National University of Computer & Emerging Sciences, Karachi
Artificial Intelligence-School of Computing
Fall 2024, Lab Manual – 11

Course Code (AI4002)	**Course: Computer Vision Lab**
Instructor(s):	**Sohail Ahmed**

Objectives:

- Introduction to Transformers
- Self-Attention Mechanism
- Vision in Transformers

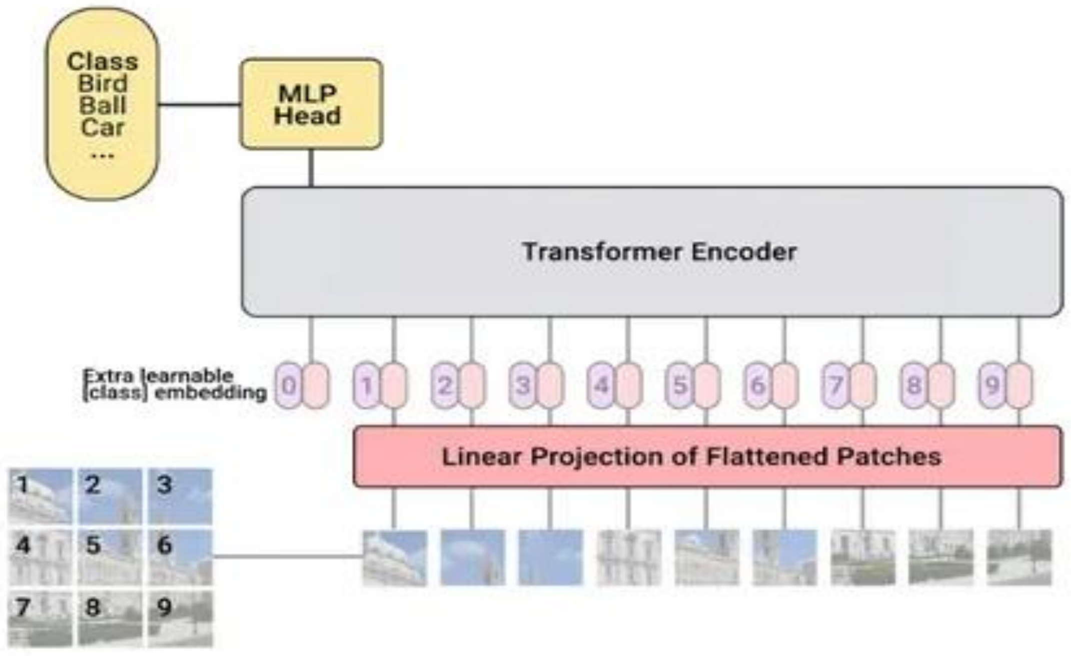
Introduction to Transformers

Over the years, we have been using Computer vision (CV) and image processing techniques from artificial intelligence (AI) and pattern recognition to derive information from images, videos, and other visual inputs. Underlying methods successfully achieve this by manipulating digital images through computer algorithms.

Researchers found that regular models had limitations in some applications, which prompted advancements in traditional deep learning and deep neural networks. This brought about the popularity of transformer models. They have the ability known as “self-attention”. This provides them with an edge over other model architectures, and researchers have introduced it extensively in natural language processing and computer vision.

What are Vision Transformers?

In simple terms, vision transformers are types of transformers used for visual tasks such as in image processing. This entails that transformers are being used in many areas, including NLP, but ViT specifically focuses on processing image-related tasks. Recently, used majorly in Generative artificial intelligence and stable diffusion.



How Do Vision Transformers Work?

Vision Transformer applies the transformer to image classification tasks with a model architecture similar to a regular transformer. It adjusts itself to allow efficient handling of images, as other models will perform for natural language processing tasks.

Key concepts of vision transformers include ‘attention’ and ‘multi-head attention’. Having an understanding of these concepts is very essential in how vision transformers work. Attention is a key mechanism unique to transformers and is the secrete to their strength. Let’s look at the transformer architecture and see how it works.

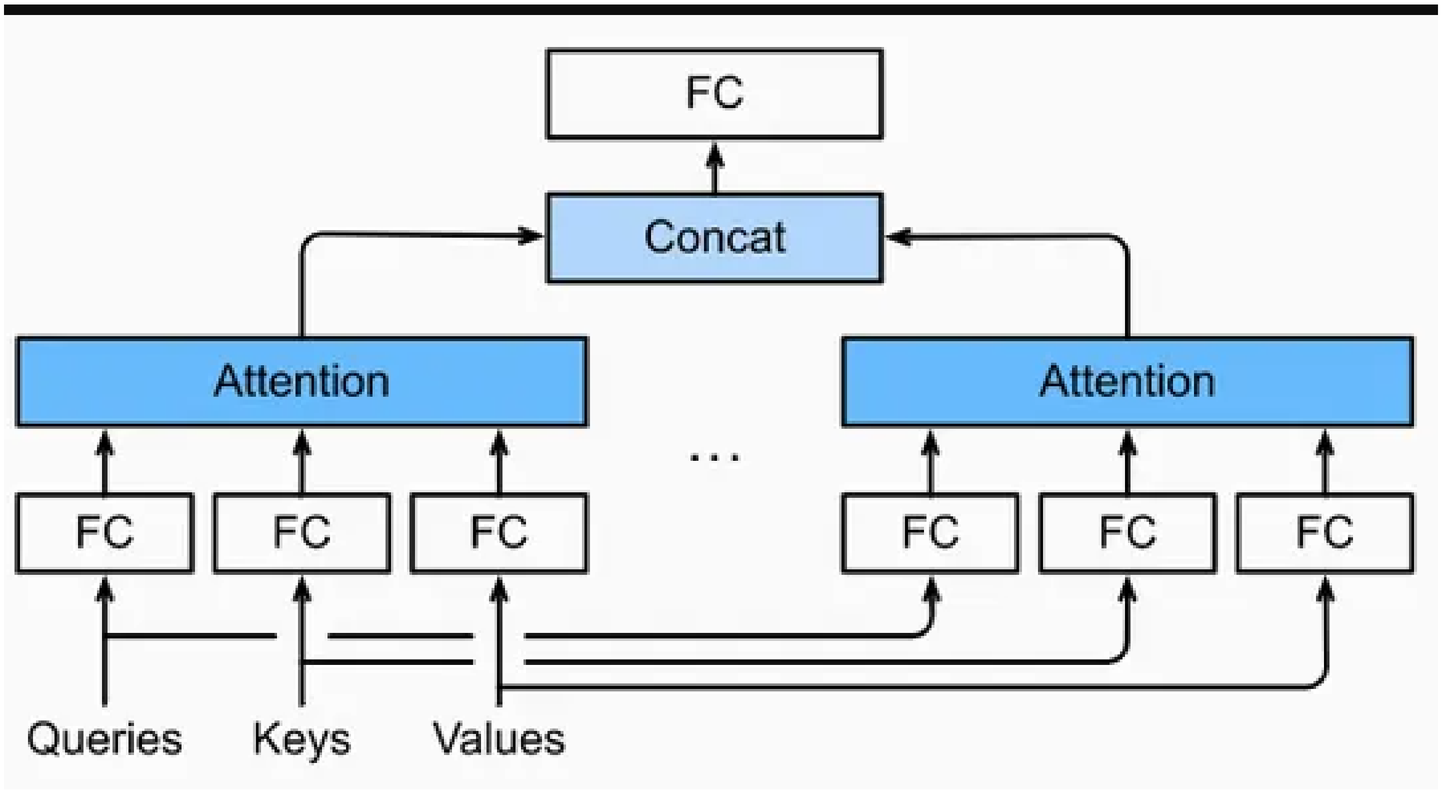
The Masked Multi-Head Attention is a central mechanism of the Transformer similar to skip-joining as in ResNet50 architecture. This means that there is a shortcut connection or skipping of some layers of the network.

$$\text{MultiheadedAttention}(Q, K, V) = \text{Concat}(\text{Attention}(XW_i^Q, XW_i^K, XW_i^V))W^O$$

Source: Wikipedia

Let’s us look at these variables briefly. Where the value of X is a concatenation of the matrix of word embeddings and the matrices:

1. **Q**: This stands for Query.
2. **K**: This stands for Key, and
3. **V**: Stands for Value



The multi-head attention calculates the attention weight of a Query token which could be the prompt of an image. Both the Key token and the Value associated with each Key are multiplied together. We can also say it calculates the relationship or attention weight between the Query and the Key and then multiplies the Value associated with each Key.

We can conclude that multi-head attention allows us to treat different parts of the input sequence differently. The model bests capture positional details since each head will separately attend to different input elements. This gives us a more robust representation.

Python Implementation of Multihead Attention

We have seen that multi-head attention transforms the consecutive weight matrices into the corresponding feature vectors representing the Queries, Keys, and Values. Lets us see an implementation module below.

```
class MultiheadAttention(nn.Module):

    def __init__(self, input_dim, embed_dim, num_heads):
        super().__init__()
        assert embed_dim % num_heads == 0, "Embedding dimension must be 0 modulo number of heads."

        self.embed_dim = embed_dim
        self.num_heads = num_heads
        self.head_dim = embed_dim // num_heads

        # Stack all weight matrices 1...h together for efficiency
        # Note that in many implementations you see "bias=False" which is optional
        self.qkv_proj = nn.Linear(input_dim, 3*embed_dim)
        self.o_proj = nn.Linear(embed_dim, embed_dim)

        self._reset_parameters()

    def _reset_parameters(self):
        # Original Transformer initialization, see PyTorch documentation
        nn.init.xavier_uniform_(self.qkv_proj.weight)
        self.qkv_proj.bias.data.fill_(0)
        nn.init.xavier_uniform_(self.o_proj.weight)
        self.o_proj.bias.data.fill_(0)

    def forward(self, x, mask=None, return_attention=False):
        batch_size, seq_length, _ = x.size()
        qkv = self.qkv_proj(x)

        # Separate Q, K, V from linear output
        qkv = qkv.reshape(batch_size, seq_length, self.num_heads, 3*self.head_dim)
        qkv = qkv.permute(0, 2, 1, 3) # [Batch, Head, SeqLen, Dims]
        q, k, v = qkv.chunk(3, dim=-1)
```

```
# Determine value outputs
values, attention = scaled_dot_product(q, k, v, mask=mask)
values = values.permute(0, 2, 1, 3) # [Batch, SeqLen, Head, Dims]
values = values.reshape(batch_size, seq_length, self.embed_dim)
o = self.o_proj(values)

if return_attention:
    return o, attention
else:
    return o
```