University of Padua

Department of Information Engineering

Master Degree in Computer Engineering

Natural Language Processing

TRANSFORMERS

Foundations and Future Developments

Alessandro Viespoli

alessandro.viesp@gmail.com

Index

Chapter 1

Transformer

Since this work depends on transformer architectures, this chapter provides the necessary background information on core components, connections, and internal transformations of the architecture. A clear understanding of these elements is essential, as our analysis focuses directly on how information flows through layers and how those layers can be modified or pruned. We also describe how the original transformer architecture has evolved in recent years, focusing on the adaptations that enable it to process image data and the design changes introduced to improve training stability, efficiency, and scalability.

1.1 General Picture

The transformer architecture was first introduced in 2017 [1] and quickly proved effective for processing sequential data such as text and time series. Although some may think that the attention mechanism was introduced alongside the transformer [1], mechanisms later referred to as attention were first used in the context of recurrent neural networks for machine translation [2]. The transformer's key innovation was to remove recurrence entirely and build the architecture solely around self-attention and feed-forward layers. Transformers have since become a dominant architecture across a wide range of domains, from natural language processing (NLP) to vision and multimodal reasoning. They power tasks such as machine translation, text and code generation, and image and audio understanding. Recent influential models such as GPT-5, Claude Opus 4.6, DINOv3 and Qwen2.5-VL demonstrate the versatility of transformers and have set new performance benchmarks across multiple modalities and tasks.

In Fig 1.1, the original transformer architecture is shown with an encoder-decoder structure. Both encoder and decoder consist of modules that can be stacked multiple times. The encoder maps an input sequence of symbol representations x to a sequence of continuous representations z. Given z, the decoder then generates an output sequence y of symbols one element at a time. At each step the model is autoregressive, consuming the previously generated symbols as additional input when generating the next.

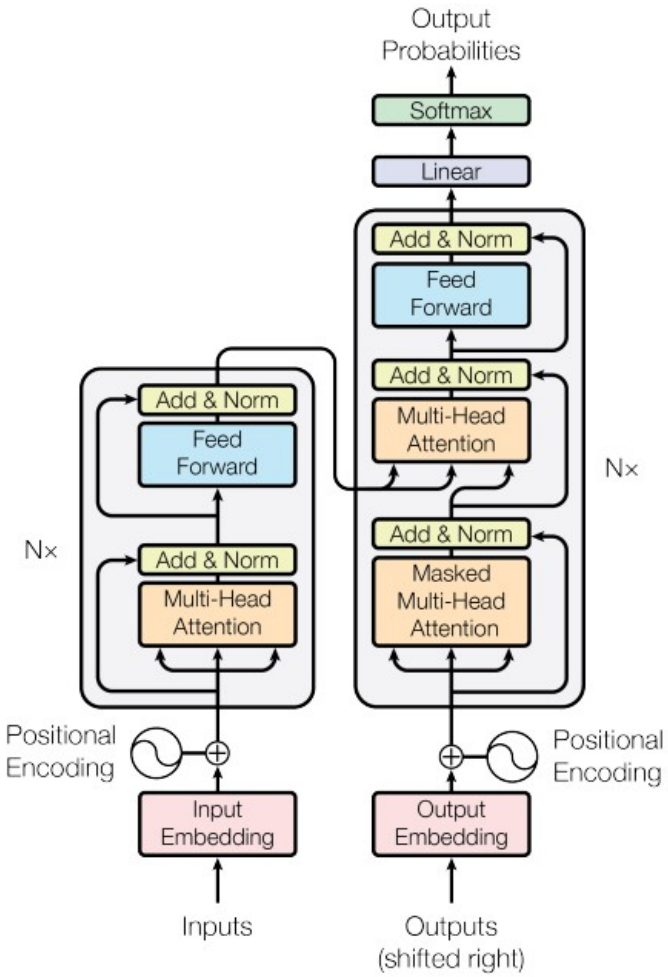
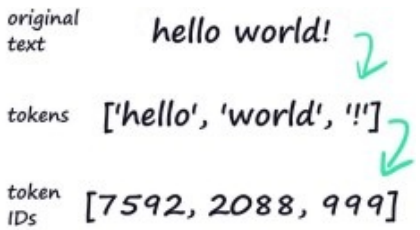


Figure 1.1: Transformer Model Architecture [1]

1.1.1 Tokenization, Embedding, Positional Encoding

Tokenization is the process of encoding a string of text into transformer-readable integers called token IDs.



The input sentence and output target sentence are first tokenized and then **embedded** in a d_{model}-dimensional space since we cannot use strings directly; the dimension d_{model} depends on the chosen model architecture.

Next, since we do not have recurrent networks that can remember how sequences are fed into a model, we must somehow provide each word in our sequence with a relative position, since a sequence depends on the order of its elements. These positions are added to the d_{model}-dimensional vector of each word through a technique called **positional encoding**. The vector that is added to each word is:

$$\vec{p}_t = \left[\sin(\omega_1 t) \quad \cos(\omega_1 t) \quad \cdots \quad \sin\left(\omega_{\frac{d_{model}}{2}} t\right) \quad \cos\left(\omega_{\frac{d_{model}}{2}} t\right)\right]_{1 \times d_{model}} \tag{1.1}$$

where $\omega_k = \frac{1}{10000^{\frac{2k}{d_{model}}}}$ and t is the token position in the sentence.

1.1.2 Self-Attention Mechanism

When performing a search query, such as looking for a specific video on YouTube, the search engine maps the query against a set of keys associated with stored videos. The algorithm then returns the best-matched videos, which can be thought of as the corresponding values. This concept can be generalized: data is often represented as key-value pairs, and a crucial step is to measure the similarity between a query and the keys. A common similarity measure is **cosine similarity**, which evaluates the angle between two vectors:

$$sim(A, B) = cos(A, B) = \frac{A \cdot B}{\|A\|\|B\|} \tag{1.2}$$

Vectors that point in similar directions yield high similarity scores, whereas vectors pointing in different directions produce lower scores. These similarity values are then passed through a softmax function, which normalizes them into a probability distribution that sums to one and sharpens the distribution, emphasizing higher similarity values while suppressing lower ones.

In our case, when processing a sequence of token embeddings, the self-attention mechanism allows each element to interact with all the other embeddings and find out who they should pay more attention to. To compute the attention matrix, the attention block receives a matrix $X \in \mathbb{R}^{n \times d_{model}}$ where each row is the embedding of one input token; for each attention layer we have a set of learnable weights W_K, W_Q, $W_V \in \mathbb{R}^{d_{model} \times d_k}$ which are used to compute the key K, query Q, value $V \in \mathbb{R}^{n \times d_k}$ matrices, where n is the token sequence length, d_{model} is the embedding dimension and d_k is the key, query, value dimension; sometimes it can happen that a different dimension for the value component of attention $d_v \neq d_k$ is chosen.
The attention matrix contains the normalized similarity scores between all queries and keys; each row corresponds to the attention distribution of one token over all tokens in the sequence, formalized as:

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{1.3}$$

where d_k serves as a scaling factor.

1.1.3 Multi-head attention

Instead of applying a single attention operation over the full d_{model}-dimensional queries, keys, and values, the attention mechanism is executed multiple times in parallel where each operation is referred to as a **head**. The motivation for multi-head attention is that each head learns independent projections and can therefore focus on different types of relationships, while a single attention head might obscure distinct types of relationships, potentially limiting the expressive power of the model.

For each head i, the input representations are linearly projected into lower-dimensional subspaces using separate learned projection matrices W_i^Q, W_i^K, $W_i^V \in \mathbb{R}^{d_{model} \times d_k}$. These projections produce transformed queries, keys, and values:

$$Q_i = XW_i^Q, \quad K_i = XW_i^K, \quad V_i = XW_i^V$$

To maintain computational efficiency, the dimensionality of each head is reduced such that

$$d_{model} = h \cdot d_k \quad \Longrightarrow \quad d_k = \frac{d_{model}}{h}$$

Thus, although attention is computed h times, each operation occurs in a lower-dimensional space, keeping the total computational cost comparable to that of single-head attention. The outputs of the h heads are concatenated and projected once more:

$$MultiHead(Q, K, V) = Concat(head_1, \ldots, head_h)W^O \tag{1.4}$$

$$head_i = Attention(Q_i, K_i, V_i) \tag{1.5}$$

where $W^O \in \mathbb{R}^{d_{model} \times d_{model}}$ is a learned output projection matrix.

1.1.4 Layer Normalization

It is important to emphasize that the transformer employs layer normalization, not batch normalization. Layer normalization stabilizes training by normalizing the activations of each individual sample across its feature dimension. Unlike batch normalization, the normalization is performed independently for each token and does not depend on other elements in the batch.

This distinction is particularly important in NLP tasks, where sentence lengths may vary across batches. If batch normalization were used, the normalization statistics would depend on the number of tokens in each batch, which can differ due to padding or variable-length sequences.

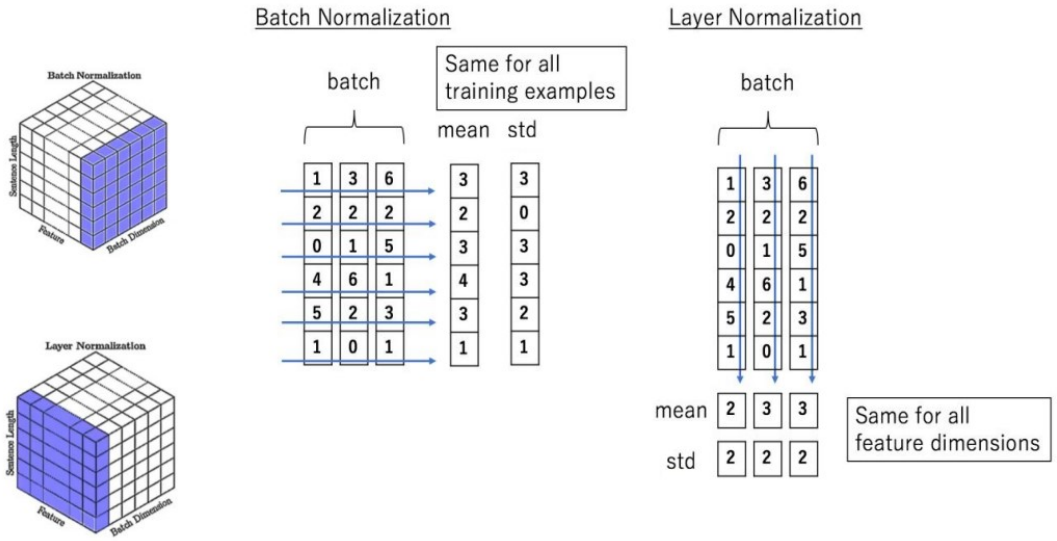


Figure 1.2: Batch Normalization versus Layer Normalization

Let $X \in \mathbb{R}^{n \times d_{model}}$ denote the input to the normalization layer, where n is the number of tokens and d_{model} is the feature dimension. Each row corresponds to one token representation, and each column corresponds to one feature. For a given token i, the mean and variance are computed across its feature dimension:

$$\mu_i = \frac{1}{d_{model}} \sum_{j=1}^{d_{model}} x_{ij} \tag{1.6}$$

$$\sigma_i^2 = \frac{1}{d_{model}} \sum_{j=1}^{d_{model}} (x_{ij} - \mu_i)^2 \tag{1.7}$$

$$\hat{x}_{ij} = \frac{x_{ij} - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}} \tag{1.8}$$

where ϵ is a small constant added for numerical stability. After normalization, learnable scale and shift parameters are applied:

$$y_{ij} = \gamma_j \hat{x}_{ij} + \beta_j \tag{1.9}$$

where $\gamma, \beta \in \mathbb{R}^{d_{model}}$ are trainable parameters.

1.1.5 Residual Connection

The residual connection adds the input of a layer to the output of the corresponding sublayer, so that each sublayer is not required to completely transform the input representation. In other words, the residual connection allows each layer to refine the embedding rather than relearning it from scratch. This mechanism helps preserve information from earlier layers, facilitates gradient propagation, and stabilizes the training of very deep networks.

This concept of residual connections was introduced in the context of deep neural networks [3] to address the degradation problem observed as model depth increases. Empirical results showed that deeper feedforward networks could exhibit higher training error than their shallower counterparts, not primarily due to overfitting or vanishing gradients, but because stacked nonlinear layers are harder to optimize, especially when the optimal mapping is close to the identity. The key idea was to reformulate layer learning through a **residual learning** paradigm: given an input X, instead of directly producing an output $\mathcal{H}(X)$, the block produces $X + \mathcal{F}(X)$. This reparameterization simplifies optimization because, when the desired transformation is close to the identity, the network only needs to learn small residual corrections (i.e. $\mathcal{F}(X) \approx 0$) rather than learning a full mapping from scratch (i.e. $\mathcal{H}(X) \approx X$).

1.1.6 Position-wise Feed-Forward Networks

In addition to the attention layer, transformer blocks contain a fully connected feed-forward network (FFN), also referred to as **multi-layer perceptron** (MLP). This network is applied independently to each token embedding, hence the name **position-wise feed-forward network**.
Let $X \in \mathbb{R}^{n \times d_{model}}$ denote the input to a FFN, where n is the sequence length and d_{model} is the embedding dimension. The feed-forward network is applied to each token representation $x_i \in \mathbb{R}^{d_{model}}$ separately and identically. The FFN consists of two linear transformations with a non-linear activation function in between:

$$\text{FFN}(x_i) = \max(0, x_i W_1 + b_1) W_2 + b_2 \tag{1.10}$$

where $W_1 \in \mathbb{R}^{d_{model} \times d_{ff}}$, $W_2 \in \mathbb{R}^{d_{ff} \times d_{model}}$ and $b_1 \in \mathbb{R}^{d_{ff}}$, $b_2 \in \mathbb{R}^{d_{model}}$ are bias terms. The intermediate dimension d_{ff} is typically larger than d_{model}.

Although the same transformation is applied at every position (i.e. the weights are shared across tokens within the same layer), different transformer blocks use different parameter sets; therefore, the FFN is position-wise but not layer-wise shared.

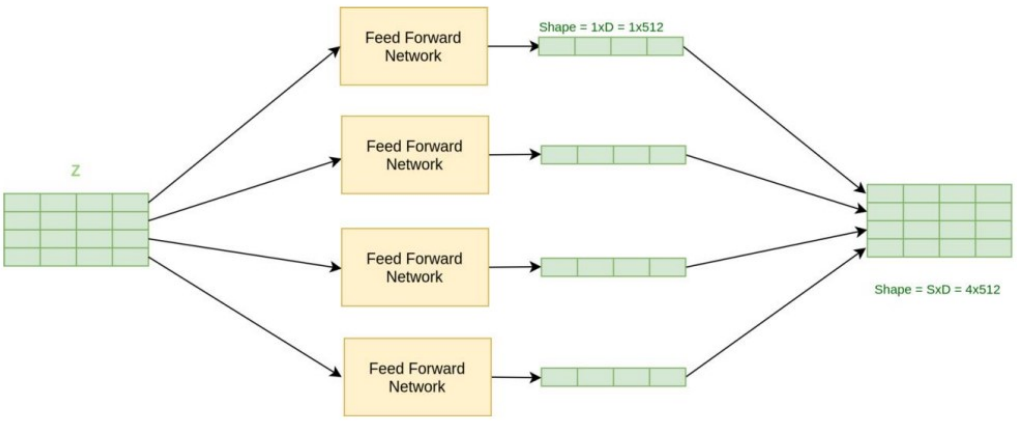
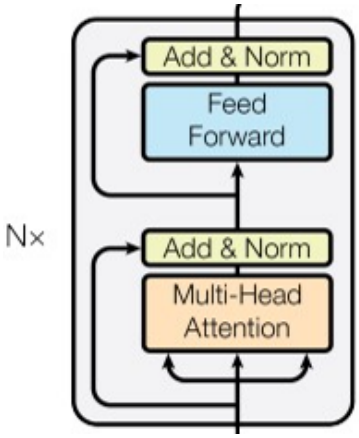


Figure 1.3: Visualization of different FFNs for each token embedding

1.2 Transformer Encoder

Having introduced the main building blocks of the transformer architecture, we now describe how these components are assembled in the encoder.
The transformer encoder consists of a stack of identical layers. Each encoder layer is composed of two main sublayers:



1. Multi-head self-attention mechanism
2. Position-wise feed-forward network (FFN)

We now describe how data flows through the network. The input sentence is first tokenized into n tokens; each token is mapped to a d_{model}-dimensional embedding vector, and positional encoding is added to incorporate order information. For the first encoder layer, X_E^0 denotes the matrix of embeddings combined with positional encodings. For subsequent layers, X_E^l represents the contextualized token representations produced by the previous encoder layer.
Now, let $X_E^l \in \mathbb{R}^{n \times d_{model}}$ be the input to an encoder layer, where n is the sequence length and d_{model} is the embedding dimension. First, the input is passed through the multi-head self-attention mechanism, then a residual connection is applied to the attention output, and the result is normalized using layer normalization:

$$Y_E^l = \text{LayerNorm}(X_E^l + \text{Attention}(X_E^l)) \tag{1.11}$$

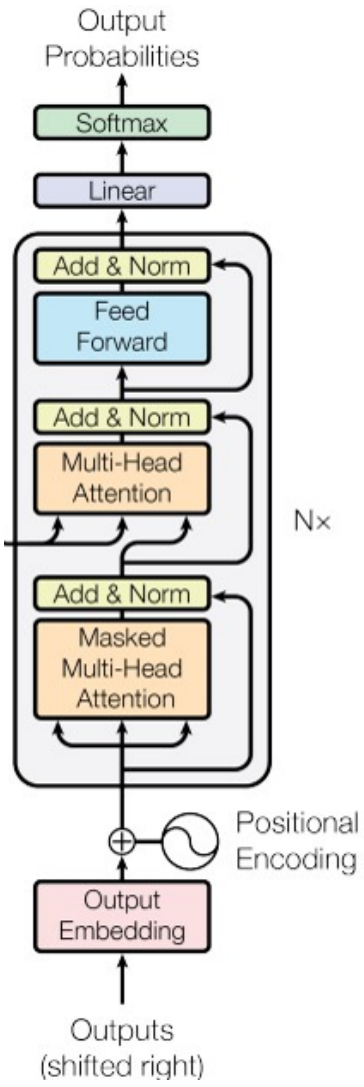
Next, the output Y_E^l is passed through the MLP layer. Again, a residual connection and layer normalization are applied:

$$X_E^{l+1} = \text{LayerNorm}(Y_E^l + \text{MLP}(Y_E^l)) \tag{1.12}$$

The resulting matrix $X_E^{l+1} \in \mathbb{R}^{n \times d_{model}}$ serves as the input to the next encoder layer.
It is important to note that the output of every sub-layer, attention or MLP, has the same dimensionality as its input.
By stacking multiple encoder layers, the model progressively refines the token representations. Together, these components allow the encoder to build increasingly abstract context-aware representations of the input sequence.

1.3 Transformer Decoder

After the encoder produces contextualized representations, the decoder takes them and generates the output sequence autoregressively. While its structure is similar to the encoder, it introduces two additional mechanisms: masked self-attention and encoder–decoder cross attention.



Let the target sentence be tokenized into m tokens. Each token is mapped to a d_{model}-dimensional embedding vector, and positional encoding is added. We denote this input by $X_D^0 \in \mathbb{R}^{m \times d_{model}}$, while the encoder output is denoted by $X_E^L \in \mathbb{R}^{n \times d_{model}}$ where n is the source sequence length.
Each decoder layer consists of three main sublayers:

1. **Masked** multi-head self-attention
2. Encoder–decoder (cross) attention
3. MLP

Given an input X_D^l, a decoder layer starts with:

$$X_l' = \text{LayerNorm}(X_D^l + \text{MaskedAttention}(X_D^l)) \tag{1.13}$$

In the second attention mechanism, the queries come from the decoder representations, while the keys and values come from the encoder output X_E^L; this allows the decoder to condition its predictions on the encoded representation of the source sentence:

$$X_l'' = \text{LayerNorm}(X_l' + \text{CrossAttention}(X_l', X_E^L)) \tag{1.14}$$

$$X_D^{l+1} = \text{LayerNorm}(X_l'' + \text{MLP}(X_l'')) \tag{1.15}$$

The decoder can be interpreted as a conditional language model: it predicts the next token in the target sequence based on the previously generated tokens and the encoded source sentence. A masking mechanism ensures that each position can only attend to earlier positions, allowing the model to maintain its **autoregressive property** while still enabling parallel computation across all positions.

1.3.1 Masked Multi-Head Attention

The masked multi-head attention in the decoder receives inputs with masks that prevent it from accessing information from future (masked) positions. Intuitively, this is like gradually uncovering the input sentence as the model generates each token.
During training, these masks are applied to the target sequences so that the model learns to generate each token based solely on the previously generated tokens. By ignoring the attention weights from future tokens, the transformer learns proper autoregressive decoding.

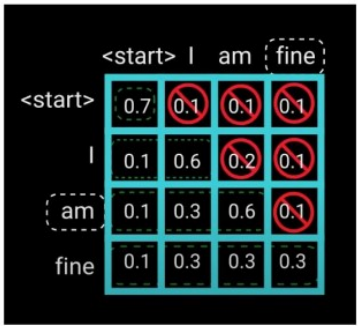


Figure 1.4: Example of causal masks for a sequence

During inference, tokens are generated one at a time, so future tokens are naturally unavailable; thus, the explicit mask used during training is no longer needed.
Formally, masked multi-head attention can be written as:

$$\text{MaskedAttention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top + M}{\sqrt{d_k}}\right)V \tag{1.16}$$

where M is the mask matrix that sets the attention scores of future positions to $-\infty$.

1.3.2 Output Projection

After passing through all N decoder layers, the final representation $X_D^N \in \mathbb{R}^{m \times d_{model}}$ is projected into the vocabulary space:

$$\text{logits} = X_D^N W_{vocab} \qquad W_{vocab} \in \mathbb{R}^{d_{model} \times |\mathcal{V}|}$$

A softmax function is applied independently to the logits at each sequence position, producing a probability distribution over the vocabulary for the next token following that position.

1.4 Modern Transformer Architectures

While the original transformer architecture established the fundamental design principles of attention-based sequence-to-sequence modeling, modern architectures have introduced several modifications to improve scalability, numerical stability, and computational efficiency. These changes include the transition from post-layer normalization to pre-layer normalization, the adoption of RMSNorm, the simplification to decoder-only stacks in large language models, and the development of structured or sparse attention mechanisms for high-resolution vision tasks.

1.4.1 From Post-LN to Pre-LN

A significant modification to the original transformer architecture concerned the order with which layer normalization is applied. This change was made because the old post-layer normalization (**Post-LN**), when scaled to larger networks, exhibited optimization instabilities during early training.
The first cases that used pre-layer normalization (**Pre-LN**) [4, 5] did not formally analyze the architectural change, instead they introduced it as a practical adjustment to improve training stability in deep decoder-only transformers. A theoretical explanation of the improved optimization behavior was later provided [6] and demonstrated that the placement of normalization significantly affects gradient propagation at initialization, explaining why Pre-LN enables stable training in very deep transformers.

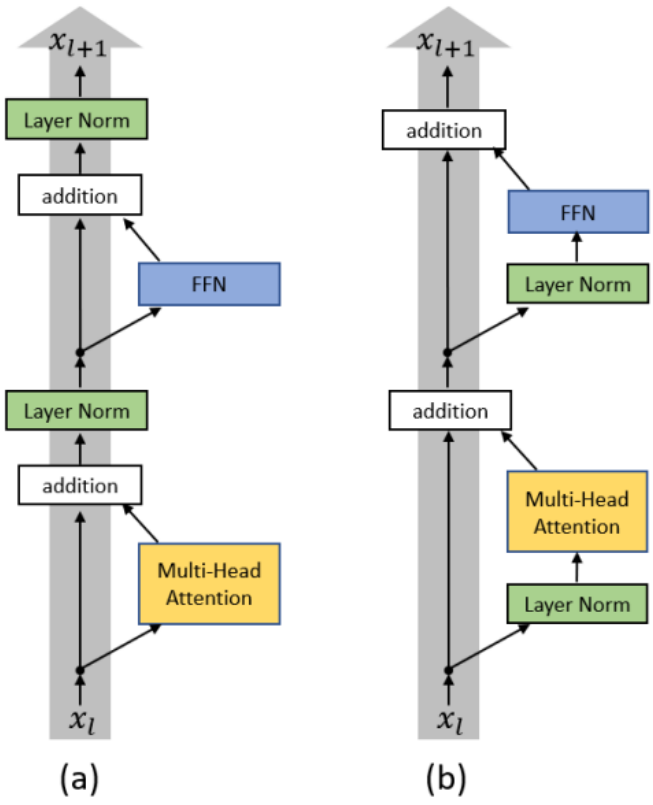


Figure 1.5: Residual Stream rapresentation for (a) Post-LN transformer layer and (b) Pre-LN transformer layer [6]

The Post-LN formulation can be traced back to Equation **??**, where layer normalization is applied **after** the residual addition. In contrast, the Pre-LN formulation applies normalization **before** the sublayer transformation, while preserving the residual branch as a pure identity mapping:

$$Y = X + \mathcal{F}\big(\text{LayerNorm}(X)\big) \tag{1.17}$$

For the purposes of this thesis, it is important to emphasize that in the Pre-LN formulation the residual pathway remains structurally independent from the sublayer transformation, as can be seen in Fig 1.5. As a result, modifications applied within the transformation $\mathcal{F}(\cdot)$ can be analyzed while preserving the integrity of the residual information flow. All transformer backbones considered in this thesis work employ the Pre-LN formulation.

1.4.2 Root Mean Square Layer Normalization

Root Mean Square Layer Normalization (**RMSNorm**) simplifies standard layer normalization (section 1.1.4) by removing the mean-centering step and regulating only the magnitude of the representation [7]. The core idea is that, in transformer architectures, stabilizing the scale of activations is often sufficient for effective optimization, while subtracting the mean is not strictly necessary.
Given an input vector $x \in \mathbb{R}^d$, RMSNorm is defined element-wise as:

$$y_i = \frac{x_i}{\text{RMS}(x)} \cdot \gamma_i \tag{1.18}$$

$$\text{RMS}(x) = \sqrt{\epsilon + \frac{1}{d}\sum_{i=1}^{d} x_i^2} \tag{1.19}$$

where γ_i is a learnable scaling parameter and ϵ is a small constant added for numerical stability.

The normalization step rescales the input vector by a single scalar factor $\text{RMS}(x)$, hence controlling its magnitude without performing mean subtraction. This radial rescaling preserves the relative proportions between components of x, however the subsequent element-wise multiplication by the learnable parameters γ_i can introduce anisotropic scaling since the parameters are not uniform, meaning that the overall resulting direction of the vector y may change with respect to x.
Nevertheless, unlike LayerNorm, RMSNorm does not re-center the representation by subtracting its mean. As a result, it avoids introducing shifts in the embedding space that alter the residual signal through additive transformations.
In Pre-LN transformer architectures, where representations accumulate along residual pathways, this property helps maintain a stable magnitude while limiting normalization-induced distortions.

1.4.3 Encoder-Only and Decoder-Only Architectures

In the years following the original transformer architecture, two architectural variants emerged depending on the task at hand: **encoder-only** and **decoder-only** models.

In encoder-only architectures, each token is allowed to attend to all other elements in the sequence, enabling the model to build rich contextual representations. For this reason, they are particularly suited for vision tasks, where the goal is to learn strong global representations of the input.
In contrast, decoder-only architectures employ masked attention, which restricts each token to attend only to previous positions in the sequence. These models are trained using a self-supervised autoregressive objective:

$$P(x) = \prod_{t=1}^{T} P(x_t \mid x_{<t}). \tag{1.20}$$

This next-token prediction objective is simple, scalable, and requires no labeled data, allowing training on massive corpora of raw text.

The shift toward decoder-only architectures in modern large language models was driven primarily by scalability considerations and objective simplicity. Tasks such as text generation, summarization, or question answering (QA) can be reformulated as a conditional prediction task by concatenating the input x and the desired output y into a single sequence, enabling the model to solve all tasks within the same autoregressive framework:

$$P(y \mid x) = \prod_{t=1}^{T} P(y_t \mid y_{<t}, x), \tag{1.21}$$

This consistency, combined with favorable scaling behavior, has led decoder-only transformers to become the dominant architecture in contemporary large language models, including models such as GPT-5, LLaMA [8], and Mistral 7B [9].

1.4.4 Modern MLP Variants

In modern architectures, even the activation functions and formulations of the MLP block have changed. As described in section 1.1.6, the original transformer used a simple two-layer FFN with ReLU activation between the layers.

Nowadays, new variants are widely adopted in state-of-the-art large-scale models; for example, models in the GPT family, LLaMA [8], and Mistral [9] replace the original ReLU MLP with SwiGLU-style MLP blocks. In these architectures, the activation function typically used is the Sigmoid Linear Unit (**SiLU**), defined as

$$\mathrm{SiLU}(x) = x \cdot \sigma(x) \tag{1.22}$$

where $\sigma(x)$ is the sigmoid function. Unlike ReLU, which is piecewise linear and zero for negative inputs, SiLU is smooth and non-monotonic, allowing small negative values to propagate with reduced magnitude rather than being entirely suppressed.

In the SwiGLU formulation, the feed-forward block becomes

$$\mathrm{MLP}_{SwiGLU}(x) = \big(\mathrm{SiLU}(W_1 x) \odot W_2 x\big) W_3 \tag{1.23}$$

where $\odot$ denotes element-wise multiplication (Hadamard product).

The transition from ReLU-based FFNs to SiLU variants was motivated empirically [10], since smooth activations such as SiLU improve gradient flow, reduce sharp activation boundaries, and lead to more stable optimization in very deep networks. When combined with multiplicative gating, this results in improved parameter efficiency and better performance for large models. These advantages have made SwiGLU MLP blocks one of the most popular choices in contemporary transformer architectures.

1.4.5 Efficient Attention

Across both language and vision domains, the dominant trend has not been the replacement of attention with fundamentally different mechanisms, but rather its structural and computational refinement. The primary computational bottleneck of the transformer architecture lies in the quadratic complexity of self-attention. The matrix multiplication $QK^\top$ in Equation 1.3 requires $\mathcal{O}(n^2 d)$ operations and $\mathcal{O}(n^2)$ memory for the attention matrix, making long sequences or high-resolution inputs computationally expensive. For LLMs, efficiency considerations are particularly critical during inference due to their deployment in production environments. Consequently, new strategies have been developed.

KV Cache

During generation, at each decoding step t, the model computes attention between the new token's query and all previous tokens. This would require recomputing the key and value representations $K_{1:t}$ and $V_{1:t}$ from scratch at every step, even though $K_{1:t-1}$ and $V_{1:t-1}$ were already computed in the previous step.

The key-value cache optimization stores these previously computed key and value tensors in memory thus, instead of recomputing attention over the entire prefix, the model computes only the new key-value pair (K_t, V_t), appends it to the cache, and reuses the stored representations. While this eliminates redundant computation, the KV cache itself introduces a significant extra memory cost.

Multi-Query Attention

To address this memory bottleneck, the multi-query attention (**MQA**) approach [11] reduces this overhead by sharing a single key and value tensor across all heads:

$$K = XW^K, \quad V = XW^V, \quad Q_i = XW_i^Q \quad \text{for } i = 1, \ldots, h$$

Each head retains its own query projection Q_i, allowing it to extract different information from the shared keys and values. This reduces KV-cache memory by a factor of h, because only one key-value pair per layer needs to be stored instead of h pairs.

Grouped-Query Attention

A generalization of MQA was later proposed, called grouped-query attention (**GQA**) [12]. Essentially, it divides the h heads into g groups, where each group shares one key-value projection.

Variant	**Heads**				**KV Pairs**
	1	2	3	4	
Standard MHA	$Q_1K_1V_1$	$Q_2K_2V_2$	$Q_3K_3V_3$	$Q_4K_4V_4$	4 pairs
MQA	Q_1KV	Q_2KV	Q_3KV	Q_4KV	1 pair
GQA ($g = 2$)	$Q_1K_1V_1$	$Q_2K_1V_1$	$Q_3K_2V_2$	$Q_4K_2V_2$	2 pairs

Table 1.1: Comparison of key-value structure across attention variants for $h = 4$ heads and $g = 2$ groups for GQA

These modifications significantly reduce memory bandwidth requirements and improve inference throughput without altering the fundamental attention mechanism. Unlike other approximations, MQA and GQA compute exact attention scores; as a result, GQA has become standard in contemporary large language models, as it preserves model quality while enabling efficient deployment.

1.5 Vision Transformer

Vision Transformers (ViTs) have significantly impacted computer vision. The main goal of the paper [13] was to show that a vanilla transformer, once adapted to deal with data from the visual domain, could compete with some of the most performant convolutional neural networks (CNNs) developed up to that point.
The Vision Transformer architecture is conceptually simple:

1. Split the input image into fixed-size patches
2. Flatten each patch into a vector
3. Project each vector into a lower-dimensional embedding
4. Add positional embeddings to retain spatial information
5. Feed the embeddings into a standard transformer Encoder
6. Pre-train the model in a supervised manner on a large-scale image dataset
7. Use the encoder's output corresponding to a special **class token**.
8. Apply a classification head (MLP) to produce the final logits.

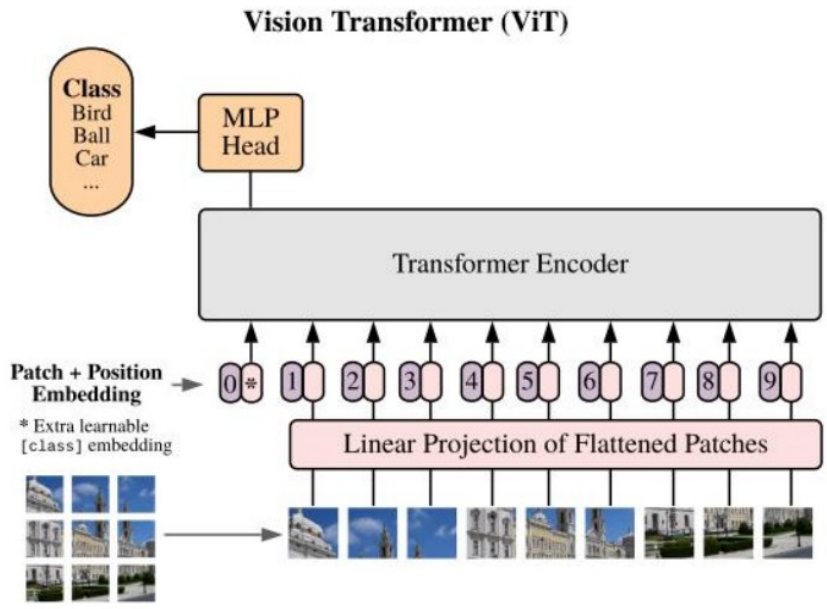


Figure 1.6: Vision Transformer model overview [13]

Unlike the original transformer for sequence-to-sequence tasks, ViT does not include a decoder. The architecture consists solely of a transformer encoder followed by a lightweight classification head. A special learnable token, commonly referred to as the class token, is prepended to the patch embeddings; its final representation is used for classification.

In practice, ViTs are typically pre-trained on very large datasets and then fine-tuned on downstream tasks. During fine-tuning, the original classification head is discarded and

replaced with a new linear layer whose output dimension matches the number of target classes. The transformer encoder weights are either fully fine-tuned or partially adapted, depending on the application and data availability.

1.5.1 Patch Embeddings as Visual Tokenization

A key conceptual shift in vision transformers is the interpretation of image patches as tokens. Given an image $x \in \mathbb{R}^{H \times W \times C}$, it is divided into non-overlapping patches of size $P \times P$, where the total number of patches is

$$N = \frac{HW}{P^2} \tag{1.24}$$

Each patch is flattened into a vector $x_p \in \mathbb{R}^{P^2 C}$ and projected into the model dimension d_{model} using a learned linear embedding:

$$z_p = x_p W_E \tag{1.25}$$

where $W_E \in \mathbb{R}^{P^2 C \times d_{model}}$. The resulting sequence $\left[z_p\right]_{p=1}^{N}$ is treated as a token sequence and processed by a standard transformer encoder.

This formulation establishes a direct parallel between visual tokenization and text tokenization.

1.5.2 Modern Vision Transformers

Since modern and heterogeneous Vision Transformer models were used in this thesis, it is worth explaining their backbones and differences.

Large-Scale Vision Transformers: DINOv2

At the time of writing, DINOv2 is not the latest model in the DINO family; DINOv3 has since been released. However, due to library and implementation constraints, DINOv2 is used in this work. The DINOv2 framework, developed by Meta AI [14], demonstrates that carefully scaled Vision Transformers can serve as strong general-purpose visual feature extractors.

Architecturally, DINOv2 retains the encoder-only transformer structure while introducing improvements in normalization, training stability, and model scaling. The backbone is typically a large ViT variant with Pre-LN blocks and modern MLP designs (e.g, SwiGLU-style activations), aligning it more closely with contemporary transformer practices observed in LLMs. Additionally, DINOv2 emphasizes the use of larger patch sizes and higher-capacity models, enabling the encoder to learn semantically rich global representations.

As discussed earlier, DINOv2 is designed primarily as a feature backbone rather than a task-specific classifier. The learned representations are optimized to be broadly transferable across tasks such as classification, segmentation, and retrieval, effectively functioning as a universal visual encoder.

Hierarchical and Structured Designs: Swin Transformer V2

Unlike the original ViT, which applies global self-attention at every layer, Swin Transformer V2 [15] adopts a hierarchical architecture designed to improve scalability for high-resolution inputs.

In the standard ViT formulation, self-attention has quadratic complexity $\mathcal{O}(n^2)$ with respect to the number of tokens n, which becomes computationally expensive as image resolution increases. SwinV2 addresses this limitation through window-based attention; instead of computing attention globally, tokens are partitioned into non-overlapping windows of size $w \times w$, and self-attention is applied *independently* within each window. This reduces the computational complexity to $\mathcal{O}(nw^2)$.

Architecturally, SwinV2 incorporates a hierarchical pyramid organized into multiple stages. Within each stage, several window-based transformer blocks operate at a fixed spatial resolution. Between stages, a patch merging operation is applied in which neighboring patches are concatenated and linearly projected, reducing the spatial resolution by a factor of two in each dimension while increasing the channel dimension. This results in a hierarchical progression:

$$(H/P, W/P, C) \;\rightarrow\; (H/2P, W/2P, 2C) \;\rightarrow\; (H/4P, W/4P, 4C) \;\rightarrow\; \ldots$$

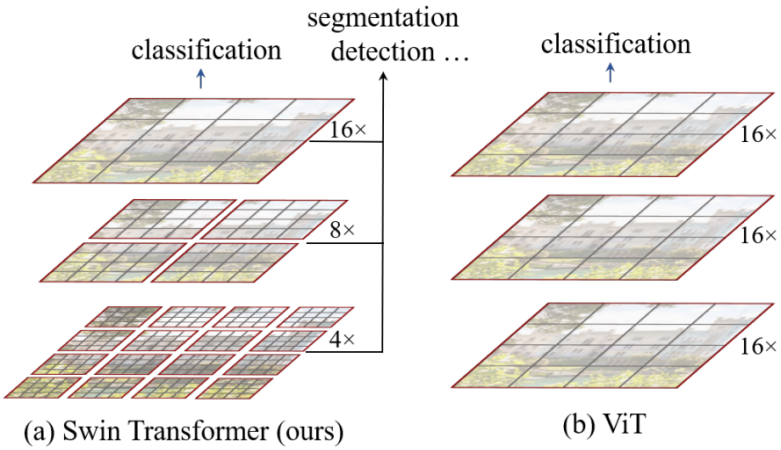


Figure 1.7: SwinV2 Hierarchical Feature Map [16]

To preserve global information exchange, SwinV2 employs **shifted window attention** in alternating layers. By shifting the window partition between successive blocks, cross-window connections are introduced without reverting to full global attention, enabling efficient propagation of information across the entire image.

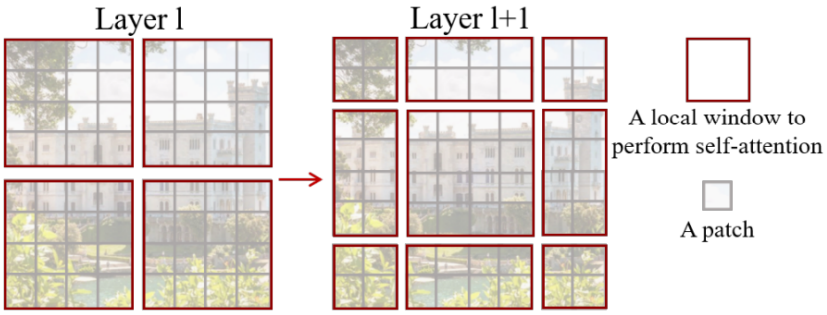


Figure 1.8: Shifted Window approach [16]

Overall, Swin Transformer V2 modifies the flat global-attention structure of ViT into a structured, hierarchical design that preserves locality while maintaining the modeling flexibility of self-attention.

Bibliography

[1] Ashish Vaswani et al. "Attention Is All You Need". In: (2023). arXiv: 1706.03762 [cs.CL]. URL: https://arxiv.org/abs/1706.03762.

[2] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. *Neural Machine Translation by Jointly Learning to Align and Translate*. 2016. arXiv: 1409.0473 [cs.CL]. URL: https://arxiv.org/abs/1409.0473.

[3] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV]. URL: https://arxiv.org/abs/1512.03385.

[4] Alexei Baevski and Michael Auli. "Adaptive Input Representations for Neural Language Modeling". In: (2019). arXiv: 1809.10853 [cs.CL]. URL: https://arxiv.org/abs/1809.10853.

[5] Rewon Child et al. "Generating Long Sequences with Sparse Transformers". In: (2019). arXiv: 1904.10509 [cs.LG]. URL: https://arxiv.org/abs/1904.10509.

[6] Ruibin Xiong et al. "On Layer Normalization in the Transformer Architecture". In: (2020). arXiv: 2002.04745 [cs.LG]. URL: https://arxiv.org/abs/2002.04745.

[7] Biao Zhang and Rico Sennrich. "Root Mean Square Layer Normalization". In: (2019). arXiv: 1910.07467 [cs.LG]. URL: https://arxiv.org/abs/1910.07467.

[8] Hugo Touvron et al. *LLaMA: Open and Efficient Foundation Language Models*. 2023. arXiv: 2302.13971 [cs.CL]. URL: https://arxiv.org/abs/2302.13971.

[9] Albert Q. Jiang et al. *Mistral 7B*. 2023. arXiv: 2310.06825 [cs.CL]. URL: https://arxiv.org/abs/2310.06825.

[10] Noam Shazeer. "GLU Variants Improve Transformer". In: (2020). arXiv: 2002.05202 [cs.LG]. URL: https://arxiv.org/abs/2002.05202.

[11] Noam Shazeer. "Fast Transformer Decoding: One Write-Head is All You Need". In: (2019). arXiv: 1911.02150 [cs.NE]. URL: https://arxiv.org/abs/1911.02150.

[12] Joshua Ainslie et al. "GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints". In: (2023). arXiv: 2305.13245 [cs.CL]. URL: https://arxiv.org/abs/2305.13245.

[13] Alexey Dosovitskiy et al. "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale". In: (2021). arXiv: 2010.11929 [cs.CV]. URL: https://arxiv.org/abs/2010.11929.

[14] Maxime Oquab et al. *DINOv2: Learning Robust Visual Features without Supervision*. 2024. arXiv: 2304.07193 [cs.CV]. URL: https://arxiv.org/abs/2304.07193.

[15] Ze Liu et al. *Swin Transformer V2: Scaling Up Capacity and Resolution*. 2022. arXiv: 2111.09883 [cs.CV]. URL: https://arxiv.org/abs/2111.09883.

[16] Ze Liu et al. *Swin Transformer: Hierarchical Vision Transformer using Shifted Windows*. 2021. arXiv: 2103.14030 [cs.CV]. URL: https://arxiv.org/abs/2103.14030.