VU Data Science in Astrophysics (Summer 2024)
Chapter 3: Data Representation

Oliver Hahn*

April 10, 2024

*oliver.hahn@univie.ac.at

1 Data Representation

1.1 Vector spaces and bases

We shall repeat here a few very basic concepts from your linear algebra class. Some you may have forgotten, some you may have not seen in exactly the way we shall need them, and maybe you have learned a different notation or terminology.

We shall consider here the case where our *sample space* is a *vector space*, which means that we can do linear algebra on it.

Vector space. A vector space $\mathcal{V}$ is a set with two operations: addition of two elements, and multiplication with a scalar, i.e. for elements $\boldsymbol{x}, \boldsymbol{y} \in \mathcal{V}$ and $\lambda, \mu \in \mathbb{R}$, we require that the following relations hold

$$\begin{aligned}
&\text{closure} && (\lambda\,\boldsymbol{x}) \in \mathcal{V} \quad \text{and} \quad (\boldsymbol{x}+\boldsymbol{y}) \in \mathcal{V} && (1)\\
&\text{commutativity} && \boldsymbol{x}+\boldsymbol{y} = \boldsymbol{y}+\boldsymbol{x} && (2)\\
&\text{distributivity} && \lambda(\boldsymbol{x}+\boldsymbol{y}) = \lambda\boldsymbol{x}+\lambda\boldsymbol{y} && (3)\\
&\text{associativity} && \lambda(\mu\boldsymbol{x}) = \mu(\lambda\boldsymbol{x}) = (\mu\lambda)\boldsymbol{x} && (4)\\
&\text{existence of neutrals } \mathbf{0} \text{ and } 1 && \boldsymbol{x}+\mathbf{0} = \boldsymbol{x} \qquad 1\,\boldsymbol{x} = \boldsymbol{x} && (5)
\end{aligned}$$

Linear combination. Given a finite number N of vectors $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_N \in \mathcal{V}$ and scalars $\lambda_1, \ldots, \lambda_N \in \mathbb{R}$, every vector $\boldsymbol{y}$ of the form

$$\boldsymbol{y} = \lambda_1\boldsymbol{v}_1 + \cdots + \lambda_N\boldsymbol{v}_N = \sum_{i=1}^{N} \lambda_i \boldsymbol{v}_i \tag{6}$$

is called a linear combination of the vectors $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_N$.

Linear independence. Consider N vectors $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_N \in \mathcal{V}$. If the equation

$$\mathbf{0} = \sum_{i=1}^{N} \lambda_i \boldsymbol{v}_i \tag{7}$$

has a nontrivial solution, i.e. at least one $\lambda_i \neq 0$, then the vectors $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_N$ are called **linearly dependent**. In the opposite case, i.e. if $\lambda_1 = \cdots = \lambda_N = 0$ is the only solution, then the vectors are said to be **linearly independent**. Equivalently, in the case of linear dependence, there is at least one vector which can be written as a linear combination of one or more of the other vectors.

Span. Consider a set of N vectors $\mathcal{A} = \{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_N\} \subseteq \mathcal{V}$. If every vector $\boldsymbol{x} \in \mathcal{V}$ can be written as a linear combination of the vectors $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_N$, then $\mathcal{A}$ is called a generating set for $\mathcal{V}$, and we say that $\mathcal{A}$ spans the vector space. This is expressed as

$$\mathcal{V} = \text{span}[\mathcal{A}] = \text{span}[\boldsymbol{v}_1, \ldots, \boldsymbol{v}_N] \tag{8}$$

Basis and dimension. A linearly independent generating set $\mathcal{B} = \{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_N\}$ is called a *basis* of the vector space. A basis also has the minimum possible number of vectors necessary to span the vector space. The *dimension* of a vector space is given by the number of vectors in its basis, i.e. $\dim(\mathcal{V}) = N$.

Subspace. Any (true) subset of a basis $\mathcal{B}' \subset \mathcal{B}$ defines a subspace $\mathcal{V}'$ of lower dimension $\dim(\mathcal{V}') < \dim(\mathcal{V})$.

Coordinate notation. Let $\mathcal{B} = \{\boldsymbol{b}_1, \ldots, \boldsymbol{b}_n\}$ be a basis. The entries of a vector in coordinate notation directly correspond to the coefficients in the basis expansion

$$\begin{bmatrix} \alpha_1 \\ \vdots \\ \alpha_n \end{bmatrix}_{\mathcal{B}} = \alpha_1 \boldsymbol{b}_1 + \cdots + \alpha_n \boldsymbol{b}_n = \sum_{i=1}^{n} \alpha_i \boldsymbol{b}_i. \tag{9}$$

Example: Canonical, standard, or Cartesian basis. The canonical basis, standard basis, or Cartesian basis of $\mathbb{R}^n$ is given by the generating set

$$\mathcal{B} = \left\{ \underbrace{\begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \ldots, \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix}}_{n \text{ vectors}} \right\} =: \{\boldsymbol{e}_1, \boldsymbol{e}_2, \ldots, \boldsymbol{e}_n\} \tag{10}$$

Whenever we express vectors with respect to this basis, we omit the explicit mention of the basis in the coordinate notation.

1.2 Scalar (inner) products

The inner product of two vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ is defined as a symmetric bilinear positive definite mapping of two vectors to the real numbers

$$\langle \boldsymbol{x}, \boldsymbol{y} \rangle : \ \mathcal{V} \times \mathcal{V} \to \mathbb{R} \tag{11}$$

Symmetric means that $\langle \boldsymbol{x}, \boldsymbol{y} \rangle = \langle \boldsymbol{y}, \boldsymbol{x} \rangle$. Bilinear means that for $\alpha \in \mathbb{R}$ it has the properties

$$\langle \alpha \boldsymbol{x}, \boldsymbol{y} \rangle = \langle \boldsymbol{x}, \alpha \boldsymbol{y} \rangle = \alpha \langle \boldsymbol{x}, \boldsymbol{y} \rangle . \tag{12}$$

and positive definite means that it obeys

$$\langle \boldsymbol{x}, \boldsymbol{x} \rangle > 0 \qquad \text{if } \boldsymbol{x} \neq \boldsymbol{0}, \text{ and } \langle \boldsymbol{0}, \boldsymbol{0} \rangle = 0. \tag{13}$$

Dot product. The dot product is a special case of the inner product. For two vectors $\boldsymbol{x} = [x_1, \ldots, x_n]^\top, \boldsymbol{y} = [y_1, \ldots, y_n]^\top$ in the standard basis, it is defined as

$$\boldsymbol{x} \cdot \boldsymbol{y} = x_1 y_1 + \cdots + x_n y_n = \sum_{i=1}^{n} x_i y_i. \tag{14}$$

Intuitively, it represents the projection of $\boldsymbol{x}$ onto $\boldsymbol{y}$ (or vice versa).

Such operations are trivially implemented in NUMPY either through the `dot` function, or the `@` operator for inner products. Note that the `*` operator is an element-wise product (also called Hadamard product $\odot$)

```
import numpy as np
x = np.array( [1, 4, 0, 2] )
y = np.array( [0, 1, 0, -2] )
print( np.dot(x,y) )
print( x@y )
print( x*y )
```

Euclidean norm. The Euclidean norm of a vector $\boldsymbol{v}$ is defined as

$$\|\boldsymbol{v}\|^2 = \|\boldsymbol{v}\|_2^2 = \boldsymbol{v} \cdot \boldsymbol{v}. \tag{15}$$

Orthogonal vectors and orthonormal bases. Two vectors $\boldsymbol{x}, \boldsymbol{y} \in \mathcal{V}$ are called orthogonal, if

$$\boldsymbol{x} \cdot \boldsymbol{y} = 0. \tag{16}$$

A basis $\mathcal{B} = \{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n\}$ is called *orthonormal* if

$$\boldsymbol{v}_i \cdot \boldsymbol{v}_j = \delta_{ij} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases} \tag{17}$$

1.3 Basis change

An extremely important concept is the concept of basis change. Consider two bases $\mathcal{B}_1 = \{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n\}$ and $\mathcal{B}_2 = \{\boldsymbol{w}_1, \ldots, \boldsymbol{w}_n\}$ of $\mathcal{V}$. We want to represent a vector

$$\boldsymbol{x} = [\alpha_1, \alpha_2, \ldots]_{\mathcal{B}_1}^\top = \alpha_1 \boldsymbol{v}_1 + \cdots + \alpha_n \boldsymbol{v}_n \tag{18}$$

in terms of the new basis $\mathcal{B}$, i.e. as

$$\boldsymbol{x} = [\beta_1, \beta_2, \ldots]_{\mathcal{B}_2}^\top = \beta_1 \boldsymbol{w}_1 + \cdots + \beta_n \boldsymbol{w}_n \tag{19}$$

Let

$$\mathrm{V} = \begin{bmatrix} | & | & & | \\ \boldsymbol{v}_1 & \boldsymbol{v}_2 & \ldots & \boldsymbol{v}_n \\ | & | & & | \end{bmatrix} \quad \text{and} \quad \mathrm{W} = \begin{bmatrix} | & | & & | \\ \boldsymbol{w}_1 & \boldsymbol{w}_2 & \ldots & \boldsymbol{w}_n \\ | & | & & | \end{bmatrix} \tag{20}$$

Then

$$\boldsymbol{x} = \boldsymbol{x} \quad \Leftrightarrow \quad \mathrm{W} \begin{bmatrix} \beta_1 \\ \vdots \\ \beta_n \end{bmatrix} = \mathrm{V} \begin{bmatrix} \alpha_1 \\ \vdots \\ \alpha_n \end{bmatrix} \quad \Leftrightarrow \quad \begin{bmatrix} \beta_1 \\ \vdots \\ \beta_n \end{bmatrix} = \mathrm{W}^{-1}\mathrm{V} \begin{bmatrix} \alpha_1 \\ \vdots \\ \alpha_n \end{bmatrix} \tag{21}$$

If the original basis is the Cartesian basis, then $V = E$ is the unit matrix (see below), and the basis change is simply given by a multiplication with the inverse basis matrix.

In PYTHON we can compute the basis change matrix $\mathrm{W}^{-1}\mathrm{V}$ conveniently. Let `v1`, `v2`, ... be the basis vectors of basis 1 (represented in terms of Cartesian coordinates `np.array([...])`), and `w1`, `w2`, ... those of basis 2, then

```
# assemble basis vectors in matrix form
V = np.array([ v1, v2, ...]) # V[0,:] = v1, V[1,:] = v2, ...
W = np.array([ w1, w2, ...]) # W[0,:] = w1, W[1,:] = w2, ...
# compute the basis change matrix
A = np.linalg.inv(W) @ V
# transform a vector in the v basis into w basis
w_vec = A.dot( v_vec )
```

Rank of a matrix. Given a column matrix of vectors $\mathrm{V} = [\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n]$. The rank r of the matrix is defined as the number of linearly independent vectors $\boldsymbol{v}_i$, i.e.

$$r = \mathrm{rank}(\mathrm{V}) = \dim(\mathrm{span}(\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n)). \tag{22}$$

As a corollary we have that the vectors $\boldsymbol{v}_i$ are linearly independent if $r = n$. Note also that $\mathrm{rank}(V) = \mathrm{rank}(V^\top)$.

1.4 Basis representations

1.4.1 Cartesian basis for nD data sets

The canonical or Cartesian basis of $\mathbb{R}^n$ is given by the generating set $\mathcal{B}_n = \{\boldsymbol{e}_1, \boldsymbol{e}_2, \ldots, \boldsymbol{e}_n\}$ Its matrix form is of course

$$\mathrm{E}_n = \left[\begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}, \ldots, \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix} \right] = \begin{bmatrix} 1 & 0 & \ldots & 0 \\ 0 & 1 & & 0 \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \ldots & 1 \end{bmatrix} \tag{23}$$

In pictorial form, we can think of them as the set depicted in the left panel (a) of Fig. 1. An image described in the Cartesian basis is therefore described by its individual pixel values. It is trivial to verify that this basis is orthonormal. A clear downside of the Cartesian basis is that all pixels are independent. In any meaningful image they won't be.

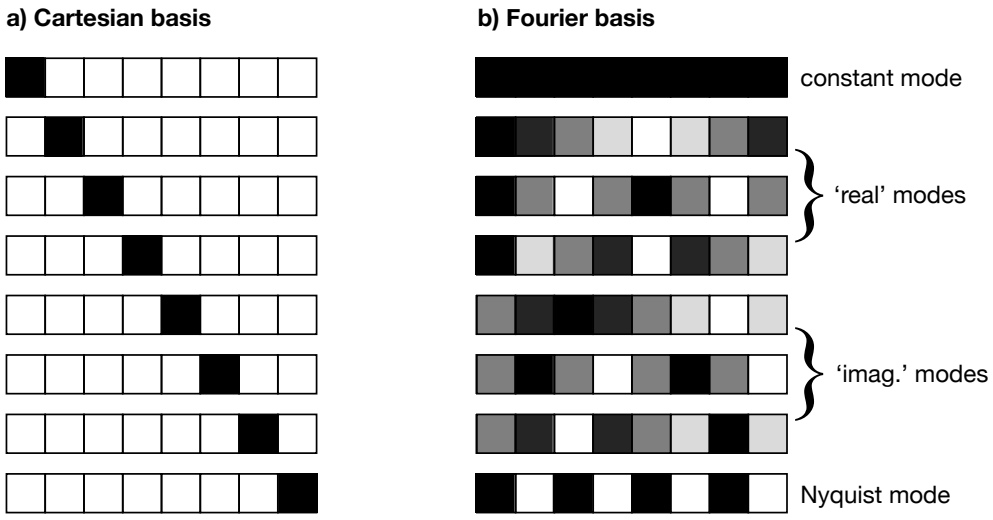

Figure 1: Two possible bases for an 8 element vector seen as a one-dimensional image: (a) the Cartesian basis, where each basis vector represents one single pixel, and (b) the Fourier basis, where the basis vectors correspond to the distinct Fourier modes representable by an 8 element vector. Changing the coefficient associated with a given Fourier basis vector affects multiple pixels.

Tensor product basis. We can build represent higher dimensional data using a tensor basis, by using the tensor product $\mathrm{E}_{ij} = \boldsymbol{e}_i \otimes \boldsymbol{e}_j \in \mathcal{B}_n \times \mathcal{B}_m$ with $\boldsymbol{e}_i \in \mathcal{B}_n$ and $\boldsymbol{e}_j \in \mathcal{B}_m$ to form a new basis. With this we can e.g. represent all two-dimensional datasets on $\mathbb{R}^{n\times m}$, which contains e.g. images with $n \times m$ pixels.

Data flattening. Sometimes it is however not necessary to resort to a tensor basis, instead it is more convenient to always work with one-dimensional data. In this case, we use the *flattening operation*, which simply maps $\mathbb{R}^{n\times m}$ with basis $\mathcal{B}_n \times \mathcal{B}_m$ to $\mathbb{R}^{nm}$ with basis $\mathcal{B}_{n\cdot m}$. In NUMPY this is very easily achieved by calling the `flatten` member function of an n-dimensional array.

1.4.2 The Fourier Basis

A very convenient basis in many cases is the Fourier basis, which decomposes a signal into its Fourier series. Let us consider the n-dimensional case of $\mathbb{R}^n$, then the Fourier basis is given by the two sets of vectors

$$\boldsymbol{c}_j = [1, \cos(2j\pi 1/n), \cos(2j\pi 2/n), \ldots, \cos(2j\pi(n-1)/n)]^\top \quad j = 0, \ldots, n/2-1 \tag{24a}$$

$$\boldsymbol{s}_j = [0, \sin(2j\pi 1/n), \sin(2j\pi 2/n), \ldots, \sin(2j\pi(n-1)/n)]^\top \quad j = 1, \ldots, n/2 \tag{24b}$$

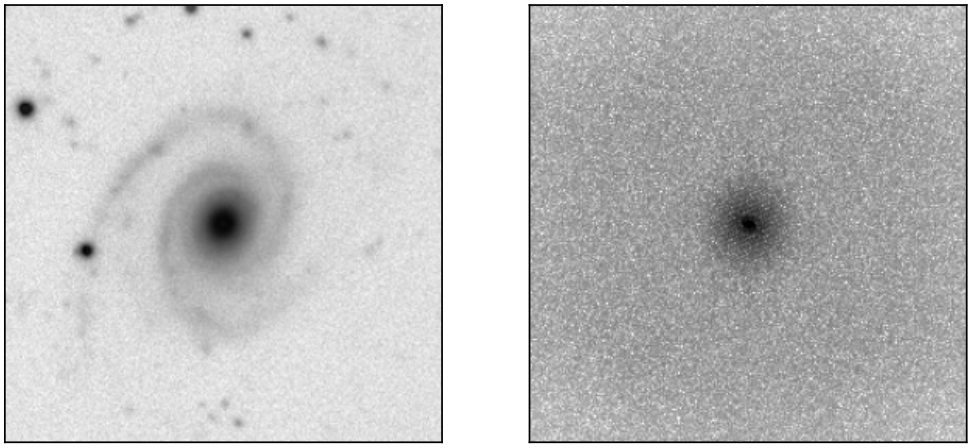

Figure 2: The gray-scale image of a galaxy in a Cartesian basis (left, $E_{256} \times E_{256}$), and in the Fourier basis (right), where the constant mode is located in the very center of the image, and we show the modulus of the complex coefficient.

where in the DFT the two are combined to a single complex vector

$$\hat{\boldsymbol{f}}_j = \boldsymbol{c}_j - \mathrm{i}\ \boldsymbol{s}_j \tag{24c}$$

with the basis matrix representation

$$\mathrm{F}_{kl} = \exp[-\mathrm{i}\, 2\pi kl/n] \tag{24d}$$

They are depicted in the right panel (b) in Figure 1, we have re-arranged them compared to the NUMPY implementation in the image. The 'real' modes correspond to the cosine modes, the 'imag.' modes to the sine modes, note that in a usual DFT (discrete Fourier transform), these are combined to a single complex number. The Nyquist mode appears in both the real and imaginary modes in principle, for a linear independent set, it can of course only appear once. The equivalent of the constant mode in the 'image' modes vanishes identically.

In contrast to the Cartesian basis, different pixels are now correlated. Can we show that this basis is nonetheless an orthonormal basis?

While we can use the basis conversion formula from above to convert a data set to the Fourier basis, this is much more efficiently implemented in the *Fast Fourier Transform* algorithm. In Figure 2, we can see the representation of a grayscale image of a galaxy with 256×256 pixels in the Cartesian basis (left), and in the Fourier basis (right). Assuming the image has been read and is stored as a 256×256 NUMPY array, the right image can be produced with the code

```
14    # assume galaxy_image contains the image
15    fgalaxy_image = np.fft.fft2( galaxy_image )
16    plt.imshow( np.fft.fftshift( np.abs( fgalaxy_image ) ) )
```

Note that `fftshift` is needed to position the constant mode in the center of the image, rather than in the corner of the image, where it would be located for the standard ordering of the DFT matrix (24d). Note that nothing has been lost or gained at this stage, the two representations of the image are equivalent, simply with respect to a different basis.

1.4.3 Dimensionality Reduction with the Fourier basis

Dimensionality reduction is the projection of a data set onto a subspace of the full vector space. If the sub space is determined by a smaller basis, the dimensionality of the data set in the subspace is trivially reduced (essentially one just discards the coefficients associated with the omitted basis vectors). Dimensionality reduction is therefore a form of *data compression*. It is obvious however that in the Cartesian basis, not much is gained. If we consider an image, then simply pixels are discarded.

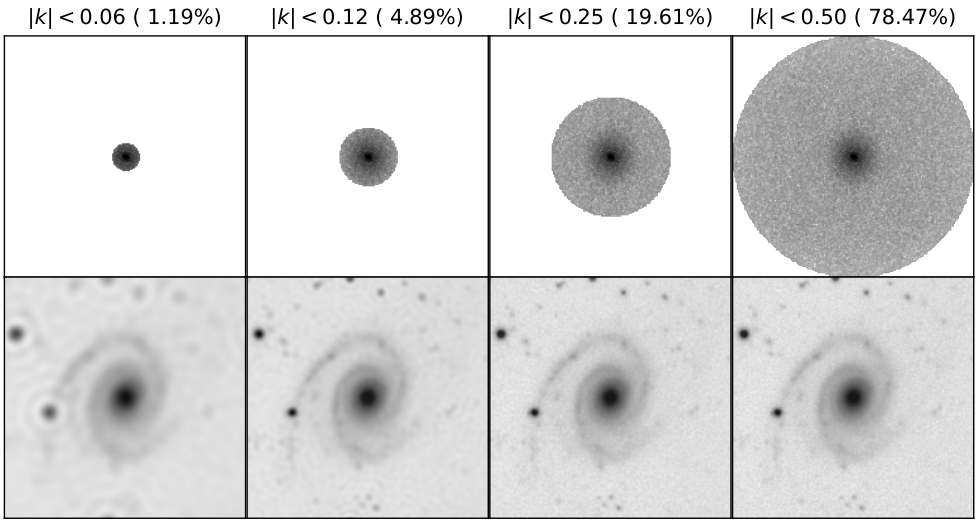

Figure 3: The reconstructed galaxy image from Fig. 2 using only a subset of the Fourier modes (corresponding to a projection on the low-frequency subspace).

In the Fourier basis the situation is dramatically changed. Let us retain only coefficients corresponding to low frequencies. Looking closely at Figure 2, we see that outside the center, the image is dominated by noise and no clear structure is visible. Using only the low-frequency modes, we can obtain a compressed representation of the image. This is shown in Figure 3. We see clearly that even with less than 1% of the coefficients, the galaxy spiral is visible. Due to the hard thresholding on coefficients (corresponding to a

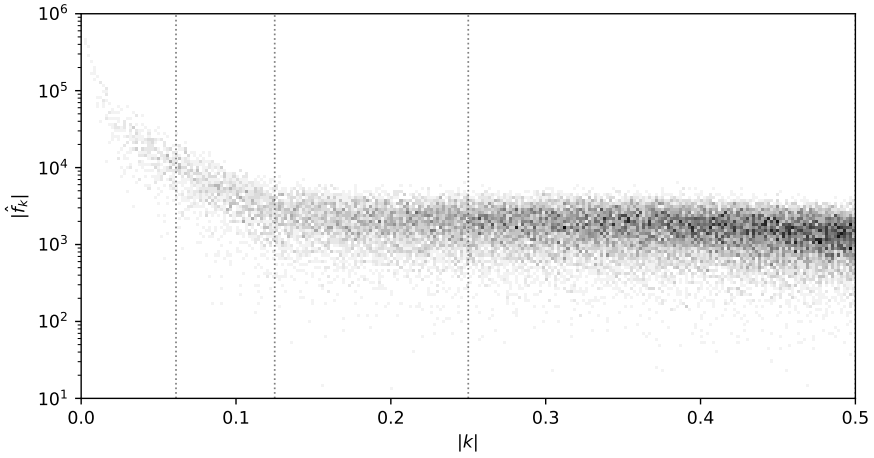

Figure 4: The amplitude of Fourier coefficients as a function of the wave number ($k = 0$ corresponding to the constant mode, $k = 1/2$ to the Nyquist mode). The flattening at larger $|k| \gtrsim 0.15$ is indicative of a dominant white noise component. The vertical dashed lines indicate the maximum modes included in the compression shown in Figure 2.

'sharp-k' low-pass filter), some ringing artefacts are visible at edges. What is important to note that the case of $\sim$5% coefficients closely cuts out the region where cooefficients are higher than the noise level. Looking closely, one sees that all features in the less compressed images are already represented. The amplitude of the Fourier coefficients as a function of the wave number ($k = 0$ corresponding to the constant mode, $k = 1/2$ to the Nyquist mode) is shown in Figure 4.

1.4.4 The Discrete Wavelet Transform

One massive shortcoming of the Fourier basis is that it is not localised. The Cartesian basis on the other hand was too localised (it did not include any non-local information). The middle ground between these two extremes is covered by *wavelets*. Wavelets, as the name suggests, are localised wave packets. Discrete wavelets are particularly useful as they form an orthonormal basis just like the Cartesian and the Fourier basis. They have a peculiarity over these other two bases however: they come in pairs (scaling functions and wavelets) and split a given signal into two distinct sets of coefficients, those representing 'coarse information' (or 'average') and those representing 'fine information' (or 'detail').

Some wavelets are shown in Fig. 5. The Haar wavelet is easiest to understand. It consists of two generators, the scaling function $\phi = [1,\, 1]$ and the wavelet $\psi = [1,\, -1]$. Computing the coefficients with respect to this wavelet basis now corresponds in 'convolving' these two stencils with the signal to obtain the average A and the detail D at level $\ell = 1$.

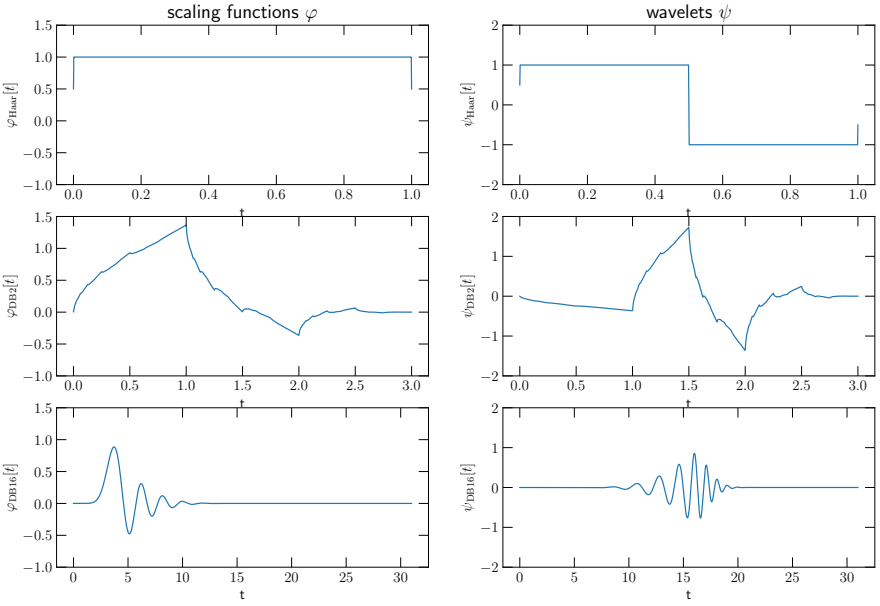

Figure 5: Some members from the family of discrete wavelets (Specifically the Haar and Daubechies wavelets of order 2 and 16). They form an orthonormal basis and for increasing order, they approach a smooth windowed Fourier transform. Each pair is mutually orthogonal. The scaling function projects the signal onto an 'average' space, the wavelet into the orthogonal 'detail' space.

Given an input signal of even length n, i.e. $\boldsymbol{f} \in \mathbb{R}^n$, $\boldsymbol{f} = [f_1, \ldots, f_n]^\top$, we have

$$A_j^{\ell=1} = \langle f, \phi \rangle_j = f_{2j-1} + f_{2j}, \qquad j = 1, \ldots, n/2 \tag{25}$$

$$D_j^{\ell=1} = \langle f, \psi \rangle_j = -f_{2j-1} + f_{2j}, \qquad j = 1, \ldots, n/2 \tag{26}$$

This process can be repeated for A^ℓ, i.e. we can continue to apply the separation into average and detail to obtain $A^{\ell+1}$ and $D^{\ell+1}$. The procedure can be extended by tensor products of the basis to higher dimensions than one. This hierarchical cascading procedure is shown in Figure 6. The discrete wavelet transform can be computed in Python using the `pywt` package. Given an image, the decomposition can be obtained using the `pywt.dwt` function, the inverse transform is computed using the `pywt.idwt` function. The decomposition and reconstruction of an image is then achieved as follows

```
17  import pywt
18
19  cA, (cH, cV, cD) = pywt.dwt2( my_image, wavelet='db2' )
20
21  re_image = pywt.idwt2( (cA, (cH, cV, cD)), wavelet='db2' )
```

In the reconstruction step, of course coefficients can be zeroed in order to achieve a

subspace projection. The higher order wavelets (beyond the Haar wavelet) need to assume boundary conditions for the data vector, this can be controlled using the `mode` argument[1]

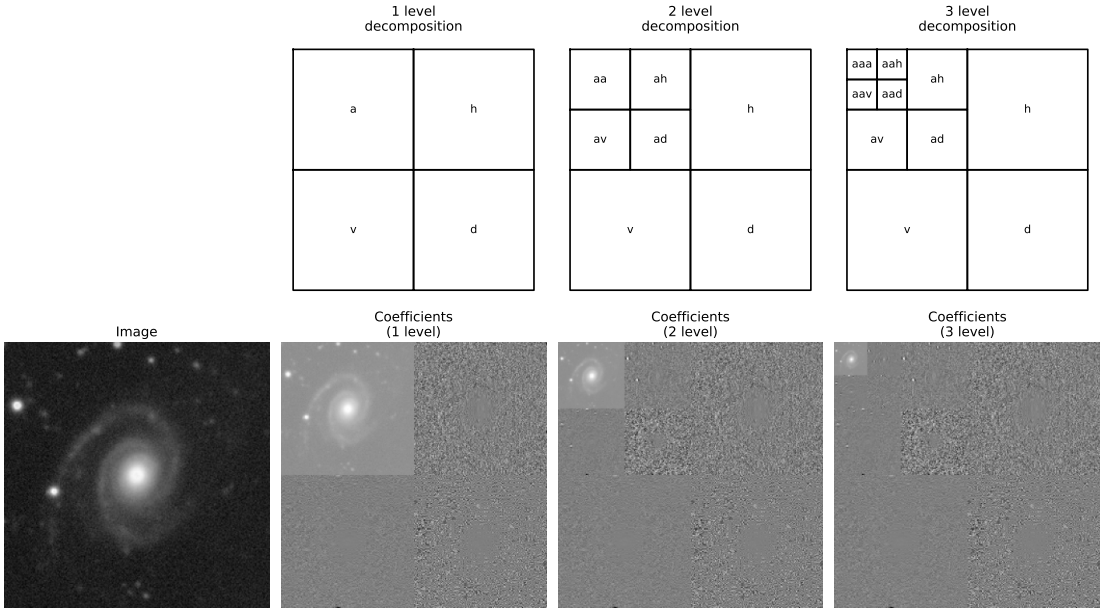

Figure 6: Hierarchical decomposition of a galaxy image using the discrete wavelet transform. At each level, the image is split in average 'A' and detail 'D' per dimension of the image. The tensor product yields four distinct basis sets $a = A \otimes A$, $d = D \otimes D$, $v = A \otimes D$ and $h = D \otimes A$. The average can once again be decomposed into four at the next higher level on so on.

Wavelets can be used for many purposes, e.g. can filtering be done now in a localised fashion rather than globally as with the Fourier basis. A particularly striking application is dimensionality reduction by keeping only the a^ℓ coeefficients of the ℓ-times projected averages. If the image is reconstructed in the inverse transform better compression can be achieved than from the Fourier basis. This is shown in Figure 7.

[1] `https://pywavelets.readthedocs.io/en/latest/ref/signal-extension-modes.html#ref-modes`

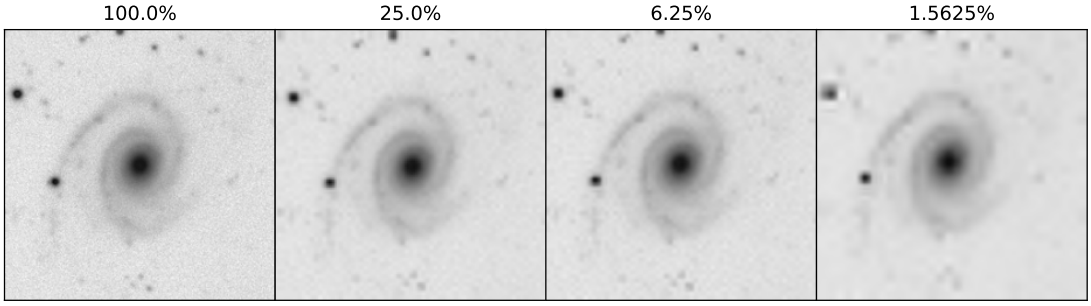

Figure 7: Image of the galaxy after projection on a low-dimensional subspace of the wavelet basis. The number of coefficients kept over the full image is indicated on top of each image. This form of dimensionality reduction is very similar to JPEG compression, as is also witnessable by the JPEG-like artefacts in the most compressed image. In this case, the 'db4' wavelet was used.

1.5 Data-Driven Basis I: Singular Value Decomposition

The basis systems we have encountered so far are **data agnostic**, i.e. they are defined independently of the data. For this reason they work reasonably well in all cases, but are rarely optimal for any given (sparse) data. A main algorithm to obtain a basis from data is the *Singular Value Decomposition* (SVD). The SVD is a data-driven basis, i.e. it is determined by the data itself. The SVD is a generalisation of the eigendecomposition of a matrix to non-square matrices.

The singular value decomposition (SVD) is a unique decomposition of a matrix $\mathrm{X} \in \mathbb{R}^{n \times m}$ so that

$$\mathrm{X} = \mathrm{U}\,\Sigma\,\mathrm{V}^\top \tag{27}$$

where

$$\mathrm{U} \in \mathbb{R}^{n \times n}, \qquad \mathrm{V} \in \mathbb{R}^{m \times m}, \quad \text{are unitary with orthonormal columns} \tag{28}$$

$$\Sigma \in \mathbb{R}^{n \times m} \quad \text{has positive entries on the diagonal and zeros otherwise.} \tag{29}$$

The n column vectors of U, i.e.

$$\mathrm{U} = \begin{bmatrix} | & | & & | \\ \boldsymbol{u}_1 & \boldsymbol{u}_2 & \dots & \boldsymbol{u}_n \\ | & | & & | \end{bmatrix} \qquad \text{with} \quad \boldsymbol{u}_i \in \mathbb{R}^n. \tag{30}$$

provide an orthonormal basis of the sample vector space. The shapes of the respective matrices are illustrated in Figure 8.

It is readily computed using the `numpy.linalg` Python package via

```
U, S, V = np.linalg.svd(X, full_matrices=False)
```

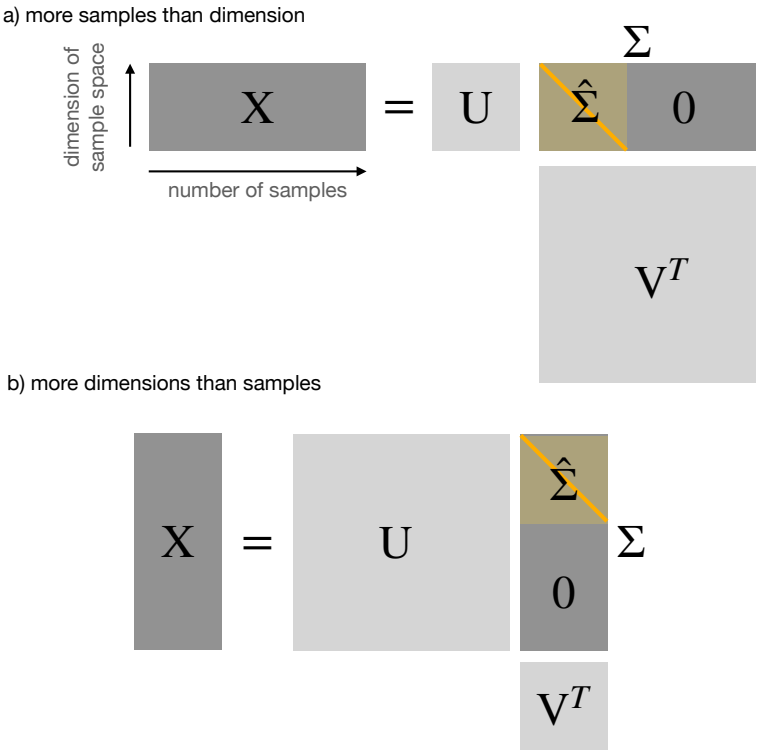

Figure 8: Illustration of the shapes of the matrices involved in the SVD.

SVD for PCA and regression. The SVD is essentially the same as PCA. In PCA, we compute the eigenvalues and eigenvectors of the data covariance matrix. This can also be achieved with SVD.

```
# let x be a vector of 1000 samples from a bivariate Gaussian distribution
# with covariance matrix [[2.0,-1.0],[-1.0,1.0]]

# compute the SVD of the mean subtracted data
xmean = np.mean(x,axis=1)
U,S,V = np.linalg.svd( x-xmean[:,np.newaxis], full_matrices=False )

t = np.linspace(-1,1,10)
# plot the data as scatter plot
plt.scatter( x[0,:], x[1,:], s=0.5)
# plot the right singular vectors
plt.plot( xmean[0] + t * U[0,0], xmean[1] + t * U[0,1] )
plt.plot( xmean[0] + t * U[1,0], xmean[1] + t * U[1,1] )
```

The results produced by this code snippet are shown in Figure 9. The SVD identifies the basis vectors which, in decreasing rank order, explain most of the variance of the data. The vectors can be used as a linear regression of the data.

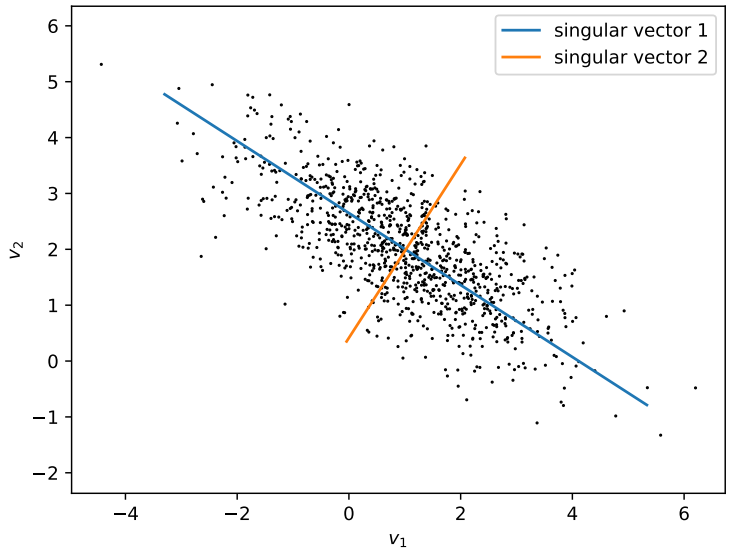

Figure 9: The SVD is essentially identical to principal component analysis (PCA) and identifies the vector space which, in decreasing rank order, explains most of the variance of the data. The vectors can be used as a linear regression of the data.

Rank truncation. A key property of the SVD is that it provides an optimal *low-rank approximation* to the matrix X. The rank r truncated matrix $\hat{\mathrm{X}}$ is obtained by zeroing all diagonal entries σ_i of $\Sigma = \mathrm{diag}(\sigma_1, \ldots, \sigma_n)$ with $i > r$, i.e. $\hat{\Sigma} = \mathrm{diag}(\sigma_1, \ldots, \sigma_r, 0, \ldots, 0)$ with

$$\hat{\mathrm{X}} = \mathrm{U}\hat{\Sigma}\mathrm{V}^\top. \tag{31}$$

As an example, we will compress a galaxy image using the SVD. The result is shown in Figure 10. The amplitude of the singular values as a function of their rank is shown in Figure 11.

Alternatively, we can think of the SVD representation as a linear combination of outer products of the singular vectors, i.e.

$$\hat{\mathrm{X}} = \sum_{i=1}^{r} \sigma_i \, \boldsymbol{u}_i \, \boldsymbol{v}_i^\top. \tag{32}$$

The outer product is a rank-1 matrix, and the sum of r rank-1 matrices is a rank-r matrix. Thinking of images, a rank-1 image has a very simple structure shown in Figure 12 for the first 16 outer products of singular vectors of the galaxy image shown in Figure 10.

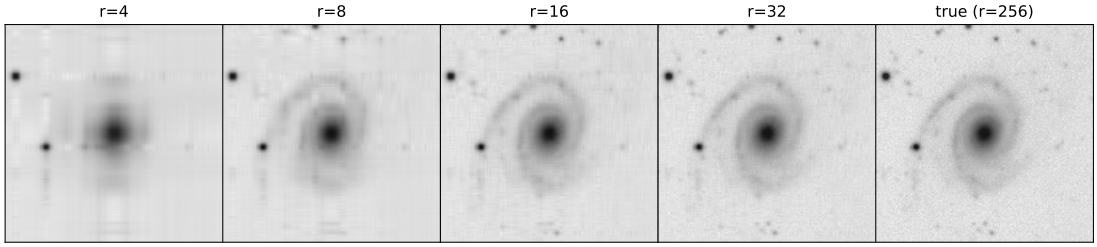

Figure 10: Rank-r SVD approximations of an individual galaxy image. The SVD low-rank approximation performs roughly as well as the other dimensionality reduction methods. The amplitude of the singular values as a function of their rank is shown in the next Figure 11.

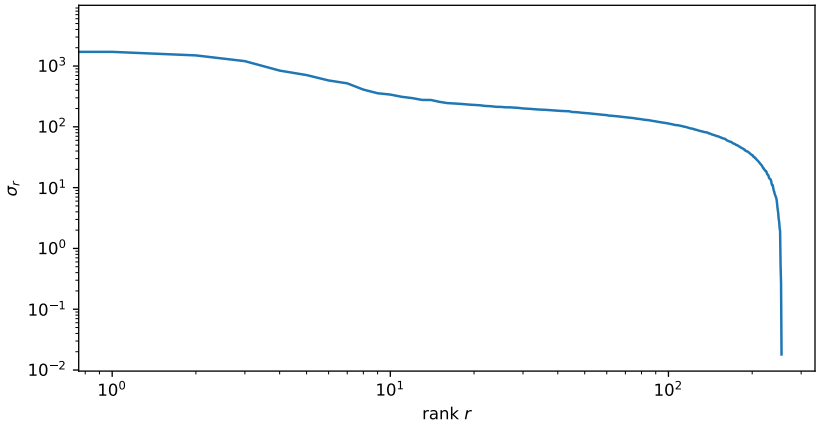

Figure 11: Magnitude of the singular values σ_r as a function of their rank r for the singular value decomposition of the galaxy image shown in Figure 10.

Theorem of Eckart-Young. The optimal rank-r least-squares approximation to a matrix X is given by the SVD rank truncated matrix $\hat{\mathrm{X}}$, i.e.

$$\underset{\hat{\mathrm{X}},\ \mathrm{s.t.rank}(\hat{\mathrm{X}})=r}{\arg\min} \quad \|\mathrm{X}-\hat{\mathrm{X}}\|_F = \mathrm{U}\hat{\Sigma}\mathrm{V}^\top, \tag{33}$$

where $\|\mathrm{X}\|_F = \sqrt{\sum_{ij} x_{ij}^2}$ is the Frobenius norm.

SVD of noise. An interesting result of SVD decompositions are the singular vectors of Brownian noise (i.e. we successively generate a data vector by adding a Gaussian random number to the previous cell, starting from zero). One finds that in this case, for a sufficiently large sample size, the SVD basis is simply the Fourier basis (see Figure 13). Hence, if there is not sufficient structure in the data, the SVD will yield a data-agnostic basis (in this case the Fourier basis).

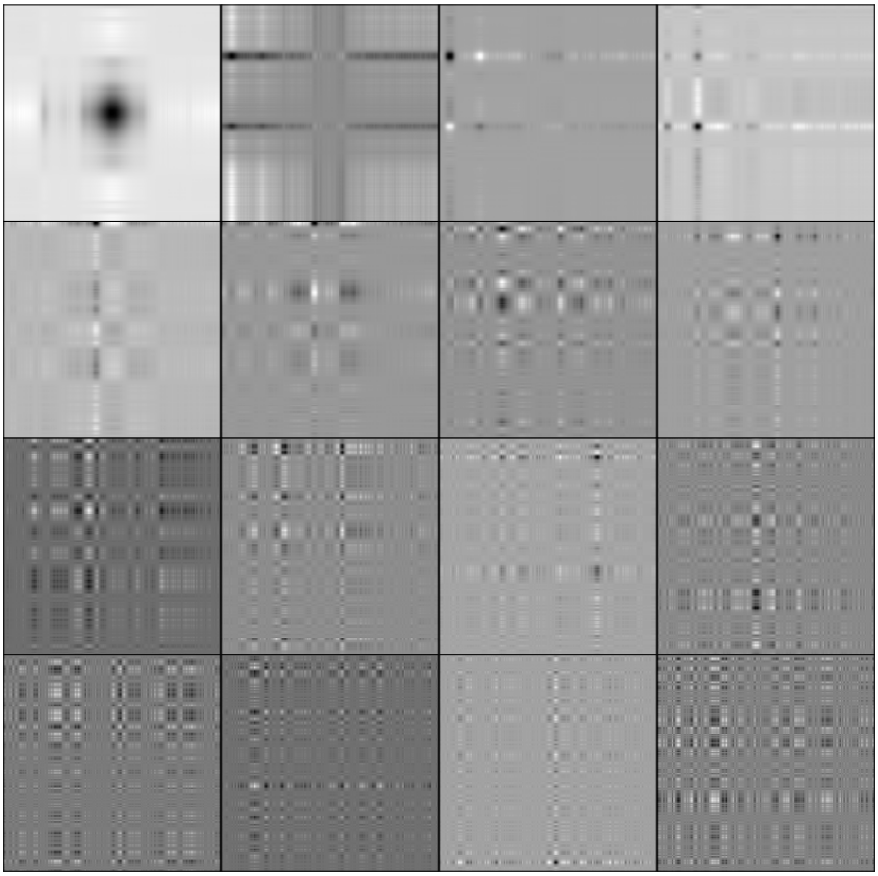

Figure 12: The outer products $\boldsymbol{u}_i \boldsymbol{v}_i^\top = \boldsymbol{u}_i \otimes \boldsymbol{v}_i$ of the singular vectors of the galaxy image shown in Figure 10. The outer products are rank-1 matrices, the sum of these 16 images (weighted by the singular values) is the $r = 16$ low-rank approximation of the galaxy image. The Eckart-Young theorem guarantees that this is the optimal rank-r approximation to the image (matrix).

1.6 Hard thresholding and the SVD

So far we have not given any guidance on how to choose the rank r of the SVD approximation other than the heuristic that it is meaningful to retain only those singular values whose amplitude exceeds the noise level. This can be formalised by using *hard thresholding*. The idea is to set all singular values σ_i to zero that are smaller than a certain threshold τ, i.e.

$$\hat{\sigma}_i = \begin{cases} \sigma_i & \text{if } \sigma_i > \tau \\ 0 & \text{otherwise} \end{cases}, \tag{34}$$

where the value of τ is determined by the amplitude of the noise.

It turns out that there is an optimal choice for the threshold τ under the assumption that

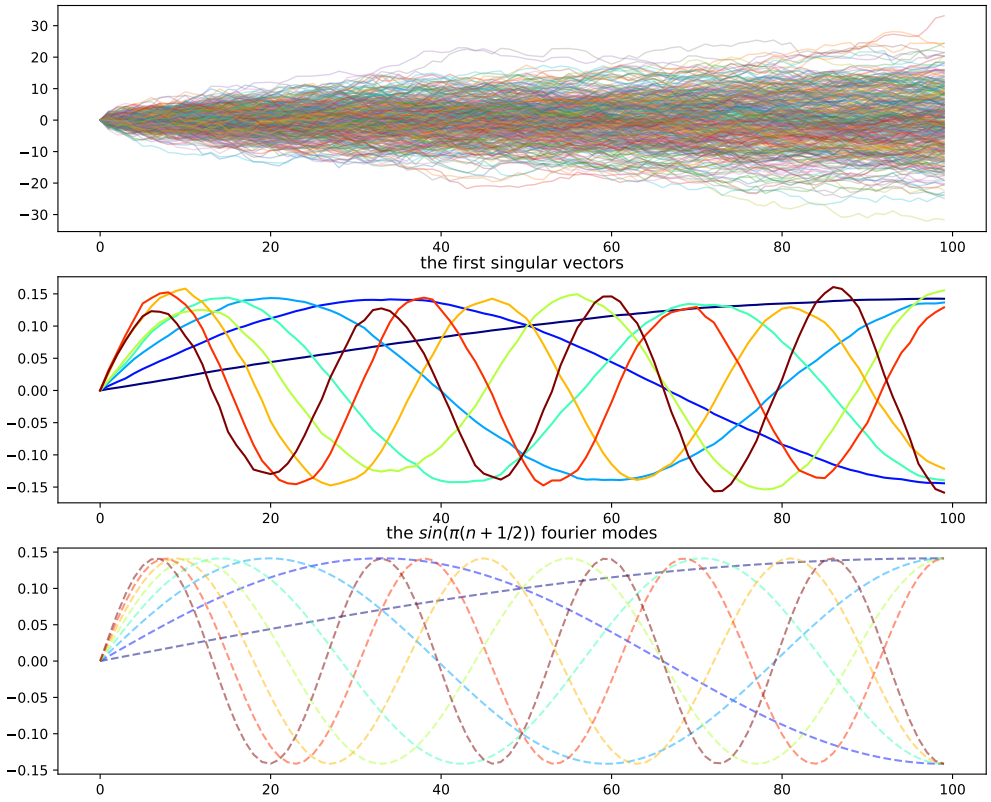

Figure 13: The singular value decomposition of Brownian walks yields the Fourier basis as the singular vectors.

a matrix has a low-rank structure contaminated with *white Gaussian noise*, i.e.

$$\mathrm{X} = \mathrm{L} + \gamma \mathrm{N} \tag{35}$$

where L is a low-rank matrix, N is a matrix of white Gaussian noise, and γ is a scaling factor. If $\mathrm{X} \in \mathbb{R}^{n \times m}$ is a square matrix, then the optimal threshold was shown by Gavish and Donoho [2014] to be given by

$$\tau = \frac{4}{\sqrt{3}} \sqrt{n} \gamma \ . \tag{36}$$

with a modified form for non-square matrices.

1.7 Robust PCA.

The SVD is a powerful tool for dimensionality reduction, but it is also sensitive to outliers. In the presence of outliers, the SVD can be severely affected. A robust version of the SVD is the *robust PCA*, proposed by Candès et al. 2011. The idea is to decompose

the data matrix X into a low-rank matrix L and a sparse matrix S, i.e.

$$\mathrm{X} = \mathrm{L} + \mathrm{S} \tag{37}$$

where L is structured low-rank and S is sparse and contains outliers and corrupt data. The goal is thus to find L and S such that

$$\min_{\mathrm{L,S}} \mathrm{rank}(\mathrm{L}) + \|\mathrm{S}\|_0 \quad \text{s.t.} \quad \mathrm{X} = \mathrm{L} + \mathrm{S} \tag{38}$$

where $\|\mathrm{S}\|_0$ is the ℓ_0 pseudo-norm, i.e. the number of non-zero elements in S. This is a combinatorial problem and is NP-hard. A relaxation is to use the ℓ_1 norm instead, i.e.

$$\min_{\mathrm{L,S}} \|\mathrm{L}\|_* + \lambda \|\mathrm{S}\|_1 \quad \text{s.t.} \quad \mathrm{X} = \mathrm{L} + \mathrm{S}, \tag{39}$$

where $\|\mathrm{L}\|_*$ is the nuclear norm of L, i.e. the sum of the singular values of L, which is a proxy for the rank of L. And $\|\mathrm{S}\|_1$ is the ℓ_1 norm of S. The parameter λ is a trade-off parameter between the low-rank and the sparse part. The minimisation can be achieved using the *alternating direction method of multipliers* (ADMM).

It can be implemented in PYTHON easily, we follow the implementation of Brunton and Kutz [2022]. We first define a `shrink` operator that subtracts τ from the entries of a matrix and sets all values with norm smaller than τ to zero:

```
36  def shrink(X, tau):
37    Y = np.maximum( np.abs(X) - tau, 0 )
38    return np.sign(X) * np.maximum(Y, 0)
```

Next one constructs the singular value thresholding operator:

```
39  def svd_threshold(X, tau):
40    U, S, V = np.linalg.svd(X, full_matrices=False)
41    return U @ np.diag( shrink(S, tau) ) @ V
```

Finally, the robust PCA algorithm is implemented as follows:

```
42  def robust_pca( X, tol=1e-7, maxit=1000 ):
43    n1, n2 = X.shape
44    mu = n1 * n2 / (4 * np.sum( np.abs( X[:] ) ))
45    lambd = 1 / np.sqrt( np.maximum(n1,n2) )
46    thresh = tol * np.linalg.norm(X)
47
48    S = np.zeros_like(X)
49    Y = np.zeros_like(X)
50    L = np.zeros_like(X)
51
52    iter = 0
53    while (np.linalg.norm(X-L-S) > thresh) and (iter < maxit):
54        L = svd_threshold( X - S + Y/mu, 1/mu )
```

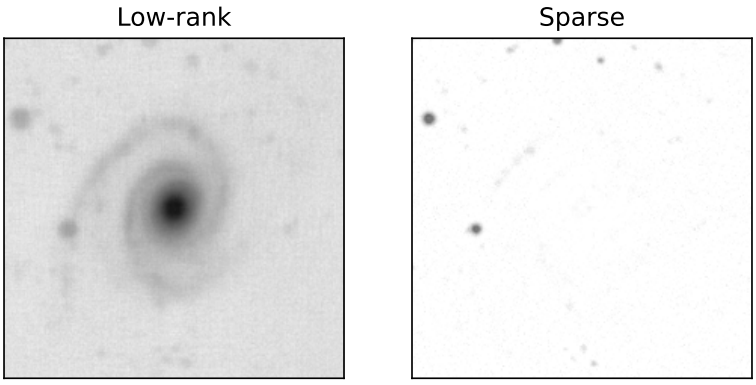

Figure 14: Robust PCA applied to an individual galaxy image. The low-rank part of the decomposition is shown in the left panel, the sparse part in the right panel.

```
55          S = shrink( X - L + Y/mu, lambd/mu )
56          Y = Y + mu * (X - L - S)
57          iter += 1
58
59      return L, S
```

1.8 Data-Driven Basis II: SVD as a linear autoencoder, autoencoders as non-linear SVDs

1.8.1 Linear autoencoders

It is interesting to note that the SVD can be written as a linear neural network. Specifically, the SVD can be thought of as a linear autoencoder. Remember that the rank-r truncated SVD is given by

$$\hat{\mathrm{X}}_{il} = \sum_{j,k} \mathrm{U}_{ij}\, \hat{\Sigma}_{jk}\, \mathrm{V}_{lk}. \tag{40}$$

if we write out the matrix multiplication explicitly.

In the context of machine learning, we call the diagonal matrix Σ the *latent space*. Since it is diagonal, we can think of the vector $\boldsymbol{\sigma} := (\sigma_1, \ldots, \sigma_r)^\top$ as the latent vector representing the object in the basis of singular vectors.

The SVD can be written as a neural network with a single dense layer with r nodes with a linear activation function. The encoder is given by the matrix multiplication with U, the decoder by the matrix multiplication with V. The network is trained to minimise the difference between input and output, i.e. the mean squared error. By the Eckart-Young theorem, we must recover the SVD in this setting.

We can implement a linear autoencoder in Keras as a neural network in a few lines of Python code. We can load the necessary Keras modules as follows

```
60  from tensorflow.keras import layers, models, optimizers
```

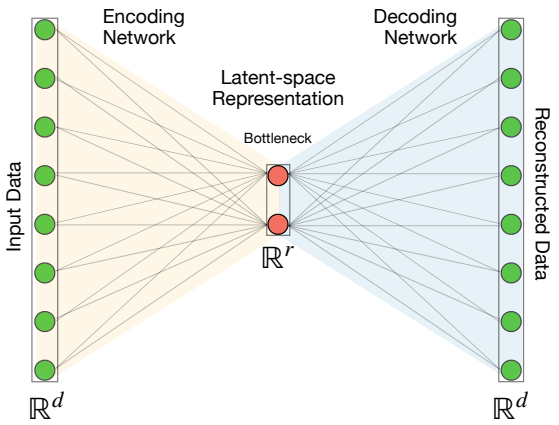

Figure 15: The architecture of a linear autoencoder – equivalent to SVD. The encoder maps the input data to a lower-dimensional representation in *latent space, the decoder maps this representation back to the original data thus reconstructing the signal from the lower-dimensional representation. The network is trained by minimising the mean squared difference between the input and the output.*

Assume we have a standardized bivariate data set X, i.e. the mean of each column is zero and the standard deviation is one. We write a linear encoder in Keras by defining an `Input` layer with two nodes (as we consider bivariate input) and a `Dense` layer with r nodes and a linear activation function:

```
61  # Define the encoder architecture
62  encoder_input  = layers.Input(shape=(2,), name='encoder_input')
63  encoder_output = layers.Dense(units=latent_dim, activation=None,
64          name='bottleneck')(encoder_input)  # Encoded to latent_dim latent space
65  linear_encoder = models.Model(encoder_input, encoder_output)
```

The decoder is defined by a `Dense` layer which has an input with r nodes and a dense layer with two nodes and a linear activation function, thus mapping back to the original dimensionality of the data:

```
66  # Define the decoder architecture
67  decoder_input = layers.Input(shape=(latent_dim,), name='decoder_input')
68  decoder_output = layers.Dense(units=2, activation=None,
69          name='decoder_output')(decoder_input)
70  linear_decoder = models.Model(decoder_input, decoder_output)
```

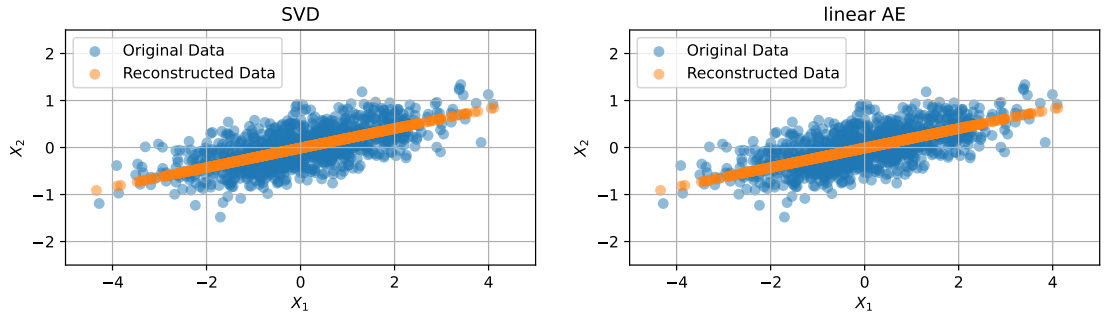

Figure 16: A linear autoencoder implemented as a neural network in Keras. The encoder consists of a single dense layer with a linear activation function, the decoder consists of a single dense layer with a linear activation function. The network is trained to minimize the difference between input and output, which, by the theorem of Eckart-Young implies that we recover the SVD.

The autoencoder is then defined as a KERAS `Model` with the encoder input and the decoder output:

```
71    linear_autoencoder = models.Model( encoder_input, linear_decoder(encoder_output) )
```

The structure of such a network is shown in Figure 15. The next step is to compile the model and train it. We can use the `adam` optimizer and the mean squared error as the loss function:

```
72    # Compile and train  the model
73  linear_autoencoder.compile(optimizer='adam', loss='mean_squared_error')
74  linear_autoencoder.fit(data, data, epochs=400, batch_size=256, shuffle=True)
```

We can use this model now for dimensional reduction by evaluating it:

```
75    # encode the data, keeping only a one-dimensional latent space
76    predictions = linear_autoencoder.predict(data)
```

The result is shown in Figure 16 in comparison to a rank-r SVD approximation. The results are identical, as expected.

1.8.2 Non-linear autoencoders

Clearly, if we have a non-linear correlation in the data, a linear autoencoder is not sufficient. In this case, we need a non-linear autoencoder. A non-linear autoencoder is a neural network that is trained to reproduce its input at its output, but with non-linear activation functions. The network is typically composed of an *encoder and a* decoder. In

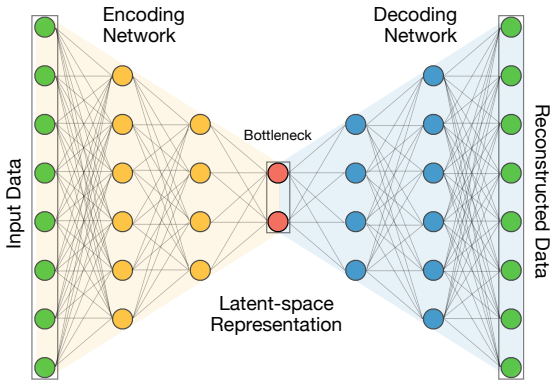

Figure 17: The architecture of a deep autoencoder. The encoder maps the input data to a lower-dimensional representation in *latent space, the decoder maps this representation back to the original data thus reconstructing the signal from the lower-dimensional representation. The network contains one or more hidden layers and non-linear activation functions. Again, the autoencoder network is trained by minimising the difference between the input and the output.*

order to make the network more expressive on can add additional hidden layers, yielding a structure such as shown in Figure 17 – a *deep autoencoder*.

We can realise a non-linear encoder by adding a `Dense` layer with a non-linear activation function, e.g. a `ReLU` or a `tanh` activation function. The following code snippet shows how to implement a deep autoencoder in KERAS:

```
77    # Define the encoder architecture:
78    # input layer
79    encoder_input  = layers.Input(shape=(2,), name='encoder_input')
80    # hidden layer
81    hidden_size = 2
82    x = layers.Dense(hidden_size, activation=activation)(encoder_input)
83    # Encoded to latent_dim latent space variables:
84    encoder_output = layers.Dense(latent_dim, activation='tanh', name='bottleneck')(x)
85
86    # Create the encoder model
87    linear_encoder = models.Model(encoder_input, encoder_output)
```

and similarly for a decoder with a non-linear activation function and a single hidden layer:

```
88    # Define the decoder architecture
89    decoder_input = layers.Input(shape=(latent_dim,), name='decoder_input')
90    # hidden layer
91    x = layers.Dense(hidden_size, activation=activation)(decoder_input)
92    # output layer
93    decoder_output = layers.Dense(2, activation=activation, name='decoder_output')(x)
94
```

```
    # Create the decoder model
    linear_decoder = models.Model(decoder_input, decoder_output)
```

Finally, we define couple again the encoder and decoder to form the autoencoder, which we compile and train as before:

```
    from tensorflow.keras import optimizers
    # Compile the model
    opt = optimizers.Adam(learning_rate=1e-2)
    linear_autoencoder.compile(optimizer=opt, loss='mean_squared_error')
    linear_autoencoder.fit(data, data, epochs=400, batch_size=256, shuffle=True)
```

We hhad to increase the learning rate in this example in order to achieve faster convergence. The results of the deep autoencoder are shown in Figure 18 in comparison to a rank-r SVD approximation. The results are identical, as expected.

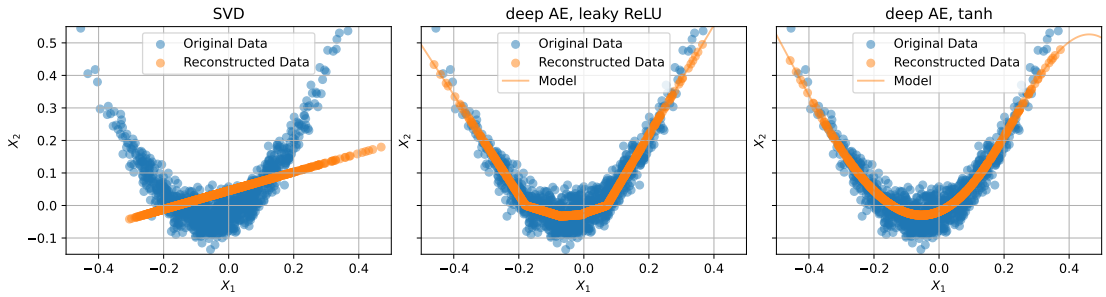

Figure 18: A deep autoencoder implemented as a neural network in Keras. The encoder consists of a single dense layer with a non-linear activation function, the decoder consists of a single dense layer with a non-linear activation function. The network is trained to minimize the difference between input and output, which yields a non-linear generalisation of the SVD.

1.9 Data-Driven Basis III: data-driven bases with SVD and autoencoders

Assume we have data of a common structure, e.g. images of galaxies, or stellar spectra. In this case, there is likely a common structure in the data that can be exploited. The SVD is a powerful tool to extract this common structure if the data is linearly separable.

Assume we have a large number of m data samples $\boldsymbol{x}_{i=1\ldots m} \in \mathbb{R}^n$ from our n-dimensional sample space (i.e. $\dim(\mathcal{V}) = n$). Let us arrange this data in a matrix

$$\mathrm{X} = \begin{bmatrix} | & | & & | \\ \boldsymbol{x}_1 & \boldsymbol{x}_2 & \ldots & \boldsymbol{x}_m \\ | & | & & | \end{bmatrix}. \tag{41}$$

Each data vector $\boldsymbol{x}_i$ can be a 'true' vector, but also a flattened image, or a higher dimensional flattened object.

1.9.1 Application of SVD to spectra

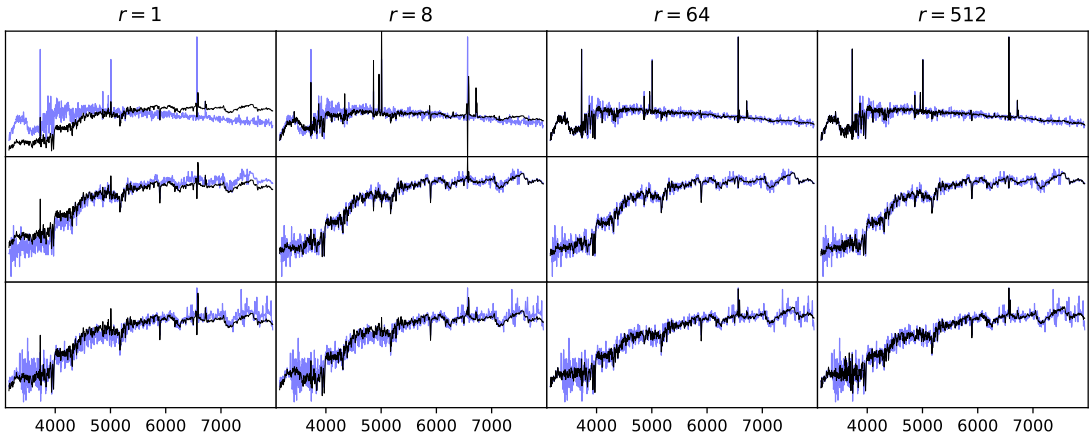

Figure 19: Low-rank SVD approximations to SDSS spectra. From top to bottom the results for three galaxies are shown. Blue lines indicate the true spectrum, black lines the low-rank approximation. From left to right, between 1 and 512 components are added to the mean spectrum (obtained from the entire sample).

Compression of SDSS spectra. This data-driven approximation property makes SVD useful for low rank data representation. As an example, in Figure 19, we show low-rank reconstructions of SDSS galaxy spectra. The singular vectors were computed from 4000 spectra. In the figure, the blue lines indicate the true spectra, the black lines a rank-r approximation based on the first r singular vectors.

1.9.2 Application to image data

Assuming that a non-linear basis can work better, we can use an autoencoder to find a common basis for the galaxy image data. We shall use a deep convolutional autoencoder for this task. THe introduction of *convolutional layers* allows the network to learn local features in the image data. Specifically, it enables the network to explicitly be aware of neighbourhood information in the image.

We will introduce convolutional neural network in more detail in a later lecture, we will here only show how to implement a convolutional autoencoder in Keras as a teaser of things to come.

Assume `SIZE` is the size of the image, and `CHANNELS` is the number of colour channels. We can define the encoder as follows:

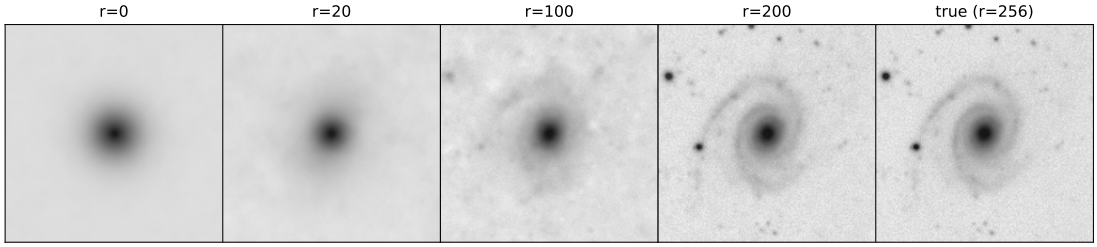

Figure 20: Rank-r full sample SVD approximations of the galaxy image. In contrast to Figure 10, the singular value decomposition is made over the sample of all galaxies (as for the galaxy spectra shown of Fig. 19). We express this particular image in the 'eigenbasis' of galaxy images. In contrast to the individual galaxy SVD approximation, the full sample SVD approximation is computed offline and can be used to project any galaxy image onto the low-rank subspace. Naturally it requires a higher r than the individual galaxy low-rank SVD approximation.

```
EMBEDDING_DIM=4

# Encoder
encoder_input = layers.Input(shape=(SIZE, SIZE, CHANNELS), name="encoder_input")
x = layers.Conv2D(32, (3, 3), strides=2, activation="relu", padding="same")(encoder_input)
x = layers.Conv2D(64, (3, 3), strides=2, activation="relu", padding="same")(x)
x = layers.Conv2D(128, (3, 3), strides=2, activation="relu", padding="same")(x)
shape_before_flattening = K.int_shape(x)[1:]  # the decoder will need this!
x = layers.Flatten()(x)
encoder_output = layers.Dense(EMBEDDING_DIM, name="encoder_output")(x)
encoder = models.Model(encoder_input, encoder_output)
encoder.summary()
```

At the same time, we define the decoder as follows:

```
# Decoder
decoder_input = layers.Input(shape=(EMBEDDING_DIM,), name="decoder_input")
x = layers.Dense(np.prod(shape_before_flattening))(decoder_input)
x = layers.Reshape(shape_before_flattening)(x)
x = layers.Conv2DTranspose( 128, (3, 3), strides=2, activation="relu", padding="same" )(x)
x = layers.Conv2DTranspose(  64, (3, 3), strides=2, activation="relu", padding="same" )(x)
x = layers.Conv2DTranspose(  32, (3, 3), strides=2, activation="relu", padding="same" )(x)
decoder_output = layers.Conv2D(CHANNELS, (3, 3), strides=1, activation="sigmoid",
    padding="same", name="decoder_output")(x)
decoder = models.Model(decoder_input, decoder_output)
decoder.summary()
```

As before, we train the autoencoder by compiling it and fitting it to the data:

```
# Autoencoder
autoencoder = models.Model(encoder_input, decoder(encoder_output))
autoencoder.compile(optimizer="adam", loss="mean_squared_error")

```

```
EPOCHS = 200
BATCH_SIZE = 128

autoencoder.fit(
    x_train,
    x_train,
    epochs=EPOCHS,
    batch_size=BATCH_SIZE,
    shuffle=True,
    validation_data=(x_test, x_test),
    callbacks=[model_checkpoint_callback, tensorboard_callback],
)
```

The result for several dimensions of the latent space is shown in Figures 21-**??**

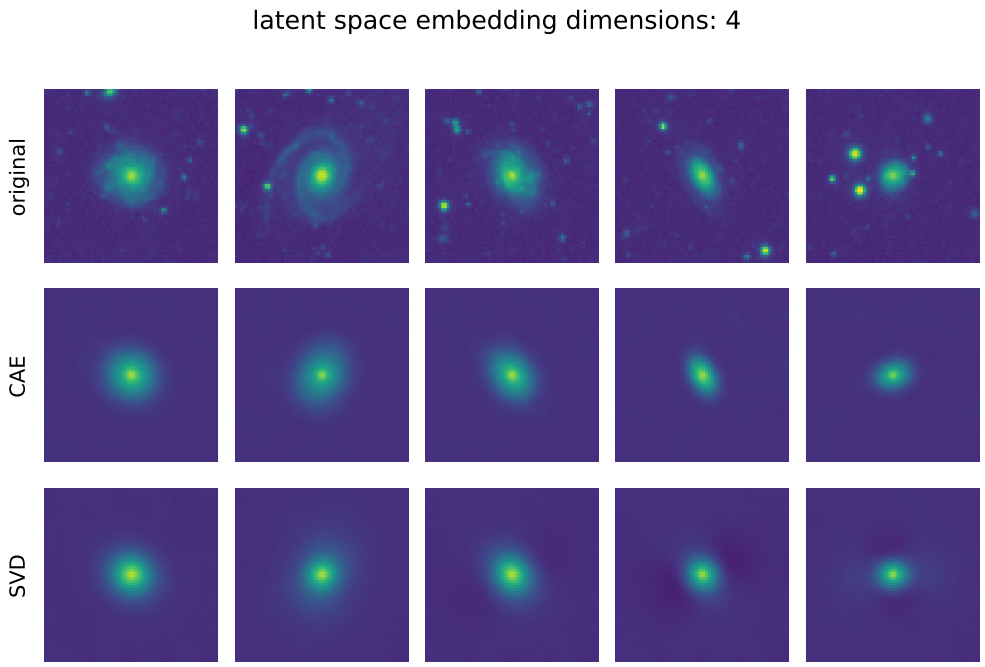

Figure 21: Comparison of a convolutional autoencoder (CAE) with an r-dimensional latent space, and a rank-r SVD approximation of a galaxy image for $r = 4$. The CAE is trained on a sample of galaxy images, the SVD is computed from the same sample. The CAE is able to capture non-linear features in the data, while the SVD is a linear decomposition.

Despite the low dimensionality of the latent space, the convolutional autoencoder model is able to capture crisp details that are not represented in the SVD approximation. This is both due to the non-linear nature of the autoencoder and the convolutional layers.

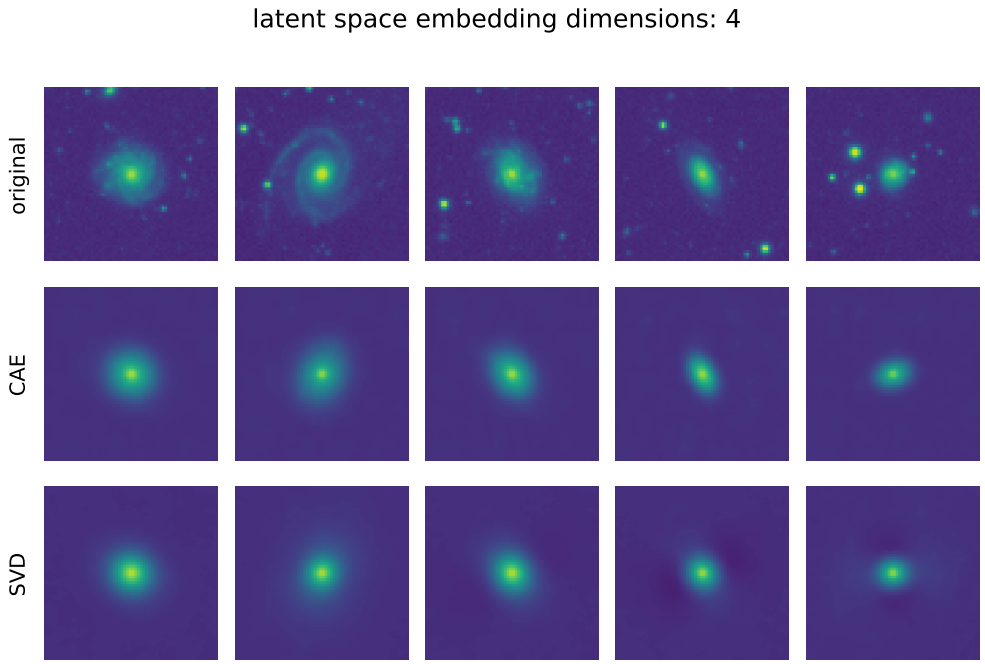

Figure 22: Same as Figure 21 but for $r = 16$.

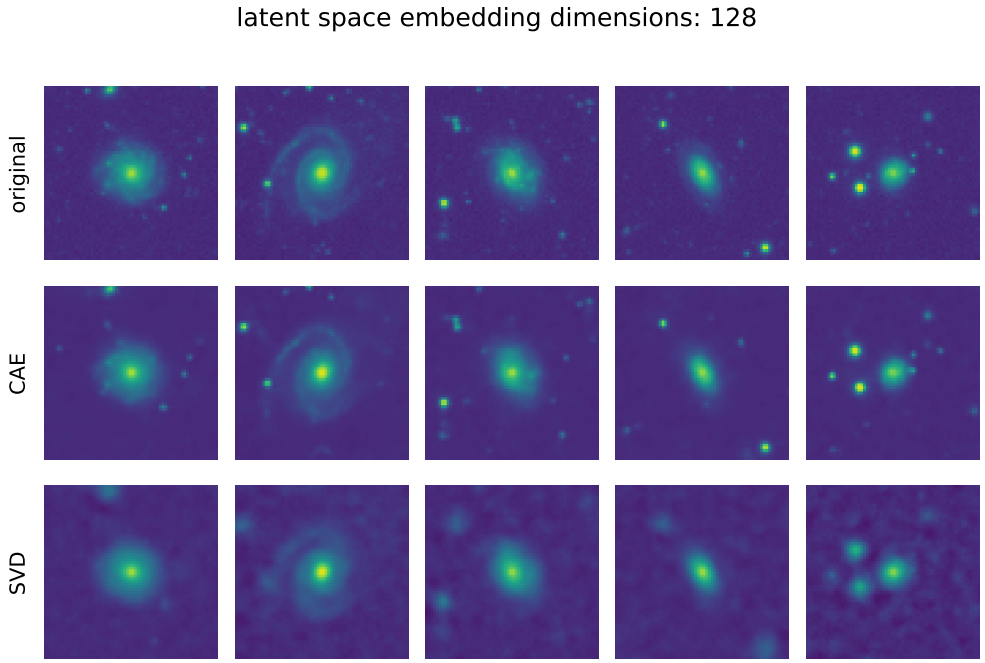

Figure 23: Same as Figure 21 but for $r = 128$.

References

Steven L. Brunton and J. Nathan Kutz. *Data-Driven Science and Engineering: Machine Learning, Dynamical Systems, and Control*. Cambridge University Press, 2 edition, 2022.

Matan Gavish and David L. Donoho. The optimal hard threshold for singular values is $4/\sqrt{3}$. *IEEE Transactions on Information Theory*, 60(8):5040–5053, 2014. doi: 10.1109/TIT.2014.2323359.