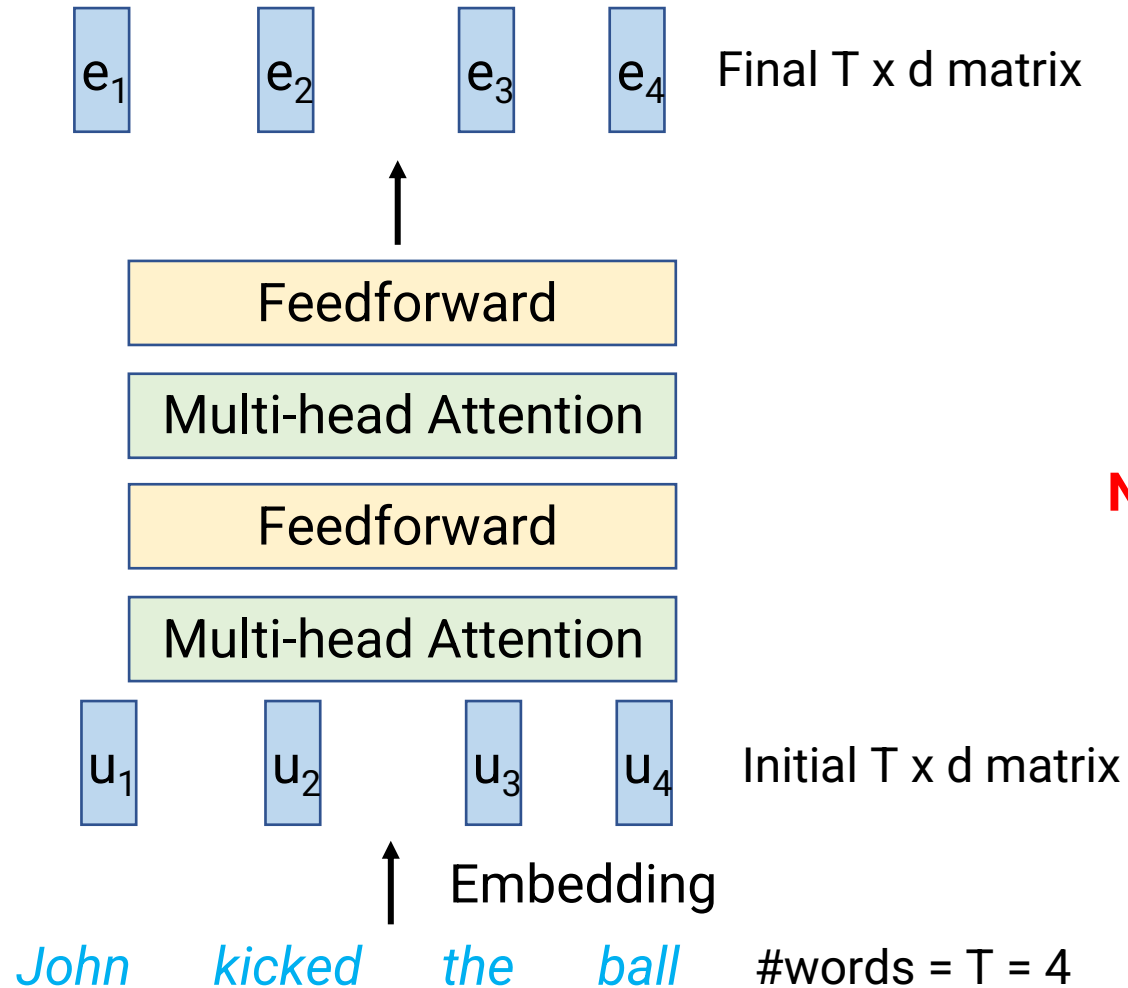


Transformers II, Pretraining

Robin Jia
USC CSCI 467, Spring 2025
March 27, 2025

Review: Transformer at a high level

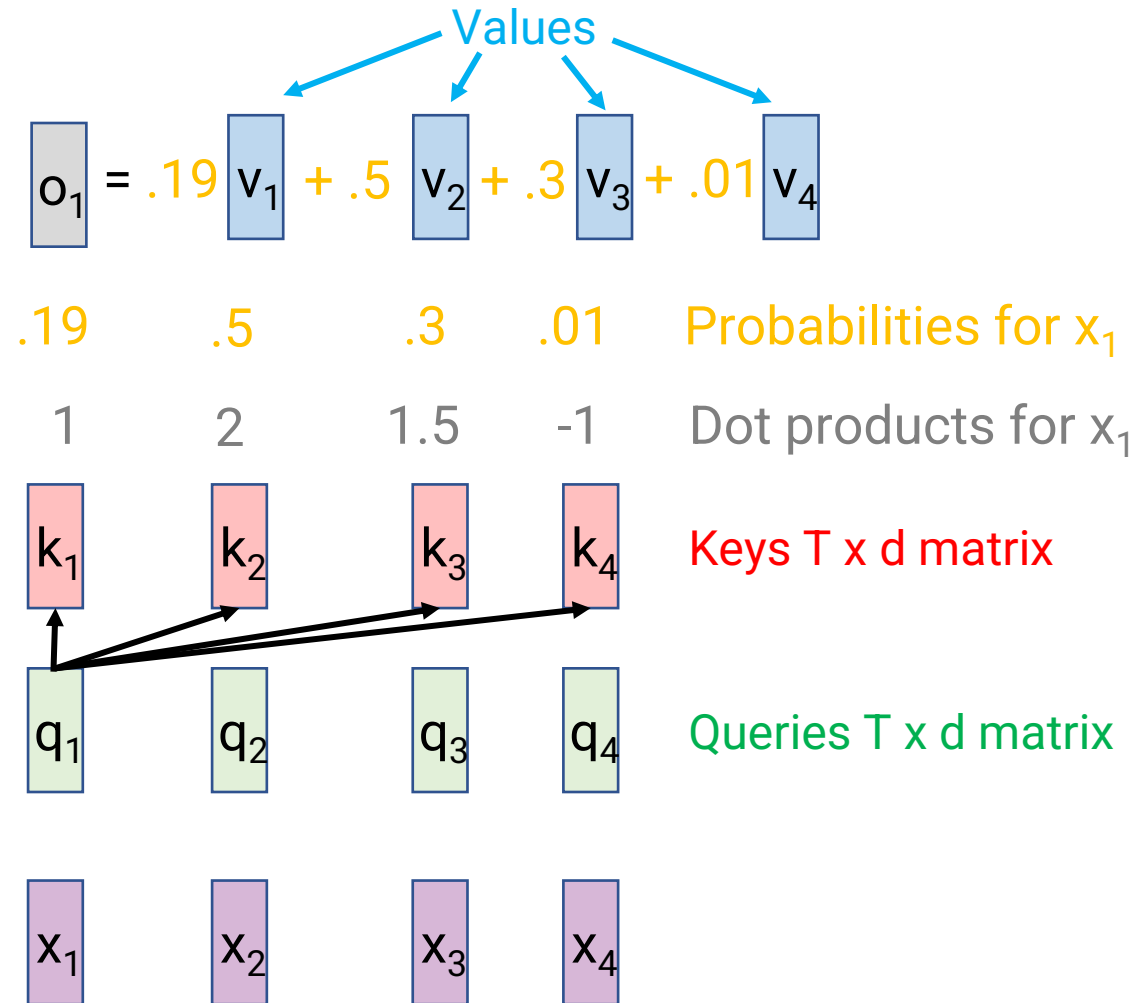


- One transformer consists of
 - Initial embeddings for each word of size d
 - Let $T = \text{\#words}$, so initially we have a $T \times d$ matrix
 - Alternating layers of
 - **"Multi-headed" attention layer**
 - Feedforward layer
 - Both take in $T \times d$ matrix and output a new $T \times d$ matrix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

New!

Familiar

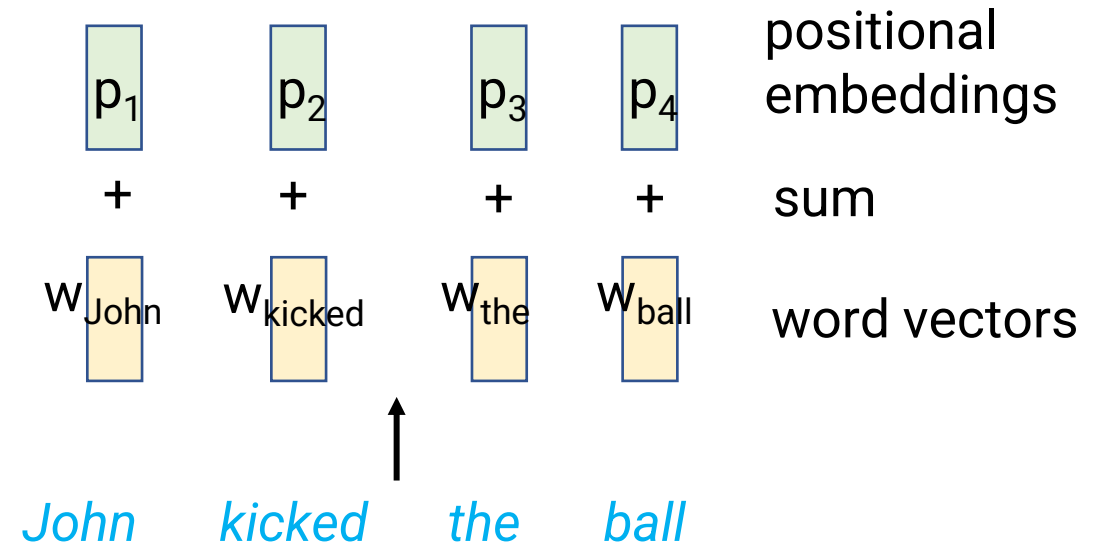
Review: Multi-headed Attention



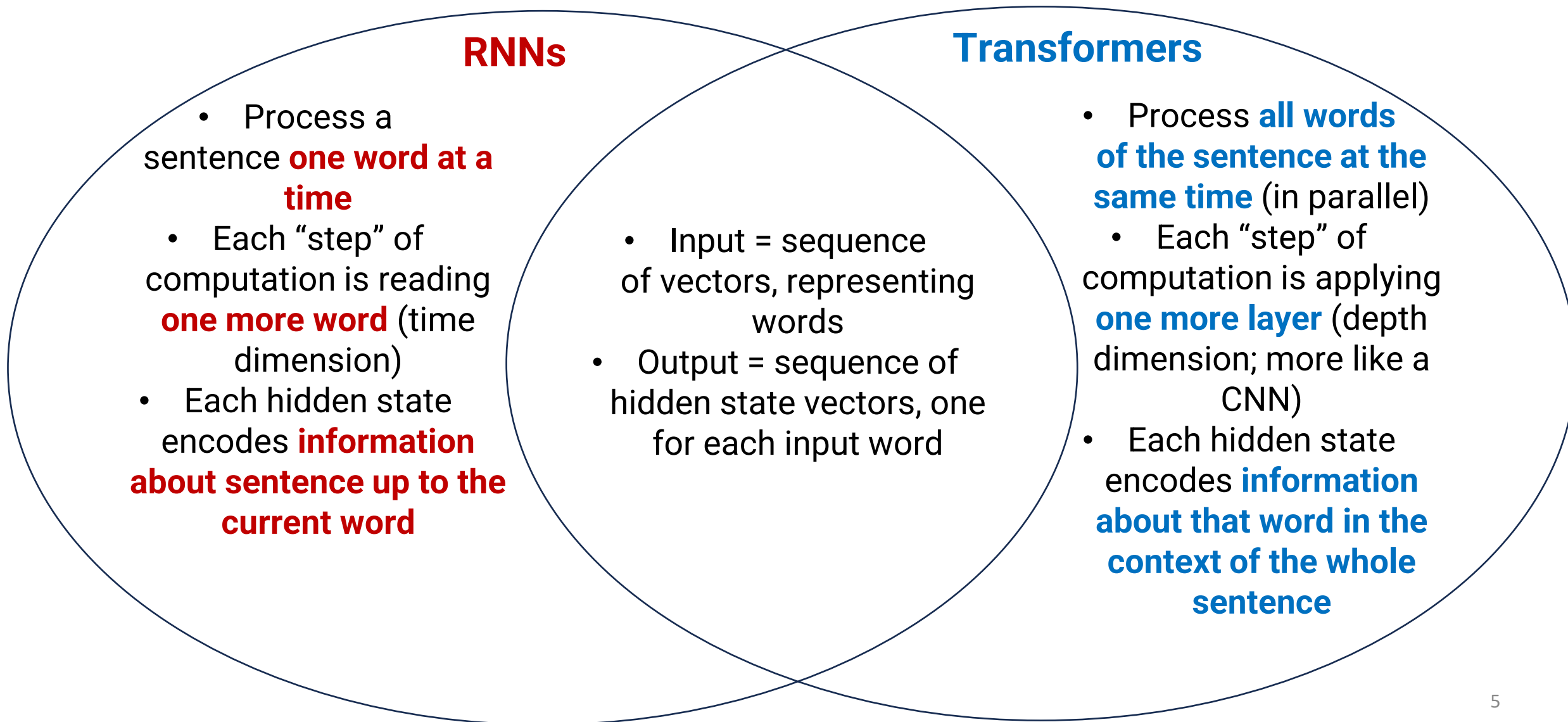
- Input: T vectors x_1, \dots, x_T each of dimension d
- Apply 3 separate linear layers to each x_t :
 - Query vectors $q_t = W^Q * x_t$
 - Keys vectors $k_t = W^K * x_t$
 - Value vectors $v_t = W^V * x_t$
- To compute output o_t :
 - Dot product q_t with each key vector k_i
 - Apply softmax to get probabilities p_i
 - Compute $o_t = \sum_{i=1}^T p_i * v_i$
- Have n heads with n different sets of parameters, then concatenate results
 - Choose $d_{\text{attn}} = d/n$ so output is also dimension d
- Parameters W^Q, W^K, W^V for each head must be learned by gradient descent

Review: Initial 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



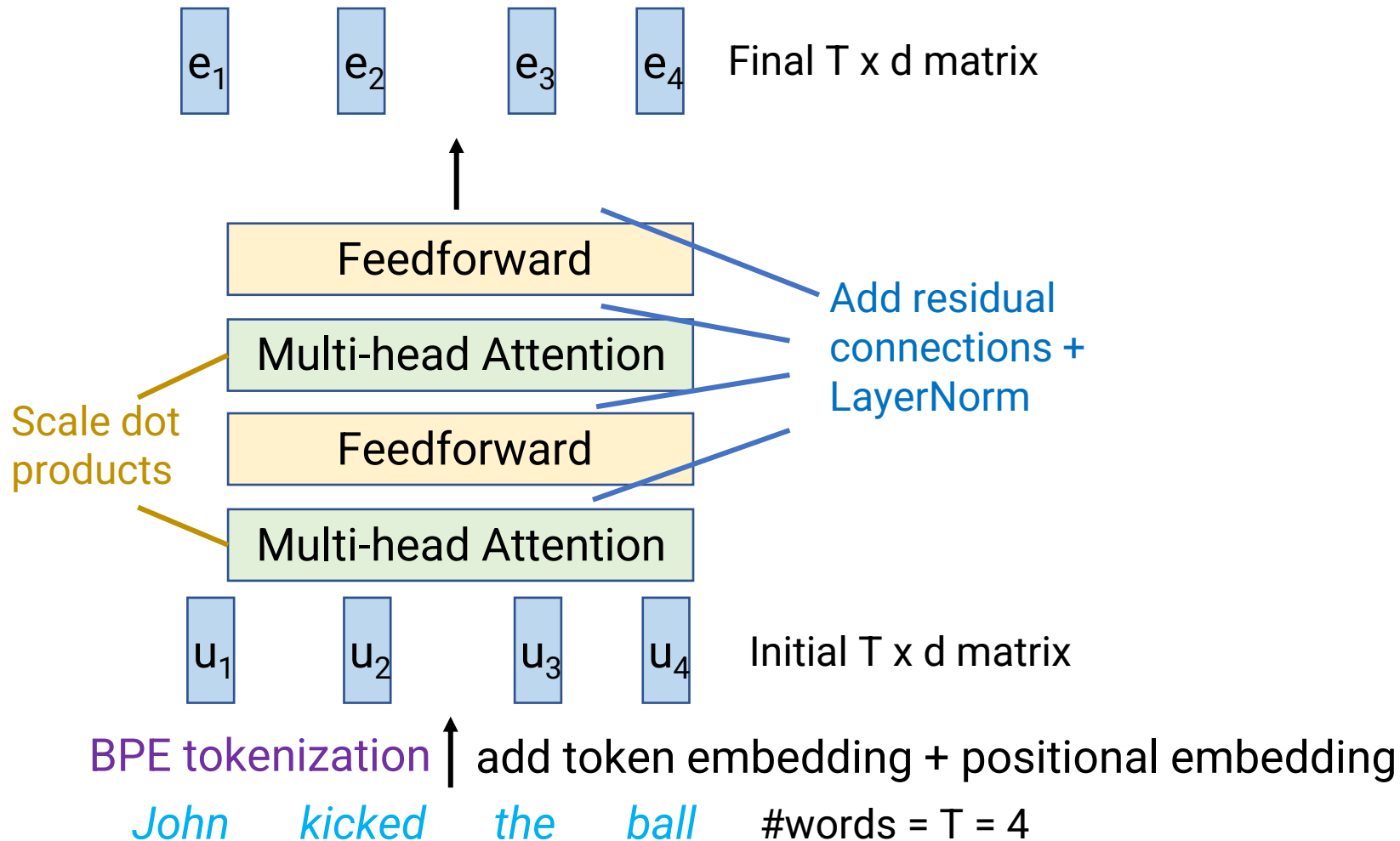
RNNs vs. Transformers (Encoders)



Today's Plan

- Transformers in full detail
- Pre-training
- Transformer decoders
- Vision Transformers

The Full Transformer



Full Transformer also includes bells and whistles:

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

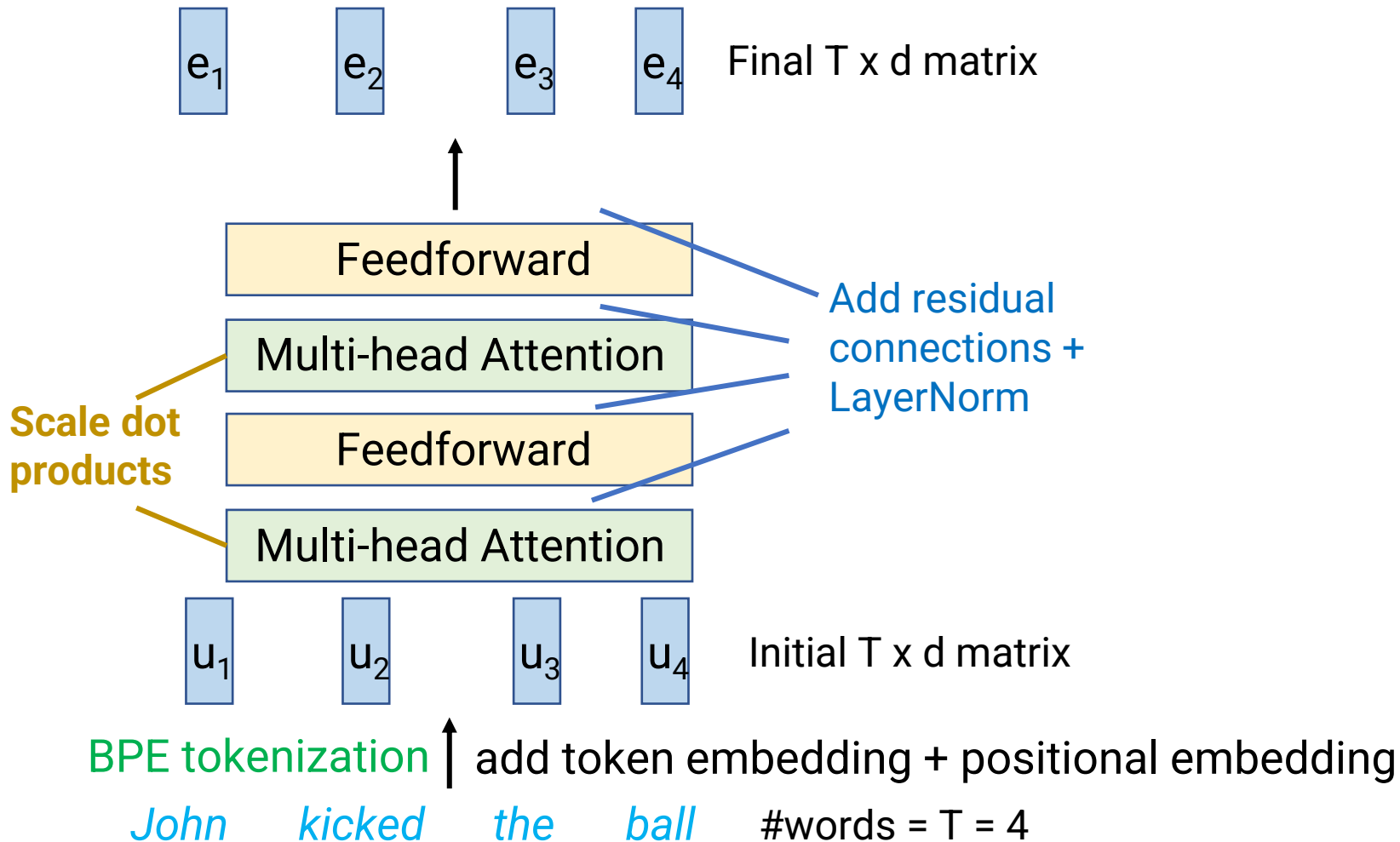
Byte Pair Encoding

- 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

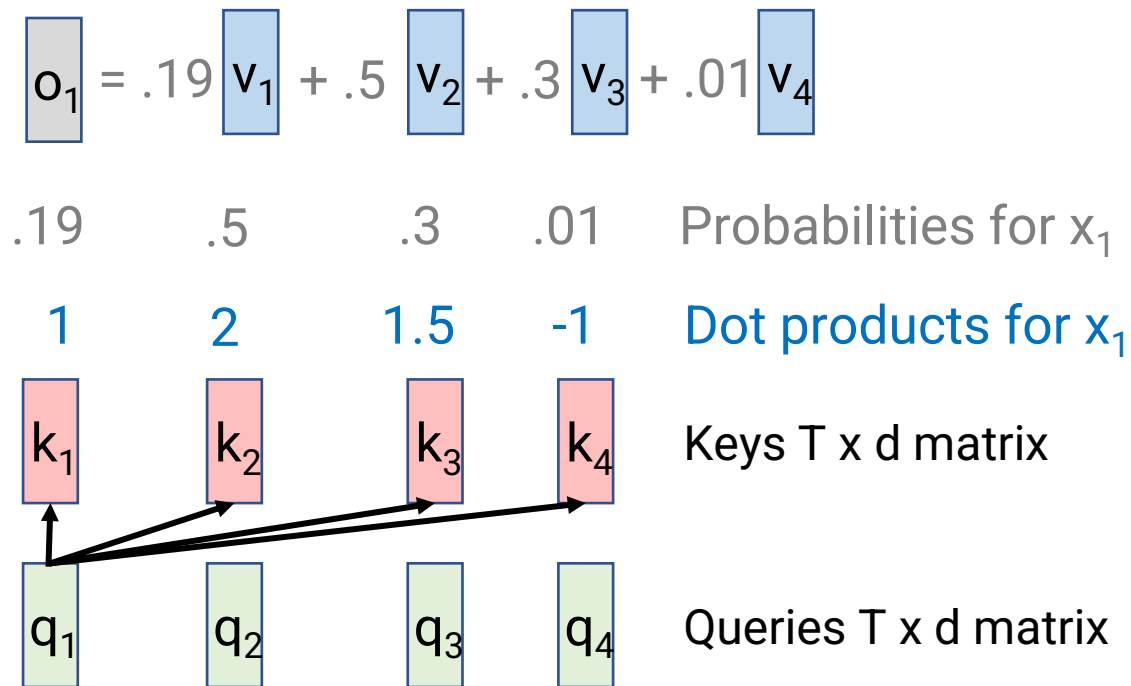
The Full Transformer



Full Transformer also includes bells and whistles:

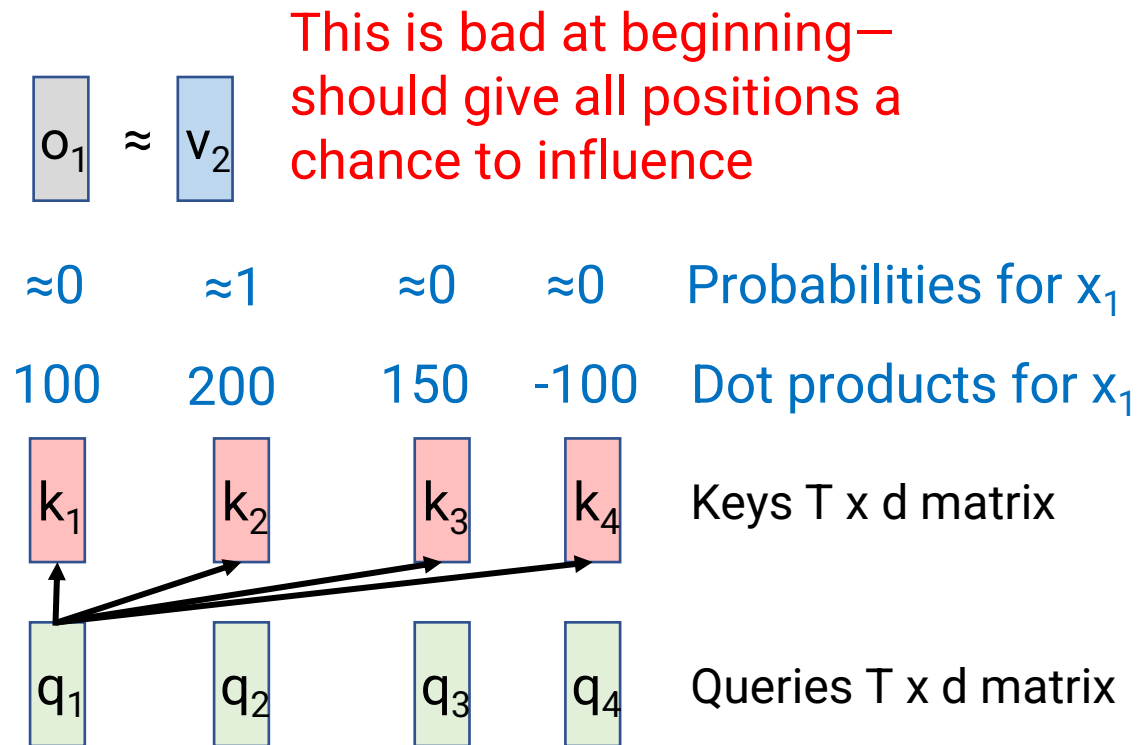
- Byte pair encoding
- Scaled dot product attention
- Residual connections between layers
- LayerNorm

Scaled dot product attention



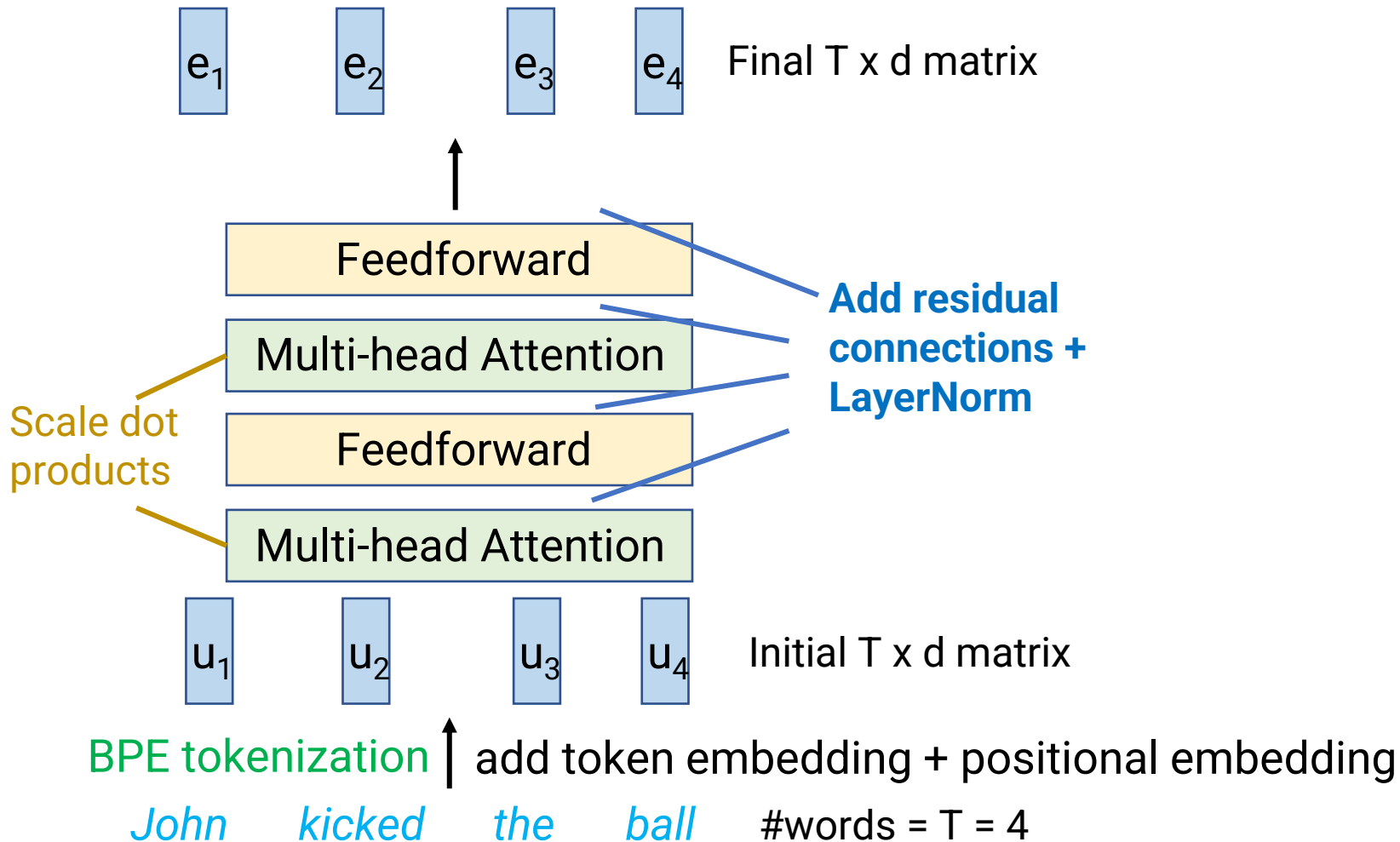
- Earlier I said, “Dot product q_1 with $[k_1, \dots, k_T]$ ”
- Actually, you take dot product and then **divide by** $\sqrt{d_{attn}}$
- Why?
 - If d large, dot product between random vectors will be large
 - This makes probabilities close to 0/1
 - Scaling dot products down encourages more even attention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

Scaled dot product attention



- Earlier I said, “Dot product q_1 with $[k_1, \dots, k_T]$ ”
- 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

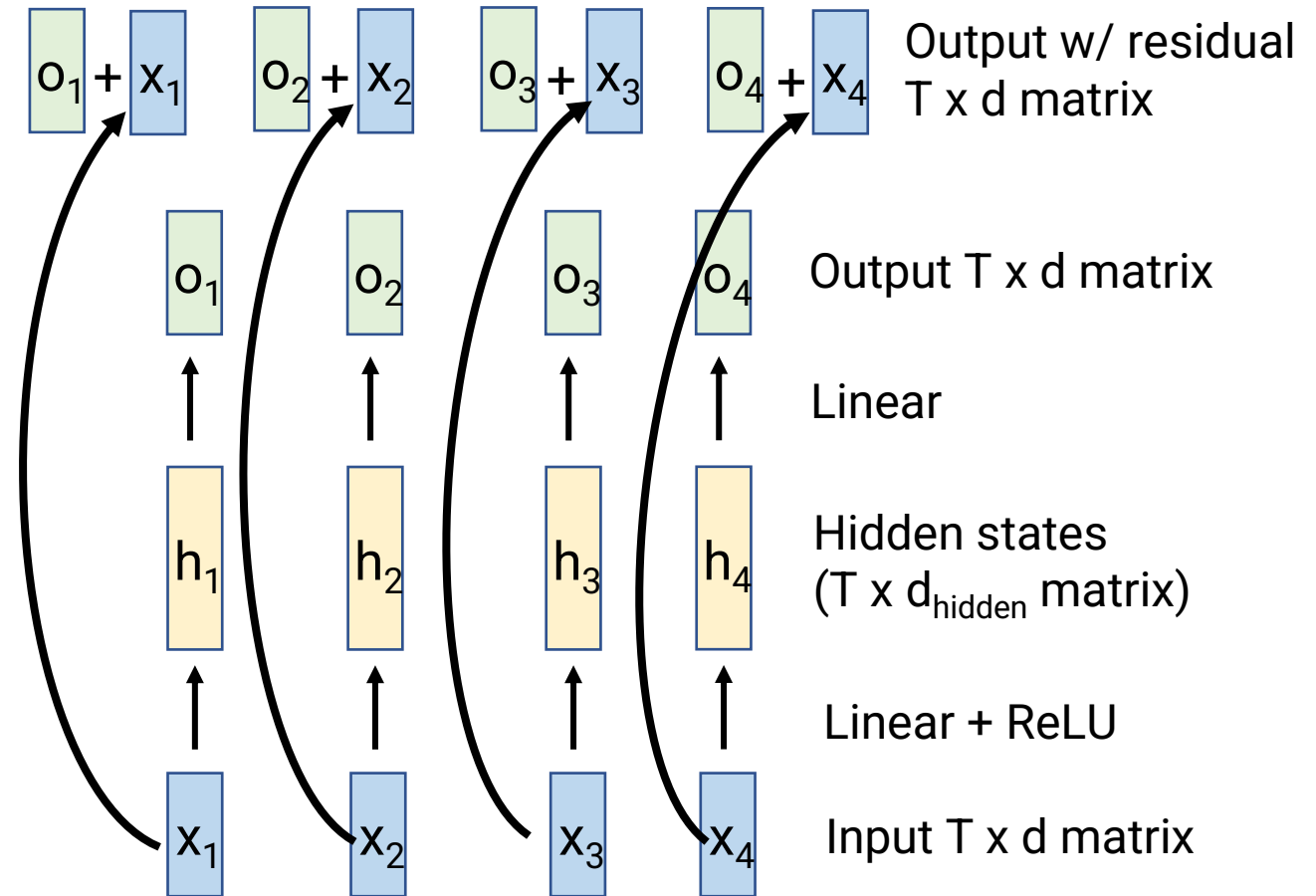


Full Transformer also includes bells and whistles:

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

Residual Connections

- Feedforward and multi-headed attention layers
 - Take in $T \times d$ matrix X
 - Output $T \times d$ 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

$$x = [100, 200, 100, 0]$$

- 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**

$$\mu = 100$$

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

$$\sigma^2 = \frac{1}{4} * (0^2 + 100^2 + 0^2 + 100^2) = 5000$$

$$y = a \odot \left(\frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} \right) + b$$

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

Normalized $x =$

$$[0, 100, 0, -100] / \sqrt{5000} \\ = [0, 1.4, 0, -1.4] \text{ (If } \epsilon \approx 0 \text{)}$$

- Parameters

Normalized x

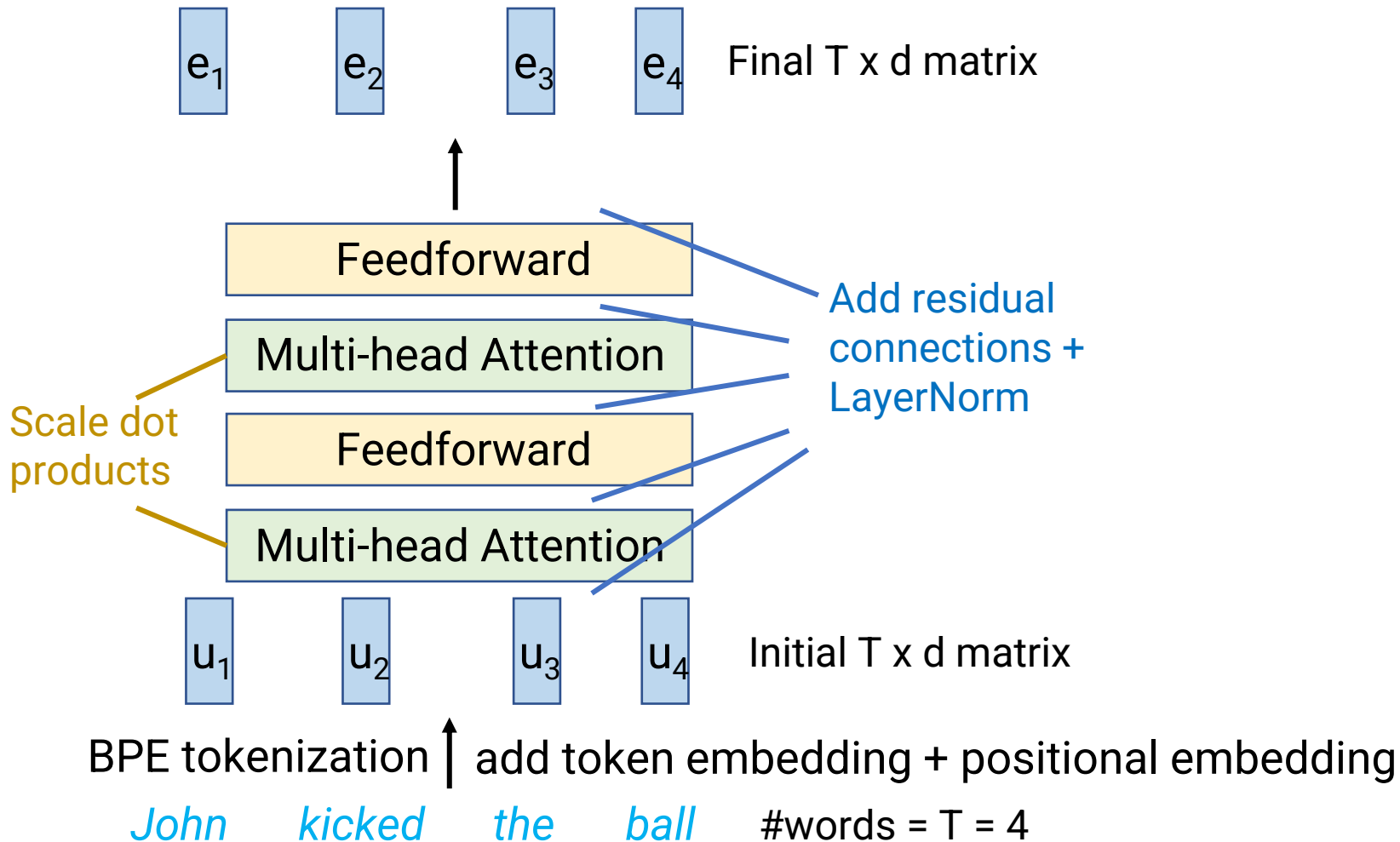
- a & b are vector parameters, let model learn good **scale**/**shift** per dimension
 - Without these, all vectors forced to have mean=0, variance=1
- ϵ is hyperparameter: Some small number to prevent division by 0

$$\text{Output} = [b_1, \\ 1.4a_2 + b_2, \\ b_3, \\ -1.4a_4 + b_4]$$

LayerNorm in Transformers

- Add Layer Normalization layer *before* every feedforward & multi-headed attention layer
 - Input: vectors x_1, \dots, x_T
 - Compute μ and σ^2 for each vector
 - Normalize each vector
 - Use the same **a** and **b** to scale/shift each vector
 - Output of each layer is $x + \text{Layer}(\text{LayerNorm}(x))$
- Why? Stabilizes optimization by avoiding very large values

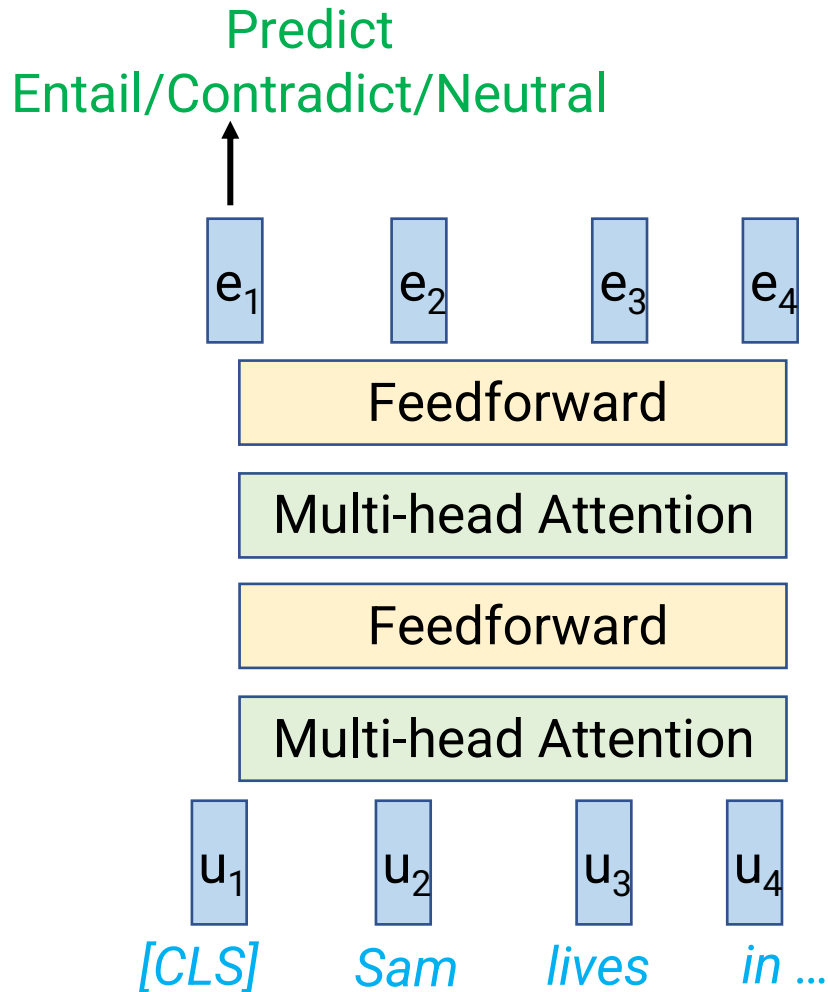
The Full Transformer



Full Transformer also includes bells and whistles:

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

Training a Transformer



- Example task: Natural Language Inference
 - Input: 2 sentences, A and B
 - Output: 3-way classification: A entails B, A contradicts B, neither
 - Performing this task well requires understanding meaning of sentences + logical relationships
- Input to Transformer: Concatenate special “CLS” token and 2 sentences together
- Output: Use CLS token’s final representation to predict
- Train on labeled data, learn to make good predictions

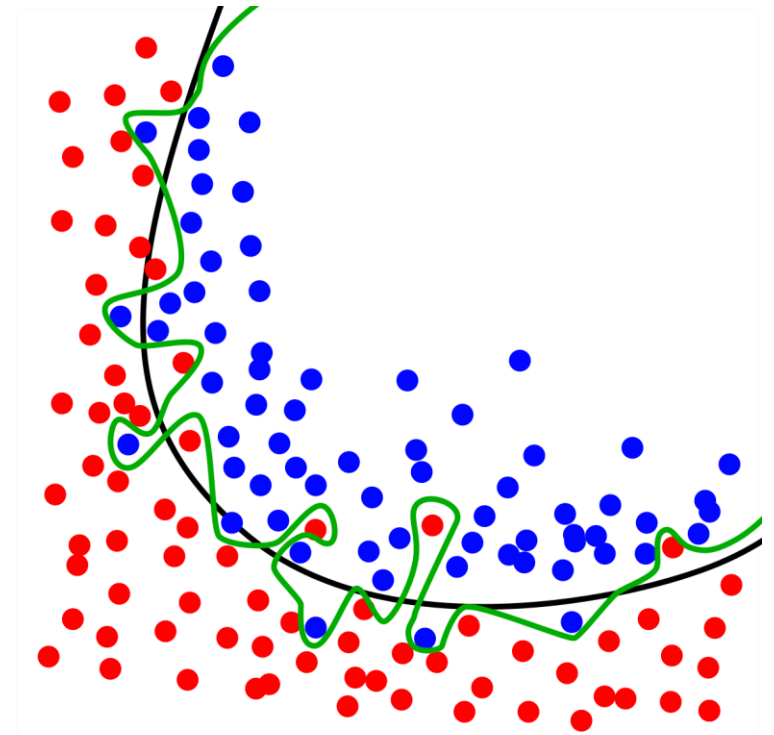
[CLS] Sam lives in Los Angeles. Sam lives in California.

Today's Plan

- Transformers in full detail
- **Pre-training**
- Transformer decoders
- Vision Transformers

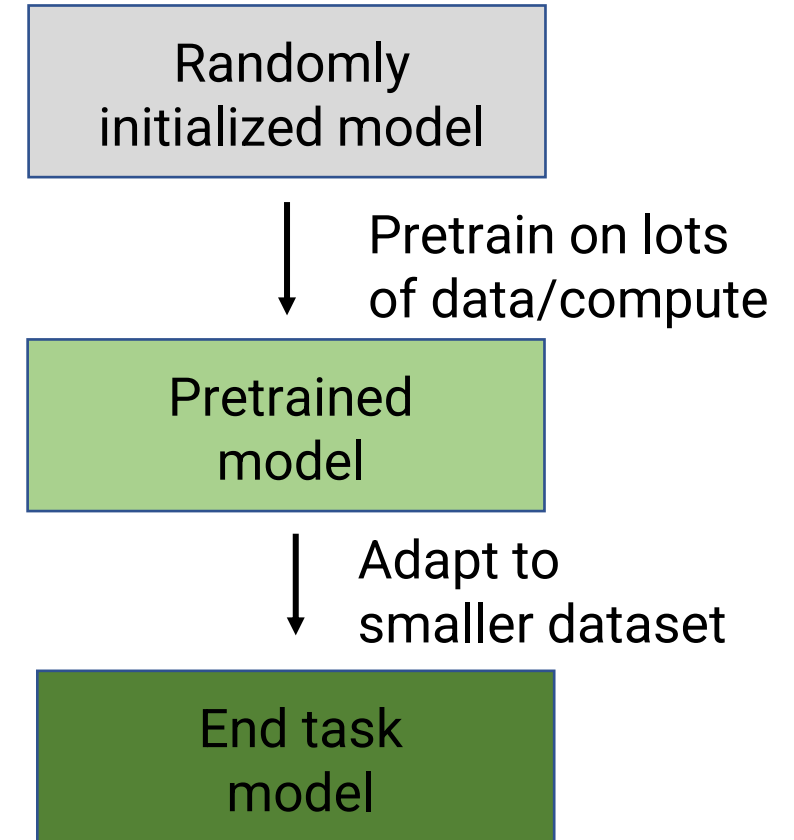
Neural Networks and Scale

- 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
 - Computationally expensive
 - Training at scale often uses GPUs



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



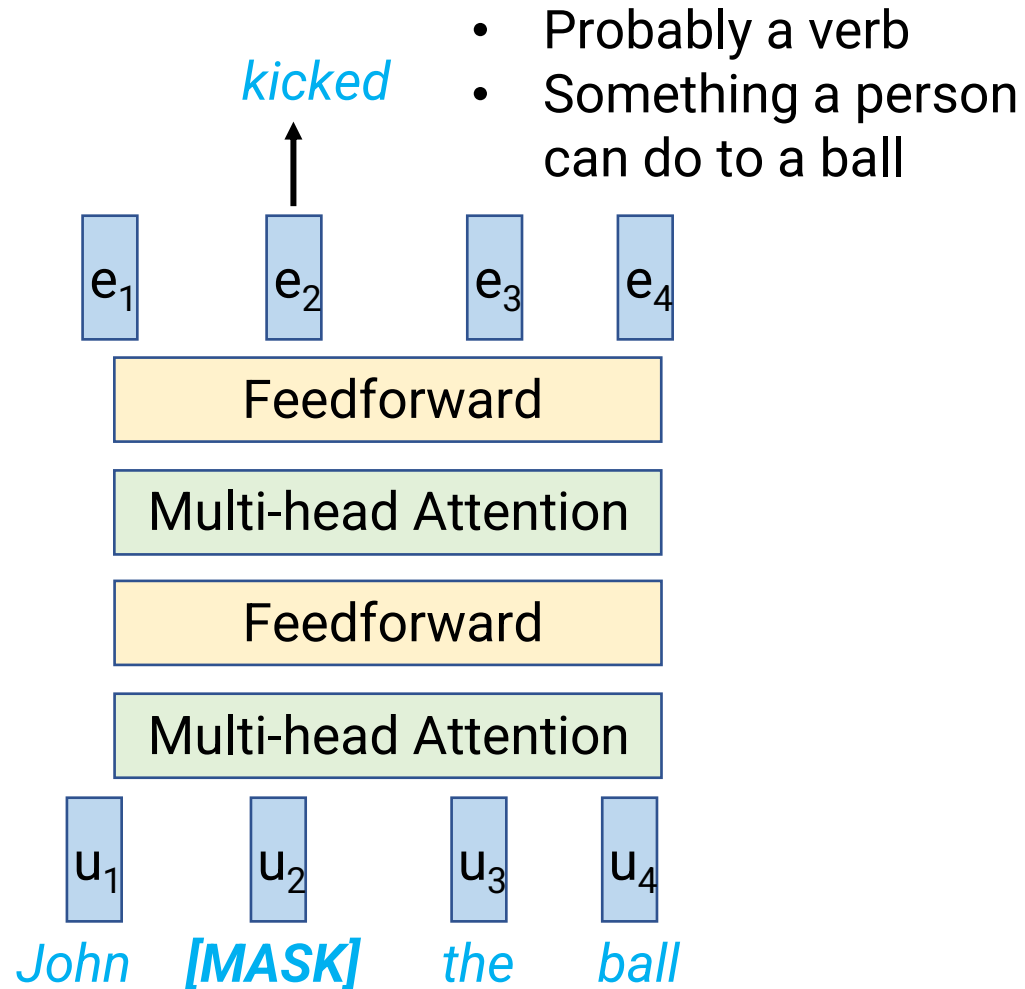
Features learned by AlexNet trained on ImageNet

ImageNet Features



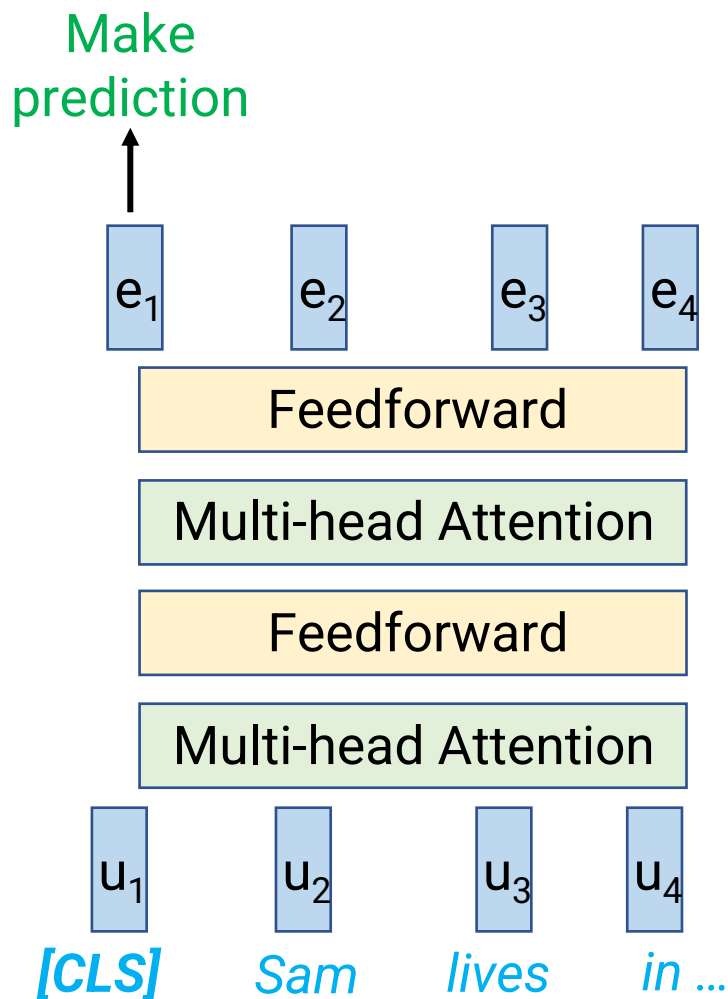
- ImageNet dataset: **14M** images, 1000-way classification
- Most applications don't have this much data
- **But the same features are still useful**
- Using "frozen" pretrained features
 - Get a (small) dataset for your task
 - Generate features from ImageNet-trained model on this data
 - Train linear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

Masked Language Modeling (MLM)



- MLM: Randomly mask some words, train model to predict what's missing
 - Doing this well requires understanding grammar, world knowledge, etc.
 - Get training data just by grabbing 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

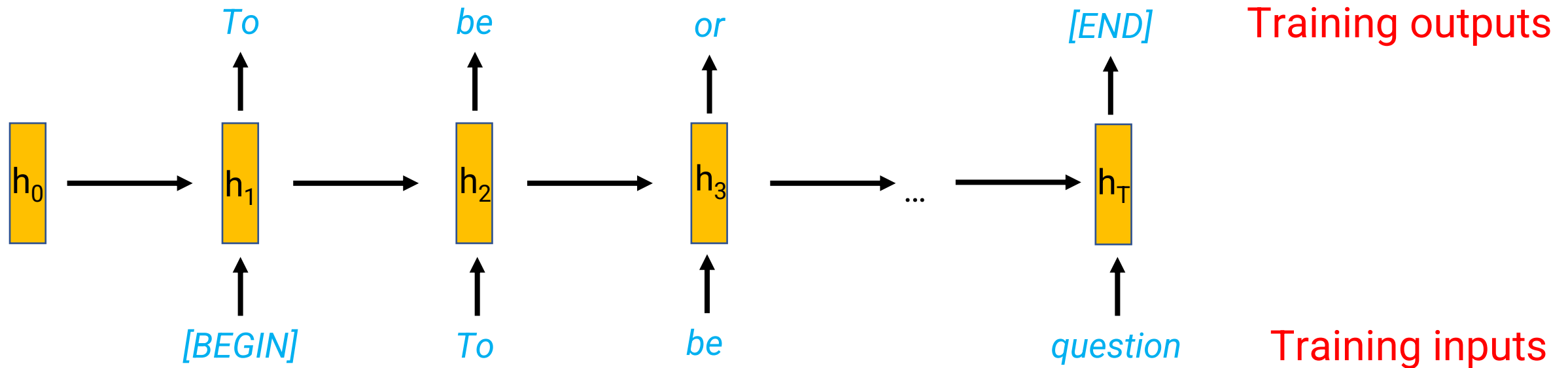


- Initialize parameters with BERT
- Add parameters that take in the output at the [CLS] position and make prediction
- Keep training all parameters (“fine-tune”) on the new task
- Point: BERT provides very good initialization for SGD

Announcements

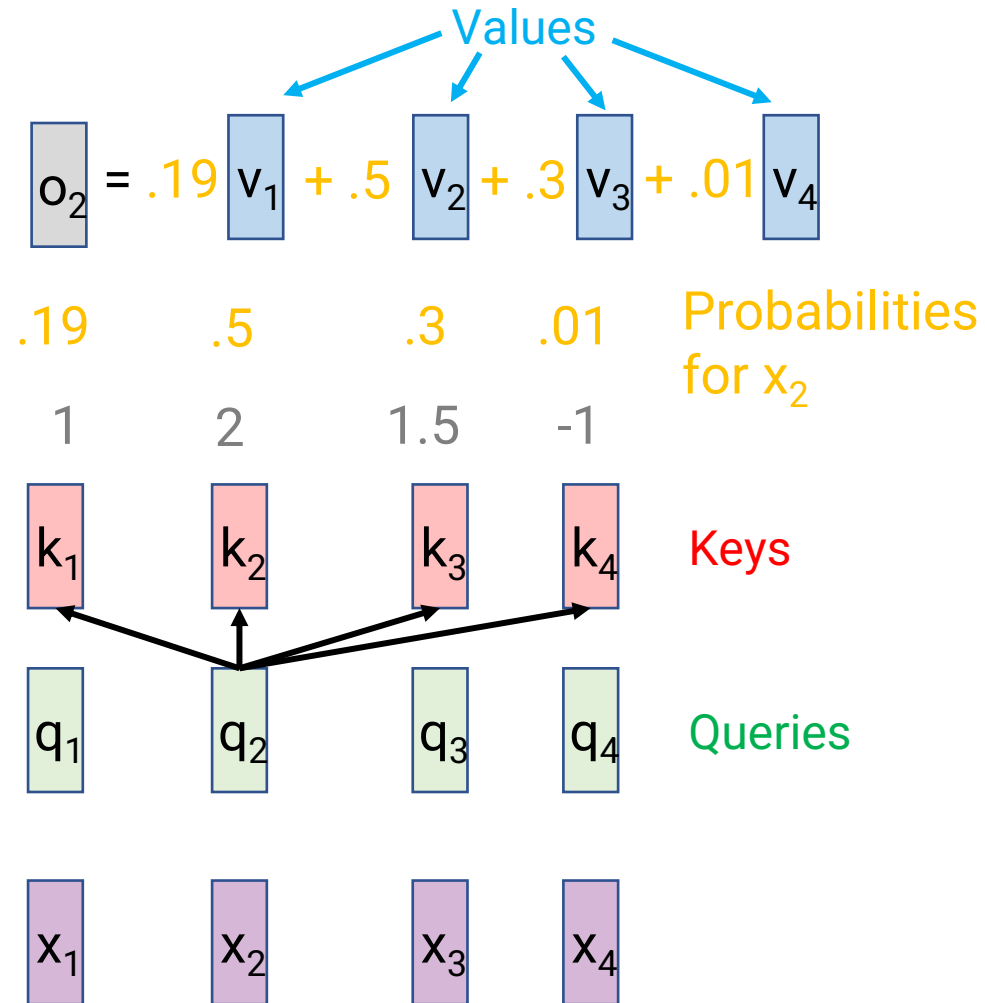
- Project midterm report due Tuesday, April 1
- HW3 released, due Tuesday, April 15
- Tomorrow's section: RNNs/Transformers in pytorch

Review: RNN Decoder Language Models



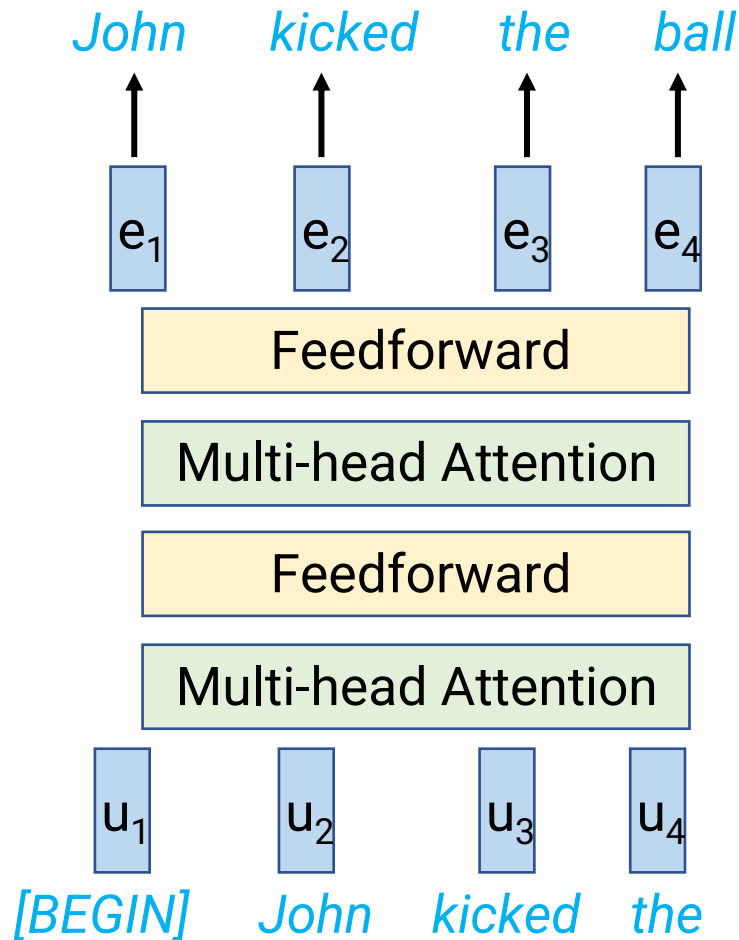
- At each step, predict the next word given current hidden state
- Test time: Model chooses a next word, that gets fed back in
- Training time: Model is fed the human-written words, tries to guess next word at every step
- RNN computations must happen in series at both training and test time
 - Each hidden state depends on the previous hidden state

Transformer autoregressive decoders



- How can we use Transformers to generate text?
- We will still generate words one at a time
- Problem: The Transformer (encoder) processes all words in parallel
 - Word 2 is allowed to attend to words 3, 4...
 - But in a decoder, words 3, 4, ... have not been chosen yet when processing word 2!
- Solution: Use a **variant** of multi-headed attention that **only allows attending to past/current words**
 - Often referred to as “causal masking”: Don’t allow looking into the future

Transformer autoregressive decoders



- Test-time behavior

- At time t , compute hidden states for current token t by attending to positions 1 through t
- Each timestep only processes the newest token, attends to previously generated hidden states
- Happens in series

Queries

[BEGIN]
John
kicked
the

[BEGIN] *John* *kicked* *the*

Keys

Transformer autoregressive decoders

- When training a decoder, it has to be “used to” only attending to past/current tokens
- Training time: Masked attention implementation trick
 - Recall: Attention computes $Q \times K^T$ ($T \times T$ matrix), then does softmax
 - But if generating autoregressively, time t can only attend to times 1 through t
 - Solution: Overwrite $Q \times K^T$ to be $-\infty$ when query index $<$ key index
 - **All timesteps happen in parallel**

Queries	[BEGIN]	10	-2	6	3
	John	0	7	2	-4
	kicked	-3	4	5	-8
	the	2	1	7	6
		[BEGIN]	John	kicked	the
		Keys			

Transformer autoregressive decoders

- When training a decoder, it has to be “used to” only attending to past/current tokens
- Training time: Masked attention implementation trick
 - Recall: Attention computes $Q \times K^T$ ($T \times T$ matrix), then does softmax
 - But if generating autoregressively, time t can only attend to times 1 through t
 - Solution: Overwrite $Q \times K^T$ to be $-\infty$ when query index $<$ key index
 - **All timesteps happen in parallel**

Queries	[BEGIN]	10	-2	6	3
	John	0	7	2	-4
	kicked	-3	4	5	-8
	the	2	1	7	6
		[BEGIN]	John	kicked	the
		Keys			

Transformer autoregressive decoders

- When training a decoder, it has to be “used to” only attending to past/current tokens
- Training time: Masked attention implementation trick
 - Recall: Attention computes $Q \times K^T$ ($T \times T$ matrix), then does softmax
 - But if generating autoregressively, time t can only attend to times 1 through t
 - Solution: Overwrite $Q \times K^T$ to be $-\infty$ when query index $<$ key index
 - **All timesteps happen in parallel**

Queries	[BEGIN]	10	$-\infty$	$-\infty$	$-\infty$
	John	0	7	$-\infty$	$-\infty$
	kicked	-3	4	5	$-\infty$
	the	2	1	7	6
		[BEGIN]	John	kicked	the
		Keys			

Transformer autoregressive decoders

- When training a decoder, it has to be “used to” only attending to past/current tokens
- Training time: Masked attention implementation trick
 - Recall: Attention computes $Q \times K^T$ ($T \times T$ matrix), then does softmax
 - But if generating autoregressively, time t can only attend to times 1 through t
 - Solution: Overwrite $Q \times K^T$ to be $-\infty$ when query index $<$ key index
 - **All timesteps happen in parallel**

Queries	[BEGIN]	1.0	0	0	0
	John	.001	.999	0	0
	kicked	.001	.356	.643	0
	the	.030	.007	.591	.372
		[BEGIN]	John	kicked	the
		Keys			

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!

Today's Plan

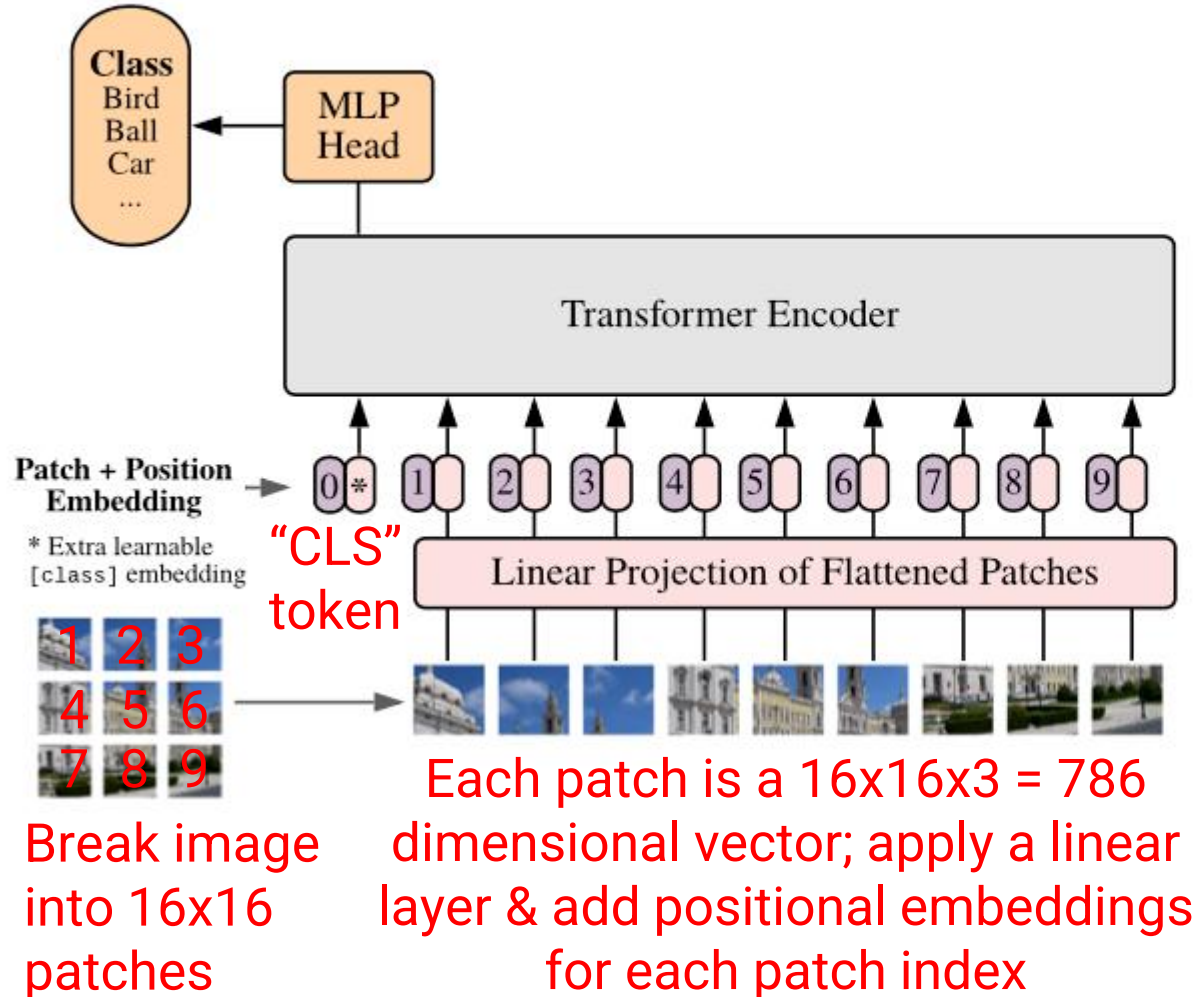
- Transformers in full detail
- Pre-training
- Transformer decoders
- **Vision Transformers**

Vision Transformers

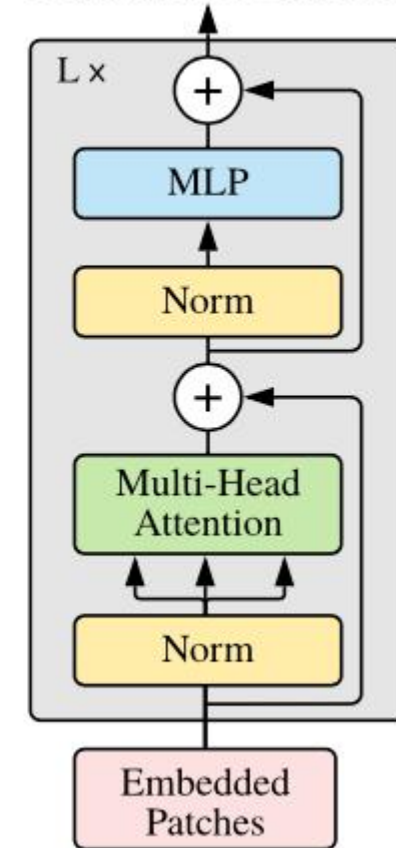
- Transformers paper came out in 2017
- By 2020, they were widely used in NLP
- Computer vision researchers: What if they're also good for images?

Vision Transformer

Vision Transformer (ViT)



Transformer Encoder

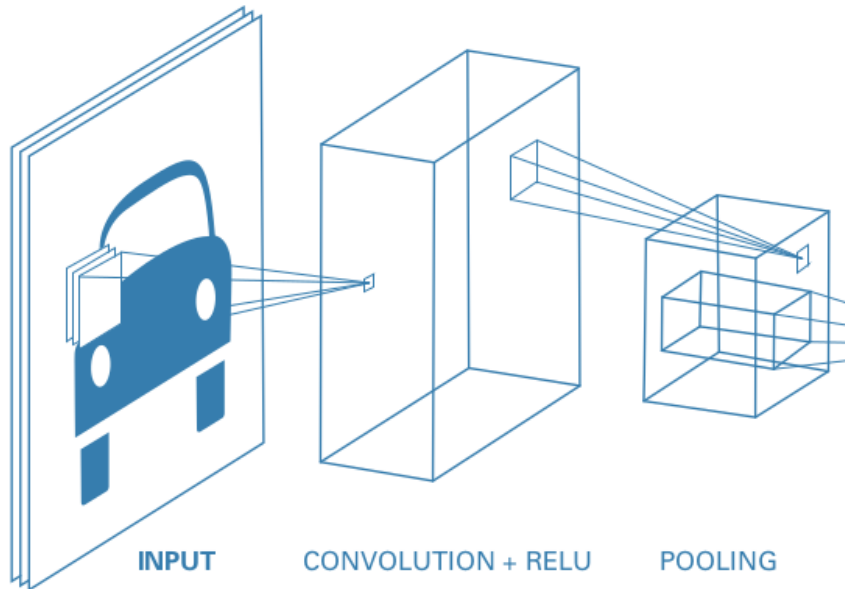


- Break images into square patches \approx tokens
- Apply a (learned) linear projection to each patch
- Add a "CLS" token
- Add positional embedding for each patch "index"
- Feed to Transformer
- Use final layer CLS representation to make prediction

CNNs vs. Vision Transformers

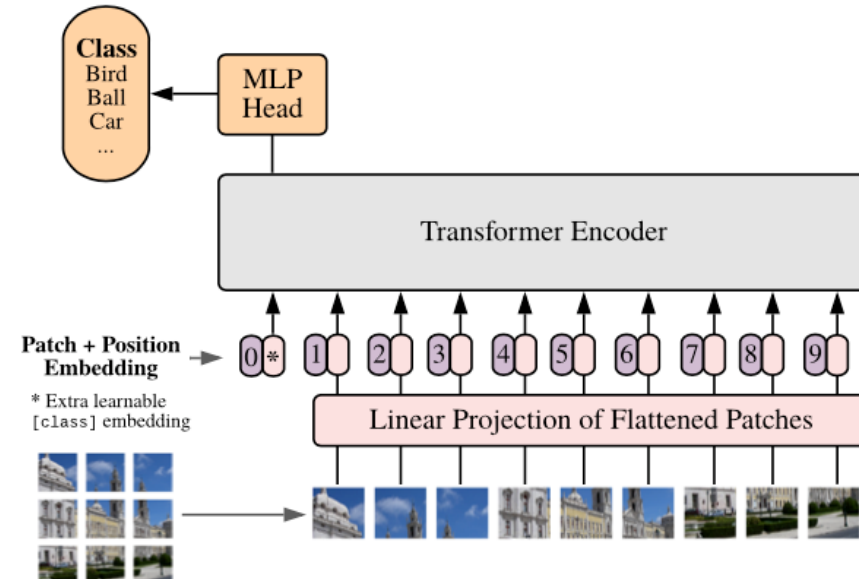
CNN

- Each neuron in 1 layer has a limited receptive field
- Strong “inductive bias”: Model has to look locally first, globally later



Vision Transformer

- Each hidden state *can* access information about a faraway part of image via attention
- Weaker “inductive bias”



Conclusion: Transformers

- “Attention is all you need”
 - Get rid of recurrent connections—all “communication” between words in sequence is handled by attention
 - Have multiple attention “heads” to learn different types of relationships between words
 - Each head has its own parameters, which enable them to learn different things
 - Plus lots of additional components to make it fit together
 - Most famous modern language models (e.g., ChatGPT) are Transformers!
- Pretraining
 - First train on large labeled or unlabeled datasets
 - Features learned are useful for other tasks with less data
- Transformers can even be used for images