A study of transformer and attention mechanics in the industry of artificial intelligence

Anna Liang

October 2024

1 Introduction

1.1 The overview background of the development of artificial intelligence

The early stages of the deep learning renaissance were driven largely by breakthroughs achieved with classical architectures such as the multilayer perceptron (MLP), convolutional neural networks (CNNs), and recurrent neural networks (RNNs). Interestingly, the core designs of these models, which facilitated many of the key advances in the 2010s, remained fundamentally similar to their original forms from nearly three decades prior. While the introduction of new techniques, including ReLU activations [37], residual connections [19], batch normalization [22], dropout [44], and adaptive learning rate schedules like Adam [27], enriched the deep learning toolbox, the underlying architectures were still recognizable as scaled-up implementations of earlier concepts.

Despite numerous research efforts proposing alternative architectures, models inspired by classical CNNs continued to dominate the field of computer vision. Likewise, in natural language processing (NLP), the long short-term memory (LSTM) architecture [20], originally proposed by Sepp Hochreiter, remained the de facto standard for most sequence-based tasks. During this period, the rapid rise of deep learning could be largely attributed to two key factors: the significant advances in computational power due to innovations in parallel computing with GPUs, and the availability of large-scale datasets enabled by cheap data storage and widespread internet access. However, alongside these advancements, we have recently witnessed a profound shift in the dominant architectures being used in both research and industry.

Currently, Transformer-based architectures have emerged as the cornerstone for a majority of NLP tasks. The standard practice for handling new tasks in this domain now involves leveraging large pretrained models, such as BERT [11], ELECTRA [10], RoBERTa [32], or Longformer [3]. These models are fine-tuned for specific tasks by adjusting the output layers to suit the new data and objectives. With increasing attention from the research community and media, OpenAI's large-scale models, including GPT-2 [40] and GPT-3 [6], have underscored the transformative impact of Transformer-based approaches.

In the realm of computer vision, the Vision Transformer (ViT) has quickly established itself as a top-performing architecture for tasks such as image recognition, object detection, semantic segmentation, and superresolution [13, 14]. Beyond vision and NLP, Transformers have also demonstrated competitive performance in fields such as speech recognition [12], reinforcement learning [38], and graph neural networks [50].

The attention mechanism, which lies at the heart of these Transformer models, has become a critical element in modern neural network architectures. This paper categorizes the development of attention mechanisms into distinct phases, exploring major breakthroughs and their influence on subsequent research across various fields.

1.2 A Brief History of the Development of Attention Mechanisms

Attention mechanisms have become a fundamental component in modern machine learning, particularly for tasks involving sequential data. They operate by assigning relative importance to each element in a sequence, facilitating more efficient and targeted processing of information. In the domain of natural language processing (NLP), this importance is quantified through "soft" weights, dynamically assigned to each word within a sentence. These soft weights enable the encoding of token embeddings across sequences that can span from tens to millions of tokens.

Unlike fixed "hard" weights, which are optimized during the backward pass of training, soft weights are computed solely during the forward pass and vary with each input step. The introduction of attention initially occurred in recurrent neural network (RNN)-based models for language translation, where the sequential nature of RNNs limited efficiency. Subsequent advancements, particularly the transformer architecture, replaced the slower serial RNN structure with a parallelized attention mechanism, significantly improving computational speed and model performance.

Drawing inspiration from the cognitive processes of human attention, the attention mechanism was designed to overcome the inherent limitations of RNNs, which tend to prioritize more recent input tokens while attenuating earlier information in the sequence. By providing equal access to any part of the input sequence, attention mechanisms enable each token to directly reference any other token, bypassing the need for sequential state dependency [48].

1.2.1 Phase 1: Emergence of Attention Mechanisms (Especially from late 1980s to Early 2010s)

The origins of attention mechanisms can be traced back to early research in neuroscience, psychology, and artificial intelligence, first explored in the context of visual attention models, and with a focus on *selective attention.* In neuroscience and cognitive psychology, selective attention was extensively studied to understand how humans focus on specific stimuli while filtering out irrelevant information [8, 5, 43]. Additionally, research showed that eye movements, particularly saccades, are influenced by cognitive processes, allowing humans to quickly scan areas of high salience in the visual field [41].

These foundational ideas influenced early neural network models that aimed to integrate *visual attention mechanisms.* For example, Fukushima introduced *selective attention* into neural networks [15]. Subsequent advancements included the VISIT and SCAN models, which applied attention mechanisms to object detection tasks[1]. By the late 1990s, models such as Itti et al.'s saliency-based neural network were developed to simulate low-level primate vision and guide attention towards regions of high salience in images [23].

In the early 2010s, a key development was the introduction of the first image classification systems inspired by the human fovea, utilizing reinforcement learning for image classification and object tracking [30]. This system, based on the selective nature of the human eye, emphasized the importance of saccadic eye movement and foveal vision, where only a small part of the visual field can be sharply resolved at any given moment.

1.2.2 Phase 2: The Rise of Attention (2015)

The year 2015 marked a "golden age" for attention mechanisms with several landmark studies:

- A novel approach to neural machine translation was proposed, addressing limitations in the traditional encoder-decoder architecture by focusing on relevant parts of the input sentence [2].
- The first visual attention model for image captioning was introduced, utilizing deep convolutional neural networks (CNNs) as encoders to selectively attend to specific regions of the image. [49]
- Two types of attention mechanisms were introduced in neural machine translation: global attention, which attends to all source words, and local attention, which focuses on a subset of source words at a time. [33]

1.2.3 Phase 3: The Expansion of Attention (2015-2016)

During this period, attention mechanisms were applied to a wider range of tasks, and novel architectures emerged:

- Memory networks such as Neural Turing Machines and end-to-end memory networks, which employed recurrent attention mechanisms, were developed [16].
- Self-attention (also called intra-attention) was successfully implemented in Long Short-Term Memory Networks (LSTMs) by replacing memory units

with memory networks, leading to improved performance across various domains [7].

- Attention models were applied to tasks such as image captioning, sentence summarization, speech recognition, video captioning, neural machine translation, and textual entailment.

1.2.4 Phase 4: The Emergence of the Transformer (2017)

The introduction of the Transformer architecture in 2017 marked a significant turning point in the development of attention mechanisms: In 2017, [45] introduced the Transformer model, a groundbreaking architecture that relies entirely on self-attention mechanisms to process sequential data. The Transformer's efficiency and ability to handle long-range dependencies without recurrence or convolution have led to its widespread adoption in various NLP tasks.

Unlike recurrent neural networks (RNNs) and convolutional neural networks (CNNs), the Transformer does not require sequential processing, which enables faster training and better scalability. The success of this architecture has spawned numerous Transformer variants, such as:

- **BERT** [11], which pre-trains bidirectional Transformers for a wide range of NLP tasks.
- **RoBERTa** [32], which optimizes BERT's pretraining methodology.
- **Longformer** [3], designed for tasks involving long documents by utilizing sparse attention mechanisms.

1.2.5 Phase 5: Current Research Directions

Attention mechanisms continue to be an active area of research, with several key trends emerging:

- **New Variants of Attention Mechanisms**: Researchers have developed various new attention mechanisms, such as relation-aware self-attention, directional self-attention, and outer product attention, each designed to address specific limitations or enhance performance.
- **Expanding Transformer Applications**: Transformers are being applied to domains beyond NLP, including image generation, speech recognition, and video analysis. Notably, Vision Transformers have become a go-to model for image tasks [13], while Conformer [17] and Speech-Transformers [12] are adapted for speech recognition tasks. This broad applicability underscores the flexibility of attention mechanisms across a variety of domains.
- **Addressing Computational Complexity**: While the Transformer architecture has demonstrated remarkable success, its computational complexity remains a challenge. Several approaches have been proposed to

mitigate this issue, such as linear Transformers [26], Reformer [28], Set Transformer [31], and Sparse Transformers [9].

2 Attention Mechanisms in Neural Networks

Attention mechanisms in neural networks can be mathematically described as a weighted combination of inputs, where the weight is determined by a similarity function (often the dot product) between a query and a key. The basic attention operation is often expressed as:

$$\sum_i \langle (\text{query})_i, (\text{key})_i \rangle (\text{value})_i \tag{1}$$

where the angled brackets denote dot product. This shows that it involves a multiplicative operation. This operation indicates that attention involves *multiplicative computations*, which have roots in earlier neural network research. For instance, multiplicative units had been studied under various names, including Group Method of Data Handling (GMDH) [25, 24], where Kolmogorov-Gabor polynomials are used to implement multiplicative units or "gates" [29], higher-order neural networks [47], multiplication units [21], sigma-pi units [39], fast weight controllers [42], and hyper-networks [18]. These multiplicative operations laid the groundwork for attention mechanisms in modern neural networks.

2.1 Nadaraya–Watson kernel regression

In statistics, *kernel regression* is a non-parametric technique to estimate the conditional expectation of a random variable. The objective is to find a non-linear relation between a pair of random variables X and Y. In any nonparametric regression, the conditional expectation of a variable Y relative to a variable X may be written:

$$\mathbf{E}(Y|X) = m(X) \tag{2}$$

where m is an unknown function. Nadaraya and Watson, both in 1964, proposed to estimate m as a locally weighted average, using a kernel as a weighting function [36, 46, 4]. The Nadaraya–Watson estimator is:

$$f(x) = \widehat{m}_h(x) = \frac{\sum_{i=1}^n K_h(x - x_i) y_i}{\sum_{i=1}^n K_h(x - x_i)} \tag{3}$$

where

$$K_h(t) = \frac{1}{h} K\left(\frac{t}{h}\right)$$

is a kernel with a bandwidth h, such that $K(\cdot)$ is of order at least 1, that is

$$\int_{-\infty}^{\infty} u K(u)\, du = 0 \tag{4}$$

Derivation. Starting with the definition of conditional expectation:

$$\mathrm{E}(Y \mid X = x) = \int y f(y \mid x)\, dy = \int y \frac{f(x,y)}{f(x)}\, dy$$

we estimate the joint distributions $f(x,y)$ and $f(x)$ using kernel density estimation with a kernel K:

$$\hat{f}(x,y) = \frac{1}{n} \sum_{i=1}^{n} K_h(x - x_i) K_h(y - y_i),$$

$$\hat{f}(x) = \frac{1}{n} \sum_{i=1}^{n} K_h(x - x_i),$$

We then get:

$$\hat{\mathrm{E}}(Y \mid X = x) = \int y \frac{\hat{f}(x,y)}{\hat{f}(x)}\, dy$$

$$= \int y \frac{\sum_{i=1}^{n} K_h(x - x_i) K_h(y - y_i)}{\sum_{j=1}^{n} K_h(x - x_j)}\, dy$$

$$= \frac{\sum_{i=1}^{n} K_h(x - x_i) \int y K_h(y - y_i)\, dy}{\sum_{j=1}^{n} K_h(x - x_j)}$$

$$= \frac{\sum_{i=1}^{n} K_h(x - x_i) y_i}{\sum_{j=1}^{n} K_h(x - x_j)},$$

which is the Nadaraya–Watson estimator.

For now, let's consider the following: denote by $\mathcal{D} \stackrel{\text{def}}{=} \{(\mathbf{k}_1, \mathbf{v}_1), \ldots (\mathbf{k}_m, \mathbf{v}_m)\}$ a database of m tuples of *keys* and *values*. Moreover, denote by q a query. Then we can define the attention over $\mathcal{D}$ as

$$\mathrm{Attention}(\mathbf{q}, \mathcal{D}) \stackrel{\text{def}}{=} \sum_{i=1}^{m} \alpha(\mathbf{q}, \mathbf{k}_i) \mathbf{v}_i, \tag{5}$$

where $\alpha(\mathbf{q}, \mathbf{k}_i) \in \mathbb{R}$ are scalar attention weights. The operation itself is typically referred to as attention pooling. The name attention derives from the fact that the operation pays particular attention to the terms for which the weight α is *significant* (i.e., large). As such, the attention over $\mathcal{D}$ generates a linear combination of values contained in the database. In fact, this contains the above example as a special case where all but one weight is zero. We have a number of special cases:

- The weights $\alpha(\mathbf{q}, \mathbf{k}_i)$ are nonnegative. In this case the output of the attention mechanism is contained in the convex cone spanned by the values v_i.

- The weights $\alpha(\mathbf{q}, \mathbf{k}_i)$ form a convex combination, i.e., $\sum_i \alpha(\mathbf{q}, \mathbf{k}_i) = \mathbf{1}$ and $\alpha(\mathbf{q}, \mathbf{k}_i) \geq 0$ for all i . This is the most common setting in deep learning.

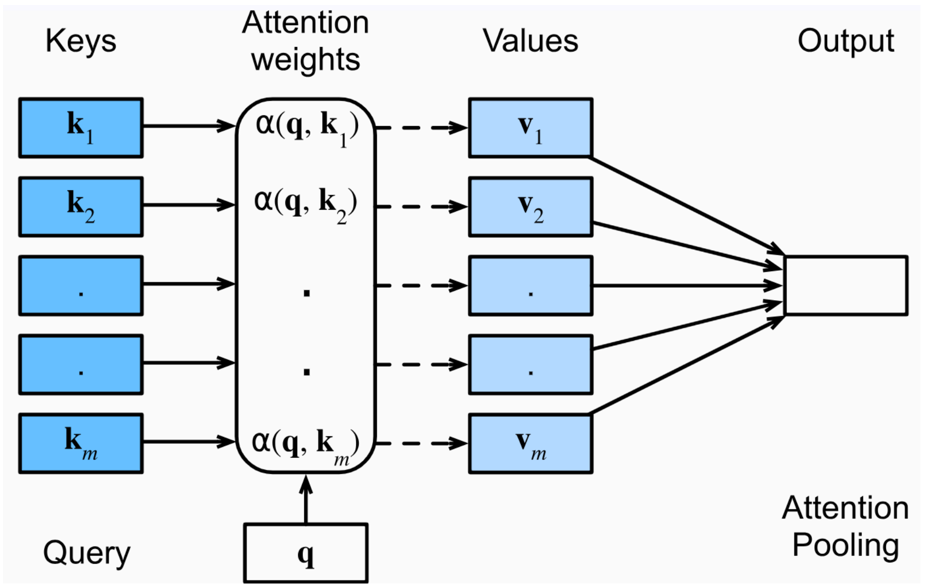

Figure 1: The attention mechanism computes a linear combination over values v_i via attention pooling, where weights are derived according to the compatibility between a query q_i and keys k_i.

- Exactly one of the weights $\alpha(\mathbf{q}, \mathbf{k}_i)$ is $\mathbf{1}$, while all others are $\mathbf{0}$. This is akin to a traditional database query.
- All weights are equal, i.e., $\alpha(\mathbf{q}, \mathbf{k}_i) = \frac{1}{m}$ for all i. This amounts to averaging across the entire database, also called average pooling in deep learning.

A common strategy for ensuring that the weights sum up to $\mathbf{1}$ is to normalize them via

$$\alpha(\mathbf{q}, \mathbf{k}_i) = \frac{\alpha(\mathbf{q}, \mathbf{k}_i)}{\sum_j \alpha(\mathbf{q}, \mathbf{k}_j)}. \tag{6}$$

In particular, to ensure that the weights are also nonnegative, one can resort to exponentiation. This means that we can now pick any function $a(\mathbf{q}, \mathbf{k})$ and then apply the softmax operation used for multinomial models to it via

$$\alpha(\mathbf{q}, \mathbf{k}_i) = \frac{\exp(a(\mathbf{q}, \mathbf{k}_i))}{\sum_j \exp(a(\mathbf{q}, \mathbf{k}_j))}. \tag{7}$$

This operation is readily available in all deep learning frameworks. It is differentiable and its gradient never vanishes, all of which are desirable properties in a model.

Note though, the attention mechanism introduced above is not the only option. For instance, we can design a *non-differentiable* attention model that can be trained using reinforcement learning methods [35]. As one would expect, training such a model is quite complex. Consequently the bulk of modern attention research follows the framework outlined in Figure 1. We thus focus our exposition on this family of differentiable mechanisms.

What is quite remarkable is that the actual "code" for executing on the set of keys and values, namely the query, can be quite concise, even though the space to operate on is significant. This is a desirable property for a network layer as it does not require too many parameters to learn. Just as convenient is the fact that attention can operate on arbitrarily large databases without the need to change the way the attention pooling operation is performed.

2.2 Attention Pooling by Similarity

Now that we have introduced the primary components of the attention mechanism, let's use them in a rather classical setting, namely *regression* and *classification* via *kernel density estimation*[36, 46]. This detour simply provides additional background: it is entirely optional and can be skipped if needed. At their core, Nadaraya–Watson estimators rely on some similarity kernel $\alpha(\mathbf{q}, \mathbf{k})$ relating queries q to keys k. Some common kernels are

$$\begin{aligned}
&\alpha(\mathbf{q}, \mathbf{k}) = \exp\left(-\frac{1}{2}\|\mathbf{q} - \mathbf{k}\|^2\right) && \text{Gaussian;} \\
&\alpha(\mathbf{q}, \mathbf{k}) = 1 \text{ if } \|\mathbf{q} - \mathbf{k}\| \leq 1 && \text{Boxcar;} \\
&\alpha(\mathbf{q}, \mathbf{k}) = \max\left(0, 1 - \|\mathbf{q} - \mathbf{k}\|\right) && \text{Epanechikov.}
\end{aligned} \tag{8}$$

All of the kernels are heuristic and can be tuned. For instance, we can adjust the width, not only on a global basis but even on a per-coordinate basis. Regardless, all of them lead to the following equation for regression and classification alike:

$$f(\mathbf{q}) = \sum_i \mathbf{v}_i \frac{\alpha(\mathbf{q}, \mathbf{k}_i)}{\sum_j \alpha(\mathbf{q}, \mathbf{k}_j)}. \tag{9}$$

In the case of a (scalar) regression with observations $(\mathbf{x}_i, y_i)$ for features and labels respectively, $\mathbf{v}_i = y_i$ are scalars, $\mathbf{k}_i = \mathbf{x}_i$ are vectors, and the query $\mathbf{q}$ denotes the new location where $\mathbf{f}$ should be evaluated.

In the case of (multiclass) classification, we use one-hot-encoding of y_i to obtain $\mathbf{v}_i$. One of the convenient properties of this estimator is that it requires no training. Even more so, if we suitably narrow the kernel with increasing amounts of data, the approach is consistent [34], i.e., it will converge to some statistically optimal solution.

2.3 Attention Pooling: Nadaraya-Watson Kernel Regression

To recapitulate, the interactions between queries q and keys k result in attention pooling. The attention pooling selectively aggregates values v to produce the output. Specifically, the Nadaraya-Watson kernel regression model proposed in 1964 is a simple yet complete example for demonstrating machine learning with attention mechanisms. We begin with perhaps the world's "dumbest" estimator for this regression problem: using average pooling to average over all the training outputs:

$$f(x) = \frac{1}{n}\sum_{i=1}^{n} y_i, \tag{10}$$

Obviously, average pooling omits the inputs x_i. Recall a better idea proposed by Nadaraya[36] and Watson[46]. From Equation 3, the Nadaraya–Watson estimator is:

$$f(x) = \widehat{m}_h(x) = \frac{\sum_{i=1}^{n} K_h(x - x_i) y_i}{\sum_{i=1}^{n} K_h(x - x_i)} \tag{11}$$

where K is a *kernel.* From the perspective of attention, we can rewrite the estimator in a more generalized form of attention pooling:

$$f(x) = \sum_{i=1}^{n} \alpha(x, x_i) y_i, \tag{12}$$

where x is the query and (x_i, y_i) is the key-value pair. Comparing Equation 11 and 12, the attention pooling here is a weighted average of values y_i. The attention weight $\alpha(x, x_i)$ in Equation 12 is assigned to the corresponding value y_i based on the interaction between the query x and the key x_i modeled by α. For any query, its attention weights over all the key-value pairs are a valid probability distribution: they are *non-negative* and sum up to **1**.

To gain intuitions of attention pooling, just consider a Gaussian kernel defined as

$$K(u) = \frac{1}{\sqrt{2\pi}} \exp(-\frac{u^2}{2}). \tag{13}$$

Plugging the Gaussian kernel into Equation 11 and Equation 12 gives

$$\begin{aligned} f(x) &= \sum_{i=1}^{n} \alpha(x, x_i) y_i \\ &= \sum_{i=1}^{n} \frac{\exp\left(-\frac{1}{2}(x - x_i)^2\right)}{\sum_{j=1}^{n} \exp\left(-\frac{1}{2}(x - x_j)^2\right)} y_i \\ &= \sum_{i=1}^{n} \operatorname{softmax}\left(-\frac{1}{2}(x - x_i)^2\right) y_i. \end{aligned} \tag{14}$$

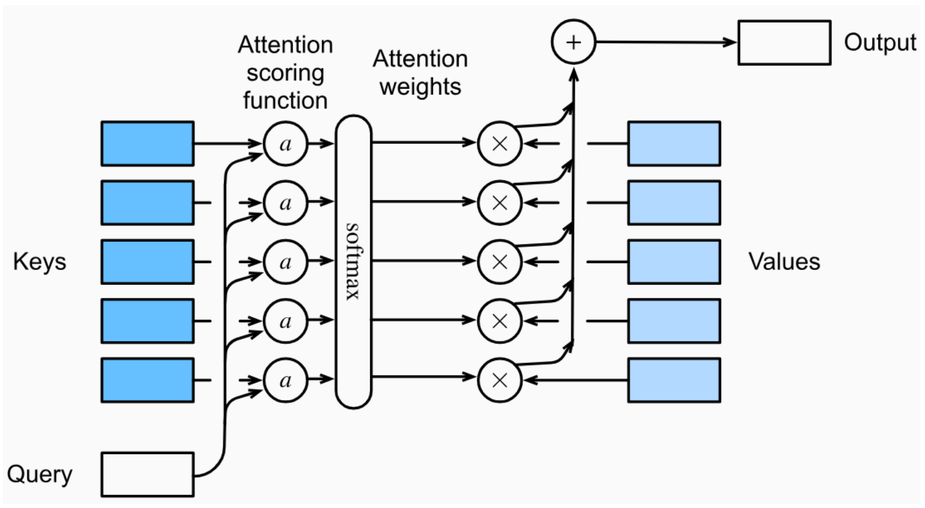

Figure 2: Computing the output of attention pooling as a weighted average of values, where weights are computed with the attention scoring function a and the *softmax* operation.

Nonparametric Nadaraya-Watson kernel regression enjoys the consistency benefit: given enough data this model converges to the optimal solution. Nonetheless, we can easily integrate learnable parameters into attention pooling. As an example, slightly different from Equation 14, in the following the distance between the query $\mathbf{x}$ and the key x_i is multiplied by a learnable parameter w:

$$\begin{aligned} f(x) &= \sum_{i=1}^{n} \alpha(x, x_i) y_i \\ &= \sum_{i=1}^{n} \frac{\exp\left(-\frac{1}{2}((x - x_i)w)^2\right)}{\sum_{j=1}^{n} \exp\left(-\frac{1}{2}((x - x_j)w)^2\right)} y_i \\ &= \sum_{i=1}^{n} \operatorname{softmax}\left(-\frac{1}{2}((x - x_i)w)^2\right) y_i. \end{aligned} \tag{15}$$

2.4 Attention Scoring Functions

In Section 2.3, a number of different distance-based kernels are displayed, including a Gaussian kernel to model interactions between queries and keys. As it turns out, distance functions are slightly more expensive to compute than *dot products*. As such, with the softmax operation to ensure nonnegative attention weights, much of the work has gone into attention scoring functions α in equation and Figure 2 that are simpler to compute.

2.4.1 Dot Product Attention

Let's review the attention function (without exponentiation) from the Gaussian kernel for a moment:

$$a(\mathbf{q}, \mathbf{k}_i) = -\frac{1}{2}\|\mathbf{q} - \mathbf{k}_i\|^2 = \mathbf{q}^\top \mathbf{k}_i - \frac{1}{2}\|\mathbf{k}_i\|^2 - \frac{1}{2}\|\mathbf{q}\|^2. \tag{16}$$

First, note that the final term depends on $\mathbf{q}$ only. As such it is identical for all $((q), \mathbf{k}_i)$ pairs. Normalizing the attention weights to $\mathbf{1}$, as is done in equation 7, ensures that this term *disappears* entirely. Second, note that both batch and layer normalization (to be discussed later) lead to activations that have well-bounded, and often constant, norms $||\mathbf{k}_i||$. This is the case, for instance, whenever the keys $\mathbf{k}_i$ were generated by a layer norm. As such, we can drop it from the definition of a without any major change in the outcome.

Last, we need to keep the order of magnitude of the arguments in the exponential function under control. Assume that all the elements of the query $\mathbf{q} \in \mathbb{R}^d$ and the key $\mathbf{k}_i \in \mathbb{R}^d$ are independent and identically drawn random variables with *zero* mean and *unit* variance. The dot product between both vectors has zero mean and a variance of d. To ensure that the variance of the dot product still remains $\mathbf{1}$ regardless of vector length, we use the *scaled dot product attention scoring function.* That is, we re-scale the dot product by $1/\sqrt{d}$. We thus arrive at the first commonly used attention function that is used, e.g., in Transformers [45]:

$$a(\mathbf{q}, \mathbf{k}_i) = \mathbf{q}^\top \mathbf{k}_i / \sqrt{d}. \tag{17}$$

Note that attention weights α still need normalizing. We can simplify this further via Equation 7 by using the *softmax* operation:

$$\alpha(\mathbf{q}, \mathbf{k}_i) = \mathrm{softmax}(a(\mathbf{q}, \mathbf{k}_i)) = \frac{\exp(\mathbf{q}^\top \mathbf{k}_i / \sqrt{d})}{\sum_{j=1} \exp(\mathbf{q}^\top \mathbf{k}_j / \sqrt{d})}. \tag{18}$$

2.4.2 Masked Softmax Operation

For decoder self-attention, all-to-all attention is *inappropriate*, because during the autoregressive decoding process, the decoder cannot attend to future outputs that has yet to be decoded. (To be more specific, it would not be predictable in the case of generating.) This can be solved by forcing the attention weights $w_{ij} = 0$ for all $i \leq j$, , called "causal masking". This attention mechanism is the "causally masked self-attention".

For instance, assume that we have three sentences list in Table 1. Since we do not want blanks in our attention model, we simply need to limit $\sum_{i=1}^{n} \alpha(\mathbf{q}, \mathbf{k}_i)\mathbf{v}_i$ to $\sum_{i=1}^{l} \alpha(\mathbf{q}, \mathbf{k}_i)\mathbf{v}_i$ for however long, $l \leq n$, the actual sentence is. Since it is such a common problem, it has a name: the *masked softmax operation.*

Actually, the implementation cheats ever so slightly by setting the values of $\mathbf{v}_i$, for $i > l$, to zero. Moreover, it sets the attention weights to a large negative number, such as -10^6, in order to make their contribution to gradients

Dive into Deep Learning
Learn to code <blank>
Hello world <blank> <blank>

Table 1: Three sample sentences for the demo of masked softmax operation

and values vanish in practice. This is done since linear algebra kernels and operators are heavily optimized for GPUs and it is faster to be slightly wasteful in computation rather than to have code with conditional (if-else) statements.

When *QKV attention* is used as a building block for an autoregressive decoder, and when at training time all input and output matrices have n rows, a masked attention variant is used:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}} + \mathbf{M})\mathbf{V}) \tag{19}$$

where the mask, $\mathbf{M} \in \mathcal{R}^{n \times n}$ is a stricly upper triangular matrix, with zeros on and below the diagonal and $-\infty$ in every element above the diagonal. The softmax output, also in $\in \mathcal{R}^{n \times n}$ is then lower triangular, with zeros in all elements above the diagonal. The masking ensures that for all $\mathbf{1} \leq i < j \leq n$, row i of the attention output is independent of row j of any of the three input matrices. The permutation invariance and equivariance properties of standard QKV attention do not hold for the masked variant.

The following is the demonstration code implemented in *pytorch*:

```
def masked_softmax(X, valid_lens):  #@save
    """Perform softmax operation by masking elements on the last
    axis."""
    # X: 3D tensor, valid_lens: 1D or 2D tensor
    def _sequence_mask(X, valid_len, value=0):
        maxlen = X.size(1)
        mask = torch.arange((maxlen), dtype=torch.float32, device=X
    .device)[None, :] < valid_len[:, None]
        X[~mask] = value # ~: bitwise NOT Operator
        return X

    if valid_lens is None:
        return nn.functional.softmax(X, dim=-1)
    else:
        shape = X.shape
        if valid_lens.dim() == 1:
            valid_lens = torch.repeat_interleave(valid_lens, shape
    [1])
        else:
            valid_lens = valid_lens.reshape(-1)
        # On the last axis, replace masked elements with a very
        # large negative value, whose exponentiation outputs 0
        X = _sequence_mask(X.reshape(-1, shape[-1]), valid_lens,
    value=-1e6)
        return nn.functional.softmax(X.reshape(shape), dim=-1)
```

2.4.3 Batch Matrix Multiplication

Another commonly used operation is to multiply batches of matrices by one another. This comes in handy when we have mini-batches of queries, keys, and values. More specifically, assume that

$$\begin{aligned}\mathbf{Q} &= [\mathbf{Q}_1, \mathbf{Q}_2, \ldots, \mathbf{Q}_n] \in \mathbb{R}^{n\times a\times b},\\ \mathbf{K} &= [\mathbf{K}_1, \mathbf{K}_2, \ldots, \mathbf{K}_n] \in \mathbb{R}^{n\times b\times c}.\end{aligned} \tag{20}$$

Then the batch matrix multiplication (BMM) computes the elementwise product

$$\mathrm{BMM}(\mathbf{Q}, \mathbf{K}) = [\mathbf{Q}_1\mathbf{K}_1, \mathbf{Q}_2\mathbf{K}_2, \ldots, \mathbf{Q}_n\mathbf{K}_n] \in \mathbb{R}^{n\times a\times c}. \tag{21}$$

BMM enables efficient parallel processing across these sequences, applying matrix multiplication across the batch and multiple heads without sacrificing performance.

2.4.4 Scaled Dot Product Attention

Let's return to the dot product attention introduced in Equation 17. In general, it requires that both the query and the key have the same vector length, say d, even though this can be addressed easily by replacing $\mathbf{q}^\top\mathbf{k}$ with $\mathbf{q}^\top\mathbf{M}\mathbf{k}$ where $\mathbf{M}$ is a matrix suitably chosen for translating between both spaces. For now assume that the dimensions match.

In practice, we often think of mini-batches for efficiency, such as computing attention for n queries and m key-value pairs, where queries and keys are of length d and values are of length v. The scaled dot product attention of queries $\mathbf{Q} \in \mathbb{R}^{n\times d}$, keys $\mathbf{K} \in \mathbb{R}^{m\times d}$, and values $\mathbf{V} \in \mathbb{R}^{m\times v}$ thus can be written as

$$\mathrm{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d}}\right)\mathbf{V} \in \mathbb{R}^{n\times v}. \tag{22}$$

Note that when applying this to a minibatch, we need the batch matrix multiplication introduced in Equation 21.

The following is the demonstration code in the case of the calculation of the dot product of *Query Matrix* and *Key Matrix* implemented in *pytorch*:

```python
import torch
import torch.nn.functional as F

# Assume the shape of Q, K, V (batch_size, num_heads, seq_len,
    depth)
batch_size, num_heads, seq_len, depth = 2, 4, 5, 8

# Randomly initialize Q, K, V tensors
Q = torch.rand((batch_size, num_heads, seq_len, depth))
K = torch.rand((batch_size, num_heads, seq_len, depth))
V = torch.rand((batch_size, num_heads, seq_len, depth))

# 1. Calculate the dot product of Q and K
# Q @ K^T: BMM will be used here, and K will be transposed into (
    batch_size, num_heads, depth, seq_len)
```

```
attention_scores = torch.matmul(Q, K.transpose(-2, -1)) / torch.
    sqrt(torch.tensor(depth).float())

# 2. Apply softmax to attention_scores
attention_weights = F.softmax(attention_scores, dim=-1)

# 3. Apply attention weight to V
output = torch.matmul(attention_weights, V) # The second
    application of BMM

print(output.shape)
# The shape of the result should be (batch_size, num_heads, seq_len
    , depth)
```

2.4.5 Additive Attention

When queries $\mathbf{q}$ and keys $\mathbf{k}$ are vectors of different dimension, we can either use a matrix to address the mismatch via $\mathbf{q}^\top \mathbf{M} \mathbf{k}$, or we can use additive attention as the scoring function.

Another benefit is that, as its name indicates, the attention is additive. This can lead to some minor computational savings. Given a query $\mathbf{q} \in \mathbb{R}^q$ and a key $\mathbf{k} \in \mathbb{R}^k$, the *additive attention scoring function* [2] is given by

$$a(\mathbf{q}, \mathbf{k}) = \mathbf{w}_v^\top \tanh(\mathbf{W}_q \mathbf{q} + \mathbf{W}_k \mathbf{k}) \in \mathbb{R}, \tag{23}$$

where $\mathbf{W}_q \in \mathbb{R}^{h \times q}$, $\mathbf{W}_k \in \mathbb{R}^{h \times k}$, and $\mathbf{w}_v \in \mathbb{R}^h$ are the learnable parameters. This term is then fed into a softmax to ensure both nonnegativity and normalization. An equivalent interpretation of Equation 23 is that the query and key are concatenated and fed into an MLP with a single hidden layer. Using tanh as the activation function and disabling bias terms, we implement additive attention as follows:

```
class AdditiveAttention(nn.Module):  #@save
    """Additive attention."""
    def __init__(self, num_hiddens, dropout, **kwargs):
        super(AdditiveAttention, self).__init__(**kwargs)
        self.W_k = nn.LazyLinear(num_hiddens, bias=False)
        self.W_q = nn.LazyLinear(num_hiddens, bias=False)
        self.w_v = nn.LazyLinear(1, bias=False)
        self.dropout = nn.Dropout(dropout)

    def forward(self, queries, keys, values, valid_lens):
        queries, keys = self.W_q(queries), self.W_k(keys)
        # After dimension expansion, shape of queries: (batch_size,
     no. of
        # queries, 1, num_hiddens) and shape of keys: (batch_size,
    1, no. of
        # key-value pairs, num_hiddens). Sum them up with
    broadcasting
        features = queries.unsqueeze(2) + keys.unsqueeze(1)
        features = torch.tanh(features)
        # There is only one output of self.w_v, so we remove the
    last
```

```
        # one-dimensional entry from the shape. Shape of scores: (
    batch_size,
        # no. of queries, no. of key-value pairs)
        scores = self.w_v(features).squeeze(-1)
        self.attention_weights = masked_softmax(scores, valid_lens)
        # Shape of values: (batch_size, no. of key-value pairs,
    value
        # dimension)
        return torch.bmm(self.dropout(self.attention_weights),
    values)
```

2.5 Attention Weights

2.5.1 Decoder Cross-Attention: Computing the Attention Weights by Dot-Product

As hand-crafting weights defeats the purpose of machine learning, the model must compute the attention weights on its own. Taking an analogy from the language of database queries, the model constructs a triple of vectors: *key*, *query*, and *value*. The rough idea is that we have a "database" in the form of a list of key-value pairs. The decoder sends in a **query** and obtains a reply in the form of a weighted sum of the **values**, where the weight is proportional to how closely the query resembles each **key**.

The decoder first processes the "`<start>`" input partially to obtain an intermediate vector h_0^d, the *0*th hidden vector of the decoder. Then, the intermediate vector is transformed by a linear map W^Q into a query vector $q_0 = h_0^d W^Q$. Meanwhile, the hidden vectors outputted by the encoder are transformed by another linear map W^K into key vectors $k_0 = h_0 W^K, k_1 = h_1 W^K, \ldots$. These linear maps provide the model with the flexibility to represent the data in the most suitable way.

Next, the query and keys are compared by taking dot products: $q_0 k_0^T, q_0 k_1^T, \ldots$. Ideally, the model learns to compute the keys and values such that $q_0 k_0^T$ is large, $q_0 k_1^T$ is small, and the rest are even smaller. This can be interpreted as applying most of the attention weight to the *0*th hidden vector of the encoder, a little to the 1st, and essentially none to the rest.

To create a properly weighted sum, we transform this list of dot products into a probability distribution over $\mathbf{0}$, $\mathbf{1}, \ldots$, using the softmax function to obtain the attention weights:

$$(w_{00}, w_{01}, \ldots) = \text{softmax}(q_0 k_0^T, q_0 k_1^T, \ldots)$$

This is then used to compute the **context vector**:

$$c_0 = w_{00} v_0 + w_{01} v_1 + \ldots$$

where $v_0 = h_0 W^V, v_1 = h_1 W^V, \ldots$ are the **value** vectors, which are linearly transformed by another matrix to give the model flexibility in representing the values. Without the matrices W^Q, W^K, and W^V, the model would be forced to use the same hidden vector for both key and value, which may not be ideal since these tasks differ.

This is the dot-attention mechanism. The version described here is "decoder cross-attention", as the output context vector is used by the decoder, while the input keys and values come from the encoder, and the query comes from the decoder, thus "cross-attention".

2.5.2 Compact Representation of the Attention Mechanism

More succinctly, we can write the attention mechanism as:

$$c_0 = \text{Attention}(h_0^d W^Q, HW^K, HW^V) = \text{softmax}((h_0^d W^Q)(HW^K)^T)(HW^V)$$

where the matrix H is the matrix whose rows are $h_0, h_1, \ldots$. Note that the querying vector h_0^d is not necessarily the same as the key-value vector h_0. In fact, it is theoretically possible for the query, key, and value vectors to all be different, though this is rarely done in practice.

2.6 Self-Attention

Self-attention is essentially the same as cross-attention, except that the query, key, and value vectors all come from the same model. Both the encoder and decoder can use self-attention, but with subtle differences.

2.6.1 Encoder Self-Attention

For encoder self-attention, we can start with a simple encoder without self-attention, such as an "embedding layer", which simply converts each input word into a vector by a fixed lookup table. This gives a sequence of hidden vectors $h_0, h_1, \ldots$. These can then be applied to a dot-product attention mechanism, resulting in:

$$\begin{aligned} h_0' &= \text{Attention}(h_0 W^Q, HW^K, HW^V) \\ h_1' &= \text{Attention}(h_1 W^Q, HW^K, HW^V) \\ &\ldots \end{aligned}$$

or more succinctly:

$$H' = \text{Attention}(HW^Q, HW^K, HW^V)$$

This can be applied repeatedly to obtain a multilayered encoder. This is the "encoder self-attention", sometimes referred to as "all-to-all attention", as the vector at every position can attend to every other position.

2.7 Multi-Head Attention

2.7.1 Multi-Head Attention Mechanism

The multi-head attention mechanism is defined as:

$$\text{MultiHead}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)\mathbf{W}^O$$

where each head is computed with QKV attention as:

$$\text{head}_i = \text{Attention}(\mathbf{Q}\mathbf{W}_i^Q, \mathbf{K}\mathbf{W}_i^K, \mathbf{V}\mathbf{W}_i^V)$$

and $\mathbf{W}_i^Q$, $\mathbf{W}_i^K$, $\mathbf{W}_i^V$, and $\mathbf{W}^O$ are learnable parameter matrices.

The permutation properties of (standard, unmasked) QKV attention apply here as well. For permutation matrices $\mathbf{A}$ and $\mathbf{B}$:

$$\text{MultiHead}(\mathbf{AQ}, \mathbf{BK}, \mathbf{BV}) = \mathbf{A}\text{MultiHead}(\mathbf{Q}, \mathbf{K}, \mathbf{V})$$

From this, we can also observe that multi-head self-attention:

$$\mathbf{X} \mapsto \text{MultiHead}(\mathbf{X}\mathbf{T}_q, \mathbf{X}\mathbf{T}_k, \mathbf{X}\mathbf{T}_v)$$

is equivariant with respect to re-ordering of the rows of the input matrix $\mathbf{X}$.

2.7.2 Bahdanau (Additive) Attention

The Bahdanau (additive) attention mechanism is defined as:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}(e)\mathbf{V}$$

where

$$e = \tanh(\mathbf{W}_Q\mathbf{Q} + \mathbf{W}_K\mathbf{K})$$

and $\mathbf{W}_Q$ and $\mathbf{W}_K$ are *learnable* weight matrices.

2.7.3 Luong Attention (General)

The Luong (general) attention mechanism is defined as:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}(\mathbf{Q}\mathbf{W}_a\mathbf{K}^T)\mathbf{V}$$

where $\mathbf{W}_a$ is a learnable weight matrix.

3 Conclusion

In conclusion, attention mechanisms, particularly those utilized within Transformer architectures, have revolutionized the field of artificial intelligence by enabling efficient handling of sequential data across various domains. By allowing models to focus on relevant parts of the input and efficiently compute dependencies over long ranges, attention mechanisms overcome many of the limitations posed by traditional architectures like CNNs and RNNs.

Multi-head attention, self-attention, and cross-attention have become foundational elements in not only natural language processing but also computer vision, reinforcement learning, and other fields. The scalability and adaptability of Transformers, alongside the evolving research in attention mechanisms, suggest that they will remain integral to AI's future developments.

However, challenges such as computational complexity, interpretability, and bias in Transformer-based models need to be addressed for broader and more responsible deployment. Future research may focus on optimizing efficiency, improving model explainability, and exploring novel architectures that can further enhance AI's capabilities across different fields.

The rapid adoption of Transformers, both in research and industry, indicates that attention-based mechanisms are likely to remain central to the ongoing evolution of artificial intelligence, fostering advancements across diverse and complex tasks.

References

[1] S. Ahmad. Visit: A neural model of attention and memory in visual information processing. *Neural Networks*, 1991.

[2] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2015.

[3] I. Beltagy, M. E. Peters, and A. Cohan. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020.

[4] Herman J. Bierens. *The Nadaraya–Watson kernel regression function estimator.* Cambridge University Press, New York, 1994.

[5] D. Broadbent. *Perception and Communication.* Pergamon Press, 1958.

[6] T. B. Brown and et al. Language models are few-shot learners. In *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS)*, 2020.

[7] J. Cheng. Long short-term memory-networks for machine reading. *arXiv preprint arXiv:1601.06733*, 2016.

[8] C. Cherry. Some experiments on the recognition of speech, with one and with two ears. *Journal of the Acoustical Society of America*, 1953.

[9] R. Child and et al. Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509*, 2019.

[10] K. Clark, M.-T. Luong, Q. V. Le, and C. D. Manning. Electra: Pre-training text encoders as discriminators rather than generators. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2020.

[11] J. Devlin and et al. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the Annual Conference of the North American Chapter of the Association for Computational Linguistics (NAACL-HLT)*, 2019.

[12] L. Dong and et al. Speech-transformer: A no-recurrence sequence-to-sequence model for speech recognition. In *Proceedings of the International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2018.

[13] A. Dosovitskiy and et al. An image is worth 16x16 words: Transformers for image recognition at scale. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2021.

[14] P. Esser and et al. Taming transformers for high-resolution image synthesis. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2021.

[15] K. Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 1980.

[16] A. Graves and et al. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

[17] A. Gulati and et al. Conformer: Convolution-augmented transformer for speech recognition. In *Proceedings of Interspeech*, 2020.

[18] David Ha and Jürgen Schmidhuber. Hypernetworks. *International Conference on Learning Representations*, pages 1–7, 2017.

[19] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[20] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 1997.

[21] John P. Hoffmann. Multiplication in neural networks. *Neural Networks*, 2:39–44, 1989.

[22] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2015.

[23] L. Itti, C. Koch, and E. Niebur. A model of saliency-based visual attention for rapid scene analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1998.

[24] A.G. Ivakhnenko and V.G. Lapa. *Cybernetic Predicting Devices.* Jprs report. CCM Information Corporation, 1973.

[25] Alexey Grigorevich Ivakhnenko. The group method of data handling; a rival of the method of stochastic approximation. 1968.

[26] A. Katharopoulos and et al. Transformers are rnns: Fast autoregressive transformers with linear attention. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2020.

[27] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[28] N. Kitaev and et al. Reformer: The efficient transformer. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2020.

[29] Andrey Kolmogorov and Dennis Gabor. On the representation of continuous functions of several variables by superpositions of continuous functions of one variable and addition. *Doklady Akademii Nauk SSSR*, 114:953–956, 1965.

[30] H. Larochelle and G. Hinton. A foveal image classification system. In *Proceedings of the International Conference on Neural Information Processing Systems (NeurIPS)*, 2010.

[31] J. Lee and et al. Set transformer: A framework for attention-based permutation-invariant neural networks. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2019.

[32] Y. Liu and et al. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[33] M. Luong, H. Pham, and C. D. Manning. Effective approaches to attention-based neural machine translation. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2015.

[34] Y.-P. Mack and B. W. Silverman. Weak and strong uniform consistency of kernel regression estimates. *Zeitschrift für Wahrscheinlichkeitstheorie und verwandte Gebiete*, 61(3):405–415, 1982.

[35] Volodymyr Mnih, Nicolas Heess, Alex Graves, et al. Recurrent models of visual attention. In *Advances in Neural Information Processing Systems*, pages 2204–2212, 2014.

[36] E. A. Nadaraya. On estimating regression. *Theory of Probability and Its Applications*, 9(1):141–142, 1964.

[37] V. Nair and G. E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2010.

[38] E. Parisotto and et al. Stabilizing transformers for reinforcement learning. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2020.

[39] Michael A. Poggio and Tomaso A. Poggio. Networks with Sigma-Pi units. *Artificial Intelligence*, 44:179–203, 1989.

[40] A. Radford and et al. Language models are unsupervised multitask learners. *OpenAI Blog*, 2019.

[41] K. Rayner. Eye movements in reading and information processing: 20 years of research. *Psychological Bulletin*, 1998.

[42] Jürgen Schmidhuber. Learning to control fast-weight memories: an alternative to recurrent nets. *Neural Computation*, 4(1):131–139, 1992.

[43] G. Sperling. The information available in brief visual presentations. *Psychological Monographs: General and Applied*, 1960.

[44] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research (JMLR)*, 2014.

[45] A. Vaswani and et al. Attention is all you need. In *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS)*, 2017.

[46] G. S. Watson. Smooth regression analysis. *Sankhyā: The Indian Journal of Statistics, Series A*, 26(4):359–372, 1964.

[47] Paul J. Werbos. Beyond regression: New tools for prediction and analysis in the behavioral sciences. *Ph.D. Thesis, Harvard University*, 1974.

[48] Wikipedia contributors. Attention (machine learning) — wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Attention_(machine_learning)`, 2023. Accessed: 2023-10-05.

[49] K. Xu and et al. Show, attend and tell: Neural image caption generation with visual attention. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2015.

[50] R. Ying and et al. Do transformers really perform bad for graph representation? In *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS)*, 2021.