TARC : Transformer Architecture Implementation in Python

“Attention is All You need” is the paper that introduced the Transformer architecture, which has since revolutionized the field of natural language processing. The researchers did not discovered the Self-Attention mechanism, but they were the first to build an architecture solely based on it, discarding the previously dominant Recurrent Neural Networks (RNNs) and Convolutional Neural Networks (CNNs).

Glossary

This glossary summarizes key terms related to the Transformer architecture and training.

- Token: The smallest unit of text processed by the model (word, subword, or character).
- Embedding: A numeric vector representation of a token in continuous space.
- Mask: A binary matrix that blocks certain positions (padding or future tokens).
- Self-Attention: Mechanism where each token attends to all other tokens in the sequence.
- Multi-Head Attention: Several attention mechanisms in parallel to capture different contexts.
- Feed-Forward Network: A small neural network applied independently at each position.
- Positional Encoding: Vectors added to embeddings to give order information to tokens.
- Encoder: The block that transforms the source sequence into contextual representations.
- Decoder: The block that generates the target sequence using the encoder's output.
- Logits: Raw values output by the model before applying softmax to get probabilities.
- Batch: A group of examples processed in parallel during training.
- Epoch: One complete pass over the training dataset.
- Loss Function: A measure of the error between predictions and expected targets.
- Backpropagation: Automatic computation of gradients to update the model's weights.
- Optimizer: Algorithm (e.g., Adam) that adjusts weights using the gradients.
- Gradient Clipping: Technique to limit gradient size and stabilize training.
- Overfitting: When the model memorizes training data but fails to generalize.
- Regularization: Methods (dropout, weight decay) to reduce overfitting.
- Beam Search: A generation strategy that explores multiple candidate sequences.
- Vocabulary (Vocab): The set of tokens the model can understand and generate.
- d_model: The dimensionality of embeddings and hidden layers in the model.
- Head Dimension: The size of sub-vectors handled by each attention head (`d_model / num_heads`).
- Query (Q): A vector asking for information from other tokens in attention.
- Key (K): A vector used as a reference to be compared against queries.
- Value (V): A vector carrying the information used to build the output.
- Score Matrix: The $Q \cdot K^T$ product, measuring similarity between tokens.
- Softmax: Function that turns scores into attention probabilities.
- Dropout: Technique that randomly zeros out neurons to prevent overfitting.
- LayerNorm: Normalization applied to vectors for stability and faster training.
- Residual Connection: Adding the input of a sub-layer back to its output to ease learning.

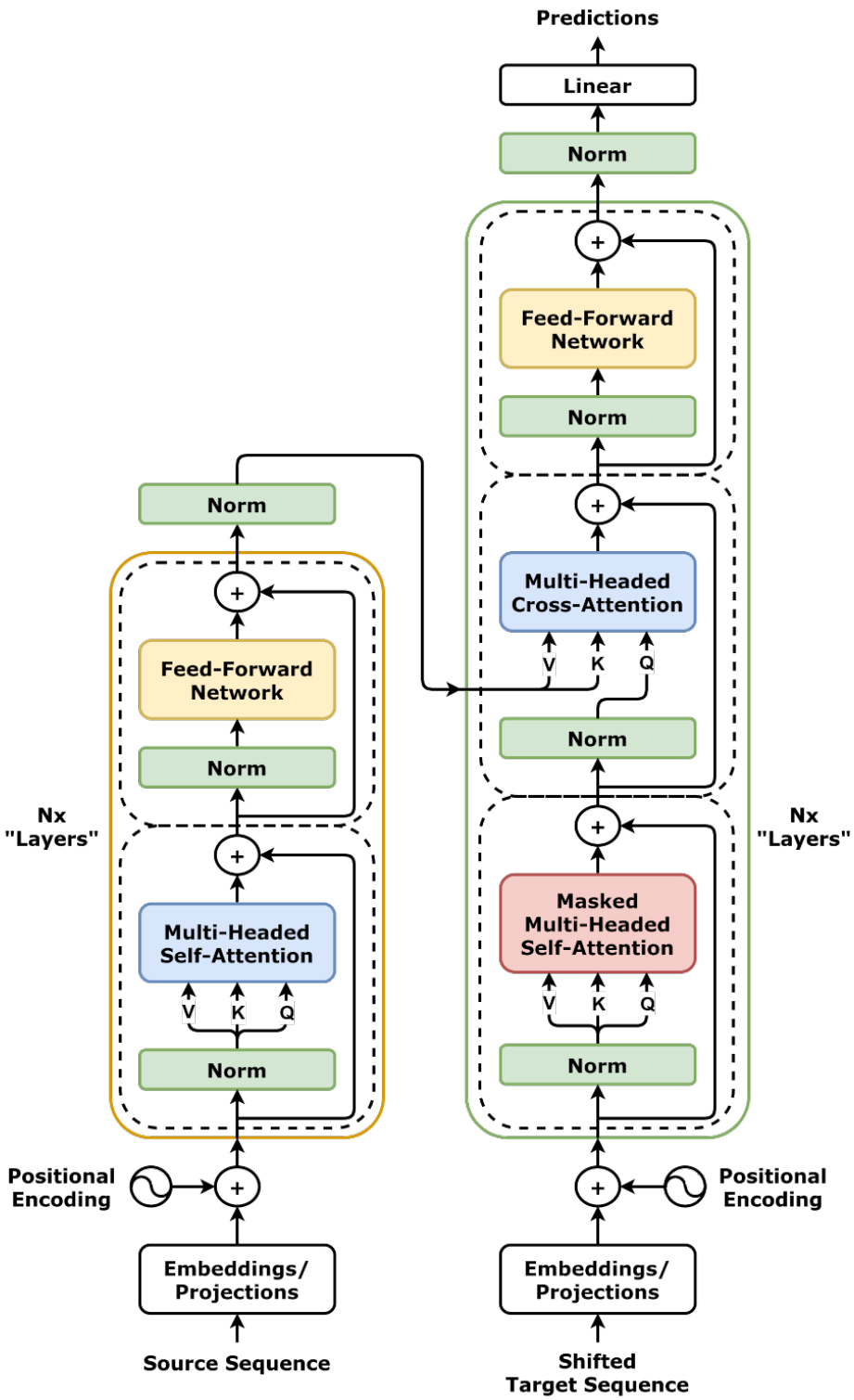

Figure 1: transformer architecture

Transformer Model

The Transformer model consists of an Encoder and a Decoder, both of which are composed of multiple layers of self-attention mechanisms and feed-forward neural networks.

The Two Main Components

These components are implemented in the `coders.py` file.

Encoder

The Encoder is responsible for processing the input sequence and generating a set of continuous representations. It consists of *multiple identical layers*, each containing two main sub-layers:

- a multi-head self-attention mechanisn, which allows the model to focus on different parts of the input sequence and capture contextual relationships between words,
- a position-wise feed-forward neural network, which further transforms the representations for each position independently.

The self-attention mechanism allows the model to weigh the importance of different words in the input sequence.

– Note : In the Encoder class, we initialize a list of EncoderLayer and we define the forward method of the nn.Module, which takes the input sequence and a source mask as arguments. The input sequence is passed through each EncoderLayer in the list, and the output is returned.

Decoder

The Decoder is responsible for generating the output sequence, one token at a time. It is also composed of *multiple identical layers*, and each layer contains three sub-layers:

- a masked multi-head self-attention mechanism, which allows each target token to attend only to earlier tokens (preventing access to future information),
- an encoder-decoder multi-head attention mechanism, which enables the Decoder to attend to the Encoder's output representations and focus on relevant parts of the source sequence,
- a position-wise feed-forward network, applied independently to each position to further transform the representations.

– Note: In the Decoder class, we initialize a list of DecoderLayer modules in the same way as the Encoder. During the forward pass, the target sequence is passed through these layers in order, and the resulting representation is then used by the final output layer to predict tokens.

Layers

The layers of the Encoder and Decoder are implemented in the same file as the classes above, `coders.py` .

EncoderLayer (Pre-LN : Pre-Layer Normalization)

This class defines a single layer of the Encoder. It consists of two main sub-layers: a multi-head self-attention mechanism and a feed-forward network as said previously, each preceded by layer normalization and followed by a residual connection.

Constructor:

- `d_model`: dimension of the input/output vectors (such as 512).

- `num_heads`: number of attention heads used in the multi-head attention.
- `d_ff`: dimension of the hidden layer in the feed-forward network (usually 4 x `d_model`, such as 2048).
- `dropout`: probability used to drop units during training to prevent overfitting.
- Initializes:
  - a multi-head attention layer + its layer normalization,
  - a feed-forward network + its layer normalization.

Forward pass:
1. Input `src` is normalized and sent through the multi-head self-attention.
2. The result is added back to the original input (residual connection).
3. The updated representation is normalized and sent through the feed-forward network.
4. Again, a residual connection adds the input of the sub-layer to its output.
5. The final enriched representation of the source sequence is returned.

This design ensures stable training (thanks to Pre-LN) and allows information to flow easily through the network via residual connections.

DecoderLayer (Pre-LN)
This class defines a single layer of the Decoder. It consists of three main sub-layers: masked self-attention, encoder-decoder attention (cross-attention), and a feed-forward network. Each sub-layer is preceded by layer normalization and followed by a residual connection.

Constructor:
- Initializes:
  - a masked multi-head self-attention layer + its layer normalization,
  - a cross multi-head attention layer (decoder attends to encoder output) + its normalization,
  - a feed-forward network + its normalization.

Forward pass:
1. Input `tgt` is normalized and sent through the masked self-attention, which prevents each position from attending to future tokens. Its result is added back to the original input (residual connection).
2. The updated sequence is normalized and sent through the encoder-decoder attention.
3. The result is normalized and passed through the feed-forward network.
4. The final enriched representation of the target sequence is returned.

This structure ensures that the decoder can use both previously generated tokens (masked self-attention) and information from the source sequence (cross-attention), while remaining stable thanks to Pre-LN.

Multi-Head Attention : Parallel Processing

The Multi-Head Attention mechanism is implemented in the `multihead.py` file.

The 'head' concept is about *splitting the model's attention* mechanism into multiple parallel attention layers, each focusing on different parts of the input sequence. Each head works with a *subset* of the model's total dimensionality (d_model / num_heads). After processing, the outputs of all heads are concatenated and linearly transformed to produce the final output.

In the MultiHeadedAttention class, we create a list of SelfAttention for each head. Each SelfAttention takes the same inputs (query, key, value, mask) but learn its own weights.

Therefore, each head can *focus on different aspects* of the input sequence, capturing a richer set of relationships and dependencies. Finally, the outputs from all heads are concatenated and passed through a linear layer to *combine the information.*

The forward method use query, key and value. They are matrices computed from the input embeddings.

Matrices :
- Query (Q): A vector calculated for each token, which is used to search for relevant information in other tokens.
- Key (K): A vector calculated for each token, which represents the content of that token. This is used to match against the Query.
- Value (V): A vector calculated for each token, which contains the actual information to be transmitted if the Key matches the Query.

We can vizualize the matrices and similarities with colors for examples.

Self-Attention : Core Mechanism

The Self-Attention mechanism is implemented in the `selfattention.py` file.

This is a mechanism that allows each token to "look at" other tokens in the sequence to gather relevant information.

The key formula is : $Attention(Q, K, V) = softmax(Q \cdot K^T / \sqrt{d_k}) \cdot V$

Where:
- Q (Query), K (Key), V (Value) are matrices derived from the input embeddings.
- d_k is the dimension of the Key vectors (used for scaling).
- softmax is a function that converts scores into probabilities.

In the product $Q \cdot K^T$, we compute the similarity between each Query and all Keys. Therefore, we obtain a score matrix that indicates *how much focus* each token should give to every other token (The attention).

In the end, the new representation for each token is a weighted sum of the Value vectors, which means that each token's new representation incorporates information from other tokens based on their relevance.

– Note : We take Q and K, multiply them, scale the result, apply softmax and multiply by V.

Feed-Forward Network

The Feed-Forward Network (FFN) is implemented in the `feedforward.py` file.

The Feed-Forward Network (FFN) is applied independently to each position in the sequence. It consists of two linear transformations with a ReLU activation in between.

The core formula is: $FFN(x) = \max(0, x W_1 + b_1) W_2 + b_2$

Positional Encoding

Since the transformer architecture does not inherently understand the order of tokens in a sequence (like RNNs do), we add positional encodings (1st, 2nd, 3rd, etc.) to the input embeddings to provide this information.

For a position pos and dimension i, the positional encoding is defined as:

- PE(pos, 2i) = sin(pos / 10000^(2i/d_model))
- PE(pos, 2i+1) = cos(pos / 10000^(2i/d_model))

Even indices use the sine function and odd indices use the cosine function.

Transformer Class

The overall Transformer model is implemented in the `transformer.py` file.

It contains the constructor method, the forward method and a method to generate output.