Attention and Transformers

Takuto Ban
University of Massachusetts
tban@umass.edu

John Wang
University of Massachusetts
jawang@umass.edu

Wenhao Yang
University of Massachusetts
wenhaoyang@umass.edu

1 Introduction

1.1 Technical Motivation

Large Language Models, such as ChatGPT, are currently dominating the world with its ability to respond to user queries with high-level expertise and intelligence. The main framework used in these models is the Transformer architecture, a state-of-the-art architecture for various sequence generation tasks such as text generation, machine translation, music generation, and protein sequence generation. Huge amounts of research efforts have been put into the Transformers, whether that is in refining their underlying architecture, optimizing their training methodologies, or exploring their applications across various fields. We want to read about the historical context in which transformers were invented, from its origins in machine translation. We want to understand the mathematical intuitions behind the techniques used in transformers, such as attention and feed-forward neural networks. We want to dive into the overall transformer architecture, examining various strategies in the architecture, as well as the training process, focusing on the computational aspects.

1.2 Mathematical Foundations

Within each attention head, every prompt is split into features and represented by high dimensional vectors called word embeddings. To start, we need to measure which features interact with each other. Specifically, we need a query and key matrix. Multiplying each feature by the query matrix transforms the feature into a vector in the query/key space. Each feature is also multiplied by the key matrix which transforms the feature into the same space. We take the inner product of every transformed feature by the query matrix and every transformed feature by the key matrix to determine the features that interact. Then, for each feature, we determine a probability distribution using a softmax function. Lastly we determine how the features interact by multiplying each by the value matrix. We modify each feature by adding the contribution of the others given by the value matrix weighted by the probability distribution of the feature.

1.3 Objectives

We focused mainly on the theory of transformers. As attention is a crucial component of transformers, we covered both transformers and attention in depth. Specifically, we researched attention history, attention mathematics, and transformers. Within attention history, we covered recurrent neural networks (RNNs) and the encoder/decoder architecture. For attention mathematics, we researched the concepts of queries, keys, values; attention pooling; attention scoring; and kernel regression as nonparametric attention. As for transformers, we looked at self-attention, multi-headed attention, and the architecture.

1.4 Roles and Responsibilities

For this project, Wenhao researched attention history. He also managed communications and logistics within the group. John researched attention mathematics. He was in charge of documentation and also led the construction of the poster. Takuto researched transformers. He kickstarted the research effort with his background knowledge and helped combine the end result into a cohesive project.

2 The Mathematics of Attention

At a high level, the attention mechanism takes in a sequence of vectors $\mathbf{x}_1, \ldots, \mathbf{x}_n \in \mathbf{R}^d$ and transforms them into a new sequence $\mathbf{z}_1, \ldots, \mathbf{z}_n \in \mathbf{R}^d$. You can interpret each $\mathbf{z}_i$ as a vector enriched by the entire input sequence $\mathbf{x}_1, \ldots, \mathbf{x}_n$. For example, suppose $\mathbf{x}_1, \ldots, \mathbf{x}_n$ represent word embeddings (i.e., $\mathbf{x}_i$ represents the semantic meaning of the ith word in the sequence). Then, $\mathbf{z}_i$ represents the ith word embedding after is absorbs the context of the words around it. To make this even more concrete, consider the phrase "agile fox". The initial vector $\mathbf{x}_i$ that describes the fox may represent the animal in a resting position without consider the context. The output vector $\mathbf{z}_i$ could represent this fox chasing prey which is a more appropriate description of the word.

2.1 Definition

With this as motivation we define

$$\mathbf{z}_i = \sum_{j=1}^{n} \alpha_{ij} x_j \ .$$

where the α_{ij} are attention weights that satisfy $\forall i, j \in [n], 0 \leq \alpha_{ij} \leq 1$ and $\forall i \in [n], \sum_{j=1}^{n} \alpha_{ij} = 1$. That is, for all i, the α_{ij} follow a probability distribution. As a result, the transformed vector $\mathbf{z}_i$ is a weighted average of the input vectors determined by the attention weights. We can generalize this definition of attention by defining

$$\mathbf{z}_i = \sum_{j=1}^{n} \alpha_{ij} v_j$$

where $v_j = V x_j$ and $V \in \mathbf{R}^{d \times d}$ transforms vectors in $\mathbf{R}^d$ to the same space. V is called a value matrix and it is full of learnable parameters [Mur22]. From here, the attention weights α_{ij} should be a measure of the influence $\mathbf{x}_j$ has on $\mathbf{x}_i$. We determine the value of α_{ij} through attention pooling.

2.2 Attention Pooling

In attention pooling, we generate α_{ij} by comparing $\mathbf{x}_i$ to $\mathbf{x}_j$. However, we do not compare them directly. Instead we compare the vectors $\mathbf{q}_i = Q\mathbf{x}_i$ where $Q \in \mathbf{R}^{q \times d}$ and $\mathbf{k}_j = K\mathbf{x}_j$ where $K \in \mathbf{R}^{q \times d}$. Q is a query matrix that transforms input vectors into a lower dimensional key-query space (i.e., $k < d$). K is a key matrix that transforms input vectors into the same key-query space. Both Q and K are full of learnable parameters.

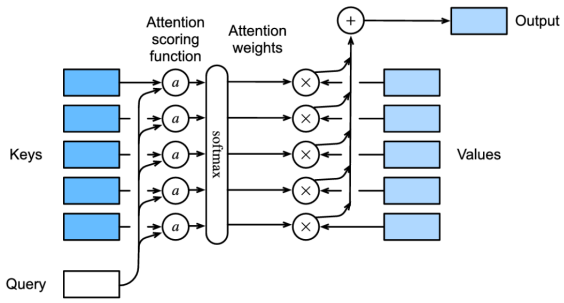
What is the benefit of comparing $\mathbf{q}_i$ to $\mathbf{k}_j$ rather than the initial vectors? We can think of $\mathbf{q}_i$ as a question that $\mathbf{x}_i$ poses. For example, if $\mathbf{x}_i$ represents a noun, $\mathbf{q}_i$ could represent the question: "Are you an adjective that describes me?". Similarly, we can think of $\mathbf{k}_j$ as an answer that $\mathbf{x}_j$ gives. For example, if $\mathbf{x}_j$ represents an adjective, $\mathbf{k}_j$ could represent the response: "Yes, I am an adjective.". Lastly, the key-query space gives the range of questions and answers that could appear. As a result, by comparing $\mathbf{q}_i$ to $\mathbf{k}_j$ we are determining how well the question posed by $\mathbf{x}_i$ is answered by $\mathbf{x}_j$.

With this intuition, in order to compare $\mathbf{q}_i$ to $\mathbf{k}_j$ we use an attention scoring function $a : \mathbf{R}^q \times \mathbf{R}^q \to \mathbf{R}$. We will cover various attention scoring functions in the next subsection. Therefore, we compare $\mathbf{q}_i$ to $\mathbf{k}_j$ using $a(\mathbf{q}_i, \mathbf{k}_j)$. As a result, the influence of $\mathbf{x}_j$ on $\mathbf{x}_i$ is given by $a(\mathbf{q}_i, \mathbf{k}_j)$. However, $\alpha_{ij} \neq a(\mathbf{q}_i, \mathbf{k}_j)$ because the $a(\mathbf{q}_i, \mathbf{k}_j)$ do not necessarily follow a valid probability distribution.

As a result, we transform the $a(\mathbf{q}_i, \mathbf{k}_j)$ into a valid probability distribution using the softmax function. Specifically,

$$\text{softmax}(a(\mathbf{q}_i, \mathbf{k}_j)) = \frac{\exp(a(\mathbf{q}_i, \mathbf{k}_j))}{\sum_{j=1}^{n} \exp(a(\mathbf{q}_i, \mathbf{k}_j))} \ .$$

Notice that the numerator of softmax transforms all negative scores to positive scores. The denominator transforms all of the now positive scores into proportions of the sum of the scores. Now that all of the components of attention have been explained, see the visual below to see all the steps involved with finding a single output vector $\mathbf{z}_i$ [Mur22].



2.3 Attention Scoring

We will end our discussion of attention mathematics with specific attention scoring functions and the motivation for each one.

2.3.1 Dot Product Scoring

The most common attention scoring function used is dot product scoring. That is, $a(\mathbf{q}_i, \mathbf{k}_j) = \mathbf{q}_i^T \mathbf{k}_j$. In dot product scoring, we assume that $\mathbf{q}_i$ and $\mathbf{k}_j$ are in the same dimension. Notice that we said earlier that $\mathbf{q}_i$ and $\mathbf{k}_j$ live in the key-query space and thus the key space is the same as the query space. In some situations, we make the key space different from the query space in which case dot product scoring is not viable.

Attention with dot product scoring can be computed very efficiently using matrix multiplication. In particular if $Z \in \mathbf{R}^{n \times d}$ has rows corresponding to our output vectors $\mathbf{z}_1, \ldots, \mathbf{z}_n$ then $Z = \text{softmax}\left(\frac{QK^T}{\sqrt{q}}\right) V$ [Vas17]. To break this down we first consider QK^T. Let $Q \in \mathbf{R}^{n \times q}$ have rows given by $\mathbf{q}_1, \ldots, \mathbf{q}_n$ and $K \in \mathbf{R}^{n \times q}$ have rows given by $\mathbf{k}_1, \ldots, \mathbf{k}_n$. Then the i, j entry of QK^T is equal to $\mathbf{q}_i^T \mathbf{k}_j$. As a result, row i of QK^T refers to the score distribution for the query $\mathbf{q}_i$ (i.e., $\mathbf{q}_i^T \mathbf{k}_1, \ldots, \mathbf{q}_i^T \mathbf{k}_n$). See visual below for more clarification.

$$QK^T = \begin{bmatrix} — & \mathbf{q}_1^T & — \\ & \vdots & \\ — & \mathbf{q}_n^T & — \end{bmatrix}_{n \times q} \begin{bmatrix} | & & | \\ \mathbf{k}_1 & \cdots & \mathbf{k}_n \\ | & & | \end{bmatrix}_{q \times n} = \begin{bmatrix} \mathbf{q}_1^T \mathbf{k}_1 & \cdots & \mathbf{q}_1^T \mathbf{k}_n \\ \vdots & \ddots & \vdots \\ \mathbf{q}_n^T \mathbf{k}_1 & \cdots & \mathbf{q}_n^T \mathbf{k}_n \end{bmatrix}_{n \times n}$$

Then, we normalize each entry in QK^T with $1/\sqrt{q}$ yielding the matrix $QK^T/\sqrt{q}$. We justify this choice with a hypothetical. Suppose that the entries $q_i(1), \ldots, q_i(q)$ in $\mathbf{q}_i$ and $k_j(1), \ldots, k_j(q)$ in $\mathbf{k}_j$ are iid random variables with expectation zero and unit variance.

$$\begin{aligned}
V[\mathbf{q}_i^T \mathbf{k}_j] &= V\left[\sum_{m=1}^{q} q_i(m) k_j(m)\right] && \text{dot product def} \\
&= \sum_{m=1}^{q} V[q_i(m) k_j(m)] && \text{independence assumption} \\
&= \sum_{m=1}^{q} E[q_i(m)^2 k_j(m)^2] - E[q_i(m) k_j(m)]^2 && \text{variance def} \\
&= \sum_{m=1}^{q} E[q_i(m)^2] E[k_j(m)^2] - E[q_i(m)]^2 E[k_j(m)]^2 && \text{independence assumption} \\
&= \sum_{m=1}^{q} E[q_i(m)^2] E[k_j(m)^2] && \text{all entries have expectation zero} \\
&= \sum_{m=1}^{q} V[q_i(m)] V[k_j(m)] = q && \text{variance def}
\end{aligned}$$

As a result, to force $V[\mathbf{q}_i^T \mathbf{k}_j]$ to have unit variance we multiply by $1/\sqrt{q}$ [ZLLS23]. From here, softmax$(Qk^T/\sqrt{q})$ is the matrix that results from having applied softmax row-wise. It is done row-wise because the score distribution for

each query $\mathbf{q}_i$ must be a valid probability distribution. Then, the i, j entry of softmax$(Qk^T/\sqrt{q})$ is the attention weight α_{ij}. Lastly, softmax$(Qk^T/\sqrt{q})V$ can be visualized below.

$$\text{softmax}\left(\frac{Qk^T}{\sqrt{q}}\right)V = \begin{bmatrix} \alpha_{11} = \text{softmax}(\mathbf{q}_1^T\mathbf{k}_1/\sqrt{q}) & \cdots & \alpha_{1n} = \text{softmax}(\mathbf{q}_1^T\mathbf{k}_n/\sqrt{q}) \\ \vdots & \ddots & \vdots \\ \alpha_{n1} = \text{softmax}(\mathbf{q}_n^T\mathbf{k}_1/\sqrt{q}) & \cdots & \alpha_{nn} = \text{softmax}(\mathbf{q}_n^T\mathbf{k}_n/\sqrt{q}) \end{bmatrix}_{n\times n} \begin{bmatrix} — & \mathbf{v}_1 & — \\ & \vdots & \\ — & \mathbf{v}_n & — \end{bmatrix}_{n\times d}$$

$$= \begin{bmatrix} — & \mathbf{z}_1 & — \\ & \vdots & \\ — & \mathbf{z}_n & — \end{bmatrix}_{n\times d}$$

Let $V \in \mathbf{R}^{n\times d}$ have rows given by $\mathbf{v}_1, \ldots, \mathbf{v}_n$. Then row i in softmax$(Qk^T/\sqrt{q})V$ is equal to a linear combination of the rows of V (i.e., the $\mathbf{v}_i$) weighted by the ith row of softmax$(Qk^T/\sqrt{q})V$ (i.e. the attention weights for $\mathbf{x}_i$). Notice that this gives the matrix $Z \in \mathbf{R}^{n\times d}$ with rows representing the $\mathbf{z}_i$.

2.3.2 Additive Scoring

We conclude our discussion of attention scoring functions with two less common examples. Suppose $\mathbf{q}_i$ and $\mathbf{k}_j$ live in different dimensions. Then we can apply the additive scoring function,

$$a(\mathbf{q}_i, \mathbf{k}_j) = \mathbf{w}_v^T \tanh(W_q\mathbf{q}_i + W_k\mathbf{k}_j)$$

where $W_q \in \mathbf{R}^{h\times q}$, $W_q \in \mathbf{R}^{h\times k}$, and $\mathbf{w}_v \in \mathbf{R}^h$ are full of learnable parameters [ZLLS23]. Notice that W_q and W_k transform $\mathbf{q}_i$ and $\mathbf{k}_j$ respectively into h dimensions and then further manipulation is done.

2.3.3 Non-parametric Scoring Motivated by Kernel Regression

Finally, we introduce nonparametric scoring motivated by kernel regression. The goal of kernel regression is similar to the goal of attention. In kernel regression, we calculate a fitted value by taking a weighted average over all other values. For example, if we have the ordered pairs $(x_1, y_1), \ldots, (x_n, y_n)$ and we need to determine $\hat{y}$ from x we evaluate

$$\hat{y} = \sum_{i=1}^{n} w_i y_i \ .$$

Like in attention we want w_i to explain the similarity between x and x_i. In kernel regression, w_i is calculated using a Gaussian kernel or some other kernel. If we are using a Gaussian kernel, we use $\mathcal{K}_{\sigma^2}(x - x_i)$ where $\mathcal{K}_{\sigma^2}(u) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}u^2\right)$ [ZLLS23]. In rough terms, $\mathcal{K}_{\sigma^2}(x - x_i)$ gives a "probability measure" of the similarity between x and x_i.

Kernel regression motivates the use of a non-parametric kernel as an attention scoring function. For example, we can use the Gaussian kernel as an attention scoring function so that $a(\mathbf{q}_i, \mathbf{k}_j) = \mathcal{K}_{\sigma^2}(\mathbf{q}_i - \mathbf{k}_j)$.

3 The RNN Architecture and RNN with Attention

3.1 RNN Architecture

Recurrent Neural Networks (RNNs) take in an input $\mathbf{x}$ and return a sequence of outputs $\mathbf{y}_1, \ldots, \mathbf{y}_n$. Depending on the type of RNN, the input may be given as a sequence $\mathbf{x}_1, \ldots, \mathbf{x}_n$ as well. At each step t, the RNN uses the previous output $\mathbf{y}_{t-1}$ (if any) and either the entire input $\mathbf{x}$ or the next input in the sequence $\mathbf{x}_t$ to update its hidden state $\mathbf{h}_t$ through weights and biases. The hidden state is then used to generate the output y_t, which is then fed back into the RNN at the next step. Depending on the type of RNN, the sequence of outputs may be used as a whole, or just the last output may be taken.

We can represent the output distribution as

$$p(\mathbf{y}_t|\mathbf{h}_t) = \text{Cat}(\mathbf{y}_t|\text{softmax}(\mathbf{W}_{hy}\mathbf{h}_t + \mathbf{b}_y))$$

where Cat is the Categorical distribution, $\mathbf{W}_{hy}$ are the weights, and $\mathbf{b}_\mathbf{y}$ is the bias. Since y_t is determined from h_t, and h_t is affected by both the input $\mathbf{x}_t$ and the previous output y_{t-1}, which is in turn dependent on h_{t-1}, each output is implicitly dependent on all previous inputs as well.

One common type of RNNs is the Seq2Seq model, which takes in the input as a sequence and uses as output the entire sequence of outputs. In the case of Seq2Seq where the input length T and output length T' are different, encoder-decoder architecture can be used. First, an encoder RNN is used to transform the input sequence into a single context vector $\mathbf{c} = f_e(\mathbf{x}_{1:T})$. Then, a decoder RNN is used to transform the context vector into a sequence of outputs $\mathbf{y}_{1:T'} = f_d(\mathbf{c})$.

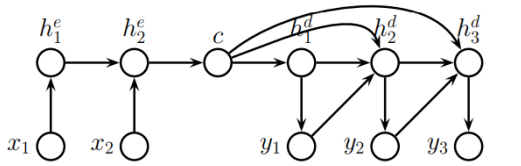


Figure 15.7: Encoder-decoder RNN architecture for mapping sequence $\boldsymbol{x}_{1:T}$ to sequence $\boldsymbol{y}_{1:T'}$.

Figure 1: Encoder-Decoder Architecture

A benefit of RNNs is that they have "unbounded memory" in that the length of the input and output sequences have no bound. However, although this is theoretically true, due to the architecture of RNNs, inputs earlier in the sequence have less of an effect on outputs later in the sequence. This is because previous inputs and outputs only affect the single hidden state, which is then later modified by all of the subsequent inputs and outputs. For uses like natural language processing and language translation, this can be a significant problem. For instance, if a long sentence is being processed, and the word "it" refers to a noun at the beginning of the sentence, the model needs to be able to take that into account, but cannot do so well with the current model of RNNs.

3.2 RNN with Attention

To solve the issue of outputs not being properly impacted by inputs, we can use attention with our RNNs.

Instead of just using the last hidden state of the encoder $\mathbf{h}_T^e$, a fixed size context vector, we now use a dynamic context vector:

$$\mathbf{c}_t = \sum_{i=1}^{T} \alpha_i(\mathbf{h}_{t-1}^d, \mathbf{h}_{1:T}^e)\mathbf{h}_i^e$$

Here, we are using attention with the previous decoder hidden state $\mathbf{h}_{t-1}^d$ as the query, all of the encoder hidden states $\mathbf{h}_{1:T}^e$ as the keys, and again the encoder hidden states as the values. Then, like before, this context vector $\mathbf{c}_t$ is used with the previous output $\mathbf{y}_{t-1}$ of the decoder to update the hidden state and generate the next output of the decoder.

The key difference is that instead of using a single, fixed-length context vector c for each step of the decoder, we now use a distinct context vector c_t, calculated using attention, for each step. This way, relevant inputs, represented by the encoder hidden states, are still able to have the necessary impact on the outputs generated by the decoder, even if inputs were located early in the input sequence.

4 Transformer

The main issue with using Attention with Recurrent Neural Network is that it processes sequences sequentially. Given an input sequence, the RNN processes the words in the token one by one, which results in its slow generation speed. The Transformer architecture, introduced in the landmark paper "Attention Is All You Need" by Vaswani et al., attempts to solve this problem [Vas17]. As implied by the title, the transformer relies solely on the attention mechanism to learn dependencies between words, completely getting rid of recurrence. This allows for a more parallelizable computation, and also results in significant improvements in generation quality compared to previous state-of-the-art methods. In this section we will be taking a look at the original Transformer architecture (which is shown in Figure 2), examining the functions and the motivations of each components.

4.1 Embedding and Positional Encoding

4.1.1 Input and Output Embedding

To even start working with sequences of text, we need to be able to convert words to numerical format. An Embedding maps a word into a vector, therefore it is essentially a look-up table. It is represented by a linear transformation $W_{d_v \times d_e}$ where d_e is the embedding dimension and d_v is the vocabulary size of the text language. Each word is given an index

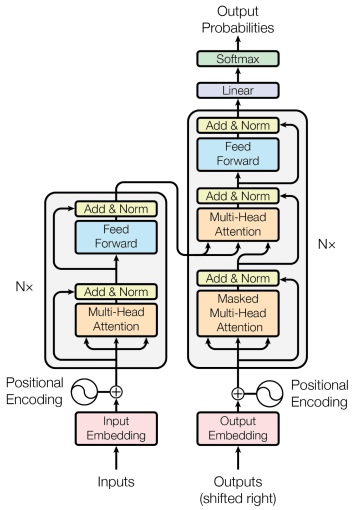


Figure 2: Original Transformer Architecture by Vaswani et al.

$i \in \{1, \cdots, d_v\}$, and its embedding is the ith row of the embedding transformation $e_i^T W$, where e_i is an one-hot vector containing all 0s except for a 1 at the ith position. The input/output embedding matrix can be calculated as $E = BW$, where each row of B is an one-hot vector of each word in the sequence. This linear transformation is trained along with the transformer model itself.

4.1.2 Positional Encoding

However, Embeddings does not incorporate positioning of the words. Attention mechanism is invariant to permutation; that is, the ordering of words does not affect the attention results. This is problematic, as language is affected by ordering; "the old man's dog" and "the man's old dog" have completely different meanings. In order to incorporate the concept of positions to the Transformer architecture, we add a Positional Encoding matrix P to the input/output embedding E. In the original Transformer, this is implemented as follows, where d is the model dimension.

$$P_{\text{pos},2i} = \sin\left(\frac{\text{pos}}{10000^{2i/d}}\right), \qquad P_{\text{pos},2i+1} = \cos\left(\frac{\text{pos}}{10000^{2i/d}}\right)$$

This incorporates both absolute positioning and relative positioning. To understand how this incorporates absolute positioning, let's examine binary numbers.

$$000, 001, 010, 011, 100, 101, 110, 111, \cdots$$

We can see that the first bit alternates every 3 numbers, the second bit alternates every 2 numbers, and the third bit alternates every number. The Positional Embedding above tries to model this behavior; each dimension (i) of the matrix has different frequencies of sinusoids, where the frequency decreases as i increases, and each word (pos) are at different positions of the sinusoids. This way, each word has values of differing frequencies, which is similar to binary numbers as seen above. In contrast, Positional Encoding incorporates relative positioning as a linear transformation, which is learnable by the attention mechanism (denote $\omega_j = 1/10000^{2j/d}$).

$$\begin{bmatrix} P_{i+\delta,2j} \\ P_{i+\delta,2j+1} \end{bmatrix} = \begin{bmatrix} \cos(\delta\omega_j) & \sin(\delta\omega_j) \\ -\sin(\delta\omega_j) & \cos(\delta\omega_j) \end{bmatrix} \begin{bmatrix} P_{i,2j} \\ P_{i,2j+1} \end{bmatrix}$$

4.2 Transformer Attentions

There are various types of attention used in the transformer; namely multi-head attention, self attention, cross attention, and masked attention.

4.2.1 Multi-Head Attention

Multi-Head Attention is simply the combination of multiple attentions into one component. Each of the attention component in multi-head attention is called an attention head, and each individual head contains its own learnable

transformations to the query, key, and value space. Intuitively, this would allow each attention head to learn different dependencies between words. For example, one attention head could learn the relationship between nouns and adjectives, another between verbs and adverbs, and so on. Computationally, multi-head attention is the concatenation of the results of each attention head, followed by a linear transformation.

$$\text{MultiHead}(Q, K, V) = \text{concat}(H_1, \cdots, H_h)W_O, \qquad \text{where } H_i = \text{Attn}(QW_{q,i}, KW_{k,i}, VW_{v,i})$$

In the Transformer architecture, every attention block is a multi-head attention, where each head is scaled dot product attention.

4.2.2 Self Attention and Cross Attention

Self Attention is Attention where the queries, keys, and values are all derived from a single matrix X (i.e. $Q = K = V = X$). Cross Attention is simply the opposite; it is Attention where the queries, keys, and values are *not* all derived from a single matrix. In the transformer architecture, the attention block in the encoder and the first attention block in the decoder are Self Attention, while the second attention block in the decoder is Cross Attention, which uses the previous decoder layer output as queries and the encoder output as keys and values.

The Vaswani paper emphasizes the importance of Self Attention to the computational efficiency of Transformers. Self Attention, Recurrence, and Convolution can all be used for mapping a variable-length sequence $(x_1, \cdots, x_n)$ to another sequence of equal length $(z_1, \cdots, z_n)$. Compared to Convolution, Self Attention allows a direct dependency between any elements in the sequence, where as Convolution is limited by its kernel size. Compared to Recurrence, Self Attention allows for a constant time computation ($O(1)$) through matrix multiplications, where as Recurrence requires linear complexity with respect to sequence length ($O(n)$). This makes Self Attention optimal for modeling dependencies in a sequence.

4.2.3 Masked Attention

Masked Attention blocks words from attending to future words, which is especially important during the training phase. Often times, we use self-supervised learning to train sequence models, where labeled data is generated from unlabeled data. For example, if we have an unlabeled data "I went to the park today.", we can create the following labeled data using chunks of the sentence; "I" $\rightarrow$ "went", "I went" $\rightarrow$ "to", "I went to" $\rightarrow$ "the", and so on. However during training, we feed the entire sentence into the Transformer, and predict the next word of each chunk of the sentence in parallel. Therefore, any word cannot attend to its proceeding words, in order to prevent words from looking "into the future". This makes the Transformer decoder an autoregressive model; where an element in the sequence is only affected by previous elements in the sequence.

Computationally, this is done by an attention mask. After computing the attention weights, we "mask" the weights not in the lower triangular region by setting the weights to $-\infty$. By doing so, applying the softmax operation would zero out the masked region, effectively disabling attention in that region.

$$\text{MaskedAttn}(Q, K, V) = \text{softmax}(\text{mask}(\frac{QK^T}{\sqrt{d}}))V$$

$$\text{where mask}(\begin{bmatrix} \bullet & \bullet & \bullet & \cdots \\ \bullet & \bullet & \bullet & \cdots \\ \bullet & \bullet & \bullet & \cdots \end{bmatrix}) = \begin{bmatrix} \bullet & -\infty & -\infty & \cdots \\ \bullet & \bullet & -\infty & \cdots \\ \bullet & \bullet & \bullet & \cdots \end{bmatrix}$$

4.3 Additional Components

Although the Transformer architecture is mainly composed of attention blocks, there are additional blocks that improves the overall performance. These are the Feed Forward Neural Networks, Residual Connections (Add), and Layer Normalization (Norm).

4.3.1 Feed-Forward Neural Networks

After the attention blocks, there are feed forward neural network blocks that are applied position-wise; each word embedding is fed into the neural network, instead of the entire chunk of text like the attention blocks. In the original Transformer, it is a two-layer neural network with a ReLU activation function in between ($\text{ReLU}(x) = \max(x, 0)$).

$$\text{FFN}(x) = \text{ReLU}(xW_1 + b_1)W_2 + b_2$$

4.3.2 Residual Connections

After each attention block and feed-forward network block, we add the block input to the output. This is referred to as residual connections, or skip connections (as seen in Figure 3), and this idea originated from image recognition using Convolutional Neural Network by He et al. [HZRS16]. In normal connections, the result is simply Block(x), but in residual connections, the result is Block$(x) + x$. Intuitively, this means that the attention blocks (and FFN blocks) incorporate additional information on top of the original text embedding, which is what we want, instead of completely replacing it with the result from attention. Computationally, this solves the vanishing gradient problem, where layers at the beginning of the network to not receive any gradients due to the depth of the network, which makes optimization difficult. Residual connections would solve this by adding the original input, which would allow the loss gradient to directly flow back to every layer and retain its magnitude.

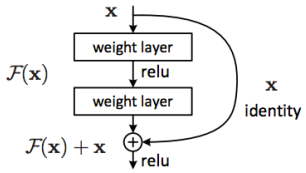


Figure 3: Residual Connection

4.3.3 Layer Normalization

Layer Normalization is a normalization method with learnable parameters γ and β. The Layer Norm is applied position-wise; therefore the mean and variance is calculated for each word across the embedding dimension.

$$\text{LayerNorm}(x) = \frac{x - \text{E}[x]}{\sqrt{\text{Var}[x]}} \cdot \gamma + \beta$$

In the original Transformer architecture, the Layer Norm is applied after the residual connection. However, most current Transformers does not implement normalization this way. A study done by Xiong et al. showed that the applying Layer Norm to the inputs before the block performed significantly better (called Pre-Layer Normalization: $x + \text{Block}(\text{LayerNorm}(x))$) compared to the original method mentioned above (called Post-Layer Normalization: $\text{LayerNorm}(x + \text{Block}(x))$) [XYH+20]. The two methods can be seen below in Figure 4.

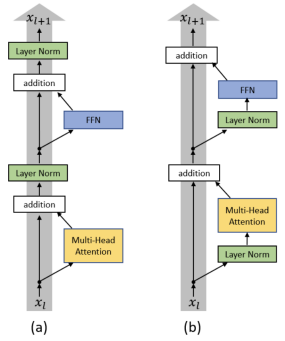


Figure 4: (a) Post-LN Transformer layer; (b) Pre-LN Transformer layer

5 Experimentation

5.1 French-English Translation using RNN and Attention

We will be implementing French-English Translation using Recurrent Neural Network with Attention, based off the original Neural Machine Translation with Attention paper [Bah14]. We based our experiments from the official PyTorch tutorial for Seq2Seq Translation [Rob17]. For the RNN, we used Gated Recurrent Unit (GRU), a specific type of RNNs

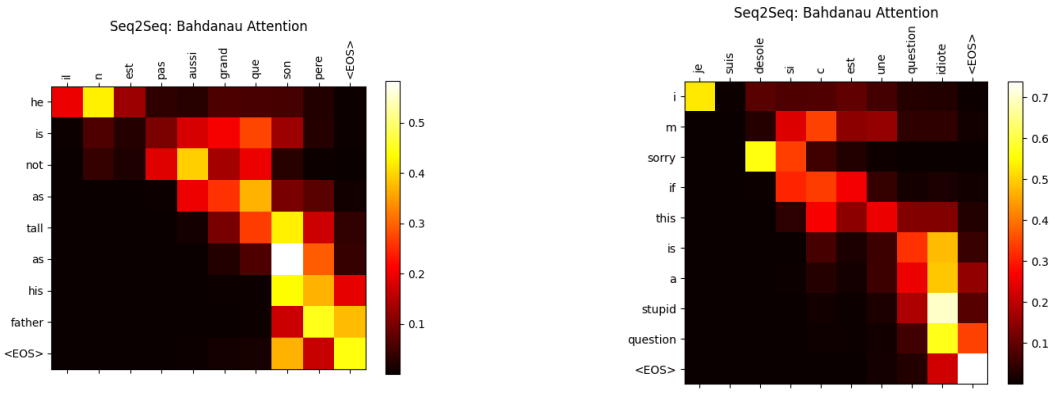


Figure 5: Bahdanau Attention Weights

that uses a gating mechanism for keeping and forgetting certain features. For the Attention mechanism, we implemented Additive Attention, also known as Bahdanau Attention.

We used the subset of the French-English dataset, specifically sentences that began with the following patterns: "I am", "He is", "She is", and so on. This subset contained about 10000 data points, which we used to train the RNN + Attention model. Figure 5 displays the attention weights between the input and output sentences, which showcases how their dependencies are modeled. Below are some sample translations, where $>$ is the input sentence, $=$ is the correct output sentence, and $<$ is the model output sentence.

```
> elles vont le tenter
= they re going to try
< they re going to try <EOS>

> je suis mecontent
= i m unhappy
< i m unhappy about anything <EOS>

> il est confronte a de nombreuses difficultes
= he is facing many difficulties
< he is confronted by many difficulties <EOS>
```

5.2 Generating Shakespeare using Transformer

We will be implementing nanoGPT, a small-scale variant of GPT-2 (Generative Pretrained Transformer). One key point is that the Transformers used in recent Large Language Models, including GPT, only uses the decoder. The original Transformer was made for machine translation, which is why it required the encoder to understand the input sentence. Text generation such as in LLMs do not require the encoder to understand the input, and instead they simply use the decoder to predict the next word successively.

We trained a character-level nanoGPT (therefore we predict the next character each iteration) using a dataset of Shakespeare text of $\sim$1M characters. The model is the decoder part of the original Transformer, but replacing Post-Layer Normalization with Pre-Layer Normalization. Below displays the text generated by nanoGPT after training on 5000 batches, each containing 16 sentences. Although it is mostly gibberish, it somewhat resembles the writing style of Shakespeare.

```
HENRY BOLINGBOKE: Oxawn; year hearthing, my shrate?
KING EDWARD IV: Dill-may, he ruckpiouse: then and leGoding live ower turn.
YORK:
MENENBUM: At the stolj Dill the make, if miscrech? My such please I creat is for cannoble! When Babe gotle he's foolse of i !
by that o subj!
SAULET: Day hels, with recoluty hour carror me?
Kind: Yet have gull to your this youth fathers.
AUTOLYCUS: Yet which resing mes Clifful faim livings
MENENA: Hy you ar mell thou do did when sless; But while here goiss; he' mades? my crown diy will this be: a well the wonth
Will Jewill is boy: a lever! And hold unjustious your budded men? A mine of pery more.
```

```
GRONWARD: Ay, to, brengely.
```

6 Discussion and Conclusion

We had minimal experience with transformers before embarking on this project so our biggest challenge was the reading process. It was difficult to find sources with relevant content. When we had found sources, it was hard to conceptualize the content. We faced this challenge by first looking at resources that described transformers and attention at a higher level. Some examples of resources we consulted first were video lectures and encyclopedia articles. Looking at the topic at a bird's eye view helped us to grasp the language presented in scholarly papers.

A more specific challenge was trying to connect the content about attention with the information we had about RNNs. The attention mechanism is similar to the RNN architecture. In attention, input vectors modify each other and in RNNs, each output depends on the last hidden state which depends on the previous output. However, these two functions have their differences. Attention provides a direct way for any input vector to modify any other vector. In RNNs, the next output depends implicitly on all of its previous outputs. However, it is hard to identify how each of the prior outputs affect the next output. As a result, we chose to explain the two topics separately. To connect attention with RNNs we decided to discuss how RNNs can use the attention mechanism to improve its unbounded memory.

Lastly, the timeline for this project was limited to a month since it was a final project for a class. Specifically, we only had a few weeks to complete the research process and synthesize the knowledge we had gained. As a result, we had to be intentional about pacing ourselves throughout the duration of the project. We settled into different roles to ensure steady progress over time. Takuto had a high-level understanding of transformers beforehand. He constructed a list of subtopics that he felt were interesting to explore. This outline really guided our research process! After we had each gone our own way and learned about our subtopic, Takuto helped us integrate all our loose ends into a cohesive project. Wenhao took on a "managing role" throughout the project. He made sure that we kept communicating with each other even if we fell behind. This helped us to stay on the same page and make progress bit by bit. He also coordinated and led our meetings so we always knew what we needed to do next. John settled into a documentation role. He recorded the progress the team had made, summarized the results, and shared it with the group. This helped the group think about what topics to discuss in the project. He also led the construction of the poster and figured out how it ought to be displayed.

Looking into the future, John, who did his research on attention, wonders about the connection between attention and neural networks. One way to think about attention is in terms of a neural network where the outputs vectors are the next layer of neurons. With this view in mind we can think about attention weights as dynamically generated. This is different from the weights in neural networks which are learned but fixed. John would like to look into other ways dynamically generated weights have been incorporated into neural networks. Another idea that intrigued John was how attention can be viewed as soft dictionary lookup. For example, a query vector is answered by every key vector whether strong or weak. John would like to explore how attention can be used to accomplish dictionary lookup in different ways.

Wenhao, who did his research on RNNs, seeks to learn more about the application of attention in other models. Attention has helped fix one of the largest flaws of RNNs, and transformers address one of the flaws in RNNs with attention. Wenhao would like to look into how attention can be used or adapted to address issues in other generative models and improve them.

Takuto, who did his research on the Transformer architecture, wants to investigate methods for making them more efficient. The main advantage of Transformer is its parallelizable computation, and Takuto wants to explore how its computation has evolved over time, such as Flash Attention and KV Cache, and hopefully discover new approach for efficient Transformers.

7 Acknowledgments

We would like to thank our professor Markos Katsoulakis for meeting with us to discuss our proposal and giving constructive comments on our poster. He made sure the scope of the project was wide enough to be substantive but narrow enough to be accomplished in the allotted time frame.

We would also like to thank 3Brown1Blue (Grant Sanderson) for his Youtube series on Neural Networks, especially the Attention and Transformer episode. His visualizations were one of the first resources we consulted to learn about the material, and helped us consolidate the ideas and intuitions behind these architectures.

Lastly, we would like to thank the reader for taking the time to learn about transformers. We hope that you enjoyed reading this report.

References

[Bah14] Dzmitry Bahdanau. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[HZRS16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[Mur22] Kevin P Murphy. *Probabilistic machine learning: an introduction*. MIT press, 2022.

[Rob17] Sean Robertson. NLP From Scratch: Translation with a Sequence to Sequence Network and Attention, 2017. Accessed: 2024-12-18, `https://pytorch.org/tutorials/intermediate/seq2seq_translation_tutorial.html`.

[Vas17] A Vaswani. Attention Is All You Need. *Advances in Neural Information Processing Systems*, 2017.

[XYH+20] Ruibin Xiong, Yunchang Yang, Di He, Kai Zheng, Shuxin Zheng, Chen Xing, Huishuai Zhang, Yanyan Lan, Liwei Wang, and Tieyan Liu. On Layer Normalization in the Transformer Architecture. In *International Conference on Machine Learning*, pages 10524–10533. PMLR, 2020.

[ZLLS23] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. *Dive into Deep Learning*. Cambridge University Press, 2023. `https://D2L.ai`.