Transformers, Part I

Robin Jia
USC CSCI 467, Spring 2024
March 19, 2024

Announcements

- Midterm grades released
- Project midterm report due Tuesday, March 26
  - 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 4

Common Exam Mistakes: 1(c)

(c) Ryan reasons that the weight should be proportional to volume, which is in units of cubic centimeters. Therefore, he think that a good formula for FW should involve features where 3 of the original features are multiplied together.

ii. (4 points) Describe **two** different ways Ryan could train a model to learn the type of formula he is looking for. Explain your answer in detail.

- Answering "Neural Network" got 1 / 2 points
  - Yes, neural networks can *approximate* any function
  - But they will never *actually* compute the product of 3 features
  - Better answer is to directly multiply the features together

Common Exam Mistakes: 1(c)

(c) Ryan reasons that the weight should be proportional to volume, which is in units of cubic centimeters. Therefore, he think that a good formula for FW should involve features where 3 of the original features are multiplied together.

ii. (4 points) Describe **two** different ways Ryan could train a model to learn the type of formula he is looking for. Explain your answer in detail.

- Partially correct answer: Use a kernelized method with a Φ(x) function that has certain properties
  - But to run a kernel method, you have to **specify the kernel function *k(x, z)***
  - To use kernel trick, must show *k(x, z)* is efficient to compute
  - A kernelized method never directly uses Φ, that's why it has different efficiency properties

Common Exam Mistakes: 5(b)

(b) (5 points) Consider a linear model with parameter vector $w \in \mathbb{R}^d$ for binary classification. For a given training dataset, we can compute the **zero-one loss** as follows:

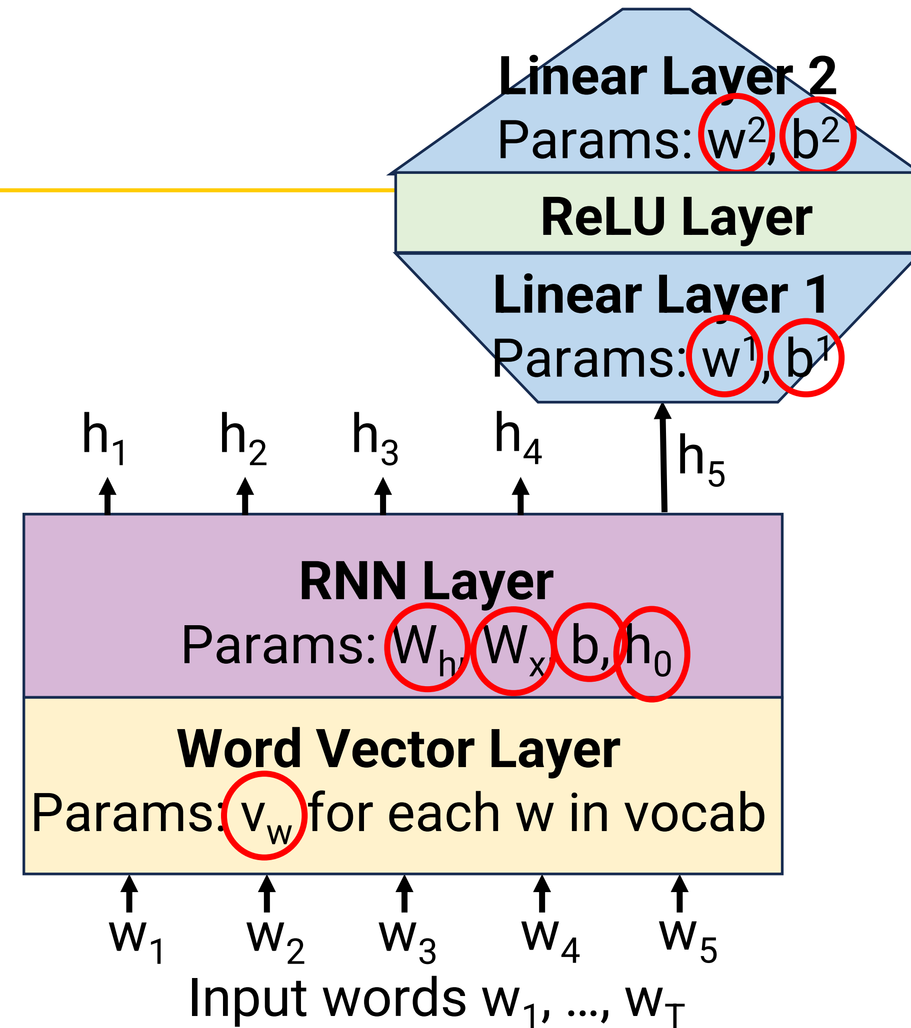
$$L(w) = \sum_{i=1}^{n} \mathbb{I}[y^{(i)} w^\top x^{(i)} \leq 0].$$

Recall that $\mathbb{I}[\cdot]$ is the indicator function that is 1 if the input is true, and 0 if it is false. Is it possible to use gradient descent to learn w by minimizing this loss function?

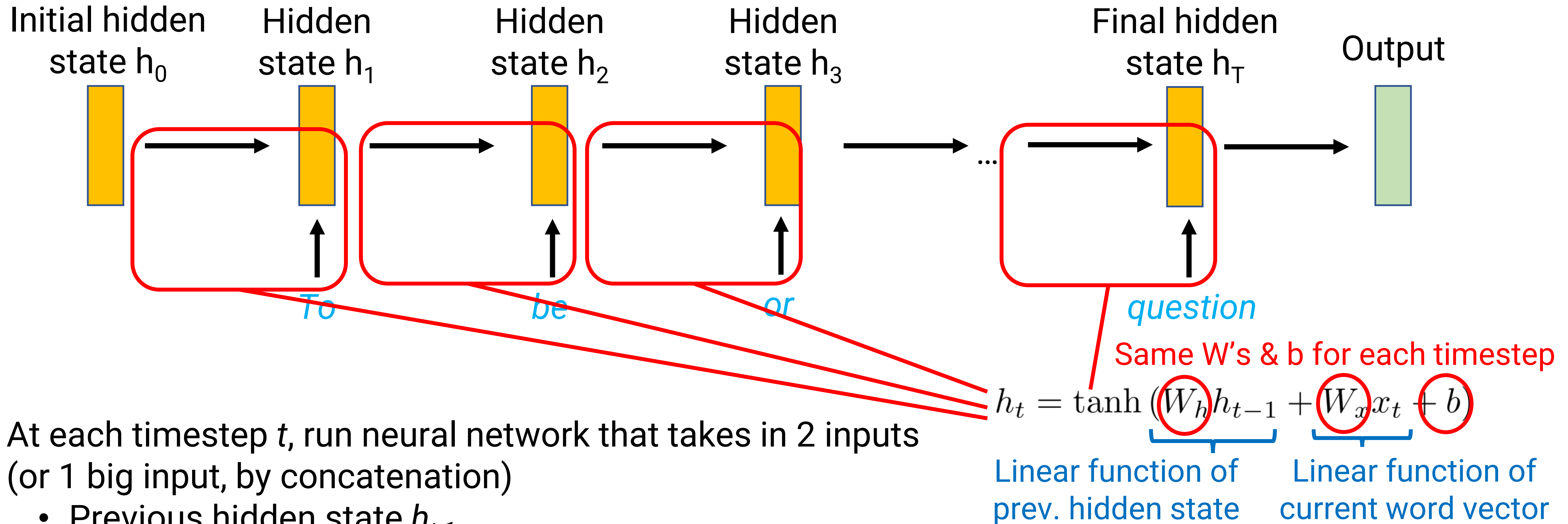
- Incomplete answer: Bad because it is not differentiable everywhere
  - Hinge loss is also not differentiable everywhere, but SVM works
  - Real problem: This function's derivative is 0 everywhere it exists
  - This means that all gradients are 0, so gradient descent does nothing

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



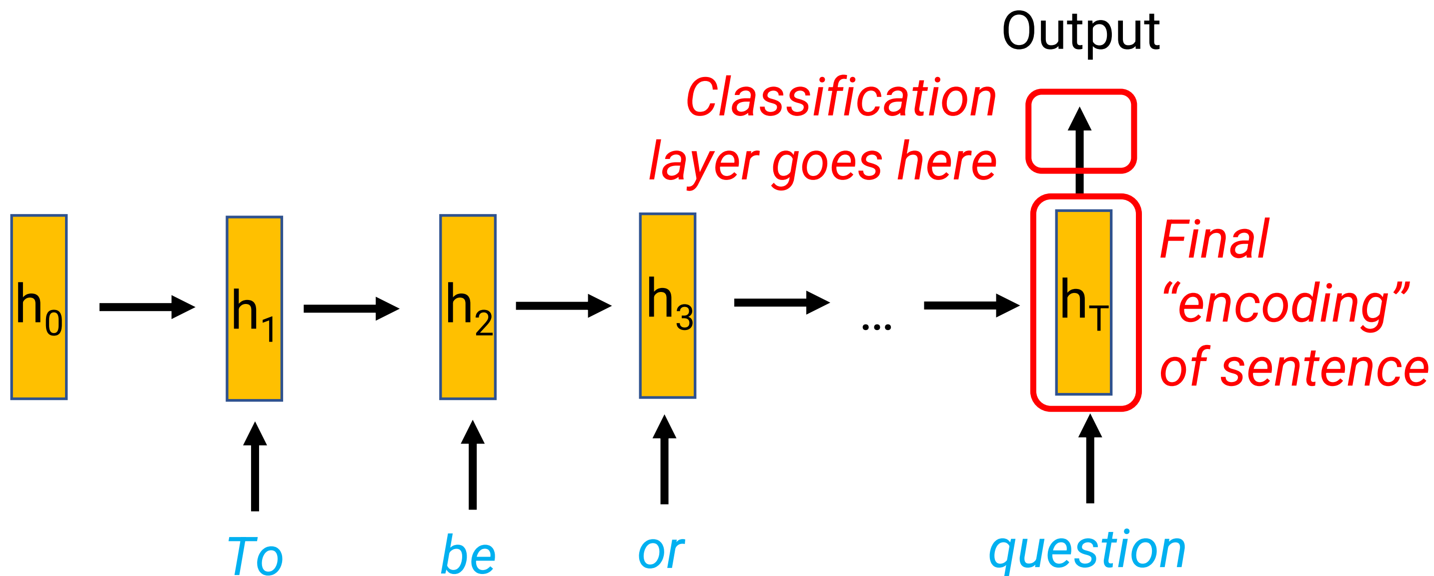
Review: RNNs



- 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

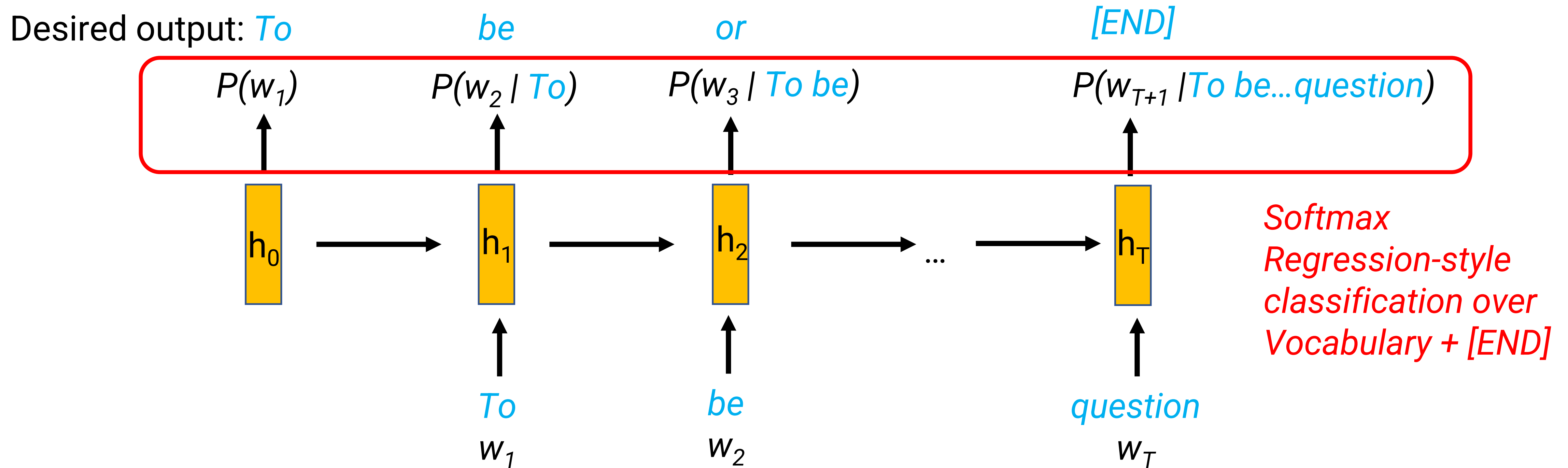
Review: Encoder vs. Decoder

Encoder model: Converts sentence to vector "encoding"



- First run an RNN over text
- Use the final hidden state as an "encoding" of the entire sequence
- Use this as features, train a classifier on top

Review: Encoder vs. Decoder



RNNs vs. Transformers (Encoders)

RNNs

- Process a sentence **one word at a time**
- Each "step" of computation is reading **one more word** (time dimension)
- Final encoding of sentence = **final word's hidden state**

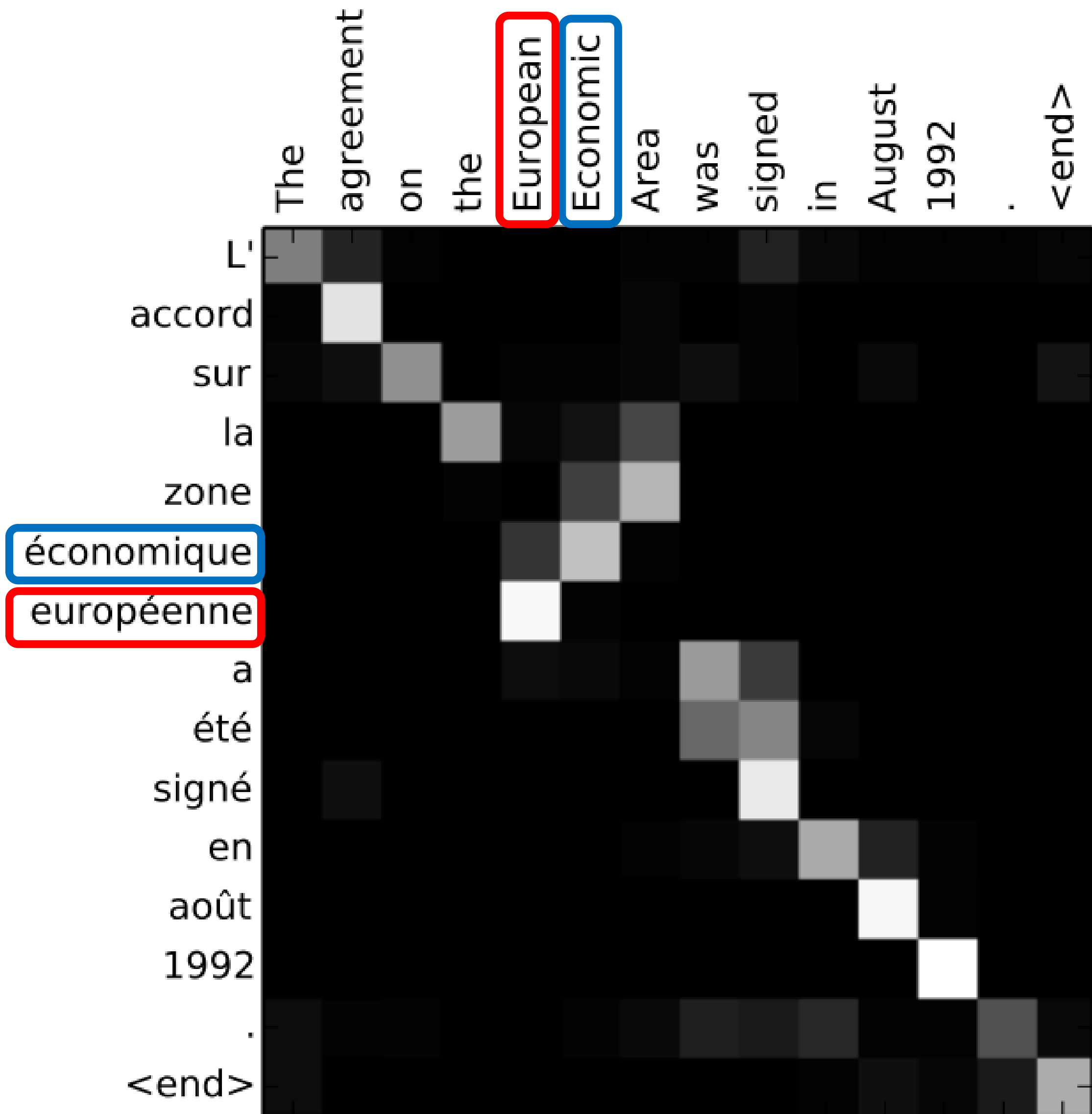
Both

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

Transformers

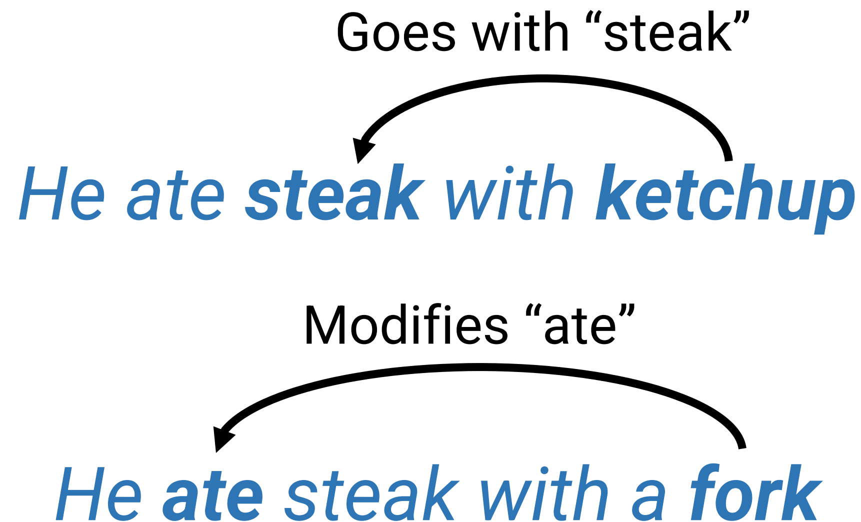
- Process **all words of the sentence at the same time** (in parallel)
- Each "step" of computation is applying **one more layer** (depth dimension; more like a CNN)
- Final encoding of sentence = **any word's hidden state** from the final layer

Review: Challenges of modeling sequences



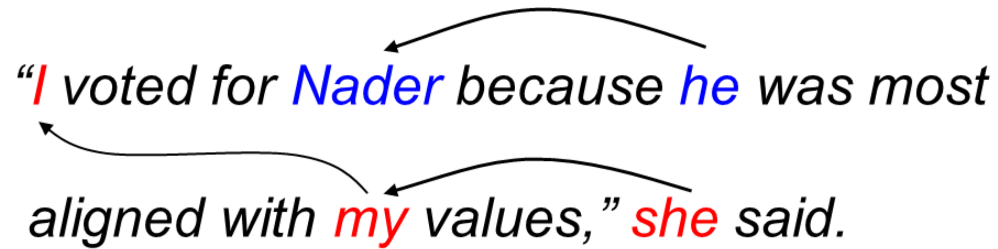
- Modeling relationships between words
  - Translation alignment

Review: Challenges of modeling sequences



- Modeling relationships between words
  - Translation alignment
  - Syntactic dependencies

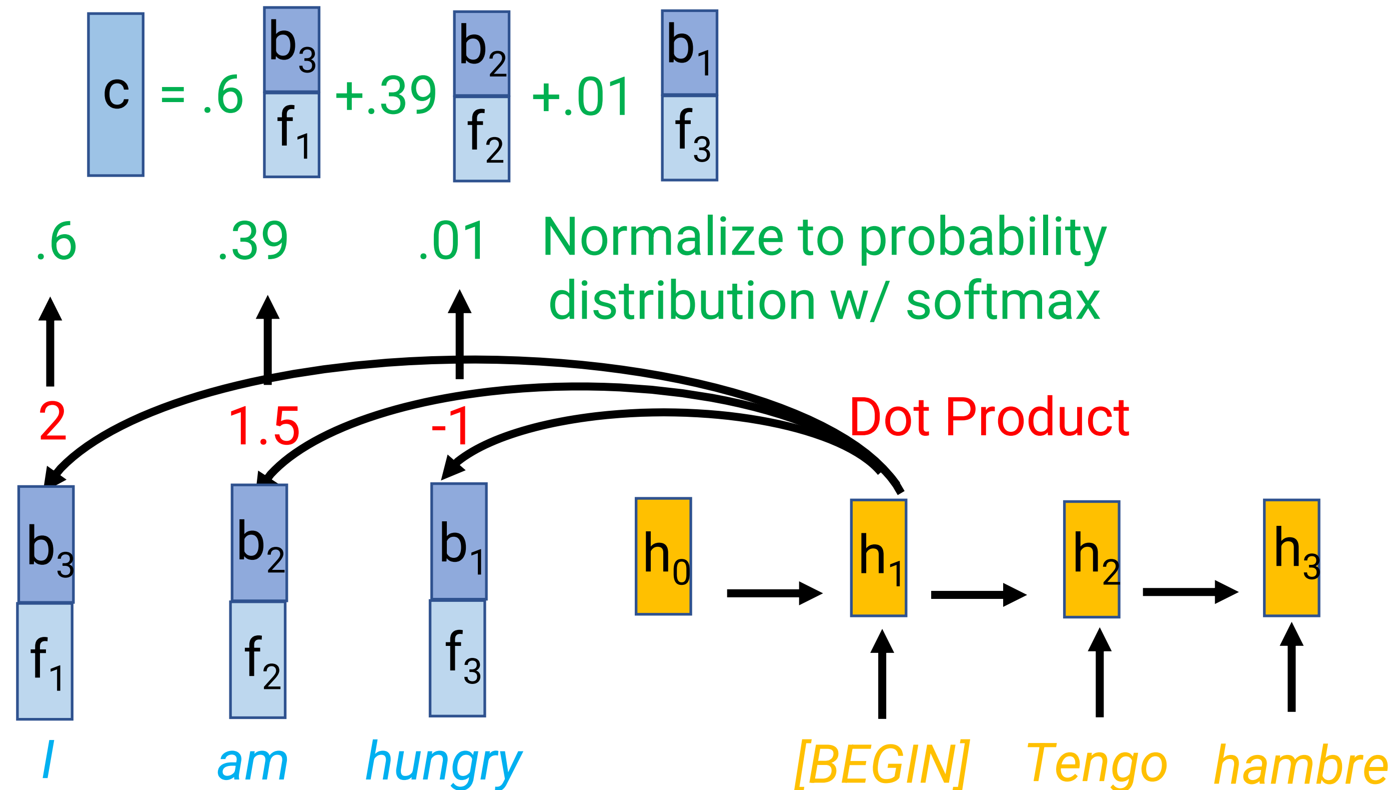
Review: Challenges of modeling sequences



- 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 | 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 $k_1, \ldots, k_T$
  - 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:

$$s_t = q^\top k_t$$

How well does the query match each key?

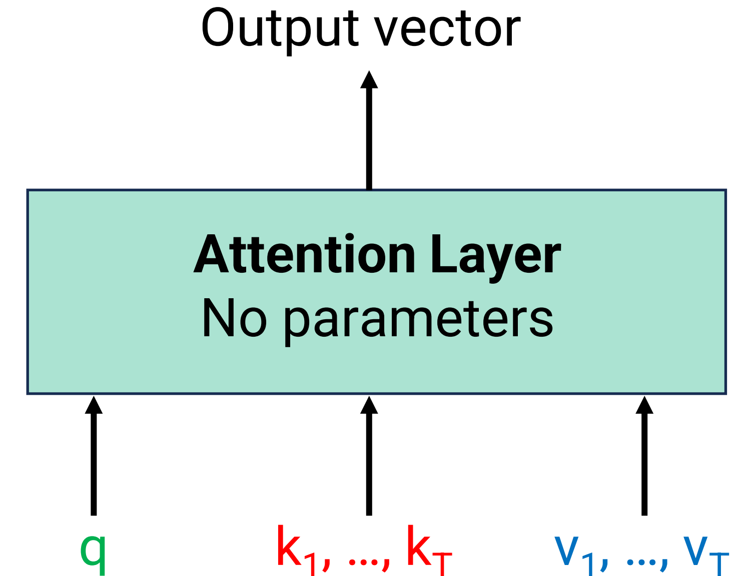
  - Softmax to get probability distribution $p_1, \ldots, p_T$:

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

  - Return weighted average of value vectors:

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

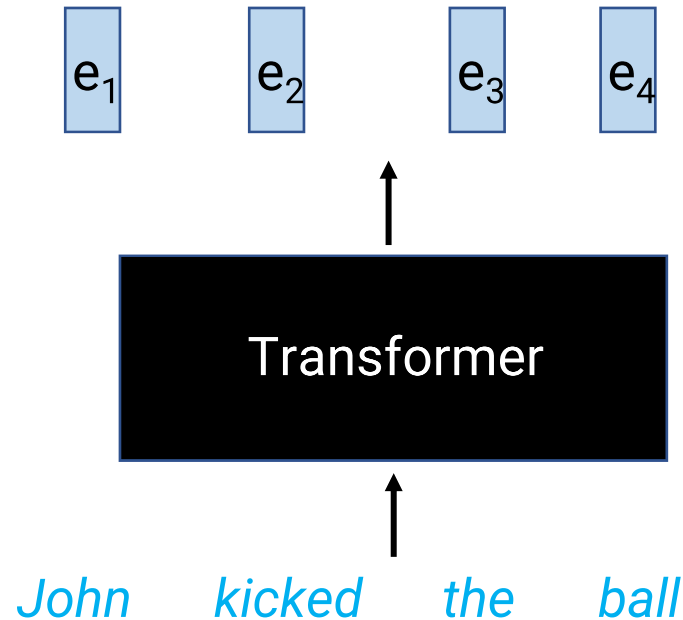
Dominated by the values corresponding to the "best-matching" keys



Today: Can we use Attention for Everything?

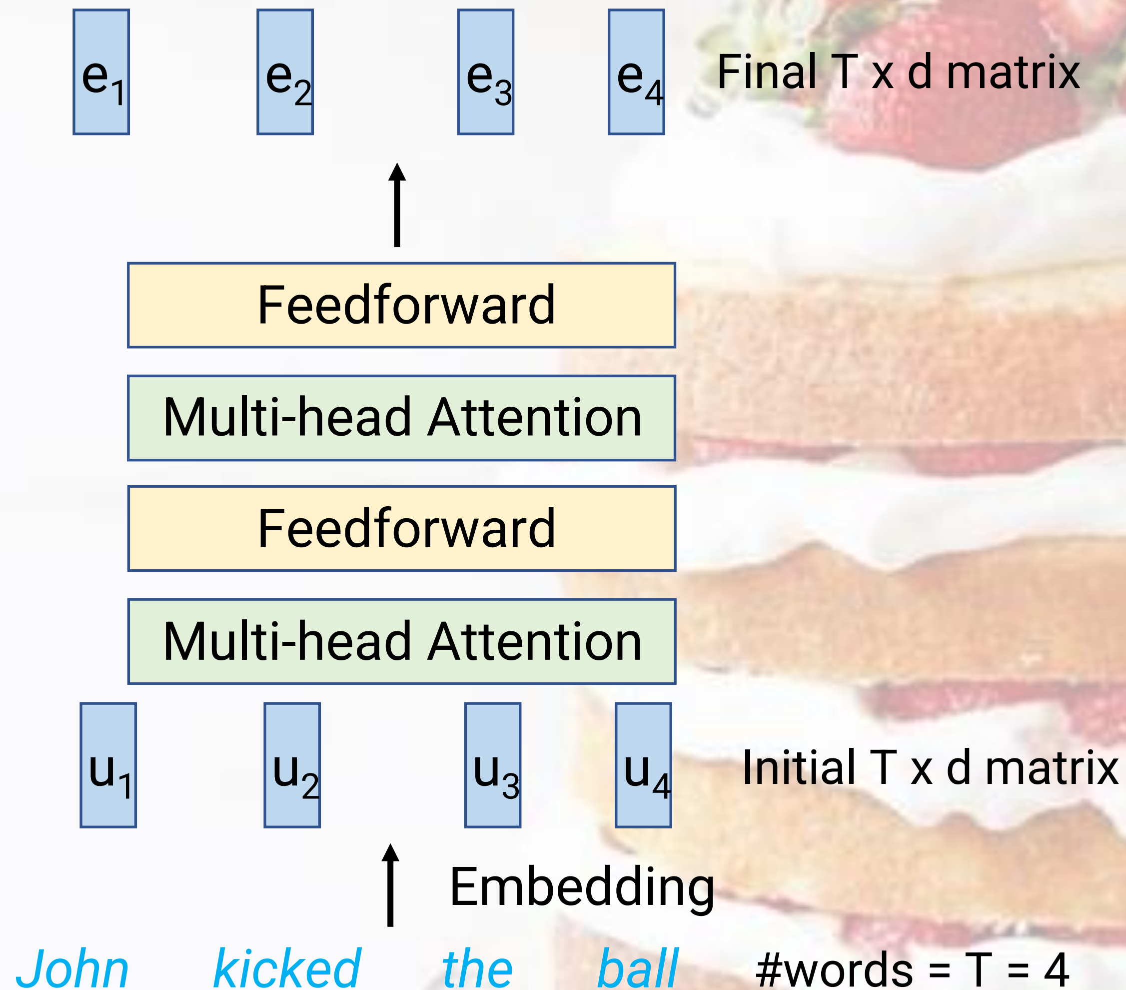
- Modeling relationships between words
  - Translation alignment
  - Syntactic dependencies
  - Coreference relationships
- Long range dependencies
  - E.g., consistency of characters in a novel
- Attention captures relationships & doesn't care about "distance," unlike RNNs
- **Let's replace RNN's with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

Today: The Transformer Architecture



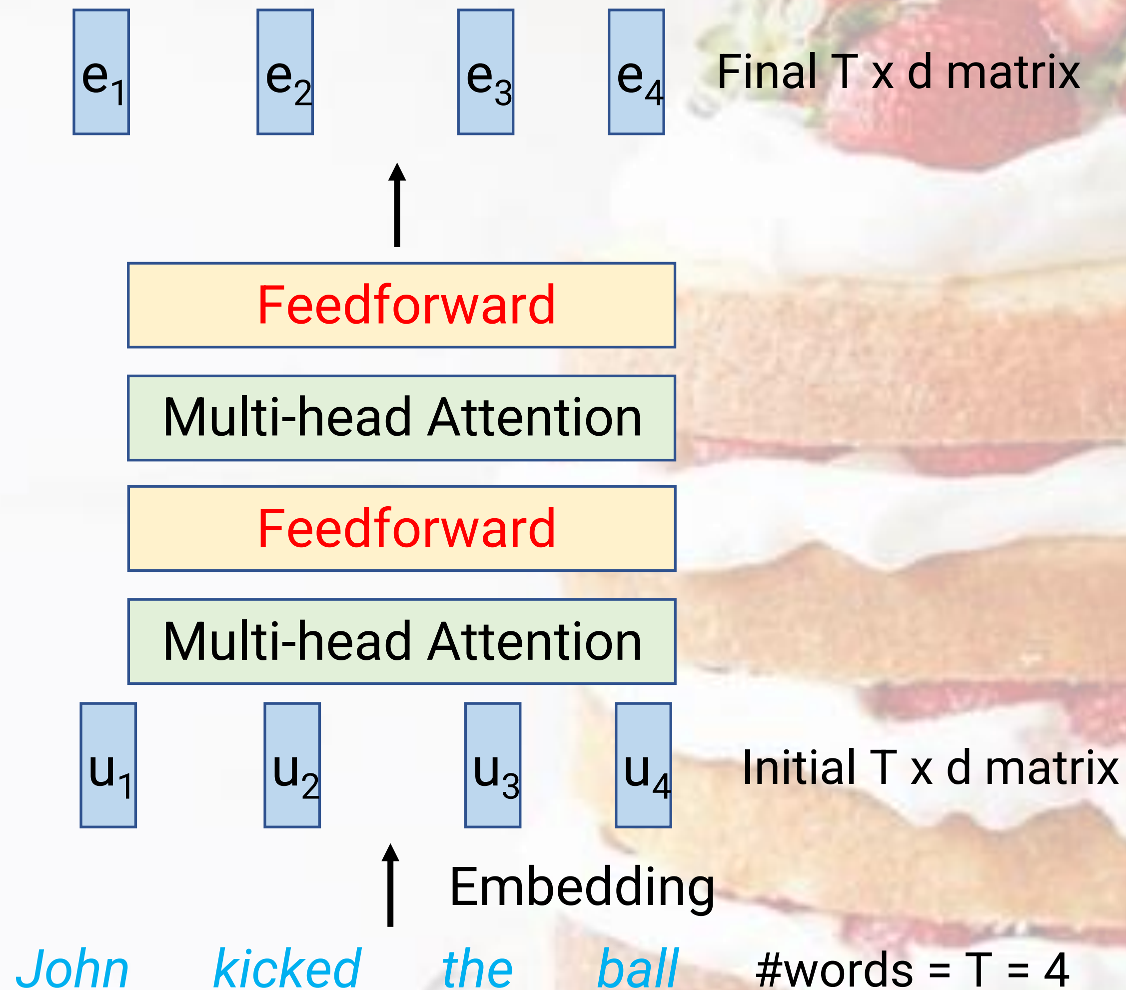
- Input: Sequence of words
- Output: Sequence of hidden state vectors, one per word
- Same “type signature” as RNN
- Motivation
  - Process all words at the same time, don’t do explicit sequential processing
  - Let **attention** figure out which words are relevant to each other
    - Whereas RNN assumes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

Transformer overview



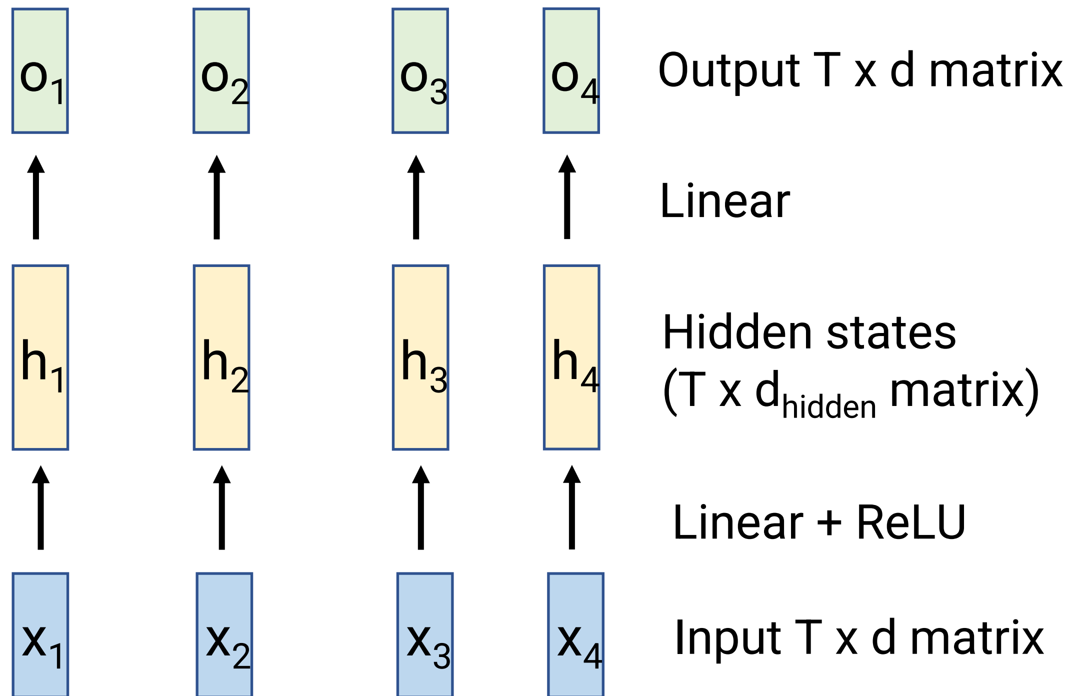
- One transformer consists of
  - Initial embeddings for each word of size d
    - Let T =#words, so initially we have a T x d matrix
  - Alternating layers of
    - "Multi-headed" attention layer
    - Feedforward layer
    - Both take in T x d matrix and output a new T x d matrix
  - Plus some bells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

Transformer overview



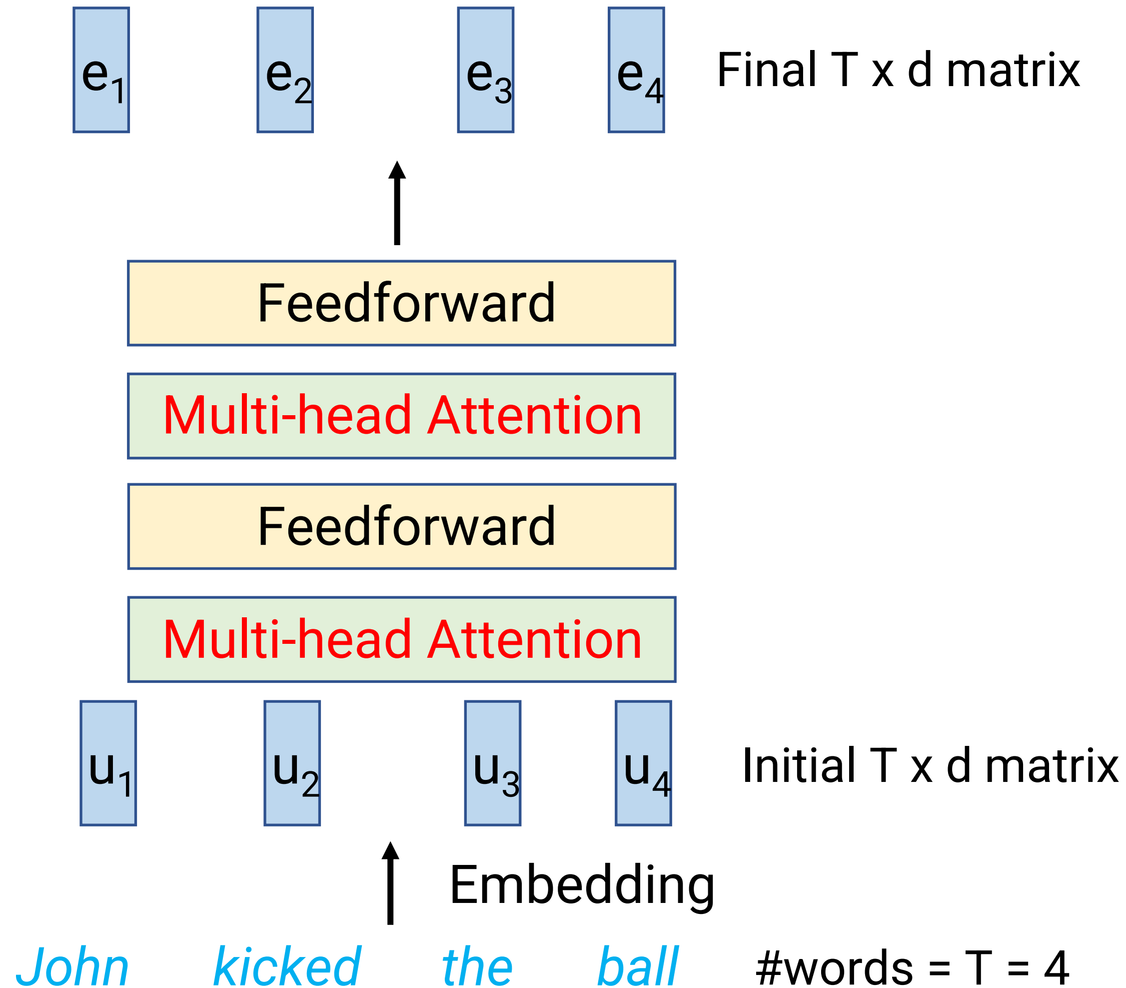
- One transformer consists of
  - Initial embeddings for each word of size d
    - Let T =#words, so initially we have a T x d matrix
  - Alternating layers of
    - "Multi-headed" attention layer
    - **Feedforward layer**
    - Both take in T x d matrix and output a new T x d matrix
  - Plus some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

Feedforward layer



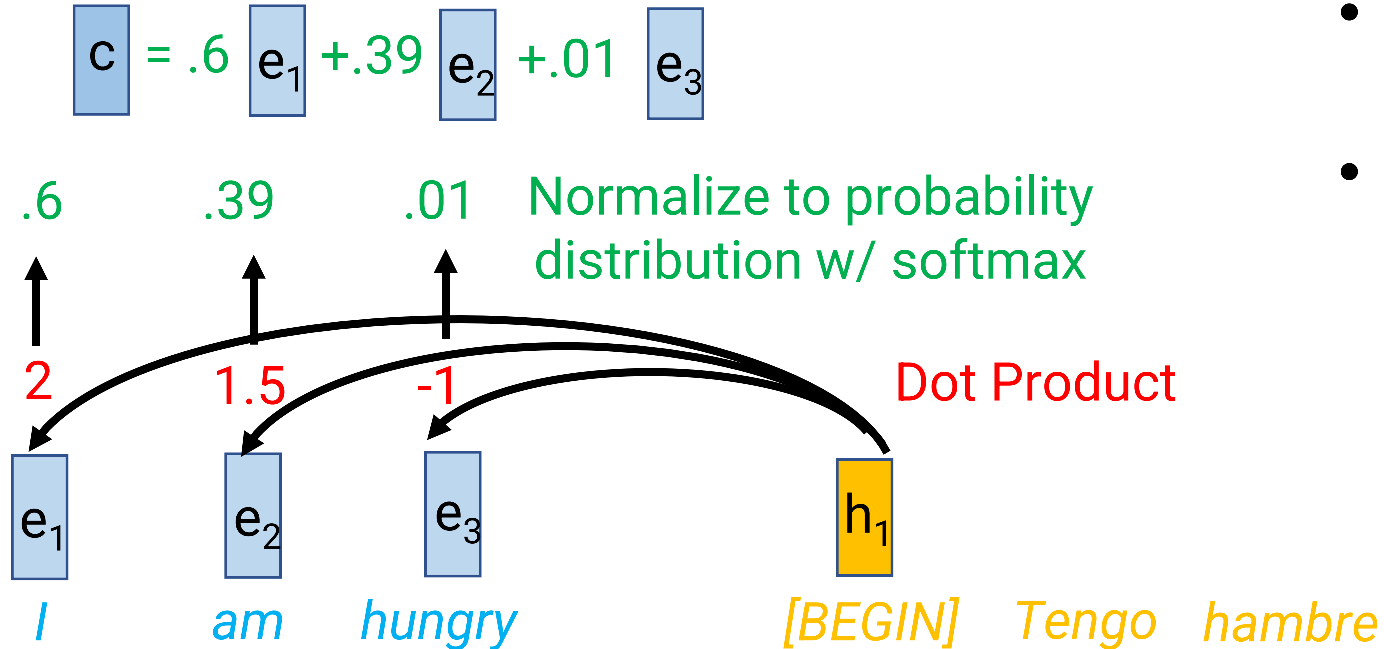
- Input: T x d matrix
- Output: Another T x d matrix
- Apply the same MLP separately to each d-dimensional vector
  - Linear layer from d to d_{hidden}
  - ReLU (or other nonlinearity)
  - Linear layer from d_{hidden} to d
- Note: No information moves between tokens here

Transformer overview



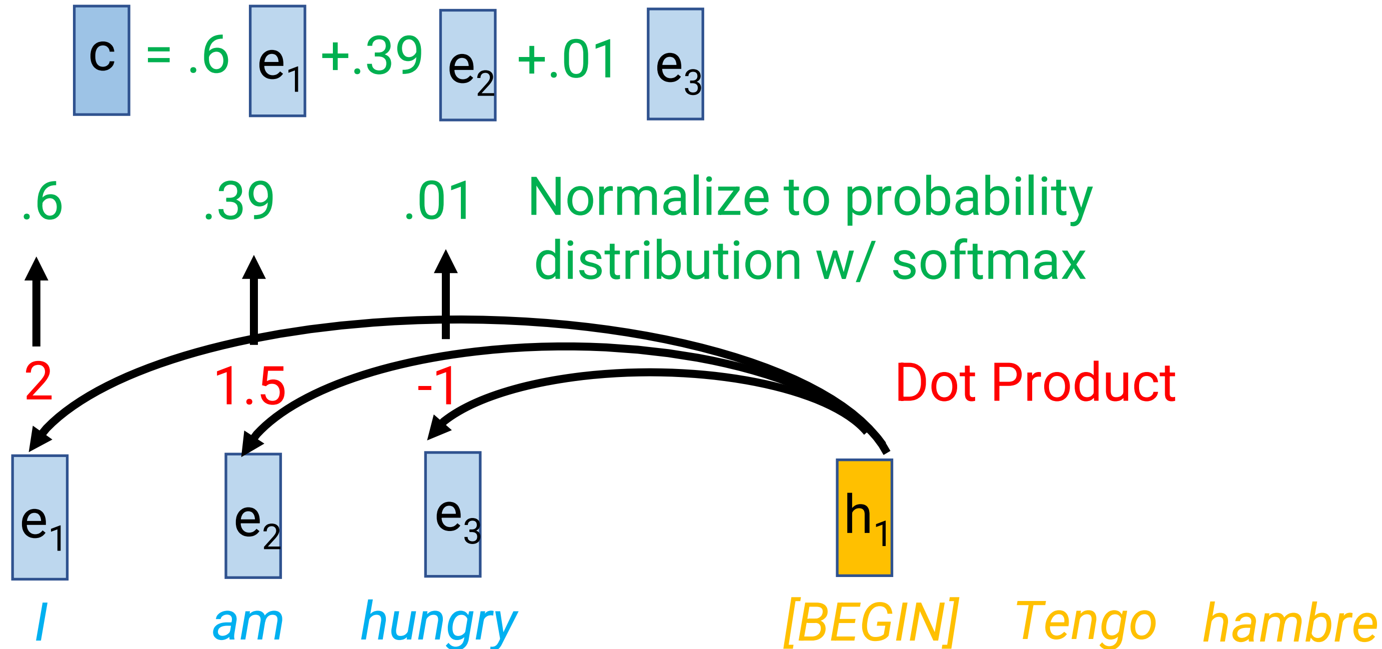
- One transformer consists of
  - Initial embeddings for each word of size d
    - Let T =#words, so initially we have a T x d matrix
  - Alternating layers of
    - **"Multi-headed" attention layer**
    - Feedforward layer
    - Both take in T x d matrix and output a new T x d matrix
  - Plus some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

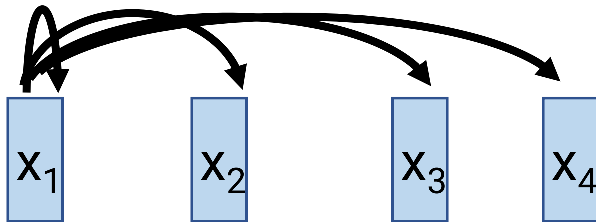


- Previously: Decoder state looks for relevant encoder states
- Self-attention: Each **encoder** state now looks for relevant (other) encoder states
- Why? Build better representation for word in context by capturing relationships to other words

Change #1: Self-attention

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

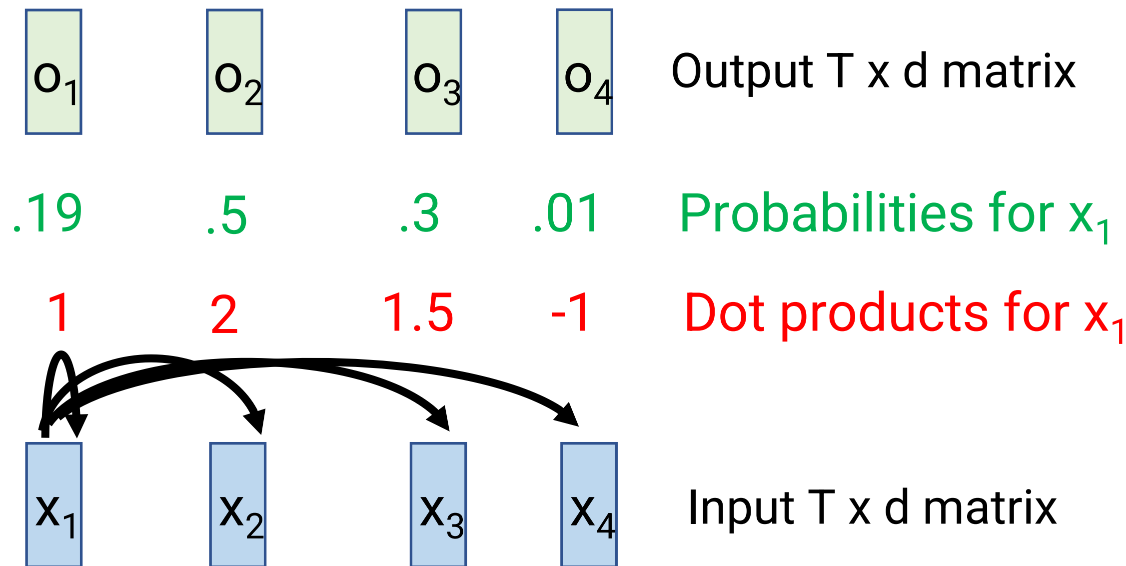
.19	.5	.3	.01	Probabilities for x_1
1	2	1.5	-1	Dot products for x_1



Input T x d matrix

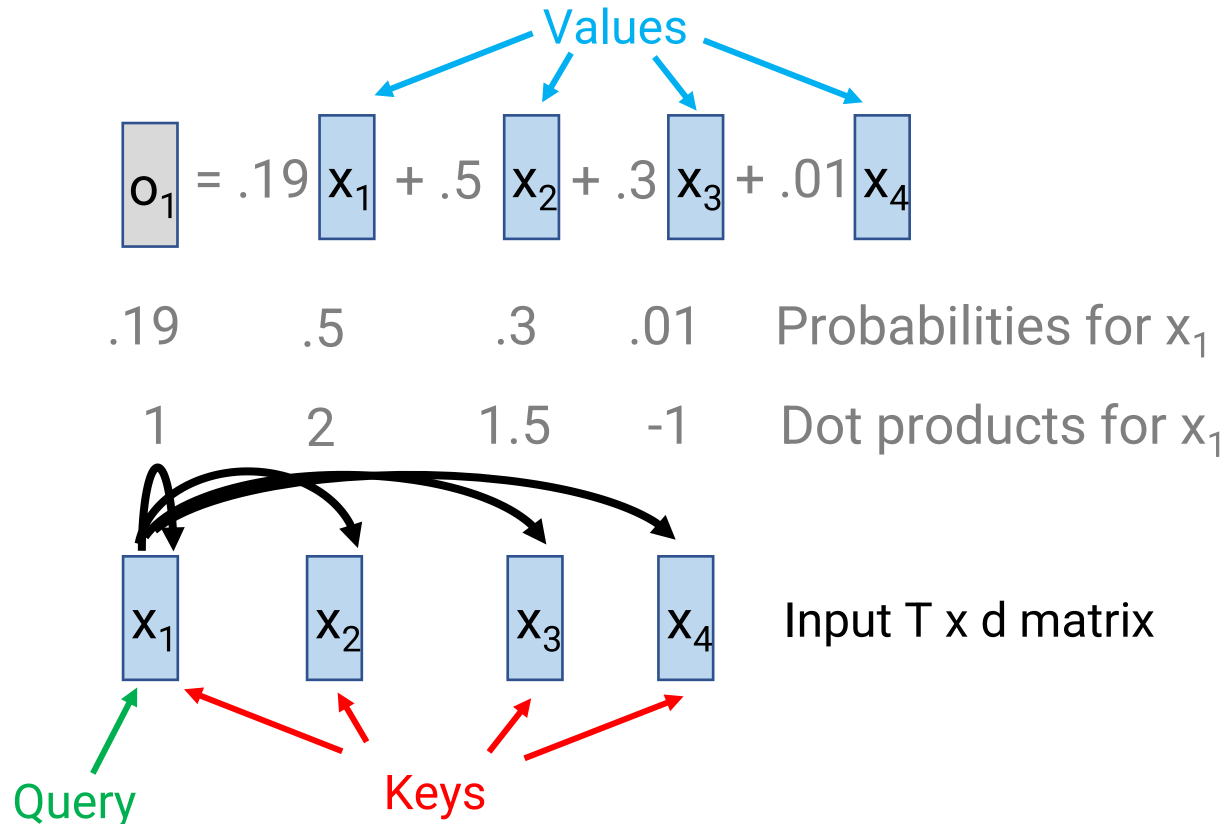
- Take x_1 and dot product it with all T inputs (including itself)
- Apply softmax to convert to probability distribution
- Compute output o_1 as weighted sum of inputs

Change #1: Self-attention



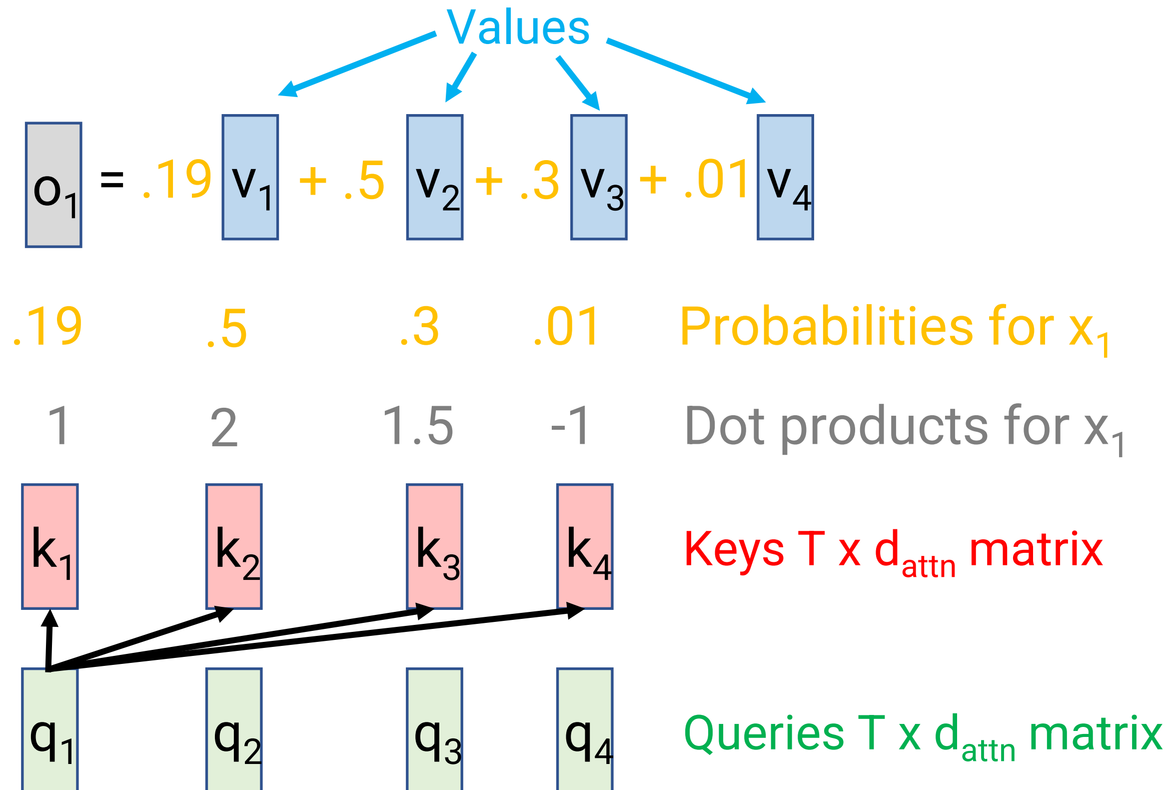
- Take x_1 and dot product it with all T inputs (including itself)
- Apply softmax to convert to probability distribution
- Compute output o_1 as weighted sum of inputs
- Repeat for t=2, 3, …, T
- Replacement for recurrence
  - RNN only allows information to flow linearly along sequence
  - Now, information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

Change #2: Separate queries, keys, and values



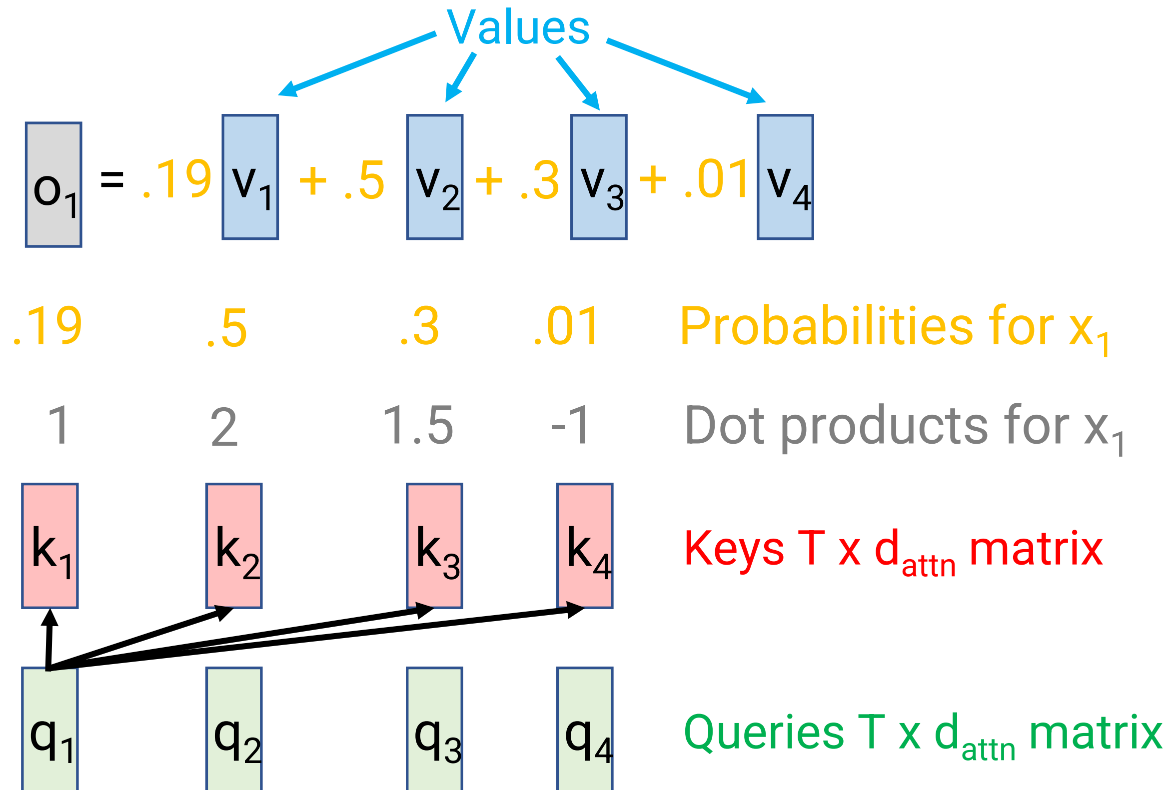
- Recall: Attention uses vectors in three different ways
  - As "query" for current index
  - As "keys" to match with query
  - As "values" when computing output
- Idea: Use separate vectors for each usage
  - What each index "looks for" different from what it "matches with"
  - What you store in output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

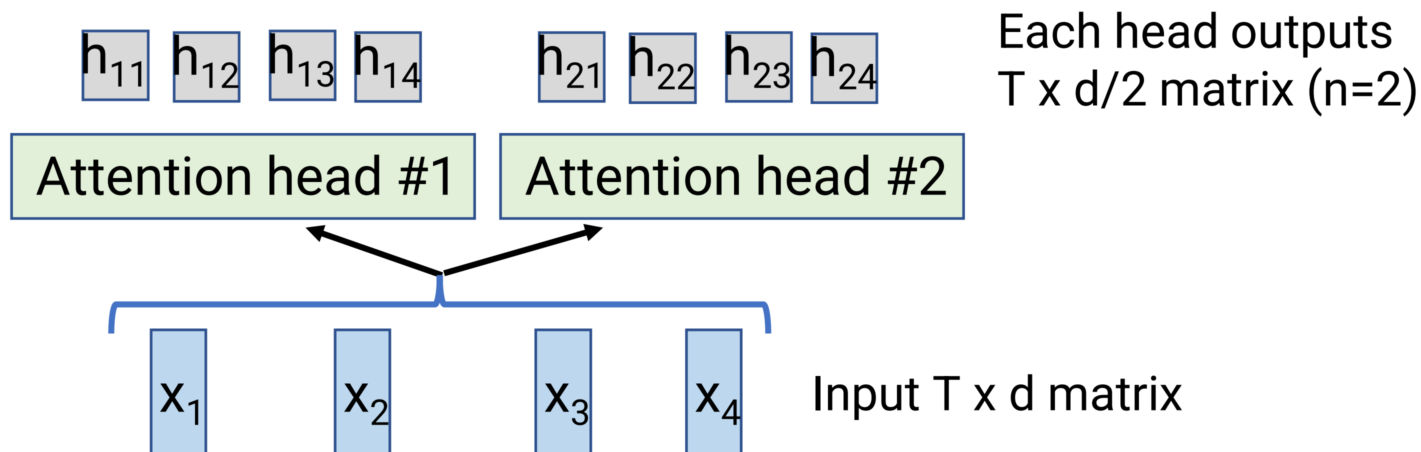
Matrix form



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

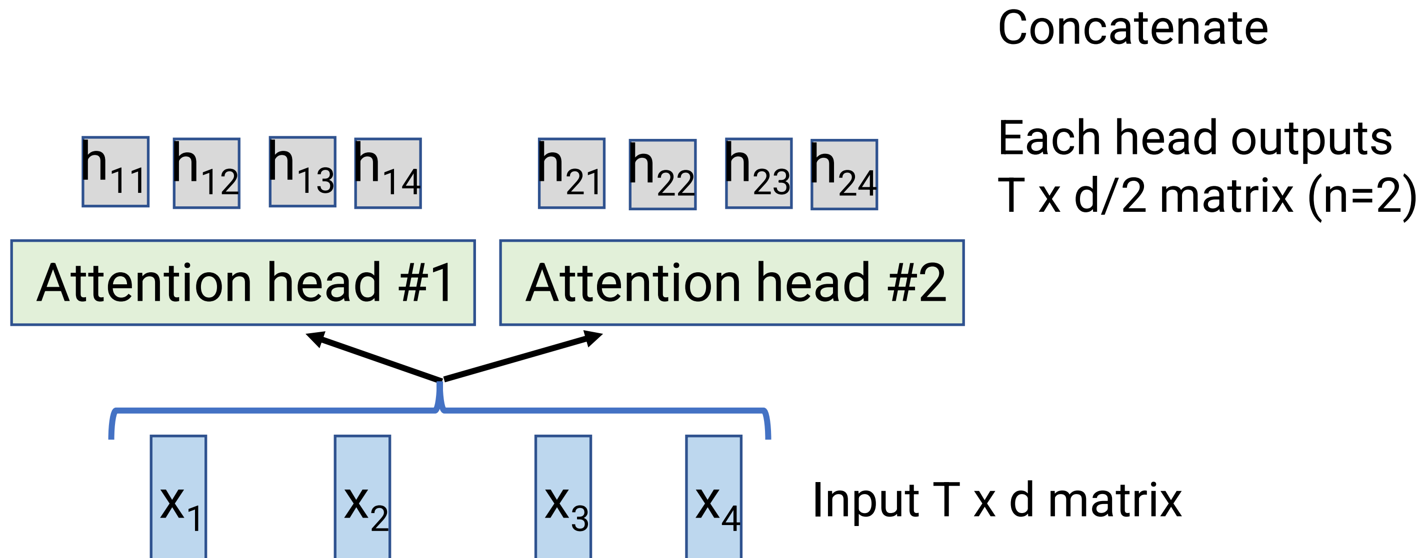
- Apply 3 separate linear layers to input matrix X (T x d_{in}) to get
  - Query matrix Q = $(W^Q * X^T)^T$
  - Keys K = $(W^K * X^T)^T$
  - Values V = $(W^V * X^T)^T$
  - Note: This adds parameters W^Q, W^K, W^V
- Compute Q x K^T (T x T matrix)
  - Each entry is dot product of one query vector with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

Change #3: Making it Multi-headed



- Instead of doing attention once, have n different "heads"
  - Each has its own parameters maps to dimension d_{attn} = d/n
  - Concatenate at end to get output of size T x d

Change #3: Making it Multi-headed

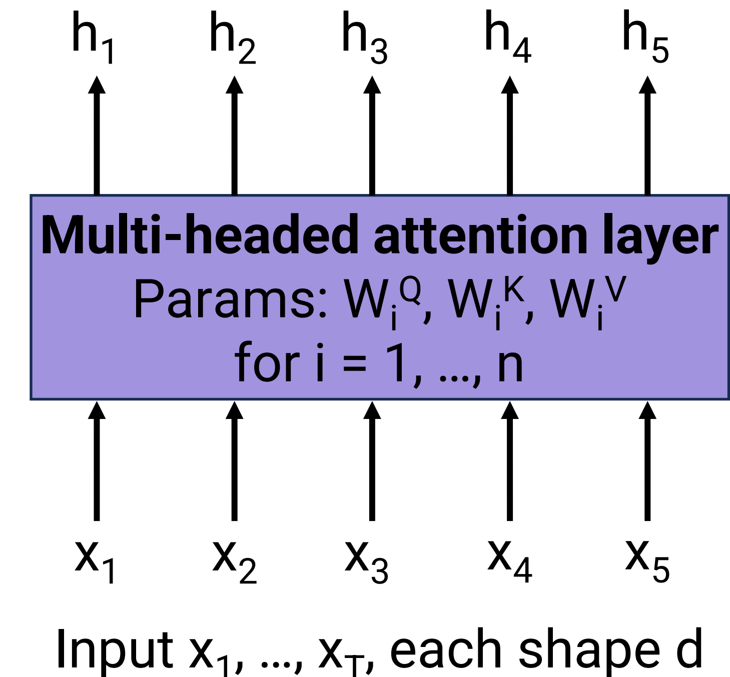


- Instead of doing attention once, have n different "heads"
  - Each has its own parameters maps to dimension d_{attn} = d/n
  - Concatenate at end to get output of size T x d
- Why? Different heads can capture different relationships between words

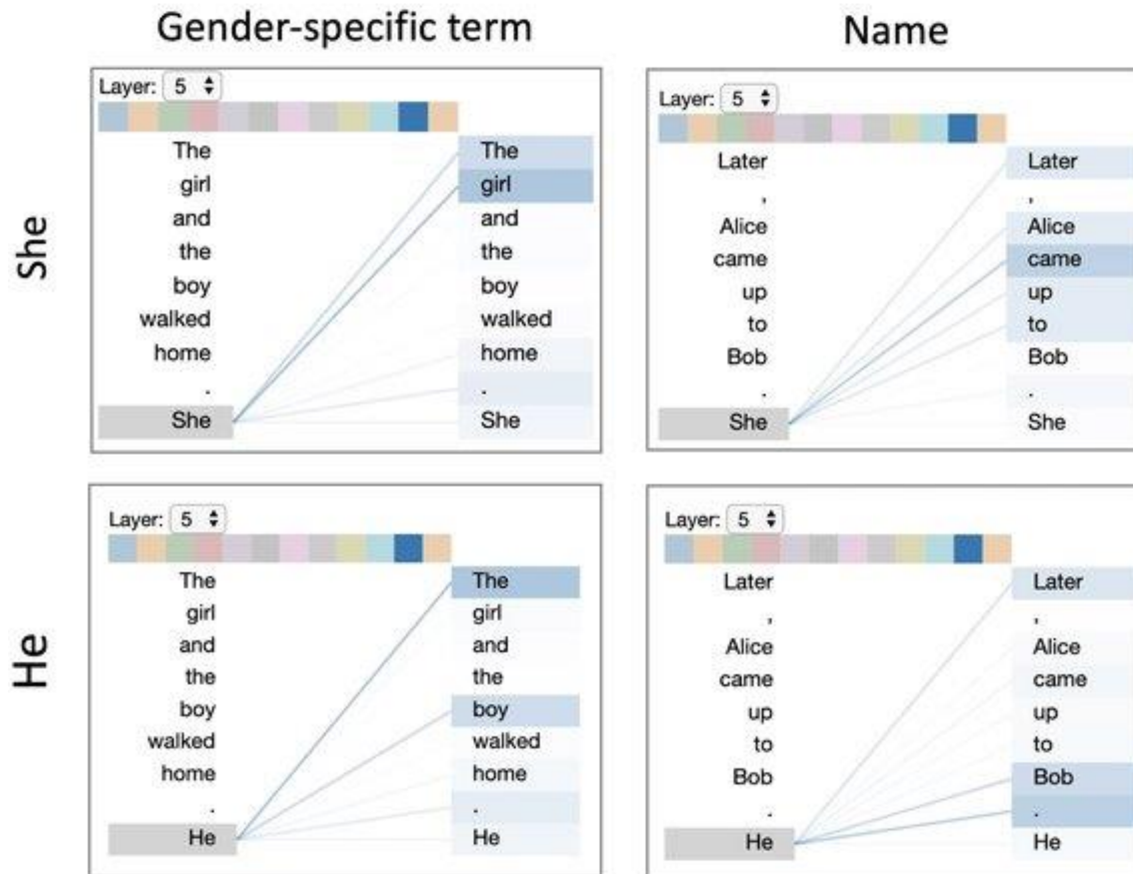
The Multi-headed Attention building block

(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()

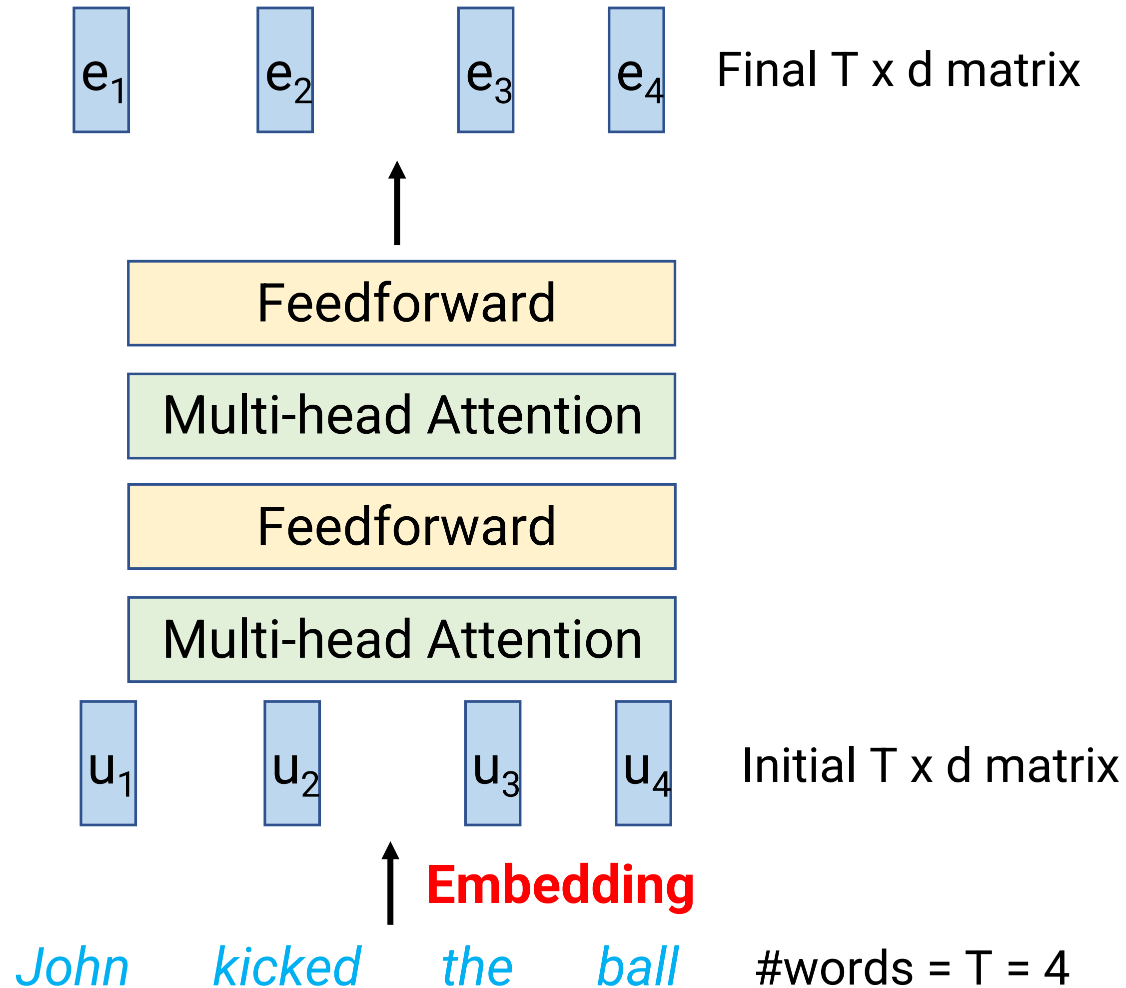


What do attention heads learn?



- 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

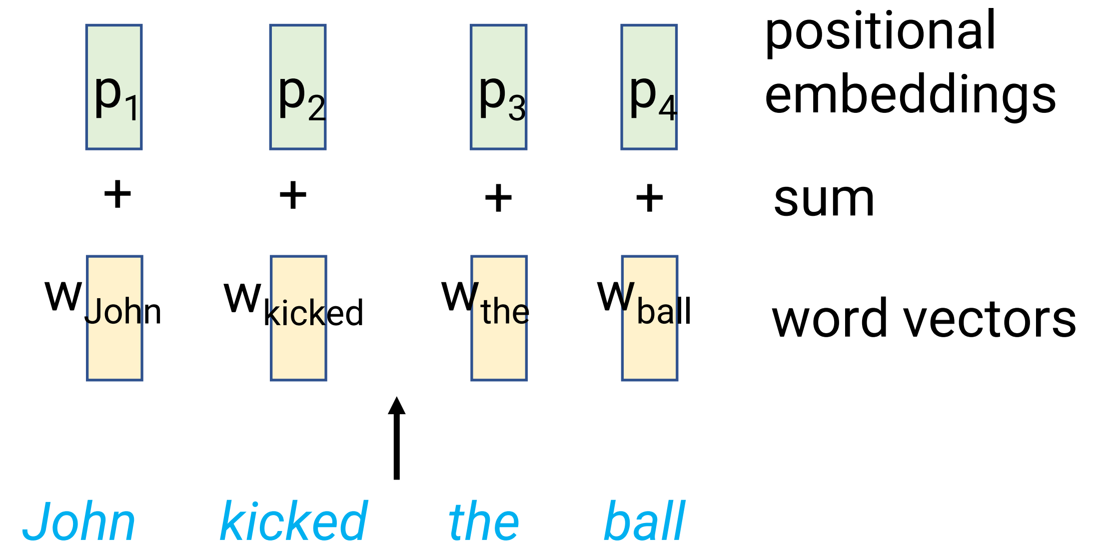
Transformer overview



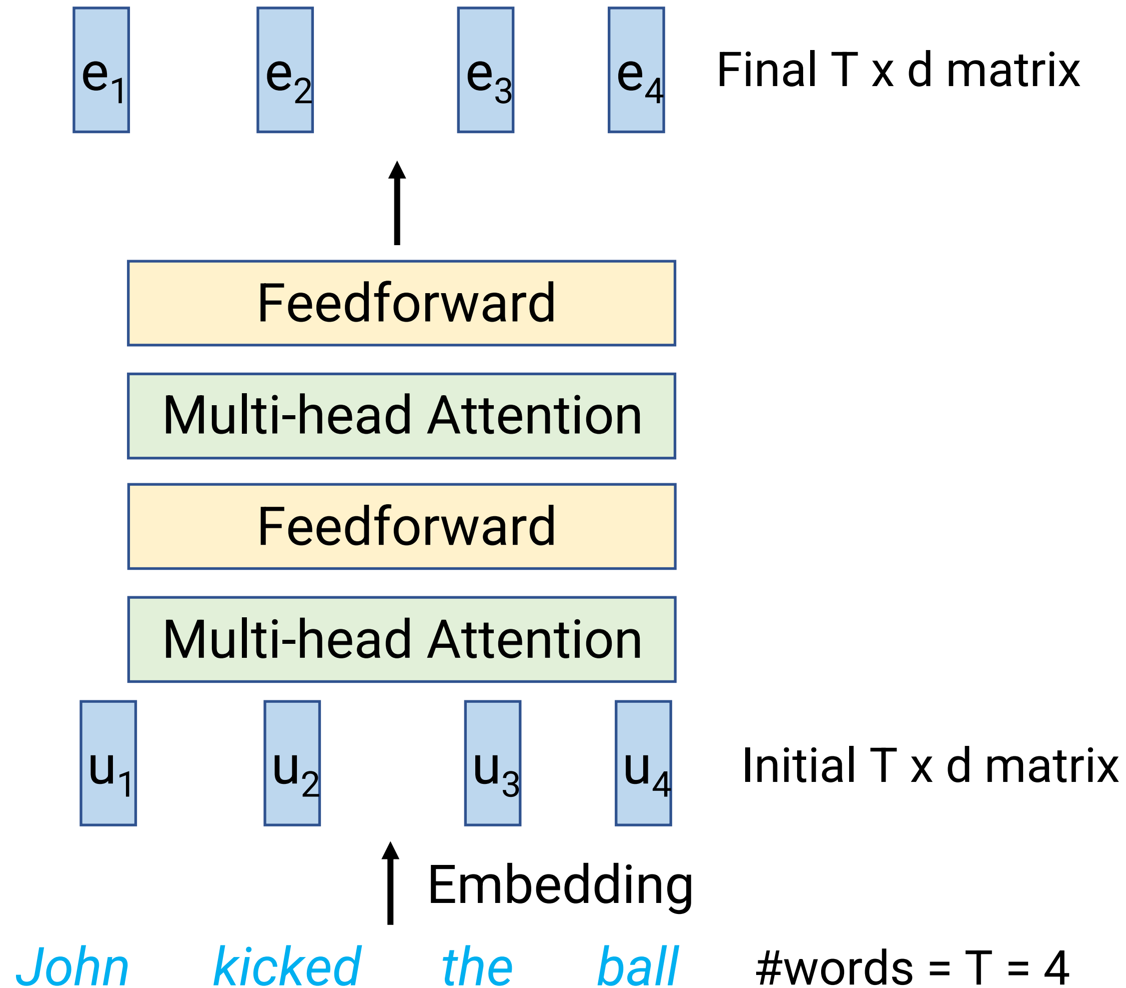
- One transformer consists of
  - **Initial embeddings** for each word of size d
    - Let T =#words, so initially we have a T x d matrix
  - Alternating layers of
    - "Multi-headed" attention layer
    - Feedforward layer
    - Both take in T x d matrix and output a new T x d matrix
  - Plus some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

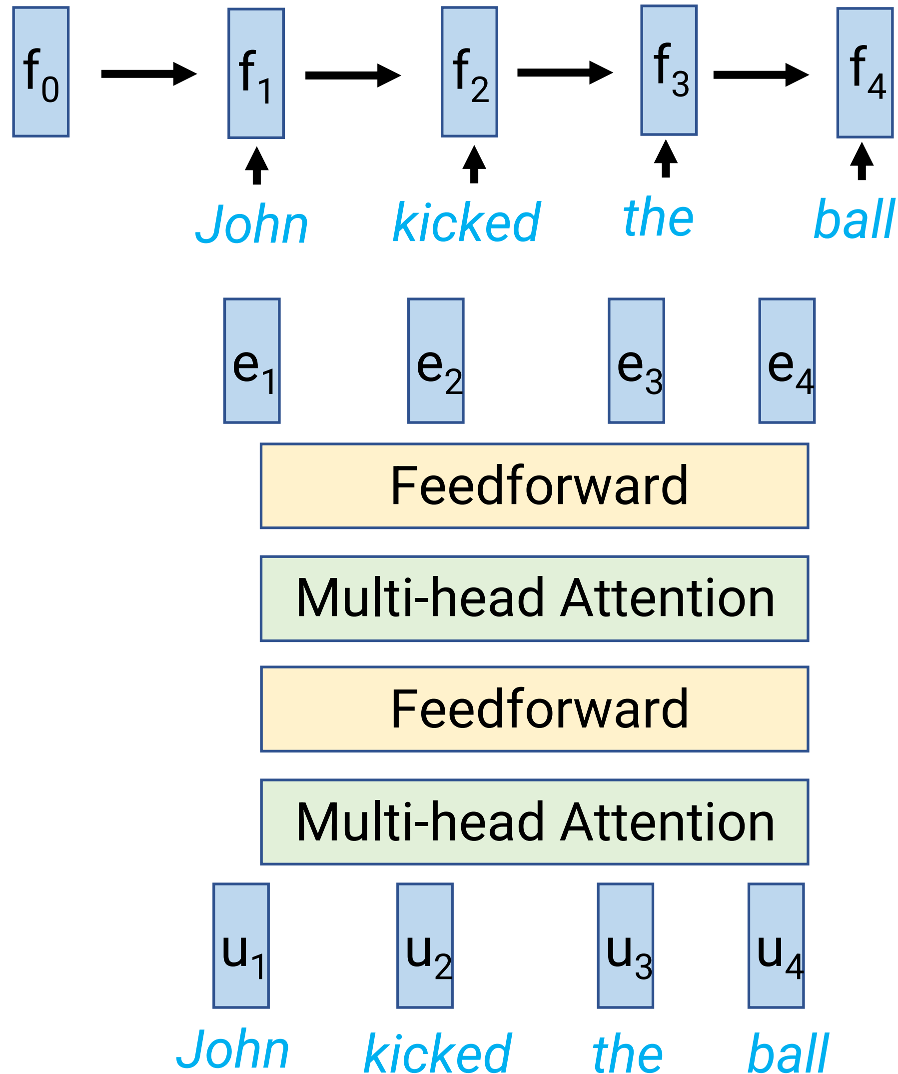


Transformer overview



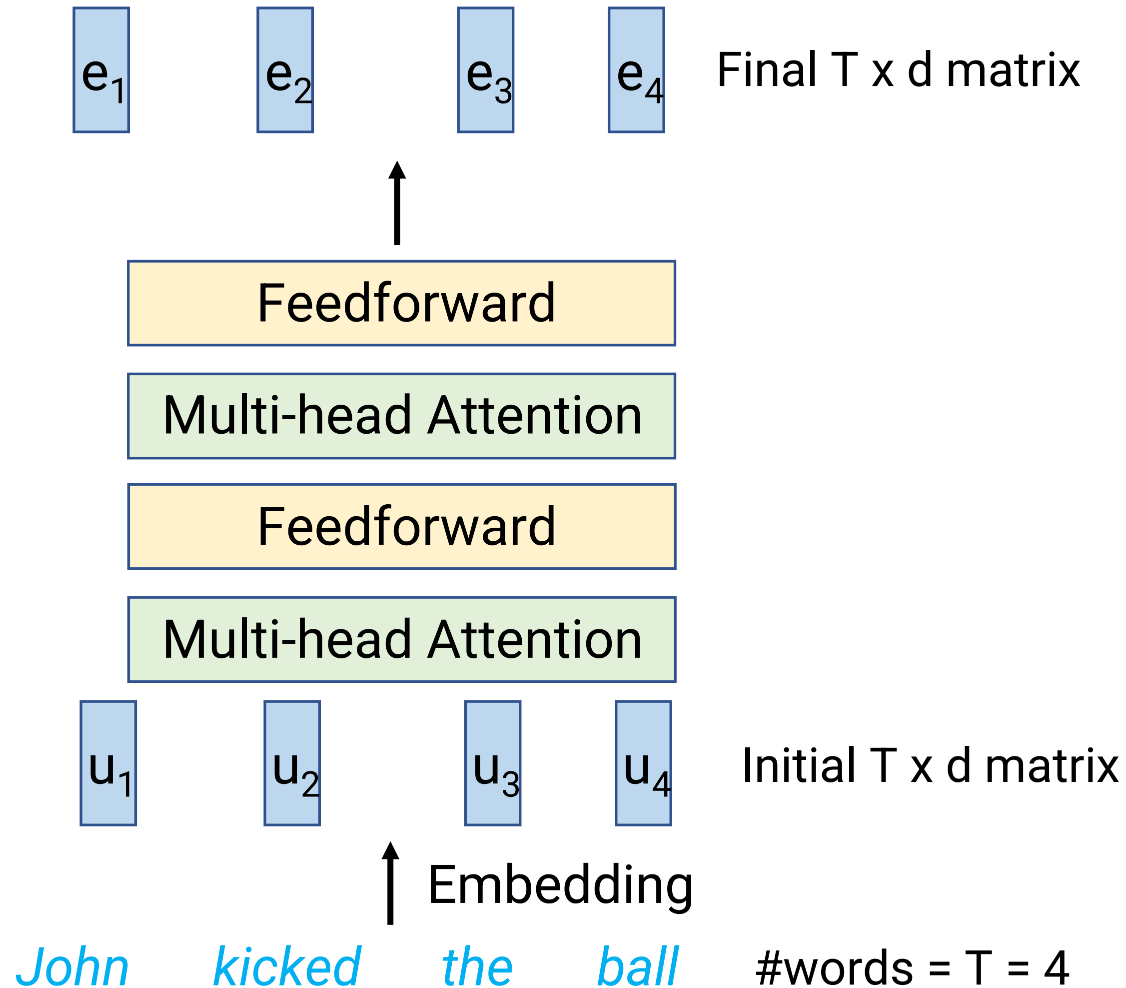
- How does a Transformer "work"?
- **Input layer**: Specify each word & its position in the sequence
- **Multi-headed attention layers**: For each word, retrieve information about related words, incorporate into the word's representation
- **Feedforward layers**: Do additional non-linear processing of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

Runtime comparison



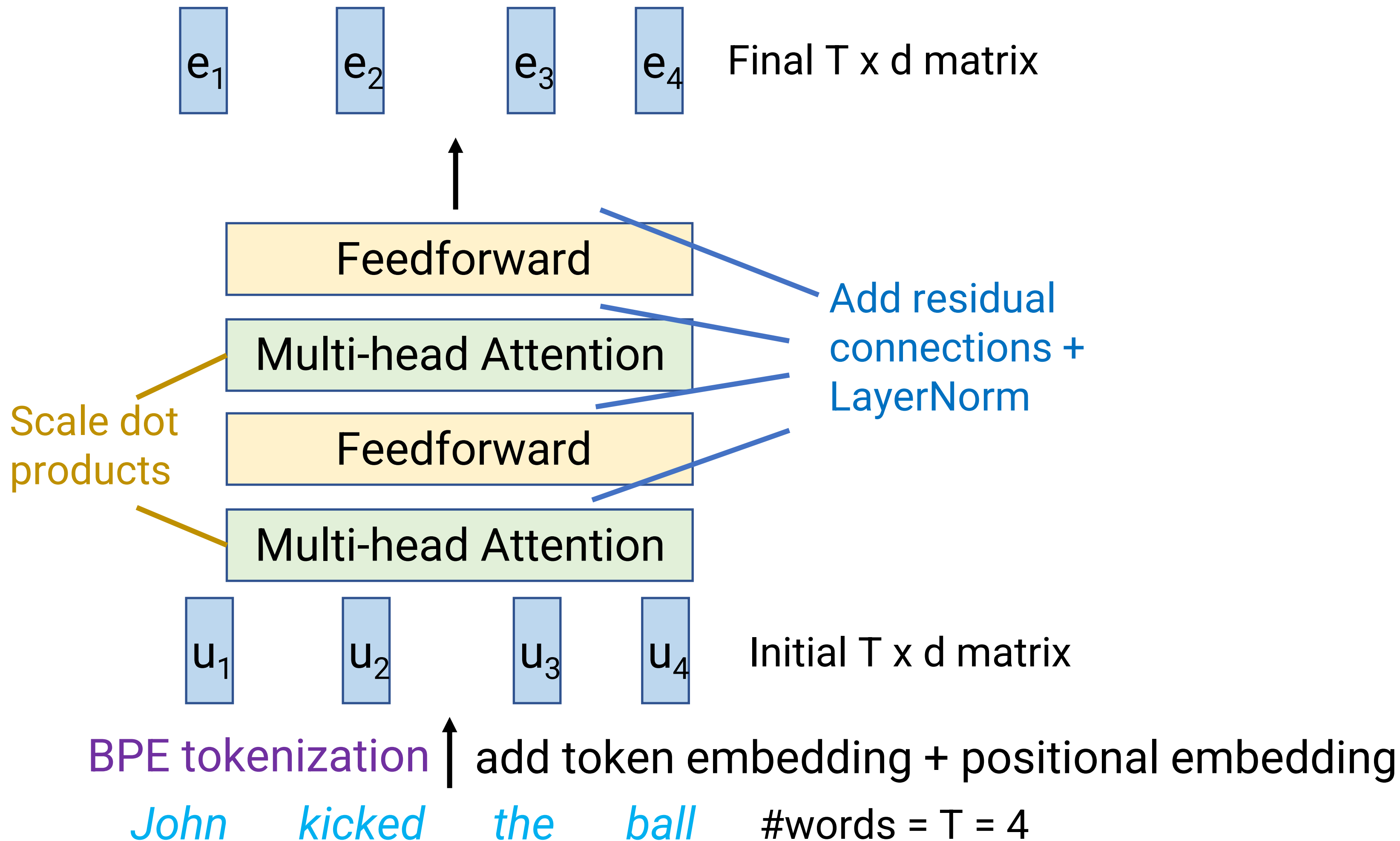
- RNNs
  - Linear in sequence length
  - But all operations have to happen in series
- Transformers
  - Quadratic in sequence length (T x T matrices)
  - But can be parallelized (big matrix multiplication)

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…**

The Full Transformer



Full Transformer also includes bells and whistles:

- Byte pair encoding
- Scaled dot product attention
- Residual connections between layers
- LayerNorm

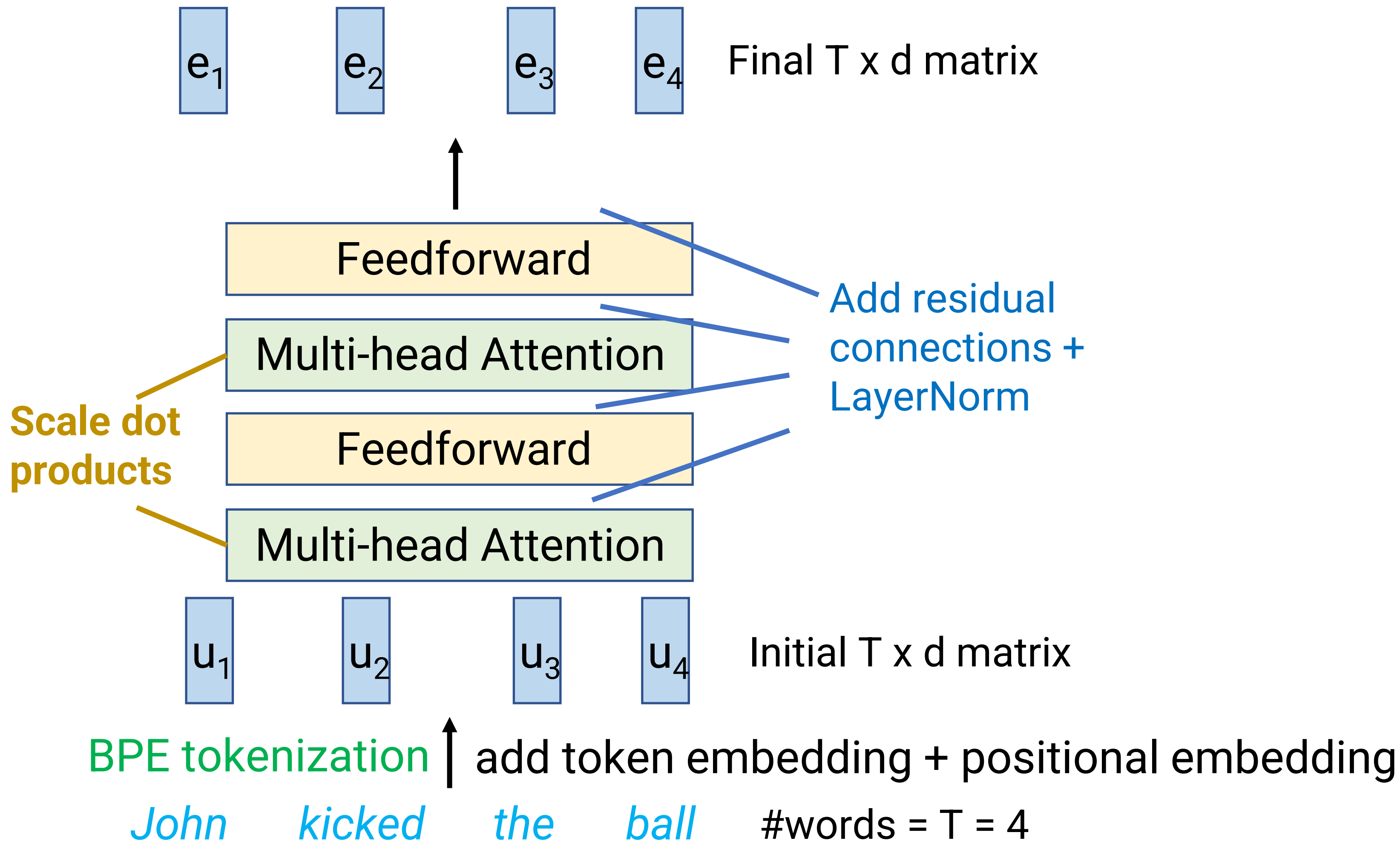
Byte Pair Encoding

- Normal word vectors have a problem: How to deal with super rare words?
  - Names? Typos?
  - Vocabulary can't contain literally every possible word…
- Solution: Tokenize string into "subword tokens"
  - Common words = 1 token
  - Rare words = multiple tokens

Aragorn told Frodo to mind Lothlorien — 6 words

'Ar', 'ag', 'orn', ' told', ' Fro', 'do', ' to', ' mind', ' L', 'oth', 'lor', 'ien' — 12 subword tokens

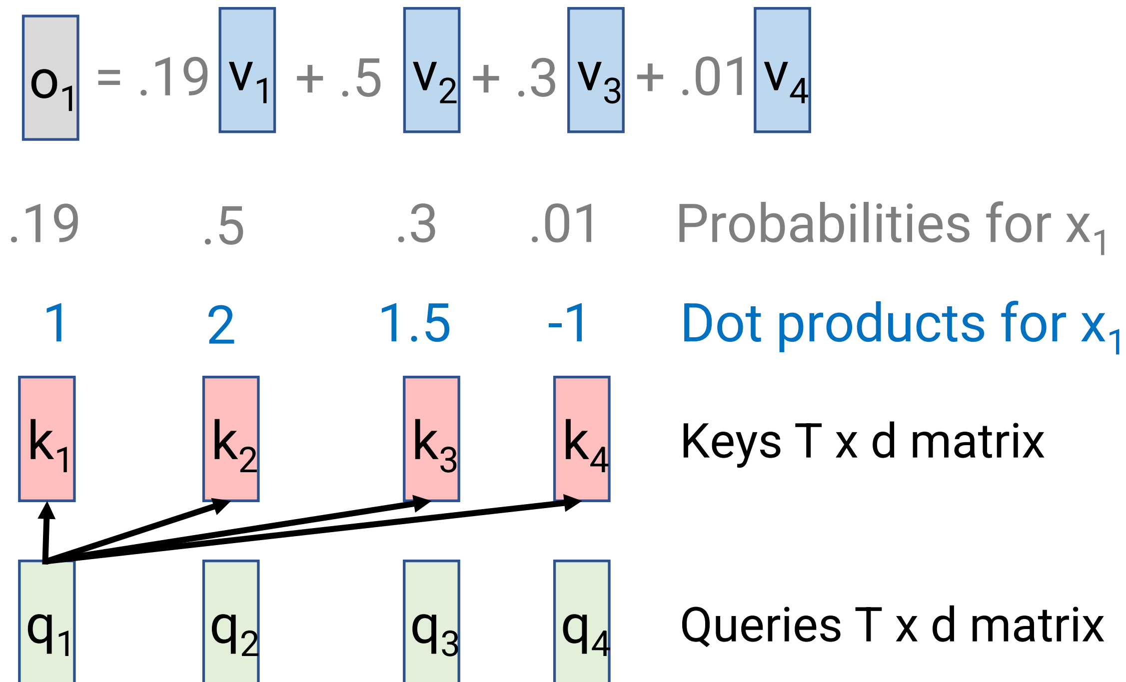
The Full Transformer



Full Transformer also includes bells and whistles:

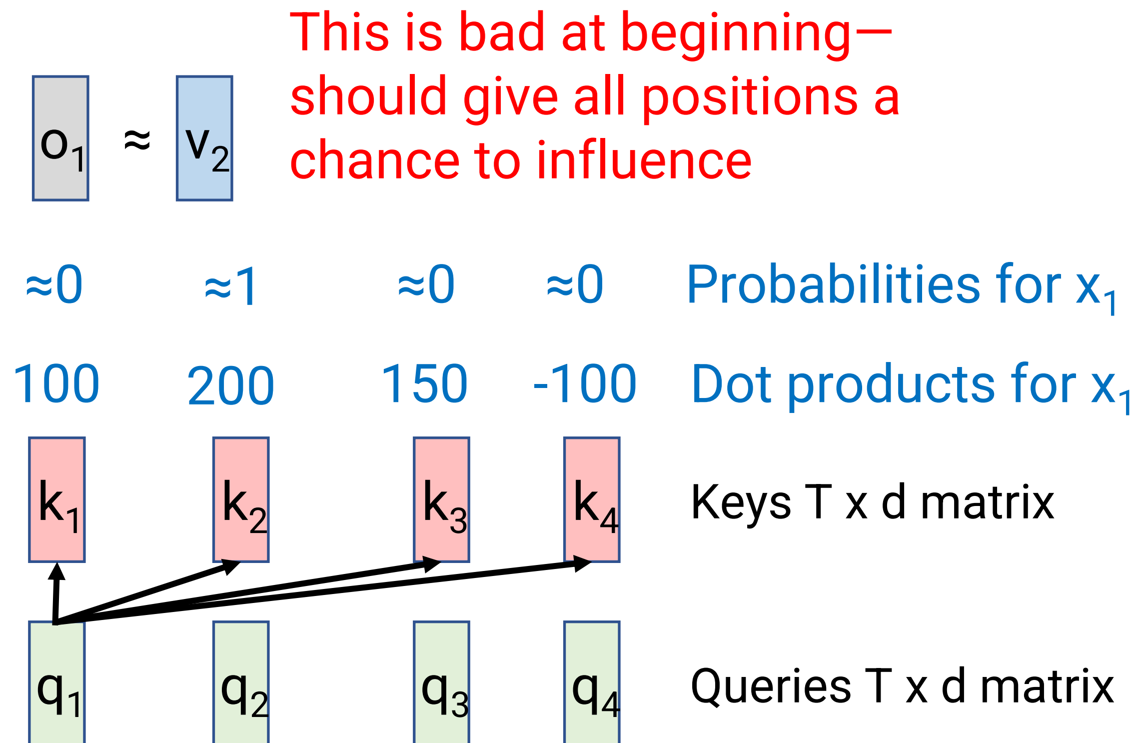
- Byte pair encoding
- **Scaled dot product attention**
- Residual connections between layers
- LayerNorm

Scaled dot product attention



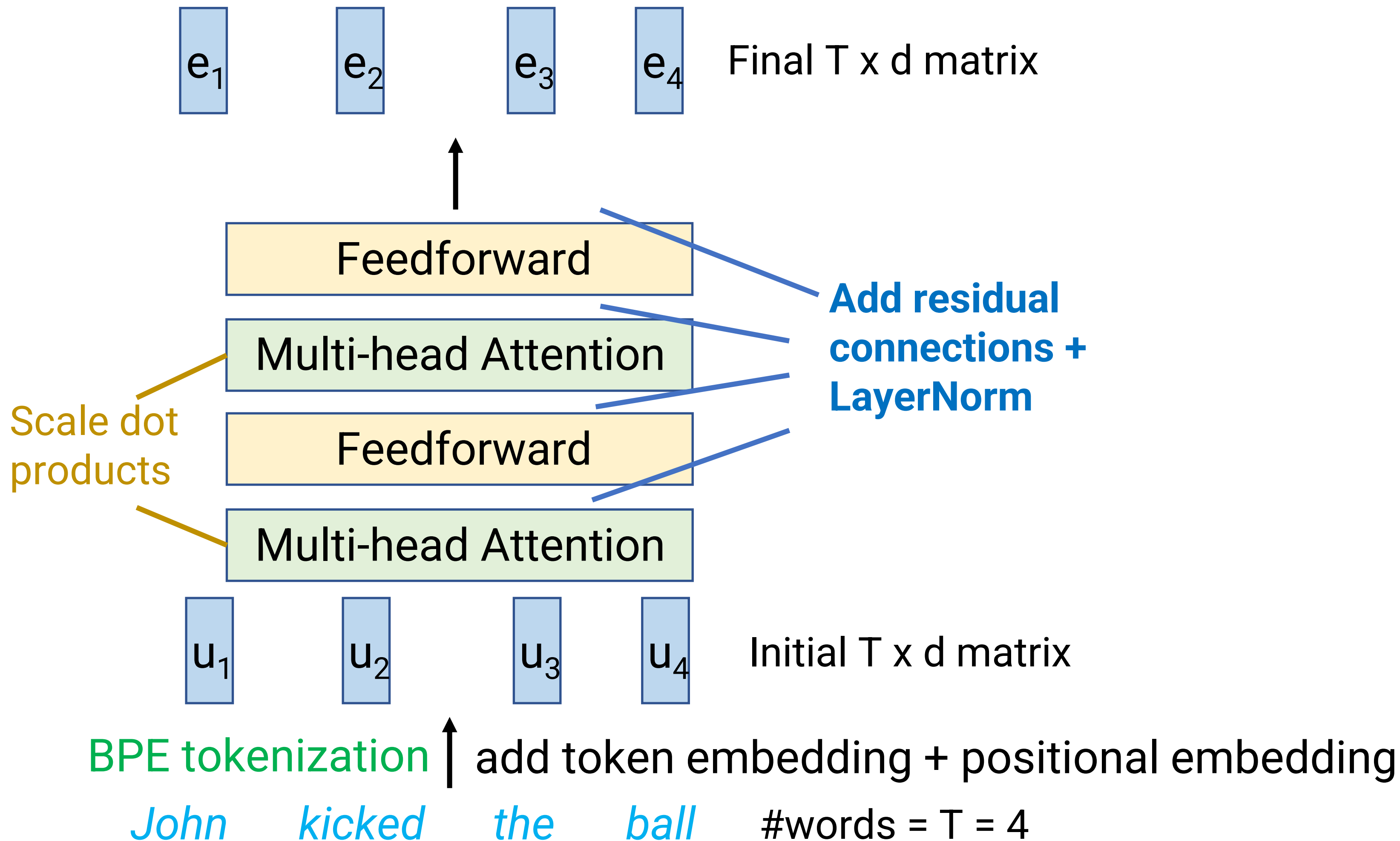
- Earlier I said, "Dot product q_1 with $[k_1, \ldots, k_T]$"
- Actually, you take dot product and then divide by $\sqrt{d_{attn}}$
- Why?
  - If d large, dot product between random vectors will be large
  - This makes probabilities close to 0/1
  - Scaling dot products down encourages more even attention at beginning

Scaled dot product attention



- Earlier I said, "Dot product q_1 with $[k_1, \ldots, k_T]$"
- Actually, you take dot product and then divide by $\sqrt{d_{attn}}$
- Why?
  - If d large, dot product between random vectors will be large
  - This makes probabilities close to 0/1
  - Scaling dot products down encourages more even attention at beginning

The Full Transformer

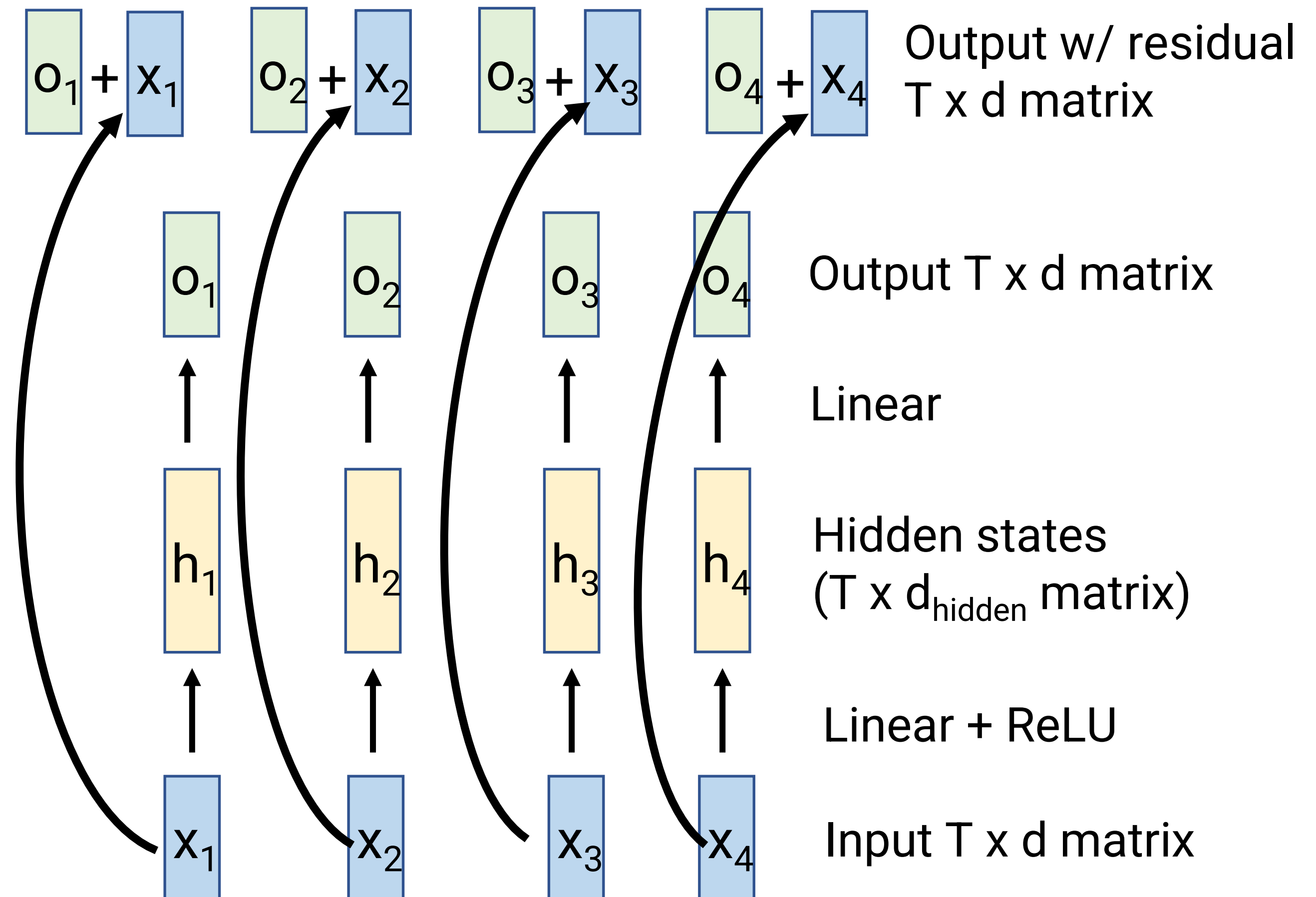


Full Transformer also includes bells and whistles:

- Byte pair encoding
- Scaled dot product attention
- **Residual connections between layers**
- **LayerNorm**

Residual Connections

- Feedforward and multi-headed attention layers
  - Take in T x d matrix X
  - Output T x d matrix O
- We add a “residual” connection: we actually use X + O as output
  - Makes it easy to copy information from input to output
  - Think of O as how much we **change** the previous value
- Same idea also common in CNNs!
  - Reduces vanishing gradient issues



Layer Normalization ("LayerNorm")

- LayerNorm is a layer/building block that "normalizes" a vector
- Input x: vector of size d
- Output y: vector of size d
- Formula:

$$\mu = \frac{1}{d}\sum_{i=1}^{d} x_i$$ Mean of components of x

$$\sigma^2 = \frac{1}{d}\sum_{i=1}^{d}(x_i - \mu)^2$$ Variance of components of x

$$y = a \cdot \frac{x - \mu}{\sqrt{\sigma^2 + \varepsilon}} + b$$

Normalized x

1. Normalize: Subtract by mean, divide by standard deviation
2. Rescale: Multiply by a, add b

- Parameters
  - a & b are scalar parameters, let model learn good scale/shift
    - Without these, all vectors forced to have mean=0, variance=1
  - ε is hyperparameter: Some small number to prevent division by 0

x = [100, 200, 100, 0]

μ = 100

$\sigma^2 = ¼ * (0^2 + 100^2 + 0^2 + 100^2) = 5000$

Normalized x =

[0, 100, 0, -100] / $\sqrt{5000}$

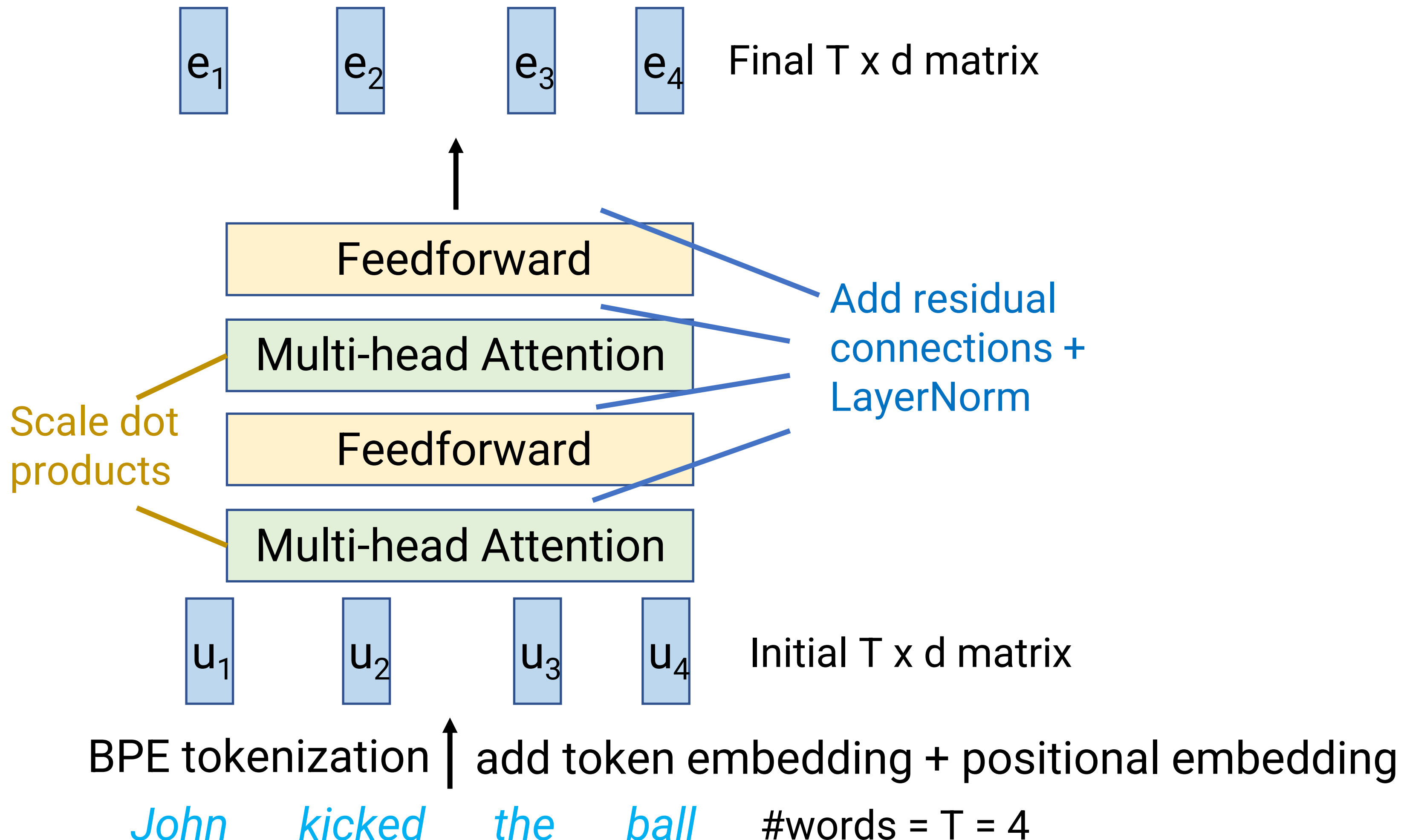
= [0, 1.4, 0, -1.4] (If ε ≈ 0)

Output = [b, 1.4a+b, b, -1.4a+b]

LayerNorm in Transformers

- After every feedforward & multi-headed attention layer, we also add Layer Normalization
  - Input: vectors $x_1, \ldots, x_T$
  - Compute μ and σ^2 for each vector
  - Normalize each vector
  - Use the same a and b to rescale each vector
- Is applied after residual connection
  - Output of each layer is $\mathrm{LayerNorm}(x + \mathrm{Layer}(x))$
- Why? Stabilizes optimization by avoiding very large values

The Full Transformer



Full Transformer also includes bells and whistles:

- Byte pair encoding
- Scaled dot product attention
- Residual connections between layers
- LayerNorm

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: Transformers as Decoders, Pre-training
  - Later: Transformers + Reinforcement Learning = ChatGPT