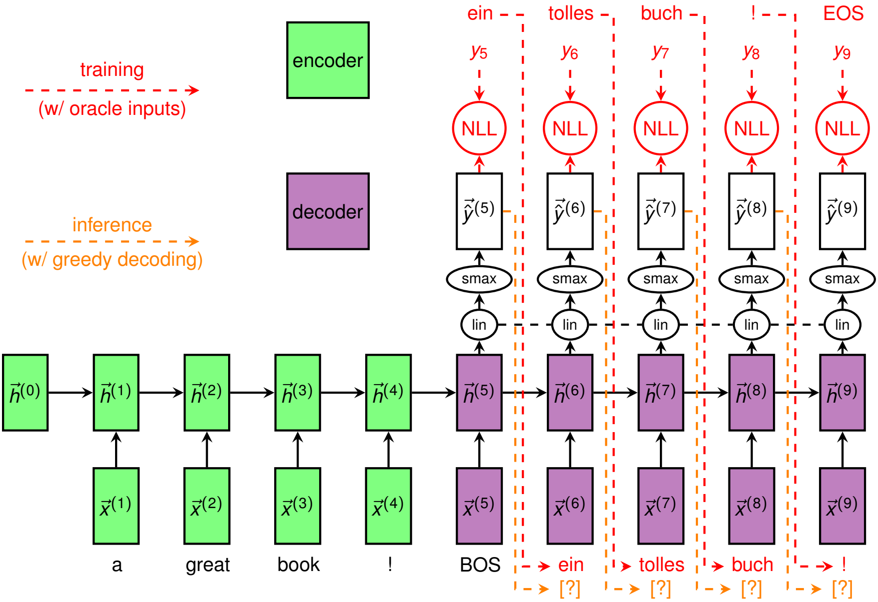
Advanced NN Architectures

Attention

Learning goals

- Understand attention mechanism
- Learn the different types of attention

ENCODER-DECODER WITH RNNS



OTHER ENCODER-DECODER APPLICATIONS

- Text summarization
- Text generation
- Keyword generation
- Automatic speech recognition
- Subtitle generation
- Question answering
- Named entity recognition
- Video or image captioning
- Part-of-speech tagging
- ...more

LIMITATIONS OF RNNS

- In an RNN, at a given point in time j, the information about all past inputs $x^{(1)} \dots x^{(j)}$ is "crammed" into the state vector $\vec{h}^{(j)}$ (and $\vec{c}^{(j)}$ for an LSTM)
- So for long sequences, the state becomes a bottleneck
- Especially problematic in encoder-decoder models (e.g., for Machine Translation)
- Solution: Attention (Bahdanau et al., 2015) – an architectural modification of the RNN encoder-decoder that allows the model to "attend to" past encoder states

ATTENTION: THE BASIC RECIPE (1)

- **Ingredients:**

- One query vector: $\mathbf{q} \in \mathbb{R}^{d_q}$
- J key vectors: $\mathbf{K} \in \mathbb{R}^{J \times d_k}$; $(\vec{k}_1 \dots \vec{k}_J)$
- J value vectors: $\mathbf{V} \in \mathbb{R}^{J \times d_v}$; $(\vec{v}_1 \dots \vec{v}_J)$
- Scoring function $a : \mathbb{R}^{d_q} \times \mathbb{R}^{d_k} \rightarrow \mathbb{R}$
  - Maps a query-key pair to a scalar ("score")
  - a may be parametrized by parameters θ_a

ATTENTION: THE BASIC RECIPE (2)

- **Step 1**: Apply a to $\vec{q}$ and all keys $\vec{k}_j$ to get scores (one per key):

$$\vec{e} = \begin{bmatrix} e_1 \\ \vdots \\ e_J \end{bmatrix} = \begin{bmatrix} a(\vec{q}, \vec{k}_1; \theta_a) \\ \vdots \\ a(\vec{q}, \vec{k}_J; \theta_a) \end{bmatrix}$$

- **Step 2**: Turn **e** into a probability distribution with the softmax function

$$\alpha_j = \frac{\exp(e_j)}{\sum_{j'=1}^{J} \exp(e_{j'})}$$

  - Note that $\sum_j \alpha_j = 1$

ATTENTION: THE BASIC RECIPE (3)

- **Step 3**: α-weighted sum over $\vec{V}$ yields one d_v-dimensional output vector:

$$\vec{o} = \sum_{j=1}^{J} \alpha_j \vec{v}_j$$

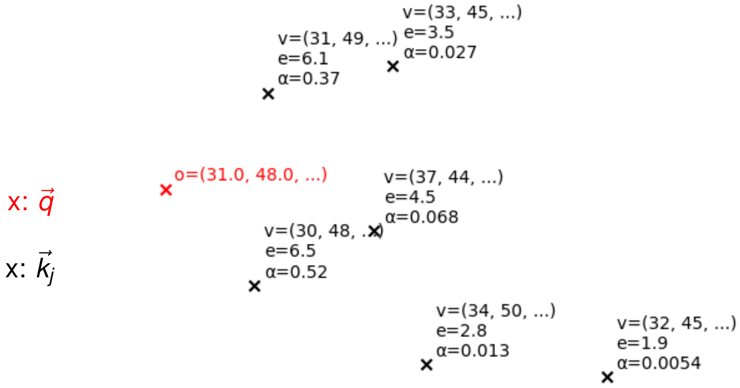
  - Intuition: α_j is how much "attention" the model pays to $\vec{v}_j$ when computing $\vec{o}$.

ATTENTION: AN ANALOGY (1)

- We have J weather stations on a map
- $\vec{K} \in \mathbb{R}^{J \times 2}$ are their geolocations (x,y coordinates)
- $\vec{V} \in \mathbb{R}^{J \times d_v}$ are their current weather conditions (temperature, humidity, etc.)
- $\vec{q} \in \mathbb{R}^2$ is a new geolocation for which we want to estimate weather conditions
- e_j is the relevance of the j'th station (e.g., $e_j = a(\vec{q}, \vec{k}_j) = \frac{1}{||\vec{q} - \vec{k}_j||_2}$), and α_j is e_j as a probability

ATTENTION: AN ANALOGY (2)

- $\vec{o}$: a weighted sum of all known weather conditions, where stations that have a small distance (high α) have a higher weight



ATTENTION IN NEURAL NETWORKS (1)

- Contrary to our geolocation example, the $\vec{q}$, $\vec{k}_j$ and $\vec{v}_j$ vectors of a neural network are produced as a function of the input and some trainable parameters
- So the *model* learns which keys are relevant for which queries, based on the training data and loss function

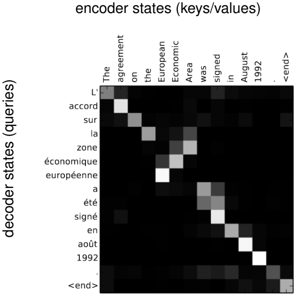


Figure from Bahdanau et al. 2015: Neural Machine Translation by Jointly Learning to Align and Translate.

ATTENTION IN NEURAL NETWORKS (2)

- No (or few) assumptions are baked into the architecture (no notion of which words are neighbors in the sentence, sequentiality, etc.)
- The lack of prior knowledge often means that the Transformer requires more training data than an RNN/CNN to achieve a certain performance
- But when presented with sufficient data, it usually outperforms them
- After Christmas: Transfer learning as a way to pretrain Transformers on **lots** of data

THE TRANSFORMER ARCHITECTURE (1)

- The Bahdanau model is still an RNN, just with attention on top.
- Architecture that consists of attention only: Transformer (Vaswani et al. (2017), “Attention is all you need”)

THE TRANSFORMER ARCHITECTURE (2)

(For simpler problems (e.g., classification, tagging), you would simply use the encoder.)

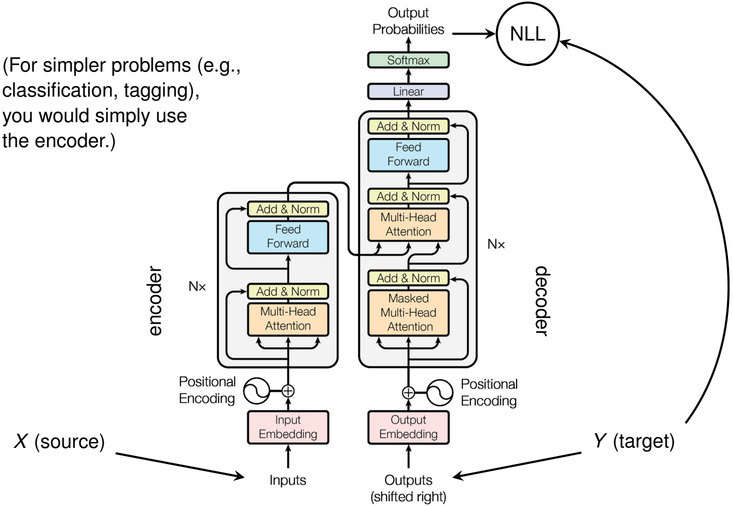


Figure from Vaswani et al. 2017: Attention is all you need

CROSS-ATTENTION AND SELF-ATTENTION

- We can use attention on many different “things”, including:
  - The pixels of images
  - The nodes of knowledge graphs
  - The words of a vocabulary
- Here, we focus on scenarios where the query, key and value vectors represent tokens (e.g., words, characters, etc.) in sequences (e.g., sentences, paragraphs, etc.).
- Cross-attention:
  - Let $X = (x_1 \dots x_{J_x})$, $Y = (y_1 \dots y_{J_y})$ be two sequences (e.g., source and target in a sequence-to-sequence problem)
  - The query vectors represent tokens in Y and the key/value vectors represent tokens in X (“Y attends to X”)
- Self-attention:
  - There is only one sequence $X = (x_1 \dots x_J)$
  - The query, key and value vectors represent tokens in X (“X attends to itself”)

CROSS-ATTENTION (1)

- Here, we describe cross-attention. Self-attention can easily be derived by assuming $\vec{X} = \vec{Y}$.
- Let $\vec{X} \in \mathbb{R}^{J_x \times d_x}$, $\vec{Y} \in \mathbb{R}^{J_y \times d_y}$ be representations of X, Y (e.g., stacked word embeddings, or the outputs of a previous layer)
- Let $\theta = \{\vec{W}^{(q)} \in \mathbb{R}^{d_y \times d_q}, \vec{W}^{(k)} \in \mathbb{R}^{d_x \times d_k}, \vec{W}^{(v)} \in \mathbb{R}^{d_x \times d_v}\}$ be trainable weight matrices
- We transform $\vec{Y}$ into a matrix of query vectors:

$$\vec{Q} = \vec{Y}\vec{W}^{(q)}$$

- We transform $\vec{X}$ into matrices of key and value vectors:

$$\vec{K} = \vec{X}\vec{W}^{(k)}; \qquad \vec{V} = \vec{X}\vec{W}^{(v)}$$

CROSS-ATTENTION (2)

- To calculate the e scores (step 1 of the basic recipe), Vaswani et al. use a parameter-less scaled dot product instead of Bahdanau's complicated FFN:

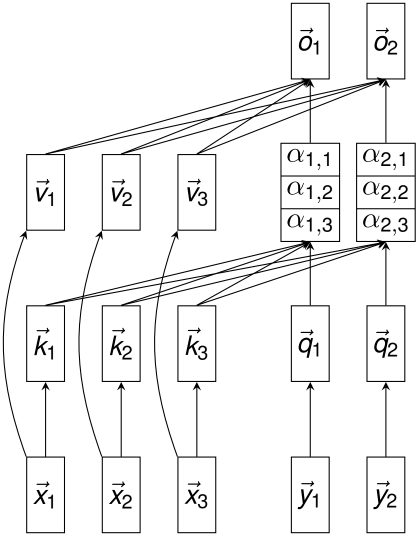
$$e_{j,j'} = a(\vec{q}_j, \vec{k}_{j'}) = \frac{\vec{q}_j^T \vec{k}_{j'}}{\sqrt{d_k}}$$

- Note: This requires that $d_q = d_k$
- Attention weights and outputs are defined like before (steps 2 and 3 of the basic recipe):

$$\alpha_{j,j'} = \frac{\exp(e_{j,j'})}{\sum_{j''=1}^{J_x} \exp(e_{j,j''})}$$

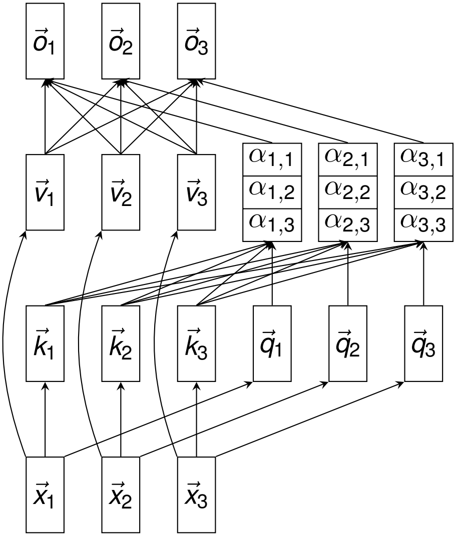
$$\vec{o}_j = \sum_{j'=1}^{J_x} \alpha_{j,j'} \vec{v}_{j'}$$

CROSS-ATTENTION (3)



Cross-attention

CROSS-ATTENTION (4)



Self-attention

PARALLELIZED ATTENTION (1)

- We want to apply our attention recipe to every query vector $\vec{q}_j$
- We could simply loop over all time steps $1 \leq j \leq J_y$ and calculate each $\vec{o}_j$ independently.
- Then stack all $\vec{o}_j$ into an output matrix $\vec{O} \in \mathbb{R}^{J_y \times d_v}$
- But a loop does not use the GPU's capacity for parallelization
- So it might be unnecessarily slow

PARALLELIZED ATTENTION (2)

- Do some inputs (e.g., $\vec{q}_j$) depend on previous outputs (e.g., $\vec{o}_{j-1}$)? If not, we can parallelize the loop into a single function:

$$\vec{O} = \mathcal{F}^{\text{attn}}(\vec{X}, \vec{Y}; \theta)$$

- Attention in Transformers is usually parallelizable, unless we are doing autoregressive inference (more on that later).
- By the way: The Bahdanau model is not parallelizable in this way, because s_i (a.k.a. the query of the $i+1$'st step) depends on c_i (a.k.a. the attention output of the i'th step), see last lecture:

The hidden state s_i of the decoder given the annotations from the encoder is computed by

$$s_i = (1 - z_i) \circ s_{i-1} + z_i \circ \tilde{s}_i,$$

where

$$\begin{aligned}
\tilde{s}_i &= \tanh\left(WEy_{i-1} + U\left[r_i \circ s_{i-1}\right] + Cc_i\right) \\
z_i &= \sigma\left(W_z Ey_{i-1} + U_z s_{i-1} + C_z c_i\right) \\
r_i &= \sigma\left(W_r Ey_{i-1} + U_r s_{i-1} + C_r c_i\right)
\end{aligned}$$

PARALLELIZED SCALED DOT PRODUCT ATTENTION (1)

- **Step 1**: The parallel application of the scaled dot product to all query-key pairs can be written as:

$$\vec{E} = \frac{\vec{Q}\vec{K}^T}{\sqrt{d_k}}; \quad \vec{E} \in \mathbb{R}^{J_y \times J_x}$$

$$\begin{array}{cc} & \rightarrow keys \rightarrow \\ \begin{array}{c}\downarrow \\ queries \\ \downarrow\end{array} & \begin{bmatrix} e_{1,1} & \dots & e_{1,J_x} \\ \vdots & \ddots & \vdots \\ e_{J_y,1} & \dots & e_{J_y,J_x} \end{bmatrix} \end{array} = \frac{1}{\sqrt{d_k}} \begin{bmatrix} - & \vec{q}_1 & - \\ & \vdots & \\ - & \vec{q}_{J_y} & - \end{bmatrix} \begin{bmatrix} | & & | \\ \vec{k}_1 & \dots & \vec{k}_{J_x} \\ | & & | \end{bmatrix}$$

PARALLELIZED SCALED DOT PRODUCT ATTENTION (2)

- **Step 2**: Softmax with normalization over the second axis (key axis):

$$\alpha_{j,j'} = \frac{\exp(e_{j,j'})}{\sum_{j''=1}^{J_x} \exp(e_{j,j''})}$$

```
>>> A = np.exp(E) / np.exp(E).sum(axis=-1, keepdims=Tru
```

- Let's call this new normalized matrix $\vec{A} \in (0,1)^{J_y \times J_x}$
- The rows of $\vec{A}$, denoted $\vec{\alpha}_j$, are probability distributions (one $\vec{\alpha}_j$ per $\vec{q}_j$)

PARALLELIZED SCALED DOT PRODUCT ATTENTION (3)

- **Step 3**: Weighted sum

$$\vec{O} = \vec{A}\vec{V};\, \vec{O} \in \mathbb{R}^{J_y \times d_v}$$

$$\begin{array}{cc} & \rightarrow d_v(\text{value dims})\rightarrow \\ \begin{array}{c}\downarrow \\ \textit{queries} \\ \downarrow\end{array} & \begin{bmatrix} o_{1,1} & \dots & o_{1,d_v} \\ \vdots & \ddots & \vdots \\ o_{J_y,1} & \dots & o_{J_y,d_v} \end{bmatrix} \end{array} = \begin{bmatrix} - & \boldsymbol{\alpha}_1 & - \\ & \vdots & \\ - & \boldsymbol{\alpha}_{J_y} & - \end{bmatrix} \begin{bmatrix} | & & | \\ \vec{v}_{:,1} & \dots & \vec{v}_{:,d_v} \\ | & & | \end{bmatrix}$$

... AS A ONE-LINER

$$\vec{O} = \mathcal{F}^{\mathrm{attn}}(\vec{X}, \vec{Y}; \theta) = \mathrm{softmax}\Big(\frac{(\vec{Y}\vec{W}^{(q)})(\vec{X}\vec{W}^{(k)})^T}{\sqrt{d_k}}\Big)(\vec{X}\vec{W}^{(v)})$$

- GPUs like matrix multiplications $\rightarrow$ usually a lot faster than RNN!
- But: The memory requirements of $\vec{E}$ and $\vec{A}$ are $\mathcal{O}(J_y J_x)$
- A length up to about 500 is usually ok on a medium-sized GPU (and most sentences are shorter than that anyway).
- But when we consider inputs that span several sentences (e.g., paragraphs or whole documents), we need tricks to reduce memory. These are beyond the scope of this lecture.