WEB INTELLIGENCE

Lecture 9:
Neural Networks and Attention Models

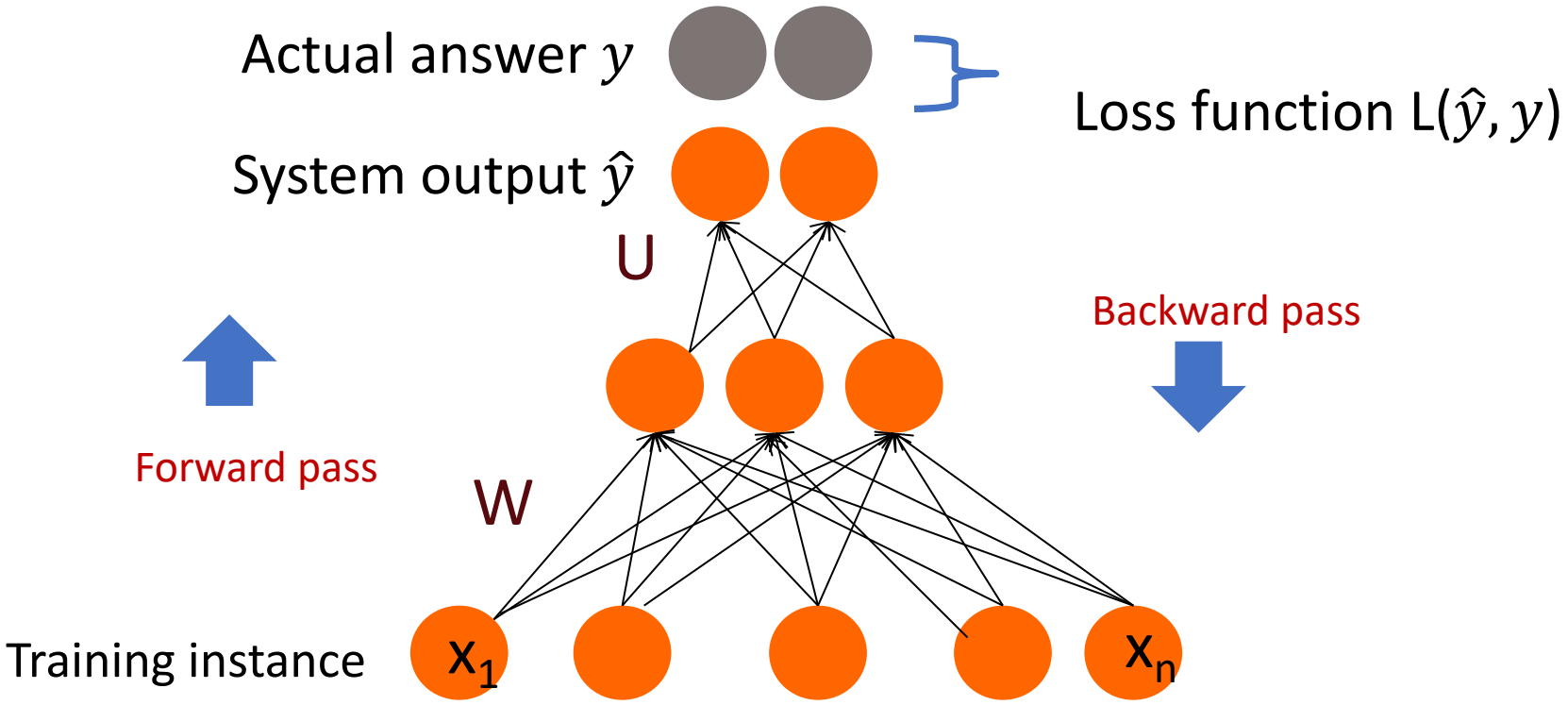
Russa Biswas

November 04, 2024

AALBORG UNIVERSITY

Based on Slides by Dan Jurafsky, Anna Goldie, John Hewitt

Training 2-layer Networks



Training 2-layer Networks

- For every training tuple (x, y)
  - Run *forward* computation to find our estimate $\hat{y}$
  - Run *backward* computation to update weights:
    - For every output node
      - Compute loss L between true y and the estimated $\hat{y}$
      - For every weight w from hidden layer to the output layer
        - Update the weight
    - For every hidden node
      - Assess how much blame it deserves for the current answer
      - For every weight w from input layer to the hidden layer
        - Update the weight

Loss function

- A measure for how far off the current answer is to the right answer
- Cross entropy loss for logistic regression:

$$L_{\text{CE}}(\hat{y}, y) = -\log p(y|x) = -[y \log \hat{y} + (1-y)\log(1-\hat{y})]$$

$$= -[y \log \sigma(w \cdot x + b) + (1-y)\log(1-\sigma(w \cdot x + b))]$$

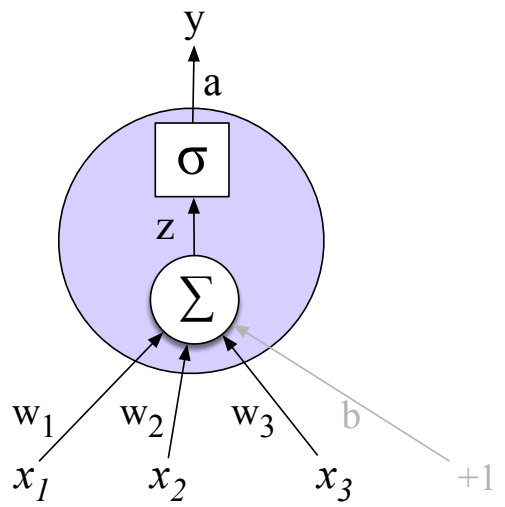
Gradient Descent for weight updates

- Use the derivative of the loss function with respect to weights $\frac{d}{dw} L(f(x;w),y)$
- To tell us how to adjust weights for each training item
  - Move them in the opposite direction of the gradient

$$w^{t+1} = w^t - \eta \frac{d}{dw} L(f(x;w),y)$$

  - A higher (faster) learning rate means that we should move *w* more on each step
  - For logistic regression

$$\frac{\partial L_{\text{CE}}(\hat{y},y)}{\partial w_j} = [\sigma(w \cdot x + b) - y] x_j$$



Using the chain rule $f(x) = u(v(x))$

$$\frac{df}{dx} = \frac{du}{dv} \cdot \frac{dv}{dx}$$

$$\frac{\partial Lz}{\partial w_i} = \frac{\partial L}{\partial y} \frac{\partial y}{\partial z} \frac{\partial z}{\partial w_i}$$

Derivative of the Loss

Derivative of the Activation

Derivative of the weighted sum

Backward Pass – Computation Graph

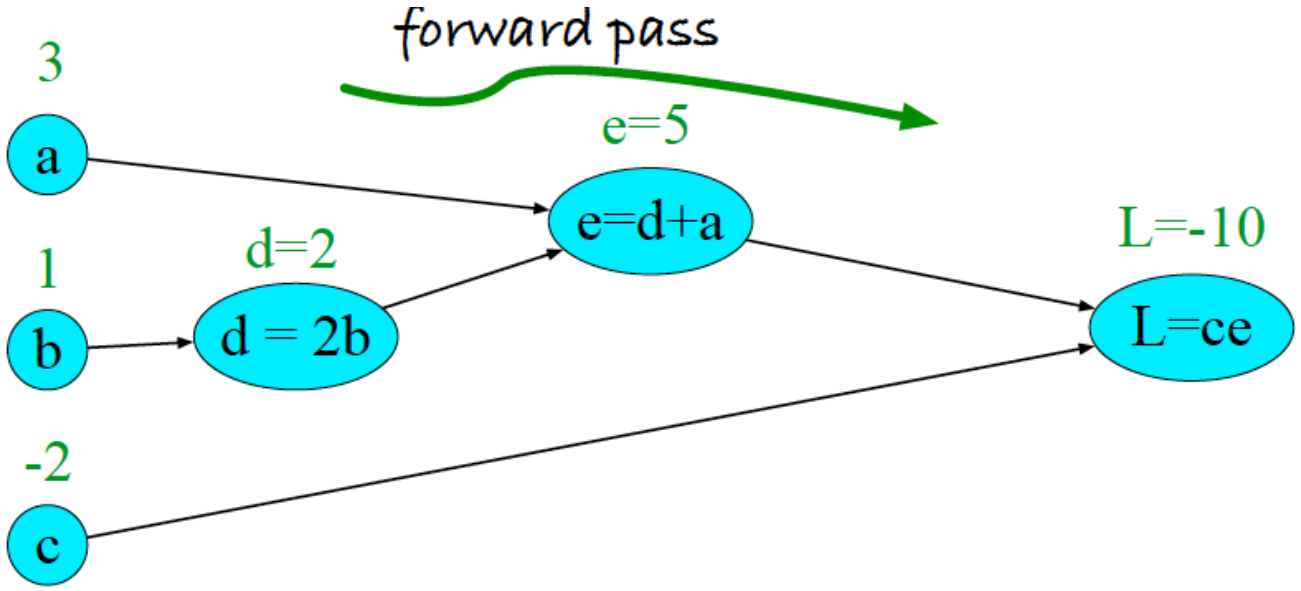
- For training, we need the derivative of the loss with respect to each weight in every layer of the network
- But the loss is computed only at the very end of the network!
- Solution: **error backpropagation** (Rumelhart, Hinton, Williams, 1986)
- **Backprop** is a special case of **backward differentiation**
- Which relies on **computation graphs**.
- A computation graph represents the process of computing a mathematical expression

Example

$$L(a,b,c) = c(a+2b)$$

Computations:

$$d = 2*b$$
$$e = a+d$$
$$L = c*e$$



Backward Pass

- The importance of the computation graph comes from the backward pass
- This is used to compute the derivatives that we'll need for the weight update

$$L(a,b,c) = c(a+2b)$$

$$d = 2*b$$

$$e = a+d$$

$$L = c*e$$

We want: $\frac{\partial L}{\partial a}$, $\frac{\partial L}{\partial b}$, and $\frac{\partial L}{\partial c}$

The derivative $\frac{\partial L}{\partial a}$, tells us how much a small change in *a* affects *L*, while others are constant

Example – contd.

$$L(a,b,c) = c(a+2b)$$

$$\begin{aligned} d &= 2*b \\ e &= a+d \\ L &= c*e \end{aligned}$$

$$L = ce \; : \quad \frac{\partial L}{\partial e} = c, \frac{\partial L}{\partial c} = e$$

$$e = a+d \; : \quad \frac{\partial e}{\partial a} = 1, \frac{\partial e}{\partial d} = 1$$

$$d = 2b \; : \quad \frac{\partial d}{\partial b} = 2$$

$$\begin{aligned} \frac{\partial L}{\partial c} &= e \\ \frac{\partial L}{\partial a} &= \frac{\partial L}{\partial e}\frac{\partial e}{\partial a} \\ \frac{\partial L}{\partial b} &= \frac{\partial L}{\partial e}\frac{\partial e}{\partial d}\frac{\partial d}{\partial b} \end{aligned}$$

Computing Backward Pass

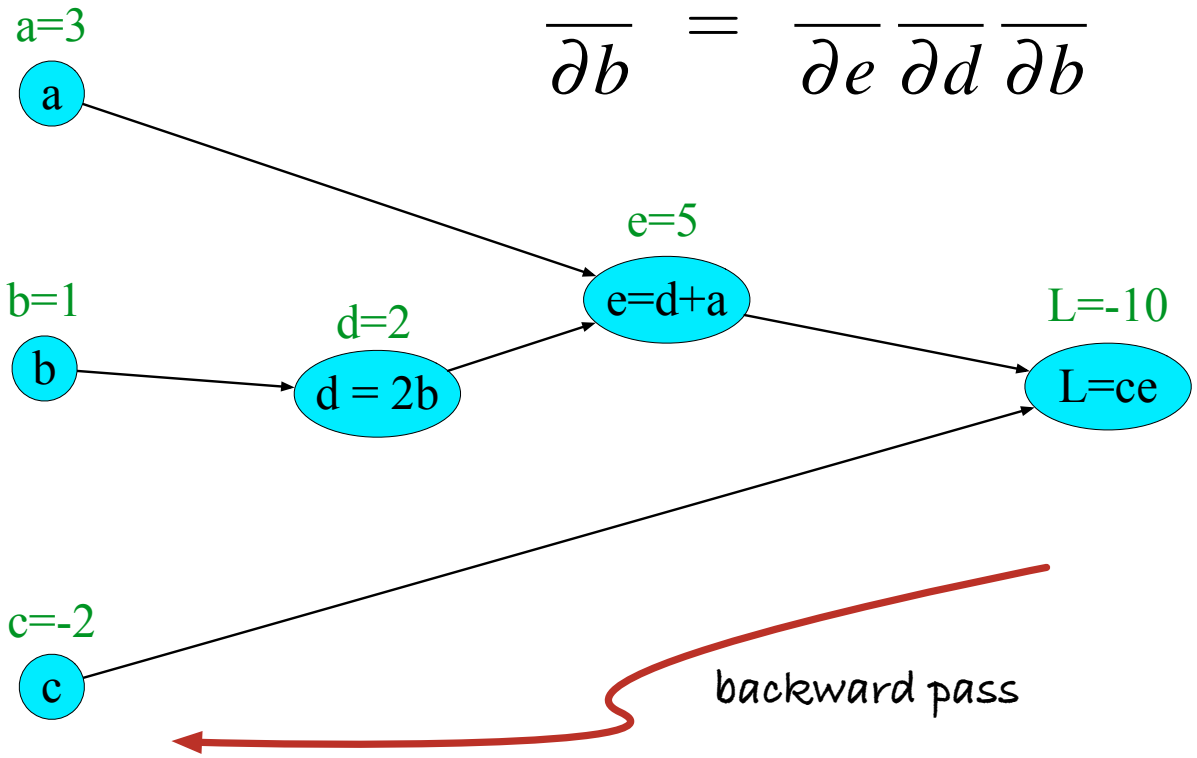
$$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial a}$$

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial d}\frac{\partial d}{\partial b}$$

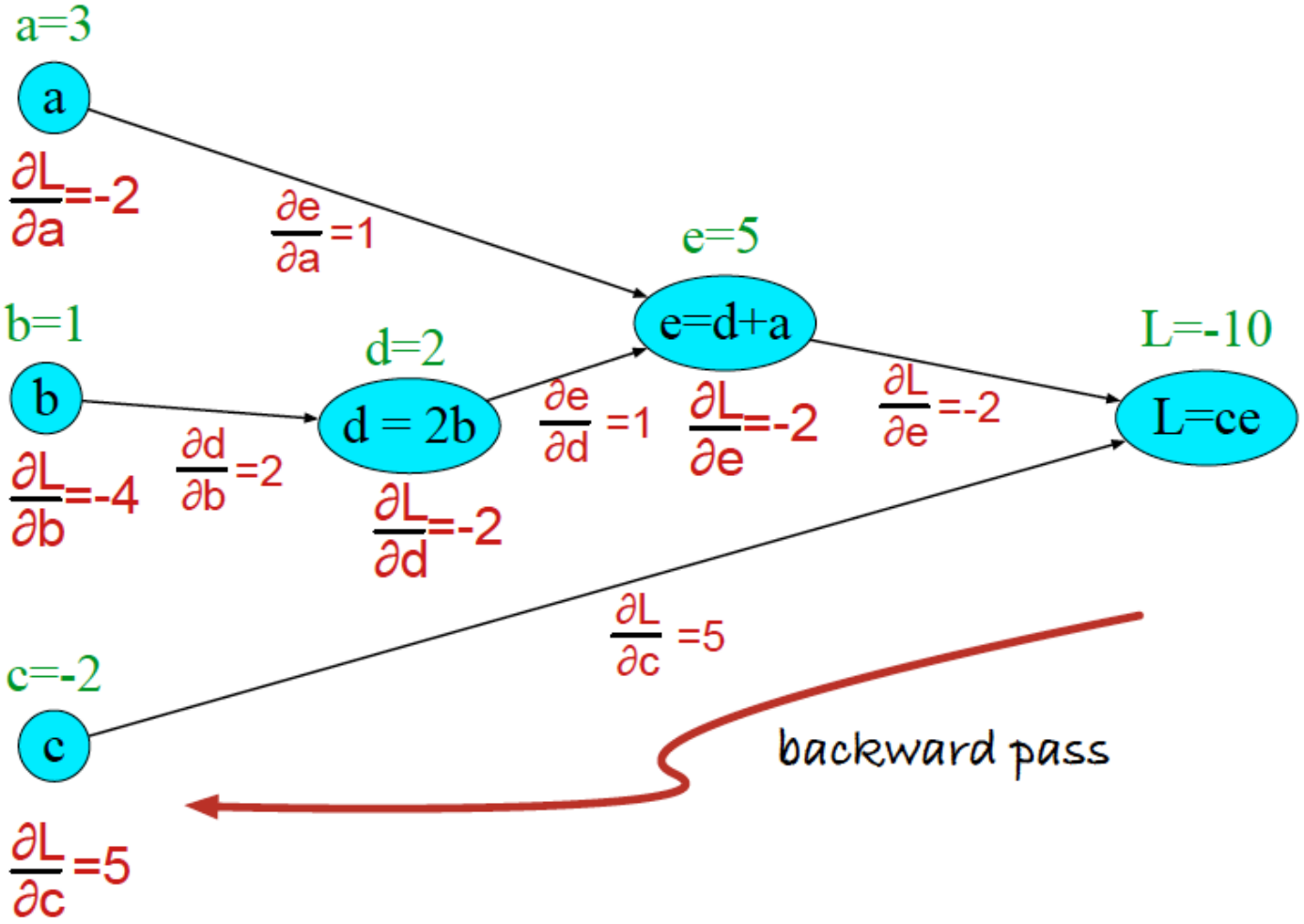
$$L = ce : \quad \frac{\partial L}{\partial e} = c, \frac{\partial L}{\partial c} = e$$

$$e = a + d : \quad \frac{\partial e}{\partial a} = 1, \frac{\partial e}{\partial d} = 1$$

$$d = 2b : \quad \frac{\partial d}{\partial b} = 2$$



Computing Backward Pass



Backward differentiation on 2-layer Network

$$
\begin{aligned}
z^{[1]} &= W^{[1]}\mathbf{x} + b^{[1]} \\
a^{[1]} &= \text{ReLU}(z^{[1]}) \\
z^{[2]} &= W^{[2]}a^{[1]} + b^{[2]} \\
a^{[2]} &= \sigma(z^{[2]}) \\
\hat{y} &= a^{[2]}
\end{aligned}
$$

$$
\frac{d\,\text{ReLU}(z)}{dz} = \begin{cases} 0 & for \quad z < 0 \\ 1 & for \quad z \geq 0 \end{cases}
$$

$$
\frac{d\sigma(z)}{dz} = \sigma(z)(1 - \sigma(z))
$$

Starting off the backward pass: $\frac{\partial L}{\partial z}$

(I'll write a for $a^{[2]}$ and z for $z^{[2]}$)

$$
\begin{aligned}
z^{[1]} &= W^{[1]}\mathbf{x} + b^{[1]} \\
a^{[1]} &= \mathbf{ReLU}(z^{[1]}) \\
z^{[2]} &= W^{[2]}a^{[1]} + b^{[2]} \\
a^{[2]} &= \boldsymbol{\sigma}(z^{[2]}) \\
\hat{y} &= a^{[2]}
\end{aligned}
$$

$$L(\hat{y}, y) = -(\mathrm{y}\log(\hat{y}) + (1-y)\log(1-\hat{y}))$$

$$L(a, y) = -(\mathrm{y}\log a + (1-y)\log(1-a))$$

$$\frac{\partial L}{\partial z} = \frac{\partial L}{\partial a}\frac{\partial a}{\partial z}$$

$$\frac{\partial L}{\partial a} = -\left(\left(y\frac{\partial \log(a)}{\partial a}\right) + (1-\mathrm{y})\ \frac{\partial \log(1-a)}{\partial a}\right)$$

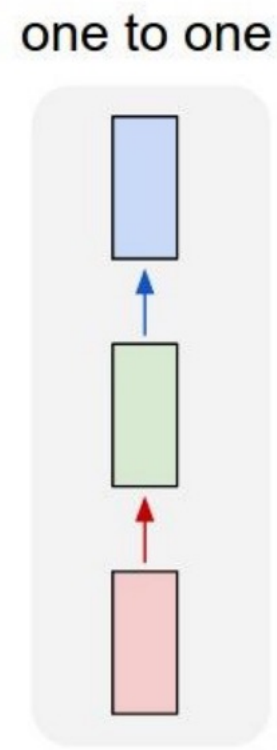
$$= -\left(\left(y\frac{1}{a}\right) + (1-\mathrm{y})\ \frac{1}{1-a}(-1)\right) = -\left(\frac{y}{a} + \frac{y-1}{1-a}\right)$$

$$\frac{\partial a}{\partial z} = a(1-a) \qquad \frac{\partial L}{\partial z} = -\left(\frac{y}{a} + \frac{y-1}{1-a}\right)a(1-a) = \mathrm{a} - \mathrm{y}$$

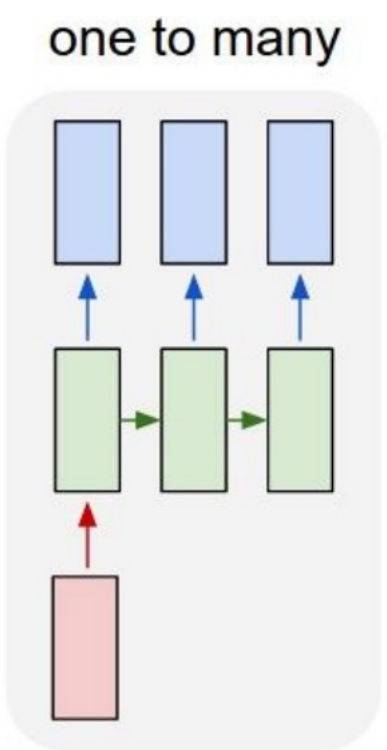
Backward Propagation

- For training, we need the derivative of the loss with respect to weights in early layers of the network
- But loss is computed only at the very end of the network!
- Solution: **backward differentiation**
- Given a computation graph and the derivatives of all the functions in it we can automatically compute the derivative of the loss with respect to these early weights.

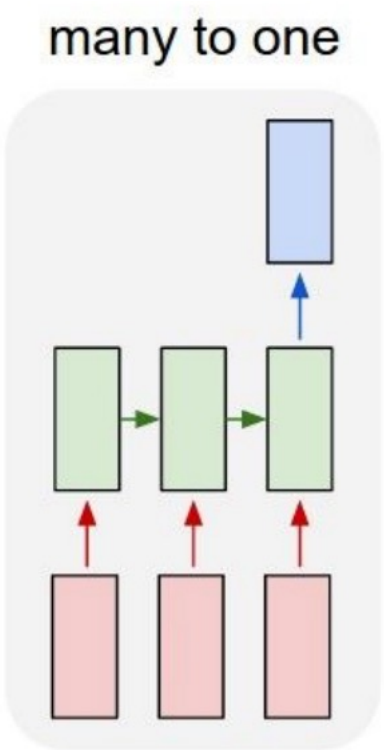
Recurrent Neural Networks



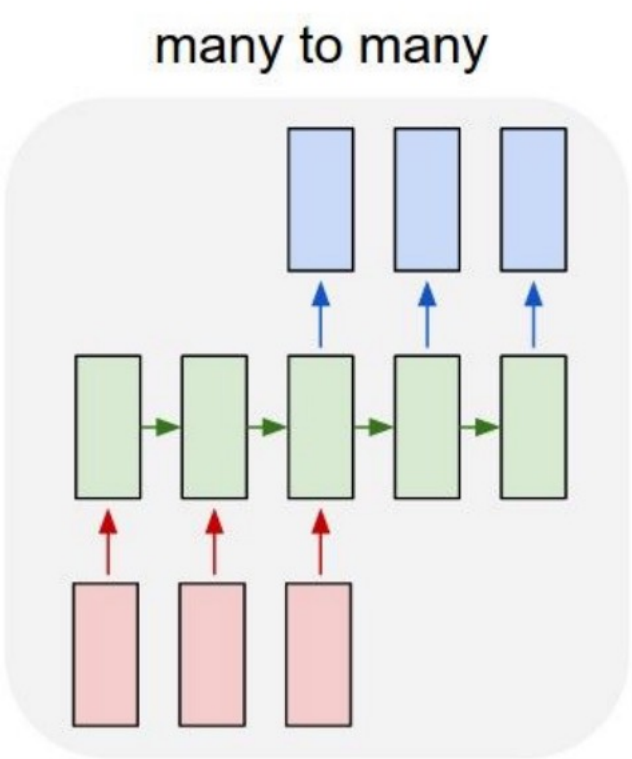
Vanilla Model



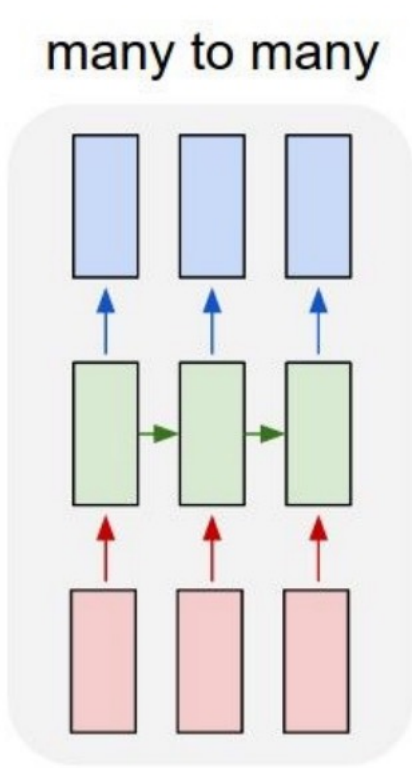
e.g. Image Captioning
image
➔ sequence of words



e.g. action prediction
sequence of video frames -> action class



E.g. Video Captioning
Sequence of video frames -> caption



e.g. Video classification on frame level

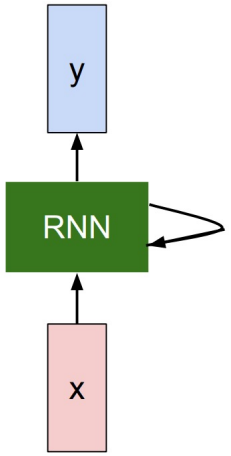
RNN – Mechanism

We can process a sequence of vectors **x** by applying a **recurrence formula** at every time step:

$$h_t = f_W(h_{t-1}, x_t)$$

new state — old state — input vector at some time step

some function with parameters W

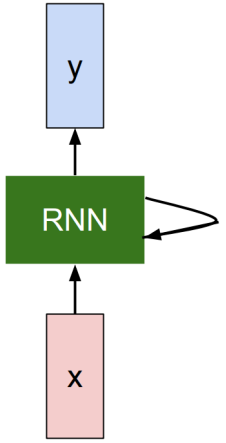


We can process a sequence of vectors **x** by applying a **recurrence formula** at every time step:

$$y_t = f_{W_{hy}}(h_t)$$

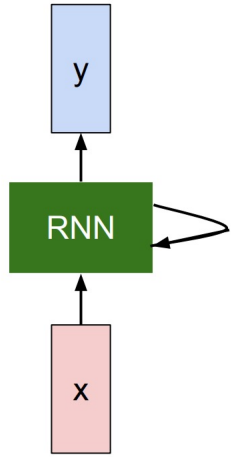
output — new state

another function with parameters W_o



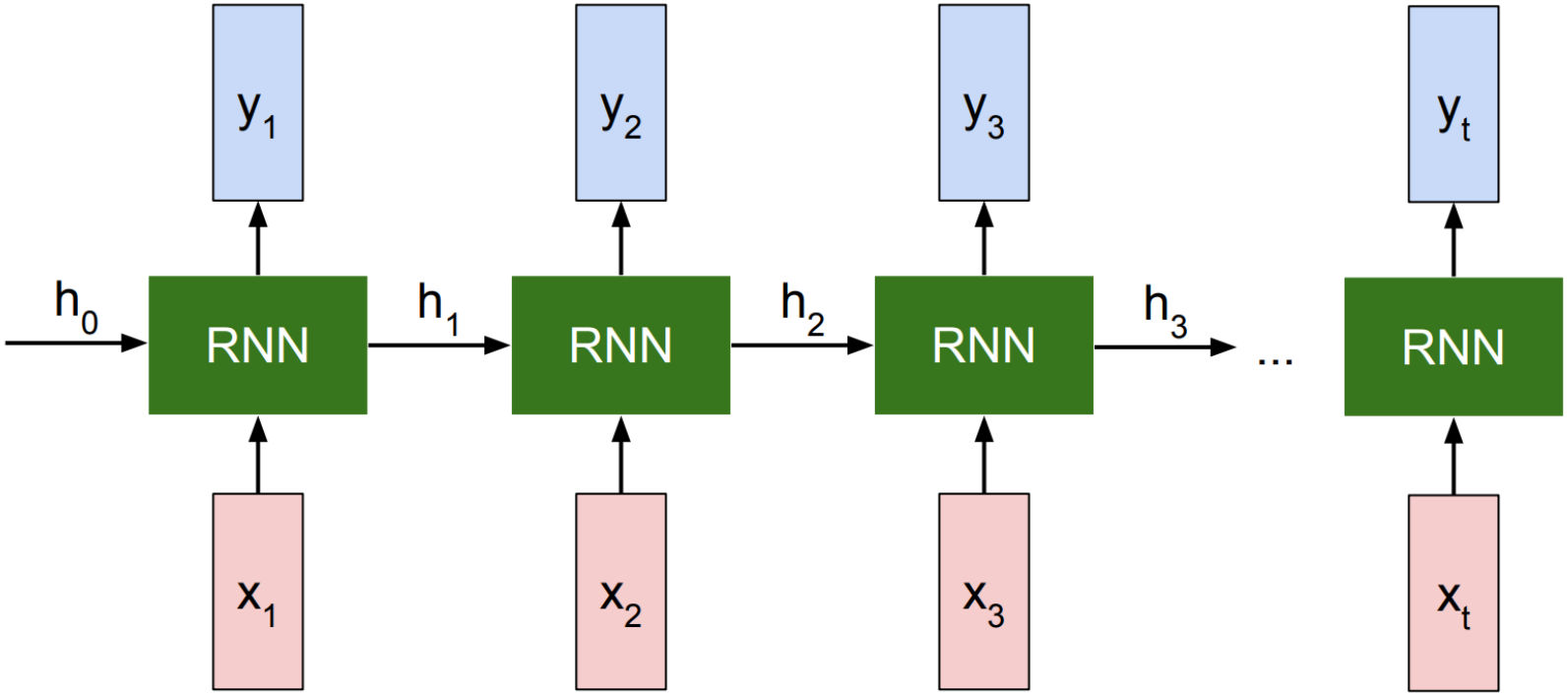
We can process a sequence of vectors **x** by applying a **recurrence formula** at every time step:

$$h_t = f_W(h_{t-1}, x_t)$$



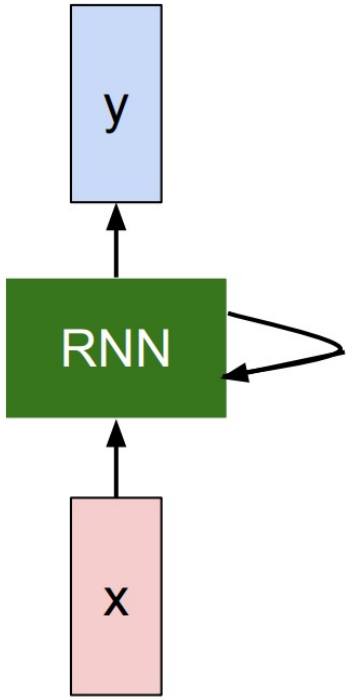
Notice: the same function and the same set of parameters are used at every time step.

RNN



(Vanilla) Recurrent Neural Network

The state consists of a single *"hidden"* vector **h**:



$$h_t = f_W(h_{t-1}, x_t)$$

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

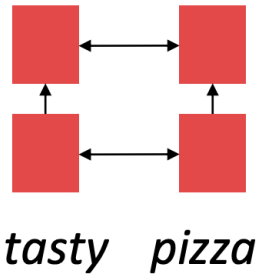
$$y_t = W_{hy}h_t$$

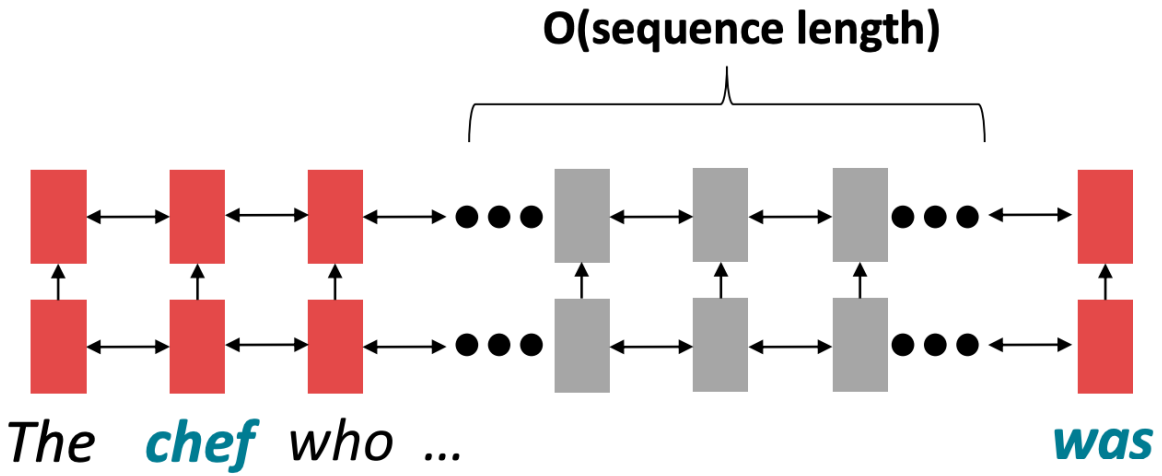
Sometimes called a "Vanilla RNN" or an "Elman RNN" after Prof. Jeffrey Elman

RNNs - Limitations

Linear Interaction Distance

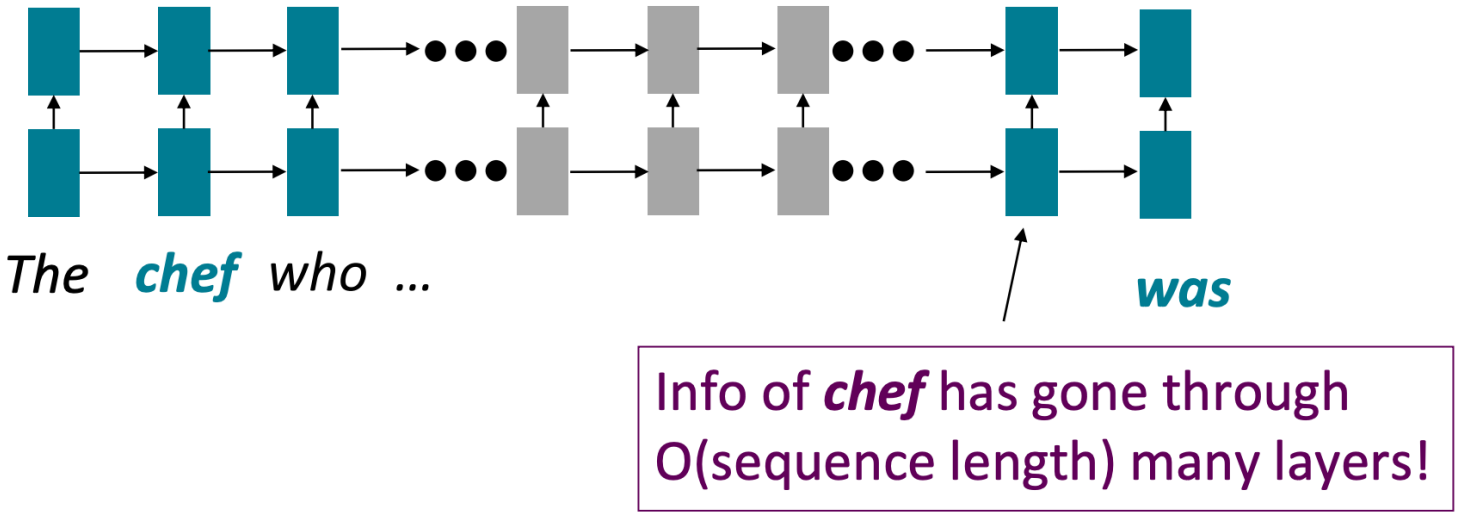
- RNNs are unrolled “left-to-right”.
- This encodes linear locality: a useful heuristic!
  - Nearby words often affect each other’s meanings
- Problem: RNNs take O(sequence length) steps for distant word pairs to interact.





RNNs - Limitations

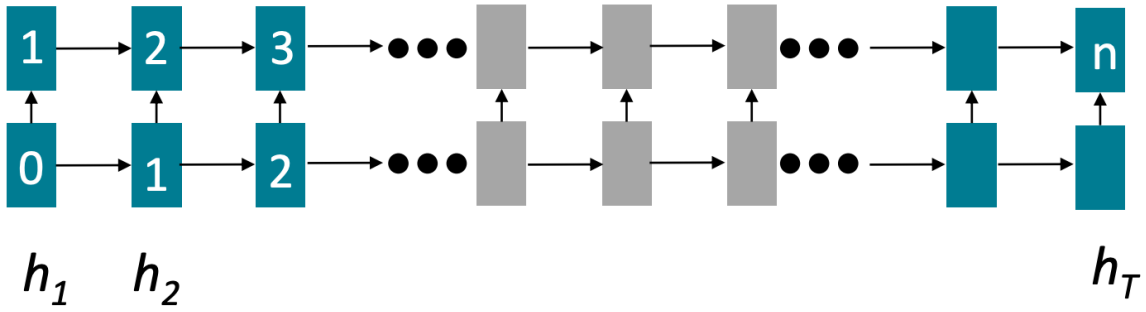
- O(sequence length) steps for distant word pairs to interact means:
  - Hard to learn long-distance dependencies (because gradient problems!)
  - Linear order of words is “baked in”; we already know linear order isn’t the right way to think about sentences



RNNs – Limitations

Lack of parallelizability

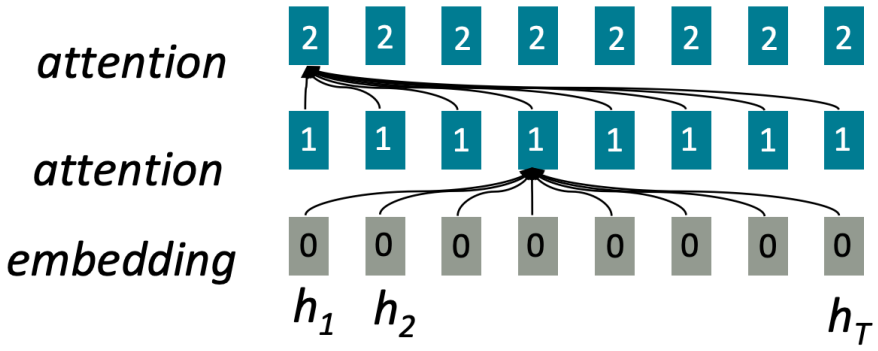
- Forward and backward passes have O(sequence length) unparallelizable operations
  - GPUs can perform a bunch of independent computations at once!
  - But future RNN hidden states can't be computed in full before past RNN hidden states have been computed
  - Inhibits training on very large datasets!



Numbers indicate min # of steps before a state can be computed

Attention - Intuition

- Attention treats each word's representation as a query to access and incorporate information from a set of values.
- Number of unparallelizable operations does not increase with sequence length.
- Maximum interaction distance: O(1), since all words interact at every layer!



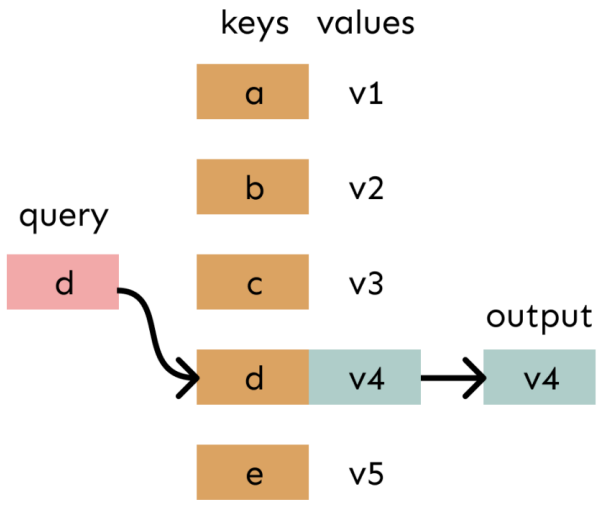
All words attend to all words in previous layer; most arrows here are omitted

Attention

- We can think of attention as performing fuzzy lookup in a key-value store.

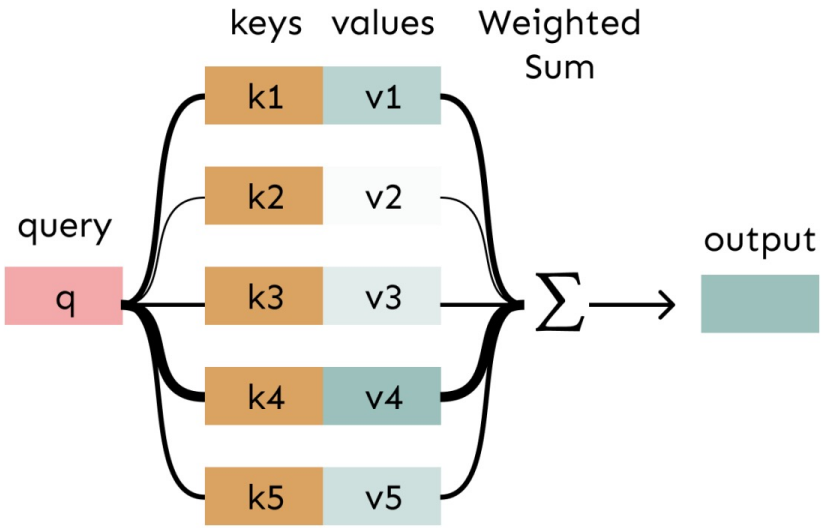
  In a lookup table, we have a table of keys that map to values.

  The query matches one of the keys, returning its value.

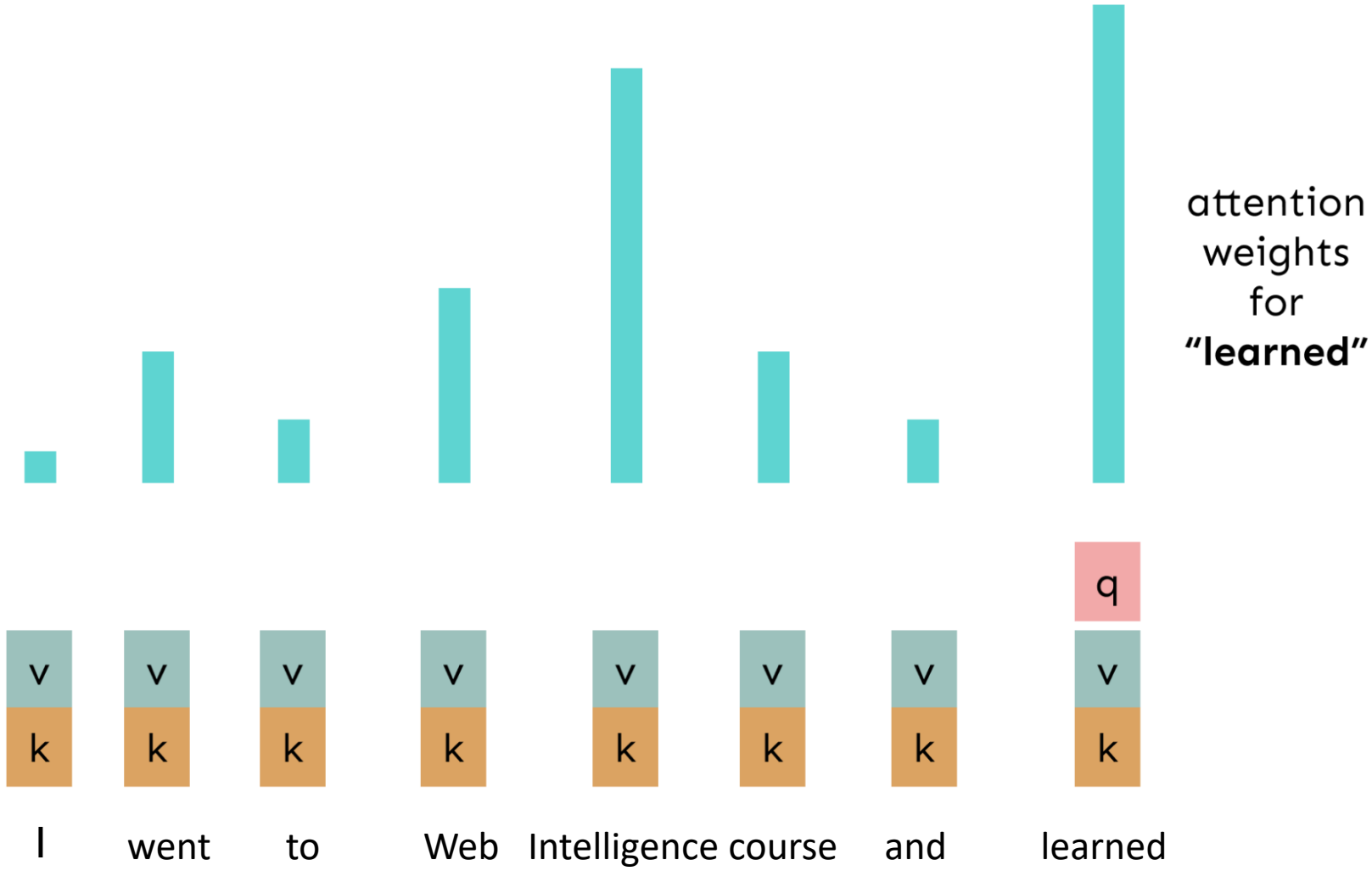


In attention,
the query matches all keys softly, to a weight between 0 and 1.

The keys' values are multiplied by the weights and summed.



Example


attention
weights
for
"learned"
q
v
k
I
went
to
Web
Intelligence
course
and
learned

Self Attention

Let $\boldsymbol{w}_{1:n}$ be a sequence of words in vocabulary V, like *Zuko made his uncle tea*.

For each $\boldsymbol{w}_i$, let $\boldsymbol{x}_i = E\boldsymbol{w}_i$, where $E \in \mathbb{R}^{d \times |V|}$ is an embedding matrix.

1. Transform each word embedding with weight matrices $\mathrm{Q}, \mathrm{K}, \mathrm{V}$, each in $\mathbb{R}^{d \times d}$

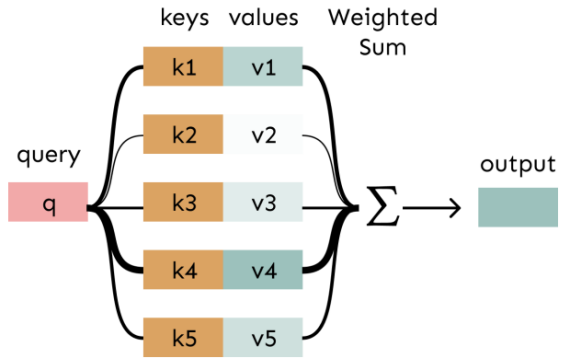
$$\boldsymbol{q}_i = Q\boldsymbol{x}_i \text{ (queries)} \qquad \boldsymbol{k}_i = K\boldsymbol{x}_i \text{ (keys)} \qquad \boldsymbol{v}_i = V\boldsymbol{x}_i \text{ (values)}$$

2. Compute pairwise similarities between keys and queries; normalize with softmax

$$\boldsymbol{e}_{ij} = \boldsymbol{q}_i^\top \boldsymbol{k}_j \qquad \boldsymbol{\alpha}_{ij} = \frac{\exp(\boldsymbol{e}_{ij})}{\sum_{j'} \exp(\boldsymbol{e}_{ij'})}$$

3. Compute output for each word as weighted sum of values

$$\boldsymbol{o}_i = \sum_j \boldsymbol{\alpha}_{ij} \boldsymbol{v}_i$$



Limitations of Self-Attention as a building block

- Doesn’t have an inherent notion of order.
- No nonlinearities for deep learning! It’s all just weighted averages
- Need to ensure we don’t “look at the future” when predicting a sequence
  - Like in machine translation
  - Or language modeling

Self Attention – Sequence Order

- Since self-attention doesn't build in order information, we need to encode the order of the sentence in our keys, queries, and values.
- Consider representing each sequence index as a vector

$\boldsymbol{p}_i \in \mathbb{R}^d$, for $i \in \{1,2,\ldots,n\}$ are position vectors

- Easy to incorporate this info into our self-attention block: just add the $\boldsymbol{p}i$ to our inputs!
- Recall that $\boldsymbol{x}i$ is the embedding of the word at index i. The positioned embedding is:

$$\tilde{\boldsymbol{x}}_i = \boldsymbol{x}_i + \boldsymbol{p}_i$$

In deep self-attention networks, we do this at the first layer! You could concatenate them as well, but people mostly just add…

Self Attention – Sequence Order

Position representation vectors learned from scratch

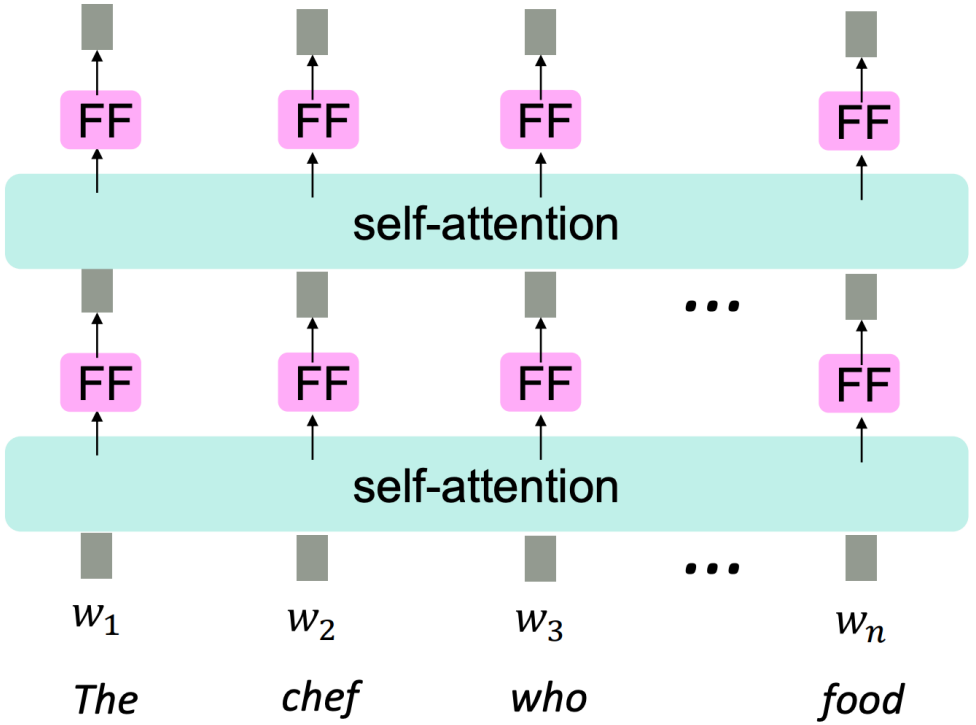
- Learned absolute position representations: Let all pi be learnable parameters
- Learn a matrix $\boldsymbol{p} \in \mathbb{R}^{d \times n}$, and let each $\boldsymbol{p}i$ be a column of that matrix
- Pros:
  - Flexibility: each position gets to be learned to fit the data
- Cons:
  - Definitely can't extrapolate to indices outside 1, … , n.
- Most systems use this!
- Sometimes people try more flexible representations of position:
  - Relative linear position attention [Shaw et al., 2018]

Self Attention – Adding Nonlinearities

- Note that there are no elementwise nonlinearities in self-attention; stacking more self-attention layers just re-averages value vectors
- Easy fix: add a feed-forward network to post-process each output vector.

$$m_i = MLP(\text{output}_i)$$
$$= W_2 * \text{ReLU}(W_1\ \text{output}_i + b_1) + b_2$$

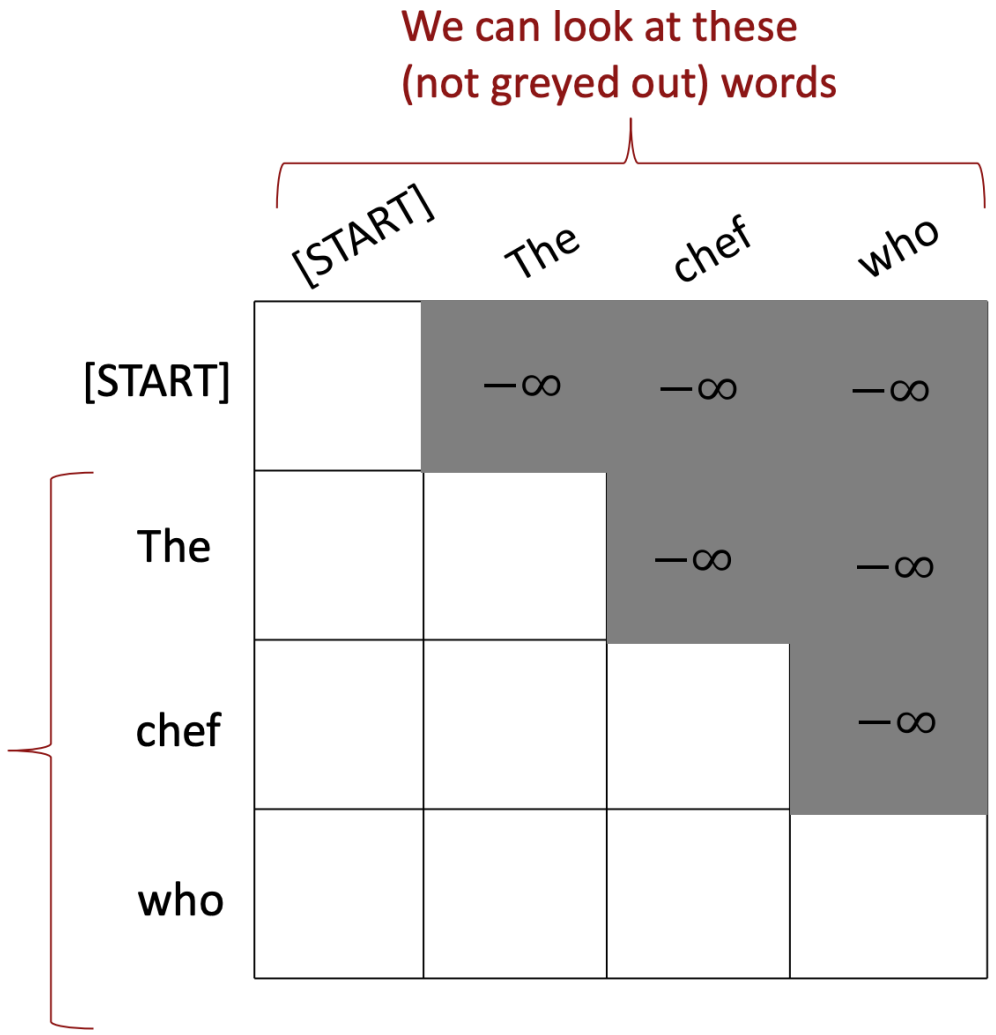
Intuition: the FF network processes the result of attention



Self Attention – Masking the future

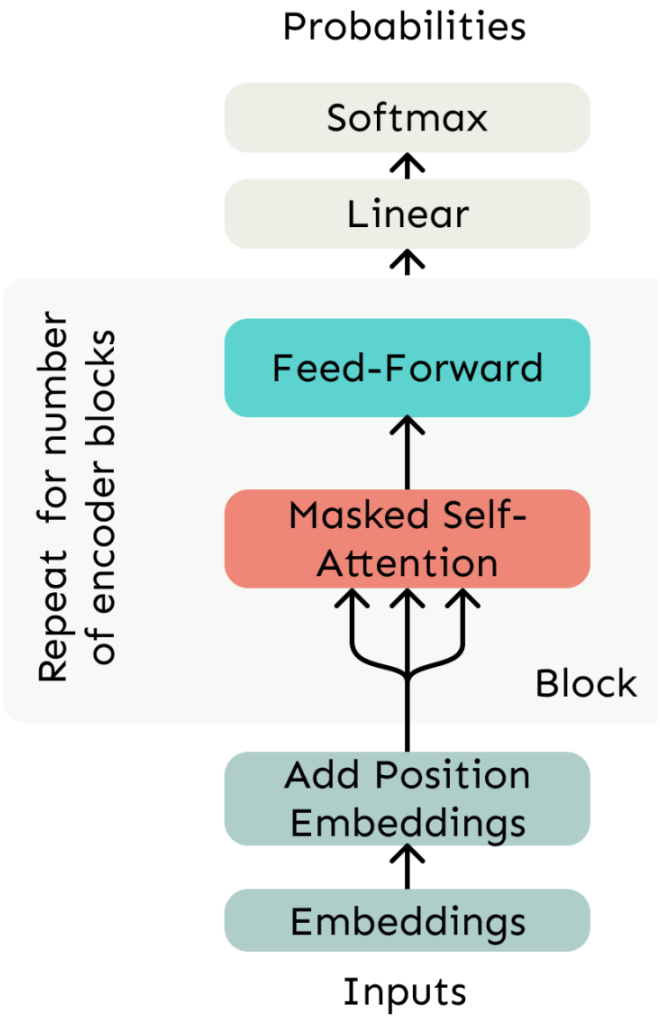
- To use self-attention in **decoders**, we need to ensure we can't peek at the future.
- At every timestep, we could change the set of **keys and queries** to include only past words. (Inefficient!)
- To enable parallelization, we **mask out attention** to future words by setting attention scores to $-\infty$.

$$e_{ij} = \begin{cases} q_i^\top k_j, j \leq i \\ -\infty, j > i \end{cases}$$



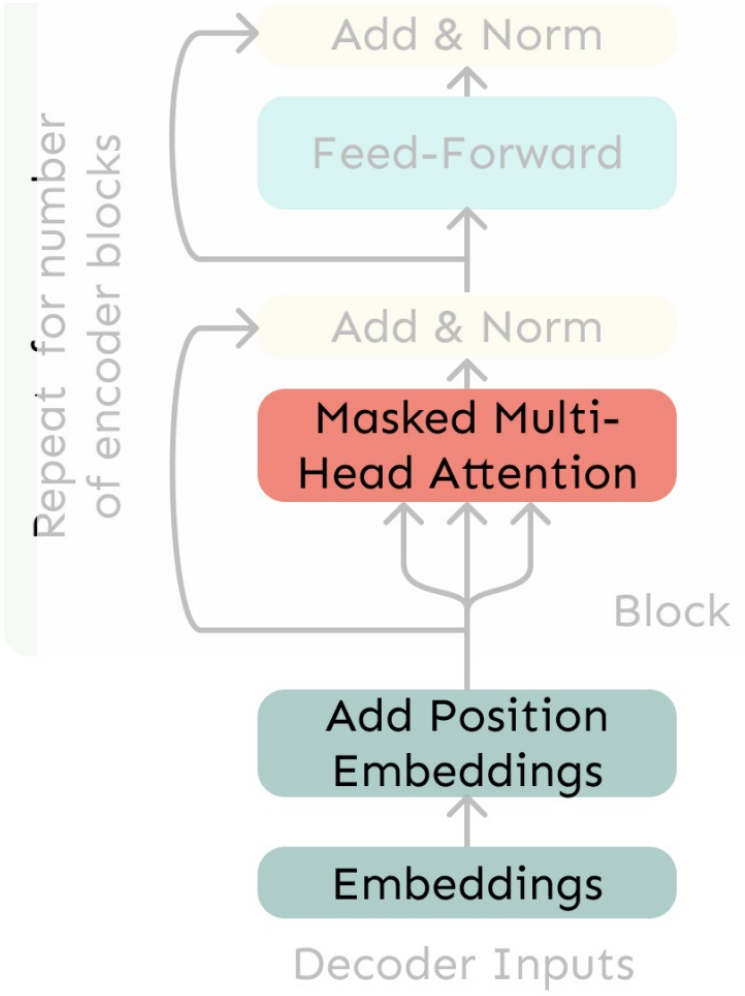
Self Attention Building Block

- **Self-attention**:
  - the basis of the method.
- **Position representations**:
  - Specify the sequence order, since self-attention is an unordered function of its inputs.
- **Nonlinearities**:
  - At the output of the self-attention block
  - Frequently implemented as a simple feed-forward network.
- **Masking**:
  - In order to parallelize operations while not looking at the future.
  - Keeps information about the future from "leaking" to the past.



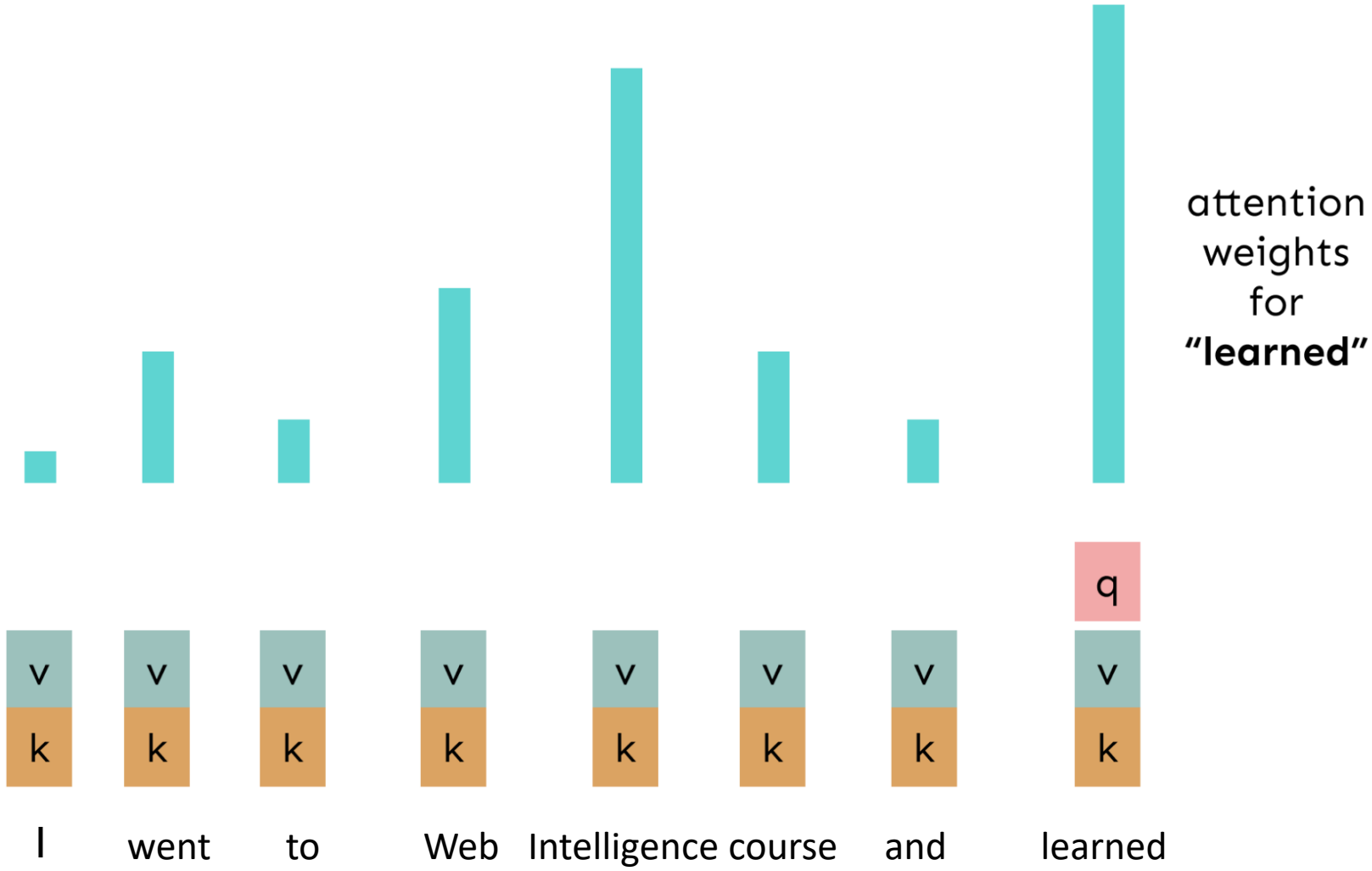
The Transformer Decoder

- A Transformer decoder is how we'll build systems like **language models**.
- It's a lot like our minimal self-attention architecture, but with a few more components.
- The embeddings and position embeddings are identical.
- We'll next replace our self-attention with **multi-head self-attention.**

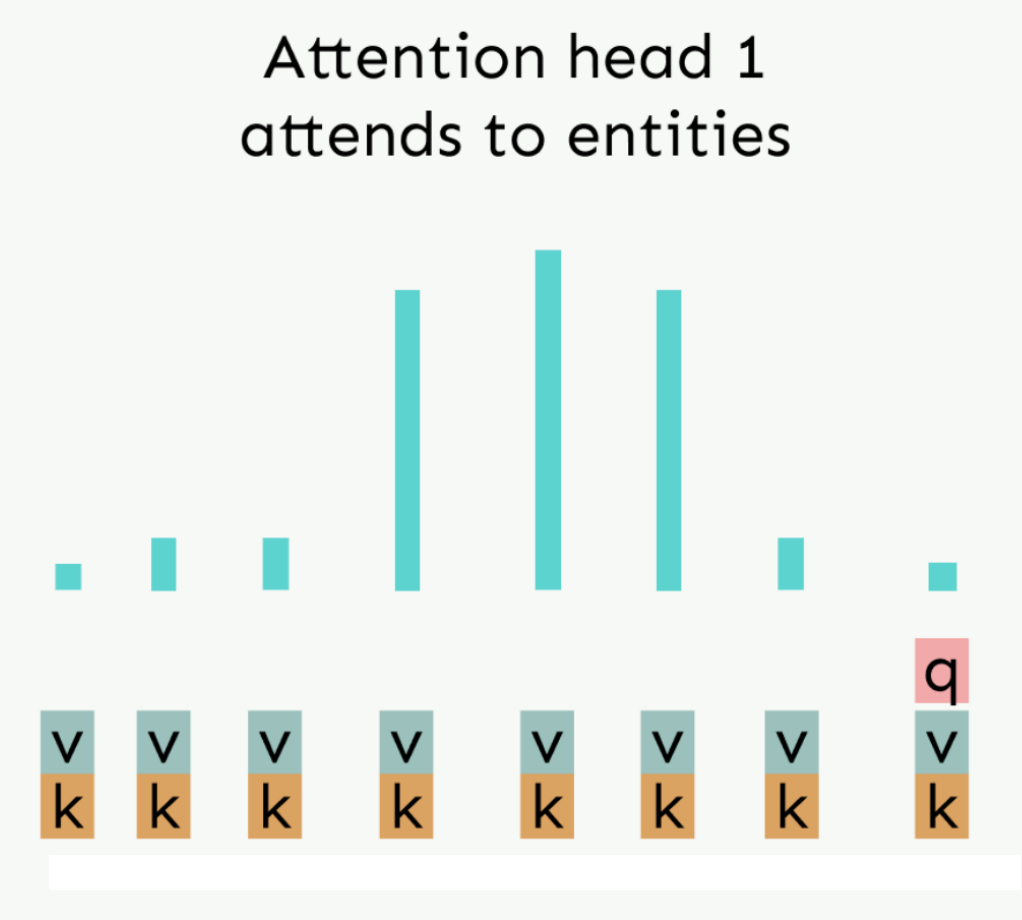


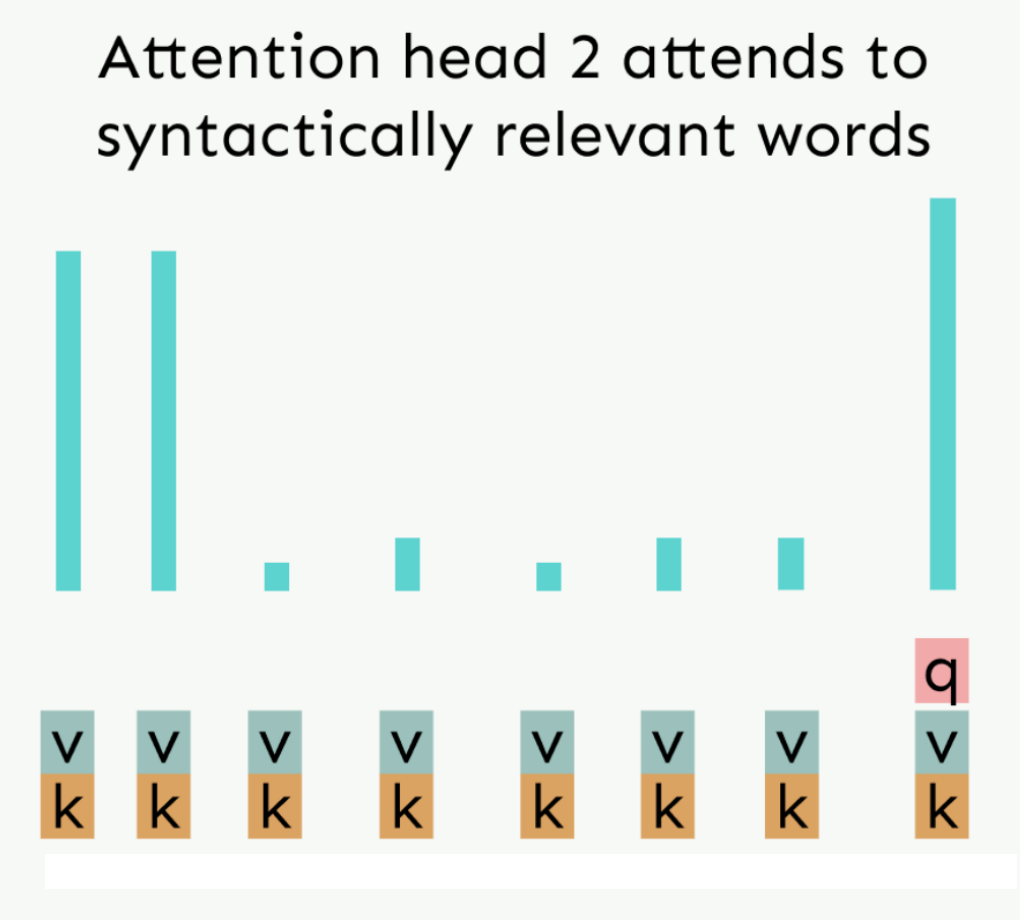
Transformer Decoder

Example


attention
weights
for
"learned"
q
v
k
I
went
to
Web
Intelligence
course
and
learned

Example





I went to Web Intelligence course and learned

Literature

- https://web.stanford.edu/class/cs224n/readings/cs224n-self-attention-transformers-2023_draft.pdf
- https://arxiv.org/abs/1810.04805