RDFIA - M2 IMA


# Practical work report 1-e
# Transformers

Done by :

Carlos GRUSS & Oussama RCHAKI

25/09/24

SORBONNE UNIVERSITÉ

# Contents

SORBONNE UNIVERSITÉ

# Objective

The goal of this practical work is to implement a simplified version of the Vision Transformer (ViT) model as introduced in the paper "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale" by Alexey Dosovitskiy et al., 2021. We will create a basic version, much smaller than the models presented in the paper, and without the use of complex data augmentation or regularization techniques. It's worth noting that such techniques are essential for transformers to perform effectively on large-scale datasets.

After manually implementing the ViT, we will explore the `timm` library, which offers pre-built implementations of various popular vision architectures.

# 1 – Self-attention

## What is the main feature of self-attention, especially compared to its convolutional counterpart. What is its main challenge in terms of computation/memory?

The main feature of self-attention is its ability to capture global dependencies within the input data by considering the relationships between all pairs of elements in the input sequence simultaneously. In the context of images, this means that each patch can attend to every other patch in the image, allowing the model to integrate information from distant parts of the image in a single layer.

In contrast, convolutional neural networks (CNNs) are inherently local. Each convolutional layer processes data within a limited receptive field defined by the kernel size, focusing on local neighborhoods. To capture global context in CNNs, multiple layers must be stacked so that the receptive field progressively grows, eventually encompassing the entire input. This hierarchical approach requires deeper networks to model long-range dependencies effectively.

The main challenge with self-attention is its computational and memory complexity. Specifically, self-attention mechanisms involve calculating attention scores between all pairs of elements in the input sequence, resulting in a computational complexity of $O(n^2)$ for sequences of length $n$. This quadratic scaling means that as the sequence length increases, the resource requirements grow rapidly.

## Equations

Let's say we have an input consisting of a certain sequence of vectors. We can organize these into a matrix $X$:

$$X = \begin{pmatrix} \text{———} & \vec{E_1} & \text{———} \\ \text{———} & \vec{E_2} & \text{———} \\ & \vdots & \\ \text{———} & \vec{E_N} & \text{———} \end{pmatrix}$$

where $N$ is the length of the input sequence and each $\vec{E_i} \in \mathbb{R}^{d_{\text{model}}}$ is the embedding for the $i$th input. The embeddings are just a vector representation of whatever input we originally had (for example, in an NLP application, a token embedding obtained with word2vec, or in a Vision Transformer the output of a convolutional layer). Note that $d_{\text{model}}$ is generally large, for example in GPT-3, $d_{\text{model}} = 12,288$.

We then want to linearly project each of these embeddings into a smaller space of dimension $d_k$ (for GPT-3, $d_k = 128$). We can do this by performing a set of matrix multiplications:

$$
\begin{aligned}
Q &= XW_Q \\
K &= XW_K \\
V &= XW_V
\end{aligned}
$$

where each of these projection matrices $W_*$ are of dimensions $d_{\text{model}} \times d_k$, and their values are learned throughout the training process. $Q, K, V$ stand for Query, Key and Value respectively. We thus obtain matrices:

$$
Q = \begin{pmatrix} \text{———} & \vec{Q_1} & \text{———} \\ \text{———} & \vec{Q_2} & \text{———} \\ & \vdots & \\ \text{———} & \vec{Q_N} & \text{———} \end{pmatrix}
$$

$$
K = \begin{pmatrix} \text{———} & \vec{K_1} & \text{———} \\ \text{———} & \vec{K_2} & \text{———} \\ & \vdots & \\ \text{———} & \vec{K_N} & \text{———} \end{pmatrix}
$$

where for instance we have $\vec{Q_i} = \vec{E_i} W_Q$.

Very simply put, we then want to, given a Query vector, recover the single or multiple Key vectors that most resemble the Query vector. To do this, we perform a dot product between each pair of vectors, which gives a measure of similarity between them. A larger dot product indicates that the vectors are more closely aligned (i.e., pointing in a similar direction) in the $d_k$-dimensional space.

$$
QK^T = \begin{pmatrix} \vec{Q_1} \cdot \vec{K_1} & \vec{Q_1} \cdot \vec{K_2} & \cdots & \vec{Q_1} \cdot \vec{K_N} \\ \vec{Q_2} \cdot \vec{K_1} & \vec{Q_2} \cdot \vec{K_2} & \cdots & \vec{Q_2} \cdot \vec{K_N} \\ \vdots & \vdots & \ddots & \vdots \\ \vec{Q_N} \cdot \vec{K_1} & \vec{Q_N} \cdot \vec{K_2} & \cdots & \vec{Q_N} \cdot \vec{K_N} \end{pmatrix}
$$

We can observe that in the $i$th row we're trying to "match" $\vec{Q_i}$ to each of the Key vectors we have, by performing the dot products between $\vec{Q_i}$ and each of the $\vec{K_j}$s. We'll have a large values where $\vec{Q_i}$ and $\vec{K_j}$ are closely aligned and a small/negative value where they're not.

For numerical/stability reasons, we divide the whole matrix by the scalar $\sqrt{d_k}$, and then perform a row-wise soft-max:

$$
A = \text{Softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)
$$

This matrix $A$ is a square $N \times N$ matrix where the $i$th row is a probability distribution of expressing the importance of each of the $N$ keys with respect to the $i$th query. In other words, $A_{ij}$ represents how much **attention** the $i$th position pays to the $j$th position.

The final step is integrating $V$, which we haven't used yet. Conceptually, we can imagine the value vectors hold the actual information we want to pay attention to when computing the output. So in $V$ we have a sequence of encoded input information, while in $A$ we have computed the importance of each input $j$ when predicting the output $i$. To do this, we perform the product:

$$Z = AV = \text{Softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

To break down this matrix product picture that each row is performing the following:

$$\vec{Z}_i = \sum_{j=1}^{N} A_{ij}\vec{V}_j$$

So each output $\vec{Z}_i$ is incorporating information from each input $\vec{V}_j$ weighted by how much the output $i$ should pay attention to the input $j$.

Observe that, starting from an input $X \in \mathbb{R}^{N\times d_{\text{model}}}$ we're obtaining a matrix $Z \in \mathbb{R}^{N\times N}$. Each row of this matrix is a vector $\vec{Z}_i$ which is a contextually enriched representation of the input embedding $\vec{E}_i$. This means that $\vec{Z}_i$ not only contains information about the $i$th input but also incorporates relevant information from other positions in the sequence, as determined by the attention weights $A_{ij}$.

# 2 – Multi-head self-attention

## Equations

In practice, the self-attention mechanism is often extended to **multi-head attention**. Instead of computing attention once, the model computes it multiple times in parallel, each with different learned projection matrices. This allows the model to capture information from different representation subspaces.

For each of the $H$ heads, we perform the self-attention operations:

$$\begin{aligned}
Q^{(h)} &= XW_Q^{(h)} \\
K^{(h)} &= XW_K^{(h)} \\
V^{(h)} &= XW_V^{(h)} \\
Z^{(h)} &= \text{Softmax}\left(\frac{Q^{(h)}{K^{(h)}}^\top}{\sqrt{d_k}}\right)V^{(h)}
\end{aligned}$$

We then concatenate the outputs from each head, and multiply by a final linear projector $W$:

$$Z = \text{Concat}\left(Z^{(1)},\ Z^{(2)},\ \ldots,\ Z^{(H)}\right)W$$

This approach allows the model to attend to multiple aspects of the input simultaneously, enhancing its ability to capture complex patterns and dependencies.

# 3 – Transformer block

## Equations

Let $X'$ be the input to the transformer encoder. The first step is to perform layer normalization on the input:

$$X = \text{LayerNorm}(X')$$

Then we proceed to process this $X$ as explained in the previous section to get $Z$. We add this to the original input of the transformer through a skip connection to get:

$$\text{MSA-Output} = Z + X'$$

We then apply Layer Normalization again, make a pass through a Multi-Layer Perceptron (MLP) of one hidden layer GeLU activation, and add this to a skip connection from the output of the Multi-Head Self-Attention network (MSA):

$$\text{Output} = \text{MLP}(\text{LayerNorm}(\text{MSA-Output})) + \text{MSA-Output}$$

with

$$\text{MLP}(x) = \text{GeLU}(x \cdot W_1 + b_1) \cdot W_2 + b_2$$

The full architecture can be observed in Figure 1. The only difference is that in our case we're not using masking (see "Mask" layer in (d)).

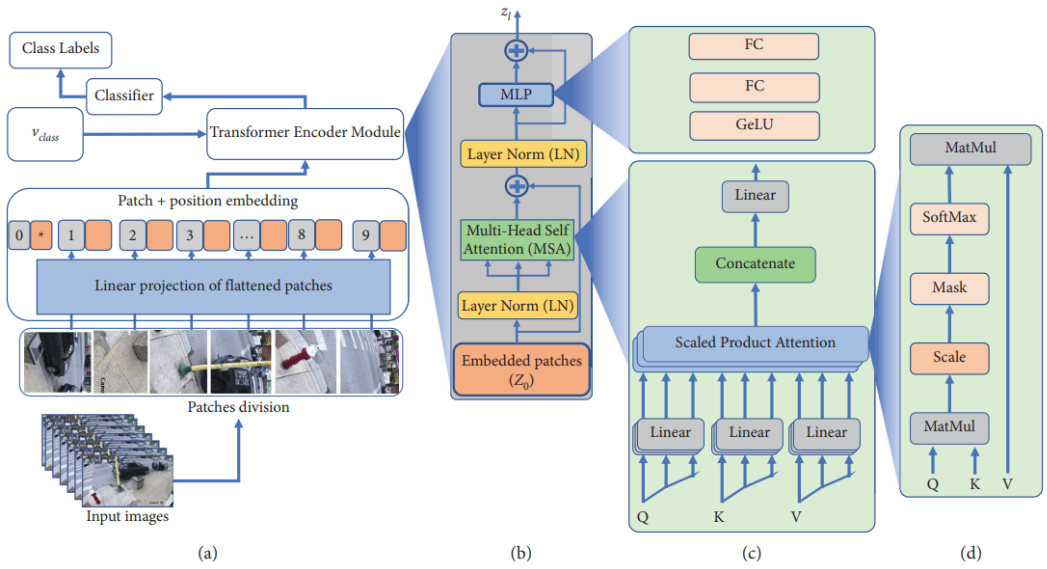


FIGURE 1 : The Vision Transformer architecture: (a) the main architecture of the model, (b) the transformer encoder module, (c) multiscale self-attention (MSA) head, and (d) the self-attention (SA) head. Hussain, A. et al (2022). Vision Transformer and Deep Sequence Learning for Human Activity Recognition in Surveillance Videos. Computational intelligence and neuroscience, 2022, 3454167. https://doi.org/10.1155/2022/3454167

# 4 – FULL VIT MODEL

## EXPLAIN WHAT IS A CLASS TOKEN AND WHY WE USE IT?

In ViT, a CLS token (short for "classification token") is a special learnable embedding that is prepended to the sequence of image patch embeddings before they are input into the Transformer encoder.

The CLS token serves as a means to aggregate information from all the image patches. After the sequence passes through the Transformer layers, the output corresponding to the CLS token captures a global representation of the entire image. This global representation is then used for classification purposes. By focusing on the CLS token's output, the model can make predictions about the overall

content or class of the image. Instead of processing and pooling over all patch embeddings, the CLS token provides a concise and effective way to summarize the entire image.

The concept of the CLS token originates from the field of natural language processing, specifically from the paper: **"BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding"** by Jacob Devlin et al., 2018. Transformer layers is used for classification tasks (e.g., sentence classification).

### Explain what is the the positional embedding (PE) and why it is important?

Positional embeddings are vectors added to the embeddings of image patches to encode their spatial positions within the input sequence. Since ViT models treat images as sequences by dividing them into fixed-size patches—and Transformers inherently lack positional awareness—these embeddings provide the necessary spatial context by indicating where each patch is located in the original image. This allows the Transformer to understand the arrangement and relationships between patches, enabling the self-attention mechanism to consider both the content and position of each patch. Incorporating positional embeddings, whether learned during training or predefined, gives the model the spatial awareness essential for accurately interpreting visual data.

The way this is generally implemented is using sinusoidal functions. In particular we're using the following definition:

$$\mathrm{PE}(i,j) = \begin{cases} \sin\left(\frac{i}{10000^{\frac{j}{d}}}\right) & \text{if } j \text{ is even} \\ \cos\left(\frac{i}{10000^{\frac{j-1}{d}}}\right) & \text{if } j \text{ is odd} \end{cases}$$

where $i$ is the embedding dimension and $j$ the position of the vector in the sequence. We visualize this in Figure 5. Here each vertical line is either a sine or a cosine looked "from above", and their period is increasing the larger the position in the sequence is.

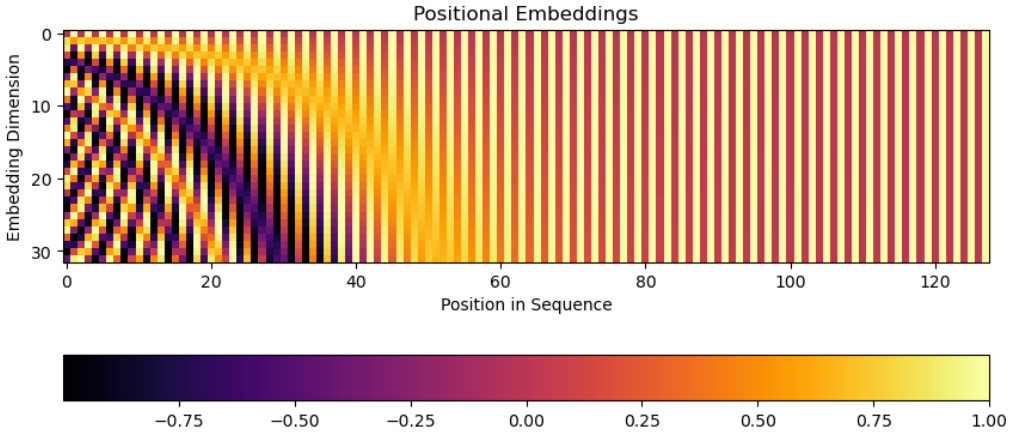


Figure 2 : Visualization of implemented positional embeddings.

# 5 – Experimental analysis

In this section, we will experiment with the hyperparameters of our Transformer model. These hyperparameters include the embedding dimension (*embed_dim*), patch size (*patch_size*), the number of Transformer blocks (*nb_blocks*), the number of heads ((*nb_head*) and the MLP ratio. By tuning these parameters, we aim to evaluate their impact on model performance and optimize the Transformer's capacity to learn intricate patterns in the data.

## Influence of the `embed_dim`

The hyperparameter `embed_dim`, or *embedding dimension*, defines the size of the vector representation for each token in the Transformer. A larger `embed_dim` allows the model to capture more nuanced and complex patterns within the data, as each token has more "space" to store information. However, increasing `embed_dim` also raises the model's computational cost and memory usage.

Through repeated experiments, we observe consistently better performance with `embed_dim` = 64. Interestingly, increasing `embed_dim` to 256 not only increases computational time but also leads to degraded performance. Indeed, spreading information across too many dimensions can dilute meaningful patterns, causing the model to focus on noise rather than valuable features. Therefore, a moderate `embed_dim` is needed.

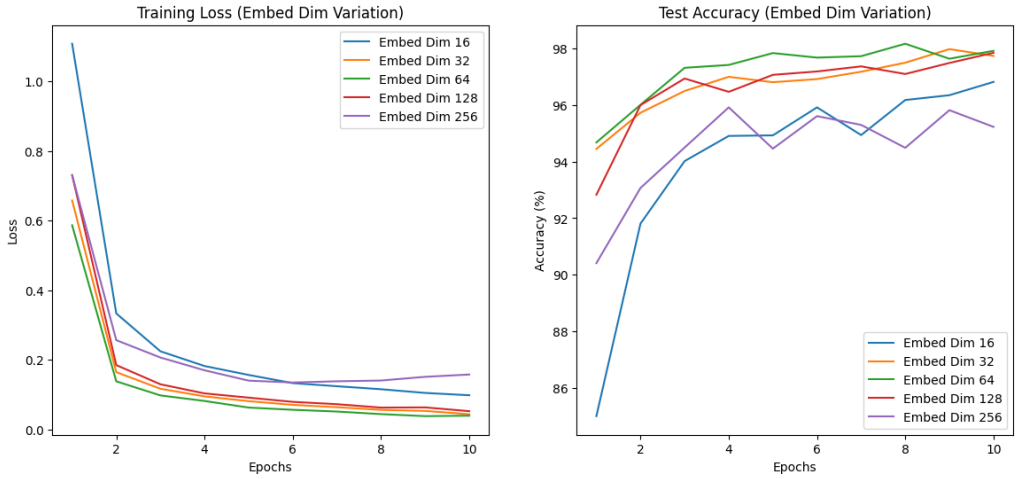


Figure 3 : Influence of the embedding dimension.

## Influence of the `patch_size`

The `patch_size` parameter in Vision Transformers (ViTs) determines the size of each patch into which an image is divided, impacting both performance and computational requirements. A smaller `patch_size`, such as 4 or 7, allows the model to capture fine-grained details by creating more tokens, which helps in learning subtle patterns in complex images. However, this also increases computational cost and memory usage due to the larger number of tokens. Conversely, a larger `patch_size`, such as 14 or 16, reduces the number of tokens, thus lowering computational requirements and training time, but at the risk of losing important details within each patch, as each one covers a larger image region. An intermediate `patch_size`=7 often strikes a balance, capturing essential details while keeping computation manageable, which is essential for optimal performance.

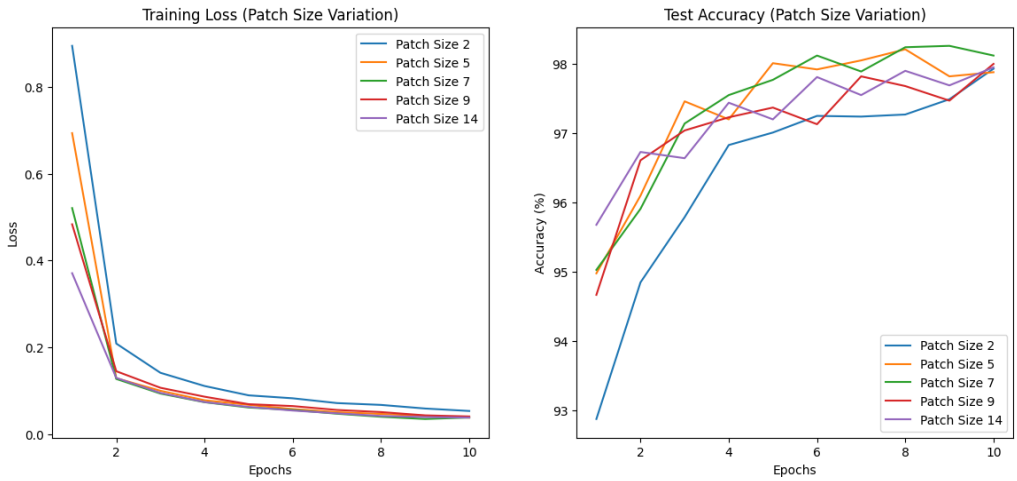


FIGURE 4 : Influence of the `patch_size`.

## INFLUENCE OF THE NUMBER OF BLOCKS

The number of Transformer blocks, affects the model's capacity to learn complex relationships and hierarchical features within the data. Increasing `nb_blocks` allows the model to capture deeper patterns, as each block refines the representation by learning additional transformations. However, a high number of blocks also increases computational cost and memory usage, and may lead to overfitting if the model becomes too complex for the training data. Conversely, a small `nb_blocks` value limits the model's ability to capture intricate patterns, potentially resulting in underfitting. Thus, finding an optimal number of blocks is crucial to balance depth of learning with computational efficiency and generalization.

Experimentally, we find that with 8 blocks, the model consistently shows good performance. This was also observed in the article **"An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale."**

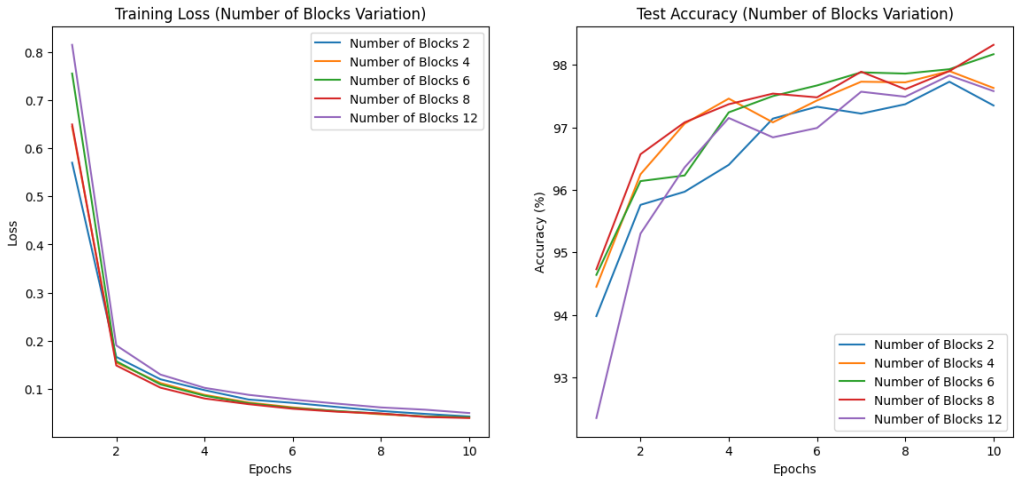


FIGURE 5 : Influence of the Number of Blocks.

## INFLUENCE OF THE NUMBER OF HEADS

The `num_heads` parameter in multi-head self-attention controls the number of distinct attention mechanisms in each Transformer layer. Increasing `num_heads` allows the model to capture a wider variety of relationships within the data, as each head can focus on different aspects of the input. However, a high `num_heads` also raises computational costs and risks overfitting by focusing on irrelevant patterns,

especially in smaller datasets. Conversely, a small `num_heads` can simplify training but may limit the model's ability to capture complex dependencies, leading to underfitting. Figure 6 shows that test accuracy improves slightly with an increase in the number of heads for an embedding size of 64; note that the number of heads must divide the embedding size, meaning that in this case the maximum number of heads possible is 32.

It is also worth mentioning that, in our study, the MNIST dataset is relatively simple, as it lacks complexity in features such as luminosity, colors, and high resolution. Thus, it is reasonable that the influence of the number of heads (or other hyperparameters) is not very significant in this case.

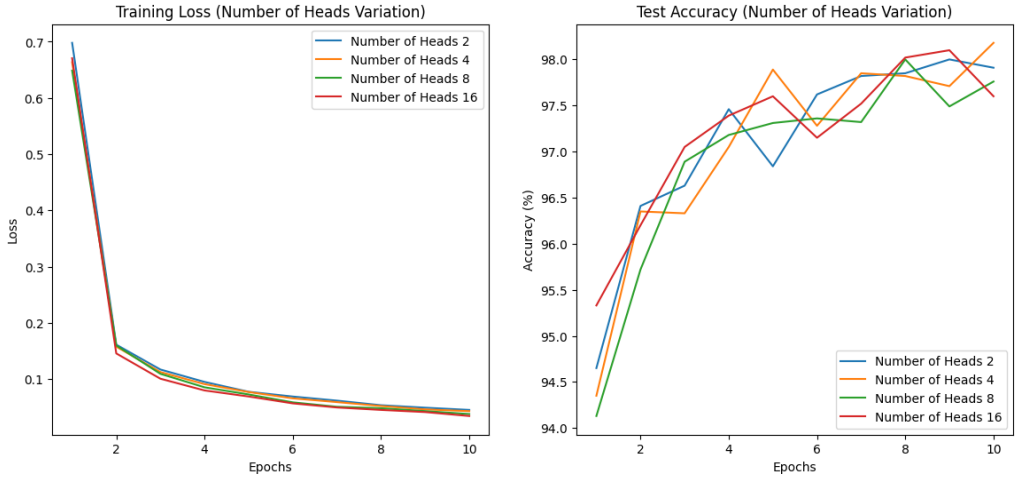


FIGURE 6 : Influence of the number of heads.

## INFLUENCE OF THE MLP RATIO

The MLP ratio parameter controls the size of the feed-forward network within each Transformer block, typically defined as a multiple of the embedding dimension. Increasing MLP ratio allows the model to capture more complex patterns by expanding the capacity of the hidden layer in the MLP, potentially improving performance on data with rich features. However, a higher MLP ratio also increases the number of parameters and computational cost. In Figure 7, we observe that increasing the MLP ratio beyond a certain point does not yield significant accuracy improvements on MNIST, as this dataset lacks the complexity typically required to benefit from a larger MLP size.

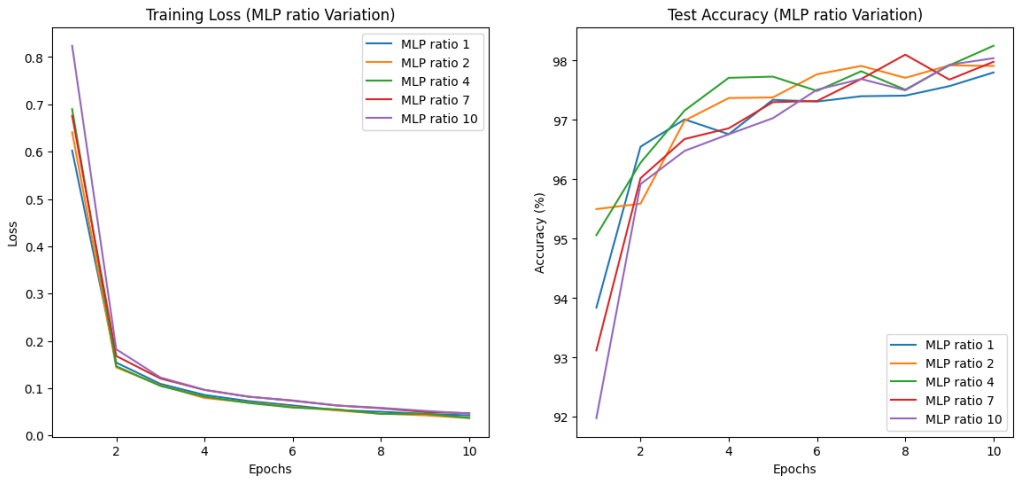


FIGURE 7 : Influence of the MLP ratio.

### What is the complexity of the transformer in terms of number of tokens? How you can we improve it?

The computational complexity of a Vision Transformer is quadratic $O(n^2)$ in the number of tokens $N$, due to the self-attention mechanism. We can improve it by reducing $N$ (e.g., using larger patches), or by using efficient attention mechanisms (like sparse or linear attention) that lower the complexity, making the model more scalable and efficient.

## 6 – Larger transformers

### What it is the problem with directly loading a model from `timm`? Explain if we have also such problem with CNNs.

The problem with directly loading a model from the `timm` library when working with ViTs is that these models are often pre-trained on datasets with specific characteristics that may not match those of the target dataset. ViTs are particularly sensitive to changes in input configurations due to their patch-based architecture. For instance:

- **Image Size**: ViTs divide images into fixed-size patches, and the number of patches (and thus the input dimension to the transformer) depends on the image size. If the dataset has images of a different size than what the pre-trained model expects, this mismatch can cause errors in the model's architecture.
- **Input Channels**: If the dataset has a different number of input channels (e.g., gray-scale images with one channel vs. RGB images with three channels), the initial embedding layer of the ViT must be adjusted accordingly.
- **Number of Classes**: The output layer of the ViT is configured for the number of classes present in the pre-training dataset.

In the case of Convolutional Neural Networks (CNNs), these issues are generally less problematic:

- **Image Size**: CNNs are more flexible with input image sizes due to the nature of convolutional layers, which can handle variable spatial dimensions. Techniques like global average pooling allow CNNs to adapt to different image sizes without significant architectural changes. Unlike ViTs, CNNs do not require the image to be divided into patches, eliminating issues related to patch size and count.
- **Input Channels**: We would still need to adjust the first convolutional layer if the input channels differ.
- **Number of Classes**: Changing the number of output classes in a CNN usually involves replacing the final fully connected layer.

### Comment the final results and provide some ideas on how to make transformer work on small datasets. You can take inspiration from some recent work.

The final results are summarized in the Table 1.

| Model | Implemented ViT | Timm ViT | Fine-tuned Timm ViT |
|---|---|---|---|
| Test accuracy after 10 epochs | **0.9664** | 0.8782 | 0.9065 |

Table 1 : Main results

We observe that the final results are decently good, even though the MNIST dataset is relatively small by transformer standards. This performance can be attributed to the pre-training performed on the Vision

Transformer (ViT) in both experiments, which provides a strong foundation for learning. However, the results are still inferior to those obtained with the manually created model, which could be due to several factors. For instance, the `timm` model is significantly larger and may require more examples to effectively learn the underlying patterns. Additionally, this model architecture is specifically designed for larger RGB images, which may not be optimal for the grayscale and simpler structure of MNIST images.

Upon further investigation, we found some common practices when training transformers on small datasets which we haven't implemented in this instance:

- Data Augmentation: Enhancing the diversity of the training data through techniques such as rotation, scaling, and cropping can help prevent overfitting and allow the transformer to generalize better from limited samples.
- Regularization Techniques: Implementing regularization methods tailored for transformers, such as dropout, stochastic depth, and layer-wise learning rate decay, can mitigate overfitting and enhance the model's ability to generalize from small datasets.
- Hybrid Architectures: Combining convolutional neural networks (CNNs) with transformer layers can harness the local feature extraction capabilities of CNNs alongside the global context modeling of transformers. This hybrid approach can be more effective on smaller datasets by capturing both fine-grained and holistic features.