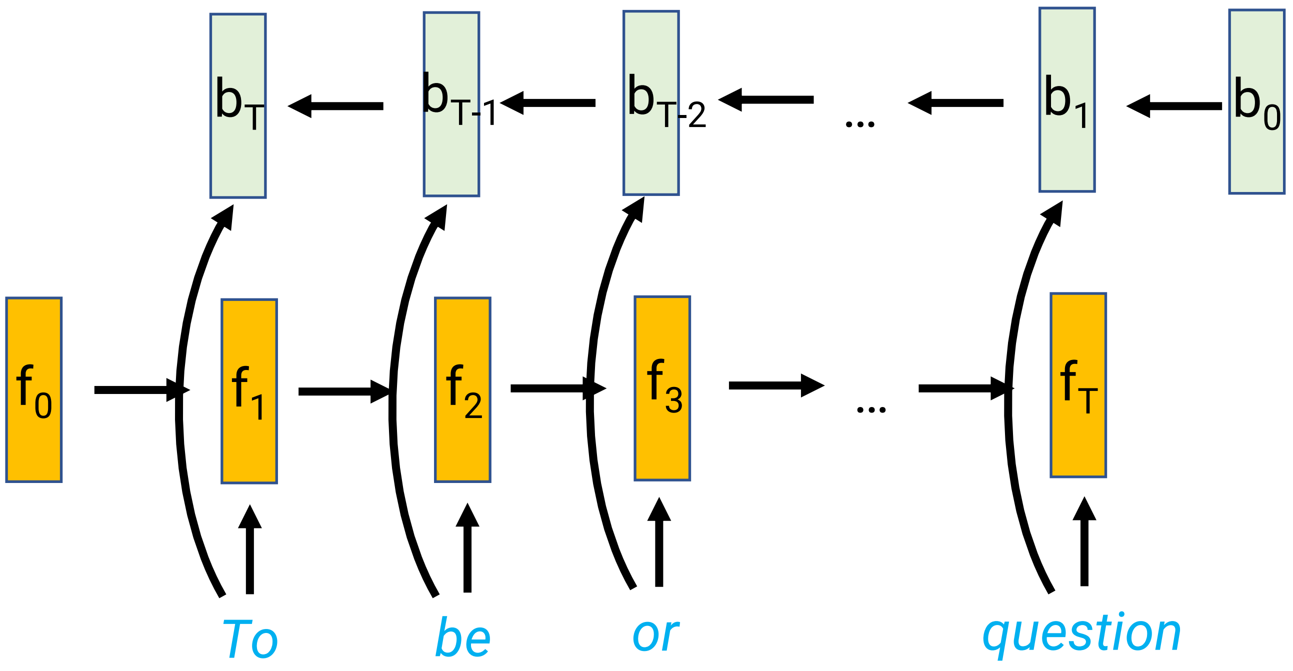
# Modern Deep Learning: Transformers, Pre-training

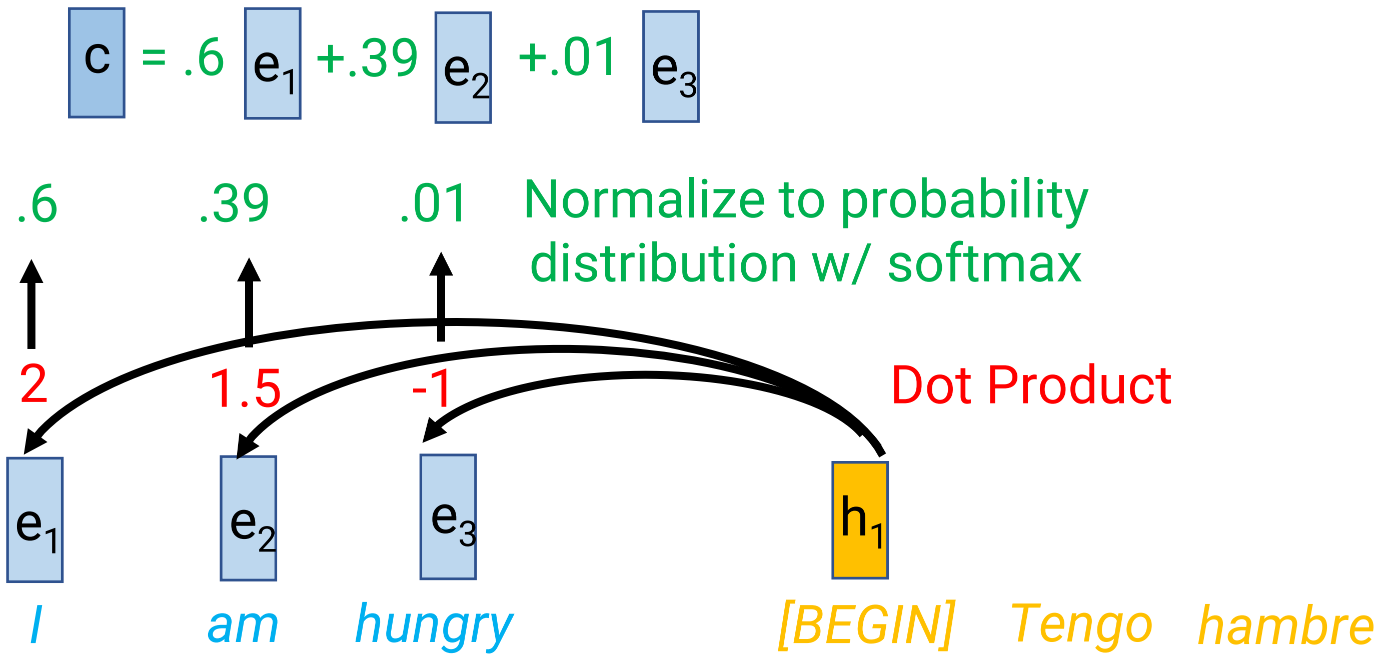
**Robin Jia**
USC CSCI 467, Spring 2023
March 2, 2023

# Review: Bi-directional RNN encoders



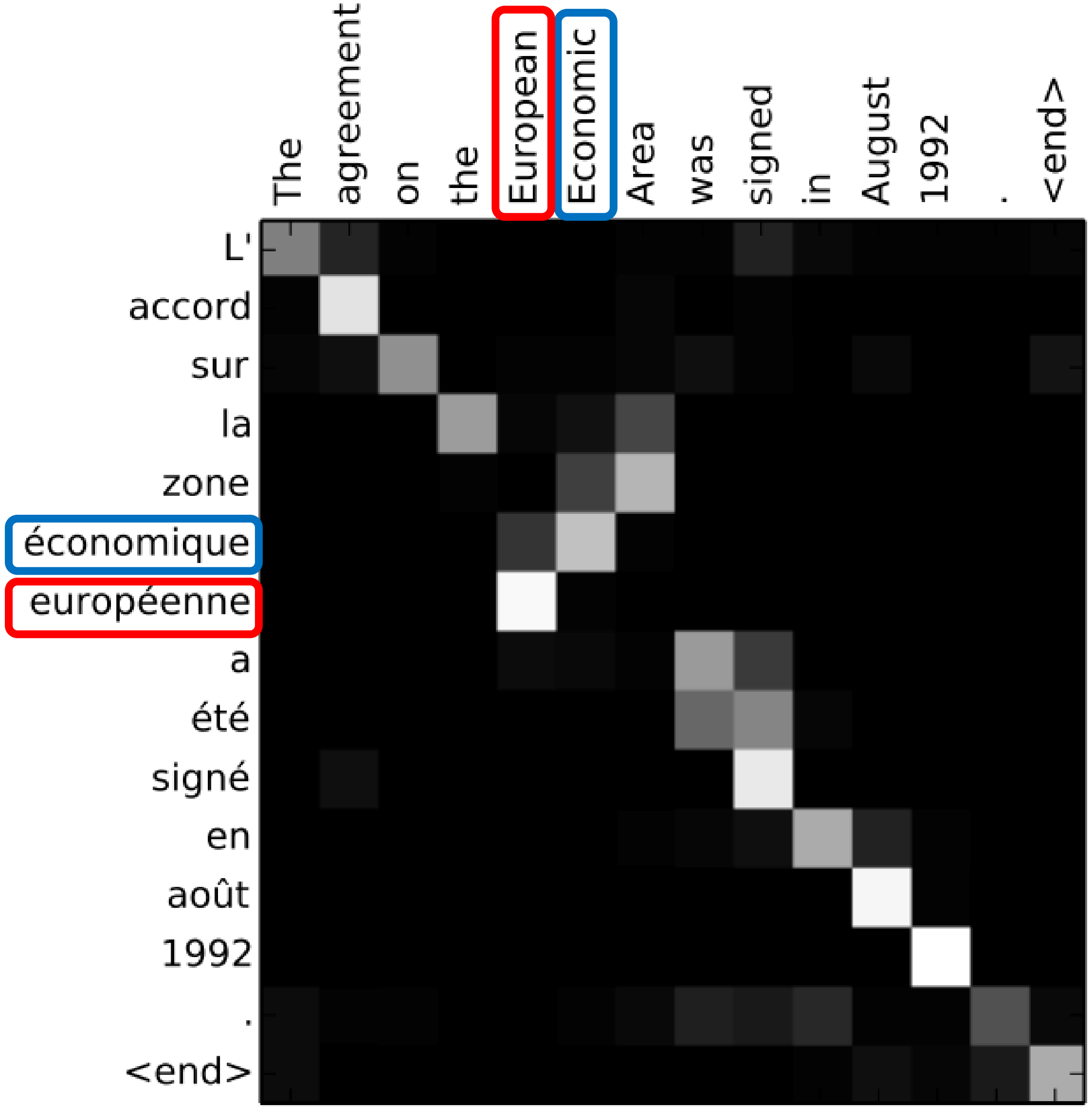

- Run one RNN left-to-right, and another one right-to-left
  - (I'll call forward-direction hidden states $f_t$, backward-direction hidden states $b_t$)
- Result: Forward and backward encodings of each token in context
  - Can just use the final 2 hidden states as features
  - Can use these as vectors to run attention over

# Review: Attention (with dot product)




- Input:
  - Encoder hidden states for each input token
  - Current decoder hidden state
- Find relevant input words
  - Dot product current decoder hidden state with all encoder hidden states
  - Normalize dot products to probability distribution with softmax
- Output: “Context” vector *c* = weighted average of encoder states based on the probabilities

# Challenges of modeling sequences




- Modeling relationships between words
  - Translation alignment

# Challenges of modeling sequences

Goes with “steak”

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

Modifies “ate”

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

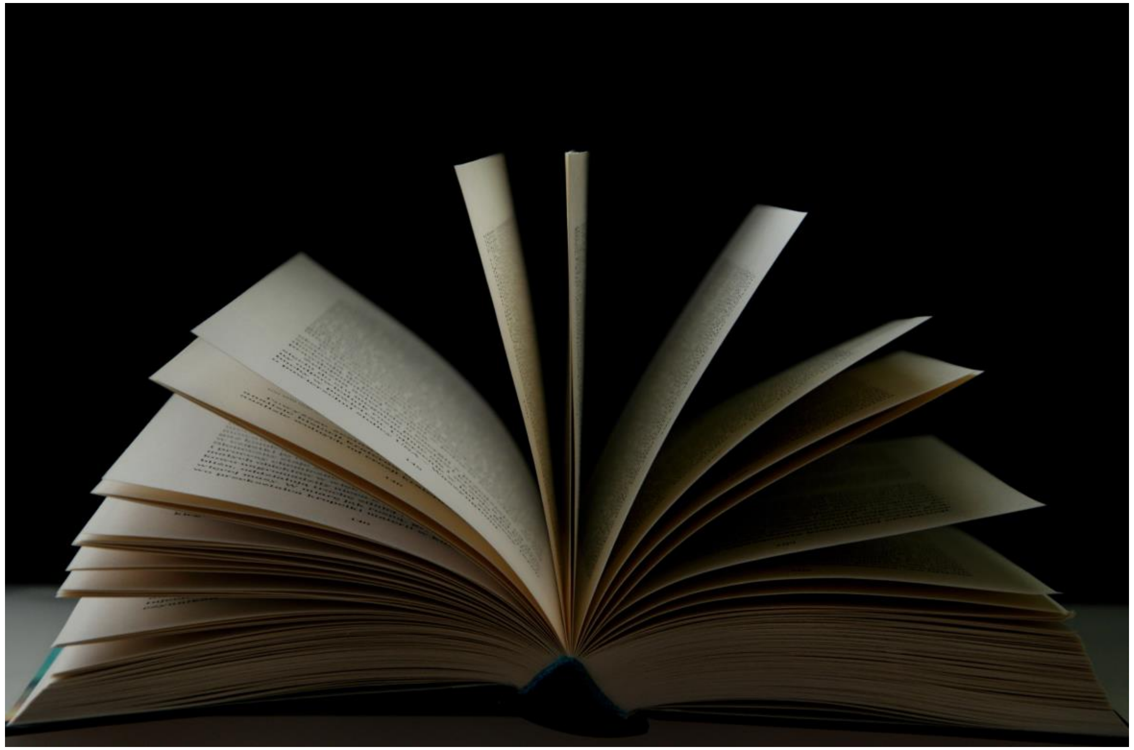
- Modeling relationships between words
  - Translation alignment
  - Syntactic dependencies

# 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

# Challenges of modeling sequences

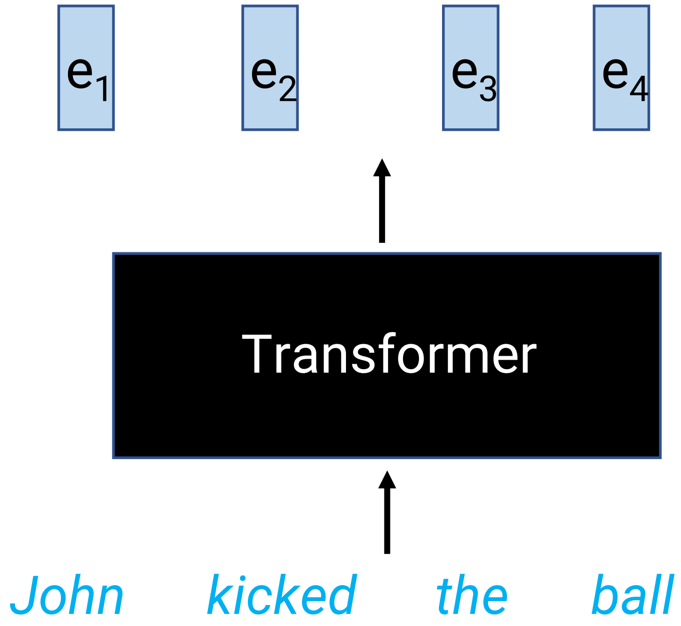


- Modeling relationships between words
  - Translation alignment
  - Syntactic dependencies
  - Coreference relationships
- Long range dependencies
  - E.g., consistency of characters in a novel
- Attention captures relationships & doesn't care about "distance"

# Outline

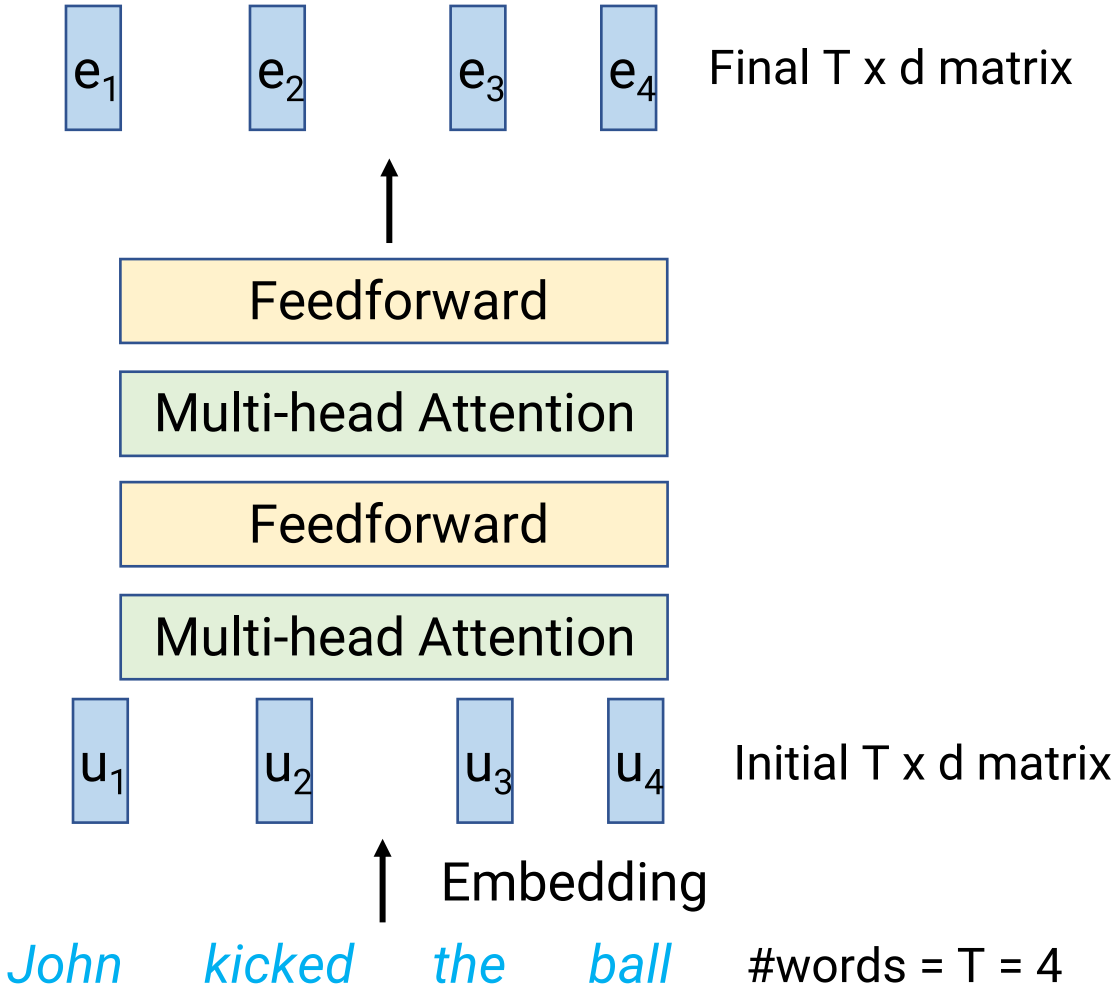
- Transformers (“Attention is all you need”)
  - Replacing recurrence with attention
  - All the bells and whistles
- Pretraining
  - Frozen features (ImageNet)
  - Fine-tuning (Masked language modeling)

# What does a Transformer (encoder) do?



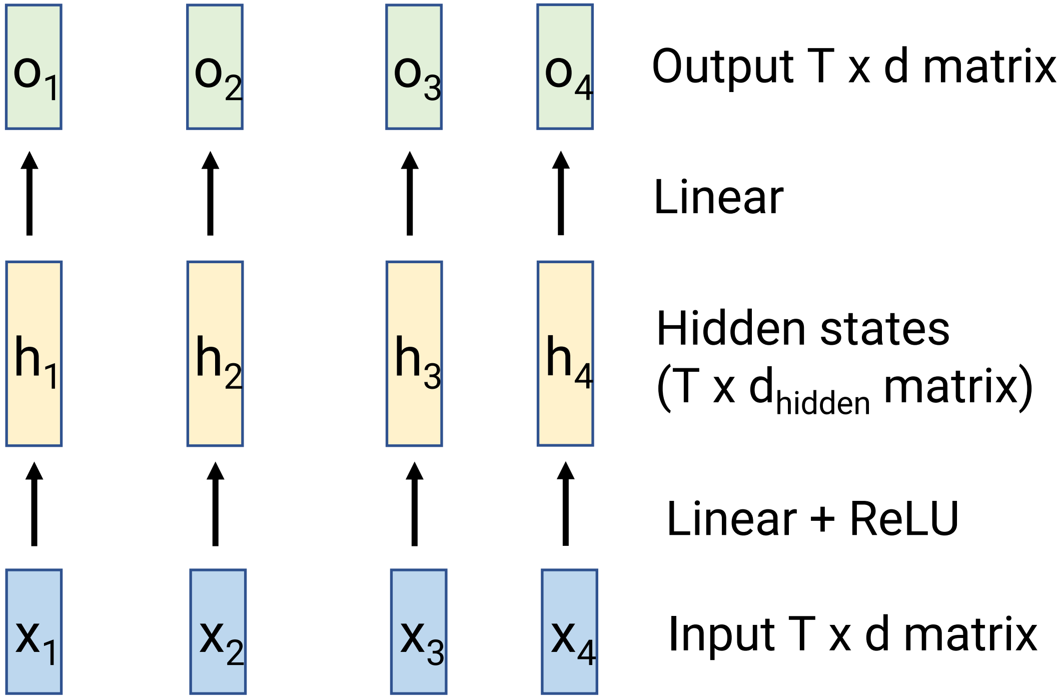

- Input: Sequence of words
- Output: Sequence of vectors, one per word
- Same “type signature” as a bi-directional RNN encoder
- Motivation
  - Don’t do explicit sequential processing
  - Instead, let attention figure out which words are relevant to each other
    - (RNN assumes sequence order is what matters)

# Transformer internals



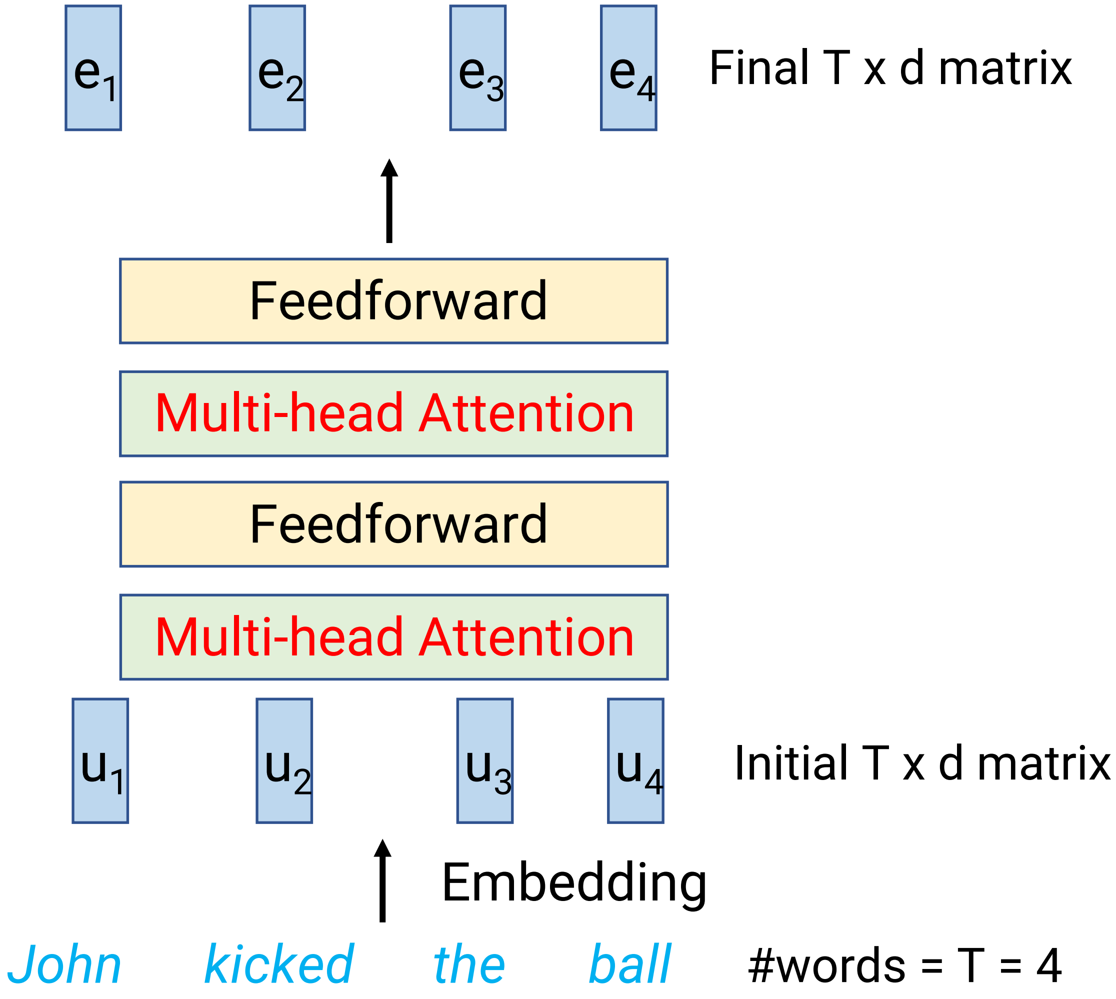

- One transformer consists of
  - Initial embeddings for each word of size d
    - Let T =#words, so initially we have a T x d matrix
  - Alternating layers of
    - "Multi-headed" attention layer
    - Feedforward layer
    - Both take in T x d matrix and output a new T x d matrix
  - Plus some bells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

# Feedforward layer



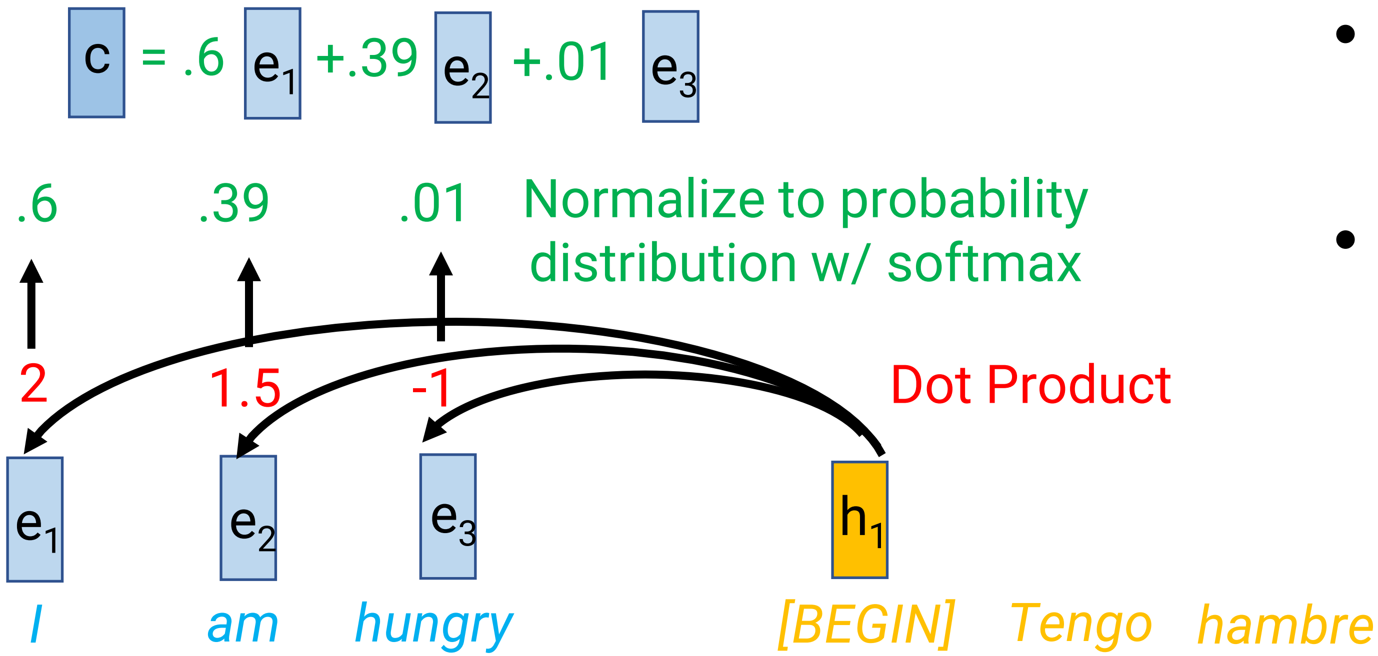

- Input: T x d matrix
- Output: Another T x d matrix
- Apply the same MLP separately to each d-dimensional vector
  - Linear layer from d to $d_{hidden}$
  - ReLU (or other nonlinearity)
  - Linear layer from $d_{hidden}$ to d
- Note: No information moves between tokens here

# Transformer internals



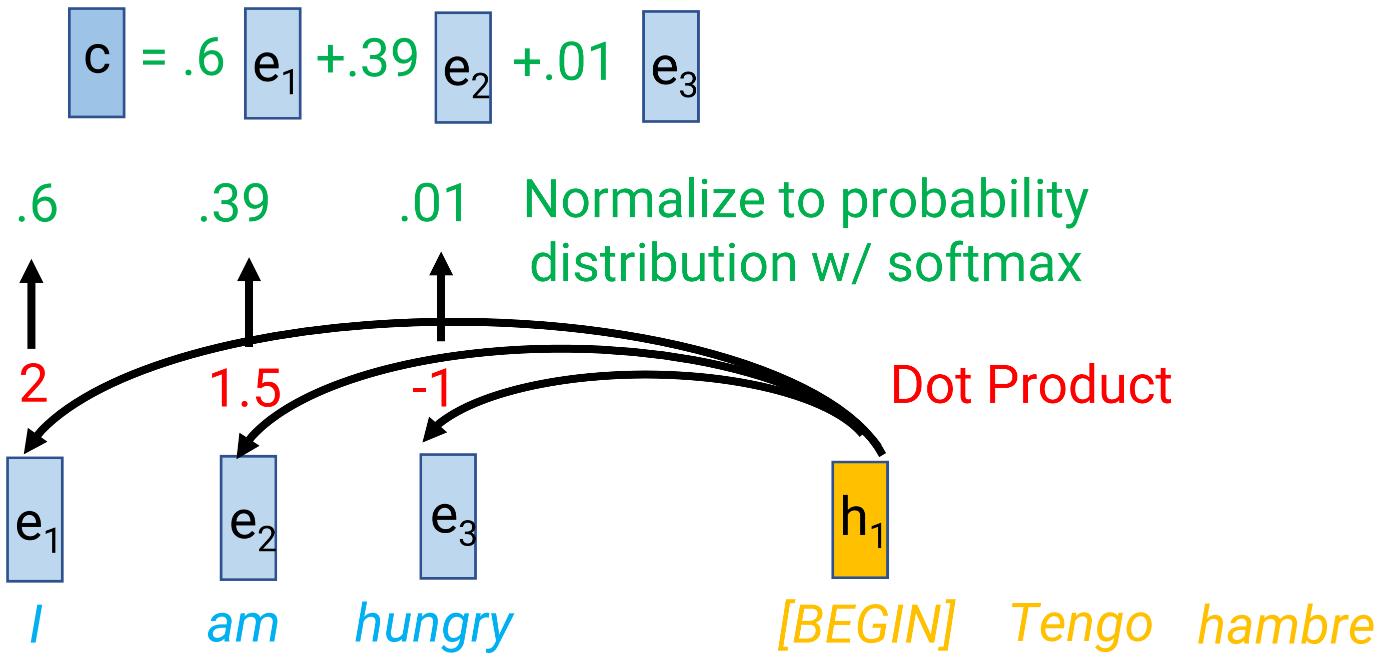

- One transformer consists of
  - Initial embeddings for each word of size d
    - Let T =#words, so initially we have a T x d matrix
  - Alternating layers of
    - **"Multi-headed" attention layer**
    - Feedforward layer
    - Both take in T x d matrix and output a new T x d matrix
  - Plus some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

# Modifying Attention



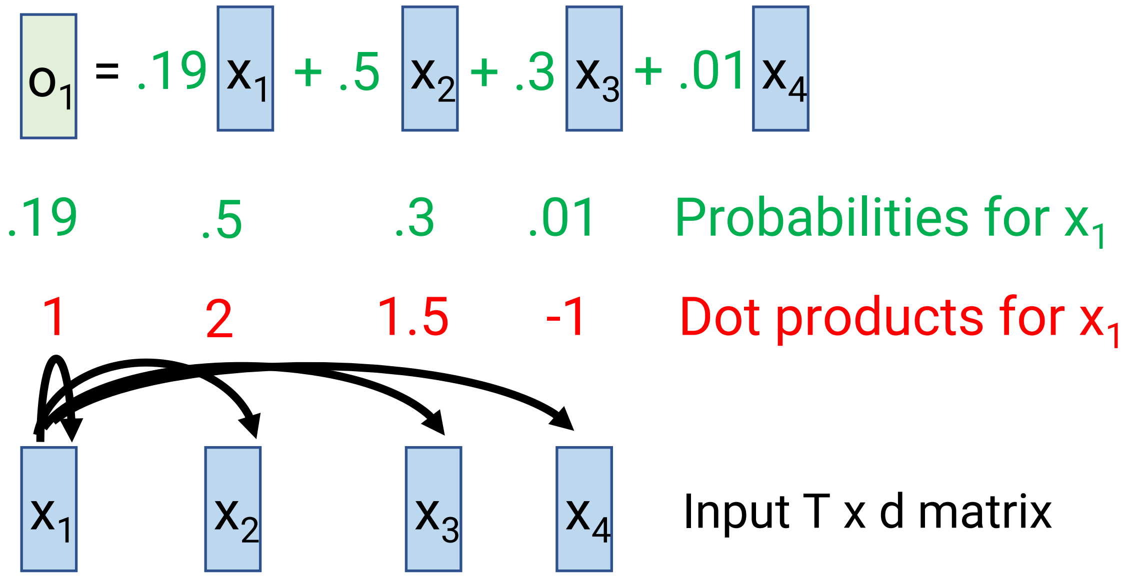

- What is a multi-headed attention layer???
- Similar to attention we've seen, but need to make 3 changes…
  - Self-attention (no separate encoder & decoder)
  - Separate queries, keys, and values
  - Multi-headed

# Change #1: Self-Attention



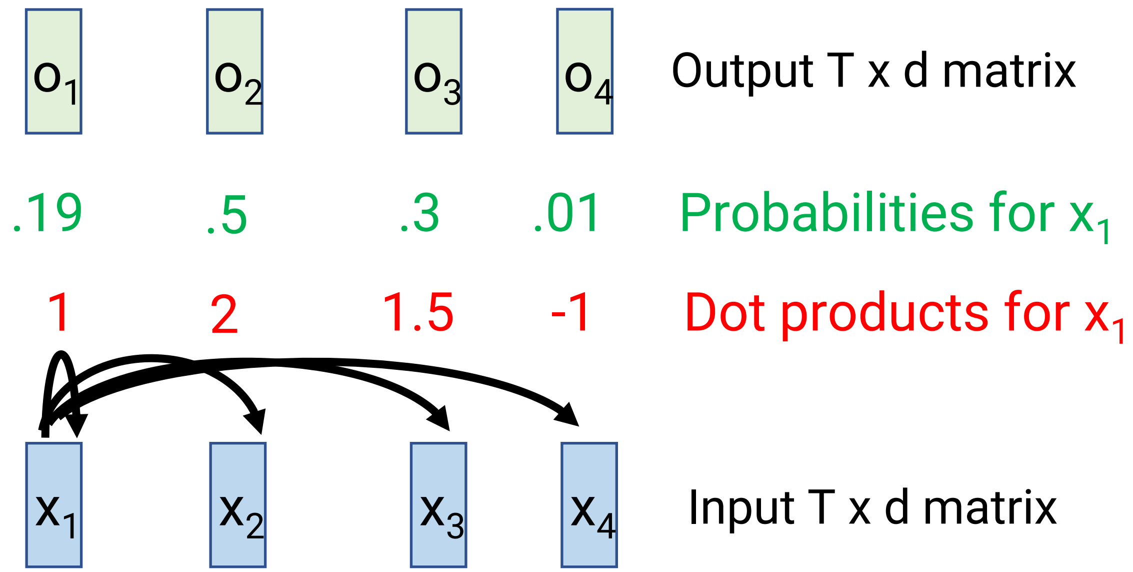

- Previously: Decoder state looks for relevant encoder states
- Self-attention: Each **encoder** state now looks for relevant (other) encoder states
- Why? Build better representation for word in context by capturing relationships to other words

# Change #1: Self-attention



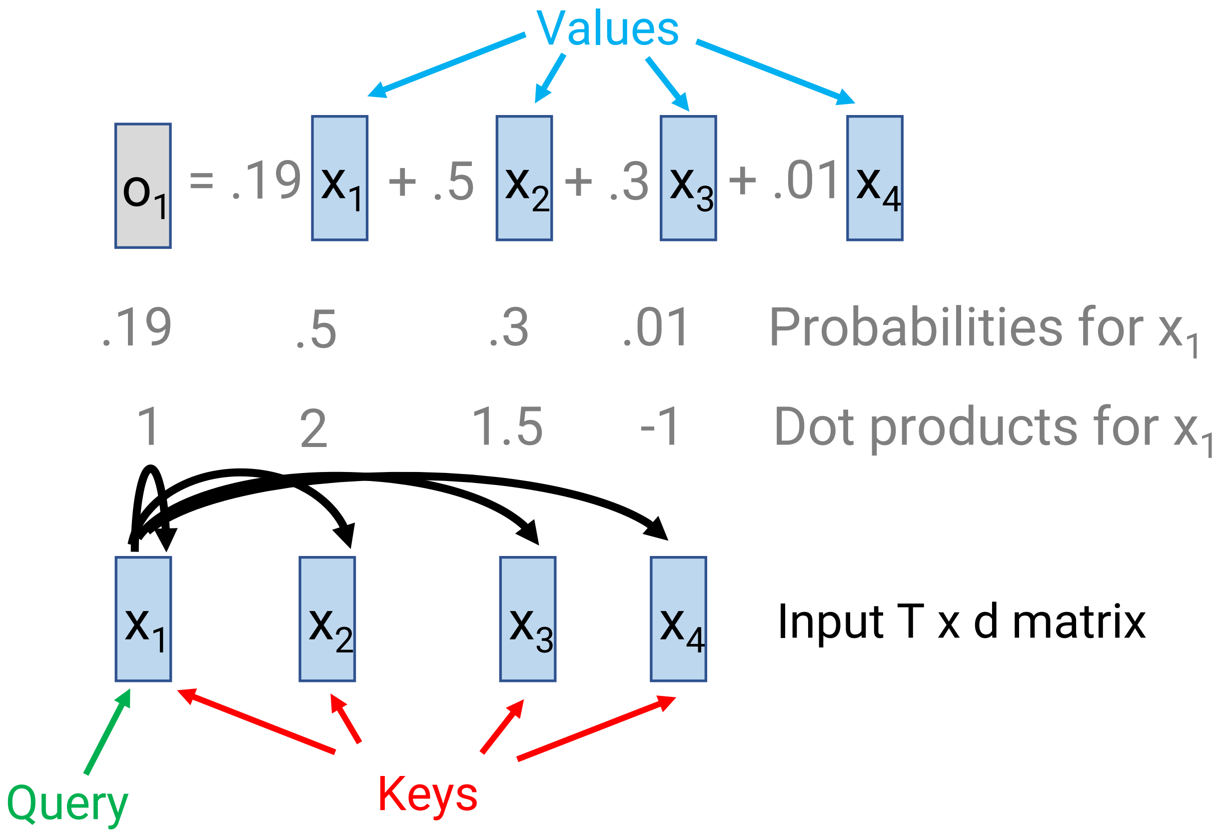

- Take $x_1$ and dot product it with all T inputs (including itself)
- Apply softmax to convert to probability distribution
- Compute output $o_1$ as weighted sum of inputs

# Change #1: Self-attention



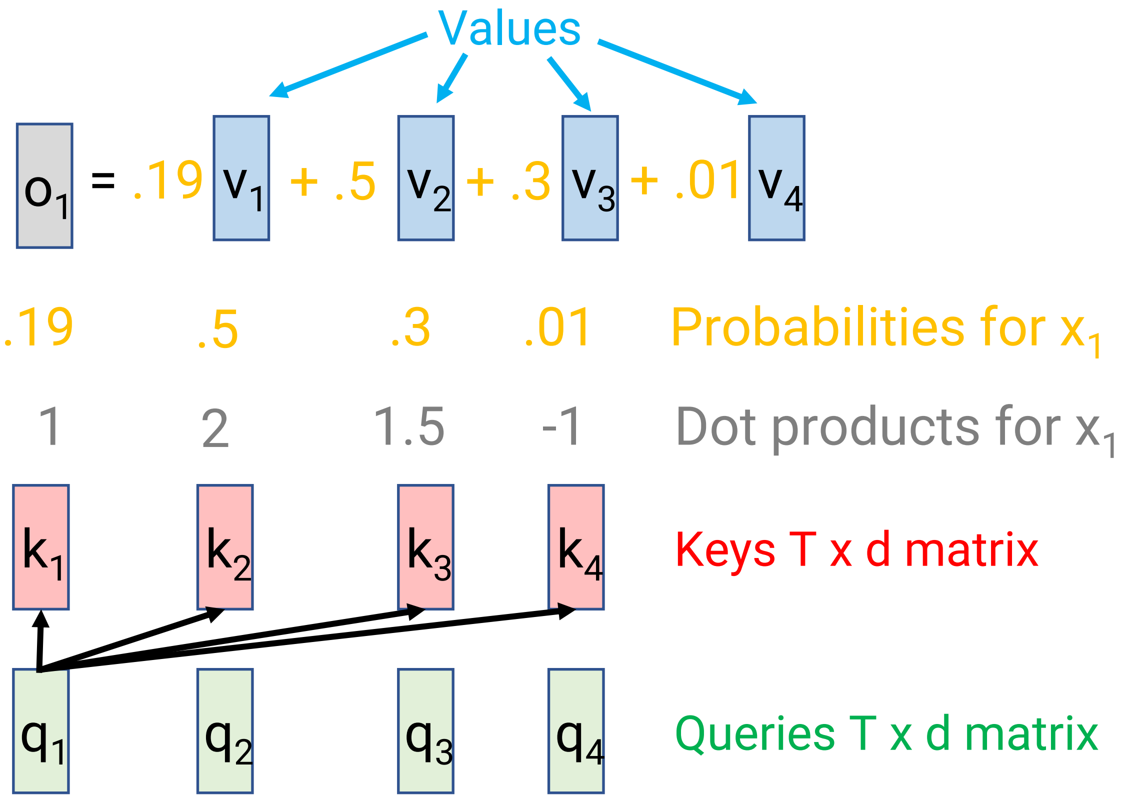

- Take $x_1$ and dot product it with all T inputs (including itself)
- Apply softmax to convert to probability distribution
- Compute output $o_1$ as weighted sum of inputs
- Repeat for t=2, 3, …, T
- Replacement for recurrence
  - RNN only allows information to flow linearly along sequence
  - Now,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

# Change #2: Separate queries, keys, and values



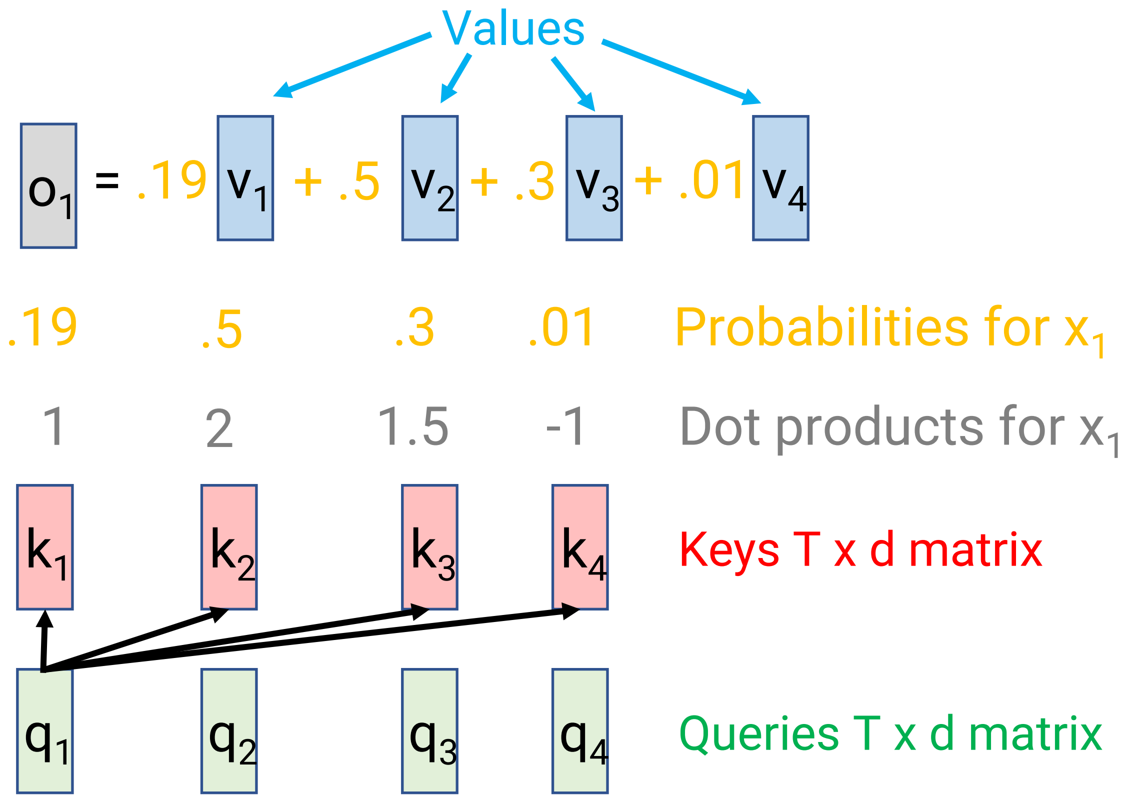

- Previously: We use input vectors in three ways
  - As "query" for current index
  - As "keys" to match with query
  - As "values" when computing output
- Idea: Use separate vectors for each usage
  - What each index "looks for" different from what it "matches with"
  - What you store in output different from what you "look for"/"match with"

# Change #2: Separate queries, keys, and values



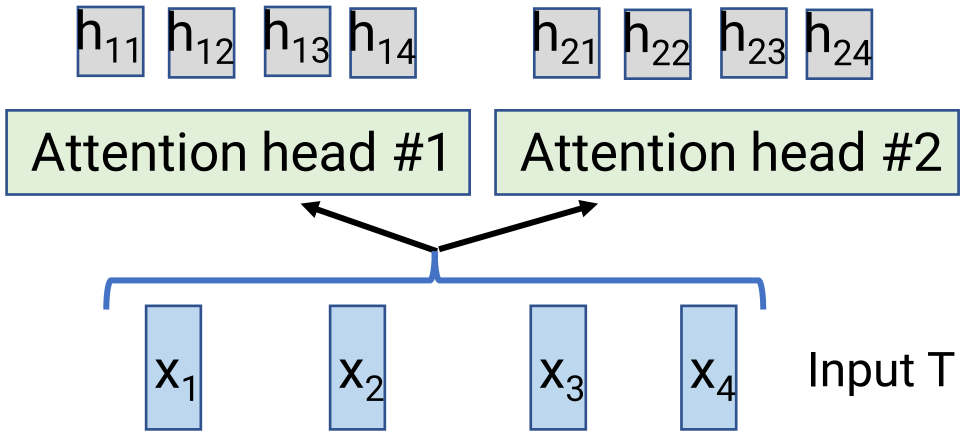

- Apply 3 separate linear layers to each of $x_1, \ldots, x_T$ to get
  - Queries $[q_1, \ldots, q_T]$
  - Keys $[k_1, \ldots, k_T]$
  - Values $[v_1, \ldots, v_T]$
  - Each linear layer maps from dimension d to dimension $d_{attn}$
- Dot product $q_1$ with $[k_1, \ldots, k_T]$
- Apply softmax to get probability distribution
- Compute $o_1$ as weighted sum of $[v_1, \ldots, v_T]$
- Repeat for t = 2, …, T

# Matrix form




- Apply 3 separate linear layers to input matrix X to get
  - Query matrix Q
  - Keys K
  - Values V
  - Each linear layer maps from dimension d to dimension $d_{attn}$
- 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
- Lessons
  - Quadratic in T
  - All you need is fast matrix multiplication
  - All indices run in parallel

# Change #3: Making it Multi-headed

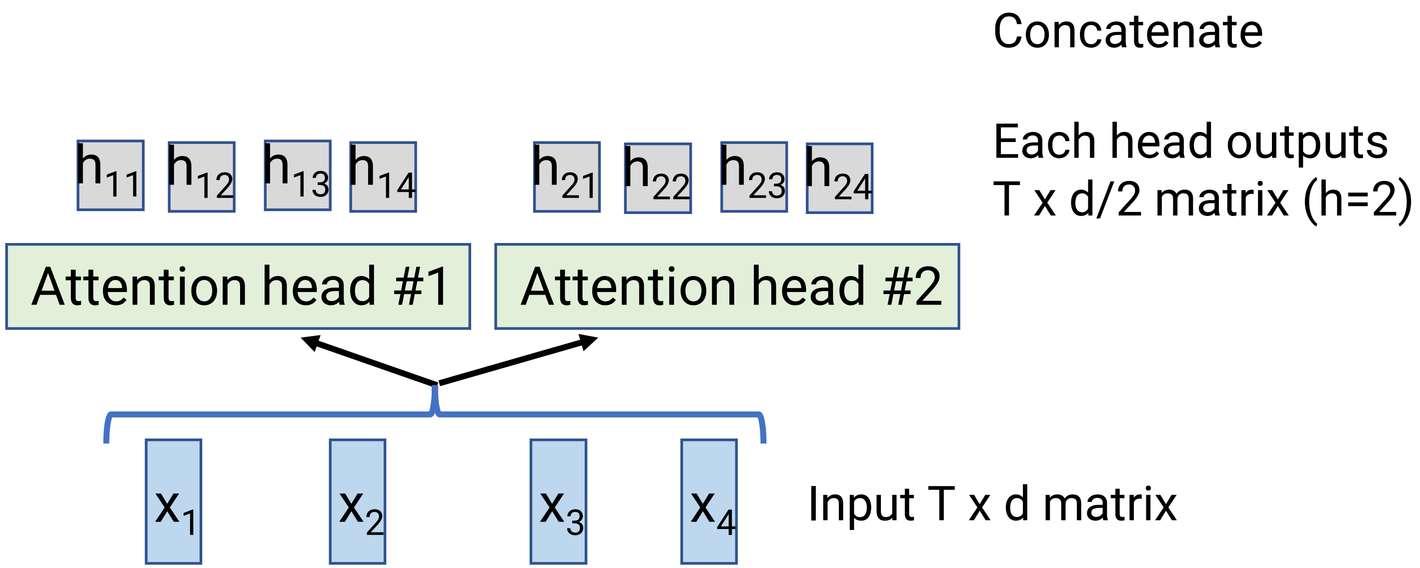



Each head outputs T x d/2 matrix (h=2)

Input T x d matrix

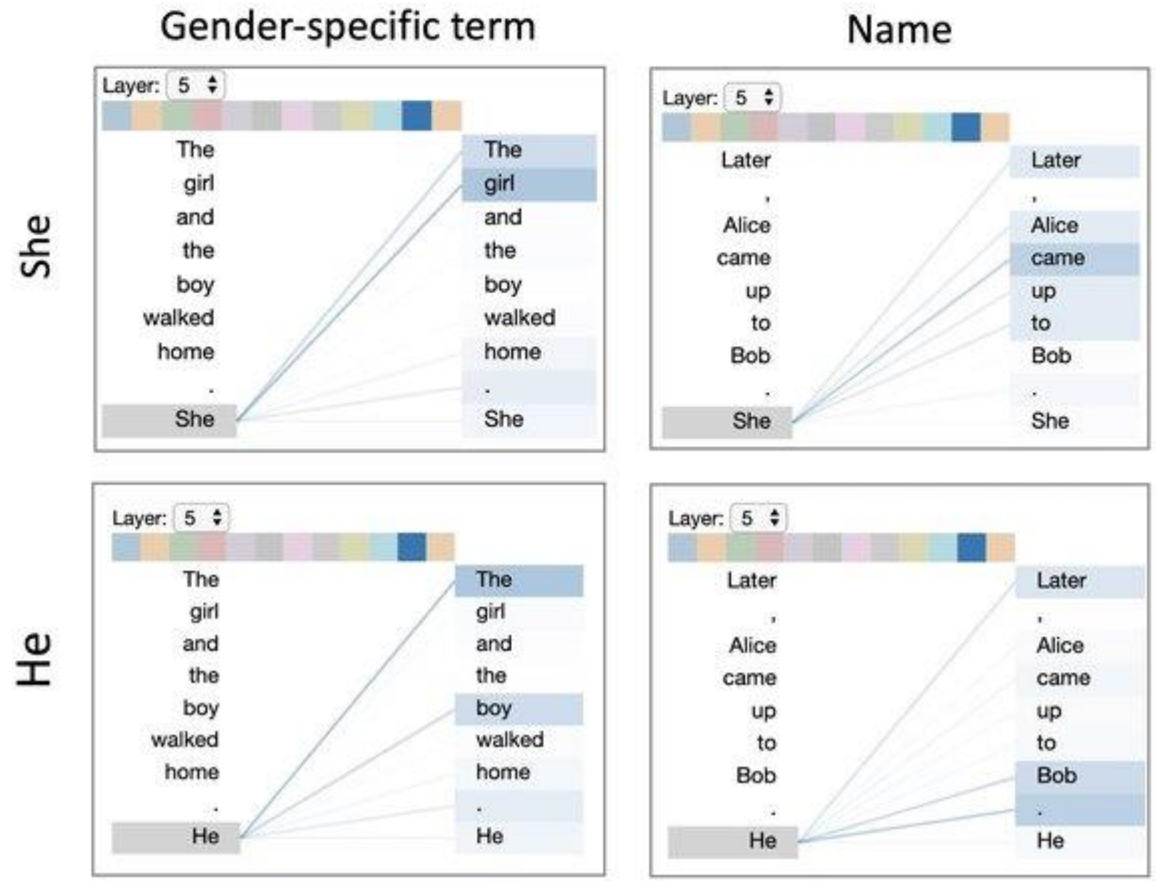
- Instead of doing attention once, have h different "heads"
  - Each has its own parameters maps to dimension $d_{attn} = d/h$
  - Concatenate at end to get output of size T x d

# Change #3: Making it Multi-headed



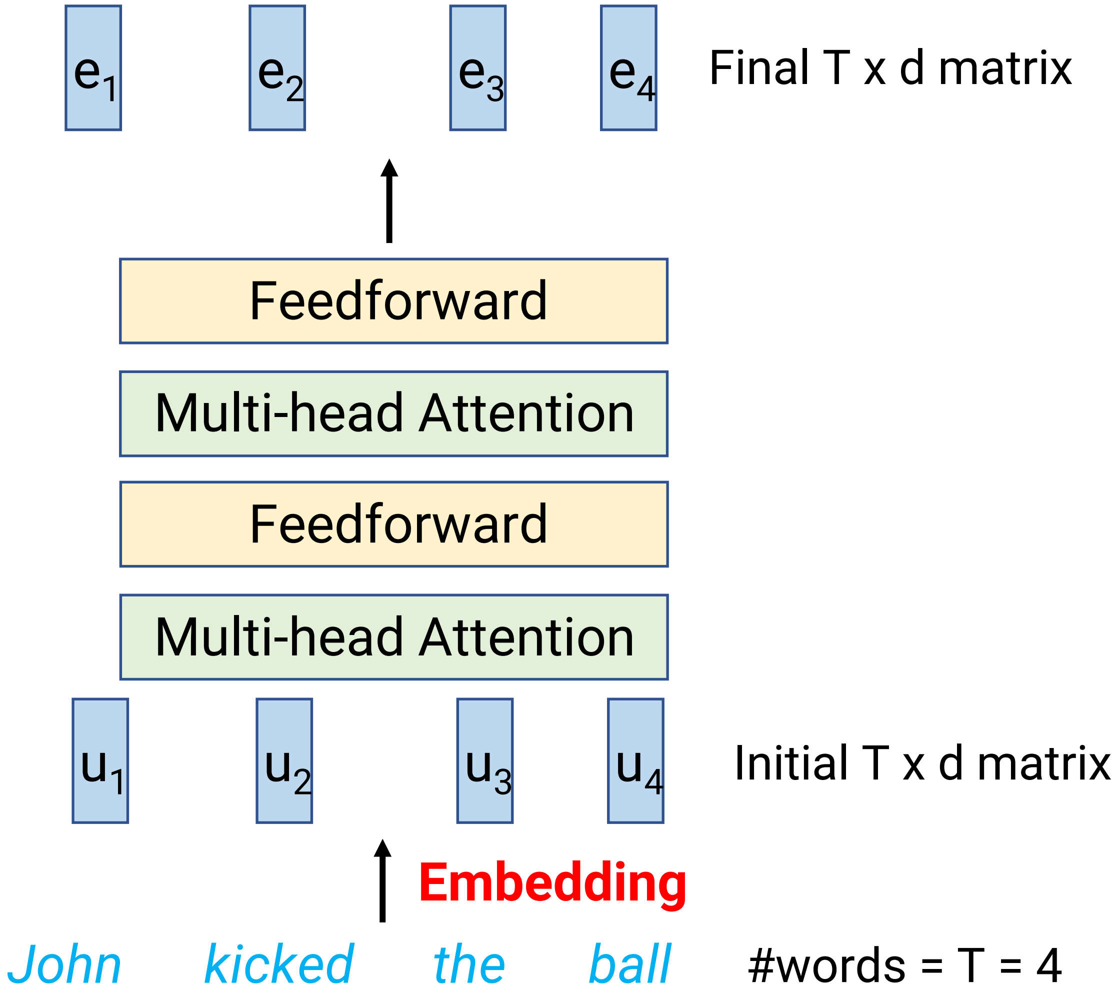

- Instead of doing attention once, have h different "heads"
  - Each has its own parameters maps to dimension $d_{attn} = d/h$
  - Concatenate at end to get output of size T x d
- Why? Different heads can capture different relationships between words

# 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

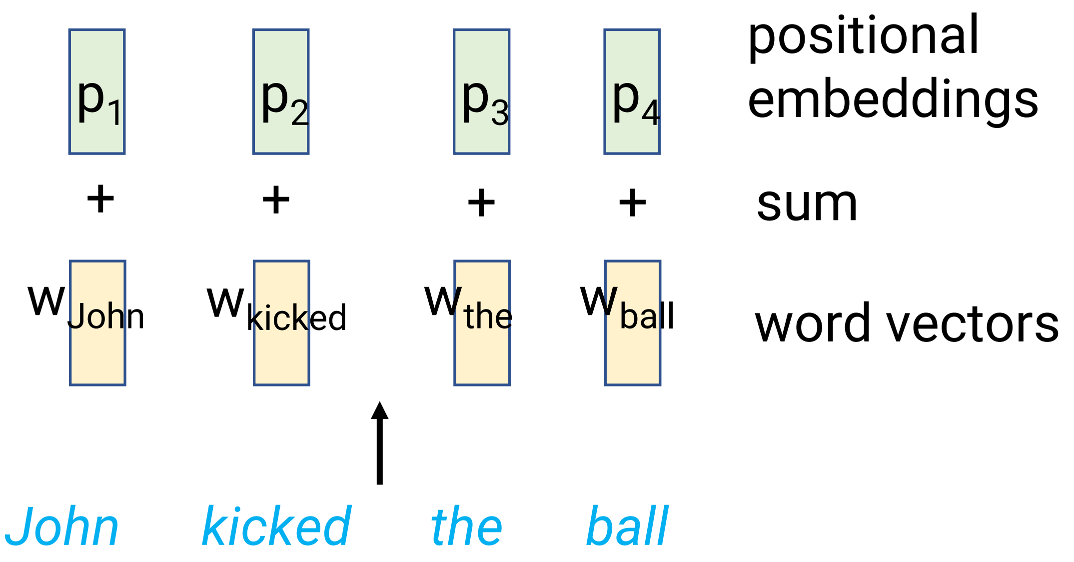
# Transformer internals



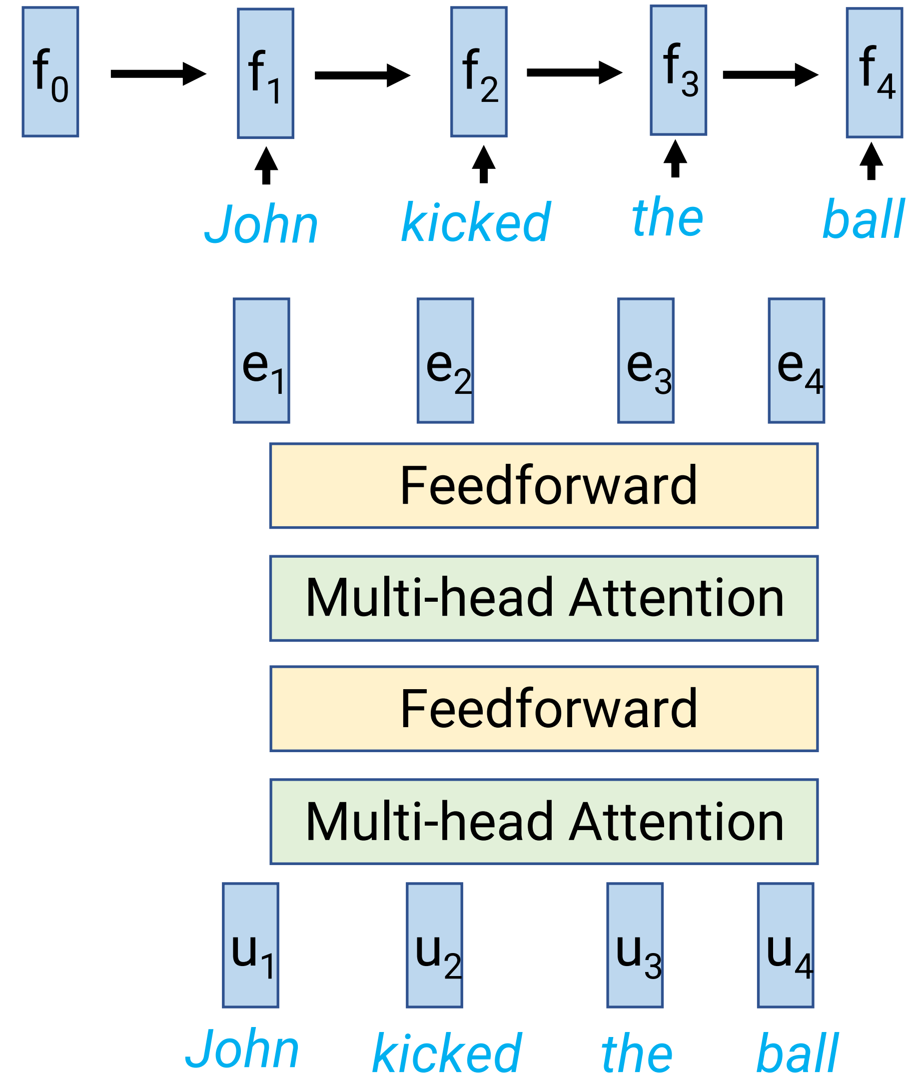

- One transformer consists of
  - **Initial embeddings** for each word of size d
    - Let T =#words, so initially we have a T x d matrix
  - Alternating layers of
    - "Multi-headed" attention layer
    - Feedforward layer
    - Both take in T x d matrix and output a new T x d matrix
  - Plus some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

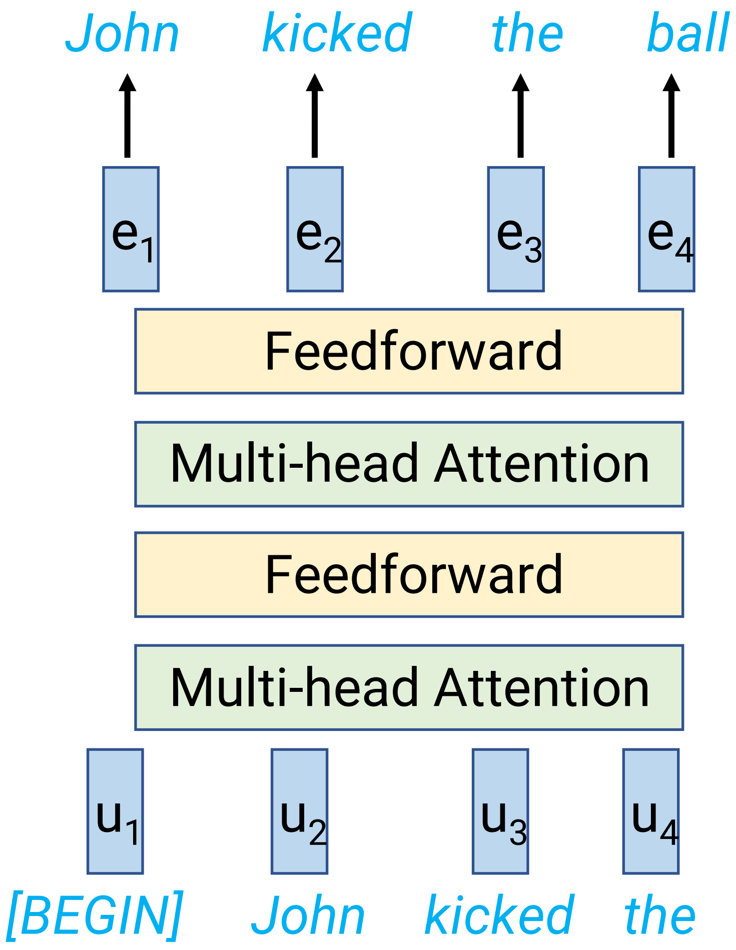


# Runtime comparison



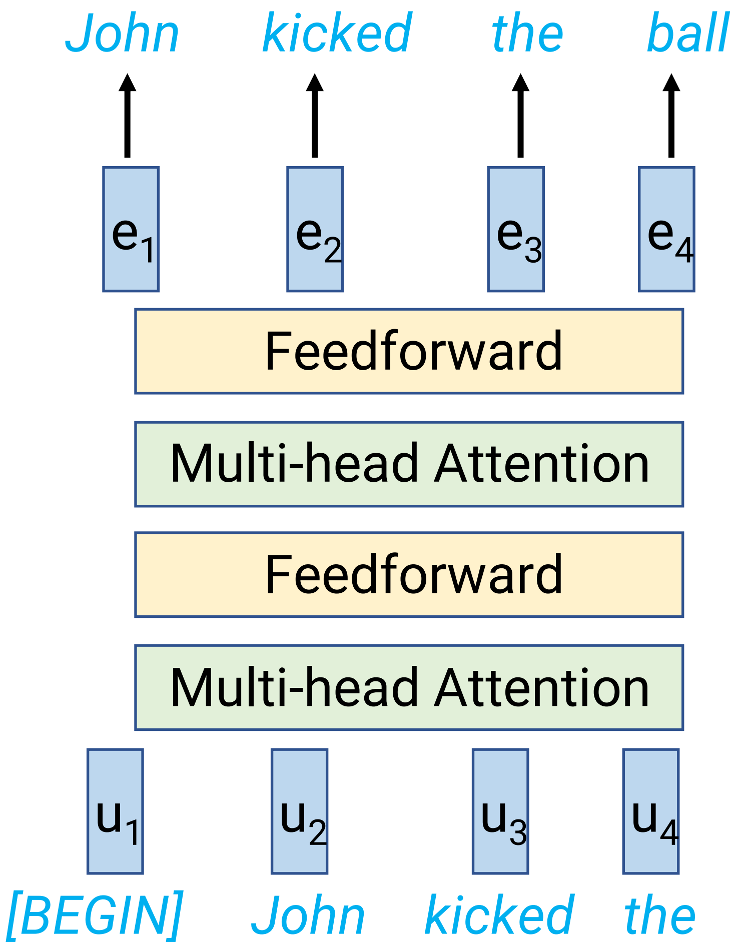

- RNNs
  - Linear in sequence length
  - But all operations have to happen in series
- Transformers
  - Quadratic in sequence length (T x T matrices)
  - But can be parallelized (big matrix multiplication)

# Transformer autoregressive decoders




- How to do autoregressive language modeling?
- Test-time
  - At time t, attend to positions 1 through t
  - Only query you have to compute is at index t (others were computed already)
  - Happens in series

# Transformer autoregressive decoders

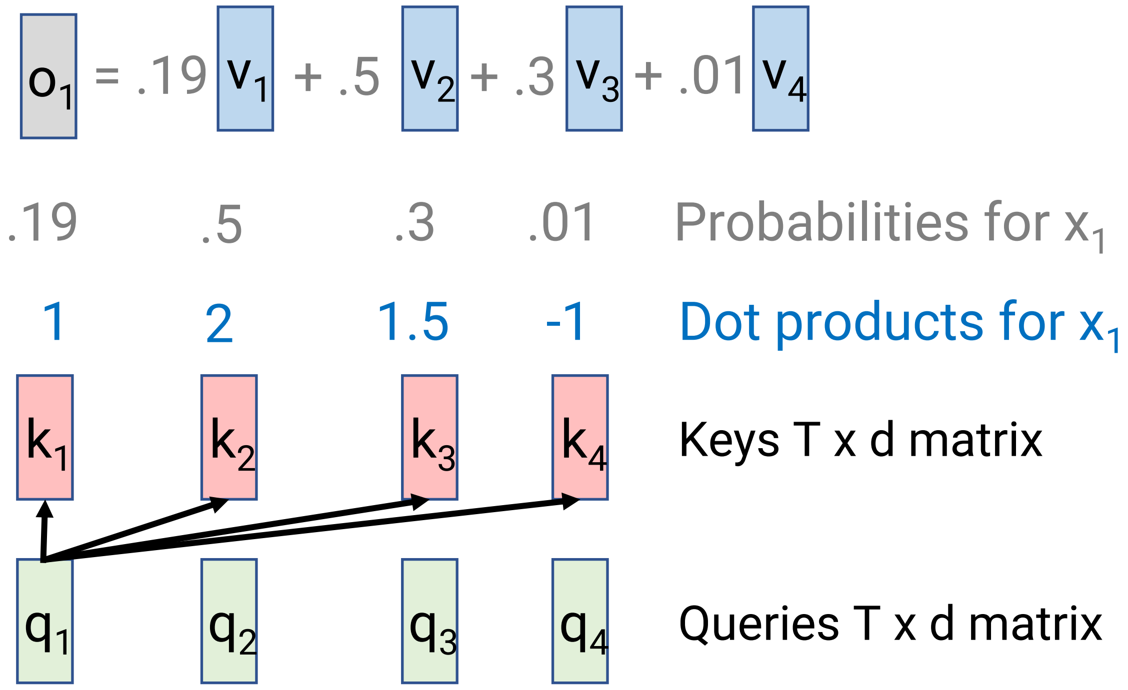



- How to do autoregressive language modeling?
- Training time: Masked attention trick
  - Recall: Attention computes Q x $K^T$ (T x T matrix), then does softmax
  - But if generating autoregressively, time t can only attend to times 1 through t
  - Solution: Overwrite Q x $K^T$ to be $-\infty$ when query index < key index
  - Still efficient/parallelizable

# Bells and whistles

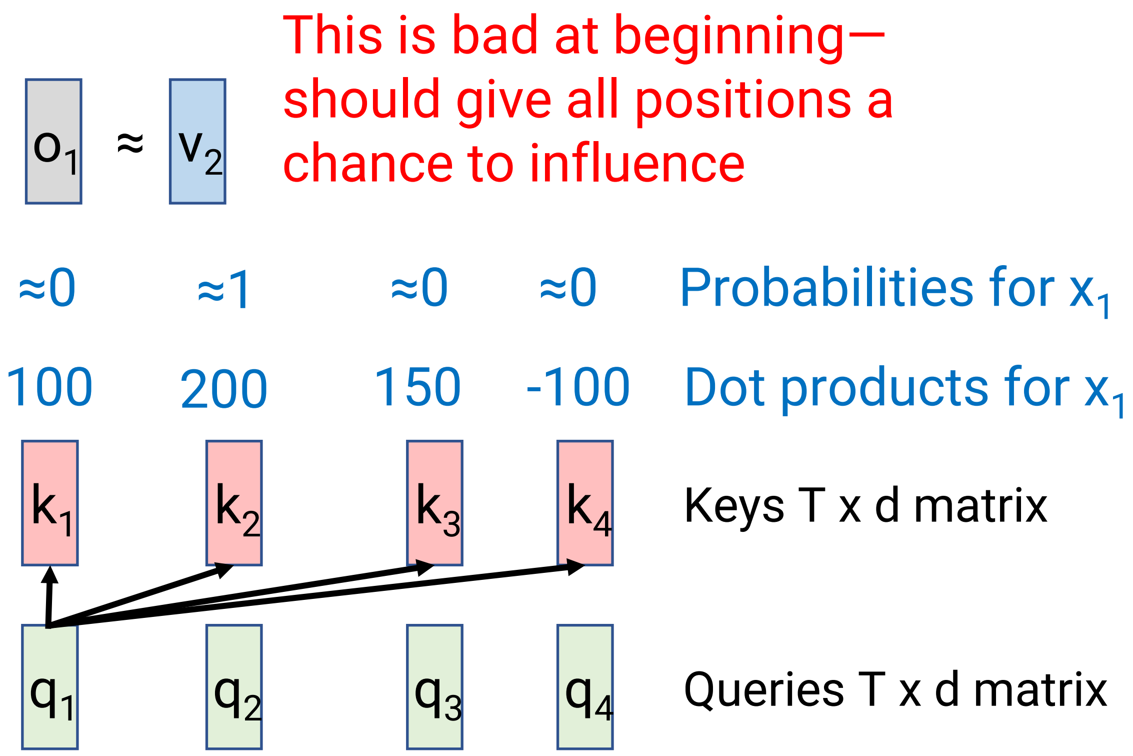
- Attention: Scaled dot products
- Residual connections
- Layer Norm
- Tokenization: Byte Pair Encoding

# 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

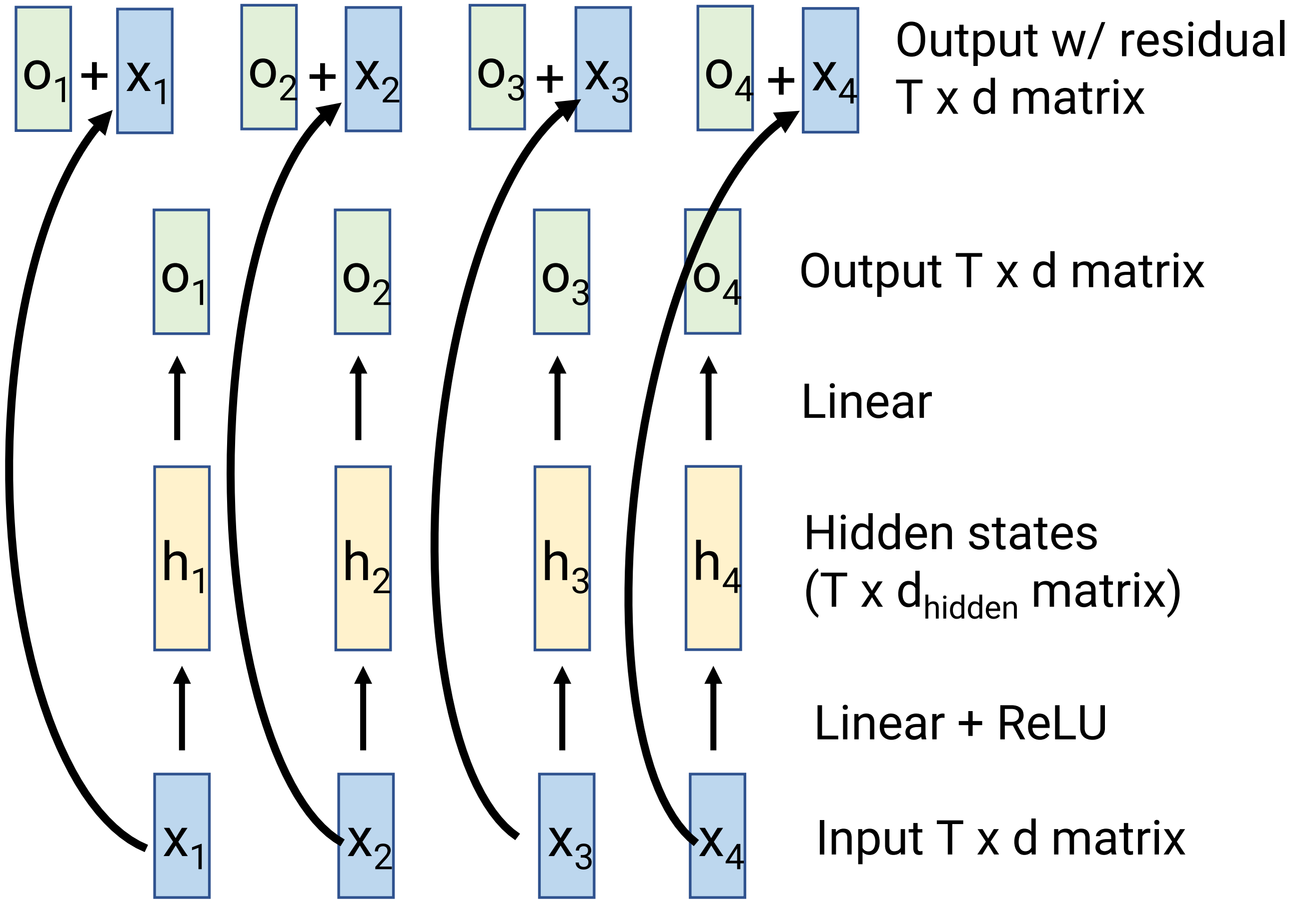
# Scaled dot product attention




- 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

# Residual Connections & Layer Norm

- 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
- Then, we add "Layer Normalization" to prevent very big or very small values

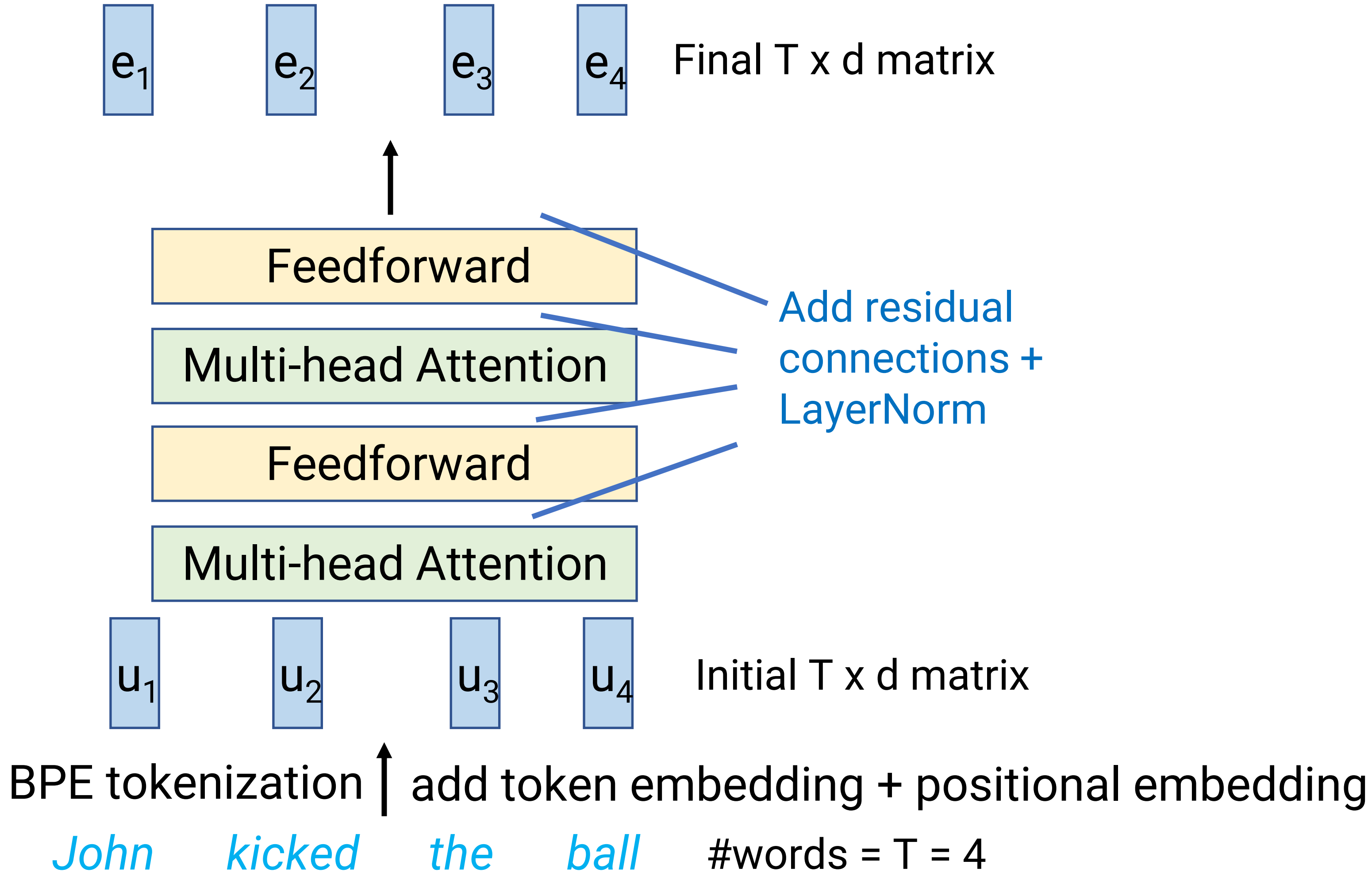


# Byte Pair Encoding

- 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 |

# Putting it all together



# Announcements

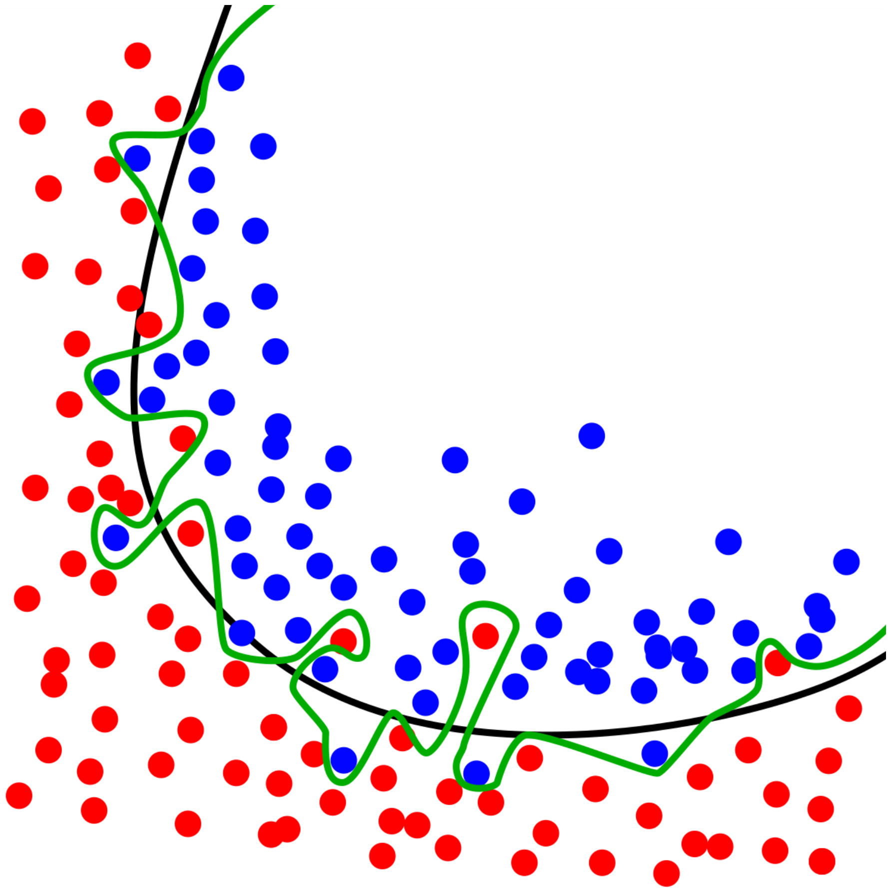
- Section tomorrow: Midterm-related topic review
- HW2 due tonight
- Midterm in-class March 9

# Outline

- Transformers (“Attention is all you need”)
  - Replacing recurrence with attention
  - All the bells and whistles
- Pretraining
  - Frozen features (ImageNet)
  - Fine-tuning (Masked language modeling)

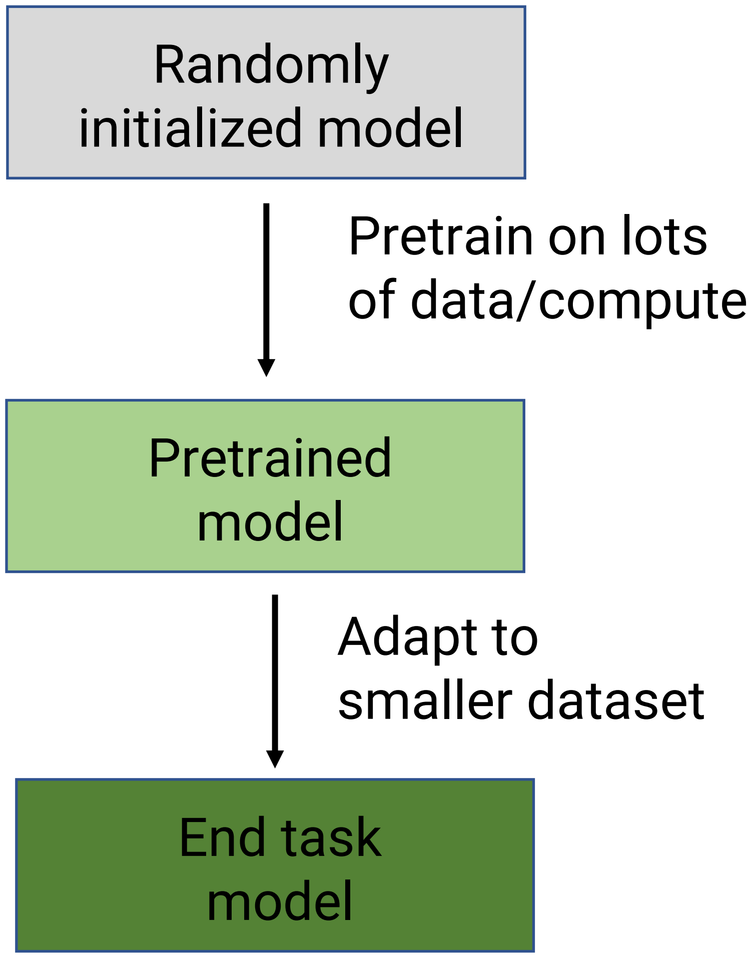
# Neural Networks and Scale

- Neural networks are very expressive, but have tons of parameters
  - Very easy to overfit a small training dataset
- Traditionally, neural networks were viewed as flexible but very **"sample-inefficient"**: they need many training examples to be good

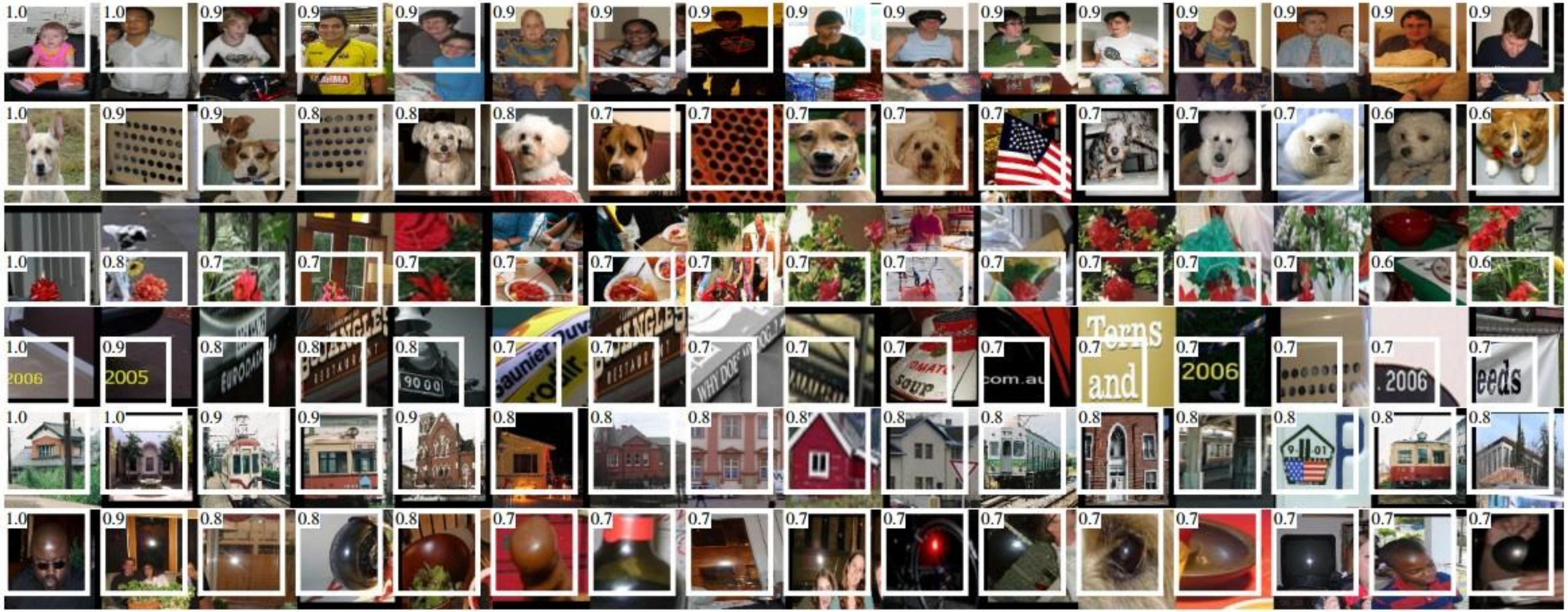


# Pretraining

- Neural networks learn to extract features useful for some training task
  - The more data you have, the more successful this will be
- If your training task is very general, these features may also be useful for other tasks!
- Hence: **Pretraining**
  - First pre-train your model on one task with a lot of data
  - Then use model's features for a task with less data
  - Upends the conventional wisdom: You can use neural networks with small datasets now, if they were pretrained appropriately!

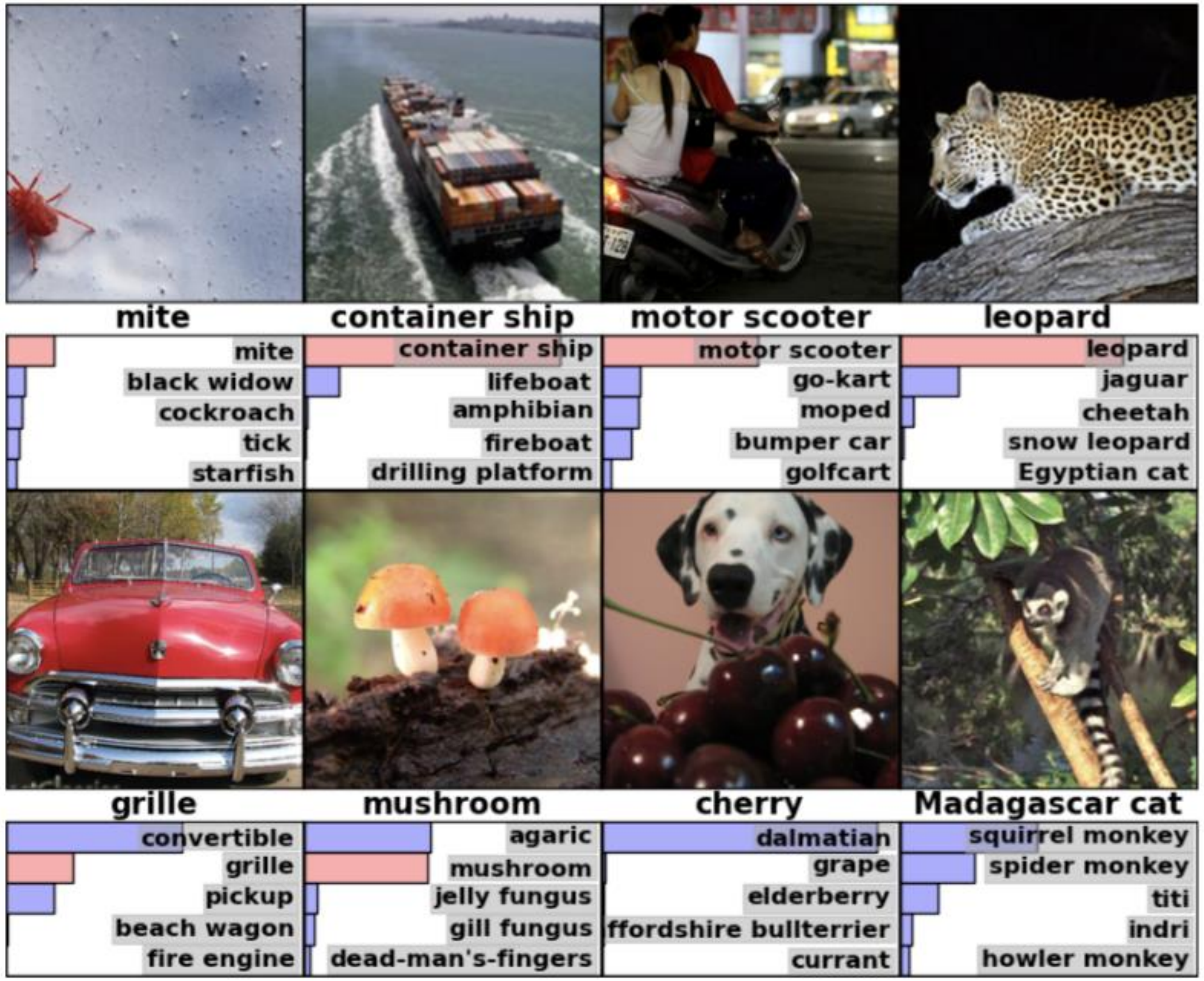


# ImageNet Features



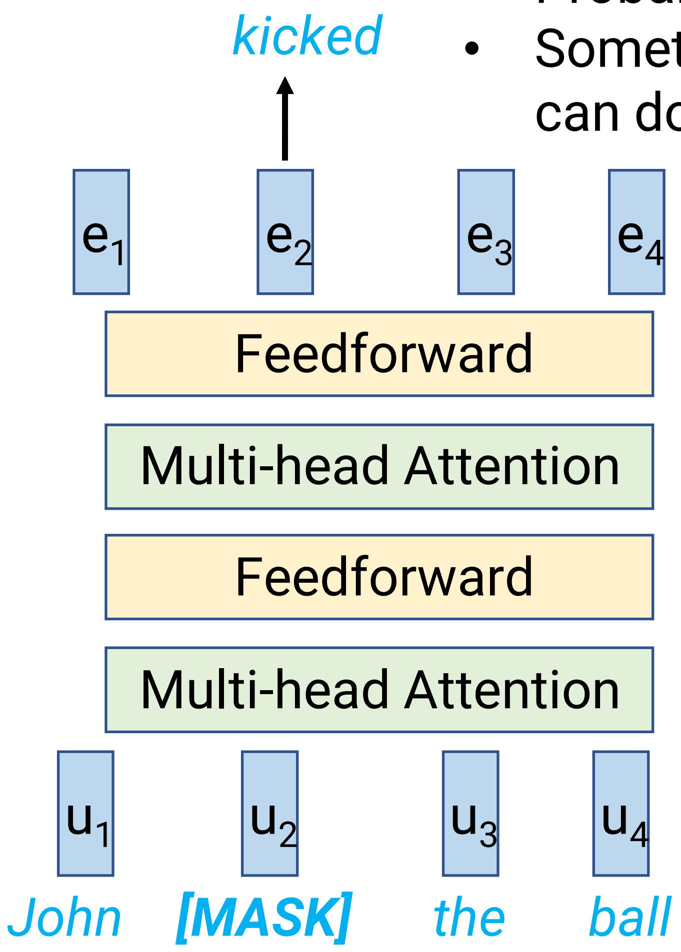

Features learned by AlexNet trained on ImageNet

# ImageNet Features




- ImageNet dataset: **14M** images, 1000-way classification
- Most applications don't have this much data
- **But the same features are still useful**
- Using "frozen" pretrained features
  - Get a (small) dataset for your task
  - Generate features from ImageNet-trained model on this data
  - Train linear classifier (or shallow neural network) using ImageNet features
  - "Frozen" because the original model is not trained further

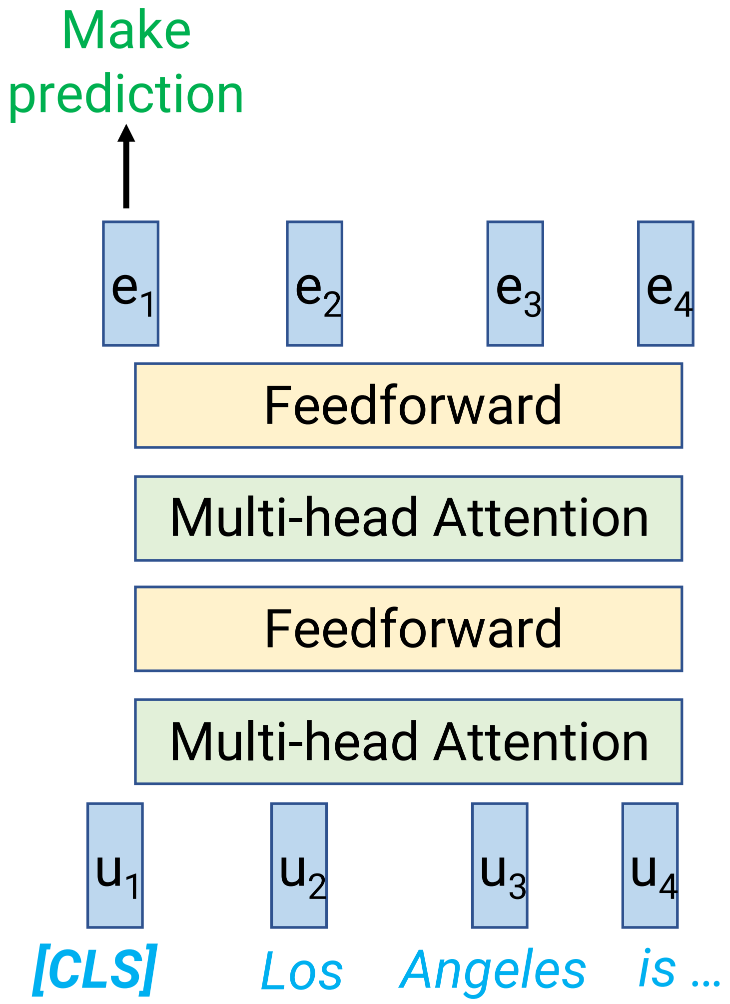
# Masked Language Modeling (MLM)




- Probably a verb
- Something a person can do to a ball

- MLM: Randomly mask some words, train model to predict what's missing
  - Doing this well requires understanding grammar, world knowledge, etc.
  - To get training data for this task, just need to find any text and randomly delete words
  - Thus: Crawl internet for text data
- Transformers are good fit due to scalability
  - Large matrix multiplications are highly optimized on GPUs/TPUs
  - Don't need lots of operations happening in series (like RNNs)
- Most famous example: BERT

# Fine-tuning




- Initialize parameters with BERT
  - BERT was trained to expect every input to start with a special token called [CLS]
- Add parameters that take in the output at the [CLS] position and make prediction
- Keep training all parameters ("fine-tune") on the new task
- Point: BERT provides very good initialization for SGD

# What about ChatGPT???

- ChatGPT appears to be a fine-tuned language model
  - Pretrained on autoregressive language modeling
  - Then fine-tuned with a method called RLHF (reinforcement learning from human feedback)
  - We’ll return to this when we talk about reinforcement learning!

# Conclusion

- Transformer architecture
  - Get rid of recurrent connections
  - Instead, all “communication” between words in sequence is handled by attention
- Pretraining
  - First train on large labeled or unlabeled datasets
  - Features learned are useful for other tasks with less data