POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Homework report

Image Joint Embedding Predictive Architecture

Scientific computing tools for advanced mathematical modelling

Authors: Giovanni Maria Bonvini, Davide Galbiati and Alessandra Gotti

Academic year: 2023-2024

Abstract

Our work introduces the Image-based Joint-Embedding Predictive Architecture (I-JEPA) [4], a self-supervised learning approach that generates highly semantic image representations without relying on ordinary hand-crafted data augmentatation, such as random scaling, cropping, and color jittering. Despite being quite effective for same-distributed datasets, this methods present some detriment and biases with different data distributions, which is why this procedures are generally called invariance methods.
Instead, I-JEPA predicts representations of various target blocks in an image from a single context block, emphasizing the importance of sampling large-scale target blocks and using informative, spatially distributed context blocks, to retrieve semnatic information. When paired with Vision Transformers (ViT), that account for semantic encoding, I-JEPA proves to be sufficiently accurate and computationally efficient on GPU's (at training level).

Cognitive learning theories suggest that biological systems adapt internal models to predict sensory inputs, a principle used in self-supervised learning, which is a type of machine learning where the system learns to predict part of its input from other parts of the input without needing labeled data. In traditional supervised learning, models are trained on a dataset where each example includes both an input and a corresponding correct output label. However, in self-supervised learning, the model creates its own labels from the input data. These methods reconstructs masked input patches to learn representations but often produce lower semantic-level representations compared to invariance-handcrafted-based approaches. As a result, they require more extensive adaptation, such as performating semantic classification tasks.
To this end, we introduce the Image-based Joint-Embedding Predictive Architecture (I-JEPA) to improve the semantic level of self-supervised representations without relying on hand-crafted augumentation. I-JEPA predicts missing information in an abstract representation space, which turns to be informative at semantic level.

The key features of our architecture are:

- **Abstract Prediction Targets**: I-JEPA adopts abstract targets instead of pixel level prediction, promoting the learning of semantic features.
- **Multi-block Masking Strategy**: The method emphasizes predicting large target blocks using an informative, spatially distributed context block, which represents an ensamble of patches.
- **Strong Representations**: I-JEPA learns strong off-the-shelf representations without hand-crafted view augmentations, outperforming pixel-reconstruction methods like basic-MAE.
- **Scalability and Efficiency**: I-JEPA is significantly faster and more efficient compared to other methods, thanks to semantic learning.

The proposed Image-based Joint-Embedding Predictive Architecture aims to predict the representations of various target blocks within an image given a context block. To do so, I-JEPA uses a Vision Transformer (ViT) architecture for the context-encoder and predictor at semantic-latent level (which is also a Trasformer Neural

Network), consisting in multiple layer working with a Attention and FeedForward mechanism that we will later explained. The core of our architecture is a fully connected MLPs that cocludes the decoding of the new-entirely predicted image, re-converting it to pixels. In our case, with a slightly difference from META-implemented I-JEPA, we evaluate the loss as a simple estimation of the difference with the recostructed missing image (pixel space) and the expected one. Differently, real I-JEPA performs a semantic loss taking the difference at semantic level between true masked patches and predicted ones. Nevertheless, differently from a elementary-MAE, our prediction occurs at a merely semantic level, which is the main strength of our work.

Contents

1. Dataset

The Dataset that we choosed to adopt is the MNIST dataset [3], both the Classical and the Fashion one [2]. The MNIST database (Modified National Institute of Standards and Technology database) is a large database of handwritten digits that is commonly used for training various image processing systems.
It is formed by a set of 70.000 grey-scale images with 28x28 pixels each. The Classical one consists of handwritten digits while the Fashion contains 10 different clothing items. Each image is associated to a label that represents the class to which it belongs.

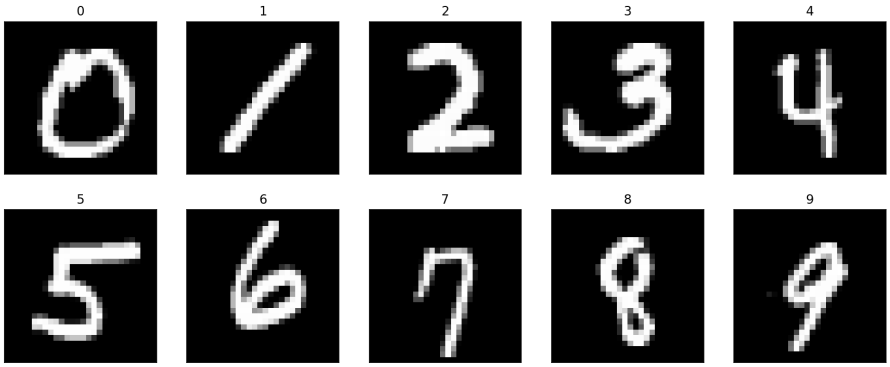

Figure 1: MNIST Classical

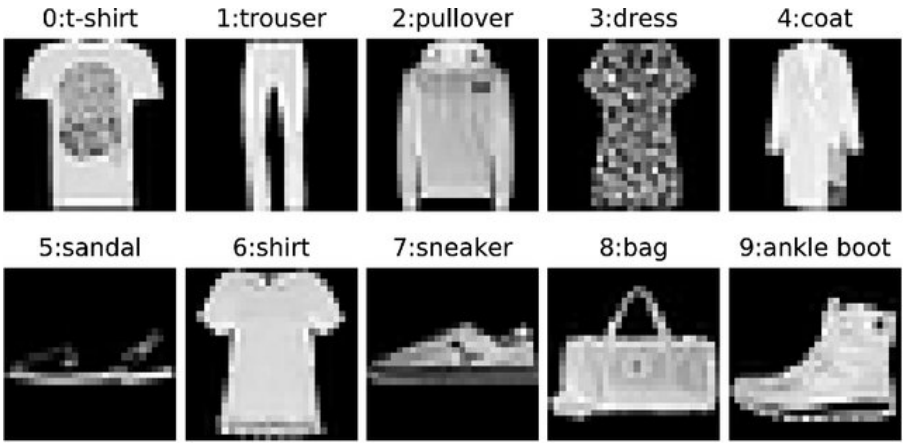

Figure 2: MNIST Fashion

2. Brief overview of ViT

Vison Transformer [1] (ViT) is an autoencoder architecture that adapts the Transformer architecture, which is highly successful in Natural Language Processing, to the domain of computer vision. This architecture is paramount for our model since it has the core parts of both the encoder and the decoder.
The key idea is to treat image patches as sequences of tokens, similar to how words are treated in text processing. The basic architecture is shown in Figure 3.

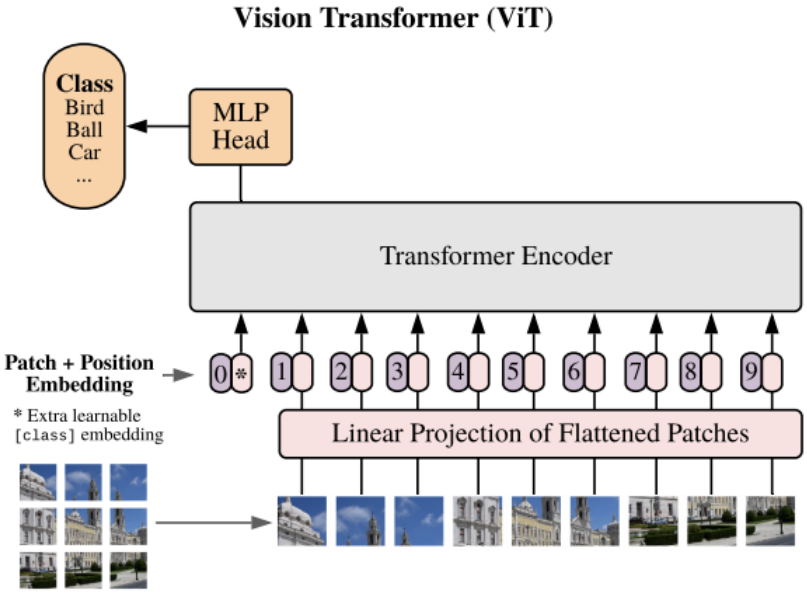

Figure 3: ViT architecture

The image is processed by the ViT in the following way:

- Input image is divided into a series of patches
- Patches are flattened and each one is associated with a positional information
- The patch + positional embedding is passed to the Transformer which outputs a vector for each embedding that contains semantic informations about the input patch

3. Masked Autoencoder (MAE) architecture

We report a schematic but detailed description of the complex architecture of the MAE autoencoder. In red we provide the Encoder architecture, while in blue the Decoder. Notice that both structures contain the Transformer component, which is at the core of the JEPA prediction.

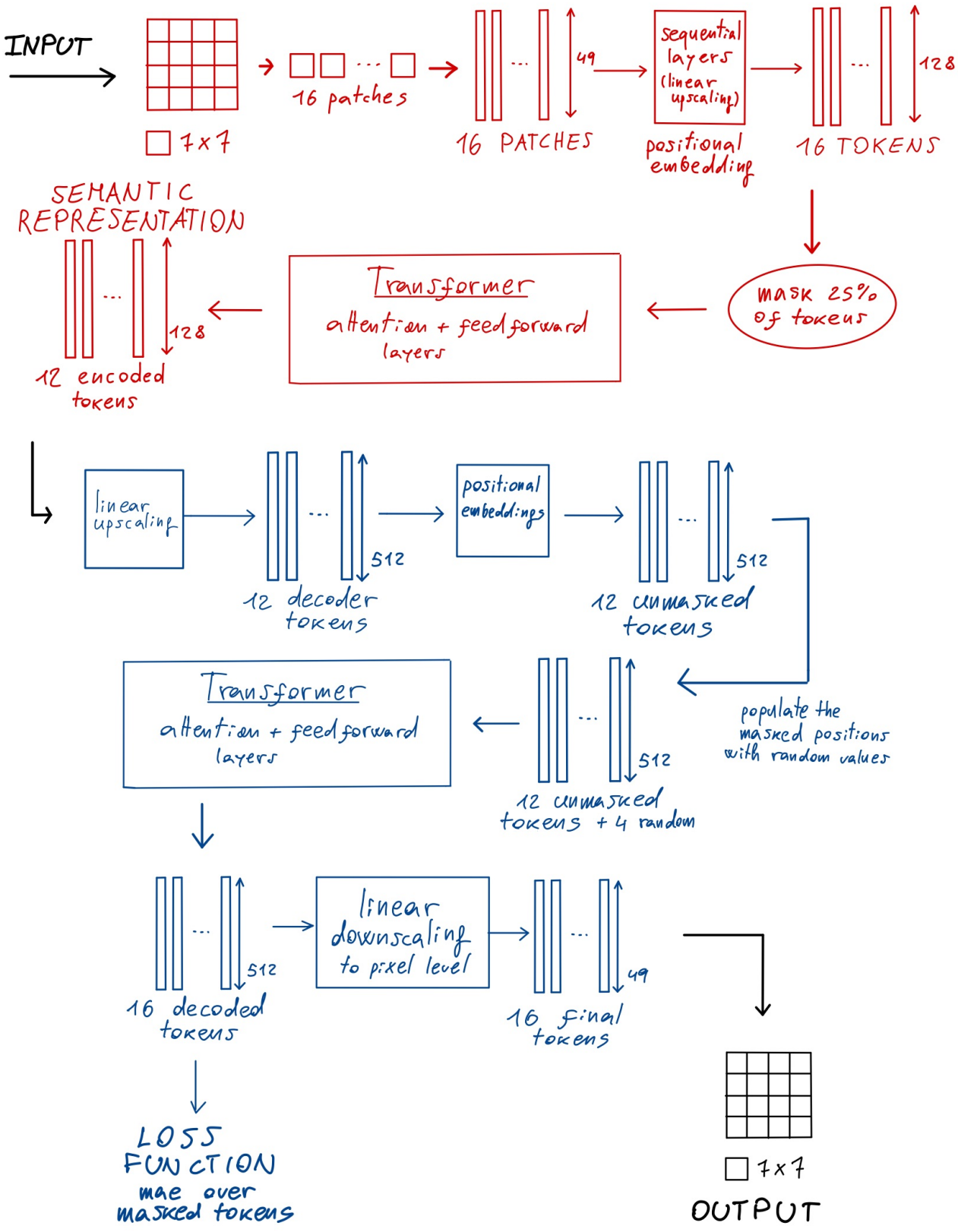

Figure 4: MAE architecture

4. Cronological steps of the Masked Autoencoder architecture (MAE)

First of all, in our code, thanks to the instance `device = torch.device('mps')` we force the machine to make use of GPU's resourches that reveal to be more efficient in all the process, especially at training level where a lot of effort is required to process the images. Thanks to the presence to lots of ALUs the training advances more rapidly in simple but large processing, like in the case of a great quantity of images.
We will discuss now, how the training process is performed, and then we will easily extend it at validation time. As first step we call the function

```
train_mae(mae_model, mnist_train_loader, num_epochs=200, device='mps')
```

that performs the training using the MNIST or Fashion-MNIST database that we described above, and 100 epochs. The optimzer we chose is Adam with a learning rate of 1e-4. The object `mae_model` is a PyTorch object that uses the class MAE and is defined as follows:

```
mae_model = MAE(
    encoder=encoder,
    decoder_dim=512,
    masking_ratio=0.25,
    decoder_depth=1,
    )
```

Such objects takes as input an **pre-trained ViT encoder** on the object dataset that is able to translate an image to a list of semantic vectors. We will discuss all the other parameters as the architecture unfolds.

4.1. Encoding

For every epoch, the training calls the method `def forward(self, img):` that performs all the operations. It's worth noticing that a lot of the following methods are proper of the `Encoder` model. As for `decoder_dim` parameter we chose it quite empirically on the power of 2 starting from 128-element decoded vector, then we tried 1024 elements, to find out that a good choice was of 512 element in the decoded space.
Every epoch trains all the images in the dataset at batches of 512 groups and divides each image in 16 pixel patches defining the alias `self.to_patch = encoder.to_patch_embedding` and hence calling

```
patches = self.to_patch(img)
```

The suitable number of patches and its px-dimension is calculated on the information of the images such as height (px) and width (px) and number of channels (1 for gray-scale images) as

```
num_patches = (image_height // patch_height) * (image_width // patch_width) #
    width, height = 7
patch_dim = channels * patch_height * patch_width
```

Unfortunately, with the computation power of our machines we were able to train nothing but gray-scale images, nonetheless the method is easily re-usable for RGB-images. In our case dividing each image in 16 patches, each one consists of 49 pixels. These dividing technique has been performed empirically looking at the 28x28 pixel origianl image, trying in ordiner to produce sufficiently large patches.
Then we defined the MAE method

```
self.patch_to_emb = nn.Sequential(*encoder.to_patch_embedding[1:])
```

that instructs for instantiating a simple neural network that simply projects each patch over a vector of 128 elements. The architecture is simple and it maily consists of a linear layer with normalization of input and outputs:

```
self.to_patch_embedding = nn.Sequential(
    nn.LayerNorm(patch_dim), #patch_dim = 49
    nn.Linear(patch_dim, dim), #dim = 128
    nn.LayerNorm(dim),
)
```

Normalization is a simple operation done at input and output level, necessary to guarantee numerical stability and efficiency of the NN. The trasformation of the data is performed first with a normalization over mean and variance as:

$$\bar{x} = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} \tag{1}$$

where $\epsilon = 1e-5$. Then, an affine trasformation scales and shifts the dataset as:
$y = \gamma \bar{x} + \beta$
Moreover, the linear projection onto a wider space, is just a simple trained linear operation that we express here in algebraic form

$$output_i = \sum_{j=1}^{49} (input_j \times W_{i,j}) + b_i \tag{2}$$

By

```
tokens = self.patch_to_emb(patches)
self.encoder.pool == "cls":
    tokens += self.encoder.pos_embedding[:, 1:num_patches]
```

We create the list of 128-element vectors of tokens adding also a positional embedding that accounts for the relative position between the patches. Every `self.encoder.pos_embedding[:, 1:num_patches]` vector is trained in the ViT and "describes" the relative position of some patch with respect to the others. This information will be semantically important for the model to predict the corrupted parts of the image. We notice here that the positional embedding of the original non-corrupted image is present at this level.
The corruption is perfomed in the following way:

```
num_masked = int(self.masking_ratio * num_patches)
rand_indices = torch.rand(batch, num_patches, device = device).argsort(dim = -1)
masked_indices, unmasked_indices = rand_indices[:, :num_masked], rand_indices[:,
    num_masked:]
# get the unmasked tokens to be encoded
tokens = tokens[batch_range, unmasked_indices]
# get the patches to be masked for the final reconstruction loss
masked_patches = patches[batch_range, masked_indices]
```

`num_masked` is calculated taking the input parameters equal to 0.25, in our case this clearly corresponds to as 4 patches are wiped out. Such corruption is randomic to enforce the natural possibility of an image to be corrupted in any possible way and to not introduce some "regularity patterns" that could spoil the force of our model.
Defining the indices of the patches to be deleted we automatically define the 12 remaining indexes. From now on thanks to the new definition `tokens = tokens[batch_range, unmasked_indices]` the model has no more information on the corrupted part of the image. It's worth noticing that corrupting the image at token level, is exactly equivalent to feed the MAE with a a corrupted pixel image. The main reason for working as we did resides in the fact that we need to use `masked_patches`, i.e. the patches that have been removed to evaulate the loss function.

4.1.1 Encoder Transformer

It is time now to produce the semantic embedding attending with the pre-trained ViT. This is performed calling the `transformer` method and defining `encoded_tokens = self.encoder.transformer(tokens)`. Such function is defined in the the encoder as

```
self.transformer = Transformer(dim = 128, depth = 1, heads = 8, dim_head = 64,
    mlp_dim = 128, dropout)
```

Such function creates a neural network with a single layer (depth = 1) with 8 attention neurons with a Feed-Transform-Forward mechanism that is capable of learning the correspondence between the encoded patches and understanding then semantic patterns.
The core function of this method is the so called `SoftMax` function that performs a scalar product between all the embeddings and outputs a distribution probability that describes how much each couple of vectors are correlated. The mathematical formulation of this tool is the following

$$attention\_weights = softmax(QK^T/\sqrt{(d_k)}) \tag{3}$$

]

$$softmax(\mathbf{z}) = \frac{\mathbf{exp(z_i)}}{\sum_{\mathbf{i}}^{\mathbf{K}} \mathbf{exp(z_k)}} \tag{4}$$

In brief, every embedded vector undergos a scalar product with the other ones thus creating a vector of (scaled-on-keys-dimension) scores. The current vector is called QUERY and the others are called KEYS. Softmax finally, transforms such scores into a distribution probability that sums to 1. Clearly, if some patch is very correlated to someother one, it will have a higher value in the corresponding place of the distribution probability vector that account for that 1-to-1 correlation.
In reality, our model works in a slightly more difficult way. In order to enhance the model's ability to focus on different parts of the input sequence simultaneouly. the transformer uses multi-head attention that involves splitting queries and keys into multiple smaller sets ($128/64 = 2$) and performs the attention mechanism in parallel. The outputs of each head are concatenated and through a linear trasformation gets the final vector of scores to give to the softmax. Then such correlation-semantic information is added to the previously defined vector and fed to FeedForward function that simply separates two attention layers with a single layer Neural Network, making use of GELU activation function. The use of GELU privides a smooth and most effective activation and it has the following mathematical form:

$$U(x) = 0.5x\left(1 + \tanh\left(\sqrt{\frac{2}{\pi}}\left(x + 0.044715x^3\right)\right)\right) \tag{5}$$

In our case, since depth is 1, only one Attention + FeedForward block is present. What follows is a snap of the architecture.

```
class FeedForward(nn.Module):
    def __init__(self, dim, hidden_dim, dropout = 0.):
        super().__init__()
        self.net = nn.Sequential(
            nn.LayerNorm(dim),
            nn.Linear(dim, hidden_dim),
            nn.GELU(),
            nn.Dropout(dropout),
            nn.Linear(hidden_dim, dim),
            nn.Dropout(dropout)
        )

class Transformer(nn.Module):
    def __init__(self, dim, depth, heads, dim_head, mlp_dim, dropout = 0.):
        super().__init__()
        self.norm = nn.LayerNorm(dim)
        self.layers = nn.ModuleList([])
        for _ in range(depth):
            self.layers.append(nn.ModuleList([
                Attention(dim, heads = heads, dim_head = dim_head, dropout = dropout)
                    ,
                FeedForward(dim, mlp_dim, dropout = dropout)
            ]))

    def forward(self, x):
        for attn, ff in self.layers:
            x = attn(x) + x
            x = ff(x) + x
```

It's worth noticing that semantic information is added to the embedded vector and now is ready to be decoded (where the prediction part takes place). The semantic vectors are assured to be the same 128 input dimension since the MLP projection in the feed forward is also 128.

4.2. Decoder

We call the function `decoder_tokens = self.enc_to_dec(encoded_tokens)` that projects the twelve 128-elements vector onto a 512-element space, for the reason we already said (empirical). Then, the positional embedding of the 12 patches (missing the 4 corrupted) is added through

```
unmasked_decoder_tokens = decoder_tokens + self.decoder_pos_emb(unmasked_indices)
```

This step is fundamental, since now the network has the information that only 12 patches are present in the encoded framework, differently from the initial tokens which were defined with the positional information of all the 16 patches. Nevertheless this last information is still present in the vectors as we can understand following

the architecture structure.
As a matter of fact we, report this last snap of code:

```
mask_tokens = repeat(self.mask_token, 'd -> b n d', b = batch, n = num_masked)
mask_tokens = mask_tokens + self.decoder_pos_emb(masked_indices)

# concat the masked tokens to the decoder tokens and attend with decoder
decoder_tokens = torch.zeros(batch, num_patches, self.decoder_dim, device=device)
decoder_tokens[batch_range, unmasked_indices] = unmasked_decoder_tokens
decoder_tokens[batch_range, masked_indices] = mask_tokens
decoded_tokens = self.decoder(decoder_tokens)
```

As we can see here we instantiate 4 tokens initially filled (first epoch) with random numbers and the positional embedding of the missing part of the image. `decoder_tokens` is the instantiaion of an object containg the 4 random patches and the unmasked embedded ones.
Now the function

```
\verb|decoded_tokens = self.decoder(decoder_tokens)|
```

actually makes the prediction making again use of the function. With the same architecture as before, the trasformer tries to predict the correlation between predicted tokens and unmasked ones trying to seek for relation between different patches.
`self.decoder()` is defines as follows

```
self.decoder = Transformer(dim = 128, depth = 1, heads = 16, dim_head = 64, mlp_dim =
    512)
```

As the model is trained, such correlation becomes more and more concrete and the prediction gains accuracy.

4.3. Loss computation

To compute the loss function we evaluate the *mse* between decoded masked tokens (linearly projected to pixel level) and the original masked patches.

4.4. Backward propagation and parameters update

The last step of the model consists obviously in updating the parameters backwardly with the loss function and advancing in the epoch cycle:

```
loss.backward()
optimizer.step()

total_loss += loss.item()
```

5. Final results

We provide here the final results obtained with the MAE architecture.

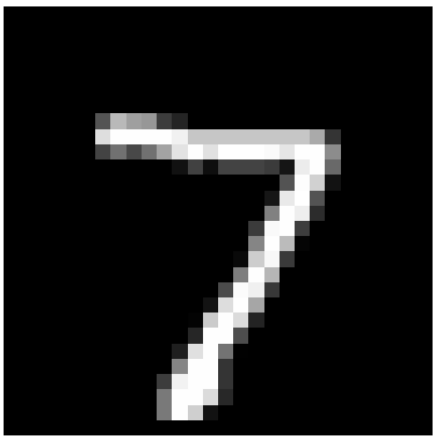

Figure 5: Original image

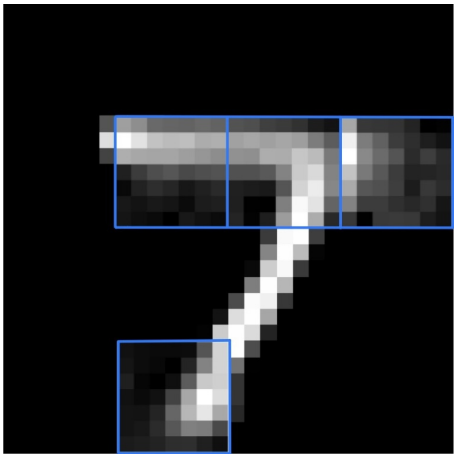

Figure 6: Reconstructed image

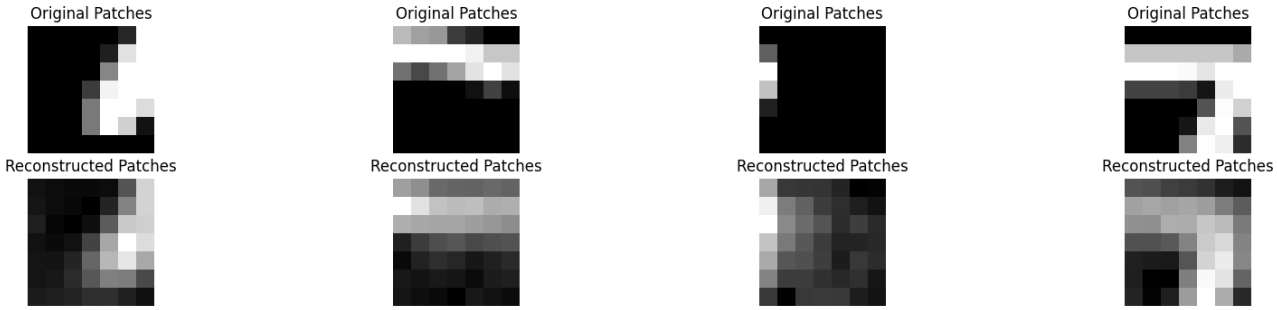

Figure 7: MAE results

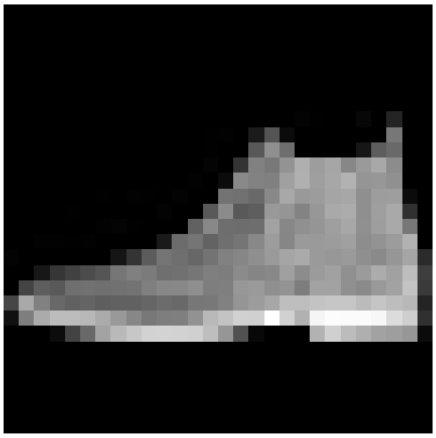

Figure 8: Original image

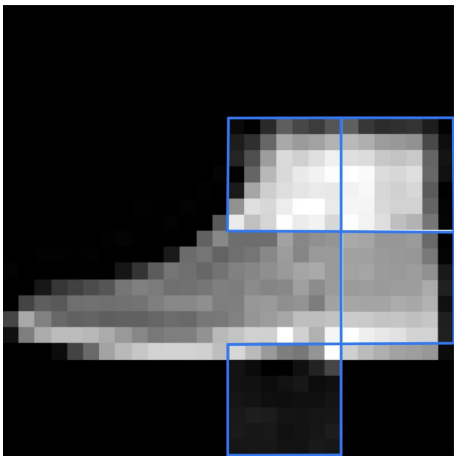

Figure 9: Reconstructed image

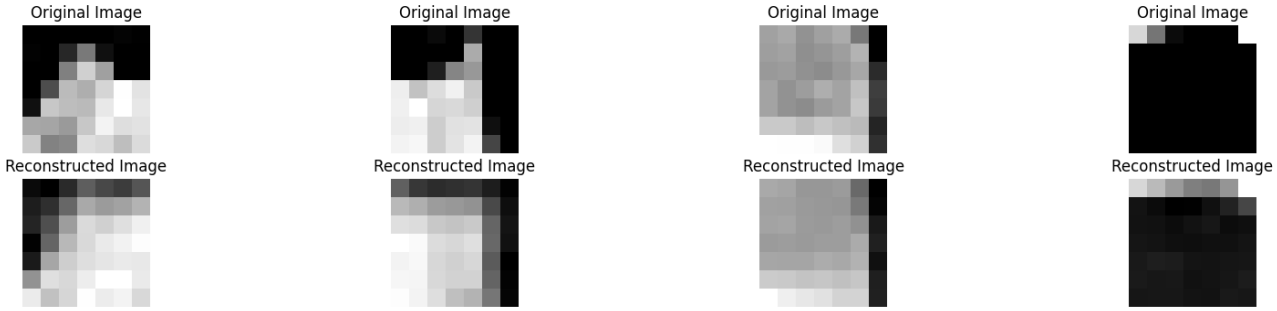

Figure 10: MAE results

We can observe that predictions are very accurate for both Classical MNIST and Fashion MNIST.

6. Conclusions

The architecture proposed proved to be an accurate and efficient model for prediction of missing components from corrupted images. This highlights the huge capabilities of semantic-level representations, suggesting a deep and meaningful comprehension of the initial corrupted raw data. This is the last step towards a fully semantic prediction, which differs from our structure only for the computation of the loss function that has to be performed at semantic level.

References

[1] https://github.com/lucidrains/vit-pytorch. MAE.

[2] https://pytorch.org/vision/0.18/generated/torchvision.datasets.fashionmnist.html. F-MNIST.

[3] https://pytorch.org/vision/main/generated/torchvision.datasets.mnist.html. MNIST.

[4] Duvall Q. Bojanowski1 P. Pascal V. Rabbat M. LeCun Y. Ballas N. Assran, M. Self-supervised learning from images with a joint-embedding predictive architecture. *Meta AI (FAIR), McGill University, Mila, Quebec AI Institute, New York University*, 2023.