Transformer Architecture

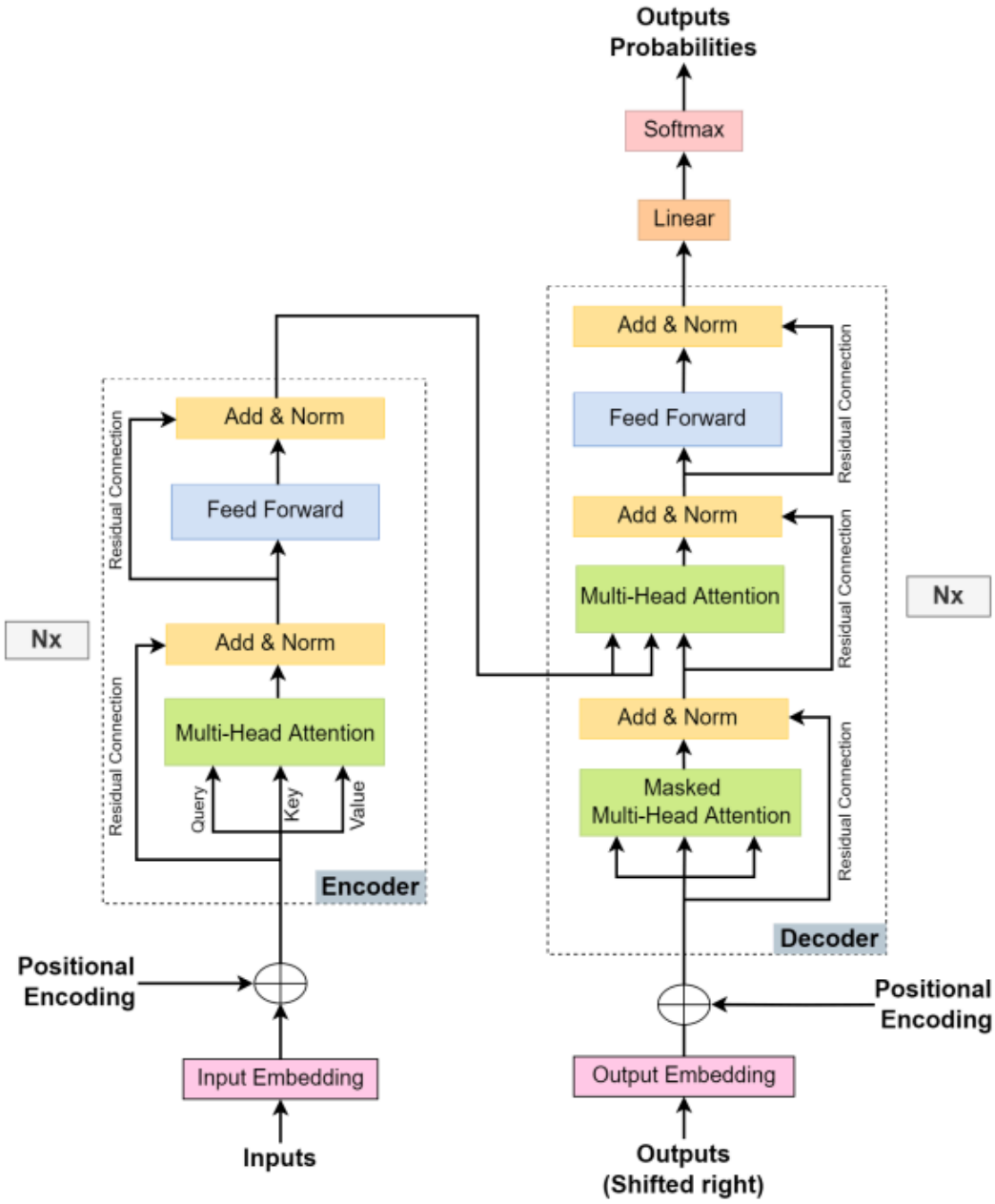

Figure 2: Transformer architecture (Vaswani et al., 2017)

Let’s understand the decoder’s layers by taking a translation example. English sentence “We are friends" is being translated to hindi “हम दोस्त हैं ”

Training Phase Decoder's execution:

Let's break down the Decoder side step by step in the context of your example, where the target input "हम दोस्त हैं" is being fed into the decoder **during the training phase.**

Step 1: Input Tokenization

The Hindi sentence "हम दोस्त हैं" is first tokenized into individual tokens (e.g., `[हम, दोस्त, हैं]`) and converted to unique token IDs using a pre-defined vocabulary.

Step 2: Word Embedding and Positional Embedding

- Each token ID is transformed into a dense vector (word embedding), capturing the semantic meaning of each word.
- Positional embeddings are added to these word embeddings to give the model a sense of the order of words in the sentence.

Now, the input to the first decoder layer is a sequence of embeddings:
`[E(हम) + P0, E(दोस्त) + P1, E(हैं) + P2]`.

Step 3: Masked Multi-Head Self-Attention

- This layer helps the decoder attend to the previous tokens in the target sequence while predicting the next token.
- A mask ensures that the decoder doesn't "cheat" by looking at future tokens during training.
    - For example, while predicting "दोस्त", the decoder only considers "हम".

This is done by:

1. Computing Query (Q), Key (K), and Value (V) vectors for each token.
2. Calculating attention scores for previous tokens using dot-product attention.
3. Using the mask to zero out attention scores for future tokens.

The output is a contextual representation of the target sequence so far.

Step 4: Encoder-Decoder Multi-Head Attention

- This layer allows the decoder to attend to the encoder's output, which contains the contextual embeddings of the source sentence "We are friends".

Here's what happens:

1. The decoder uses the self-attended output from Step 3 as the Query (Q).
2. It uses the encoder output as Key (K) and Value (V).
3. Attention scores are calculated to align tokens in the target sentence with relevant tokens in the source sentence.

For example:

- "हम" might align with "We".
- "दोस्त" might align with "friends".

Step 5: Feed-Forward Neural Network

- Each token's representation is passed through a position-wise FFN to apply additional transformations.
- The FFN includes two linear layers with a ReLU activation in between. This step is necessary to learn the "non linearity of the data"

Step 6: Add & Norm Layers

- Residual connections (skip connections) ensure that gradients flow better during training.
- Each sub-layer (e.g., attention, FFN) adds its output back to its input, and the sum is normalized.

Step 7: Output Token Prediction

- After passing through all the decoder layers, the output is a contextual embedding for each token in the target sequence.
- A **linear layer** maps these embeddings to vocabulary scores.

Why a Linear Layer?

The decoder's final output at each time step is a **vector** (contextual embedding) that represents the token being predicted. This vector is still in a high-dimensional space and needs to be transformed into probabilities over the entire vocabulary to predict the next word.

How It Works

1. **Input to the Linear Layer**:
   Each token's **embedding** (output of the last decoder layer) is a vector of size Dmodel (e.g., 512 or 1024 dimensions).
2. **Linear Transformation**:
   The linear layer applies a learned transformation matrix W of size $|V| \times$Dmodel

   where:

   - $|V|$ is the size of the vocabulary (e.g., 30,000 words for many language models).
   - Dmodel is the size of the embedding vector.

   Mathematically:

   $$\text{Scores} = W \cdot \text{Embedding (Coming from previous step)} + b$$

   Here:

   - W projects the embedding into a space where each dimension corresponds to a vocabulary token.
   - b is a bias vector of size $|V|$.

   Output of the Linear Layer:
   The result is a vector of size $|V|$, where each element represents a raw score (called **logits**) for a vocabulary token.

   The linear layer transforms this vector into a **logits vector** with a size equal to the vocabulary size.

   Since our vocabulary has only 3 words, the output will be a vector of size 3: Logits Vector=[2.1,0.5,1.8]

From Scores to Probabilities

After the linear layer:

1. The **softmax function** is applied to the scores. This converts them into probabilities for each token in the vocabulary:

$$P(\text{Token}_i) = \frac{\exp(\text{Score}_i)}{\sum_{j=1}^{|V|} \exp(\text{Score}_j)}$$

   - $P(\text{Token}_i)$ is the probability of the i^{th} token in the vocabulary.
   - The sum of probabilities across all vocabulary tokens is 1.
2. The token with the highest probability is selected as the prediction.

Example

---

3. Softmax Function

The softmax function converts the logits into probabilities, ensuring they sum to 1:

$$P(\text{word}_i) = \frac{\exp(\text{logit}_i)}{\sum_{j=1}^{3} \exp(\text{logit}_j)}$$

For the logits $[2.1, 0.5, 1.8]$:

- $P(\text{"हम"}) = \frac{\exp(2.1)}{\exp(2.1)+\exp(0.5)+\exp(1.8)} \approx 0.59$
- $P(\text{"दोस्त"}) = \frac{\exp(0.5)}{\exp(2.1)+\exp(0.5)+\exp(1.8)} \approx 0.08$
- $P(\text{"हैं"}) = \frac{\exp(1.8)}{\exp(2.1)+\exp(0.5)+\exp(1.8)} \approx 0.33$

---

4. Argmax Selection

The decoder picks the word with the highest probability:

$$\text{argmax}([0.59, 0.08, 0.33]) = 0.59 \quad \Rightarrow \quad \text{"हम"}$$

Thus, "हम" is the selected token for this time step.

The predicted token ("हम") is then fed back into the decoder during training to predict the next token.

Iterative Generation

1. At the next time step, "हम" will be part of the input to the decoder.
2. The process repeats: the decoder produces a hidden state, the linear layer maps it to logits, softmax generates probabilities, and the model selects the next word via argmax.

This process repeats for each decoder layer and for each time step until the entire target sequence is processed.

Training Feedback Loop

During training, the ground truth tokens ("हम", "दोस्त", "हैं") are used as inputs at each step to minimize the loss between predicted tokens and actual tokens.

Key Differences Between Training and Inference

1. **Training Phase**: The ground truth target sequence (e.g., "हम दोस्त हैं") is provided to the decoder at each step, enabling teacher forcing.
2. **Inference Phase**: The decoder must generate the target sequence one token at a time without ground truth. Instead, it uses its own previously predicted tokens as input.

Doubt: During training phase , do decoder not use previously predicted tokens as input ?? I think the only difference during inferencing is lack of ground truth.

Answer: **Clarification:**

During the **training phase**, the decoder primarily uses **teacher forcing**, where the ground truth tokens are used as inputs at each time step rather than the model's own predicted tokens. This is a major distinction between training and inference.

Step-by-Step Process for Inference

Let's use the same example: translating "We are friends" into Hindi, where the target sentence is "हम दोस्त हैं".

Step 1: Input to the Encoder

- The source sentence "We are friends" is tokenized, embedded, and passed through the encoder.
- The encoder outputs contextual embeddings for each token in the source sentence.

Step 2: Decoding Starts

The decoding process happens iteratively, one token at a time:

Iteration 1: Predict the First Token

1. **Initialize Input**:
    - The decoder starts with a special token, such as `<START>` (or its embedding), as the initial input.
2. **Masked Self-Attention**:

    - Since there is only one token (`<START>`), no masking is needed at this stage.
3. **Encoder-Decoder Attention**:
    - The decoder attends to the encoder's output to align `<START>` with relevant parts of the source sentence.
4. **Output Token Prediction**:
    - The linear layer and softmax produce a probability distribution over the vocabulary.
    - The most probable token (e.g., "हम") is selected as the first prediction.

Iteration 2: Predict the Second Token

1. **Input Update**:
    - The decoder now receives the embedding of the predicted token ("हम") as input, along with the `<START>` token embedding.
2. **Masked Self-Attention**:
    - The decoder computes attention over the sequence `[<START>, हम]`, but future positions remain masked.
3. **Encoder-Decoder Attention**:
    - The decoder again attends to the encoder's output using the contextual embeddings of `<START>` and "हम".
4. **Output Token Prediction**:
    - The linear layer and softmax produce probabilities, and the most probable token (e.g., "दोस्त") is selected as the second prediction.

Iteration 3: Predict the Third Token

1. **Input Update**:
    - The decoder receives the embeddings of `[<START>, हम, दोस्त]`.
2. **Repeat the Attention Mechanisms**:
    - Masked self-attention ensures the decoder only attends to previously predicted tokens.
    - Encoder-decoder attention aligns these tokens with the source sentence.
3. **Output Token Prediction**:
    - The decoder predicts the next token (e.g., "हैं").

Step 3: Stopping Condition

- The decoder continues generating tokens until a special `<END>` token is predicted or a maximum sequence length is reached.

Example Walkthrough: Translating "We are friends" to "हम दोस्त हैं"

1. Encoder processes "We are friends" → Outputs contextual embeddings.
2. Decoder predicts:

- Step 1: `<START>` → "हम".
- Step 2: `<START>, हम` → "दोस्त".
- Step 3: `<START>, हम, दोस्त` → "हैं".
- Step 4: `<START>, हम, दोस्त, हैं` → `<END>` (stopping condition).

This distinction between **non-autoregressive** and **autoregressive** behavior in Transformers is fundamental to understanding how the model operates during **training** and **inference**. Let me explain this in detail.

Non-Autoregressive During Training

In the training phase:

- **Ground Truth Tokens Are Available**: The entire target sequence (e.g., "हम दोस्त हैं") is provided to the decoder. This allows **teacher forcing**, where the correct tokens are fed to the decoder at every time step, regardless of its own predictions.

Why Non-Autoregressive?

- The decoder processes the entire target sequence **in parallel**. It does not depend on the outputs of the previous time steps because the correct tokens (ground truth) are directly provided.
- All tokens of the target sequence are passed through the decoder layers simultaneously, which speeds up training.

Autoregressive During Inference

In the inference phase:

- **Ground Truth Tokens Are Not Available**: The model must generate the target sequence token by token. Each predicted token is fed back into the decoder to predict the next one.

Why Autoregressive?

- The decoder generates the target sequence **one step at a time**. Predictions at step t depend on all previous predictions [<START>,y1,y2,…,yt−1].
- This sequential dependency makes the process **autoregressive**.

Why This Difference?

1. **Training Optimization**:

    - During training, the goal is to efficiently compute loss across the entire sequence. Providing ground truth tokens enables this to be done in parallel.
2. **Inference Challenges**:
    - At inference time, the ground truth is unknown. Predictions must depend on the model's own outputs, requiring a step-by-step (autoregressive) process.

Summary

- **Training**: Non-autoregressive, thanks to parallelism enabled by ground truth tokens.
- **Inference**: Autoregressive, generating one token at a time by feeding previous predictions back into the model.