# Transformers

CNN, RNN, LSTM are sequential architectures that limited the possibilities to train on the same scale of data that showed value for computer vision

- Transformers processes sequintial data with much more Parallization.

* Types of modifications to the transformers block.

| Modification | Transformer |
|---|---|
| Lightweight transformers | |
| | Lite Transformer [274]<br>Funnel Transformer [66]<br>DeLighT [180] |
| Cross-block connectivity | |
| | Realformer [107]<br>Transparent Attention [19] |
| Adaptive computation time | |
| | Universal Transformer [69]<br>Conditional Computation Transformer [18]<br>DeeBERT [276] |
| Recurrent | |
| | Transformer-XL [67]<br>Compressive Transformer [204]<br>Memformer [287] |
| Hierarchical | |
| | HIBERT [296]<br>Hi-Transformer [270] |
| Different architectures | |
| | Macaron Transformer [174]<br>Sandwich Transformer [201]<br>Differentiable Architecture Search [299] |

* Transformer block changes:

- Decreasing memory footprint and compute.
- Adding connections between transformers blocks.
- Adaptive computation time (allow early stopping during training).
- Recurrence or hierarchical structure.
- changing the architecture more drastically.

* Parts of transformer block:
- positional encodings.
- multi-head attention.
- residual connections with layer normalization.
- position-wise feedforward network.

TABLE 1.2 Types of modifications to the multi-head attention module

| Modification | Transformer |
|---|---|
| Low-rank | |
| | Performer [53]<br>Nystromformer [277]<br>Synthesizer [241] |
| Attention with prior | |
| | Gaussian Transformer [102]<br>Realformer [107]<br>Synthesizer [241]<br>Longformer [25] |
| Improved multi-head attention | |
| | Talking-heads Attention [227]<br>Multi-Scale Transformer [234] |
| Complexity reduction | |

| Modification | Transformer |
|---|---|
| | Longformer [25]<br>Reformer [142]<br>Big Bird [292]<br>Performer [53]<br>Routing Transformer [214] |
| Prototype queries | |
| | Clustered Attention [256]<br>Informer [302] |
| Clustered key-value memory | |
| | Set Transformer [151]<br>Memory Compressed Transformer [167]<br>Linformer [259] |

Kinds of positional encoding:-
- absoulte positional encodings (standard transformer).
- relative " " (Transformer-XL).
- hybird encodings have absoulte and relative.
- implicit encodings that provide information about sequence.

* Pre-training Methods:-

- encoder only :- BERT
- decoder only :- GPT-3
- encoder-decoder :- T5 and ByT5

TABLE 1.4 Libraries and Tools

| Organization | Language and Framework | API | Pre-trained |
|---|---|---|---|
| AllenNLP | Python and PyTorch | Yes | Yes |
| HuggingFace | Jax, PyTorch, and TensorFlow | Yes | Yes |
| Google Brain | TensorFlow | Yes | |
| GluonNLP | MXNet | Yes | Yes |

* Encoder-Decoder Architecture:-

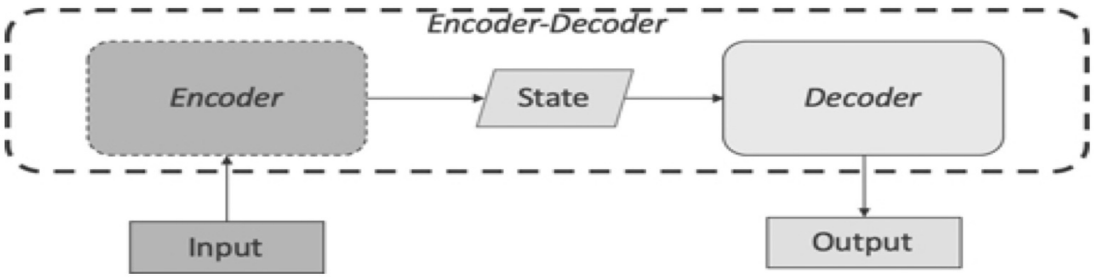



Figure 2.1 Encoder-Decoder architecture.

- The encoder component takes a variable-length sequence and converts it into a fixed-length output-state.
- The decoder component takes a fixed-length state and converts it back into a variable-length output.

* Sequence-to-Sequence

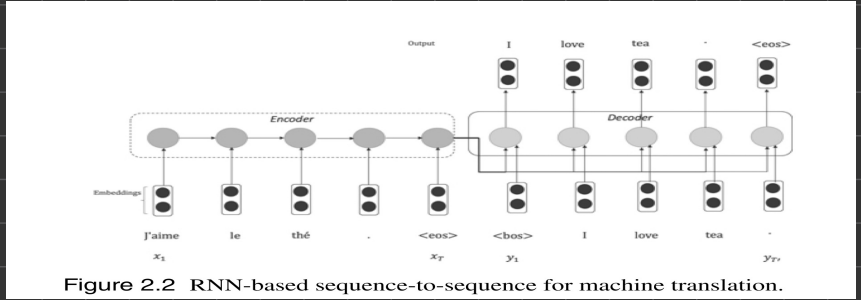



Figure 2.2 RNN-based sequence-to-sequence for machine translation.

• Encoder:-

Sentence is tokenized into words and words are mapped into feature vectors.

$$\mathbf{h}_t = f(\mathbf{h}_{t-1}, \mathbf{x}_t)$$

→ Input: generates a new hidden state.

→ hidden state

→ previous hidden state

$$\mathbf{c} = m(\mathbf{h}_1, \cdots, \mathbf{h}_T)$$

→ mapping function.

→ context vector encodes the information of the entire input sequence.

* Embedding mappings transform the input $x_1, \cdots, x_T$ to $\mathbf{x}_1, \cdots, \mathbf{x}_T$ vectors.

* Decoder:

$$P(\mathbf{y}_{t'}|\mathbf{y}_{t'-1}, \cdots, y_1, \mathbf{c}) = \text{softmax}(\mathbf{s}_{t-1}, \mathbf{y}_{t'-1}, \mathbf{c})$$

Decoder predicts a probability distribution for output tokens on each time step, and softmax gives the distribution over the words.

- Loss function:- cross-entropy is used for optimization.

$$\max_{\theta} \frac{1}{N} \sum_{n=1}^{N} \log p_\theta(\mathbf{y}^{(n)} | \mathbf{x}^{(n)})$$

- RNNs have issues with vanishing and explosive gradients.

* Attention Mechanism:-

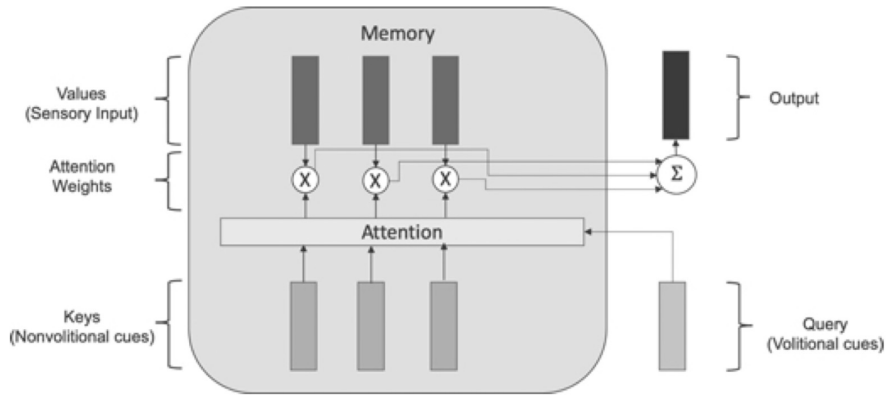



Figure 2.3 Attention mechanism showing query, keys, values, and output vector interactions.

- Attention mechanism can be considered as a memory with keys and values.

To formalize, let us consider the memory unit consisting of $n$ key-value pairs $(\mathbf{k}_1, \mathbf{v}_1), \ldots, (\mathbf{k}_n, \mathbf{v}_n)$ with $\mathbf{k}_i \in \mathbb{R}^{d_k}$ and $\mathbf{v}_i \in \mathbb{R}^{d_v}$. The attention layer receives an input as query $\mathbf{q} \in \mathbb{R}^{d_q}$ and returns an output $\mathbf{o} \in \mathbb{R}^{d_v}$ with same shape as the value $\mathbf{v}$.

- The attention layer measures the similarity between the query and the key using a score function $\alpha$ which returns $a_1, \ldots, a_n$ and $k_1, \ldots, k_n$.

$$a_i = \alpha(\mathbf{q}, \mathbf{k}_i)$$

$a_i$: score; $\alpha$: score function

- Attention weights are computed as softmax f. on the score:

$$\mathbf{b} = \text{softmax}(\mathbf{a})$$

Each element of $\mathbf{b}$ is

$$b_i = \frac{\exp(a_i)}{\sum_j \exp(a_j)}$$

- Output is the sum of the attention weights and values.

$$\mathbf{o} = \sum_{i=1}^{n} b_i \mathbf{v}_i$$

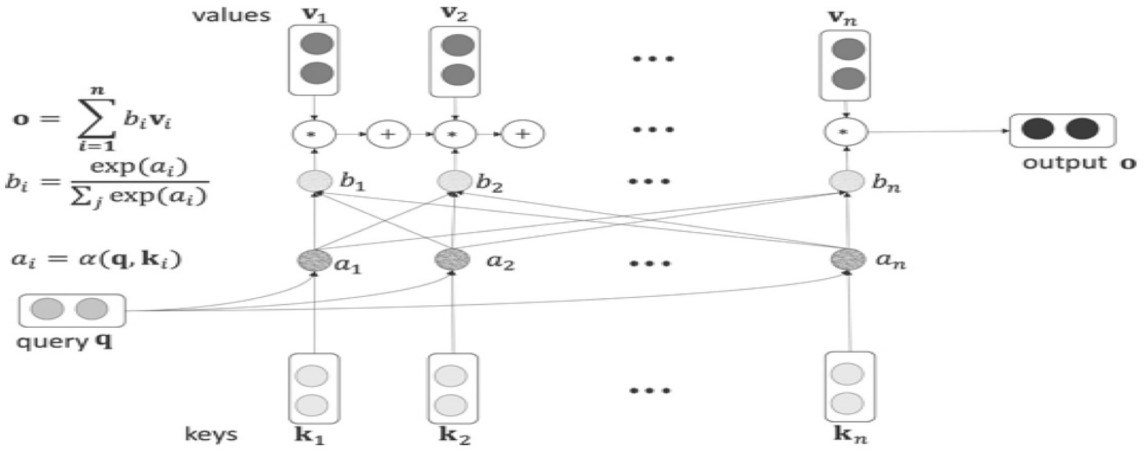



Figure 2.4 Attention mechanism showing query, keys, values, and output vector interactions.

* Types of Score-based Attention:

- Dot-product:

$$\alpha(\mathbf{q}, \mathbf{k}) = \mathbf{q} \cdot \mathbf{k}$$

- has no parameters to tune.

- Scaled dot product:

divides the dot product by $\sqrt{d_k}$.

dim of k

To remove the influence of dimension $d_k$

$$\alpha(\mathbf{q}, \mathbf{k}) = \frac{\mathbf{q} \cdot \mathbf{k}}{\sqrt{d_k}}$$

# * Attention-Based Sequence-to Sequence:

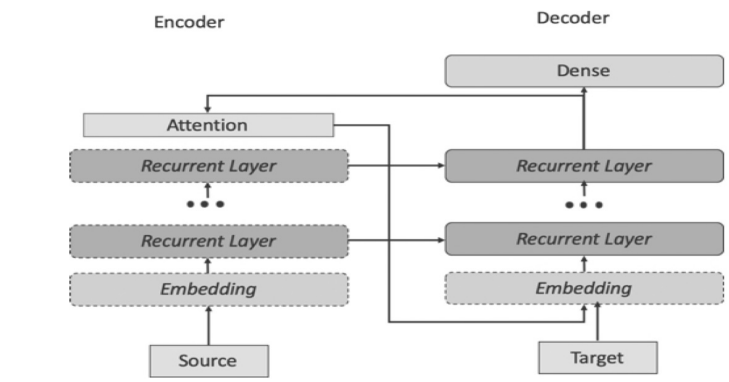


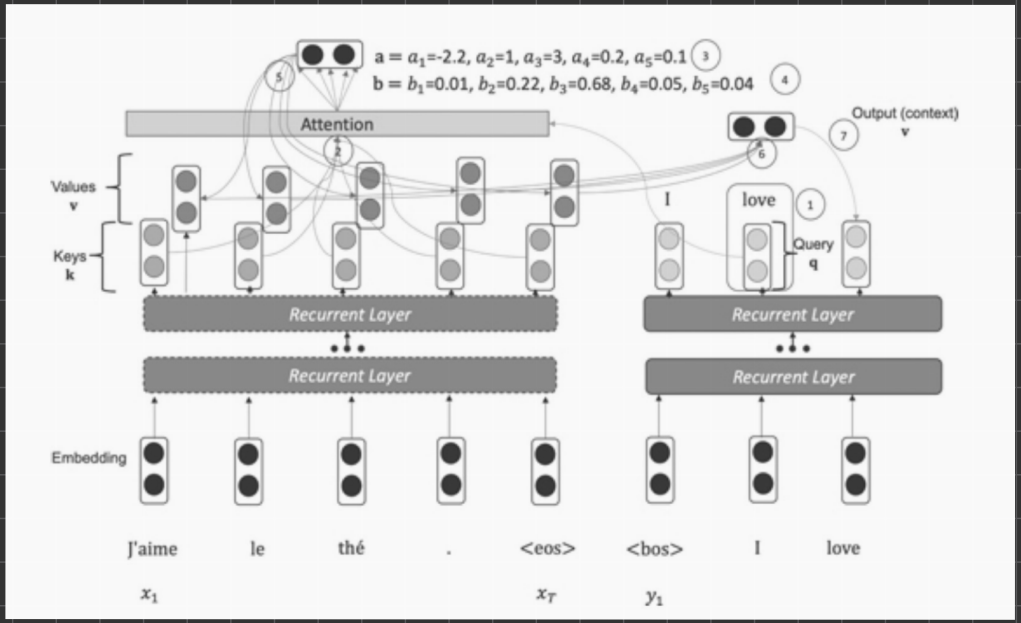

Figure 2.5 Encoder-decoder with attention layer.

- The output of last encoder state are used as keys and values.
- ? = ? = ? = decoder ? at time t-1 is used as query
- output from Attention layer 0 the Context variable used for the next decoder state.

example:

French Translation.

- Encoder state for tokens {J'aime, le, thé ... } are keys and values.
- Decoder at time t-1 generated the tokens I, Love
- Decoder output at time t-1, love, flows into attention layer as query.



* Transformers: based on encoder-decoder architecture.
- Combines the advantages of CNN to parallelize the computations. and RNN to capture long-range, sequential information.

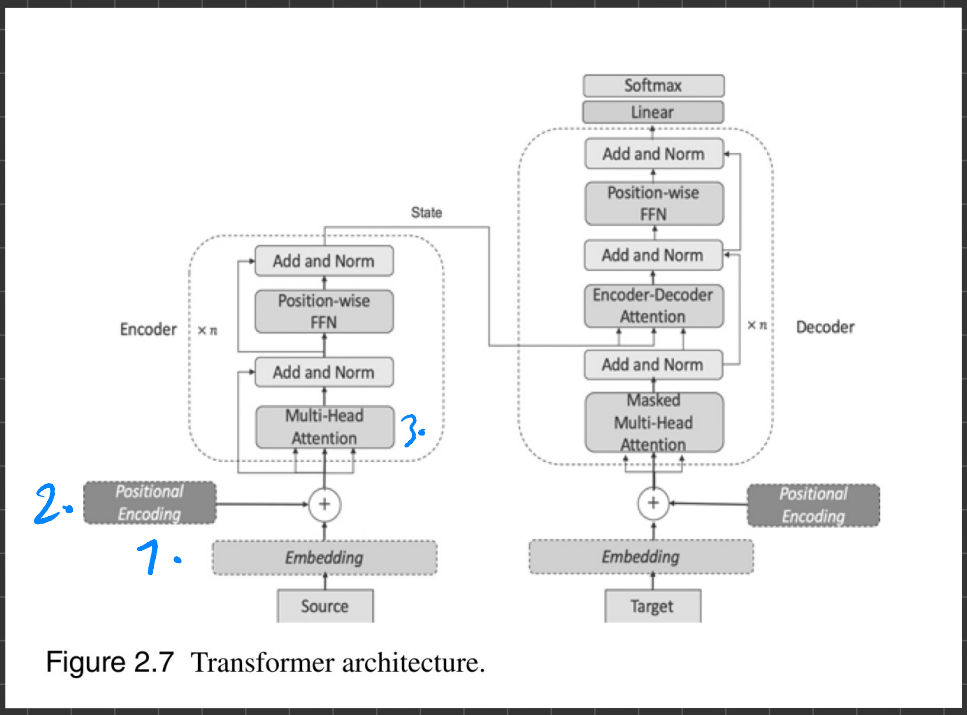



Figure 2.7 Transformer architecture.

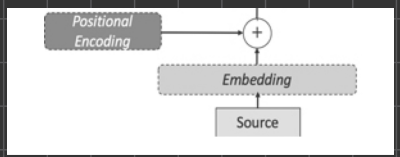
1. Word Embedding:
Convert sentence of length $l$ to Matrix (vector) $\mathbf{W}$ of dim. $(l, d)$.

2. Positional Encoding: (word order information)

$$\mathbf{P}_{i,2j} = \sin(i/1000^{2j/d})$$ → even

$$\mathbf{P}_{i,2j+1} = \cos(i/1000^{2j/d})$$ → odd

⇒ the distance between tow time-steps across sentences is various lengths.



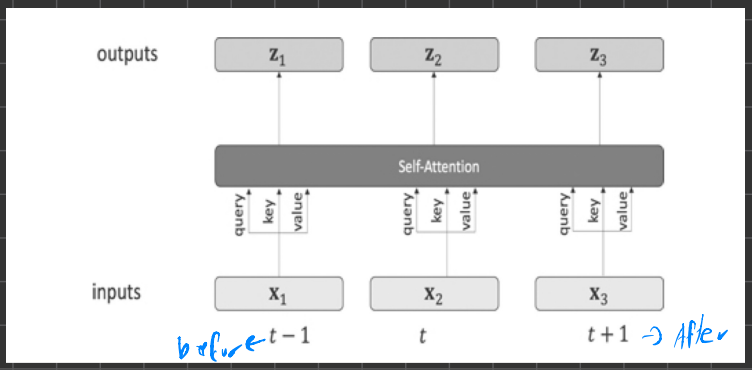

$$\mathbf{X} = \mathbf{W} + \mathbf{P} \in \mathbb{R}^{l \times d}.$$

Input representation ← X

W: Word embeddings

P: positional encoding

# 3. Self-attention:




- Input vectors $x_i$ converted to the vectors $z_i$ through the self-attention layer.

- Each input vector ($x_i$) has 3 diffrent vectors ($Q_i, K_i, V_i$).
  query, key, value

- q,k,v are obtained the input vector ($x_i$) at time i on the learnable weight metrices. $W_q$, $W_k$ and $W_v$.

- Weight metrices are randomly initialized and weights learning from the training.

- For the first attention layer of en. and de., the Inputs are the summition of word embedding (w) and positional encodings (p).

- roles:

1. query vector $q_i$: to combine with every other key vectors. $\sum_{j=0}^{p} q_i k_j^T$ to influence the weights for its own output $z_i$

2. key vector $k_i$: to be matched with every other query vectors to get similarity with query and to influence the output through query-key product scoring.

3. value vector $v_i$: is extracting information by combining with output of the query-key scores to get the output vector $z_i$.

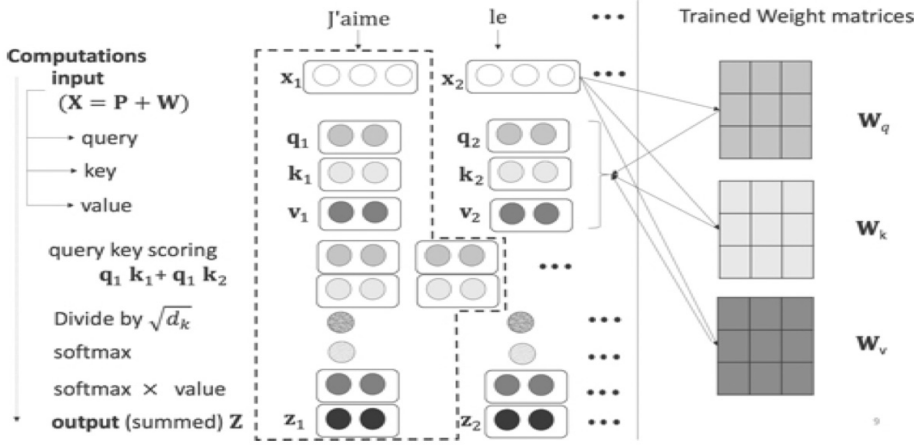



Figure 2.11 The dotted lines show the complete flow of computation for one input through a self-attention layer.

Instead of a vector computation for each token $i$, input matrix $\mathbf{X} \in \mathbb{R}^{l \times d}$ where $l$ is the maximum length of the sentence and $d$ is the dimension of the inputs, combines with each of the query, key, and value matrices as a single computation given by

$$attention(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}(\frac{\mathbf{Q}\mathbf{K}^{\mathrm{T}}}{\sqrt{\mathbf{d}_{\mathbf{k}}}})\mathbf{V}$$

## * Multi-head Attention:-

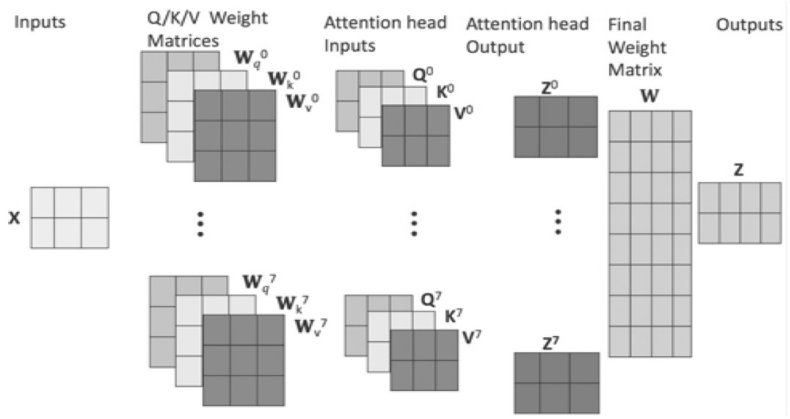



Figure 2.12 Multi-head attention.

$$head_i = attention(\mathbf{W}_q{}^i\mathbf{Q}, \mathbf{W}_k{}^i\mathbf{K}, \mathbf{W}_v{}^i\mathbf{V})$$

$$multihead(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \mathbf{W}_O \text{concat}(head_1, \ldots, head_h)$$

- each head can learn something different.
- multi head Attention have multiple sets of q/k/v weight metrics.
- generating output matrices $Z_i$.

- Output matrices from each head are concatenated and multiplied with a Additional weight matrix $W_o$ to get a single final matrix $Z$ with vectors $z_i$ as output for each input $x_i$.

* Masked multi-head attention:-

- Only previous target tokens need to be present and others to be masked.

- This is implemented by having a masking weight matrix $M$ that has $-\infty$ for future tokens and 0 for previous tokens.

This computation is inserted after the scaling of the multiplication of **Q** and $\mathbf{K}^T$ and before the softmax so that the softmax results in the actual scaled values for previous tokens and the value 0 for future tokens.

$$masked\ Attention(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^{\mathrm{T}}+\mathbf{M}}{\sqrt{\mathrm{d_k}}}\right)\mathbf{V}$$

* Residuals and layer Normalization:-

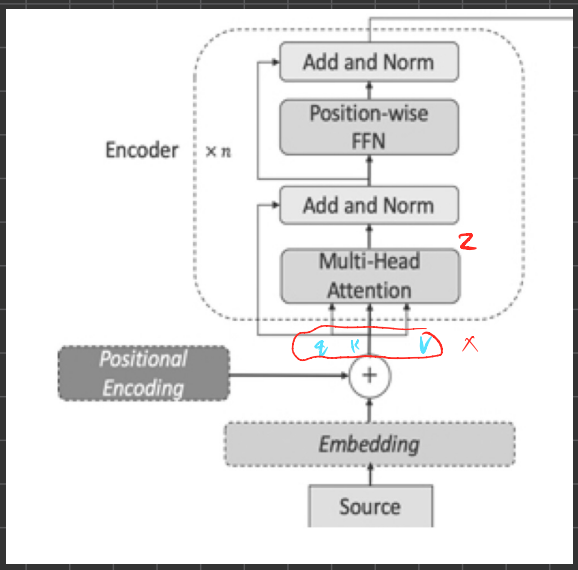
- The input X are circuited to the output Z and both are added and passed through layer Normalization addAndNorm (X + Z)

- Layer normalization ensures each layer to have 0 mean and a unit (1) variance.

- For each hidden unit, $h_i$ we can compute:-

$$h_i = \frac{g}{\sigma}(h_i - \mu)$$

where $g$ is the gain variable (can be set to 1), $\mu$ is the mean given by $\frac{1}{H}\sum_{i=1}^{H} h_i$ and $\sigma$ is the standard deviation given by $\sqrt{\frac{1}{H}(h_i - \mu)^2}$.



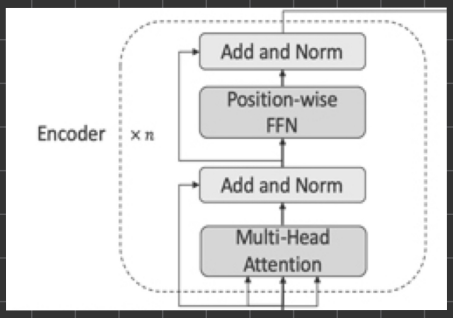

* Position wise Feed-forward Networks:-

- For each position, similar linear transformations with a ReLU activation in between is performed.

$$FFN(\mathbf{x}) = \max(0, \mathbf{x}\mathbf{W}_1 + b_1)\mathbf{W}_2 + b_2$$

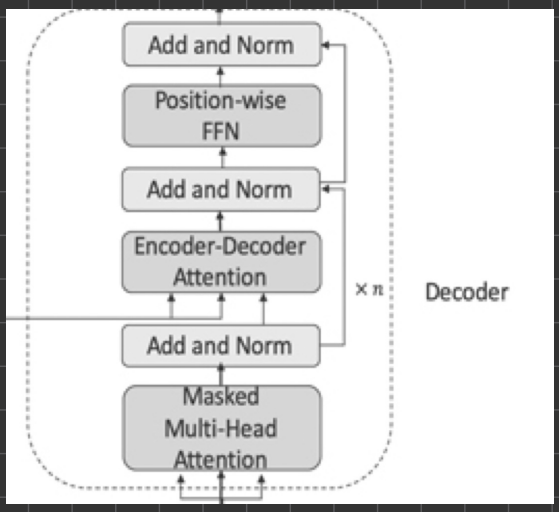
## * Encoder:

- encoder block consists of n blocks of {Multi-Head Attention, addAndNorm, FFN, addandNorm}.
- Multiheadattention in encoder attends to the input or the source.




## * Decoder:

- Multi-head-attention in decoder attends to the target attention between masked outputs with themselves.
- The encoder-decoder attention layer creates attention between the source and the target.



# * Bidirectional Encoder Representation from Transformers (BERT)

- BERT uses unlabeled text to pre-trained language models that can be fine-tuned with a simple additional output layer and a reasonably sized labeled dataset to produce state-of-the-art performance.

## * Architecture:

- Core layers:

- The capacity of the BERT architecture is characterized by:

i. Number of transformer layers $L$.

ii. Size of hiden representations $H$.

iii. Number of bidirectional self-attention heads $A$.

- Input and output Representation:

- BERT prefixes a special [cls] token. => for any input

- Hidden vector of this token in the last layer will be used as an aggregate representation for the entrie input sequence.

=> classification tasks.

- BERT concatenates the sentences into one sequence with a separator token [SEP]

- BERT uses a learned segment embding to indicate to wich sentence a token belongs.

- " " Position embedding layer to encode the order of tokens within the input.

Input: is the sum of the token embedding, segment embedding and positional embedding for each token.

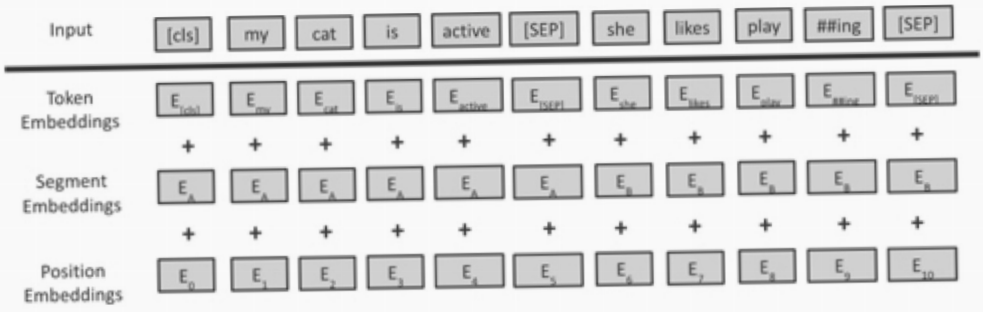



Figure 3.1 BERT input configuration [71].

## Masked Language Model (MLM):

The idea is to randomly mask out a percantage of the input sequence tokens, replacing them with the special [MASK] token.

- modified input sequence runs through BERT.
- output representations of masked tokens are fed into a softmax layer over the word piece as practiced.

## Next Sentence Prediction (NSP):

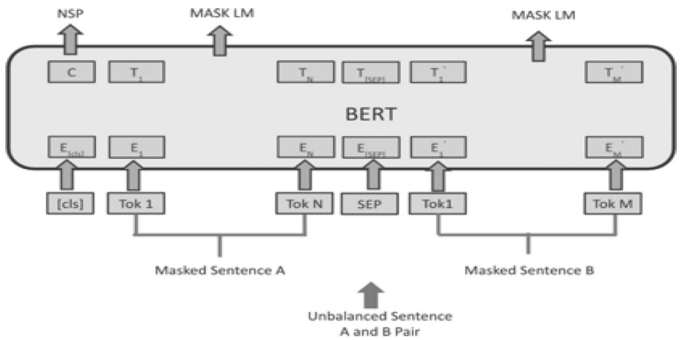



Figure 3.2 BERT pre-training architecture [71].

- the first sentence is prefixed wit [S] token, then the two sentences are delimited by special token [SEP].

To summarize, using the same basic architecture as BERT, RoBERTa has shown that the following design changes significantly improves RoBERTa performance over BERT.

1. Pre-training the model for longer time
2. Using bigger batches
3. Using more training data
4. Removing the NSP pre-training task
5. Training on longer sequences
6. Dynamically changing the masking pattern applied to the training data.