inzva DLSG

Transformers

Week9 & Week10

1 Basic of Transformers and Importance of NLP

1.1 Developing Transformers and Basics

1.1.1 What is Transformers

Transformer is a revolutionary deep learning architecture significantly impacting natural language processing (NLP). Introduced by Google researchers in 2017 in the paper titled ”Attention is All You Need,” [1]. After introducing this model, many successful models such as BERT, GPT, T5 have been developed. These models have shown superior performance in many NLP tasks such as language understanding, translation, and summarization.

1.1.2 The Development of Transformer

Traditionally, models used in Natural Language Processing (NLP) relied heavily on recurrent neural networks (RNNs) and long short-term memory (LSTM) models. However, these models were often ineffective in managing dependencies over long texts and had limited parallel processing capabilities. The Transformer model, by utilizing the attention mechanism, addresses these shortcomings by focusing on each input element independently, thus better managing long-distance dependencies. Additionally, its parallel processing capability enables rapid training on large datasets.

1.2 Advantages of Transformer Over RNN and LSTM Models

1.3 Recurrent Neural Networks (RNN)

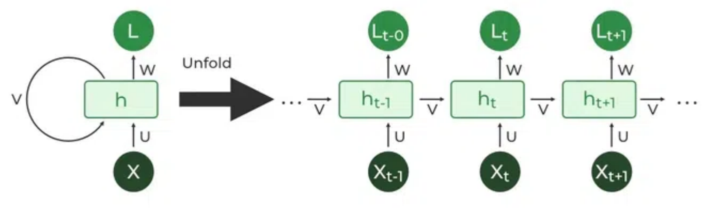

Figure 1: Rnn Architecture

RNNs are neural networks capable of predicting future steps using information from previous steps in sequential data. The fundamental building block of RNNs is cells that process the input along with the hidden state and pass it to the next time step.
Figure 1, h_t is the hidden state, x_t is the input, y_t is the output, W_h, W_x, W_y are weight matrices, and b_h, b_y are bias terms. However, RNNs may struggle to carry information over time, which creates issues in learning long-term dependencies. This problem is known as the ”vanishing gradient” problem.

1.4 Long Short-Term Memory (LSTM)

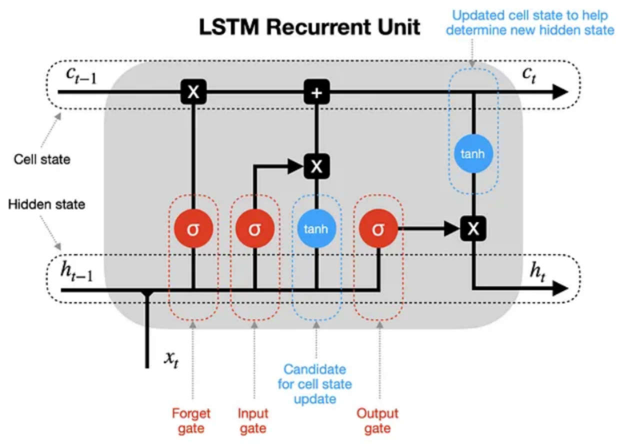

Figure 2: Lstm Architecture

LSTMs are a type of RNN developed to solve the problem of learning long-term dependencies. LSTM cells contain gates that control the flow of information, such as the input gate, forget gate, and output gate.

Figure 2, f_t is the forget gate, i_t is the input gate, $\tilde{C}_t$ is the cell state, C_t is the updated cell state, o_t is the output gate, and σ is the sigmoid function. These gates allow for more effective learning of long-term dependencies. However, LSTMs can also be computationally inefficient for large datasets and very long sequences.

1.5 Transformer Model

The Transformer model was developed to overcome the limitations of RNN and LSTM models. The Transformer takes a completely different approach to processing sequential data and uses the attention mechanism.

1.5.1 Attention Mechanism

Transformer replaces the previous sequential processing structure with the attention mechanism. This mechanism allows the model to focus attention on specific parts of each word during processing. As a result, it can better understand the context of each word and effectively manage long-distance dependencies.

1.5.2 Parallel Processing Capability

Sequential models like RNNs and LSTMs need to complete previous steps before moving to the next, making parallel processing challenging and slowing down training on large datasets. In contrast, the Transformer model can independently process inputs in parallel due to its attention mechanism, speeding up the training process and enhancing scalability.

1.5.3 Handling Long-Distance Dependencies

In RNNs and LSTM models, gradient problems can occur over time, making it difficult to manage dependencies over long texts or documents effectively. The Transformer model overcomes this issue by efficiently processing long-distance dependencies, allowing it to successfully perform comprehensive NLP tasks.

1.6 Conclusion

Transformer has significantly advanced NLP compared to traditional RNN and LSTM-based models. Its attention mechanism, parallel processing capability, scalability, and effective management of long-distance dependencies are fundamental to its success.

In this section, we have explored the fundamentals of Transformer, its development, and detailed advantages over traditional RNN and LSTM models. Don't worry if there are intimidating terms, we will examine the entire transformer architecture in detail in the next sections.

Let's explore together the power of Transformers.

Figure 3: power of Transformers

2 Attention is All We Need

Before diving into the Transformers architecture, let's understand the main feature that distinguishes this architecture from others, as we mentioned above: the self-attention mechanism.

2.1 Self Attention

2.1.1 Intuiton

Let's start with a creative example. Imagine we have the sentence: **"Dragons breathe fire."** In this sentence, we need to pay attention to other words to understand each word's context. For instance, to correctly understand the word "breathe," we need to consider the words "Dragons" and "fire."

Why Self-Attention? Traditional models like RNNs and LSTMs are designed to capture sequential dependencies but can struggle with long sequences. Specifically, understanding the relationship between the first and last words of a sentence can

Figure 4: Dragon breathes fire.

be challenging. The self-attention mechanism solves this problem by computing the relationships between all words. In other words, while calculating the context of each word, it considers all the other words in the sentence.

2.1.2 More Technical Example

In the self-attention mechanism, three vectors are calculated for each word: Query (Q), Key (K), and Value (V). These vectors are used to determine the relationships between words.

Example: Let's take the sentence "Dragons breathe fire." For each word in this sentence, we will calculate the Q, K, and V vectors.

Let's define the initial embedding vectors as follows:

- "Dragons": [1, 0, 0]
- "breathe": [0, 1, 0]
- "fire": [0, 0, 1]

We will calculate the Q, K, and V vectors from these embedding vectors:

$$Q = W_q X$$
$$K = W_k X$$
$$V = W_v X$$

Here, X is the embedding vector of the word, and W_q, W_k, and W_v are learnable weight matrices. Let's keep the weight matrices simple:

$$W_q = W_k = W_v = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Thus, the Q, K, and V vectors are:

- "Dragon":
  - Q: [1, 0, 0]
  - K: [1, 0, 0]
  - V: [1, 0, 0]
- "breathes":
  - Q: [0, 1, 0]
  - K: [0, 1, 0]
  - V: [0, 1, 0]
- "fire":
  - Q: [0, 0, 1]
  - K: [0, 0, 1]
  - V: [0, 0, 1]

Now, let's calculate the self-attention scores for the word "breathes."

1. Calculating the Scores: First, we calculate the scores by multiplying the Q vector of "breathes" with the K vectors of other words:

$$\text{score}_{breathe,dragon} = Q_{breathes} \cdot K^T_{dragons} = [0,1,0] \cdot [1,0,0]^T = 0$$
$$\text{score}_{breathes,breathes} = Q_{breathes} \cdot K^T_{breathes} = [0,1,0] \cdot [0,1,0]^T = 1$$
$$\text{score}_{breathes,fire} = Q_{breathes} \cdot K^T_{fire} = [0,1,0] \cdot [0,0,1]^T = 0$$

Thus, the scores are:

$$\text{scores} = [0,1,0]$$

2. Scaling the Scores:

$$\text{scaled scores} = \frac{\text{scores}}{\sqrt{d_k}}$$

Here, $d_k = 3$:

$$\text{scaled scores} = \frac{[0,1,0]}{\sqrt{3}} = [0, \frac{1}{\sqrt{3}}, 0]$$

3. Applying Softmax to the Scores:

$$\text{attention weights} = \text{softmax(scaled scores)}$$

Applying softmax:

$$\text{attention weights} = \text{softmax}([0, \frac{1}{\sqrt{3}}, 0]) = [0.2119, 0.5762, 0.2119]$$

4. Weighting and Summing the Values: Finally, we obtain the updated vector for "breathe" by weighting the value vectors with the attention weights:

$$\text{output}_{breathes} = 0.2119 \cdot V_{dragon} + 0.5762 \cdot V_{breathe} + 0.2119 \cdot V_{fire}$$
$$\text{output}_{breathes} = 0.2119 \cdot [1,0,0] + 0.5762 \cdot [0,1,0] + 0.2119 \cdot [0,0,1]$$
$$\text{output}_{breathes} = [0.2119,0,0] + [0,0.5762,0] + [0,0,0.2119]$$
$$\text{output}_{breathes} = [0.2119, 0.5762, 0.2119]$$

As a result of these steps, we obtain the updated embedding vector for the word "breathe." This vector becomes more meaningful by considering the relationships with the other words.

2.1.3 Representing Vectors with Parallel Matrices

Instead of performing these operations for each word individually, we can make the computations more efficient by performing them in parallel using matrices. When we construct the Q, K, and V matrices for all words:

- Q matrix:
$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$
- K matrix:
$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$
- V matrix:
$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Following these steps to calculate self-attention scores:

1. Calculating the Scores:

$$\text{scores} = QK^T$$

$$\text{scores} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}^T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

2. Scaling the Scores:

$$\text{scaled scores} = \frac{\text{scores}}{\sqrt{d_k}}$$

Here, $d_k = 3$:

$$\text{scaled scores} = \begin{bmatrix} \frac{1}{\sqrt{3}} & 0 & 0 \\ 0 & \frac{1}{\sqrt{3}} & 0 \\ 0 & 0 & \frac{1}{\sqrt{3}} \end{bmatrix}$$

3. Applying Softmax to the Scores:

$$\text{attention weights} = \text{softmax}(\text{scaled scores})$$

Applying softmax to each row, since all rows and columns are the same, the softmax results:

$$\text{attention weights} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

4. Weighting and Summing the Values:

$$\text{output} = \text{attention weights} \cdot V$$

$$\text{output} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

These steps allow us to determine the context of each word by considering its relationship with other words. For example, calculating how much attention "breathe" should pay to "Dragons" and "fire."

By performing these operations in parallel on matrices, we save time and increase the efficiency of the calculations. This way, we can apply the self-attention mechanism quickly and effectively even on long sentences.

2.2 Multi-Head Attention

Figure 5: Dragons breathe fire.

Intuition

Why Multi-Head Attention? While self-attention allows a model to focus on various parts of a sentence, multi-head attention enhances this capability by employing multiple attention mechanisms (or "heads") simultaneously. This enables the model to grasp different dimensions of the relationships between words, such as syntactic structure or semantic meaning.

How Multi-Head Attention Works

Multi-head attention involves executing multiple self-attention mechanisms simultaneously. Each head operates with its own set of learned weights, offering a unique perspective on the input data. The outputs from all heads are then concatenated and transformed to generate the final result.

Detailed Steps

Let's break down the process using a simple example.

- **Linear Projections:** For each head, we apply a linear transformation to the input embeddings to get the query (Q), key (K), and value (V) matrices.

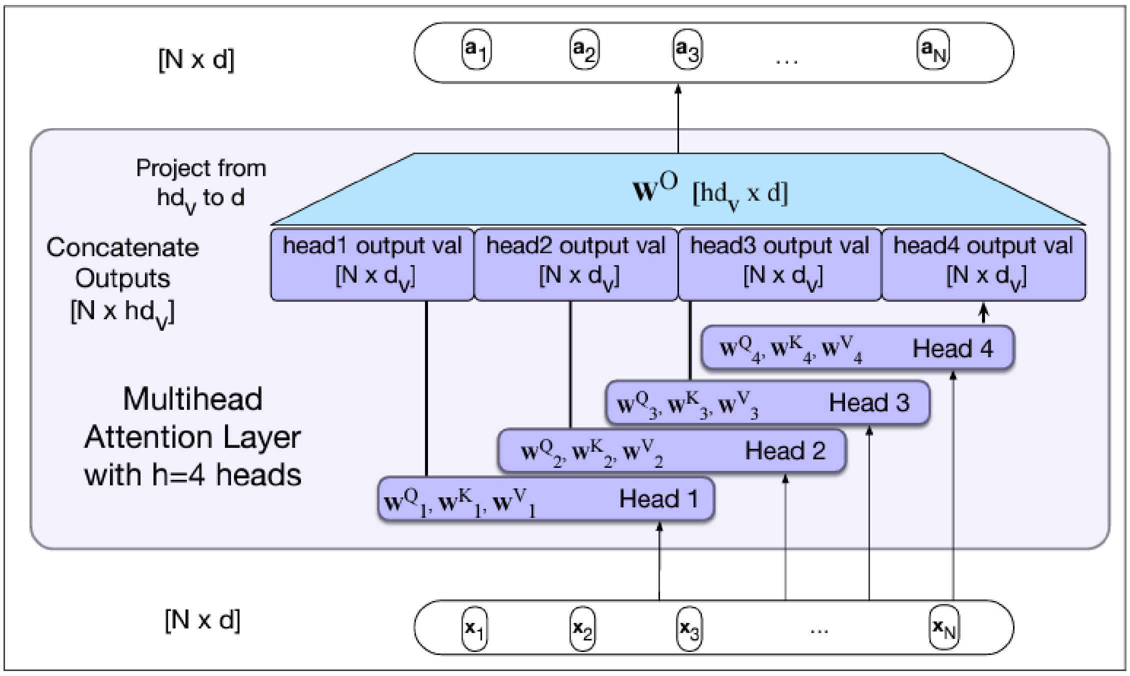
[N x d]
a1
a2
a3
...
aN
Project from hdV to d
W^O [hdV x d]
Concatenate Outputs [N x hdV]
head1 output val [N x dV]
head2 output val [N x dV]
head3 output val [N x dV]
head4 output val [N x dV]
WQ4, WK4, WV4 Head 4
WQ3, WK3, WV3 Head 3
WQ2, WK2, WV2 Head 2
WQ1, WK1, WV1 Head 1
Multihead Attention Layer with h=4 heads
[N x d]
x1
x2
x3
...
xN

Figure 6: Multi-head self-attention: Each of the multi-head self-attention layers is provided with its own set of key, query, and value weight matrices. The outputs from each of the layers are concatenated and then projected to **d**, thus producing an output of the same size as the input so the attention can be followed by layer norm and feed-forward, and layers can be stacked.

$$Q_i = XW_{q_i}, \quad K_i = XW_{k_i}, \quad V_i = XW_{v_i}$$

Here, W_{q_i}, W_{k_i}, and W_{v_i} are the weight matrices for the i-th head.

- **Scaled Dot-Product Attention:** Each head performs scaled dot-product attention as described earlier.

$$\text{Attention}(Q_i, K_i, V_i) = \text{softmax}\left(\frac{Q_i K_i^T}{\sqrt{d_k}}\right) V_i$$

- **Concatenation:** The outputs of all the heads are concatenated.

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \text{head}_2, \ldots, \text{head}_h)W_O$$

Here, W_O is the weight matrix that projects the concatenated outputs back to the original dimensionality.

Before moving on to the example, we will share a table that explains what we have said so far in a much more descriptive way. We hope it will be more understandable this way.

Example

Consider the modified example sentence: ”Dragons breathe fire. They are mythical creatures.”

- Let's assume we have 2 heads for simplicity. Each heads represent one sentence.
- Each head will have its own set of Q, K, and V matrices.

For the first head:

$$Q_1 = XW_{q_1}, \quad K_1 = XW_{k_1}, \quad V_1 = XW_{v_1}$$

For the second head:

$$Q_2 = XW_{q_2}, \quad K_2 = XW_{k_2}, \quad V_2 = XW_{v_2}$$

Each head performs self-attention:

$$\text{head}_1 = \text{Attention}(Q_1, K_1, V_1)$$

$$\text{head}_2 = \text{Attention}(Q_2, K_2, V_2)$$

The outputs are then concatenated:

$$\text{MultiHead} = \text{Concat}(\text{head}_1, \text{head}_2)W_O$$

This allows the model to consider different aspects of the input sentence simultaneously, leading to a richer representation.

3 Tranformers Architecture

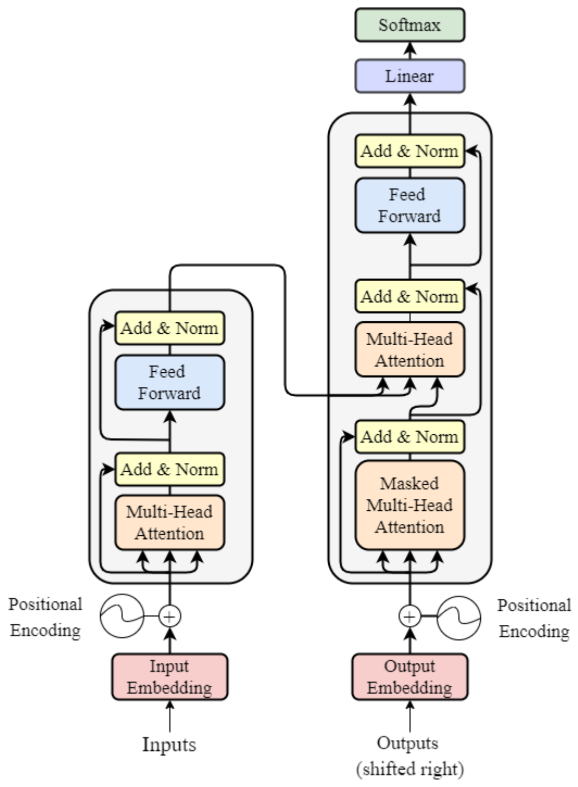

Figure 7: Architecture of Transformers

So far, we have learned about the basic components of the transformer architecture, including multi-head attention and feed-forward layers. However, there are two key components that we have not yet covered in detail: positional encoding and residual connections (Add & Norm layers). Let's dive into these components to understand their roles and how they contribute to the overall architecture.

Positional Encoding

Why Positional Encoding? Transformers do not have a built-in sense of the order of words. Unlike RNNs, which process sequences in order, transformers process all words simultaneously. This means we need a way to inject some information about the position of words into the model. Positional encoding provides this information.

How Does It Work?

Positional encoding adds unique information to each word based on its position in the sentence. This allows the model to differentiate between the words based on their positions. The positional encodings are vectors of the same dimension as the embeddings and are added to the embeddings.

The formulas for calculating positional encodings are as follows:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d}}}\right)$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d}}}\right)$$

Here, pos is the position, i is the dimension, and d is the total dimension of the positional encoding.

Example

Consider the sentence "Dragons breathe fire." Suppose our embeddings and positional encodings are both of dimension 4.

- Embedding for "Dragons": $[0.1, 0.2, 0.3, 0.4]$
- Positional Encoding for position 1: $[0.0, 0.84, 0.91, 0.54]$
- Combined: $[0.1 + 0.0, 0.2 + 0.84, 0.3 + 0.91, 0.4 + 0.54] = [0.1, 1.04, 1.21, 0.94]$

By adding these positional encodings, the model now has information about the word "Dragons" and its position in the sentence.

Residual Connections and Layer Normalization

Why Residual Connections?

Training deep neural networks can be challenging due to problems like vanishing gradients. Residual connections help by providing a shortcut path for the gradients to flow, making it easier to train deep networks.

How Does It Work?

In a transformer, residual connections are added around each sub-layer (like the multi-head attention and feed-forward layers). This means that the input to each sub-layer is added to its output before passing it to the next layer. This helps in preserving the original information and improving gradient flow.

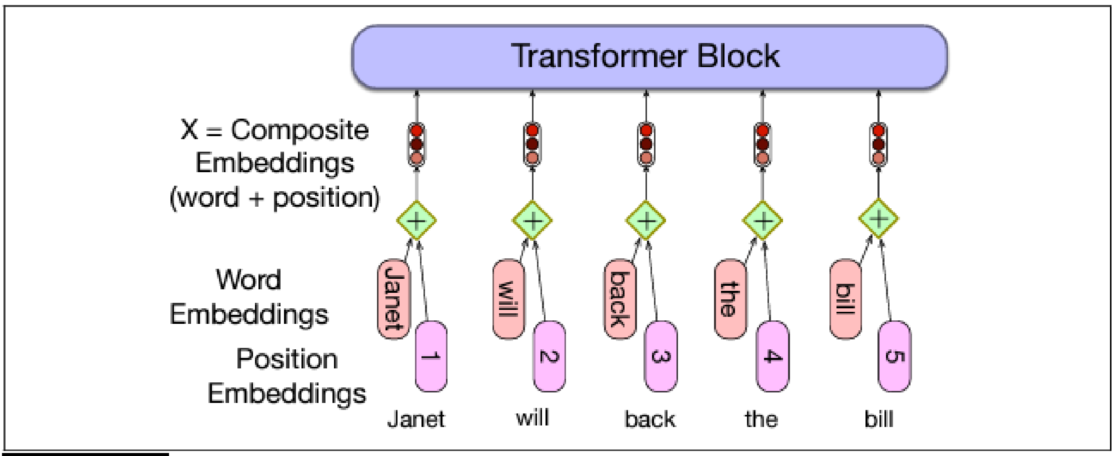

Figure 8: A simple way to model position: add an embedding of the absolute position to the token embedding to produce a new embedding of the same dimenionality

Layer Normalization

After the addition, layer normalization is applied. This normalizes the output to have zero mean and unit variance, stabilizing and speeding up training.

Example

Let's take the output of a multi-head attention layer and see how residual connections and layer normalization are applied.

- Multi-Head Attention Output: $\text{Output}_{MHA} = [0.5, 0.6, 0.4, 0.3]$
- Add Input (Residual Connection): Suppose the input to the multi-head attention was $[0.1, 0.2, 0.3, 0.4]$.

$$\text{Added Output} = \text{Output}_{MHA} + \text{Input} = [0.5, 0.6, 0.4, 0.3] + [0.1, 0.2, 0.3, 0.4] = [0.6, 0.8, 0.7, 0.7]$$

- Layer Normalization: Applying layer normalization to the added output:

  To normalize, we calculate the mean and variance of $[0.6, 0.8, 0.7, 0.7]$:

$$\text{mean} = \frac{0.6 + 0.8 + 0.7 + 0.7}{4} = 0.7$$

$$\text{variance} = \frac{(0.6 - 0.7)^2 + (0.8 - 0.7)^2 + (0.7 - 0.7)^2 + (0.7 - 0.7)^2}{4} = 0.005$$

$$\text{Normalized Output} = \frac{[0.6, 0.8, 0.7, 0.7] - 0.7}{\sqrt{0.005}} = [-1.41, 1.41, 0, 0]$$

By applying residual connections and layer normalization, the model stabilizes and accelerates the training process. More simplify representation:

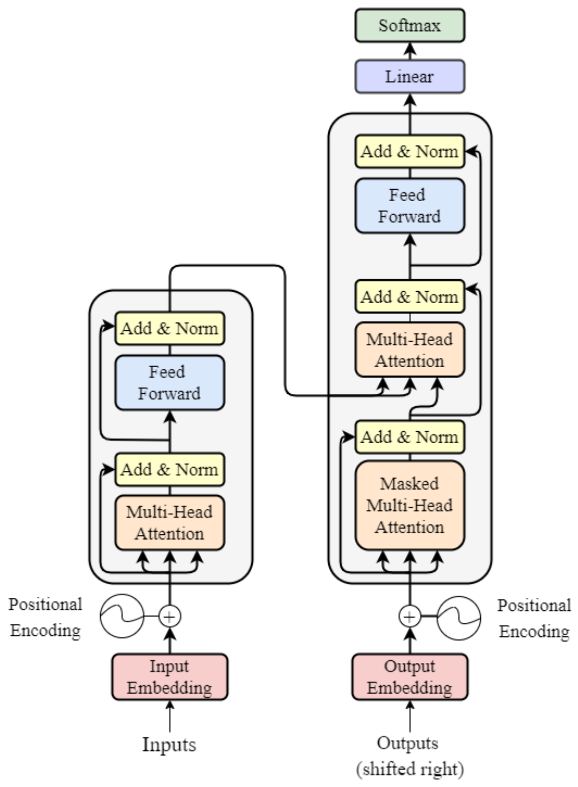

Figure 9: Architecture of Transformers

4 Tranformers Architecture

So far, we have learned about the basic components of the transformer architecture, including multi-head attention and feed-forward layers. However, there are two key components that we have not yet covered in detail: positional encoding and residual connections (Add & Norm layers). Let's dive into these components to understand their roles and how they contribute to the overall architecture.

Positional Encoding

Why Positional Encoding? Transformers do not have a built-in sense of the order of words. Unlike RNNs, which process sequences in order, transformers process all words simultaneously. This means we need a way to inject some information about the position of words into the model. Positional encoding provides this information.

How Does It Work?

Positional encoding adds unique information to each word based on its position in the sentence. This allows the model to differentiate between the words based on their positions. The positional encodings are vectors of the same dimension as the embeddings and are added to the embeddings.

The formulas for calculating positional encodings are as follows:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d}}}\right)$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d}}}\right)$$

Here, *pos* is the position, i is the dimension, and d is the total dimension of the positional encoding.

Example

Consider the sentence "Dragons breathe fire." Suppose our embeddings and positional encodings are both of dimension 4.

- Embedding for "Dragons": $[0.1, 0.2, 0.3, 0.4]$
- Positional Encoding for position 1: $[0.0, 0.84, 0.91, 0.54]$
- Combined: $[0.1 + 0.0, 0.2 + 0.84, 0.3 + 0.91, 0.4 + 0.54] = [0.1, 1.04, 1.21, 0.94]$

By adding these positional encodings, the model now has information about the word "Dragons" and its position in the sentence.

Residual Connections and Layer Normalization

Why Residual Connections?

Training deep neural networks can be challenging due to problems like vanishing gradients. Residual connections help by providing a shortcut path for the gradients to flow, making it easier to train deep networks.

How Does It Work?

In a transformer, residual connections are added around each sub-layer (like the multi-head attention and feed-forward layers). This means that the input to each sub-layer is added to its output before passing it to the next layer. This helps in preserving the original information and improving gradient flow.

Layer Normalization

After the addition, layer normalization is applied. This normalizes the output to have zero mean and unit variance, stabilizing and speeding up training.

Example

Let's take the output of a multi-head attention layer and see how residual connections and layer normalization are applied.

- Multi-Head Attention Output: $\text{Output}_{MHA} = [0.5, 0.6, 0.4, 0.3]$
- Add Input (Residual Connection): Suppose the input to the multi-head attention was $[0.1, 0.2, 0.3, 0.4]$.

$$\text{Added Output} = \text{Output}_{MHA} + \text{Input} = [0.5, 0.6, 0.4, 0.3] + [0.1, 0.2, 0.3, 0.4] = [0.6, 0.8, 0.7, 0.7]$$

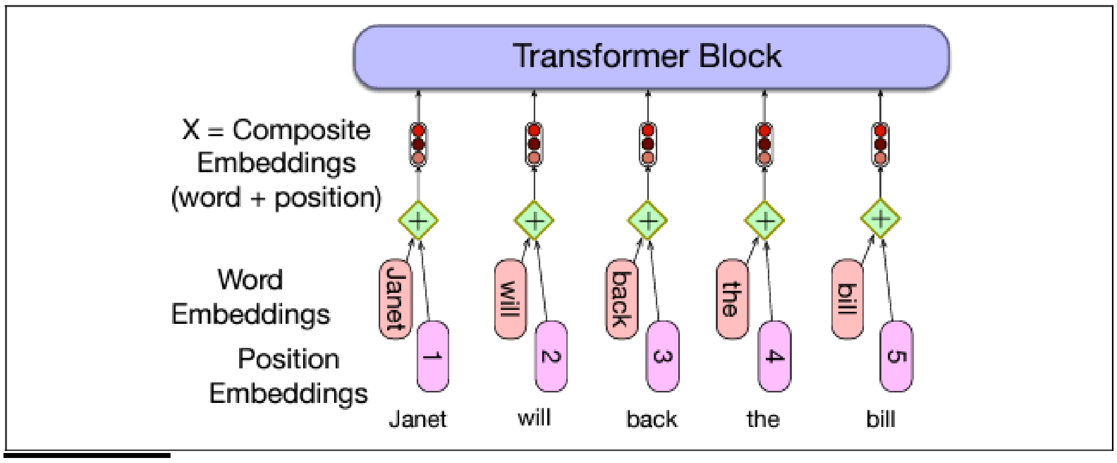

Figure 10: A simple way to model position: add an embedding of the absolute position to the token embedding to produce a new embedding of the same dimenionality

- Layer Normalization: Applying layer normalization to the added output:

  To normalize, we calculate the mean and variance of $[0.6, 0.8, 0.7, 0.7]$:

$$\text{mean} = \frac{0.6 + 0.8 + 0.7 + 0.7}{4} = 0.7$$

$$\text{variance} = \frac{(0.6 - 0.7)^2 + (0.8 - 0.7)^2 + (0.7 - 0.7)^2 + (0.7 - 0.7)^2}{4} = 0.005$$

$$\text{Normalized Output} = \frac{[0.6, 0.8, 0.7, 0.7] - 0.7}{\sqrt{0.005}} = [-1.41, 1.41, 0, 0]$$

By applying residual connections and layer normalization, the model stabilizes and accelerates the training process. More simplify representation:

5 Transformer Block

Although the encoder-decoder layer is introduced together in the base architecture of our model, at this stage we will only introduce the encoder part for the sake of simplicity. The architecture will be mentioned again in the next section. Transformer blocks are multi-layered structures that process input vectors to produce richer contextual representations. The fundamental components of a transformer block include:

1. **Self-Attention Layer**
2. **Feedforward Network**
3. **Residual Connections**
4. **Layer Normalization**

1. Self-Attention Layer

The self-attention layer computes the relationships between each word and all other words in the input sequence to generate contextual representations. The process involves the following steps:

- **Calculation of Query (Q), Key (K), and Value (V) Vectors:** Query, key, and value vectors are computed for each word. These vectors are used to determine the relationships between words.

  $$Q = XW^Q, \quad K = XW^K, \quad V = XW^V$$

  where X represents the input vectors, and W^Q, W^K, W^V are learnable weight matrices.

- **Calculation of Attention Scores:** The attention scores are computed by taking the dot product of the query vector of each word with the key vectors of all other words.

  $$\text{score}(Q_i, K_j) = Q_i \cdot K_j^T$$

- **Scaling of Scores:** The computed scores are scaled to stabilize the softmax application.

  $$\text{scaled\_score}(Q_i, K_j) = \frac{Q_i \cdot K_j^T}{\sqrt{d_k}}$$

  where d_k is the dimension of the query and key vectors.

- **Softmax Application:** The scaled scores are normalized using the softmax function to obtain attention weights.

  $$\alpha_{ij} = \text{softmax}(\text{scaled\_score}(Q_i, K_j))$$

- **Weighted Sum of Values:** The value vectors are weighted by the attention weights and summed to produce the output vector.

  $$\text{output}_i = \sum_j \alpha_{ij} V_j$$

2. Feedforward Network

The output of the self-attention layer is passed through a two-layer feedforward network that operates independently on each position. **output as x input**:

- **First Layer:**

  $$\text{FFN}_1(x) = \text{ReLU}(xW_1 + b_1)$$

  where W_1 and b_1 are learnable weight matrix and bias term.

- **Second Layer:**

  $$\text{FFN}_2(x) = xW_2 + b_2$$

  where W_2 and b_2 are learnable weight matrix and bias term.

3. Residual Connections

Residual connections help to prevent the loss of information and facilitate learning by adding the input vector to the output vector of each sub-layer before passing it to the next layer.

$$\text{output}_{\text{residual}} = x + \text{layer\_output}$$

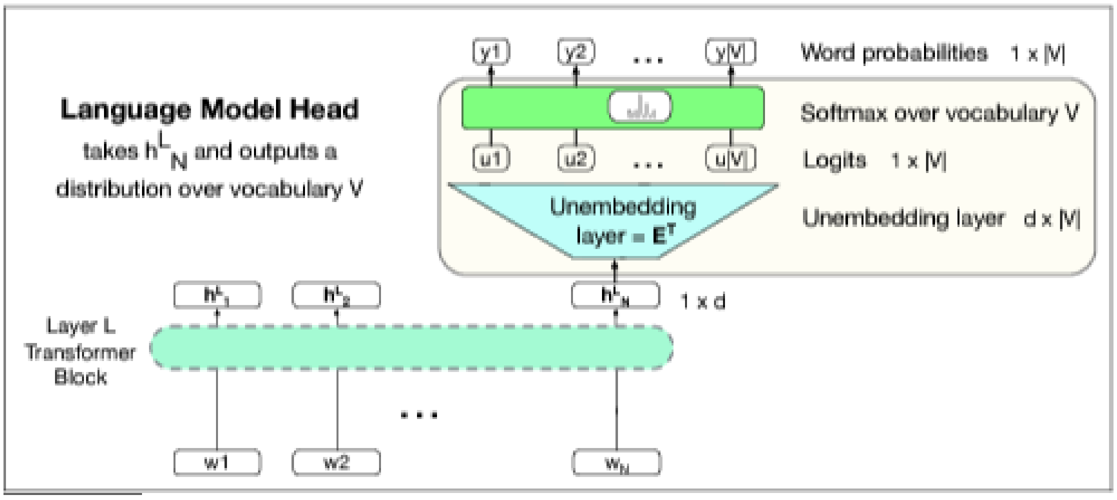
Language Model Head
takes h^L_N and outputs a distribution over vocabulary V
y1
y2
...
y|V|
Word probabilities 1 x |V|
Softmax over vocabulary V
u1
u2
...
u|V|
Logits 1 x |V|
Unembedding layer = E^T
Unembedding layer d x |V|
h^L_1
h^L_2
h^L_N
1 x d
Layer L Transformer Block
w1
w2
w_N

Figure 11: The language modeling head: the circuit at the top of a transformer that maps from the output embedding for token N from the last transformer layer (hL N) to a probability distribution over words in the vocabulary V.

4. Layer Normalization

The output of the residual connections is normalized to stabilize and accelerate the training process.

$$\text{normalized\_output} = \frac{\text{output}_{\text{residual}} - \mu}{\sigma}$$

where μ and σ are the mean and standard deviation of the output vectors.

Step-by-Step Operation of a Transformer Block

1. **Input vectors (X) are taken.**
2. **Query (Q), Key (K), and Value (V) vectors are computed.**
3. **Attention scores are calculated and scaled.**
4. **Softmax is applied to obtain attention weights.**
5. **Value vectors are weighted and summed to produce output vectors.**
6. **Outputs are passed through the feedforward network.**
7. **Residual connections are added, and layer normalization is applied.**

These steps illustrate how transformer blocks process information from input to output and create contextual representations for each word.

5.1 Next Token Prediction

One of the primary functions of transformer models is to predict the next word in a given text sequence. This process allows the model to effectively perform tasks such as text completion and language modeling by utilizing its language understanding and generation capabilities.

$$P(\text{NextToken} \mid \text{PreviousTokens}) = \text{softmax}(W_{\text{proj}} \cdot \text{DecoderOutput})$$

This process involves the model calculating the probability of the next word at each step and determining which word is most likely during text generation.

Language Modeling Head

The language modeling head is a critical component in transformer models for tasks such as text generation and language modeling. It takes the final layer's output and converts it into a probability distribution to predict the next word.

Process Details

1. **Extracting the Final Layer Output:** The output of the last token from the transformer's final layer (usually h_N) is taken. This output vector contains contextual information and is a d-dimensional vector.

$$h_N \in \mathbb{R}^d$$

2. **Calculating the Logit Vector:** The final token's output is passed through a linear layer to compute the logit vector. The logit vector contains a score for each word in the vocabulary ($|V|$).

$$u = h_N W^O$$

   Here, W^O is the weight matrix of the linear layer.

3. **Applying Softmax:** The logit vector is normalized using the softmax function to obtain a probability distribution. This distribution helps the model predict the likelihood of each word being the next word.

$$y = \text{softmax}(u)$$

4. **Word Selection:** A word is selected from the generated probability distribution. This word is used as the next word in the text generation process.

Weight Tying

The language modeling head often shares weights with the embedding matrix (weight tying). This allows the model to use the same weight matrix for both word embeddings and word predictions, reducing the number of parameters and simplifying learning.

$$u = h_N E^T$$

Here, E is the embedding matrix, and E^T is its transpose.

Applications of the Language Modeling Head

The language modeling head enables transformer models to be effectively used in tasks such as text completion, summarization, machine translation, and question answering. This head allows the model to use contextual information to accurately predict words and generate meaningful texts.

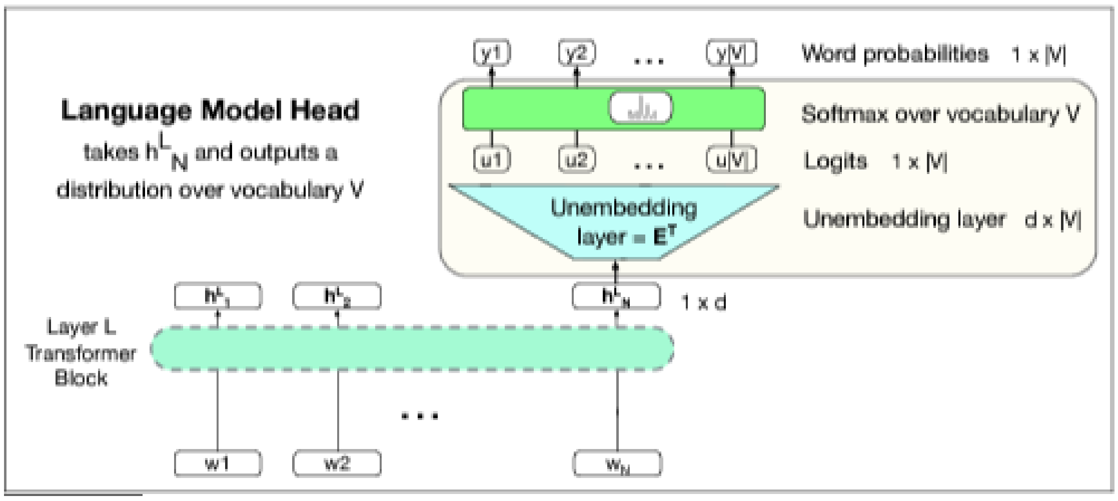
Language Model Head
takes h^L_N and outputs a distribution over vocabulary V
y1
y2
...
y|V|
Word probabilities 1 x |V|
Softmax over vocabulary V
u1
u2
...
u|V|
Logits 1 x |V|
Unembedding layer = E^T
Unembedding layer d x |V|
h^L_1
h^L_2
h^L_N
1 x d
Layer L Transformer Block
w1
w2
w_N

Figure 12: The language modeling head: the circuit at the top of a transformer that maps from the output embedding for token N from the last transformer layer (hL N) to a probability distribution over words in the vocabulary V.

Text Completion

In text completion tasks, the model predicts the continuation of a given text prefix. The language modeling head generates the most likely continuation based on the given prefix.

Prefix Text: ”Harry looked around the dimly lit room, noticing the shadow of a figure”

Generated Text: ”lurking near the old, dusty bookshelf.”

5.1.1 Question Answering

In question answering tasks, the model provides answers to questions based on text context. The language modeling head predicts the most likely answer based on the context of the question.

Question: ”Who gave Harry the Marauder’s Map in his third year at Hogwarts?”

Answer: ”Fred and George Weasley”

5.2 Conclusion

The language modeling head enables transformer models to perform effectively across various NLP tasks. This head allows the model to use contextual information to accurately predict words and generate meaningful texts. Transformer models, with the language modeling head, successfully execute text generation processes, showcasing superior performance in language processing tasks.

6 Next Token Prediction

One of the primary functions of transformer models is to predict the next word in a given text sequence. This process allows the model to effectively perform tasks such as text completion and language modeling by utilizing its language understanding and generation capabilities.

$$P(\text{NextToken} \mid \text{PreviousTokens}) = \text{softmax}(W_{\text{proj}} \cdot \text{DecoderOutput})$$

This process involves the model calculating the probability of the next word at each step and determining which word is most likely during text generation.

Language Modeling Head

The language modeling head is a critical component in transformer models for tasks such as text generation and language modeling. It takes the final layer's output and converts it into a probability distribution to predict the next word.

Process Details

1. **Extracting the Final Layer Output:** The output of the last token from the transformer's final layer (usually h_N) is taken. This output vector contains contextual information and is a d-dimensional vector.

   $$h_N \in \mathbb{R}^d$$

2. **Calculating the Logit Vector:** The final token's output is passed through a linear layer to compute the logit vector. The logit vector contains a score for each word in the vocabulary ($|V|$).

   $$u = h_N W^O$$

   Here, W^O is the weight matrix of the linear layer.

3. **Applying Softmax:** The logit vector is normalized using the softmax function to obtain a probability distribution. This distribution helps the model predict the likelihood of each word being the next word.

   $$y = \text{softmax}(u)$$

4. **Word Selection:** A word is selected from the generated probability distribution. This word is used as the next word in the text generation process.

Weight Tying

The language modeling head often shares weights with the embedding matrix (weight tying). This allows the model to use the same weight matrix for both word embeddings and word predictions, reducing the number of parameters and simplifying learning.

$$u = h_N E^T$$

Here, E is the embedding matrix, and E^T is its transpose.

Applications of the Language Modeling Head

The language modeling head enables transformer models to be effectively used in tasks such as text completion, summarization, machine translation, and question answering. This head allows the model to use contextual information to accurately predict words and generate meaningful texts.

Text Completion

In text completion tasks, the model predicts the continuation of a given text prefix. The language modeling head generates the most likely continuation based on the given prefix.

Prefix Text: ”Harry looked around the dimly lit room, noticing the shadow of a figure”

Generated Text: ”lurking near the old, dusty bookshelf.”

Question Answering

In question answering tasks, the model provides answers to questions based on text context. The language modeling head predicts the most likely answer based on the context of the question.

Question: ”Who gave Harry the Marauder’s Map in his third year at Hogwarts?”

Answer: ”Fred and George Weasley”

Conclusion

The language modeling head enables transformer models to perform effectively across various NLP tasks. This head allows the model to use contextual information to accurately predict words and generate meaningful texts. Transformer models, with the language modeling head, successfully execute text generation processes, showcasing superior performance in language processing tasks.

Encoder-Decoder Architecture in Transformers

The encoder-decoder architecture is fundamental to the functioning of transformer models, especially in tasks involving sequence-to-sequence learning such as machine translation. This architecture allows the model to convert an input sequence into a different output sequence, which can be in a different language or format.

Encoder

The encoder’s primary job is to process the input sequence and create a set of continuous representations (or embeddings) that encapsulate the information present in the input. The encoder is composed of multiple identical layers, each containing two main components: a multi-head self-attention mechanism and a position-wise feedforward neural network.

1. Input Embedding and Positional Encoding

- Input tokens are first converted into dense vectors (embeddings).
- Positional encodings are added to these embeddings to incorporate the order of tokens.

$$X_{\text{input}} = [E[w_1] + PE[1], E[w_2] + PE[2], \ldots, E[w_N] + PE[N]]$$

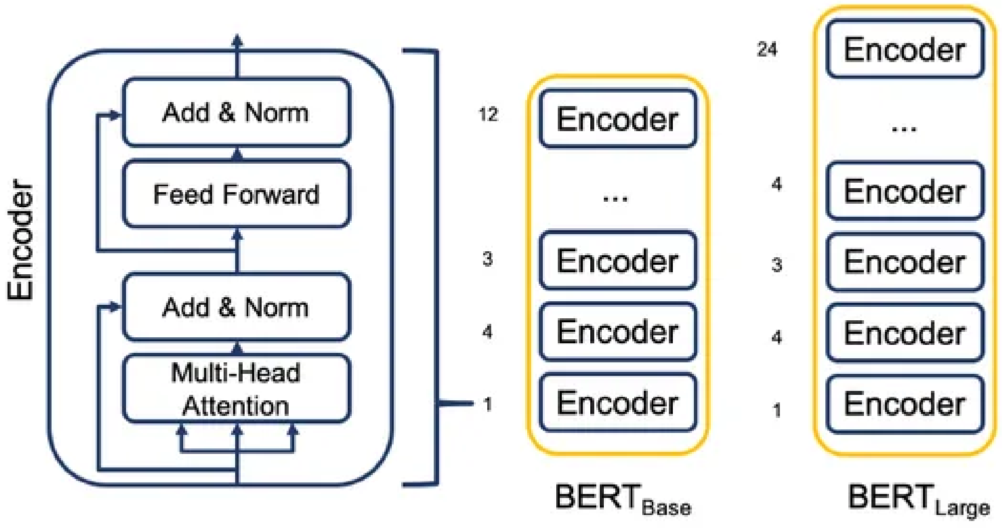

Figure 13: Encoder Architecture

2. Self-Attention Mechanism

The self-attention mechanism allows each token to focus on other tokens in the sequence, enabling the model to capture contextual relationships.

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

Here, Q, K, and V are the query, key, and value matrices, which are derived from the input embeddings.

3. Feedforward Neural Network

The output of the self-attention layer is passed through a feedforward neural network, which consists of two linear transformations with a ReLU activation in between.

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

4. Residual Connections and Layer Normalization

Residual connections are used around each sub-layer (attention and feedforward) followed by layer normalization to stabilize the training process.

$$\text{output}_{\text{residual}} = \text{LayerNorm}(x + \text{Sublayer}(x))$$

The output of the encoder is a set of context-rich representations of the input sequence, which are then passed to the decoder.

Encoder-Based Language Models

Encoder-based language models are designed to create rich contextual embeddings for input sequences, which can be used for various natural language processing (NLP) tasks such as text classification, named entity recognition, and sentiment analysis. Some of the most well-known encoder-based language models include BERT, RoBERTa, and DistilBERT.

BERT (Bidirectional Encoder Representations from Transformers)

BERT, developed by Google, is a transformer-based model that uses a bidirectional approach to understand the context of a word based on its surroundings. It is pre-trained on a large corpus of text in a self-supervised manner, using two main tasks: masked language modeling (MLM) and next sentence prediction (NSP).

- **Masked Language Modeling (MLM):** Randomly masks some of the tokens in the input and then predicts those masked tokens based on the context provided by the other tokens in the sequence.
- **Next Sentence Prediction (NSP):** Predicts whether a given sentence is likely to follow a previous sentence, enabling the model to understand sentence relationships.

BERT has achieved state-of-the-art results on many NLP benchmarks and has been fine-tuned for a wide range of specific NLP tasks.

RoBERTa (Robustly Optimized BERT Pretraining Approach)

RoBERTa, developed by Facebook, is an optimized version of BERT. It improves upon BERT by modifying key hyperparameters and training the model on a larger dataset for a longer period. RoBERTa removes the next sentence prediction objective and focuses solely on the masked language modeling task. These changes lead to significant performance improvements on various NLP tasks.

DistilBERT

DistilBERT, developed by Hugging Face, is a smaller, faster, and lighter version of BERT. It uses a technique called knowledge distillation, where a smaller model (student) is trained to replicate the behavior of a larger model (teacher). DistilBERT retains 97% of BERT's performance while being 60% faster and 40% smaller in size. This makes it more suitable for deployment in resource-constrained environments.

Applications of Encoder-Based Language Models

Encoder-based models like BERT, RoBERTa, and DistilBERT are widely used in various NLP applications, including:

- **Text Classification:** Categorizing text into predefined categories.
- **Named Entity Recognition (NER):** Identifying and classifying entities (e.g., names of people, organizations) in text.
- **Question Answering:** Answering questions based on a given context.
- **Sentiment Analysis:** Determining the sentiment expressed in a piece of text.

These models have significantly advanced the field of NLP by providing powerful tools for understanding and generating human language.

Decoder

The decoder's job is to generate the output sequence one token at a time. The decoder is also composed of multiple identical layers, each containing three main components: a masked multi-head self-attention mechanism, a multi-head attention mechanism over the encoder's output, and a position-wise feedforward neural network.

1. Input Embedding and Positional Encoding

- The previously generated tokens in the output sequence are embedded and added to positional encodings.

$$Y_{\text{input}} = [E[w_1] + PE[1], E[w_2] + PE[2], \ldots, E[w_M] + PE[M]]$$

2. Masked Self-Attention Mechanism

The masked self-attention mechanism ensures that the predictions for the current position depend only on the known outputs at previous positions.

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

Here, the mask prevents attending to future tokens.

3. Encoder-Decoder Attention Mechanism

The second attention mechanism allows the decoder to focus on relevant parts of the input sequence by attending to the encoder's output.

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

In this case, K and V are the encoder's output representations, and Q comes from the previous decoder layer.

4. Feedforward Neural Network

Similar to the encoder, the output from the attention layer is passed through a feedforward neural network.

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

5. Residual Connections and Layer Normalization

Residual connections and layer normalization are applied similarly to the encoder.

$$\text{output}_{\text{residual}} = \text{LayerNorm}(x + \text{Sublayer}(x))$$

6. Final Linear and Softmax Layer

The final output is transformed into logits for each token in the vocabulary, and a softmax layer converts these logits into probabilities.

$$P(\text{token}) = \text{softmax}(W_{\text{proj}} \cdot \text{output}_{\text{decoder}})$$

Decoder-Based Language Models

Decoder-based language models are designed to generate text by predicting the next word in a sequence given the previous words. These models are particularly effective for tasks such as text generation, translation, and summarization. Some of the most well-known decoder-based language models include GPT, GPT-2, and GPT-3.

GPT (Generative Pre-trained Transformer)

GPT, developed by OpenAI, is a transformer-based model that uses a unidirectional (left-to-right) approach to generate text. It is pre-trained on a large corpus of text in an unsupervised manner, using a language modeling objective. This means that the model learns to predict the next word in a sentence based on the previous words.

- **Unidirectional Language Modeling:** The model generates text by predicting the next word in a sequence, using only the context of the previous words.
- **Pre-training and Fine-tuning:** GPT is pre-trained on a large dataset and can be fine-tuned on specific tasks to improve performance.

GPT has been shown to perform well on a variety of NLP tasks, including text completion, translation, and question answering.

GPT-2

GPT-2 is an improved version of GPT, with a larger model size and trained on more data. It demonstrates remarkable text generation capabilities, producing coherent and contextually relevant paragraphs of text.

- **Increased Model Size:** GPT-2 has 1.5 billion parameters, compared to the 110 million parameters of the original GPT.
- **Unsupervised Learning:** It is trained in an unsupervised manner, allowing it to generate text that is contextually rich and diverse.

GPT-2 has been widely used in applications such as chatbots, content creation, and automated summarization.

GPT-3

GPT-3, also developed by OpenAI, is the largest and most powerful version in the GPT series, with 175 billion parameters. It can generate highly realistic and contextually appropriate text, making it suitable for a wide range of applications.

- **Massive Scale:** GPT-3's large model size allows it to understand and generate complex text with high accuracy.
- **Few-Shot Learning:** GPT-3 can perform tasks with minimal task-specific training data, often referred to as "few-shot" learning.

GPT-3 has been used in numerous applications, from virtual assistants to creative writing and beyond.

Applications of Decoder-Based Language Models

Decoder-based models like GPT, GPT-2, and GPT-3 are widely used in various NLP applications, including:

- **Text Generation:** Creating coherent and contextually appropriate text based on a given prompt.
- **Machine Translation:** Translating text from one language to another.
- **Summarization:** Generating concise summaries of longer texts.
- **Dialogue Systems:** Powering chatbots and virtual assistants.

These models have significantly advanced the field of NLP by providing powerful tools for generating and understanding human language.

Encoder-Decoder Workflow

1. Encoding Phase

The input sequence is passed through the encoder, producing a set of context-rich representations.

$$\text{EncoderOutput} = \text{Encoder}(X_{\text{input}})$$

2. Decoding Phase

The decoder uses these encoder outputs along with the previously generated tokens to produce the next token in the sequence.

$$\text{DecoderOutput} = \text{Decoder}(Y_{\text{input}}, \text{EncoderOutput})$$

3. Prediction

The final decoder output is passed through a linear layer and softmax to produce a probability distribution over the vocabulary for the next token.

$$\text{NextToken} \sim \text{softmax}(W_{\text{proj}} \cdot \text{DecoderOutput})$$

This process repeats until the entire output sequence is generated.

Conclusion

The encoder-decoder architecture in transformers is powerful for sequence-to-sequence tasks. The encoder processes the input sequence into context-rich representations, and the decoder generates the output sequence one token at a time, attending to the encoder's output to maintain contextual consistency. This architecture, combined with the self-attention mechanism, allows transformers to handle complex dependencies and generate high-quality translations, summaries, and other sequence-based outputs.

Encoder-Decoder-Based Language Models

Encoder-decoder-based language models are designed for tasks that involve transforming an input sequence into an output sequence, such as machine translation, summarization, and question answering. This architecture leverages both an encoder to process the input sequence and a decoder to generate the output sequence. Some of the most well-known encoder-decoder-based language models include the original Transformer model, T5, and BART.

Transformer

The original Transformer model, introduced by Vaswani et al. in the paper "Attention is All You Need," is a pioneering encoder-decoder model. It uses self-attention mechanisms in both the encoder and decoder to handle long-range dependencies and allows for parallel processing, significantly improving the efficiency of training.

- **Self-Attention:** The model uses self-attention in both the encoder and decoder to understand the context and relationships between words.
- **Parallel Processing:** Unlike RNNs, the Transformer can process all tokens in the sequence simultaneously, making it highly efficient.

The original Transformer model laid the foundation for many subsequent advancements in NLP and has been widely adopted for tasks like machine translation and text summarization.

T5 (Text-To-Text Transfer Transformer)

T5, developed by Google, is an encoder-decoder model that treats every NLP task as a text-to-text problem. It converts the input into a text sequence and generates the output as another text sequence, allowing the same model architecture to be applied across diverse NLP tasks.

- **Unified Framework:** T5 uses a consistent text-to-text framework for all tasks, making it versatile and easy to fine-tune on specific tasks.
- **Pre-trained and Fine-tuned:** T5 is pre-trained on a large corpus and can be fine-tuned on specific datasets to achieve state-of-the-art performance on various NLP benchmarks.

T5 has been used for tasks such as translation, summarization, and question answering, demonstrating its flexibility and power.

BART (Bidirectional and Auto-Regressive Transformers)

BART, developed by Facebook, is an encoder-decoder model that combines the strengths of bidirectional and autoregressive transformers. It is trained by corrupting text and then reconstructing the original text, making it effective for text generation and comprehension tasks.

- **Denoising Autoencoder:** BART is trained as a denoising autoencoder, where it corrupts the input text and learns to reconstruct it, making it robust and effective for various text generation tasks.

- **Versatile Applications:** BART is used for a range of tasks, including summarization, translation, and question answering.

BART has shown strong performance on multiple NLP benchmarks, showcasing its effectiveness in both understanding and generating text.

Applications of Encoder-Decoder-Based Language Models

Encoder-decoder models like the original Transformer, T5, and BART are widely used in various NLP applications, including:

- **Machine Translation:** Converting text from one language to another.
- **Summarization:** Generating concise summaries of longer texts.
- **Question Answering:** Providing answers to questions based on a given context.
- **Text Generation:** Creating coherent and contextually appropriate text.

These models have significantly advanced the field of NLP by providing powerful tools for transforming and generating human language.

7 Addition

The performance of llm has shown to be 3 factors:

- model size (the number of parameters not counting embeddings)
- dataset size
- amount of computer used for training

References

[1] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. *Advances in neural information processing systems 30* (2017).