Transformers, Part I

Robin Jia
USC CSCI 467, Spring 2025
March 25, 2025

Announcements

- Midterm grades released
  - Regrade requests open through this Friday, March 28
- Project midterm report due Tuesday, April 1
  - 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”)
  - If any questions/issues, reach out to your CP
- HW3 releasing soon, due April 15

Common Exam Mistakes: 2(d)i.

(d) Deqing switches from a linear regression model to a multi-layer perceptron with one hidden layer. Given an input $x \in \mathbb{R}^d$, this new model computes the function

$$f(x) = c^\top g(Ax + b) + d,$$

where $A \in \mathbb{R}^{h \times d}$, $b \in \mathbb{R}^h$, $c \in \mathbb{R}^h$, and $d \in \mathbb{R}$ are parameters. g is an activation function and h is the size of the hidden layer.

i. (5 points) Suppose that Deqing chooses g to be the identity function $g(z) = z$, and that he adds a Gaussian noise vector to the data just like in the earlier parts. This is equivalent in expectation to applying **what type** of regularization to **what quantity**?

- Most people got 2(c): When $f(x) = w^\top x + b$, this is applying L2 regularization to w
  - i.e., Trying to make the L2 norm of w small
- In this new network, we have $f(x) = (c^\top A)x + c^\top b + d$
- So you immediately conclude this is applying L2 regularization to $c^\top A$

Common Exam Mistakes: 3(a)

Consider the following operations that *may* be associated with the training of a neural network:

A. Acquire dataset Only do it once

B. Apply kernel trick Irrelevant

C. Backward pass 2. Requires forward pass to compute gradients

D. Forward pass 1. Has to come first

E. Group training data into batches Only do it once per epoch

F. Manually derive formula for the gradient Not needed thanks to backpropagation

G. Solve normal equations Irrelevant

H. Update model parameters 4. Parameters "move" in direction of velocity

I. Update velocity term for momentum 3. Velocity formula depends on gradient

(a) (6 points) From the above choices, list all of the operations that must occur in **every** step of stochastic gradient descent, in the **order** that they must be performed. Assume we are training a neural network using momentum.

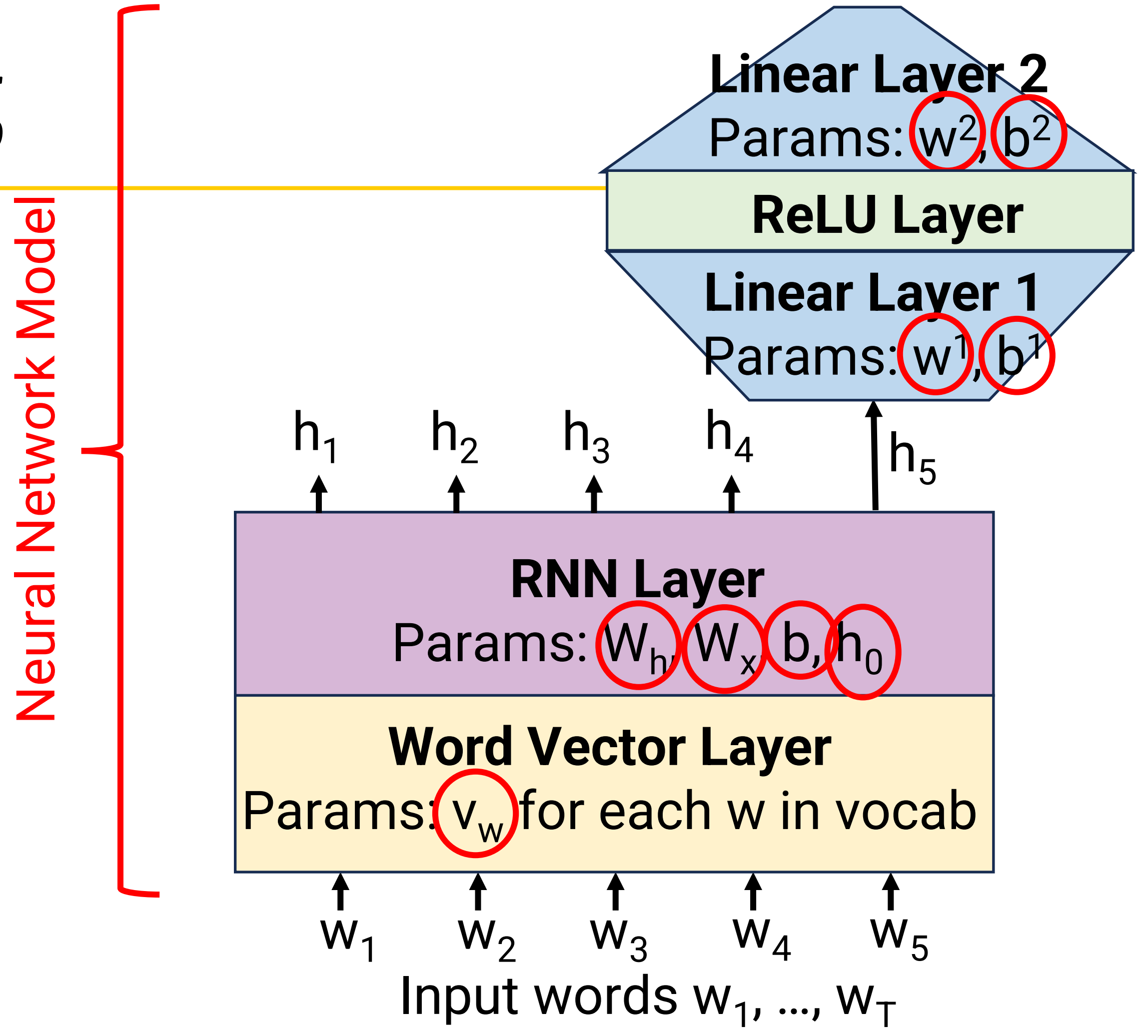
Common Exam Mistakes: 4(d)

(d) (2 points) **True** or $\boxed{\textbf{False:}}$ A multi-layer perceptron with two hidden layers can express functions that cannot be approximated by any multi-layer perceptron with one hidden layer.

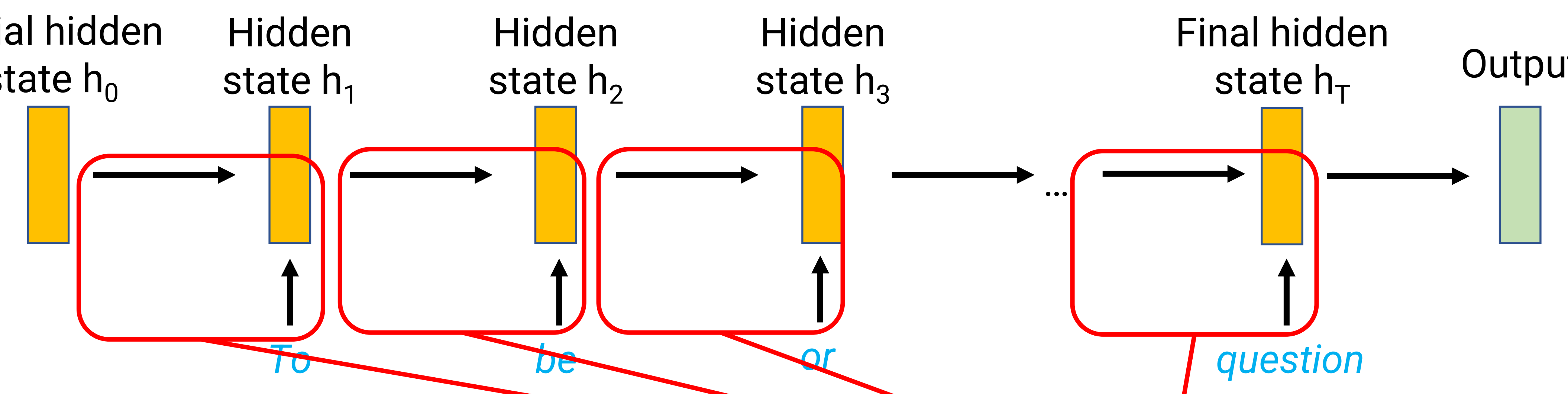
- Sounds true but is false!
  - Having 2 hidden layers sounds more powerful than having 1 hidden layer
- But 1 hidden layer networks are **universal approximators**—can approximate any other function
- How is this possible? You might need a **much wider** network with 1 hidden layer to approximate a network with 2 hidden layersould be impractical
- So in practice, deeper networks often work better

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



Review: RNNs

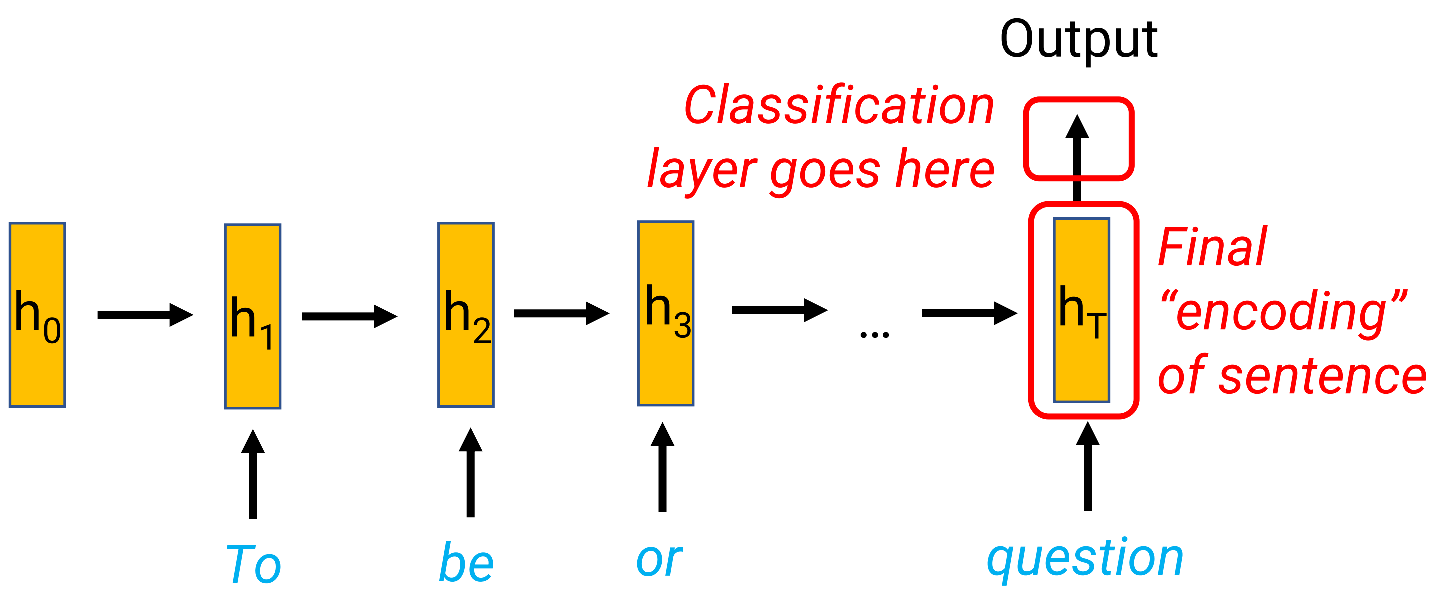


$$h_t = \tanh\left(\underbrace{W_h h_{t-1}}_{\text{Linear function of prev. hidden state}} + \underbrace{W_x x_t}_{\text{Linear function of current word vector}} + b\right)$$

- 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

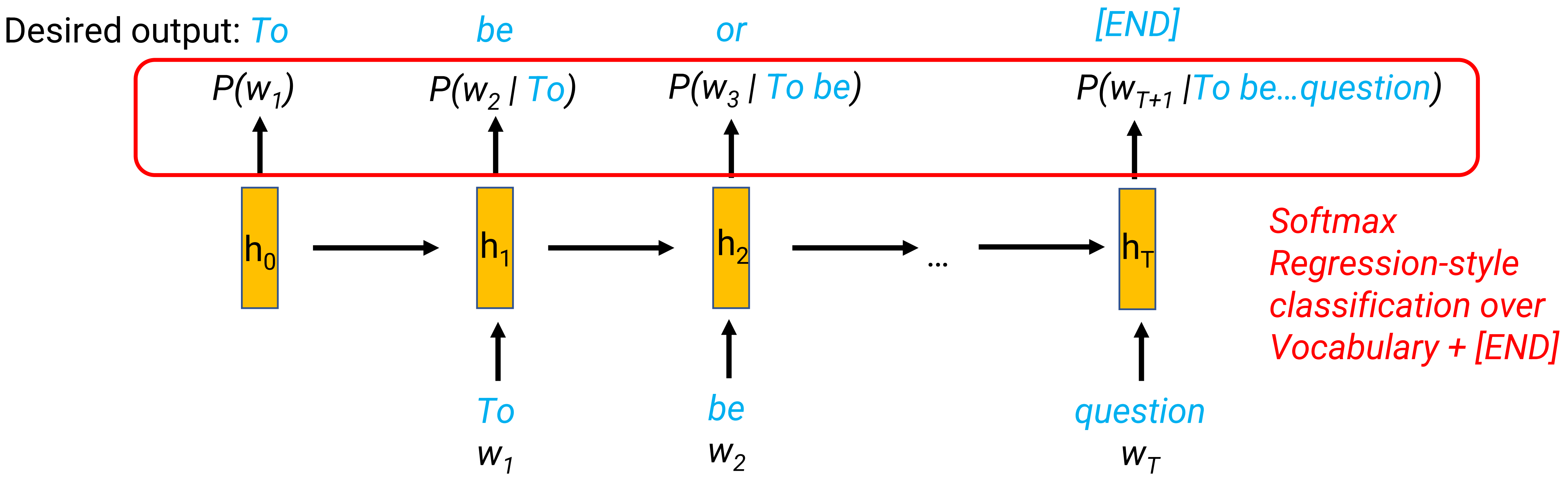
Encoder model: Converts sentence to vector “encoding”



- 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)

RNNs

- Process a sentence **one word at a time**
- Each “step” of computation is reading **one more word** (time dimension)
- Each hidden state encodes **information about sentence up to the current word**

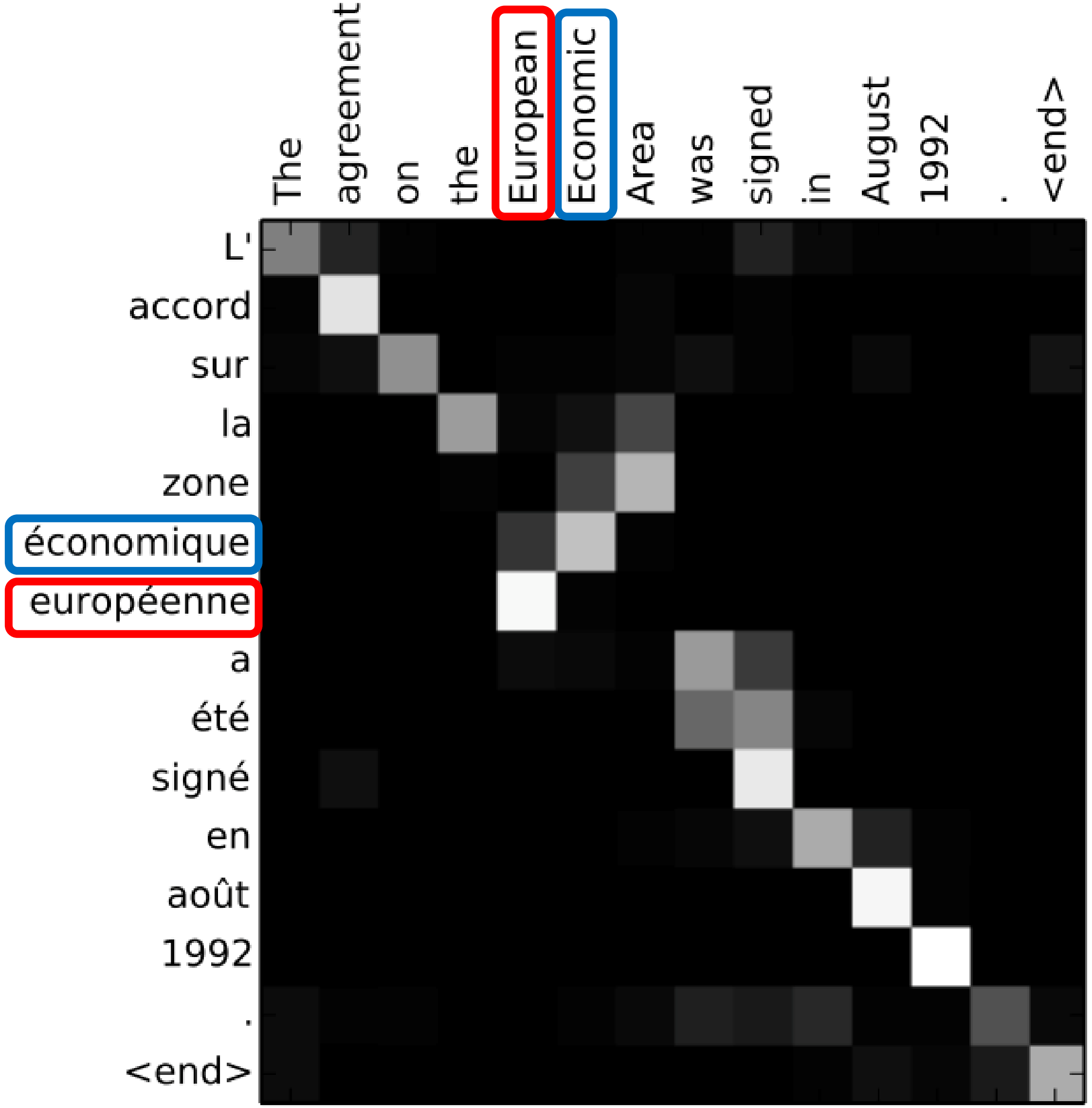
Both

- Input = sequence of vectors, representing words
- Output = sequence of hidden state vectors, one for each input word

Transformers

- Process **all words of the sentence at the same time** (in parallel)
- Each “step” of computation is applying **one more layer** (depth dimension; more like a CNN)
- Each hidden state encodes **information about that word in the context of the whole sentence**

Review: Challenges of modeling sequences



- Modeling relationships between words
  - Translation alignment

Review: Challenges of modeling sequences

Goes with "steak"

*He ate **steak** with **ketchup***

Modifies "ate"

*He **ate** steak with a **fork***

- Modeling relationships between words
  - Translation alignment
  - Syntactic dependencies

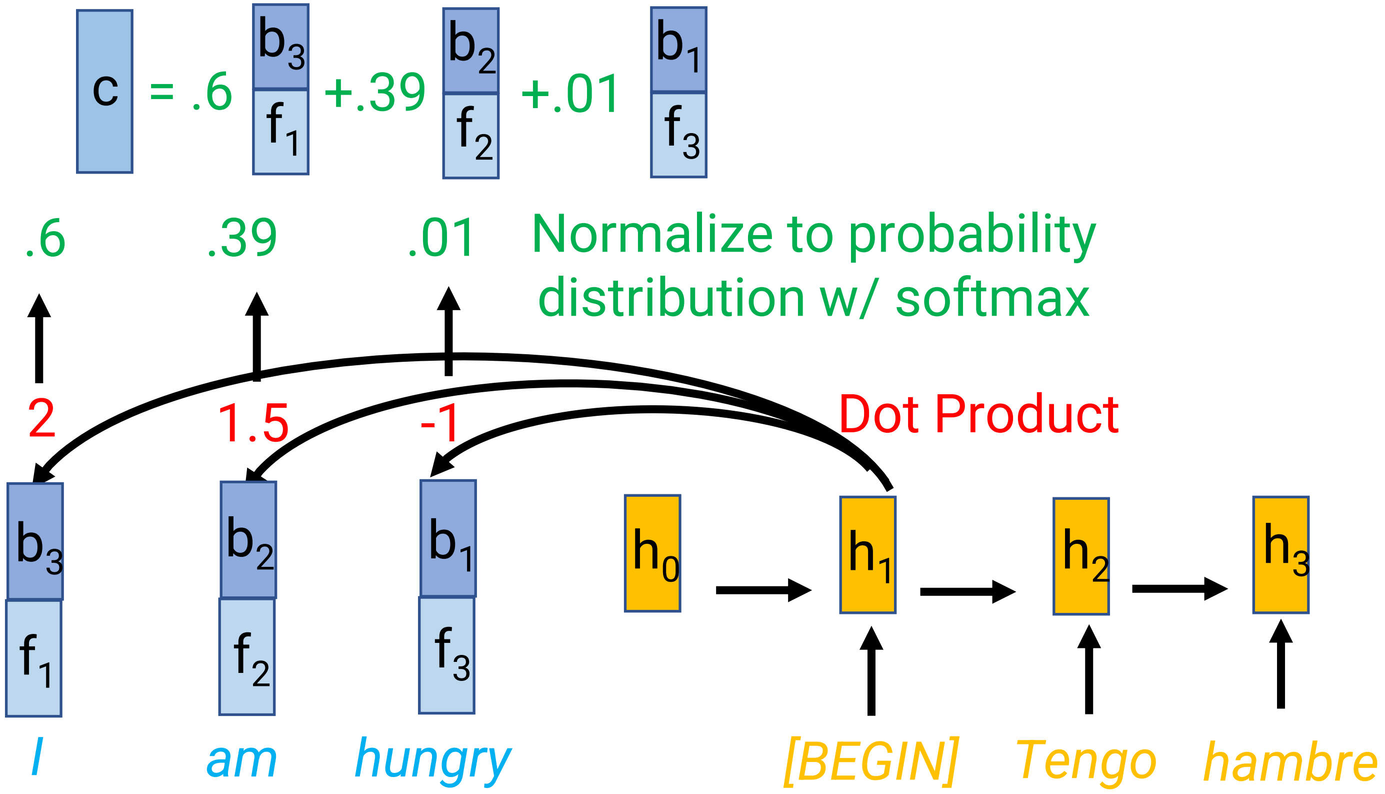
Review: Challenges of modeling sequences

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

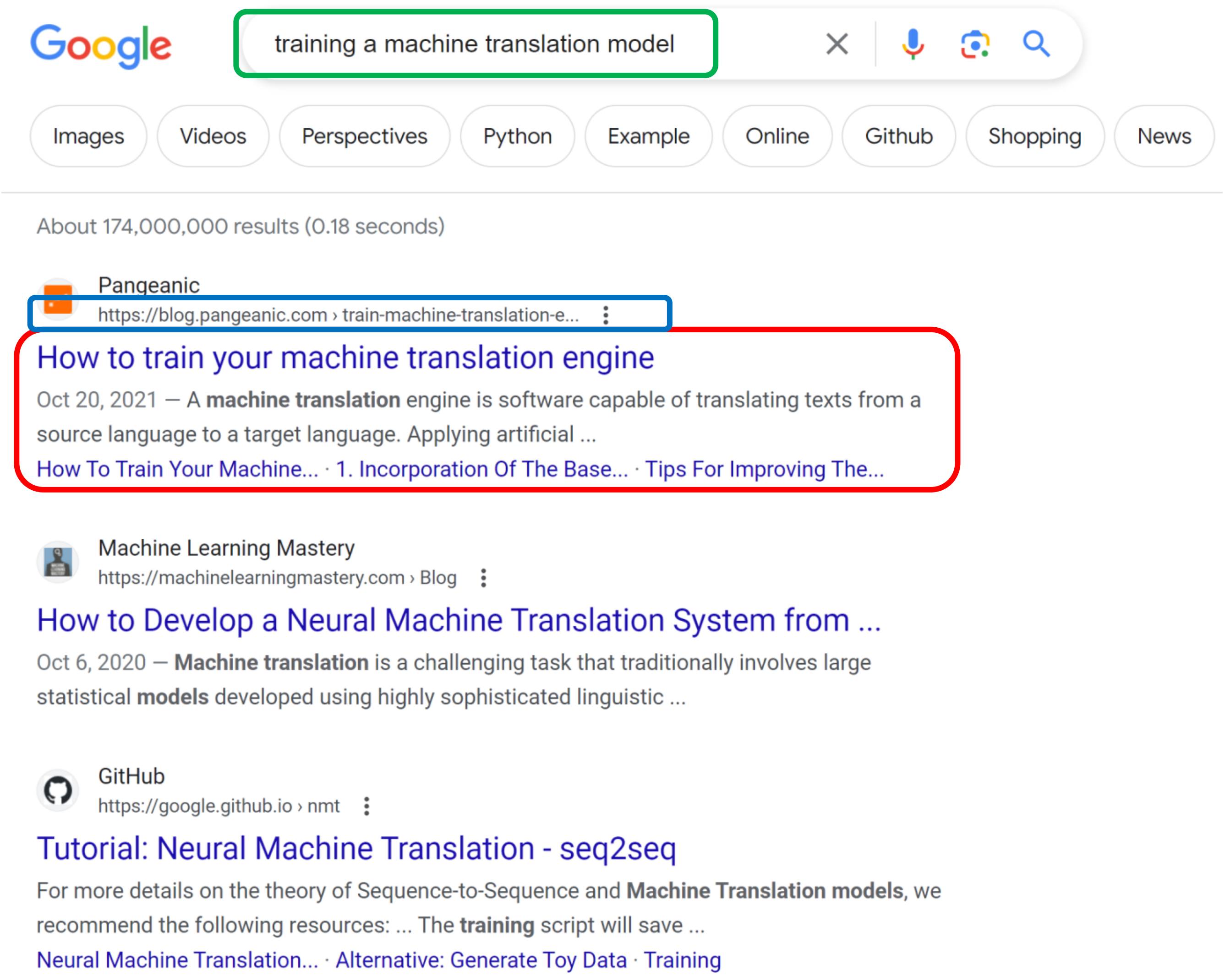
- Modeling relationships between words
  - Translation alignment
  - Syntactic dependencies
  - Coreference relationships

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



- 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 $k_1, \ldots, k_T$
  - Value vectors $v_1, \ldots, v_T$
- Output (also vector of length *d*)
  - Dot product q with each key vector k_t to get score s_t:

How well does the query match each key?

$$s_t = q^\top k_t$$

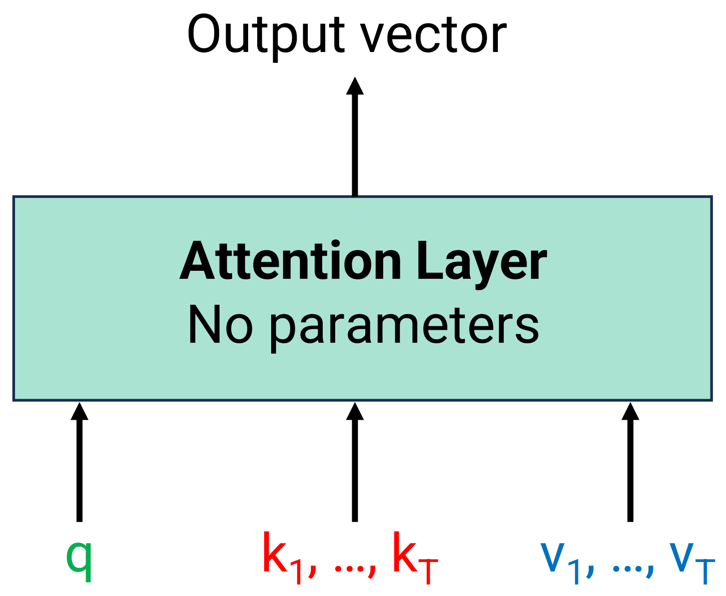
  - Softmax to get probability distribution $p_1, \ldots, p_T$:

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

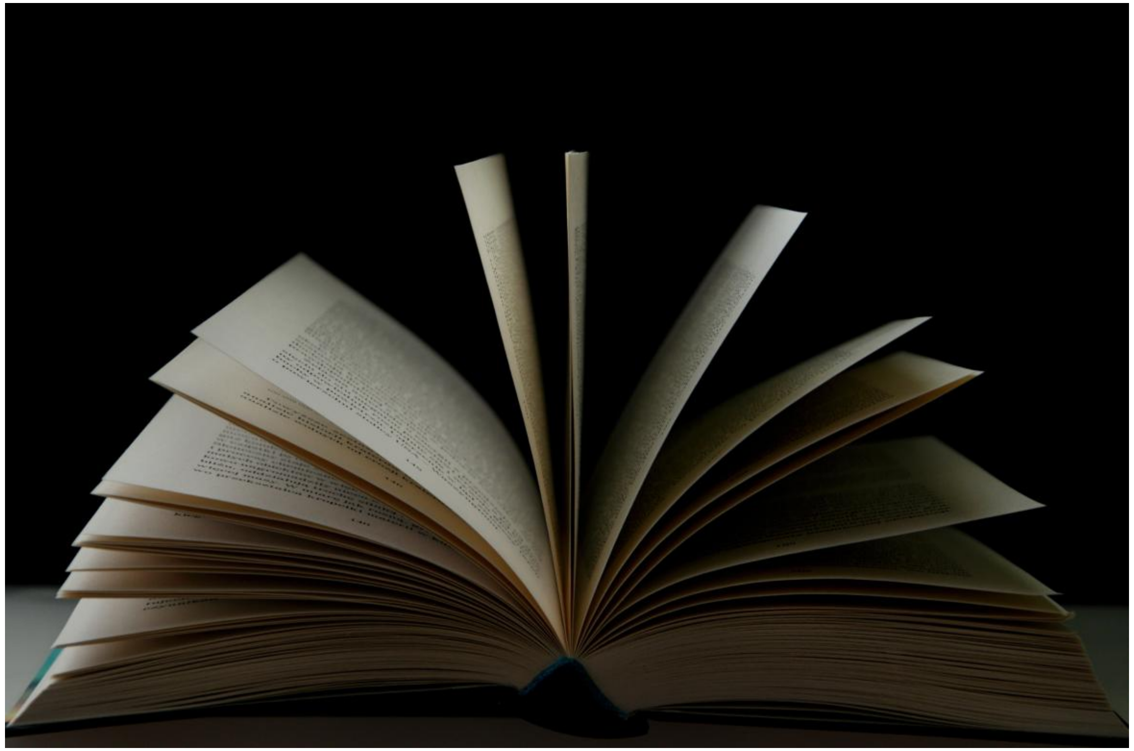
  - Return weighted average of value vectors:

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

Dominated by the values corresponding to the "best-matching" keys

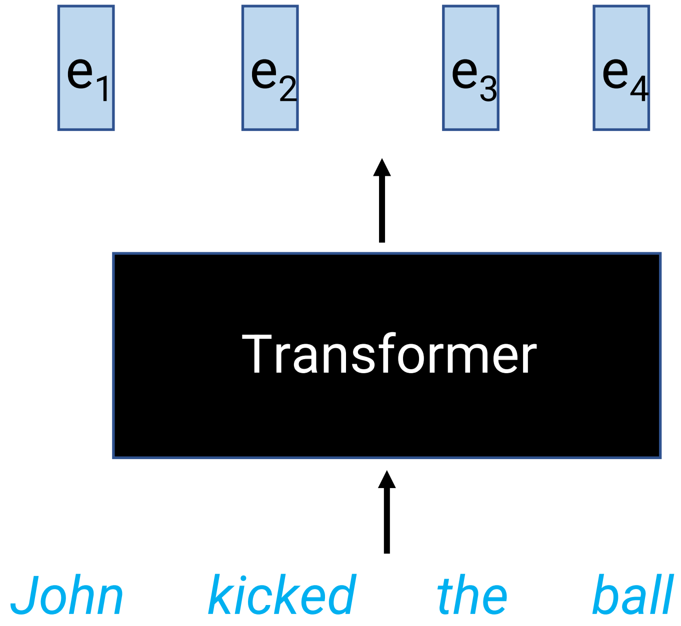


Today: Can we use Attention for Everything?



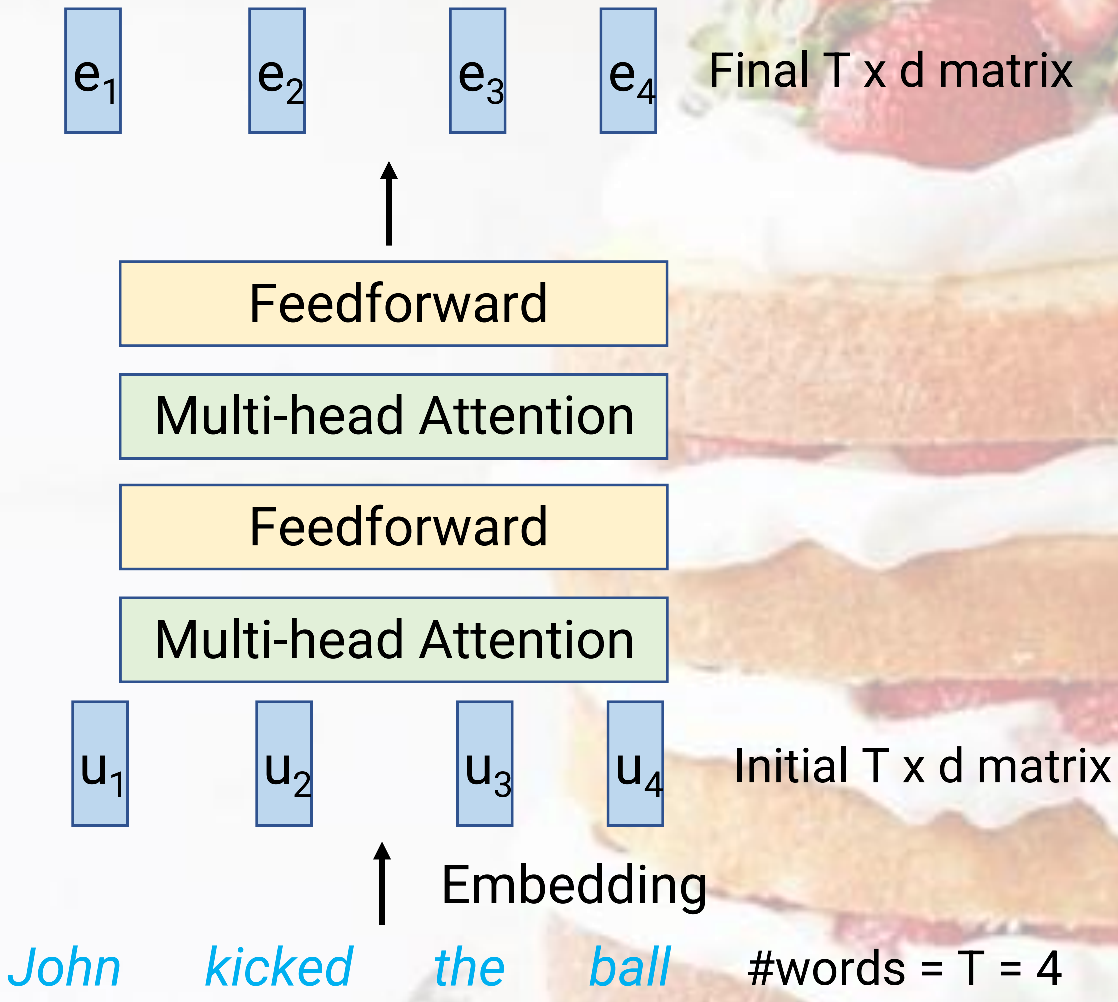
- Modeling relationships between words
  - Translation alignment
  - Syntactic dependencies
  - Coreference relationships
- Long range dependencies
  - E.g., consistency of characters in a novel
- Attention captures relationships & doesn't care about "distance," unlike RNNs
- **Let's replace RNN's with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

Today: The Transformer Architecture



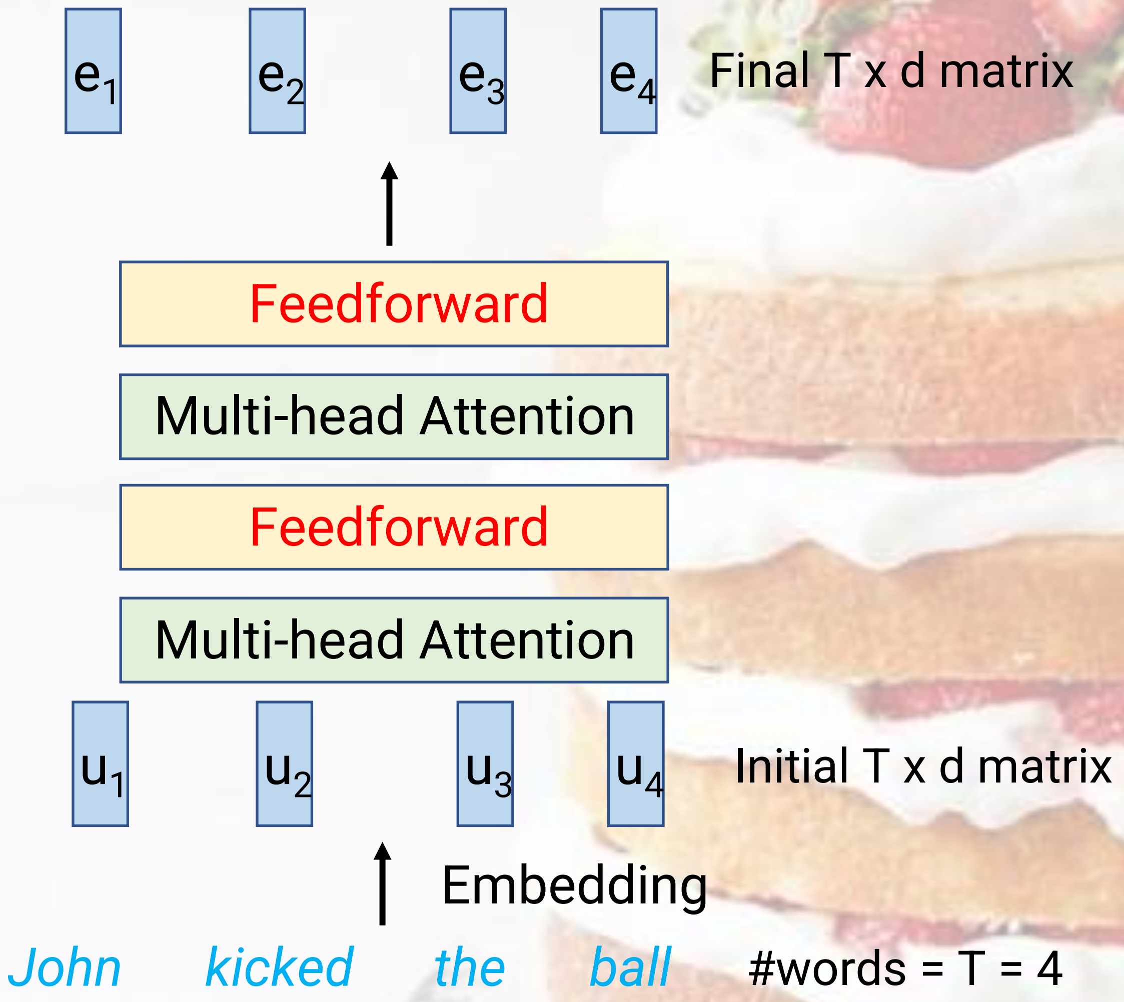
- Input: Sequence of words
- Output: Sequence of hidden state vectors, one per word
- Same "type signature" as RNN
- Motivation
  - Process all words at the same time, don't do explicit sequential processing
  - Let **attention** figure out which words are relevant to each other
    - Whereas RNN assumes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

Transformer overview



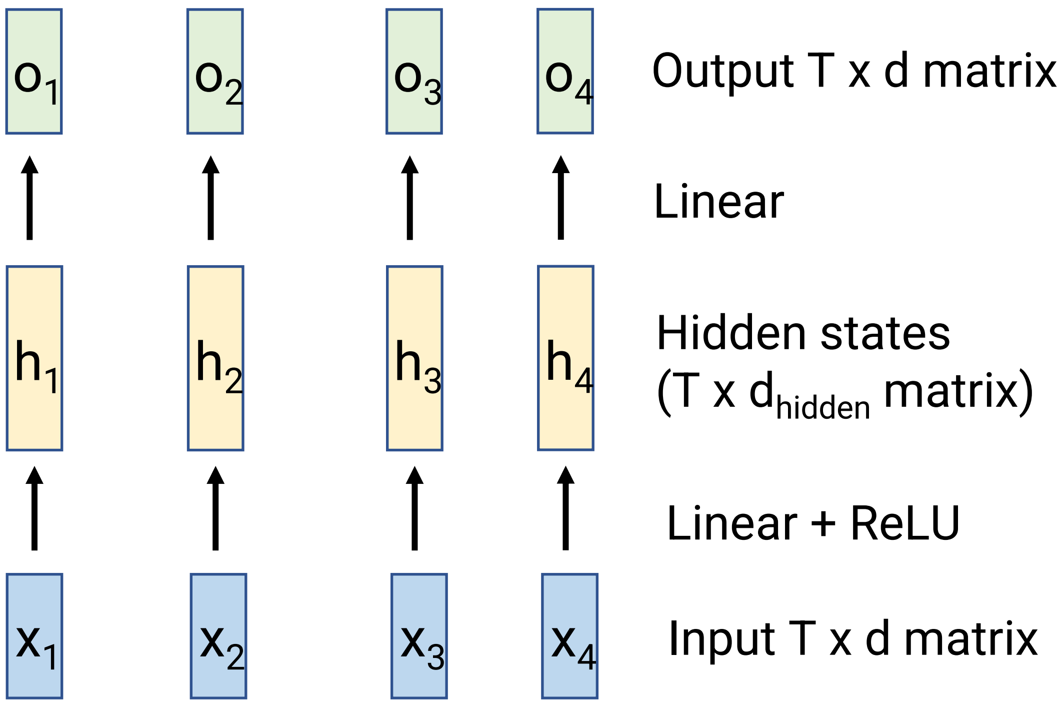
- One transformer consists of
  - Initial embeddings for each word of size d
    - Let T =#words, so initially we have a T x d matrix
  - Alternating layers of
    - “Multi-headed” attention layer
    - Feedforward layer
    - Both take in T x d matrix and output a new T x d matrix
  - Plus some bells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

Transformer overview



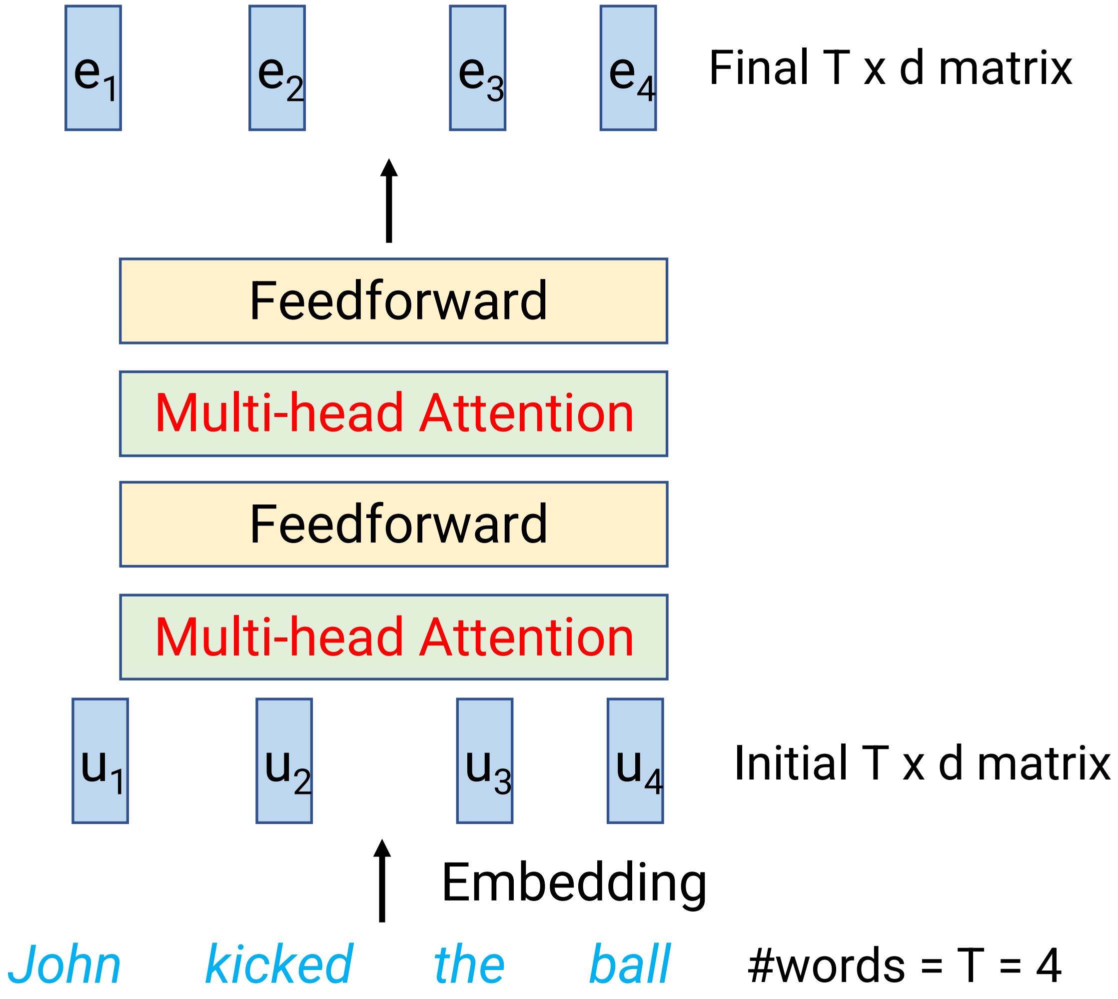
- One transformer consists of
  - Initial embeddings for each word of size d
    - Let T =#words, so initially we have a T x d matrix
  - Alternating layers of
    - "Multi-headed" attention layer
    - **Feedforward layer**
    - Both take in T x d matrix and output a new T x d matrix
  - Plus some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

Feedforward layer



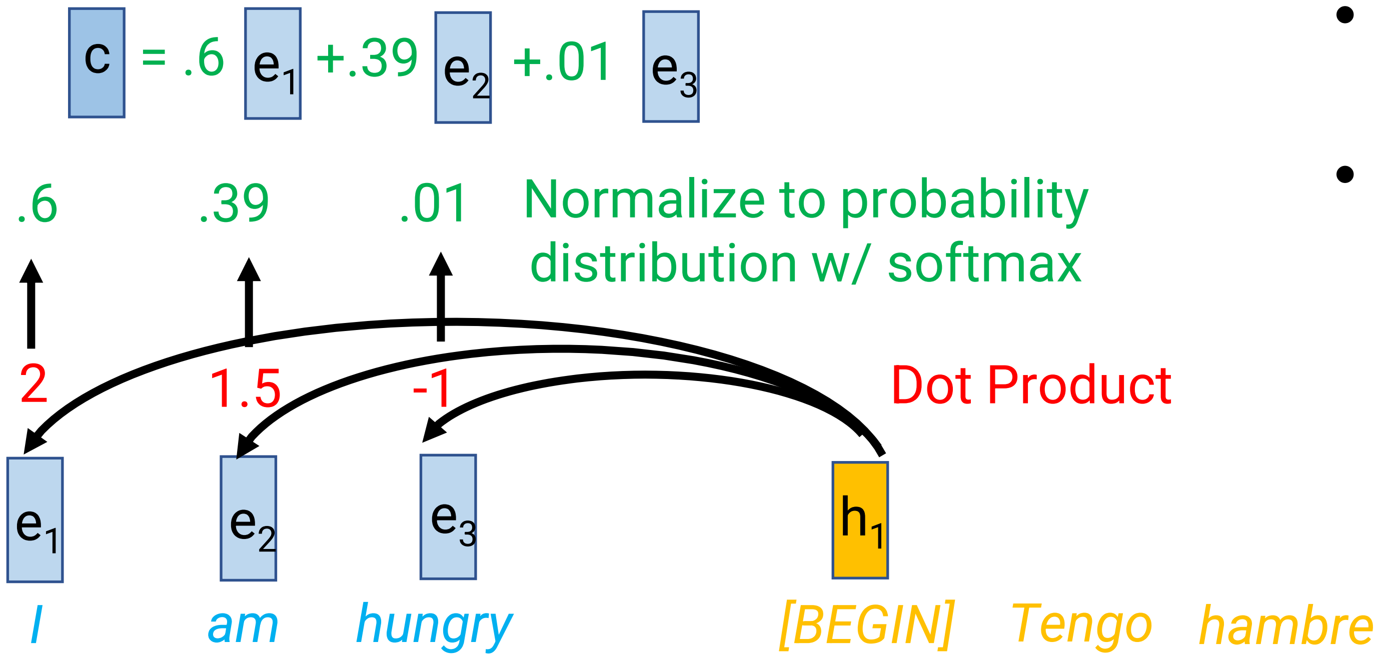
- Input: T x d matrix
- Output: Another T x d matrix
- Apply the same MLP separately to each d-dimensional vector
    - Linear layer from d to d_{hidden}
    - ReLU (or other nonlinearity)
    - Linear layer from d_{hidden} to d
- Note: No information moves between tokens here

Transformer overview



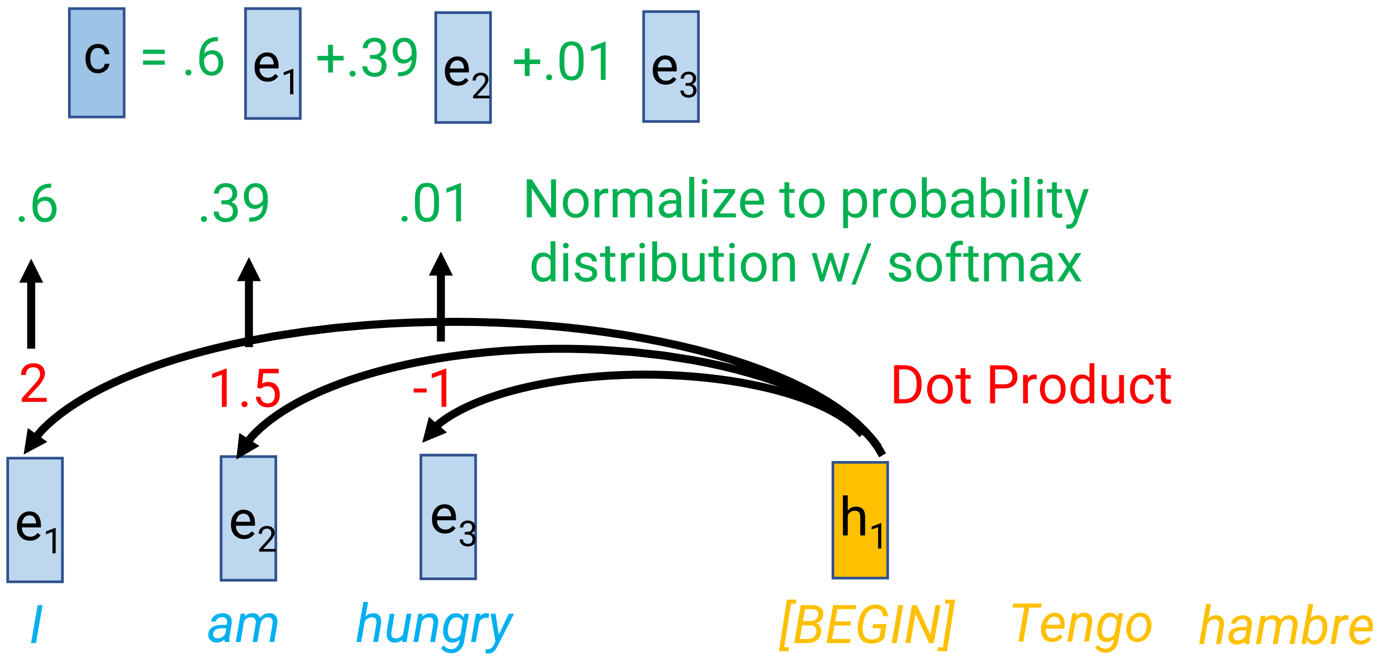
- One transformer consists of
  - Initial embeddings for each word of size d
    - Let T =#words, so initially we have a T x d matrix
  - Alternating layers of
    - **"Multi-headed" attention layer**
    - Feedforward layer
    - Both take in T x d matrix and output a new T x d matrix
  - Plus some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

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

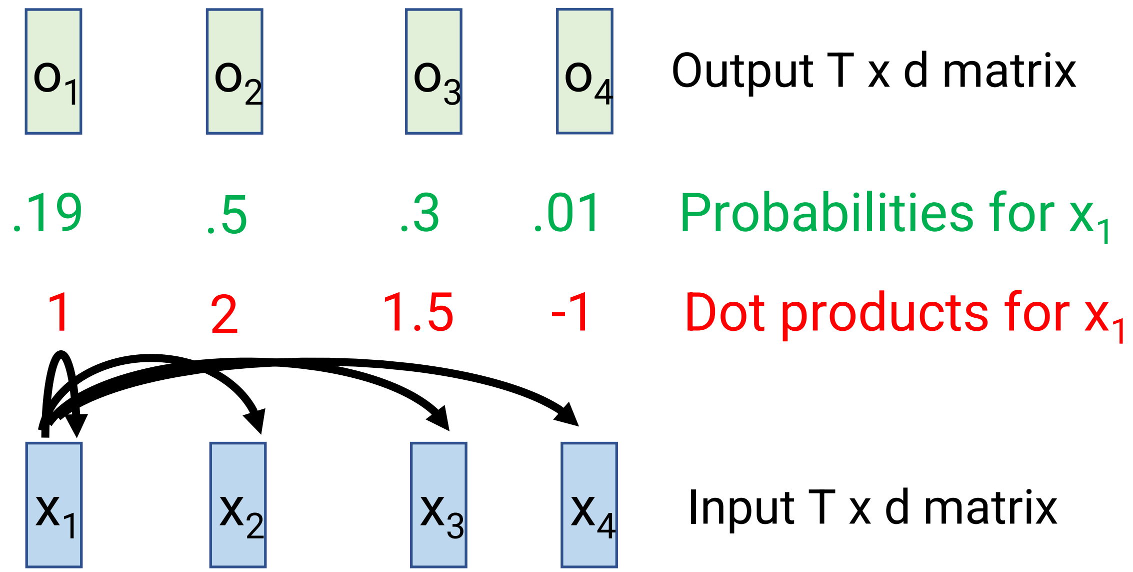
.19 .5 .3 .01 Probabilities for x_1

1 2 1.5 -1 Dot products for x_1

x_1 x_2 x_3 x_4 Input T x d matrix

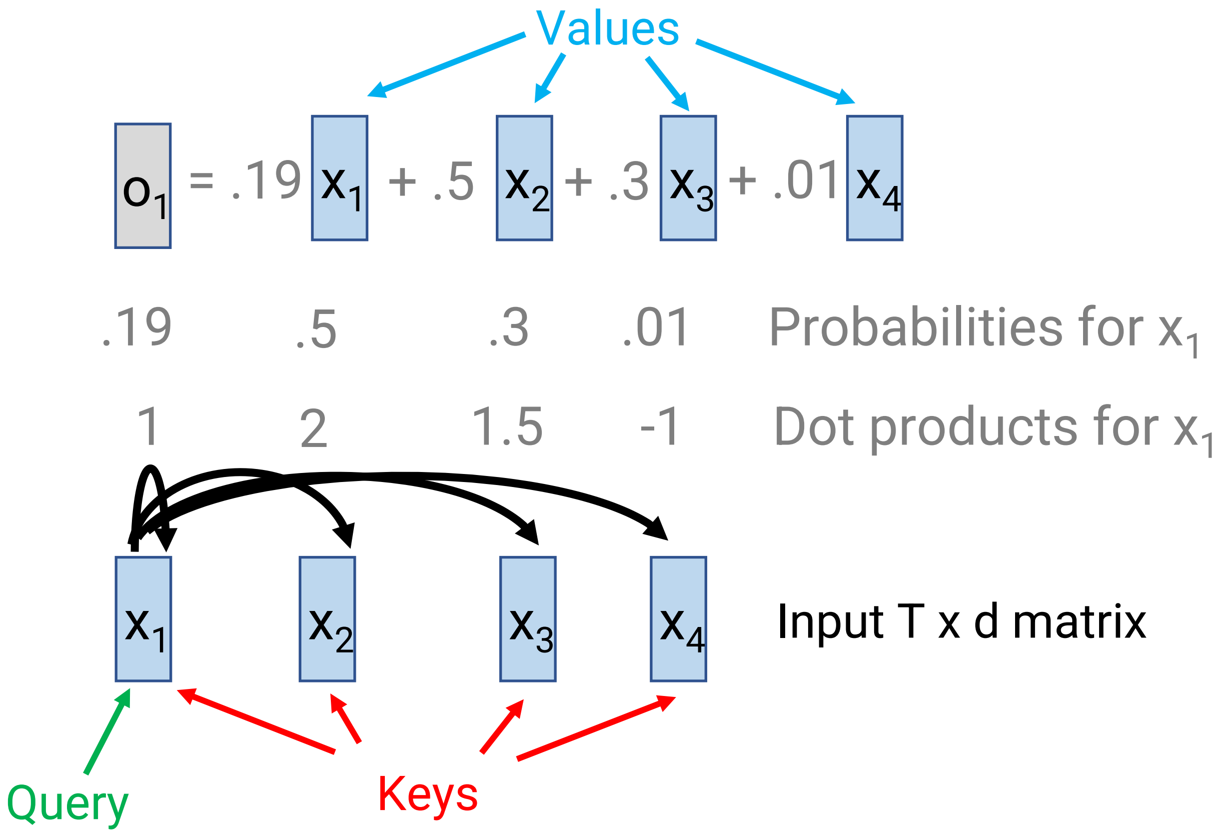
- Take x_1 and dot product it with all T inputs (including itself)
- Apply softmax to convert to probability distribution
- Compute output o_1 as weighted sum of inputs

Change #1: Self-attention



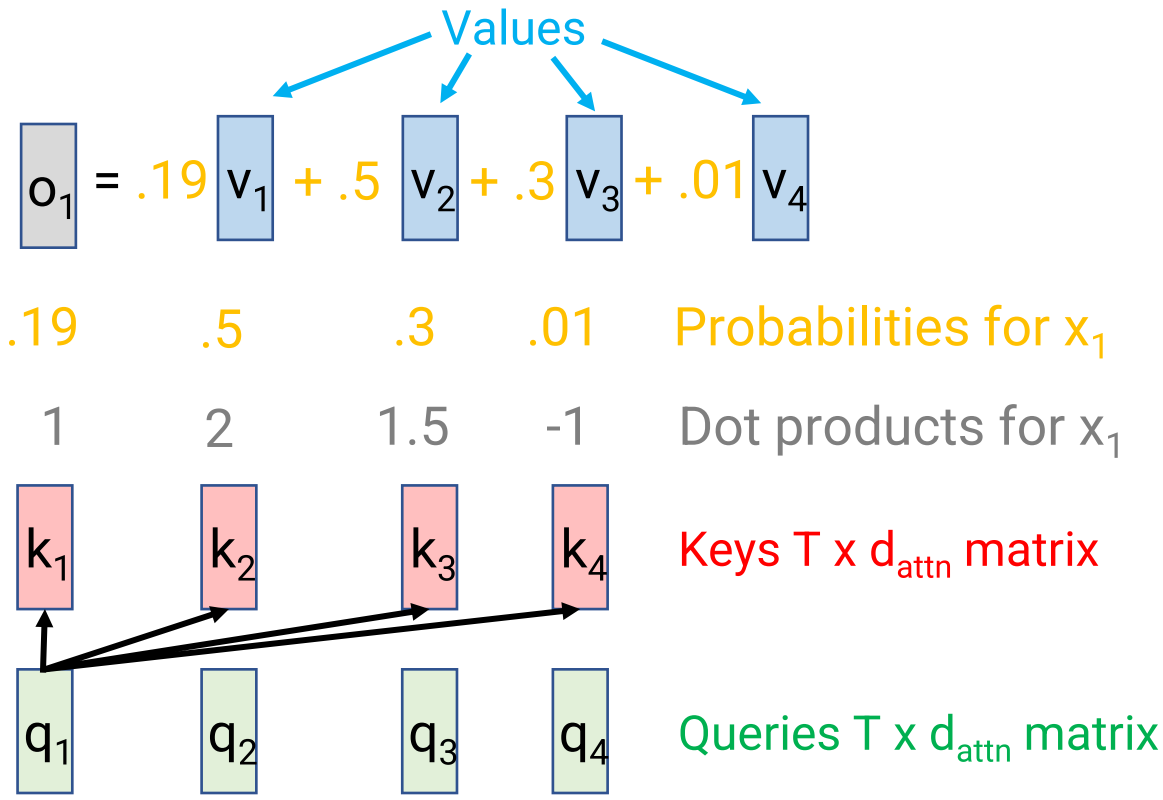
- Take x_1 and dot product it with all T inputs (including itself)
- Apply softmax to convert to probability distribution
- Compute output o_1 as weighted sum of inputs
- Repeat for t=2, 3, …, T
- Replacement for recurrence
    - RNN only allows information to flow linearly along sequence
    - Now,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

Change #2: Separate queries, keys, and values



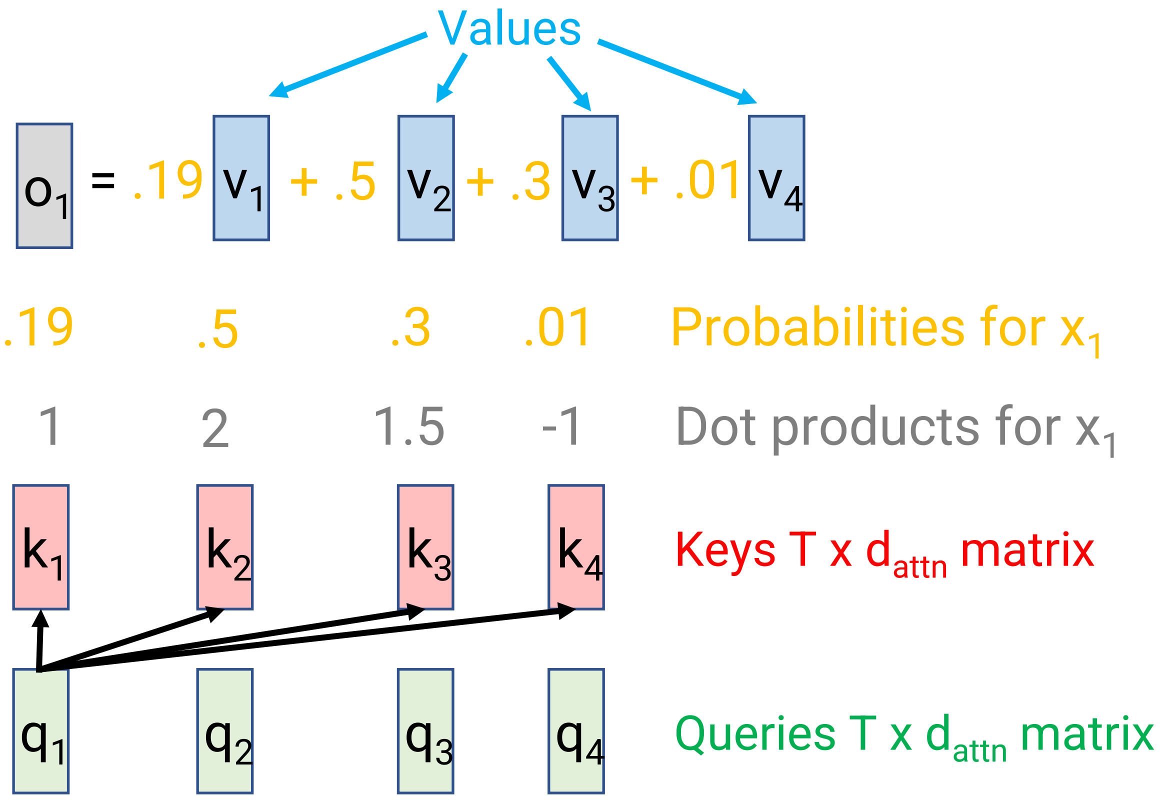
- Recall: Attention uses vectors in three different ways
  - As "query" for current index
  - As "keys" to match with query
  - As "values" when computing output
- Idea: Use separate vectors for each usage
  - What each index "looks for" different from what it "matches with"
  - What you store in output 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, …, x_T to get
  - Queries $[q_1, \ldots, q_T]$, each $q_t = W^Q * x_t$
  - Keys $[k_1, \ldots, k_T]$, each $k_t = W^K * x_t$
  - Values $[v_1, \ldots, v_T]$, 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 $[k_1, \ldots, k_T]$
- Apply softmax to get probability distribution
- Compute o_1 as weighted sum of $[v_1, \ldots, v_T]$
- Repeat for t = 2, …, T

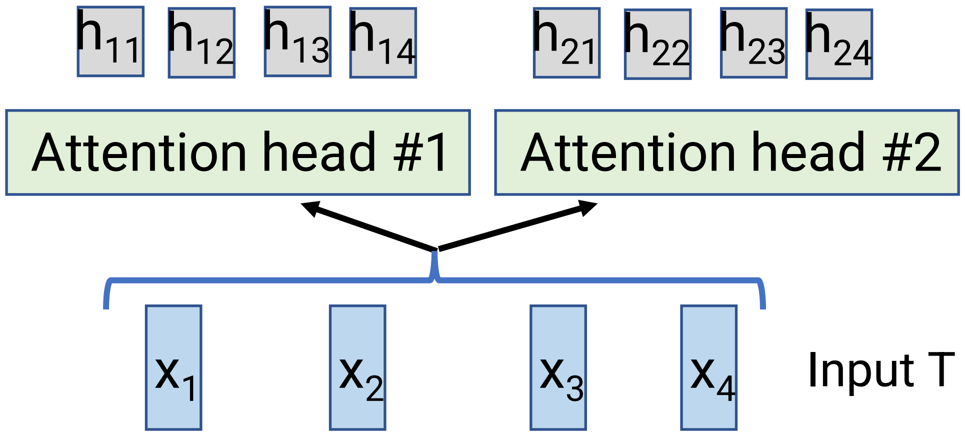
Matrix form



- 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 x d_{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 x K^T (T x 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 x V

Change #3: Making it Multi-headed

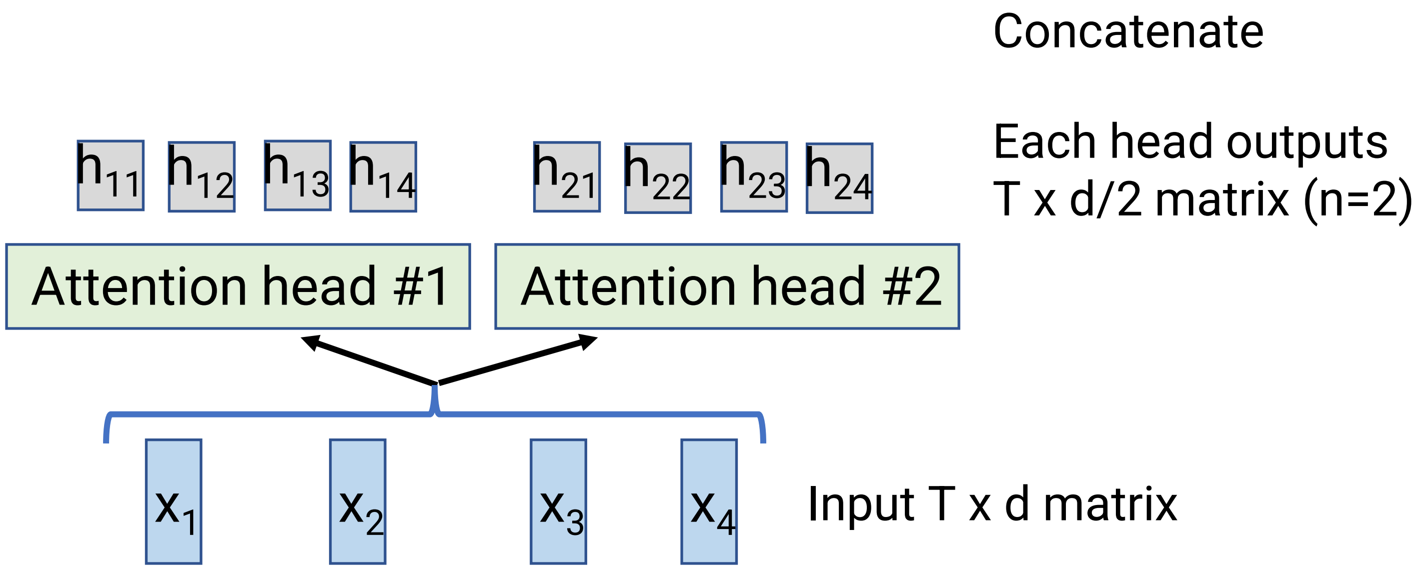


Each head outputs T x d/2 matrix (n=2)

Input T x d matrix

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

Change #3: Making it Multi-headed



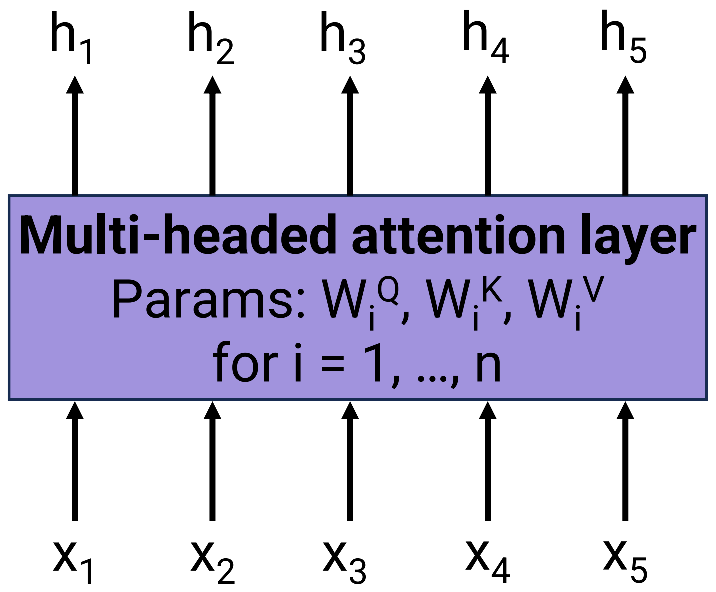
- Instead of doing attention once, have n different "heads"
  - Each has its own W^Q, W^K, W^V parameters that map to dimension $d_{attn} = d/n$
  - Concatenate at end to get output of size T x d
- Why? Different heads can capture different relationships between words

The Multi-headed Attention building block

(9) Multi-headed Attention Layer

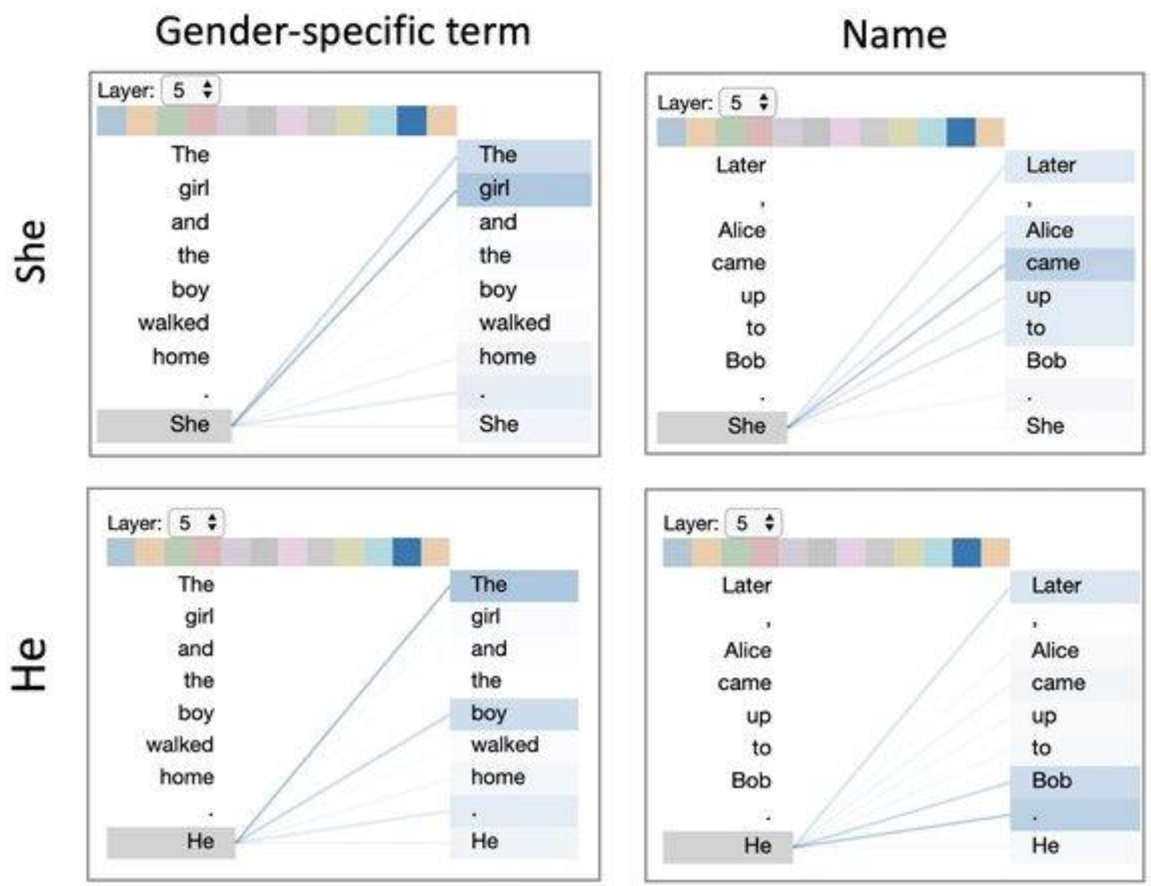
- Input: List of vectors $x_1, \ldots, x_T$, each of size d
  - Equivalent to a T x d matrix
- Output: List of vectors $h_1, \ldots, h_t$, each of size d
  - Equivalent to another T x d matrix
- Formula: For each head i:
  - Compute Q, K, V matrices using W_i^Q, W_i^K, W_i^V
  - Compute self attention output using Q, K, V to yield T x d_{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_{attn} x d
  - (# of heads n is hyperparameter, $d_{attn} = d/n$)
- In pytorch: nn.MultiheadAttention()

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



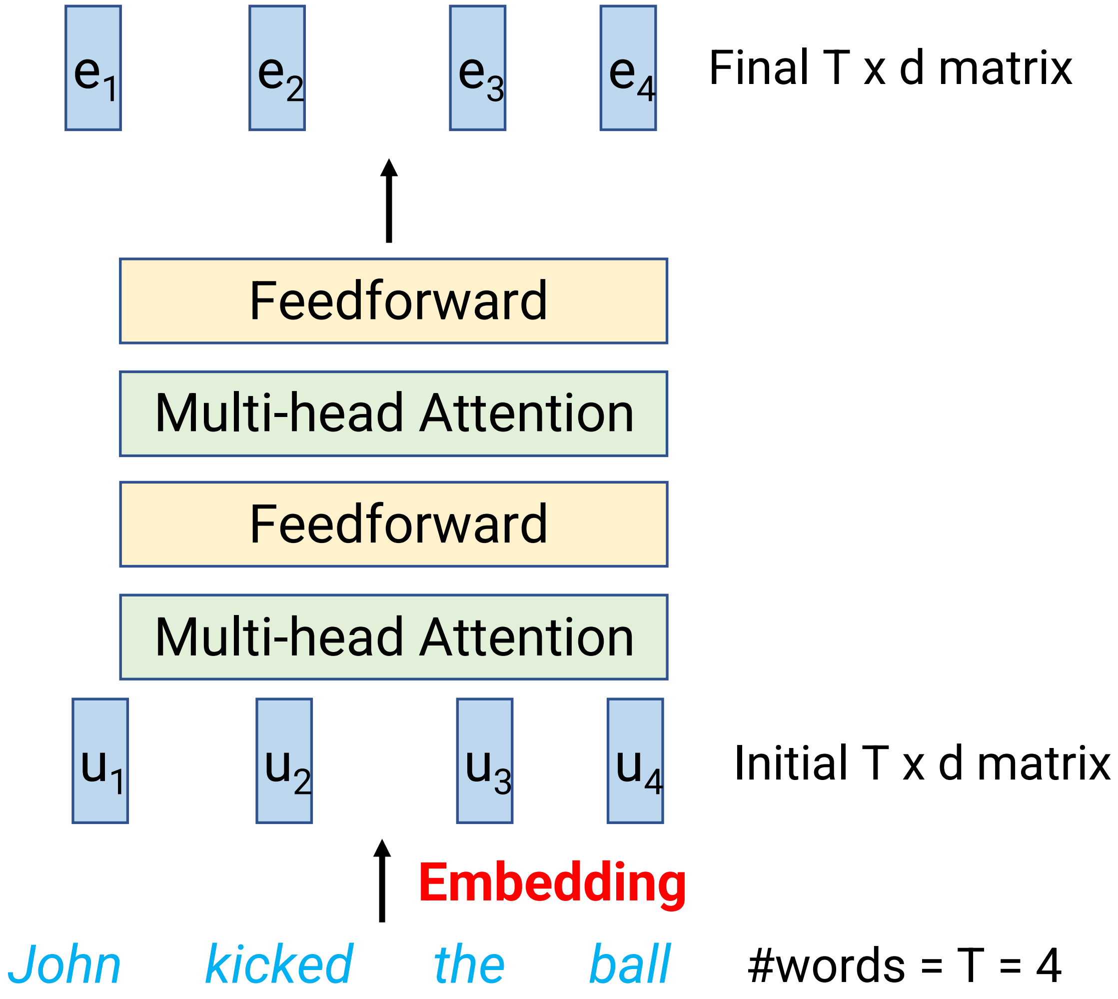
Input $x_1, \ldots, x_T$, each shape d

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

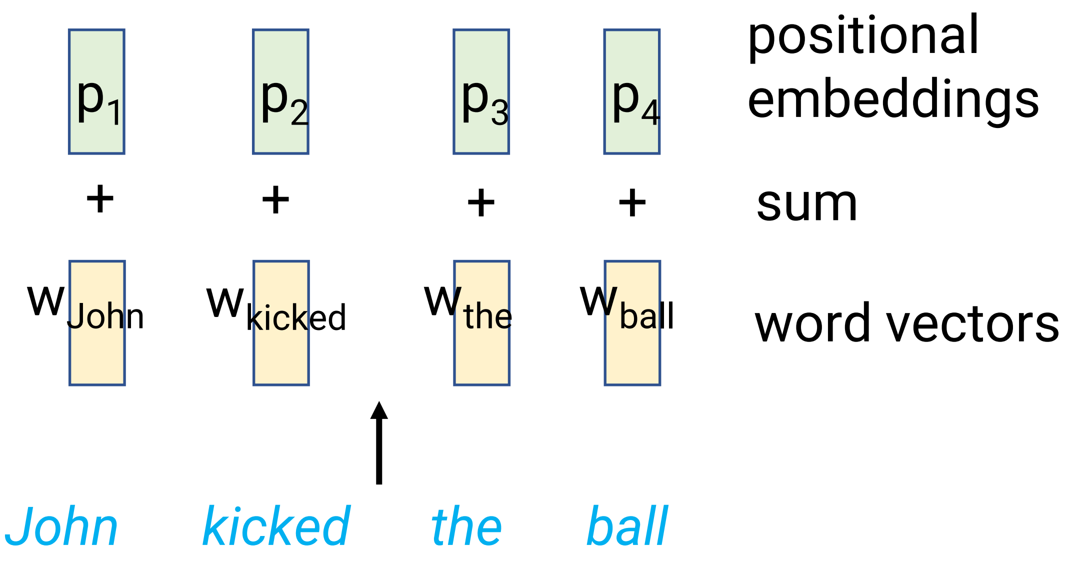
Transformer overview



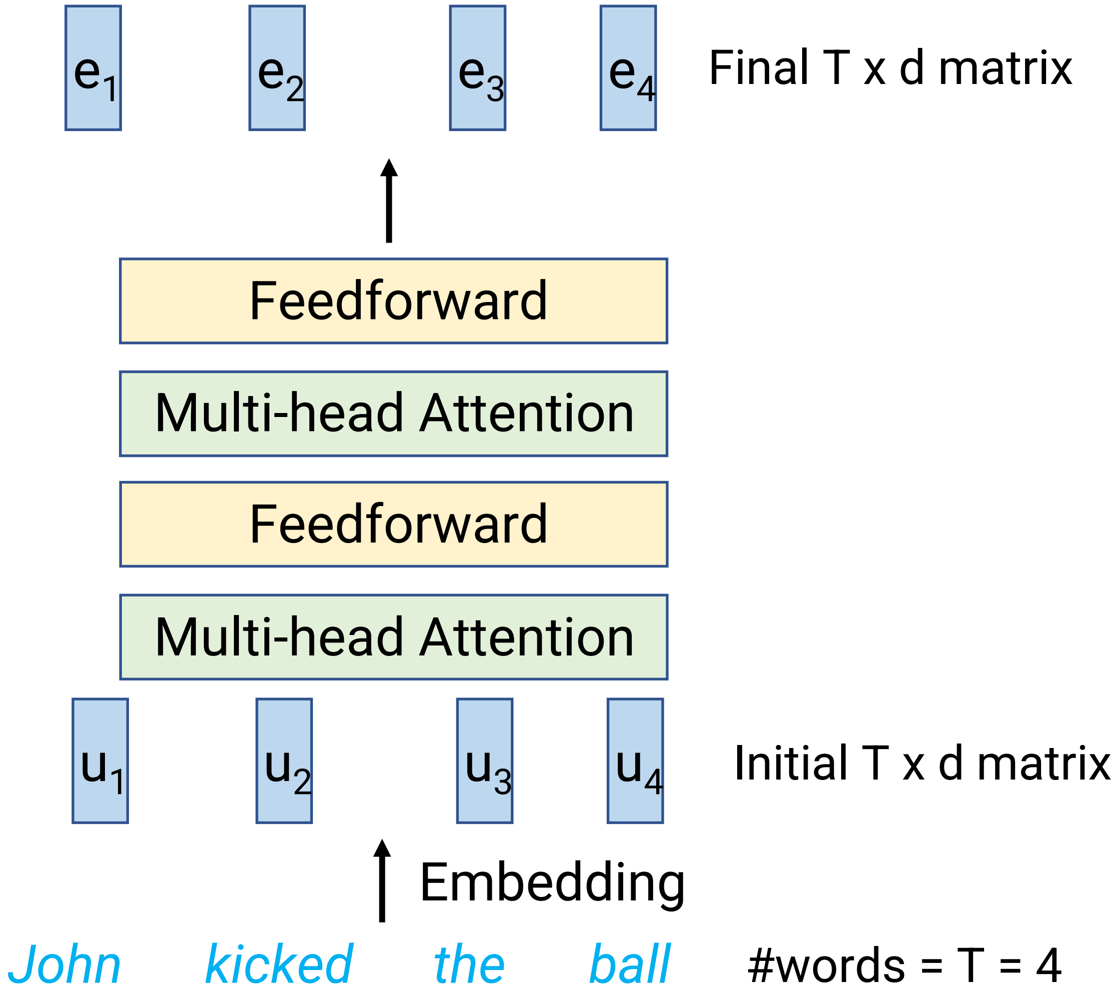
- One transformer consists of
  - **Initial embeddings** for each word of size d
    - Let T =#words, so initially we have a T x d matrix
  - Alternating layers of
    - "Multi-headed" attention layer
    - Feedforward layer
    - Both take in T x d matrix and output a new T x d matrix
  - Plus some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

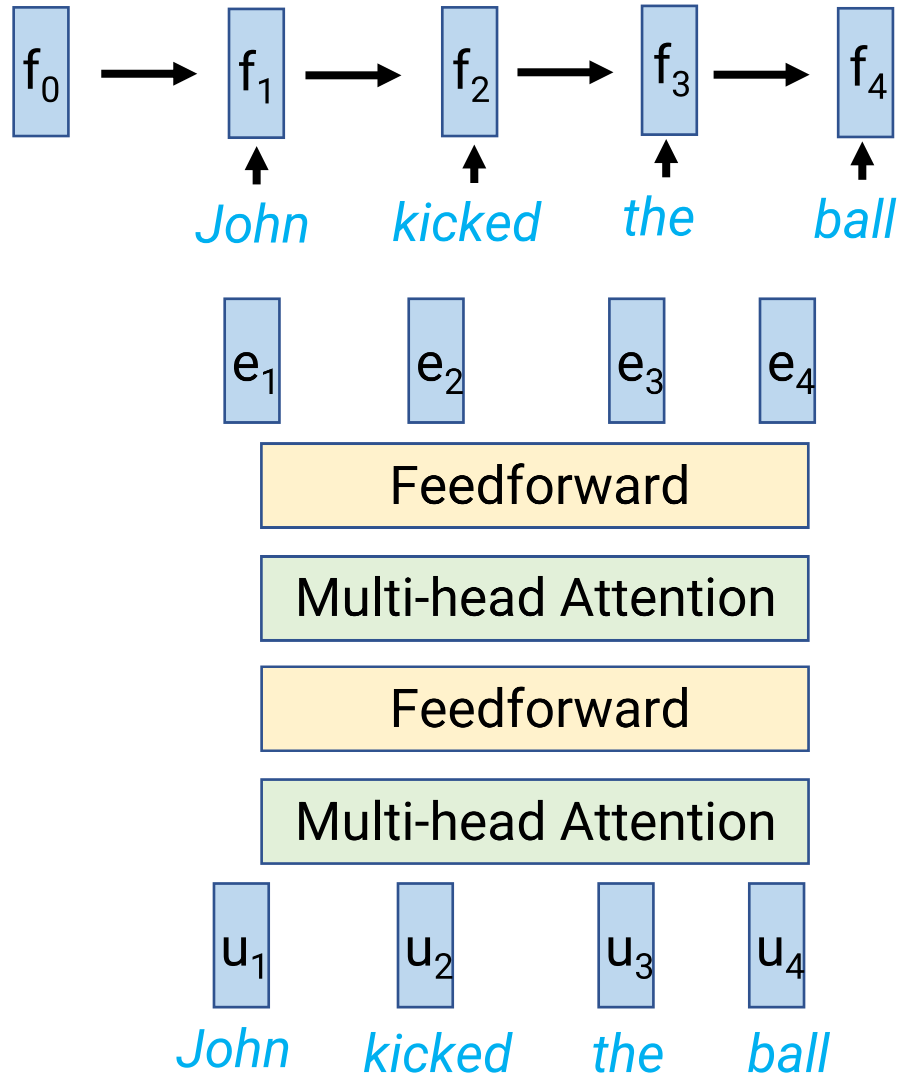


Transformer overview



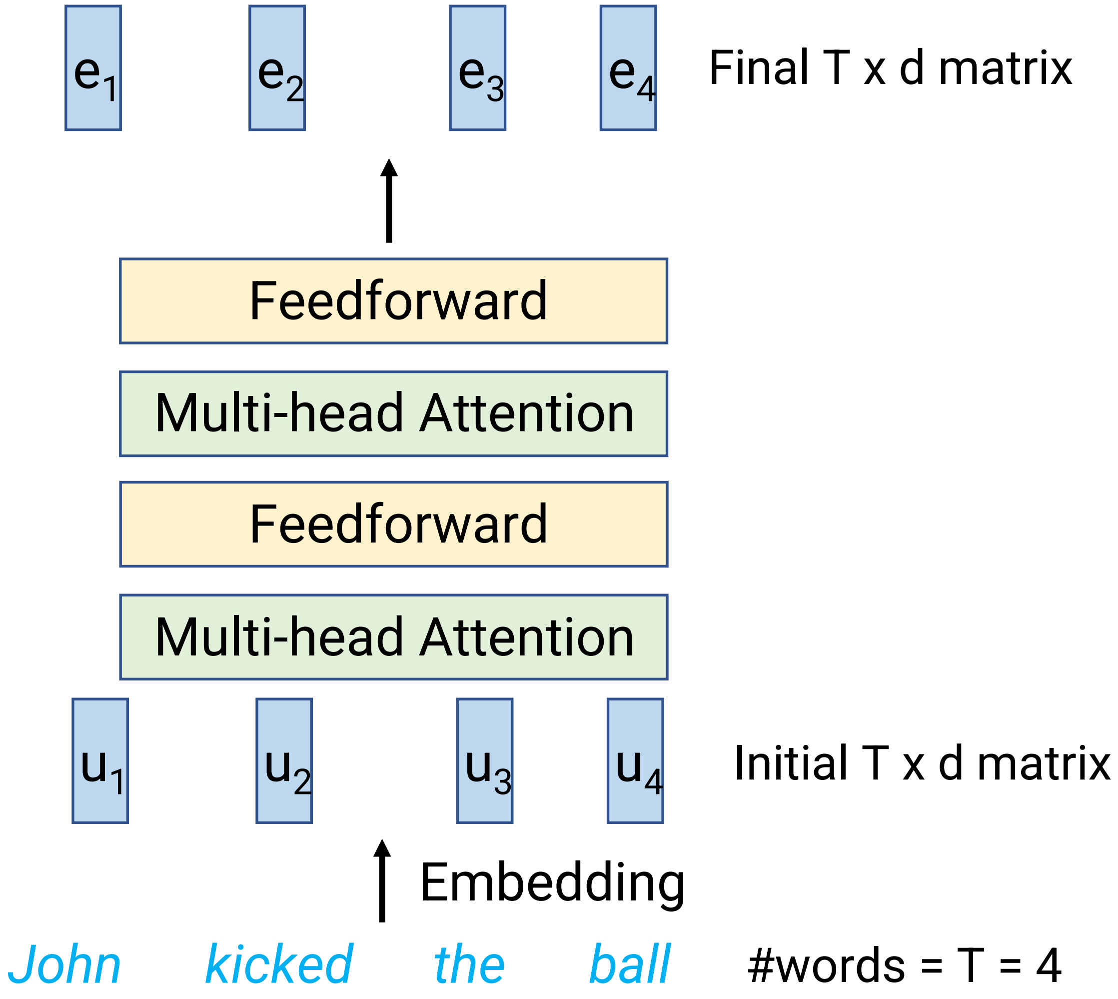
- How does a Transformer "work"?
- **Input layer**: Specify each word & its position in the sequence
- **Multi-headed attention layers**: For each word, retrieve information about related words, incorporate into the word's representation
- **Feedforward layers**: Do additional non-linear processing of 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 x 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 x d matrix
  - Alternating layers of
    - "Multi-headed" attention layer
    - Feedforward layer
    - Both take in T x d matrix and output a new T x 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