VIETNAM GENERAL CONFEDERATION OF LABOR

**TON DUC THANG UNIVERSITY**

**FACULTY OF INFORMATION TECHNOLOGY**



**Final Report**

**Deep Learning**

Advised by

PhD. Lê Anh Cường

HO CHI MINH CITY,2025

VIETNAM GENERAL CONFEDERATION OF LABOR

**TON DUC THANG UNIVERSITY**

**FACULTY OF INFORMATION TECHNOLOGY**



**Lê Hải Đăng – 521H0112**

**Huỳnh Đăng Khoa – 522H0104**

**Châu Bảo Nhân - 522H0093**

**Final Report**

**Deep Learning**

Advised by

PhD. Lê Anh Cường

HO CHI MINH CITY,2025

## Declaration of Originality

**TON DUC THANG UNIVERSITY**

I hereby declare that this thesis is the result of our own work and has been generated under the supervision of PhD. Le Anh Cuong. The research, findings, and conclusions presented in this thesis are original and have not been previously published in any form. All data presented in tables and figures for analysis, evaluation, and assessment were collected by the author(s) from various sources, which are clearly cited in the references.

Furthermore, any comments, evaluations, or data from other authors or organizations used in this thesis are properly cited and referenced.

I take full responsibility for any academic misconduct found in this thesis. Ton Duc Thang University is not liable for any copyright infringements caused by me during this work (if any).

# SUMMARY

This study focuses on exploring the mechanisms and operations of various advanced attention types that are widely used in modern deep learning architectures. The attention mechanisms covered in this work include Self-Attention, Multi-Query Attention (MQA), Grouped-Query Attention (GQA), FlashAttention. For each type, we analyze the core idea, computation process, advantages, limitations, and their role in improving the efficiency and scalability of large-scale neural networks.

Alongside the theoretical analysis, we also build a practical model for text extraction from images. Our approach combines a Convolutional Neural Network (CNN) for extracting spatial visual features with a Transformer Decoder to generate the corresponding text sequences. This CNN + Transformer-Decoder architecture is designed to handle multimodal input effectively, making it well-suited for OCR tasks where both visual understanding and sequential generation are required

# CATEGORY

[Declaration of Originality 3](#_Toc30836)

[SUMMARY 4](#_Toc29554)

[CATEGORY 5](#_Toc11342)

[List of images 6](#_Toc30308)

[PART 1 – ATTENTION 7](#_Toc9104)

[1.1) Self-Attention 7](#_Toc32117)

[1.2) MQA 12](#_Toc125)

[1.3) GQA 16](#_Toc26224)

[1.4) Flash Attention 20](#_Toc17377)

[PART 2 – BUILDING A MODEL TO SOLVE THE PROBLEM 25](#_Toc10336)

[2.1 Introduction to OCR model 25](#_Toc17509)

[2.2 CNN for Visual Feature Extraction 26](#_Toc5118)

[2.3 Transformer Decoder for text generation 26](#_Toc31143)

[2.4 Attention Mechanism between Visual Features and Text Decoder 27](#_Toc19711)

[2.4.1. Self-Attention (Decoder Only): 28](#_Toc27230)

[2.4.2. Cross-Attention (Decoder-to-CNN Features): 28](#_Toc12479)

[2.4.3. Multi-Head Attention: 29](#_Toc24142)

[Summary: 30](#_Toc31103)

[2.5 Build a model to extract text from image 30](#_Toc2690)

[2.5.1 Character Level Tokenizer 30](#_Toc1317)

[2.5.2 Create dataset class 33](#_Toc19433)

[2.5.3 Positional Encoding 35](#_Toc21641)

[2.5.4 OCR model using CNN + Transformer Decoder 37](#_Toc24848)

[REFERENCES 56](#_Toc20663)

## List of images

[Image 1 . Attention Idea 8](#_Toc14213)

[Image 2 . Tokenize + Embbeding 8](#_Toc17979)

[Image 3 .Create a Query vector 9](#_Toc23051)

[Image 4 . Key vector 10](#_Toc15780)

[Image 5 . Mask 10](#_Toc21558)

[Image 6 . MQA example 13](#_Toc5521)

[Image 7 . Multi-Query architecture 13](#_Toc11340)

[Image 8 .Multi head with stand alone Query 14](#_Toc19991)

[Image 9 .Grouped - Query Attention architecture 17](#_Toc26529)

[Image 10 . GQA adjustment 18](#_Toc16576)

[Image 11 . Self-attention compute process on GPU 20](#_Toc8888)

[Image 12 . Memory Hierachy 21](#_Toc6611)

[Image 13 . Tiling processs 23](#_Toc29868)

[Image 14 : Transformer decoder 27](#_Toc30422)

[Image 15 : Self attention 28](#_Toc8654)

[Image 16 : Cross attention 29](#_Toc20050)

[Image 17 : Multi head attention 30](#_Toc24976)

[Image 18 : CharTokenizer 33](#_Toc1238)

[Image 19 : Create Dataset 34](#_Toc15266)

[Image 20 : Formula of positional encoding 35](#_Toc21310)

[Image 21 : Positional Encoding 36](#_Toc16474)

[Image 22 : Model's architecture 37](#_Toc45)

[Image 23 : self-attention 39](#_Toc25293)

[Image 24 : Token vector 40](#_Toc23183)

[Image 25 : First step of self-attention 41](#_Toc4651)

[Image 26 : Compute attention score 42](#_Toc6945)

[Image 27 : Softmax formula 42](#_Toc19632)

[Image 28 : Standard equation for normalization 44](#_Toc24417)

[Image 29 : Normalization 45](#_Toc20979)

[Image 30 : Feed Forward 45](#_Toc19121)

[Image 31 : Transformer Decoder block 46](#_Toc124)

[Image 32 : Convert 47](#_Toc20744)

[Image 33 : Looking up token embedding in an embedding layer 49](#_Toc5662)

[Image 34 : Generating positional embeddings with sine and cosine function 50](#_Toc6632)

[Image 35 : Structure of a decoder-only transformer model 51](#_Toc19080)

[Image 36 : Evaluation 52](#_Toc3984)

[Image 37 : Output1 54](#_Toc30742)

[Image 38 : Output2 55](#_Toc28274)

[Image 39 : Deploy the model 56](#_Toc18157)

## 

# PART 1 – ATTENTION

Attention mechanisms are foundational in modern deep learning model, particularly in Large Language Models (LLMs). This part will explores advanced variants such as Self-Attention, Multi-Query Attention (MQA), Grouped-Query Attention (GQA), FlashAttention.

## Self-Attention

#### Mechanism and Idea

Self-Attention is a mechanism that enables a model to focus on different parts of a sequence when generating representations.

Example : in normal sentences we use adjective to support noun.

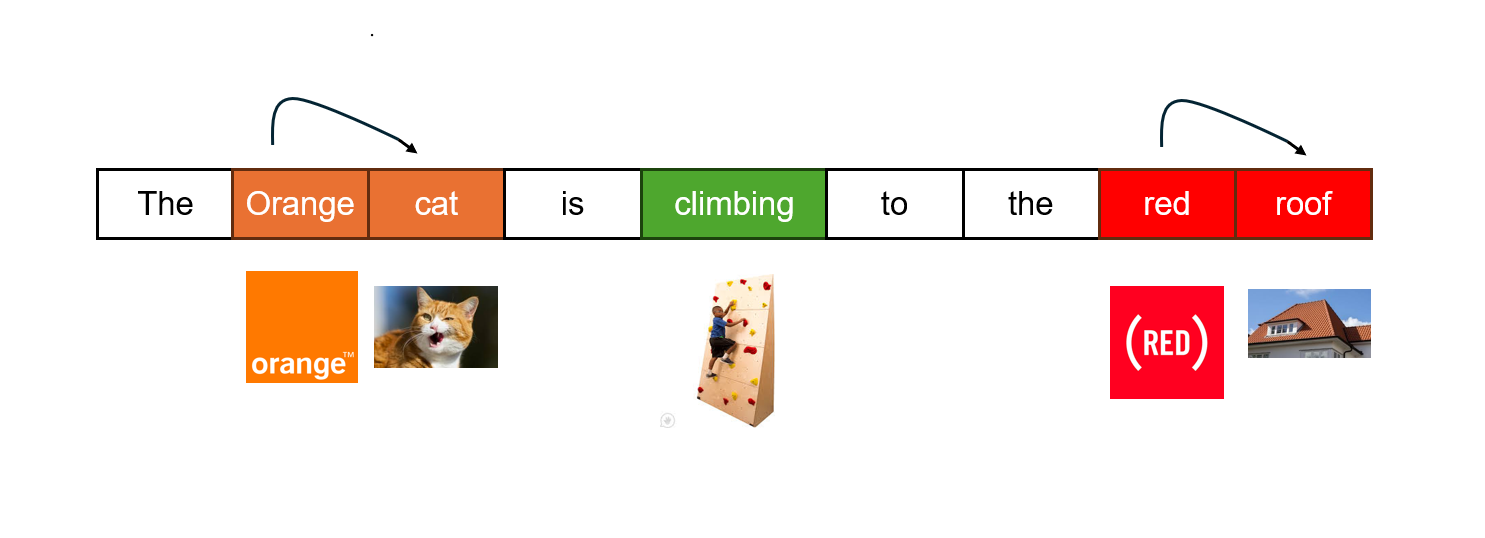


Image 1. Attention Idea

Similar to what we do, computer use it own way to know which word need to be focus by encode each word to high dimensional vector and calculate to learn it meaning. The step by step of self-attention are :

* *Preprocess input sentence :*

Computers do not directly understand human language so the first thing we need to do is convert it to something that computer can understand, in this case is a sequence of vector to present it meaning. This process called Tokenize and Embedding words.

A screenshot of a computer

AI-generated content may be incorrect.

Image 2. Tokenize + Embbeding

* *Create Q, K, V vectors :*

*\*Query vector (Q) :* is a special vector use to present information of request for a token. It is used to compare with Key vector to know how each token is related to each other. For example, if a word “Cat” want to know what is it color, it have to ask all others token like “hey, what is my color ?” in this case the question is represent the Query vector. We create a Q vector by dot product an embedded token vector with query matrix (WQ).

A screenshot of a computer

AI-generated content may be incorrect.

Image 3.Create a Query vector

*\*Key vector (K) :* Similar to Q, K vector is represent for the answer of Q vector use to compare with Q to find which suit it most. In short K vector is a vector of a token that will get a high score if it go with the right Q vector. Example : if Q vector have a meaning like “What is my color ?” and there are many other K vector have meaning like “The” , “Orange”, “Roof”, ..... each of these will be dot product with Q vector to find out what suit Q most by the highest value. In this case it is “Orange”.

A screenshot of a computer

AI-generated content may be incorrect.

Image 4. Key vector

* Mask

Before moving to Value vector, we want to discuss mask techniques. In the attention pattern we want to avoid latter words influence to earlier word, to prevent so we can simply put ) to all value of latter words. The reason for using ) to help Attention pattern stay normalized after SoftMax.

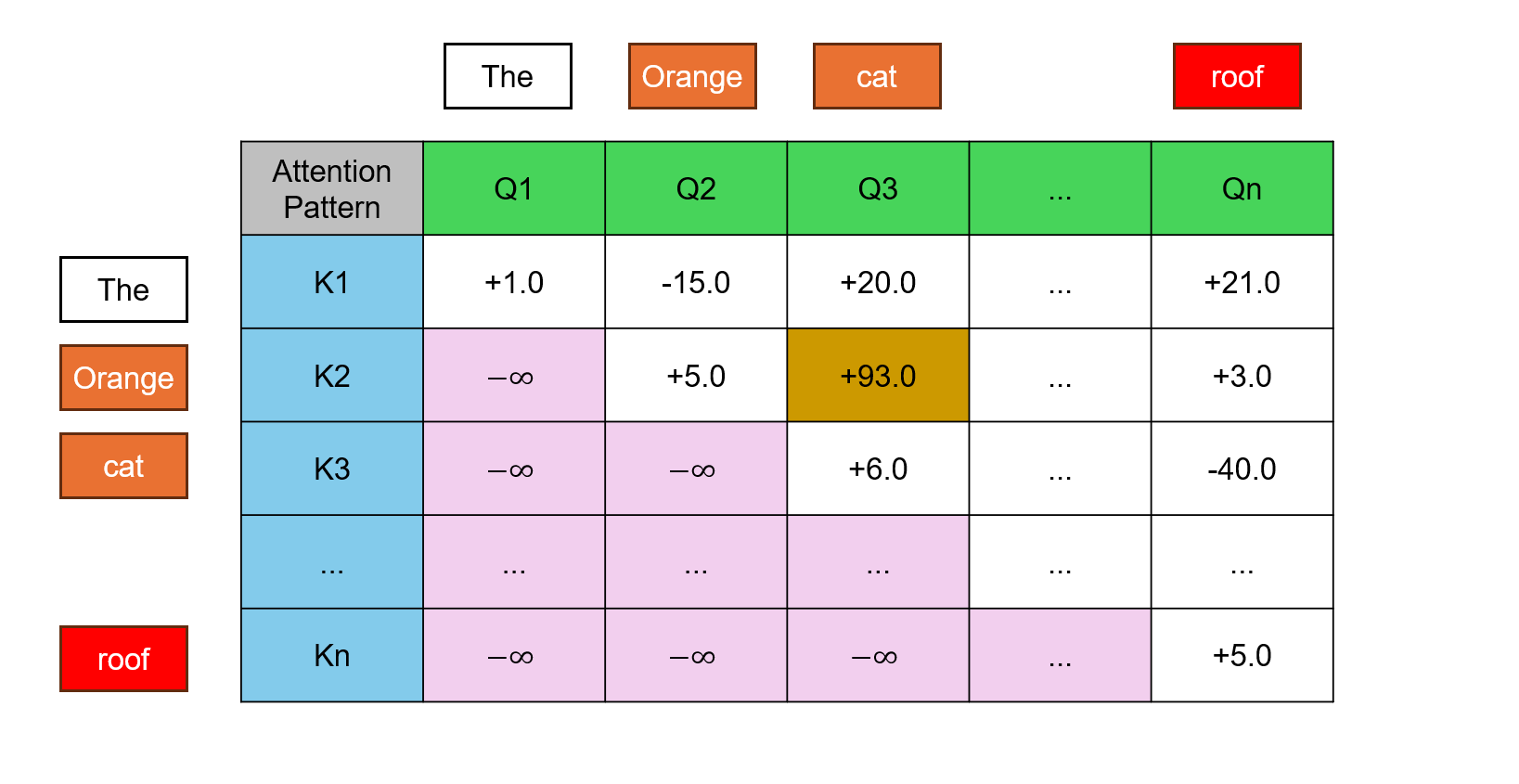
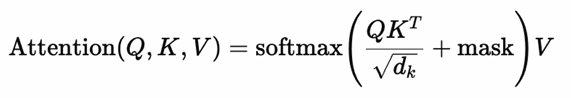


Image 5. Mask

*\*Value vector (V) :* After knowing which token is relate to each other by compare it Q and K vector, we need to actually update the embedding, allowing words to pass information to whichever words they’re relevant to. To create a value vector, we dot product each embedded tokens with a value matrix.

* *Attention Score :*

To know related level of each word to other we need to use Q,K,V vector to calculate with this formula :



#### Complexity

The computational complexity of self-attention depends on the size of the Query, Key, and Value matrices, each of which has a shape of Rnxd, where n is the sequence length and d is the embedding dimension.

To compute the attention output, the following steps are performed:

1. **Dot-product between Q and Kᵀ**:
   * This operation produces a matrix of shape n×nn \times nn×n, representing attention scores between all token pairs.
   * Complexity: ***O(n2 . d)***, since each score is computed via a dot product of d-dimensional vectors.
2. **Softmax over attention scores**:
   * For each of the n queries, we compute softmax over n elements.
   * Complexity: ***O(n2)***
3. **Weighted sum with V**:
   * The softmax results (shape n×n) are multiplied with V (shape n×d) to produce the final output.
   * Complexity: ***O(n2 . d)***

Putting it all together, the overall computational complexity of self-attention is:

#### Advantages and Disadvantages

* ***Advantages:*** self-attention can captures global dependencies, allows each token to attend to all other tokens in the sequence, regardless of their position. This enables the model to capture long-range dependencies effectively.
* ***Disadvantages:*** High computational cost with complexity ***O(n2 . d)*** and Requires positional encoding, self-attention cannot understand position of token and to add manually.

## MQA

#### Mechanism and Idea

In Transformer architecture, Multi-Query Attention (MQA) is an optimization of the standard multi-head attention mechanism. Unlike Multi-Head Attention (MHA), where each attention head has its own set of Queries, Keys, and Values, MQA uses distinct Query projections for each head but shares a single set of Keys and Values across all heads.

This approach significantly reduces memory and computational requirements, especially during inference in large-scale language models.

Example: In the sentence *“The orange cat is climbing to the red roof with its claws,”* we want to determine what the word “it” refers to. While a single-head attention mechanism may struggle to distinguish whether “it” refers to “cat” or “roof,” MQA allows different attention heads to analyze the input from multiple perspectives—while still reusing the same Key and Value representations, improving both efficiency and contextual understanding.

A diagram of a graph

AI-generated content may be incorrect.

Image 6. MQA example

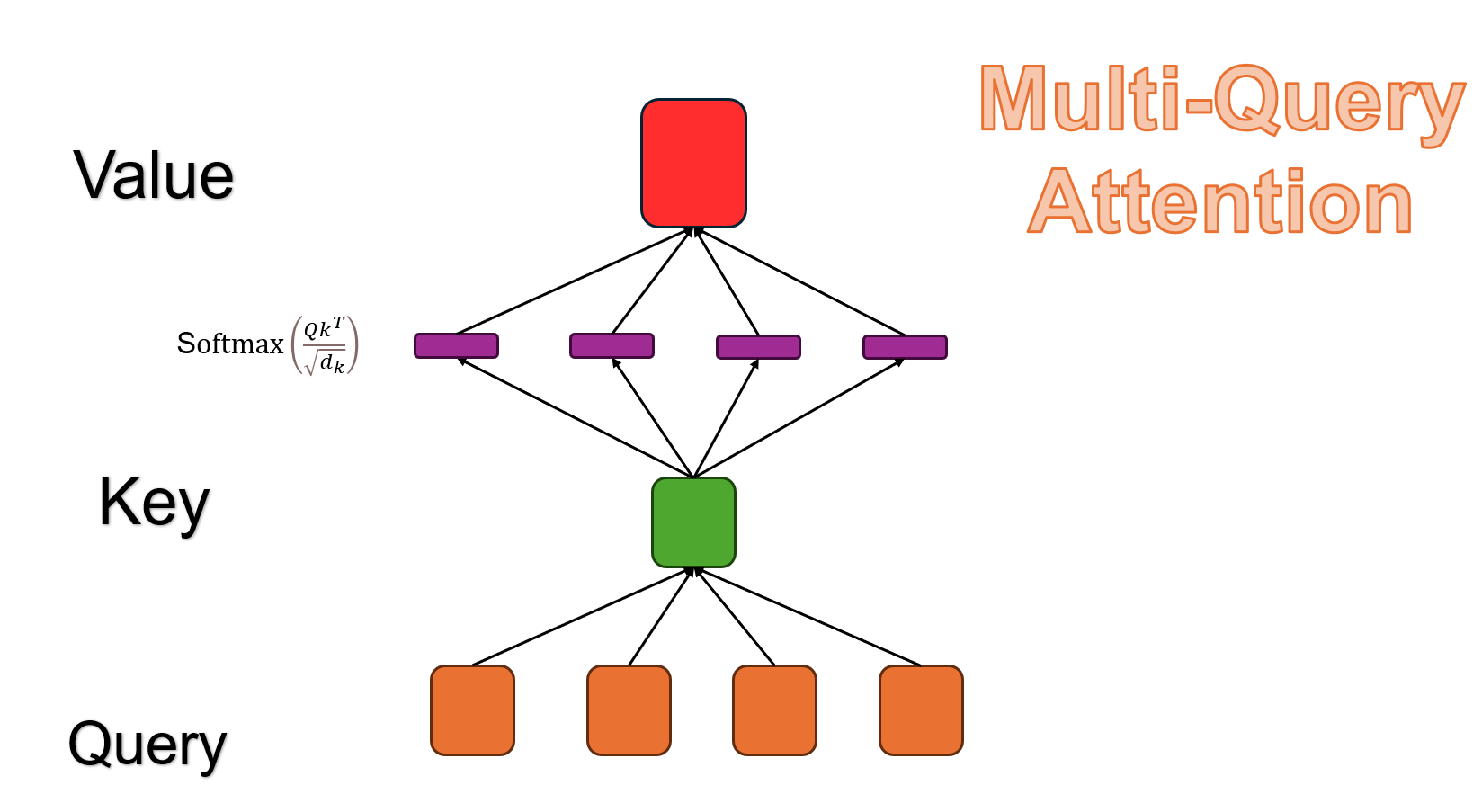


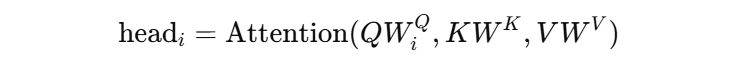
Image 7. Multi-Query architecture

Each attention head can **focus on different aspects of the input** such as syntactic structure, semantic meaning, or positional relationships.

A group of tables with numbers and symbols

AI-generated content may be incorrect.

Image 8.Multi head with stand alone Query



Where :

* i is the index number of head.
* Wi is learned projection matrices for head i.

Then all of these head will be concated together to create a new attention that have all information of all aspect.



WO  is the output projection matrix after concatenation.

#### Complexity

Multi-Query Attention (MQA) is an optimized variant of multi-head attention, where each attention head has a unique Query projection, but all heads **share the same Key and Value matrices**. This significantly reduces memory and computational cost, particularly during inference in large-scale models.

Assuming:

* n: sequence length
* d: embedding dimension
* h: number of heads

The attention computation includes:

1. **Projecting Queries (separately for each head)**  
   Each head computes its own Query:

***Qi = X for i = 1,...,h***

Complexity: ***O(n . d . h)***

1. **Projecting Keys and Values (shared across heads)**  
   Only one shared projection for Key and one for Value:

***K = XWK , V = XWV***

Complexity: ***O(n . d)***

1. **Computing Attention Scores**  
   Each head computes QiKT Rnxn

Complexity: ***O(h . n2)***

1. **Applying attention to Values**  
   Use attention weights to compute weighted sum over shared Value  
   Complexity: ***O(h . n . d)***

Putting it all together, the overall complexity of MQA is:

Compared to standard multi-head attention ***O(h . n2 . d)***, MQA reduces memory usage and inference time by avoiding redundant Key and Value computations across heads.

#### Advantages and Disadvantages

* ***Advantages:*** by sharing Key and Value projections across all heads, MQA significantly reduces both memory usage and computation time compared to standard Multi-Head Attention (MHA), especially during inference on long sequences. Although Keys and Values are shared, each head still has its own Query projection. This allows MQA to capture different contextual aspects of the input sequence across heads, like traditional MHA.
* ***Disadvantages:*** Sharing the same Key and Value matrices across heads may limit the diversity of information each head can access, potentially reducing model expressiveness compared to full MHA. While MQA reduces per-head computation, it does **not solve the O(n2)** bottleneck in attention score computation, which remains a challenge for very long sequences.

## GQA

#### Mechanism and Idea

Grouped-Query Attention (GQA) is a hybrid between Multi-Head Attention (MHA) and Multi-Query Attention (MQA). In GQA, attention heads are divided into groups, where each group shares the same Key and Value projections, but each head within the group maintains its own Query projection.

This design offers a balance between computational efficiency and representational diversity, making it particularly suitable for large models that require faster inference without fully sacrificing attention quality.

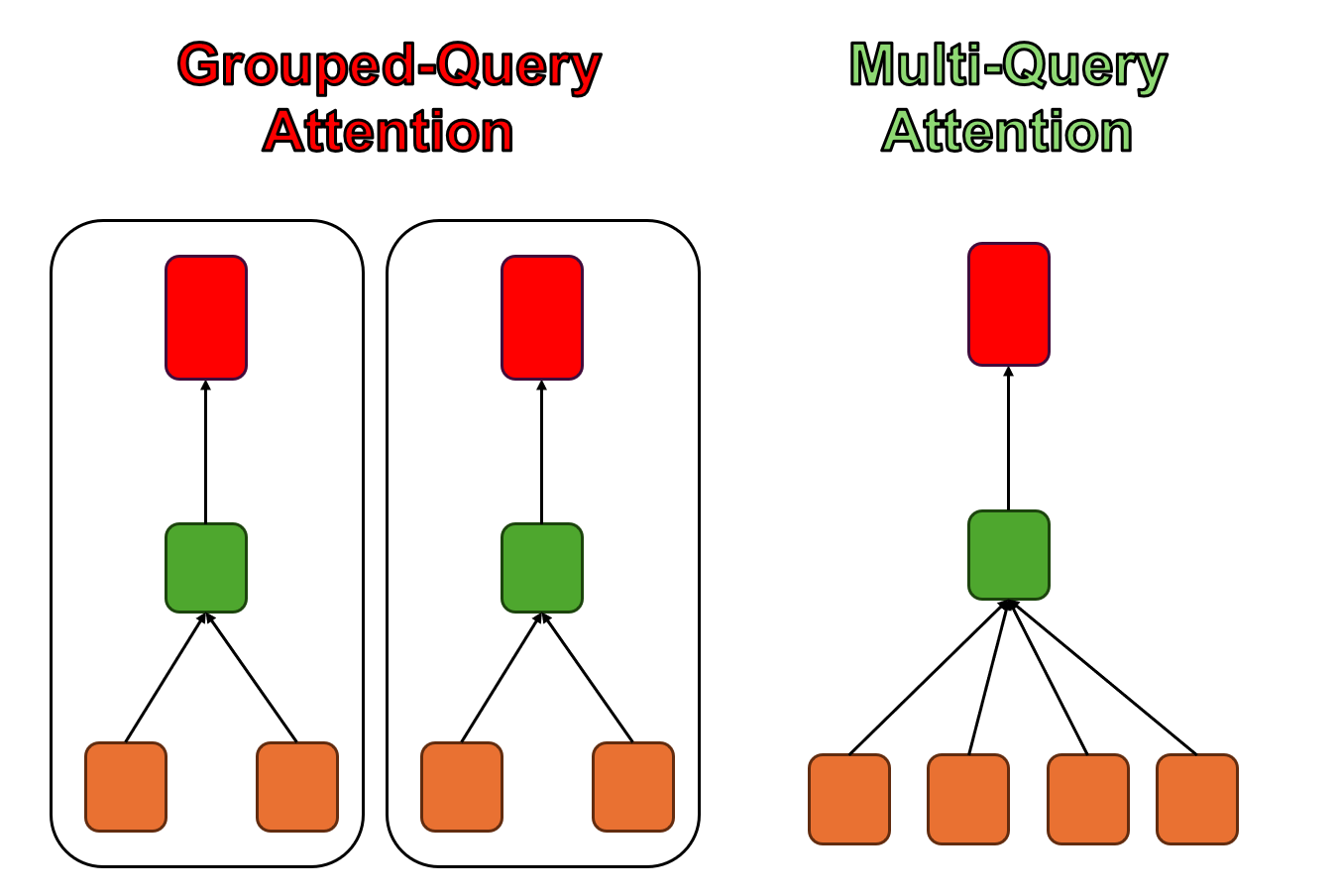


Image 9.Grouped - Query Attention architecture

Grouped-Query Attention (GQA) provides a flexible design that interpolates between Multi-Query Attention (MQA) and Multi-Head Attention (MHA).  
Specifically, if the number of groups is set to **1**, GQA becomes equivalent to MQA, where all heads share the same Key and Value.

On the other hand, if the number of groups equals the number of heads (i.e., one group per head), GQA becomes equivalent to standard MHA, where each head has its own Key and Value projections.

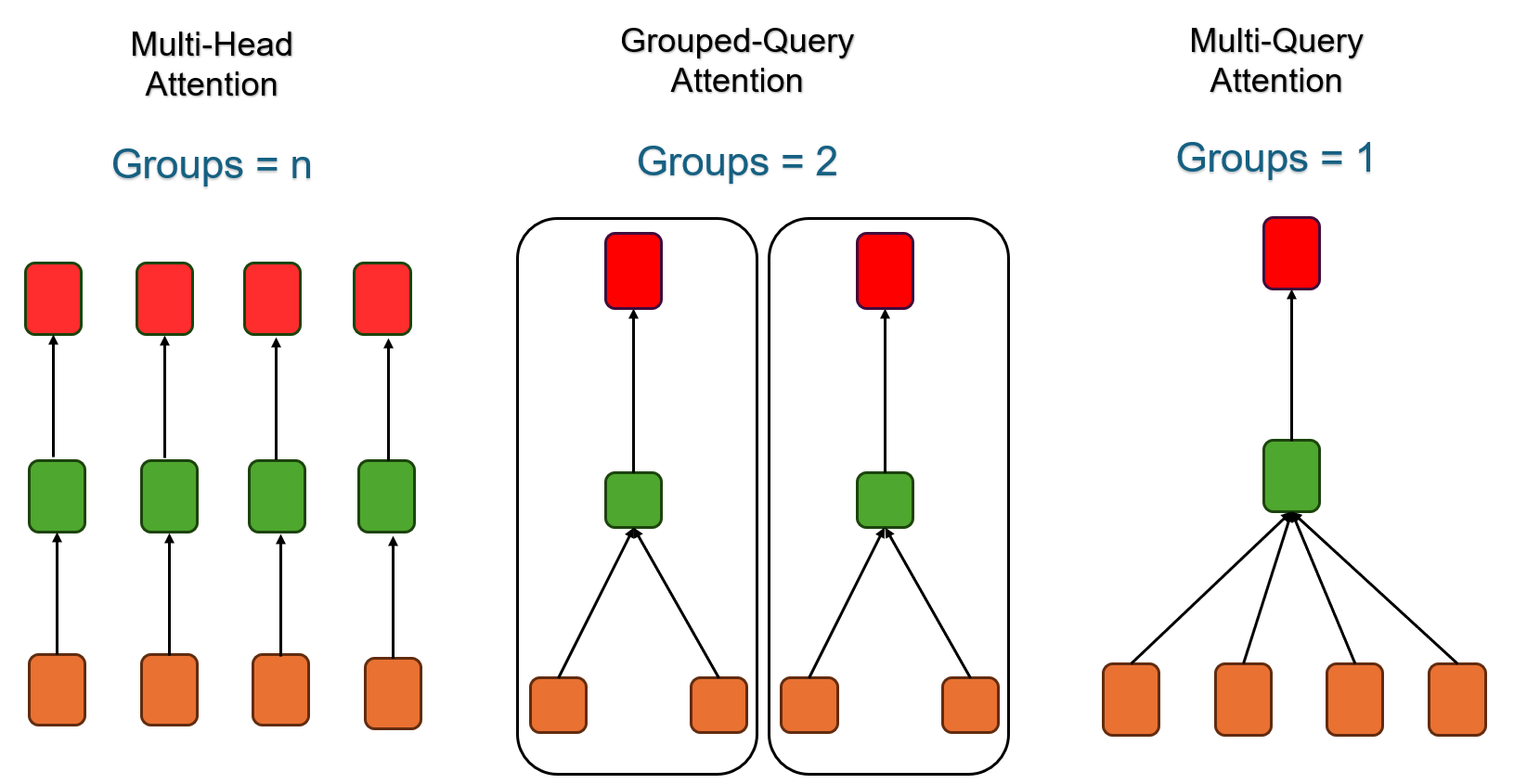


Image 10. GQA adjustment

Let:

* h: number of attention heads
* g: number of key-value groups (where g < h)
* n: sequence length
* d: embedding dimension

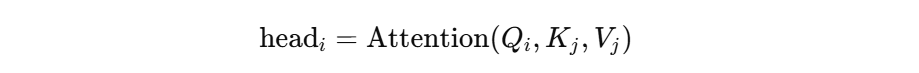
In GQA:

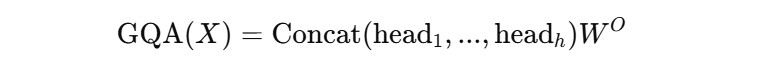
* Similar to MQA each **head** has its own **Query** projection:
* The **Key** and **Value** are shared **within each group**, not per head.  
  That is, for group j=1,...,g, we compute:

***Kj = X , V = X***

* Each head i is assigned to a group j, and uses its own Qi​ but **shared** Kj,Vj​.

Then, the attention output for each head i is:

Finally, as in standard attention:



#### Complexity

 Key/Value projections are computed once per group → ***O(g . n . d)***

 Query projections are still per head → ***O(h . n . d)***

 Attention score computation per head → ***O(h . n2)***

**Overall complexity**:

*This lies between the complexity of MHA and MQA:*

***OMQA < OGQA < OMHA***

#### Advantages and Disadvantages

* ***Advantages:*** by sharing Keys and Values across groups rather than individual heads, GQA reduces memory and compute cost compared to full MHA. Unlike MQA, GQA allows **multiple sets of Key/Value**, giving the model more flexibility and representation power across attention heads and the group size g can be tuned to find a good balance between speed and quality, depending on model size and task complexity.
* ***Disadvantages:*** Requires choosing a suitable number of groups g, which introduces a **tuning hyperparameter**. Like MHA and MQA, GQA still computes full attention scores → maintains ***O(h . n2)*** dependency on sequence length.

## Flash Attention

FlashAttention is an optimized implementation of the self-attention mechanism that significantly reduces memory usage and computation time by leveraging fused kernels and tiling techniques on modern hardware (e.g., GPUs).

Unlike architectural variants (like MHA or GQA), FlashAttention does not change the mathematical formulation of attention. Instead, it focuses purely on improving the efficiency of implementation.

A diagram of a computer

AI-generated content may be incorrect.

Image 11. Self-attention compute process on GPU

**Image 11** illustrates the standard self-attention process, which performs three separate steps:

1. computing attention scores (S = QKT),
2. applying softmax,
3. computing the final output (O = AV).

Each step involves loading and writing large intermediate matrices (Q, K, S, A, V) from high-bandwidth memory (HBM), causing **frequent memory accesses** and significant performance bottlenecks.

A diagram of a computer hardware system

AI-generated content may be incorrect.

Image 12. Memory Hierachy

The core idea behind **FlashAttention** is to **minimize the cost and latency of memory access** by maximizing computation within **GPU on-chip memory (SRAM)**. As shown in *Image 12*, GPU SRAM provides extremely high bandwidth (up to 19 TB/s) but has very limited capacity, whereas GPU HBM offers larger storage (e.g., 40 GB) at much lower bandwidth (1.5 TB/s). Standard self-attention mechanisms frequently read and write large intermediate matrices (such as attention scores and softmax outputs) to HBM, causing significant performance bottlenecks.

FlashAttention overcomes this limitation by applying **tiling, online softmax** and **kernel fusion**, enabling attention computation to be carried out **block by block entirely in SRAM** without storing large intermediate results. This approach greatly reduces memory traffic, improves throughput, and allows attention to scale efficiently to longer sequences, while producing the same output as traditional self-attention.

* *Tiling :* By applying an **I/O-aware algorithm**, the **tiling** technique divides large input data into smaller blocks that fit entirely within fast on-chip memory (such as GPU SRAM). This allows each block to be processed independently without reading or writing intermediate results to slower memory like HBM. As a result, tiling **minimizes memory traffic**, reduces latency, and ensures that computations remain efficient even when working with long sequences or large attention matrices.

*Example : with two 4x4 matrix dot product.*

A screenshot of a math game

AI-generated content may be incorrect.

A white paper with black arrows and numbers

AI-generated content may be incorrect.

Image 13. Tiling processs

* *Online softmax :* In FlashAttention, the standard softmax operation cannot be directly applied because the attention scores are computed in **blocks (tiles)** and the **entire score vector is not available at once**. Traditional softmax requires full access to a row to compute both the maximum and the normalization term, which is incompatible with block-wise computation in limited on-chip memory.

To address this, **online softmax** is used, a streaming algorithm that incrementally computes the maximum and the normalization sum as each tile is processed. This enables softmax to be computed **without storing the full vector in memory**. In addition, **safe softmax** is applied alongside to ensure **numerical stability**, especially when working with large values, by subtracting the running maximum from each element before exponentiation.

In Online softmax there are 2 step to perform:

* 1. **Streaming max and sum :** 
     + **Max :** *m = max(m, xi)*
     + **Sum :** *s = s + exi-m*
  2. **Normalize :**

***Online Softmax(xi) =***

* *Safe softmax :* By applying the **log-sum-exp transformation**, values are kept smaller to prevent numerical overflow.

***Safe Softmax(xi) =***

Subtracting max(x) does not change the result of SoftMax (since it affects both numerator and denominator equally), but it ensures:

* xi – max(x) ≤ 0 to avoid overflow when computing ex.
* improves **numerical stability** when working with very large or very small negative numbers.
* *Fusion kernel :*  With the help of **online SoftMax**, Flash Attention can compute all three steps — attention score computation (S = QKT), SoftMax (A), and value weighting (O = AV) — in a **single fused kernel**, as illustrated in *Image 11*.  
  Unlike standard self-attention, which performs these steps separately, Flash Attention executes them in one continuous pipeline.  
  This technique significantly **reduces both memory usage and computation time**, making it far more efficient than conventional self-attention.

#### Complexity

FlashAttention eliminates this overhead through **tiling**, **online softmax**, and **kernel fusion**, resulting in:

* **Reduced memory complexity:**

#### Advantages and Disadvantages

* **Advantages :** By sharing Keys and Values across groups rather than individual heads, Grouped-Query Attention (GQA) reduces memory and computation costs compared to full Multi-Head Attention (MHA). Unlike MQA, it supports multiple Key/Value sets, offering greater flexibility and representational power. The number of groups ggg can also be tuned to balance speed and performance based on the task.
* **Disadvantages :** It introduces an extra hyperparameter — the number of groups — which must be carefully selected. Moreover, like MHA and MQA, GQA still computes full attention scores, maintaining a quadratic **O(n2)** time complexity with respect to sequence length.

# PART 2 – BUILDING A MODEL TO SOLVE THE PROBLEM

**2.1 Introduction to OCR model**

Optical Character Recognition (OCR) aims to extract textual content from images. In recent deep learning-based approaches, a hybrid architecture that combines Convolutional Neural Networks (CNNs) with Transformer Decoders has shown state-of-the-art performance in image-to-text tasks. This section explains in detail how each component contributes to the OCR pipeline, particularly focusing on the use of **ResNet-50 pretrained model** as the CNN backbone and the **Transformer Decoder with Multi-Head Attention** for text generation.

At first, we need to preprocess the data, such as cropping, applying blur reduction, or enhancing image quality. These steps help ensure that the input images are clean and standardized before being fed into the OCR model. After preprocessing, the images are tokenized and converted into embeddings, followed by positional encoding and attention mechanisms within the decoder-only transformer to generate the final text predictions.

**2.2 CNN for Visual Feature Extraction**

In this OCR system, a **Convolutional Neural Network (CNN)**—specifically, **ResNet-50**—is used to extract high-level visual features from the input image. The image is first preprocessed (resized, normalized), then passed through ResNet-50, a deep residual network with skip connections that allow for stable training and efficient learning.

+ The output of ResNet-50 is a **2048-dimensional feature vector**, representing the entire image’s abstract visual semantics.

+ This feature vector is passed through a **linear projection layer** to reduce its dimensionality to **512**, aligning with the expected input size (d\_model = 512) of the Transformer Decoder.

In essence, the CNN transforms raw pixel data into a dense, informative representation that summarizes the textual content of the image in vector space. This visual embedding serves as the "memory" input for the Transformer Decoder.

**2.3 Transformer Decoder for text generation**

The **Transformer Decoder** is responsible for generating the target text sequence, one character at a time. It is composed of several stacked **TransformerDecoderLayers**, each containing self-attention, cross-attention, and feedforward sublayers.

#### Key Components:

**+ Token Embedding**: Each character in the target sequence is embedded into a 512-dimensional vector.

**+ Positional Encoding**: Since Transformers lack inherent sequence ordering, sinusoidal positional encodings are added to retain the order of characters.

**+ Autoregressive Decoding**: At each time step, the decoder uses only the previously generated tokens to predict the next token.

This design allows the model to generate sequences of arbitrary length, adjusting its output dynamically based on the image context and previously decoded characters.

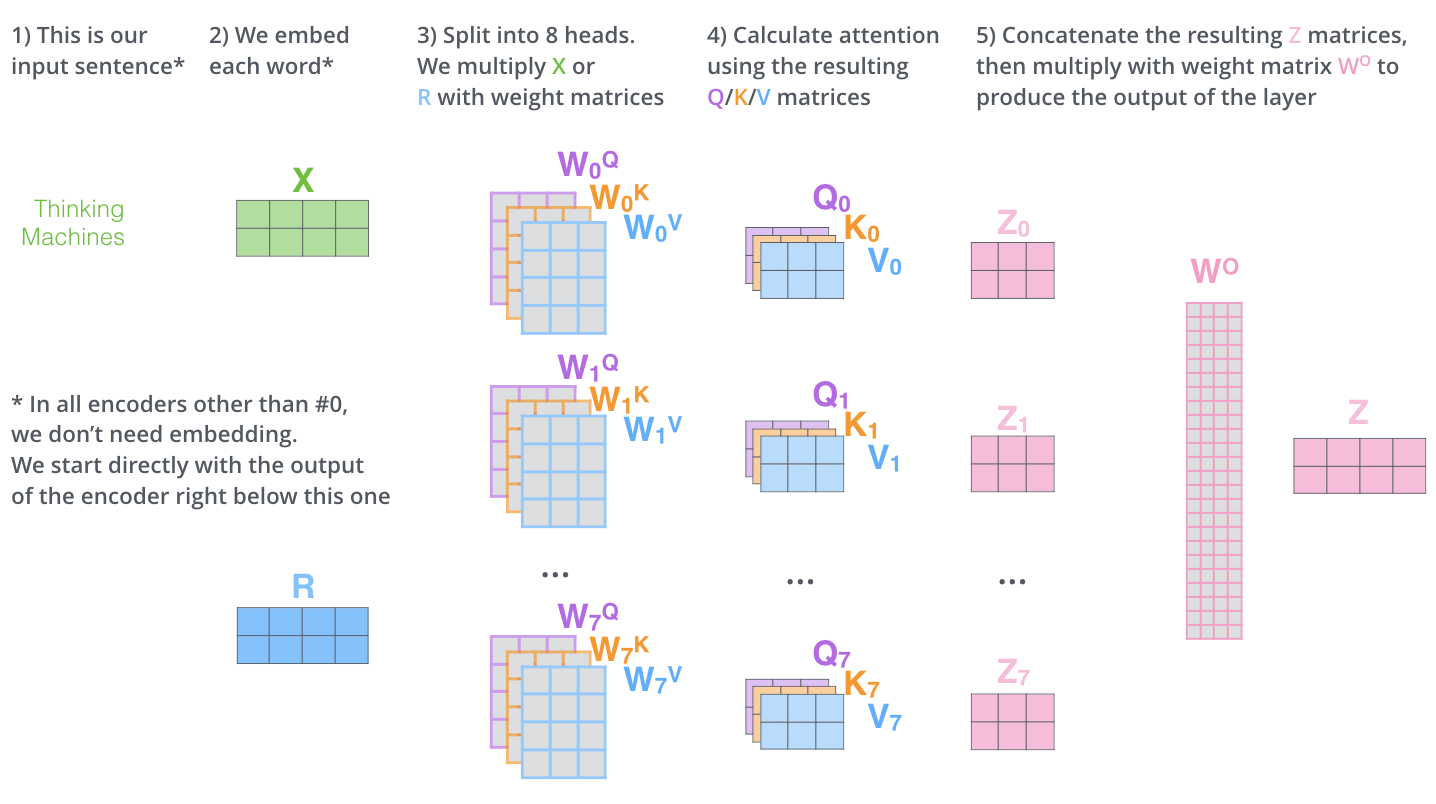


Image 14: Transformer decoder

## 2.4 Attention Mechanism between Visual Features and Text Decoder

The core strength of the architecture lies in its **attention mechanism**, which governs how the model aligns the extracted visual features with the generated text.

### 2.4.1. ****Self-Attention**** (Decoder Only):

Enables the decoder to attend to previously generated characters.

Helps the model understand context within the target sequence.

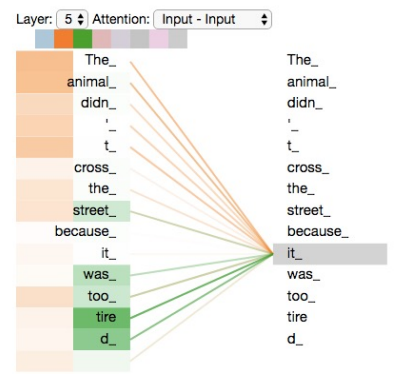


Image 15: Self attention

### 2.4.2. ****Cross-Attention**** (Decoder-to-CNN Features):

The decoder queries the CNN feature vector (memory) at each time step.

It computes a relevance score for each visual embedding dimension and weighs it accordingly.

This process guides the decoder to focus on the most relevant parts of the visual feature when generating a specific character.

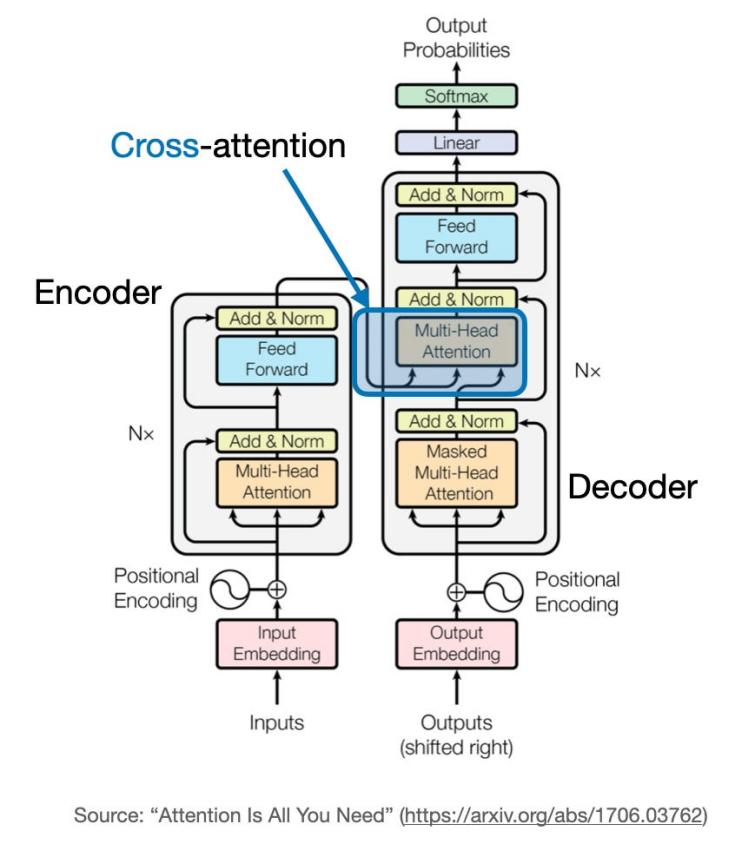


Image 16: Cross attention

### 2.4.3. ****Multi-Head Attention****:

Each head learns to attend to different aspects of the visual representation or sequence structure.

This parallel attention mechanism enables the decoder to capture complex and diverse alignment patterns between image and text.

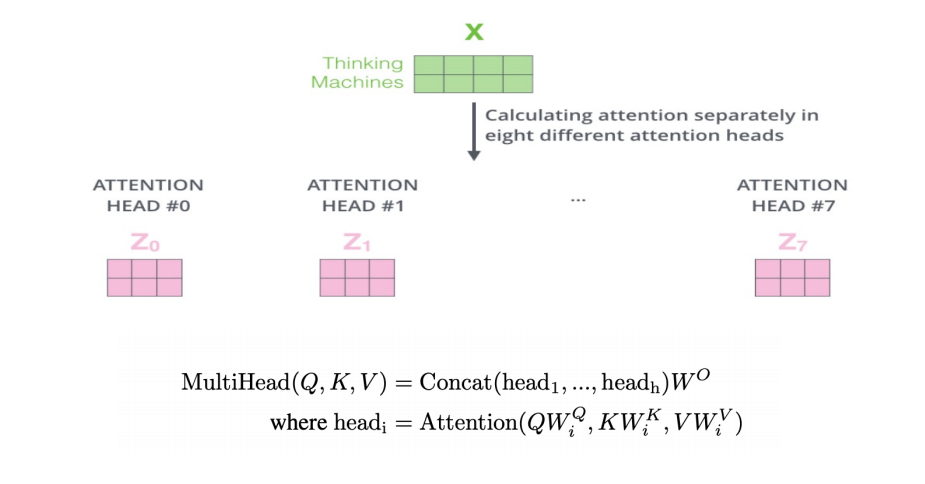


Image 17: Multi head attention

### Summary:

The combination of **CNN and Transformer Decoder** forms an effective architecture for OCR tasks:

+ The **CNN (ResNet-50)** acts as the visual encoder, compressing spatial image information into a compact feature vector.

+ The **Transformer Decoder** autoregressively generates text, relying on both previous output tokens and visual context.

+ The **attention mechanism**, particularly the **cross-attention** with multiple heads, allows the model to intelligently align the decoded text with the relevant parts of the image representation.

This architecture proves highly adaptable to real-world OCR challenges, including varying fonts, orientations, and noisy backgrounds.

**2.5 Build a model to extract text from image**

**2.5.1 Character Level Tokenizer**

In any sequence-to-sequence OCR task, converting textual data into a machine-readable numerical format is essential for training and inference. This section introduces and explains the structure, design rationale, and role of the CharTokenizer class, which handles both the encoding and decoding of character sequences for our OCR model.

**2.5.1.1 Design and Functionality of the Tokenizer**

**Vocabulary Creation**: The tokenizer constructs a vocabulary by extracting all unique characters found in the training dataset. This includes both alphanumeric characters and special symbols. Special tokens such as the start-of-sequence (<SOS>), end-of-sequence (<EOS>), and padding (<PAD>) tokens are also added to the vocabulary.

· **Special Tokens**:

+ <SOS>: Marks the beginning of a sequence and is used to indicate the start of text input during inference.

+ <EOS>: Marks the end of a sequence, signaling when the model should stop generating text.

+ <PAD>: Used to pad shorter sequences to a uniform length, ensuring that all sequences in a batch are of equal length.

#### ****2.5.1.2 Encoding Process****

**+ Text to Indices**: The encode method converts a given text string into a sequence of integer indices based on the tokenizer’s vocabulary. The text is first prefixed with the <SOS> token and suffixed with the <EOS> token.

**+ Padding**: If the encoded sequence is shorter than the specified max\_len, the remaining positions are filled with the <PAD> token. This ensures that all sequences have a consistent length, which is crucial for batch processing in deep learning models.

#### ****2.5.1.3 Decoding Process****

**Indices to Text**: After the model generates a sequence of indices, the decode method is used to convert the numerical representation back into a readable string. During this process, the special tokens (<SOS>, <EOS>, and <PAD>) are removed, and the remaining characters are joined together to form the final output text.

#### ****2.5.1.4 Vocabulary Size****

The vocab\_size property gives the total number of unique tokens in the tokenizer's vocabulary. This is an important parameter because it determines the size of the model’s embedding layer, which must accommodate all possible token values.

#### ****2.5.1.5. Suitability for OCR Tasks****

**Character-Level Tokenization**: For OCR tasks, character-level tokenization is beneficial because it enables the model to handle various types of text, including uncommon words, misspelled characters, and noisy data. The model doesn’t need to worry about unknown words since it operates at the granularity of individual characters.

**Handling Fragmented or Structured Text**: OCR outputs often include fragmented or distorted text, such as partial words or symbols. Character-level tokenization helps the model better handle such cases.

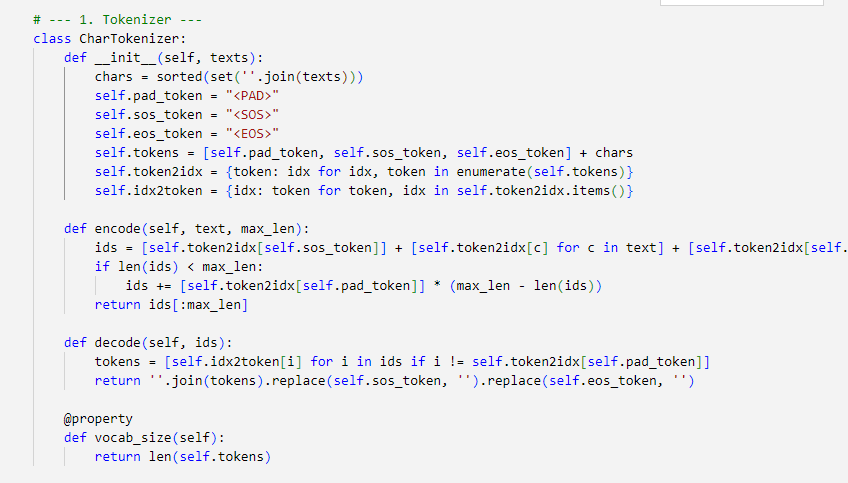


Image 18: CharTokenizer

**2.5.2 Create dataset class**

The OCRDataset class is a crucial component in preparing the dataset for the OCR model. It serves as a custom dataset class that is compatible with PyTorch’s DataLoader, facilitating the loading and transformation of image-text pairs for training.

**Functionality:**

**Data Loading:** The class takes in a list of sample pairs (image filename and corresponding text), and for each sample, it loads the image from the specified directory and retrieves the corresponding text label.

**Image Transformation:** To standardize the image input, the class applies a series of transformations. These include resizing the image to a fixed dimension (64x256), converting it to a tensor format, and normalizing the pixel values. The normalization ensures that the model receives inputs that are centered around zero with a standard deviation of one, which typically aids in training stability and model convergence.

**Text Tokenization:** The text labels associated with each image are tokenized using the provided tokenizer. The tokenizer converts each character of the text into a corresponding integer index, padding the sequence if necessary to a predefined maximum length (in this case, 64). This step allows the model to process the text in a format it can understand, with padding ensuring consistent sequence lengths across different samples.

**Output:** The class returns both the processed image and the tokenized label as a tuple. This format is directly compatible with PyTorch’s training loop, allowing efficient batching and feeding into the model.

**Purpose in the OCR Pipeline:**  
The OCRDataset class plays a key role in the data preprocessing pipeline, ensuring that both the images and their corresponding text labels are transformed into the necessary format for model training. By combining image transformation and text tokenization, it simplifies the workflow and allows for efficient handling of data in deep learning tasks.

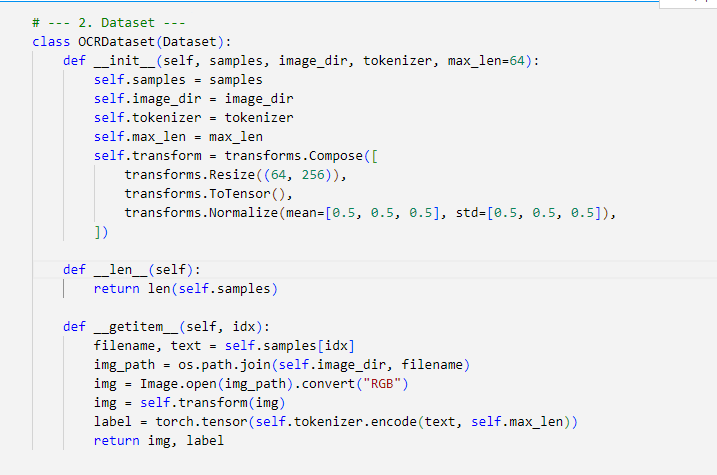


Image 19: Create Dataset

**2.5.3 Positional Encoding**

The PositionalEncoding class is a key component in incorporating positional information into the input embeddings of a Transformer model. Since Transformers do not inherently process sequential data in a temporal order (unlike RNNs or CNNs), positional encoding is used to inject information about the relative or absolute position of tokens in a sequence.

**Functionality:**

**Initialization:** The class initializes a matrix pe of shape (max\_len, d\_model), where max\_len is the maximum sequence length, and d\_model is the dimensionality of the input embeddings. The matrix is filled with sine and cosine functions at different wavelengths to encode positional information. The sine and cosine functions are applied alternatively to even and odd positions in the embedding vector, which ensures that each position in the sequence has a unique representation.

**Positional Encoding Calculation:** The positional encoding is computed using the formula:

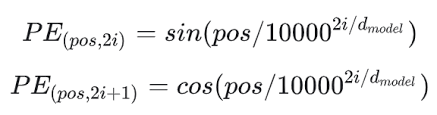


Image 20: Formula of positional encoding

Where pos is the position and i is the dimension index. The sine function is applied to the even indices and the cosine function to the odd indices.

**Buffer Registration:** The computed positional encodings are registered as a buffer within the model, which ensures they are not treated as model parameters and are not updated during backpropagation.

**Forward Pass:** During the forward pass, the positional encoding is added to the input tensor x. The encoded positional information is combined with the token embeddings to provide each token in the sequence with both its content (from the token embedding) and its position in the sequence.

**Purpose in the OCR Pipeline:**  
The PositionalEncoding class is critical in enabling the Transformer model to handle sequential data, such as text. By adding unique positional encodings to each token, the model can understand the order of the characters in the input sequence, which is essential for accurate text recognition. Without positional encoding, the Transformer model would treat all tokens as independent, which would significantly hinder its performance on sequential tasks like OCR.

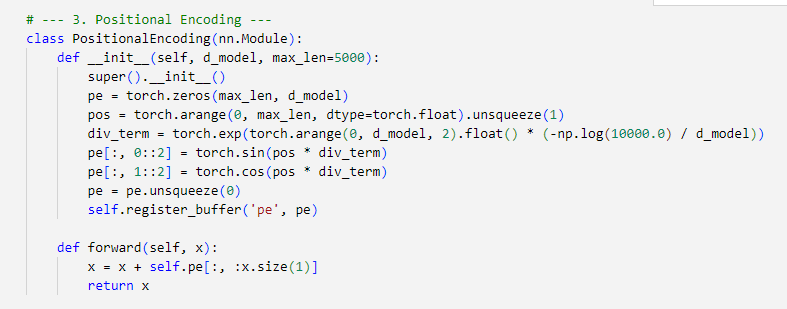


Image 21: Positional Encoding

**2.5.4 OCR model using CNN + Transformer Decoder**

Here are model’s architecture:

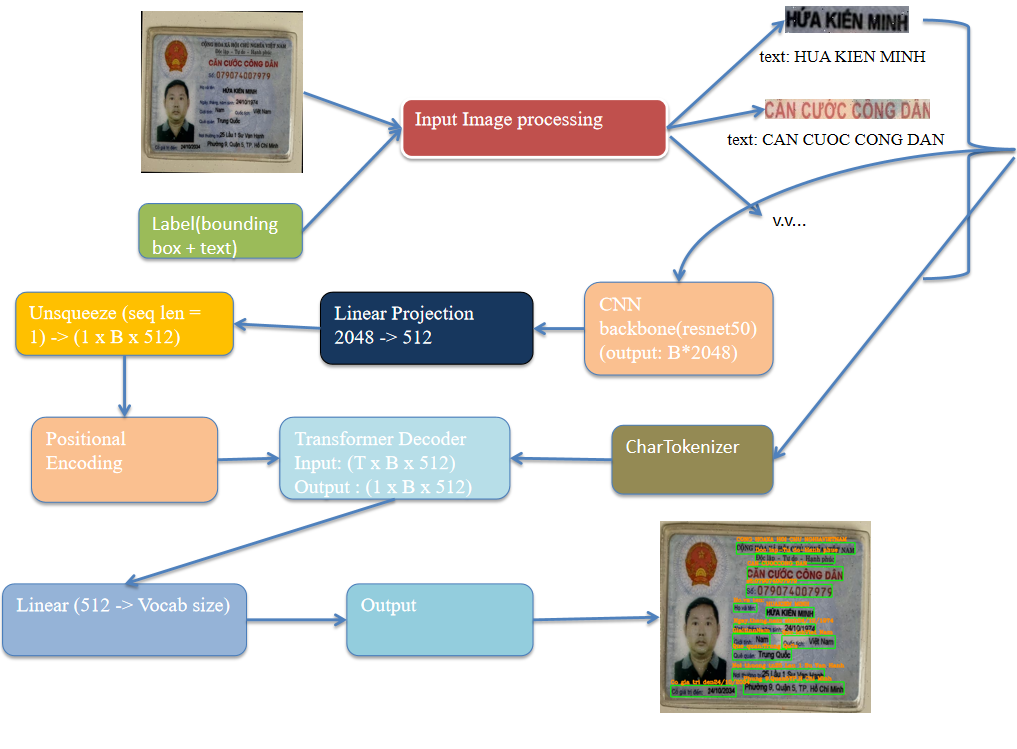


Image 22: Model's architecture

**2.5.4.1 Overview of the architecture:**

The OCR model is based on two main components: **CNN Backbone** and **Transformer Decoder**. Specifically:

**+ CNN Backbone (ResNet-50)**: Used to extract features from the input image. ResNet-50, a deep network with 50 layers, is a powerful model pretrained on large image datasets. It helps extract crucial features from images without processing each pixel manually.

**+ Transformer Decoder**: This part is used to decode the extracted features into a sequence of text. Transformer is a powerful architecture for sequence processing, with an attention mechanism that helps the model learn long-term relationships between elements in the sequence.

**2.5.4.2 Attention Mechanism in the model**

The **Attention** mechanism in the Transformer plays a crucial role by allowing the model to "attend" to different parts of the input. In this OCR model, Attention helps the model focus on relevant areas of the image when decoding the features into text.

**+ Self-Attention**: During the decoding process, each word in the sequence being predicted attends to other words in the sequence, allowing the model to understand the relationships between words.

**+ Multi-Head Attention**: To improve the learning of complex relationships, the model uses multiple "attention heads", each focusing on different parts of the input, enabling the model to learn from different perspectives.

**2.5.4.3 CNN Backbone(using resnet-50)**

The CNN (Convolutional Neural Networks) in this model is used as a key component for extracting spatial features from the input image. CNNs are highly effective in recognizing image features, including objects, shapes, textures, and other signals that may appear in text.

**ResNet-50**: ResNet (Residual Network) is a type of deep CNN designed to address the vanishing gradient problem in very deep networks. ResNet-50 is a version of ResNet with 50 layers, allowing the model to learn complex features from images without encountering training issues.

**Deep Network with Residual Connections**: ResNet uses residual blocks that allow gradients to flow more efficiently through deep networks, enabling deeper networks to be trained without the gradient vanishing problem. These residual connections ensure that important signals are retained during training.

**Feature Extraction**: After passing through the CNN, the input image is transformed into a high-dimensional vector (2048 dimensions) representing the image's features. This reduces the complexity of processing individual pixels and instead allows the model to learn higher-level features, such as words or text patterns in the image.

**2.5.4.4 Transformer Decoder**

The Transformer Decoder is a critical part of the model, responsible for decoding the extracted features from CNN into text sequences. The Transformer architecture is particularly effective for sequence-to-sequence tasks, such as language translation and OCR, because it uses attention mechanisms to focus on different parts of the input when generating output.

At first, we need to understand about Masked, multihead attention.

“An attention function [maps] a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors. The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key.” - from paper title “ Attention is all you need”.

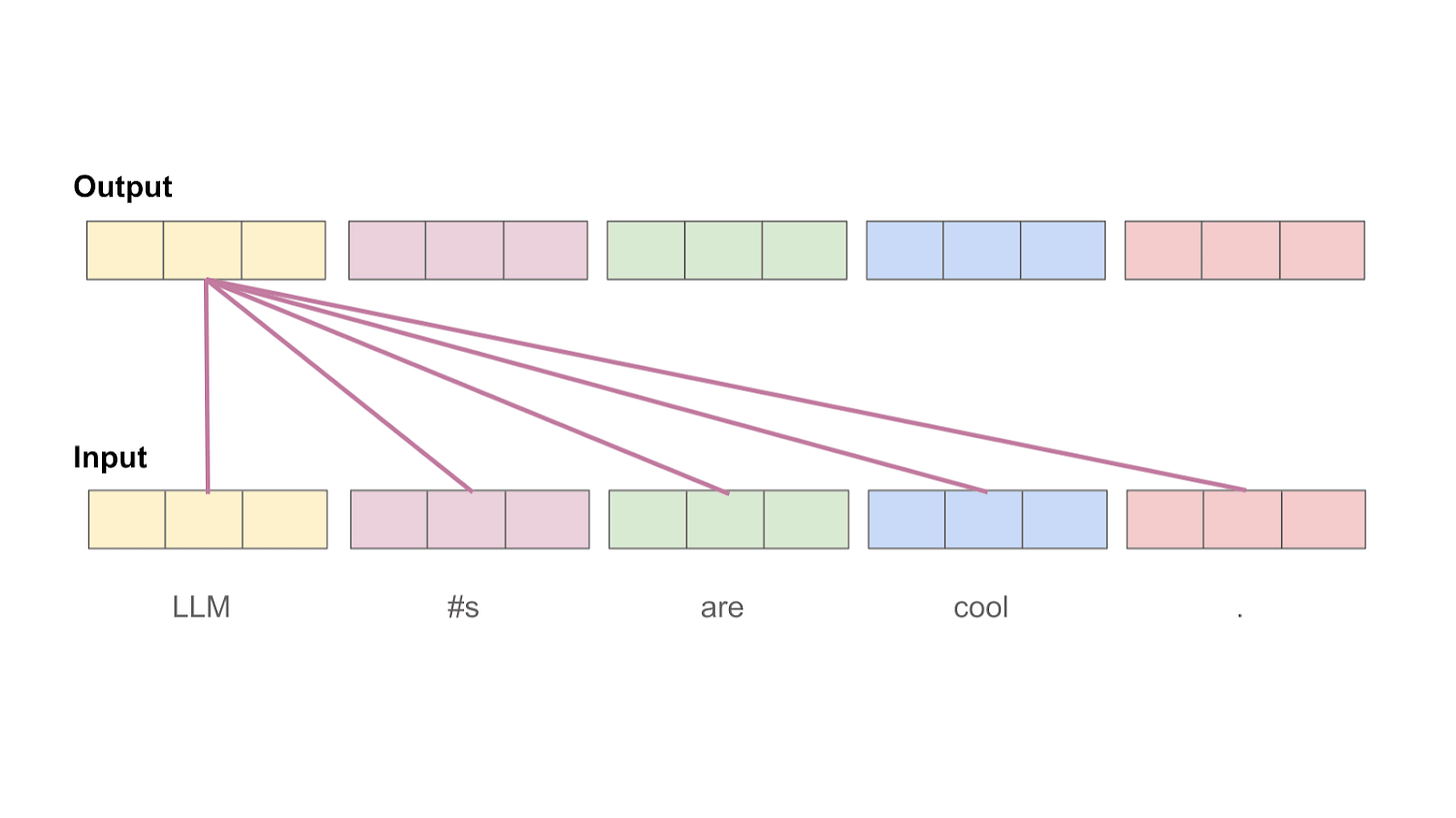


Image 23: self-attention

The input to a self-attention layer is a batch of token sequences, where each token is represented by an embedding vector. For an OCR model, where each token corresponds to a character or sub-word extracted from an image, the input tensor has the shape [B, T, d], where:

**+ B** is the batch size.

**+ T** is the length of each token sequence (corresponding to the number of characters in a word or sentence).

**+ d** is the dimensionality of the token embeddings.

For simplicity, let's consider a single sequence of tokens. The self-attention layer receives this sequence as a tensor of shape [1, T, d] and performs attention operations to determine how each token relates to every other token in the sequence. This same operation extends naturally to a batch of sequences with shape [B, T, d].

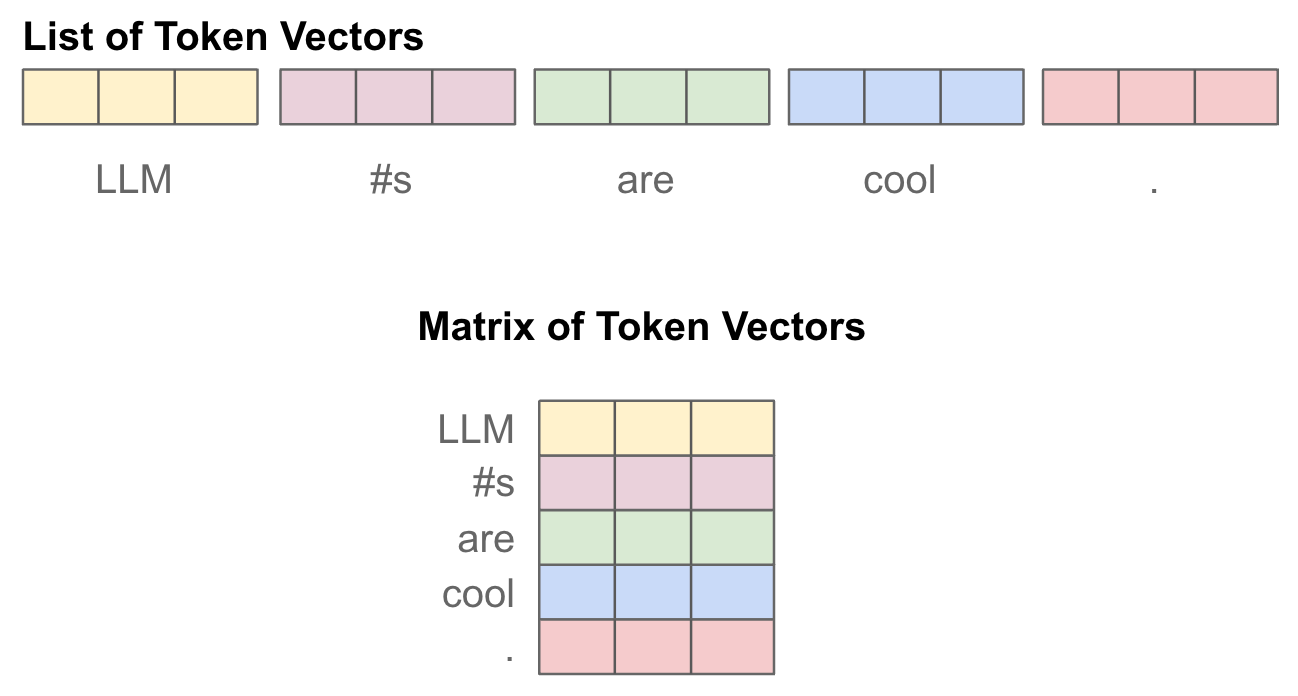


Image 24: Token vector

The first step of self-attention is to perform three separate (linear) projections of the token vectors in our input sequence, forming key, query, and value vector sequences. To do this, we have three weight matrices—corresponding to the key, query, and value projections—that are used to project each of the input token vectors, forming new sequences of transformed token vectors. Because we do this three times, we end up with three separate sequences of token vectors; see below.

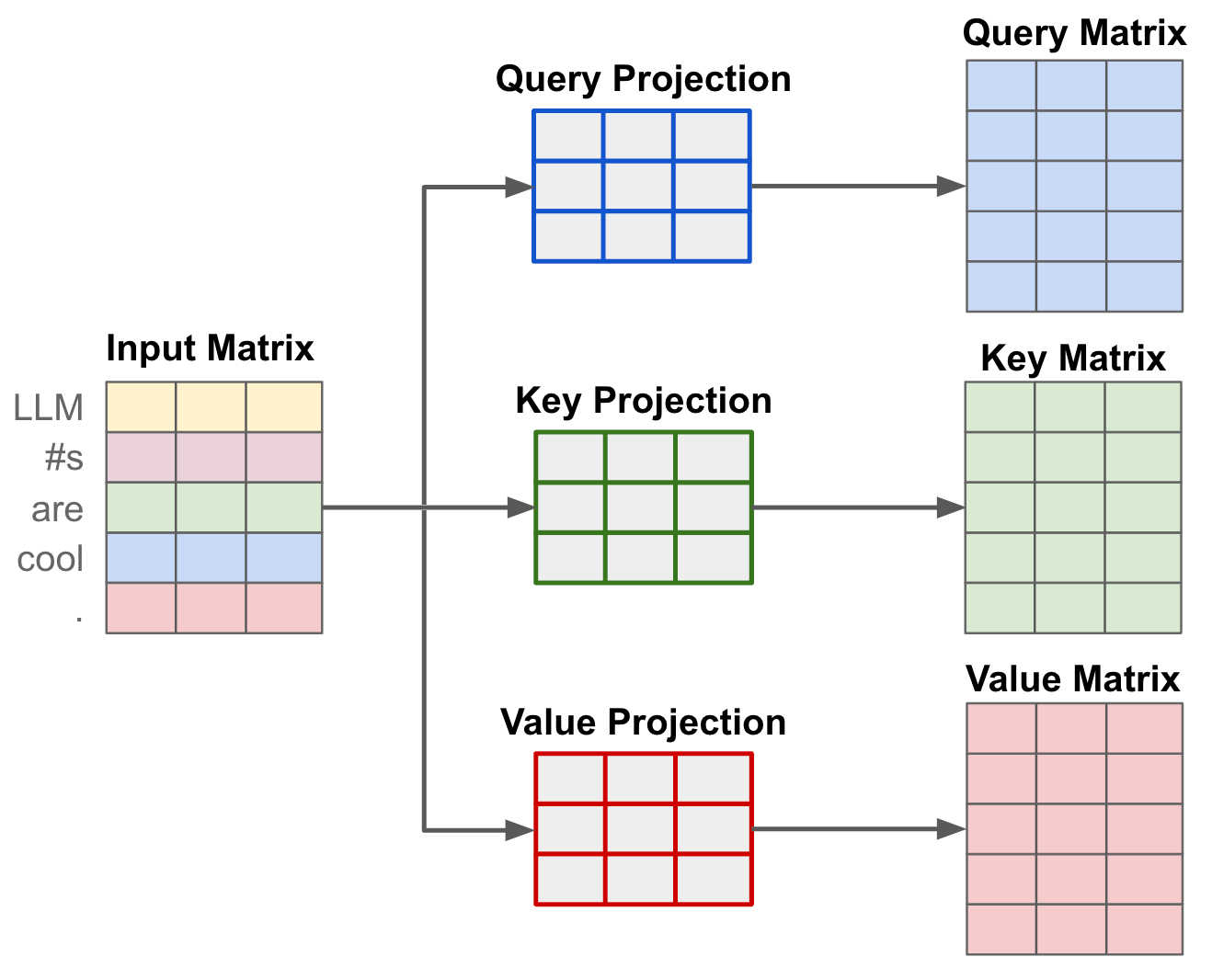


Image 25: First step of self-attention

After projecting the input, the attention scores are computed using the **key** and **query** vectors for each token in the sequence. For each pair of tokens [i, j] within the sequence, we calculate an attention score a[i,j]a[i, j]a[i,j] that determines how much token j should contribute to the new representation of token i.

To compute the attention score, we take the **dot product** of the query vector for token i and the key vector for token j. The result of this dot product is then scaled and passed through a softmax function to normalize the scores to the range [0, 1]. This process quantifies the relationship between tokens i and j in the sequence, indicating how much token j should be "attended to" when updating the representation of token i.

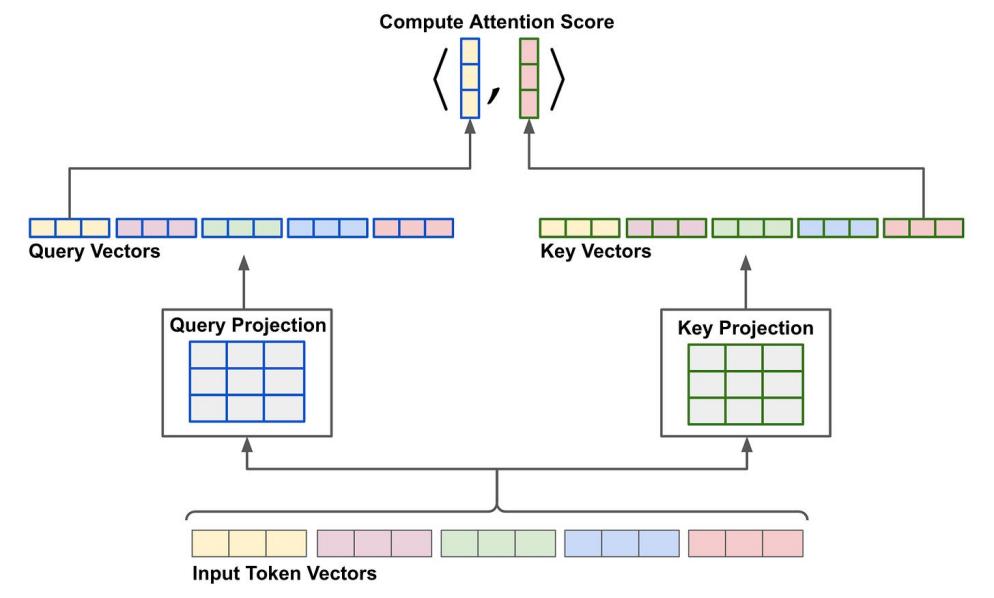


Image 26: Compute attention score

To efficiently compute all pairwise attention scores in a sequence, we stack the **query** and **key** vectors into two matrices and perform a matrix multiplication. Specifically, we multiply the **query matrix** (of shape [T, d]) with the **transposed key matrix** (of shape [d, T]), resulting in an **attention matrix** of size [T, T]. This matrix contains the attention scores for every pair of tokens [i, j] within the sequence.

Next, we scale the values in the attention matrix by dividing each score by the square root of the dimensionality ddd. This scaling helps improve training stability, as it prevents excessively large values in the attention scores during the early stages of training. After scaling, we apply the **softmax** operation to each row of the attention matrix. The softmax ensures that the attention scores for each token fall within the range [0, 1], forming a valid probability distribution that sums to 1 for each token.

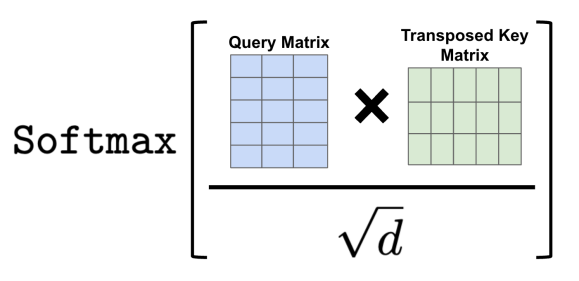


Image 27: Softmax formula

Once we have computed the attention scores, deriving the output of the **self-attention** mechanism is straightforward. The output for each token is simply a weighted combination of **value vectors**, where the weights correspond to the attention scores.

To compute this output efficiently for a batch, we stack all value vectors into a matrix and perform a matrix multiplication between the **attention matrix** (of shape [T, T]) and the **value matrix** (of shape [T, d]). This gives us the transformed representation for each token.

Notably, the self-attention mechanism preserves the size of the input sequence. For each token in the sequence, the model produces a transformed output vector of the same dimensionality ddd as the input token vectors. If we expand this matrix multiplication, we can see that each token’s output representation is essentially a weighted average of the value vectors, with the weights determined by the attention scores. This ensures that the output incorporates relevant information from all tokens based on their relationships as determined by the attention mechanism.

But we need to consider one thing, when we add more layer to neural network, the activations can grow very large or become very small. This lead to unstable gradients that hinder learning. I think an effective approach is normalization, which ensures that the activations are controlled and stable.Normalization techniques, such as **Batch Normalization** and **Layer Normalization**, prevent the intermediate activations from growing too large or shrinking too small during training. These methods stabilize the network and improve the model's ability to converge faster.I will show how Layer Normalization works in the context of transformer architecture.

Here's 2 most common forms of normalization works in the context of transformer architectures:

**+ Batch Normalization** normalizes across the batch dimension, which makes it suitable for tasks that involve large datasets with consistent batch sizes, but it isn't ideal for sequence models like transformers because the sequence length may vary.

**+ Layer Normalization**, on the other hand, normalizes the activations within each individual sequence (or token) across all the feature dimensions. This technique is particularly effective in models like transformers, where each input token in a sequence might require normalization independently of the batch size.

In my model I’ve use nn.TransformerDecoderLayer,this class will use Layer Normalization and this can be applied to both the **self-attention** and **feed-forward layers** to stabilize the activations after each operation. Typically, PyTorch applies Layer Normalization at the input and output of the attention and feed-forward sub-layers, helping to prevent gradient issues and improve training stability.

Here’s the standard equation for normalization

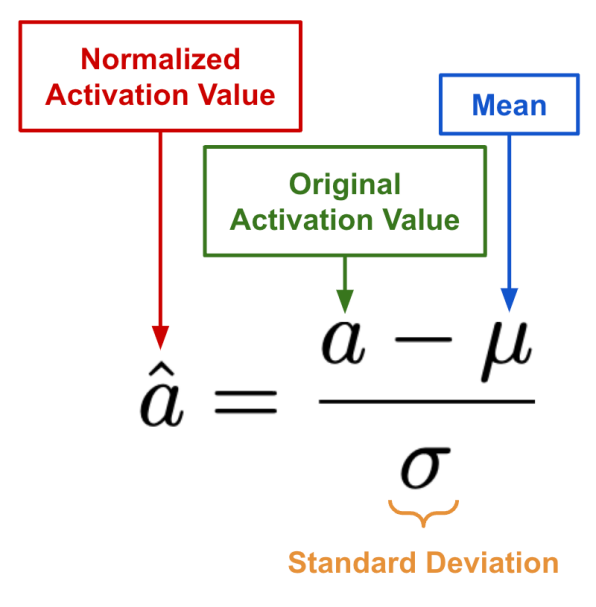


Image 28: Standard equation for normalization

And the picture below will show the differentiation of each Normalization forms.

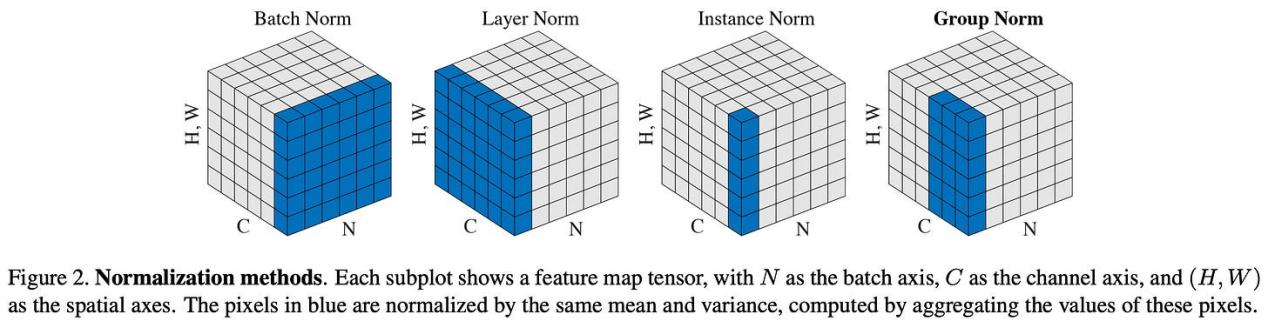


Image 29: Normalization

Layer Normalization (LayerNorm) is crucial for stabilizing training and improving performance in OCR models, especially those using CNN and Transformer Decoder. Unlike Batch Normalization, which depends on batch size, LayerNorm normalizes across the features of each individual sample, making it more suitable for OCR tasks where input sequences (text in images) can vary in length and structure. It helps prevent instability in the model, speeds up convergence, and improves generalization, especially in tasks that involve text recognition from images with varying fonts, sizes, or orientations. By normalizing the activations, LayerNorm enhances the model's ability to learn meaningful patterns from diverse input data, ultimately improving accuracy.

**FEED FORWARD TRANSFORMATION**

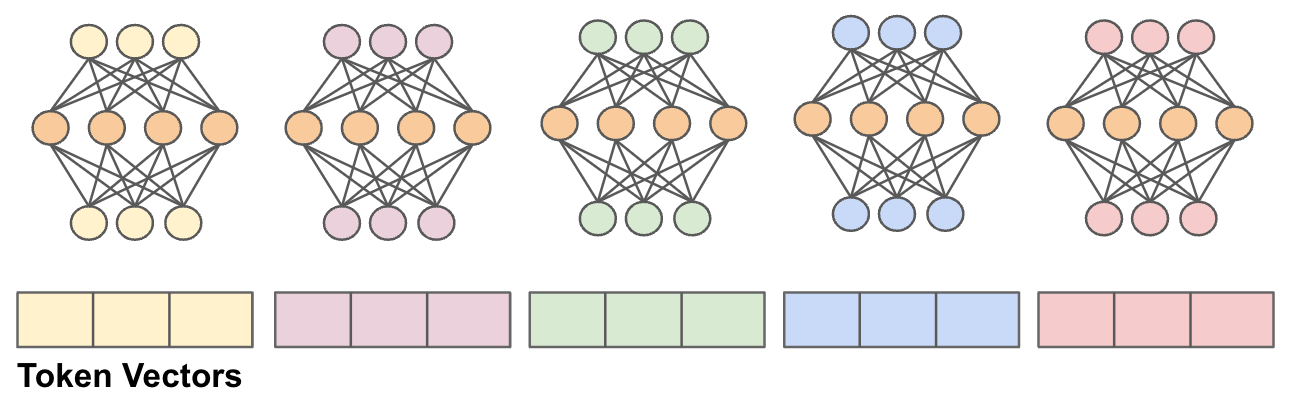


Image 30: Feed Forward

The feed-forward transformation in a decoder-only transformer block plays a critical role in processing each token individually. It passes each token vector through a small neural network consisting of two linear layers, with a non-linear activation function (like ReLU or GELU) applied in between. The hidden dimension of this neural network is typically larger than the input dimension of the token vector, often four times larger in models like GPT and GPT-2. This larger hidden dimension allows the model to learn more complex features, enhancing its capacity to represent intricate patterns within the input tokens. By applying this transformation, the model can better capture the relationships and nuances in the data, crucial for tasks like OCR where accurate text recognition and interpretation are essential.

Finally, I’m putting it together for better understanding of Transformer decoder block :

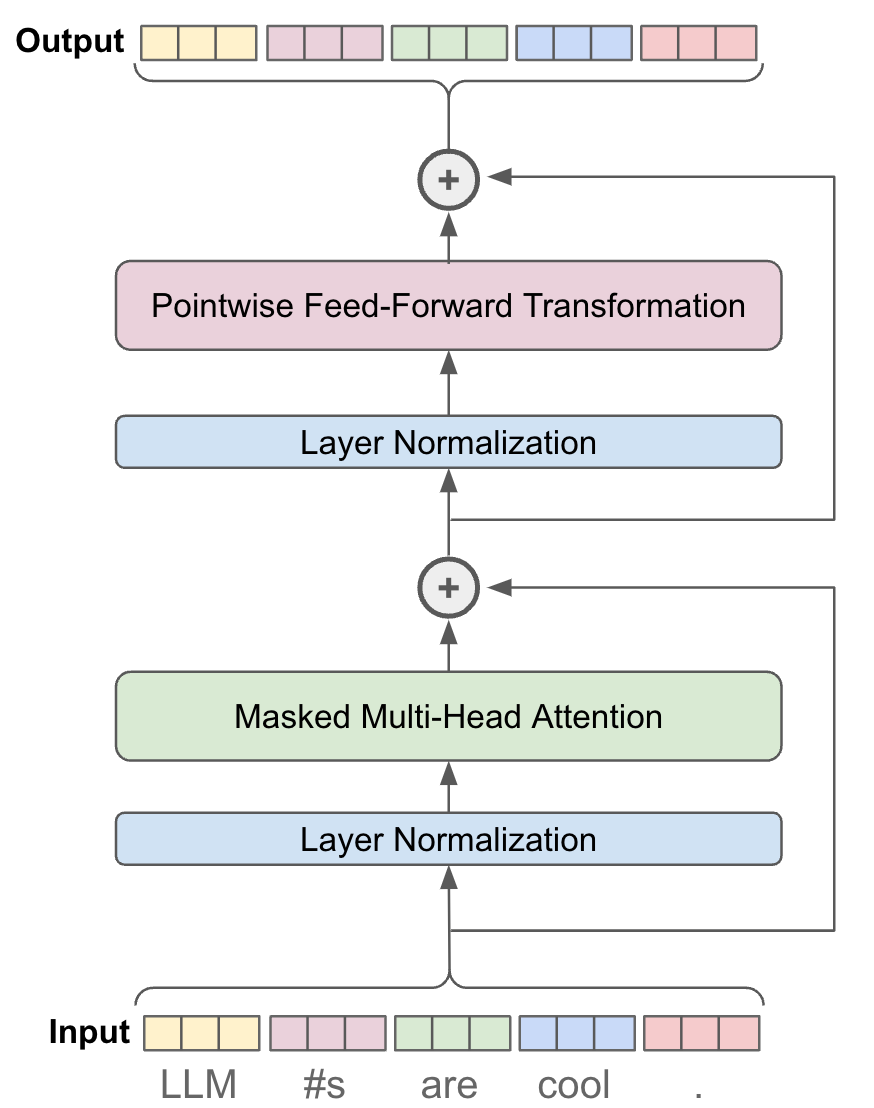


Image 31: Transformer Decoder block

Now, we explore the complete decoder-only transformer architecture, which is built from the core components discussed earlier. While most of the complexity lies in the self-attention mechanism, we also need to understand a few additional steps—such as preparing model inputs and using outputs for text prediction or generation—to fully grasp how the architecture functions end-to-end.

Constructing the Model’s input:As outlined previously, the input to a Transformer Decoder block is expected to be a (batched) sequence of token vectors, typically represented as a tensor with shape [B,T,d][B, T, d][B,T,d], where BBB is the batch size, TTT is the sequence length, and ddd is the embedding dimension. However, in OCR tasks, the Decoder does not receive a textual prompt like in language models, but rather a **ground-truth text label** associated with the input image. How do we convert this label into a sequence of token vectors?

**Tokenization.** The Transformer Decoder receives the raw label (text string) as input during training. The first step is to **tokenize** this label into a sequence of discrete characters (or sub-word units), which are commonly referred to as **tokens**. These tokens are then mapped to corresponding embedding vectors through an embedding layer, forming the sequential input [B,T,d][B, T, d][B,T,d] required by the Decoder.

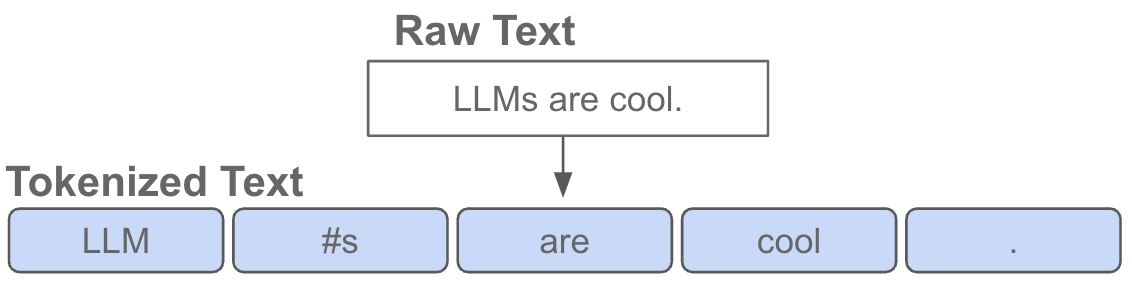


Image 32: Convert

The tokenization process in our OCR system is handled by a custom **character-level tokenizer**, rather than relying on common algorithms like Byte-Pair Encoding (BPE), SentencePiece, or WordPiece. This tokenizer, named CharTokenizer, operates at the granularity of individual characters instead of subwords or words. It builds a small, task-specific vocabulary by collecting all unique characters present in the training dataset. The vocabulary also includes three special tokens: <PAD> (padding), <SOS> (start of sequence), and <EOS> (end of sequence), which play critical roles during training and inference in sequence generation.

The tokenizer implements two essential functions:

**+ Encode**: Converts a raw string (label) into a fixed-length sequence of token IDs, prepending <SOS> and appending <EOS>, with padding if necessary.

**+ Decode**: Converts a sequence of token IDs back into a human-readable string, removing padding and special tokens.

Although often underestimated, tokenization is a crucial component in OCR model training. It directly affects how labels are represented and learned by the Transformer Decoder. A mismatch or error in tokenization may lead to subtle, hard-to-debug performance issues. In the character-level approach, every character is preserved, making it well-suited for OCR tasks where fine-grained control over each symbol is essential, especially in domains with noisy, short, or unusual text patterns.

While this tokenizer is simpler than subword-based ones used in large language models, it is highly effective for the OCR domain where precision at the character level is paramount.

**Token Embeddings.** Once the character-level tokenizer (CharTokenizer) has encoded the text label into a sequence of token IDs, the next step is to convert these discrete tokens into continuous embedding vectors. This is achieved through a **learnable embedding layer**, which is an integral part of the decoder-only Transformer architecture. The embedding layer is implemented as a matrix of shape [V, d], where:

V is the size of the character vocabulary (including <PAD>, <SOS>, and <EOS>),d is the embedding dimension (i.e., the size of each token vector).

Each token ID corresponds to an index in this matrix, and the embedding for a token is retrieved by simply indexing into the embedding layer. In other words, every character token—whether it’s a letter, digit, punctuation mark, or special token—is mapped to a dense vector representation of size d. These embeddings serve as the input to the decoder and are refined during training to help the model learn character-level patterns in OCR outputs.

This simple yet effective mechanism allows the model to operate on symbolic character data in a differentiable space, making it possible for the Transformer Decoder to perform sequence generation accurately and efficiently.

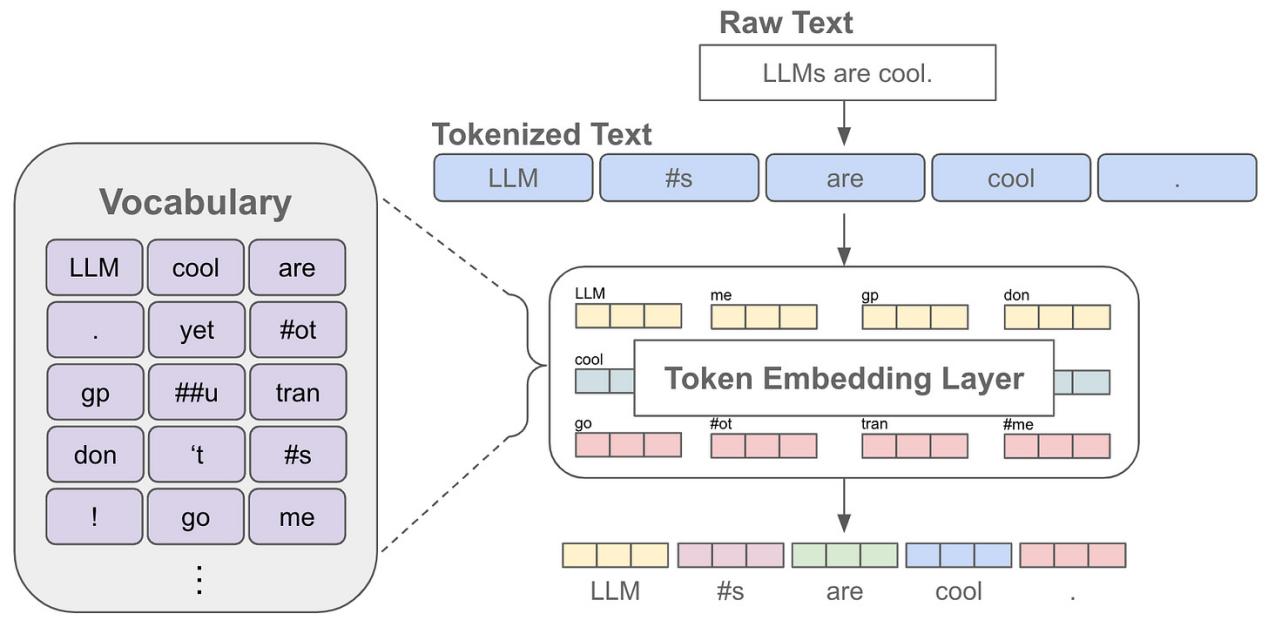


Image 33: Looking up token embedding in an embedding layer

**Positional Embeddings.** After converting the character tokens into embedding vectors, we obtain a tensor of shape [B, T, d], where:

+ B is the batch size,

+ T is the sequence length (after padding if necessary),

+ d is the embedding dimension.

However, since the Transformer architecture lacks recurrence or convolution, it **has no inherent understanding of the order of tokens in the sequence**. To address this, we add **positional embeddings** to the token embeddings to encode information about the position of each token.

Each position in the sequence is assigned a unique embedding vector of dimension d. These positional embeddings can either be:

**+ Learned** as part of the model parameters during training,

+ Or **generated using fixed rules**, such as the sinusoidal functions introduced in the original Transformer paper ([1]).

By adding positional embeddings to the token embeddings, the model is able to distinguish between character sequences like "A1" and "1A", which is crucial for accurate text recognition in OCR tasks. The final input to the decoder is the **element-wise sum of token embeddings and positional embeddings**, which is then processed by the subsequent Transformer layers.

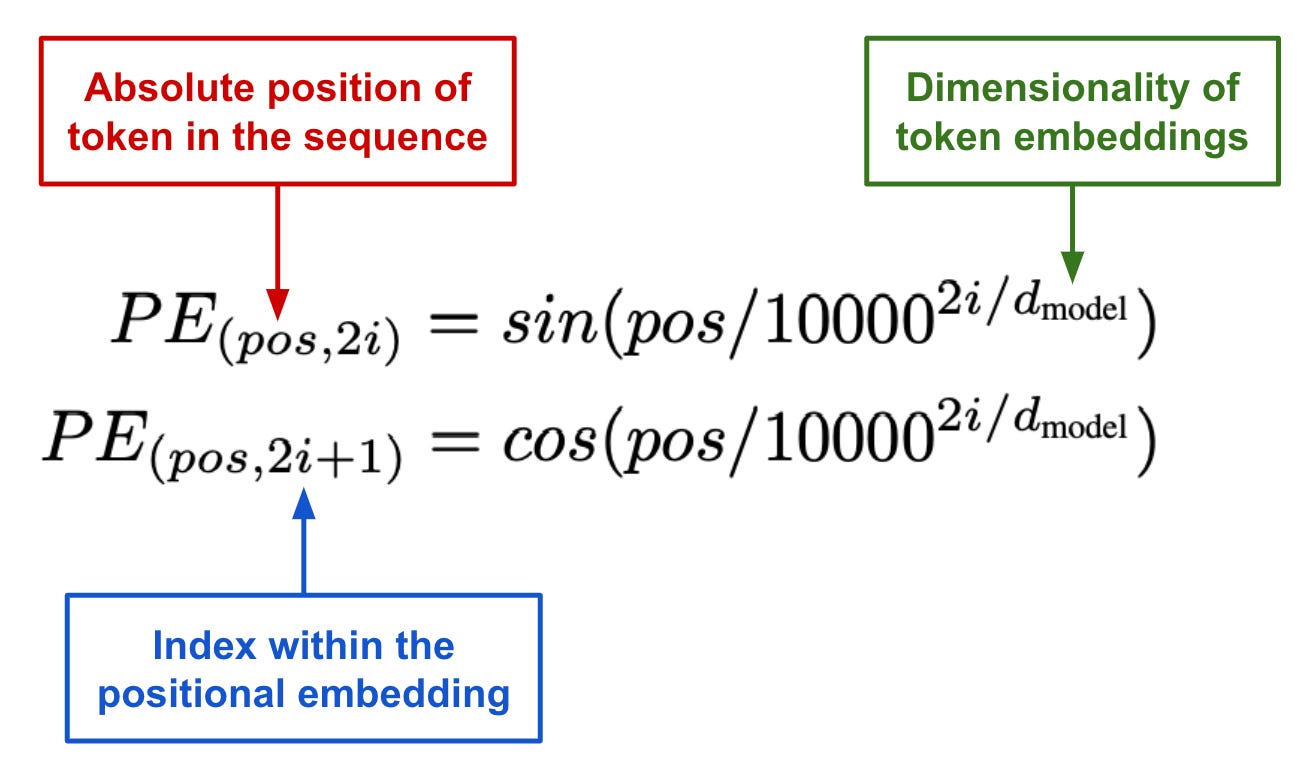


Image 34: Generating positional embeddings with sine and cosine function

These approaches are referred to as “absolute” positional embedding strategies, as the embedding used is determined by the token's absolute position in the sequence. In the context of OCR, this approach works by embedding the position of each character token relative to its position in the sequence. However, as we will discuss later, absolute positional embedding strategies face limitations when generalizing to sequences longer than those seen during training. This becomes particularly relevant in OCR, where the length of input sequences can vary. As a result, more flexible and generalizable positional embedding strategies have been proposed to address this issue.

After all, here is the full Decoder Transformer model:

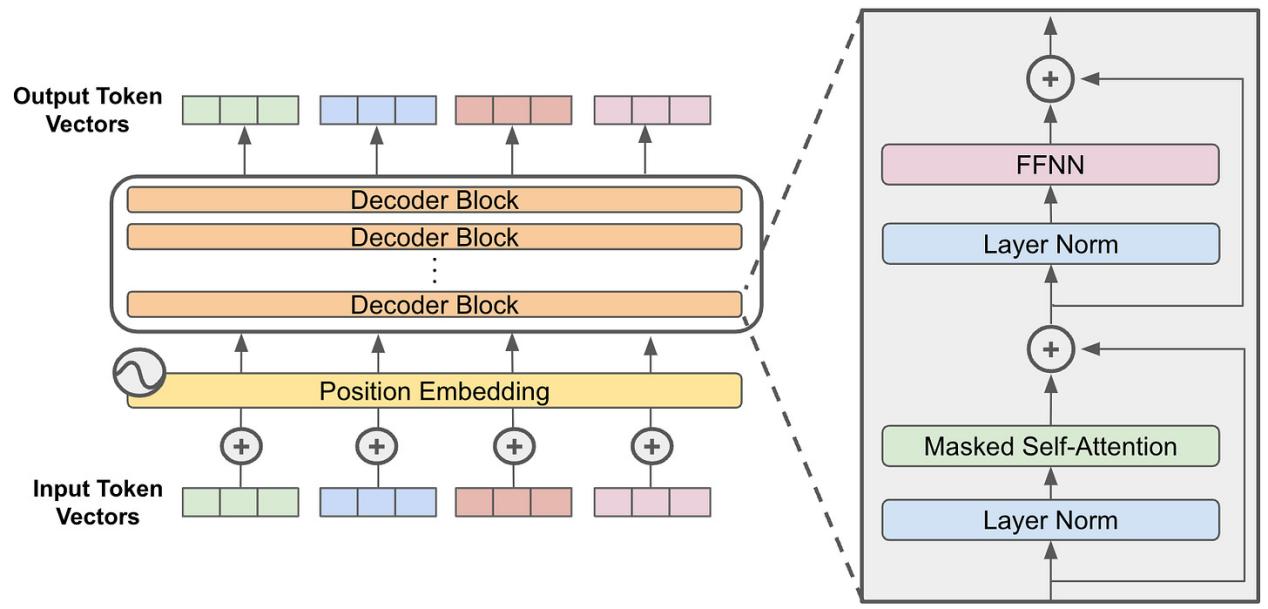


Image 35: Structure of a decoder-only transformer model

**2.5.4.5 Model’s operation**

The model operates through the following steps:

**+ Feature Extraction from Images**: The input image is passed through the **CNN Backbone (ResNet-50)**. This CNN extracts high-level features from the image, which are typically representations of shapes, textures, and objects that are essential for identifying text.

**+ Feature Projection**: The extracted features, typically high-dimensional, are then projected into a lower-dimensional space through a linear transformation. This is done via a **fully connected layer** to match the embedding vector size used by the Transformer, typically 512 dimensions.

**+ Embedding and Attention**: The target text sequences (the text the model is trying to generate) are converted into embeddings using an **embedding layer**. These embeddings are further enhanced with **Positional Encoding**, which adds information about the position of each token in the sequence. The attention mechanism then allows the model to "attend" to important parts of the sequence, giving higher attention to relevant tokens based on the input image.

**+ Decoding with Transformer**: The processed features and the target text sequences are fed into the **Transformer Decoder**. The decoder uses the attention mechanism to decode the image features into a sequence of text, where each token (or word) is generated step by step.

**+ Output**: The final output is a predicted sequence of text, generated from the image. Each word in the output is predicted based on the features extracted from the image, allowing the model to generate highly accurate textual information from images containing text.

**2.5.4.6 Model Evaluation**

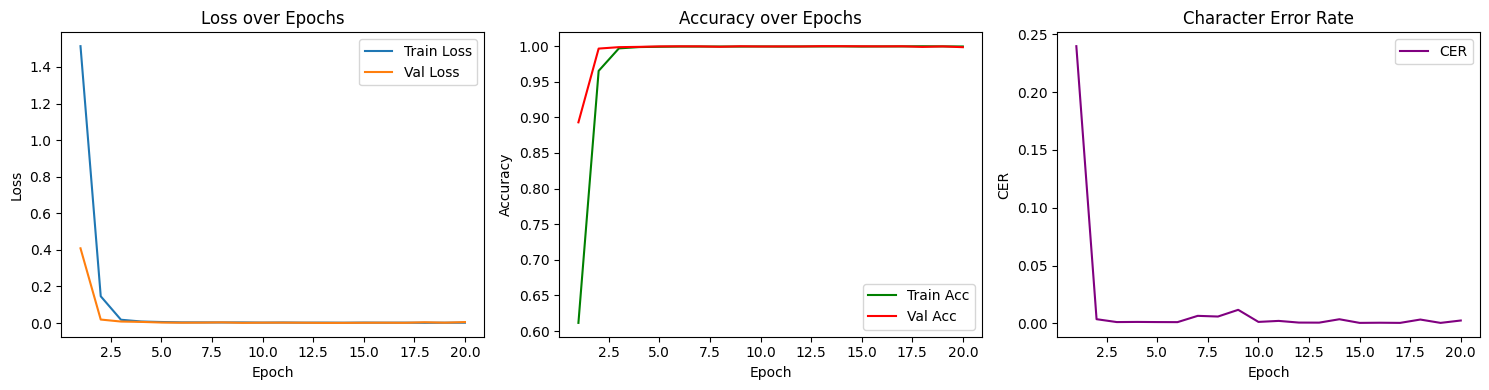


Image 36: Evaluation

**Loss over Epochs**:  
 The first chart illustrates the training loss (Train Loss) and validation loss (Val Loss) over the epochs. A decreasing training loss indicates that the model is effectively learning from the training data. If the validation loss stabilizes or begins to increase while the training loss continues to decrease, this may suggest overfitting. Overfitting occurs when the model learns the training data too well, including noise and outliers, which negatively impacts its ability to generalize to unseen data.

**Accuracy over Epochs**:  
 The second chart depicts both training accuracy (Train Acc) and validation accuracy (Val Acc). A high and stable accuracy in both datasets indicates that the model can recognize characters correctly. Consistency between training and validation accuracy is crucial; it suggests that the model is not just memorizing the training data but is also capable of generalizing its learning to new, unseen examples. This balance is vital for ensuring robust performance in real-world applications of OCR.

**Character Error Rate (CER)**:  
 The final chart shows the Character Error Rate (CER), which should ideally decrease over time. A lower CER signifies that the model is improving its ability to correctly identify characters within the text. In OCR applications, minimizing CER is critical, as it directly correlates with the accuracy of the text recognition process. A consistently low CER indicates a well-performing OCR model.

**2.5.4.7 Preprocessing data**

Before go to the training section, we need to preprocessing data for these images. In this OCR task, preprocessing data plays a crucial role in ensurung that the model receives only images containing a single line of text along with the respectively label. The raw data consists of original images containing multiple lines of text in a image and a JSON file , where each image is associated with a list of text lines, each containing the recognized text and its bounding box coordinates.

To process the data, the program iterates through each image, and for every text line, it converts the bounding box into a rectangular region by computing the minimum and maximum x and y coordinates. The line image is then cropped from the original image using the Python Imaging Library (PIL). Each cropped line image is saved as a separate file, named using the format {image\_name}\_crop\_{index}.png. Simultaneously, the corresponding text content (label) is recorded and stored in a new label file that lists each cropped image along with its associated text.

The result of this preprocessing step is a clean, standardized dataset in which each image corresponds to a single line of text and includes a ground-truth label. This format is convenient for training a line-level OCR model using a CNN backbone combined with a Transformer Decoder. By standardizing the input data in this way, we reduce noise, improve accuracy, and enhance the effectiveness of the training process.

**2.5.4.8 Sample output of the model**



Image 37: Output1



Image 38: Output2

**2.5.4.9 End-to-End Deployment of a CNN + Transformer OCR System**

After completing the training phase, the OCR line model was successfully deployed on a web application to enable real-time usage. The deployment allows users to upload an image containing text, after which the system automatically detects and crops each text line based on pre-trained bounding box detection (e.g., via PaddleOCR or similar). Each cropped line is then passed through the trained OCR model, which outputs the corresponding text.

The web interface was designed to be user-friendly, providing functionalities such as uploading images, displaying original and cropped image lines, and showing the recognized text results. In addition, visualization of attention maps can be integrated to help users understand how the model focuses on different parts of the input during text generation. This deployment demonstrates the practical application of the model and enables broader accessibility for end users without requiring technical knowledge of the underlying model.

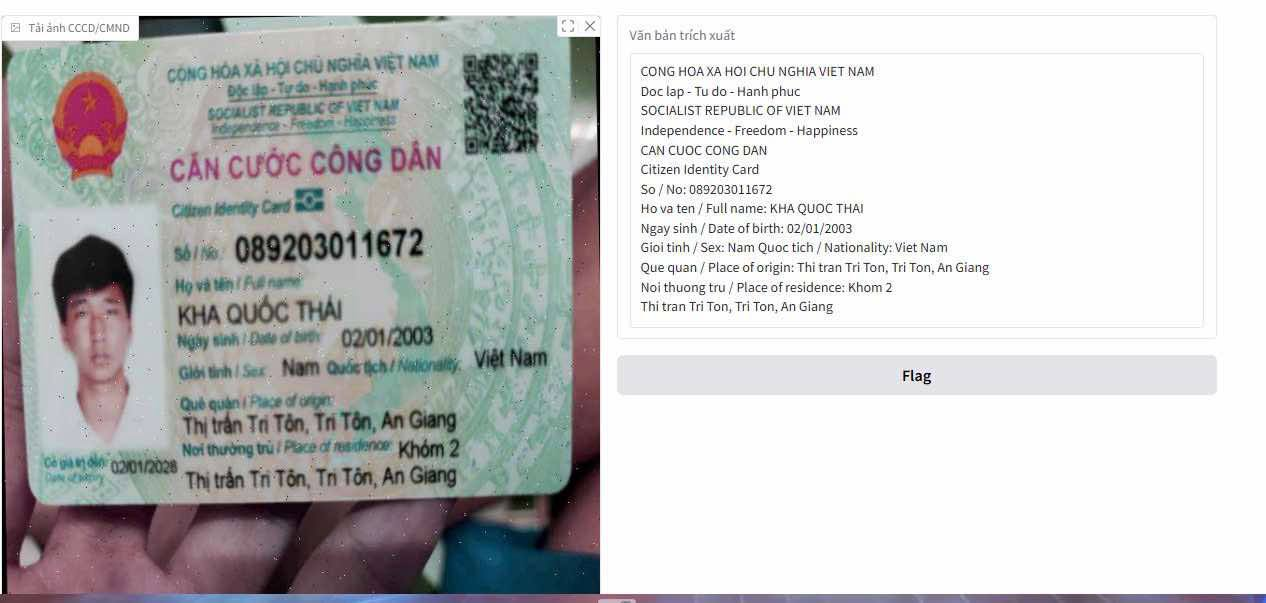


Image 39: Deploy the model

# 

# REFERENCES

 Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I., "Attention is All You Need," *in Advances in Neural Information Processing Systems (NeurIPS)*, 2017, pp. 5998–6008.

 Dao, T., Fu, D., Ermon, S., Rudra, A., and Ré, C., "FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness," *in Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

 Choromanski, K., Likhosherstov, V., Dohan, D., Song, X., Gane, A., Sarlós, T., Hawkins, P., Davis, J., Mohiuddin, A., Kaiser, Ł., and Belanger, D., "Rethinking Attention with Performers," *in International Conference on Learning Representations (ICLR)*, 2021.

 Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J., "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer," *Journal of Machine Learning Research (JMLR)*, vol. 21, no. 140, pp. 1–67, 2020.

 Li, J., Zhu, Q., and Wong, K.-F., "Grouped-Query Attention for Improving Vision Transformers," *arXiv preprint arXiv:2202.10427*, 2022.

 Nguyễn, H. T., Trần, M. T., Lê, N. P., Phạm, D. M., and Nguyễn, N. C., "Vintern-1B: An Efficient Multimodal Large Language Model for Vietnamese," *arXiv preprint arXiv:2408.12480*, 2024.

 Viblo Blog, "Transformers - Người Máy Biến Hình Biến Đổi Thế Giới NLP," *Viblo.asia*. [Online]. Available: <https://viblo.asia/p/transformers-nguoi-may-bien-hinh-bien-doi-the-gioi-nlp-924lJPOXKPM>