Transformer

1. Introduction (00:00:00)

- **What is a Transformer?**
  - A Transformer is a type of **neural network architecture** designed to handle **sequential data** like text. It was introduced in the paper **"Attention is All You Need"** by Vaswani et al. (2017).
  - Unlike older models like **RNNs** or **LSTMs**, Transformers don't process data sequentially (word by word). Instead, they process **all words in a sentence at once**, making them faster and more efficient.
- **Why Transformers?**
  - **Parallel Processing**: Transformers process all words in parallel, which speeds up training and inference.
  - **Self-Attention Mechanism**: This allows the model to focus on different parts of the sentence when processing each word, capturing **long-range dependencies** (e.g., understanding how words far apart in a sentence relate to each other).
  - **Scalability**: Transformers can handle **large datasets** and are highly scalable, making them suitable for modern NLP tasks.
- **Applications**:
  - Transformers are used in models like **BERT**, **GPT**, and **T5**, which have revolutionized NLP by achieving **state-of-the-art (SOTA)** results in tasks like:
    - **Machine Translation**: Translating text from one language to another (e.g., Google Translate).
    - **Text Summarization**: Generating concise summaries of long documents.
    - **Question Answering**: Answering questions based on a given context (e.g., ChatGPT).

2. What and Why Transformers (00:04:07)

- **Why Transformers?**
  - **Problem with RNNs/LSTMs**:
    - RNNs process data **sequentially** (word by word), which makes them slow and hard to scale.
    - They struggle with **long-range dependencies** (e.g., understanding how the first word in a sentence relates to the last word).
  - **Solution: Transformers**:
    - Transformers process **all words in parallel**, making them faster and more efficient.
    - They use **self-attention** to capture relationships between all words in a sentence, regardless of distance.
- **Key Features**:

    - **Self-Attention Mechanism**: Allows the model to focus on different parts of the sentence when processing each word.
    - **Parallel Processing**: All words are processed simultaneously, speeding up training and inference.
    - **Scalability**: Transformers can handle large datasets and are highly scalable.

3. Basic Architecture of Transformers (00:22:21)

- **Encoder-Decoder Structure**:
    - The Transformer consists of two main components: the **Encoder** and the **Decoder**.
    - **Encoder**: Processes the input sequence (e.g., a sentence) and generates a set of **representations** (annotations) that capture the context of each word in the sequence.
    - **Decoder**: Generates the output sequence (e.g., translated text) one word at a time, using the context provided by the encoder.
- **Key Components**:
    - **Self-Attention Mechanism**: Allows the model to weigh the importance of different words in the input sequence relative to each other.
    - **Feed-Forward Neural Network**: A simple neural network applied to each word independently after the self-attention mechanism.
    - **Positional Encoding**: Adds information about the position of each word in the sequence, since Transformers don't have a built-in notion of word order.

4. Self-Attention Architecture (00:36:29)

- **Self-Attention Mechanism**:
    - Self-attention allows the model to focus on different parts of the input sequence when processing each word.
    - **Process**:
        1. **Queries (Q)**, **Keys (K)**, and **Values (V)**: For each word, the model computes three vectors: a query, a key, and a value.
            - **Query (Q)**: Represents the word for which we are calculating attention.
            - **Key (K)**: Represents all the words in the sequence and is used to compare with the query.
            - **Value (V)**: Holds the actual information that will be aggregated to form the output.
        2. **Attention Scores**: The model computes the dot product between the query vector of the current word and the key vectors of all words in the sequence. This gives a score that indicates how much attention to give to each word relative to the current word.
        3. **Scaling**: The scores are scaled by dividing by the square root

of the dimension of the key vectors ($\sqrt{d_k}$). This prevents the dot product from becoming too large, which can lead to unstable gradients during training.
4. **Softmax**: Apply the softmax function to the scaled scores to get attention weights (probabilities that sum to 1).
5. **Weighted Sum of Values**: Multiply the attention weights by the value vectors and sum them up to get the final output for the current word.

- **Example**:
  - For the sentence "The cat sat", the self-attention mechanism computes how much attention to give to "The", "cat", and "sat" when processing each word.

5. Multi-Head Attention (01:38:32)

- **Multi-Head Attention**:
  - Instead of computing a single set of attention scores, the Transformer uses **multiple attention heads**. Each head learns to focus on different parts of the input sequence, allowing the model to capture different types of relationships between words.
  - **Process**:
    1. **Split into Heads**: The input embeddings are split into multiple heads, and each head computes its own set of queries, keys, and values.
    2. **Compute Attention**: Each head computes attention scores independently.
    3. **Concatenate**: The outputs from all heads are concatenated and then multiplied by a weight matrix to produce the final output of the multi-head attention layer.
- **Why Multi-Head?**
  - Multi-head attention allows the model to focus on different parts of the sequence simultaneously, capturing different types of relationships (e.g., syntactic, semantic).

6. Feed-Forward with Multi-Head Attention (01:48:46)

- **Feed-Forward Neural Network**:
  - After the multi-head attention layer, the output is passed through a **feed-forward neural network (FFN)**. This is a simple neural network applied to each word independently.
  - The FFN consists of two linear transformations with a ReLU activation in between:
    $\text{FFN}(x) = \text{ReLU}(xW1+b1)W2 + b2$
- **Purpose**:
  - The FFN adds **non-linearity** to the model and helps it learn more complex patterns in the data.

7. Positional Encoding (01:57:24)

- **Positional Encoding**
  - Transformers process tokens in parallel, so they lack a natural sense of order. Positional encoding provides this order information by adding a position-specific vector to each input embedding.
- 🔷 **Why Positional Encoding is Needed**
  - Unlike RNNs, which process words sequentially and inherently capture order, Transformers process all words in parallel.
  - Without positional information, the Transformer wouldn't know if "The cat chased the dog" is different from "The dog chased the cat."
- 🔷 **How Positional Encoding Works**
  - **Positional Encoding Vectors:**
    - A fixed-size vector is added to the input embeddings of each word to encode its position in the sequence.
    - The vector dimensions match the embedding size so that positional information integrates well with the input embeddings.
  - **Mathematical Formula:**
    Positional encoding is computed using sinusoidal functions (sine and cosine) because they provide a unique, smooth representation for each position and allow the model to generalize to longer sequences.

8. Layer Normalization in Transformers (02:27:27)

- Layer normalization is applied to stabilize and speed up training. It normalizes the activations of each layer to have a mean of 0 and a standard deviation of 1.

✅ **Purpose:**
✔ Prevents internal covariate shift (changing distribution of layer inputs).
✔ Ensures stable gradients during backpropagation.
✔ Helps the model converge faster and improves generalization.

9. Complete Encoder Architecture (03:01:03)

The Encoder processes input sequences and generates meaningful contextual representations. It consists of **N identical layers** (usually 6 or 12).

🔶 **Main Components:**

1. **Input Embeddings + Positional Encoding**
   - Input tokens are converted to embeddings.
   - Positional Encoding is added to capture word order.
2. **Multi-Head Self-Attention**
   - Computes attention scores using Query (Q), Key (K), and Value (V) matrices.
   - Multiple attention heads capture different aspects of the input.

3. **Add & Norm**
    - Output is added to input (residual connection) and passed through **Layer Normalization**.
4. **Feed-Forward Network (FFN)**
    - Two linear layers with a ReLU activation in between.
5. **Add & Norm**
    - Output of FFN is added to input and passed through Layer Normalization.

✅ Key Points:

- Encoder captures both local and long-range dependencies.
- Uses residual connections and normalization for stability and faster training.
- Outputs contextual embeddings for each token.

10. Decoder Plan of Action (03:30:56)

The Decoder generates output sequences based on the encoder's contextual representations. It consists of **N identical layers** (usually 6 or 12).

🔶 Main Components:

1. **Input Embeddings + Positional Encoding**
    - Output tokens are converted to embeddings.
    - Positional Encoding is added to capture word order.
2. **Masked Multi-Head Self-Attention**
    - Similar to encoder self-attention but masked to prevent attending to future tokens.
    - Ensures the model generates output step-by-step.
3. **Add & Norm**
    - Output is added to input (residual connection) and normalized using **Layer Normalization**.
4. **Encoder-Decoder Multi-Head Attention**
    - Queries come from the decoder's output.
    - Keys and values come from the encoder's output.
    - Helps the decoder align with the encoder's context.
5. **Add & Norm**
    - Output is added to input and normalized.
6. **Feed-Forward Network (FFN)**
    - Two linear layers with a ReLU activation in between.
7. **Add & Norm**
    - Output of FFN is added to input and normalized.

✅ Key Points:

- Decoder generates one token at a time based on previous tokens.
- Masked attention ensures proper autoregressive generation.
- Encoder-decoder attention aligns generated output with input context.

11. Linear and Softmax Layer (04:47:00)

The final step in the Transformer architecture is to convert the decoder's output into a probability distribution over the vocabulary.

🔶 **Steps:**

1. **Linear Layer**
   - The decoder's output (a vector) is passed through a **Linear Layer** (fully connected layer).
   - This transforms the output dimension to match the size of the vocabulary.
   - If the vocabulary size is V, the output will be a vector of size V.
2. **Softmax Layer**
   - Softmax converts the output vector into a probability distribution.
   - Each value represents the probability of a specific token in the vocabulary being the next token.

✅ **Key Points:**

- The token with the highest softmax value is selected as the next token.
- The model is trained using **cross-entropy loss** based on these probabilities.

=> Let's walk through the **entire scenario of a Transformer** from start to end using a **simple example**. We'll use the task of **machine translation** (translating a sentence from English to French) to explain how the Transformer works step-by-step.

Example Task: Machine Translation

- **Input Sentence (English)**: "The cat sat on the mat."
- **Output Sentence (French)**: "Le chat s'est assis sur le tapis."

Step 1: Input Embedding (00:22:21)

- **What Happens**:
  - Each word in the input sentence is converted into a **vector** (a list of numbers) using an **embedding layer**.
  - These vectors capture the meaning of the words in a numerical form.
- **Example**:
  - "The" → [0.2, 0.5, 0.1, ...]
  - "cat" → [0.7, 0.3, 0.9, ...]
  - "sat" → [0.4, 0.8, 0.2, ...]
  - (Each word is represented by a vector of fixed size, e.g., 512 dimensions.)

Step 2: Positional Encoding (01:57:24)

- **What Happens**:
  - Since Transformers process all words in parallel, they don't know the order of words in the sentence. To fix this, **positional**

encodings are added to the word embeddings.
   - These encodings are based on sine and cosine functions and provide information about the position of each word in the sentence.
- **Example**:
   - "The" (position 1) → Add positional encoding for position 1.
   - "cat" (position 2) → Add positional encoding for position 2.
   - "sat" (position 3) → Add positional encoding for position 3.

Step 3: Encoder (03:01:03)

- **What Happens**:
   - The input sentence (with embeddings and positional encodings) is passed through the **Encoder**.
   - The Encoder consists of multiple layers, each containing:
      1. **Multi-Head Self-Attention**: Computes attention scores for each word in the sentence.
      2. **Feed-Forward Neural Network**: Adds non-linearity to the model.
      3. **Layer Normalization**: Stabilizes the training process.
- **Self-Attention in Detail**:
   - For each word, the Encoder computes **queries (Q)**, **keys (K)**, and **values (V)**.
   - The model calculates how much attention to give to each word in the sentence when processing a specific word.
   - For example, when processing "cat", the model might focus more on "The" and "sat" because they are related to "cat".
- **Output of Encoder**:
   - The Encoder outputs a set of **contextualized representations** for each word in the sentence. These representations capture the meaning of each word in the context of the entire sentence.

Step 4: Decoder (03:30:56)

- **What Happens**:
   - The Decoder generates the output sentence (in French) one word at a time, using the context provided by the Encoder.
   - The Decoder also consists of multiple layers, each containing:
      1. **Masked Multi-Head Self-Attention**: Prevents the model from attending to future words in the output sequence.
      2. **Encoder-Decoder Attention**: Attends to the Encoder's output to incorporate context from the input sentence.
      3. **Feed-Forward Neural Network**: Adds non-linearity to the model.
      4. **Layer Normalization**: Stabilizes the training process.
- **Masked Self-Attention**:
   - When generating the first word ("Le"), the Decoder can only attend

    to itself (since no other words have been generated yet).
  - When generating the second word ("chat"), the Decoder can attend to "Le" and itself, but not to future words like "s'est" or "assis".
- **Encoder-Decoder Attention**:
  - The Decoder uses the Encoder's output to understand the context of the input sentence. For example, when generating "chat", the Decoder might focus on "cat" in the input sentence.

Step 5: Linear and Softmax Layer (04:47:00)

- **What Happens**:
  - The output of the Decoder is passed through a **linear layer** to produce **logits** (raw scores) for each word in the French vocabulary.
  - The logits are then passed through a **softmax function** to produce probabilities for each word.
  - The word with the highest probability is selected as the output.
- **Example**:
  - When generating the first word, the model might produce the following probabilities:
    - "Le" → 0.9
    - "La" → 0.1
    - "Un" → 0.0
  - The model selects "Le" as the first word.
  - This process is repeated for each word in the output sentence.

Step 6: Output Sentence

- **What Happens**:
  - The Decoder generates the output sentence one word at a time, using the context from the Encoder and its own previous outputs.
- **Example**:
  - The final output sentence is: "Le chat s'est assis sur le tapis."

Summary of the Transformer Workflow

1. **Input Embedding**: Convert words into numerical vectors.
2. **Positional Encoding**: Add information about word positions.
3. **Encoder**: Process the input sentence using self-attention and feed-forward layers to generate contextualized representations.
4. **Decoder**: Generate the output sentence one word at a time, using masked self-attention and encoder-decoder attention.
5. **Linear and Softmax**: Convert the Decoder's output into probabilities and select the most likely word.
6. **Output**: Produce the final translated sentence.

Example Walkthrough

- **Input Sentence**: "The cat sat on the mat."
- **Step-by-Step Translation**:
  1. **Input Embedding**: Convert "The", "cat", "sat", "on", "the", "mat" into vectors.
  2. **Positional Encoding**: Add positional information to each word.
  3. **Encoder**: Compute self-attention and generate contextualized representations for each word.
  4. **Decoder**:
     - Generate "Le" (attending to "The" and "cat").
     - Generate "chat" (attending to "cat" and "Le").
     - Generate "s'est" (attending to "sat" and previous words).
     - Continue until the full sentence is generated.
  5. **Output Sentence**: "Le chat s'est assis sur le tapis."