Understanding Encoders and Decoders

Minor in AI - IIT ROPAR

26th March, 2025

From Sequential Listening to Smart Attention

Imagine you're at a family gathering. Everyone is having conversations in small groups, some talking about food, some about politics, and some kids are just playing around. Now imagine your curious younger cousin pulls you aside and asks, "Hey, can you tell me what everyone here is talking about?"

To do that, you'd have to observe each group, pay attention to who is saying what, and try to understand which conversations are related, which ones are just noise, and who is influencing whom. You're not just listening to people in a sequence, you're attending to everyone, all at once, figuring out which voices are more relevant, and which ones matter less in understanding the full picture.

This is exactly what **transformers** do in language processing.

Earlier, machines used to read sentences word by word — like listening to people talk in a long line, one by one. But now, transformers let machines do what you just did: attend to everyone at once, understand relevance, and pick out the important parts.

Let's now jump into this technical yet beautiful journey.

The Journey Begins: The Limitation of Sequential Models

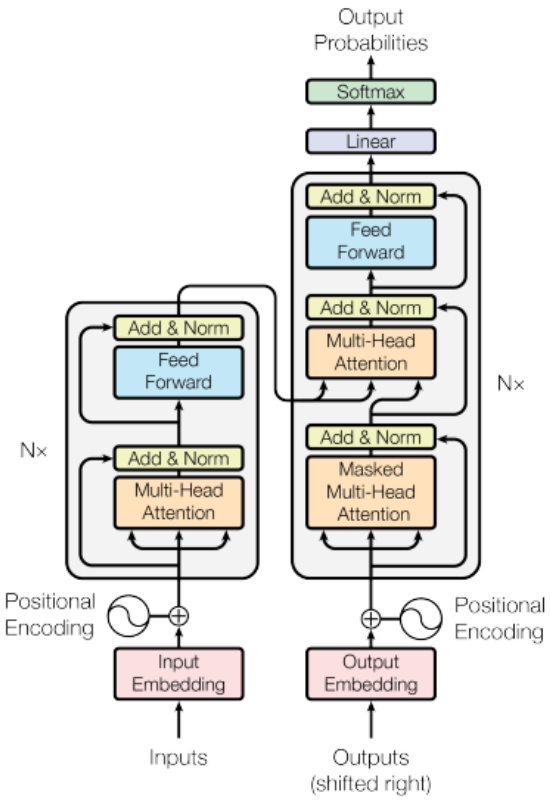
Previously, models like RNNs and LSTMs were used to process text. They worked well for many tasks but had a key limitation — they processed input **sequentially**:

- The model had to read a sentence word by word.
- This made training slow and hard to parallelize.
- They struggled to remember long-term dependencies.

This is where the **Transformer architecture** (Vaswani et al., 2017) came in and changed the game.

What Makes Transformers Special?

Transformers allow us to look at all words in a sentence **simultaneously**, and figure out their **relationships**. This is done using the **attention mechanism**, specifically, **self-attention**.



The Trinity: Query, Key, and Value Vectors

Each word is first embedded into a vector. Then for each word, we derive:

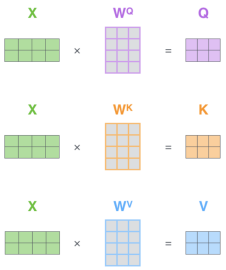
- **Query (Q):** What this word is looking for.
- **Key (K):** What this word has to offer.
- **Value (V):** What this word carries as content.

These are derived using learned matrices:

$$Q = X \cdot W_Q, \quad K = X \cdot W_K, \quad V = X \cdot W_V$$

where X is the input matrix of embeddings.

Example: If $X \in \mathbb{R}^{4\times 8}$ and $W_Q \in \mathbb{R}^{8\times 5}$, then $Q \in \mathbb{R}^{4\times 5}$.



Self-Attention Mechanism: How Words Understand Each Other

Let us now dive deep into the concept of self-attention, which lies at the heart of the Transformer architecture. The main goal of self-attention is to allow each word in a sentence to look at, or attend to, other words in the same sentence and decide how much importance it should give them when computing its own contextual representation.

Suppose we are focusing on a particular word, say the third word in a sentence. Its query vector is denoted as Q_3. We want to evaluate how relevant all other words in the sentence are to this word. To achieve this, we compare Q_3 with the key vectors of all other words, i.e., $K_1, K_2, K_3, \ldots, K_n$. This is accomplished through the following steps:

Step 1: Compute Similarity (Dot Product + Scaling)

To assess the relatedness between words, we compute the dot product between the query vector Q_i (for example, Q_3) and each key vector K_j. The dot product acts as a similarity measure — a higher value indicates a stronger alignment or relevance between the two vectors.

$$\text{score}_{i,j} = \frac{Q_i \cdot K_j^T}{\sqrt{d_k}}$$

Here, d_k is the dimensionality of the key vectors. The division by $\sqrt{d_k}$ is known as *scaling*, which is a crucial step. Without scaling, the dot product values can become large when d_k is high, which may push the softmax (next step) into regions with small gradients, hindering the learning process. Scaling helps maintain stable gradients and efficient learning.

After this computation, for each word i, we obtain a list of raw similarity scores with all other words j in the sentence.

Step 2: Convert Similarities into Probabilities (Softmax)

Once similarity scores are obtained for a given query Q_i, we convert them into attention weights using the softmax function:

$$\alpha_{i,j} = \text{softmax}(\text{score}_{i,j})$$

The softmax operation performs two important functions:

- It converts the raw similarity scores into values between 0 and 1.
- It ensures that the resulting values sum to 1, thereby forming a probability distribution.

These probabilities indicate how much attention the i-th word should pay to the j-th word. If $\alpha_{i,3}$ is large, it implies that word i finds word 3 highly relevant.

For instance, in the sentence "The animal didn't cross the street because it was too tired," the word "it" (at position i) is more semantically linked to "the animal" than to "the street." Thus, the attention weight $\alpha_{i,j}$ assigned to "the animal" would be higher.

Step 3: Generate the New Word Representation (Weighted Sum of Values)

Having determined how much each word should be attended to, we use the attention weights $\alpha_{i,j}$ to compute a weighted average of the value vectors:

$$\text{Attention}(Q_i, K, V) = \sum_j \alpha_{i,j} V_j$$

Each value vector V_j holds the actual content or representation of the word at position j. By summing these vectors weighted by their attention scores $\alpha_{i,j}$, we generate a new, context-sensitive representation for word i.

This final vector for word i is known as a *context-aware embedding*. It incorporates information not only about the word itself but also about other words in the sentence, weighted by their importance or relevance.

Repeat for All Words

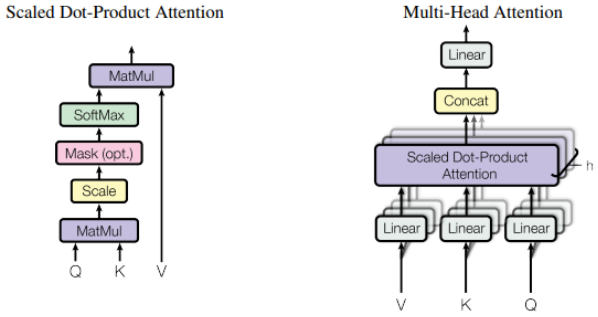
This process is carried out for every word in the sentence. The result is that each word ends up with a new, enriched vector that reflects its meaning in the context of the entire sentence.

Example: If $Q, K \in \mathbb{R}^{4\times5}$, then $QK^T \in \mathbb{R}^{4\times4}$, and multiplying with $V \in \mathbb{R}^{4\times6}$ gives output $\in \mathbb{R}^{4\times6}$.

Why Multiple Heads? Multi-Head Attention (MHA)

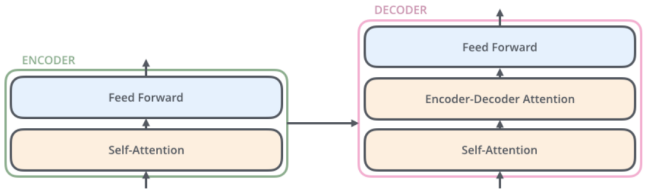
Each head learns different relationships.

- Create h heads, each with its own W_Q^i, W_K^i, W_V^i.
- For each head:
$$\text{head}_i = \text{Attention}(QW_Q^i, KW_K^i, VW_V^i)$$
- Concatenate all heads:
$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O$$



Example: If 8 heads each output 64-dimensional vectors, the result is 512-dimensional before projection.

Putting It All Together: Encoder-Decoder Structure



Encoder: Understanding the Input

The Transformer architecture is structured as an Encoder-Decoder model, particularly useful in tasks like machine translation, where an input sentence (e.g., in English) is transformed into an output sentence (e.g., in French).

The encoder is responsible for analyzing and understanding the input sentence. It consists of multiple identical layers (typically 6 in the original Transformer), and each layer has two sub-layers:

- Multi-Head Self-Attention Layer
- Feed-Forward Neural Network (FFN)

Each of these sub-layers is surrounded by:

- Residual (Skip) Connections
- Layer Normalization

Let the input sentence be represented as a sequence of tokens:

$$X = \{x_1, x_2, \ldots, x_n\}$$

Each token is embedded and positionally encoded:

$$E = \text{PositionalEncoding}(\text{Embedding}(X)) \in \mathbb{R}^{n \times d_{\text{model}}}$$

1. Multi-Head Self-Attention

We compute the query, key, and value matrices:

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

The self-attention mechanism:

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

Multiple heads are computed and concatenated:

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

Each head:

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

2. Feed-Forward Network (FFN)

The FFN is applied to each position independently:

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

3. Residual Connection + Layer Normalization

Each sub-layer is wrapped as:

$$\text{LayerNorm}(x + \text{Sublayer}(x))$$

So, the final encoder output is:

$$Z = \text{Encoder}(X) \in \mathbb{R}^{n \times d_{\text{model}}}$$

Decoder: Generating the Output

The decoder generates the output sequence one token at a time, using both the encoder output and previously generated tokens. Each decoder layer has:

- Masked Multi-Head Self-Attention
- Cross-Attention (Encoder-Decoder Attention)
- Feed-Forward Network (FFN)

Let the output sequence be:

$$Y = \{y_1, y_2, \ldots, y_t\}$$

1. Masked Multi-Head Self-Attention

Decoder inputs are embedded and positionally encoded:

$$E_Y = \text{PositionalEncoding}(\text{Embedding}(Y)) \in \mathbb{R}^{t \times d_{\text{model}}}$$

Compute:

$$Q = E_Y W^Q, \quad K = E_Y W^K, \quad V = E_Y W^V$$

Masked attention:

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

2. Cross-Attention (Encoder-Decoder Attention)

The decoder attends to the encoder output Z:

$$\text{CrossAttention}(Q, K = Z, V = Z) = \text{softmax}\left(\frac{QZ^\top}{\sqrt{d_k}}\right) Z$$

3. Feed-Forward Network and Output

Like the encoder:

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

Final output probabilities:

$$P(y_t \mid y_1, \ldots, y_{t-1}, X) = \text{softmax}(\text{FFN}(\text{Decoder Output}))$$

Generation Process: A Loop

- Start with a special `<SOS>` token.
- Generate y_1.
- Feed y_1 back into the decoder to generate y_2.
- Continue until `<EOS>` token is generated.

The Decoder-Only Model: GPT

The GPT (Generative Pre-trained Transformer) model adopts only the decoder portion of the original Transformer architecture. Unlike encoder-decoder models that require the entire input sequence upfront (e.g., for translation), GPT is designed for autoregressive next-word prediction. This means that given a sequence of tokens, the model is trained to predict the next token in the sequence.

Example:

- Input: "The sun is"
- Output: "shining"

At the core of GPT lies a stack of transformer decoder blocks. Each block includes a masked multi-head self-attention mechanism followed by a position-wise feed-forward network, both wrapped with residual connections and layer normalization. The masking ensures that at each position, the model can only attend to the current and previous positions—never future ones—thereby preserving the autoregressive property.

The input tokens are first embedded into dense vectors and enriched with positional encodings to retain word order. These are then processed through the decoder layers. At each layer, the self-attention mechanism computes query, key, and value matrices using learned weight matrices W_Q, W_K, and W_V, enabling the model to determine which earlier tokens are most relevant for predicting the next word.

Once the final hidden states are computed, they are passed through a linear layer followed by a softmax to produce a probability distribution over the vocabulary for the next token.

Training:

- The model is trained using a cross-entropy loss between the predicted token probabilities and the true next token in the sequence.
- During this process, the parameters of the self-attention (i.e., W_Q, W_K, W_V), the output projection layers, and the feed-forward networks are updated via backpropagation.

The GPT model generates text by starting with a special token (e.g., `<BOS>`), predicting the first token, appending it to the input, and repeating the process until a stopping condition (like an `<EOS>` token) is met.

Enhancements: Layer Norm and Residual Connections

- **Residual Connections:** Add input to layer output.
- **Layer Normalization:** Normalize across layer to stabilize training.

Summary Table

Concept	Description
Q, K, V	Learned representations for each word (Query, Key, Value)
Self-Attention	Measures similarity between words in a sentence
Scaled Dot Product	$QK^T/\sqrt{d_k}$
Softmax	Normalizes attention scores
Multi-Head Attention	Uses multiple heads to learn diverse patterns
Encoder	Contextualizes input sentence
Decoder	Generates output sentence step by step
Residual + LN	Stabilize and preserve info through network
GPT	Decoder-only model for next-word prediction

Key Takeaways

- **Transformers revolutionized NLP** by enabling models to attend to all words in a sequence simultaneously using the self-attention mechanism, overcoming the limitations of sequential models like RNNs and LSTMs.
- **The core idea behind attention** is to allow each word to weigh the importance of all other words in the input, producing context-aware representations for every word.
- **Query, Key, and Value vectors** form the foundation of self-attention. Each word generates these vectors, and attention scores are calculated using scaled dot-product similarity between queries and keys.
- **Softmax is applied to attention scores** to convert them into a probability distribution, which is then used to compute a weighted sum of the value vectors—forming the new contextual embedding for each word.
- **Multi-Head Attention (MHA)** allows the model to capture multiple types of relationships between words by running multiple self-attention operations in parallel and concatenating their results.
- **The encoder-decoder architecture** is especially powerful for sequence-to-sequence tasks like machine translation, where the encoder processes the input and the decoder generates the output step by step.
- **GPT uses only the decoder block** of the Transformer, focusing on autoregressive generation by predicting the next token based on previously generated tokens.

- **Masking in GPT** ensures the model doesn't look ahead during training, preserving causality and enabling one-token-at-a-time generation.
- **Training Transformers involves cross-entropy loss**, and the model parameters—including attention weights and feed-forward layers—are updated using backpropagation.
- **Residual connections and layer normalization** are critical architectural components that help stabilize and speed up training in deep networks.