

# Transformer Language Models

**Slido:** <https://app.sli.do/event/kLfanHdyTj7DT2iuwxEJMs>

**Yu Meng**

University of Virginia  
[yumeng5@virginia.edu](mailto:yumeng5@virginia.edu)

Oct 1, 2025

## Overview of Course Contents

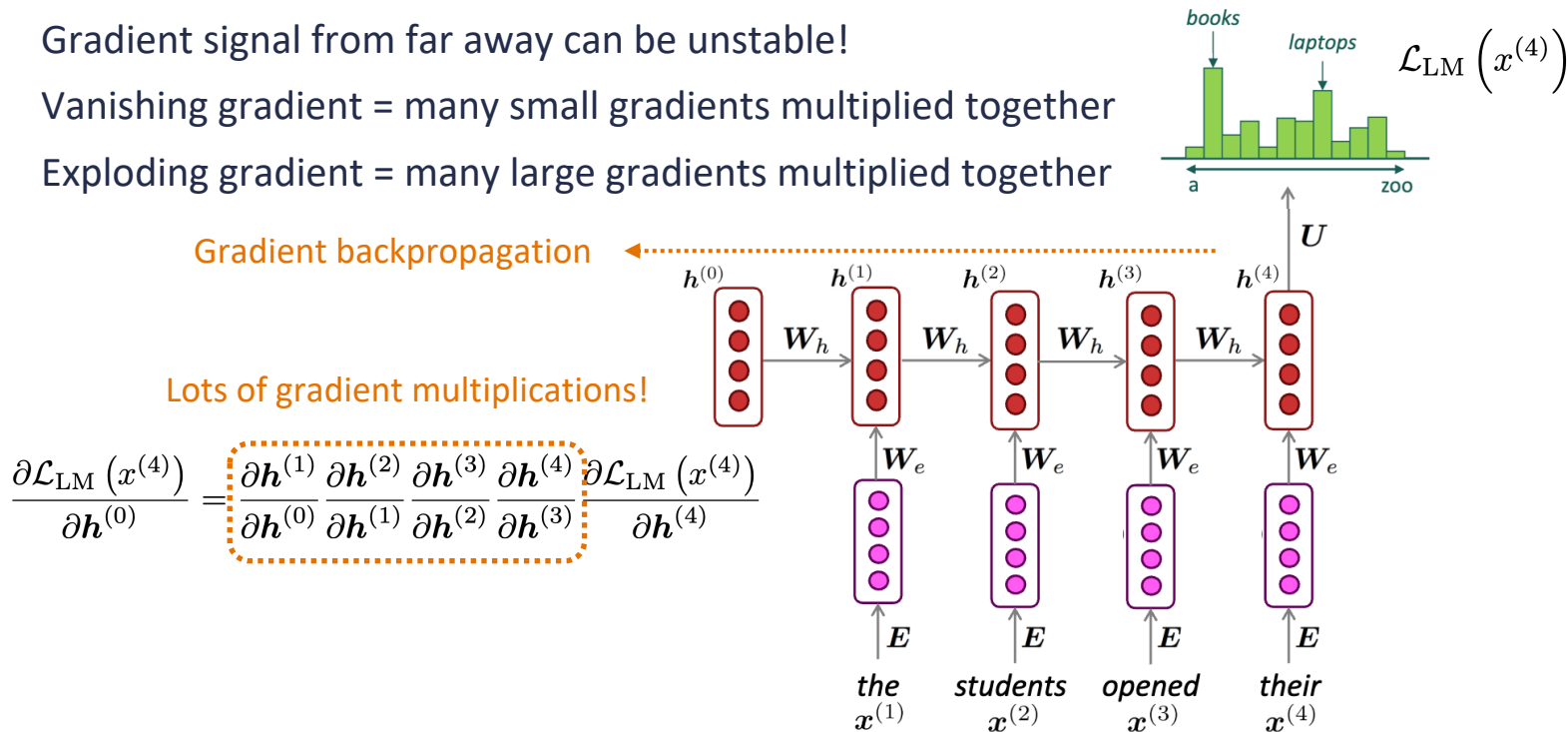
- Week 1: Logistics & Overview
- Week 2: N-gram Language Models
- Week 3: Word Senses, Semantics & Classic Word Representations
- Week 4: Word Embeddings
- Week 5: Sequence Modeling & Recurrent Neural Networks (RNNs)
- **Week 6: Language Modeling with Transformers**
- Week 9: Large Language Models (LLMs) & In-context Learning
- Week 10: Knowledge in LLMs and Retrieval-Augmented Generation (RAG)
- Week 11: LLM Alignment
- Week 12: Reinforcement Learning for LLM Post-Training
- Week 13: LLM Agents + Course Summary
- Week 15 (after Thanksgiving): Project Presentations

## Reminder

- Assignment 2 grades released; contact Wei-Lin ([wlchen@virginia.edu](mailto:wlchen@virginia.edu)) if you have questions
- No class next week (instructor travel)
- Next lecture date: 10/15 guest lecture (held on Zoom; more info later)
- Assignment 3 due next Monday (10/6) 11:59pm

## (Recap) Vanishing & Exploding Gradient

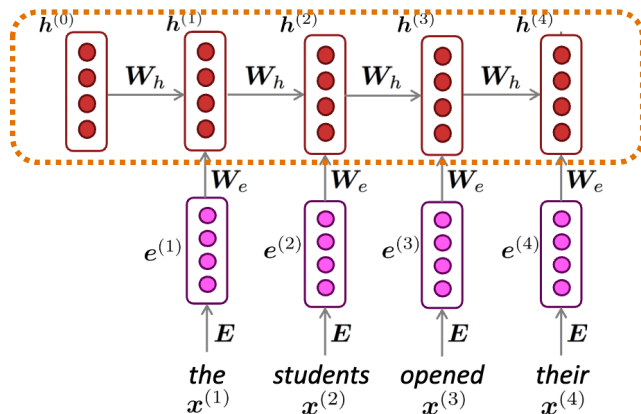
- Gradient signal from far away can be unstable!
- Vanishing gradient = many small gradients multiplied together
- Exploding gradient = many large gradients multiplied together





## (Recap) Difficulty in Capturing Long-Term Dependencies

- RNNs are theoretically capable of remembering information over arbitrary lengths of input, but they struggle in practice with long-term dependencies
- RNNs use a fixed-size hidden state to encode an entire sequence of variable length; the hidden state is required to compress a lot of information
- RNNs might give more weight to the most recent inputs and may ignore or “forget” important information at the beginning of the sentence while processing the end

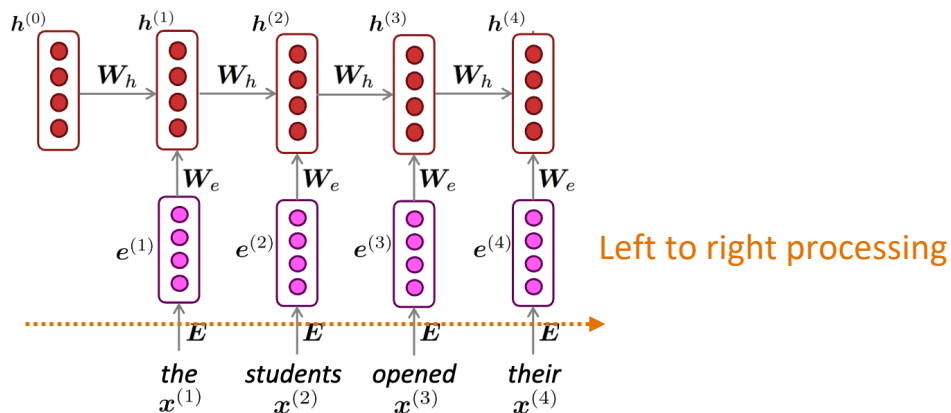


Fixed size hidden states!



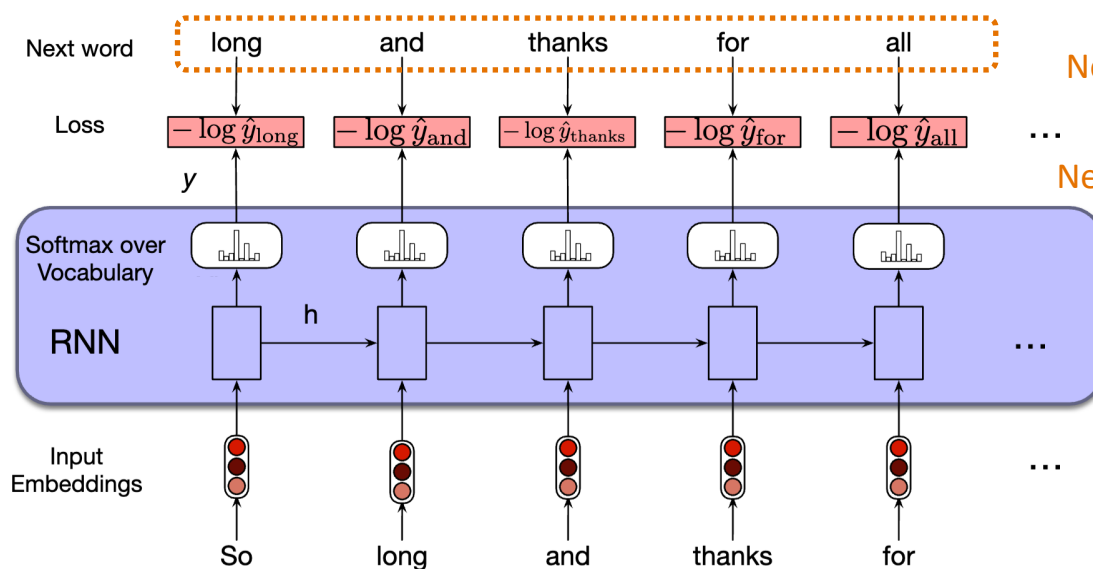
## (Recap) Lack of Bidirectionality

- RNNs process the input sequence step by step from the beginning to the end (left to right for English)
- At each time step, the hidden state only has access to the information from the past without being able to leverage future contexts
- Example: “The bank is on the river” -> the word “bank” can be correctly disambiguated only if the model has access to the word “river” later in the sentence



## (Recap) Exposure Bias

- **Teacher forcing/exposure bias:** during RNN training, the model always receives the **correct** next word from the training data as input for the next step
- When the model has to predict sequences on its own, it may perform poorly if it hasn't learned how to correct its own mistakes



During training:  
Next word = actual next word

During generation:  
Next word = model's prediction

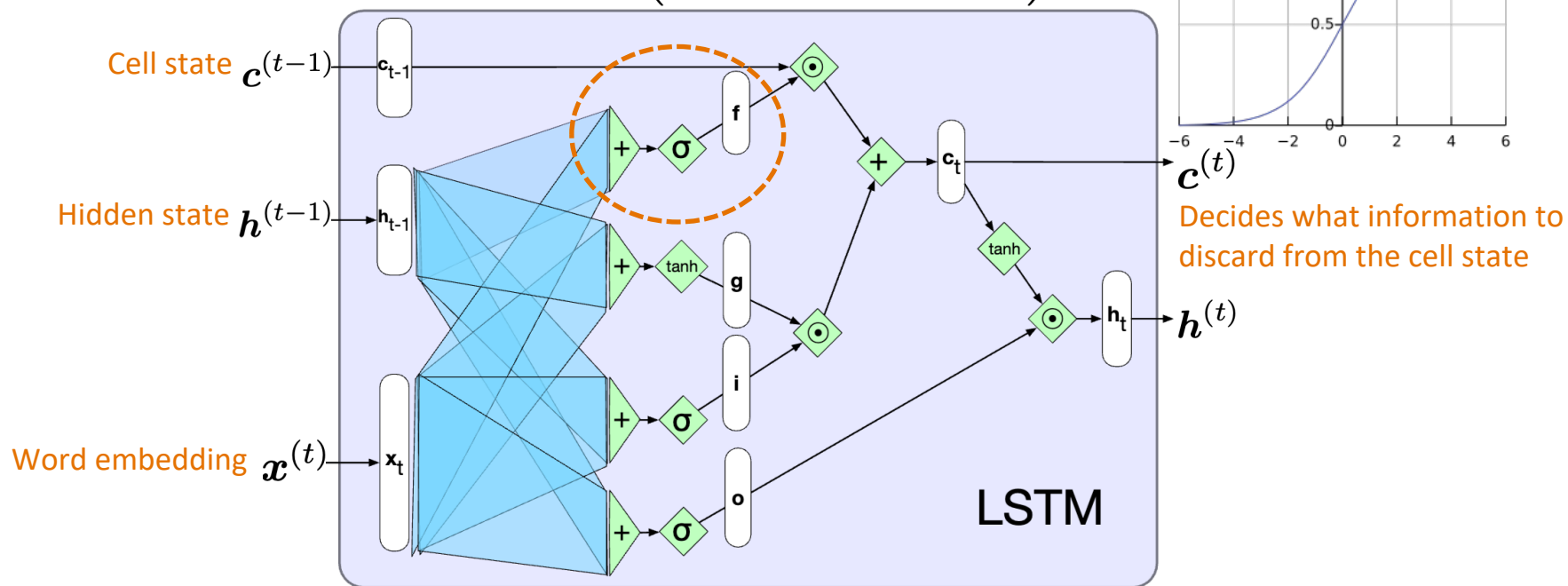
## (Recap) Long Short-Term Memory (LSTM)

- Challenge in RNNs: information encoded in hidden states tends to be local; distant information gets lost
- LSTM design intuition:
  - Remove information no longer needed from the context
  - Add information likely to be needed for future time steps
- Inputs at each time step:
  - Word embedding of the current word
  - Hidden state from the previous time step
  - **Memory/cell state**
- Three gates:
  - Forget gate
  - Add/input gate
  - Output gate

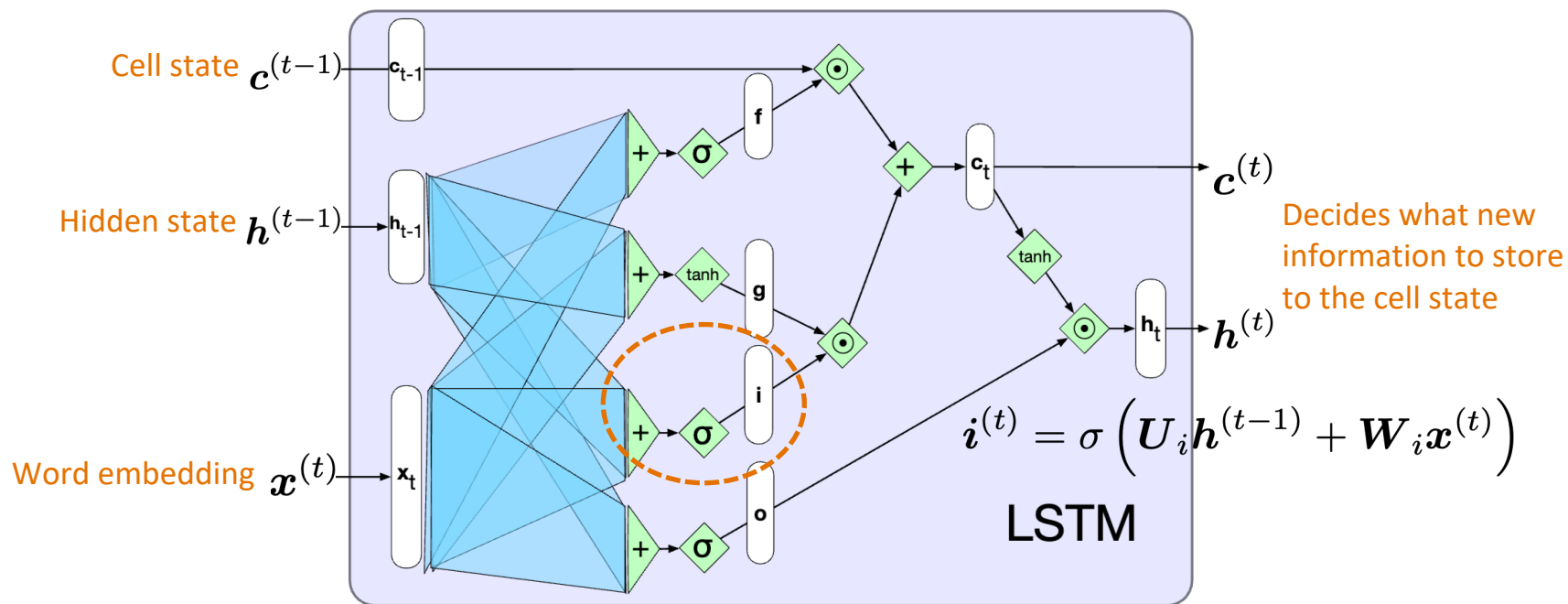


## (Recap) LSTM Computation (Forget Gate)

$$f^{(t)} = \sigma \left( U_f h^{(t-1)} + W_f x^{(t)} \right)$$

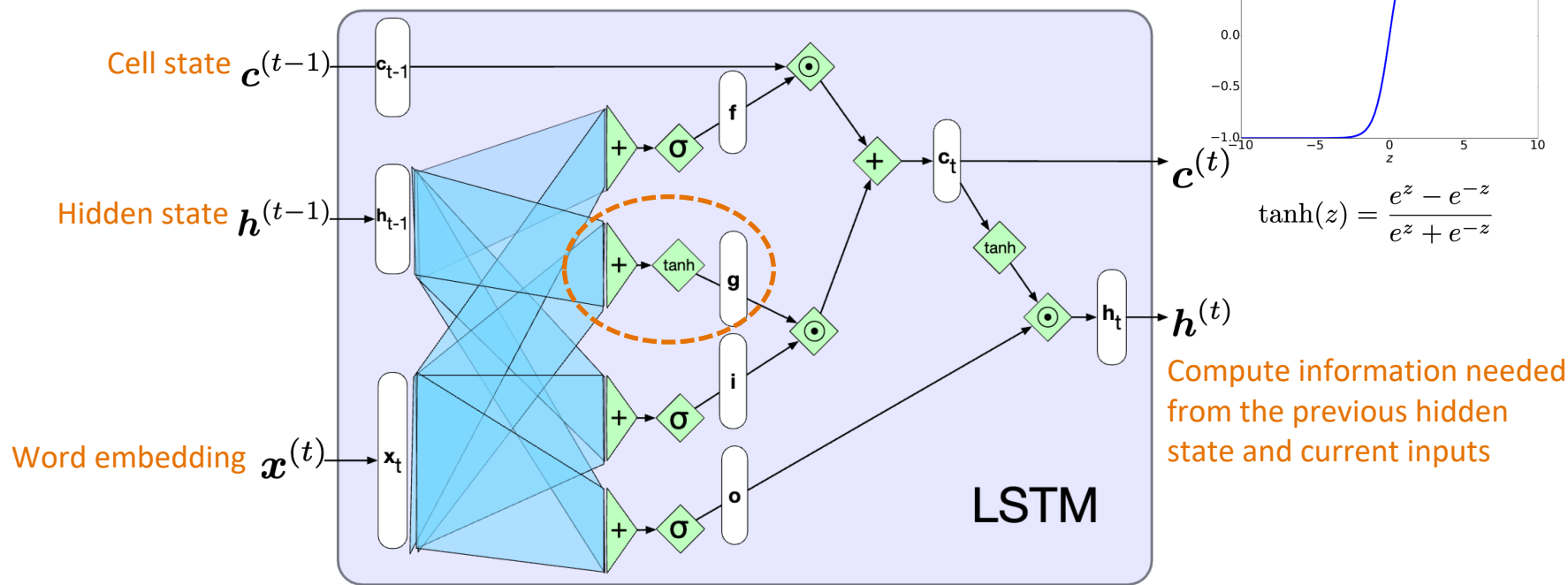


## (Recap) LSTM Computation (Add/Input Gate)



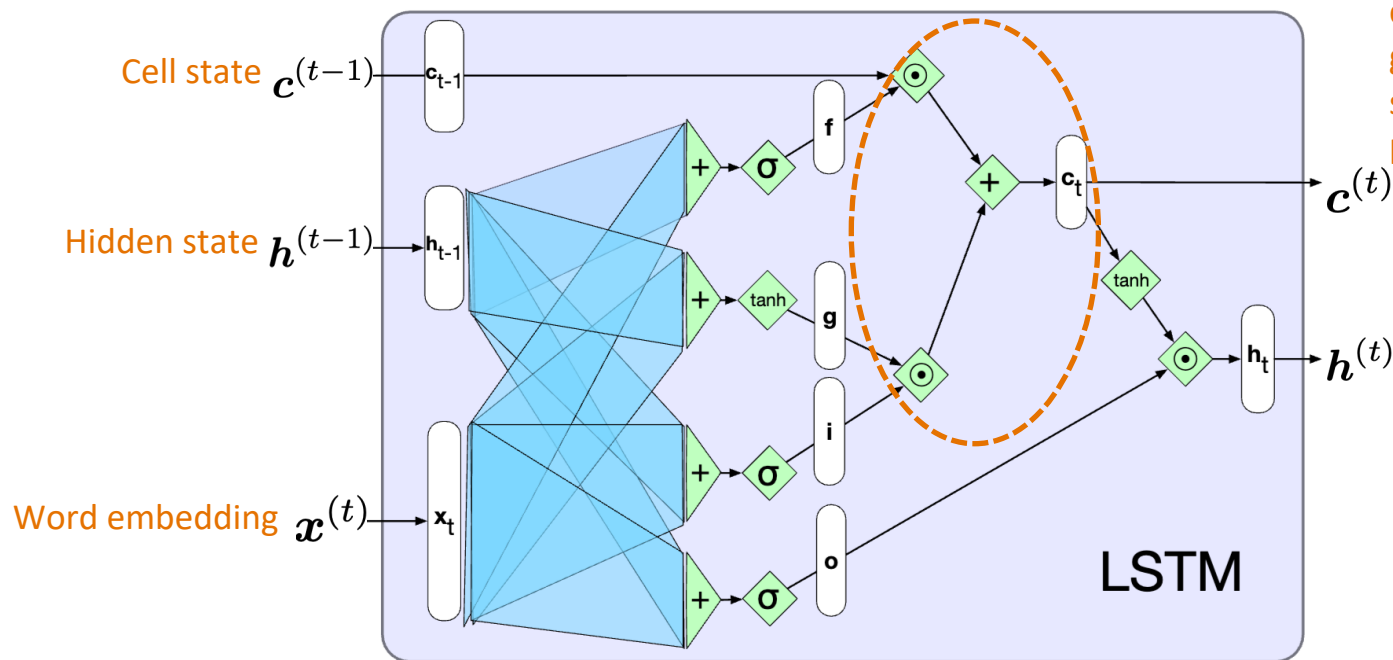
## (Recap) LSTM Computation (Candidate Cell State)

$$g^{(t)} = \tanh \left( U_g h^{(t-1)} + W_g x^{(t)} \right)$$



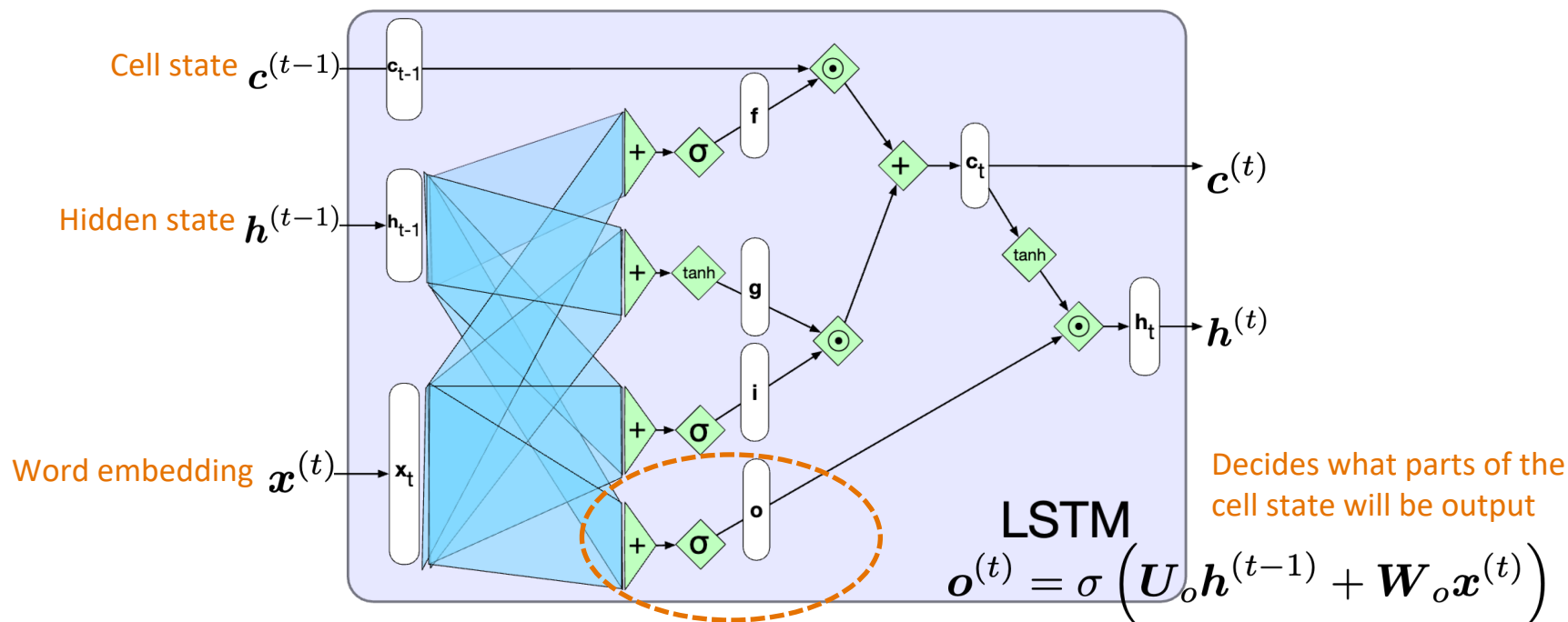
## (Recap) LSTM Computation (Cell State Update)

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

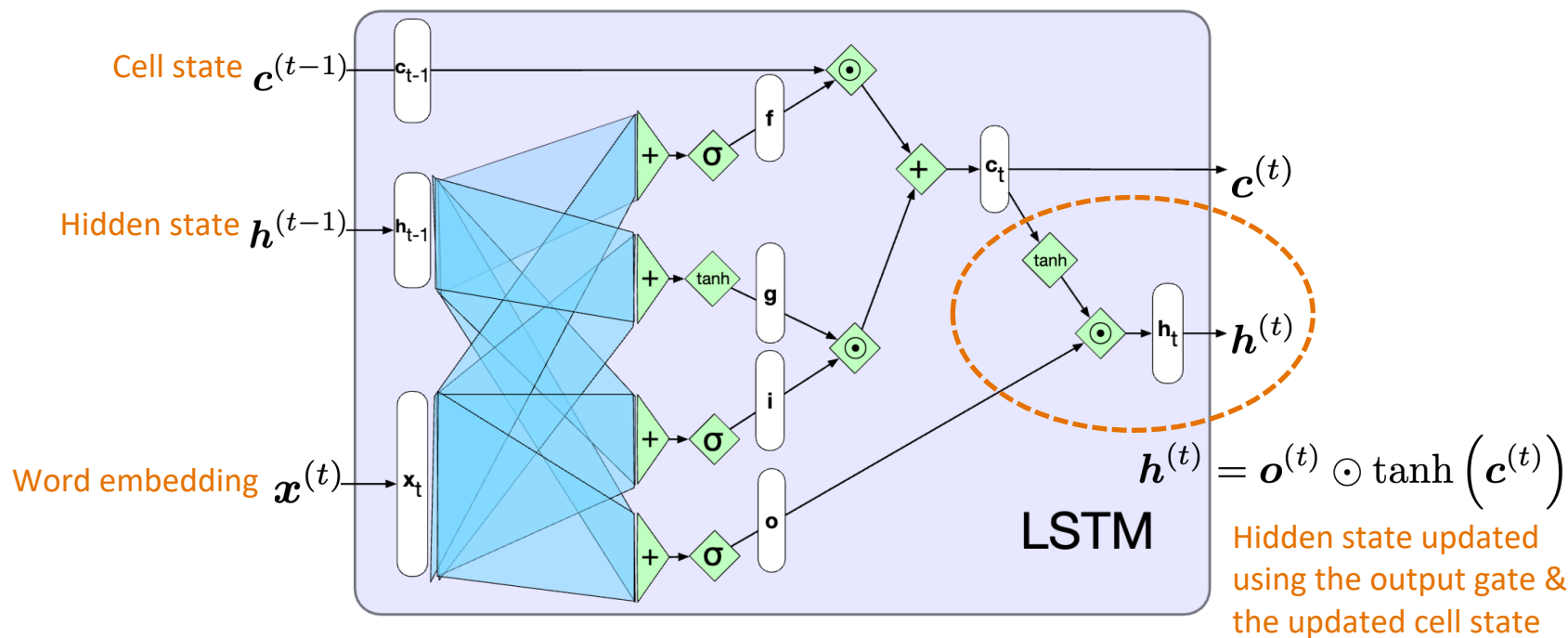


Cell state updated by combining the input gate, candidate cell state, forget gate & previous cell state

## (Recap) LSTM Computation (Output Gate)

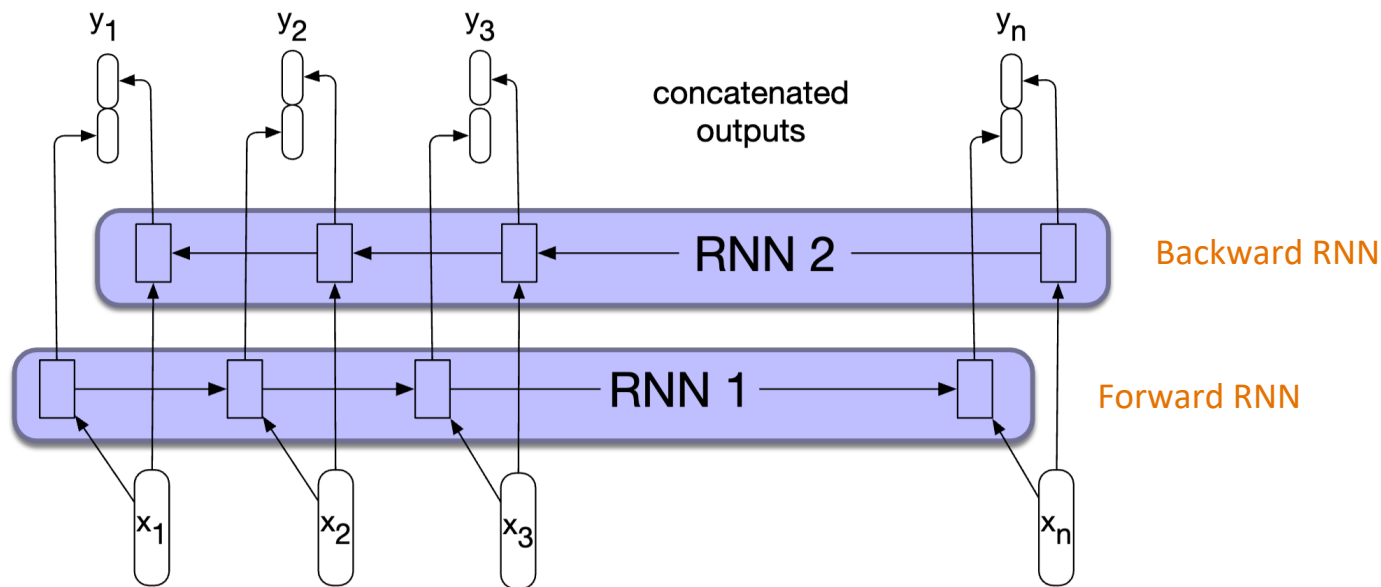


## (Recap) LSTM Computation (Hidden State Update)



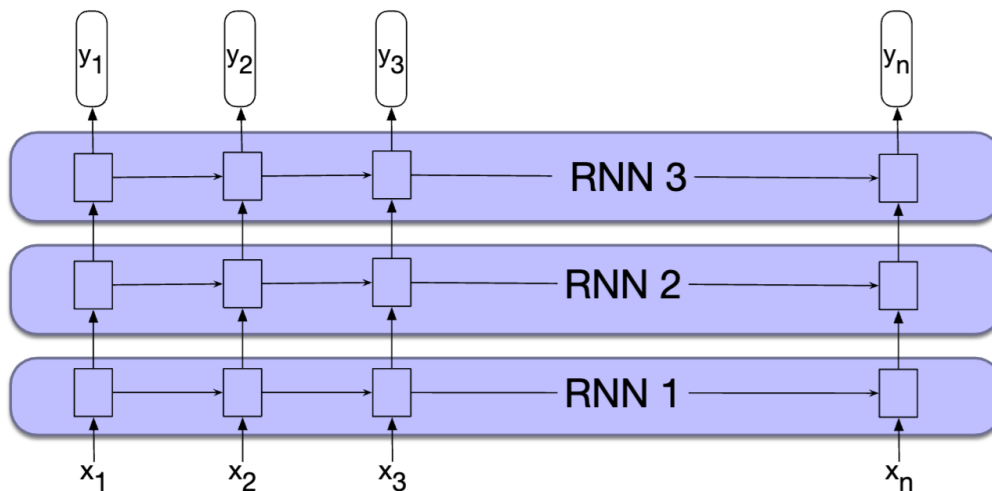
## (Recap) Bidirectional RNNs

- Separate models are trained in the forward and backward directions
- Hidden states from both RNNs are concatenated as the final representations



## (Recap) Deep RNNs

- We can stack multiple RNN layers to build deep RNNs
- The output of a lower level serves as the input to higher levels
- The output of the last layer is used as the final output







## (Recap) Transformer: Overview

- Transformer is a specific kind of sequence modeling architecture (based on DNNs)
- Use attention to replace recurrent operations in RNNs
- The most important architecture for language modeling (almost all LLMs are based on Transformers)!

---

### Attention Is All You Need

---

Ashish Vaswani\*  
Google Brain  
avaswani@google.com

Noam Shazeer\*  
Google Brain  
noam@google.com

Niki Parmar\*  
Google Research  
nikip@google.com

Jakob Uszkoreit\*  
Google Research  
usz@google.com

Llion Jones\*  
Google Research  
llion@google.com

Aidan N. Gomez\* †  
University of Toronto  
aidan@cs.toronto.edu

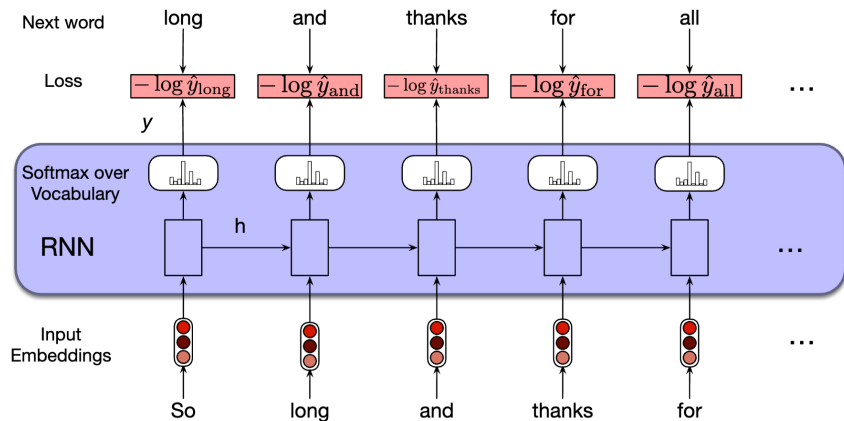
Lukasz Kaiser\*  
Google Brain  
lukaszkaiser@google.com

Illia Polosukhin\* ‡  
illia.polosukhin@gmail.com

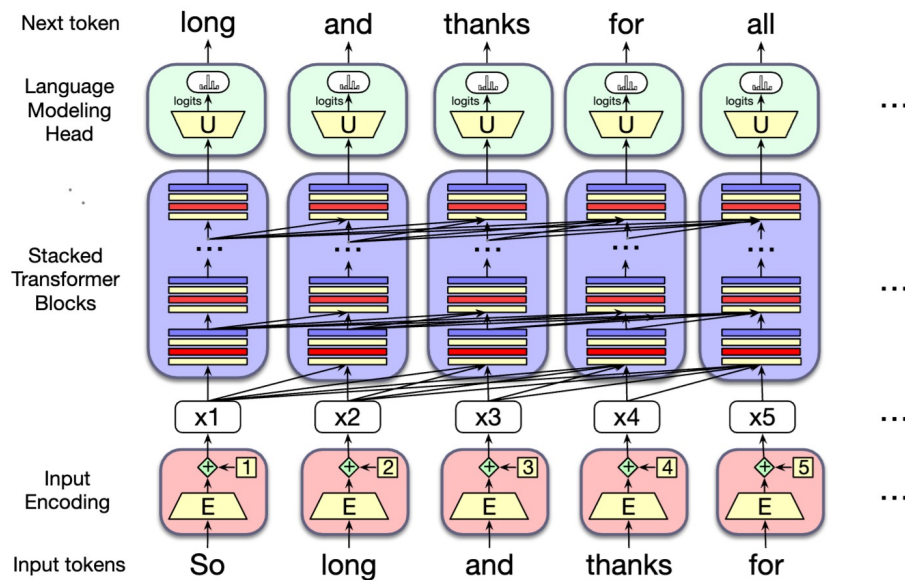
# (Recap) Transformer vs. RNN

## RNN

(recurrent computations)



## Transformer (self-attention computations)



## (Recap) Transformer: Motivation

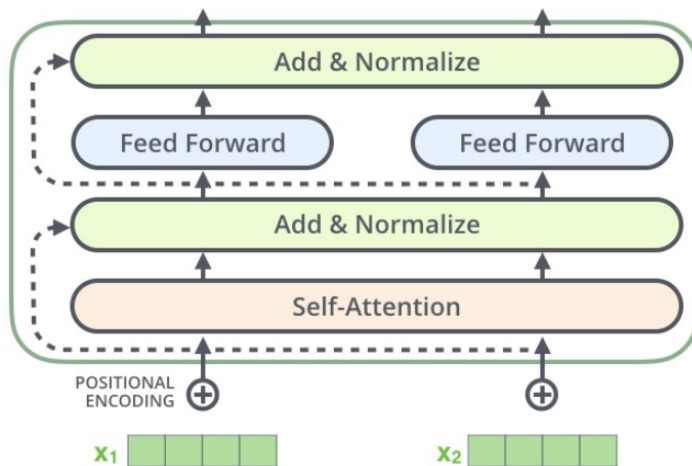
- Parallel token processing
  - RNN: process one token at a time (computation for each token depends on previous ones)
  - Transformer: process all tokens in a sequence in parallel
- Long-term dependencies
  - RNN: bad at capturing distant relating tokens (vanishing gradients)
  - Transformer: directly access any token in the sequence, regardless of its position
- Bidirectionality
  - RNN: can only model sequences in one direction
  - Transformer: inherently allow bidirectional sequence modeling via attention

## (Recap) Transformer Layer

Each Transformer layer contains the following important components:

- Self-attention
- Feedforward network
- Residual connections + layer norm

Transformer layer



## (Recap) Self-Attention: Intuition

- Attention: weigh the importance of different words in a sequence when processing a specific word
  - “When I’m looking at this word, which other words should I pay attention to in order to understand it better?”
- **Self-attention**: each word attends to other words in the **same** sequence
- Example: “The quick brown fox jumps over the lazy dog.”
  - Suppose we are learning attention for the word “**jumps**”
  - With self-attention, “**jumps**” can decide which other words in the sentence it should focus on to better understand its meaning
  - Might assign high attention to “fox” (the subject) & “over” (the preposition)
  - Might assign less attention to words like “the” or “lazy”

## Self-Attention: Example

Derive the center word representation as a weighted sum of context representations!

Center word representation

Context word representation

$$\mathbf{a}_i = \sum_{x_j \in \mathbf{x}} \alpha_{ij} \mathbf{x}_j, \quad \sum_{x_j \in \mathbf{x}} \alpha_{ij} = 1$$

Attention score  $i \rightarrow j$ , summed to 1

Context word (key) \_ Center word (query)

The	The
chicken	chicken
didn't	didn't
cross	cross
the	the
road	road
because	because
it	it
was	was
too	too
tired	tired

Current word = "it"

## Self-Attention: Attention Score Computation

- Attention score is given by the softmax function over vector dot product

$$\mathbf{a}_i = \sum_{x_j \in \mathbf{x}} \alpha_{ij} \mathbf{x}_j, \quad \sum_{x_j \in \mathbf{x}} \alpha_{ij} = 1$$

$$\alpha_{ij} = \text{Softmax}(\mathbf{x}_i \cdot \mathbf{x}_j)$$

Center word (query) representation

Context word (key) representation

- Why use two copies of word representations for attention computation?
  - We want to reflect the different roles a word plays (as the target word being compared to others, or as the context word being compared to the target word)
  - If using the same copy of representations for attention calculation, a word will (almost) always attend to itself heavily due to high dot product with itself!

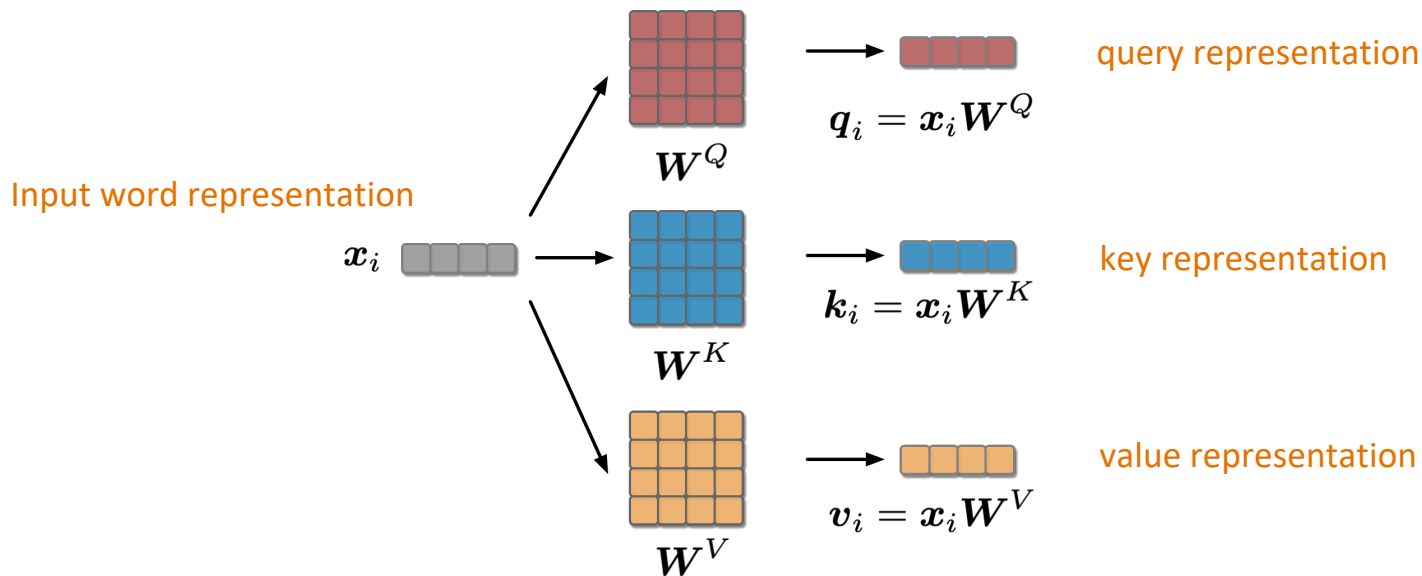
## Self-Attention: Query, Key, and Value

- Each word in self-attention is represented by three different vectors
  - Allow the model to flexibly capture different types of relationships between tokens
- **Query (Q):**
  - Represent the current word seeking information about
- **Key (K):**
  - Represent the reference (context) against which the query is compared
- **Value (V):**
  - Represent the actual content associated with each token to be aggregated as final output



## Self-Attention: Query, Key, and Value

Each self-attention module has three weight matrices applied to the input word vector to obtain the three copies of representations



## Self-Attention: Overall Computation

- Input: single word vector of each word  $\mathbf{x}_i$
- Compute Q, K, V representations for each word:

$$\mathbf{q}_i = \mathbf{x}_i \mathbf{W}^Q \quad \mathbf{k}_i = \mathbf{x}_i \mathbf{W}^K \quad \mathbf{v}_i = \mathbf{x}_i \mathbf{W}^V$$

- Compute attention scores with Q and K
  - The dot product of two vectors usually has an expected magnitude proportional to  $\sqrt{d}$
  - Divide the attention score by  $\sqrt{d}$  to avoid extremely large values in softmax function

$$\alpha_{ij} = \text{Softmax} \left( \frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d}} \right)$$

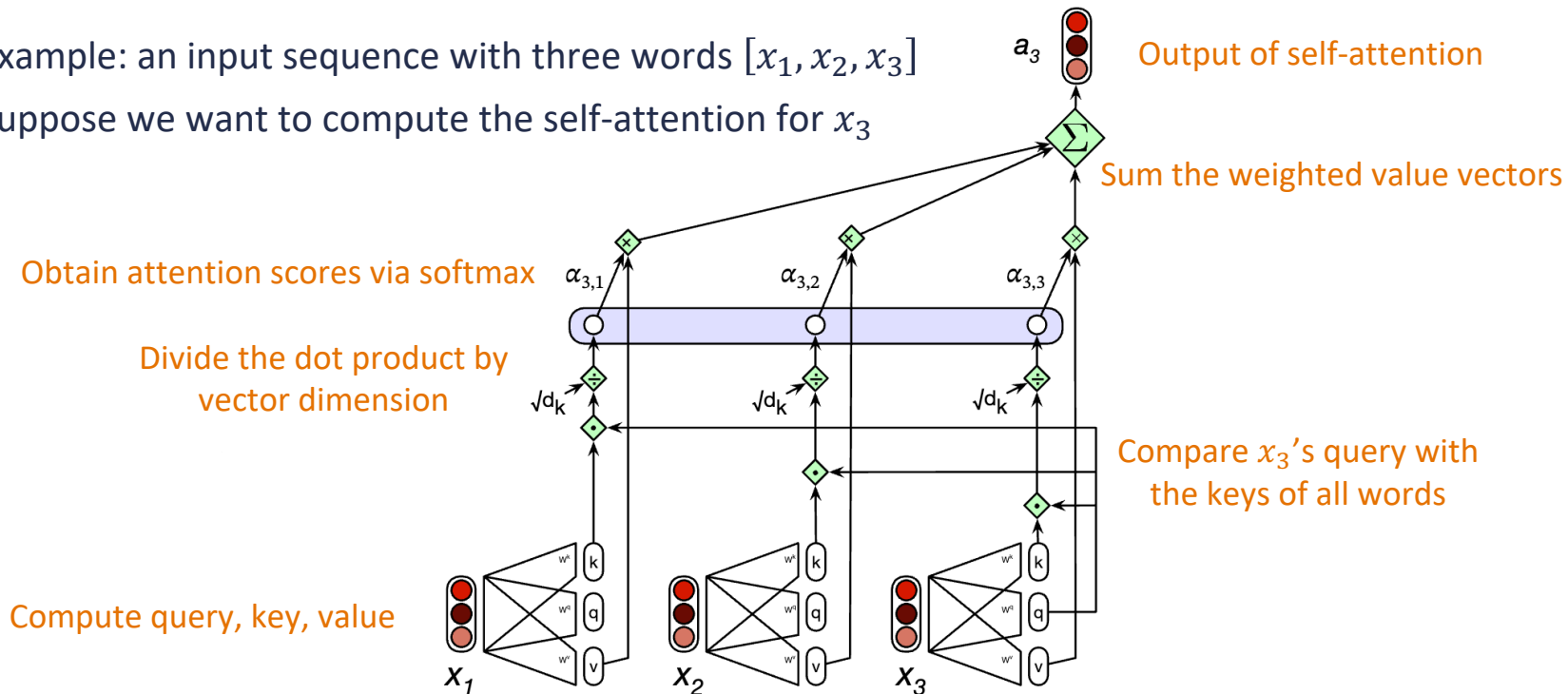
..... Dimensionality of  $q$  and  $k$

- Sum the value vectors weighted by attention scores

$$\mathbf{a}_i = \sum_{\mathbf{x}_j \in \mathbf{x}} \alpha_{ij} \mathbf{v}_j$$

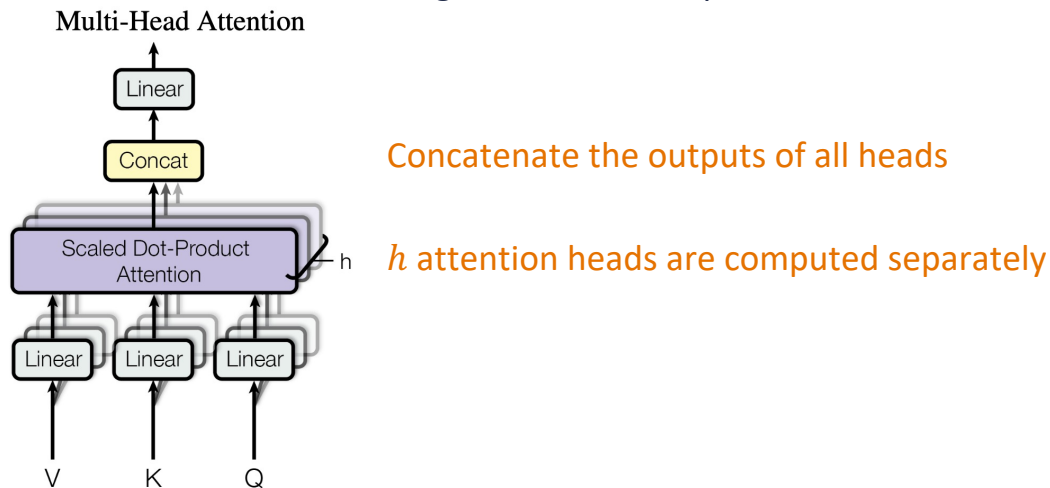
## Self-Attention: Illustration

- Example: an input sequence with three words  $[x_1, x_2, x_3]$
- Suppose we want to compute the self-attention for  $x_3$



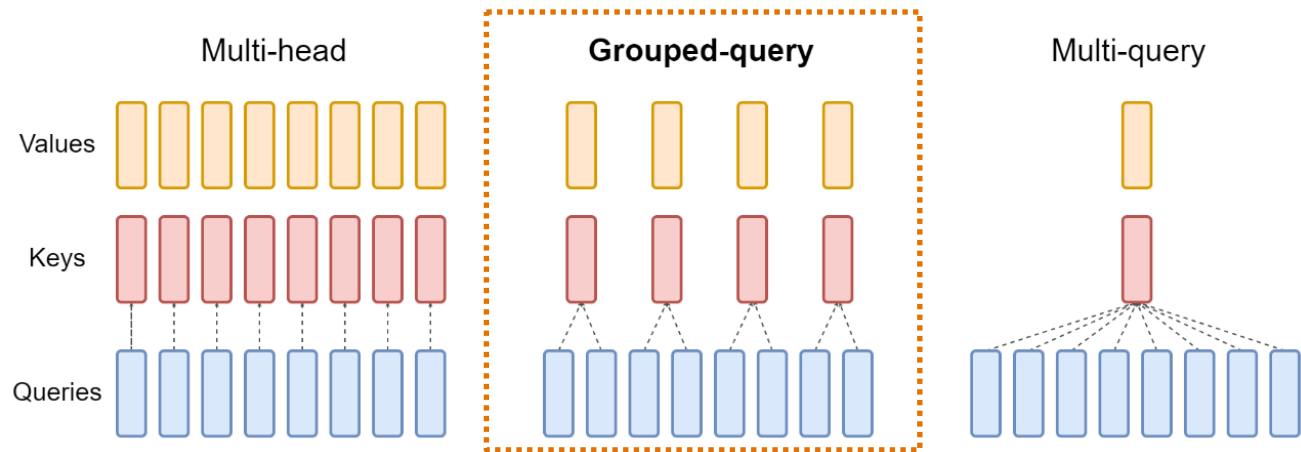
## Multi-Head Self-Attention

- Transformers use multiple attention heads for each self-attention module
- Intuition:
  - Each head might attend to the context for different purposes (e.g., particular kinds of patterns in the context)
  - Heads might be specialized to represent different linguistic relationships



## Multi-Head Self-Attention Variants

- Multi-query attention ([Fast Transformer Decoding: One Write-Head is All You Need](#)): share keys and values across all attention heads
- Grouped-query attention ([GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints](#)): share keys and values within groups of heads



Used in latest LLMs (e.g., Llama3)


Figure source: <https://arxiv.org/pdf/2305.13245>

## Parallel Computation of QKV

- Self-attention computation performed for each token is independent of other tokens
- Easily parallelize the entire computation, taking advantage of the efficient matrix multiplication capability of GPUs
- Process an input sequence with  $N$  words in parallel

Compute QKV for one word:  $\mathbf{q}_i = \mathbf{x}_i \mathbf{W}^Q \quad \mathbf{k}_i = \mathbf{x}_i \mathbf{W}^K \quad \mathbf{v}_i = \mathbf{x}_i \mathbf{W}^V \in \mathbb{R}^d$

Stacking  $N$  input vectors:  $\mathbf{Q} = \mathbf{X} \mathbf{W}^Q \quad \mathbf{K} = \mathbf{X} \mathbf{W}^K \quad \mathbf{V} = \mathbf{X} \mathbf{W}^V \in \mathbb{R}^{N \times d}$



$$\mathbf{X} = \begin{bmatrix} \text{---} & \mathbf{x}_1 & \text{---} \\ \text{---} & \mathbf{x}_2 & \text{---} \\ \dots & \dots & \dots \\ \text{---} & \mathbf{x}_N & \text{---} \end{bmatrix}$$

## Parallel Computation of Attention

Attention computation can also be written in matrix form

Compute attention for one word:  $a_i = \text{Softmax} \left( \frac{q_i \cdot k_j}{\sqrt{d}} \right) \cdot v_j$

Compute attention for one  $N$  words:  $A = \text{Softmax} \left( \frac{QK^\top}{\sqrt{d}} \right) V$   $N$

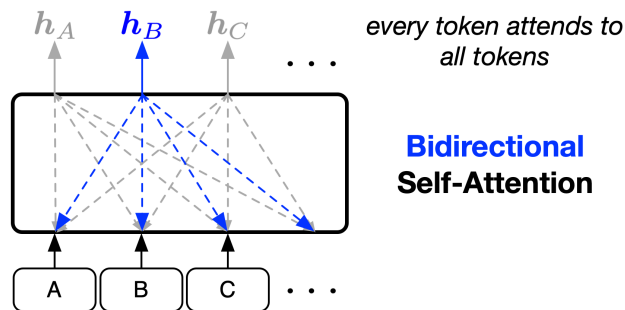
Attention matrix

q1•k1	q1•k2	q1•k3	q1•k4
q2•k1	q2•k2	q2•k3	q2•k4
q3•k1	q3•k2	q3•k3	q3•k4
q4•k1	q4•k2	q4•k3	q4•k4

$N$

## Bidirectional vs. Unidirectional Self-Attention

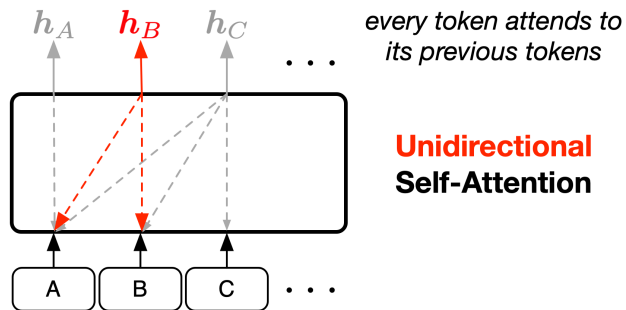
- Self-attention can capture different context dependencies
- **Bidirectional** self-attention:
  - Each position attends to all other positions in the input sequence
  - Transformers with bidirectional self-attention are called Transformer **encoders** (e.g., BERT)
  - Use case: natural language understanding (NLU) where the entire input is available at once, such as text classification & named entity recognition





## Bidirectional vs. Unidirectional Self-Attention

- Self-attention can capture different context dependencies
- Unidirectional** (or **causal**) self-attention:
  - Each position can only attend to earlier positions in the sequence (including itself).
  - Transformers with unidirectional self-attention are called Transformer **decoders** (e.g., GPT)
  - Use case: natural language generation (NLG) where the model generates output sequentially



upper-triangle portion set to  $-\infty$

N

$q1 \cdot k1$	$-\infty$	$-\infty$	$-\infty$
$q2 \cdot k1$	$q2 \cdot k2$	$-\infty$	$-\infty$
$q3 \cdot k1$	$q3 \cdot k2$	$q3 \cdot k3$	$-\infty$
$q4 \cdot k1$	$q4 \cdot k2$	$q4 \cdot k3$	$q4 \cdot k4$

## Agenda

- Position Encoding
- Tokenization
- Other Transformer Modules

## Position Encoding

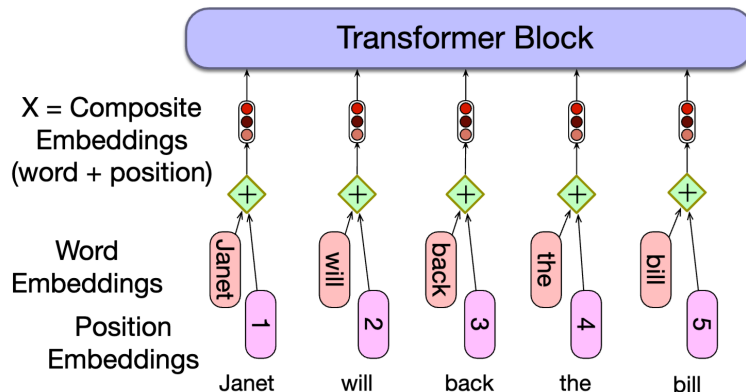
- Motivation: inject positional information to input vectors

$$q_i = x_i W^Q \quad k_i = x_i W^K \quad v_i = x_i W^V \in \mathbb{R}^d$$

$$a_i = \text{Softmax} \left( \frac{q_i \cdot k_j}{\sqrt{d}} \right) \cdot v_j$$

When  $x$  is word embedding,  $q$  and  $k$  do not have positional information!

- How to know the word positions in the sequence? Use position encoding!



## Position Encoding Methods

- Absolute position encoding (the original Transformer paper)
  - Learn position embeddings for each position
  - Not generalize well to sequences longer than those seen in training
- Relative position encoding ([Self-Attention with Relative Position Representations](#))
  - Encode the relative distance between words rather than their absolute positions
  - Generalize better to sequences of different lengths
- Rotary position embedding ([RoFormer: Enhanced Transformer with Rotary Position Embedding](#))
  - Apply a rotation matrix to the word embeddings based on their positions
  - Incorporate both absolute and relative positions
  - Generalize effectively to longer sequences
  - Widely-used in latest LLMs

## Agenda

- Position Encoding
- Tokenization
- Other Transformer Modules



## Tokenization: Overview

- Tokenization: splitting a string into a sequence of tokens
- Simple approach: use whitespaces to segment the sequence
  - One token = one word
  - We have been using “tokens” and “words” interchangeably
- However, segmentation using whitespaces is not the approach used in modern large language models

Multiple models, each with different capabilities and price points. Prices can be viewed in units of either per 1M or 1K tokens. You can think of tokens as pieces of words, where 1,000 tokens is about 750 words.

## Limitation of Word-Based Segmentation

- Out-of-vocabulary (OOV) issues:
  - Cannot handle words never seen in our training data
  - Reserving an [UNK] token for unseen words is a remedy
- Subword information:
  - Loses subword information valuable for understanding word meaning and structure
  - Example: “unhappiness” -> “un” + “happy” + “ness”
- Data sparsity and exploded vocabulary size:
  - Require a large vocabulary (vocabulary size = number of unique words)
  - The model sees fewer examples of each word (harder to generalize)



## Single-Character Segmentation?

- How about segmenting sequences by character?
  - No OOV issue
  - Small vocabulary size
- Increased sequence length:
  - Significantly increases the length of input sequences
  - Transformer's self-attention has quadratic complexity w.r.t. sequence length!
- Loss of word-level semantics:
  - Characters alone often don't carry semantic meaning/linguistic patterns



## Subword Tokenization

- Strike a balance between character-level and word-level tokenization
  - Capture meaningful subword semantics
  - Handle out-of-vocabulary words better
  - Efficient sequence modeling
- Three common algorithms:
  - Byte-Pair Encoding (BPE): [Sennrich et al. \(2016\)](#)
  - WordPiece: [Schuster and Nakajima \(2012\)](#)
  - SentencePiece: [Kudo and Richardson \(2018\)](#)
- Subword tokenization usually consists of two parts:
  - A token learner that takes a raw training corpus and induces a **vocabulary** (a set of tokens)
  - A token segmenter that takes a raw sentence and **tokenizes** it according to that vocabulary

## Byte-Pair Encoding (BPE) Overview

- BPE is the most commonly used tokenization algorithm in modern LLMs
- Intuition: start with a character-level vocabulary and iteratively merges the most frequent pairs of tokens
- **Initialization:** Let vocabulary be the set of all individual characters: {A, B, C, D, ..., a, b, c, d, ....}
- **Frequency counting:** count all adjacent symbol pairs (could be a single character or a previously merged pair) in the training corpus
- **Pair merging:** merge the most frequent pair of symbols (e.g. 't', 'h' => "th")
- **Update corpus:** replace all instances of the merged pair in the corpus with the new token & update the frequency of pairs
- **Repeat:** repeat the process of counting, merging, and updating until a predefined number of merges (or vocabulary size) is reached



## BPE: Token Learner

Token learner of BPE

```
function BYTE-PAIR ENCODING(strings  $C$ , number of merges  $k$ ) returns vocab  $V$   
  
   $V \leftarrow$  all unique characters in  $C$            # initial set of tokens is characters  
  for  $i = 1$  to  $k$  do                             # merge tokens til  $k$  times  
     $t_L, t_R \leftarrow$  Most frequent pair of adjacent tokens in  $C$   
     $t_{NEW} \leftarrow t_L + t_R$                      # make new token by concatenating  
     $V \leftarrow V + t_{NEW}$                            # update the vocabulary  
    Replace each occurrence of  $t_L, t_R$  in  $C$  with  $t_{NEW}$    # and update the corpus  
  return  $V$ 
```

## BPE Example

Suppose we have the following corpus

low low low low low lowest lowest newer newer newer  
newer newer newer wider wider wider new new

**corpus**

5   l o w \_  
2   l o w e s t \_  
6   n e w e r \_  
3   w i d e r \_  
2   n e w \_

Special “end-of-word” character  
(distinguish between subword units  
vs. whole word)

**vocabulary**

\_, d, e, i, l, n, o, r, s, t, w



## BPE: Counting & Merging

The adjacent symbol pair with the highest frequency is “er” (count = 9)

low low low low low lowest lowest newer newer newer  
newer newer newer wider wider wider new new

**corpus**

5 l o w \_  
2 l o w e s t \_  
6 n e w e r \_  
3 w i d e r \_  
2 n e w \_

Merge “er”



**corpus**

5 l o w \_  
2 l o w e s t \_  
6 n e w e r \_  
3 w i d e r \_  
2 n e w \_

**vocabulary**

\_, d, e, i, l, n, o, r, s, t, w



**vocabulary**

\_, d, e, i, l, n, o, r, s, t, w, er

## BPE: Counting & Merging

The adjacent symbol pair with the highest frequency is “er\_” (count = 9)

low low low low low lowest lowest newer newer newer  
newer newer newer wider wider wider new new

**corpus**

5 l o w \_  
2 l o w e s t \_  
6 n e w er \_  
3 w i d er \_  
2 n e w \_

Merge “er\_”



**corpus**

5 l o w \_  
2 l o w e s t \_  
6 n e w er\_  
3 w i d er\_  
2 n e w \_

**vocabulary**

\_, d, e, i, l, n, o, r, s, t, w, er



**vocabulary**

\_, d, e, i, l, n, o, r, s, t, w, er, er\_

## BPE: Counting & Merging

The adjacent symbol pair with the highest frequency is “ne” (count = 8)

low low low low low lowest lowest newer newer newer  
newer newer newer wider wider wider new new

**corpus**

5 l o w \_  
2 l o w e s t \_  
6 n e w e r \_  
3 w i d e r \_  
2 n e w \_

Merge “ne”



**corpus**

5 l o w \_  
2 l o w e s t \_  
6 n e w e r \_  
3 w i d e r \_  
2 n e w \_

**vocabulary**

\_, d, e, i, l, n, o, r, s, t, w, e r, e r \_



**vocabulary**

\_, d, e, i, l, n, o, r, s, t, w, e r, e r \_, ne

## BPE: Counting & Merging

Continue the process to merge more adjacent symbols

low low low low low lowest lowest newer newer newer  
newer newer newer wider wider wider new new

**corpus**

5 l o w \_  
2 l o w e s t \_  
6 n e w e r\_  
3 w i d e r\_  
2 n e w \_

**merge**

**current vocabulary**

(ne, w)	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new
(l, o)	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo
(lo, w)	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo, low
(new, er_)	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo, low, newer_
(low, _)	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo, low, newer_, low_





## BPE: Token Segmenter

- Once we learn our vocabulary, we need a token segmenter to tokenize an unseen sentence (from test set)
- Just run (greedily based on training data frequency) on the merge rules we have learned from the training data on the test data
- Example:
  - Assume the merge rules: [(e, r), (er, \_), (n, e), (ne, w), (l, o), (lo, w), (new, er\_), (low, \_)]
  - First merge all adjacent “er”, then all adjacent “er\_”, then all adjacent “ne”...
  - “newer\_” from the test set will be tokenized as a whole word
  - “lower\_” from the test set will be tokenized as “low” + “er\_”

low low low low low lowest lowest newer newer newer  
newer newer newer wider wider wider new new

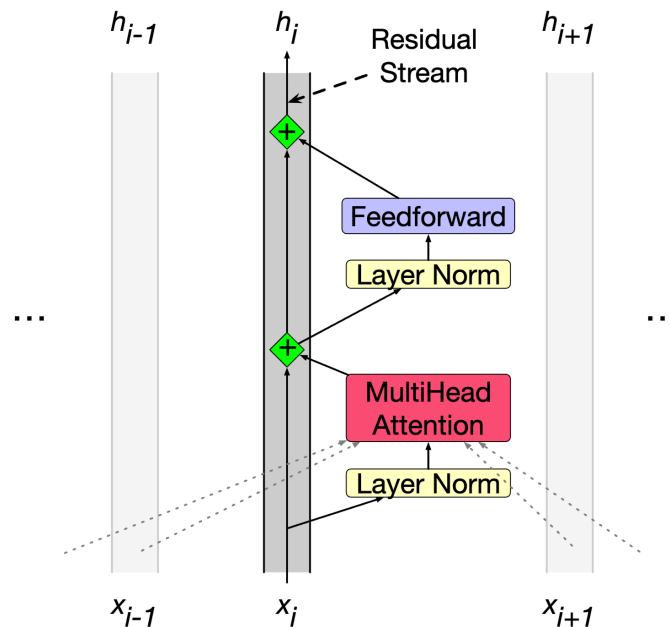
“lower\_” is an unseen word from the training set

## Agenda

- Position Encoding
- Tokenization
- Other Transformer Modules

## Transformer Block

- Modules in Transformer layers:
  - Multi-head attention
  - Layer normalization (LayerNorm)
  - Feedforward network (FFN)
  - Residual connection



## Layer Normalization: Motivation

- Proposed in [Ba et al. \(2016\)](#)
- The distribution of inputs to DNN can change during training – “internal covariate shift”
- Slow down the training process: the model constantly adapts to changing distributions

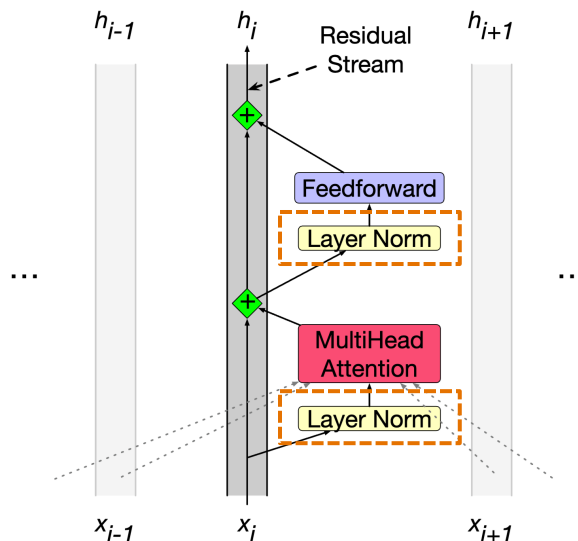


Figure source: <https://web.stanford.edu/~jurafsky/slp3/8.pdf>

## Layer Normalization: Solution

- Normalize the input vector  $\mathbf{x}$ 
  - Calculate the mean & standard deviation over the input vector dimensions


$$\mu = \frac{1}{d} \sum_{i=1}^d x_i \quad \sigma = \sqrt{\frac{1}{d} \sum_{i=1}^d (x_i - \mu)^2}$$

- Apply normalization

$$\hat{\mathbf{x}} = \frac{\mathbf{x} - \mu}{\sigma}$$

- Learn to scale and shift the normalized output with parameters

$$\text{LayerNorm}(\mathbf{x}) = \gamma \frac{\mathbf{x} - \mu}{\sigma} + \beta$$

  
Learnable parameters

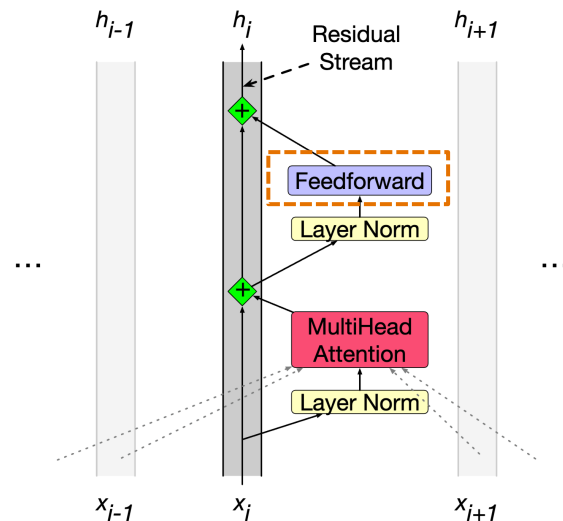


## Feedforward Network (FFN)

- FFN in Transformer is a 2-layer network (one hidden layer, two weight matrices)

$$\text{FFN}(x_i) = \text{ReLU}(x_i W_1) W_2$$

- Apply non-linear activation after the first layer
- Same weights applied to every token
- Weights are different across different Transformer layers

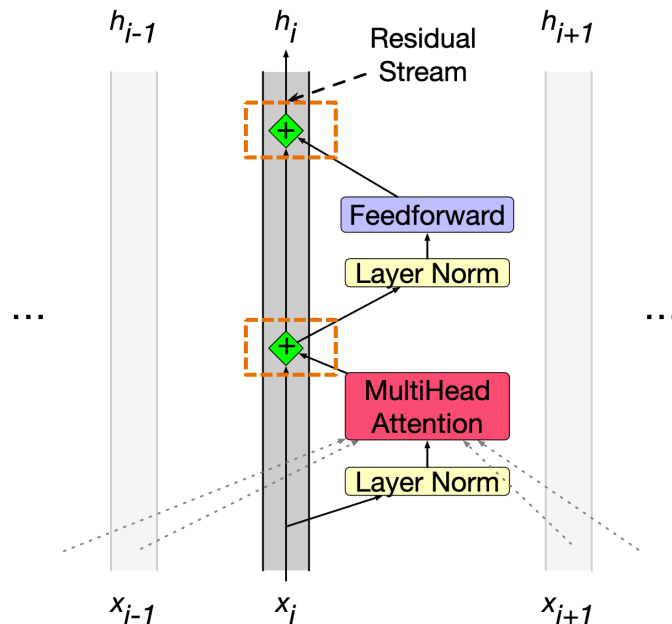


## Residual Connections

- Add the original input to the output of a sublayer (e.g., attention/FFN)

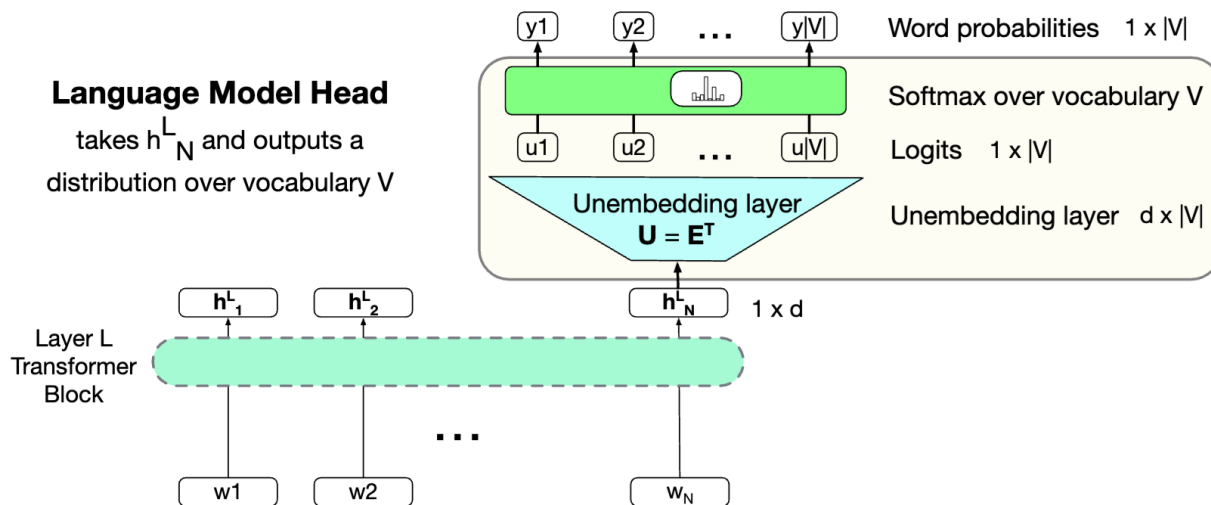
$$\mathbf{y} = \mathbf{x} + f(\mathbf{x})$$

- Benefits
  - Address the vanishing gradient problem
  - Facilitate information flow across the network
  - Help scale up model



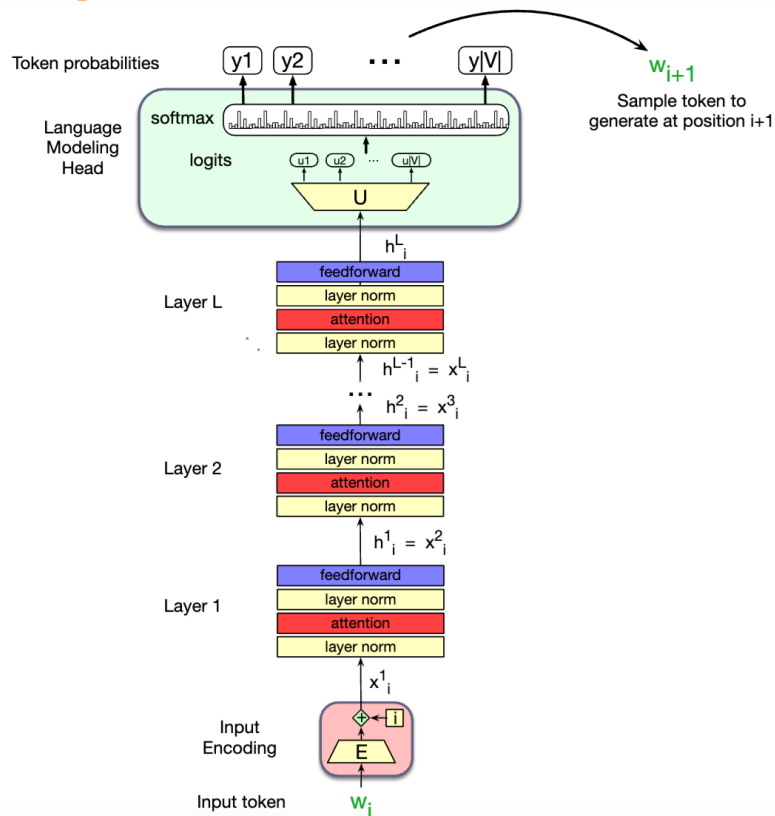
## Language Model Head

- Language model head is added to the final layer
- Usually apply the weight tying trick (share weights between input embeddings and the output embeddings)



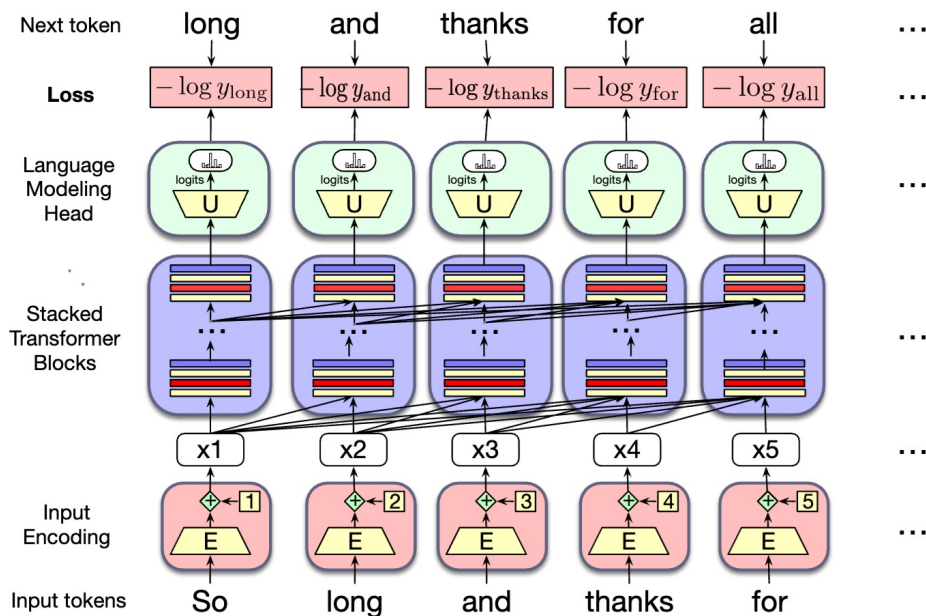


# Transformer Language Model: Overview



# Transformer Language Model Training

Use cross-entropy loss to train Transformers for language modeling (like RNN LMs)





# Thank You!

**Yu Meng**

University of Virginia

[yumeng5@virginia.edu](mailto:yumeng5@virginia.edu)