

Transformers, Part I

Robin Jia
USC CSCI 467, Fall 2026
October 21, 2026

Announcements

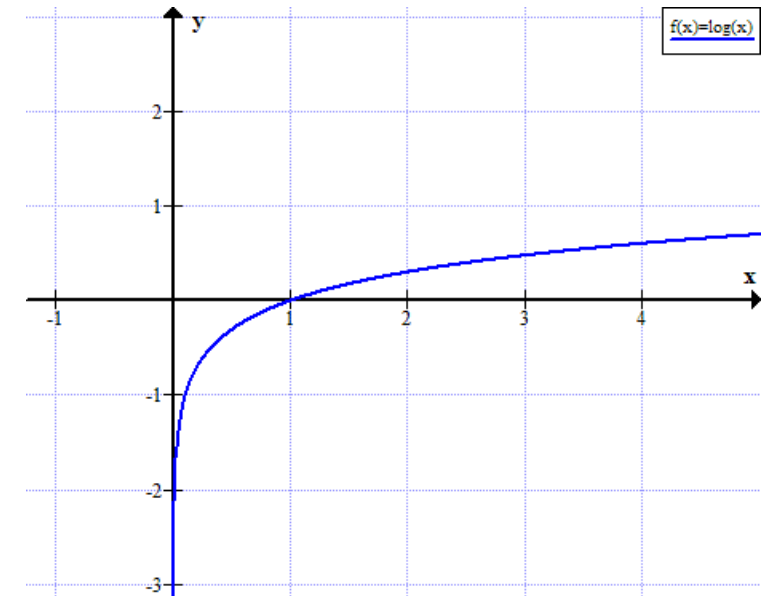
- Midterm grades released
 - Regrade requests open through this next Tuesday, October 27
- Project midterm report due Monday, November 2
 - Aiming to finish grading project proposals by this Friday, each team is paired with one TA/CP
 - Main goal: Obtain needed data & have a full pipeline that processes data, trains a model, and gets some results
 - Compare this model with some baseline (either an even simpler model or a non-learning method)
 - Results may or may not be “good”—just a starting point for final model
 - Analyze errors and identify possible sources of improvement
 - Full description on course website (click on “Final Project Information”)
- HW3 releasing soon, due November 9

Common Exam Mistakes: 1d

(d) Now, consider what happens if we set $\lambda = 0$ instead. Consider the same message $x = (\text{students}, \text{help})$.

- i. (4 points) What will the model do now when asked to make a prediction on this input?
- ii. (4 points) Would computing the scores in log space change your answer to the question above? Explain your answer.

- Part (i): You get $P(x, y=R) = 0$ and $P(x, y=U) = 0$
- Part (ii): Taking logs doesn't help! In fact **$\log(0)$ is undefined**
 - Many people wrote $\log(0) = 0$ or 1
 - Log space helps when multiplying small probabilities, not when probabilities are equal to 0



Common Exam Mistakes: 2b/c

(b) (4 points) Derive the expression for $\nabla_w L(w)$, the gradient of $L(w)$ with respect to w .

Solution: The gradient is:

$$\frac{1}{n} \sum_{i=1}^n \left(2 \cdot (w^\top x^{(i)} - y^{(i)}) \cdot x^{(i)} \right) + 2\lambda w$$

- Many people had expressions like $2\lambda ||w||$ or $2\lambda \sum_{j=1}^d w_j$
- You should **immediately** know this is wrong because the gradient is a vector and these expressions are scalars!
 - Important to make sure your equations “type-check”

Common Exam Mistakes: 3a

Ryan is training a neural network, but he is being secretive about his exact methods. Using your detective skills, you have uncovered the following clues:

- Ryan's model includes two linear layers, one RNN layer, and one tanh layer (not necessarily in that order).
- Besides these layers, the model only has one other layer, which is of an unknown type.

...

- The model is doing a classification task where each input is a text document (as a list of words).

(a) (5 points) What must the “unknown” layer in Ryan's model be? Explain your reasoning.

If input is a list of words, the model **needs** a word vector layer!

Common Exam Mistakes: 5a

(b) (3 points) Spider-Man and Ned are building a web portal to classify emergency complaints from across Brooklyn. For their initial prototype, they need an automated system to tag each complaint with any applicable labels: "Superhuman Threat," "Property Damage," and/or "Evacuation Needed." (Note that a single complaint can receive multiple tags simultaneously i.e., the categories are not mutually exclusive). Which of these models is the best fit?

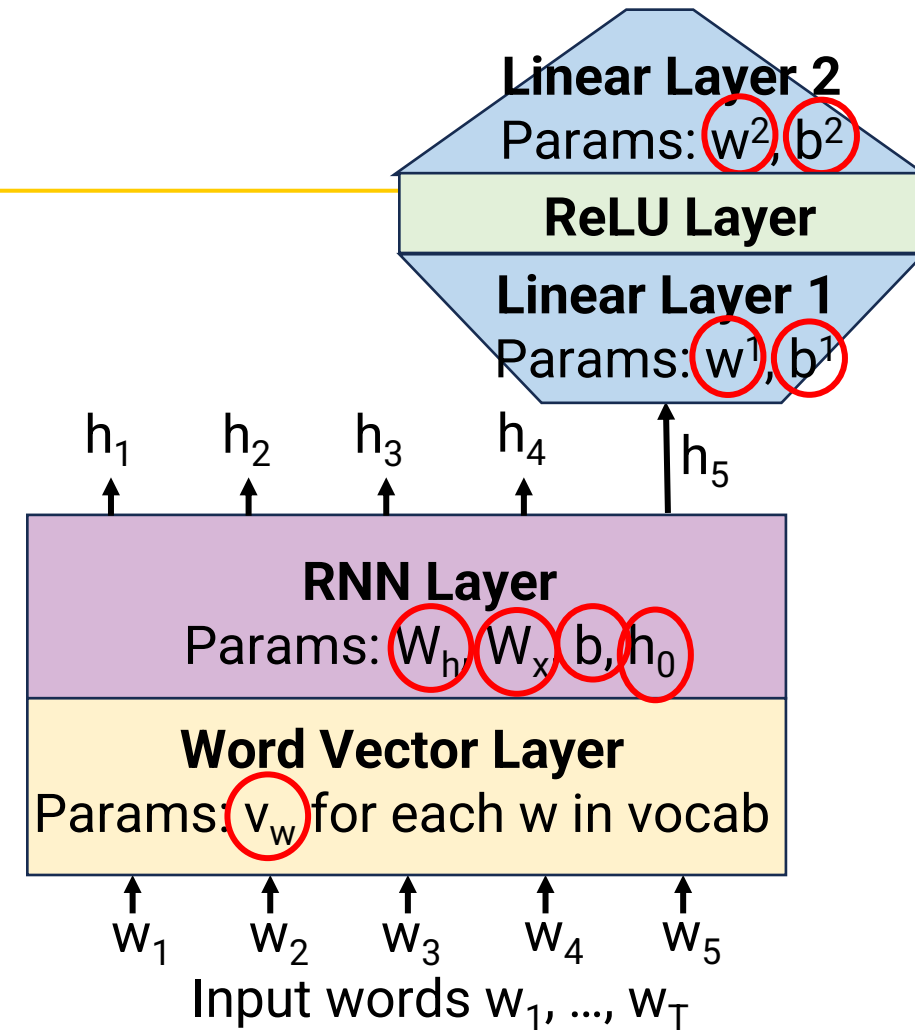
- A. Linear Regression
- ☒ B. Logistic Regression
- C. Softmax Regression
- D. word2vec

- Be careful: Softmax Regression does not apply because **labels are not mutually exclusive**, while softmax just chooses one best class
- Should think of each label as a separate yes/no binary classification problem

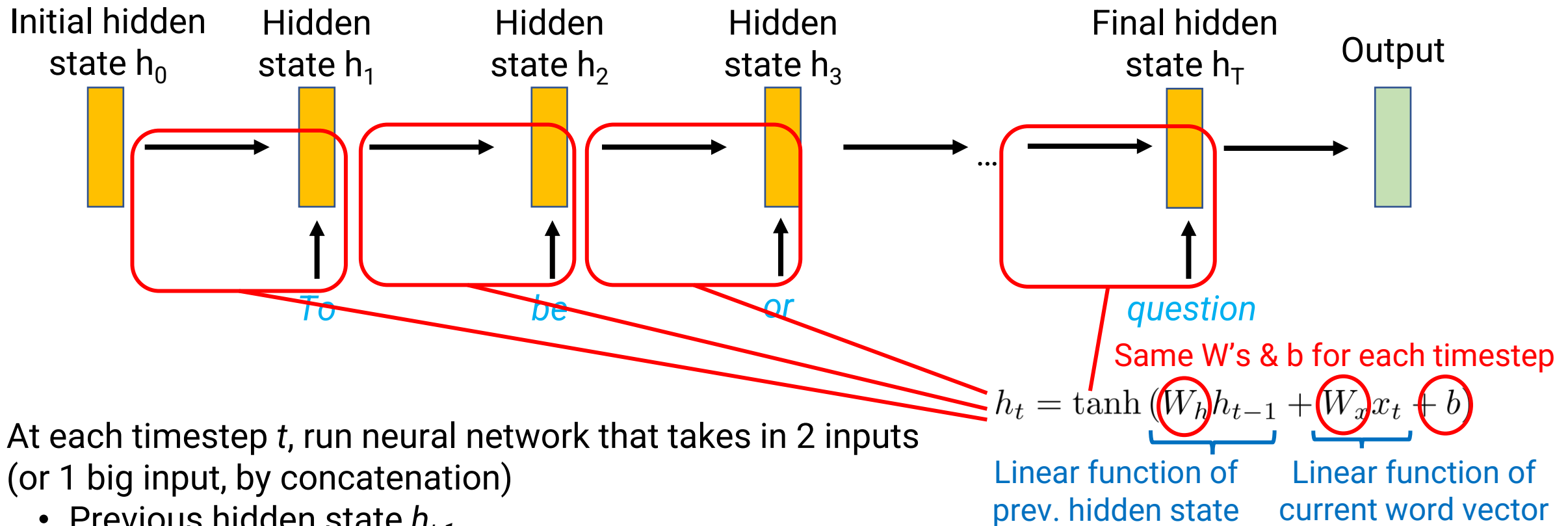
Review: Deep Learning

- Task: Specifies the inputs & outputs
 - Sentiment classification: Input = sentence, Output = positive/negative
 - Object recognition: Input = picture, Output = type of object
- Model: We combine building blocks that can transform the input to the output
 - **With parameters**: Linear layer, Convolutional layer, RNN layer, Word vector layer
 - **No parameters**: sigmoid/tanh/ReLU, max pooling, addition,
- Training: Minimize loss of our model's outputs compared to the true outputs by updating **parameters** of all layers (that have them)
 - Do this by gradient descent
 - Backpropagation computes gradient w.r.t. every parameter

Neural Network Model



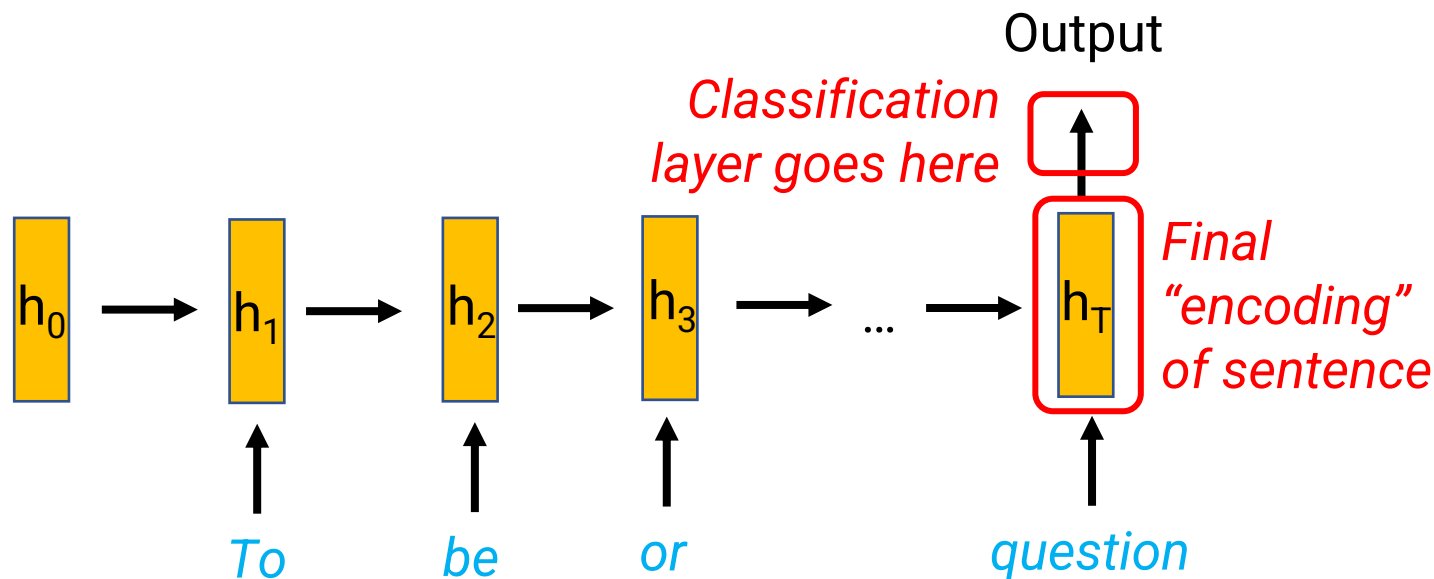
Review: RNNs



- At each timestep t , run neural network that takes in 2 inputs (or 1 big input, by concatenation)
 - Previous hidden state h_{t-1}
 - Vector for current word x_t
- Learn linear function of both inputs, add bias, apply non-linearity
- Parameters: Recurrence params (W_h , W_x , b), initial hidden state h_0 , word vectors

Review: Encoder vs. Decoder

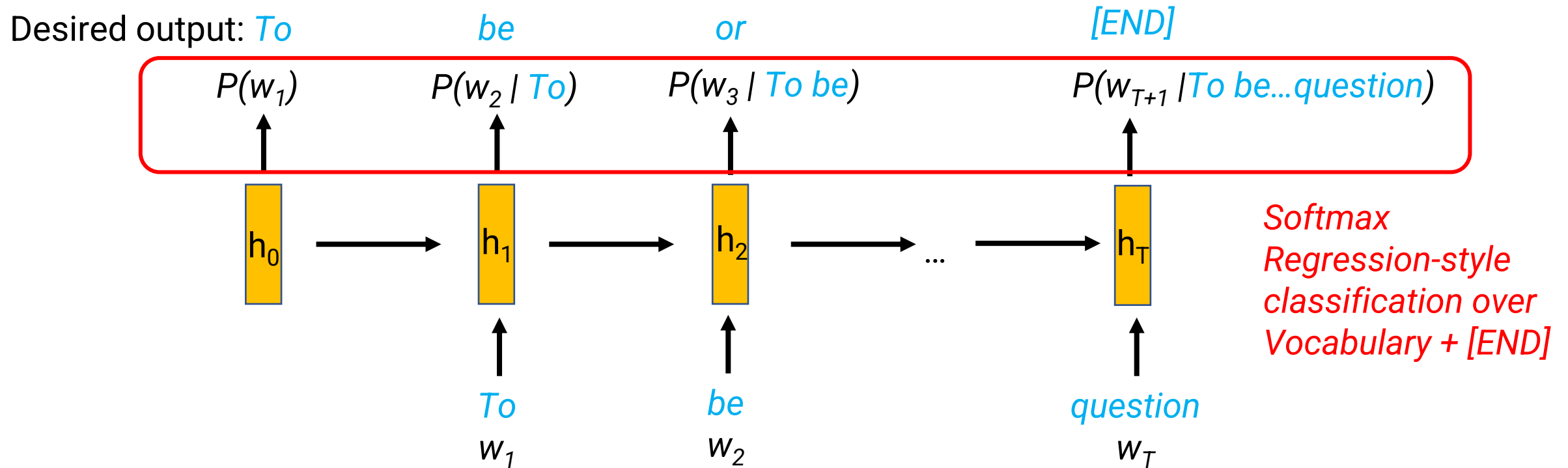
Encoder model: Converts sentence to vector “encoding”



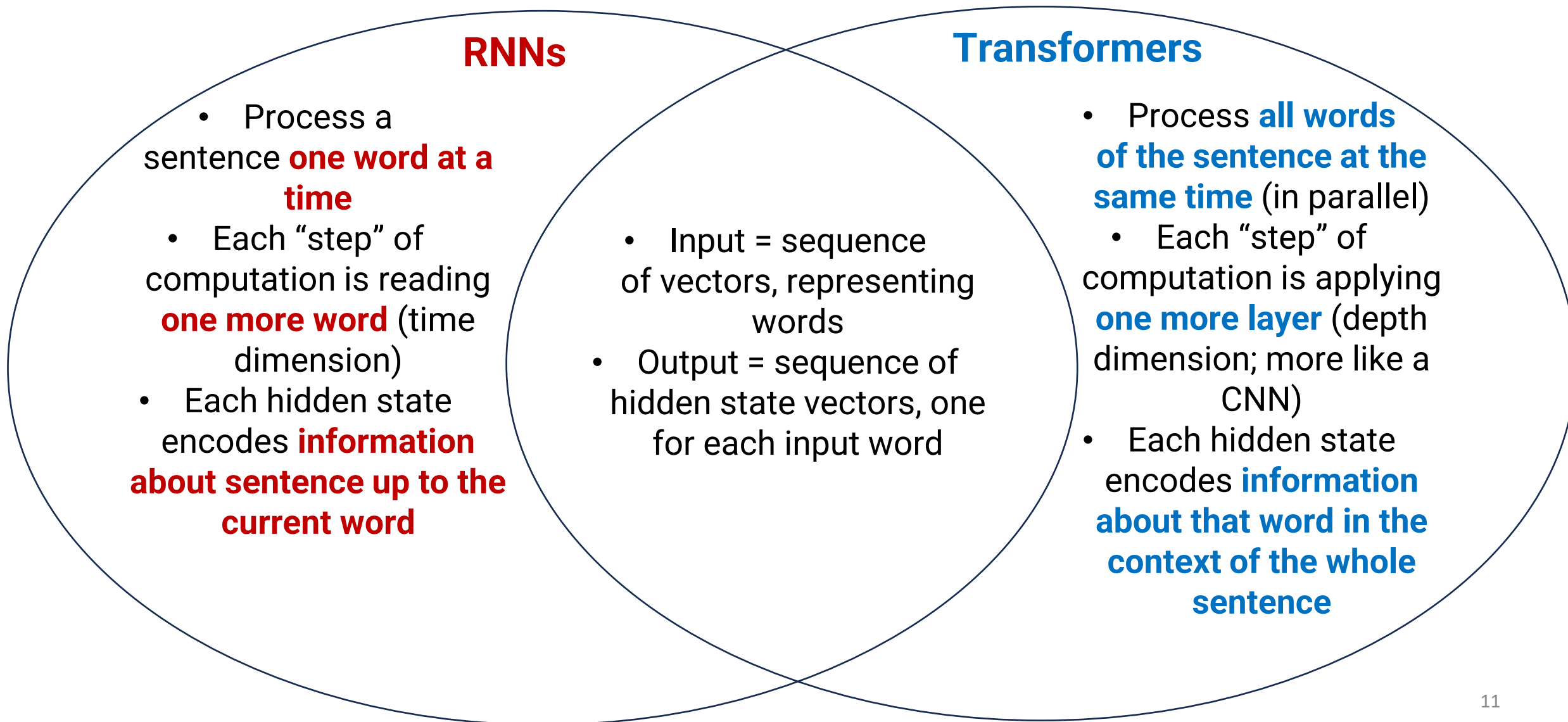
- First run an RNN over text
- Each hidden state is an “encoding” of the entire sequence up to the current timestep
- Use this as features, train a classifier on top

Review: Encoder vs. Decoder

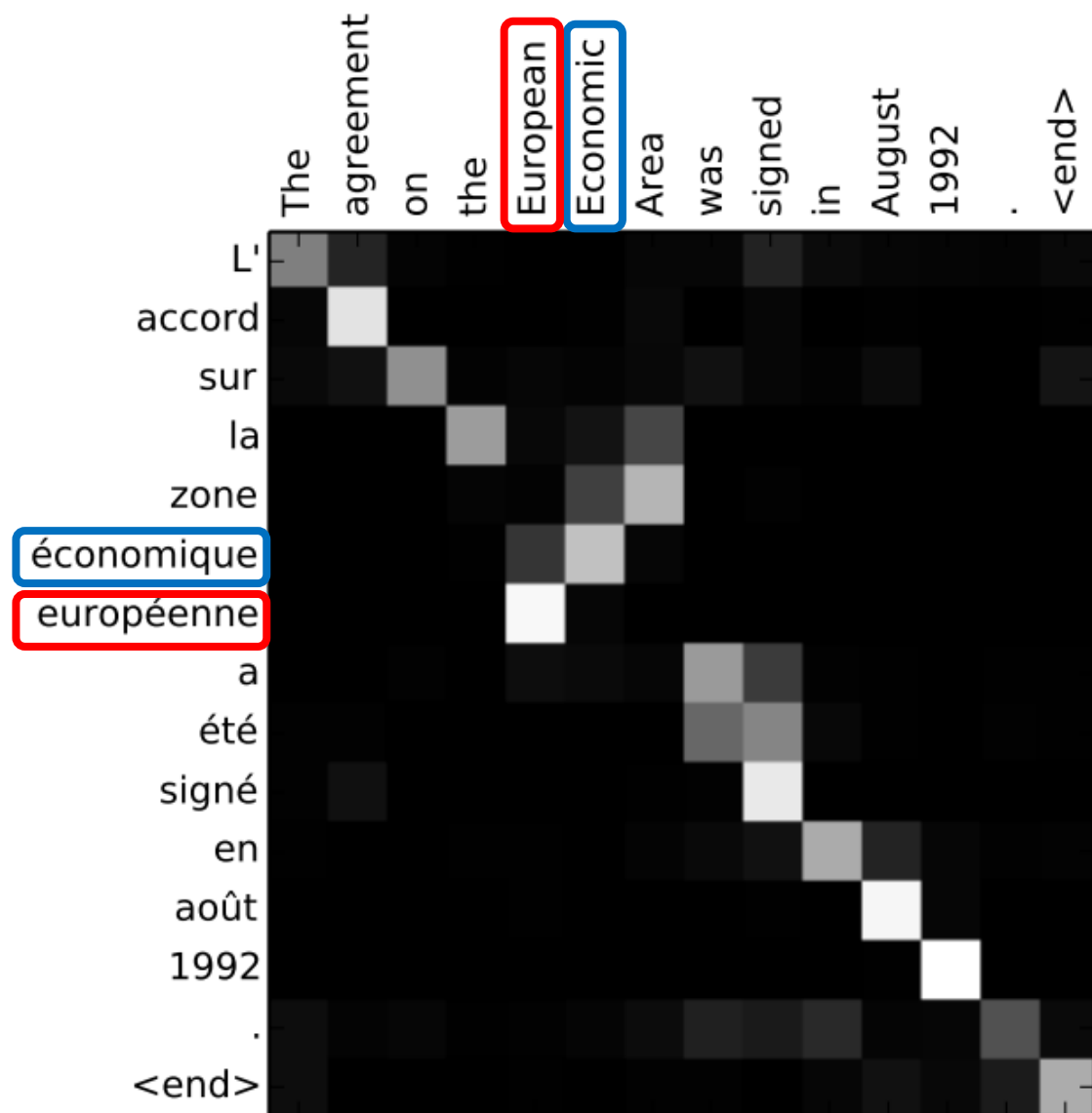
Decoder model: Generates words one at a time



RNNs vs. Transformers (Encoders)

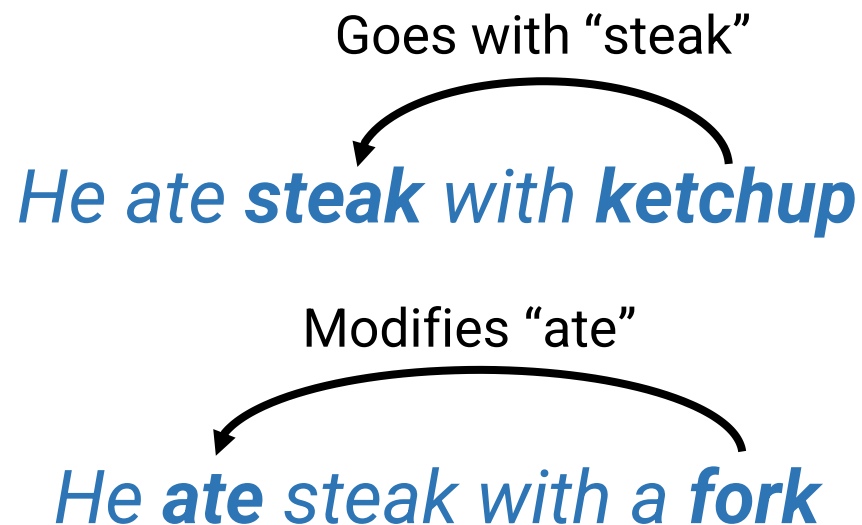


Review: Challenges of modeling sequences



- Modeling relationships between words
 - Translation alignment

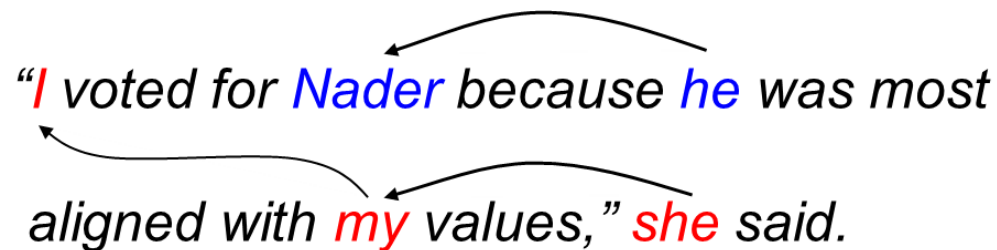
Review: Challenges of modeling sequences



- Modeling relationships between words
 - Translation alignment
 - Syntactic dependencies

Review: Challenges of modeling sequences

"I voted for Nader because he was most aligned with my values," she said.

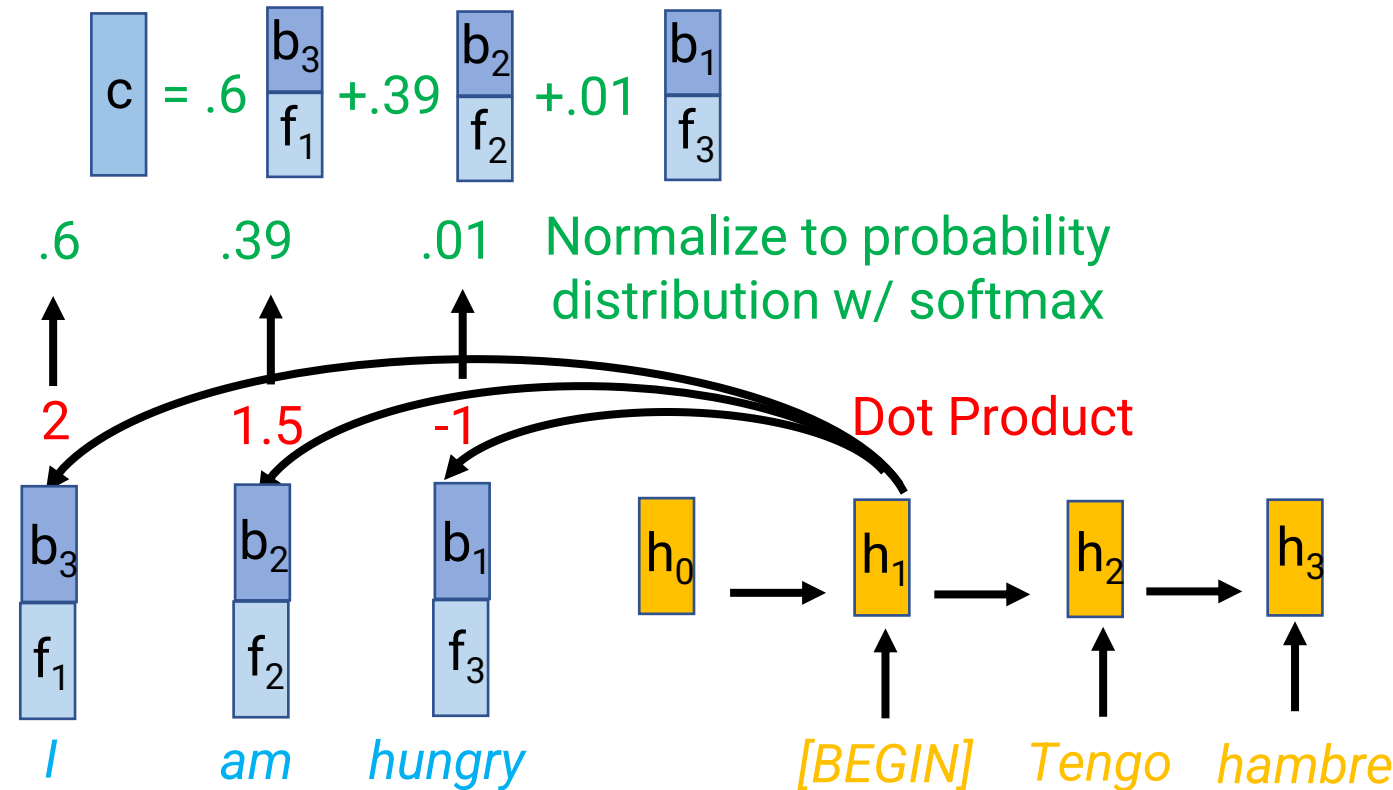


The diagram illustrates coreference relationships in the sentence "I voted for Nader because he was most aligned with my values," she said. Curved arrows connect the pronouns to their referents: an arrow from "I" to "she", an arrow from "he" to "Nader", and an arrow from "my" to "values".

- Modeling relationships between words
 - Translation alignment
 - Syntactic dependencies
 - Coreference relationships

Review: Attention

- Compute **similarity** between **decoder** hidden state and each **encoder** hidden state
 - E.g., **dot product**, if same size
- Normalize similarities to **probability distribution with softmax**
- Output: “Context” vector c = weighted average of encoder states based on the probabilities
 - No new parameters (like ReLU/max pool)



Review: Attention as Retrieval

Google search results for "training a machine translation model". The search bar is highlighted with a green box. Below the search bar, there are tabs for Images, Videos, Perspectives, Python, Example, Online, Github, Shopping, and News. The results show "About 174,000,000 results (0.18 seconds)". The first result is from Pangeanic, titled "How to train your machine translation engine", with a snippet about machine translation engines. The second result is from Machine Learning Mastery, titled "How to Develop a Neural Machine Translation System from ...", with a snippet about machine translation as a challenging task. The third result is from GitHub, titled "Tutorial: Neural Machine Translation - seq2seq", with a snippet about sequence-to-sequence models and training scripts.

training a machine translation model

Images Videos Perspectives Python Example Online Github Shopping News

About 174,000,000 results (0.18 seconds)

Pangeanic
https://blog.pangeanic.com › train-machine-translation-e...
How to train your machine translation engine
Oct 20, 2021 — A **machine translation** engine is software capable of translating texts from a source language to a target language. Applying artificial ...
How To Train Your Machine... · 1. Incorporation Of The Base... · Tips For Improving The...

Machine Learning Mastery
https://machinelearningmastery.com › Blog
How to Develop a Neural Machine Translation System from ...
Oct 6, 2020 — **Machine translation** is a challenging task that traditionally involves large statistical **models** developed using highly sophisticated linguistic ...

GitHub
https://google.github.io › nmt
Tutorial: Neural Machine Translation - seq2seq
For more details on the theory of Sequence-to-Sequence and **Machine Translation models**, we recommend the following resources: ... The **training** script will save ...
Neural Machine Translation... · Alternative: Generate Toy Data · Training

- Consider a search engine:
 - **Queries**: What you are looking for
 - E.g., What you type into Google search
 - **Keys**: Summary of what information is there
 - E.g., Text from each webpage
 - **Values**: What to give the user
 - E.g., The URL of each webpage

Review: Attention

(8) Attention Layer

- Inputs (all vectors of length d):
 - Query vector q
 - Key vectors $k_1, \dots, k_T$
 - Value vectors $v_1, \dots, v_T$
- Output (also vector of length d)
 - Dot product q with each key vector k_t to get score s_t :
$$s_t = q^\top k_t$$
 - Softmax to get probability distribution $p_1, \dots, p_T$:

How well does the query match each key?

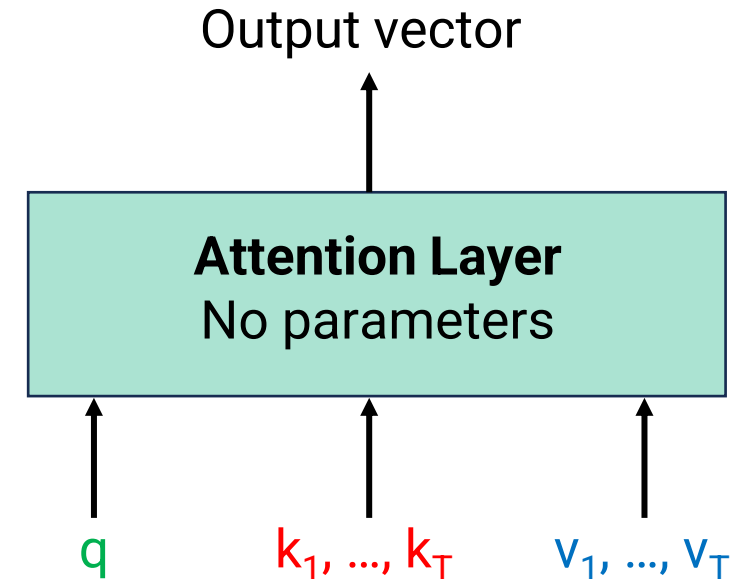


$$p_t = \frac{e^{s_t}}{\sum_{j=1}^T e^{s_j}}$$

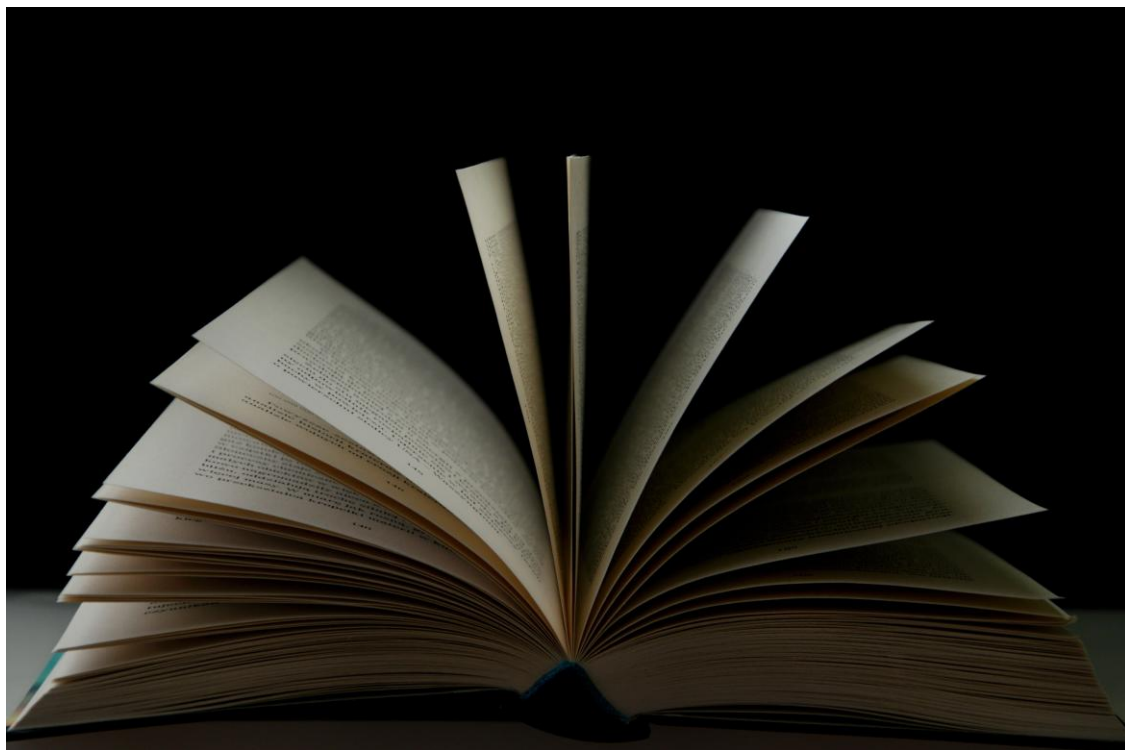
- Return weighted average of value vectors:

$$\sum_{t=1}^T p_t v_t$$

Dominated by the values corresponding to the “best-matching” keys

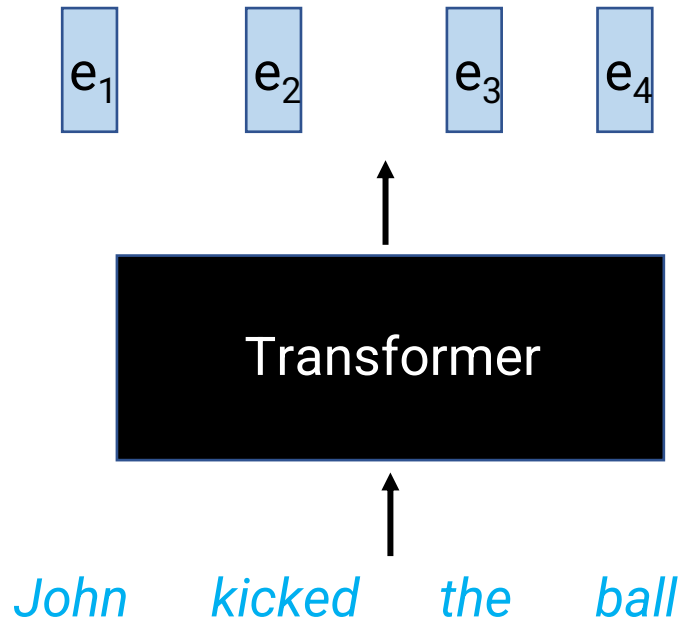


Today: Can we use Attention for Everything?



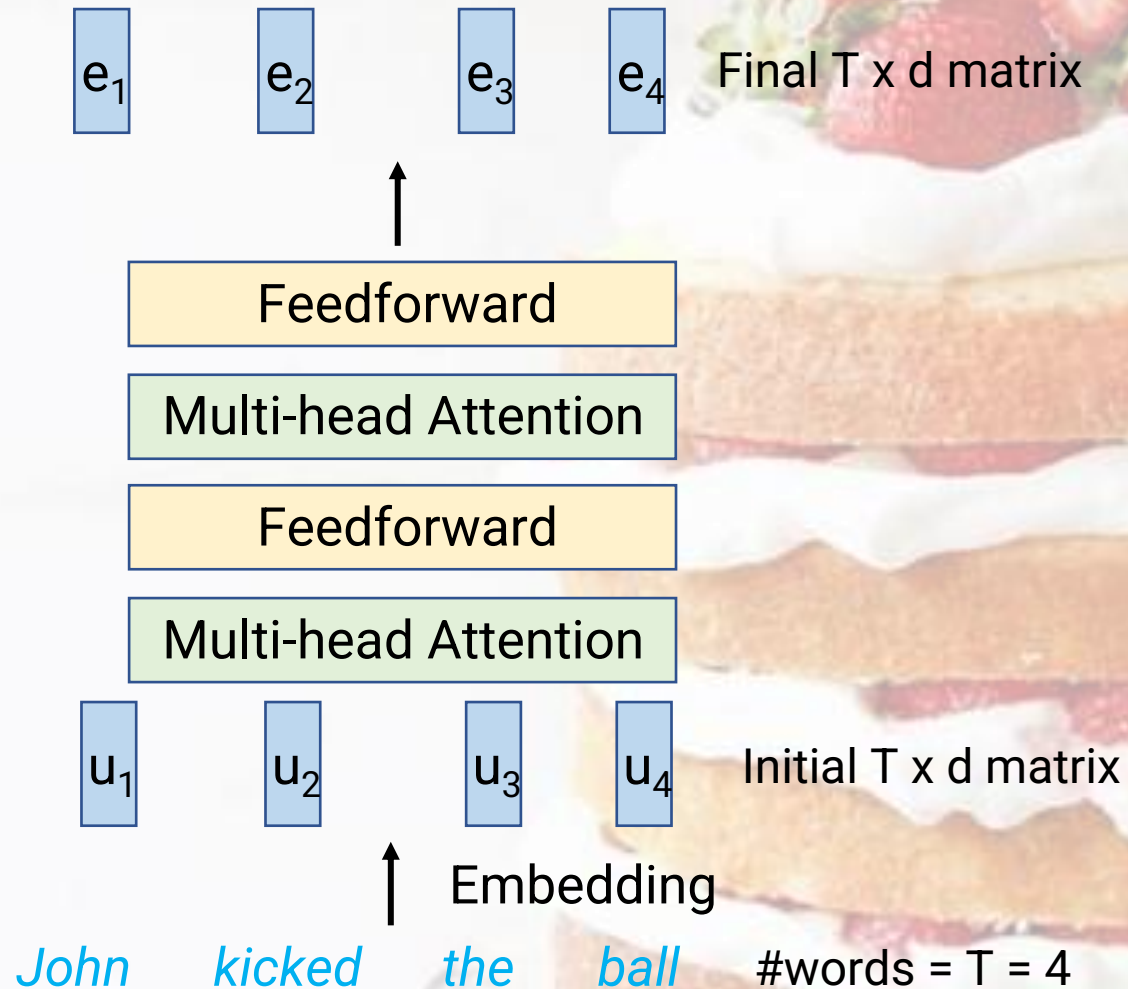
- Modeling relationships between words
 - Translation alignment
 - Syntactic dependencies
 - Coreference relationships
- Long range dependencies
 - E.g., consistency of characters in a novel
- Attention captures relationships & doesn't care about "distance," unlike RNNs
- **Let's replace RNN's with an architecture based solely on MLP's + attention**

Today: The Transformer Architecture



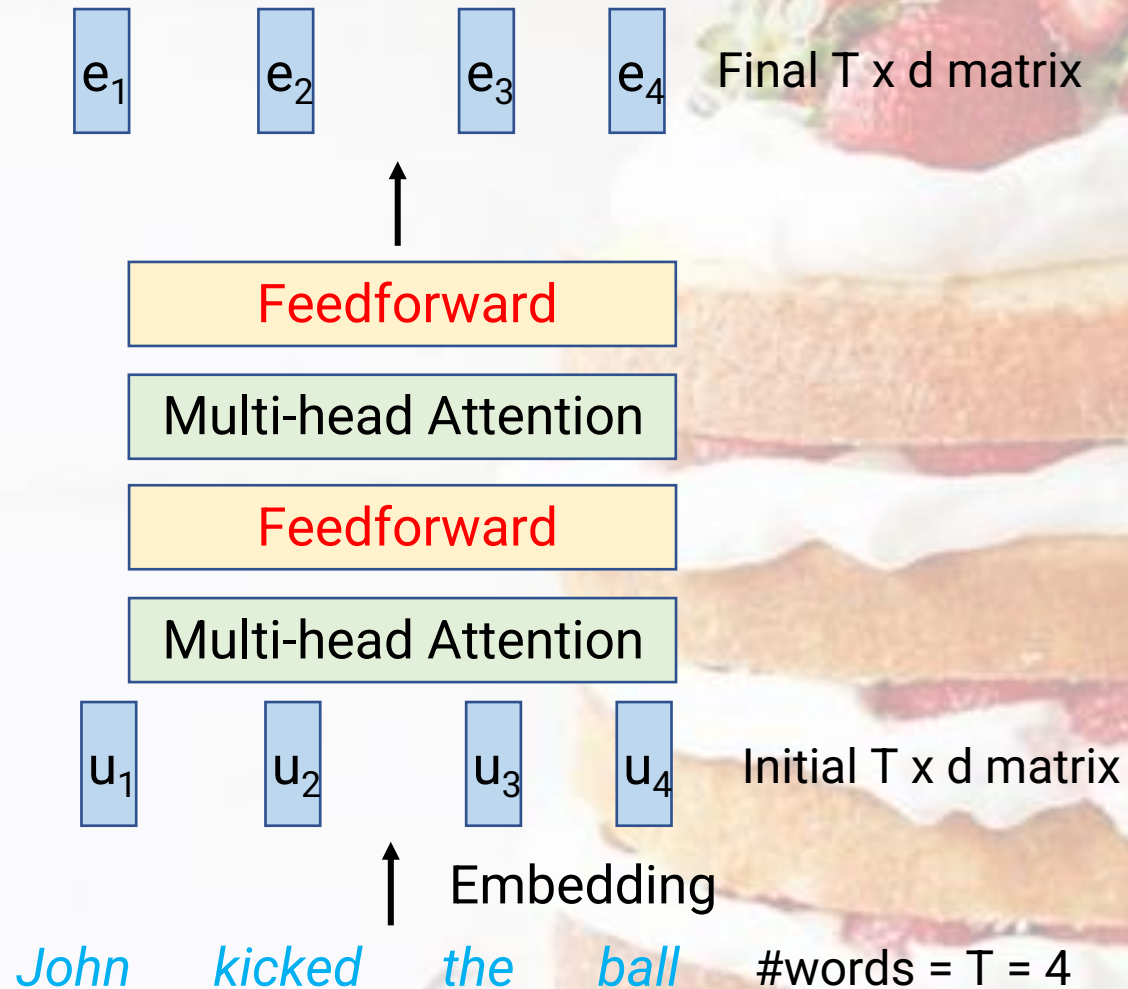
- Input: Sequence of words
- Output: Sequence of hidden state vectors, one per word
- Same “type signature” as RNN
- Motivation
 - Process all words at the same time, don’t do explicit sequential processing
 - Let **attention** figure out which words are relevant to each other
 - Whereas RNN assumes sequence order is what matters
 - “Attention is all you need”

Transformer overview



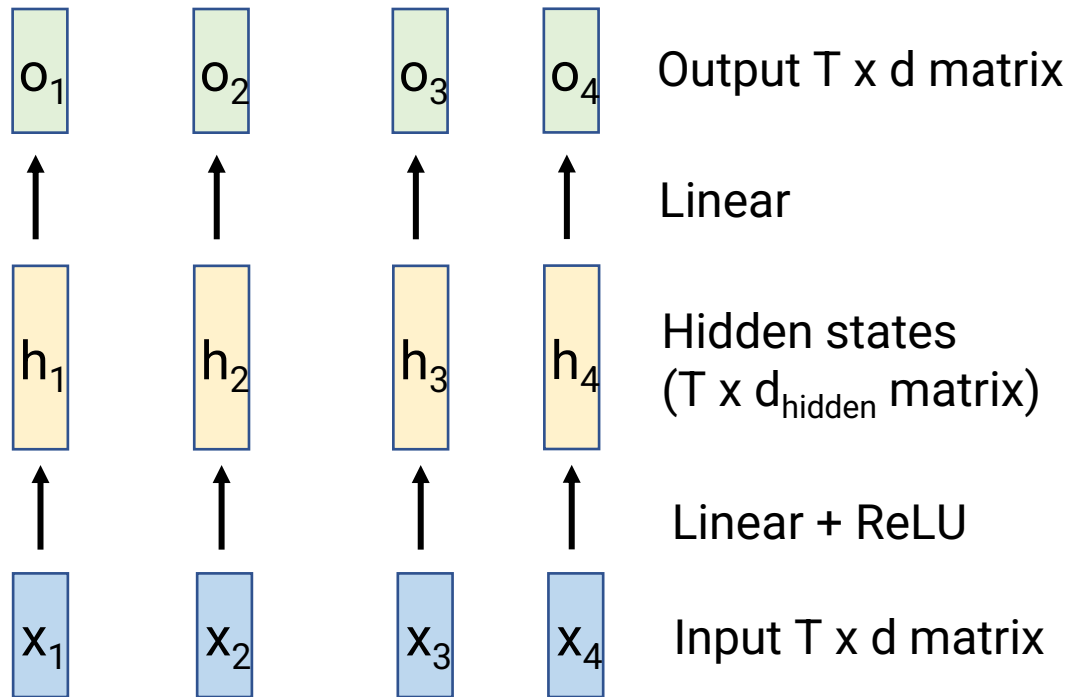
- One transformer consists of
 - Initial embeddings for each word of size d
 - Let $T = \text{\#words}$, so initially we have a $T \times d$ matrix
 - Alternating layers of
 - “Multi-headed” attention layer
 - Feedforward layer
 - Both take in $T \times d$ matrix and output a new $T \times d$ matrix
 - Plus some bells and whistles...

Transformer overview



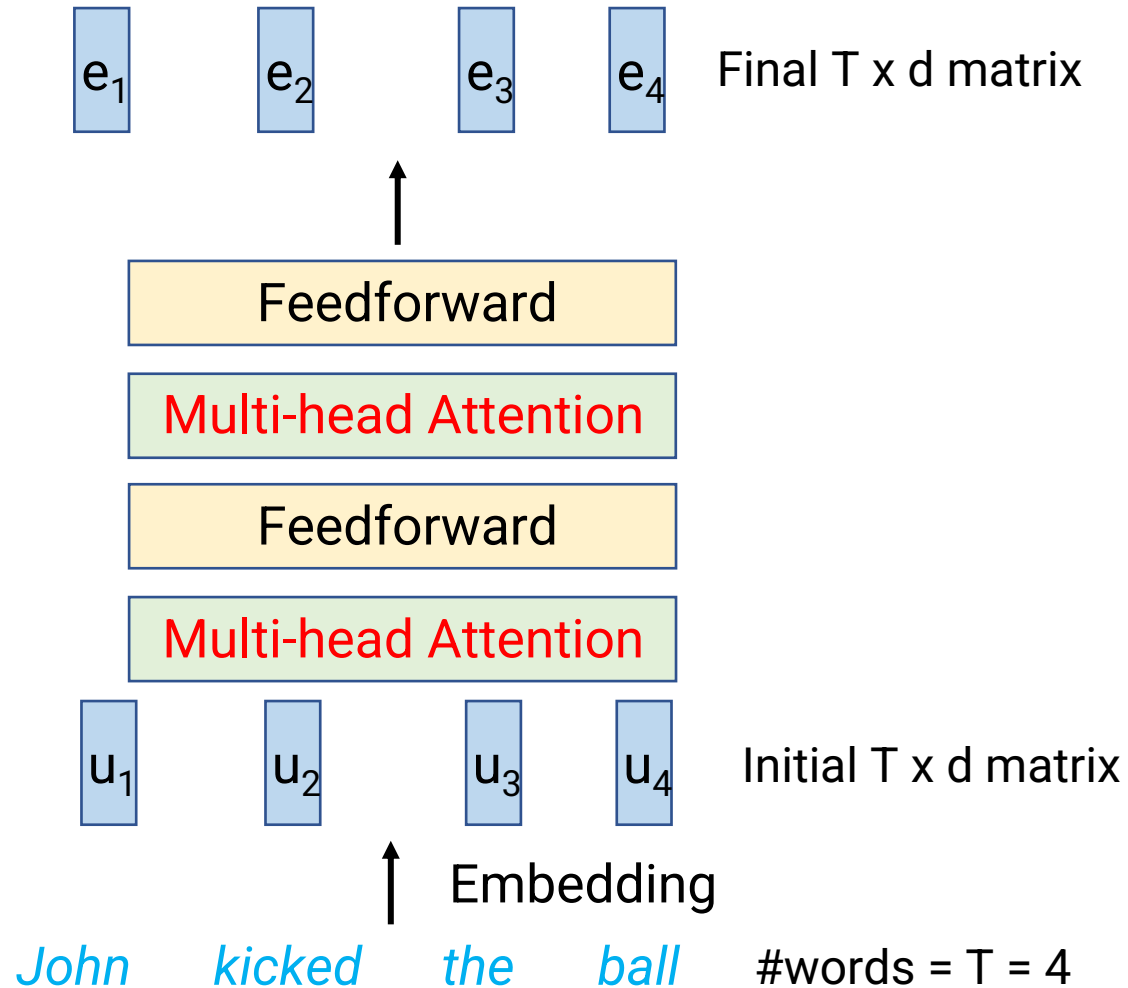
- One transformer consists of
 - Initial embeddings for each word of size d
 - Let $T = \text{\#words}$, so initially we have a $T \times d$ matrix
 - Alternating layers of
 - “Multi-headed” attention layer
 - **Feedforward layer**
 - Both take in $T \times d$ matrix and output a new $T \times d$ matrix
 - Plus some bells and whistles...

Feedforward layer



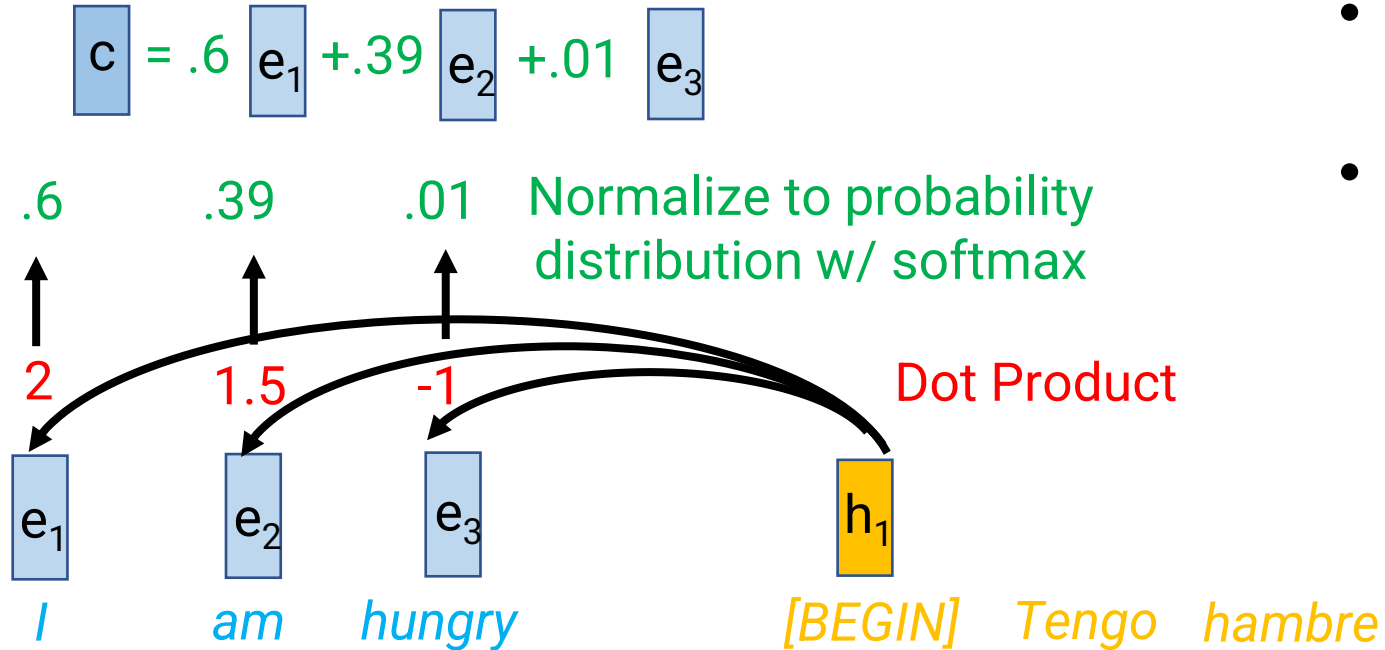
- Input: $T \times d$ matrix
- Output: Another $T \times d$ matrix
- Apply the same MLP separately to each d -dimensional vector
 - Linear layer from d to d_{hidden}
 - ReLU (or other nonlinearity)
 - Linear layer from d_{hidden} to d
- Note: No information moves between tokens here

Transformer overview



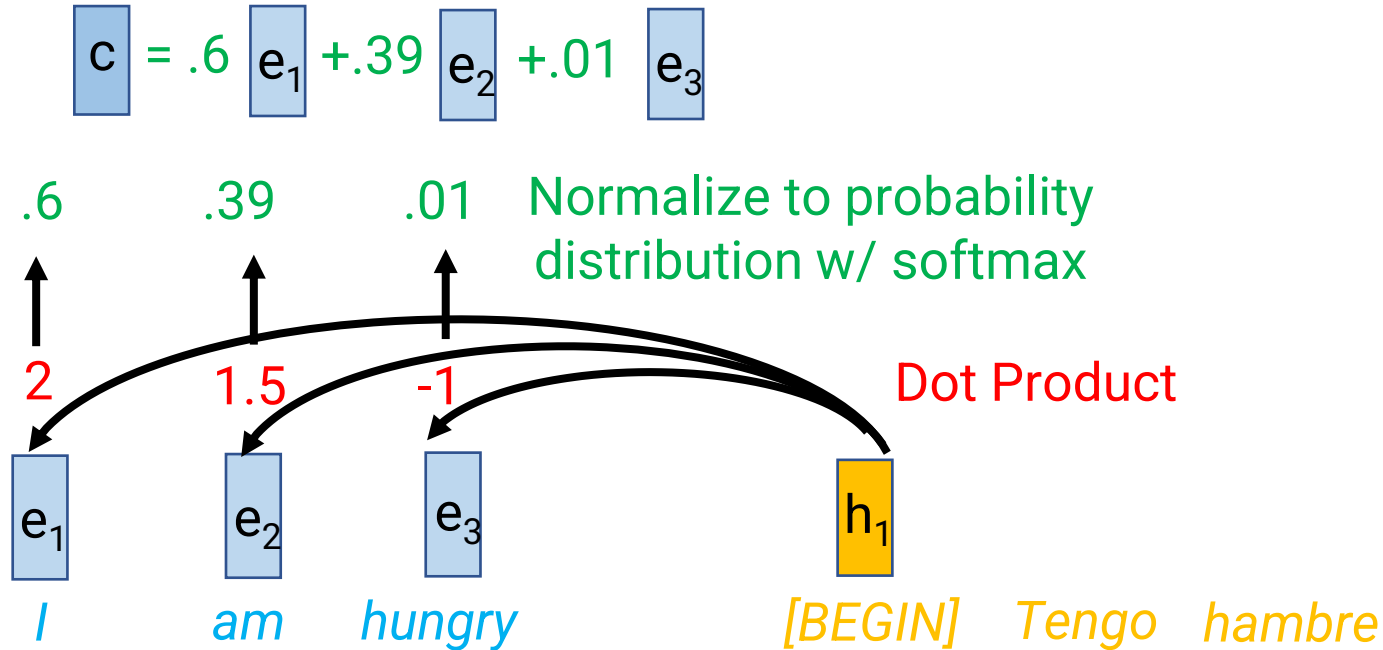
- One transformer consists of
 - Initial embeddings for each word of size d
 - Let $T = \text{\#words}$, so initially we have a $T \times d$ matrix
 - Alternating layers of
 - **"Multi-headed" attention layer**
 - **Feedforward layer**
 - Both take in $T \times d$ matrix and output a new $T \times d$ matrix
 - Plus some bells and whistles...

Modifying Attention



- What is a multi-headed attention layer???
- Similar to attention we've seen, but need to make 3 changes...
 - Self-attention (no separate encoder & decoder)
 - Separate queries, keys, and values
 - Multi-headed

Change #1: Self-Attention



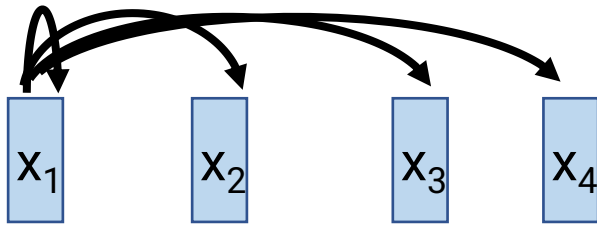
- Previously: Decoder state looks for relevant encoder states
- Self-attention: Each **encoder** state now looks for relevant (other) encoder states
- Why? Build better representation for word in context by capturing relationships to other words

Change #1: Self-attention

$$o_1 = .19x_1 + .5x_2 + .3x_3 + .01x_4$$

.19 .5 .3 .01 Probabilities for x_1

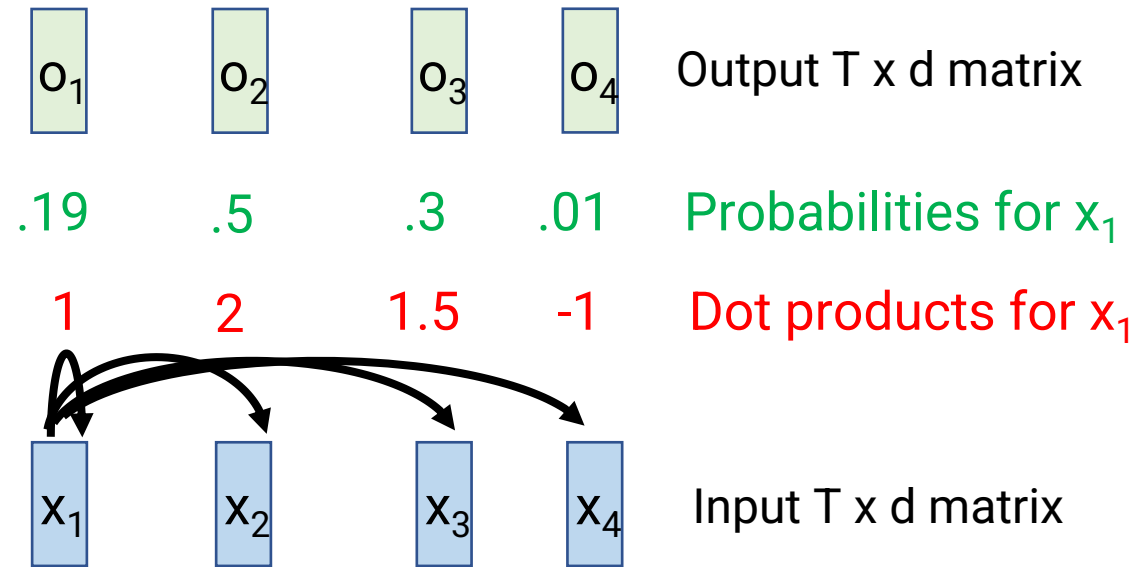
1 2 1.5 -1 Dot products for x_1



Input T x d matrix

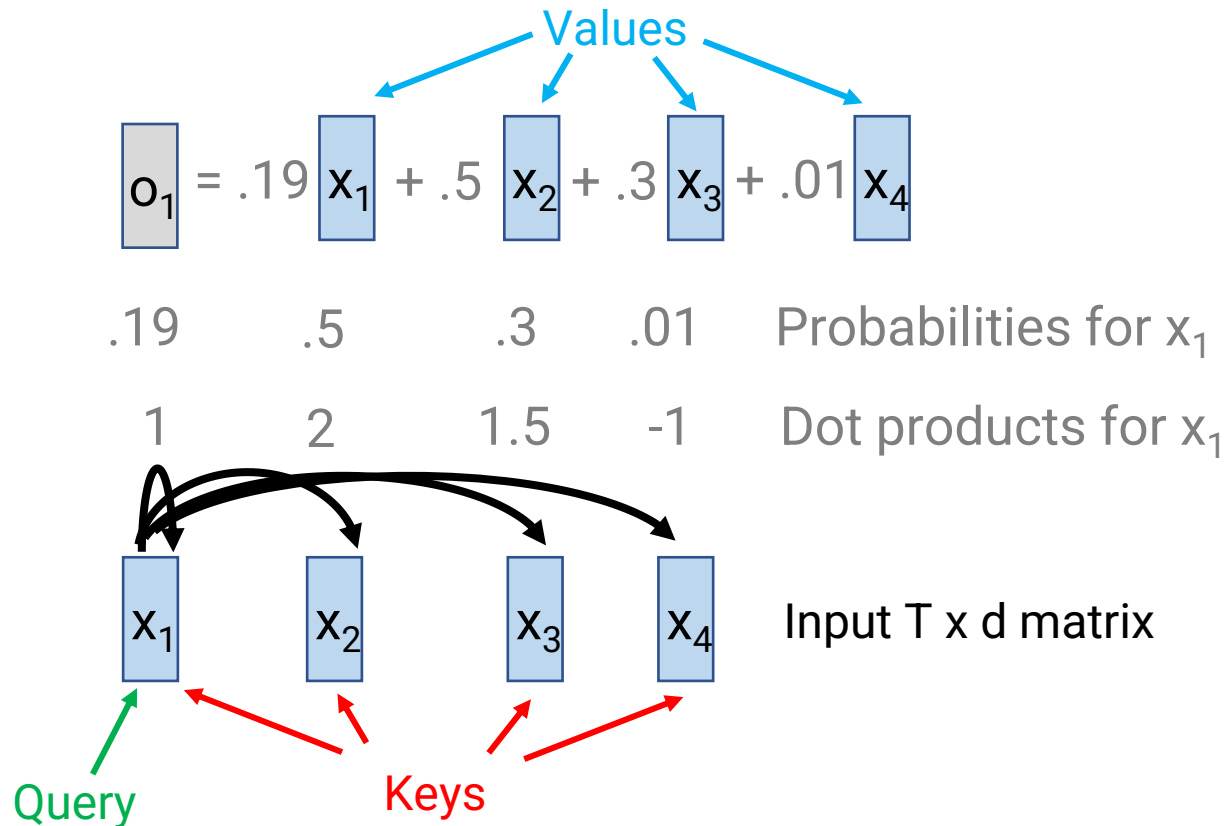
- Take x_1 and dot product it with all T inputs (including itself)
- Apply softmax to convert to probability distribution
- Compute output o_1 as weighted sum of inputs

Change #1: Self-attention



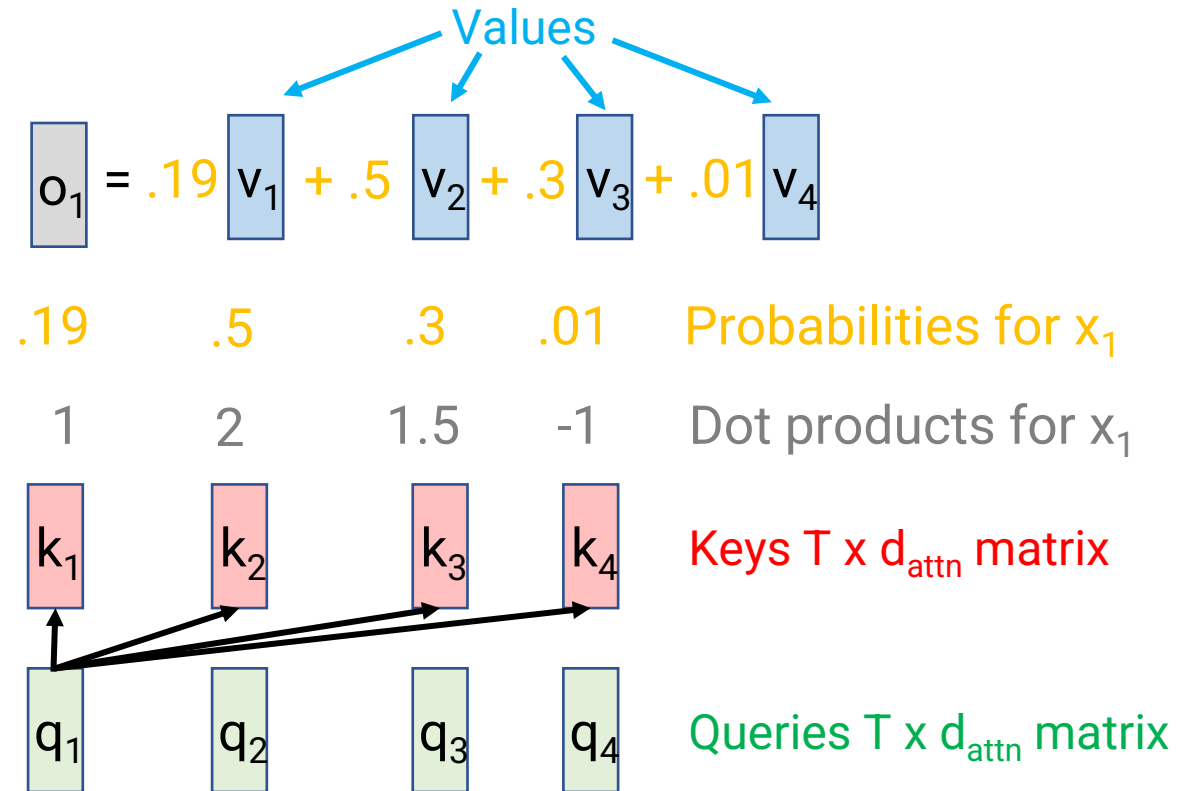
- Take x_1 and dot product it with all T inputs (including itself)
- Apply softmax to convert to probability distribution
- Compute output o_1 as weighted sum of inputs
- Repeat for $t=2, 3, \dots, T$
- Replacement for recurrence
 - RNN only allows information to flow linearly along sequence
 - Now, information can flow from any index to any other index, as determined by attention

Change #2: Separate queries, keys, and values



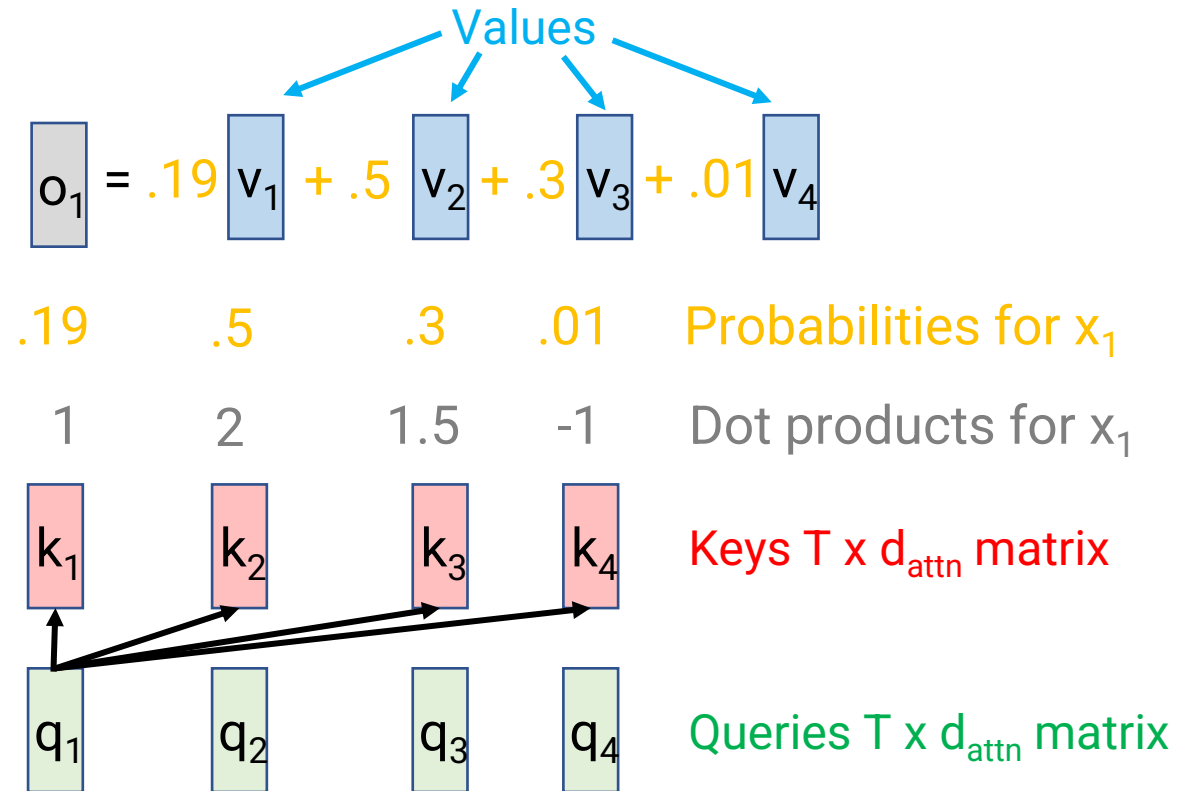
- Recall: Attention uses vectors in three different ways
 - As “**query**” for current index
 - As “**keys**” to match with query
 - As “**values**” when computing output
- Idea: Use separate vectors for each usage
 - What each index “**looks for**” different from what it “**matches with**”
 - What you **store in output** different from what you “**look for**”/“**match with**”

Change #2: Separate queries, keys, and values



- Apply 3 separate linear layers to each of $x_1, \dots, x_T$ to get
 - Queries $[q_1, \dots, q_T]$, each $q_t = W^Q * x_t$
 - Keys $[k_1, \dots, k_T]$, each $k_t = W^K * x_t$
 - Values $[v_1, \dots, v_T]$, each $v_t = W^V * x_t$
 - Note: This adds parameters W^Q, W^K, W^V
 - Each linear layer maps from dimension d to dimension d_{attn}
- Dot product q_1 with $[k_1, \dots, k_T]$
- Apply softmax to get probability distribution
- Compute o_1 as weighted sum of $[v_1, \dots, v_T]$
- Repeat for $t = 2, \dots, T$

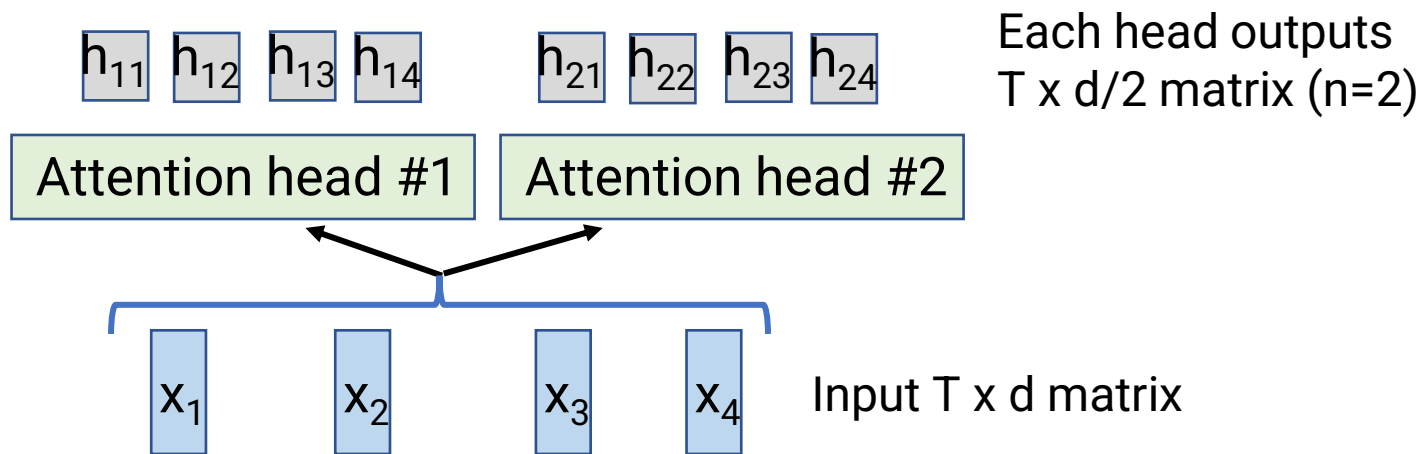
Matrix form



- Quadratic in T
- All you need is fast matrix multiplication
- All indices run in parallel

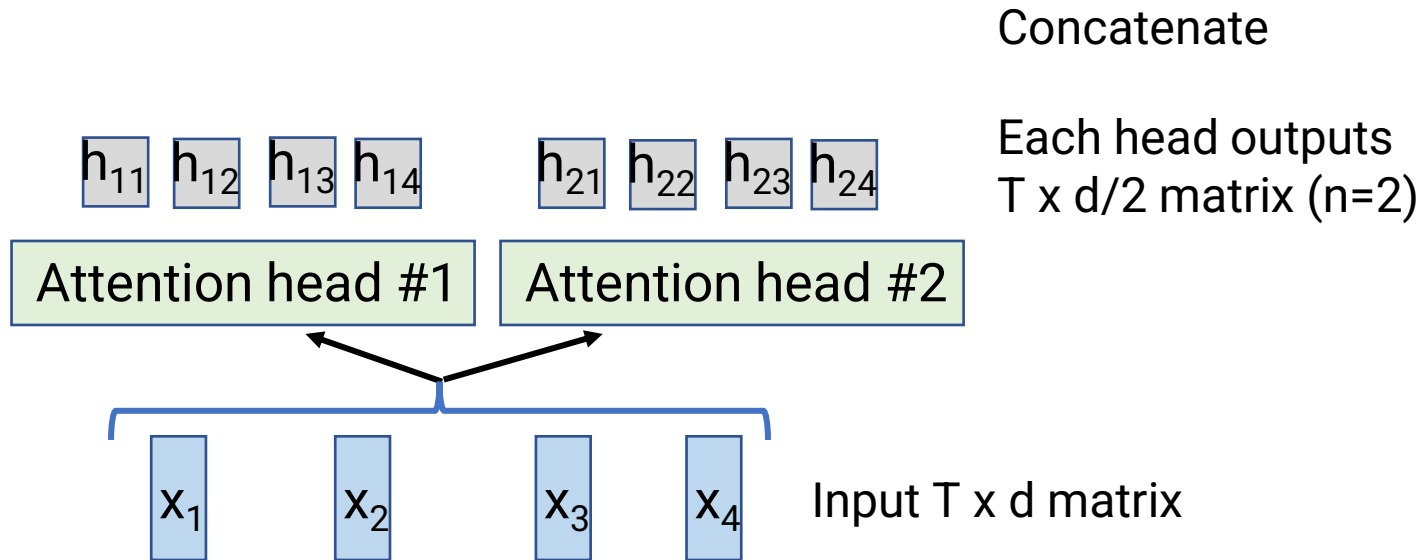
- Apply 3 separate linear layers to input matrix X ($T \times d_{\text{in}}$) to get
 - Query matrix $Q = (W^Q * X^T)^T$
 - Keys $K = (W^K * X^T)^T$
 - Values $V = (W^V * X^T)^T$
 - Note: This adds parameters W^Q, W^K, W^V
- Compute $Q \times K^T$ ($T \times T$ matrix)
 - Each entry is dot product of one query vector with one key vector
- Normalize each row with softmax to get matrix of probabilities P
- Output = $P \times V$

Change #3: Making it Multi-headed



- Instead of doing attention once, have n different “heads”
 - Each has its own W^Q, W^K, W^V parameters that map to dimension $d_{\text{attn}} = d/n$
 - Concatenate at end to get output of size $T \times d$

Change #3: Making it Multi-headed



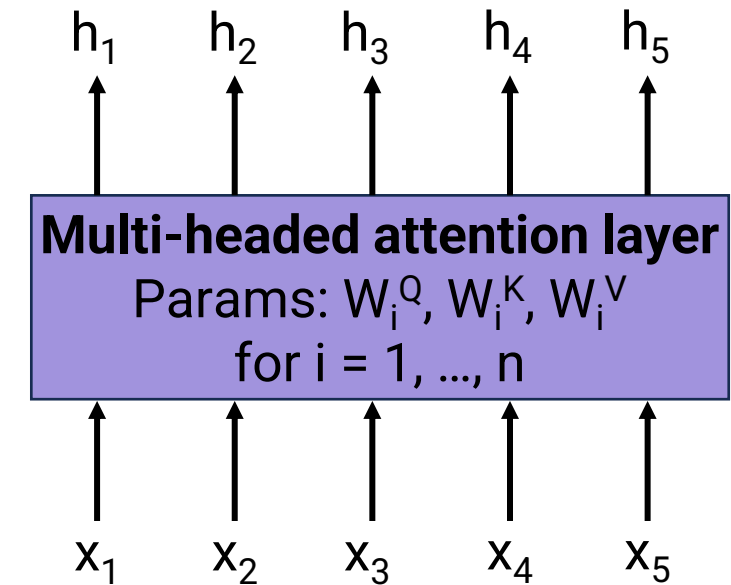
- Instead of doing attention once, have n different “heads”
 - Each has its own W^Q, W^K, W^V parameters that map to dimension $d_{\text{attn}} = d/n$
 - Concatenate at end to get output of size $T \times d$
- Why? Different heads can capture different relationships between words

The Multi-headed Attention building block

(9) Multi-headed Attention Layer

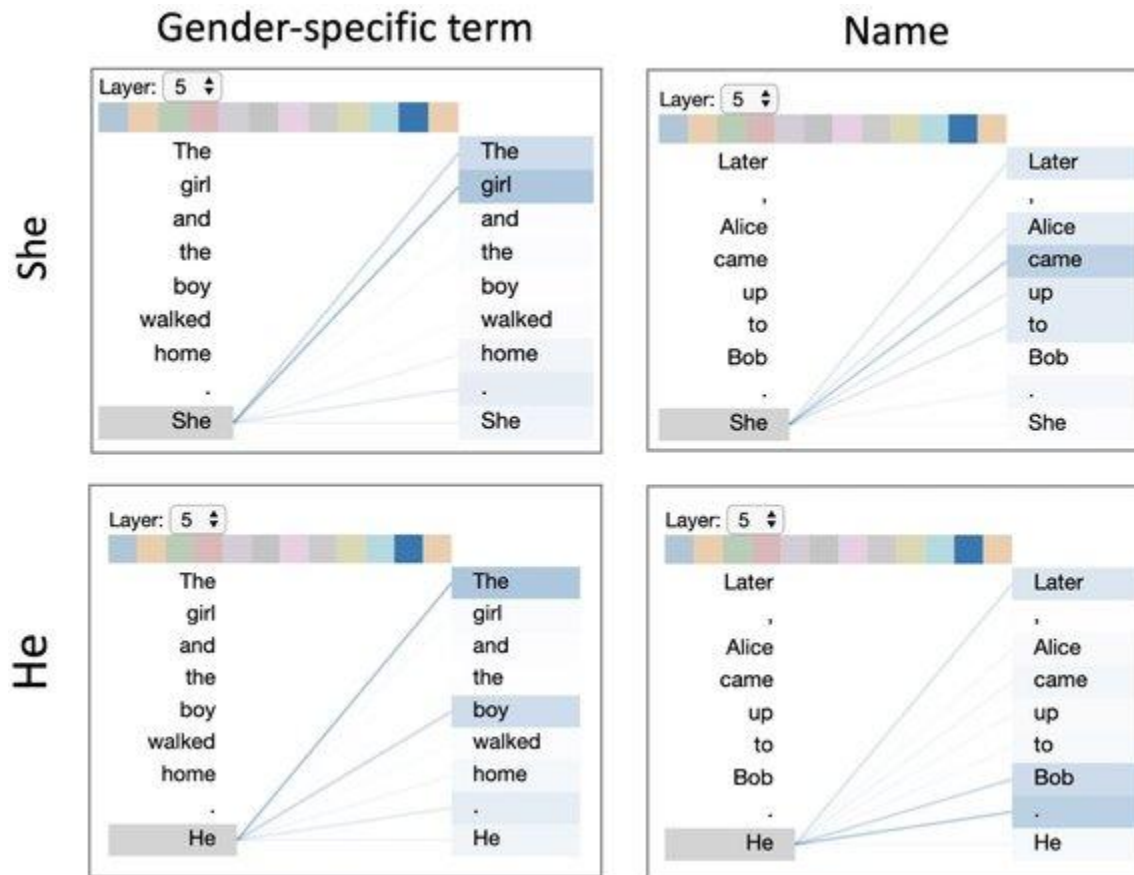
- Input: List of vectors $x_1, \dots, x_T$, each of size d
 - Equivalent to a $T \times d$ matrix
- Output: List of vectors $h_1, \dots, h_T$, each of size d
 - Equivalent to another $T \times d$ matrix
- Formula: For each head i :
 - Compute Q, K, V matrices using W_i^Q, W_i^K, W_i^V
 - Compute self attention output using Q, K, V to yield $T \times d_{\text{attn}}$ matrix
 - Finally, concatenate results for all heads
- Parameters:
 - For each head i , parameter matrices W_i^Q, W_i^K, W_i^V of size $d_{\text{attn}} \times d$
 - (# of heads n is hyperparameter, $d_{\text{attn}} = d/n$)
- In pytorch: `nn.MultiheadAttention()`

Output $h_1, \dots, h_T$, each shape d



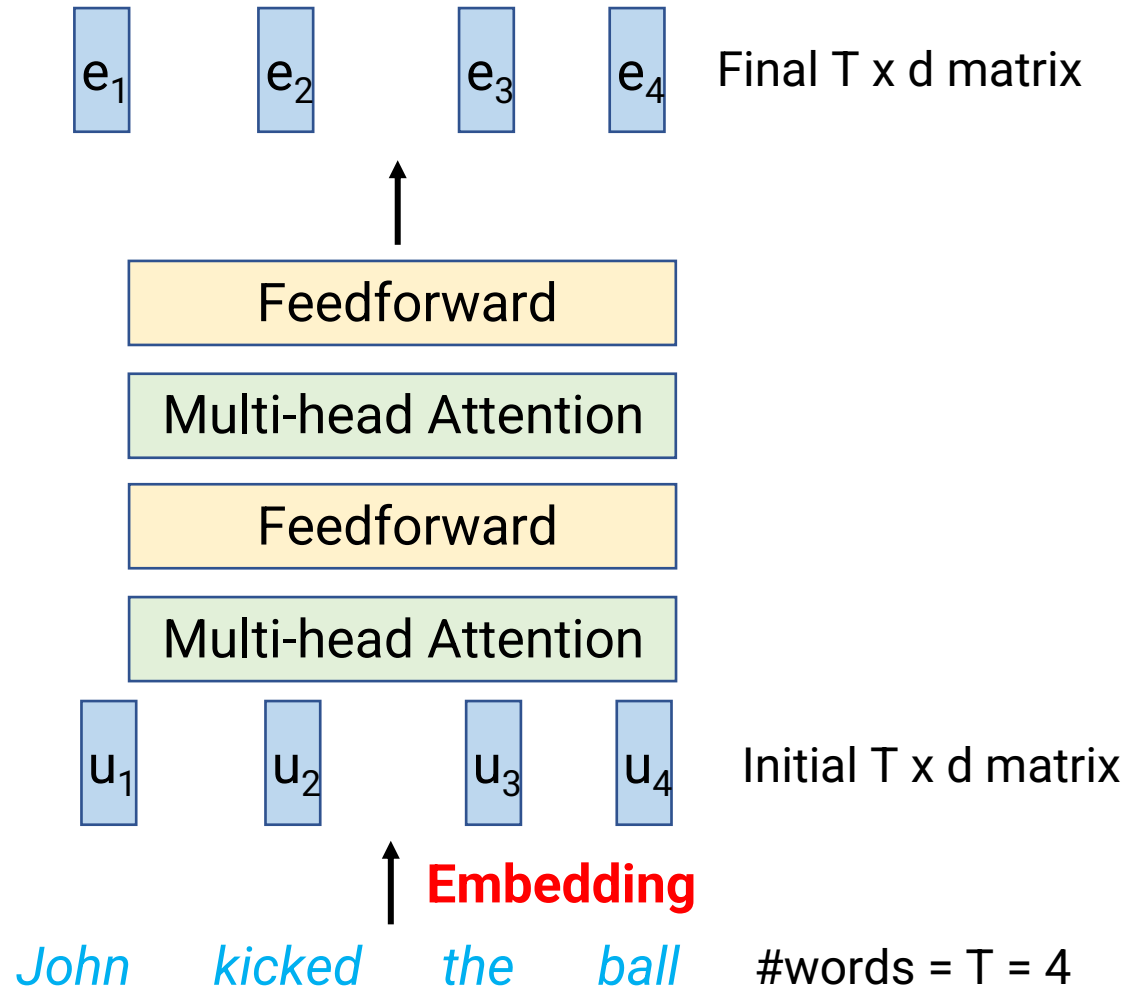
Input $x_1, \dots, x_T$, each shape d

What do attention heads learn?



- This attention head seems to go from a pronoun to its antecedent (who the pronoun refers to)
- Other heads may do more boring things, like point to the previous/next word
 - In this way, can do RNN-like things as needed
 - But attention also can reach across long ranges

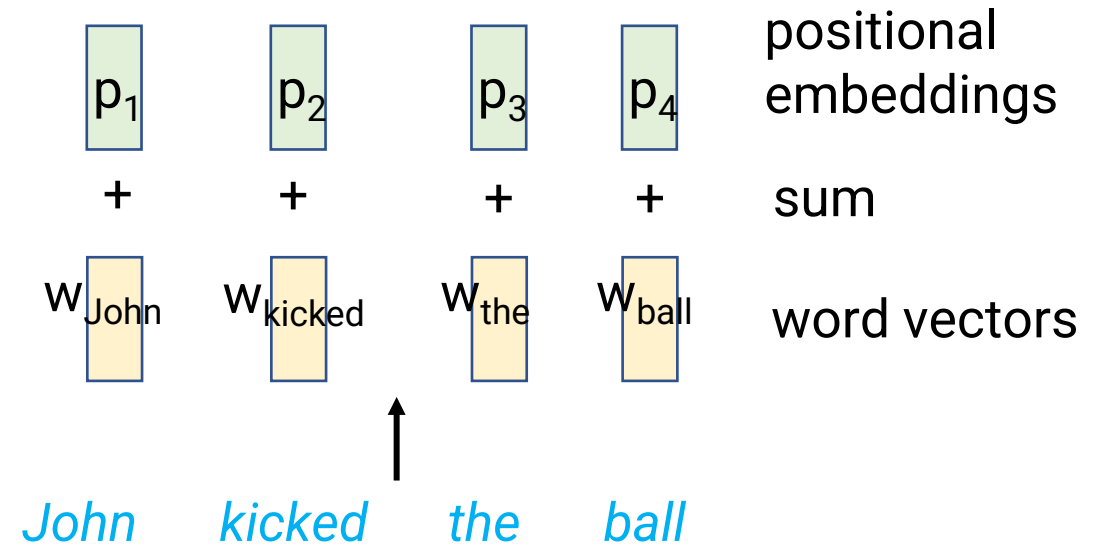
Transformer overview



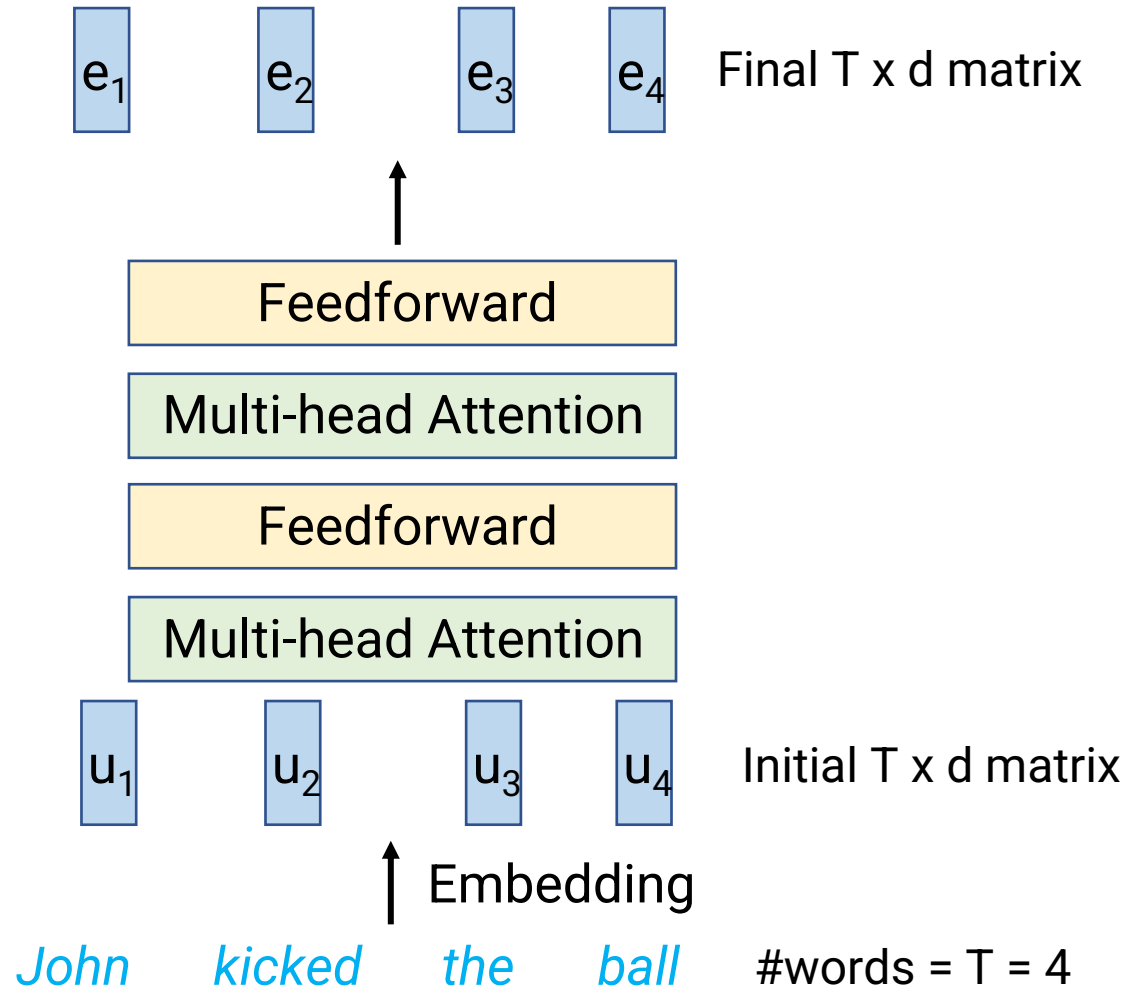
- One transformer consists of
 - **Initial embeddings** for each word of size d
 - Let T = #words, so initially we have a $T \times d$ matrix
 - Alternating layers of
 - "Multi-headed" attention layer
 - Feedforward layer
 - Both take in $T \times d$ matrix and output a new $T \times d$ matrix
 - Plus some bells and whistles...

Embedding layer

- As before, learn a vector for each word in vocabulary
- Is this enough?
 - Both attention and feedforward layers are **order invariant**
 - Need the initial embeddings to also encode order of words!
- Solution: **Positional embeddings**
 - Learn a different vector for each index
 - Gets added to word vector at that index

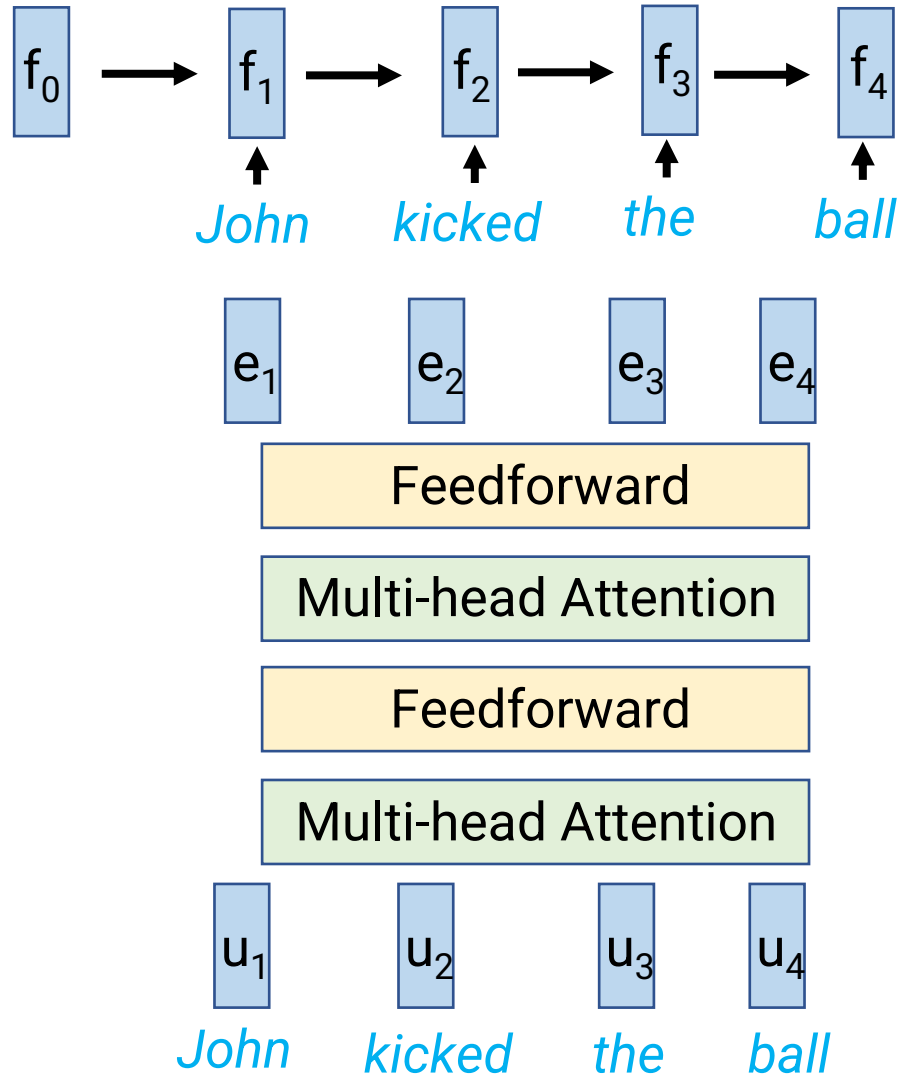


Transformer overview



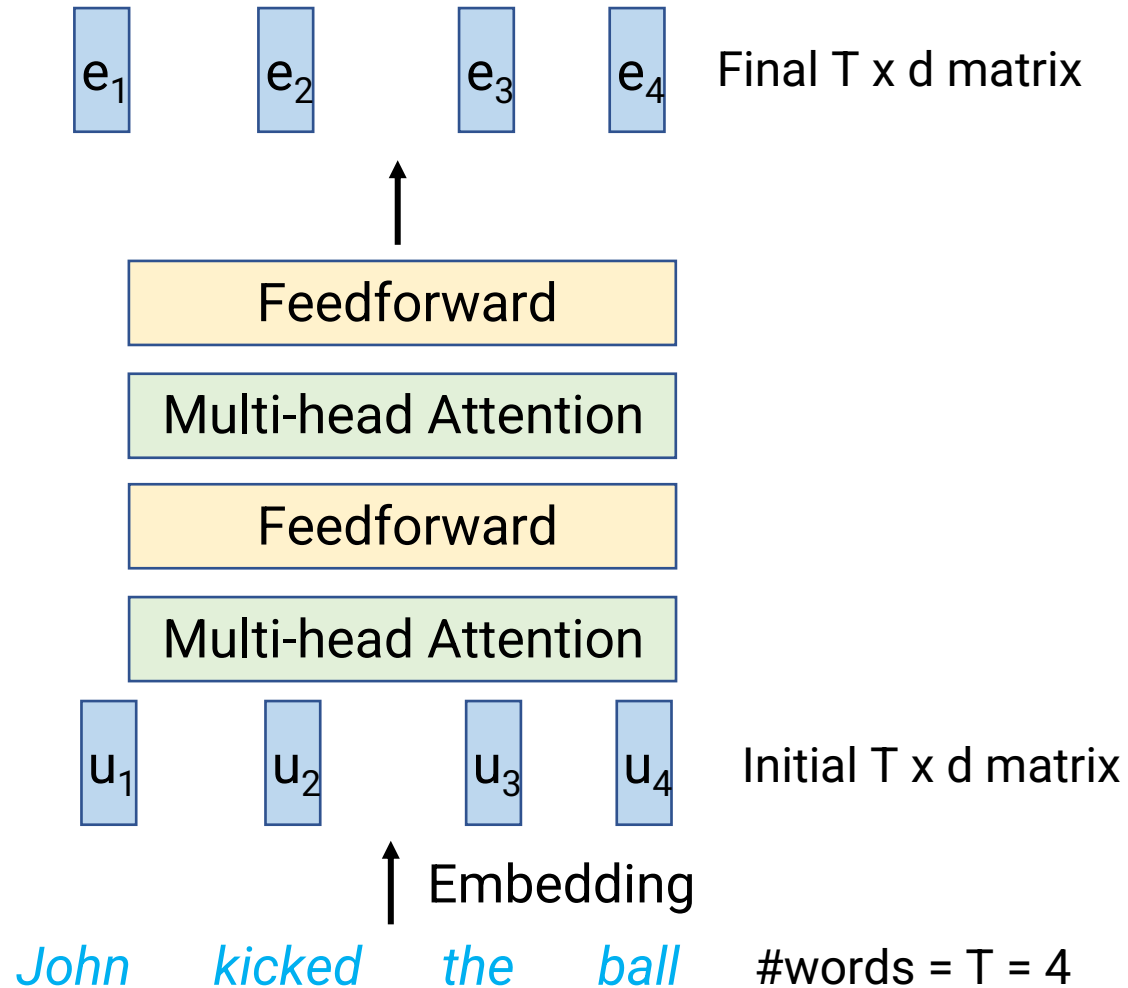
- How does a Transformer “work”?
- **Input layer:** Specify each word & its position in the sequence
- **Multi-headed attention layers:** For each word, retrieve information about related words, incorporate into the word’s representation
- **Feedforward layers:** Do additional non-linear processing of the information we have about the each word (independently)

Runtime comparison



- RNNs
 - Linear in sequence length
 - But all operations have to happen in series
- Transformers
 - Quadratic in sequence length ($T \times T$ matrices)
 - But can be parallelized (big matrix multiplication)

Transformer overview



- One transformer consists of
 - Initial embeddings for each word of size d
 - Let $T = \#words$, so initially we have a $T \times d$ matrix
 - Alternating layers of
 - "Multi-headed" attention layer
 - Feedforward layer
 - Both take in $T \times d$ matrix and output a new $T \times d$ matrix
 - **Plus some bells and whistles...more next time**

Conclusion: Transformers

- “Attention is all you need”
 - Get rid of recurrent connections
 - Instead, all “communication” between words in sequence is handled by attention
 - Have multiple attention “heads” to learn different types of relationships between words
- Most famous modern language models (e.g., ChatGPT) are Transformers!
- Next time: More Transformer details, Transformers as Decoders, Pre-training
- Later: Transformers + Reinforcement Learning = ChatGPT