Transformer From Scratch With PyTorch🔥

2024 | © LUIS FERNANDO TORRES

Table of Contents

Introduction

In 2017, the Google Research team published a paper called *"Attention Is All You Need"*, which presented the Transformer architecture and was a paradigm shift in Machine Learning, especially in Deep Learning and the field of natural language processing.

The Transformer, with its parallel processing capabilities, allowed for more efficient and scalable models, making it easier to train them on large datasets. It also demonstrated superior performance in several NLP tasks, such as sentiment analysis and text generation tasks.

The archicture presented in this paper served as the foundation for subsequent models like GPT and BERT. Besides NLP, the Transformer architecture is used in other fields, like audio processing and computer vision. You can see the usage of Transformers in audio classification in the notebook **Audio Data: Music Genre Classification.**

Even though you can easily employ different types of Transformers with the 🤗 **Transformers** library, it is crucial to understand how things truly work by building them from scratch.

In this notebook, we will explore the Transformer architecture and all its components. I will use PyTorch to build all the necessary structures and blocks, and I will use the **Coding a Transformer from scratch on PyTorch, with full explanation, training and inference** video posted by **Umar Jamil** on YouTube as reference.

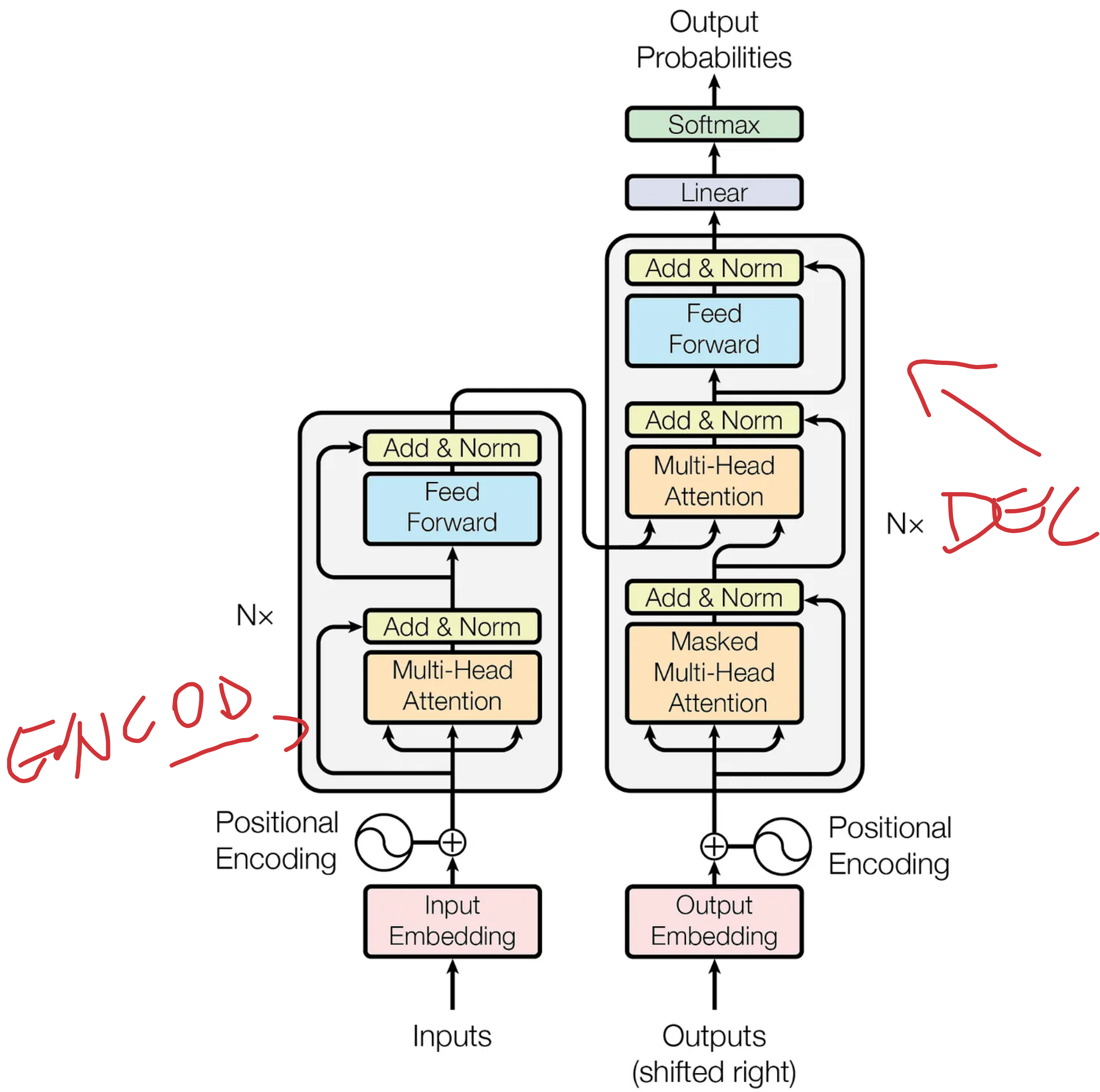
Let's start by importing all the necessary libraries.

In [1]:
```python
# Importing libraries

# PyTorch
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader, random_split
from torch.utils.tensorboard import SummaryWriter

# Math
import math

# HuggingFace libraries
from datasets import load_dataset
from tokenizers import Tokenizer
from tokenizers.models import WordLevel
from tokenizers.trainers import WordLevelTrainer
from tokenizers.pre_tokenizers import Whitespace

# Pathlib
from pathlib import Path

# typing
from typing import Any

# Library for progress bars in loops
from tqdm import tqdm

# Importing library of warnings
import warnings
```

```
/opt/conda/lib/python3.10/site-packages/scipy/__init__.py:146: UserWarning: A NumP
y version >=1.16.5 and <1.23.0 is required for this version of SciPy (detected ver
sion 1.24.3
  warnings.warn(f"A NumPy version >={np_minversion} and <{np_maxversion}"
```

Transformer Architecture

Before coding, let's take a look at the Transformer architecture.



Source: Attention Is All You Need

The Transformer architecture has two main blocks: the **encoder** and the **decoder**. Let's take a look at them further.

Encoder: It has a *Multi-Head Attention* mechanism and a fully connected *Feed-Forward* network. There are also residual connections around the two sub-layers, plus layer normalization for the output of each sub-layer. All sub-layers in the model and the embedding layers produce outputs of dimension $d_{model} = 512$.

Decoder: The decoder follows a similar structure, but it inserts a third sub-layer that performs multi-head attention over the output of the encoder block. There is also a modification of the self-attention sub-layer in the decoder block to avoid positions from attending to subsequent positions. This masking ensures that the predictions for position i depend solely on the known outputs at positions less than i.

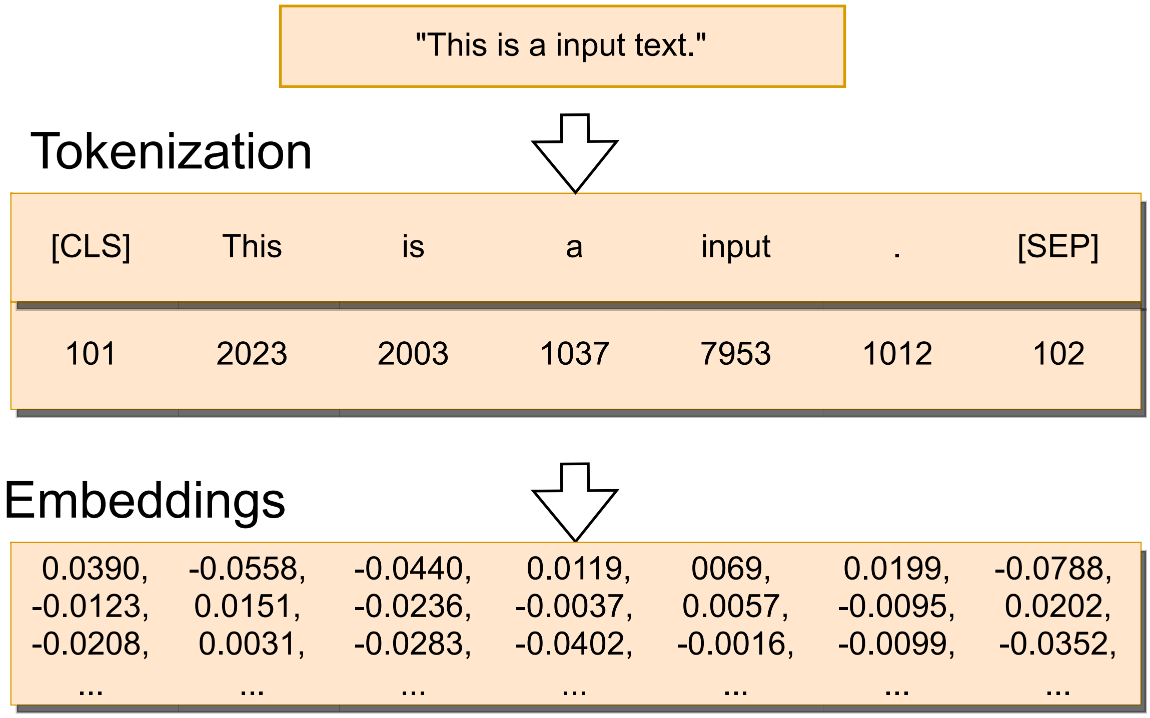
Both the encoder and decode blocks are repeated N times. In the original paper, they defined $N = 6$, and we will define a similar value in this notebook.

Input Embeddings

When we observe the Transformer architecture image above, we can see that the Embeddings represent the first step of both blocks.

The `InputEmbedding` class below is responsible for converting the input text into numerical vectors of `d_model` dimensions. To prevent that our input embeddings become extremely small, we normalize them by multiplying them by the $\sqrt{d_{model}}$.

In the image below, we can see how the embeddings are created. First, we have a sentence that gets split into tokens—we will explore what tokens are later on—. Then, the token IDs—identification numbers—are transformed into the embeddings, which are high-dimensional vectors.



Source: vaclavkosar.com

In [2]:
```
# Creating Input Embeddings
class InputEmbeddings(nn.Module):

    def __init__(self, d_model: int, vocab_size: int):
        super().__init__()
        self.d_model = d_model # Dimension of vectors (512)
        self.vocab_size = vocab_size # Size of the vocabulary
        self.embedding = nn.Embedding(vocab_size, d_model) # PyTorch layer that cor

    def forward(self, x):
        return self.embedding(x) * math.sqrt(self.d_model) # Normalizing the variar
```

Positional Encoding

In the original paper, the authors add the positional encodings to the input embeddings at the bottom of both the encoder and decoder blocks so the model can have some information about the relative or absolute position of the tokens in the sequence. The positional encodings have the same dimension d_{model} as the embeddings, so that the two vectors can be summed and we can combine the semantic content from the word embeddings and positional information from the positional encodings.

In the `PositionalEncoding` class below, we will create a matrix of positional encodings `pe` with dimensions `(seq_len, d_model)`. We will start by filling it with 0s.We will then apply the sine function to even indices of the positional encoding matrix while the cosine function is applied to the odd ones.

$$\text{Even Indices } (2i): \quad \text{PE}(\text{pos}, 2i) = \sin\left(\frac{\text{pos}}{10000^{2i/}}\right.$$

$$\text{Odd Indices } (2i+1): \quad \text{PE}(\text{pos}, 2i+1) = \cos\left(\right.$$

We apply the sine and cosine functions because it allows the model to determine the position of a word based on the position of other words in the sequence, since for any fixed offset k, PE_{pos+k} can be represented as a linear function of PE_{pos}. This happens due to the properties of sine and cosine functions, where a shift in the input results in a predictable change in the output.

In [3]:

```
# Creating the Positional Encoding
class PositionalEncoding(nn.Module):

    def __init__(self, d_model: int, seq_len: int, dropout: float) -> None:
        super().__init__()
        self.d_model = d_model # Dimensionality of the model
        self.seq_len = seq_len # Maximum sequence length
        self.dropout = nn.Dropout(dropout) # Dropout layer to prevent overfitting

        # Creating a positional encoding matrix of shape (seq_len, d_model) filled
        pe = torch.zeros(seq_len, d_model)

        # Creating a tensor representing positions (0 to seq_len - 1)
        position = torch.arange(0, seq_len, dtype = torch.float).unsqueeze(1) # Tra

        # Creating the division term for the positional encoding formula
        div_term = torch.exp(torch.arange(0, d_model, 2).float() * (-math.log(10000

        # Apply sine to even indices in pe
        pe[:, 0::2] = torch.sin(position * div_term)
        # Apply cosine to odd indices in pe
        pe[:, 1::2] = torch.cos(position * div_term)

        # Adding an extra dimension at the beginning of pe matrix for batch handlin
        pe = pe.unsqueeze(0)

        # Registering 'pe' as buffer. Buffer is a tensor not considered as a model
        self.register_buffer('pe', pe)

    def forward(self,x):
        # Addind positional encoding to the input tensor X
```

```
        x = x + (self.pe[:, :x.shape[1], :]).requires_grad_(False)
        return self.dropout(x) # Dropout for regularization
```

Layer Normalization

When we look at the encoder and decoder blocks, we see several normalization layers called ***Add & Norm***.

The `LayerNormalization` class below performs layer normalization on the input data. During its forward pass, we compute the mean and standard deviation of the input data. We then normalize the input data by subtracting the mean and dividing by the standard deviation plus a small number called epsilon to avoid any divisions by zero. This process results in a normalized output with a mean 0 and a standard deviation 1.

We will then scale the normalized output by a learnable parameter `alpha` and add a learnable parameter called `bias`. The training process is responsible for adjusting these parameters. The final result is a layer-normalized tensor, which ensures that the scale of the inputs to layers in the network is consistent.

In [4]:
```
# Creating Layer Normalization
class LayerNormalization(nn.Module):

    def __init__(self, eps: float = 10**-6) -> None: # We define epsilon as 0.00000
        super().__init__()
        self.eps = eps

        # We define alpha as a trainable parameter and initialize it with ones
        self.alpha = nn.Parameter(torch.ones(1)) # One-dimensional tensor that will

        # We define bias as a trainable parameter and initialize it with zeros
        self.bias = nn.Parameter(torch.zeros(1)) # One-dimensional tenso that will

    def forward(self, x):
        mean = x.mean(dim = -1, keepdim = True) # Computing the mean of the input d
        std = x.std(dim = -1, keepdim = True) # Computing the standard deviation of

        # Returning the normalized input
        return self.alpha * (x-mean) / (std + self.eps) + self.bias
```

Feed-Forward Network

In the fully connected feed-forward network, we apply two linear transformations with a ReLU activation in between. We can mathematically represent this operation as:

$$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2 \qquad (3)$$

W_1 and W_2 are the weights, while b_1 and b_2 are the biases of the two linear transformations.

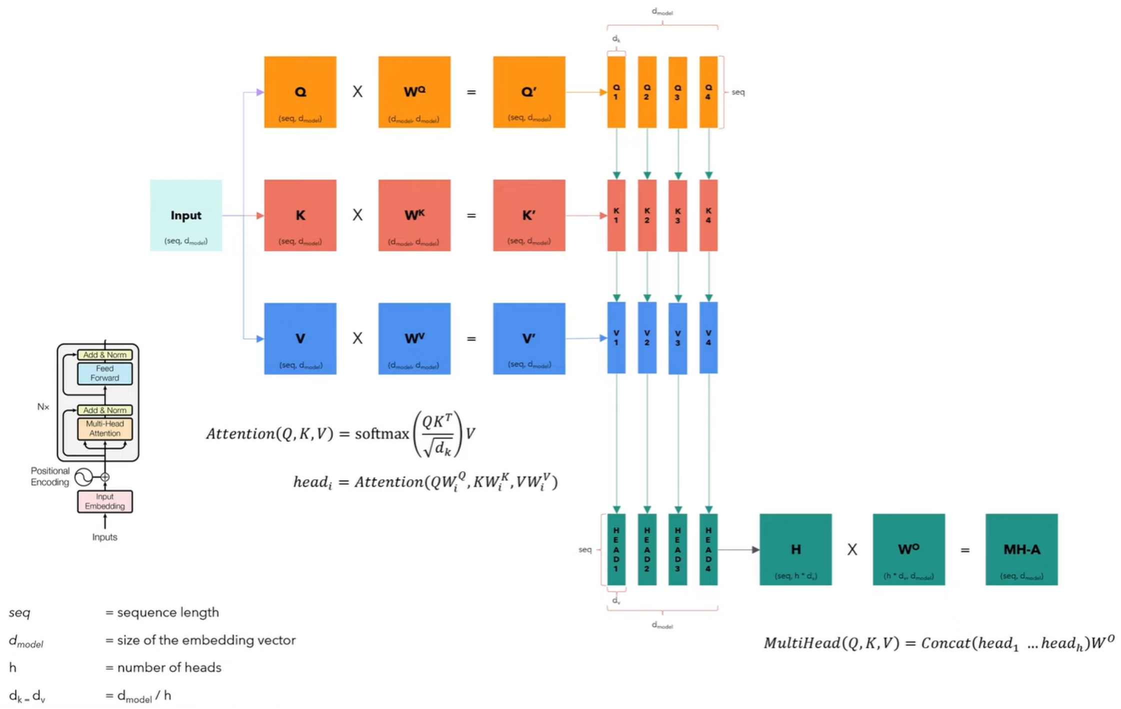
In the `FeedForwardBlock` below, we will define the two linear transformations— `self.linear_1` and `self.linear_2` —and the inner-layer `d_ff`. The input data will first pass through the `self.linear_1` transformation, which increases its dimensionality from `d_model` to `d_ff`. The output of this operation passes through the ReLU activation function, which introduces non-linearity so the network can learn more complex patterns, and the `self.dropout` layer is applied to mitigate overfitting. The final operation is the `self.linear_2` transformation to the dropout-modified tensor, which transforms it back to the original `d_model` dimension.

In [5]:
```
# Creating Feed Forward Layers
class FeedForwardBlock(nn.Module):

    def __init__(self, d_model: int, d_ff: int, dropout: float) -> None:
        super().__init__()
        # First linear transformation
        self.linear_1 = nn.Linear(d_model, d_ff) # W1 & b1
        self.dropout = nn.Dropout(dropout) # Dropout to prevent overfitting
        # Second linear transformation
        self.linear_2 = nn.Linear(d_ff, d_model) # W2 & b2

    def forward(self, x):
        # (Batch, seq_len, d_model) --> (batch, seq_len, d_ff) -->(batch, seq_len,
        return self.linear_2(self.dropout(torch.relu(self.linear_1(x))))
```

Multi-Head Attention

The Multi-Head Attention is the most crucial component of the Transformer. It is responsible for helping the model to understand complex relationships and patterns in the data.

The image below displays how the Multi-Head Attention works. It doesn't include `batch` dimension because it only illustrates the process for one single sentence.



Source: YouTube: Coding a Transformer from scratch on PyTorch, with full explanation, training and inference by Umar Jamil.

The Multi-Head Attention block receives the input data split into queries, keys, and values organized into matrices Q, K, and V. Each matrix contains different facets of the input, and they have the same dimensions as the input.

We then linearly transform each matrix by their respective weight matrices W^Q, W^K, and W^V. These transformations will result in new matrices Q', K', and V', which will be split into smaller matrices corresponding to different heads h, allowing the model to attend to information from different representation subspaces in parallel. This split creates multiple sets of queries, keys, and values for each head.

Finally, we concatenate every head into an H matrix, which is then transformed by another weight matrix W^o to produce the multi-head attention output, a matrix $MH - A$ that retains the input dimensionality.

In [6]:

```
# Creating the Multi-Head Attention block
class MultiHeadAttentionBlock(nn.Module):

    def __init__(self, d_model: int, h: int, dropout: float) -> None: # h = number
        super().__init__()
        self.d_model = d_model
        self.h = h

        # We ensure that the dimensions of the model is divisible by the number of
        assert d_model % h == 0, 'd_model is not divisible by h'
```