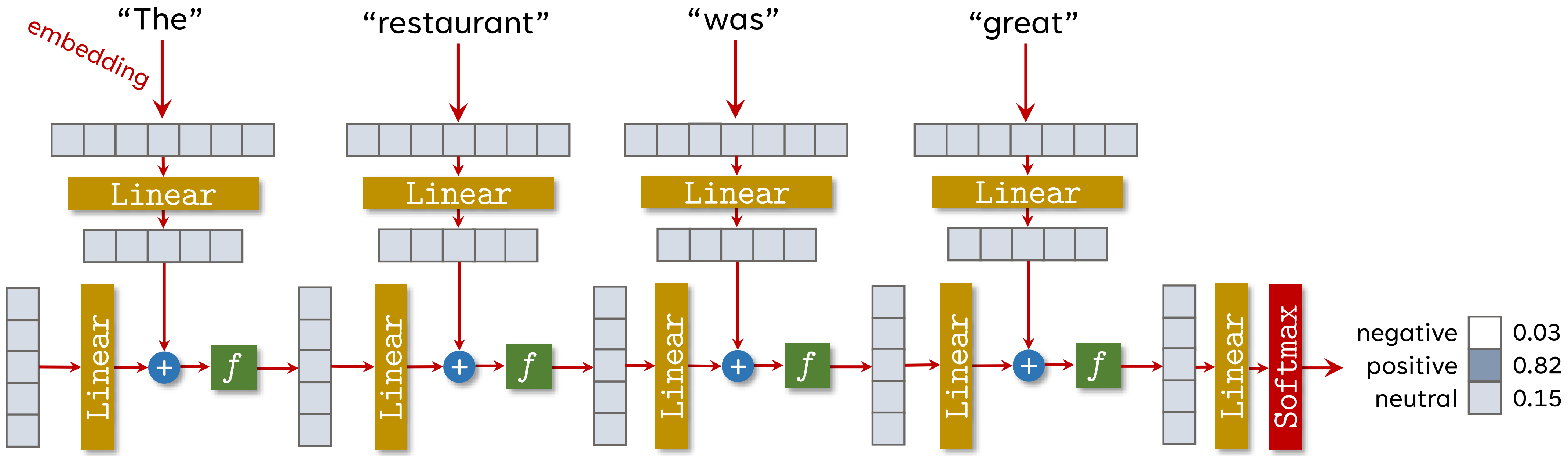
CS 490: NATURAL LANGUAGE PROCESSING

Dan Goldwasser, Abulhair Saparov

Lecture 8: Attention and Transformers

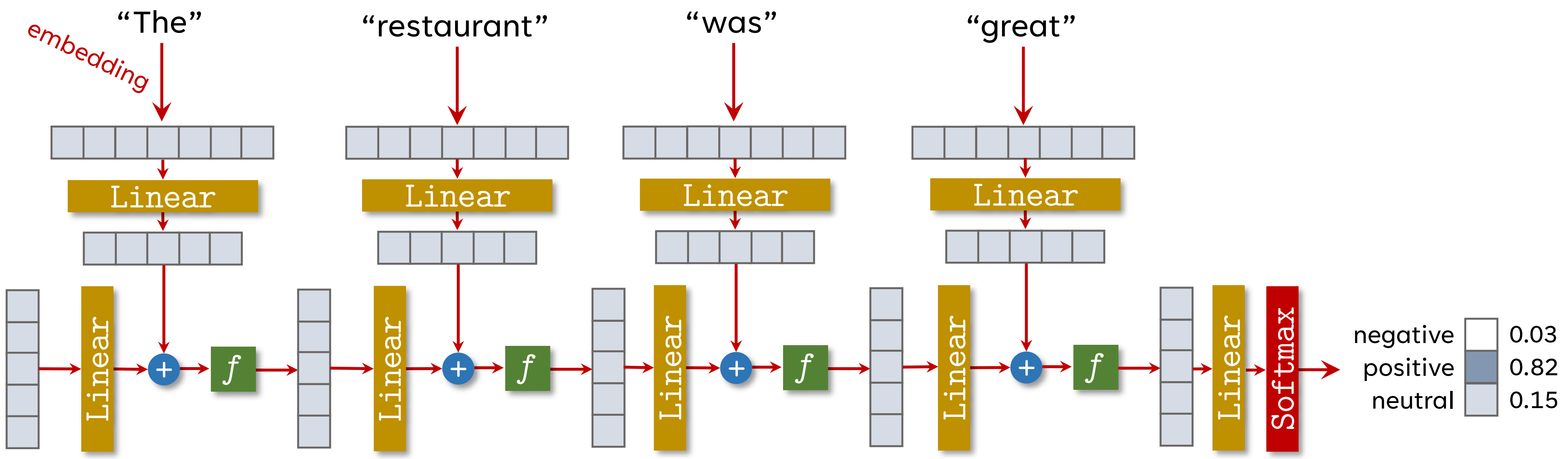
RECAP FROM PREVIOUS LECTURE

- We discussed the recurrent neural network (RNN) architecture.



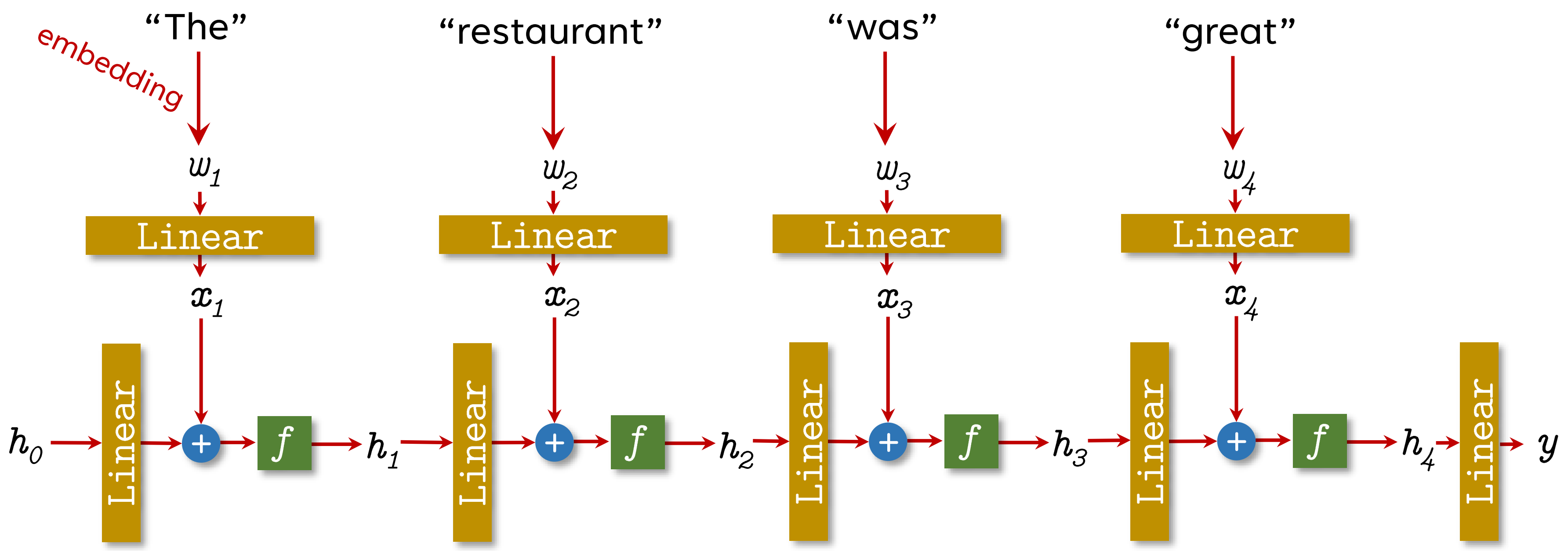
RECAP FROM PREVIOUS LECTURE

- Models the sequential (word-by-word) processing nature of human language processing.



RECAP FROM PREVIOUS LECTURE

- Models the sequential (word-by-word) processing nature of human language processing.



VANISHING/EXPLODING GRADIENTS

- Suppose we have a long RNN (lots of tokens).

$$y = g_n(g_{n-1}(\ldots g_2(g_1(x_1, W_1))\ldots))$$

- Each g_i is an RNN "unit", written more simply.
- x_1 is the first word, and W_1 is the weight matrix in the linear layer after x_1.
- What is the gradient with respect to W_1?

$$\begin{aligned}\nabla_{W_1} y &= g_n'(\ldots) \cdot \nabla_{W_1} g_{n-1}(\ldots) \\ &= g_n'(\ldots) \cdot g_{n-1}'(\ldots) \cdot \nabla_{W_1} g_{n-2}(\ldots) \\ = \ldots &= g_n'(\ldots) \cdot g_{n-1}'(\ldots) \cdot \ \ldots \ \cdot g_2'(\ldots) \cdot g_1'(\ldots) \cdot x_1^T\end{aligned}$$

VANISHING/EXPLODING GRADIENTS

- Note that this is a product containing many terms.
- If the terms are > 1, their product will grow exponentially in n.
- If the terms are < 1, their product will shrink to 0 exponentially in n.
- This is called the exploding or vanishing gradient problem.
- This is also an issue for very deep networks (containing many layers).

$$\begin{aligned}\nabla_{W_1}\mathbf{y} &= g_n'(\ldots) \cdot \nabla_{W_1} g_{n-1}(\ldots) \\ &= g_n'(\ldots) \cdot g_{n-1}'(\ldots) \cdot \nabla_{W_1} g_{n-2}(\ldots) \\ = \ldots &= g_n'(\ldots) \cdot g_{n-1}'(\ldots) \cdot \ . \ . \ . \ \cdot g_2'(\ldots) \cdot g_1'(\ldots) \cdot x_1^{\mathrm{T}}\end{aligned}$$

LONG-TERM DEPENDENCIES

- Suppose we have some NLP task where the input is a document or string of words.
- If the expected output for this task requires information about words that occur far from the end of the input,

  This is a long-term dependency (or a long-range dependency).
- Why are they important in natural language?

LONG-TERM DEPENDENCIES

- Consider the following sentences:
  - “`The cat, which was very hungry, chased the mouse.`”
  - “`The children sitting under the tree know the farmers.`”
- Suppose we ask questions such as:
  - Who chased the mouse?
  - Who knows the farmers?

LONG-TERM DEPENDENCIES

- Consider the following sentences:
  - “`Without her contributions would be impossible.`”
    - What is the subject of this sentence?
    - What is the main verb?
  - “`The old man the boat.`”
  - “`I convinced her children are noisy.`”
    - (I convinced her that children are noisy)
- These are called garden path sentences.

LONG-TERM DEPENDENCIES

- Long-term dependencies appear in almost every NLP task.
- Coreference resolution:
  - “`I couldn’t fit the trophy in the bag because it was too big.`”
- Translation:
  - Consider translating from an SVO language into an VSO language.
  - E.g., English into Arabic.
  - “The cat, which was very hungry, chased the mouse”
  - [“chased”] [“the cat, which was very hungry”] [“the mouse”]

LONG-TERM DEPENDENCIES

- It is possible for an RNN to compute long-term dependencies.
- For example, we can construct an RNN to memorize the input sequence:
  - Input: word embedding w_t and previous state h_{t-1}
  - Suppose hidden dimension is Nd
    - where N is the sequence length, and d the embedding dimension.
  - The linear layer on the word simply takes the embedding of w_t and appends many zeros to the end of it.
    - The first d elements of the output are the embedding of w_t, and the last $(N-1)d$ elements are zero.
  - The linear layer on the hidden state rotates the dimensions of h_{t-1} by d positions.

LONG-TERM DEPENDENCIES

- It is possible for an RNN to compute long-term dependencies.
- For example, we can construct an RNN to memorize the input sequence:
  - Input: word embedding w_t and previous state h_{t-1}
  - This construction is simply concatenating each word embedding into a long hidden state vector (which is initially zero).
  - Therefore, the hidden state is able to keep track of long-term dependencies.
    - It has memorized the entire input!

EXPRESSIVENESS VS LEARNABILITY

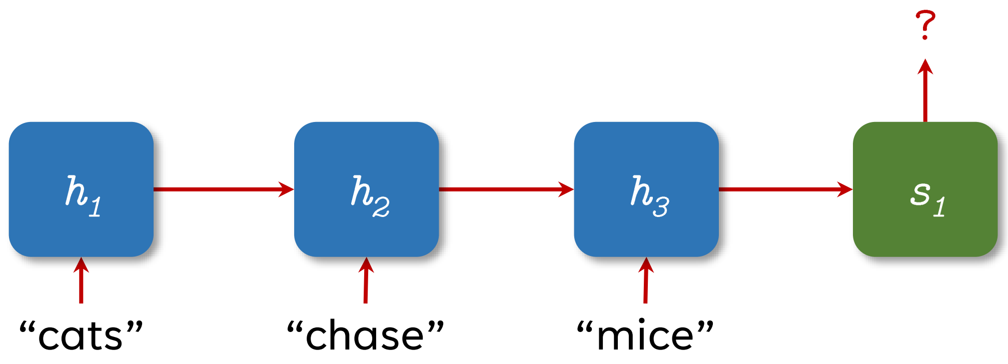
- The problem is that this network is *difficult to learn*.
  - Due to the vanishing/exploding gradients problem.
- But this is a common problem with highly expressive models.
  - E.g., MLPs, RNNs, transformers
  - These models are able to express a vast number of algorithms.
  - But the question of whether they can learn all such algorithms from data is entirely separate.
  - Oftentimes, there are algorithms that these models can express, but have difficulty learning.

LONG-TERM DEPENDENCIES

- How do we solve the problem of long-term dependencies?
- By design RNNs tend to depend more on more recent inputs.
- In addition, RNNs can only store a limited amount of information in the hidden state vector.
  - The amount of information that can be stored is independent of the length of the input.
- But is there a fundamentally different way to model long-term dependencies?
  - What if we relax the assumption that the next hidden state only depends on the previous hidden state?

ATTENTION

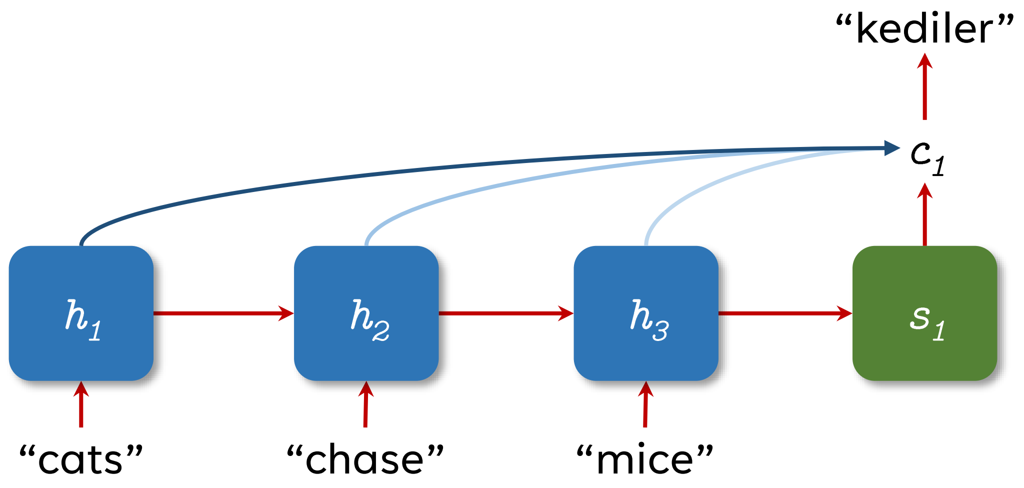
- The attention mechanism was designed to more explicitly model dependencies between words that are very far apart.
- Suppose we have an RNN performing a machine translation task.



[Bahdanau et al., 2015]

ATTENTION

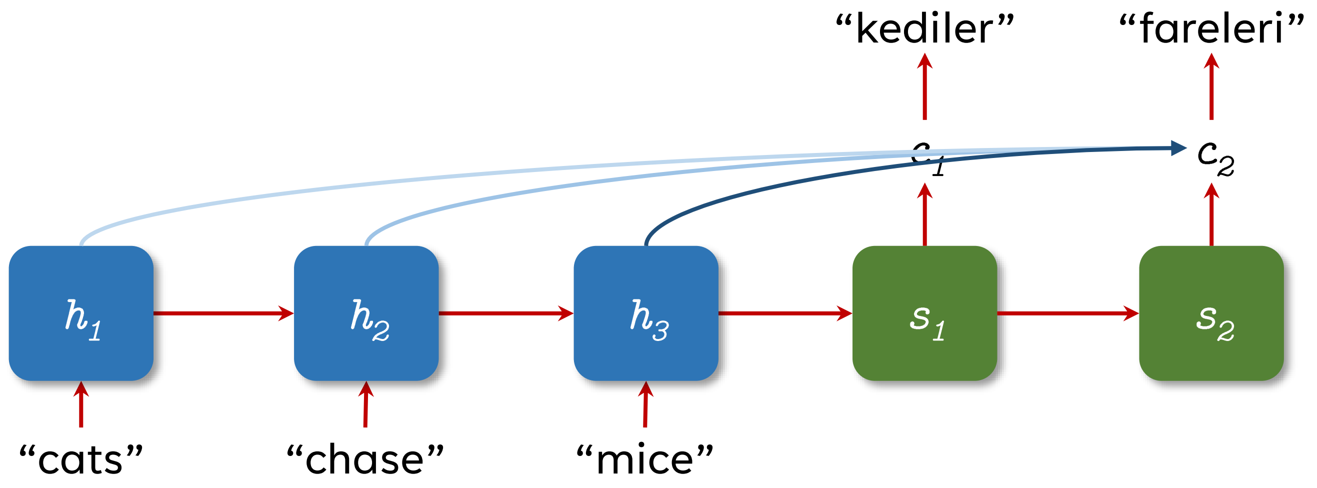
- Basic idea: Compute a linear sum of the vectors corresponding to the input.
- The weights in this linear sum are called attention weights.



[Britz, Goldie, Lyong, and Le 2017]

ATTENTION

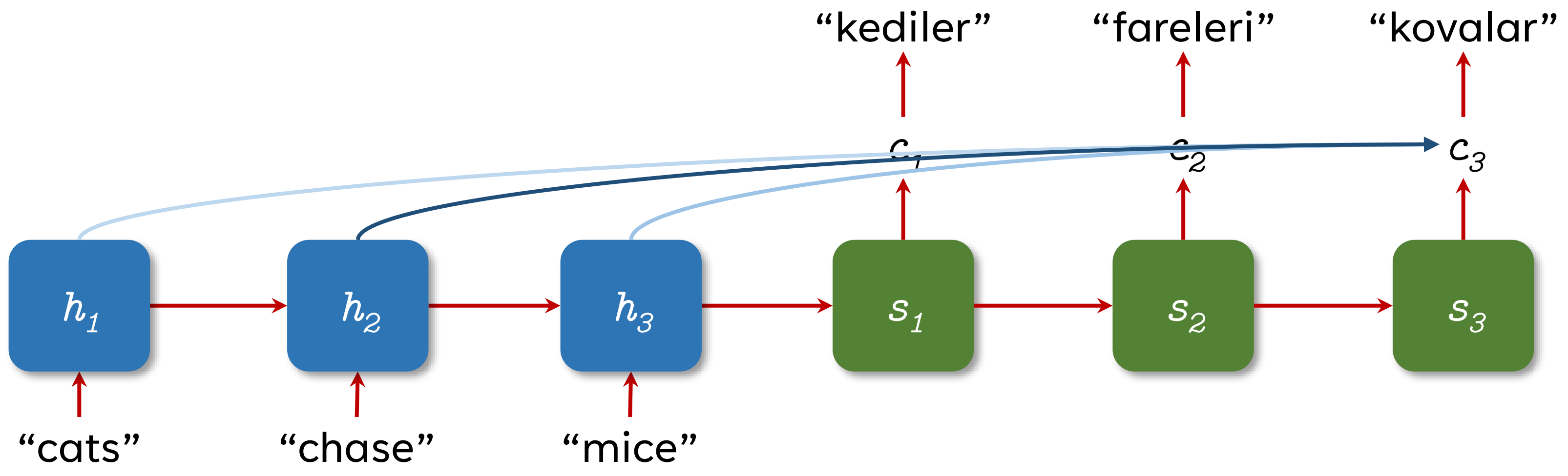
- Basic idea: Compute a linear sum of the vectors corresponding to the input.
- The weights in this linear sum are called attention weights.



[Britz, Goldie, Lyong, and Le 2017]

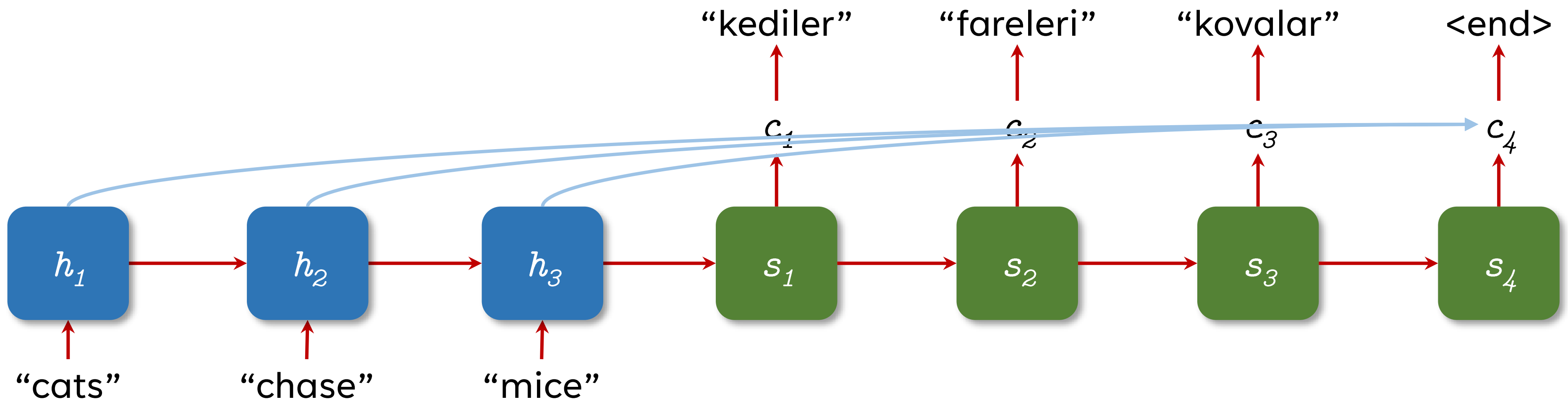
ATTENTION

- Basic idea: Compute a linear sum of the vectors corresponding to the input.
- The weights in this linear sum are called attention weights.



ATTENTION

- Basic idea: Compute a linear sum of the vectors corresponding to the input.
- The weights in this linear sum are called attention weights.
- The entire input no longer needs to be encoded in the hidden state.



[Britz, Goldie, Lyong, and Le 2017]

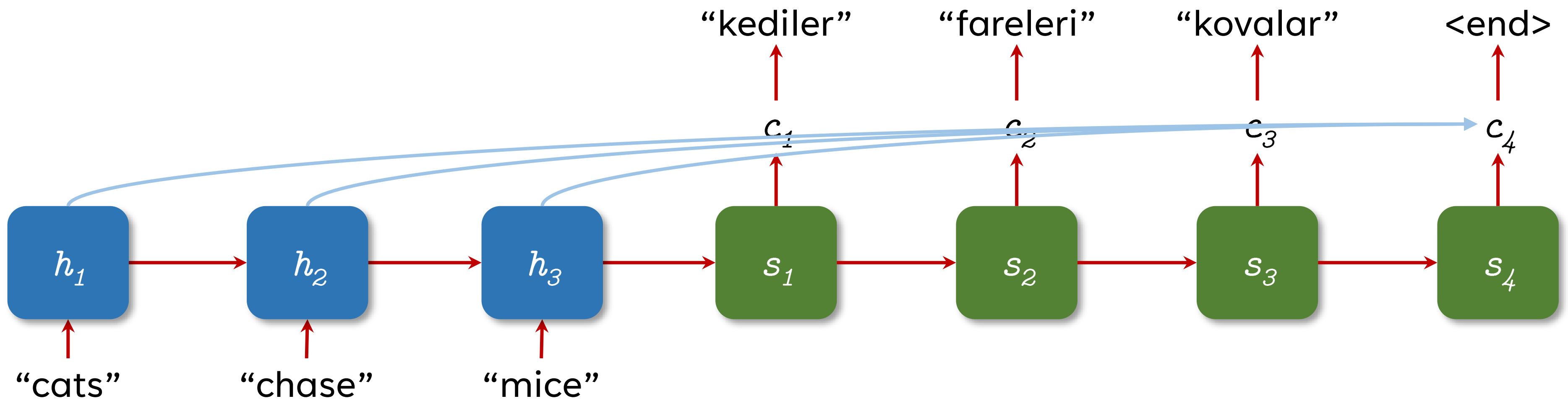
ATTENTION

$c_i = \sum_{j=1}^{n} a_{ij} h_j$ where a_{ij} are attention weights.

$a_{ij} \propto score(s_i, h_j)$

Note the "∝" (proportional to) symbol.

For each i, we first compute the scores between s_i and all h_j, and then normalize.



ATTENTION

$c_i = \sum_{j=1}^{n} a_{ij} h_j$ where a_{ij} are attention weights.

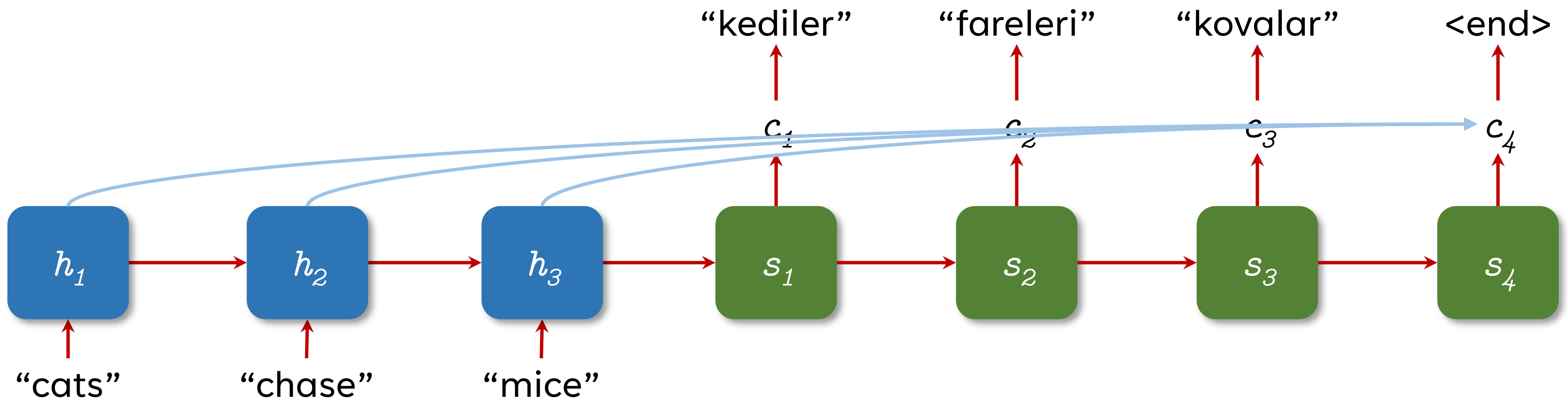
$a_{ij} \propto score(s_i, h_j)$

$score(x,y) = x^{\mathrm{T}} y$

$score(x,y) = (W_1 x)^{\mathrm{T}} (W_2 y)$

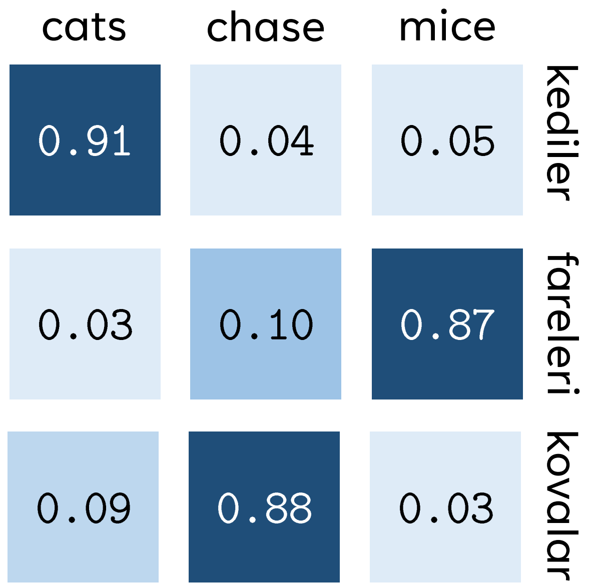
etc...

There are lots of options for this scoring function.
Notice everything is still differentiable, so we can still use gradient descent for training.



ATTENTION

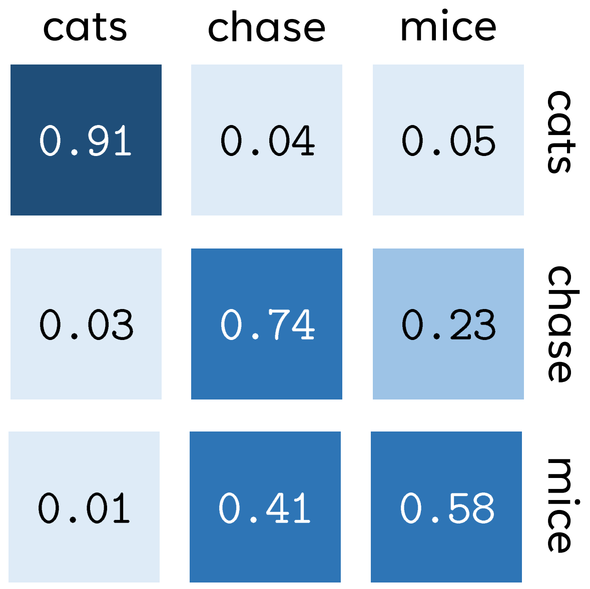
- We can also view the attention weights as a matrix.
- This is an example of cross-attention:
    - Attention weights are computed between two sequences.



cats	chase	mice	
0.91	0.04	0.05	kediler
0.03	0.10	0.87	fareleri
0.09	0.88	0.03	kovalar

SELF-ATTENTION

- Self-attention: Attention weights are computed between tokens of the same sentence.



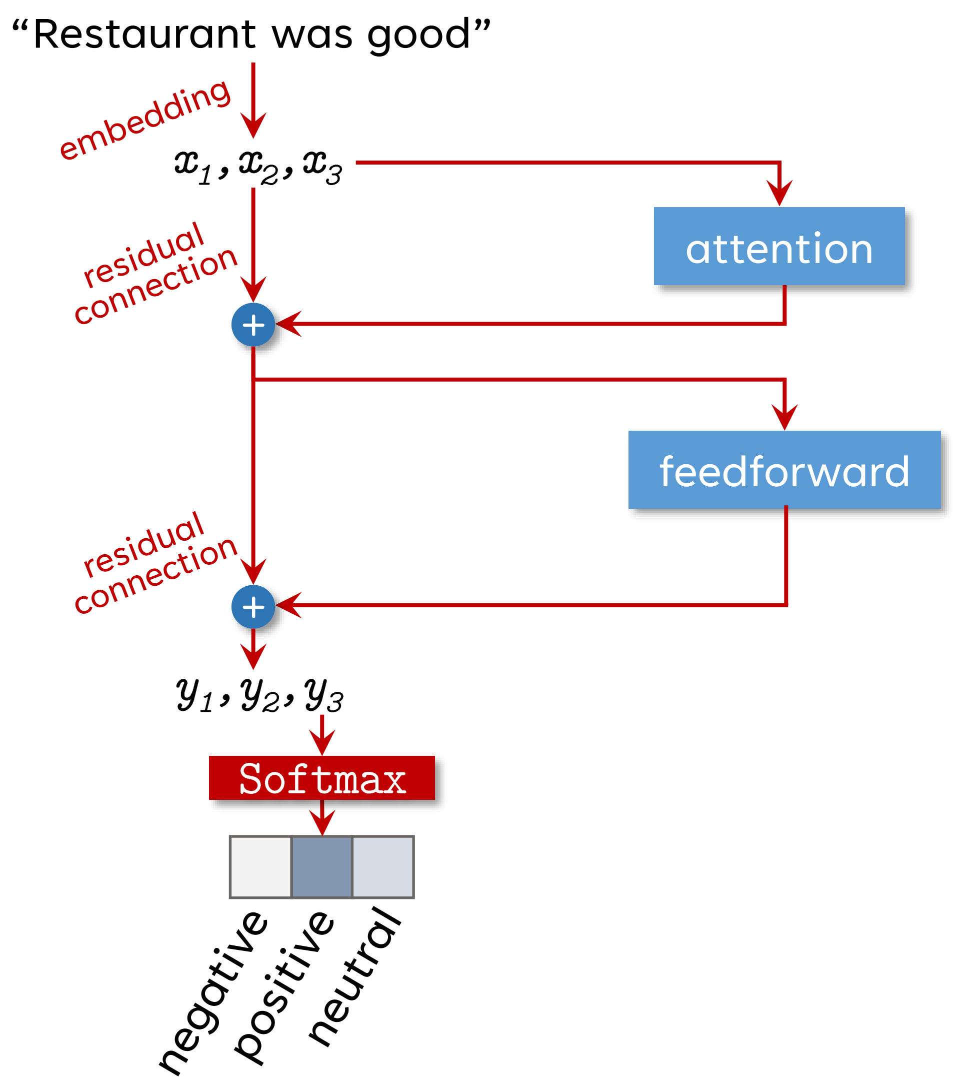
	cats	chase	mice
cats	0.91	0.04	0.05
chase	0.03	0.74	0.23
mice	0.01	0.41	0.58

TRANSFORMER

- RNNs suffer from poor parallelizability.
- We must compute each successive hidden state sequentially.
- What if we removed the recurrent aspect, and we model inter-word dependencies entirely via the attention mechanism?
- The transformer architecture is one way to do this (Vaswani et al., 2017).

TRANSFORMER

- We embed each word into a d_{model}-dimensional vector.
- We can make predictions from any output vector y_i.
- One common choice is to make predictions from the last y_i.
- Why do we have residual connections?



Suppose $g(x)$ performs the operations of a single transformer layer *without* residual connections.

Suppose x is a function of w, a weight from an earlier layer.

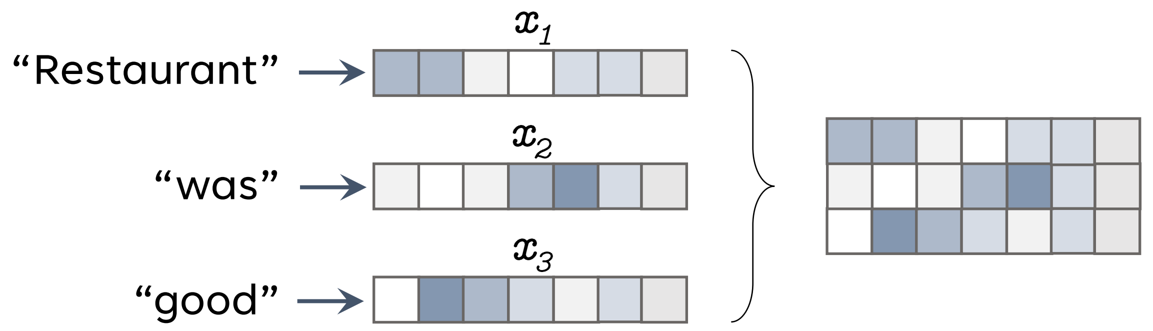
What is the gradient $\frac{\partial}{\partial w}g(x)$?

$$\frac{\partial}{\partial w}g(x) = g'(x)\frac{\partial x}{\partial w}$$

If we had a residual connection, each transformer layer would perform $x + g(x)$.

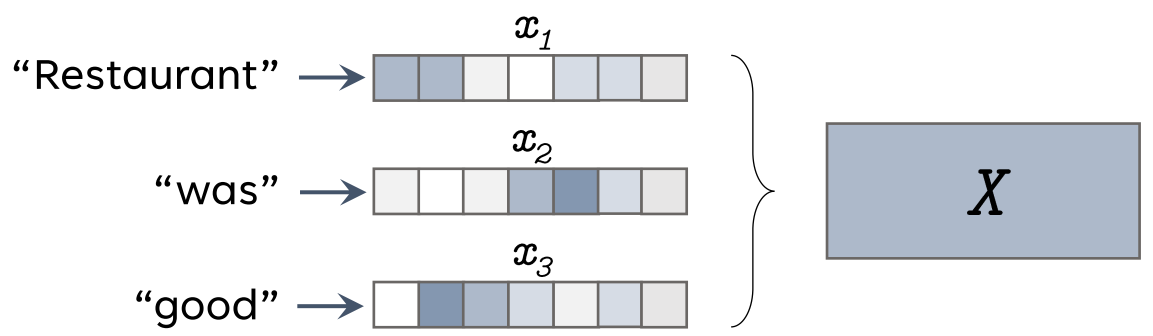
$$\frac{\partial}{\partial w}(g(x) + x)$$
$$= g'(x)\frac{\partial x}{\partial w} + \frac{\partial x}{\partial w}$$
$$= (g'(x) + 1)\frac{\partial x}{\partial w}$$

SIMPLIFYING NOTATION



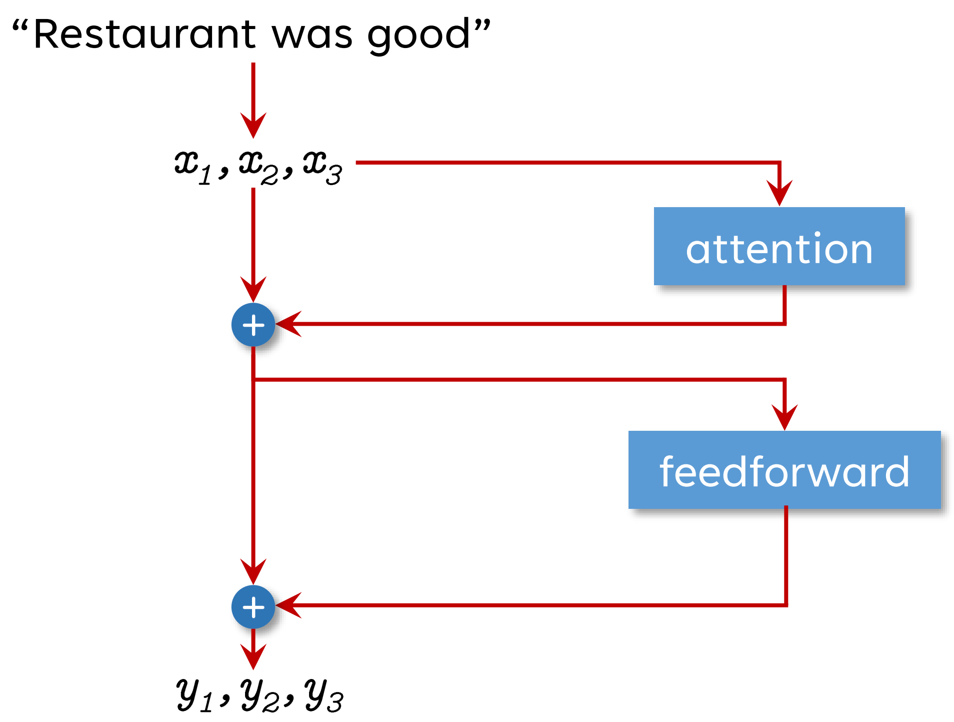
This matrix is the input to the transformer
(and to the attention and FF layers)

SIMPLIFYING NOTATION

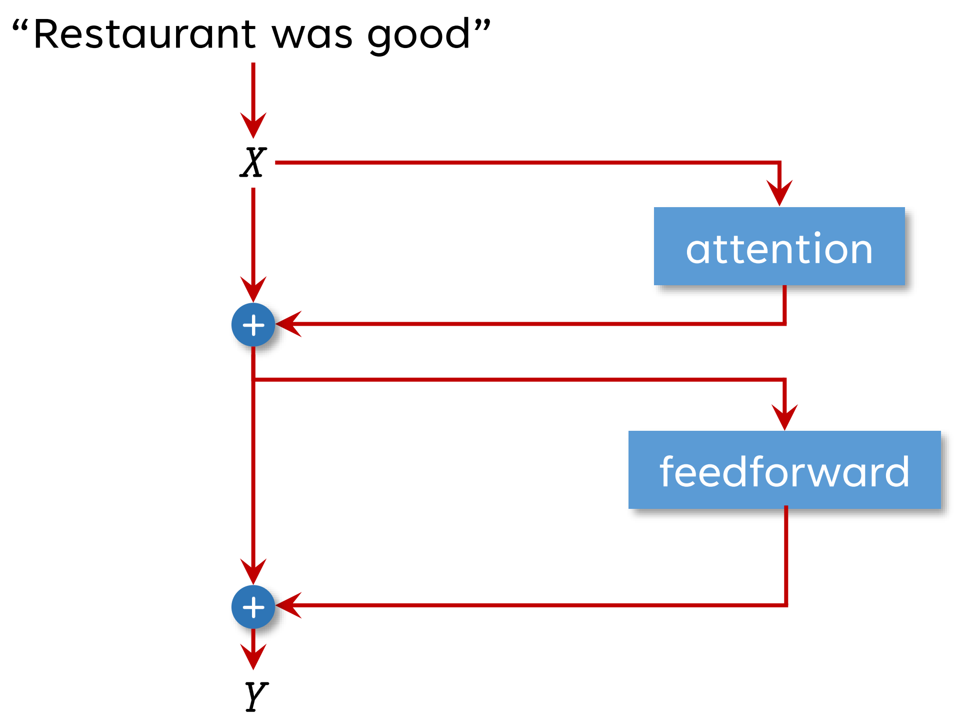


This matrix is the input to the transformer
(and to the attention and FF layers)

TRANSFORMER



TRANSFORMER



FEEDFORWARD LAYER

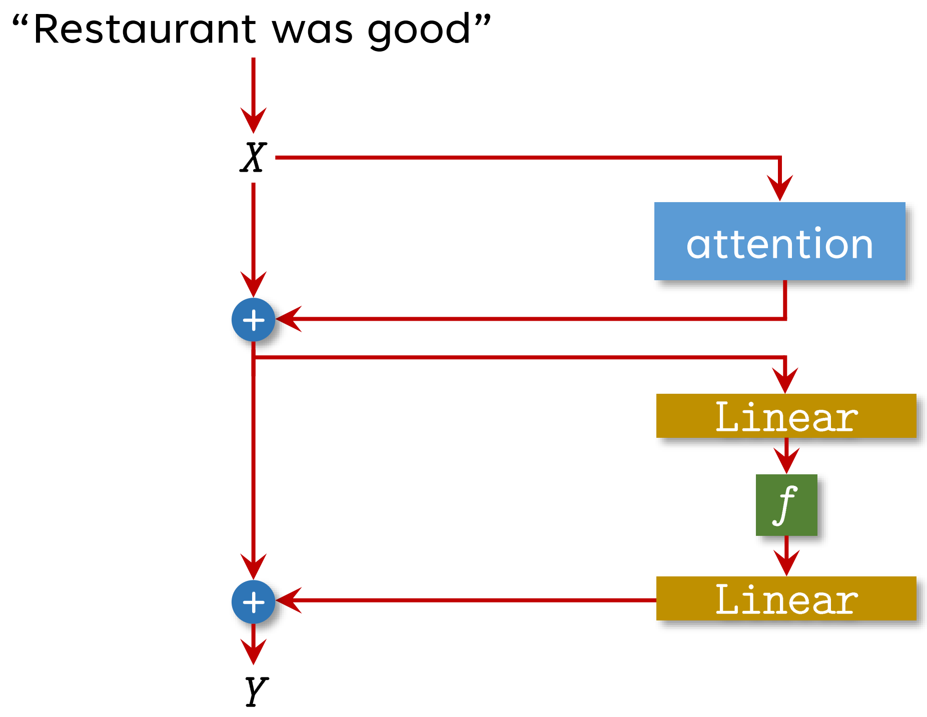
$$X^{out} = W_2 f(W_1 X^{in} + b_1) + b_2$$

Where W_1 is a weight matrix with dimension $d_{ff} \times d_{model}$,

And W_2 is a weight matrix with dimension $d_{model} \times d_{ff}$.

So the nonlinear operation is performed in a higher-dimensional space,

before being projected back to a d_{model}–dimensional output.



FEEDFORWARD LAYER

$$X^{out} = W_2 f(W_1 X^{in} + b_1) + b_2$$

Important: this FF operation is performed on each of the input vectors independently:

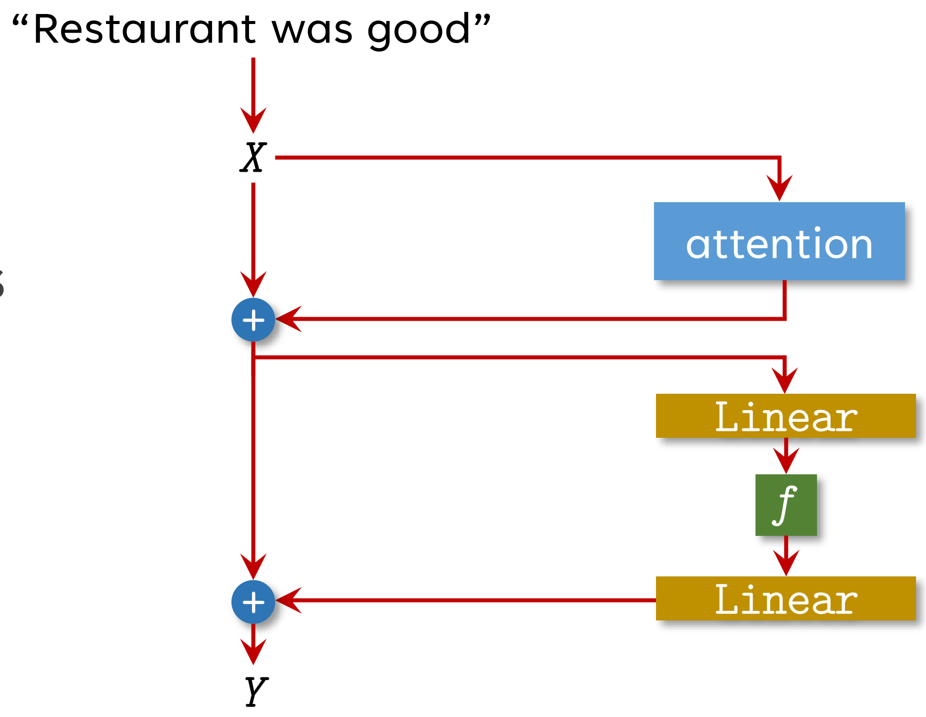
$$x_1^{out} = W_2 f(W_1 x_1^{in} + b_1) + b_2$$

$$x_2^{out} = W_2 f(W_1 x_2^{in} + b_1) + b_2$$

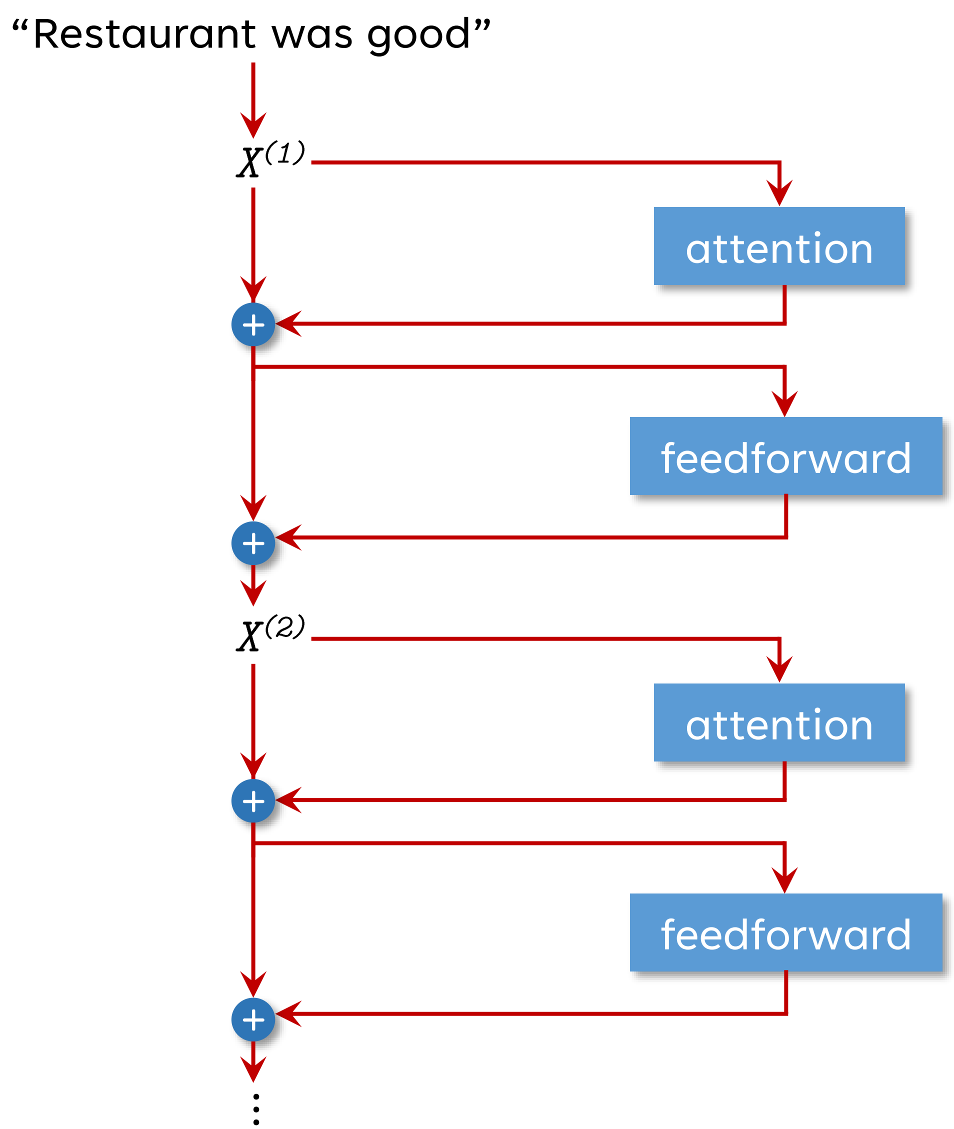
$$x_3^{out} = W_2 f(W_1 x_3^{in} + b_1) + b_2$$

etc…

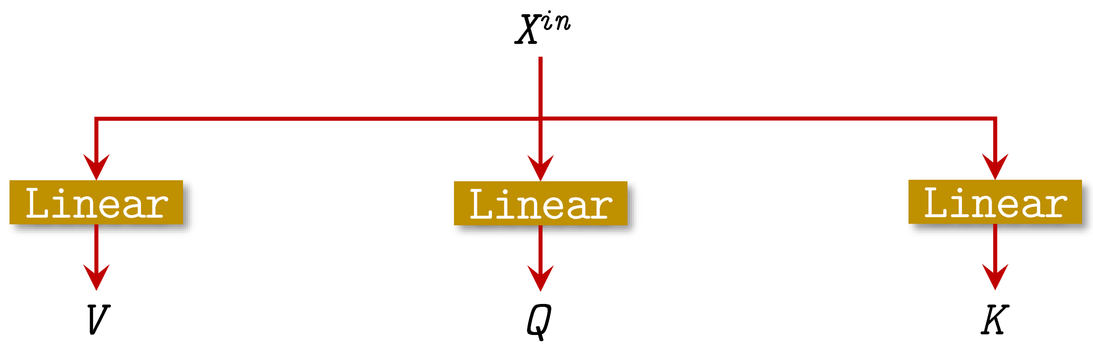
No information is shared between token embeddings in the FF layer.



ADDING MORE LAYERS

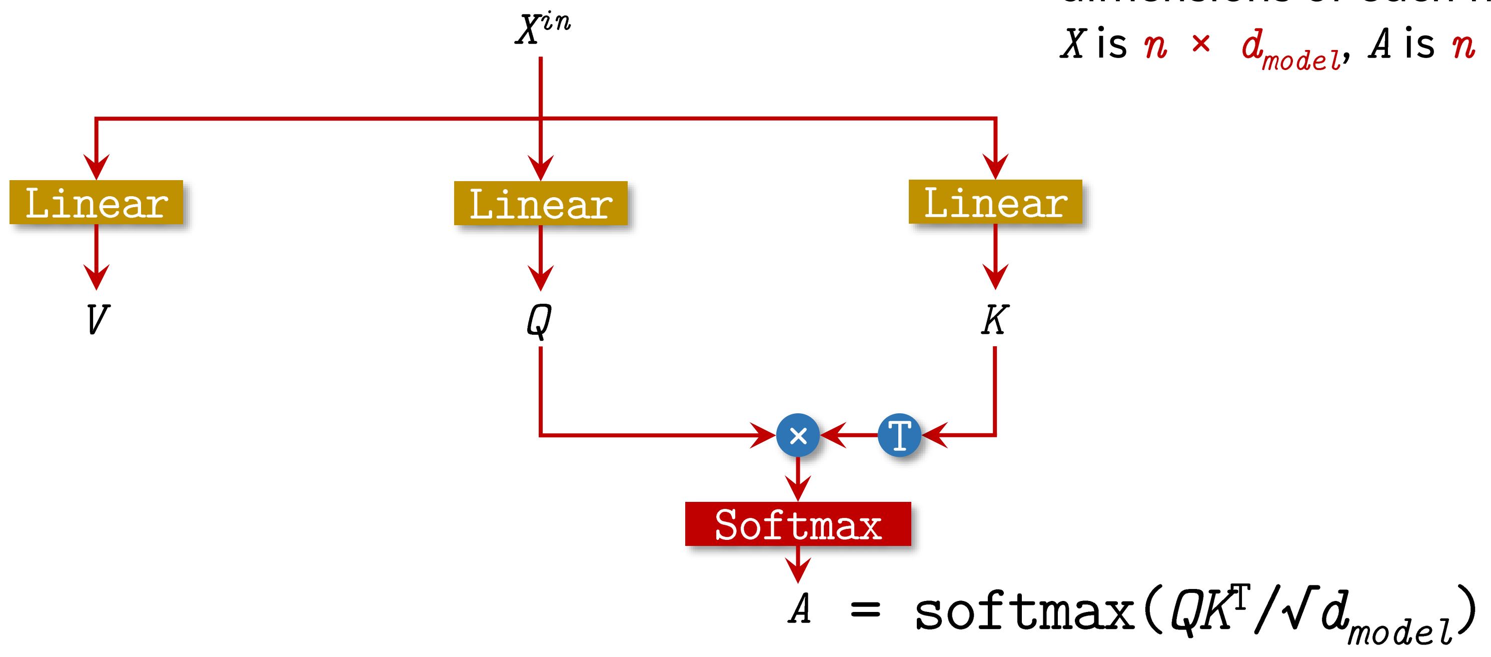


ATTENTION LAYER



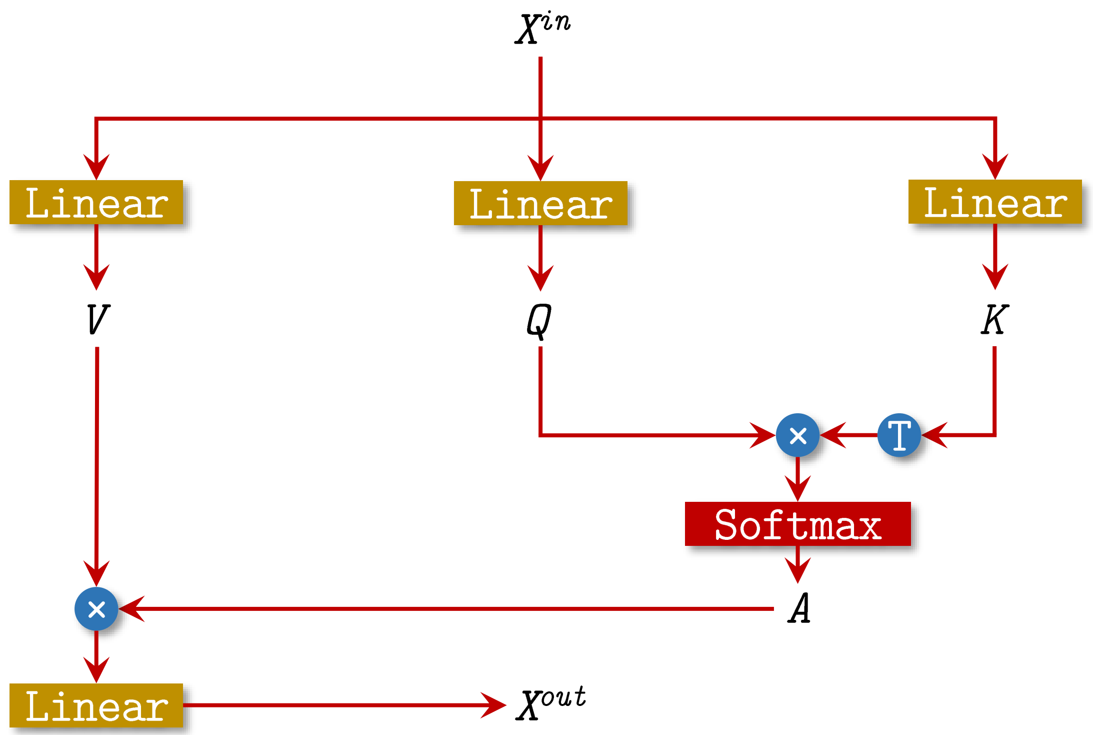
ATTENTION LAYER

It helps to think about the dimensions of each matrix: X is $n \times d_{model}$, A is $n \times n$, etc.



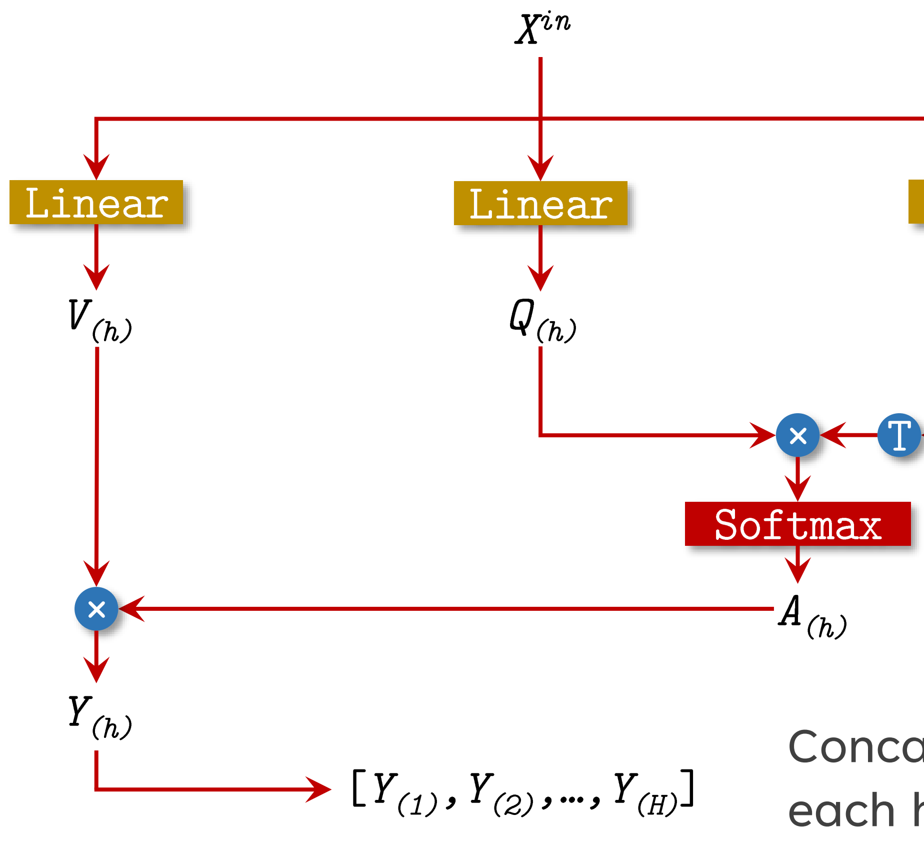
The attention matrix A describes the dependencies between different tokens of the input.
E.g., $A_{i,j}$ describes how strongly the token at index i depends on the token at index j.
We can choose to mask the attention matrix: we force $A_{i,j} = 0$ if $j > i$.
That is, we only allow each token to depend on previous tokens, but not future tokens.

ATTENTION LAYER



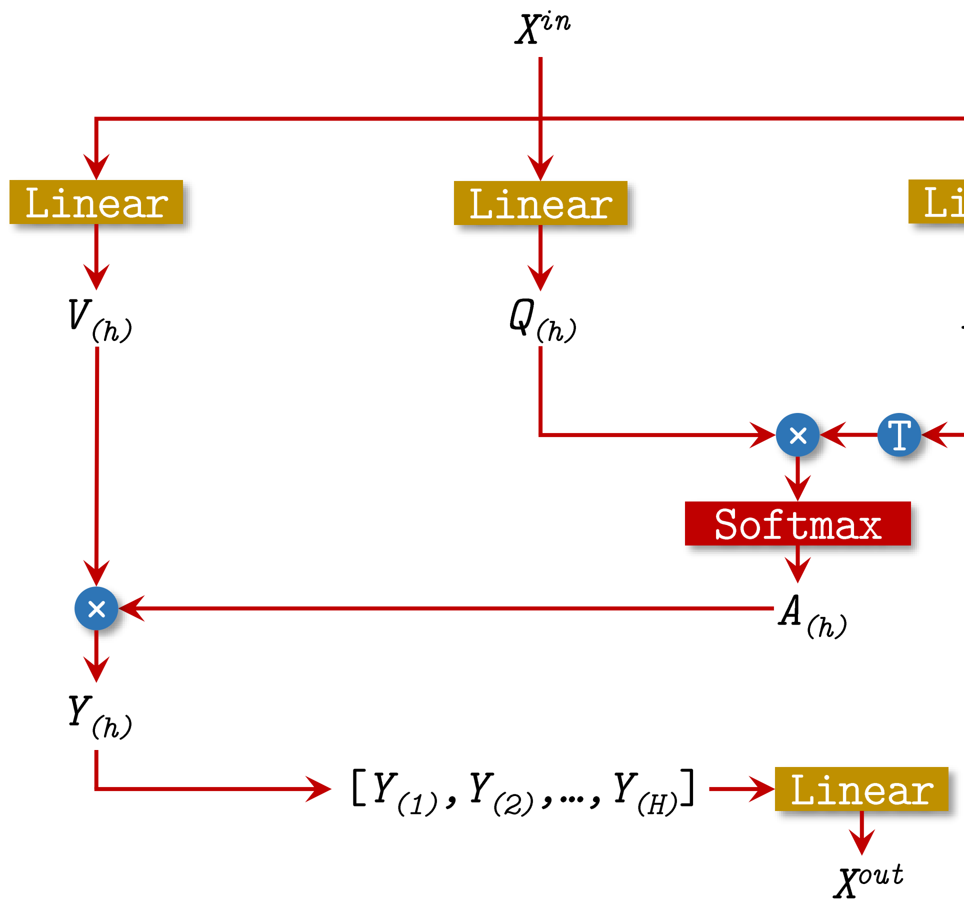
This is the output of the attention layer.
But recall that due to the residual connections, the actual output is $X^{in} + X^{out}$.

MULTI-HEAD ATTENTION



Concatenate the outputs from each head into one large matrix.

MULTI-HEAD ATTENTION



The resulting matrix is too large, since we need to add it back to X^{in}. It's dimension is $n \times Hd_{model}$. So we use a linear layer to resize it. Due to the residual connection, we add the output back to the input: $X^{in} + X^{out}$

WHY MULTIPLE HEADS?

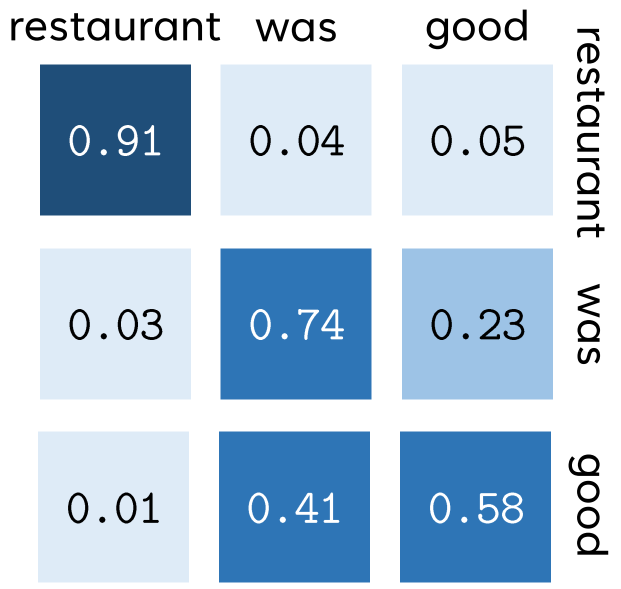
- Different attention heads can perform different computations.
- For example one attention head can compute syntactic relations:
    - “I run a small business” vs “I went for a run”
    - To compute the part-of-speech of “run”, it helps to attend the word immediately before: “I” vs “a”.
    - “a run” indicates that “run” is a noun.
    - “I run” indicates that “run” is a verb.
- A second attention head can compute semantic information:
    - What is the subject of the run?
    - In both examples above, the subject is “I”.

WHY MULTIPLE HEADS?

- Multiple heads can save us from needing more layers to perform complex computations.
- More heads and fewer layers -> More parallelizable!

CAUSAL MASK

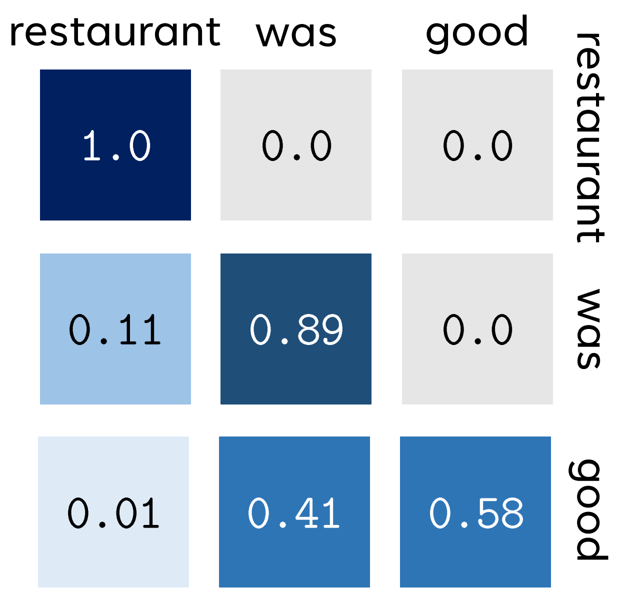
- Example of attention matrix without mask:



	restaurant	was	good
restaurant	0.91	0.04	0.05
was	0.03	0.74	0.23
good	0.01	0.41	0.58

CAUSAL MASK

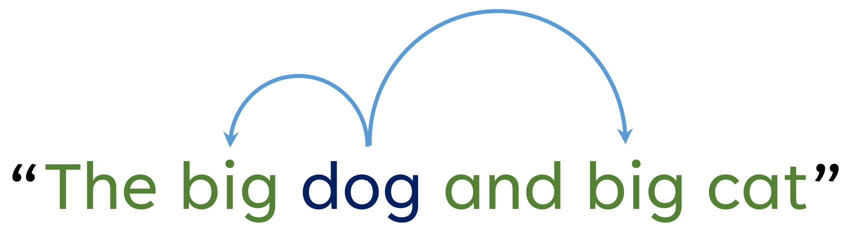
- Example of attention matrix with a causal mask:



	restaurant	was	good
restaurant	1.0	0.0	0.0
was	0.11	0.89	0.0
good	0.01	0.41	0.58

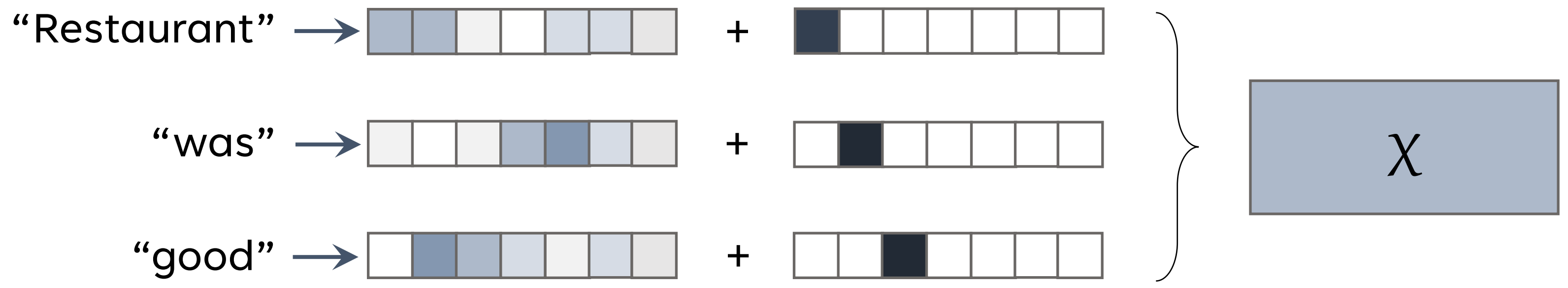
POSITIONAL EMBEDDING

- Suppose the input X to the attention layer has no position information (it only has word information).
- And suppose we have no causal mask.
- The attention layer is not able to compute the relative positions of tokens:
  - E.g. it can’t determine which token immediately follows any other token.
  - Suppose the word “dog” has high attention weight with “big”.
  - Since the embeddings of both “big” words are identical, the attention weight between dog and both big’s are the same.



POSITIONAL EMBEDDING

- Thus, position information is explicitly added to the embeddings:
- There are many kinds of positional embeddings.
- Token embeddings and positional embeddings can be summed, multiplied, or concatenated, etc.
    - Lots of ways to incorporate position information into embeddings.



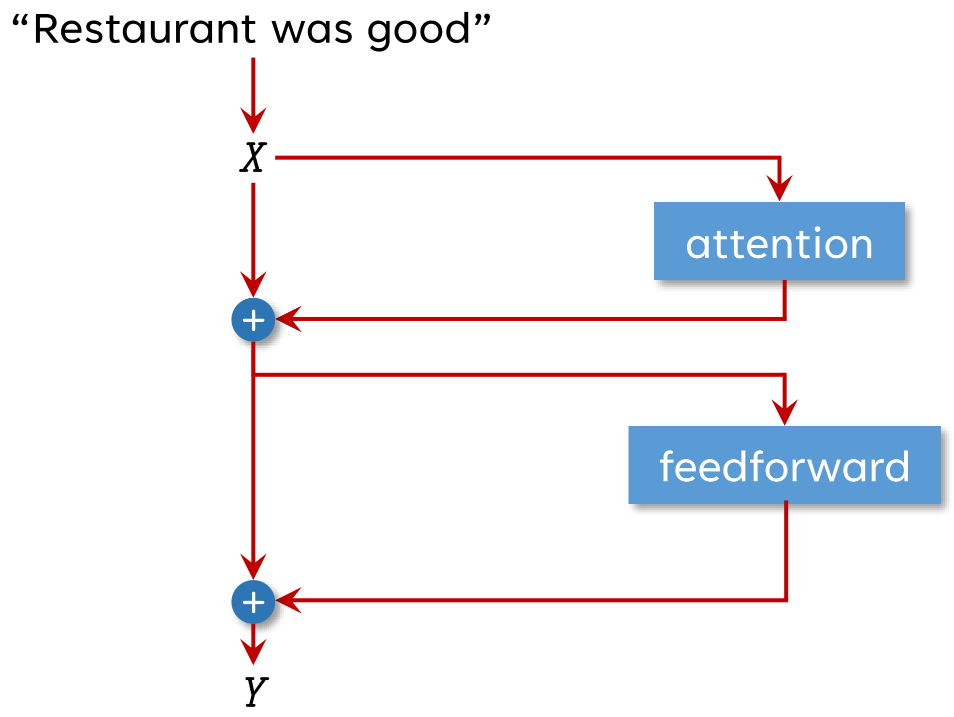
LAYER NORMALIZATION

- Suppose we are training a transformer on a classification task, so we have a `softmax` operation at the end of the network.
- Also suppose the input embeddings have large magnitude,
- It's very likely that the magnitude of the embeddings stays large throughout the transformer layers, up to the last softmax operation.
- Recall that the derivative of the softmax is close to zero if the input is a large positive or negative value.
  - Hint: The logistic function (i.e., sigmoid) is equivalent to softmax in two dimensions.
- Thus, in this example, the gradient would be very close to zero, and training would be extremely slow.

LAYER NORMALIZATION

- Thus, transformers with many layers can also sometimes suffer from vanishing gradients.
- To avoid this, transformers use layer normalization.
    - Idea: Keep the activations close to 0 and not too negative or too positive.

LAYER NORMALIZATION

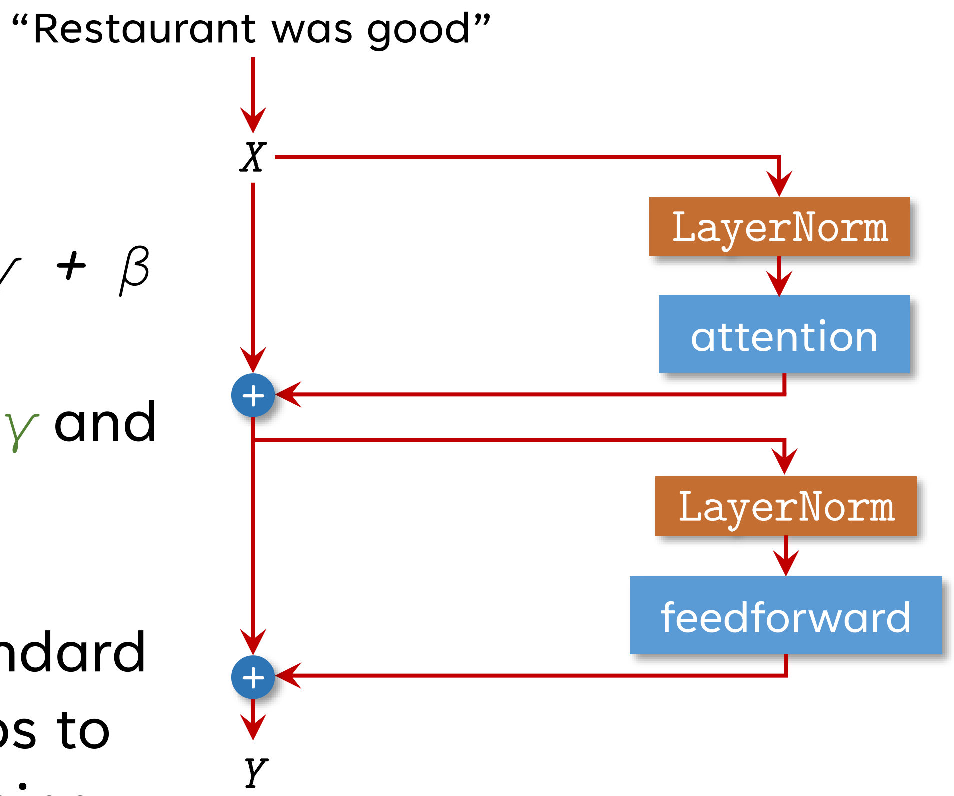


LAYER NORMALIZATION

$$\mathtt{LayerNorm}(x_i) = \frac{x_i - avg(x_i)}{\sqrt{var(x_i) + \varepsilon}} \circ \gamma + \beta$$

Where ε is a small fixed constant, γ and β are vectors of learnable weights.

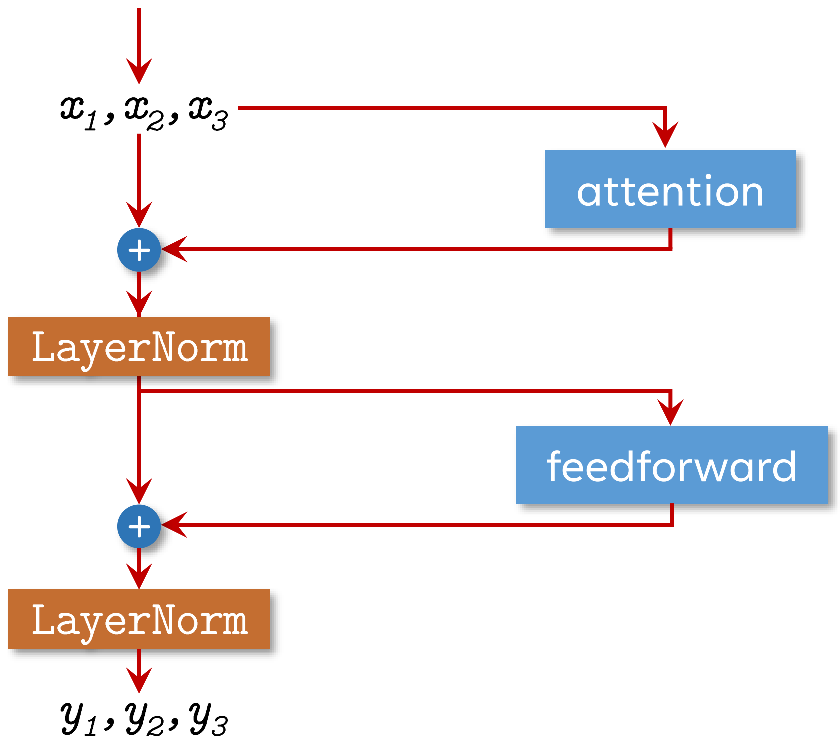
Since we scale the input by its standard deviation, layer normalization helps to prevent the activations from attaining very large positive or negative values.



POST-LAYER NORMALIZATION

The original transformer paper used post-layer normalization.

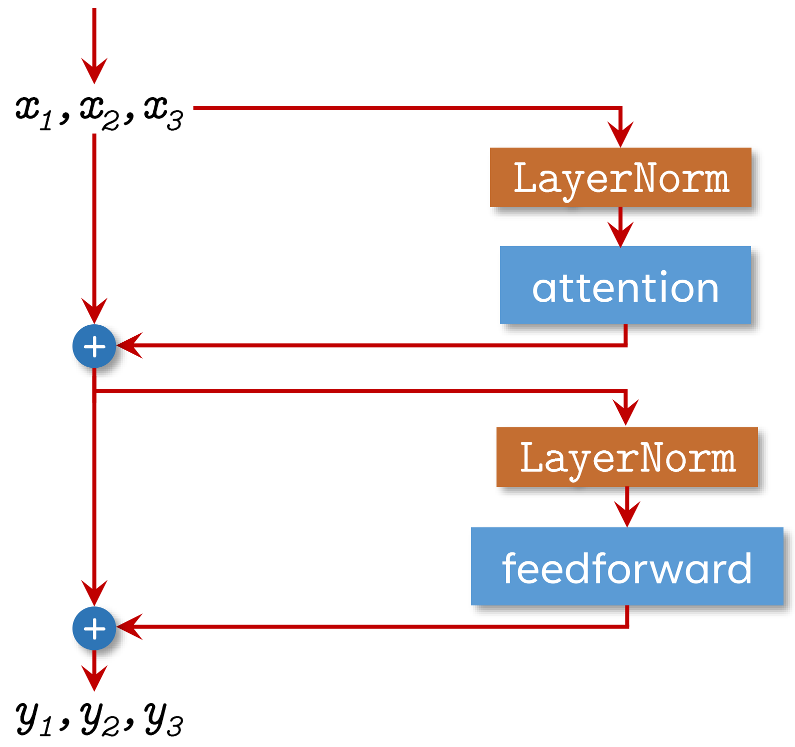
Layer normalization was applied on the residual stream (i.e., *after* residual connection).



[Vaswani et al., 2017]

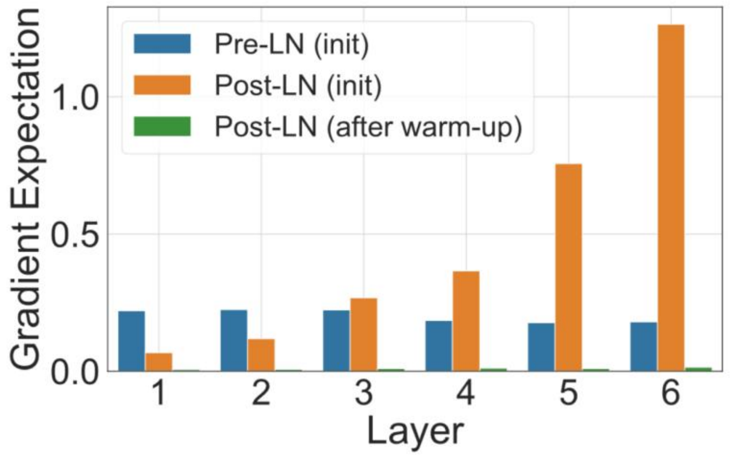
PRE-LAYER NORMALIZATION

Xiong et al., 2020, proposed moving the layer normalization *before* the attention and FF blocks.

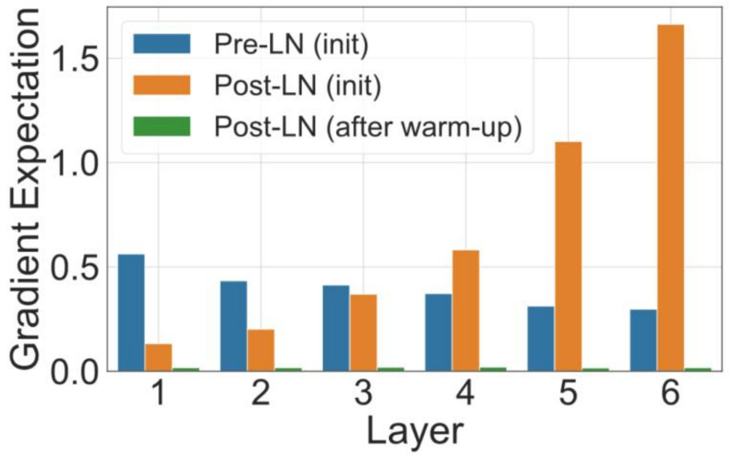


PRE-LAYER NORMALIZATION

- Xiong et al., 2020, showed that the magnitudes of the gradients are more uniform across layers when using pre-layer normalization.
- Hypothesis: Learning occurs at a more uniform rate across layers when using pre-layer norm.



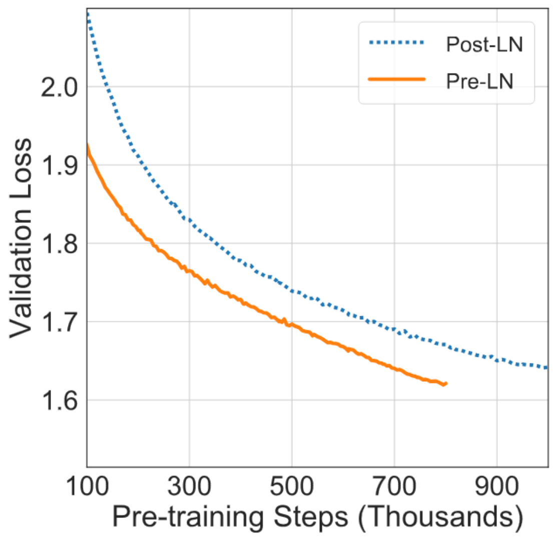
(a) W^1 in the FFN sub-layers



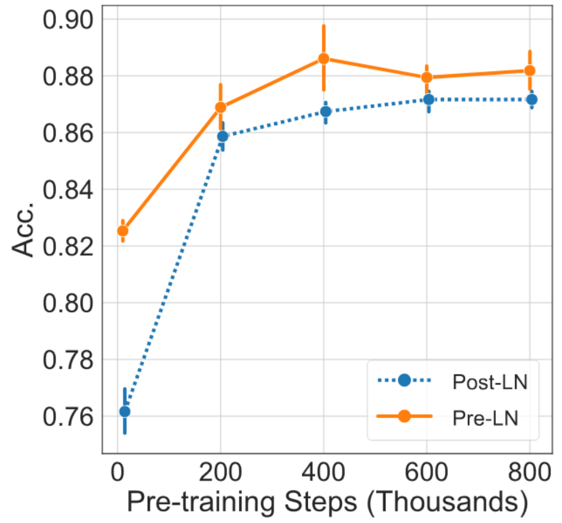
(b) W^2 in the FFN sub-layers

PRE-LAYER NORMALIZATION

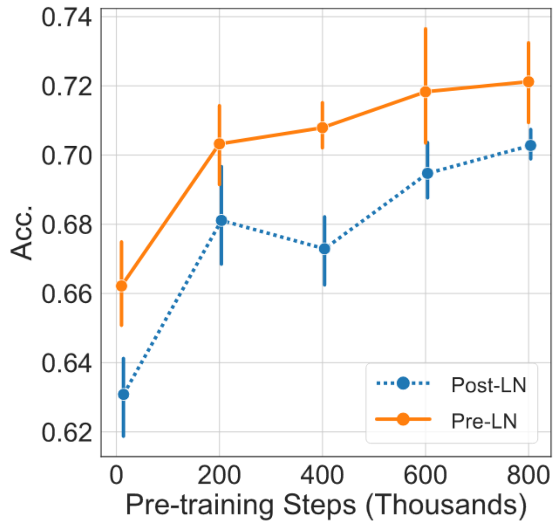
- Measure empirical performance on masked language modeling, semantic similarity (Microsoft Research Paragraph Corpus), and textual entailment (Recognizing Textual Entailment dataset).



(a) Validation Loss on BERT



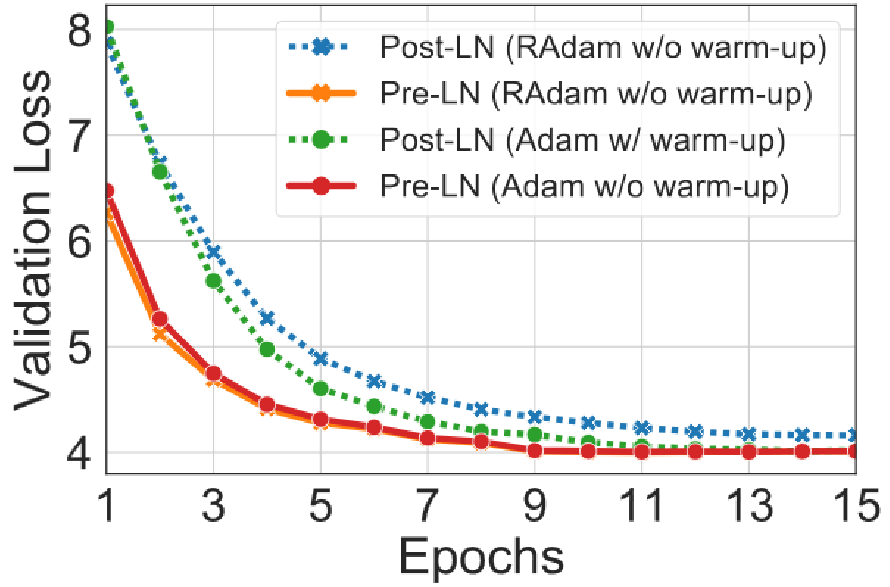
(b) Accuracy on MRPC



(c) Accuracy on RTE

PRE-LAYER NORMALIZATION

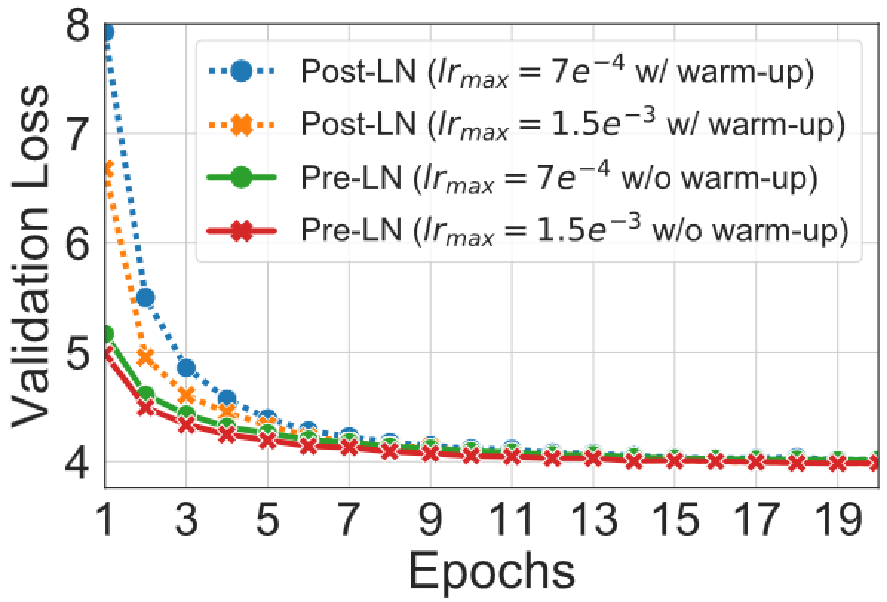
- Measure empirical performance on machine translation.



(a) Validation Loss (IWSLT)

PRE-LAYER NORMALIZATION

- Measure empirical performance on machine translation.



(c) Validation Loss (WMT)

RMS NORMALIZATION

- Zhang and Sennrich, 2019, proposed a simpler alternative to layer normalization, called root mean square layer normalization, or RMSNorm.

$$\text{LayerNorm}(x_i) = \frac{x_i - avg(x_i)}{\sqrt{var(x_i) + \varepsilon}} \circ \gamma + \beta$$

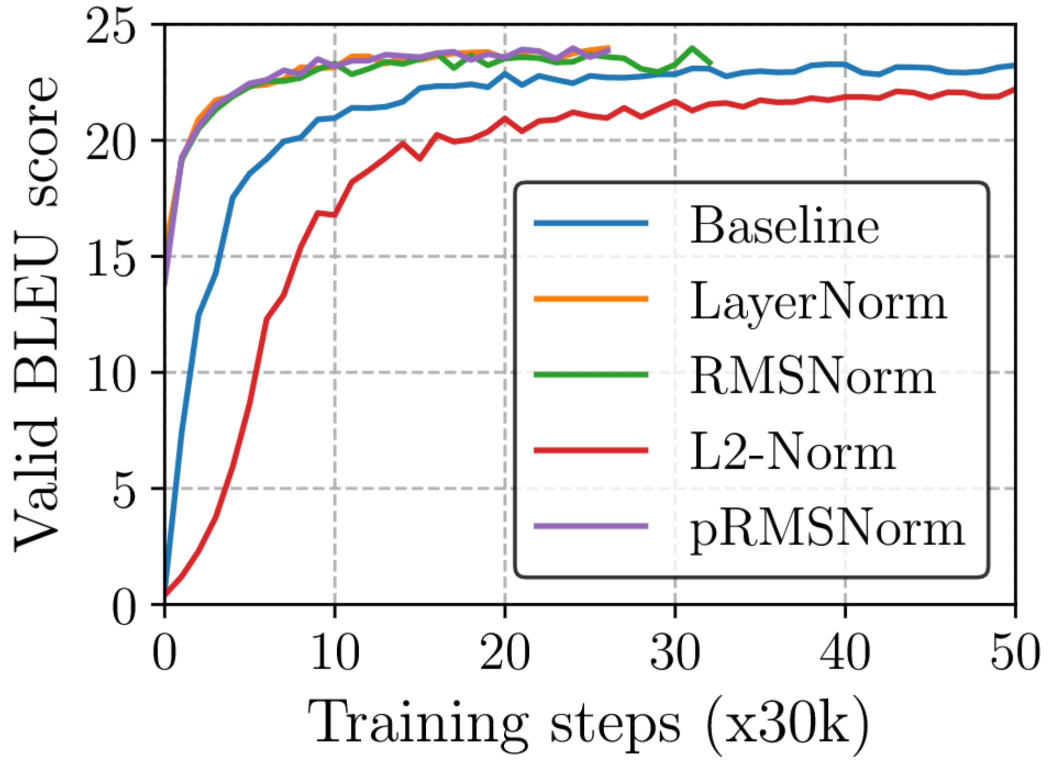
where ε is a small fixed constant, γ and β are vectors of learnable weights.

$$\text{RMSNorm}(x_i) = \frac{x_i}{RMS(x_i)} \circ \gamma \text{ where } RMS(x_i) = \sqrt{\frac{1}{n}\sum_{j=1}^{n} x_{i,j}^2}$$

- Note that RMSNorm is faster to compute.

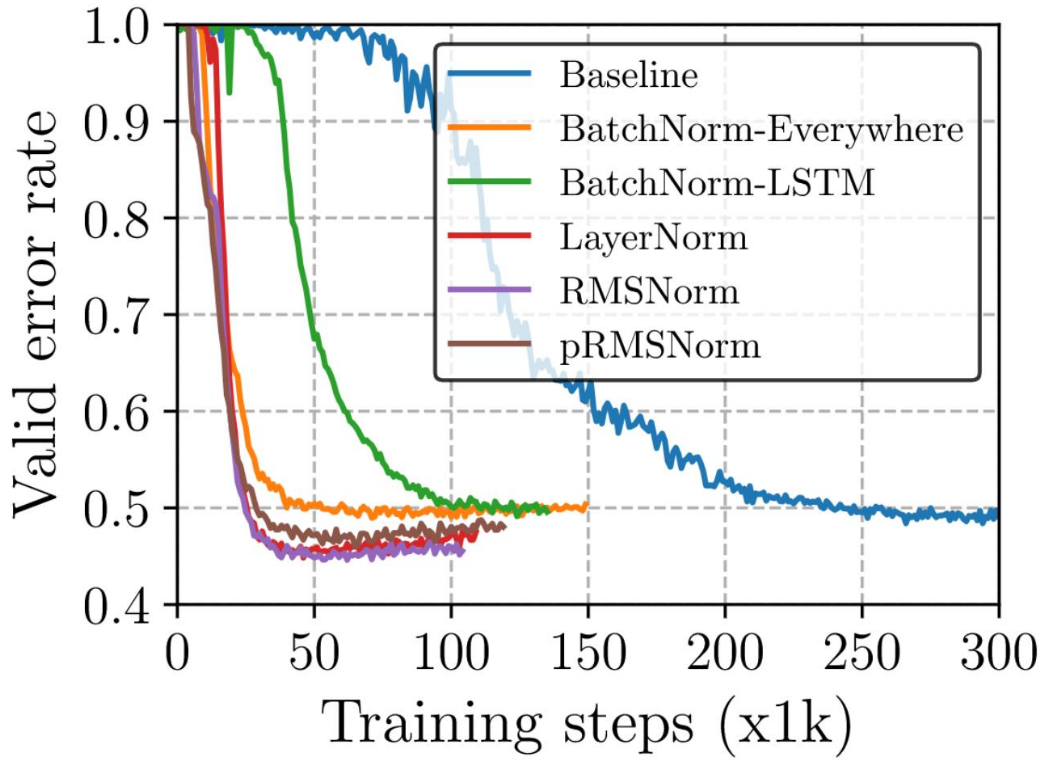
RMS NORMALIZATION

- Measure empirical performance on machine translation.



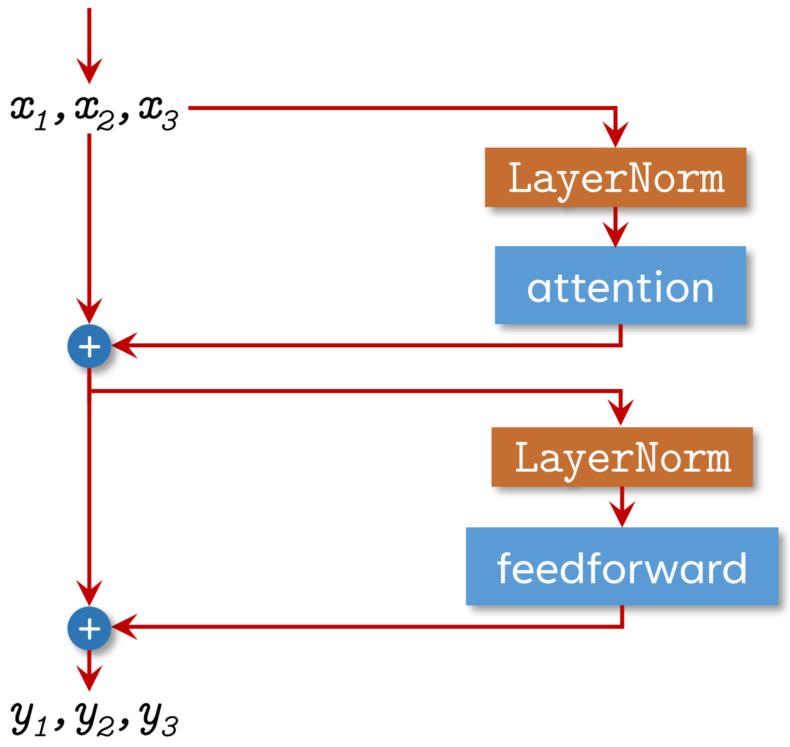
RMS NORMALIZATION

- Measure empirical performance on question answering.



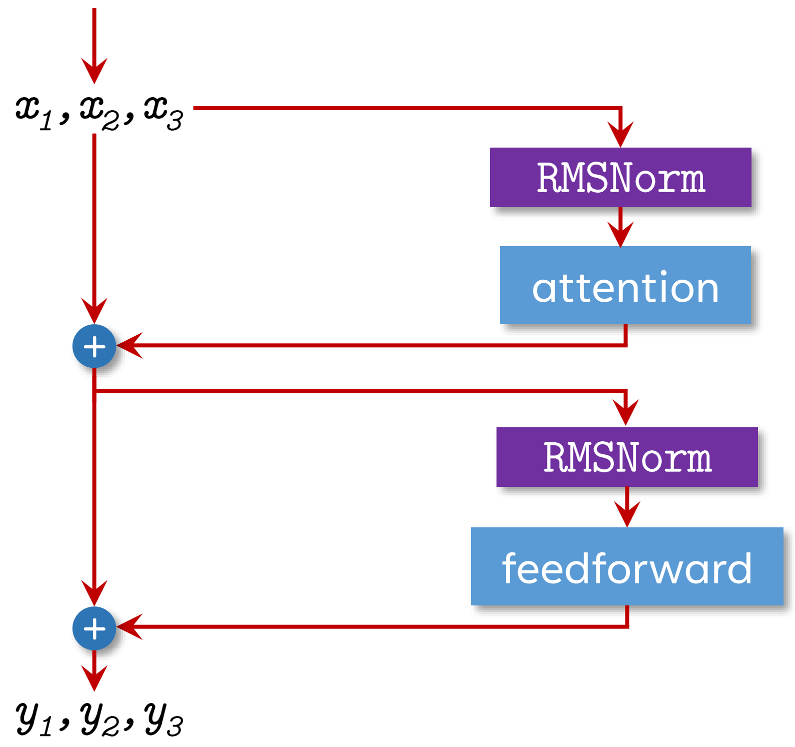
PRE-LAYER NORMALIZATION AND RMSNORM

- Older transformer models (e.g., GPT-2) used pre-layer normalization.



PRE-LAYER NORMALIZATION AND RMSNORM

- Older transformer models (e.g., GPT-2) used pre-layer normalization.
- Recent models (e.g., Llama, DeepSeek, Qwen, Olmo) typically use pre-layer normalization with RMSNorm.



QUESTIONS?