# Homework 1 report

## Machine Learning for Signal Processing

Institute of Computer Science and Engineering

04/02/2024


***Supervised by :***
Do Huu Phu
dohuuphu25.ee11@nycu.edu.tw

***Produced par :***
PAULY Alexandre
alexandre.pauly@cy-tech.fr

# Contents

# 1 Introduction

**Context:**

As part of the Machine Learning for Signal Processing module, an image processing homework in Python has been initiated. This project is part of an educational approach aiming to concretize the theoretical concepts taught within this module, with a focus on image reconstruction through processing algorithms.

Our approach has focused on facial recognition. This field offers an opportunity to apply different image processing techniques through non-negative matrix factorization (NMF) and autoencoders (AE): two fundamental techniques in machine learning and data processing.

**Objective:**

In this report, we focus on understanding and implementing NMF and AE from scratch. The main objective is to explain in detail each step of the implementation of these two techniques, focusing on the underlying algorithms, network architectures, and training methods. Additionally, we compare the performance of NMF and AE on a dataset of real images to better understand their respective advantages and limitations. This study aims to provide a solid foundation for the practical use of these techniques in various machine learning and data processing applications.

# 2 Implementation

## 2.1 Non-negative Matrix Factorization (NMF)

Nonnegative Matrix Factorization (NMF) is a technique used to decompose a matrix into two non-negative matrices. The idea is to represent the input data as linear combinations of basis vectors, where these basis vectors and the combination coefficients are all non-negative. This makes NMF particularly useful for feature extraction and dimensionality reduction in domains such as computer vision and signal processing.

To implement NMF, we start by loading and preprocessing the data. In our case, we use a dataset of images, where each image is represented as a matrix of pixels. Then, we initialize matrices W and H with random values. These matrices represent the basis vectors and combination coefficients, respectively.

```
def initialize_nmf(V, k):
    # Dimensions initializations
    m, n = V.shape

    # Random matrix initialization
    W = np.random.rand(m, k)
    H = np.random.rand(k, n)

    return W, H
```

***Code 1 :*** *W and H initialization*

We then use an iterative algorithm to update matrices W and H to minimize the reconstruction error between the original and reconstructed data. This algorithm can be based on gradient descent methods or other optimization techniques. Here, it updates the values of matrices W and H until convergence is achieved or the maximum number of iterations is reached, avoiding infinite loops.

```
def update_H(V, W, H):
    # Separate calculation of numerator and denominator
    numerator = np.dot(W.T, V)
    denominator = np.dot(np.dot(W.T,W),H)

    # H update
    H *= numerator / denominator

    return H

def update_W(V, W, H):
    # Separate calculation of numerator and denominator
```

```
    numerator = np.dot(V, H.T)
    denominator = np.dot(np.dot(W, H), H.T)

    # W update
    W *= numerator / denominator

    return W
```

***Code 2 :*** *W and H update*

Thus, NMF mixes the data by identifying and grouping common features of the output images. This allows for a compact representation of the data while preserving important features for reconstruction. Furthermore, it is also possible to vary the number of latent components to adjust the quality of data representation and the reconstruction capacity. The more components are extracted, the more variations will be captured, but this can also lead to overfitting.

```
def nmf(V, k, max_iter=1000, tol=1e-6):
    # Step 1 : W and H initialization
    W, H = initialize_nmf(V, k)

    # Iteration until convergence or max iterations
    for i in range(max_iter):
        # Step 2 - W and H update
        H = update_H(V, W, H)
        W = update_W(V, W, H)

        # Cost function
        cost = np.sum((X - np.dot(W, H))**2)

        # Check convergence
        if cost < tol:
            print(f"Convergence achieved after. {i+1}
               iterations.")
            break

    return W, H
```

***Code 3 :*** *NMF application*

Finally, we visualize the results by displaying both the original images and the reconstructed images from NMF, along with a comparison to the results of the NMF implementation from the scikit-learn library. This allows us to evaluate the quality of the reconstruction and understand how NMF represents the input data as combinations of basis vectors.

**Advantages:**

- Very simple to implement.
- Analyzing the W and H components allows highlighting interesting characteristics such as the emergence of common features. By displaying these patterns, we can better understand the types of information captured by each latent component.

**Disadvantages:**

- NMF is a linear method and may struggle to capture complex nonlinear relationships between the data.
- The choice of the number of latent components can significantly influence performance, as well as the number of iterations.

## 2.2 Autoencoder (AE)

Just like NMF, the principle of the autoencoder aims to build a model capable of learning a compact representation of the data by reducing their dimensionality while preserving important features. This is done using two main parts: the encoder and the decoder.

The encoder consists of convolutional layers that transform the input data into a lower-dimensional representation. Here, the encoder is defined as a sequence of PyTorch convolutional layers. Each is followed by a ReLU activation function.

```
# Encoder
encoder = nn.Sequential(
    nn.Conv2d(1, 16, kernel_size=3, stride=2, padding=1),
    nn.ReLU(),
    nn.Conv2d(16, 32, kernel_size=3, stride=2, padding=1),
    nn.ReLU()
)
```

***Code 4 :*** *Encoder*

Conversely, the decoder is responsible for data reconstruction. It uses deconvolutional layers to increase the dimensionality of the latent representation and bring it back to the size of the input data. Basically, it reproduces the inverse path of the encoder and again uses a ReLU activation function on the first layer, followed by a sigmoid activation function to normalize the values of the reconstructed pixels on the output layer.

```
# Decoder
decoder = nn.Sequential(
    nn.ConvTranspose2d(32, 16, kernel_size=3, stride=2,
       padding=1, output_padding=1),
    nn.ReLU(),
```

```
    nn.ConvTranspose2d(16, 1, kernel_size=3, stride=2,
       padding=1, output_padding=1),
    nn.Sigmoid()
)
```

***Code 5 :*** *Decoder*

Once the encoder and decoder are defined, the autoencoder model is assembled by combining the two parts into a single architecture. The weights of the encoder and decoder layers are initialized randomly, and the entire model is trained simultaneously using a loss function based on the mean squared error (MSE) between the reconstructed data and the original data.

During training, the Adam optimizer is used to adjust the model's weights to minimize the loss function. The data is divided into mini-batches for training using a PyTorch DataLoader.

```
# Initialization of variables
num_epochs = 10           # Number of epochs
losses = []               # Loss values
k = 10                    # Number images to display
criterion = nn.MSELoss() # Mean Squared Error loss function
optimizer = optim.Adam(list(encoder.parameters()) + list(
   decoder.parameters()), lr=0.001) # Adam optimizer with
   learning rate 0.001

# For each epochs
for epoch in range(num_epochs):
    # Initialization of variable
    epoch_loss = 0 # Loss per epoch

    # For each images
    for data in dataloader:
        # Initialization of variables
        inputs, = data        # Data
        optimizer.zero_grad() # Gradients

        # Encoding
        encoded = encoder(inputs)

        # Decoding
        decoded = decoder(encoded)[:, :, :h, :w]

        # Loss calculation
        loss = criterion(decoded, inputs)
```

```
        loss.backward()

        # Weights update
        optimizer.step()

        # Sum of loss
        epoch_loss += loss.item()

    # Loss per epoch
    epoch_loss /= len(dataloader)
    losses.append(epoch_loss)

    # Message display
    print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {epoch_loss
        :.4f}')

# Results
with torch.no_grad():
    encoded_X = encoder(X_tensor)  # Encoded data
    decoded_X = decoder(encoded_X) # Decoded data
```

***Code 6 :*** *Autoencoder*

Once training is complete, the data is encoded using the encoder and decoded using the decoder to obtain the reconstructed images. Finally, the original images and the reconstructed images are displayed side by side to visually evaluate the performance of the autoencoder in reconstructing the input data, which we will discuss in the next section.

**Advantages :**

- This method can capture complex nonlinear relationships between data, making them more suitable for modeling highly nonlinear data such as faces.
- Capable of precisely reconstructing input data by minimizing reconstruction error (see Section 3).
- Once trained, autoencoders can generate new data by randomly sampling from the latent distribution.

**Disadvantages:**

- Requires a large amount of data to learn meaningful representations, which can be problematic in scenarios where data is scarce.
- Due to their complex architecture, autoencoders are more challenging to implement and optimize.
- Autoencoders can be prone to overfitting, especially when trained on small datasets.

# 3 Results

The data used in this study comes from the "Labeled Faces in the Wild" (LFW) dataset, which is often used for facial recognition and image classification. This dataset contains images of faces from different individuals, captured under various conditions such as lighting, viewing angle, and facial expression. Each image is represented as a matrix of grayscale pixels.

**Figure 1:** Data from LFW

Now that the methods and data have been presented, we can analyse the results.

## 3.1 Non-negative Matrix Factorization (NMF)

**Visual quality and accuracy:**

Overall, the reconstruction of images by NMF is simplified in details but remains faithful to the general facial shapes. This allows for the distinction of some facial features, although details like glasses or facial traits may not be fully captured, especially with a limited number of iterations. By increasing the number of iterations, for example to 1000, the quality of the reconstruction significantly improves, offering sharper images, although they may still lack some details.

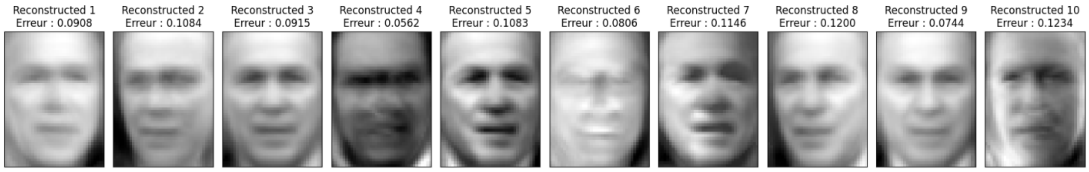


**Figure 2:** NMF with 10 latent components and 1000 iterations

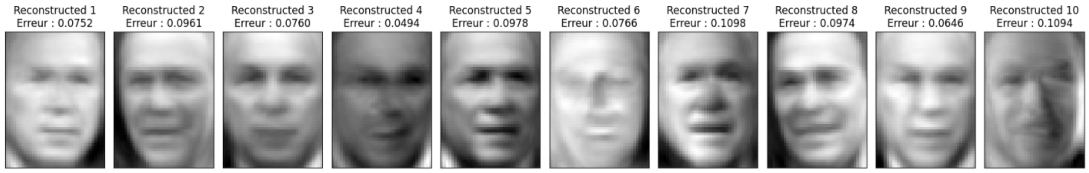


**Figure 3:** NMF with 20 latent components and 1000 iterations

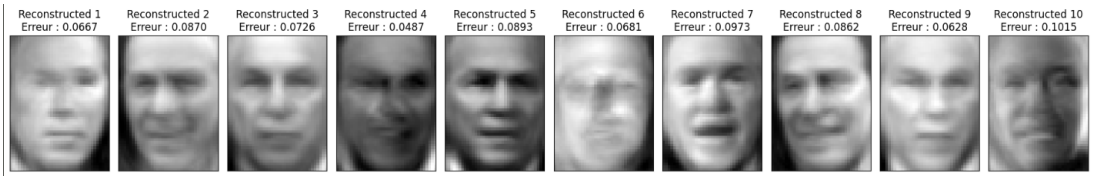


**Figure 4:** NMF with 30 latent components and 1000 iterations

As mentioned earlier, we can clearly observe the difference between 100 and 1000 iterations for 30 latent components. Even though the image is adequately reconstructed, it still appears generally pixelated.

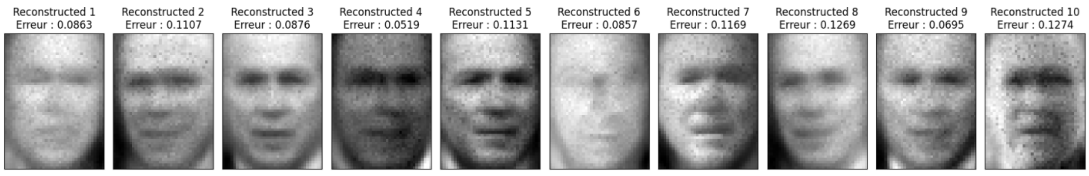


**Figure 5:** NMF with 30 latent components and 100 iterations

However, one advantage of NMF is the ability to analyze its resulting matrices, such as the W matrix from the following results where 10 latent components group common features across all faces in the database.

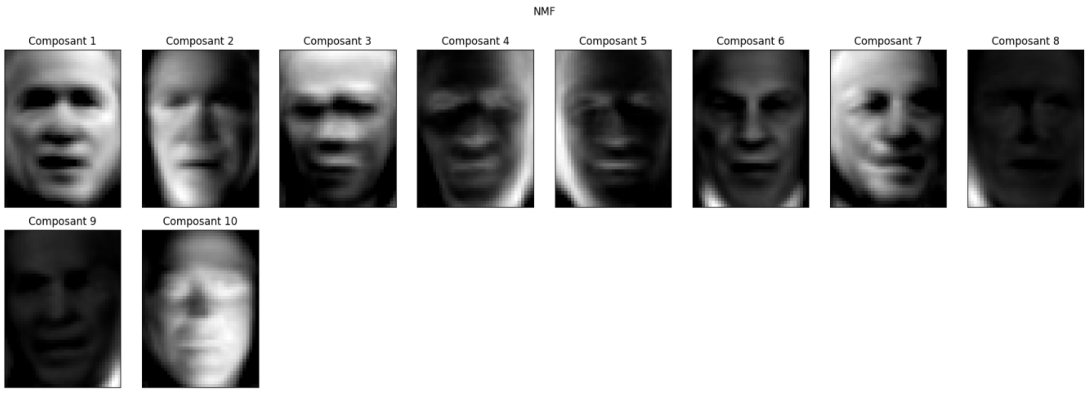


**Figure 6:** W matrix of the NMF with 10 latent components and 1000 iterations

**Training time :**

Regarding training time, NMF requires a significant investment. The required time heavily depends on the number of iterations needed to converge to a satisfactory solution. Thus, training time can vary from a few seconds for 100 iterations (5 to 10 seconds) to several minutes for 1000 iterations (on average 1 minute to 1 minute 30 seconds), which can be considered lengthy, especially for obtaining high-quality reconstructions.

## 3.2 Autoencoder (AE)

**Visual quality and accuracy:**

Overall, the image reconstruction by the autoencoder is of very high quality and very faithful to the original data. This is reflected in the low reconstruction errors for each image, especially when the autoencoder is trained for a number of epochs greater than 10. Below this threshold, the results may be slightly more pixelated, but without significant loss of information. Additionally, an improvement in the sharpness of the images and the amount of information reconstructed is observed as the number of epochs increases, as can be seen in the following cases :

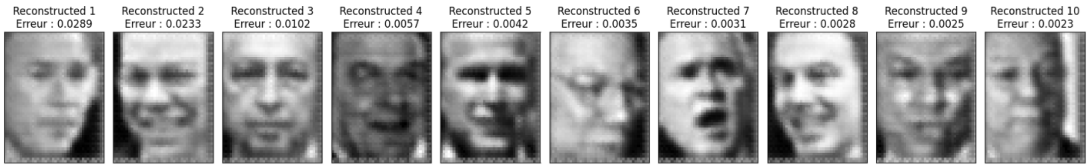


**Figure 7:** AE with 10 epochs

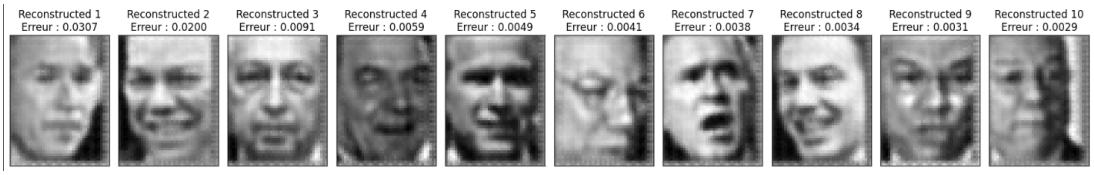


**Figure 8:** AE with 20 epochs

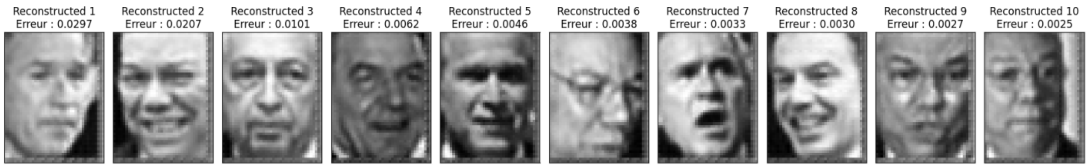


**Figure 9:** AE with 30 epochs

To be more precise, as the number of epochs increases, the appearance of certain features becomes more noticeable. Specifically, certain wrinkles or other details that appear more prominently than in the example with 10 epochs, and this is visible in most of the images. But to focus on one of them. The first one reveals some traits around the mouth as the epochs increase, or even the last image reveals the subject's glasses.

**Training time:**

In terms of efficiency, the autoencoder proves to be very effective. The training time for reconstructions ranging from 5 to 20 seconds on 10 to 50 epochs is relatively short, especially considering the high number of images in the dataset (1217 images).

# 4 Conclusion

The implementation and comparative analysis of Non-Negative Matrix Factorization and Autoencoder in the context of image reconstruction revealed significantly different results and interesting observations.

Firstly, we explored in detail the functioning of each method, highlighting their fundamental principles, architectures, and training processes. NMF was introduced as a matrix decomposition technique aimed at extracting non-negative features from data, while AE was described as a deep learning model capable of learning compact data representations using an encoder and a decoder.

Next, we applied these methods to a dataset of face images from the "Labeled Faces in the Wild" dataset. Practical implementation showed that NMF, although simpler in design, produces minimal reconstructions in terms of details, while AE offers more precise reconstructions with superior visual quality.

The analysis of the results revealed that AE has advantages in terms of visual quality and training speed, while NMF offers a more intuitive and interpretable approach while requiring more time to converge to quality reconstructions. However, NMF allows for additional analysis with the W and H matrices to highlight similar features.

In conclusion, although both methods have advantages and disadvantages, the choice between NMF and AE will depend on the specific requirements of the application, including reconstruction quality, training speed, and interpretability of the results. This work has thus contributed to a better understanding of image reconstruction techniques and their performances in different application contexts.

# References

[1] Lee, Daniel, and H. Sebastian Seung. "Algorithms for non-negative matrix factorization." Advances in neural information processing systems 13 (2000). https://proceedings.neurips.cc/paper_files/paper/2000/file/f9d1152547c0bde01830b7e8bd60024c-Paper.pdf

[2] Squires, Steven, Adam Prügel Bennett, and Mahesan Niranjan. "A variational autoencoder for probabilistic non-negative matrix factorization." arXiv preprint arXiv:1906.05912 (2019). https://arxiv.org/pdf/1906.05912.pdf