Word 2 Vec

One-hot vector ⇒ orthogonal to each other

⇒ no natural notion of "similarity"

High dimensional vector ⇒ semantic spaces

⇒ vectors with similar meaning are closeby to each other

- **Defination of Similarity**: Two vectors are similar ⇔ high dot product
- **Turning Similarity into Probability**: softmax function $\mathbb{R}^n \to (0,1)^n$

② exponentiation makes everything positive

① dot product ⇔ similarity

$$P(o|c) = \frac{\exp(u_o^T v_c)}{\sum_{w \in V} \exp(u_w^T v_c)}$$

③ normalize over entire vocabulary to give probability distribution

$$\text{softmax}(x_i) = \frac{\exp(x_i)}{\sum_{j=1}^{n} \exp(x_j)} = p_i$$

softmax: amplifies probability of largest x_i; still assigns some probabilities to smaller x_i

☆ Interesting semantic patterns emerge in the scaled vectors, and they are linear.

Gradient Descent

Update equation in matrix notation:

$$\theta^{new} = \theta^{old} - \alpha \nabla_\theta J(\theta)$$

α: step size or learning rate

Update equation for each single parameter θ_j:

$$\theta_j^{new} = \theta_j^{old} - \alpha \frac{\partial}{\partial \theta_j^{old}} J(\theta)$$

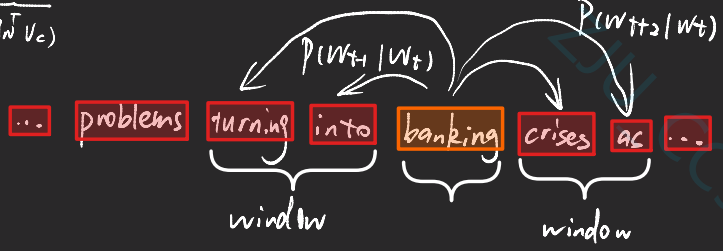
$J(\theta)$: loss function

- **Problem**: very expensive to compute the gradient of the entire corpus
- **Solution**: Stochastic gradient descent (SGD)

  Repeatedly sample windows, only compute the gradient of the window and use it to update the entire corpus.

Main Idea of Word2Vec:

- Start with random word vectors
- Iterate through each word position in the whole corpus.
- Try to predict surrounding words using word vectors: $P(o|c) = \frac{\exp(u_o^T v_c)}{\sum_{w \in V} \exp(u_w^T v_c)}$



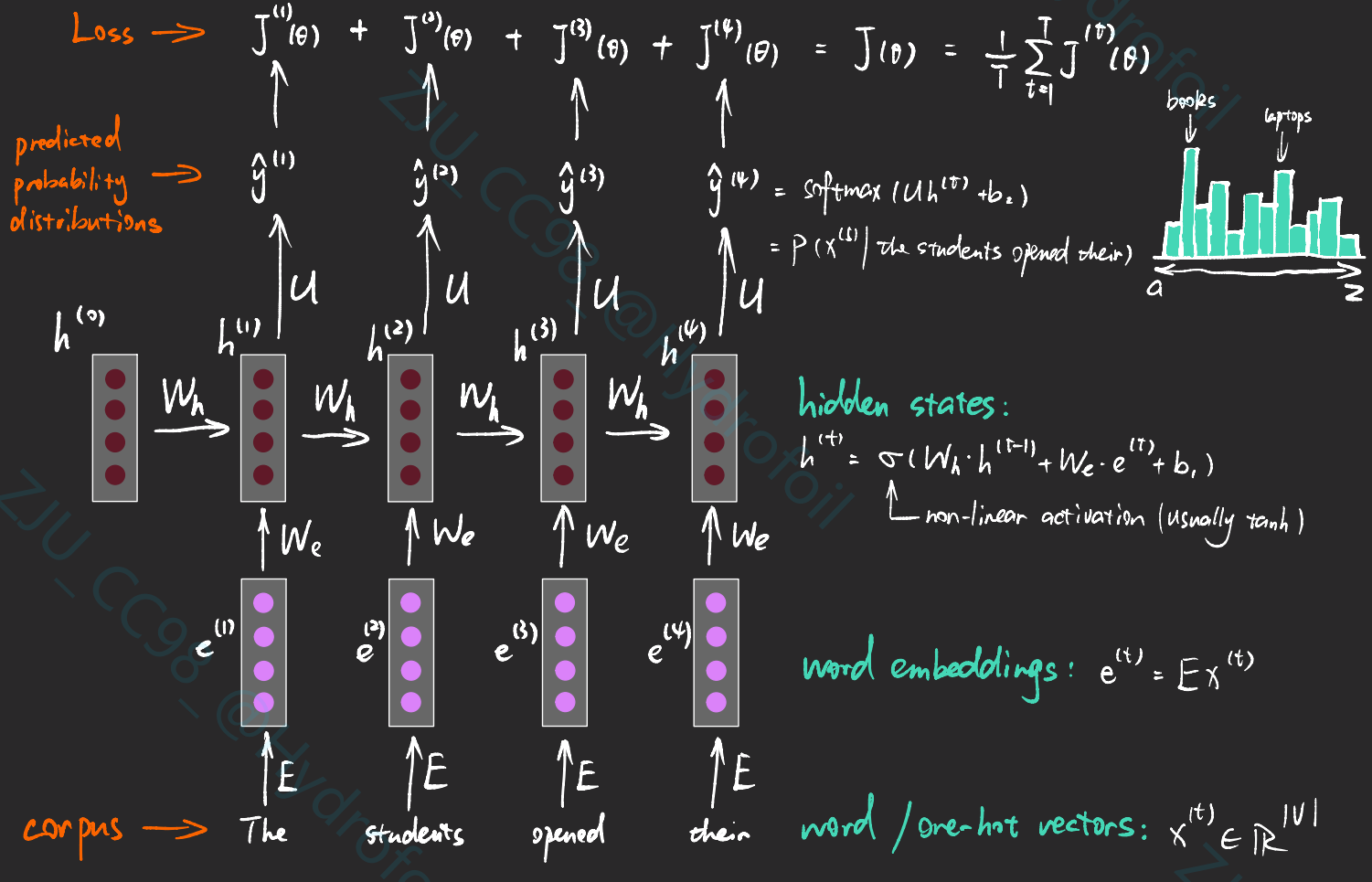
- Learning: Update vectors so each word is at more appropriate position in the semantic space.

☆ Polysemy: different senses of a word reside in a linear weighted sum in standard word embeddings like word2vec

$v_{pike} = a_1 v_{pike_1} + a_2 v_{pike_2} + a_3 v_{pike_3}$

where $a_1 = \frac{f_1}{f_1+f_2+f_3}$, etc., for frequency f.

RNN:



hidden states: $h^{(t)} = \sigma(W_h \cdot h^{(t-1)} + W_e \cdot e^{(t)} + b_1)$

↳ non-linear activation (usually tanh)

word embeddings: $e^{(t)} = Ex^{(t)}$

word / one-hot vectors: $x^{(t)} \in \mathbb{R}^{|V|}$

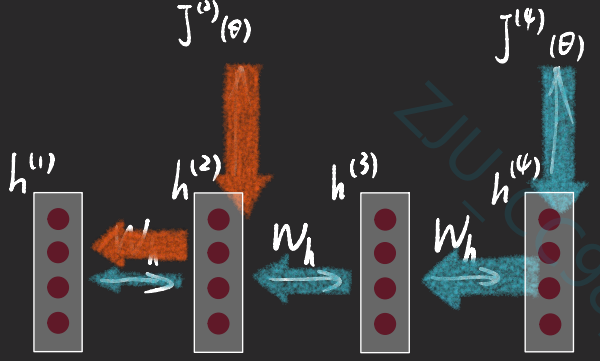
$$J^{(t)}(\theta) = \text{cross\_entropy}(y^{(t)}, \hat{y}^{(t)}) = -\sum_{w \in V} y^{(t)} \log \hat{y}_w^{(t)} = -\log \hat{y}_{x_{t+1}}^{(t)}$$

RNN Advantages:

- Can process any length input.
- Computation for step t can use information from many step back.
- Model size doesn't increase for longer input.

RNN Disadvantages:

- Recurrent computation is slow (compute in loop, not parallel)
- In practice, difficult to access information from many steps back due to Vanishing Gradients.



- Exploding Gradients.

LSTM:

Forget gate: controls what's kept vs. forgotten, from previous cell state.

Input gate: controls what parts of the new cell content are written to cell.

Output gate: controls what parts of cell are output to hidden state.

sigmoid

$$\begin{pmatrix} f^{(t)} \\ i^{(t)} \\ o^{(t)} \end{pmatrix} = \sigma \left(\begin{pmatrix} W_f \\ W_i \\ W_o \end{pmatrix} h^{(t-1)} + \begin{pmatrix} U_f \\ U_i \\ U_o \end{pmatrix} x^{(t)} + \begin{pmatrix} b_f \\ b_i \\ b_o \end{pmatrix} \right)$$

New cell content: the new content that is to be written to the cell.

$$\tilde{c}^{(t)} = \tanh(W_c h^{(t-1)} + U_c x^{(t)} + b_c)$$

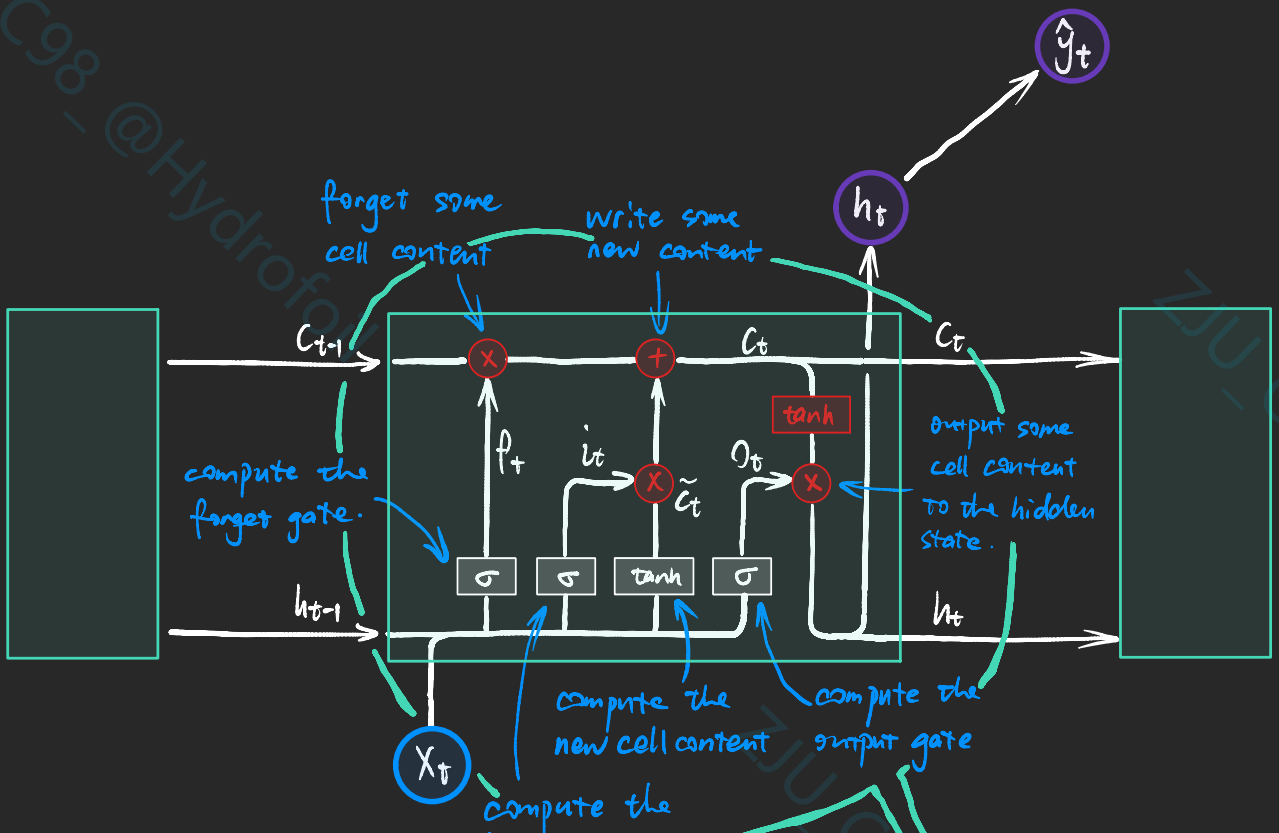
Cell state: erase ("forget") some content from last cell state, and write ("input") some new content.

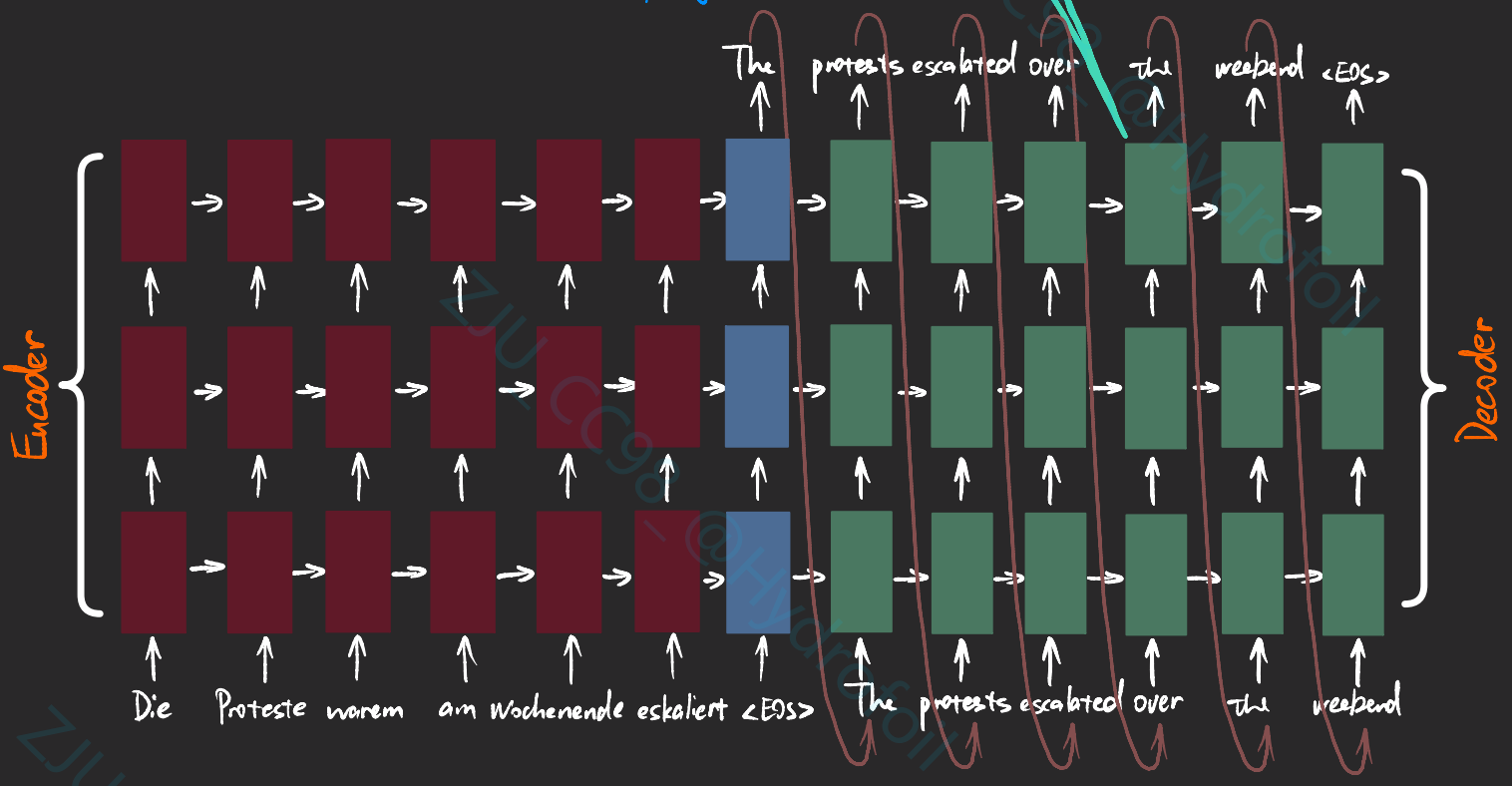
$$c^{(t)} = f^{(t)} \odot c^{(t-1)} + i^{(t)} \odot \tilde{c}^{(t)}$$

Hidden state: read ("output") some content from the cell.

$$h^{(t)} = o^{(t)} \odot \tanh c^{(t)}$$

Multi-layer deep encoder-decoder machine translation net

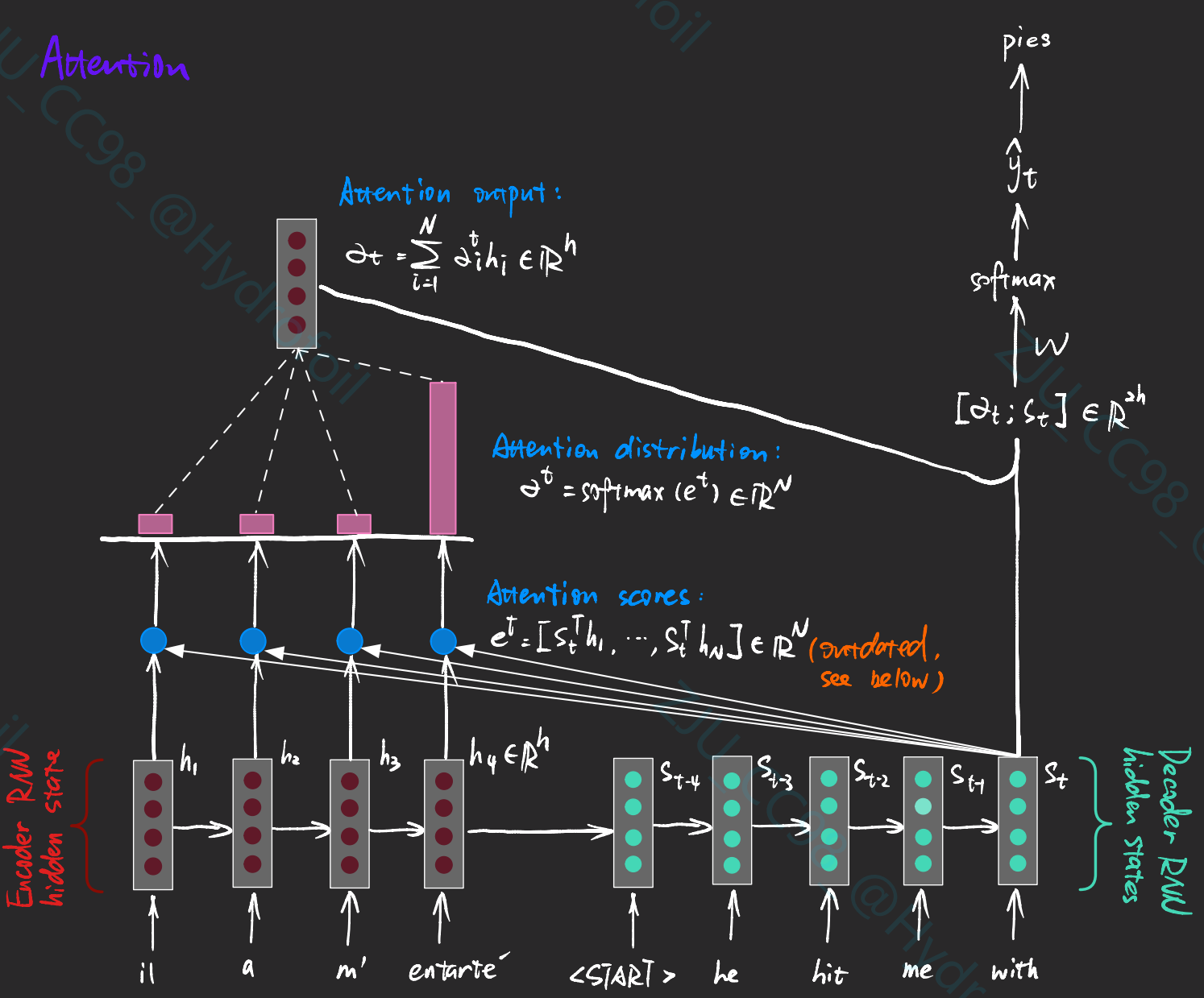




sequence to sequence model: ① input a sequence → ② output a sequence.

Bottleneck problem of sequence to sequence model: Encoding needs to capture all information about the source sentence, and stuff it into the last vector of the encoder RNN. It's difficult when the source sentence is long.

Attention



Computing attention scores: compute $e \in \mathbb{R}^N$ from $h_1, \cdots, h_N \in \mathbb{R}^{d_1}$ and $s \in \mathbb{R}^{d_2}$

- Basic dot-product attention: $e_i = s^T h_i \in \mathbb{R}$ (assumes $d_1 = d_2$)
- Multiplicative attention: $e_i = s^T W h_i \in \mathbb{R}$
  where $W \in \mathbb{R}^{d_2 \times d_1}$ is a weight matrix
  problem: too many parameters in W when $d_1 d_2$ is big.
- Reduced-rank multiplicative attention: $e_i = s^T (U^T V) h_i = (Us)^T (Vh_i)$
  For low rank matrices $U \in \mathbb{R}^{k \times d_2}$, $V \in \mathbb{R}^{k \times d_1}$, $k \ll d_1, d_2$

Attention improves NMT performance

- provides a more "human-like" model (look back to the source sentence, rather than remember it all)
- solves the bottleneck problem
- helps with the vanishing gradient problem
- provides some interpretability (we see what the decoder was focusing on)

Self-Attention

- word sequence: $w_{1:n} = [w_1, \dots, w_n] \in \mathbb{R}^{|V| \times n}$ (one-hot)
  where $|V|$ is the vocabulary size.
- embedding: for each w_i, let $x_i = E w_i \in \mathbb{R}^{d \times 1}$
  where $E \in \mathbb{R}^{d \times |V|}$ is the embedding matrix.
- position embedding: $\tilde{x}_i = x_i + p_i$ where p_i is the position vector.
- transform each word embedding with weight matrices Q, K, V in $\mathbb{R}^{d \times d}$:

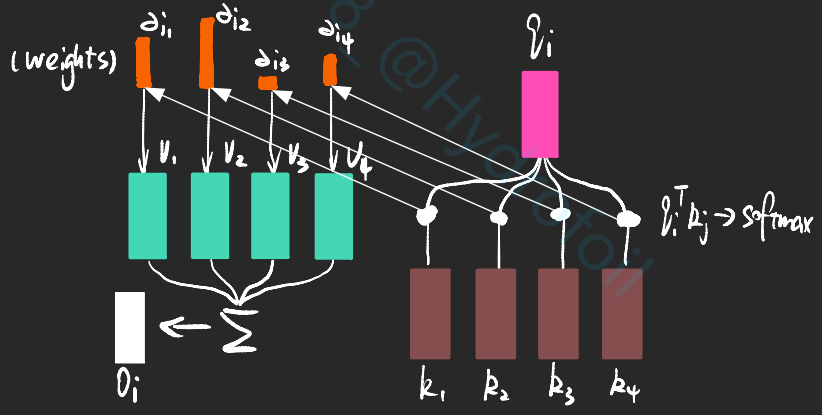
$q_i = Q x_i$ (queries) $\quad k_i = K x_i$ (keys) $\quad v_i = V x_i$ (values) $\quad q_i, k_i, v_i \in \mathbb{R}^{d \times 1}$

- compute pairwise similarities with keys and queries, normalize with softmax:

$$e_{ij} = q_i^T k_j \in \mathbb{R} \qquad \alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{j'} \exp(e_{ij'})} \in \mathbb{R}$$

- compute output for each word as weighted sum of values:

$$o_i = \sum_j \alpha_{ij} v_j \in \mathbb{R}^{d \times 1}$$



- future mask: to use self-attention in the decoders, we need to ensure we can't peek at the future. Otherwise the model can't learn anything during the training.

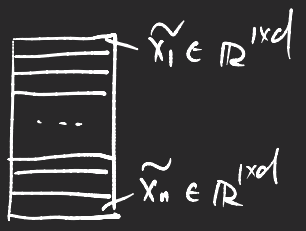
$$e_{ij} = \begin{cases} q_i^T k_j, & j \le i \\ -\infty, & j > i \end{cases}$$

We can look at these (not shadowed out) words

For encoding these words

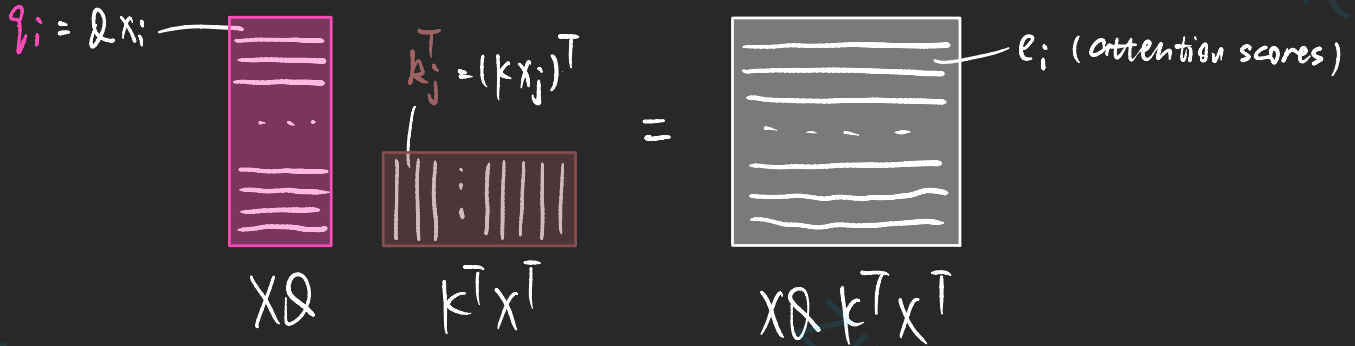
	<START>	The	chef	who
<START>		$-\infty$	$-\infty$	$-\infty$
The			$-\infty$	$-\infty$
chef				$-\infty$
who				

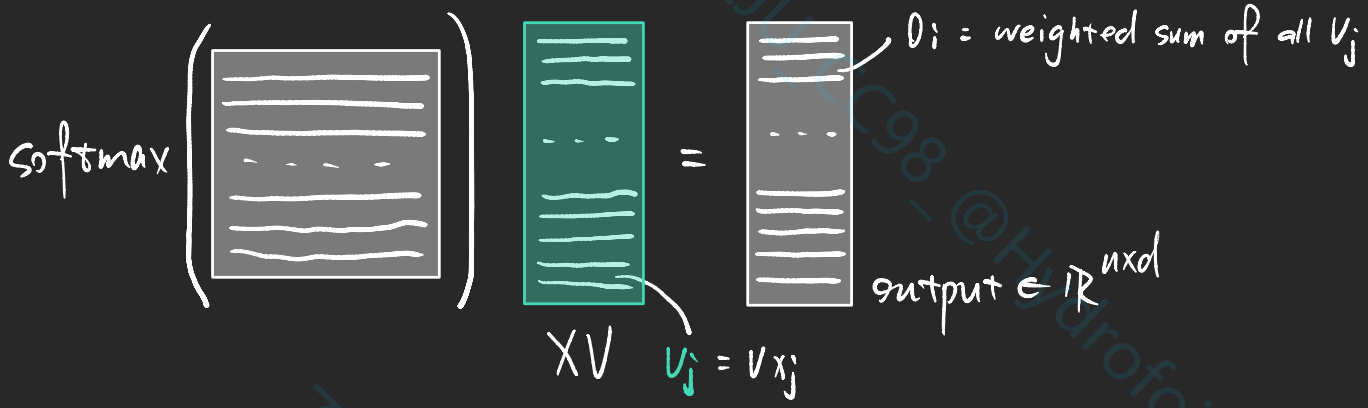
Sequence-Stacked form of Attention:

$X = [\tilde{x}_1; \tilde{x}_2; \cdots; \tilde{x}_n] \in \mathbb{R}^{n \times d}$ i.e. $X =$ 

$XK \in \mathbb{R}^{n \times d}$, $XQ \in \mathbb{R}^{n \times d}$, $XV \in \mathbb{R}^{n \times d}$

output $= \text{softmax}(XQ(XK)^T)XV \in \mathbb{R}^{n \times d}$



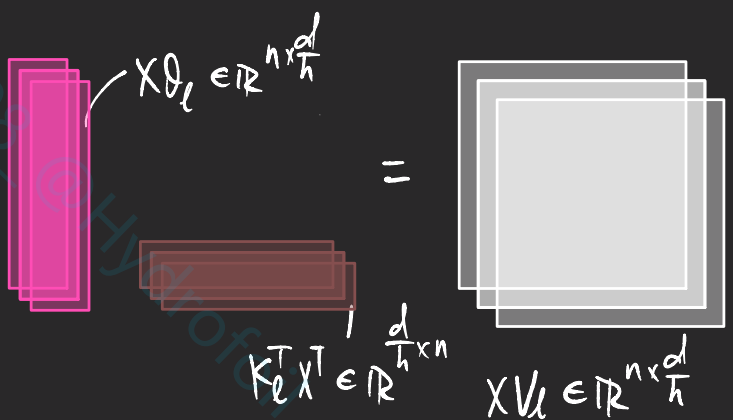


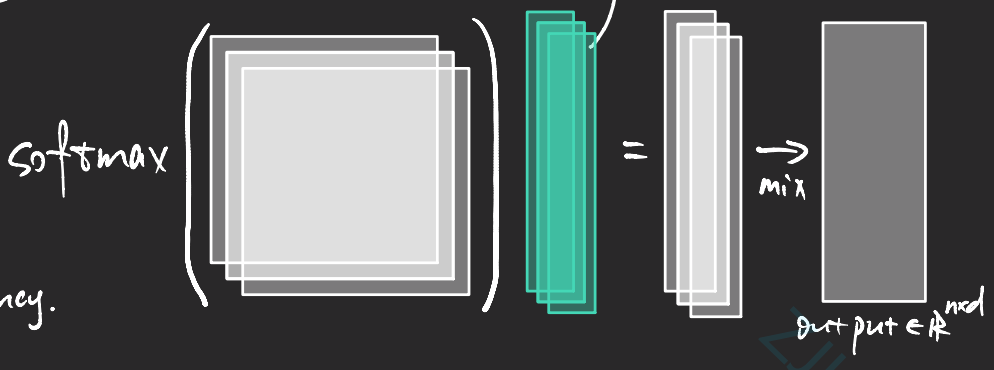
Multi-Head Attention

Let, $Q_\ell, K_\ell, V_\ell \in \mathbb{R}^{d \times \frac{d}{h}}$, where h is the number of attention heads, and ℓ ranges from 1 to h.

So that $XQ_\ell \in \mathbb{R}^{n \times \frac{d}{h}}$, likewise for XK_ℓ and XV_ℓ.

The projection down to lower dimensional space $\mathbb{R}^{n \times \frac{d}{h}}$ is for computational efficiency.





Scaled Dot Product: $\text{output}_i = \text{Softmax}\left(\frac{XQ_iK_i^TX^T}{\sqrt{d/h}}\right)\cdot XV_i$

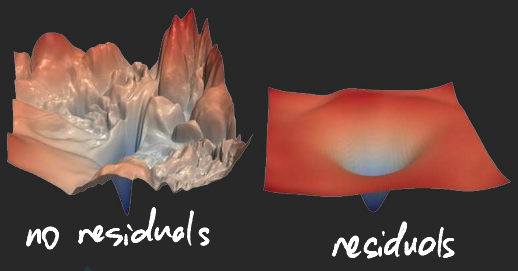
This is to prevent the case when dimensionality d becomes large

⇒ dot products between vectors tend to become large

⇒ inputs to the softmax function can be large

⇒ the gradient is small.

Residual connections:



- when the gradient of the layer is small, it's at least 1 due to the additive gate.
- make the loss surface smoother, and better to escape local optimum.

Layer normalization:

- problem: Difficult to train the parameters of a given layer because its input from the layer beneath keeps shifting.
- solution: mean: $\mu = \frac{1}{H}\sum_{i=1}^{H} a_i$

standard diviation: $\sigma = \sqrt{\frac{1}{H}\sum_{i=1}^{H}(a_i-\mu)^2}$

$\Rightarrow$ output $= \frac{x-\mu}{\sigma+\varepsilon}\cdot\gamma+\beta$

Transformer