Transformers

Slido: https://app.sli.do/event/k5TBSCako9oeaZZZYuWkfK

Yu Meng
University of Virginia
yumeng5@virginia.edu

Sept 29, 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

UNIVERSITY of VIRGINIA

Reminder

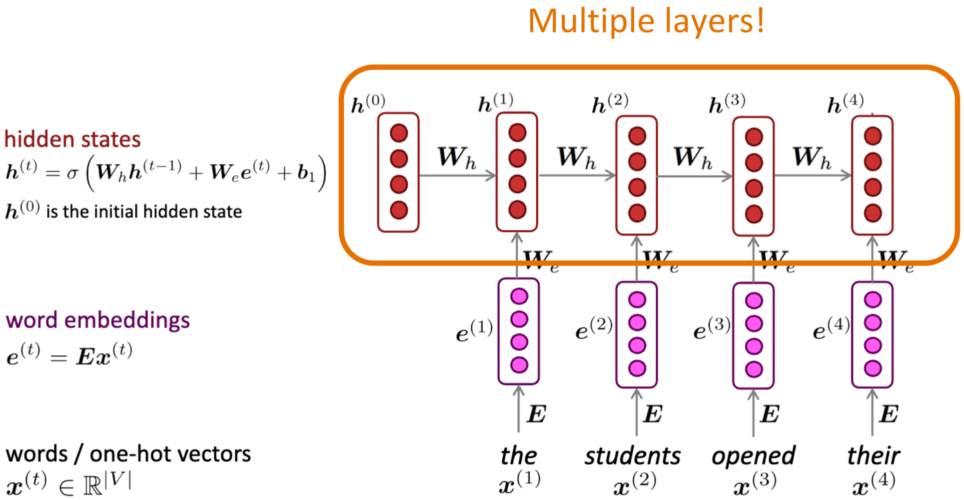
- Project proposal grades & feedback released
- Assignment 3 released; due date: 10/06 11:59pm

UNIVERSITY of VIRGINIA

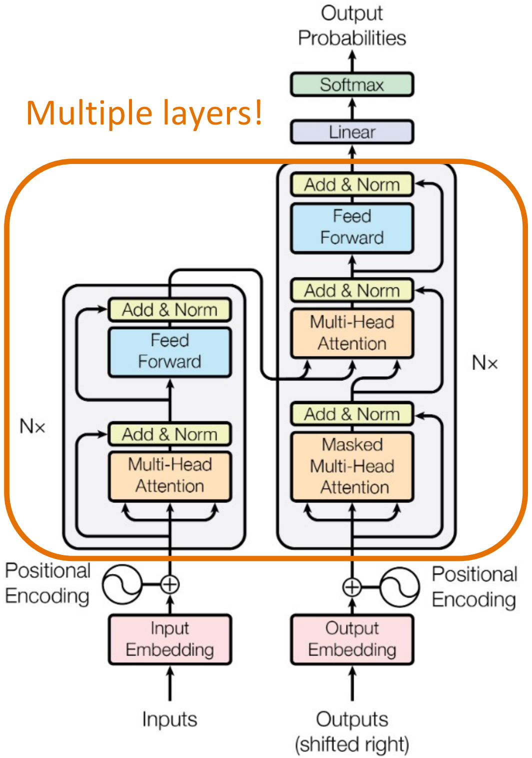
(Recap) Sequence Modeling: Overview

- Use deep learning methods to understand, process, and generate **text sequences**
- Goals:
  - Learn context-dependent representations
  - Capture long-range dependencies
  - Handle complex relationships among large text units
- Sequence modeling architectures are based on deep neural networks (DNNs)!
  - Language exhibits hierarchical structures (e.g., letters form words, words form phrases, phrases form sentences)
  - DNNs learn multiple levels of abstraction across layers, allowing them to capture low-level patterns (e.g., word relations) in lower layers and high-level patterns (e.g., sentence meanings) in higher layers
  - Each layer in DNNs refines the word representations by considering contexts at different granularities (shorter & longer-range contexts), allowing for contextualized understanding of words and sequences

(Recap) Sequence Modeling Architectures



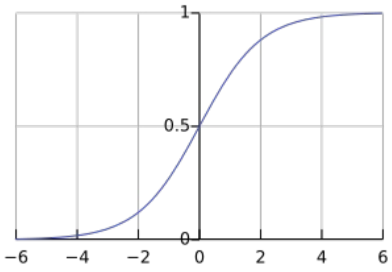
RNN neural networks:
https://web.stanford.edu/class/cs224n/slides/cs224n-spr2024-lecture05-rnnlm.pdf



Transformer: https://arxiv.org/pdf/1706.03762

(Recap) Neural Network: Basic Unit (Perceptron)

- Input: $\boldsymbol{x} = [x_1, x_2, x_3]$
- Model parameters (weights & bias): $\boldsymbol{w} = [w_1, w_2, w_3]$ & b
- Linear computation: $z = \boldsymbol{w} \cdot \boldsymbol{x} + b$
- Nonlinear activation: $a = \sigma(z)$



$y = \sigma(x)$

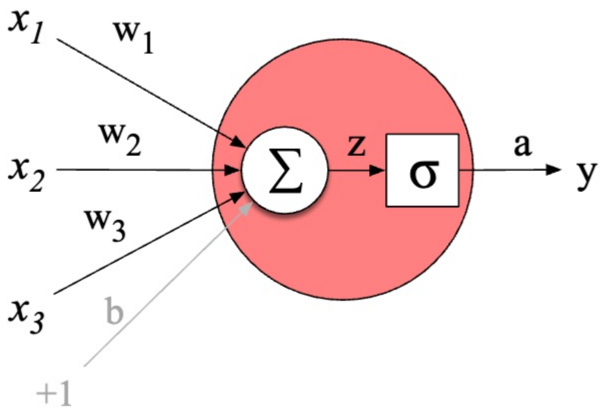


Figure source: https://web.stanford.edu/~jurafsky/slp3/6.pdf

(Recap) Basic Unit (Perceptron): Example

- Input: $\boldsymbol{x} = [0.5, 0.6, 0.1]$
- Model parameters (weights & bias): $\boldsymbol{w} = [0.2, 0.3, 0.9]$ & $b = 0.5$
- Linear computation: $z = \boldsymbol{w} \cdot \boldsymbol{x} + b = 0.87$
- Nonlinear activation: $a = \sigma(z) = \dfrac{1}{1 + \exp(-0.87)} \approx 0.70$

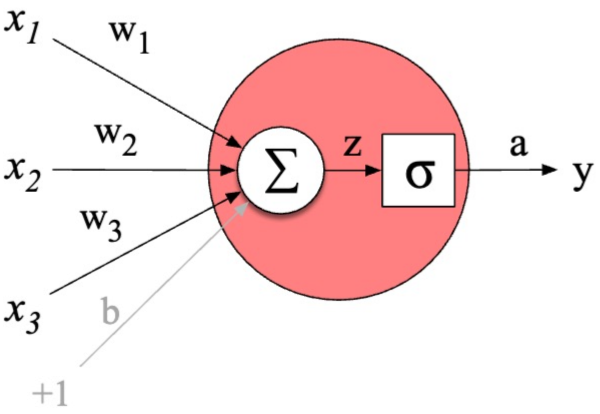
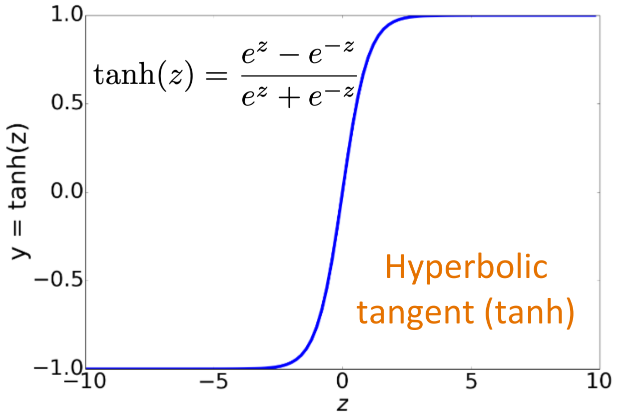


Figure source: https://web.stanford.edu/~jurafsky/slp3/6.pdf

(Recap) Common Non-linear Activations

- Why non-linear activations?
- Stacking linear operations will only result in another linear operation
- We wish our network to model complex, non-linear relationships between inputs and outputs



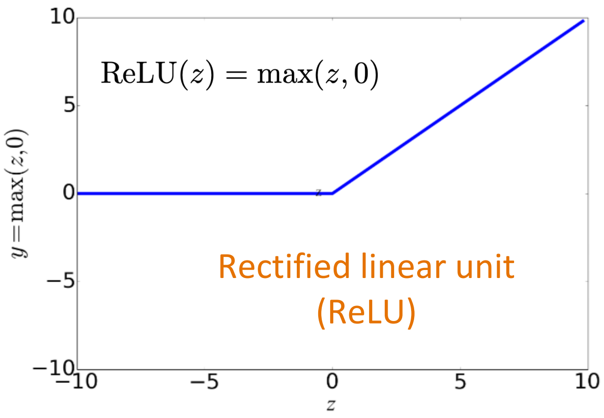


Figure source: https://web.stanford.edu/~jurafsky/slp3/6.pdf

UNIVERSITY of VIRGINIA

(Recap) Feedforward Network (FFN)

- Feedforward network (FFN) = multi-layer network where the outputs from units in each layer are passed to units in the next higher layer
- FFNs are also called multi-layer perceptrons (MLPs)
- Model parameters in each layer in FFNs: a weight matrix $\boldsymbol{W}$ and a bias vector $\boldsymbol{b}$
  - Each layer has multiple hidden units
  - Recall: a single hidden unit has a weight vector and a bias parameter
  - Weight matrix: combining the weight vector for each unit
  - Bias vector: combining the bias for each unit

(Recap) Example: 2-layer FFN

- Input: $\boldsymbol{x} = [x_1, x_2, \ldots, x_{n_0}]$
- Model parameters (weights & bias): $\boldsymbol{W} \in \mathbb{R}^{n_1 \times n_0}$, $\boldsymbol{U} \in \mathbb{R}^{n_2 \times n_1}$ & $\boldsymbol{b} \in \mathbb{R}^{n_1}$
- Forward computation:

First layer: $\boldsymbol{h} = \sigma(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$

Non-linear function (element-wise)

Second layer: $\boldsymbol{z} = \boldsymbol{U}\boldsymbol{h}$

Output: $\boldsymbol{y} = \text{softmax}(\boldsymbol{z})$

Convert to probability distribution

$$= \left[\frac{\exp(z_1)}{\sum_{j=1}^{n_2} \exp(z_j)}, \ldots, \frac{\exp(z_{n_2})}{\sum_{j=1}^{n_2} \exp(z_j)} \right]$$

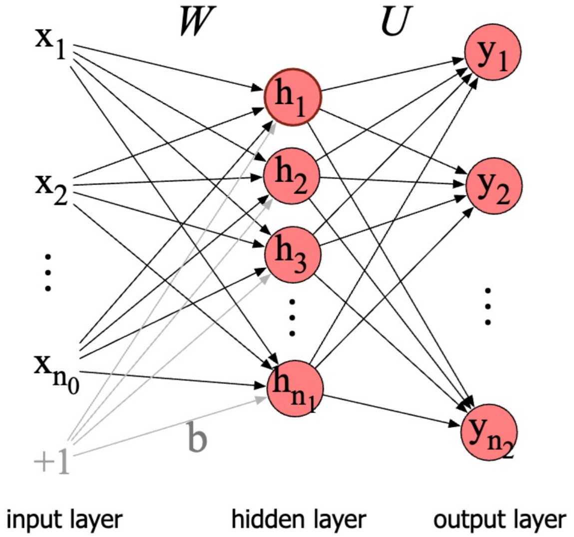


Figure source: https://web.stanford.edu/~jurafsky/slp3/13.pdf

(Recap) Training Objective

- We'll need a **loss function** that models the distance between the model output and the gold/desired output
- The common loss function for classification tasks is **cross-entropy** (CE) loss

K-way classification (K classes): $$\mathcal{L}_{\mathrm{CE}}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = -\sum_{k=1}^{K} y_k \log \hat{y}_k$$

Model output probability

Ground-truth probability

Usually a one-hot vector (one dimension is 1; others are 0): $\boldsymbol{y} = [0, \ldots, 1, \ldots, 0]$

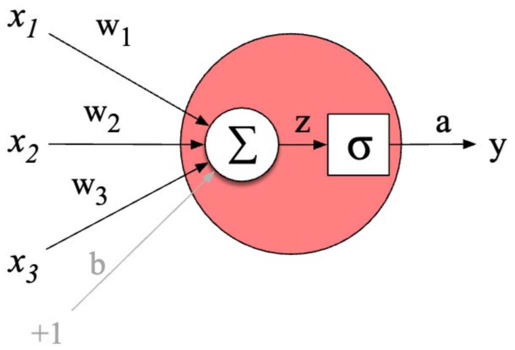
$$\mathcal{L}_{\mathrm{CE}}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = -\log \hat{y}_c = -\log \frac{\exp(z_c)}{\sum_{j=1}^{K} \exp(z_j)}$$

Also called "negative log likelihood (NLL) loss"

c is the ground-truth class

(Recap) Model Training (Forward Pass)

- Most optimization methods for DNNs are based on gradient descent
- First, randomly initialize model parameters
- In each optimization step, run two passes
  - **Forward pass**: evaluate the loss function given the input and current model parameters



$$z = \boldsymbol{w} \cdot \boldsymbol{x} + b$$
$$y = \sigma(z)$$

(Recap) Model Training (Backward Pass)

- Most optimization methods for DNNs are based on gradient descent
- First, randomly initialize model parameters
- In each optimization step, run two passes
  - **Forward pass**: evaluate the loss function given the input and current model parameters
  - **Backward pass:** update the parameters following the opposite direction of the gradient

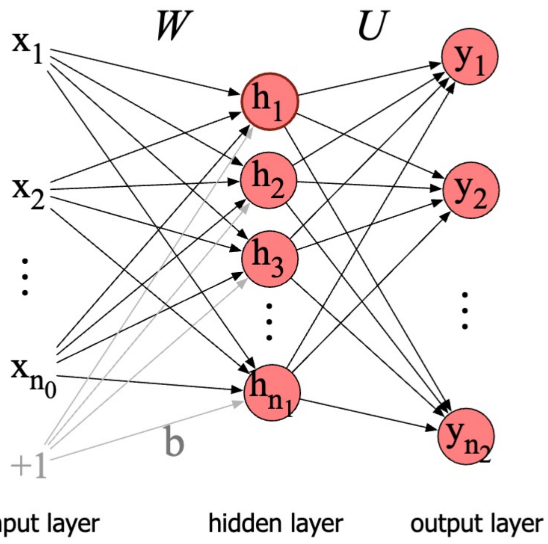
$$\boldsymbol{w}^{(t+1)} \leftarrow \boldsymbol{w}^{(t)} - \eta \nabla_{\boldsymbol{w}} \mathcal{L}(\hat{\boldsymbol{y}}, \boldsymbol{y})$$

- Gradient computed via the chain rule $\nabla_{\boldsymbol{w}} \mathcal{L}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \frac{\partial \mathcal{L}}{\partial \boldsymbol{w}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{y}} \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{z}} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{w}}$

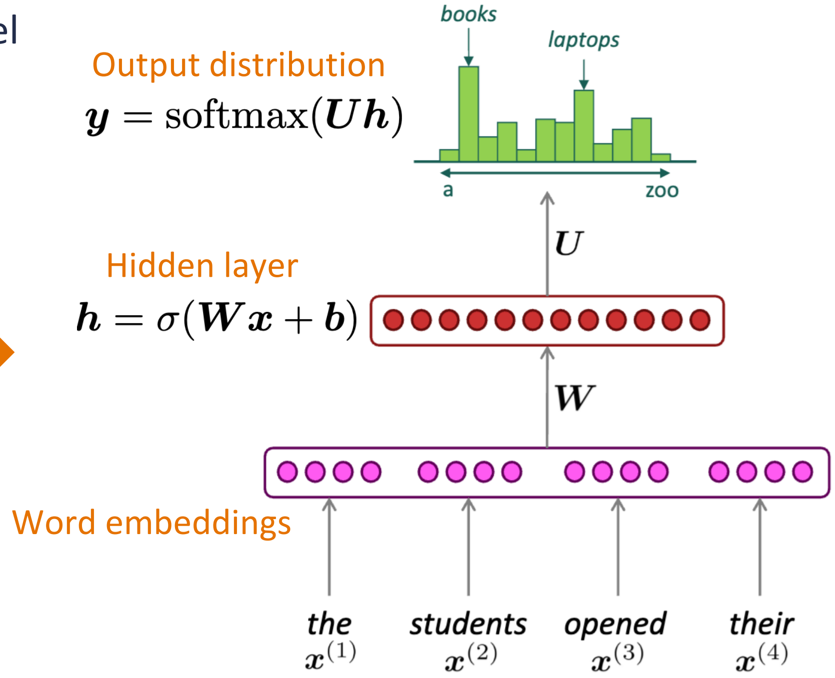
Gradient computation taken care of by deep learning libraries
(e.g., PyTorch)

(Recap) Simple Neural Language Model

Instantiate FFN as a neural language model



2-layer FFN



2-layer neural language model

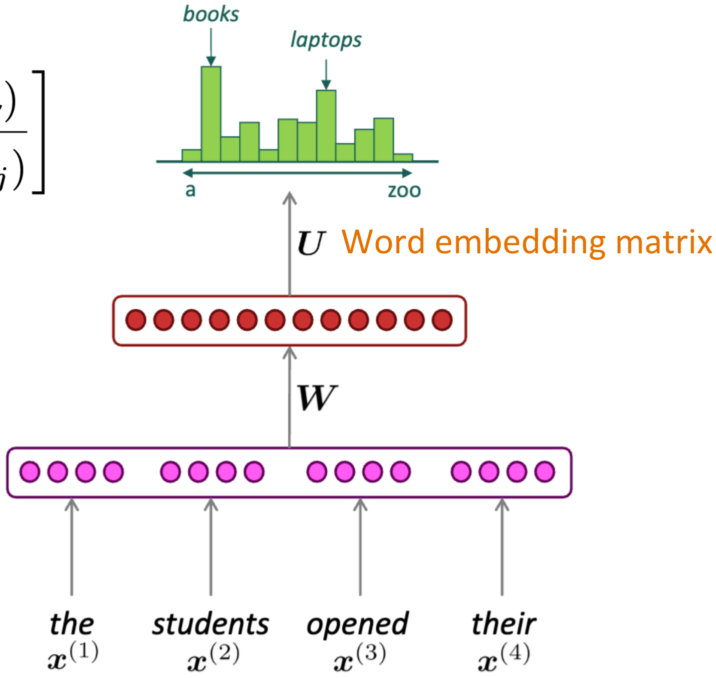
Figure source: https://web.stanford.edu/class/cs224n/slides/cs224n-spr2024-lecture05-rnnlm.pdf

(Recap) Benefits of Neural Language Models

Output distribution

$$\boldsymbol{y} = \text{softmax}(\boldsymbol{U}\boldsymbol{h}) = \Bigg[\underbrace{\frac{\exp(\boldsymbol{u}_1 \cdot \boldsymbol{h})}{\sum_{j=1}^{|\mathcal{V}|}\exp(z_j)}, \ldots, \frac{\exp(\boldsymbol{u}_{|\mathcal{V}|} \cdot \boldsymbol{h})}{\sum_{j=1}^{|\mathcal{V}|}\exp(z_j)}}_{|\mathcal{V}|\text{-dimensions}}\Bigg]$$

- Address sparsity issue:
    - Strictly positive probability on every token in the vocabulary
    - Semantically similar words tend to have similar probabilities



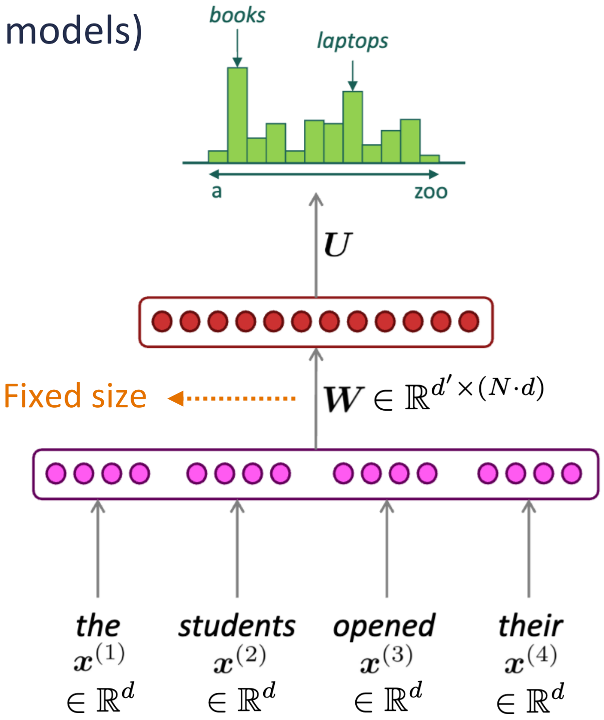
UNIVERSITY of VIRGINIA

(Recap) Limitations of (Simple) Neural Language Models

- Context window is fixed (same as N-gram language models)
- Increasing N will enlarge $\boldsymbol{W}$

Concatenated word embeddings

$$\boldsymbol{x} = \boldsymbol{x}^{(1)} \oplus \boldsymbol{x}^{(2)} \oplus \cdots \oplus \boldsymbol{x}^{(N)} \in \mathbb{R}^{N \cdot d}$$

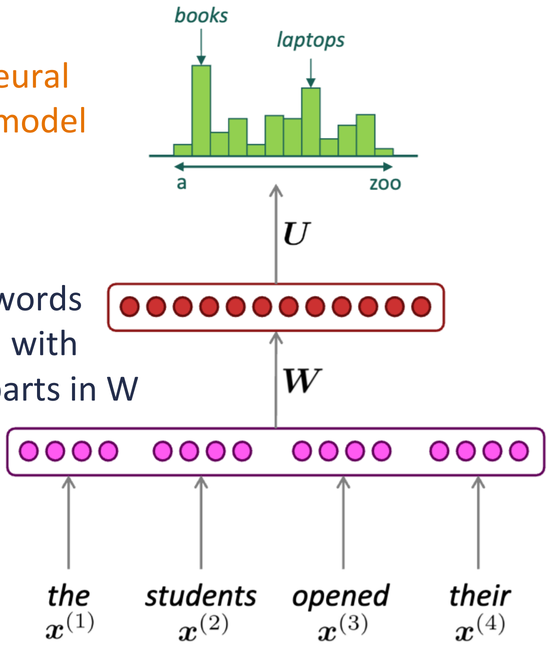


(Recap) Recurrent Neural Network (RNN) Overview

A neural language model that can process inputs of arbitrary lengths

Simple neural language model

Different words multiplied with different subparts in W



Recurrent neural language model

Reuse the same weights for all words

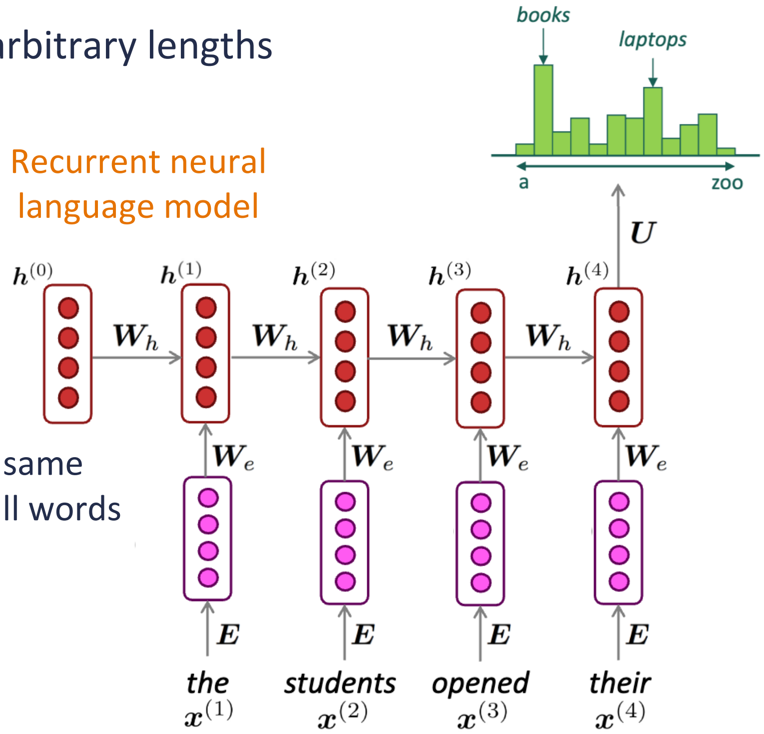


Figure source: https://web.stanford.edu/class/cs224n/slides/cs224n-spr2024-lecture05-rnnlm.pdf

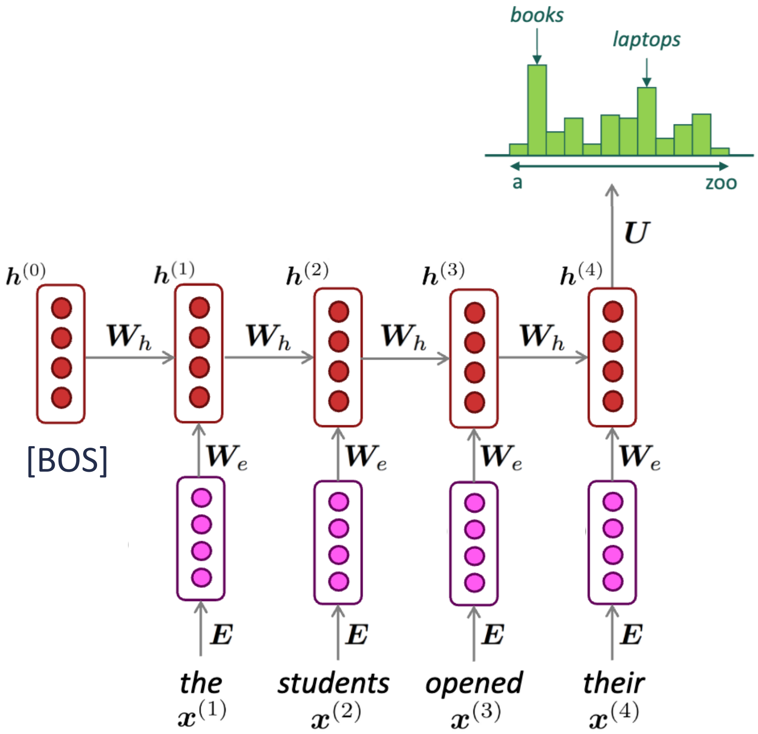
(Recap) RNN Computation

- Hidden states in RNNs are computed based on
  - The hidden state at the previous step (memory)
  - The word embedding at the current step
- Parameters:
  - $\boldsymbol{W}_h$: weight matrix for the recurrent connection
  - $\boldsymbol{W}_e$: weight matrix for the input connection

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

Hidden states at the previous word (time step)

Word embedding of the current word (time step)



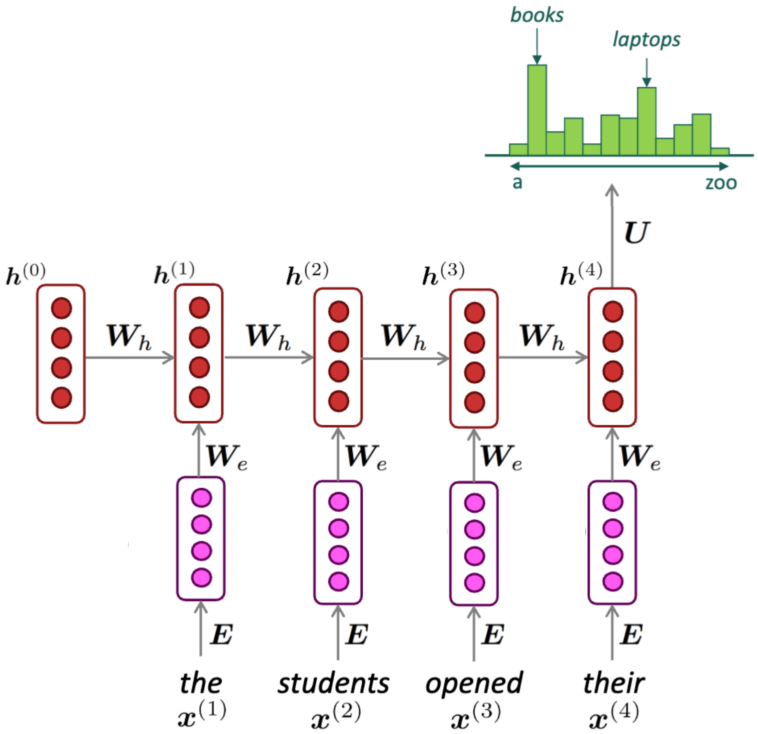
(Recap) RNN Computation

- Input: $\boldsymbol{x} = [\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \cdots, \boldsymbol{x}^{(N)}]$
- Initialize $\boldsymbol{h}^{(0)}$
- For each time step (word) in the input:
  - Compute hidden states:

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

  - Compute output:

$$\boldsymbol{y}^{(t)} = \text{softmax}\left(\boldsymbol{U}\boldsymbol{h}^{(t)}\right)$$



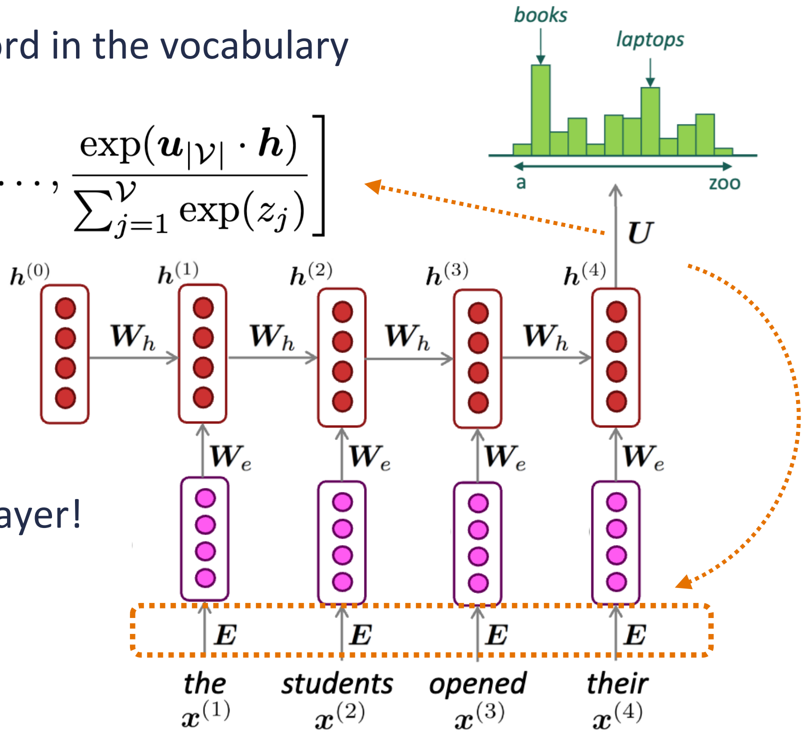
(Recap) RNN Weight Tying

- Role of matrix $\boldsymbol{U}$: score the likelihood of each word in the vocabulary

$$\boldsymbol{y} = \mathrm{softmax}(\boldsymbol{U}\boldsymbol{h}) = \left[\frac{\exp(\boldsymbol{u}_1 \cdot \boldsymbol{h})}{\sum_{j=1}^{|\mathcal{V}|} \exp(z_j)}, \ldots, \frac{\exp(\boldsymbol{u}_{|\mathcal{V}|} \cdot \boldsymbol{h})}{\sum_{j=1}^{\mathcal{V}} \exp(z_j)}\right]$$

$$\boldsymbol{U} \in \mathbb{R}^{|\mathcal{V}| \times d}$$

Same dimensionality of the word embedding matrix!



- Use the same input embeddings in the softmax layer!
- Weight tying benefits:
  - Improve learning efficiency & effectiveness
  - Reduce the number of parameters in the model

(Recap) RNN for Language Modeling

- Recall that language modeling predicts the next word given previous words

$$p(\boldsymbol{x}) = p\left(x^{(1)}\right) p\left(x^{(2)} | x^{(1)}\right) \cdots p\left(x^{(n)} | x^{(1)}, \ldots, x^{(n-1)}\right) = \prod_{t=1}^{n} p\left(x^{(t)} | x^{(1)}, \ldots, x^{(t-1)}\right)$$

- How to use RNNs to represent $p\left(x^{(t)} | x^{(1)}, \ldots, x^{(t-1)}\right)$?

Output probability at (t-1) step: $\boldsymbol{y}^{(t-1)} = \mathrm{softmax}\left(\boldsymbol{U}\boldsymbol{h}^{(t-1)}\right) := f\left(\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(t-2)}, \boldsymbol{x}^{(t-1)}\right)$

$\boldsymbol{h}^{(t-1)}$ is a function of $\boldsymbol{h}^{(t-2)}$ and $\boldsymbol{x}^{(t-1)}$: $\boldsymbol{h}^{(t-1)} = \sigma\left(\boldsymbol{W}_h \boldsymbol{h}^{(t-2)} + \boldsymbol{W}_e \boldsymbol{x}^{(t-1)}\right) := g\left(\boldsymbol{h}^{(t-2)}, \boldsymbol{x}^{(t-1)}\right)$

$\boldsymbol{h}^{(t-2)}$ is a function of $\boldsymbol{h}^{(t-3)}$ and $\boldsymbol{x}^{(t-2)}$: $\boldsymbol{h}^{(t-2)} = \sigma\left(\boldsymbol{W}_h \boldsymbol{h}^{(t-3)} + \boldsymbol{W}_e \boldsymbol{x}^{(t-2)}\right) := g\left(\boldsymbol{h}^{(t-3)}, \boldsymbol{x}^{(t-2)}\right)$

$$\vdots \qquad\qquad \vdots$$

$\boldsymbol{h}^{(1)}$ is a function of $\boldsymbol{h}^{(0)}$ and $\boldsymbol{x}^{(1)}$: $\boldsymbol{h}^{(1)} = \sigma\left(\boldsymbol{W}_h \boldsymbol{h}^{(0)} + \boldsymbol{W}_e \boldsymbol{x}^{(1)}\right) := g\left(\boldsymbol{h}^{(0)}, \boldsymbol{x}^{(1)}\right)$

(Recap) RNN Language Model Training

Train the output probability at each time step to predict the next word

$$\mathcal{L}_{\mathrm{LM}}(\boldsymbol{x}) = \frac{1}{n}\sum_{t=1}^{n}\mathcal{L}_{\mathrm{CE}}\left(\hat{\boldsymbol{y}}^{(t)}, \boldsymbol{y}^{(t)}\right) = \frac{1}{n}\sum_{t=1}^{n} -\log \hat{y}^{(t)}_{x^{(t)}} = \frac{1}{n}\sum_{t=1}^{n} -\log \frac{\exp\left(x^{(t)}\right)}{\sum_{w'\in\mathcal{V}}\exp(w')}$$

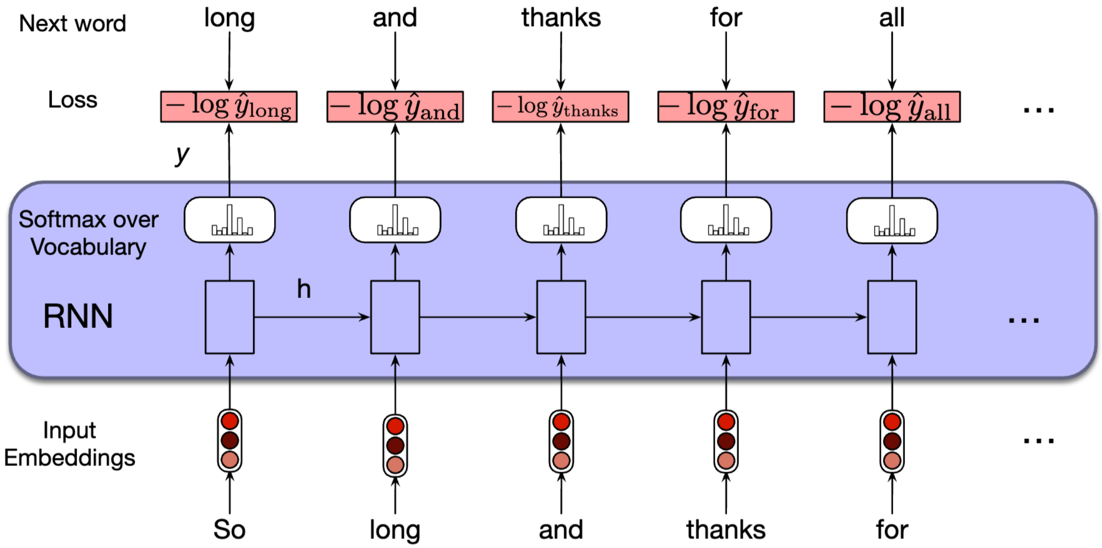


Figure source: https://web.stanford.edu/~jurafsky/slp3/13.pdf

(Recap) RNN for Text Generation

- Input [BOS] (beginning-of-sequence) token to the model
- Sample a word from the softmax distribution at the first time step
- Use the word embedding of that first word as the input at the next time step
- Repeat until the [EOS] (end-of-sequence) token is generated

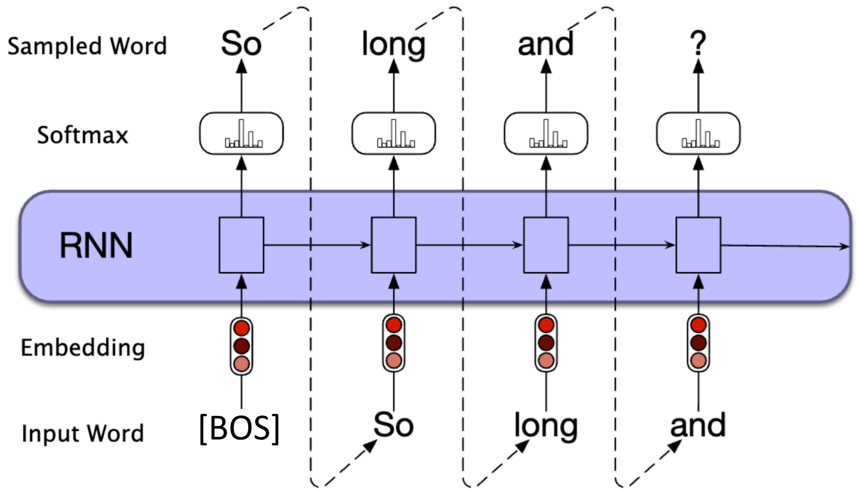


Figure source: https://web.stanford.edu/~jurafsky/slp3/13.pdf

Agenda

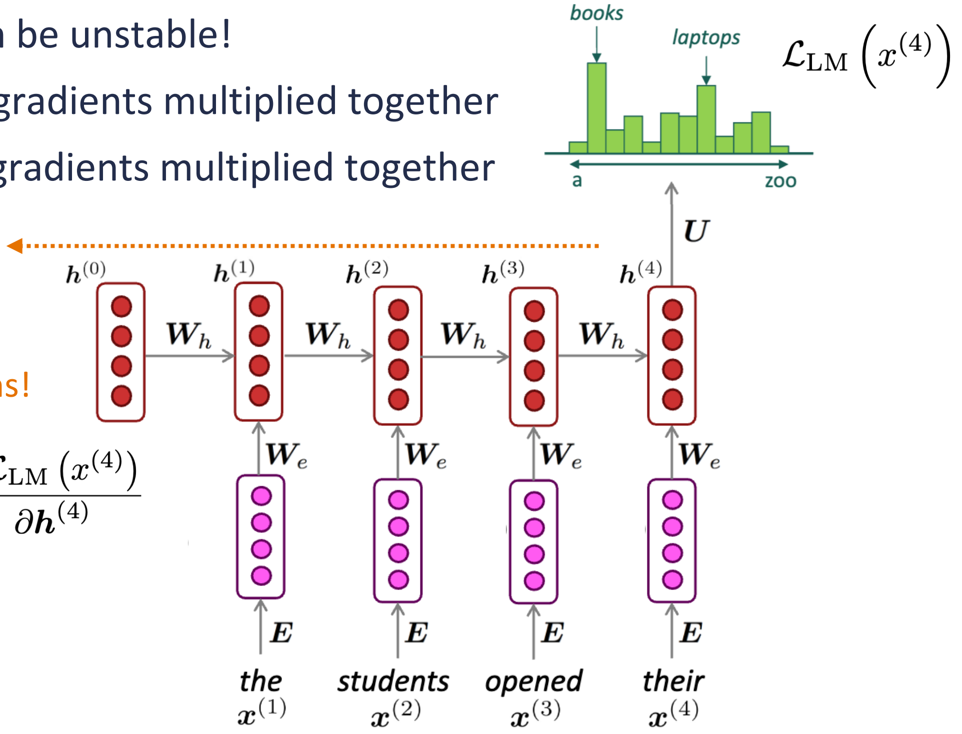
- RNN Limitations
- Advanced RNNs
- Transformer Overview
- Self-Attention

UNIVERSITY of VIRGINIA

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

Gradient backpropagation

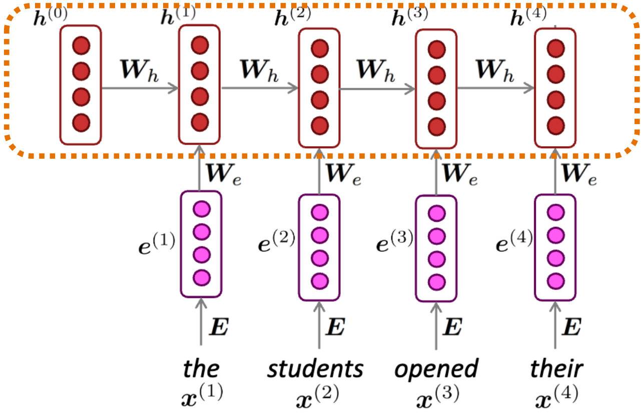


Lots of gradient multiplications!

$$\frac{\partial \mathcal{L}_{\mathrm{LM}}\left(x^{(4)}\right)}{\partial \boldsymbol{h}^{(0)}} = \boxed{\frac{\partial \boldsymbol{h}^{(1)}}{\partial \boldsymbol{h}^{(0)}} \frac{\partial \boldsymbol{h}^{(2)}}{\partial \boldsymbol{h}^{(1)}} \frac{\partial \boldsymbol{h}^{(3)}}{\partial \boldsymbol{h}^{(2)}} \frac{\partial \boldsymbol{h}^{(4)}}{\partial \boldsymbol{h}^{(3)}}} \frac{\partial \mathcal{L}_{\mathrm{LM}}\left(x^{(4)}\right)}{\partial \boldsymbol{h}^{(4)}}$$

Difficulty in Capturing Long-Term Dependencies

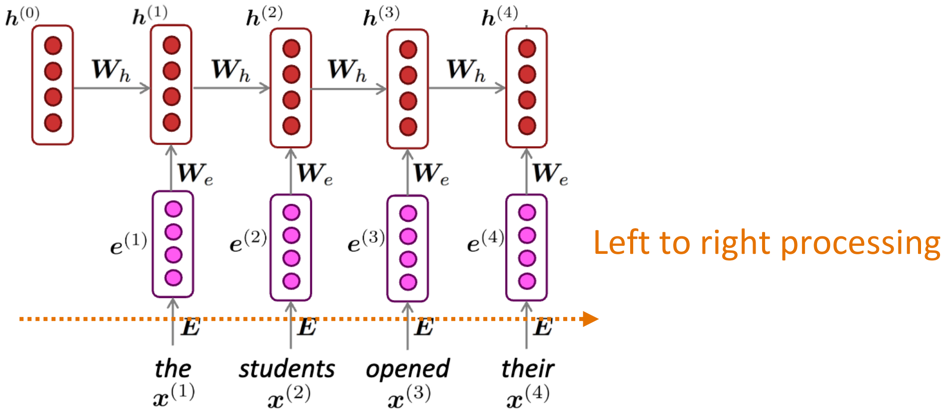
- RNNs are theoretically capable of remembering information over arbitrary lengths of input, but they struggle in practice with long-term dependencies
- RNNs use a fixed-size hidden state to encode an entire sequence of variable length; the hidden state is required to compress a lot of information
- 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!

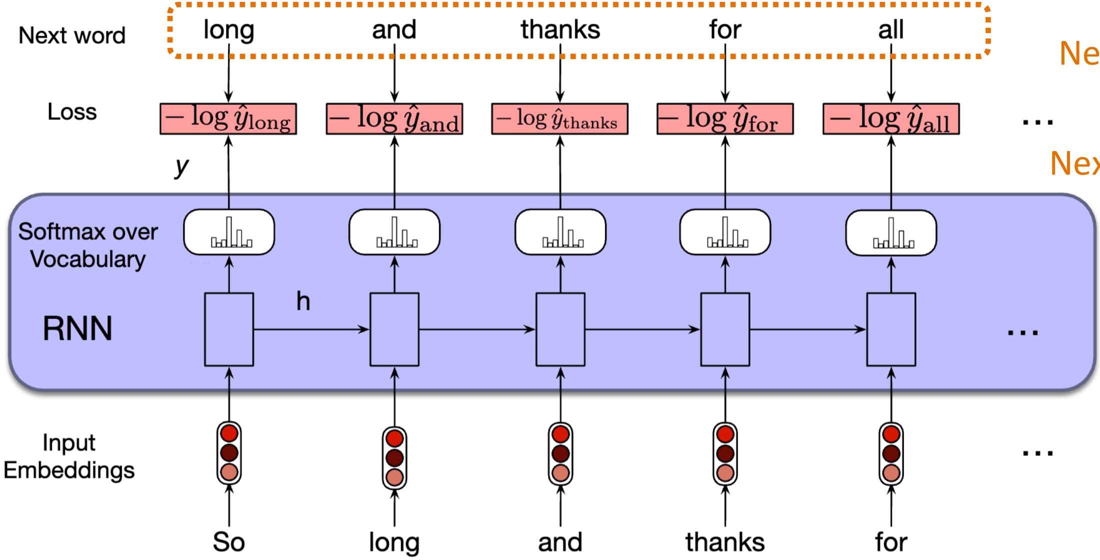
Lack of Bidirectionality

- 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



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

Agenda

- RNN Limitations
- Advanced RNNs
- Transformer Overview
- Self-Attention

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

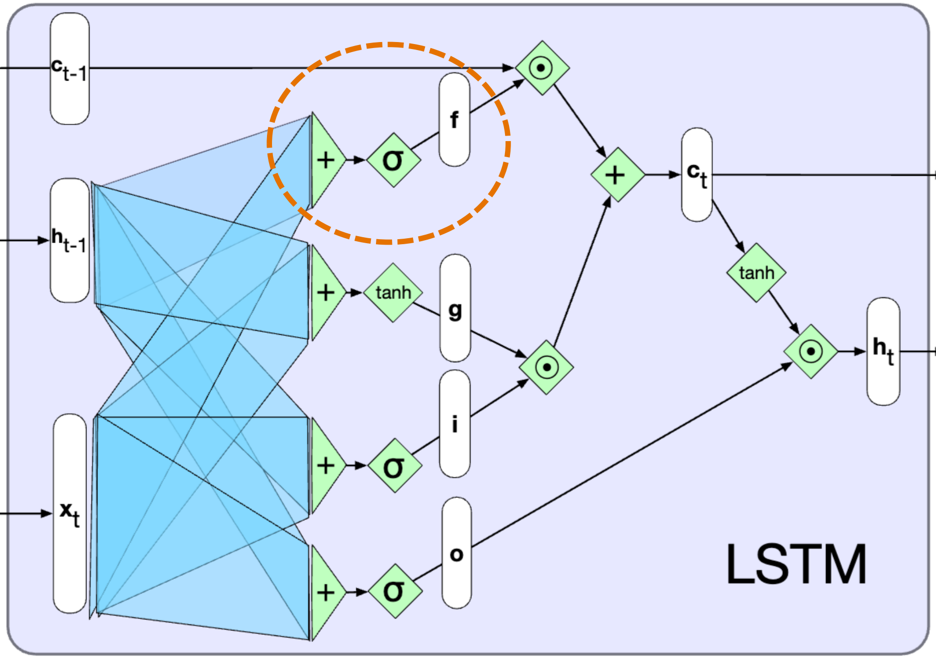
LSTM Computation (Forget Gate)

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

Cell state $\boldsymbol{c}^{(t-1)}$

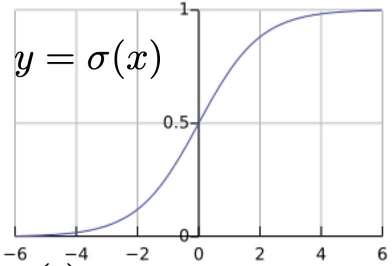
Hidden state $\boldsymbol{h}^{(t-1)}$

Word embedding $\boldsymbol{x}^{(t)}$



$\boldsymbol{c}^{(t)}$

$\boldsymbol{h}^{(t)}$



Decides what information to discard from the cell state

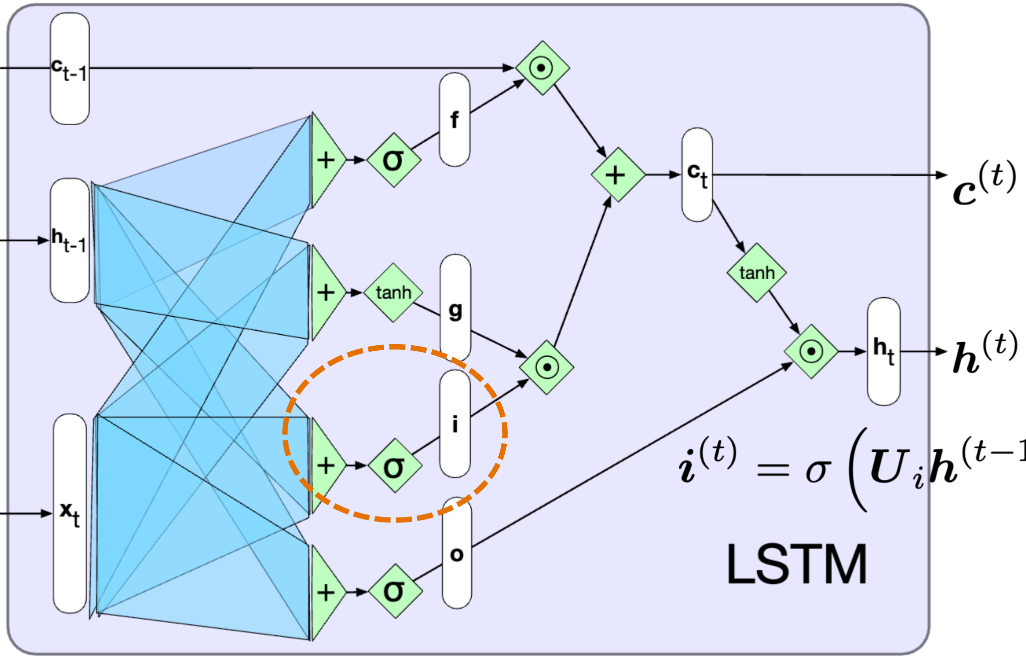
Figure source: https://web.stanford.edu/~jurafsky/slp3/13.pdf

LSTM Computation (Add/Input Gate)

Cell state $\boldsymbol{c}^{(t-1)}$

Hidden state $\boldsymbol{h}^{(t-1)}$

Word embedding $\boldsymbol{x}^{(t)}$



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

Decides what new information to store to the cell state

Figure source: https://web.stanford.edu/~jurafsky/slp3/13.pdf

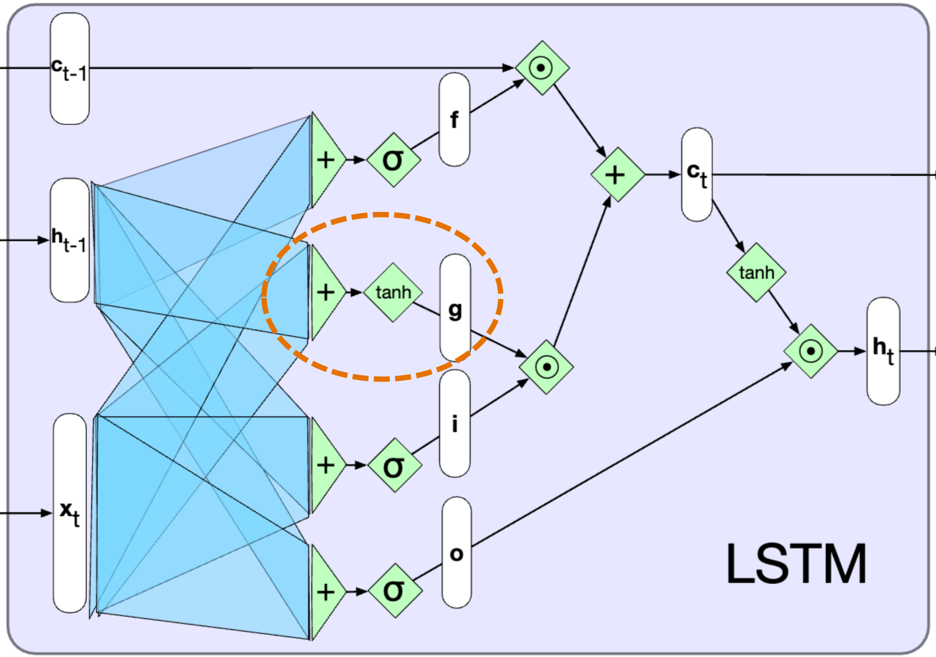
LSTM Computation (Candidate Cell State)

$$\boldsymbol{g}^{(t)} = \tanh\left(\boldsymbol{U}_g \boldsymbol{h}^{(t-1)} + \boldsymbol{W}_g \boldsymbol{x}^{(t)}\right)$$

Cell state $\boldsymbol{c}^{(t-1)}$

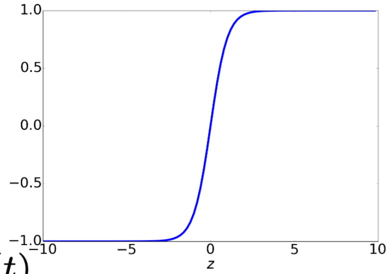
Hidden state $\boldsymbol{h}^{(t-1)}$

Word embedding $\boldsymbol{x}^{(t)}$



$\boldsymbol{c}^{(t)}$

$\boldsymbol{h}^{(t)}$



$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

Compute information needed from the previous hidden state and current inputs

Figure source: https://web.stanford.edu/~jurafsky/slp3/13.pdf

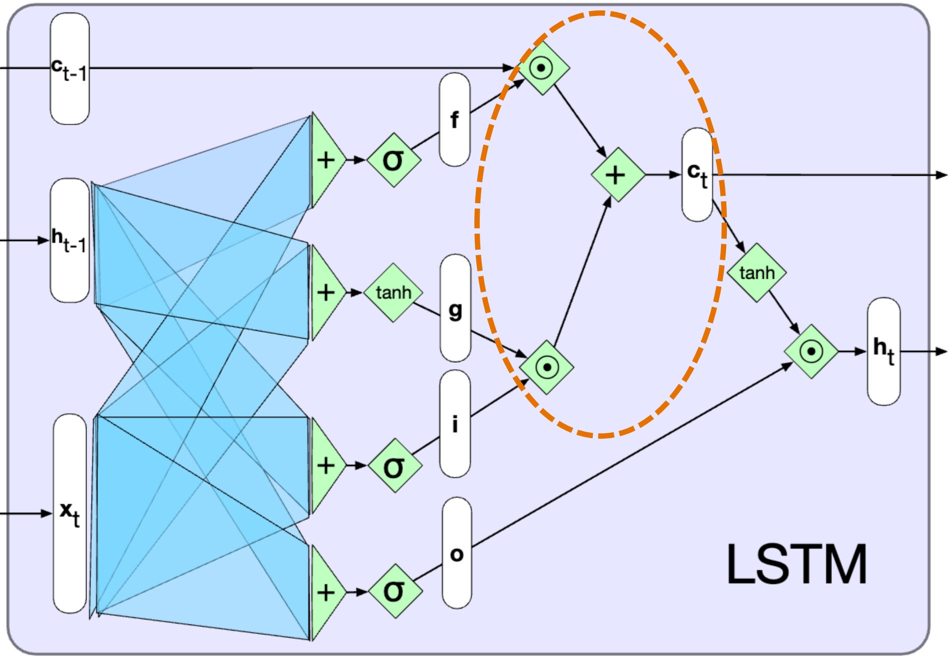
LSTM Computation (Cell State Update)

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

Cell state $\boldsymbol{c}^{(t-1)}$

Hidden state $\boldsymbol{h}^{(t-1)}$

Word embedding $\boldsymbol{x}^{(t)}$



$\boldsymbol{c}^{(t)}$

$\boldsymbol{h}^{(t)}$

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

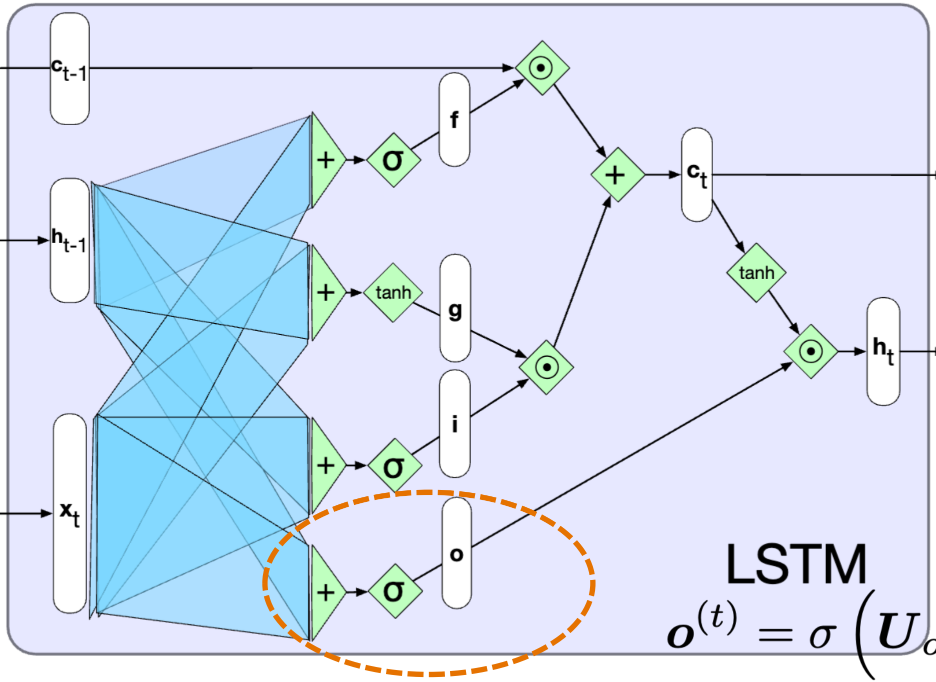
Figure source: https://web.stanford.edu/~jurafsky/slp3/13.pdf

LSTM Computation (Output Gate)

Cell state $\boldsymbol{c}^{(t-1)}$

Hidden state $\boldsymbol{h}^{(t-1)}$

Word embedding $\boldsymbol{x}^{(t)}$



$\boldsymbol{c}^{(t)}$

$\boldsymbol{h}^{(t)}$

Decides what parts of the cell state will be output

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

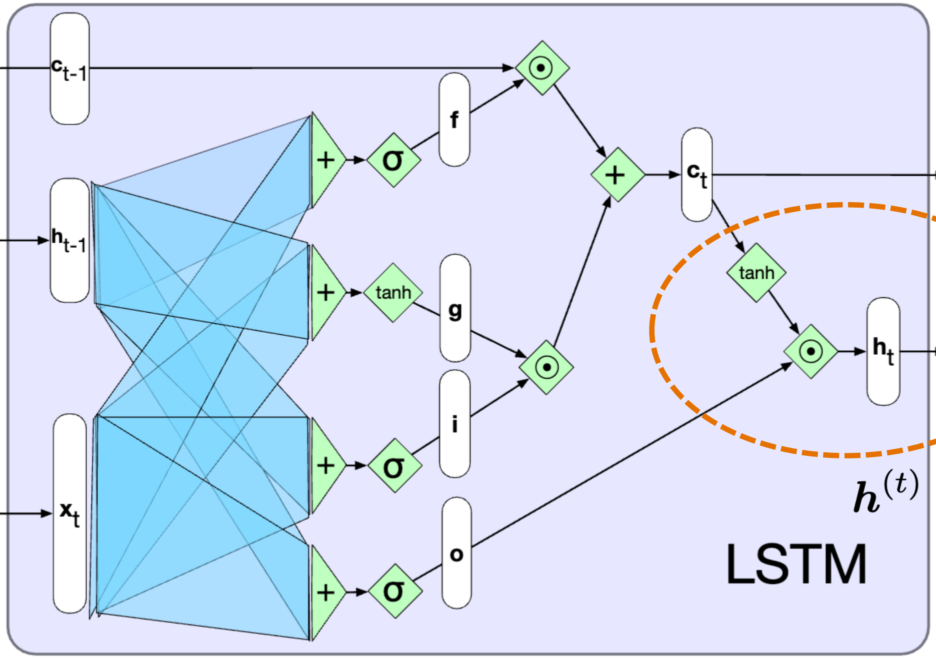
Figure source: https://web.stanford.edu/~jurafsky/slp3/13.pdf

LSTM Computation (Hidden State Update)

Cell state $\boldsymbol{c}^{(t-1)}$

Hidden state $\boldsymbol{h}^{(t-1)}$

Word embedding $\boldsymbol{x}^{(t)}$



$\boldsymbol{c}^{(t)}$

$\boldsymbol{h}^{(t)}$

$$\boldsymbol{h}^{(t)} = \boldsymbol{o}^{(t)} \odot \tanh\left(\boldsymbol{c}^{(t)}\right)$$

Hidden state updated using the output gate & the updated cell state

Figure source: https://web.stanford.edu/~jurafsky/slp3/13.pdf

Bidirectional RNNs

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

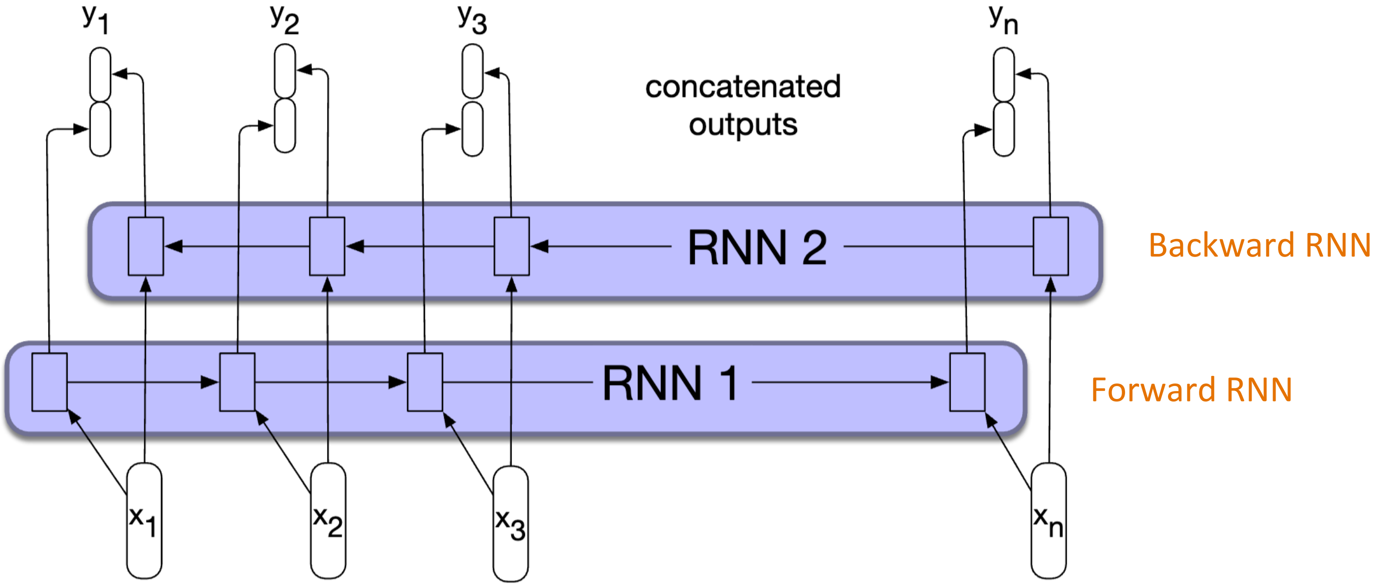


Figure source: https://web.stanford.edu/~jurafsky/slp3/13.pdf

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

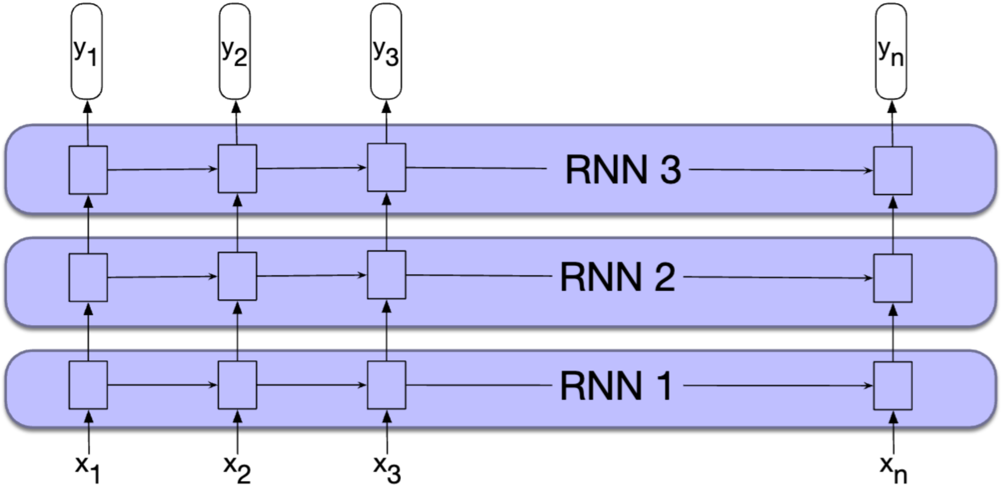


Figure source: https://web.stanford.edu/~jurafsky/slp3/13.pdf

UNIVERSITY of VIRGINIA

Summary: Sequence Modeling

- Sequence modeling goals:
  - Learn context-dependent representations
  - Capture long-range dependencies
  - Handle complex relationships among large text units
- Use deep learning architectures to understand, process, and generate text sequences
- Why DNNs?
  - The multi-layer structure in DNNs mirrors the hierarchical structures in language
  - DNNs learn multiple levels of semantics across layers: low-level patterns (e.g., relations between words) in lower layers & high-level patterns (e.g., sentence meanings) in higher layers

Summary: Neural Language Models

- Address the sparsity issue in N-gram language models by computing the output distribution based on distributed representations (with semantic information)
- Simple neural language models based on feedforward networks suffer from the fixed context window issue
  - Can only model a fixed number of words (similar to N-gram assumption)
  - Increasing the context window requires adding more model parameters

Summary: Recurrent Neural Networks

- General idea: Use the same set of model weights to process all input words
- RNNs as language models
  - Theoretically able to process infinitely long sequences
  - Practically can only keep track of recent contexts
- Training issues: vanishing & exploding gradients
- LSTM is a prominent RNN variant to keep track of both long-term and short-term memories via multiple gates

Agenda

- RNN Limitations
- Advanced RNNs
- Transformer Overview
- Self-Attention

UNIVERSITY of VIRGINIA

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

Łukasz Kaiser* Google Brain lukaszkaiser@google.com

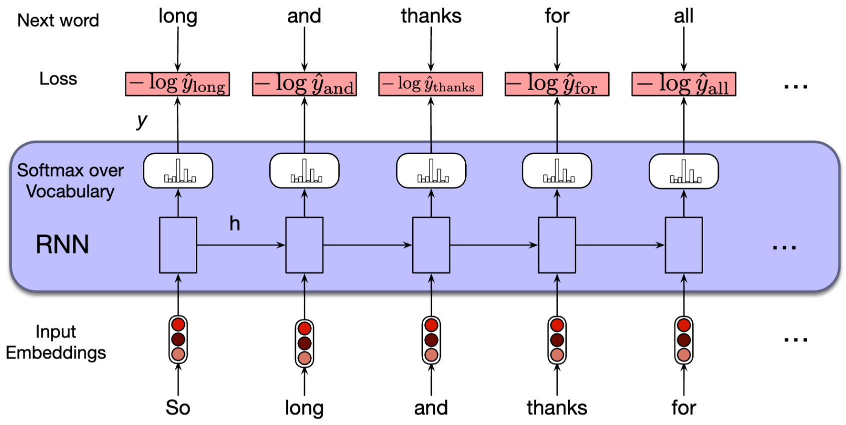
Illia Polosukhin* ‡ illia.polosukhin@gmail.com

Transformer: https://arxiv.org/pdf/1706.03762

UNIVERSITY of VIRGINIA

Transformer vs. RNN

RNN
(recurrent computations)



Transformer
(self-attention computations)

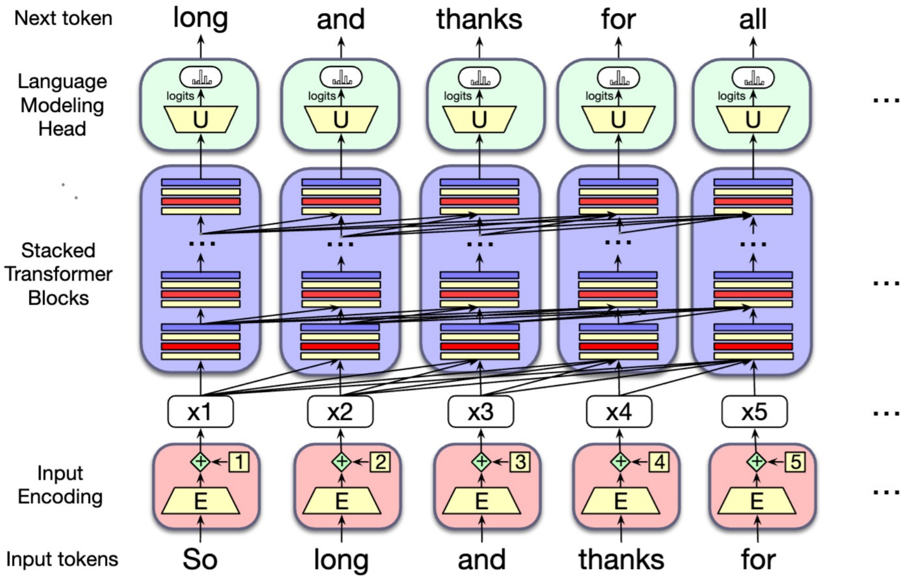


Figure source: https://web.stanford.edu/~jurafsky/slp3/8.pdf

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

Transformer Layer

Each Transformer layer contains the following important components:

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

Transformer layer

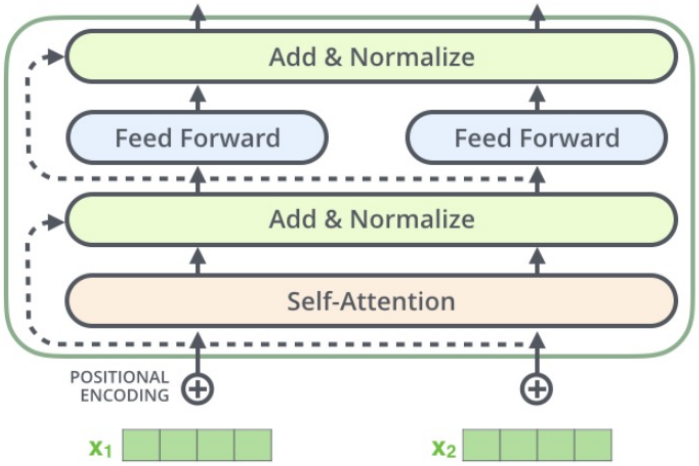


Figure source: https://jalammar.github.io/illustrated-transformer/

Agenda

- RNN Limitations
- Advanced RNNs
- Transformer Overview
- Self-Attention

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

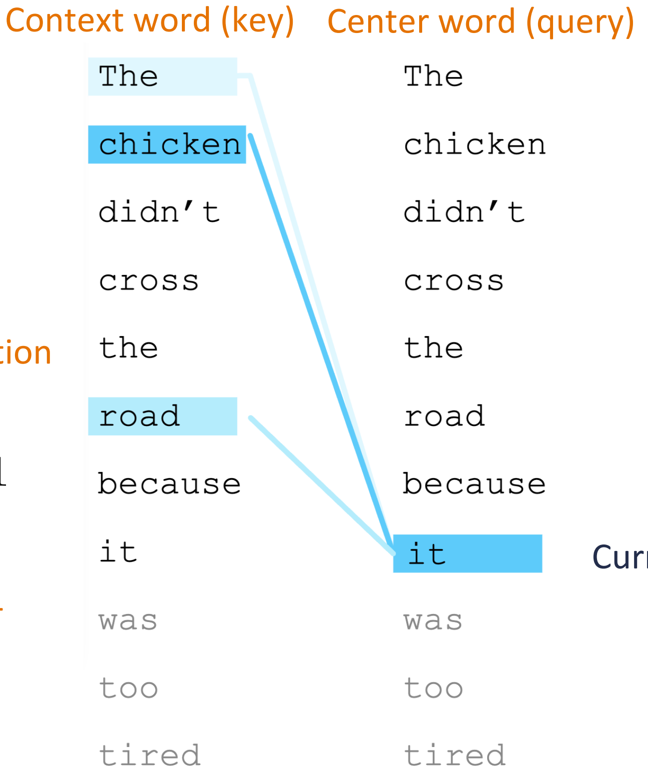
Derive the center word representation as a weighted sum of context representations!

Center word representation

Context word representation

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

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



Current word = "it"

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

Self-Attention: Attention Score Computation

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

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

$$\alpha_{ij} = \mathrm{Softmax}(\boldsymbol{x}_i \cdot \boldsymbol{x}_j)$$

Center word (query) representation

Context word (key) representation

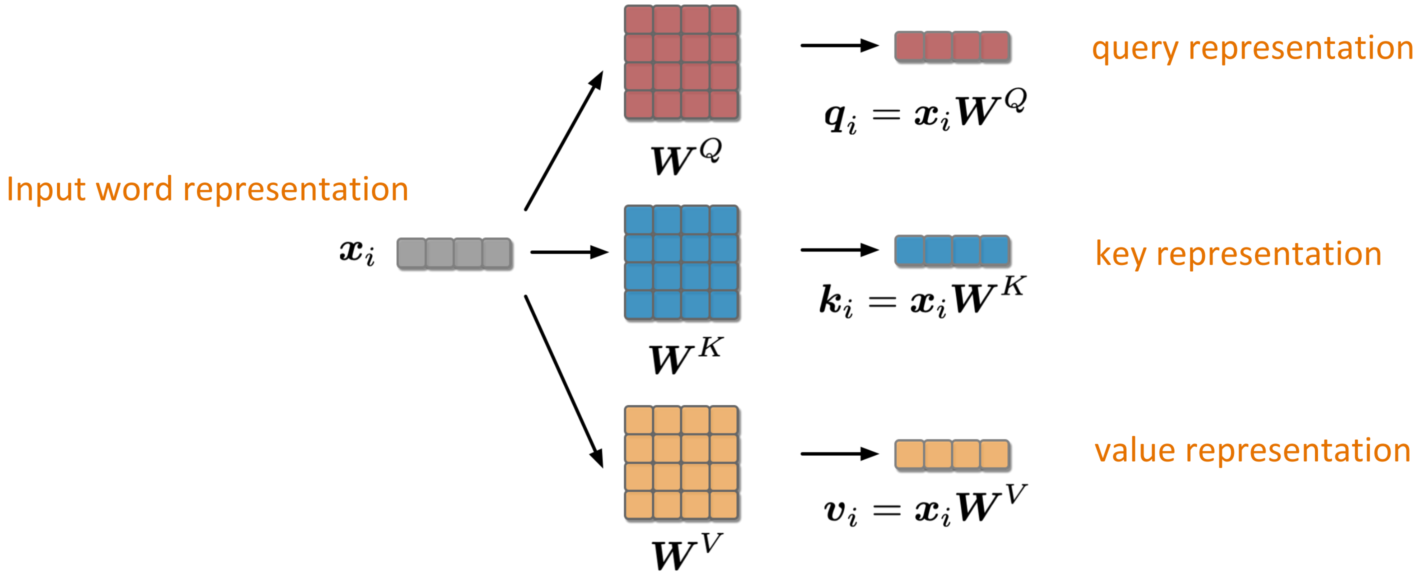
- Why use two copies of word representations for attention computation?
  - We want to reflect the different roles a word plays (as the target word being compared to others, or as the context word being compared to the target word)
  - If using the same copy of representations for attention calculation, 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 $\boldsymbol{x}_i$
- Compute Q, K, V representations for each word:

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

- 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

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

Dimensionality of q and k

- Sum the value vectors weighted by attention scores

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

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

Output of self-attention

Sum the weighted value vectors

Obtain attention scores via softmax

Divide the dot product by vector dimension

Compare x_3's query with the keys of all words

Compute query, key, value

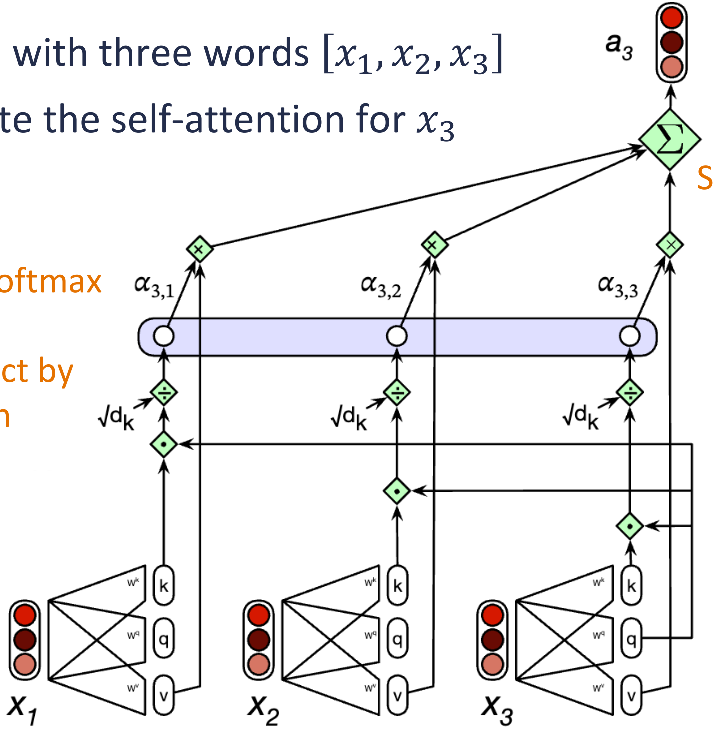
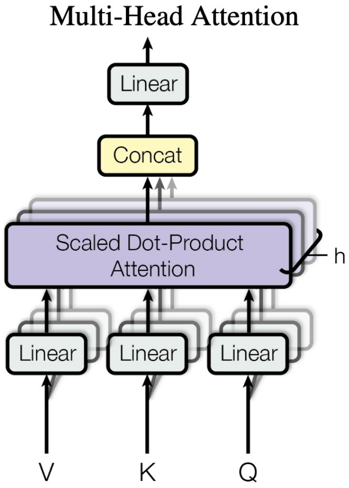


Figure source: https://web.stanford.edu/~jurafsky/slp3/8.pdf

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



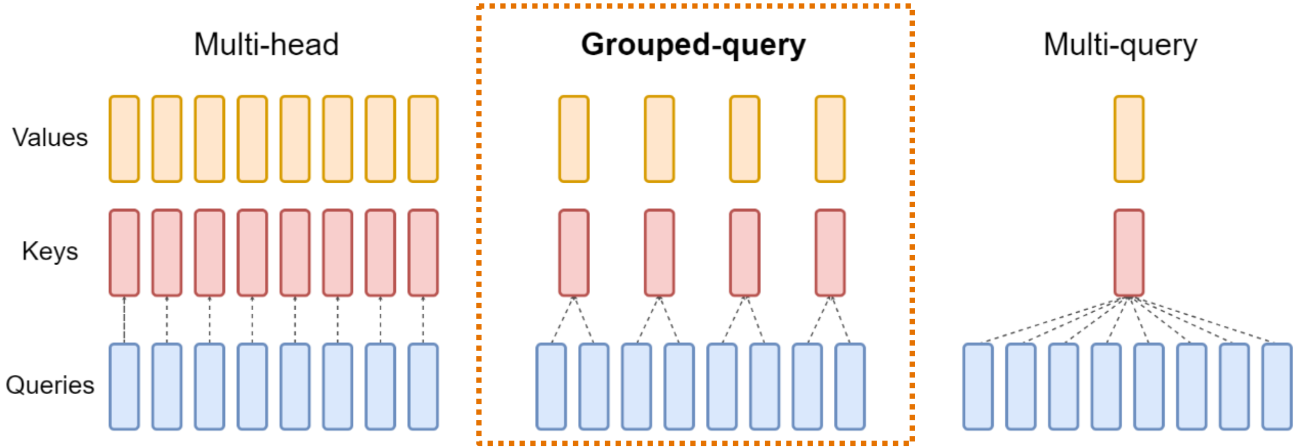
Concatenate the outputs of all heads

h attention heads are computed separately

Figure source: https://arxiv.org/pdf/1706.03762

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: $\boldsymbol{q}_i = \boldsymbol{x}_i \boldsymbol{W}^Q \quad \boldsymbol{k}_i = \boldsymbol{x}_i \boldsymbol{W}^K \quad \boldsymbol{v}_i = \boldsymbol{x}_i \boldsymbol{W}^V \in \mathbb{R}^d$

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

$$\boldsymbol{X} = \begin{bmatrix} \text{———} & \boldsymbol{x}_1 & \text{———} \\ \text{———} & \boldsymbol{x}_2 & \text{———} \\ \cdots & \cdots & \cdots \\ \text{———} & \boldsymbol{x}_N & \text{———} \end{bmatrix}$$

UNIVERSITY of VIRGINIA

Parallel Computation of Attention

Attention computation can also be written in matrix form

Compute attention for one word: $\boldsymbol{a}_i = \mathrm{Softmax}\left(\frac{\boldsymbol{q}_i \cdot \boldsymbol{k}_j}{\sqrt{d}}\right) \cdot \boldsymbol{v}_j$

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

Attention matrix

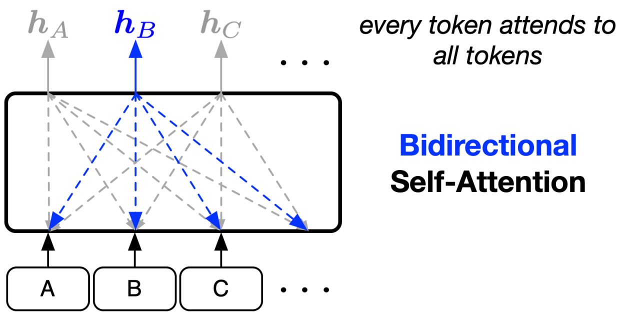
	q1•k1	q1•k2	q1•k3	q1•k4
N	q2•k1	q2•k2	q2•k3	q2•k4
	q3•k1	q3•k2	q3•k3	q3•k4
	q4•k1	q4•k2	q4•k3	q4•k4

N

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

Bidirectional vs. Unidirectional Self-Attention

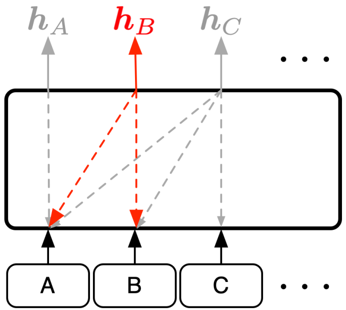
- Self-attention can capture different context dependencies
- **Bidirectional** self-attention:
  - Each position to attend to all other positions in the input sequence
  - Transformers with bidirectional self-attention are called Transformer **encoders** (e.g., BERT)
  - Use case: natural language understanding (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 -inf



every token attends to its previous tokens

Unidirectional Self-Attention

N

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

N

Position Encoding

- Motivation: inject positional information to input vectors

$$\boldsymbol{q}_i = \boldsymbol{x}_i \boldsymbol{W}^Q \quad \boldsymbol{k}_i = \boldsymbol{x}_i \boldsymbol{W}^K \quad \boldsymbol{v}_i = \boldsymbol{x}_i \boldsymbol{W}^V \in \mathbb{R}^d$$

$$\boldsymbol{a}_i = \mathrm{Softmax}\left(\frac{\boldsymbol{q}_i \cdot \boldsymbol{k}_j}{\sqrt{d}}\right) \cdot \boldsymbol{v}_j$$

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

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

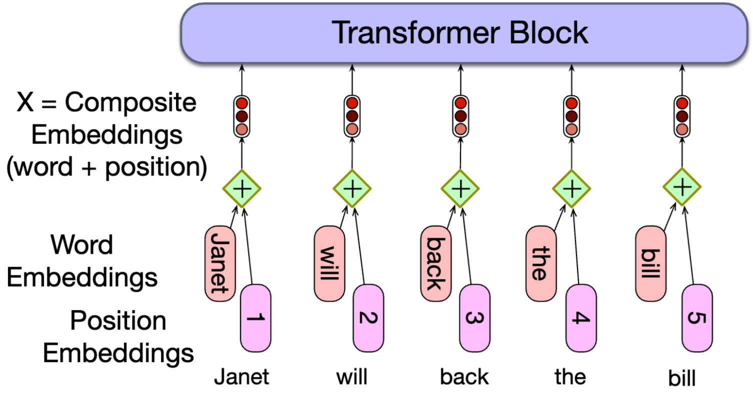


Figure source: https://web.stanford.edu/~jurafsky/slp3/8.pdf

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

Thank You!

Yu Meng
University of Virginia
yumeng5@virginia.edu