CS 577: NATURAL LANGUAGE PROCESSING

Abulhair Saparov

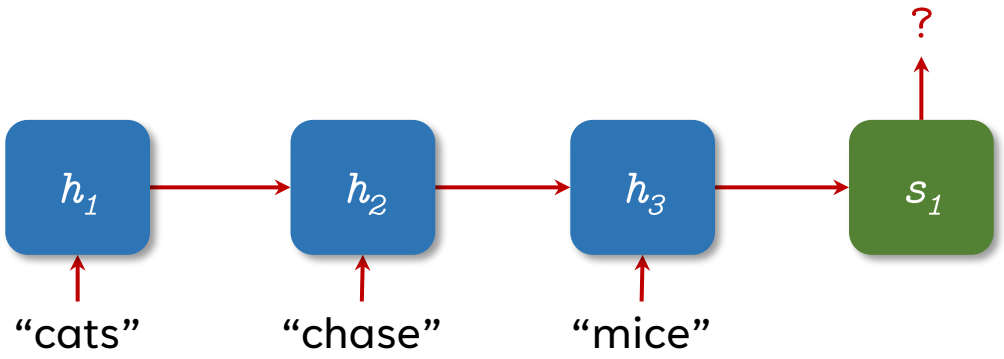
Lecture 6: Attention and Transformers

LONG-TERM DEPENDENCIES

- Last lecture, we discussed long short-term memory (LSTM) networks and gated recurrent units (GRUs).
- They were designed to solve the problem of long-term dependencies in classical RNNs.
- 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.
- 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

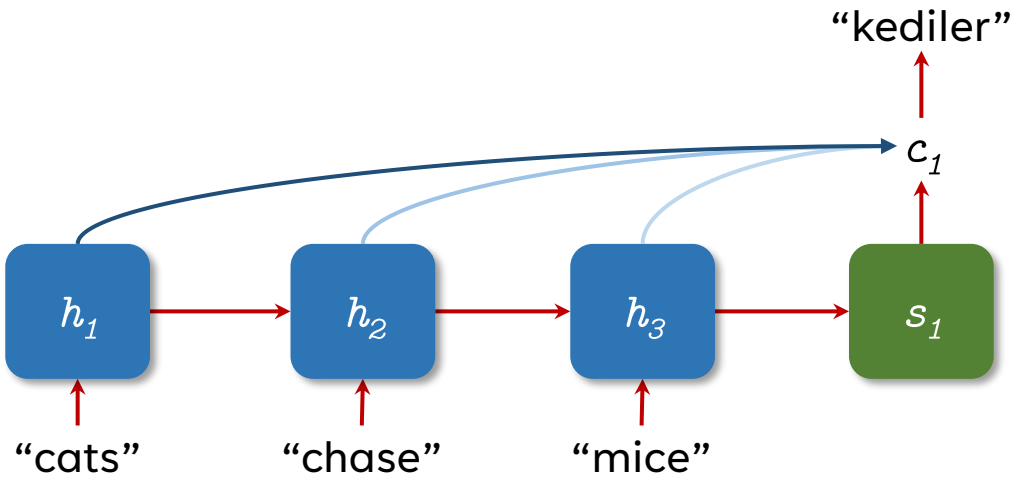
- The attention mechanism was designed to more explicitly model dependencies between words that are very far apart.
- Suppose we have an RNN performing a machine translation task.



[Bahdanau et al., 2015]

ATTENTION

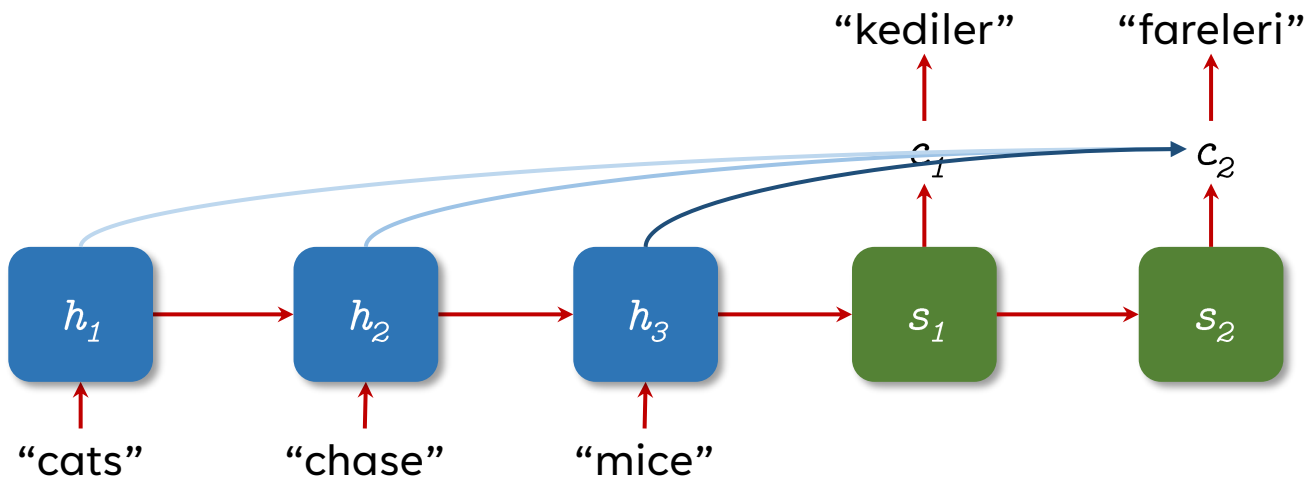
- Basic idea: Compute a linear sum of the vectors corresponding to the input.
- The weights in this linear sum are called attention weights.



[Britz, Goldie, Lyong, and Le 2017]

ATTENTION

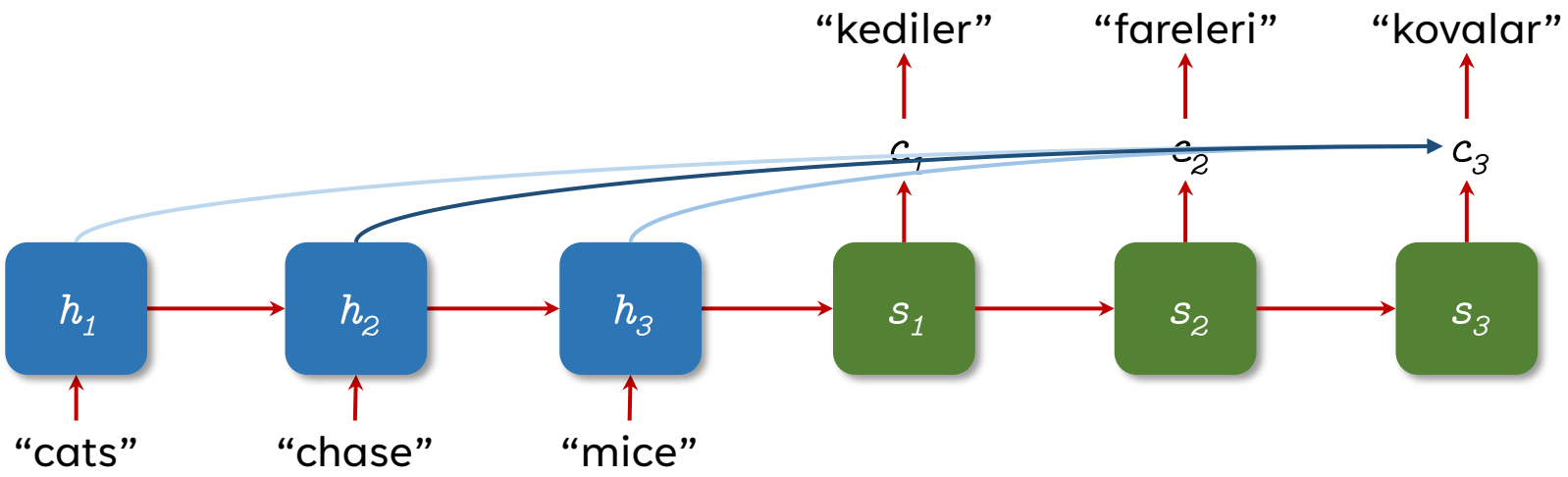
- Basic idea: Compute a linear sum of the vectors corresponding to the input.
- The weights in this linear sum are called attention weights.



[Britz, Goldie, Lyong, and Le 2017]

ATTENTION

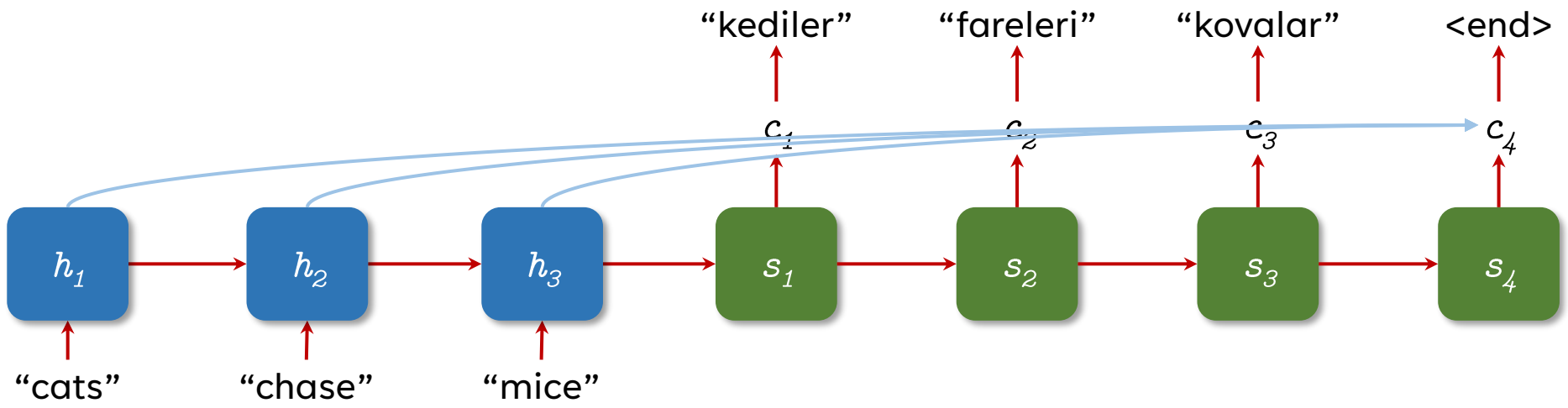
- Basic idea: Compute a linear sum of the vectors corresponding to the input.
- The weights in this linear sum are called attention weights.



[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.
- The entire input no longer needs to be encoded in the hidden state.



[Britz, Goldie, Lyong, and Le 2017]

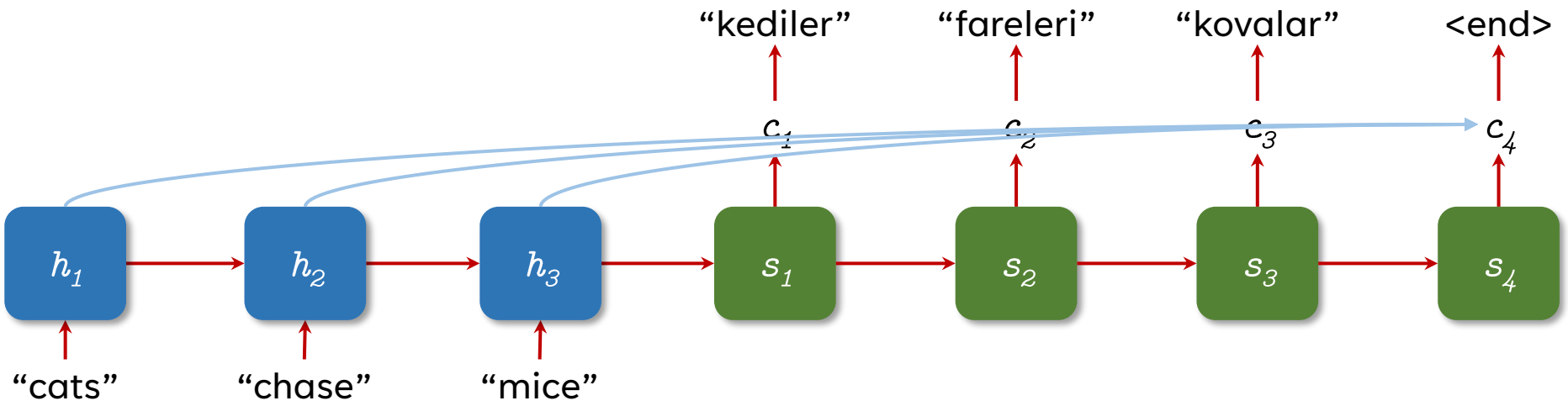
ATTENTION

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

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

Note the "$\propto$" (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.

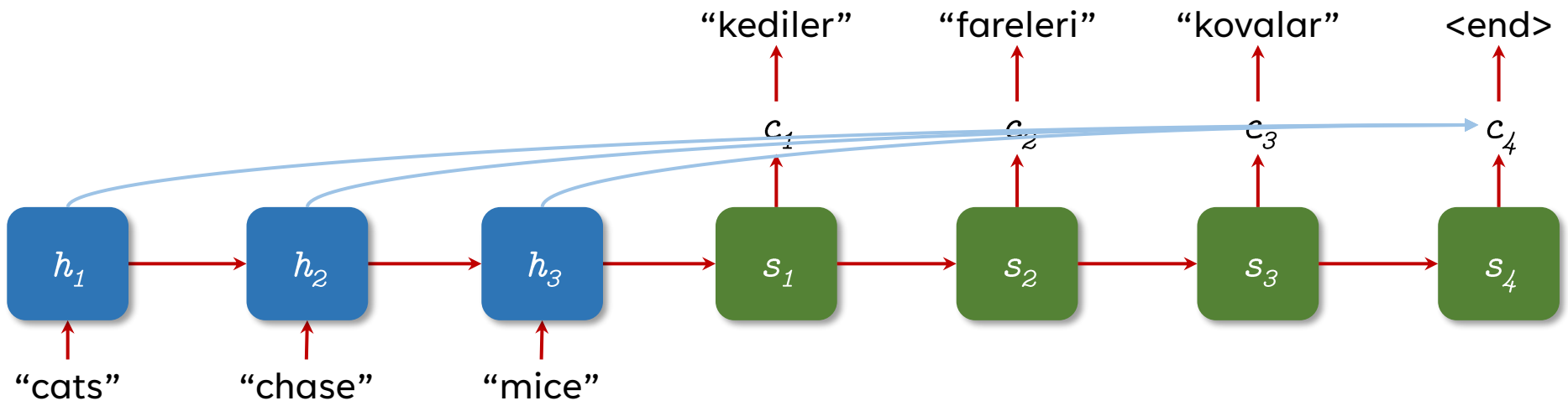
$a_{ij} \propto score(s_i, h_j)$

$score(x, y) = x^T y$

$score(x, y) = (W_1 x)^T (W_2 y)$

etc...

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

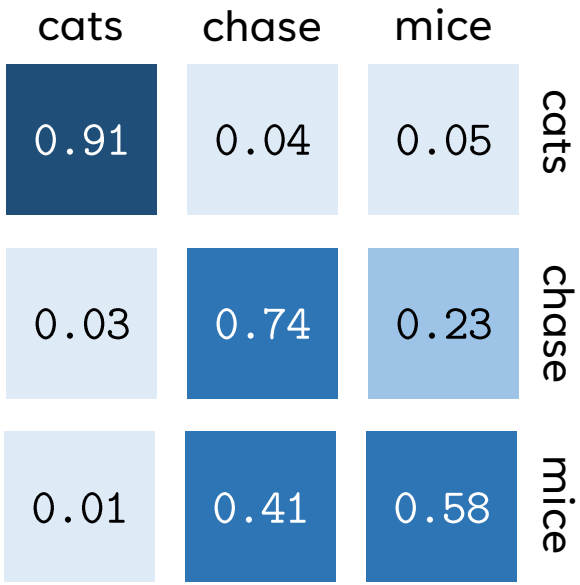


ATTENTION

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

SELF-ATTENTION

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

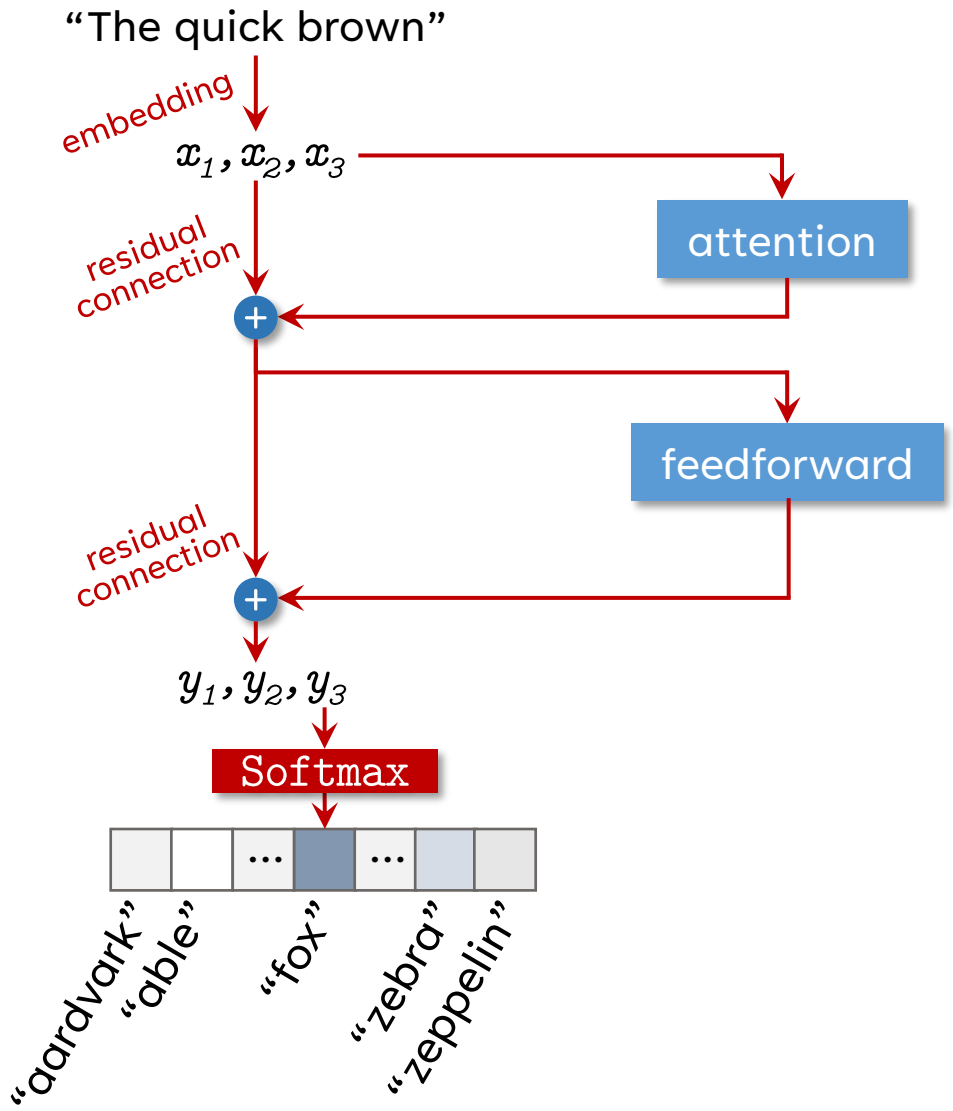


TRANSFORMER

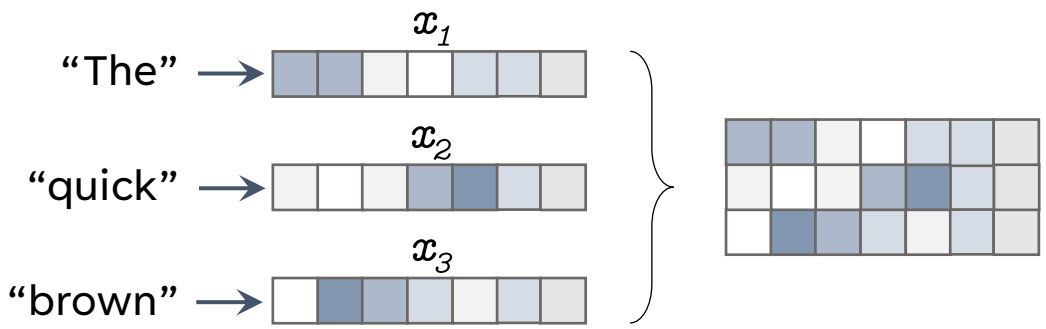
- RNNs suffer from poor parallelizability.
- We must compute each successive hidden state sequentially.
- What if we removed the recurrent aspect, and we model inter-word dependencies entirely via the attention mechanism?
- The transformer architecture is one way to do this (Vaswani et al., 2017).

TRANSFORMER

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

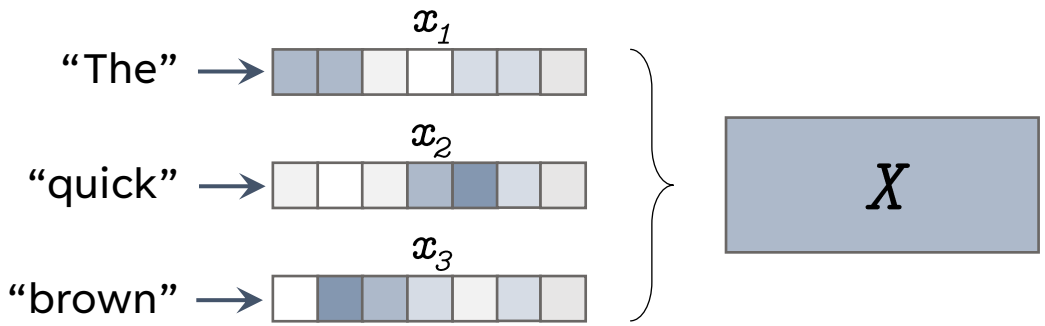


SIMPLIFYING NOTATION



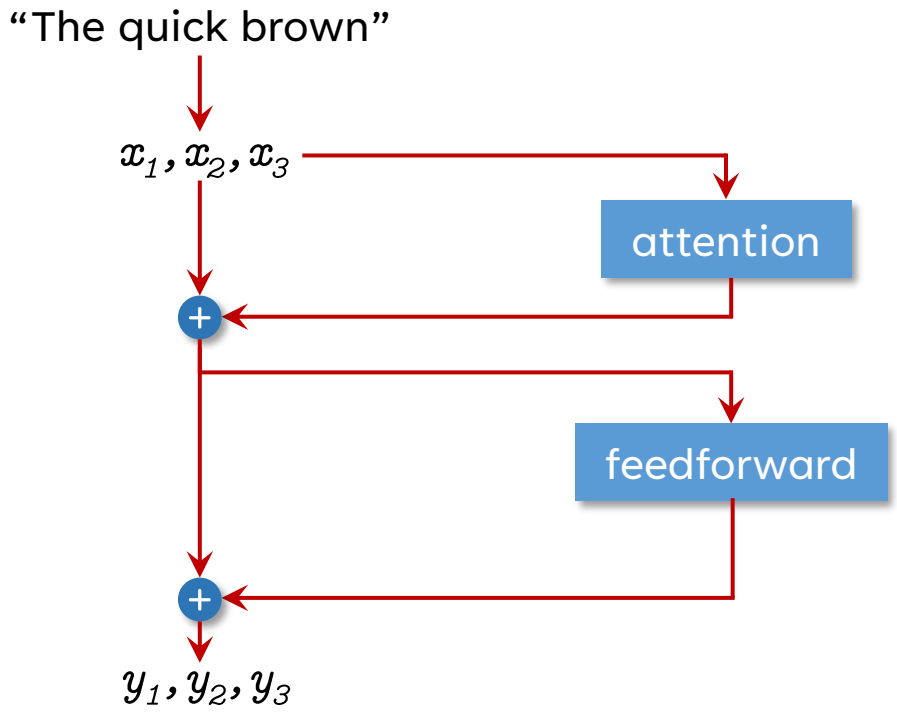
This matrix is the input to the transformer
(and to the attention and FF layers)

SIMPLIFYING NOTATION

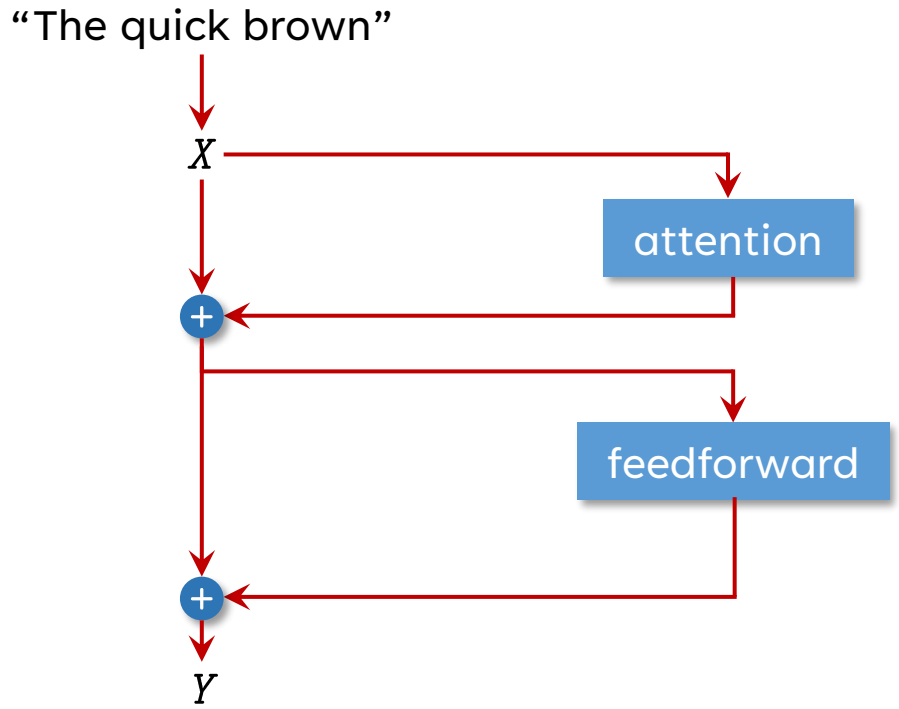


This matrix is the input to the transformer
(and to the attention and FF layers)

TRANSFORMER



TRANSFORMER



FEEDFORWARD LAYER

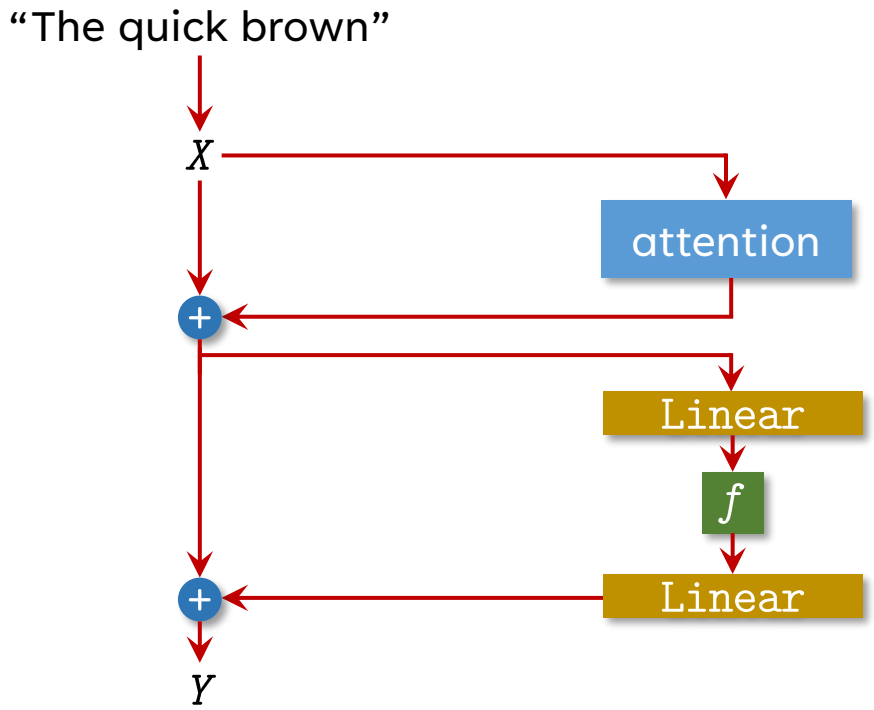
$$X^{out} = W_2 f(W_1 X^{in} + b_1) + b_2$$

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:

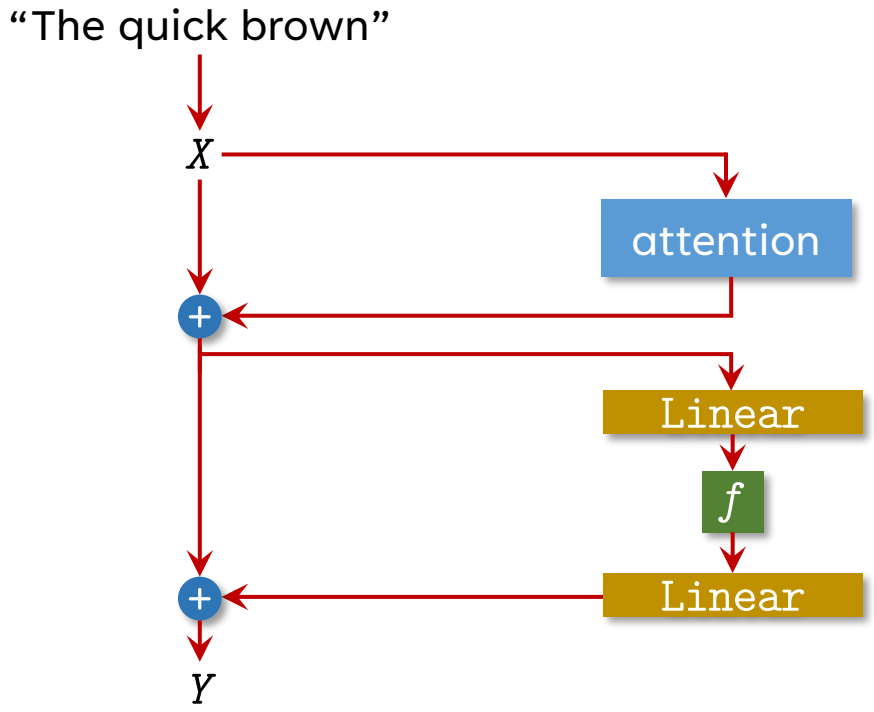
$$x_1^{out} = W_2 f(W_1 x_1^{in} + b_1) + b_2$$

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

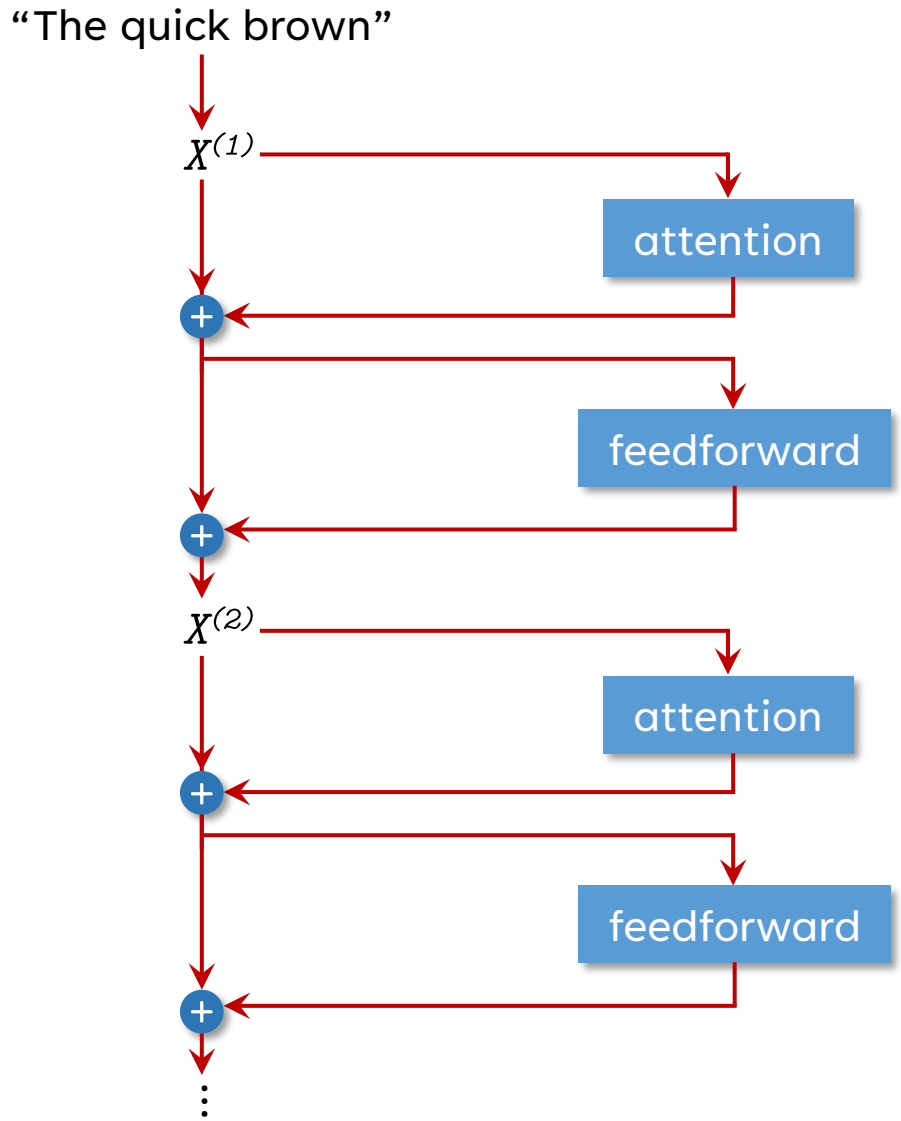
$$x_3^{out} = W_2 f(W_1 x_3^{in} + b_1) + b_2$$

etc…

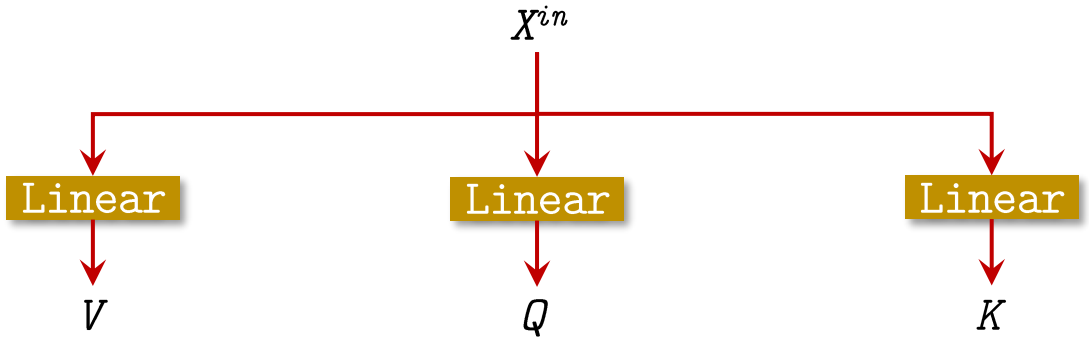
No information is shared between embeddings in the FF layer.



ADDING MORE LAYERS

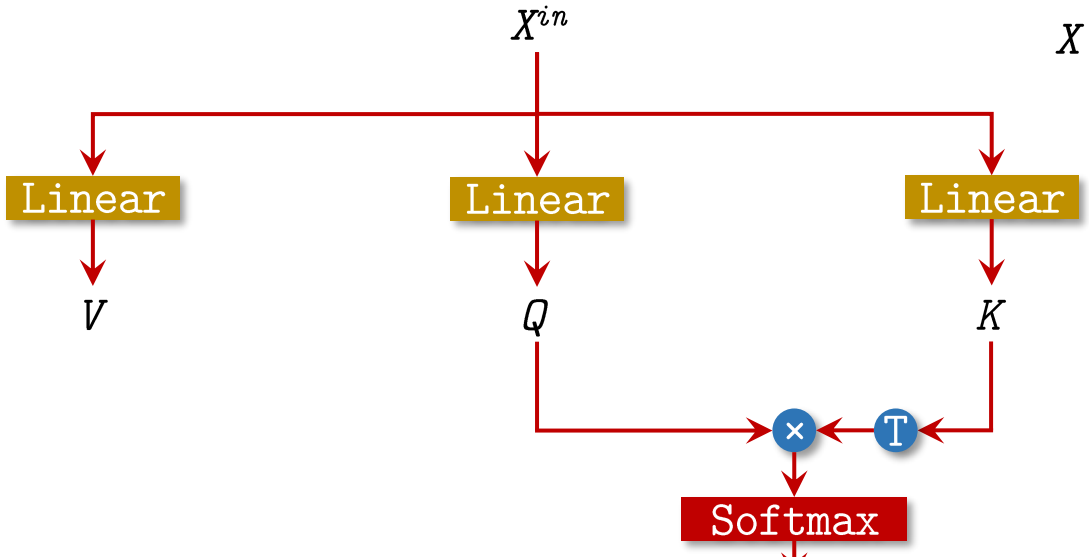


ATTENTION LAYER



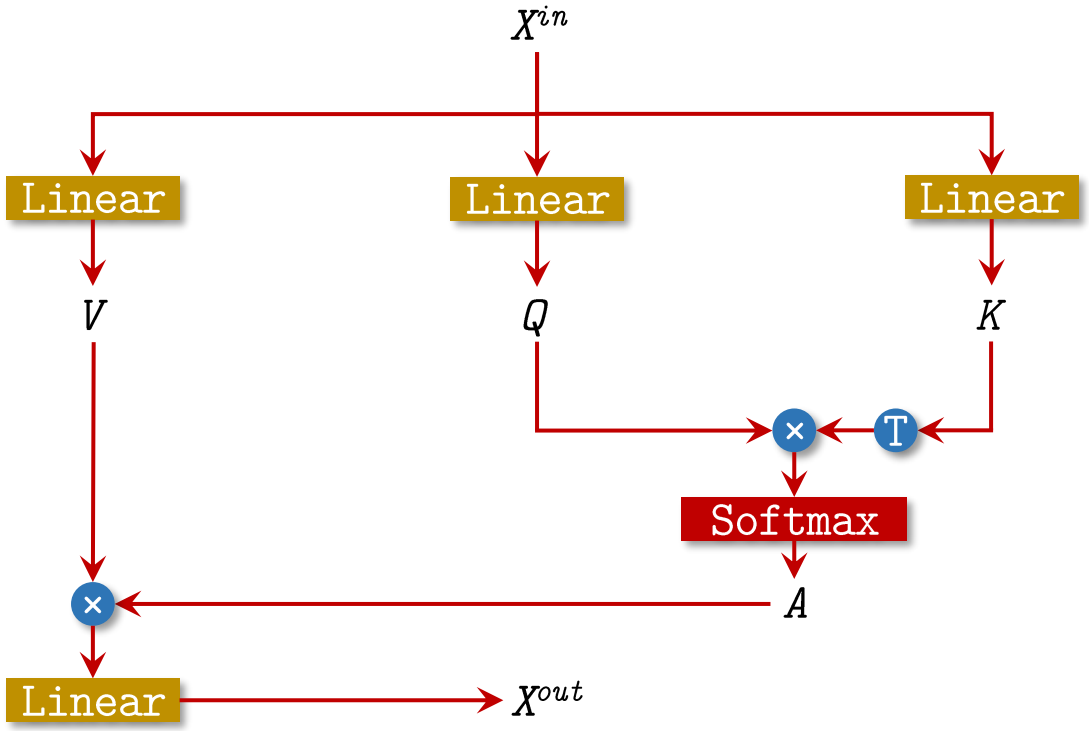
ATTENTION LAYER

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



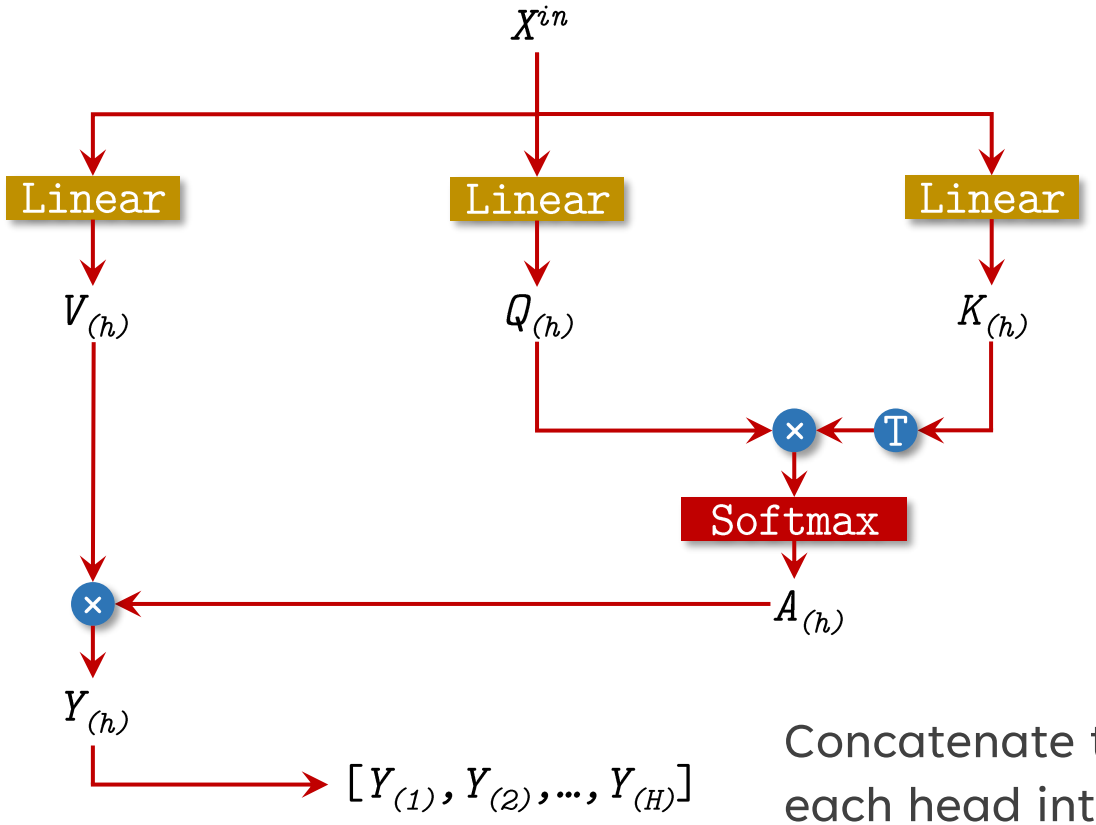
The attention matrix A describes the dependencies between different tokens of the input.
E.g., $A_{i,j}$ describes how strongly the token at index i depends on the token at index j.
We can choose to mask the attention matrix: we force $A_{i,j} = 0$ if $j > i$.
That is, we only allow each token to depend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

ATTENTION LAYER



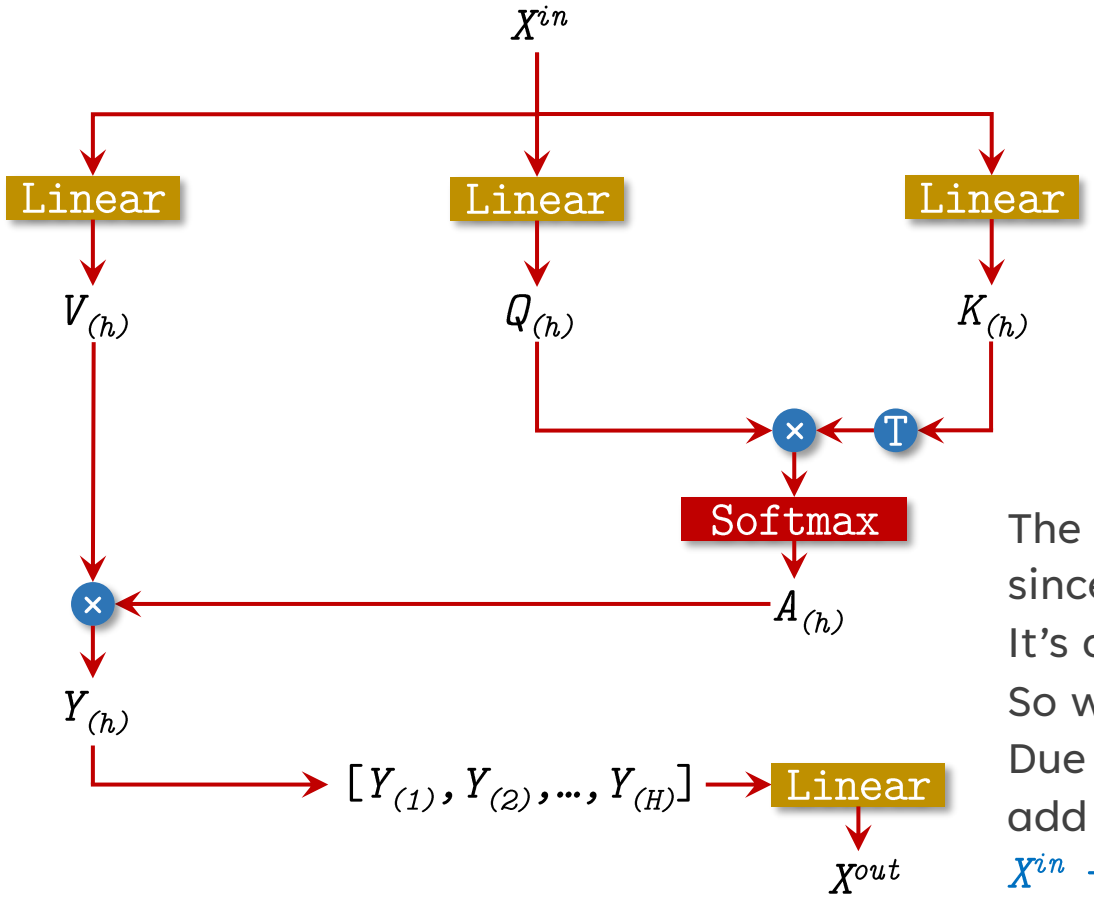
This is the output of the attention layer.
But recall that due to the residual connections, the actual output is $X^{in} + X^{out}$.

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?

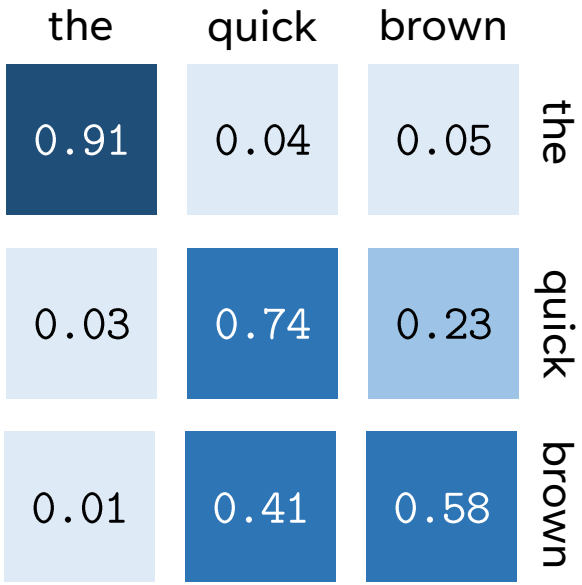
- Different attention heads can perform different computations.
- For example one attention head can compute syntactic relations:
  - “I run a small business” vs “I went for a run”
  - To compute the part-of-speech of “run”, it helps to attend the word immediately before: “I” vs “a”.
  - “a run” indicates 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!

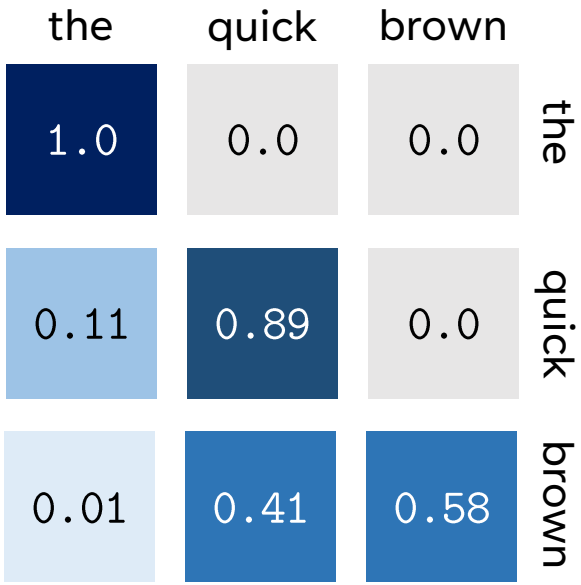
CAUSAL MASK

- Example of attention matrix without mask:



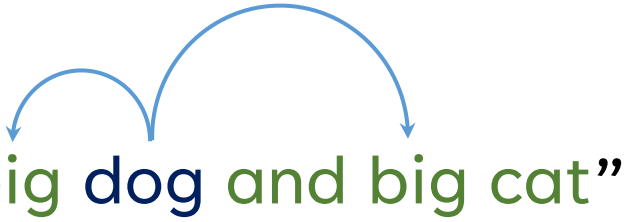
CAUSAL MASK

- Example of attention matrix with a causal mask:



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.

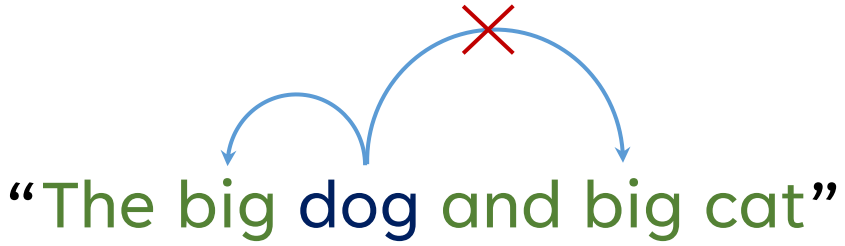


POSITIONAL EMBEDDING

- 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.

POSITIONAL EMBEDDING

- But if we use the causal mask for attention, the transformer may be able to compute positions, even without positional embeddings.
- In the earlier example, if we use a causal mask, then "dog" cannot attend to the later "big."

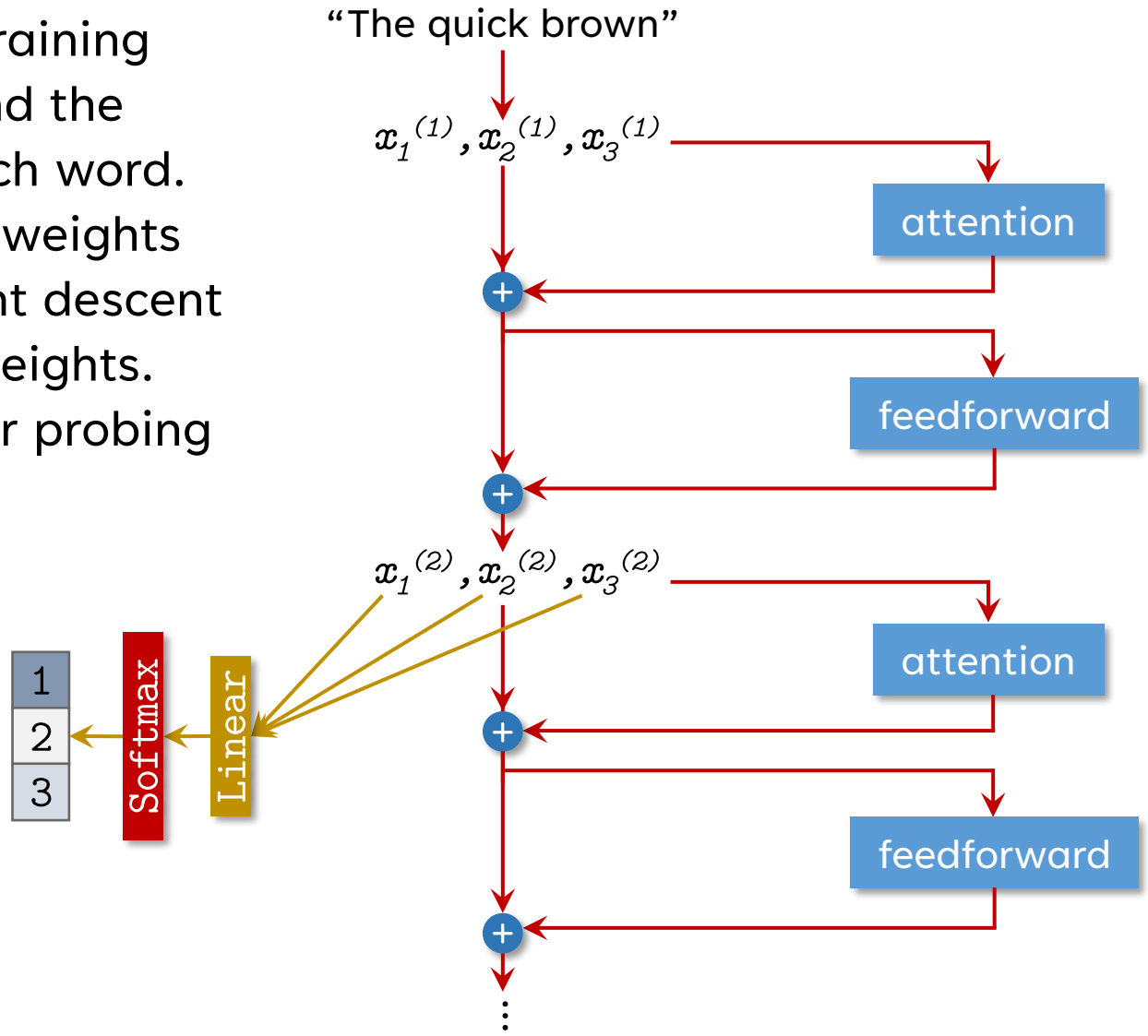


POSITIONAL EMBEDDING

- But if we use the causal mask for attention, the transformer may be able to compute positions, even without positional embeddings.
- We can test this by training multiple language models using different positional encodings.
- Then, we fix the weights of the transformer and train a linear probe to predict the absolute position of each token:
  - For each layer L of the transformer, add a linear layer from each vector in the layer's output ${x_i}^{(L)}$ and a softmax to predict the absolute position i.

LINEAR PROBING

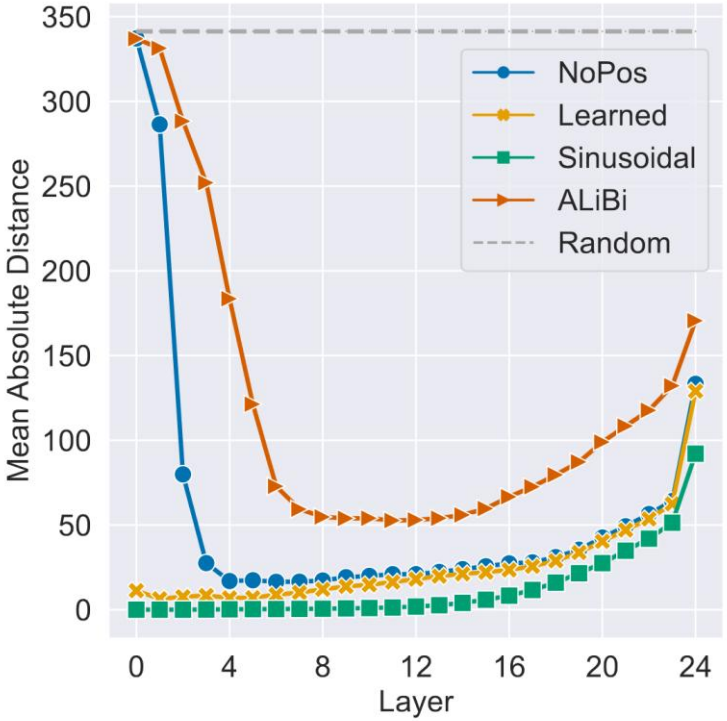
- We have supervised training examples of inputs and the correct position of each word.
- Keep the transformer weights fixed, and use gradient descent to learn the probe's weights.
- Note: we can do linear probing on any model!



[Haviv et al., 2022]

TRANSFORMERS CAN LEARN POSITION FROM CAUSAL MASK

- Train a probe at each layer of the transformer.
- Measure how accurately the probe can predict each word's position.



[Haviv et al., 2022]

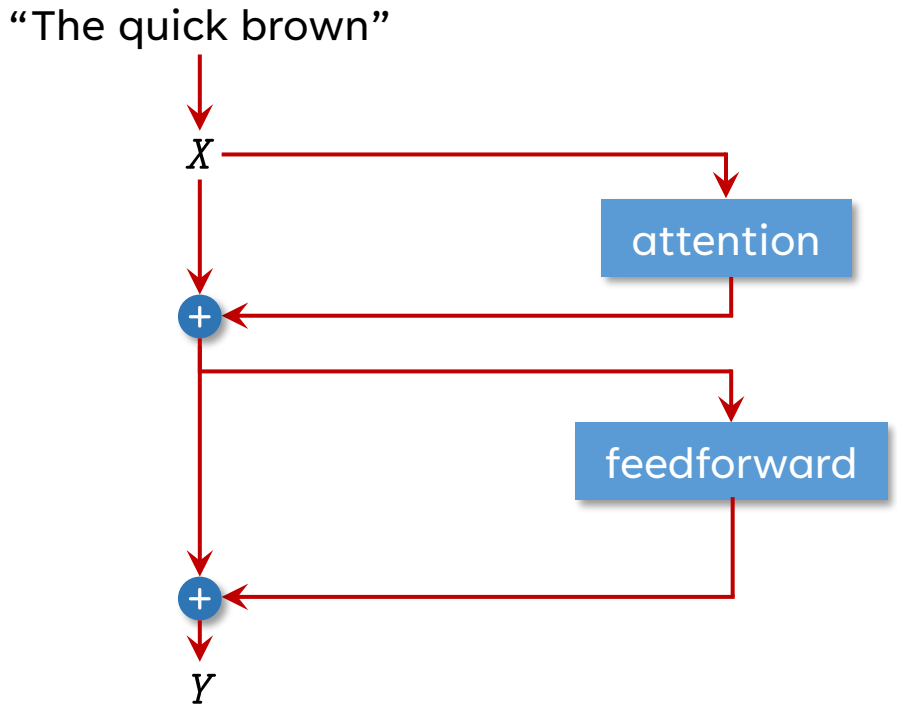
LAYER NORMALIZATION

- 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.

LAYER NORMALIZATION

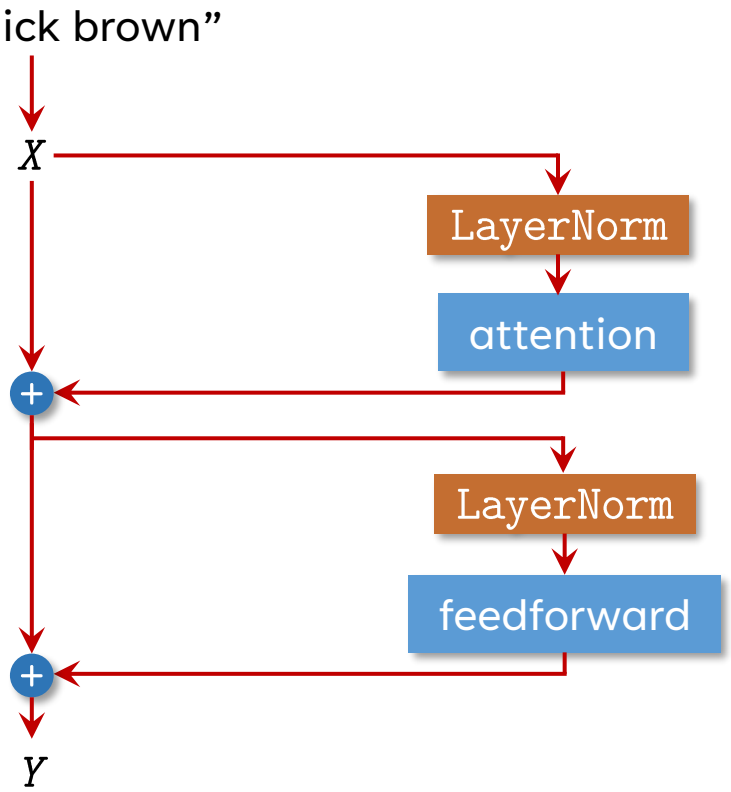


LAYER NORMALIZATION

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

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

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



QUESTIONS?