NTNU – Trondheim
Norwegian University of
Science and Technology

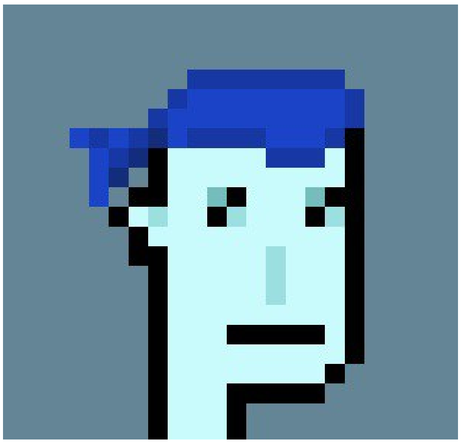
NTNU, Institutt for Matematiske fag

# TMA4320 vår 2022 - Industriell Matematikk-Prosjekt

Prosjektansvarlig: Martin Ludvigsen

March 28, 2022

## Dimensionality reduction and noise removal of face images with Non-Negative Matrix Factorization



Prosjektperiode: 25. Mars - 8. April

## Praktisk informasjon

Innleveringsfrist: Fredag 8. April, kl. 23:59
Språk: Engelsk (ikke alle som retter snakker norsk)
Format: Jupyter Notebook.
Vurderingsansvarlig: Martin Ludvigsen og Ergys Çokaj.
Mail: `martilud@ntnu.no`, `ergys.cokaj@ntnu.no`.

## 1 Introduction

The quickly growing field of machine learning has shown great potential in solving a wealth of tasks in a large variety of fields. An important subset of machine learning is dimensionality reduction, where the goal is to transform high-dimensional data into low-dimensional data that carries meaningful information about the original data. In many ways, dimensionality reduction is at the core of intelligence, i.e the capacity to understand, store and recall information. As the available data in the world is increasing at absurd and perhaps alarming rates, scientists and engineers need methods for reducing the amount of data, removing redundancies and extracting important structures. A field where this is important is images and computer vision. Datasets of images are usually large in size, but can be efficiently compressed with dimensionality reduction. Learning these compressed representations can be useful not only for understanding the data, but for a wealth of tasks including image denoising, which we will attempt to do this project.

In this project we will investigate a dimensionality reduction method called non-negative matrix factorization (NMF) and its application for image datasets.

## 2 Matrix factorization

We will now discuss dimensionality reduction in the context of matrix factorization. The goal is to discuss the so-called Non-negative Matrix Factorization (NMF), where a non-negative (meaning it has no negative components) $m \times n$ matrix $A$ is decomposed into two matrices

$$A \approx A_{\text{NMF}} = WH, \tag{1}$$

where $W$ is an $m \times d$ non-negative matrix, and $H$ is a $d \times n$ non-negative matrix, where $d$ can be significantly smaller than both $m$ and $n$. An illustration of NMF is shown in figure 1.

In this text, we call the matrix $WH$ the (NMF) reconstruction of $A$.

In many applications where matrix factorization is used, the columns of $A$ represent a data point, also called a sample, and the rows represent what is usually called a feature, which represents some quantity of interest. The opposite is of course also possible, but in this text we assume the the former.

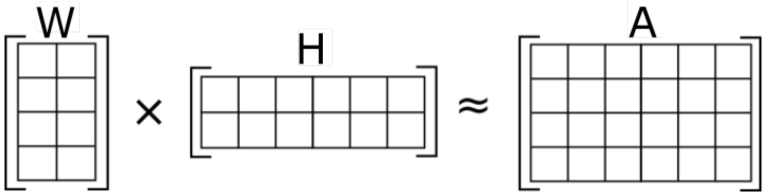


Figure 1: An illustration of Non-negative matrix factorization. A non-negative matrix $A$ is decomposed into two non-negative matrices $W$ and $H$. Note that $W$ and $H$ combined have less components than $A$. Source: https://en.wikipedia.org/wiki/Non-negative_matrix_factorization

| Application | Components of $A$ | Columns of $A$ | Rows of $A$ | Columns of $W$ |
|---|---|---|---|---|
| Movie ratings | Movie rating | Movie rater | Movie | Different genres |
| Cancer screening | Cancer risk | Patient | Time | Risk profiles |
| Spotify | Time listening | User | Song | Genres (Daily mix[1]) |
| Imaging | Pixel values | Images | Pixels | Image features |

Table 1: Examples of applications of NMF.

With a decomposition like NMF, we can write the $i$-th column of $A_{\text{NMF}}$, denoted $a_i$ as

$$a_i = Wh_i, \tag{2}$$

that is, a (non-negative) linear combination of the columns of $W$ with the weights given by the $i$-th column of $H$. In this sense, the columns of $W$ can be interpreted as approximate non-negative basis vectors for the range of $A$, and each column vector of $W$ should capture a different "hidden" feature of the matrix $A$. The columns of $W$ can also be called clusters. If $A$ is a square matrix, the columns of $W$ can be related to the eigenvectors of $A$, which we will see shortly.

Table 1 shows some concrete examples of different applications where NMF and similar techniques can be performed, and what $A$ and $W$ (hopefully) represent in these cases.

There are many reasons why applying a technique like the NMF can be useful. Firstly, it can drastically reduce the amount of data, which can make large datasets more manageable. Secondly, we can learn the underlying features (clusters) of the dataset. This can be useful to better understand large amounts of data, but is also useful in making predictions. Large companies like Netflix, Spotify and Youtube use these kinds of techniques to make predictions about what type of content you probably want to consume by clustering together users which prefer the same kind of content. More compact representations of data also has uses in problems like denoising, which is what we will investigate towards the end of this project.

[1] More complicated algorithms are most likely used to create personalized playlists, but NMF can nevertheless be applied to cluster which users listen to what type of music, and by extension be used to recommend music of a certain genre.

## 2.1 Movie example

To concertize things further, let us introduce a toy example. Assume we are given a matrix of 6 different users rating of 6 different movies between 1 and 5.

$$\begin{array}{r|cccccc|} & \text{User 1} & \text{User 2} & \text{User 3} & \text{User 4} & \text{User 5} & \text{User 6} \\ \text{Avengers: Infinity War} & 5 & 1 & 3 & 1 & 3 & 2 \\ \text{LotR: Fellowship of the Ring} & 2 & 4 & 1 & 4 & 1 & 4 \\ \text{Avengers: Endgame} & 4 & 1 & 5 & 1 & 4 & 2 \\ \text{LotR: Two Towers} & 1 & 4 & 2 & 5 & 2 & 4 \\ \text{Spiderman: Homecoming} & 3 & 1 & 5 & 3 & 3 & 2 \\ \text{LotR: Return of the King} & 2 & 4 & 2 & 4 & 2 & 4 \end{array}$$

In this case there are two "genres" of movies, Marvel movies and Lord of the Rings movies. The users are mostly split between liking one genre or the other. For example, user 1 seems to like the Marvel movies, but not the Lord of the Rings movies, and vice versa for user 2. All ratings are positive, meaning that the matrix $A$ containing the ratings is a non-negative matrix. We can thus attempt to factorize the matrix using NMF with $d = 2$ in hopes of "splitting" the dataset in two. The NMF is here calculated with the algorithm that will be shown in later sections[2].

$$\begin{bmatrix} 5 & 1 & 3 & 1 & 3 & 2 \\ 2 & 4 & 1 & 4 & 1 & 4 \\ 4 & 1 & 5 & 1 & 4 & 2 \\ 1 & 4 & 2 & 5 & 2 & 4 \\ 3 & 1 & 5 & 3 & 3 & 2 \\ 2 & 4 & 2 & 4 & 2 & 4 \end{bmatrix} \approx \begin{array}{c} \begin{matrix} w_1 & \quad w_2 \end{matrix} \\ \begin{bmatrix} 1.03 & 0.24 \\ 0.08 & 1.29 \\ 1.24 & 0.21 \\ 0.15 & 1.41 \\ 0.99 & 0.48 \\ 0.28 & 1.26 \end{bmatrix} \end{array} \begin{array}{c} \begin{matrix} h_1 & \quad h_2 & \quad h_3 & \quad h_4 & \quad h_5 & \quad h_6 \end{matrix} \\ \begin{bmatrix} 3.43 & 0.11 & 3.74 & 0.55 & 2.84 & 1.04 \\ 0.77 & 2.98 & 0.83 & 3.27 & 0.88 & 2.87 \end{bmatrix} \end{array}$$

We clearly see that the first column of $W$ has large values for the rows with the Marvel movies and small values for the rows with the Lord of the Rings movies, and vice versa for the other column. Thus, we have found two clusters in our dataset, where each column of $W$ correspond to these different clusters. Similarly, each column of $H$ correspond to each of the 6 users. We clearly see that each user prefers either Marvel movies or Lord of the Rings movies based on the two numbers in the columns of $H$ (larger number means larger preference). For example, the first column of $H$ has a large value corresponding to the first column of $W$, and a small value corresponding to the second column of $H$. This is again more or less equivalent to the fact that user 1 prefers Marvel movies and dislikes Lord of the Rings movies, as we have already mentioned. We can split both the

[2] And as we will discuss in later sections, the NMF is not unique, and we only show one possible solution here.

movies and users into two groups, which is perhaps easier to interpret than the individual ratings for each individual movie, which is what we started with.

It is also worth noting that we went from $6 \times 6 = 36$ components in $A$ to $2 \times 2 \times 6 = 24$ components in $W$ and $H$ combined. Thus, we have reduced the dimensions, and the total amount of data. The amount of data reduction is usually much more significant for larger datasets.

## 2.2 Some other factorization techniques.

The next subsection aims to lay a theoretical groundwork for matrix factorization, to hopefully provide some intuition about NMF.

### 2.2.1 Eigenvalue decomposition

An important tool from linear algebra is matrix diagonalization. A square, diagonalizable $n \times n$ matrix $A$ can be written as

$$A = PDP^{-1}, \tag{3}$$

where $D$ is a diagonal matrix $n \times n$ matrix and $P$ is an $n \times n$ invertible matrix. This is also called the eigendecomposition of the matrix $A$, because it can be calculated by selecting $D$ as the eigenvalues of $A$ and the columns of $P$ containing the eigenvectors of $A$. In the particular case where the eigenvectors of $A$ form an orthonormal basis, such that $P^T P = I$, matrix diagonalization can be written in the simpler form

$$A = PDP^T. \tag{4}$$

This is in particular the case when $A$ is what is called normal, which means it satisfies

$$A^T A = AA^T. \tag{5}$$

Any symmetric matrix ($A = A^T$) is also normal. Note that we can also write a diagonalizable matrix as the product of two matrices

$$A = WH, \tag{6}$$

with for example $W = P$ and $H = DP^{-1}$, where $W$ and $H$ are not necessarily non-negative. In the case where $W$ and $H$ are non-negative, this is of course an example of a NMF of the matrix $A$. In particular, the columns of $W$, which correspond to interesting features of $A$, are the eigenvectors of $A$. The eigenvectors that correspond to large eigenvalues carry more information about the matrix.

While diagonalization is useful, it has some weaknesses. This decomposition only exists if $A$ is square and diagonalizable, which is equivalent to the eigenvectors of $A$ being a basis for $\mathbb{R}^n$.

### 2.2.2 Singular Value Decomposition

There exists a generalization of matrix diagonalization called the singular value decomposition (SVD), which we will briefly introduce. The details here are not important, so don't be worried if you find this overwhelming. SVD is incredibly useful and finds its use in most fields imaginable, like quantum physics, statistics, numerics, and of particular importance to us: dimensionality reduction.

Any (potentially non-square) $m \times n$ matrix $A$ has a SVD decomposition

$$A = U\Sigma V^T, \tag{7}$$

where $U$ is an orthogonal ($U^T U = UU^T = I$) $m \times m$ matrix, $V$ is an orthogonal $n \times n$ matrix and $\Sigma$ is a rectangular, diagonal $m \times n$ matrix containing the so-called singular values of $A$. The singular values are usually ordered from largest to smallest, so that the largest singular value appears in the upper left corner of $\Sigma$. In the case where $A$ is normal with positive eigenvalues, the SVD coincides with the diagonalization given in (4), and the singular values are just the eigenvalues of $A$.

Similarily to equation (6), this implies that any matrix $A$ can be decomposed into two matrices $A = WH$, with for example $W = U$, $H = \Sigma V^T$.

When the SVD is applied to data equivalently to how we applied the NMF in the example in the previous section, it is usually called Principal Component Analysis (PCA), which is more or less the most common and most studied dimensionality reduction method [3].

We now come to the important part. In the definition of NMF given above, we asserted that we can attempt to factorize a $m \times n$ matrix $A$ into "small" matrices. For example, we can have $A \approx WH$ where $W$ is a $m \times d$, with $d < m$. The columns of $W$ often contain redundant information, and not all columns are equally important for reconstructing $A$. This should not come as a surprise. For a square matrix $A$, it is possible that the columns are linearly dependent, in which case some of its eigenvalues are 0. These linearly dependant columns are essentially redundant, and do not represent any new (useful) information. This leads us to the notion of reduced SVDs.

### 2.2.3 Reduced SVD

Reduced SVDs is a variant of SVD where we remove (or never compute) the least important parts of the SVD. Denote $U_d$ the $m \times d$ matrix obtained by removing the $m - d$ last columns of $U$. Similarily, denote $V_d^T$ the $d \times n$ matrix obtained by removing the last $n - d$ columns of $V_d$ (rows of $V_d^T$) and $\Sigma_d$ the $d \times d$ matrix obtained by removing rows and columns from $\Sigma$. A reduced SVD is then a factorization on the form

$$A \approx U_d \Sigma_d V_d^T. \tag{8}$$

There are two important results here. Firstly, if $A$ has only $d$ non-zero singular values (eigenvalues if $A$ is normal), then we have equality $A = U_d \Sigma_d V_d^T$. In other words, we

[3] More on this in TMA4267 Lineære statistiske modeller and TMA4268 Statistisk læring.

can safely remove redundant information and still achieve a perfect reconstruction. This is called the compact SVD, because it produces a much more compact representation of $A$. What is important for us is that this is a kind of dimensionality reduction. We can remove superfluous dimensions to receive a more compact representation.

This is not even the final form of reduced SVDs, and we can go even further beyond. Remembering that the singular values are usually sorted from largest to smallest, it is feasible to remove columns and rows corresponding to non-zero singular values. This means we are discarding "useful" information. The main point is that some components are more important for reconstructing $A$ than others, and the most important components are the ones that correspond to large singular values. This is called truncated SVD. An illustration of the SVD, compact and truncated SVD is shown in figure 2. It is possible to show that the best low-rank approximation to a $m \times n$ matrix $A$ is the truncated SVD. In other words, the solution to

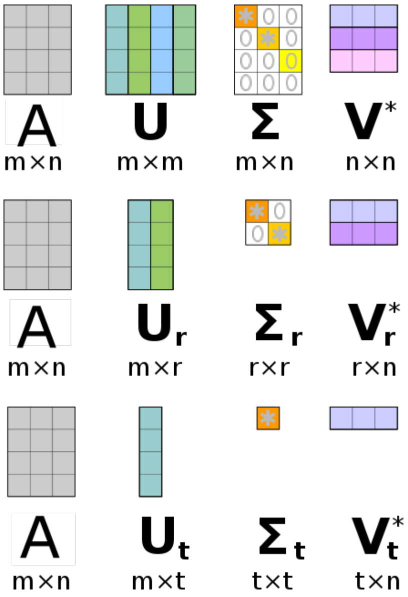


Figure 2: Top: Full SVD of matrix $A$. Middle: Compact SVD of $A$ obtained by removing redundant rows and colums. Bottom: Truncated SVD obtained by removing further rows and columns. Note that the reduced SVD perfectly reconstructs $A$, while the truncated SVD is in this case the best rank 1 approximation to $A$. Source: https://en.wikipedia.org/wiki/Singular_value_decomposition#Reduced_SVDs.

$$\min_{\hat{A}\in\mathbb{R}^{m\times n}} \|A-\hat{A}\|_F, \quad \text{s.t rank}(\hat{A}) = d, \tag{9}$$

is the truncated SVD $\hat{A} = U_d \Sigma_d V_d^T$. $\|.\|_F$ here refers to the Frobenius norm, which is one natural way of talking about distance between matrices. This becomes very important when the dimensions $m$ and $n$ are large, making the matrix $A$ hard to interpret, and the full SVD costly (and redundant) to compute.

The takeaway is that it is reasonable to look for compact factorizations of matrices. However, some of the results we stated here only hold for the SVD. What happens if we enforce non-negativity, which is what is done in NMF?

### 2.3 Non-negative Matrix factorization

The NMF of a $m \times n$ matrix $A$ can be obtained by solving the problem

$$\min_{W,H} \|A - WH\|_F, \quad \text{s.t } W, H \text{ non-negative}, \tag{10}$$

that is, find the non-negative $m \times d$ matrix $W$ and non-negative $d \times n$ matrix $H$ where $d$ is usually chosen as much smaller than $n$ and $m$. This type of problem is what is called a constrained optimization problem. These types of problems are often impossible to solve analytically, and hard to solve numerically[4].

From the discussion in the previous sections, a few things are apparent. The NMF of a matrix is a non-unique approximation, so there are usually infinitely many solutions $W$ and $H$. If you find this non-intuitive, think of the NMF for scalars: Find two non-negative numbers $w$ and $h$ so that when you multiply them together you get as close to a non-negative number $a$ as possible,

$$\min_{w,h} |a - wh|, \quad \text{s.t } w, h \geq 0. \tag{11}$$

The solution to this is obviously any $w$ and $h$ satisifying $w = a/h$ and $h = a/w$, which there are infinitely many of.

In multiple dimensions, things are only slightly more complicated. In particular, it is also possible that infinitely many reconstructions $WH$ solve the NMF problem (you will test this in task 1).

With eigendecompositions and the SVD, we can be sure that we can perfectly recreate the matrix $A$. In particular, a rank $d$ matrix $A$ can be perfectly recreated by a truncated SVD with $d$ components. One would similarly in general expect that a rank $d$ matrix $A$ can be perfectly recreated by an NMF with $d$ components, but this is not always the case, and there is no theoretical guarantee this is true. It is only possible for the NMF to perfectly recreate $A$ if the NMF captures the non-negative features of $A$, and it is in general hard to say for what matrices this property holds. This is also something that you will test in task 1. We can however, expect that as $d$ increases up to the rank of $A$, the NMF yields a better approximation to $A$.

Why would we want to calculate the NMF as opposed to the SVD? There are a couple of arguments to why this makes sense:

- Images, are inherently non-negative, so it makes sense to try to decompose an image into a combination of other non-negative vectors, that is, other images. If non-negativity is not enforced, interpreting the resulting decomposition becomes difficult.

[4] More on this in TMA4180 Optimization 1

- In the case where $A$ has rank $d$ less than $m$ and $n$, we know that the matrix contains redundant information. We can thus hope that the NMF is able to reduce the redundant information, similarly to the truncated SVD.

- NMF has been researched for the last few decades because it has some interesting properties. We will in particular see that it produces interesting basis vectors, stored in $W$.

- The non-negativity constraint can prevent so-called overfitting, which will become relevant in task 3.

On the other hand, NMF has a less developed theory, and is in general much more expensive to compute than truncated SVD.

### 2.4 Numerical Algorithm for NMF

The algorithm we will use is a multiplicative update rule proposed by Lee and Seung [1]. The algorithm updates $W$ and $H$ are

$$H_{k+1} \leftarrow H_k \odot (W_k^T A) \oslash (W_k^T W_k H_k), \tag{12}$$

$$W_{k+1} \leftarrow W_k \odot (A H_{k+1}^T) \oslash (W_k H_{k+1} H_{k+1}^T). \tag{13}$$

The subscript $k$ denotes the iteration. Here $\odot$ denotes elementwise product and $\oslash$ denotes elementwise division. For example with $2 \times 2$ matrices

$$A \odot B = \begin{bmatrix} a_1 & a_2 \\ a_3 & a_4 \end{bmatrix} \odot \begin{bmatrix} b_1 & b_2 \\ b_3 & b_4 \end{bmatrix} = \begin{bmatrix} a_1 b_1 & a_2 b_2 \\ a_3 b_3 & a_4 b_4 \end{bmatrix}, \quad A \oslash B = \begin{bmatrix} a_1/b_1 & a_2/b_2 \\ a_3/b_3 & a_4/b_4 \end{bmatrix}. \tag{14}$$

This is also called Hadamard product and Hadamard division. Make sure you understand the difference between the matrix product $AB$ and the elementwise matrix product $A \odot B$. We can also define the updates

$$(H_{k+1})_{ij} \leftarrow (H_k)_{ij} \cdot \frac{(W_k^T A)_{ij}}{(W_k^T W_k H_k)_{ij}}, \tag{15}$$

$$(W_{k+1})_{ij} \leftarrow (W_k)_{ij} \cdot \frac{(A H_{k+1}^T)_{ij}}{(W_k H_{k+1} H_{k+1}^T)_{ij}}, \tag{16}$$

where $(A)_{ij}$ denotes the component at the $i$-th row and $j$-th column of the matrix $A$.

It can be shown that this algorithm converges to a local minimizer of (10) and in particular, the updates satisfy

$$\|A - W_{k+1} H_{k+1}\|_F \leq \|A - W_k H_k\|_F, \tag{17}$$

It is here quite crucial that the algorithm only finds a local minimizer. This means that the algorithm can only find an approximate solution to (10), and convergence can be slow depending on initial conditions.

We thus have two sources of error in the NMF approximation: $A$ might not be suitable for the non-negative factorization, and the algorithm might not converge to a good solution.

This update rule is quite popular because it is easy to implement, though there exist other, potentially more efficient algorithms. This algorithm has a few caveats

- One needs to ensure that $H$ and $W$ are initialized to positive matrices (why this is true is part of task 1). One of the simplest ways of doing this is to initialize $H$ and $W$ with random positive elements, for example by sampling from a uniform distribution between 0 and 1 with `np.random.uniform`. In order to make this efficient, it is useful to sample all the random numbers required in one go, and reshape the resulting vector. For example, if you need $n$ uniform random numbers between 0 and 1, you can do something like `np.random.uniform(0.0, 1.0, n)`. You should also scale the matrices appropriately, which can be done by multiplying the randomly initialized vectors with $\sqrt{\text{mean}(A)/d}$. Use `np.mean` for this.

- It is possible that an element of the denominator of either update is 0. One of the easiest ways to avoid this is to add some small number $\delta$ to each element of the denominator whenever the division is calculated. Adding this term can actually have additional beneficial effects to the algorithm. Suitable values are $\delta \in [10^{-9}, 10^{-6}]$.

- There exists converge criterions that can be used to determine whether or not the algorithm has converged [2]. However, the optimality conditions can be difficult and expensive to calculate. The simplest solution, which we will apply, is to just to stop the iterations after a certain number of iterations.

.

**Algorithm 1** Lee and Seung's multiplicative update rule for NMF

```
Input: $m \times n$ matrix $A$, number of components $d$ and safe division constant $\delta$.
Output: Approximate NMF factorization $W$ and $H$ so that $A_{\text{NMF}} = WH$.

Initialize $(W_0)_{ij} \sim \text{Unif}(0,1)$ and $(H_0)_{ij} \sim \text{Unif}(0,1)$.
Scale initial values $W_0 \leftarrow \sqrt{\text{mean}(A)/d} \times W_0$, $H_0 \leftarrow \sqrt{\text{mean}(A)/d} \times H_0$
for $k = 1$ to maxiter do
    Update $H_{k+1} \leftarrow H_k \odot W_k^T A \oslash (W_k^T W_k H_k + \delta)$
    Update $W_{k+1} \leftarrow W_k \odot A H_{k+1}^T \oslash (W_k H_{k+1} H_{k+1}^T + \delta)$
end for
```

Here are some useful NumPy functions for implementing such an algorithm:

- Calculating the transpose of a matrix $A$ can easily be done in NumPy by typing `A.T` (where `A` is a NumPy array), or using the `np.transpose()` function.

- There are a few ways to calculate the matrix product $AB$. The most common way in Python 3 is to write `A @ B`, but this is not compatible with Python 2. You can also use `np.dot(A,B)`, which is equivalent to `A.dot(B)` and `np.matmul(A,B)`. Note that $A$ and $B$ must be of proper dimensions, and I guarantee that over the course of the project you will get errors because your matrices won't be of proper shape. You can check the shape of an array by printing `A.shape`, which returns a tuple of the dimensions of the array.

- For componentwise multiplication and division you can most likely just write `A * B` and `A / B`, but you can also use `np.multiply` and `np.divide` to have more control.

- Another important function is `np.reshape`, which can be used to change the shape of a matrix. It can also be called like `A.reshape(newshape)`, where `newshape` is a tuple containing the new shape. You might not need this for task 1, but you will certainly need it for task 2 and 3.

- You will be asked to calculate $\|A-WH\|_F$, which can be done using `np.linalg.norm`. In order to get the Frobenius norm you can use something like `np.linalg.norm(A - np.dot(W,H), 'fro')`.

- Because the initial vectors are random, I would recommend also taking in what is called a random seed as input of the function, and calling the function `np.random.seed` with this input at the beginning of the function. This way, you get consistent results each run.

- Finally, you might find some use for the `np.nonzero` and `np.where` functions, but you will have to figure out how to use them yourself!

## 3 Task 1

Even though the tasks are given in a list, try to answer them in a fluent text. Make sure that every question has been answered in your reports.

We will first investigate some of the properties of the algorithm. You don't need to answer the next few questions in great detail. Short and concise answers are better.

- **a)** Assume that $A$ is non-negative and that $W$ and $H$ are initialized as positive. Show that all iterates $W_k$ and $H_k$ are non-negative.

- **b)** Confirm that $H_{k+1} = H_k$ (that is, $H_k$ is a fixed point) if $(W^T A) \oslash (W^T WH)$ is a matrix of ones. Show that this happens if $WH = A$.

- **c)** Are there any difficulties with setting the initializations $H_0$ and $W_0$ equal to matrices containing only zeros?

Throughout the rest of this task, we will investigate the four matrices

$$A_1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad A_2 = \begin{bmatrix} 1 & 2 \\ 1 & 1 \\ 1 & 2 \end{bmatrix}, \quad A_3 = \begin{bmatrix} 2 & 1 & 1 \\ 2 & 1 & 1 \\ 1 & 1 & 2 \end{bmatrix}, \quad A_4 = \begin{bmatrix} 2 & 1 & 0 \\ 1 & 2 & 3 \\ 0 & 3 & 3 \end{bmatrix}.$$

In the following experiments, it might be wise to select the maximum number of iterations to 1000 to ensure that the algorithm has converged. These tasks should also be used to test if your implementation is correct. Because we use a random initialization, you might get slightly different result each time you run the code. You can use `np.random.seed` to make sure you get the same results each time. In any case, feel free to run the code several times, but only report a single run.

- **d)** Implement the NMF algorithm 1. Calculate the NMF of $A_1$ and $A_2$ with $d = 1$ with two different random initializations. Print $W$, $H$, $WH$ and $\|A - WH\|_F$ for both experiments (you can choose whether you want to present the results as an output from a code chunk or in a table, but a table is more readable). What do you observe? What quantities are unique? (Hint: For $A_1$ you should get $\|A - WH\| = 1$ as long as the initialization satisfies non-negativity).

- **e)** Calculate the NMF of $A_1$ and $A_2$ with $d = 2$. Is $\|A_1 - WH\|_F$ close to zero? Is the result reasonable given the matrices? (Hint: for $A_2$ you might get stuck at a local minimizer, particularly if you use few iterations)

We will now extend the algorithm slightly. Extend your implementation so that for each iteration, you store the value of $\|A - W_k H_k\|_F$ and return an array containing these values at the end of the program. I recommend not using any append functions for this, but allocate the whole array before running the loop.

- **f)** What is the rank of $A_3$ and $A_4$? (Hint: A rank $r$ matrix has $r$ non-zero eigenvalues/singular values. Use for example `np.linalg.eig` or `np.linalg.svd`.).

- **g)** For $A_3$ and $A_4$, run the NMF algorithm for $d = 1$, $d = 2$ and $d = 3$ and plot $\|A - W_k H_k\|_F$ as a function of number of iterations in two separate plots (one for $A_3$ and one for $A_4$). Scale the $y$-axis logarithmically, using for example `plt.semilogy`. Is equation (17) satisfied? How do the results depend on $d$?

Make sure that your code works as expected before moving on to task 2.

# 4 Images

We now turn to the area of interest in this project, images. In this project we represent a digital image $v$ as a discrete vector[5]. An image has width $m_x$ and height $m_y$, where $m_x$ and $m_y$ are integers. These integers together are usually called the resolution of the

[5] Or tensor.

image. At each coordinate exists the smallest discernible image element, usually called a pixel. Each of these pixels can have up to three color channels, with one of the most common models being the RGB (red, green, blue) model. In the RGB model, each pixel is represented by three numerical values, one for each of the colors red, green and blue. The linear combination of these three colors can effectively represent color images. If you look at your computer or phone screen really close, you might see that it consists of rows and columns of small red, green and blue lights. Thus, an $m_x \times m_y$ RGB image can be represented by a $m_x \times m_y \times 3$ NumPy array.

An image stored on a computer has a quality called the bit depth, which is how many bits are used to store the value for each color channel of each pixel. One of the most common bit depths used for images is 8-bit, which means that each color of each pixel is an integer value between 0 and 255. When there are three color channels, this is also referred to as 24-bit, as each pixel requires 24 bits of storage. This might not sound like much, but 24-bit corresponds to $16,777,216$ different colors, which is more than what a human eye can discern. In digital image processing, it is often more convenient to represent each color channel of each pixel as a real number between 0 and 1, and more or less ignore this bit depth. We will throughout this project assume that each color channel of each pixel has a value between 0 and 1 (which is important for functions like `plt.imshow` to work properly). Note that this means that images are non-negative vectors.

It is also common for images to have a fourth channel, usually called the alpha channel, which represents opacity, which is another word for transparency. Such an image is called an RGBA image, where the "A" stands for "alpha". This allows for different images to be layered on top of each other. In this project, we are mainly concerned with pixel values with opacity 0 or 1, that is, pixels with no transparency and completely transparent pixels respectively. The opaque pixels are the pixels that constitute the image, and the transparent pixels constitute the background of the image. An example of a picture with and without alpha channels is shown in figure 3

## 4.1 CryptoPunks dataset

CryptoPunks [6] is one of the most popular Non Fungible Tokens (NFT), a form of digital currency, which has become popular over the last few years. For our purposes, a CryptoPunk is a digital $24 \times 24$ RGBA image of a so-called "cyberpunk" character [7]. A dataset of all CryptoPunks can be found on Kaggle: https://www.kaggle.com/tunguz/cryptopunks, but they will also be handed out with this text.

An illustration of a handful of CryptoPunks is shown in figure 4.

There are 10.000 algorithmically generated CryptoPunks, where some of the different traits, like hairstyles, glasses, cigarettes etc. are shared by several different characters.

[6] https://larvalabs.com/cryptopunks

[7] Technically, a CryptoPunk is data stored on the Ethereum blockchain, which sounds like SciFi jargon, but is actually true. As of October 2021, the entire collection of CryptoPunks was estimated to be worth around 5 billion US dollars, which also sounds like something out of a dystopian SciFi, but is also true.

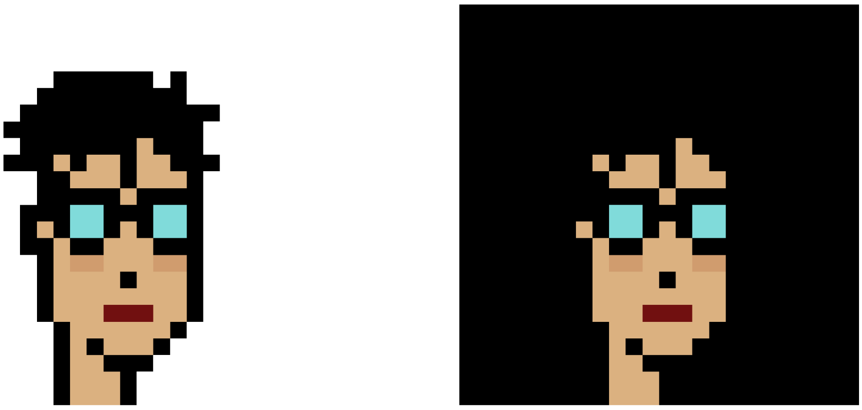


Figure 3: Left: Image with alpha channel and transparent pixels. Right: Image without alpha channel, pixels that were transparent are now black. Note that some of the opaque pixels merge with the background without opacity. In these images you can also clearly separate the pixels from each other, which is often impossible for higher resolution images.

Some of these features are rarer than others. For example, in the bottom row there is a ”Zombie” character, which is among the more rare types of characters.

Investigating figure 4, one quickly notices the figures have a lot of similarity, and of particular importance to us is that the figures are perfectly centered. This means that shared features appear at the exact same pixels for different images. For example, the hat of the most lower right character is shared by 7 other characters, and in each of the different images this hat appears at the exact same pixels, and also the exact same colors. This is crucial if one wants to use linear factorization techniques like NMF. If some characters were shifted, rotated or scaled, a method like NMF would completely fail. Effects like these cannot be accurately represented with linear combinations, and one would have to use non-linear methods like neural networks[8].

The fact that the features are centered and shared is also of great importance to us. This means that we should more or less be able to decompose each image into image components, so that each image can be described as a sum of these different image components. For example, a character can be described as the sum of a few facial features: ”image of a blue hat” + ”image of a dark-skinned, angular face” + ”image of red lipstick” + ”image of green eyes” +... Finding these different features can (hopefully) be done with NMF. An illustration of this (which was made using NMF) is shown in figure 5.

[8] And don't worry, almost all of STEM is converging towards neural networks, so you will learn about them at some point.

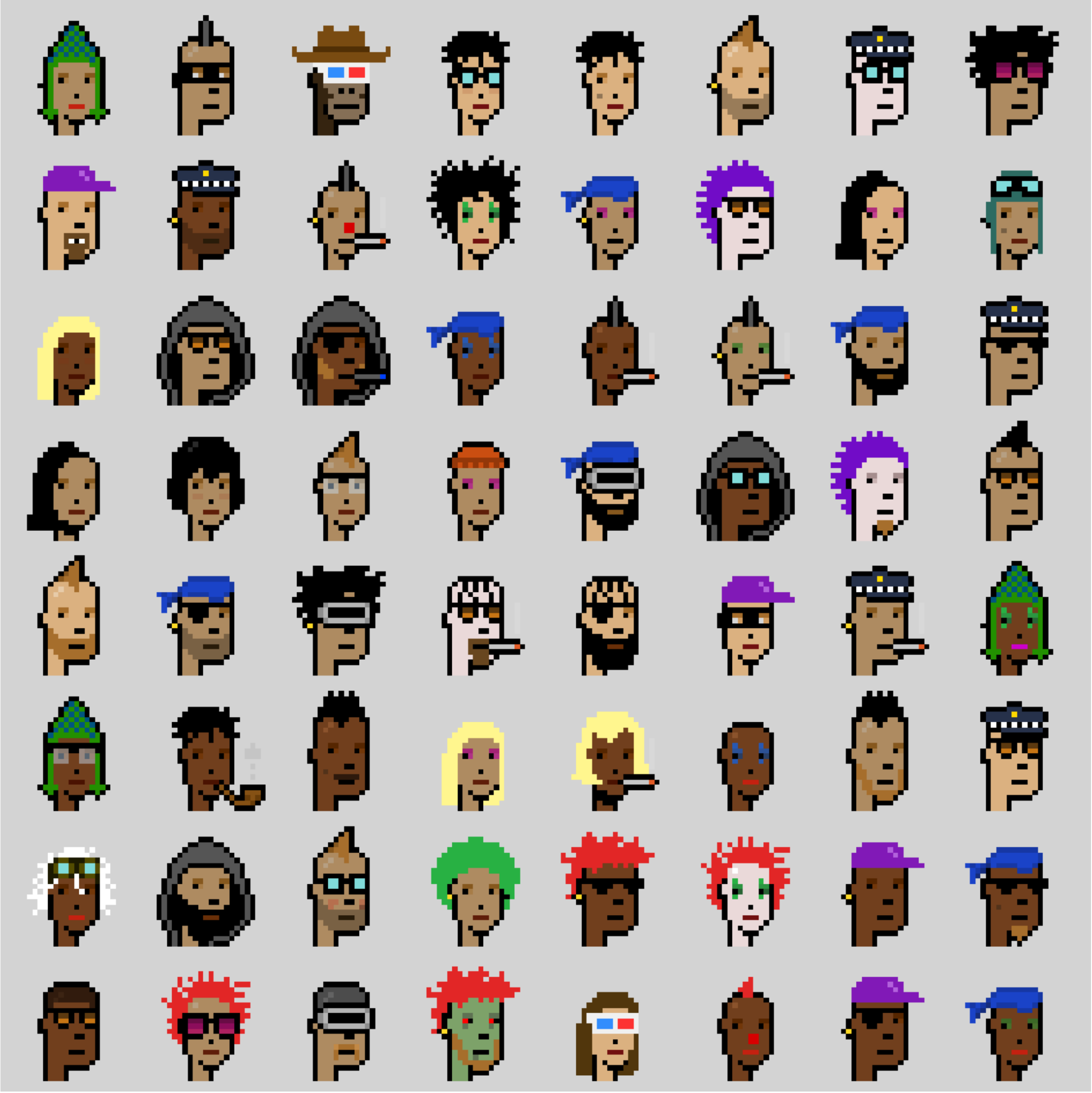


Figure 4: 64 different CryptoPunks plotted on a gray background. The gray areas are parts of the images with 0 opacity.

### 4.2 Our approach

There are two main concerns for applying NMF to images: reshaping the data matrix and the opacity. $N$ RGBA CryptoPunk images can be stored in a $24 \times 24 \times 4 \times N$ array, but NMF must be applied to an $m \times n$ array. Therefore, we have to reshape the array before applying NMF. When plotting and visualising we need to reshape the array back to suitable size.

Because the opacity is either 0 or 1 for every pixel, it does not really make sense to

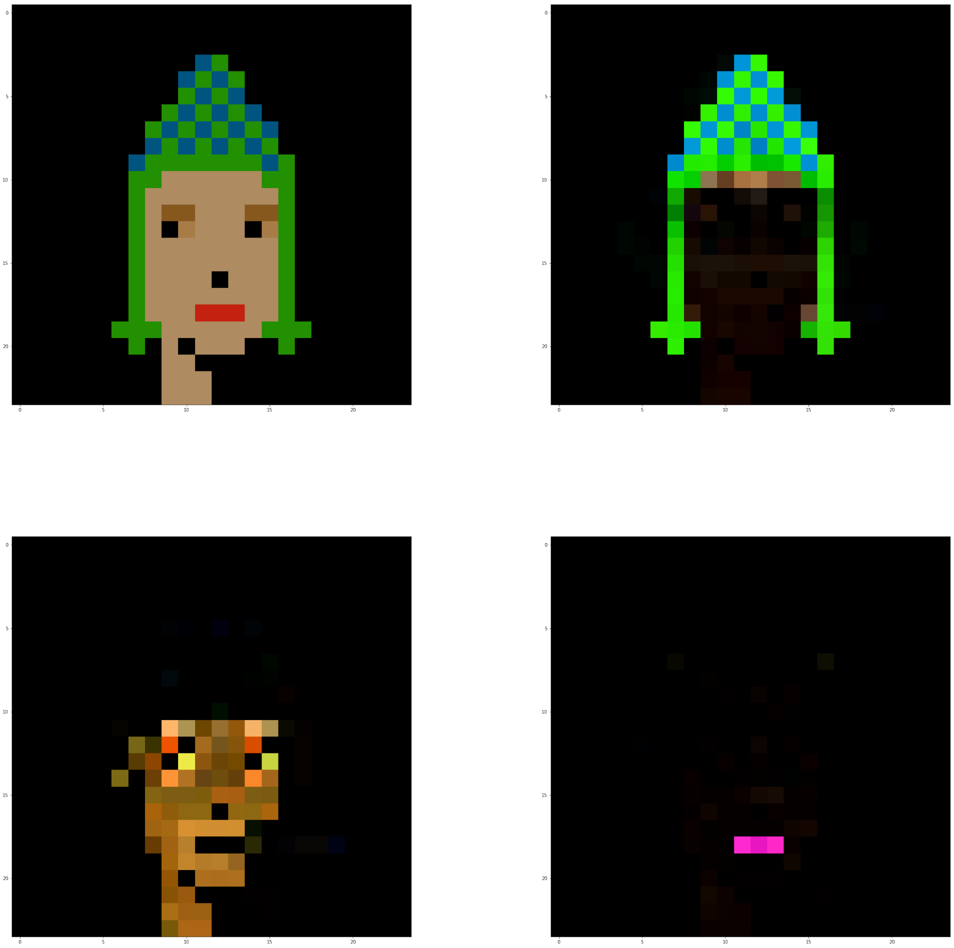


Figure 5: In the top left is a CryptoPunk character without opacity channel. The other images are three example image features (columns of $W$ obtained from applying NMF with $d = 64$, $N = 1000$) that can be added together to approximate the character, along with the rest of the basis vectors. In particular, here the "face" in the bottom left is a rather poor approximation of the face of the character, and more features are required.

learn it by matrix factorization, at least not in the form we have introduced it. Trying to learn the opacity also yields some other artifacts we want to avoid. Therefore, we only learn the color channels, and assume the opacity channel known. The approach is

1. Load images and store them in an $24 \times 24 \times 4 \times N$ array. A function that does this is given in the handed out code.

2. Store only the fourth channel (the opacity channel) in a $24 \times 24 \times N$ array.

3. Reshape the remaining three color channels into a $24 \cdot 24 \cdot 3 \times N = 1728 \times N$ array. This is the matrix $A$ whose NMF we want to calculate.

4. Fit the NMF to this matrix.

5. When plotting the reconstructions $WH$, we add the opacity channel we stored earlier. When plotting these images it is very possible that some pixel values will be larger than 1 (we assumed that images have pixel values between 0 and 1). In this case you will either have to clip the values to 1 or rescale the images by dividing each pixel value of an image by the maximal pixel value of that image. Rescaling yields a truer presentation of the reconstruction, but also makes the image darker and possibly discolored. Therefore, it might be smartest to clip the images (`plt.imshow` does this automatically but throws an annoying warning, so you should probably do it yourself). Whatever you do should not make a large difference.

One drawback of this approach is that we "lose" the opacity in the columns of $W$, which should contain image components. This means that when we plot these columns, there will be no way of discerning black pixels from the background, similarly to what was shown in figure 3.

If you need to save your results to an image file, I strongly suggest using `plt.imshow` and `plt.figsave` as shown in the handed out code as opposed to the `plt.imsave` function. The reason is that `plt.imsave` adds so-called anti-aliasing, which does not really work well with pixelated images as they simply become blurry.

# 5 Task 2

You will now investigate the dataset and apply the NMF to discover the image components of the dataset. You are free to select how many images $N$ you want to use, but we recommend $N = 500$. More images means the approach is much more computationally intensive. Similarly, selecting $N$ smaller is much less computationally expensive, but can lead to the resulting NMF looking quite differently. In particular, if $N$ is chosen low, the columns of $W$ will carry less independent information. We have purposefully given you opportunity to experiment with different parameters, and a good report should contain at least a tiny amount of experimentation (but not so much that you do not have time to answer the rest of the tasks!).

- **a)** Investigate the dataset and make sure you are familiar with it. Plot 64 different images from the dataset on a $8 \times 8$ grid of subplots (See handed out code).

- **b)** Along each pixel and color channel calculate the mean and using for example `np.mean` (Hint: use `axis = -1`) and plot the resulting image. The mean should be a $24 \times 24$ image with 4 channels. Does this yield any useful information about the dataset? Are there any pixels that have no opacity for all images? What can you say about the rank of the matrix containing the images?

- **c)** Calculate the NMF of the 3 color channels as explained above using $d = 64$. Plot the columns of $W$ interpreted as RGB images (each columns should be a $24 \times 24 \times 3$ vector so you can use the handed out plotting function after reshaping). Explain and discuss what you see. Does the NMF capture the important features of the dataset? You can also try with different values for $d$ to see how this affects the results.

- **d)** With $d = 64$, calculate the 64 reconstructions $WH$ corresponding to the images you plotted in 2a) and plot the reconstructions (again, you can use the handed out plotting function after reshaping). Are the reconstructions overall good? How do the reconstructions deviate from the original images (if they deviate at all)? In particular, are all features of the images equally well reconstructed?

- **e)** For each iteration $k$, calculate $\|A - W_k H_k\|_F$ and plot it as a function of iterations similarily to what you did in Task 1, but now for $d = 16, 32, 64$. Are the results reasonable? Does it look like the algorithm has converged?

- **f)** For a wide range of $d$, for example $d = 16, 64, 128, 256$, do an NMF and calculate $\|A - WH\|_F$ [9]. Make sure you are using a high number of maximum iterations (at least 1000 iterations), and if your computing resources and time allows it, try to use more values for $d$. Plot this quantity as a function of $d$. Discuss the resulting plot. Specifically, how would you expect $\|A - WH\|_F$ to depend on $d$?

# 6 Denoising

We will now see some of the true power of NMF and dimensionality reduction methods. We can apply them to solve a variety of tasks, including denoising. If you are starting to feel overwhelmed, do not worry. The rest of the project should be very quick if you have done Task 2, as you can reuse code for all tasks.

## 6.1 Noise

Noise is unwanted modifications of a signal (like images), that occur for a variety of reasons. Noise is very common in all of signal (and particularly image) processing, and there exists many ways of attempting to remove it. We will consider additive Gaussian noise

$$A_{\text{noisy}} = A + \sigma E, \tag{18}$$

where $\sigma > 0$ is a scalar noise level and $E$ is a matrix with the same shape as $A$ where all components are assumed to be realizations of the standard normal distribution (mean

[9] It is worth noting here that the Frobenius norm is not neccessarily the best way to describe distance between images. For example, two face images where there is a slight skin color difference can have a very large difference in norm, but two face images that have very different eyes can have a small difference in norm. The Frobenius norm does not necessarily reflect what the human brain considers similar, only that the pixel values are close or far.

equal to 0 and variance equal to 1, usually denoted $\mathcal{N}(0, 1)$). This is a common noise model because most physical noise behaves similarly to Gaussian distributions, and can be assumed to be independent of the original image. Although modern smartphone cameras are relatively sophisticated, you will most likely see some Gaussian [10] noise when you take images, particularily if you take photos in dark environments (just try to hold your hand in front of your smartphone camera).

Similarly to task 2, we will assume the opacity channel is known, and that only the color channels have noise on them. Furthermore, we will only assume that there is noise on color channels that are non-zero in that pixel (this might seem arbitrary, but it avoids some difficulties). We also clip value to lie between 0 and 1 after adding noise, so that noisy images are still plottable images. An example is shown in figure 6.

This noise model is more complicated than the purely additive noise given in equation (18), but the great thing about dimensionality reduction methods like NMF is that it does not care about the specifics of the noise model. It just works.

## 6.2 Denoising with NMF

The idea of noise removal with NMF is quite simple: We simply fit an NMF to $A_{\text{noisy}}$, and the reconstruction $WH$ will then contain denoised reconstructions, and the quality of the reconstruction depends on $d$. The reason for this is that when we learn a lower dimensional representation of the data, the signal (i.e, the original images) will be more important than the noise. When we discussed the SVD, one of the central points was that not all singular vectors (or eigenvectors) are as important for reconstructing the data, and we can discard information that is essentially just noise. You can also interpret each column of $W$ to be a sort of mean of many noisy images with the same feature. As you know from statistics, taking the mean of many noisy (stochastic) variables leads to reduction in variance (noise).

As the model complexity of the NMF increases ($d$ increases), the model will learn more of the noise, i.e the columns of $W$ will contain more noise and the reconstructions $WH$ will be noisy as well. If $d$ is chosen low, the model will learn little noise, but the reconstructions will also be poor as we saw in Task 2, as we discard too much useful information. This is one of the central concepts of machine learning, and is called overfitting and underfitting. Overfitting is when a model essentially fits the noise and not the underlying structure of the data, while underfitting is when a model is unable to capture the complexities of the data. One of the main challenges of machine learning is selecting not only a model that is able to learn something useful from the data, but selecting a model complexity that avoids underfitting and overfitting.

Something that is also worth noting about applying NMF is that the approach is completely unsupervised as opposed to supervised, two other central concepts in machine learning. In supervised denoising, the denoising model is fitted with knowledge of the noiseless data, while in unsupervised learning, the model is fitted only with the noisy

[10] Technically, the noise that occurs when taking images, so called "shot noise", can be modelled as Poisson distributed (light can be modelled as discrete photons, so a discrete probability distribution is required!), which is essentially equivalent to Gaussian distributed for large numbers.

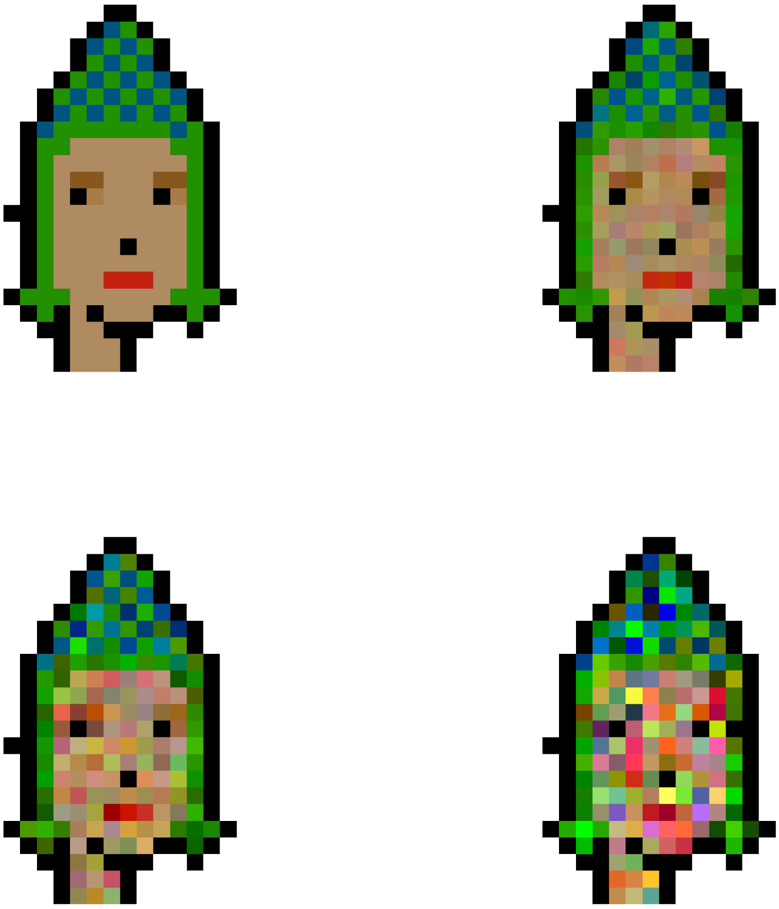


Figure 6: In the top left is an original image. Moving from left to right, top to bottom, the rest of the images are noisy images with $\sigma = 0.05, 0.1, 0.2$ respectively. Note that with our noise model, the black pixels are noiseless, and pixels where some of the RGB colors are 0 only have noise on the non-zero colors.

data. While supervised methods are usually much more effective (there exists a bunch of AI denoisers, upscalers, object recognition and compression algorithms online that are created using supervised methods), they also require a lot of data, and are much more prone to overfitting. Unsupervised denoising methods can be used when original noiseless images are not available. In this project we will simulate noise, and use the noiseless images in order to assess the effectiveness of our method.

## 7 Task 3

Throughout this task, we suggest you choose the noise level relatively high, for example $\sigma = 0.1$ for $N = 500$. You will most likely get much better denoising results for larger $N$, but this can be computationally prohibitive.

- **a)** Add noise to the images (see handed out code), and plot 64 images like you did in task 2a). Calculate $\|A_{\text{noisy}} - A\|_F$, where $A_{\text{noisy}}$ is the matrix containing noisy images and $A$ is the matrix containing the original images.

- **b)** Using the same approach as task 2, fit an NMF using $d = 64$ to the noisy images. Plot the columns of $W$ and the reconstructions like you did in 2c) and 2d). Are the reconstructions less noisy than the noisy images? Compare the columns of $W$ and the reconstructions to the results you got in task 2.

- **c)** As you did in task 2, fit an NMF to the noisy images for a wide range of $d$, calculate the reconstruction error $\|A - WH\|_F$ (important: $A$ is here the original images, NOT the noisy images). Plot this with the error you calculated in 2f), where noiseless images were used. If you have a large enough range of $d$ and large enough noise you should see that the reconstruction error for the noisy images is somewhere between "U"-shaped and "L"-shaped, and at some point the error should increase for large enough $d$. Explain why the plot looks like this (try to include the words "underfitting" and "overfitting"). Approximately, what value of $d$ is the "best fit", that provides the lowest reconstruction error?

- **d)** Include a suitable conclusion to the entire project. What did you find? What are the advantages and disadvantages of applying NMF for images, in particular for denoising?

## References

[1] D. Lee and H. S. Seung, "Algorithms for non-negative matrix factorization," in Advances in neural information processing systems, Vol. 13, edited by T. Leen, T. Dietterich, and V. Tresp (2001).

[2] J. Kim and H. Park, "Toward faster nonnegative matrix factorization: a new algorithm and comparisons," in 2008 eighth ieee international conference on data mining (IEEE, 2008), pp. 353–362.