Deep Learning Review, Transformers

Robin Jia
USC CSCI 467, Fall 2023
October 17, 2023

Parameters & Hyperparameters

Parameters

- Numbers that **directly determine** the model's predictions
- **Must be learned**
  - Usually by choosing parameter values that minimize some loss function
- Example: **w** & **b** for logistic regression, which makes prediction

$$P(y=1 \mid x) = \sigma(\mathbf{w}^T x + \mathbf{b})$$

Hyperparameters

- Numbers that **influence** which parameters are learned
  - Thus, they *indirectly* influence model's predictions
- **Cannot be learned**—must be chosen before learning starts
  - Hyperparameter tuning: Can try learning many times with different hyperparameters, then pick the one with best development accuracy
- Example: **λ** for L2 regularization

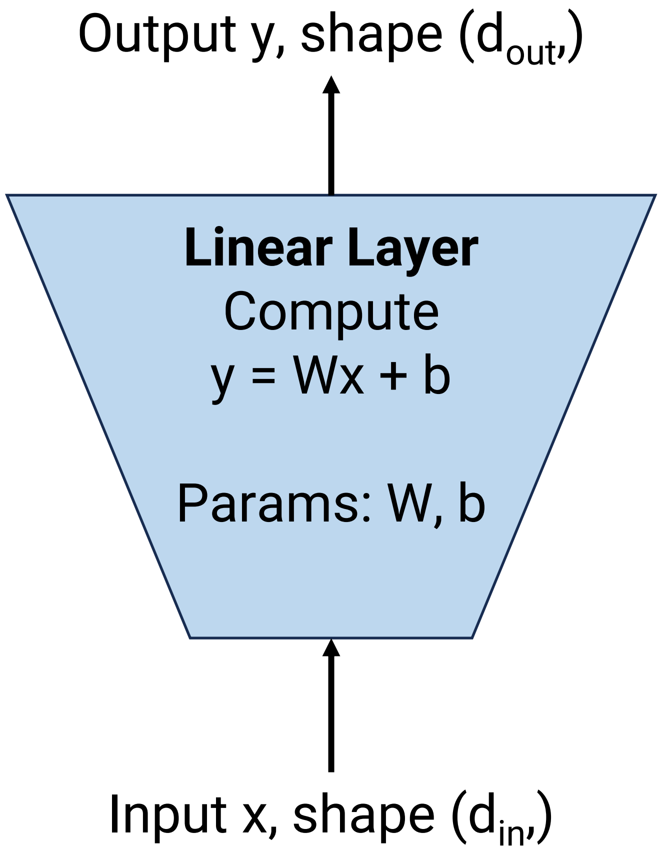
Deep Learning Review

- Neural Network = Many "layers" stacked on top of each other
  - Layers built from a core set of building blocks
  - Arrangement of layers is called an "architecture"
- Each layer takes in some input and computes some output

The Basic "Building Blocks"

(1) Linear Layer

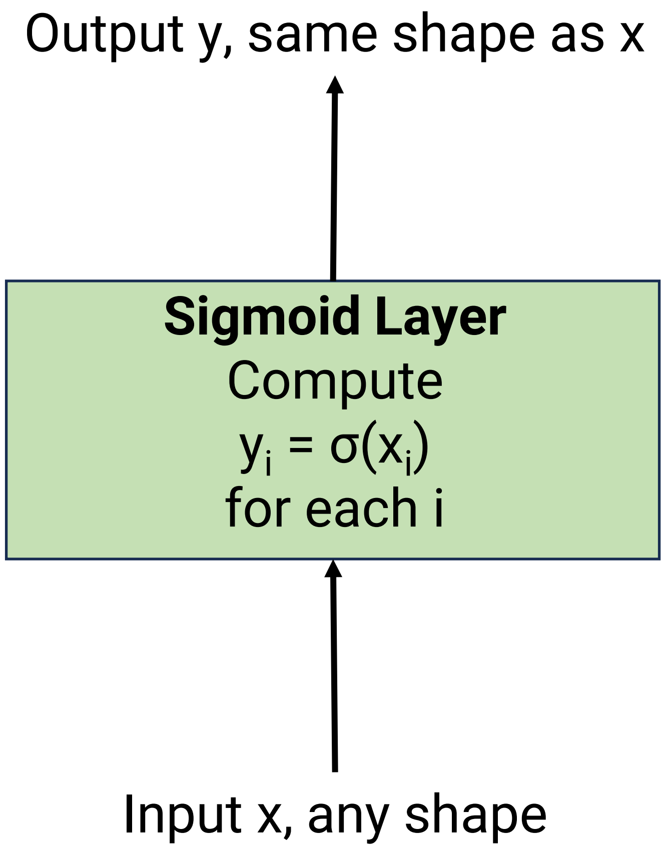
- Input x: Vector of dimension d_{in}
- Output y: Vector of dimension d_{out}
- Formula: $y = Wx + b$
- Parameters
  - W: d_{out} x d_{in} matrix
  - b: d_{out} vector
- In pytorch: nn.Linear()



The Basic “Building Blocks”

(2) Non-linearity Layer

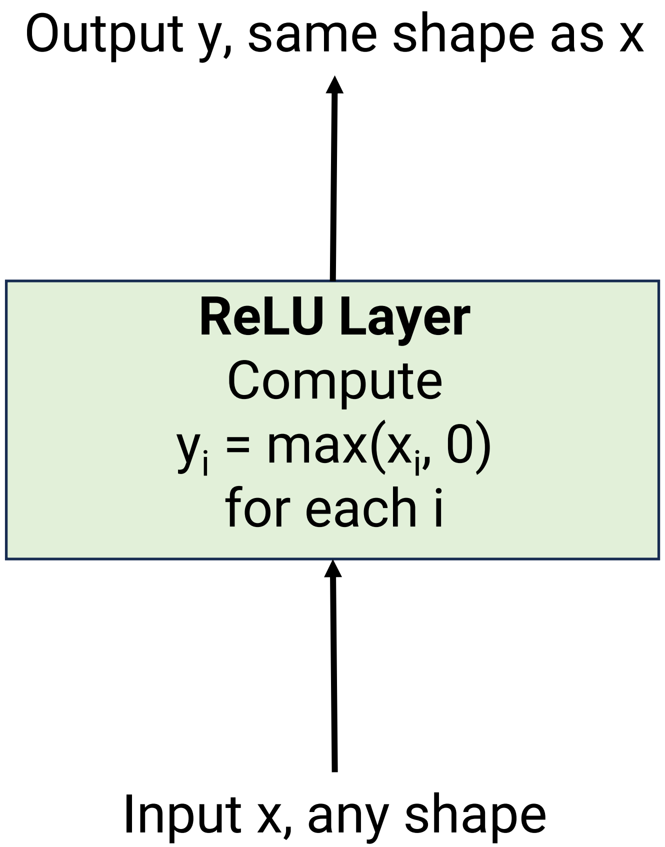
- Input x: Any number/vector/matrix
- Output y: Number/vector/matrix of same shape
- Possible formulas:
  - Sigmoid: $y = \sigma(x)$, elementwise
  - Tanh: $y = \tanh(x)$, elementwise
  - Relu: $y = \max(x, 0)$, elementwise
- Parameters: None
- In pytorch: torch.sigmoid(), nn.functional.relu(), etc.



The Basic "Building Blocks"

(2) Non-linearity Layer

- Input x: Any number/vector/matrix
- Output y: Number/vector/matrix of same shape
- Possible formulas:
  - Sigmoid: $y = \sigma(x)$, elementwise
  - Tanh: $y = \tanh(x)$, elementwise
  - Relu: $y = \max(x, 0)$, elementwise
- Parameters: None
- In pytorch: torch.sigmoid(), nn.functional.relu(), etc.



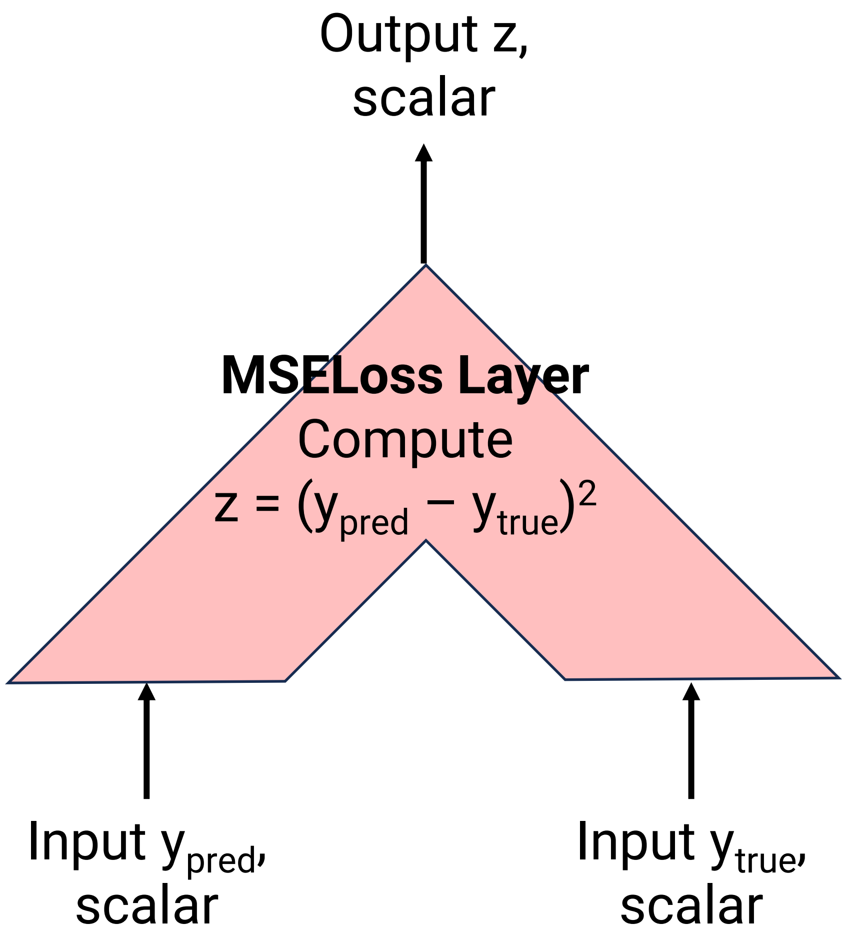
The Basic “Building Blocks”

(3) Loss Layer

- Inputs:
  - y_{pred}: shape depends on task
  - y_{true}: scalar (e.g., correct regression value or class index)
- Output z: scalar
- Possible formulas:
  - Squared loss: y_{pred} is scalar, $z = (y_{pred} - y_{true})^2$
  - Softmax regression loss: y_{pred} is vector of length C,

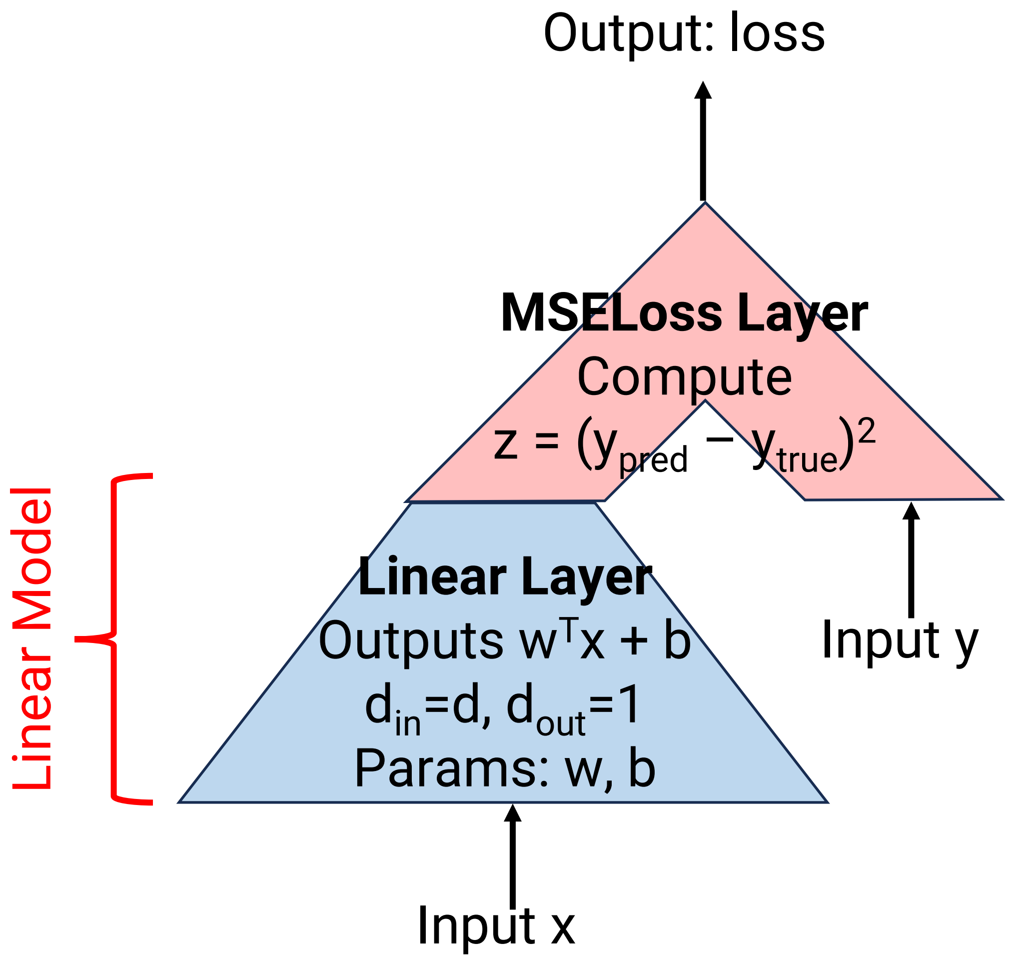
$$z = -\left(y_{\text{pred}}[y_{\text{true}}] - \log \sum_{i=1}^{C} \exp(y_{\text{pred}}[i]) \right)$$

- Parameters: None
- In pytorch: nn.MSELoss(), nn.CrossEntropyLoss(), etc.



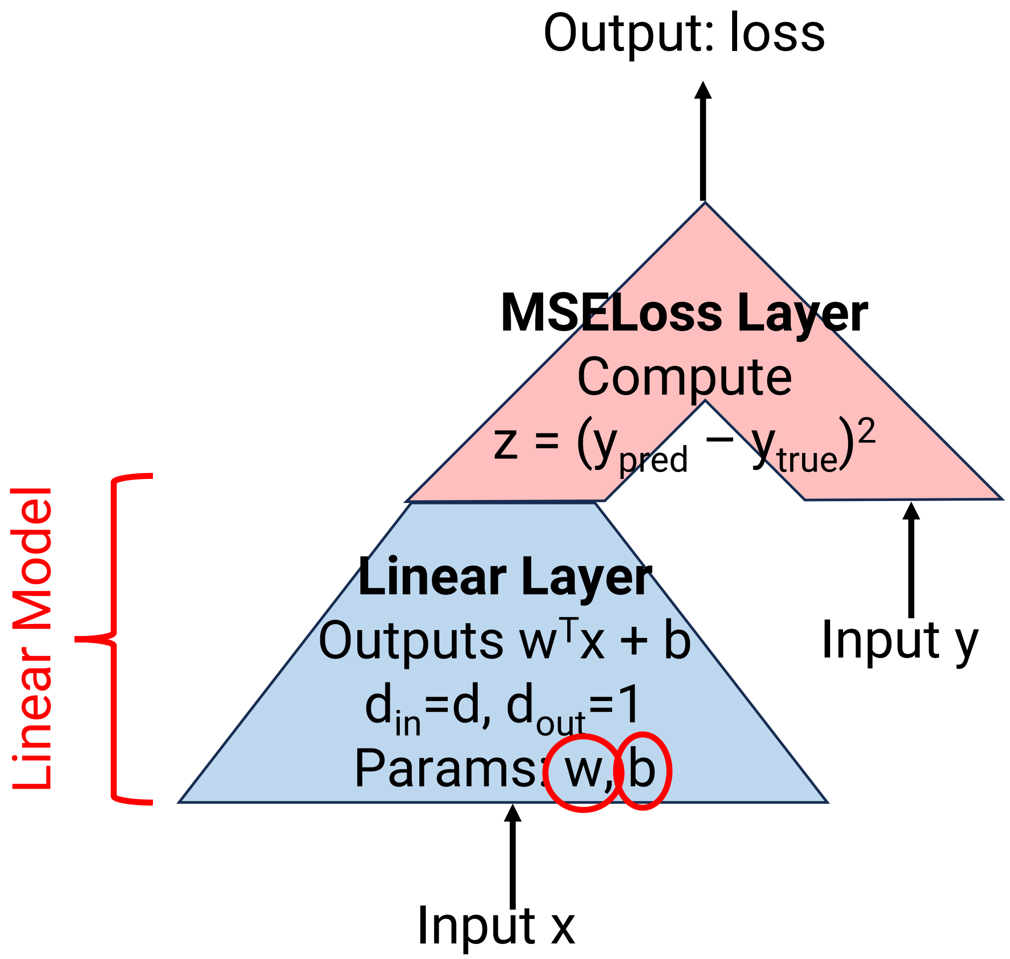
Building Linear Regression

- Step 1: Compute the loss on one example
  - Training example is (x, y)
  - x is vector of length d, y is scalar



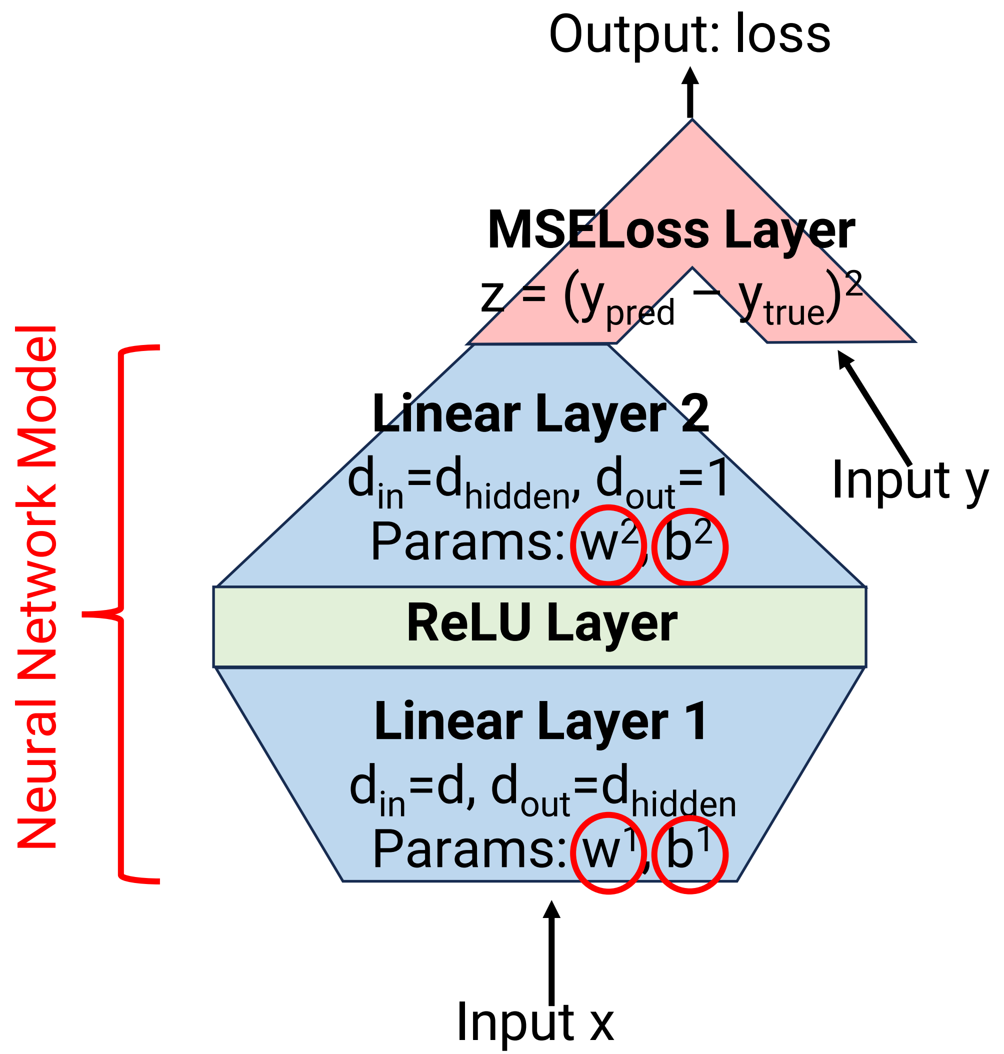
Building Linear Regression

- Step 1: Compute the loss on one example
  - Training example is (x, y)
  - x is vector of length d, y is scalar
- Step 2: Compute gradient of loss with respect to all parameters
- Step 3: Update all parameters with gradient descent update rule



Building an MLP (for regression)

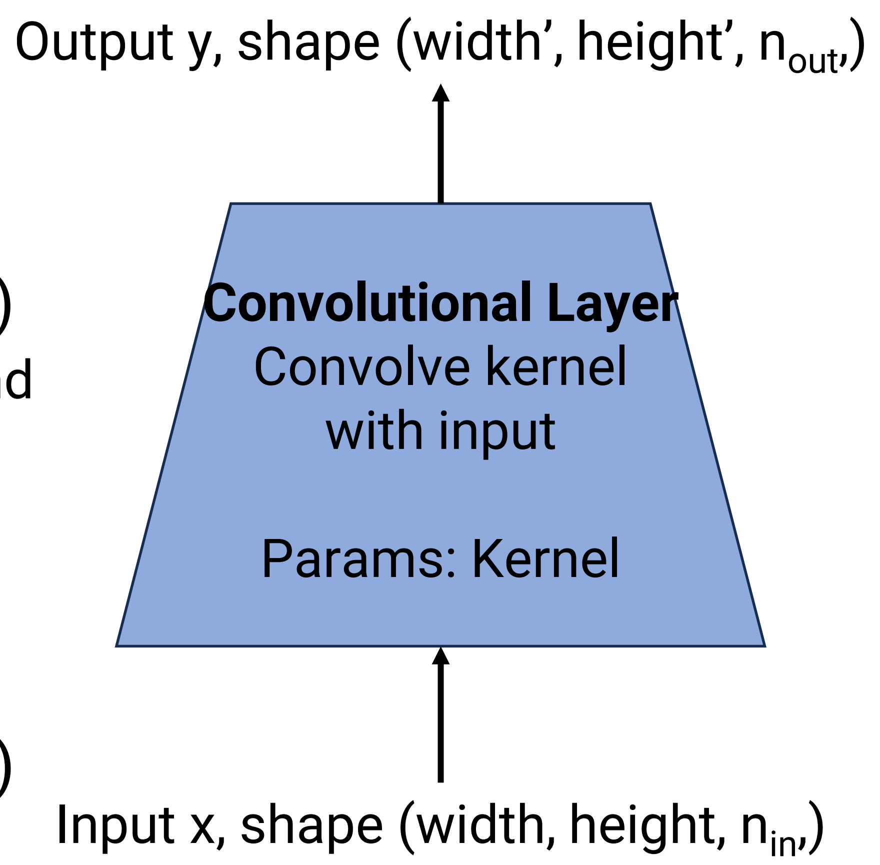
- **Steps for training are exactly the same:**
- Step 1: Compute the loss on one example
  - Training example is (x, y)
  - x is vector of length d, y is scalar
- Step 2: Compute gradient of loss with respect to all parameters
  - **No matter how many/which layers we use, backpropagation can automatically compute gradient of loss with respect to parameters**
- Step 3: Update all parameters with gradient descent update rule



CNN “Building Blocks”

(4) Convolutional Layer

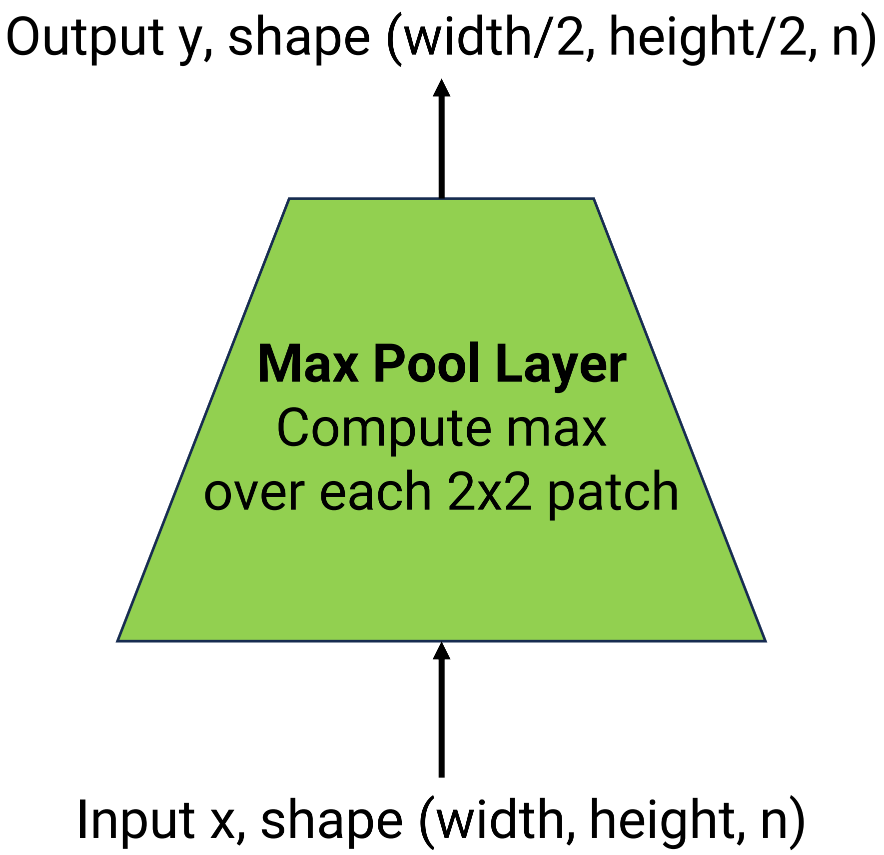
- Input x: Tensor of dimension (width, height, n_{in})
  - n_{in}: Number of input channels (e.g. 3 for RGB images)
- Output y: Tensor of dimension (width’, height’, n_{out})
  - width’, height’: New width & height, depends on stride and padding
  - n_{out}: Number of output channels
- Formula: Convolve input with kernel
  - Recall: This is in fact a linear operation
- Parameters: Kernel params of shape (K, K, n_{in}, n_{out})
- In pytorch: nn.Conv2d()



CNN "Building Blocks"

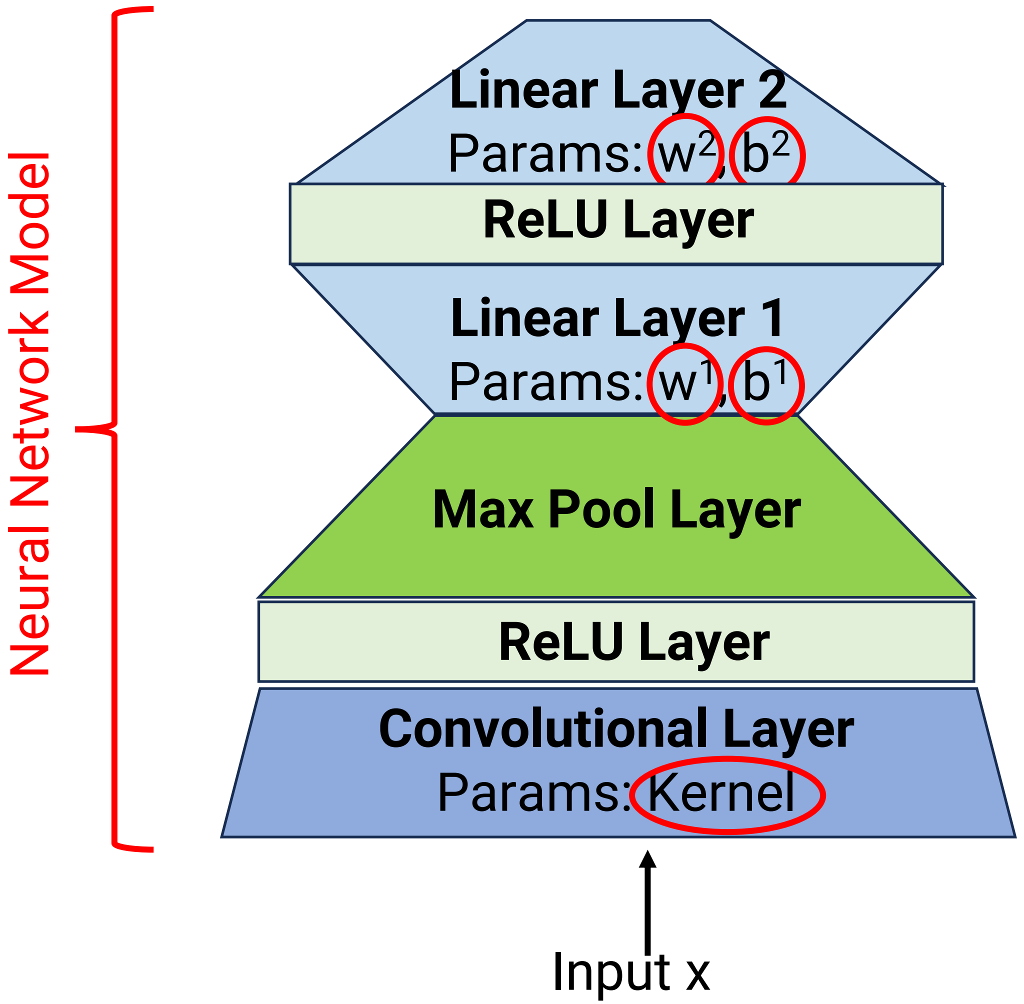
(5) Max Pooling layer

- Input x: Tensor of dimension (width, height, n)
  - n: Number of channels
- Output y: Tensor of dimension (width/2, height/2, n)
- Formula: In each 2x2 patch, compute max
- Parameters: None
- In pytorch: nn.MaxPool2d()



Building a CNN Model

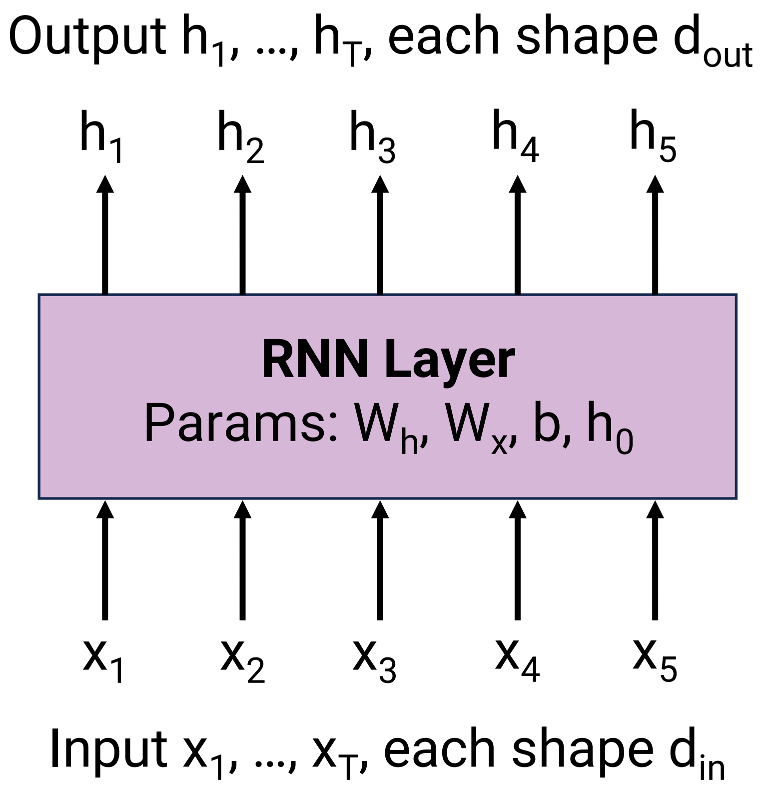
- A generic CNN architecture
  - First use conv + relu + pool to extract features
  - Then use MLP to make final prediction
- Basic steps are still all the same
  - Backpropagation still works
- Gradient descent needed to update all parameters



RNN "Building Blocks"

(6) RNN Layer

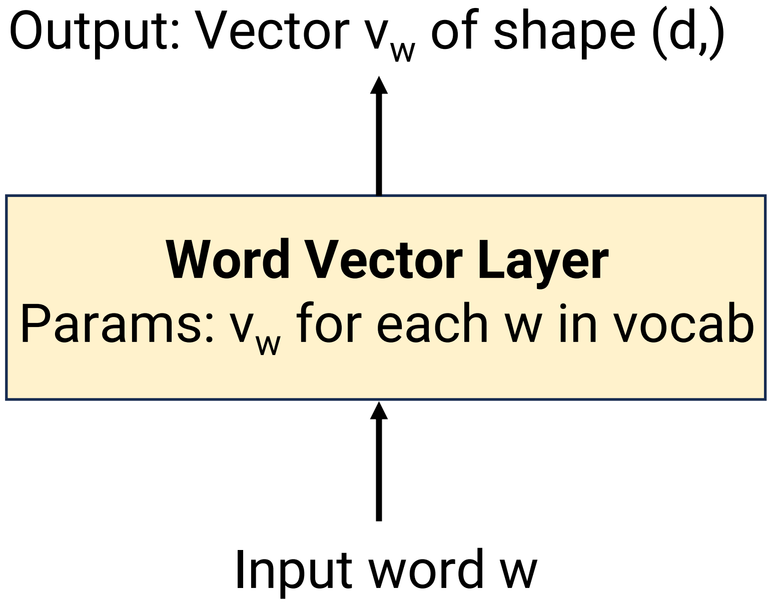
- Input: List of vectors $x_1, \ldots, x_T$, each of size d_{in}
  - E.g., x_t is word vector for t-th word in sentence
  - Equivalent to a T x d_{in} matrix
- Output: List of vectors $h_1, \ldots, h_t$, each of size d_{out}
  - d_{out}: Dimension of hidden state
  - Equivalent to a T x d_{out} matrix
- Formula (Elman RNN): $h_t = \tanh\left(W_h h_{t-1} + W_x x_t + b\right)$
- Parameters:
  - W_h: Matrix of shape (d_{out}, d_{out})
  - W_x: Matrix of shape (d_{out}, d_{in})
  - b: Vector of shape (d_{out},)
  - h_0: Vector of shape (d_{out},)
- In pytorch: nn.RNN(), nn.LSTM(), etc.



RNN “Building Blocks”

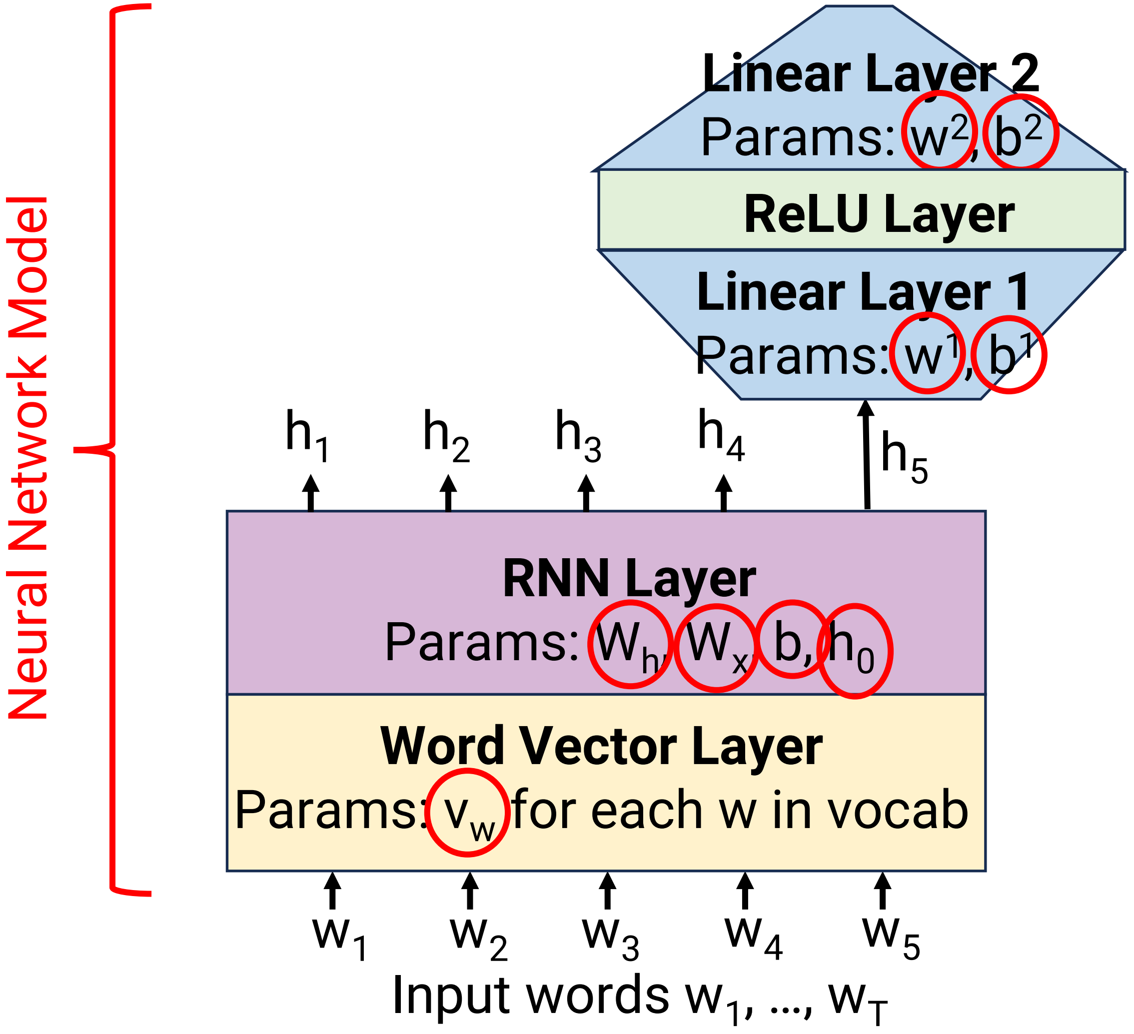
(7) Word Vector Layer

- Input w: A word
  - Must be in the vocabulary
  - Can also input list of words
- Output: A vector of length d
  - If input is many words, output is list of vectors corresponding to each word
- Formula: Return `word_vecs[w]`
- Parameters:
  - For each word w in vocabulary, there is a word vector parameter v_w of shape d
  - Think of this as a dictionary called `word_vecs`, where the keys are words & values are learned parameter vectors
- In pytorch: nn.Embedding()

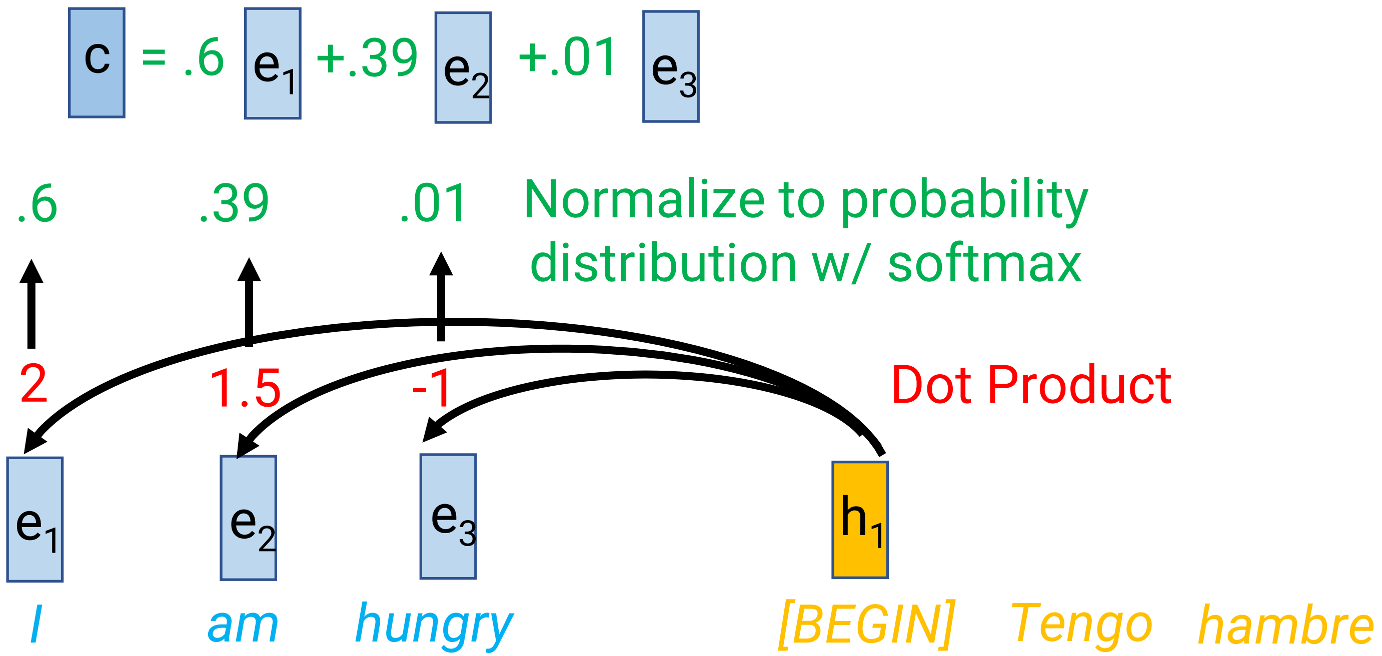


Building an RNN encoder model

- A generic RNN architecture
  - Map each word to a vector
  - Feed word vectors to RNN to generate list of hidden states
  - Feed final hidden state to MLP to make final prediction
- Basic steps are still all the same
  - Backpropagation still works
- Gradient descent needed to update all parameters



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

Attention Layer as a Building Block

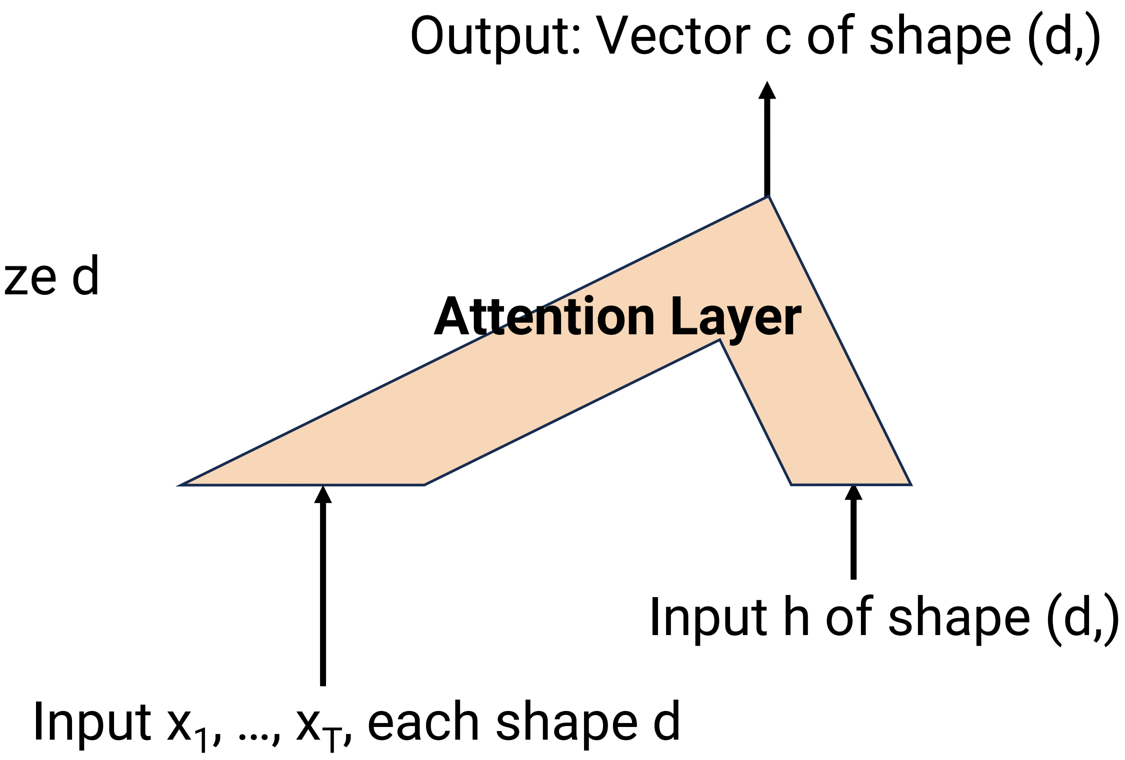
(8) Attention Layer

- Inputs:
  - $x_1, \ldots, x_T$: List of vectors to attend to, size d
  - h: "query" vector to decide what to attend to, size d
- Output c: Convext vector of size d
- Formula:

$$p_t = \frac{\exp(h^\top x_t)}{\sum_{i=1}^{T} \exp(h^\top x_i)} \forall i = \{1, \ldots, T\}$$

$$c = \sum_{t=1}^{T} p_t x_t$$

- Parameters: None
- In pytorch: Implement with sequence of basic operations



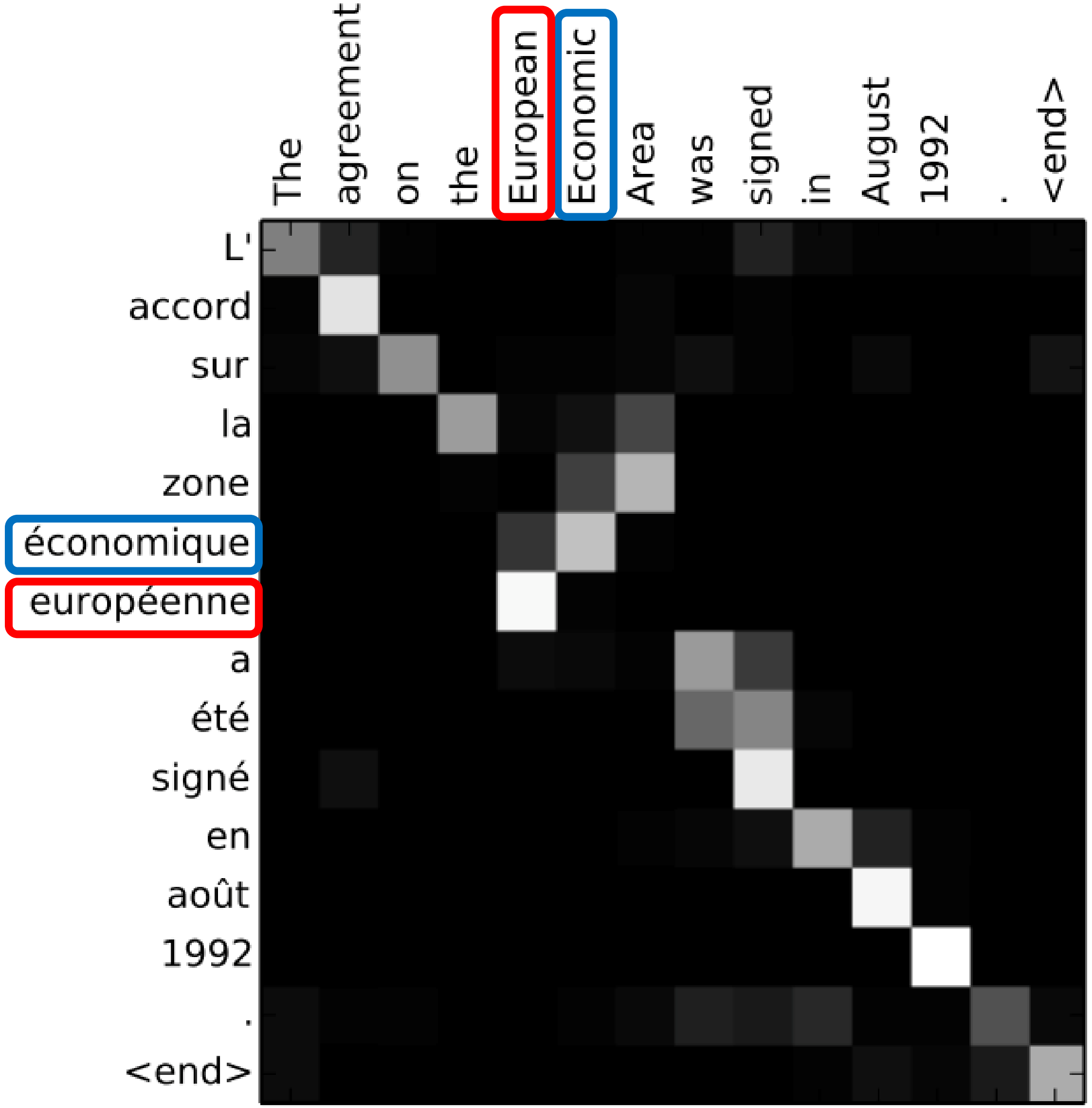
Summary: Neural Network Building Blocks

- Neural network components are like lego bricks
  - Can be assembled in many different ways
  - Some have parameters, some don't
- Training strategy is always the same
  - (1) Compute loss
  - (2) Take gradient of loss w.r.t. parameters
  - (3) Gradient descent
- Backpropagation works on any architecture
- **So, when we discuss neural architectures, we only need to discuss the forward pass**
  - Backpropagation takes care of gradients
  - Gradient descent takes care of learning parameters

Announcements

- Midterm grades released
- Project Proposal grades & feedback released
- Midterm report due October 31
  - 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

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***

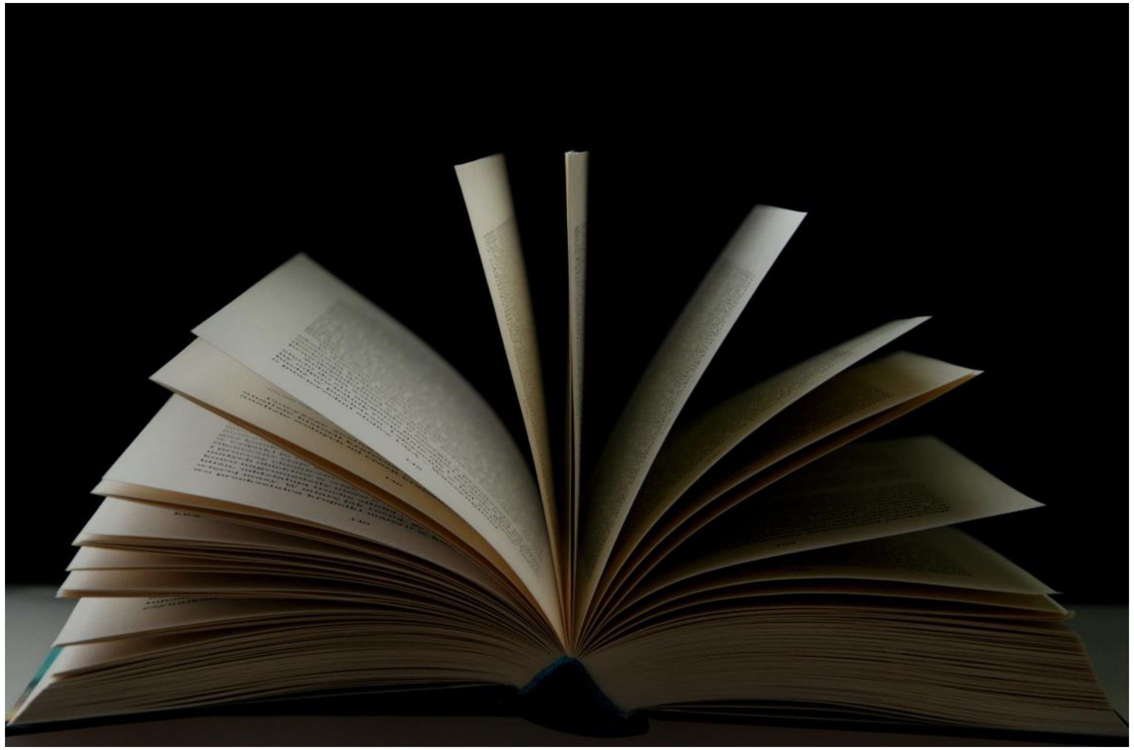
- Modeling relationships between words
  - Translation alignment
  - Syntactic dependencies

Challenges of modeling sequences

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

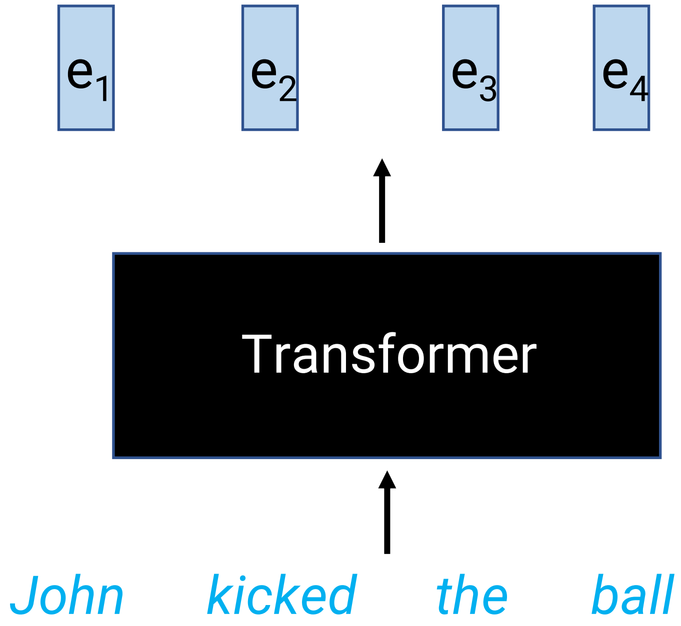
- Modeling relationships between words
  - Translation alignment
  - Syntactic dependencies
  - Coreference relationships

Challenges of modeling sequences



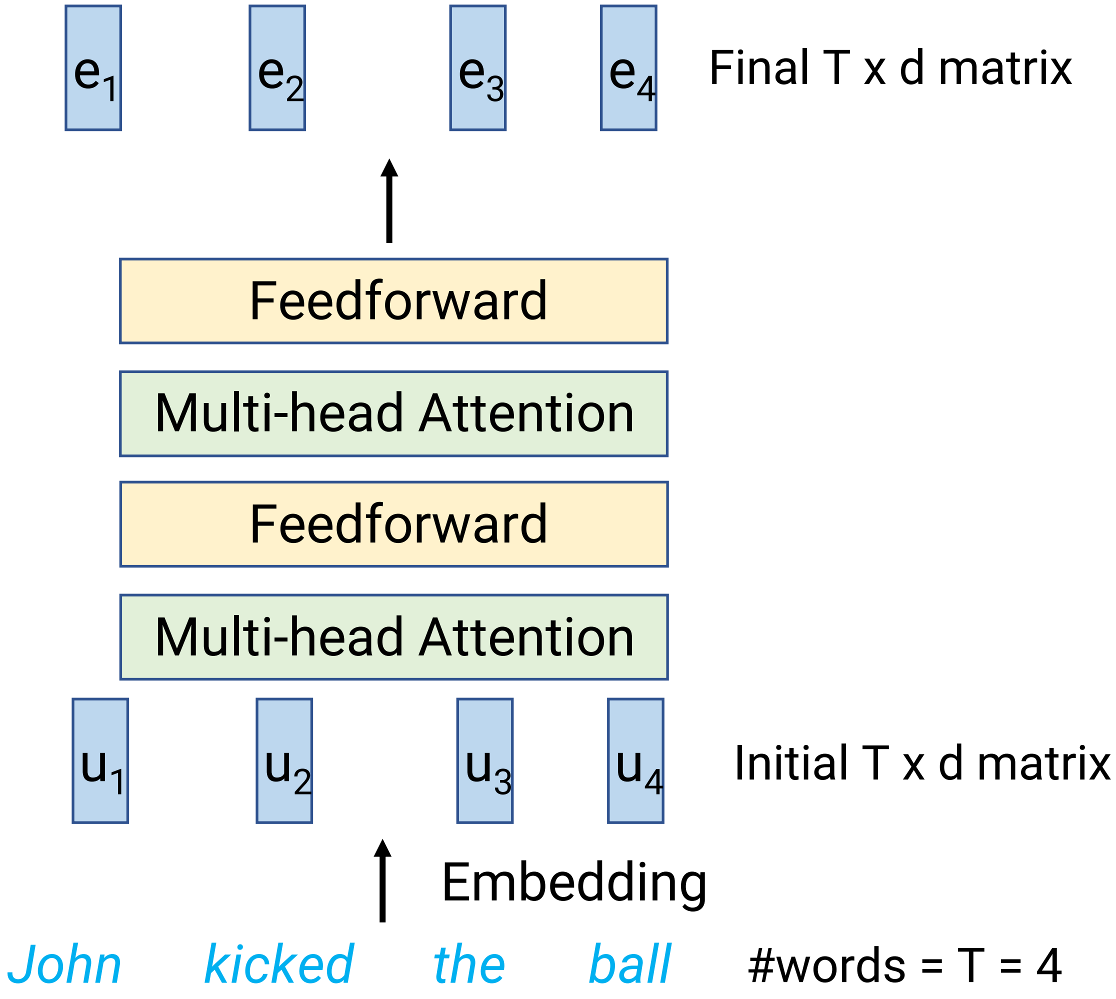
- Modeling relationships between words
  - Translation alignment
  - Syntactic dependencies
  - Coreference relationships
- Long range dependencies
  - E.g., consistency of characters in a novel
- Attention captures relationships & doesn’t care about “distance”

Today: The Transformer Architecture



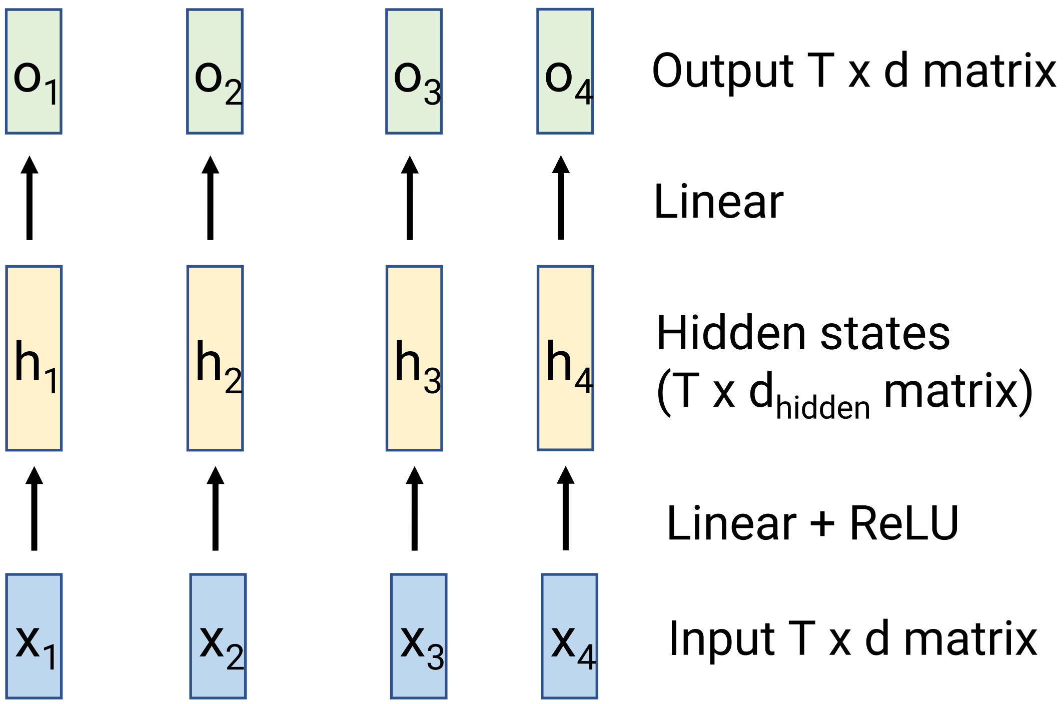
- Input: Sequence of words
- Output: Sequence of vectors, one per word
- Same “type signature” as RNN
- Motivation
  - Don’t do explicit sequential processing
  - Instead, let **attention** figure out which words are relevant to each other
    - RNN assumes sequence order is what matters
  - “Attention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

Transformer internals



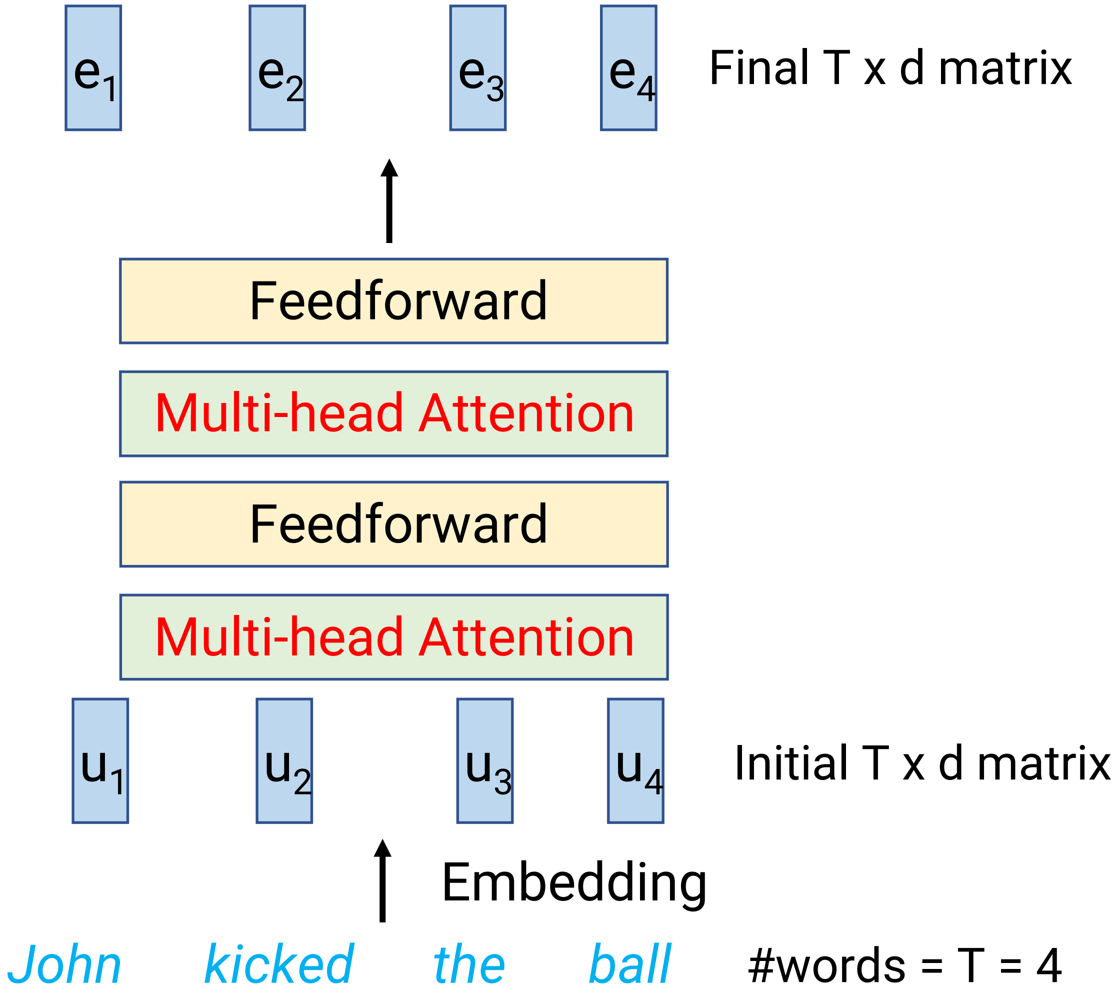
- One transformer consists of
  - Initial embeddings for each word of size d
    - Let T =#words, so initially we have a T x d matrix
  - Alternating layers of
    - “Multi-headed” attention layer
    - Feedforward layer
    - Both take in T x d matrix and output a new T x d matrix
  - Plus some bells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

Feedforward layer



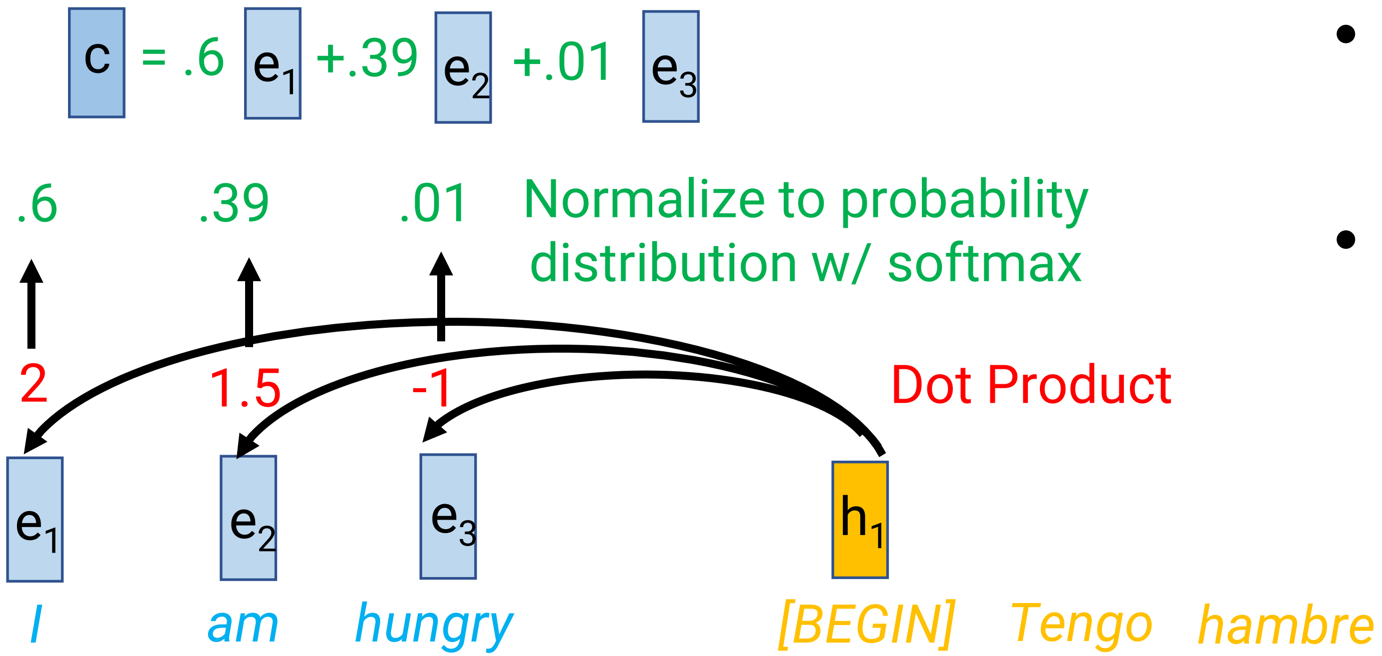
- Input: T x d matrix
- Output: Another T x d matrix
- Apply the same MLP separately to each d-dimensional vector
  - Linear layer from d to d_{hidden}
  - ReLU (or other nonlinearity)
  - Linear layer from d_{hidden} to d
- Note: No information moves between tokens here

Transformer internals



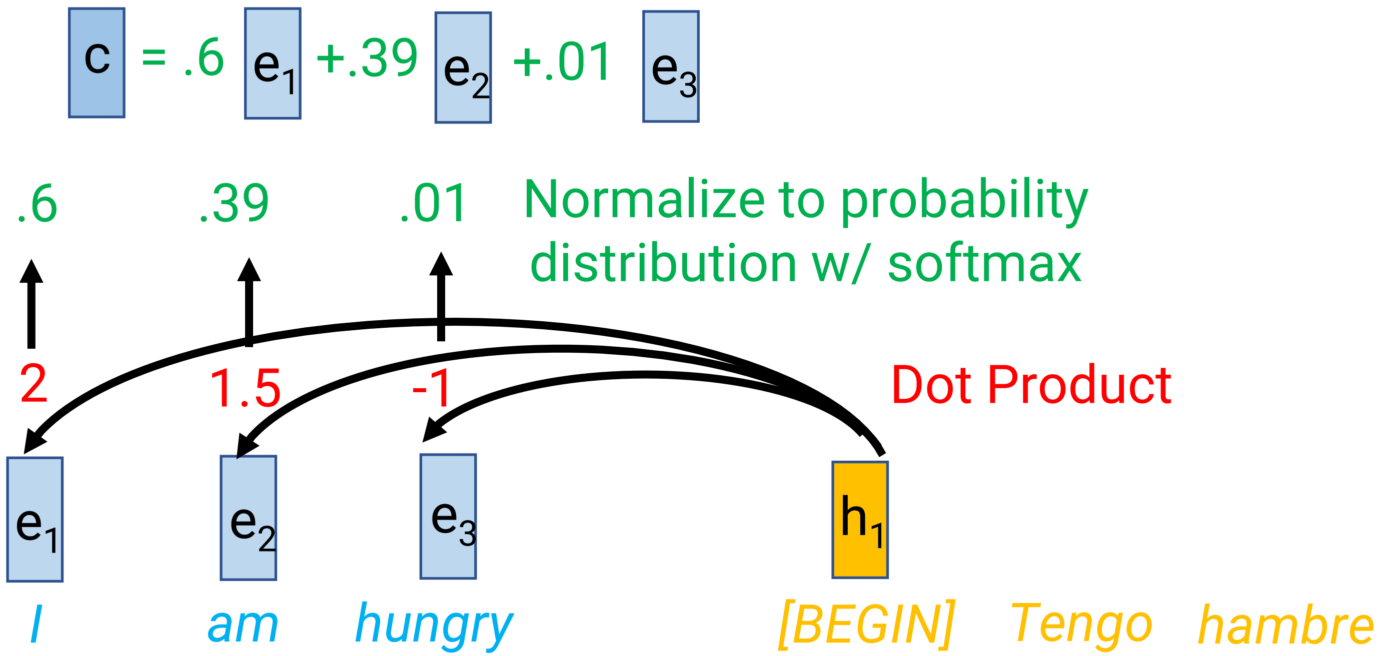
- One transformer consists of
  - Initial embeddings for each word of size d
    - Let T =#words, so initially we have a T x d matrix
  - Alternating layers of
    - **"Multi-headed" attention layer**
    - Feedforward layer
    - Both take in T x d matrix and output a new T x d matrix
  - Plus some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

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

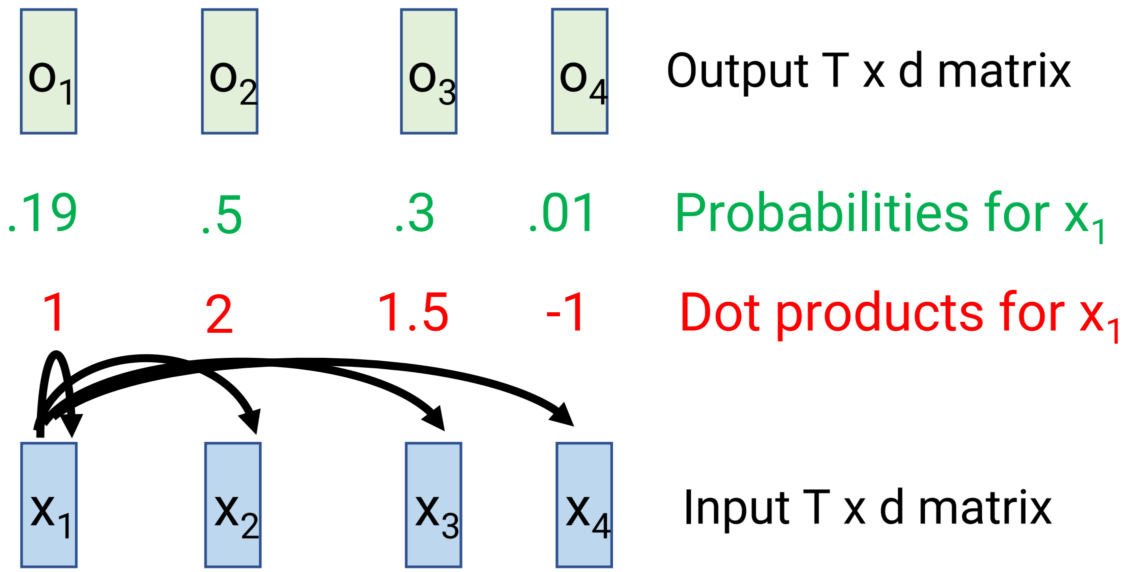
.19 .5 .3 .01 Probabilities for x_1

1 2 1.5 -1 Dot products for x_1

x_1 x_2 x_3 x_4 Input T x d matrix

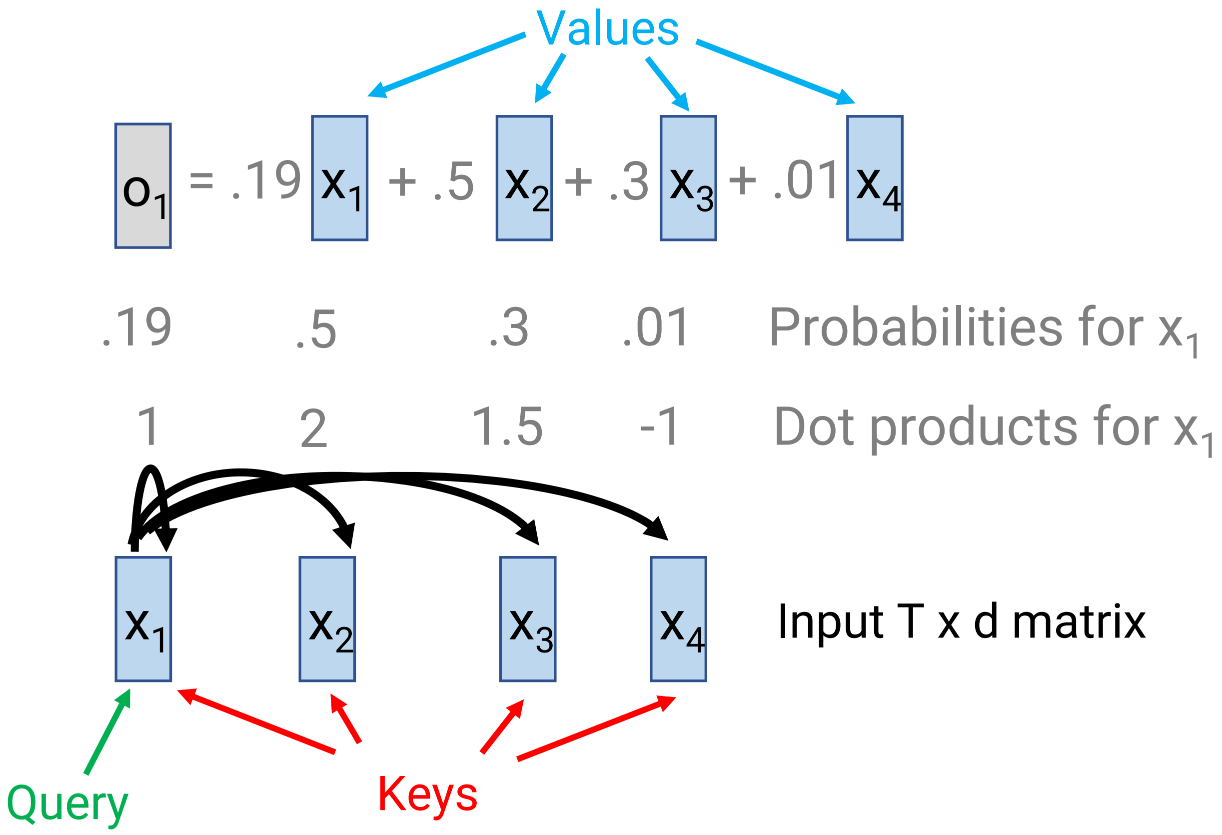
- Take x_1 and dot product it with all T inputs (including itself)
- Apply softmax to convert to probability distribution
- Compute output o_1 as weighted sum of inputs

Change #1: Self-attention



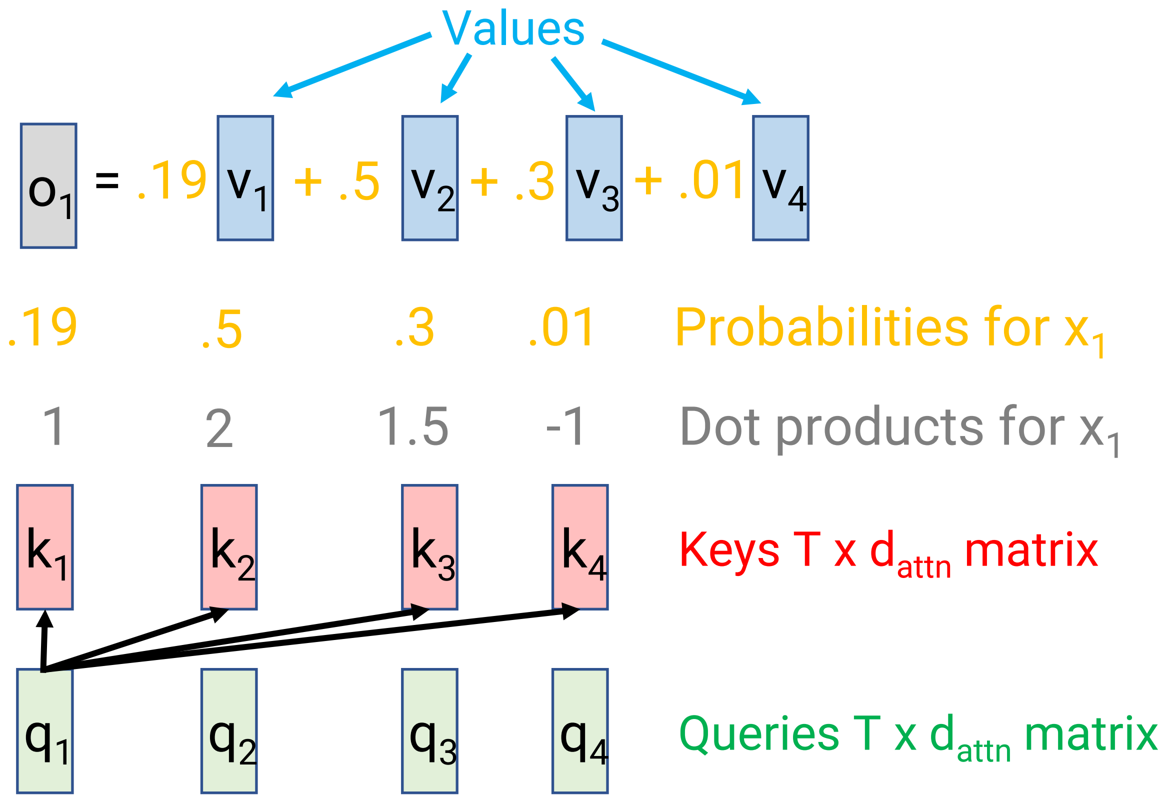
- Take x_1 and dot product it with all T inputs (including itself)
- Apply softmax to convert to probability distribution
- Compute output o_1 as weighted sum of inputs
- Repeat for t=2, 3, …, T
- Replacement for recurrence
  - RNN only allows information to flow linearly along sequence
  - Now, information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

Change #2: Separate queries, keys, and values



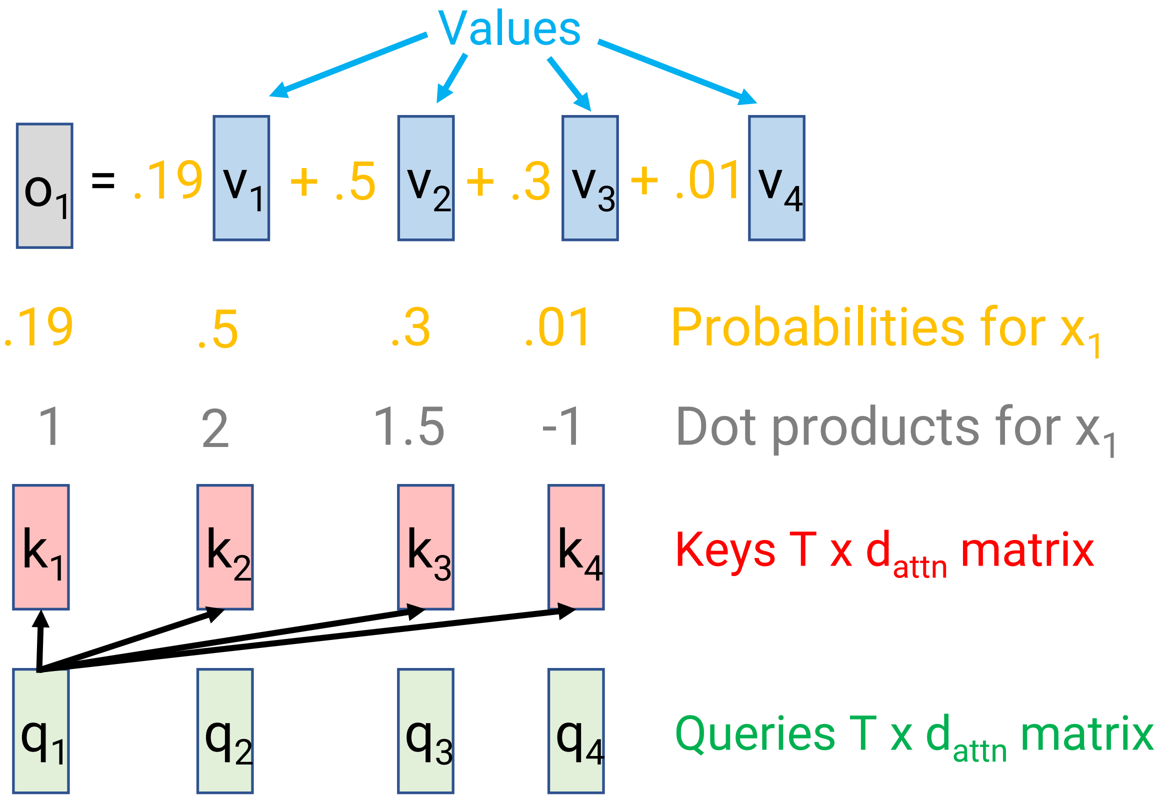
- Previously: We use input vectors in three ways
  - As “query” for current index
  - As “keys” to match with query
  - As “values” when computing output
- Idea: Use separate vectors for each usage
  - What each index “looks for” different from what it “matches with”
  - What you store in output different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

Change #2: Separate queries, keys, and values



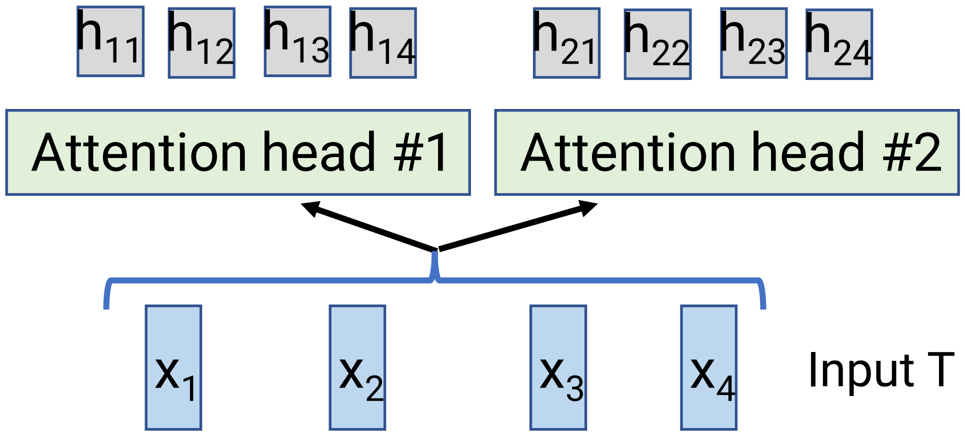
- Apply 3 separate linear layers to each of $x_1, \ldots, x_T$ to get
  - Queries $[q_1, \ldots, q_T]$
  - Keys $[k_1, \ldots, k_T]$
  - Values $[v_1, \ldots, v_T]$
  - Note: This adds parameters W^Q, W^K, W^V
  - Each linear layer maps from dimension d to dimension d_{attn}
- Dot product q_1 with $[k_1, \ldots, k_T]$
- 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



- Apply 3 separate linear layers to input matrix X to get
  - Query matrix Q
  - Keys K
  - Values V
  - Note: This adds parameters W^Q, W^K, W^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

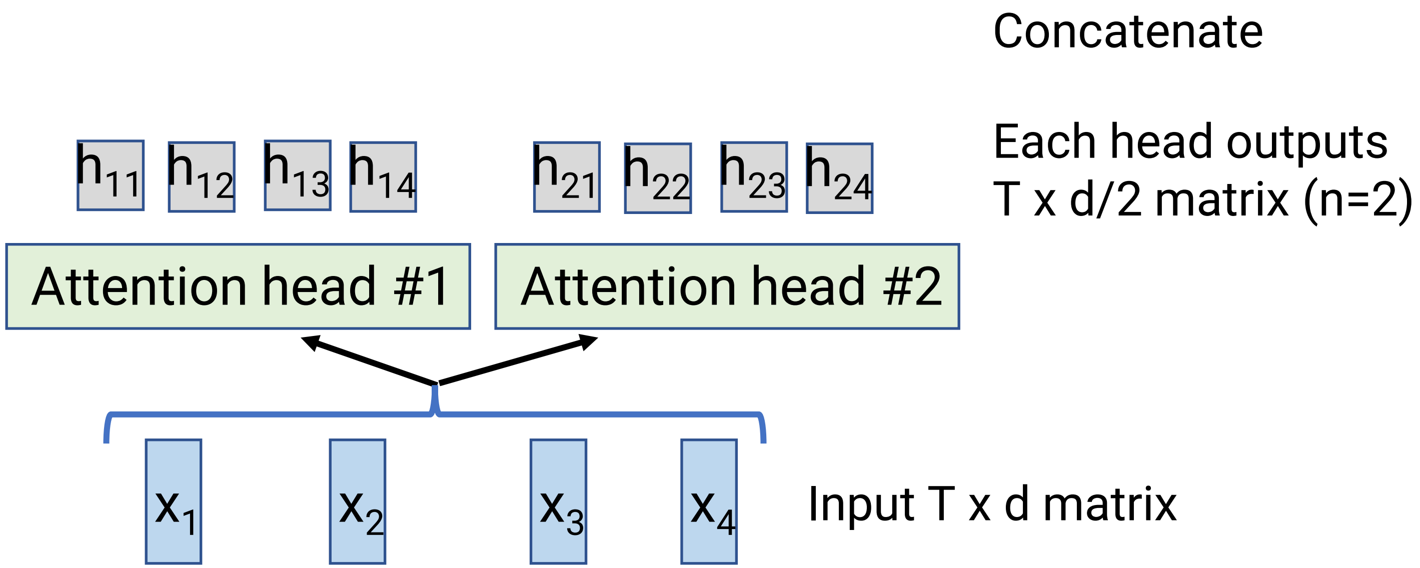


Each head outputs T x d/2 matrix (n=2)

Input T x d matrix

- 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

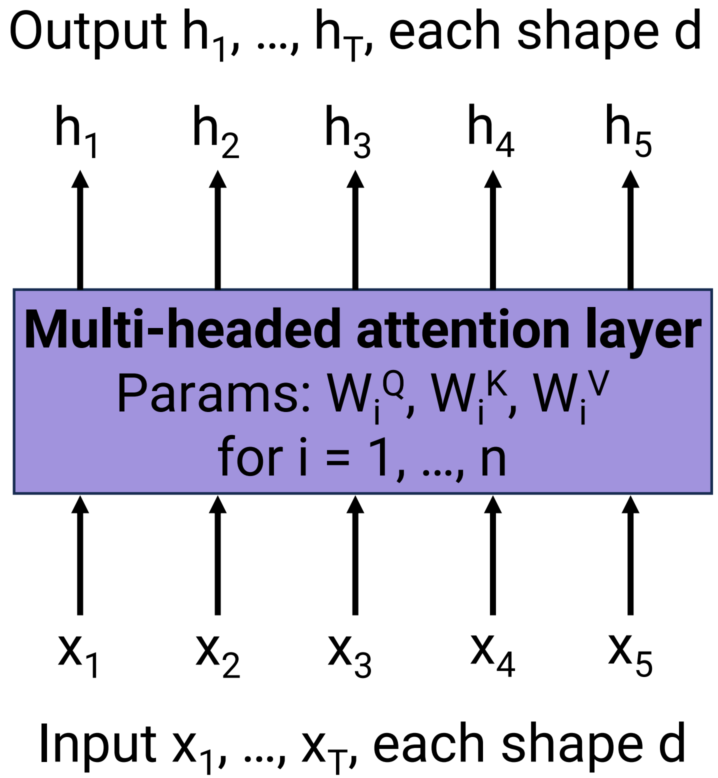


- Instead of doing attention once, have n different "heads"
  - Each has its own parameters maps to dimension $d_{attn} = d/n$
  - Concatenate at end to get output of size T x d
- Why? Different heads can capture different relationships between words

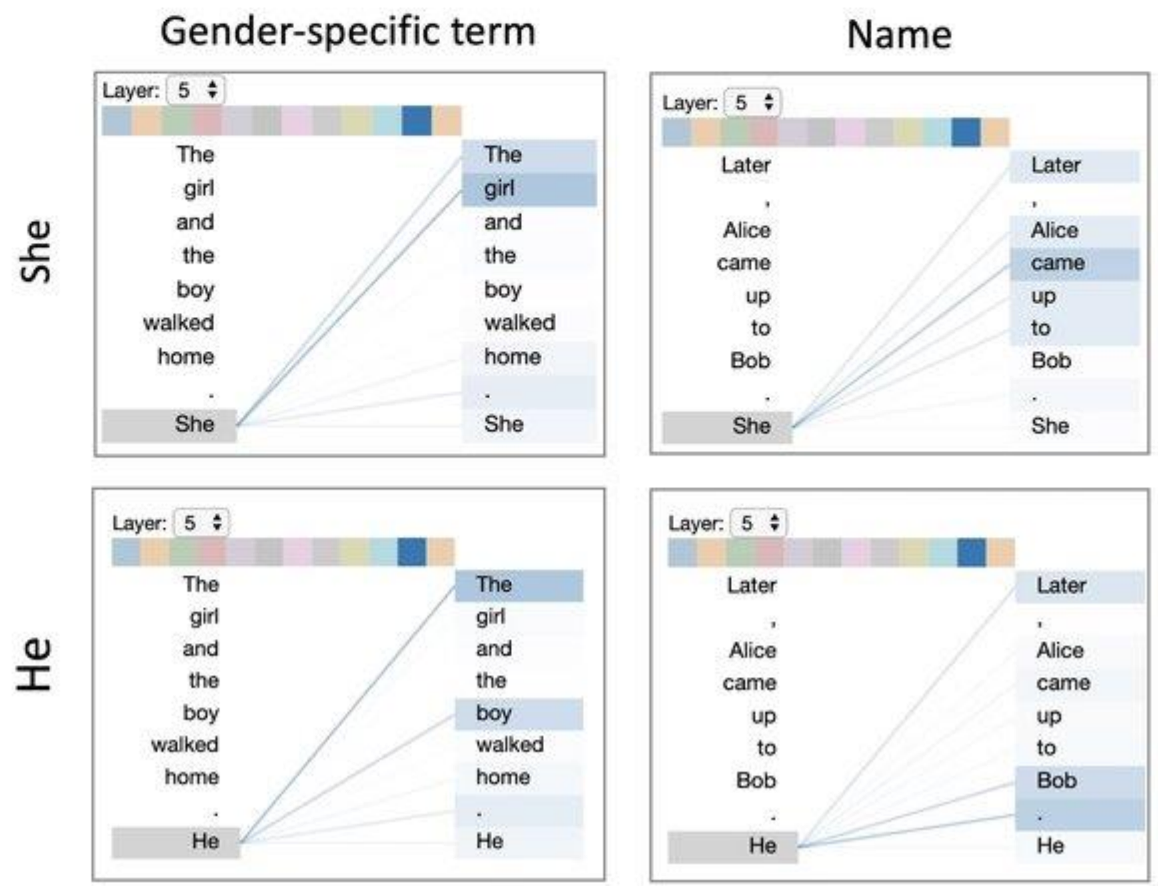
The Multi-headed Attention building block

(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()

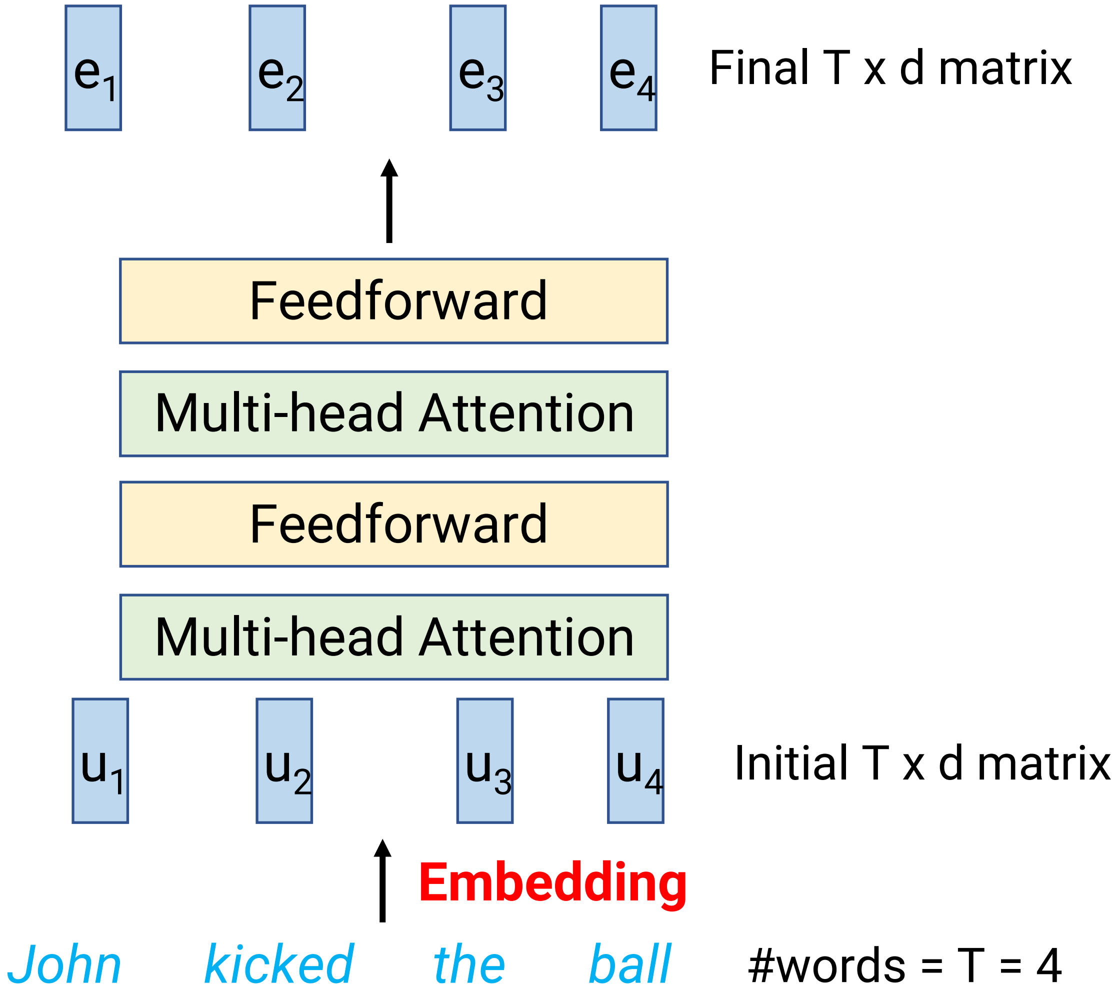


What do attention heads learn?



- 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

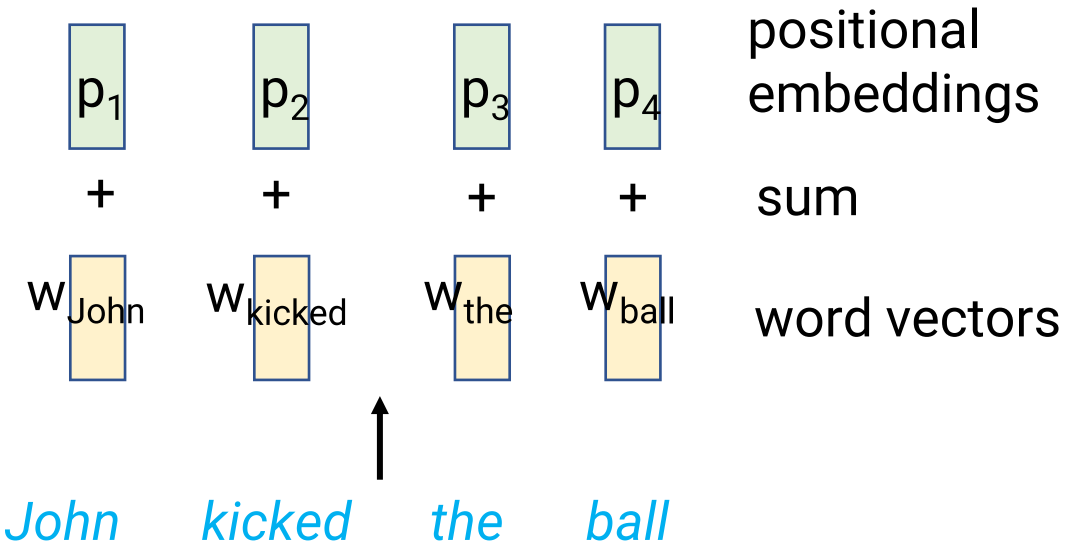
Transformer internals



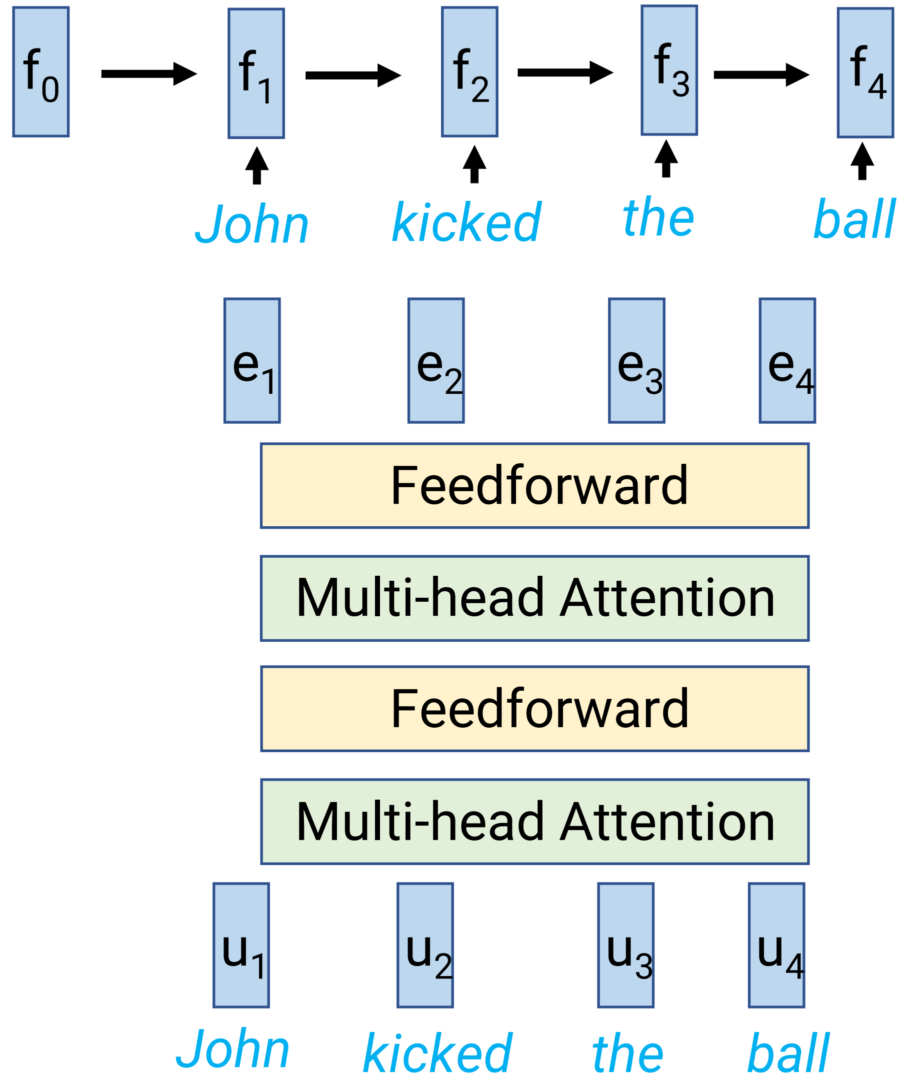
- One transformer consists of
  - **Initial embeddings** for each word of size d
    - Let T =#words, so initially we have a T x d matrix
  - Alternating layers of
    - "Multi-headed" attention layer
    - Feedforward layer
    - Both take in T x d matrix and output a new T x d matrix
  - Plus some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

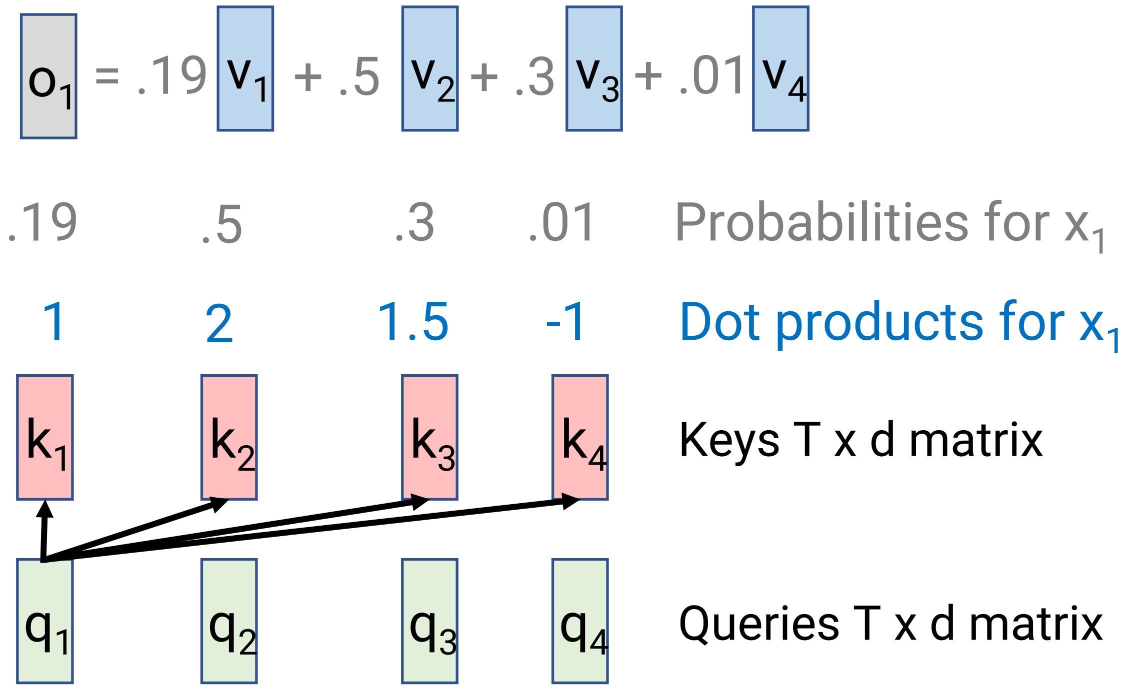


- RNNs
  - Linear in sequence length
  - But all operations have to happen in series
- Transformers
  - Quadratic in sequence length (T x T matrices)
  - But can be parallelized (big matrix multiplication)

Bells and whistles

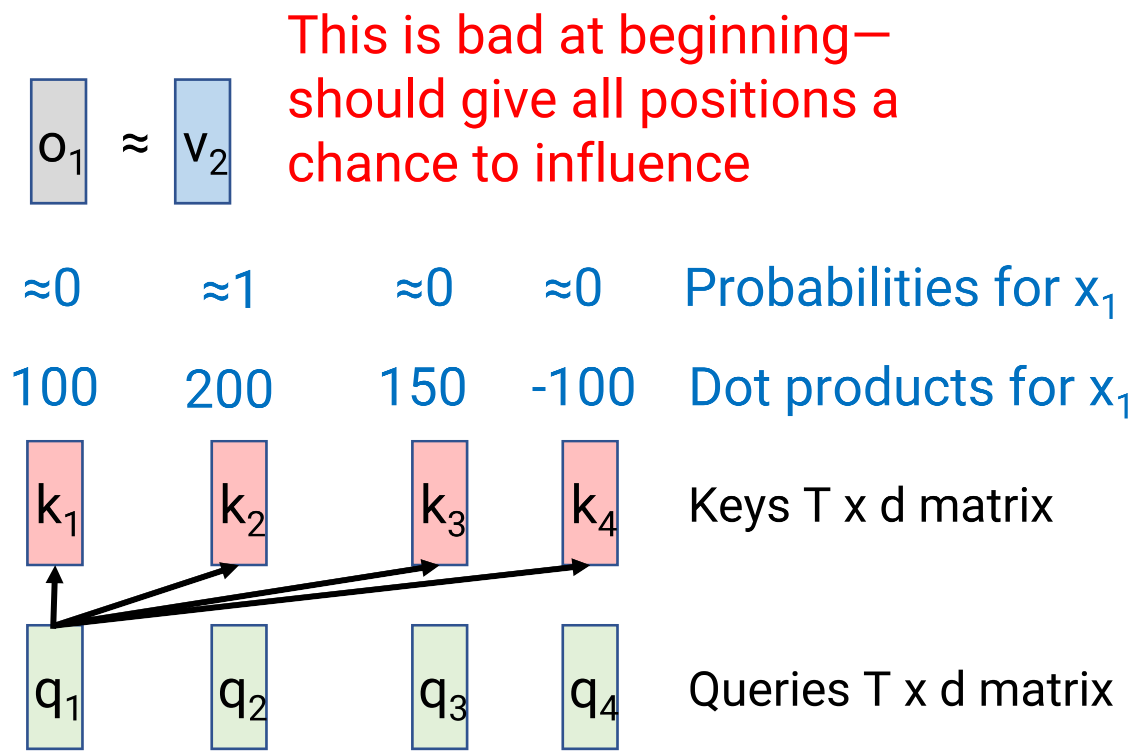
- Attention: Scaled dot products
- Residual connections
- Layer Norm
- Tokenization: Byte Pair Encoding

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

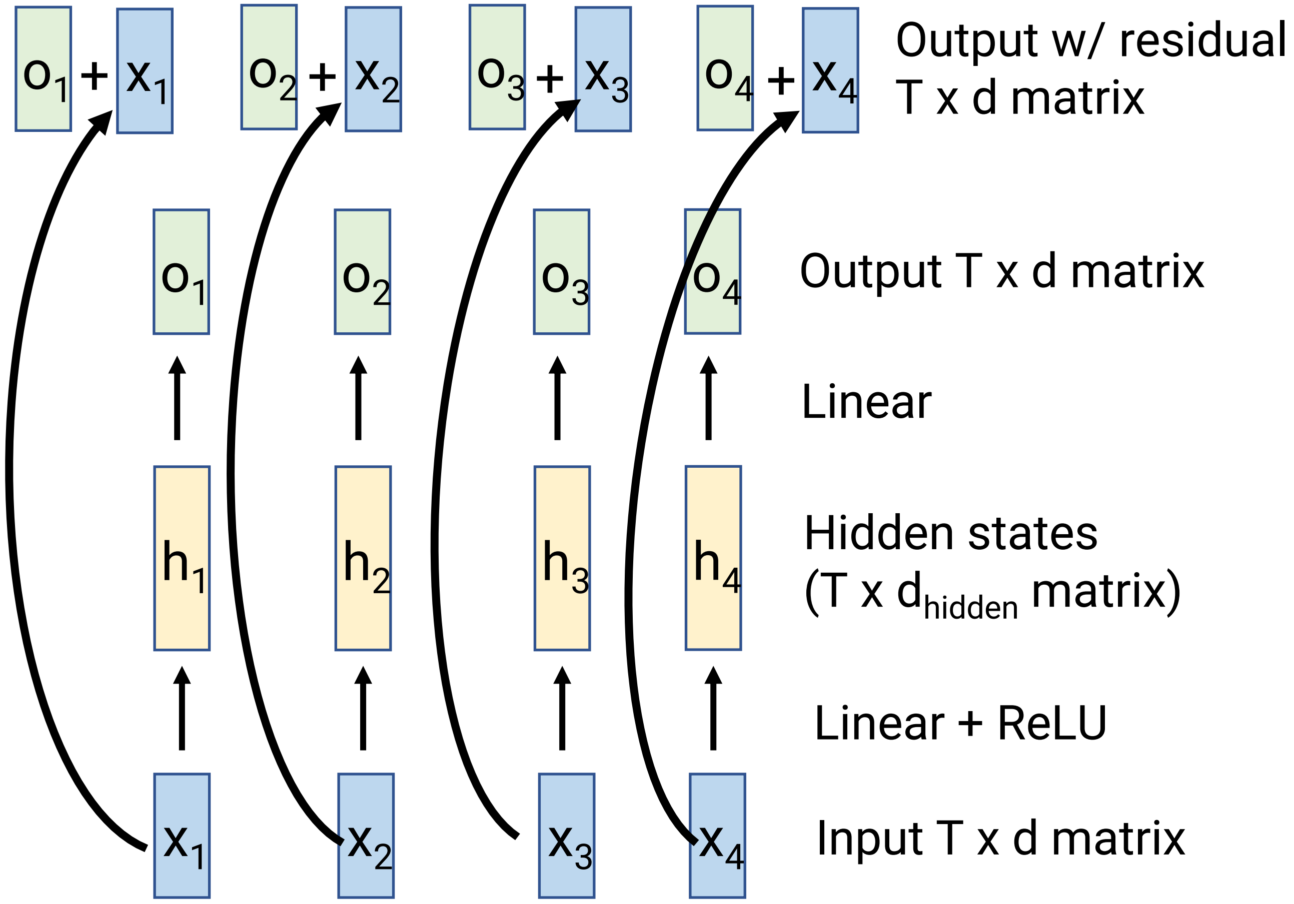
Scaled dot product attention



- 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
  - Also reduces vanishing gradient issues
  - Think of O as how much we **change** the previous value
- Then, we add "Layer Normalization" to prevent very big or very small values

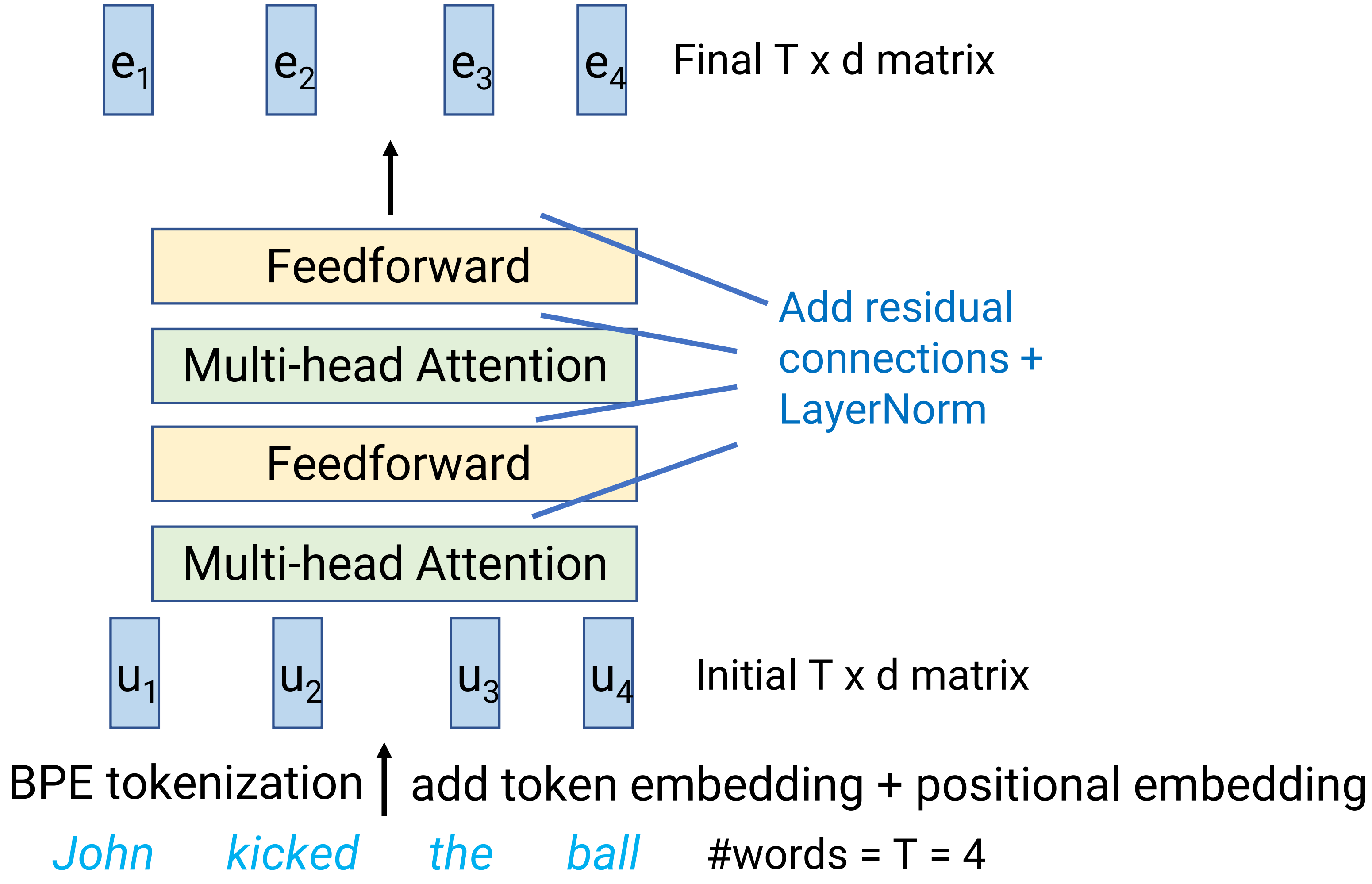


Byte Pair Encoding

- 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



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