



Transformer Language Models

Yu Meng

University of Virginia
yumeng5@virginia.edu

Oct 07, 2024

Reminders

- Assignment 3 due this Friday!
- Project midterm proposal guideline released (due 10/18):
https://docs.google.com/document/d/12-f2KQRH2kYBohxJLj_E6gzfj1vulmnuEVBbyXBAiY/edit?usp=sharing

Join at
slido.com
#8315 018





Overview of Course Contents

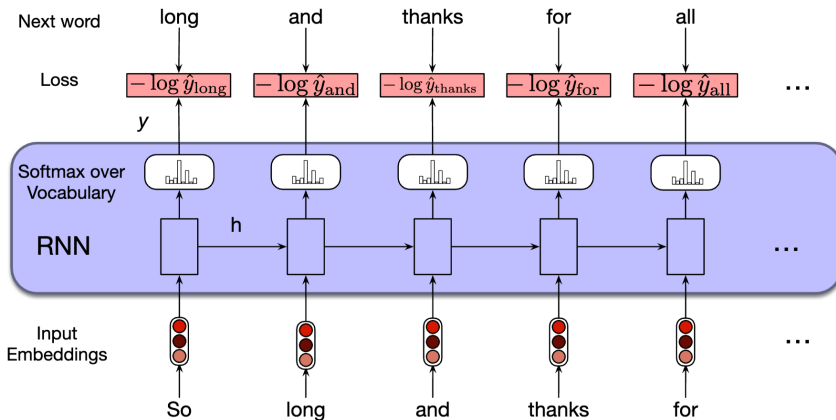
- Week 1: Logistics & Overview
- Week 2: N-gram Language Models
- Week 3: Word Senses, Semantics & Classic Word Representations
- Week 4: Word Embeddings
- Week 5: Sequence Modeling and Neural Language Models
- **Week 6-7: Language Modeling with Transformers (Pretraining + Fine-tuning)**
- Week 8: Large Language Models (LLMs) & In-context Learning
- Week 9-10: Knowledge in LLMs and Retrieval-Augmented Generation (RAG)
- Week 11: LLM Alignment
- Week 12: Language Agents
- Week 13: Recap + Future of NLP
- Week 15 (after Thanksgiving): Project Presentations



(Recap) Transformer vs. RNN

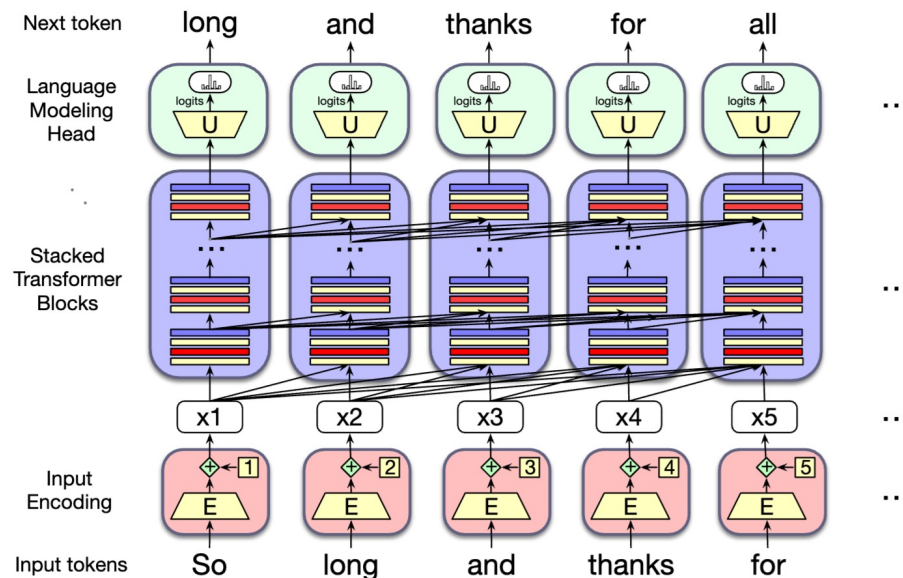
RNN

(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

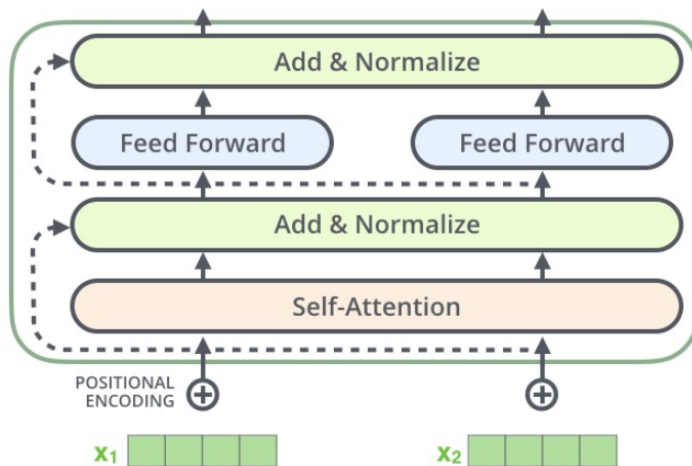


(Recap) Transformer Layer

Each Transformer layer contains the following important components:

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

Transformer layer





(Recap) Self-Attention: 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"



(Recap) 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



(Recap) 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

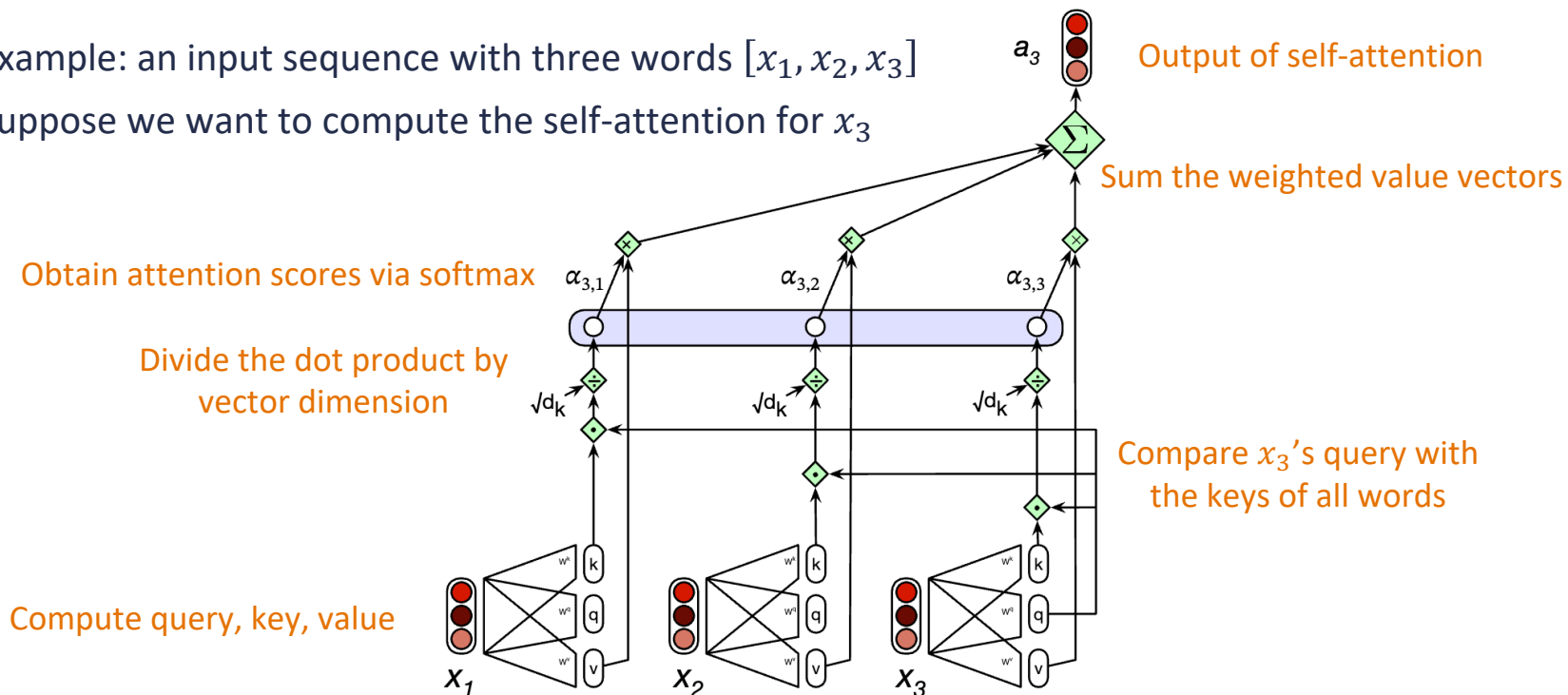
- Sum the value vectors weighted by attention scores

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



(Recap) Self-Attention: Illustration

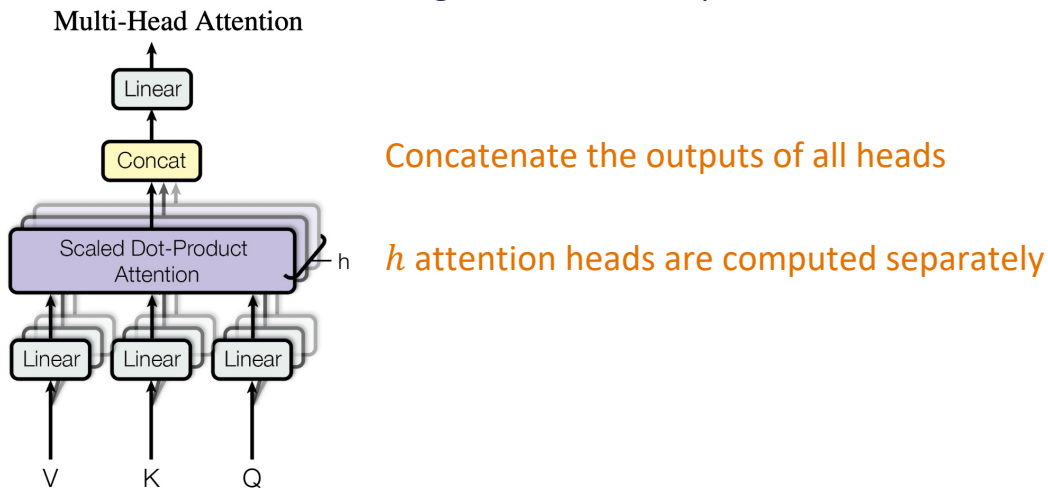
- Example: an input sequence with three words $[x_1, x_2, x_3]$
- Suppose we want to compute the self-attention for x_3





(Recap) 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






(Recap) 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: $\mathbf{Q} = \mathbf{X} \mathbf{W}^Q \quad \mathbf{K} = \mathbf{X} \mathbf{W}^K \quad \mathbf{V} = \mathbf{X} \mathbf{W}^V \in \mathbb{R}^{N \times d}$

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





(Recap) Parallel Computation of Attention

Attention computation can also be written in matrix form

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

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

Attention is **quadratic** in the length of the input: need to compute dot products between each pair of tokens in the input

Attention matrix

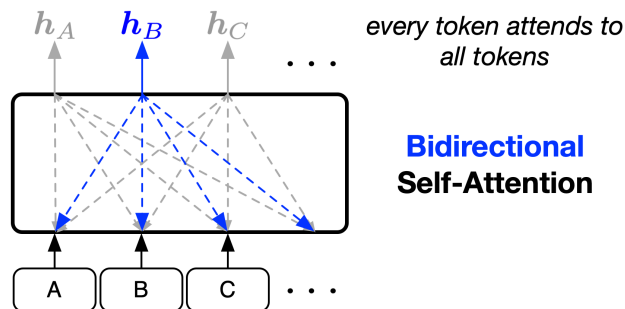
q1•k1	q1•k2	q1•k3	q1•k4
q2•k1	q2•k2	q2•k3	q2•k4
q3•k1	q3•k2	q3•k3	q3•k4
q4•k1	q4•k2	q4•k3	q4•k4

N



(Recap) Bidirectional Self-Attention

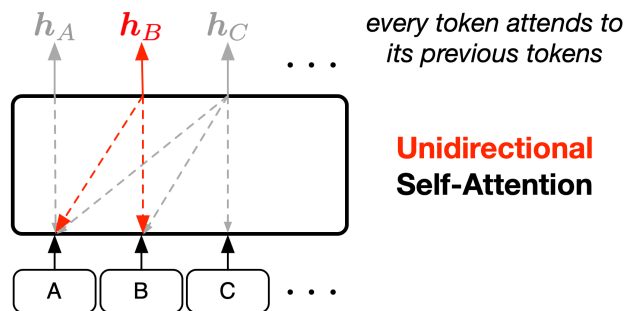
- Self-attention can capture different context dependencies
- **Bidirectional** self-attention:
 - Each position attends to all other positions in the input sequence
 - Transformers with bidirectional self-attention are called Transformer **encoders** (e.g., BERT)
 - Use case: natural language understanding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





(Recap) 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 $-\infty$

	$q_1 \cdot k_1$	$-\infty$	$-\infty$	$-\infty$
	$q_2 \cdot k_1$	$q_2 \cdot k_2$	$-\infty$	$-\infty$
	$q_3 \cdot k_1$	$q_3 \cdot k_2$	$q_3 \cdot k_3$	$-\infty$
	$q_4 \cdot k_1$	$q_4 \cdot k_2$	$q_4 \cdot k_3$	$q_4 \cdot k_4$

N

N

Agenda

- Position Encoding
- Tokenization
- Other Transformer Modules

Join at
slido.com
#8315 018





Position Encoding

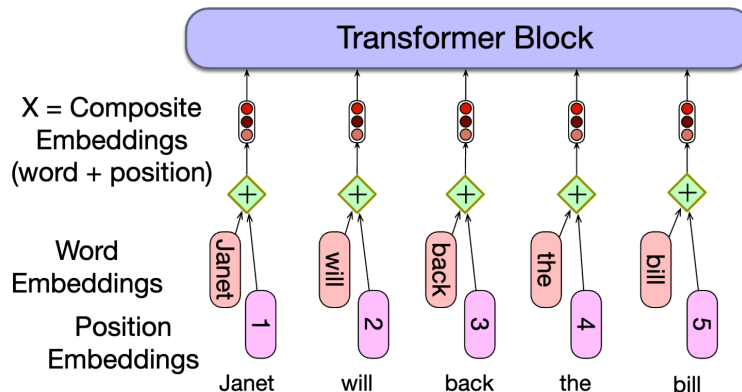
- Motivation: inject positional information to input vectors

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

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

When \mathbf{x} is word embedding, \mathbf{q} and \mathbf{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

Join at
slido.com
#8315 018





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?

Join at
slido.com
#8315 018



- 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

Join at
slido.com
#8315 018



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 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, 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, er, er



vocabulary

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

BPE: Counting & Merging

Join at
 slido.com
 #8315 018



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

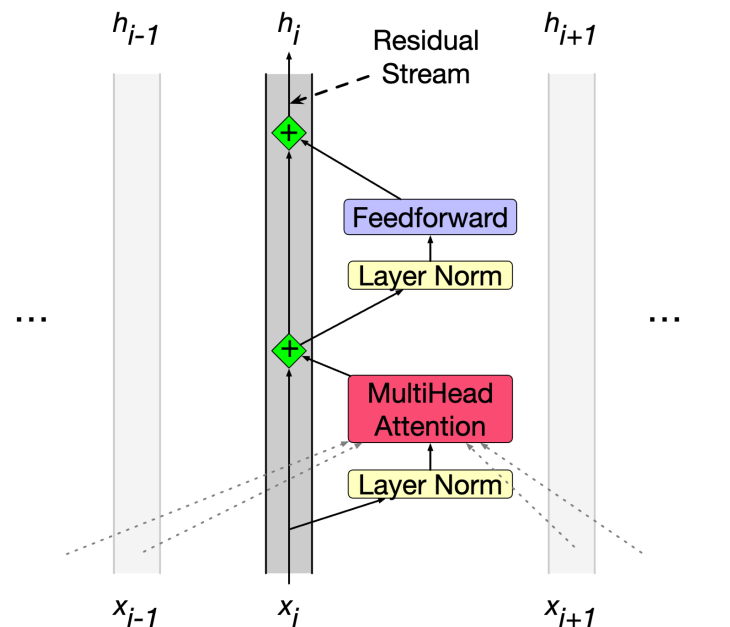
Join at
slido.com
#8315 018





Transformer Block

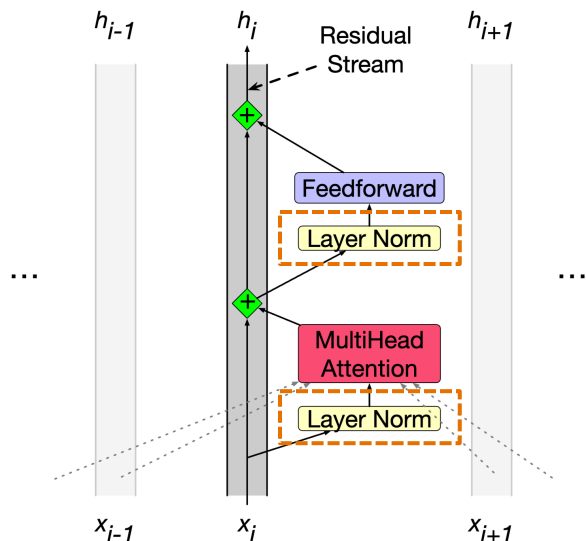
- Modules in Transformer layers:
 - Multi-head attention
 - Layer normalization (LayerNorm)
 - Feedforward network (FFN)
 - Residual connection





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





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

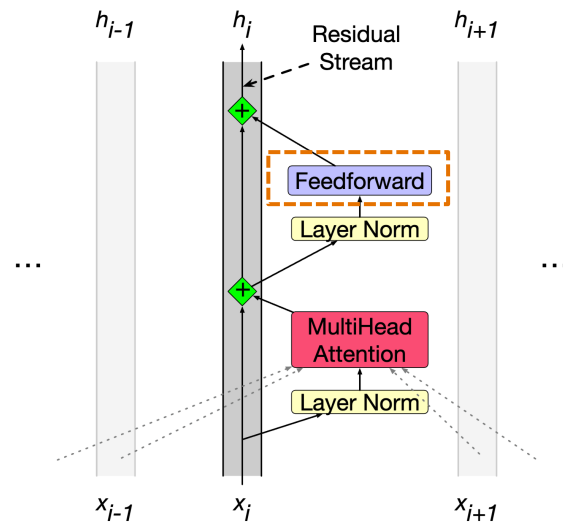


Feedforward Network (FFN)

- 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



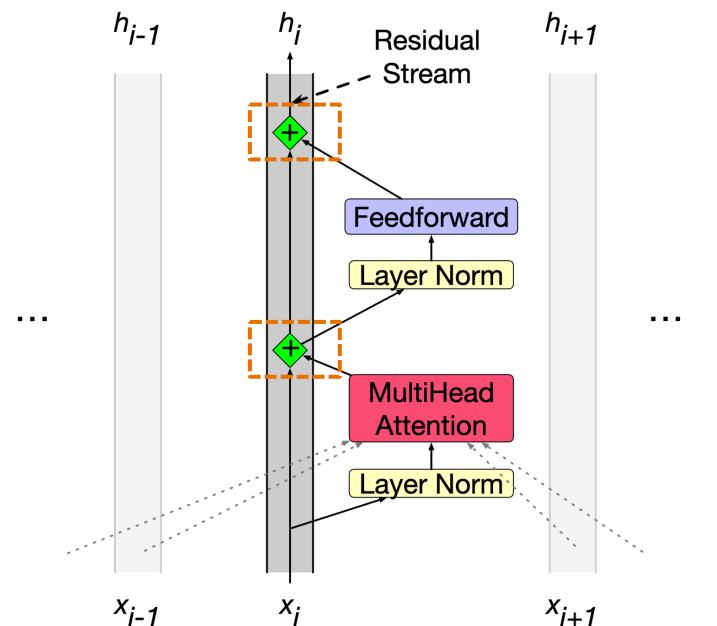


Residual Connections

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

$$y = x + f(x)$$

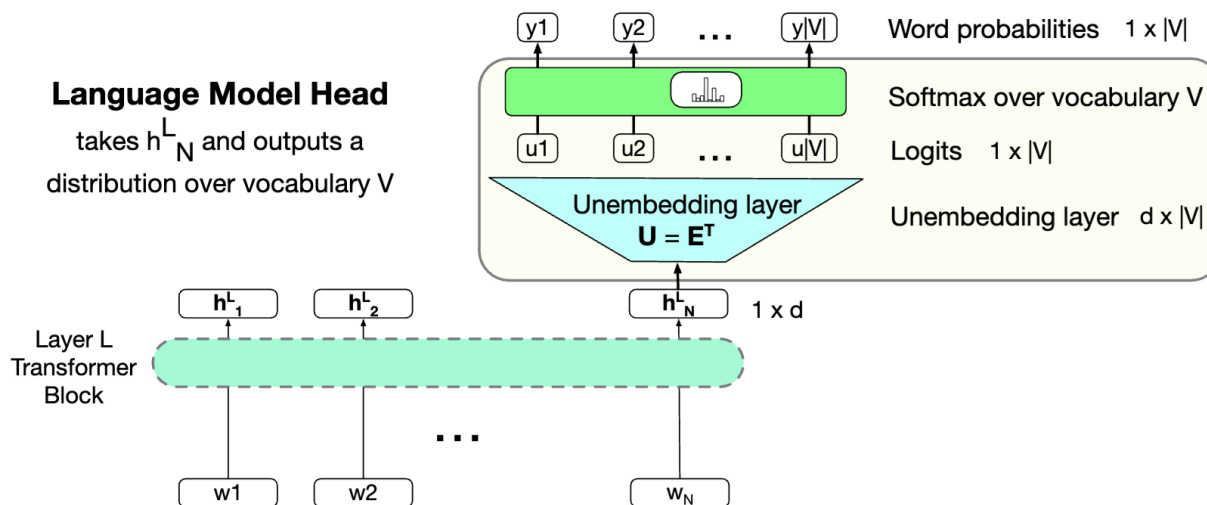
- Benefits
 - Address the vanishing gradient problem
 - Facilitate information flow across the network
 - Help scale up model





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

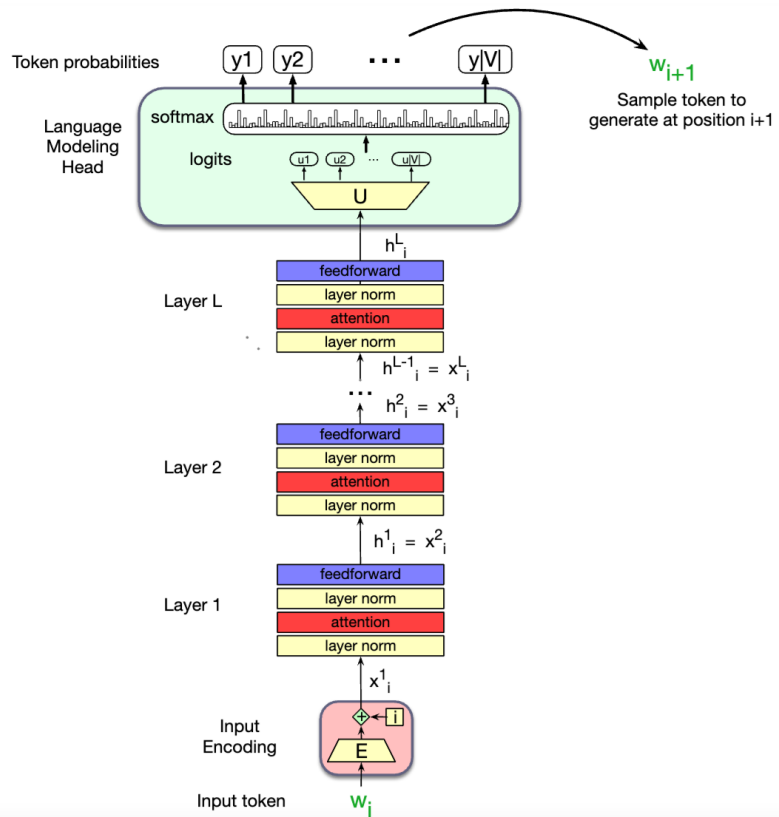


Figure source: <https://web.stanford.edu/~jurafsky/slp3/9.pdf>



Thank You!

Yu Meng

University of Virginia

yumeng5@virginia.edu