Machine Learning (CE 40477)

Fall 2024

Ali Sharifi-Zarchi

CE Department
Sharif University of Technology

December 8, 2024

1. Contextualized Word Embeddings
2. Recurrent Neural Networks
3. Attention Mechanism
4. Types of Attention
5. Positional Encoding
6. Multi-Head Attention
7. References

Attention is All You Need!

Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com
Noam Shazeer* Google Brain noam@google.com
Niki Parmar* Google Research nikip@google.com
Jakob Uszkoreit* Google Research usz@google.com
Llion Jones* Google Research llion@google.com
Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu
Łukasz Kaiser* Google Brain lukaszkaiser@google.com
Illia Polosukhin* ‡ illia.polosukhin@gmail.com

Abstract

The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. Experiments on two machine translation tasks show these models to be superior in quality while being more parallelizable and requiring significantly less time to train. Our model achieves 28.4 BLEU on the WMT 2014 English-to-German translation task, improving over the existing best results, including ensembles, by over 2 BLEU. On the WMT 2014 English-to-French translation task, our model establishes a new single-model state-of-the-art BLEU score of 41.0 after training for 3.5 days on eight GPUs, a small fraction of the training costs of the best models from the literature.

https://arxiv.org/abs/1706.03762

Attention: A Game-Changer in Natural Language Processing

- Imagine trying to read a book while your attention is scattered. You'd miss crucial details.
- Attention mechanism solve this problem in Neural Networks.
- This mechanism allows model to **"attend"** the most relevant information from the entire sequence.

1 Contextualized Word Embeddings

2 Recurrent Neural Networks

3 Attention Mechanism

4 Types of Attention

5 Positional Encoding

6 Multi-Head Attention

7 References

Limitations of Word2vec

- One vector for each word type
- Word2vec has challenges in polysemous words, e.g. `Light`
- Words don't appear in isolation. The word use (e.g., syntax and semantics) depends on its context.
- **Why not learn the representations for each word in its context?**

Contextualized Word Embeddings

بعد از یک روز طولانی در باغ‌وحش، وقتی به خانه برگشتم، یک لیوان شیر خوردم و شیر آبی که چکه می‌کرد را بستم؛ اما صدای غرش شیر قفس بغلی هنوز در گوشم می‌پیچید.

Figure is generated by Flux.1

Contextualized Word Embeddings

Compute contextual vector:

$$\mathbf{c_k} = f(w_k \mid w_1, w_2, \ldots, w_n) \in \mathbb{R}^d$$

Examples:

$f(\textit{light} \mid \text{Please turn off the } \overbrace{\textit{light}}^{} \text{ .})$

$\neq$

$f(\textit{light} \mid \text{This box is very } \underbrace{\textit{light}}_{} \text{ to carry.})$

How do we implement the context function f?

1 Contextualized Word Embeddings

2 Recurrent Neural Networks

3 Attention Mechanism

4 Types of Attention

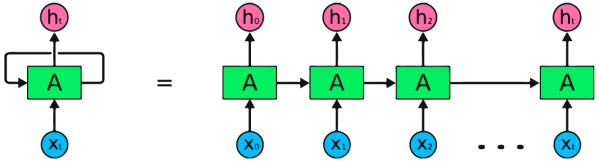
5 Positional Encoding

6 Multi-Head Attention

7 References

Recurrent Neural Networks

- To process each word, we need to remember the previous words.
- How do we create this memory? By maintaining a **hidden state**.
- Each hidden state captures information about:
  - Current word
  - All previous words in the sequence



where h_t serves as contextualized representation, containing memory of previous words.

RNN's Hidden State Update

- Process sequences by maintaining a hidden state.
- Update state sequentially: $h_t = f_W(h_{t-1}, x_t)$
- This recurrence formula is applied at each time step to process a sequence of vectors x.
- The same function and the same set of parameters are used for each word.

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

new state

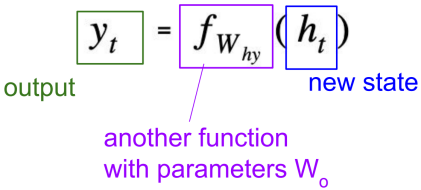
old state

input vector at some time step

some function with parameters W

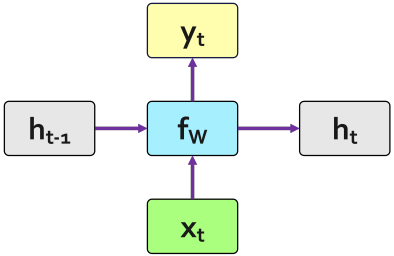
RNN Output Generation

- After updating the hidden state, RNN generates outputs at each time step.
- The output y_t is typically computed as a function of the current hidden state h_t, often passed through a layer (like a softmax layer) for classification or regression tasks.



(Vanilla) Recurrent Neural Network

- The state of the RNN consists of a single hidden vector h_t, which updates as new inputs are processed.



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

$$\downarrow$$

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

$$y_t = W_{hy}h_t$$

Basic Numerical Example of RNN - Hidden State Update (Part 1)

- Consider a simple RNN with the following parameters:
  - Input $x_t = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$
  - Previous hidden state $h_{t-1} = \begin{bmatrix} 0.5 \\ -0.5 \end{bmatrix}$
  - Weight matrices:
$$W_{hh} = \begin{bmatrix} 0.1 & 0.2 \\ -0.1 & 0.3 \end{bmatrix}, W_{xh} = \begin{bmatrix} 0.4 & 0.5 \\ 0.6 & 0.7 \end{bmatrix}$$
  - Bias vector $b = \begin{bmatrix} 0.1 \\ 0.2 \end{bmatrix}$
- The RNN updates its hidden state using the equation:
$$h_t = W_{hh} h_{t-1} + W_{xh} x_t + b$$
$$h_t = \begin{bmatrix} 0.1 & 0.2 \\ -0.1 & 0.3 \end{bmatrix} \begin{bmatrix} 0.5 \\ -0.5 \end{bmatrix} + \begin{bmatrix} 0.4 & 0.5 \\ 0.6 & 0.7 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \begin{bmatrix} 0.1 \\ 0.2 \end{bmatrix}$$

Basic Numerical Example of RNN - Hidden State Update (Part 2)

- Continuing from the previous slide, the calculation after adding the bias term is:

$$h_t = \begin{bmatrix} 0.05 - 0.1 \\ -0.05 - 0.15 \end{bmatrix} + \begin{bmatrix} 0.4 \\ 0.6 \end{bmatrix} + \begin{bmatrix} 0.1 \\ 0.2 \end{bmatrix}$$

$$h_t = \begin{bmatrix} 0.45 \\ 0.6 \end{bmatrix}$$

The hidden state is then passed through the activation function (tanh):

$$h_t = f\left(\begin{bmatrix} 0.45 \\ 0.6 \end{bmatrix}\right)$$

$$h_t = \begin{bmatrix} \tanh(0.45) \\ \tanh(0.6) \end{bmatrix}$$

Basic Numerical Example of RNN - Output Computation

- Continuing from the previous slide, the updated hidden state is:

$$h_t = f\left(\begin{bmatrix} 0.45 \\ 0.6 \end{bmatrix}\right)$$

- The output y_t is computed as:

$$y_t = W_{hy} h_t$$

$$y_t = \begin{bmatrix} 0.8 & 0.9 \end{bmatrix} \begin{bmatrix} \tanh(0.45) \\ \tanh(0.6) \end{bmatrix}$$

$$y_t = \begin{bmatrix} 0.8 \cdot 0.4228 + 0.9 \cdot 0.5370 \end{bmatrix}$$

$$y_t = \begin{bmatrix} 0.3382 + 0.4833 \end{bmatrix}$$

$$y_t = \begin{bmatrix} 0.8215 \end{bmatrix}$$

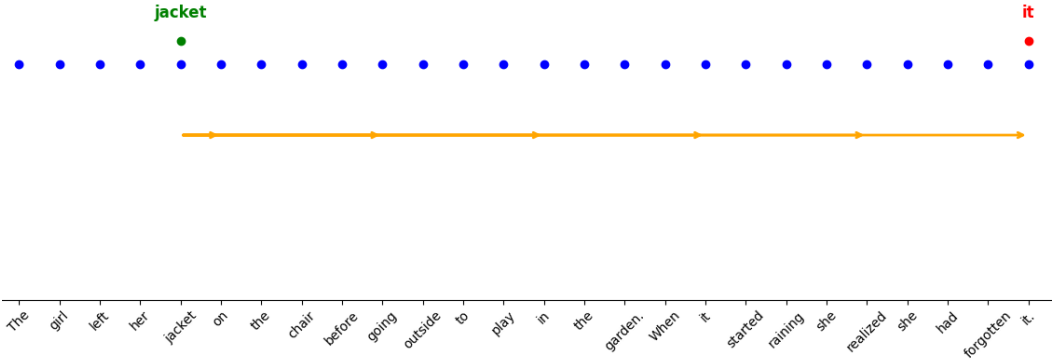
Challenges of RNNs

While improved versions of vanilla RNNs attempt to resolve some issues, in many cases they still struggle with:

- Long-term dependencies
- Vanishing/exploding gradients
- Sequential computation (can't parallelize)

Challenges of RNNs

RNNs face challenges in retaining information over long sequences, such as when pronouns refer to distant words in the sequence.



So, what solution do we have?

1 Contextualized Word Embeddings

2 Recurrent Neural Networks

3 Attention Mechanism

4 Types of Attention

5 Positional Encoding

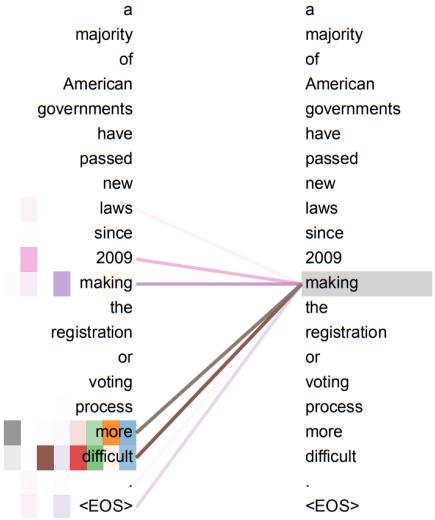
6 Multi-Head Attention

7 References

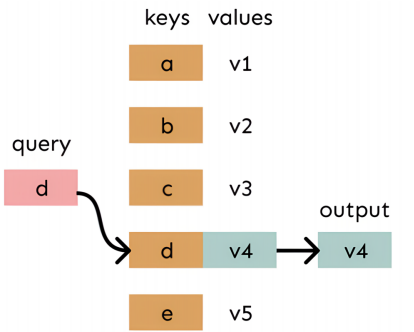
Why Attention?

- Enhances the model's ability to focus on relevant parts of the input.
- Can compute relationships between inputs regardless of their position.
- Inspired by human visual attention.
- **Key Idea**
  - Let the model learn which parts of the input are important for each output.

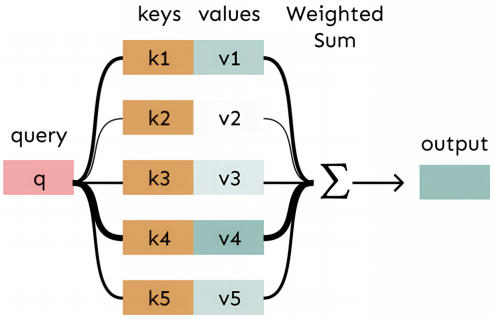
Why Attention?



Attention as a Soft, Averaging Lookup Table



In a lookup table, we have a table of keys that map to values. The query matches one of the keys, returning its value.

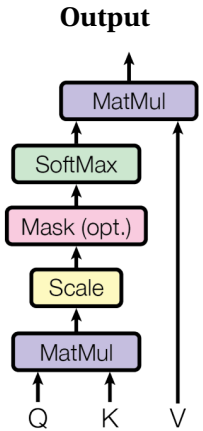


In the attention, the query matches all keys softly, to a weight between 0 and 1. The keys' values are multiplied and summed by the weights.

Attention Mechanism

- An attention function maps a query and a set of key-value pairs to an output.
- The query, keys, values, and the output are represented as vectors.
- Components of the Attention Mechanism are:
  - Query (Q): What we're looking for
  - Key (K): What we match against
  - Value (V): What we retrieve
- The most common attention function is **Scaled Dot Product Attention**, which will be described in the following slides.

Scaled Dot-Product Attention



$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Scaled Dot-Product Attention

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

- The query and keys are in dimension d_k.
- Dot products of the query with all keys are computed, divided by $\sqrt{d_k}$
- The softmax function is applied to convert scores to probabilities (get attention distribution).

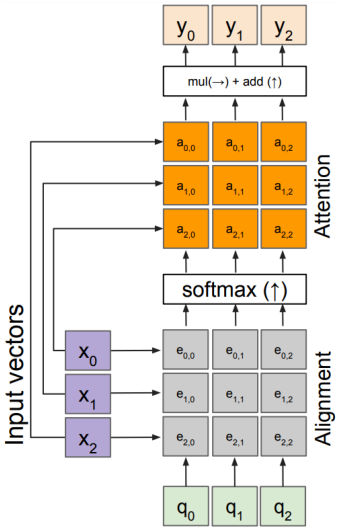
Scaling Factor in Attention (Part 1)

- The attention function computes the dot product between the query and key.
- For **large** d_k, the dot products can become very large, which **pushes the softmax function** into regions where it has **extremely small gradients**, making the model less effective at learning.
- To address this, the dot product is scaled by $\frac{1}{\sqrt{d_k}}$, ensuring softmax operates in a stable range.

Scaling Factor in Attention (Part 2)

- Scaling improves training stability, especially for large d_k values.
- This adjustment avoids softmax saturation and enhances model performance.

General Attention Layer



Outputs:

Context vectors: $\mathbf{y}$ (*shape:* D_v)

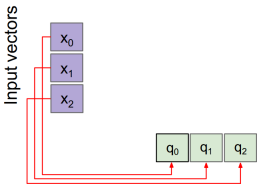
Operations:

- Key vectors: $\mathbf{k} = W_k^T \mathbf{x}$
- Value vectors: $\mathbf{v} = W_v^T \mathbf{x}$
- **Alignment**: $e_{i,j} = \frac{\mathbf{q}_i \cdot \mathbf{k}_i}{\sqrt{D_k}}$
- **Attention**: $\mathbf{a} = \texttt{softmax}(\mathbf{e})$
- Output: $y_j = \sum_i a_{i,j} \mathbf{v}_i$

Inputs:

- Input vectors: $\mathbf{x}$ (*shape:* $N \times D$)
- Queries: $\mathbf{q}$ (*shape:* $M \times D_k$)

Self-Attention Layer



Operations:

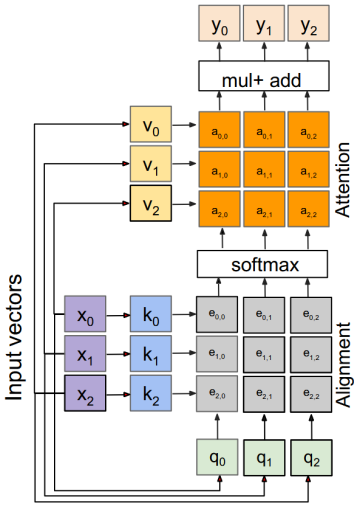
- Key vectors: $\mathbf{k} = W_k^T \mathbf{x}$
- Value vectors: $\mathbf{v} = W_v^T \mathbf{x}$
- Query vectors: $\mathbf{q} = W_q^T \mathbf{x}$
- **Alignment**: $e_{i,j} = \frac{\mathbf{q}_j \cdot \mathbf{k}_i}{\sqrt{D_k}}$
- **Attention**: $\mathbf{a} = \texttt{softmax}(\mathbf{e})$
- Output: $y_j = \sum_i a_{i,j} \mathbf{v}_i$

Inputs:

- Input vectors: $\mathbf{x}$ (*shape:* $N \times D$)
- ~~Queries: $\mathbf{q}$ (*shape:* $M \times D_k$)~~

- We can calculate the query vectors from the input vectors, therefore defining a **"self-attention"** layer.
- Instead, query vectors are calculated using a Fully Connected (FC) layer.
- **No input query vectors anymore.**

Self-Attention Layer



Outputs:

Context vectors: $\mathbf{y}$ (*shape:* D_v)

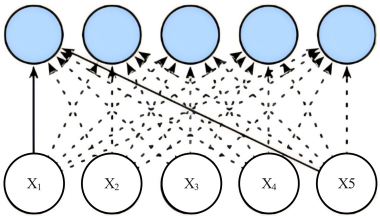
Operations:

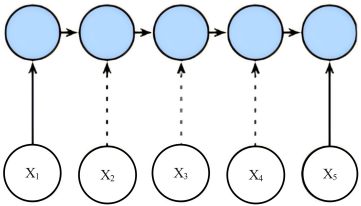
- Key vectors: $\mathbf{k} = W_k^T \mathbf{x}$
- Value vectors: $\mathbf{v} = W_v^T \mathbf{x}$
- Query vectors: $\mathbf{q} = W_q^T \mathbf{x}$
- **Alignment**: $e_{i,j} = \frac{\mathbf{q}_j \cdot \mathbf{k}_i}{\sqrt{D_k}}$
- **Attention**: $\mathbf{a} = \texttt{softmax}(\mathbf{e})$
- Output: $y_j = \sum_i a_{i,j} \mathbf{v}_i$

Inputs:

- Input vectors: $\mathbf{x}$ (*shape:* $N \times D$)

Self-Attention vs RNN





Self-Attention:

- Directly accesses key information at any position in the input sequence.
- Allows the model to **"attend"** to the most relevant information in the sequence.

RNN:

- Processes input sequentially, making distant information harder to access.
- Relies on the hidden state at the current time step to propagate information.

Self-Attention: Core Theorem

Self-Attention Properties

Self-attention layers can model dependencies between all elements in an input sequence in parallel, capturing long-range relationships efficiently.

- Global connectivity
- Parallel computation
- Position-independent weighting
- $O(n^2)$ complexity for sequence length n

Proof Component 1: Parallel Processing

- Matrix multiplication enables parallel computation:

$$S = QK^T = \begin{bmatrix} (q_1 k_1^T) & \cdots & (q_1 k_n^T) \\ \vdots & \ddots & \vdots \\ (q_n k_1^T) & \cdots & (q_n k_n^T) \end{bmatrix}$$

- All n^2 interactions computed simultaneously
- No sequential dependencies between computations

Key Insight

Unlike RNNs, there is no need to wait for previous timesteps

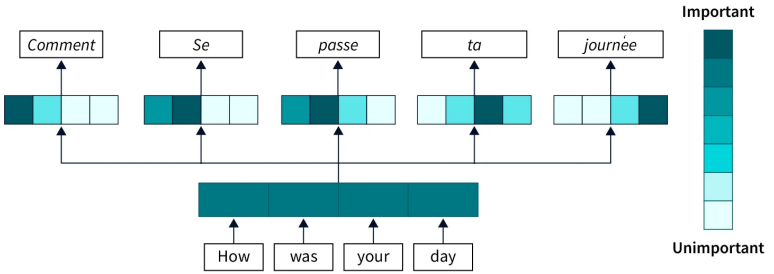
Proof Component 2: Global Dependencies

- Attention weights between positions i and j:

$$\alpha_{ij} = \frac{\exp(q_i^T k_j / \sqrt{d_k})}{\sum_{l=1}^{n} \exp(q_i^T k_l / \sqrt{d_k})}$$

  - Properties:
  - α_{ij} depends only on compatibility of i and j
  - No distance-based attenuation
  - Softmax ensures $\sum_j \alpha_{ij} = 1$

Proof Component 2: Global Dependencies



Proof Component 3: Computational Efficiency

Complexity Analysis

- Matrix multiply: $O(n^2 d)$
- Softmax: $O(n^2)$
- Total: $O(n^2 d)$

Optimizations

- Sparse attention
- Linear attention
- Sliding window

Memory-Computation Tradeoff

More memory than RNNs ($O(n)$), but enables parallelization.

Implementation Details

- Practical considerations:
  - Scale factor $\sqrt{d_k}$ prevents vanishing gradients
  - Mask for causal attention (decoder)
  - Dropout for regularization

$$\text{MaskedAttention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}} + M\right)V$$

where $M_{ij} = -\infty$ for masked positions

Formal Proof Conclusion

Theorem Verification

Self-attention satisfies all claimed properties:

1. Parallel computation: Matrix operations
2. Global dependencies: Direct pairwise attention
3. Efficient computation: $O(n^2 d)$ complexity
4. Learnable patterns: Attention weights

Important Note

These properties make self-attention particularly suitable for:

- Natural language processing
- Graph neural networks
- Computer vision (with modifications)

1 Contextualized Word Embeddings

2 Recurrent Neural Networks

3 Attention Mechanism

4 Types of Attention

5 Positional Encoding

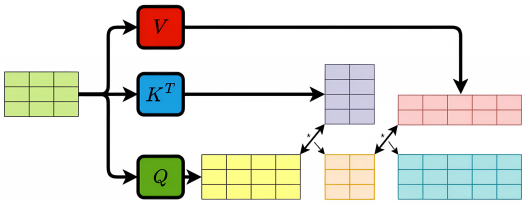
6 Multi-Head Attention

7 References

Self-Attention vs Cross-Attention

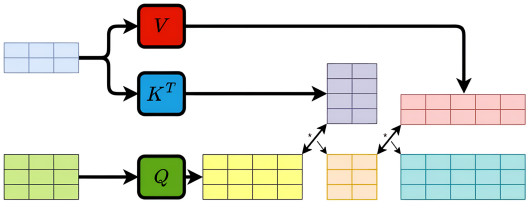
Self-Attention

- Queries, Keys, and Values are derived from the same sequence.
- Each position attends to all other positions in the sequence.



Cross-Attention

- Queries come from one sequence.
- Keys and Values come from another sequence.



1 Contextualized Word Embeddings

2 Recurrent Neural Networks

3 Attention Mechanism

4 Types of Attention

5 Positional Encoding

6 Multi-Head Attention

7 References

Positional Encoding

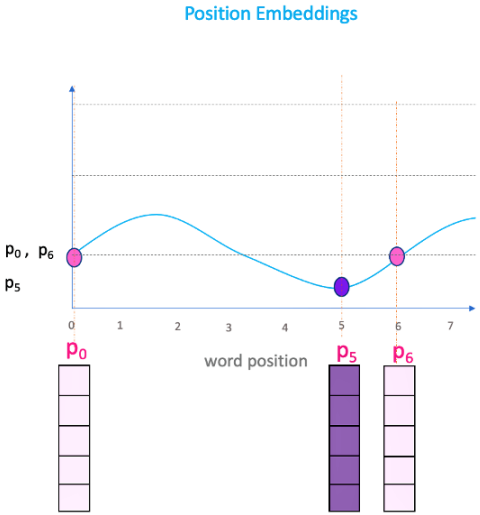
- Problem: Attention is position-agnostic
- Solution: Add position information to embeddings
- Using sinusoidal functions:

$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d_{model}})$$

$$PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d_{model}})$$

- Allows model to learn relative positions
- Works for sequences of any length

Positional Encoding



1 Contextualized Word Embeddings

2 Recurrent Neural Networks

3 Attention Mechanism

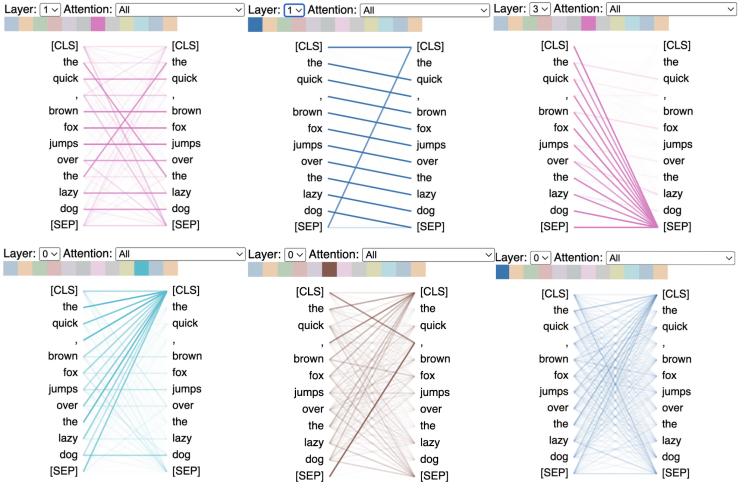
4 Types of Attention

5 Positional Encoding

6 Multi-Head Attention

7 References

Multi-Head Attention



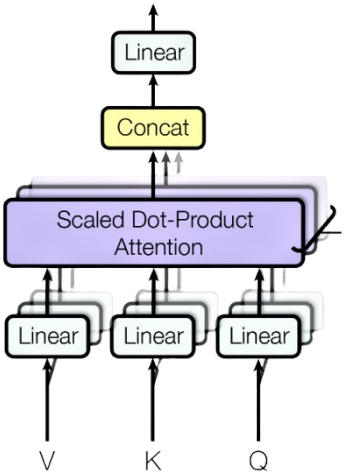
Multi-Head Attention

- Instead of single attention, use multiple heads
- Each head can focus on different aspects
- Process in parallel and concatenate

$$\text{MultiHead}(Q, K, V) = \text{Concat}(head_1, \ldots, head_h) W^O$$

$$head_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Multi-Head Attention



Multi-Head Attention Benefits

- Multiple representation subspaces
- Can capture different types of relationships:
  - Syntactic dependencies
  - Semantic relationships
  - Long-range dependencies
- Improves model capacity and stability

Contributions

These slides are authored by:

- Faezeh Sarlakifar

1 Contextualized Word Embeddings

2 Recurrent Neural Networks

3 Attention Mechanism

4 Types of Attention

5 Positional Encoding

6 Multi-Head Attention

7 References

[1] F.-F. Li and Z. Durante, *CS231n Lectures*.
Stanford University, June 2024.
Updated June 3, 2024.

[2] C. Manning, *CS224n Lectures*.
Stanford University, June 2024.

[3] M. Soleymani Baghshah, "Deep learning." Lecture slides.

Any Questions?