FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Usage of neural network for non-negative factorization
Student:	Tomáš Gregor
Supervisor:	doc. Ing. Ivan Šimeček, Ph.D.
Study program:	Informatics
Branch / specialization:	Knowledge Engineering
Department:	Department of Applied Mathematics
Validity:	until the end of summer semester 2023/2024

Instructions

1. NMF is an iterative, approximative method of matrix factorization. This method is also suitable for image compression[1].
2. Study other existing methods for image processing by neural networks. Identify suitable methods and technologies for input matrices initialization.
3. Propose and implement a neural network for initializing matrices for image compression using the NMF algorithm. Perform experiments with different hyperparameter configurations
4. Measure performance improvements against more traditional initialization techniques.

[1] Pikna, M.: Evaluating performance of an image compression scheme based on non-negative matrix factorization. Diploma thesis, CTU FIT, 2019.

Electronically approved by Ing. Magda Friedjungová, Ph.D. on 4 February 2023 in Prague.

Bachelor's thesis

THE USE OF NEURAL NETWORKS FOR IMAGE COMPRESSION USING NON-NEGATIVE MATRIX FACTORIZATION

Tomáš Gregor

Faculty of Information Technology
Department of Applied Mathematics
Supervisor: doc. Ing. Ivan Šimeček, Ph.D.
May 11, 2023

Citation of this thesis: Gregor Tomáš. *The use of neural networks for image compression using non-negative matrix factorization.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023.

Contents

Acknowledgments vii

Declaration viii

Abstract ix

Introduction 1

1 Introduction to Non-negative Matrix Factorization 5

1.1 NMF algorithms 6

1.2 NMF initialization methods 8

1.2.1 Random-based algorithms 9

1.2.2 Clustering-based algorithms 9

1.2.3 Low-rank-based algorithms 9

2 Artificial Neural Networks 11

2.1 Back-Propagation 13

2.2 Loss functions 14

2.3 Activation functions 14

2.4 Optimizers 16

2.5 Architecture 16

2.5.1 Autoencoders 17

2.5.2 Convolutional Neural Networks 18

2.6 Regularisation 19

3 Proposed model 21

3.1 Data modeling 22
3.1.1 Data aquisition 22
3.1.2 Data preprocessing 22
3.1.3 Statistical model of factor matrices 23
3.2 Description of the model 25
3.2.1 Sliding window 26
3.2.2 Semi-autoencoder 27

4 Implementation 29

4.1 Chosen technologies 30
4.1.1 Programming language 30
4.1.2 Jupyter notebooks 30
4.1.3 Neural network framework 30
4.1.4 Supporting libraries 31
4.2 Training infrastructure 31
4.3 Hyperparameter tuning 31

5 Evaluation 35

5.1 Choosing best hyperparameter configuration 35
5.2 Comparison with other NMF initialization methods 38

Conclusion and future work 41

Abbreviations 47

Definitions 48

List of Figures

1.1 Comparion of NMF and PCA representations 6

2.1 A visualization of a simple neural network 12

2.2 Functional representation of the feedforward network 13

2.3 Linear Unit function 15

2.4 Rectified Lienar Unit funciton 15

2.5 Sigmoid funciton 15

2.6 Softmax for a vector (v_1, v_2) 15

2.7 A visualization under-complete autonecoder 17

2.8 An example of a typical CNN network. 18

3.1 The mean and standard deviation data of the W and H matrices 22

3.2 Polynomial curves fitted to the mean and standard deviation data of the W and H matrices 23

3.3 The final statistical models for the mean and standard deviation data of the W and H matrices. 24

3.4 Visualisation of the sliding window component of the WNet model 26

3.5 Visualisation of the semi-autoencoder component of the WNet model 27

4.1 An example of results of training the NMFNet model 33

5.1 Result of the hyperparameter tuning for each hyperparameter 35

5.2 Correlation of the funnel reduction factor with the model training time 36

5.3 Comparison of the results of the MAE and MSE loss functions. 36

5.4 The training results of the NMFNet model chosen for evaluation 37

5.5 The results of comparison between the initialization methods. 38

5.6 A sample of the images reconstructed directly from the initial settings. 39

List of Tables

3.1 The values of parameters chosen for the NMF decompositions. 24

4.1 Values of the training loop hyperparameters. 29

4.2 The values of the optimizer hyperparameters. 29

4.3 Hyperparameters and their values chosen for the grid search 32

4.4 Grid search result statistics 32

5.1 The best-found hyperparameter configuration. 37

5.2 Percentage of the evaluation dataset on which the NMFNet model performed relative to the other methods 39

I would like to thank my supervisor for the advice given and the guidance during the writing of this thesis. I am also thankful for the support of my friends, whom kept me in good spirits even in times of strife, and my family, for providing me with a loving home to come back to.

Declaration

In Prague on May 11, 2023 ....

Abstract

Nonnegative matrix factorization is a method of matrix factorization, that can approximate data by a low-rank representation. This representation can be exploited for reducing the size of an image file while keeping most of the visual quality. An initialization of the decomposition algorithm is needed to produce the low-rank approximation. In this work, we propose the NMFNet neural model to accomplish the task of this initialization. The model is then compared to other initialization techniques used in practice. Random initialization, NNDSVD initialization and K-means clustering were chosen for this comparison. Our model was shown to compare favorably to these methods.

Keywords Image compression, Nonnegative matrix factorization, Nonnegative matrix factorization initialization, Machine learning, Artificial neural network, Autoencoder

Abstrakt

Nezáporná faktorizace matic je metoda maticové faktorizace, která data reprezentuje v prostoru nižšího řádu. Tato reprezentace může být využita při kompresi obrázků. Algoritmy pro výpočet tohoto rozkladu potřebují nějaké prvotní nastavení, tato práce se zabývá problémem hledání tohoto prvotního nastavení, které vyústí v co nejlepší možnou kvalitu komprimovaného obrázku. Navrhli jsme neurální model NMFNet, který takové prvnotní nastavení najde a tento model jsme porovnali s technikami běžně používanými v praxi. Pro toto porovnání byly zvoleny 3 techniky: náhodná inicializace, NNDSVD inicializace a K-means shlukování. Bylo ukázáno, že náš model má dobré výsledky ve srovnání s těmito technikami.

Klíčová slova Komprese obrázků, Nezáporná faktorizace matic, Inicializace nezáporné faktorizace matic, Strojové učení, Umělá neuronová síť, Autoencoder

Introduction

Matrix factorization techniques have been widely used over the past few decades to a great effect in recommender systems, genome analysis, image processing and many other applications. The idea behind matrix factorization is to take a large matrix and decompose it into two or more usually smaller matrices. From these matrices, the original matrix can be reconstructed with some degree of approximation by multiplying these smaller matrices. The goal of this exercise is to concentrate the information of the large matrix in useful ways so insights can be extracted.

The other benefit of the decomposition into smaller matrices is the potential reduction in space needed to store the data, thus it can be used for data compression. This work will use a Non-negative Matrix Factorization (NMF) based strategy to compress images. We build on the work of Marek Pikna, who in his master's thesis proposed an NMF-based compression scheme and compared it with state-of-the-art compression algorithms[1].

NMF is a method of factorizing non-negative matrices – for example images – into two also non-negative matrices, that can be much smaller (meaning can have much fewer elements combined) than the initial matrix. This method, however, is highly dependent on the initial setting of its algorithm. With a good initial setting, NMF can find the solution to the decomposition much faster and the solution can be of much higher quality. Without it, the algorithms are not guaranteed to converge at all. Unfortunately, there does not exist a simple rule for picking the correct initialization method for the problem a researcher is trying to solve. A try-and-see approach is often chosen for deciding which of the strategies is to be used.

Artificial Neural Networks (ANNs) are state-of-the-art machine learning models, with applications across almost every domain. Their high adaptability, ability to learn complex relationships from the data and resistance to the data representation are just some of the advantages they bring to the table and why they are used so heavily by the machine learning community.

In this work, we try to join these 2 methods, ANN for the initialization and NMF for obtaining a more efficient representation of the data, for the purpose of image compression. We hope to improve Pikna's compression scheme, specifically to reduce the number of iterations in the NMF step.

Goals

The goal of this work is to use an ANN to find a good first setting of an NMF algorithm and compare our results with more traditional initialization techniques. The idea of using ANN for NMF initialization has not yet been explored much in the literature and we hope to gain some useful insights into the viability of this strategy.

In order to achieve this goal, we must first:

- analyze existing NMF initialization strategies,
- analyze different ANN architectures useful for image processing,

This will be the loose structure of the theoretical part of this work, the practical part will be structured as follows:

- propose a model for NMF initialization,
- choose technologies best suited for our task,
- train the model and determine the best possible hyperparameter configuration,
- compare our model to more traditional initialization strategies.

Related works

In his master's thesis [1], Marek Pikna proposed a compression scheme based on Nonnegative Matrix Factorization (NMF). He then experimented with different representations of the image data and the NMF algorithm parameters, namely the rank of the decomposition and the number of iterations of the algorithm. Some initialization methods were also tried.

His scheme uses an algorithm based on multiplicative updates with squared Euclidean distance. This algorithm is further discussed in Section 1.1. Much thought was given to the color representation of the image. The "naive RGB", "separate RGB" and the "Y'$C_B C_R$" compression schemes were developed. In the naive RGB scheme, the 3 color channels of the image first merged into a single matrix, this matrix was then decomposed using the NMF algorithm. The separate RGB scheme used 3 decompositions, each color channel was decomposed by itself.

The Y'$C_B C_R$ compression was the most interesting. The Y'$C_B C_R$ color space leverages the way the human eye perceives colors. The eye is significantly more sensitive to green than the other two primary colors. Instead of using the 3 color channels for representing the primary colors, Y'$C_B C_R$ combines the colors in certain proportions in the Y' channel of the image. This channel is called **luma**. As a result, the luma channel contains the black-and-white representation of the image, and the C_B and C_R channels are then used to add color to produce the final colored image. The compression scheme using the Y'$C_B C_R$ color space further prevented the loss of visual quality of the image by compressing only the C_B and C_R channels of the image, leaving the luma untouched.

For each of these spaces, the rank and number of iterations variables were tested. The Y'$C_B C_R$ scheme performed the best. Three of the NMF initialization methods were used: random initialization, random Vcol initialization and the Non-Negative Double Singular Value Decomposition (NNDSVD). These methods are described in Section 1.2 of this work. Interestingly the NNDSVD was significantly outperformed by the random initializations in the image quality tests. On the other hand, some artifacts were introduced on artificial images (solid color images) by random initialization. The random and random Vcol initializations performed about the same. The compression time for decomposition with a small rank, e.g. 10, and 300 iterations is stated to be around 10 seconds, when increasing the rank to 150, compression time rose to 20 seconds. These figures however are to be taken as a relative measure because no hardware details were provided.

Chapter 1

Introduction to Non-negative Matrix Factorization

▶ **Definition 1.1.** ***Non-negative Matrix Factorization (NMF)*** *is a method of matrix factorization. Given matrix $V \in \mathbb{R}_{0+}^{n \times m}$, find matrices $W \in \mathbb{R}_{0+}^{n \times r}$ and $H \in \mathbb{R}_{0+}^{r \times m}$, such that*

$$V \approx WH. \tag{1}$$

When V is interpreted as a data matrix, where m columns represent m data samples and n rows represent n features, matrix W is a basis matrix containing r row base vectors, serving as the latent features, and H is a coefficient matrix by which each data sample can be approximatively reconstructed by multiplying the corresponding column of H with the base matrix W.[2, 3, 4]

NMF was first described in [2] by Lee and Seung. They proposed this matrix factorization technique as an alternative to other state-of-the-art machine learning (ML) methods of the time. They see a parts-based representation as the main advantage of NMF in contrast to for example Principal Component Analysis (PCA) that describes the data globally. The method achieves this by constraining the decomposition by non-negativity, in other words only additive combinations are allowed in the representation.[4] This is perhaps best demonstrated by the now-famous comparison of the learned representation of images of faces by NMF and PCA algorithms. As can be seen in Fig. 1.1 the pure additivity of NMF leads to parts-based feature extraction and thus to better interpretability.

A highly useful property of NMF is the low-rank representation it produces. This property is particularly useful in the context of data compression and feature extraction. As can be seen from Def. 1.1, when r is chosen such that $r < \frac{\min(n,m)}{2}$, the number of elements of the V matrix is strictly lower than the number of elements of the W and H matrices combined.

Low-rank approximation algorithms have been a topic of discussion in the ML community for many years. The dimensionality reduction in the data allows for valuable insights to be gained and great interpretability of the results. It is important to note, that under certain conditions NMF can even derive the ground truth from the data, and thus can be used as a generative model.[5, 4]

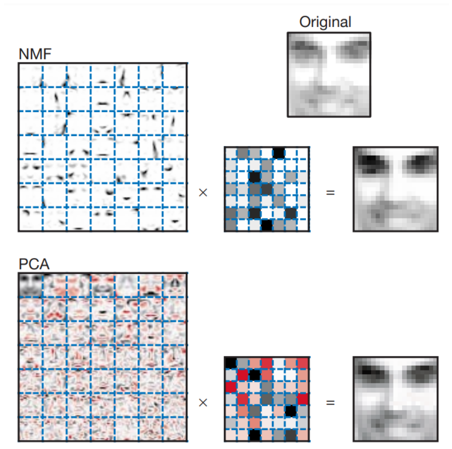

Figure 1.1 Comparion of NMF and PCA representations. The left matrices are the matrices composed of the learned basis vectors (the W matrix for the NMF). Each square of these basis matrices represents a single basis vector, which has been folded to the shape of the original image for a better illustration. The middle matrices are the coordinate vectors corresponding to the original image, the vectors were folded in the same manner as the basis vectors. The values of the elements of the vectors are represented using color: white represents 0, the more black an element is, the more positive the value, conversely the more red, the more negative.[2]

Many subcategories exist for NMF, the most fundamental of which is the Basic NMF. Basic NMF is the problem described above, the only constraint is the non-negativity of the V, W and H matrices. The specific algorithms used for computing its solution are described in Section 1.1 below. Exact NMF is a problem where we try to decide if a non-approximating (exact) solution exists for the factorization, and if so, find the solution. The algorithms for solving Exact NMF are however proven to be NP-hard, and therefore the algorithms for finding the exact solution have limited application in practice.[6]

Many Constraint NMF subproblems exist, where additional constraints are imposed on the decomposition, such as sparseness, orthogonality or discrimination. The goal of additional constraints is to improve NMF performance in specific tasks, such as classification or clustering, or improving the uniqueness of the solution. Generalized versions of the NMF also have their application: Semi-NMF drops the non-negativity constraint on the V and H matrices, therefore it can be used on any data matrix; Non-negative Tensor Factorization decomposes tensors (a generalization of a matrix to multi-dimensional space) instead of matrices.[4]

Since the Exact NMF problem is NP-hard, the other variants often use iterative algorithms for finding the solution. A direct consequence of this is the need for some initial setting of the W and H matrices. The reason behind this is the iterative algorithms only improve the previous solution, instead of computing the solution outright and so some initial setting is required. Plenty of methods have been explored for this task, they can be categorized into 4 categories: Random, Clustering, Low-rank and Heuristic.[7] Specific algorithms are described in the section below.

1.1 NMF algorithms

The classic approach to computing a solution of the decomposition is through alternating minimalization of an objective function. The two widely used objective functions are the Square of Euclidian Distance (SED) and the Generalized Kullback-Leibler Divergence (GKLD). SED function is based on minimizing the sum of the squares of the entry-wise differences of the V and WH matrices. The advantage of this function is its simplicity and ease of computation. It is described by the equation below:

$$SED(W, H) = \frac{1}{2} \|V - WH\|_2^2 . \tag{2}$$

GKLD is more widely used in practice than the SED. Interestingly the function is not symmetrical, and so it cannot be called a distance, instead the function measures the relative entropy of the matrices. It is described by the equation below:

$$D_{KL}(V, WH) = \sum_{i,j} (V_{i,j} \log \frac{V_{i,j}}{(WH)_{i,j}} - V_{i,j} + (WH)_{i,j}). \tag{3}$$

It is important to note, that both functions are convex in W and H separately, but not in both W and H together. This implies that a convergence to a global minimum is not realistic and only a local minimum convergence can be expected. To be more precise, the algorithm converges only to a stationary point, as proven by [8]. The original Multiplicative Updates (MU) algorithm, developed by Lee and Seung[3], is still widely used as a baseline. "... [The algorithm] is simple and parameter-free with a low cost per iteration, but [it] converges slowly due to a first-order convergence rate."[4]

Algorithm 1: A generic MU framework

```
  Input  : $V \in \mathbb{R}^{m \times n}, r \in \mathbb{N}_+$
  Output: $W \in \mathbb{R}^{m \times r}, H \in \mathbb{R}^{r \times n}$
1 Initialize $W$ and $H$
2 while Stop condition not reached do
3     Apply the update rule for $W$
4     Apply the update rule for $H$
5 end
6 return $W, H$
```

For each of the objective functions, different update rules must be derived. For the SED objective function the update rules are defined as follows:

$$H_{\alpha\mu} \leftarrow H_{\alpha\mu} \frac{(W^T V)_{\alpha\mu}}{(W^T W H)_{\alpha\mu}} \qquad W_{i\alpha} \leftarrow W_{i\alpha} \frac{(V H^T)_{i\alpha}}{(W H H^T)_{i\alpha}}. \tag{4}$$

The update rules for the GKLD objective function are defined as follows:

$$H_{\alpha\mu} \leftarrow H_{\alpha\mu} \frac{\sum_i W_{i\alpha} V_{i\mu}/(WH)_{i\mu}}{\sum_k W_{k\alpha}} \qquad W_{i\alpha} \leftarrow W_{i\alpha} \frac{\sum_\mu H_{\alpha\mu} V_{i\mu}/(WH)_{i\mu}}{\sum_\nu H_{\alpha\nu}}. \tag{5}$$

Other algorithms of note are the Alternating Non-negative Least Squares[9] and Quasi-Newton Optimalization[10], these however are not relevant to this work since the compression scheme proposed by Pikna uses the multiplicative updates algorithm.

1.2 NMF initialization methods

The initial setting of the factor matrices is crucial to fast convergence and quality of the result. In practice, random initialization strategies are employed most frequently, their advantage being simplicity and very fast initialization. The disadvantages of these strategies are slow convergence and results are not reproducible. As a result, factorizations using this method must be performed multiple times to make sure the solution found was in fact the best possible.[7]

The idea behind the clustering-based algorithms is the NMF is very similar to K-Means clustering and in some cases even mathematically equivalent.[11] The similarity of these methods leads in some cases to an initialization that is close to the convergence point of NMF, thus reducing the number of iterations needed. The method also leads to great interpretability, since the clustering can summarise the data well. The application of clustering for initialization has great results in environmental research and signal and image processing.

Low-rank initialization for NMF uses some other low-rank representation method to initialize the factor matrices. The most popular choices are Singular Value Decomposition (SVD), Non-Negative Double SVD (NNDSVD), Non-negative Principal Component Analysis and Non-negative Independent Component Analysis. These approaches try to combine the advantages of NMF with other low-rank representation methods, hence producing a better solution for specific tasks.[7]

Heuristic initialization is the most recent and the least explored area of the initialization algorithms for NMF. Some notable algorithms are based on Population-Based Algorithms and the Genetic Algorithm The problem with using heuristic initialization for NMF is the lack of proof of convergence, despite that, they can have good results in certain applications.[12]

In the subsections below are listed and briefly described initialization strategies used in literature. If no initialization strategy is specified for the H matrix, random initialization is used or the values are computed from W by the least squares method.

1.2.1 Random-based algorithms

Random initialization sets the factor matrices with random positive values.

Random Vcol initialization constructs each column of W as an average of p randomly selected columns of V.[13]

Co-occurrence initialization first computes a co-occurrence matrix of the data, then chooses r of its columns as W.[13, 14]

Gabor wavelet initialization sets the values of W based on the Gabor wavelet[15, 16], it is particularly useful when extracting features from images. The wavelet generates a specified number of transformations of the original data, these transformed images are then used as columns of W.

1.2.2 Clustering-based algorithms

K-means initialization performs standard K-mean clustering on the data, the number of clusters is chosen as r. The coordinates of the cluster centroids are then used as columns of W

Fuzzy C-means initialization performs C-mean clustering on the data, and the number of clusters is chosen to be r. This assigns a membership value to each of the data points for each cluster, this matrix is then used as W.

Hierarchical clustering initialization groups the data attributes instead of the data points themselves. The matrix rank is used as a distance metric, so linearly dependent attributes fall into the same cluster.[17]

1.2.3 Low-rank-based algorithms

Singular Value Decomposition initialization first computes SVD of the data, the negative elements of U and V matrices are set to 0. r first singular values are then chosen and the corresponding truncated matrices U and V are used as W and H.

Non-Negative Double Singular Value Decomposition initialization is based on two runs of SVD, the first approximates the data and the second approximates the positive parts of the first decomposition.[18]

Non-negative Principal Component Analysis initialization performs PCA on the data, the basis vectors of r first principal components are then used as columns of W with their negative elements set to 0.

Chapter 2

Artificial Neural Networks

Artificial Neural Networks are a class of machine learning models, that use units called **neurons** to approximate some function based on a sample of data points from that function. As the reader might have guessed, the use of the term ”neural” in the name of these models is not a coincidence. The first ANN model was proposed by American psychologist Frank Rosenblatt in 1958. He described a model, called The perceptron, based on the structure of neurons in the human eye. This model was then trained and it learned to recognize simple shapes in images. For example, it could distinguish the letter ”X” from the letter ”E”.[19, 20]

A biological neuron is a type of brain cell with 3 distinct parts: the cell body, the dendrites and the axons. The dendrites are a part of the cell responsible for receiving signals from other neurons. Vice versa the axon is responsible for sending signals to the other neurons by connecting to their dendrites. The more signals are sent over this neural connection, the stronger the link becomes. It is believed that from this behavior, consciousness and the ability to learn emerges, though none have yet been able to describe this process fully.[21]

Analogously neuron in an ANN is a function that maps n inputs (a parallel to dendrites) to usually one output (a parallel to axon), though more outputs are not disallowed. The domain of inputs and outputs is arbitrary, they can be scalars, vectors, matrices and even tensors over the real or the complex set. This work focuses on a category of ANNs: Feedforward Neural Networks. The word feedforward in their name represents the fact they can be expressed by a directed acyclic graph with source and sink nodes and information flows through them in accordance with this graph, from the inputs (sources) to the outputs (sinks).

The neurons are generally organized in layers. The first layer of the network is called the **input layer**, the final layer is called the **output layer**. All the layers between the input layer and the output layer are called the **hidden layers**. For a better visual representation see Fig. 2.1. The number of layers in a model determines its **depth**. The number of neurons in a layer is the layer’s **width**.

The edges of the ANN graph represent connections between the neurons, each of the connections has a weight. The output of the first neuron is first multiplied by the weight of the connection and is then given to the second neuron as an input. The values of the weights of the connections are learned parameters. At the beginning of the learning process, they are initialized to small random values.

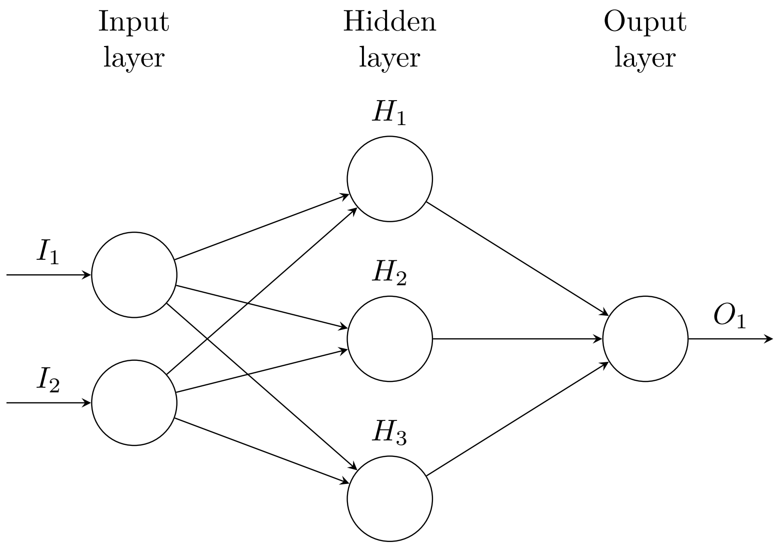

Figure 2.1 A visualization of a simple neural network

The function, which maps the neuron's inputs to the outputs is called the **activation function**. The specific types of activation functions are discussed in Section 2.3 below. The desirable properties of an activation function are as follows:

- the function domain is equal or a superset of the set of all possible inputs,
- the function is continuously differentiable[1].

The former property is self-explanatory, a situation in which a neuron cannot map an input to an output is quite undesirable. The latter property is related to the learning process. For each neuron, we want to be able to compute the gradient of its activation function, in other words, to tell how quickly each of the input variables changes the output of the neuron and with it, how it affects the network's error. If we then know the amplitude of the error of the neuron's output, we can adjust the neuron's weights and parameters based on the derivation. The method of obtaining the gradient for the neurons is called back-propagation, backprop for short. Backprop is discussed in Section 2.1 of this chapter.

The process of teaching an ANN is very similar to teaching any other model. First, a **loss function** must be defined. A loss function $L(y, \widehat{y})$ compares the desired output y with the actual output of the network $\widehat{y}$ and yields a scalar error. As discussed, this error can then be used together with the neuron gradients to update the network's parameters. This process is called **optimization**. Types of loss functions and optimization algorithms are discussed in Sections 2.2 and 2.4 respectively.

[1]Though functions with non-existing differentials at points are also widely used, with explicitly defined point differentials.

2.1 Back-Propagation

In this section, we'll describe the process of computing the neuron gradients. Back-propagation is the most popular method of obtaining the gradients, but it is not the only method. It is helpful to think of each layer of an ANN as a function. This function is defined by the activation functions of the neurons inside the layer. If we find the gradient of the whole layer, implicitly we have also found the gradients for each of the neurons. The network then can be thought of as a composite function consisting of these layer functions. Fig. 2.2 helps to illustrate this point.

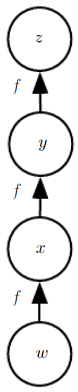

Figure 2.2 Functional representation of the feedforward network. Variable w serves as the network's input, variables x and y are the network's hidden states and z is the network's output. The function f is the representation of the layer transformation, in this case, the network has the same 3 layers.[22]

The network from the example can then be described with this equation:

$$z = f(f(f(x))), \tag{1}$$

where x is the input data and z is the prediction.

With this understanding of the model, deriving the gradient of the network's output with respect to the inputs can be calculated using the chain rule of calculus. Goodfellow describes the chain rule of calculus for obtaining the layer function derivatives in [22] thusly:

Let x be a real number, and let f and g both be functions mapping from a real number to a real number. Suppose that $y = g(x)$ and $z = f(g(x)) = f(y)$. Then the chain rule states that

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}.$$

From this, he then generalizes the rule for vectors and derives this equation, for the calculation of the network's gradient:

$$\Delta_x z = \frac{\partial z}{\partial y}^T \Delta_y z,$$

where $\partial y\ \partial x$ is the $n \times m$ Jacobian matrix of g.

With this calculation, we can determine the gradient of every layer with respect to the output of the network. If we then know the gradient of the output error, we can adjust the layer's parameters accordingly.

2.2 Loss functions

The loss function, sometimes called the objective function or the cost function, determines the model's error for a given pair of expected and predicted outcomes[2]. It is important when designing any ML model, to choose the loss function well. The goal of the model is to minimize the loss function and a bad choice can lead to the model producing undesirable solutions. The rest of this section is dedicated to describing 3 popular loss functions used in practice.

Mean Absolute Error (MAE) computes the average absolute difference between the expected and predicted output. This function is useful when we want to fit the model close to the median of the data since the function minimizes when the areas with high data density are fitted well. The main problem of this function lies in the failure to punish the error of the outlying data.

$$MAE(Y, \widehat{Y}) = \frac{1}{n} \sum |Y - \widehat{Y}|.$$

Mean Squared Error (MSE) computes the average squared difference of the expected and predicted output. In comparison with the MAE, MSE minimizes when the mean of the data is well-fitted. MSE also fits outlying data well, because it punishes the expected-predicted output distance progressively.

$$MSE(Y, \widehat{Y}) = \frac{1}{n} \sum (Y - \widehat{Y})^2.$$

Cross entropy is used mainly for classification tasks. It measures the number of bits needed to represent the expected probability distribution in the learned predicted probability distribution.

$$H = -\mathbb{E}_{X,Y \in \widehat{p}_{data}} \log p_{model}(Y|X),$$

where $\mathbb{E}$ is the expected value and p is the probability. It is interesting to note, that the formula is very similar to the calculation of entropy for a given discrete probability distribution.

2.3 Activation functions

The field of activation functions is a very diverse one. In fact, most new functions used in practice are not even published, unless they bring some clear and significant improvement. Here we list some of the most popular functions used.

Linear Unit (LU) is a very simple activation function. It uses the linear function with w and b as coefficient and intercept respectively. A network consisting only of linear units is itself linear and cannot learn to approximate a non-linear function effectively.

[2]Or a set of expected-predicted output pairs.

Rectified Linear Unit (ReLU) is a function $g(x) = max(0, x)$. It is used on top of the linear unit as a non-linearization operator. It is widely used today for its simplicity and great results. The unit suffers from the "dead ReLU" problem, where the input of the function is always negative, and thus the function always outputs a 0. The gradient of the functions is in this case flat and so the unit never unlearns this behavior. This property can be beneficial in applications, where sparse parameter space is desired.

Sigmoid function used to be the most popular activation function until it was overtaken by the ReLU. The sigmoid maps the whole real number line to an interval $(0, 1)$. This is a great property for binary classification tasks because we can interpret the value of the function as a probability, that the input belongs to a class. Inversely the function suffers from the "vanishing gradient" problem. When given an extreme input, positive or negative, a change in that output has an infinitesimal effect on the gradient. This can lead to a slow convergence when left unchecked.

Softmax function is a generalization of the sigmoid function for vectors. Values of all the elements of the vector are bounded to an interval $(0, 1)$ with the additional condition that L^1 norm of the output vector is equal to 1. In other words, we can think of the softmax as a probability distribution over the input vector's elements.

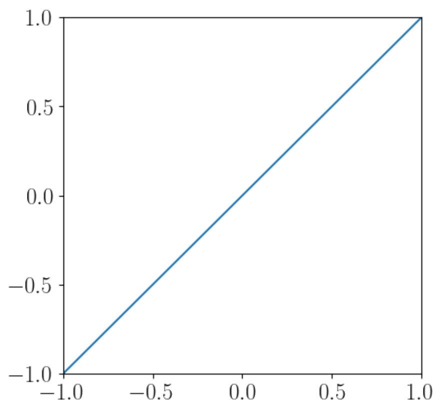

Figure 2.3 Linear Unit function

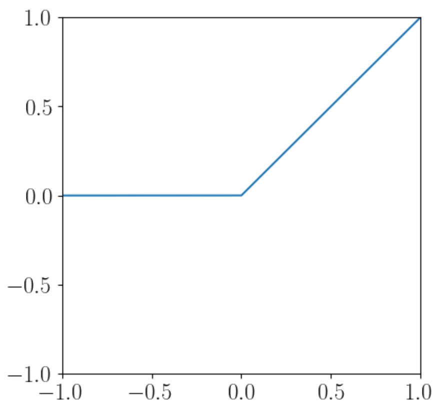

Figure 2.4 Rectified Lienar Unit funciton

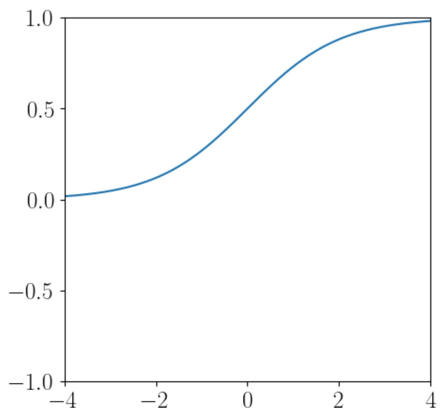

Figure 2.5 Sigmoid funciton

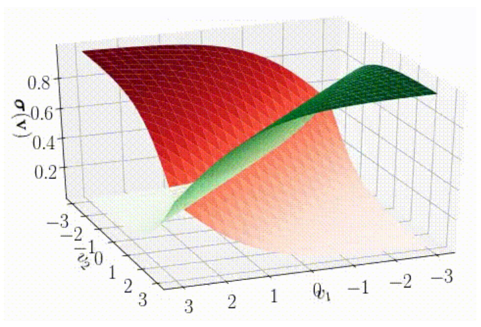

Figure 2.6 Softmax for a vector (v_1, v_2)[23]

2.4 Optimizers

The role of the optimizer in the neural network is to update the parameters of the network according to the gradient obtained from the back-propagation algorithm. Most optimizers are based on the Gradient Descent (GD) algorithm. GD is an iterative convex optimization algorithm, i.e. it minimizes convex functions, in our case, it minimizes the loss function. As the reader might have guessed, the problem in ANNs, and in other ML algorithms where GD is used, is the loss function is not strictly convex with respect to the model's parameters. This can result in the optimization finding some local non-optimal minimum instead of the optimum.

The addition of the so-called ”momentum” to the GD algorithm helps to improve its optimum-finding performance. The idea of the momentum is to take into consideration the gradients of the previous training steps instead of just the immediate gradient. The momentum causes the model to ”overshoot” the local minimum and in so doing can find a path to a better result. The overshooting, however, causes the convergence to be slower.

To increase the speed of convergence adaptive step size of the update step can be implemented. This technique increases the magnitude of parameter updates at the beginning of the learning process and gradually decreases it when approaching the optimum. Here we list a few of the most popular GD-based optimizer algorithms:

Stochastic Gradient Descent (SGD) is the standard GD algorithm for iterative convex optimization,

SGD with momentum uses the previous gradients in concord with the current gradient to optimize the parameters, the method is more resistant to converging to a local minimum instead of the global minimum than the standard SGD,

AdaGrad improves the rate of convergence of SGD with adaptive step size, with respect to all previous steps, however, this can lead to reducing the learning rate to effectively zero,

AdaDelta improves on the AdaGrad by introducing 2^{nd} moment of the previous gradients, also only a specified number of previous gradients is taken into account,

Adam further improves on AdaDelta by also using momentum, Adam is the most used algorithm in practice.

Algorithms not based on GD also exist, such as the genetic algorithm and the ant colony optimization, but these approaches are out of the scope of this work.

2.5 Architecture

The architecture of an ANN dictates, how many layers the network has, how many neurons are in each of the layers and how the neurons are connected. The rest of this section is dedicated to describing two architectures used for processing image data.

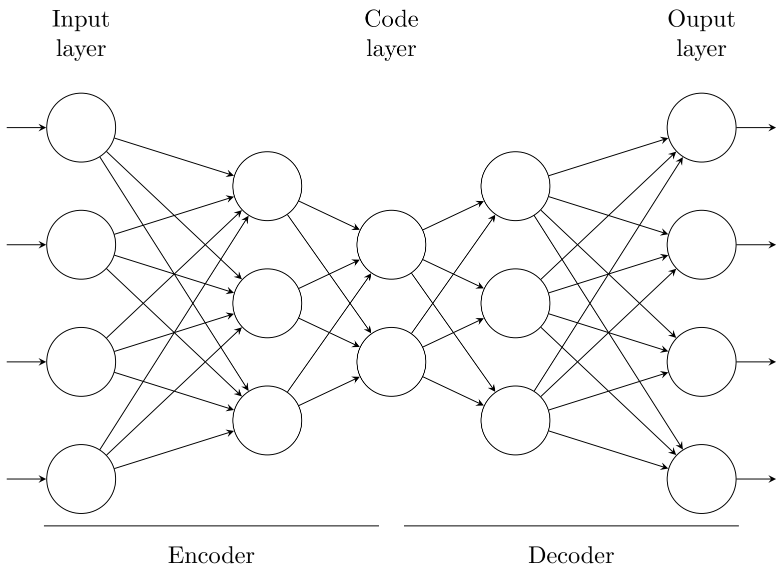

Figure 2.7 A visualization under-complete autonecoder

2.5.1 Autoencoders

Autoencoder is a special case of the feedforward network. It consists of two subnetworks, the encoder and the decoder. The output layer of the encoder acts as the input layer of the decoder, it is called the **code** layer or more colloquially the bottleneck. We will focus mainly on a type of autoencoder called the under-complete autoencoder, where the dimension of the code layer is less than the dimension of the input layer. Autoencoders with equal or higher dimension exist but are out of the scope of this work. The encoder and the decoder usually have the same architecture, just with the direction of the connections reversed. Fig. 2.7 provides some illustration.

The goal of the autoencoder is to reproduce the input of the network as closely as possible and in doing so learn a low-rank representation of the data on the code layer. Since the dimension of the code layer is smaller than the input's, the autoencoder has to represent the most important features of the data, to reproduce it most closely. This low-dimensional representation can then be used in many applications, namely data compression, feature representation, data clustering and others.

When the encoder a the decoder are linear and MSE loss function is used, the representation is similar to PCA. When used with non-linear functions, the autoencoder can then be interpreted as a non-linear generalization of the PCA.[24]

A problem one must face when designing an autoencoder is the model's capacity. When an autoencoder is too large, i.e. has too many layers or the layers are too wide, it can learn to encode the training data as an index. A network with even a single neuron in its code layer can then reproduce the data perfectly. The code representation in this case brings no value to a researcher and the model is unusable on data outside its training dataset.

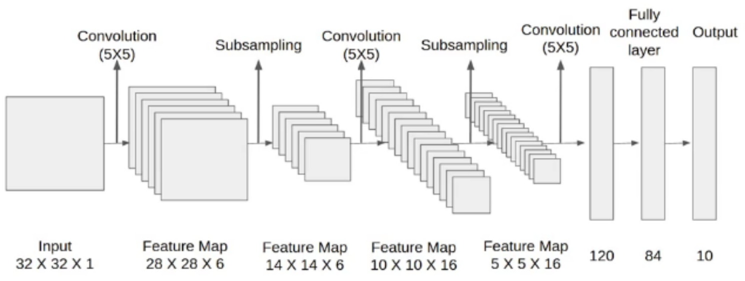

Figure 2.8 An example of a typical CNN network. Specifically, it is an overview of the LeNet-5 architecture.[25]

2.5.2 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a type of ANNs built specifically for image processing[3]. The problem with processing image data by a simple fully-connected feedforward network is the sheer number of connections from the input layer to the first hidden layer. Consider a greyscale 256-by-256-pixel image as an input of a network, for each neuron in the hidden layer, 65,536 connections need to be made, that is 65,536 parameters for the optimizer to optimize. CNNs sidestep this issue by first reducing the number of inputs to the fully-connected layers, by introducing alternating convolution and pooling layers. These layers are described below.

A convolution layer convolves the input by a relatively small matrix of parameters. To get back to our example, given 256-by-256 image as an input, we convolve it by, for example, a 5-by-5 parameter matrix. This operation extracts some features from the otherwise information-sparse image, to be used by subsequent layers. The convolution layer produces its output by using only 25 parameters instead of the 65 thousand, needed by a fully-connected layer. This approach has multiple advantages when processing images:

- the layer ignores any spatial dependency, which is useful for object detection since we are indifferent to where in the image the object is,
- the use of fixed parameters over the whole image prevents the model from finding features specific to the dataset.[26]

A non-linear function is often used with the convolution operation, to help with extracting non-linear features. Usually, after the convolution layer, a pooling layer follows. Pooling operation condenses the information. It does so by effectively downsampling the data. For example, one of the most common pooling methods splits the data into 2-by-2 submatrices and selects only the maximum value from that matrix. This reduces the size of the data by a factor of 4. After a few convolution-pooling layers a standard feedforward network usually follows to produce the final output of the model.

[3]CNNs can and are used on other types of data, e.g. audio data or time-series data.

2.6 Regularisation

In ML, the goal of regularisation is to reduce the model's error outside the training data. This is usually done at the expense of accuracy on the training data. The problem is if the model fits the training data, which is only a flawed representation of the ground truth, with 100% accuracy. It then cannot generalize well on the data not contained in the training dataset. This problem is called **overfitting** and it is not specific to ANNs, in fact, every ML model experiences this phenomenon to some extent. The rest of this section describes some standard methods of ANN regularisation.

Norm penalties are a way to control the parameter space. By adding the L^1 or the L^2 norm of the parameters, multiplied by a coefficient, to the loss function we can guide the values of the parameters in a certain way. The L^1 norm helps to induce sparseness in the network by driving the parameters either to 0 or a large value. The L^2 norm is more computationally efficient and helps to keep the parameters small.

Dropout is a very popular technique to keep the model from overfitting. It is similar to bagging. When computing the outputs, we remove some hidden units from the network and try to reduce the error of this sub-model. The units are then returned to the model and another set of units is removed. In doing so, dropout trains many sub-models, that when put together resist overfitting by compensating for each other's errors. Dropout also hastens the training process, because we do not need to compute the gradients of the removed units.

Drop connect is similar to dropout. Instead of removing units, this technique selects a set of connections to be removed.

Chapter 3

Proposed model

As discussed in the introduction of this work, the best-performing scheme from Pikna's thesis was the one using the Y'$C_B C_R$ color space. The scheme achieved good results by compressing only the C_B and C_R channels of the image, our goal is to improve the initialization of the algorithm used in this scheme.

To this end, we developed the **NMFNet** model. NMFNet is a neural model whose objective is to produce a good first initialization for the NMF algorithm for the C_B and C_R image channels. Since the model has to be able to process any image, we can safely assume, that a single model can be used for the initialization of both the C_B and the C_R channels. Consider an image with some C_B and C_R channels, there surely exists an image with switched values for the channels and therefore the sets of all possible evaluations of the channels are the same.

▶ **Observation 1.** The best possible initialization of the NMF iterative algorithm is the solution of the decomposition. When the algorithm is initialized this way, no iterations are needed since the solution is already good enough.

If the NMFNet model can approximate the NMF decomposition sufficiently enough, we can then effectively skip a number of iterations of the NMF algorithm, potentially cutting down on the computation time of the decomposition. The ability to approximate the result of the NMF algorithm was the primary focus when designing the model. The selection of training data was essential in achieving the approximation. A detailed explanation of data processing and generation is discussed in Section 3.1 below.

The size of a neural model, i.e. the number of parameters, is an important metric in determining the size of the dataset needed to train the network and the power of the hardware needed to perform the training. With an increasing number of parameters, the size of the dataset and the memory needed to compute the gradients of the network must grow also. For this reason, it is important not to make the model too big, e.g. to have unnecessarily many layers or to have too many connections between the layers. The method for reducing the number of parameters needed in the NMFNet model is described in Section 3.2 below.

3.1 Data modeling

3.1.1 Data aquisition

The images used for the development of the NMFNet model and subsequent validation and testing are a part of the RAISE raw image dataset[27]. The dataset offers a collection of 8156 unprocessed images from 3 different cameras. The images are further categorized into 7 categories: Outdoor Photo, Indoor photo, Landscape Photo, Nature Photo, People Photo, Objects Photo and Buildings Photo. Each photo can belong to multiple categories at once.

The images use the TIFF file format with lossless compression. 3 different resolutions of the images can be found in the dataset, images taken with the Nikon D7000 have a resolution of 4928 by 3264 pixels, images from the Nikon D90 have 4288 by 2848 pixels and images from the Nikon D40 have 3008 by 2000 pixels. The file sizes range from around 5 to 25 MB, with the D40 having the smallest files and the D7000 the largest.

3.1.2 Data preprocessing

First, the dataset was standardized. Two of the images were corrupted and consequently were removed from the dataset. Also, some images were taken in portrait orientation, these images were transposed. When loading each image into memory, a transformation was made. First, the images were resized to 308 by 204 pixels, this was required because processing larger images was computationally challenging. In order to mitigate some overfitting, the images were randomly flipped in both directions. Finally, the images were converted from the RGB color space to the Y'$C_B C_R$ color space, and the pixel values scaled to the interval $[0, 1]$.

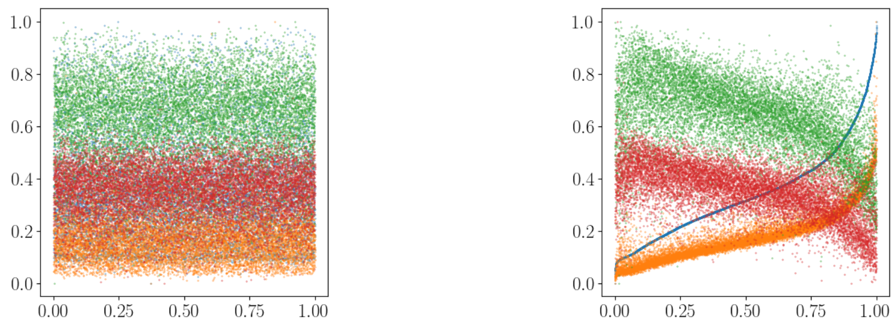

Figure 3.1 Left: data from the decompositions before sorting. Right: data from the decompositions after sorting. It can be observed, that the relationship between the mean and the standard deviation of each of the matrices is linear. The relationship of the values between the two matrices can be observed to be inverse: when the mean and standard deviation of one of the matrices are small, the parameters of the other matrix are large.

3.1.3 Statistical model of factor matrices

To teach the model to approximate the NMF decomposition well, we need to be able to compare the model's output with the real underlying data decomposition, or at the very least, with some good approximation of it. Since the Exact NMF algorithm is NP-hard and real-world data most likely does not have an exact decomposition, it is practically impossible to obtain the underlying decomposition of a real image.

Suppose one is able to randomly generate the W and H matrices. From these artificial factors, one is then able to fabricate an artificial image by multiplying the artificial factors. Images generated this way can consequently be used to train the model and the model's output can be compared directly with the actual factors, from which the images were generated. The factors obtained when decomposing real images using NMF are however not fully random matrices. In this work, we make an assumption, that elements of the factor W come from a normal distribution $\mathcal{N}_W(\mu_W, \sigma_W^2)$ and the elements of the H factor come from a normal distribution $\mathcal{N}_H(\mu_H, \sigma_H^2)$.

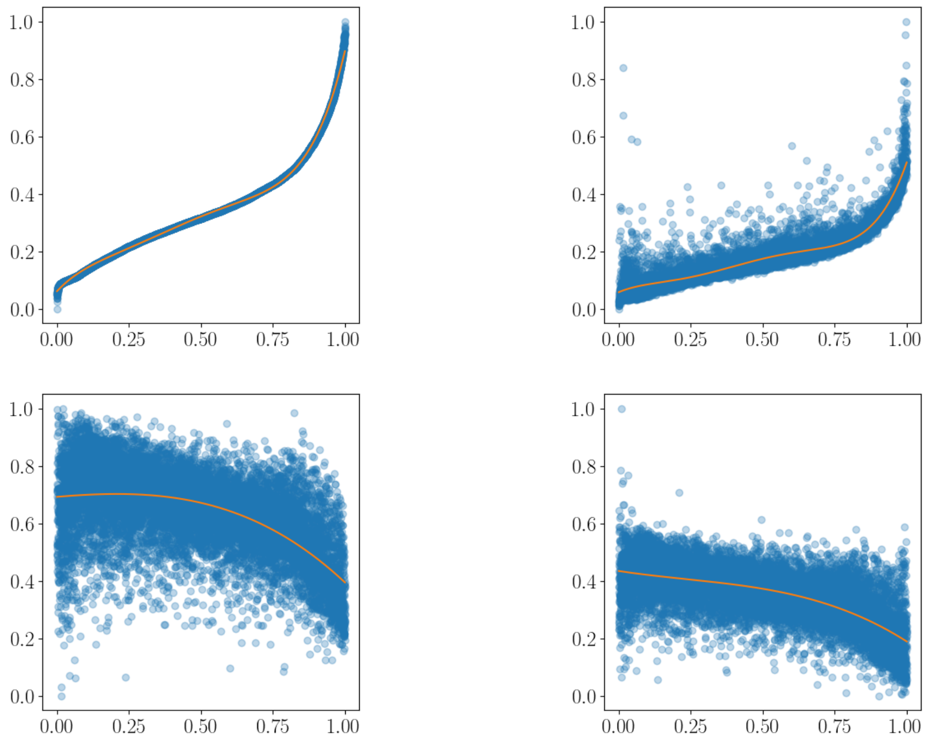

Figure 3.2 Polynomial curves fitted to the mean and standard deviation data of the W and H matrices. Top left: the model for μ_W, top right: the model for σ_W, bottom left: the model for μ_H, bottom right: the model for σ_H.

With this assumption in mind, a statistical model was developed for each of the distributions' parameters: μ_W, σ_W, μ_H and σ_H. This statistical model allows us to emulate factors of real images to a reasonable precision. An NMF decomposition was first performed on all the images from the dataset. The chosen values of parameters of the decomposition can be seen in Table 3.1.

A sample mean and standard deviation were calculated for each factor of each image decomposition, normalized to an interval of $[0, 1]$. Minimum and maximum values of the data are

Algorithm	Rank	Number of iterations
SED-MU	20	400

Table 3.1 The values of parameters chosen for the NMF decompositions.

stored for each parameter, to be able to rescale the parameter back to the original scale. The values are then put into an array as quadruplets. The array was then sorted by the $\widehat{\mu}_W$, to reveal any correlation between the parameters. A strong correlation between the parameters can be observed in Fig. 3.1.

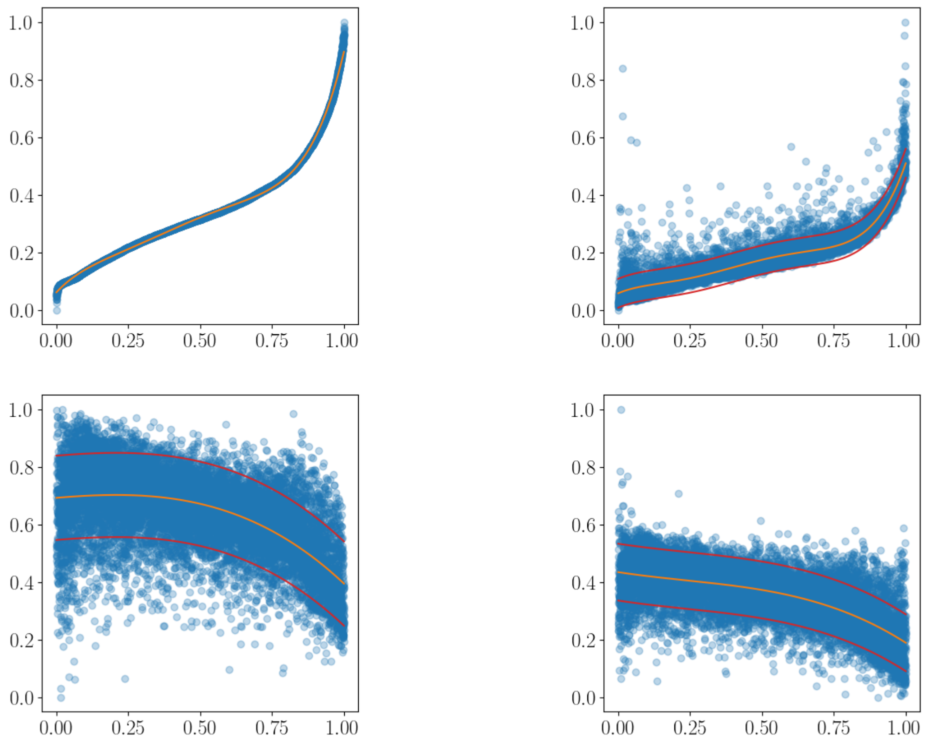

Figure 3.3 The final statistical models for the mean and standard deviation data of the W and H matrices. Top left: the model for μ_W, top right: the model for σ_W, bottom left: the model for μ_H, bottom right: the model for σ_H.

With this data in hand, a model we created for each of the variables. The idea is to generate a random number z in the interval $[0, 1]$, corresponding to the x axis of the right graph in Fig. 3.1 and get corresponding values for our parameters. To this end, a continuous model must be created since we want to be able to assign parameter values to an arbitrary value of z.

A polynomial curve was first fitted to each of the parameter's data. A polynomial of the 5^{th} degree was chosen for the mean and standard deviation of the W matrix and a polynomial of the 3^{rd} degree was chosen for the mean and standard deviation of the H matrix. A standard least squares method was used. The fitted curves can be seen in Fig. 3.2. The fitted curves give us a good middle value of a given parameter for a given z. For our model to be more accurate, a random variance must be added to the value. This is not needed for $\widehat{\mu}_W$, since the data was sorted by this value and therefore it is fully described by fitted curve. For the other parameters, the standard deviation was calculated for the data of each of the parameters, we shall label these values σ_{σ_W}, σ_{μ_H} and σ_{σ_H}.

The variance along the fitted curves is modeled naively, the value of $\sigma_\sigma(\sigma_\mu)$ is multiplied by a correction coefficient $\sigma_\sigma^c(\sigma_\mu^c)$ and a random value from the interval $[-1, 1]$. The correction coefficient was introduced to adjust the amplitude of the variance. The values of the correction coefficient for each parameter were chosen thusly:

$$\sigma_{\mu_W}^c := 0, \quad \sigma_{\sigma_W}^c := 0.5, \quad \sigma_{\mu_H}^c := 1, \quad \sigma_{\sigma_H}^c := 1.$$

A visualization of the final models can be seen in Fig. 3.3.

After the base parameter value is generated, the value is rescaled back into the original scale, using the original minimum and maximum parameter values. Algorithm 2 summarizes the process of generating a parameter value from the seed value z.

Algorithm 2: Generate the value of parameter α

```
  Input  : z ∈ $[0, 1]$
  Input  : min ∈ $\mathbb{R}^{+0}$
  Input  : max ∈ $\mathbb{R}^{+0}$
  Input  : $\sigma_\alpha \in \mathbb{R}^{+0}$
  Input  : $\sigma_\alpha^c \in \mathbb{R}^{+0}$
  Input  : p_model ∈ $[0, 1] \to [0, 1]$
  Output: Value of the parameter
   // Generate a random number
1 variance ← random([-1, 1])
   // Find the middle value corresponding to z
2 value ← p_model($z$)
   // Add the random variance to the value
3 value ← value + variance $* \sigma_\alpha^c * \sigma_\alpha$
   // Rescale the value
4 value ← value $* (max - min)$
5 value ← value $+ min$
6 return value
```

3.2 Description of the model

The NMFNet model consists of two neural networks, called WNet and HNet. The goal of these submodels is to predict the W or the H matrix respectively, from the image data. The architecture of these models is the same except for the layer widths. The WNet model processes the image horizontally, and so its width is determined by the width of the image, and vice versa, the HNet model processes the image vertically and so its size is determined by the image height. Considering the submodels are identical, save the width of the layers, only the WNet model will be described in the rest of the section, the HNet model simply has the input image matrix transposed.

The WNet model consists of two main components: the sliding window and the semi-autoencoder. The sliding window is similar in function to the kernel in CNNs, it reduces the number of parameters needed to process the input by sharing the values of parameters. The semi-autoencoder part of the model takes the data from the sliding window and produces a single row of the W matrix. The sliding window and the semi-autoencoder are described in more detail in the subsections below.

3.2.1 Sliding window

▶ **Observation 2.** When reconstructing the original matrix V by multiplying the factor matrices, the i-th row of the W matrix affects only the i-th row of the approximation, and vice versa, the j-th column of the H matrix affects only the j-th column.

The sliding window was introduced to exploit Observation 2 from the beginning of this chapter. By focusing only on a part of the image, we can produce the W matrix by parts, as opposed to trying to solve the whole decomposition at once. The window selects a number of adjacent rows from the image and passes to the semi-autoencoder, to be reduced into a single row of the W matrix. The number of selected rows is called the **window width**. The window width must always be a positive odd integer.

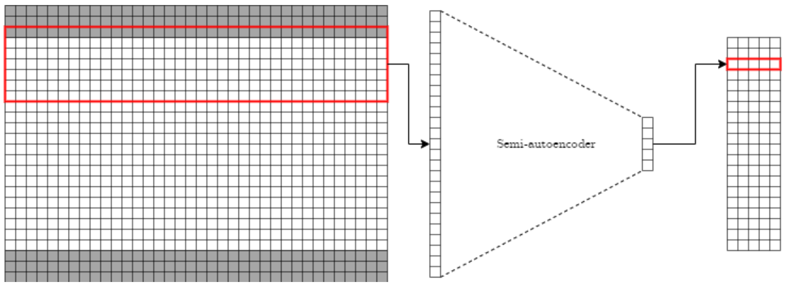

Figure 3.4 Example of the sliding window part. On the left, the image data is represented by the white grid and the sliding window is illustrated by the red rectangle. In the example, the width of the sliding window is chosen to be 7, using the formula below, we calculate $k = 3$, and so padding wide 3 pixels is added to the top and bottom of the image data, represented by the gray grids. Data from the window is then flattened to a vector and processed by the semi-autoencoder, which produces a single row of the W matrix. The window starts at the top of the data and moves down one pixel in each step. In the example, 3^{rd} step of the window can be seen, which produces 3^{rd} row of the W matrix, highlighted by the red rectangle on the right.

To produce all n rows of the W matrix, the window has to be able to make n steps, therefore, padding rows must be introduced when the window width is larger than 1. Zero padding was used in our model. k rows are added to both the top and the bottom of the image.

$$k = \frac{(width - 1)}{2}.$$

The window serves a dual purpose. Firstly, it reduces the number of parameters needed to process the image. The number of parameters in a neural model, that would try to process the whole image at once, without some means of reducing the number of parameters, would make it infeasible to train the network in a reasonable time even on today's cutting-edge hardware. Secondly, it simulates the locality of the original NMF decomposition.

3.2.2 Semi-autoencoder

The semi-autoencoder part of the WNet performs the actual encoding of a part of the image, received from the sliding window, into a row of W. The semi-autoencoder is simply the encoder part of the autoencoder model discussed in Section 2.5.1. We need just the encoder because the model is required to produce only the encoding. In our model, the decoder is substituted by the multiplication of the W and H factors.

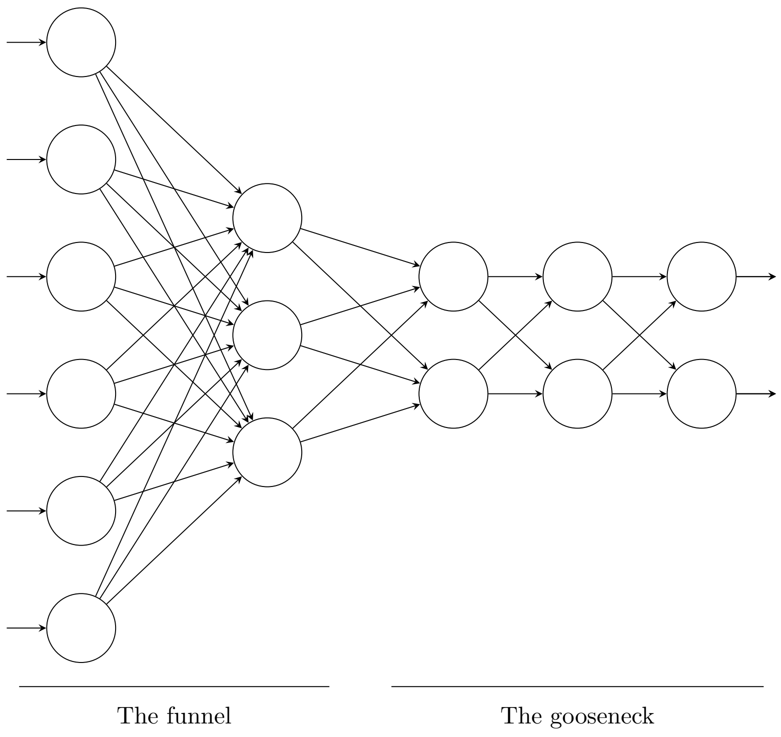

Figure 3.5 The semi-autoencoder takes the flattened data from the sliding window and reduces it down, to produce a single row of W. In the example, the funnel consists of the leftmost 2 layers, the reduction factor is set to 2, hence the width of the second layer is a half of the first. The gooseneck consists of the 3 layers on the right, ergo the gooseneck depth is 3. The width of the input layer is dictated by the number of elements in the sliding window, and the width of the output layer is the number r, in this case, $r = 2$.

The semi-autoencoder used in our model has two parts: the funnel and the gooseneck. The function of the funnel is to progressively reduce the width of the layers, down to width r, which is the final width. It does so with exponential decay of the layer width, each layer is n times smaller than the one before. n is called the **funnel reduction factor** and is a hyperparameter of the model. When the reduction would produce a layer with a width less than r, the width of the layer is instead set to r.

The gooseneck of the semi-autoencoder directly follows the funnel. The architecture of this part of the semi-autoencoder is very simple, the gooseneck is just a series of densely connected layers with width r. The number of layers in the gooseneck is called the **gooseneck depth** and is a hyperparameter of the model. The semi-autoencoder must always have a gooseneck depth of at least 1.

All the layers in the autoencoder except the last use the LeakyReLU activation function.

LeakyReLU is an augmentation of the classic ReLU function, the difference being, instead of setting all negative values to zero, it uses the following formula:

$$\text{LeakyReLU}(x) = \begin{cases} x, & \text{if } x > 0. \\ 0.01x, & \text{otherwise.} \end{cases}$$

This solves the ”dead ReLU” problem by introducing a small gradient for the negative values.

When training the model, the output layer of the model uses the Linear Unit activation function. This is desirable because when the model produces a negative number in one of the outputs, it can correct that behavior since the gradient is non-zero, if the ReLU function would be used in training, the neuron would become dead. During inference, however, the output layer’s gradient is inconsequential and so the ReLU activation function is used, to prevent the model from producing factors with negative elements.

Chapter 4

Implementation

The training loop of the NMFNet model followed the standard training scheme: the training was done in several epochs, after each epoch a validation of the model was performed; each epoch was comprised of a number of training steps; in each step, the model was trained on a batch of data and a gradient descent step was performed.

Normally, the validation after each epoch is performed on the whole validation dataset, for the sake of expediting the learning process, the validation was only done on a batch of data, from the validation dataset. In the table below we list the values chosen for our training loop, the values were determined empirically during the development of our model.

Number of epochs	Number of steps in an epoch	Training batch size	Validation batch size
15	100	32	64

Table 4.1 Values of the training loop hyperparameters.

The Adam optimizer was chosen for the training since it is the best-performing optimizer today. The optimizer has a number of parameters, which need to be set. We have chosen to mostly adhere to the values recommended in the PyTorch framework, however, the learning rate was lowered and a small weight decay was added. The values chosen are listed in the table below.

Learning rate	β_1	β_2	Weight decay
$1 \cdot 10^{-4}$	0.9	0.999	$1 \cdot 10^{-5}$

Table 4.2 The values of the optimizer hyperparameters.

4.1 Chosen technologies

4.1.1 Programming language

The Python programing language was chosen for our implementation. It is the most popular programming language used in machine learning today.[28] Reasons for Pythons's large following are many: the simple syntax and enforcement of indentation in the code lead to very readable code, the weak type system is great for fast prototype development, a rich ecosystem of libraries and frameworks allows for most problems to be solved in just a few lines of code and thanks to the large community of Python users documentation is plentiful and troubleshooting fast.

The greatest weakness of Python lies in its slow runtime. A simple nested iteration can take multiple times longer in Python when compared to the same implementation in more performance-minded languages like C or Go.[29] In the machine learning domain, where the training of a fully performance-optimized model can take weeks or even months, this property of Python would be quite problematic. The lackluster performance of Python is circumvented by the use of libraries written in performance-minded languages, so all the critical computation takes place outside the Python interpreter, and Python itself serves only as a sort of orchestrator of these high-performance libraries.

Other options were considered, namely C++, Scala and Julia, which are also popular among ML experts. Pyhon was however chosen over these languages primarily for its mentioned ease of use. The subjective preference of the language and the expansive ecosystem of tools and libraries were also major factors.

4.1.2 Jupyter notebooks

In the workflow of an ML expert, it is important to be able to visualize and annotate the data as one is processing it, Jupyter is an application that provides this ability.

Jupyter works with Jupyter notebooks. A notebook is a file, that is divided into cells. A cell can either contain code or simple text, Jupyter can then execute these cells using a persistent environment, called a kernel. The role of the kernel is to interpret the code in the notebook's cells and to hold the execution context in memory, so cells can access variables and functions defined in other cells.

The advantage of this approach is one must not evaluate the entire program from the beginning each time a change is made to the code. When working with a large amount of data, which can take minutes or hours to be processed, it is beneficial to load the data into the memory once, and then work with it inside the persistent environment of the Jupyter kernel. Jupyter notebooks also provide the ability to describe the data in the text cells or to visualize the data directly inside the notebook, thanks to the ability to show images and plots.

4.1.3 Neural network framework

"Usability over performance" is the first design principle of ML framework called PyTorch. The framework leads the user to be very explicit with the code they write, making it easy to debug.

Even though performance is secondary to the PyTorch development team, the framework still has reasonable performance and can work with hardware accelerators and scale to many machines.[30] Pytorch is currently the most popular framework among researchers. In recent years, the number of papers published with PyTorch as their framework of choice has risen sharply, as compared to TensorFlow, this phenomenon is attributed to PyTorch's ease-of use.[31]

Another option for our implementation is the TensorFlow framework. It was developed by Google LLC and it is the most popular ML framework used today for machine learning.[32] The first implementation attempts of our model were done using the TensorFlow framework, but unfortunately, some technical issues were encountered when training on multiple graphical accelerators, which we were not able to resolve promptly. The implementation was then moved to the PyTorch framework and its less complex and more beginner-friendly structure allowed us to complete the development.

4.1.4 Supporting libraries

This subsection contains a list of libraries used in the development, training or evaluation of the NMFNet model.

Numpy is a mathematical library implemented in C. It provides a simple application interface for working with vectors and matrices in Python while having great performance.

Pillow is a library for working with images in Python. It can load, transform and convert images in many common formats. In the implementation, it was mainly used for preprocessing the dataset.

Nimfa is a matrix decomposition library. It implements common NMF algorithms and initialization techniques. In the implementation, the library was used for the development of the statistical model of the data and the evaluation of the final model.

Scikit Learn is an ML library, which implements many standard ML models such as Decision trees, Ridge regression and many others. In the implementation, the library was used for the K-means initialization of the NMF.

Pandas focuses on making the processing of tabular data easy and intuitive. The library was mainly used in the evaluation phase.

4.2 Training infrastructure

The development, training and evaluation of the NMFNet model were carried out on a system equipped with Intel® Xeon® Gold 6254 processor with a total of 18 physical cores and 36 threads, 128 GB of DDR4 ECC memory and two Nvidia A100 40 GB graphical accelerators. Access to this system was granted to us by the Faculty of Information Technology CTU in Prague.

4.3 Hyperparameter tuning

A grid search approach was chosen for tuning the hyperparameter values. In grid search, a number of hyperparameters were chosen together with ranges of values for each of the hyperparameters. For each combination of these hyperparameters, a model was then trained and the best hyperparameter combination was determined. 4 hyperparameters were chosen in our training, they are listed together with their chosen values below:

Hyperparameter	Values
Loss function	MAE, MSE
Sliding window width	1, 3, 5, 7, 9, 11, 13, 15
Funnel reduction factor	2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 5.5, 6.0, 6.5, 7.0, 7.5, 8.0, 8.5, 9.0, 9.5, 10.0
Gooseneck depth	1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Table 4.3 Hyperparameters and their values chosen for the grid search

The choice of the ranges of values for each of the hyperparameters was informed by preliminary testing during the development of the model. The number of combinations of the hyperparameter values is 2720, for each of these combinations a model was trained. In order to reduce the real-world training time, the set of all hyperparameter combinations was split into 12 disjunct subsets, 12 models were then trained concurrently, each on one of the subsets.

During the training process, a number of metrics were recorded. For every training step, the loss of the step was tracked. At the beginning of the training process, an image was randomly chosen and saved as a reference, after each training epoch has ended, the reference image was decomposed using the NMFNet model. The image was then reconstructed and the result was saved. These sample images, taken during the training were invaluable in determining the quality of the decomposition. At the end of every epoch, the validation loss and SSIM metrics were also tracked.

Here we list some statistics about the training time and the sizes of the models trained.

Time statistics	
Total real-world training duration	4 days and 8 hours
Total computational time	52 days and 8 hours
Average model training time	27 minutes and 42 seconds
Models trained per hour	26 models per hour
Minimum model training time	4 minutes and 57 seconds
Maximum model training time	1 hour and 9 minutes

Model size statistics	
Minimum number of parameters	13 920
Maximum number of parameters	20 476 806
Average number of parameters	2 665 834
Median number of parameters	1 622 014

Table 4.4 Grid search result statistics

The progress of every model during training followed roughly the same path. The models started with a relatively high loss but quickly improved until a loss function floor was reached. The models usually reached the loss floor in 200 to 600 steps. This behavior produced an easily discernable ”elbow” in the graph of the training loss of each model. It was observed that the models usually exhibited the best subjective performance around the elbow of the graph and started to overfit from that point on.

An example of the typical progress of the training of a model is illustrated in Fig. 4.1.

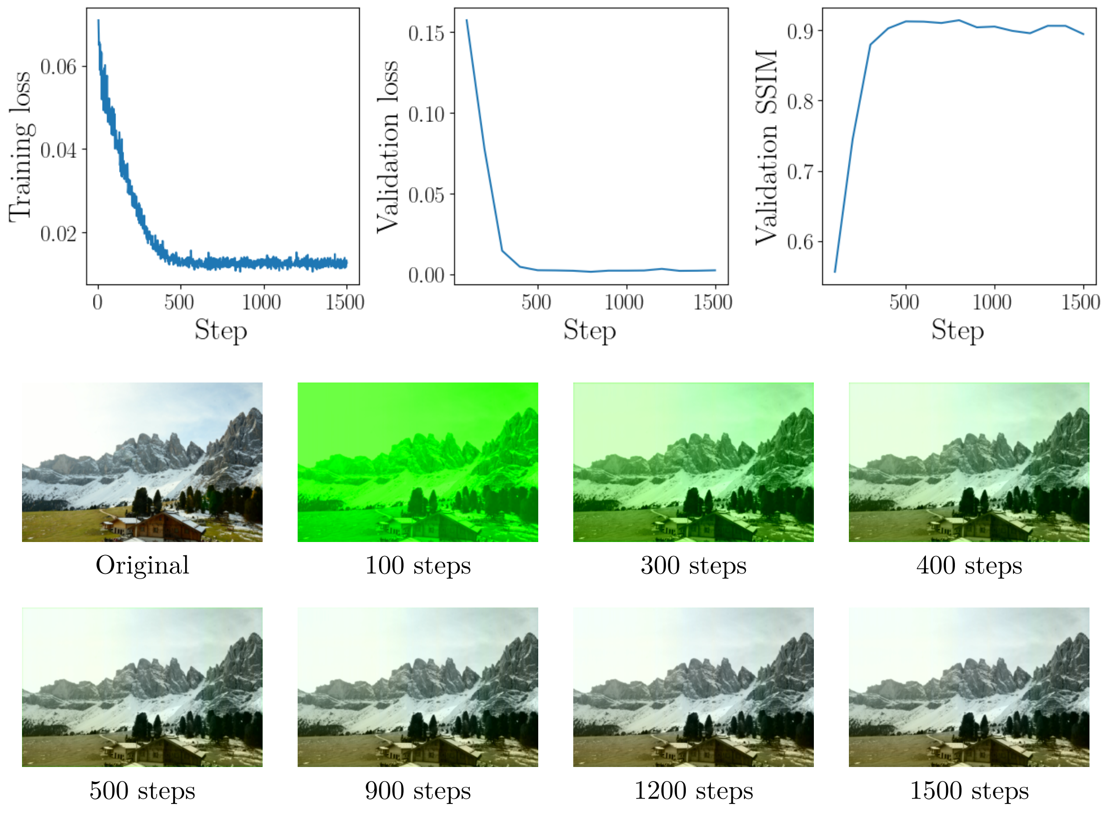

Figure 4.1 Top: three graphs showing the values of metrics in each step of the training. Bottom: a sample of the images generated during each validation step. During training, each model typically reached some local minimum, in the example, this happened around step 500, which creates an ”elbow” in the graph. From the sample images can be seen that the model doesn’t really improve from that point on, some models even started overfitting, which was indicated by a purple tint.

Chapter 5

Evaluation

5.1 Choosing best hyperparameter configuration

After the grid search concluded an analysis was performed on the data collected during the training. The main goal of this analysis was to determine if any of the hyperparameters explored in the hyperparameter tuning had a significant effect on the quality of the decomposition.

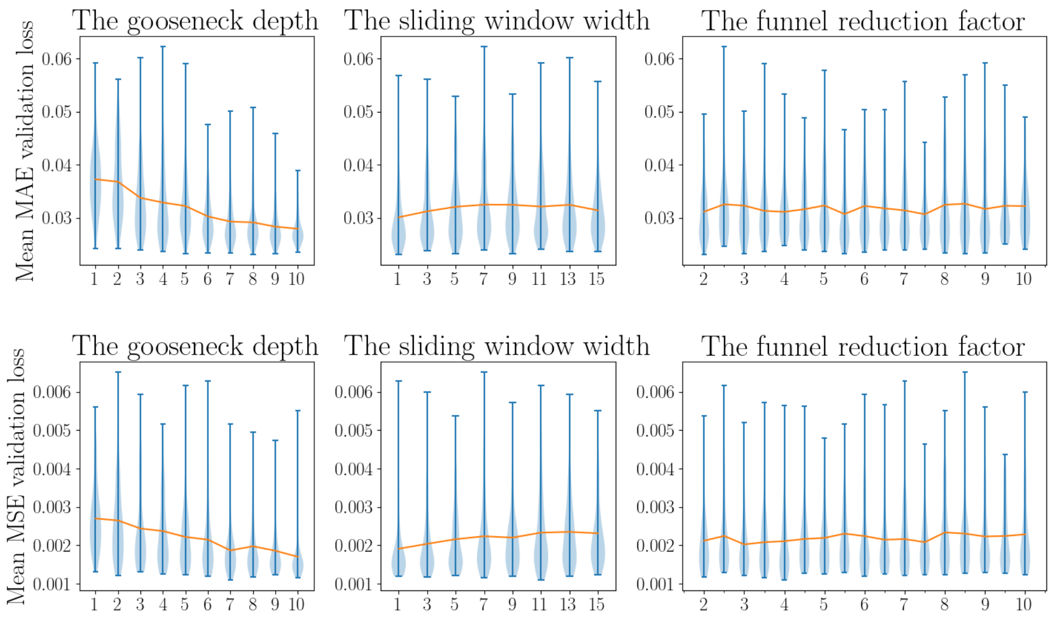

Figure 5.1 Top 3 charts: the results for the MAE loss function. Bottom 3 charts: results for the MSE loss function. On the x-axis of each chart, the values for each hyperparameter are marked, and the y-axis corresponds to the maximum value of the loss function reached during the training. Each of the blue violins indicates the distribution of the maximal loss function values for each choice of the hyperparameter value. The orange line connects the mean value of the loss for each hyperparameter value.

The data was first separated based on the loss function used in the training of the model since the absolute value of the validation loss is incomparable between the MSE and the MAE. Next, the data were aggregated by the values of each of the other hyperparameters: the sliding window width, the funnel reduction factor and the gooseneck depth. Correction of the hyperparameter values was examined for both the validation loss and the validation SSIM, however, no significant correlation was discovered between the hyperparameter values and the validation SSIM values and therefore will not be discussed.

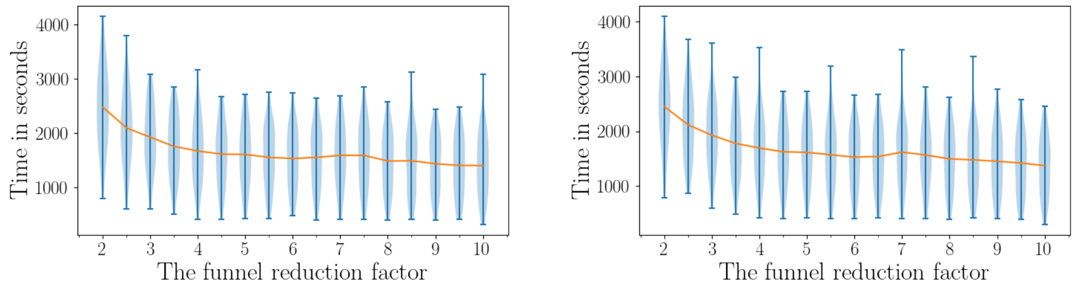

Figure 5.2 Left: the effect of the funnel reduction factor on training time for the MAE loss function. Right: the effect of the funnel reduction factor on training time for the MSE loss function. The width of the blue violins describes the distribution of the values for each of the hyperparameter values. The orange line connects the mean of each violin.

Analysis of the validation loss revealed a number of insights. A strong correlation between the gooseneck depth and the loss value was identified, a larger number of layers in the gooseneck corresponds to a smaller loss. A less significant correlation was found for the sliding window width. Interestingly the lower loss was achieved when the width of the window was small, contrary to our hypothesis. A wider window was expected to produce a lower loss. No real link between the funnel reduction factor and the validation loss achieved. To identify a good value for this hyperparameter, we examined its effect on the training time of the model.

With an increasing funnel reduction factor, the training time of a model decreased. The best value of the funnel reduction factor was chosen to be 7.5 since the value had good performance in reducing both the loss function and the training time. Performance of the MAE and MSE loss functions was then compared. As stated above, the absolute values of these functions are incomparable, and so the validation SSIM was chosen as the metric for determining, which loss functions yielded better results. The SSIM values for each of the functions were again aggregated.

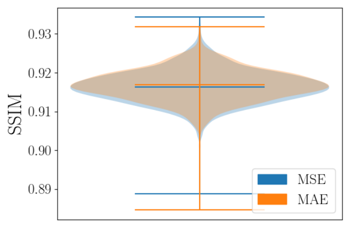

Figure 5.3 The two loss functions performed nearly identically. The top horizontal line is the maximum SSIM achieved, and the bottom horizontal line is the minimum. The middle line is the modus of the values for each loss function. The width of the violins describes the distributions of the values.

The difference in the performance of the two loss functions was found to be minuscule, but the MAE loss was found to be slightly better.

After the analysis of the data collected during the hyperparameter tuning, the best-performing values of the hyperparameters were chosen for the model used for comparison with the other initialization methods. The best found hyperparameter configuration is presented in the table below.

Loss function	The sliding window width	The funnel reduction factor	The gooseneck depth
MAE	1	7.5	10

Table 5.1 The best-found hyperparameter configuration.

Ten models with this configuration were then trained, with the training parameters slightly tweaked. The number of epochs was increased to 80 and the number of steps in each epoch was reduced to 20, in total each model was trained in 1600 steps. The model state was also saved after each epoch in addition to the validation, this allowed us to pick the best possible parameter configuration based on the subjective analysis of the images generated in each validation step. The training loss, validation loss and validation SSIM together with the images generated in the validation steps during training of the model chosen, for the evaluation can be seen in Fig. 5.4.

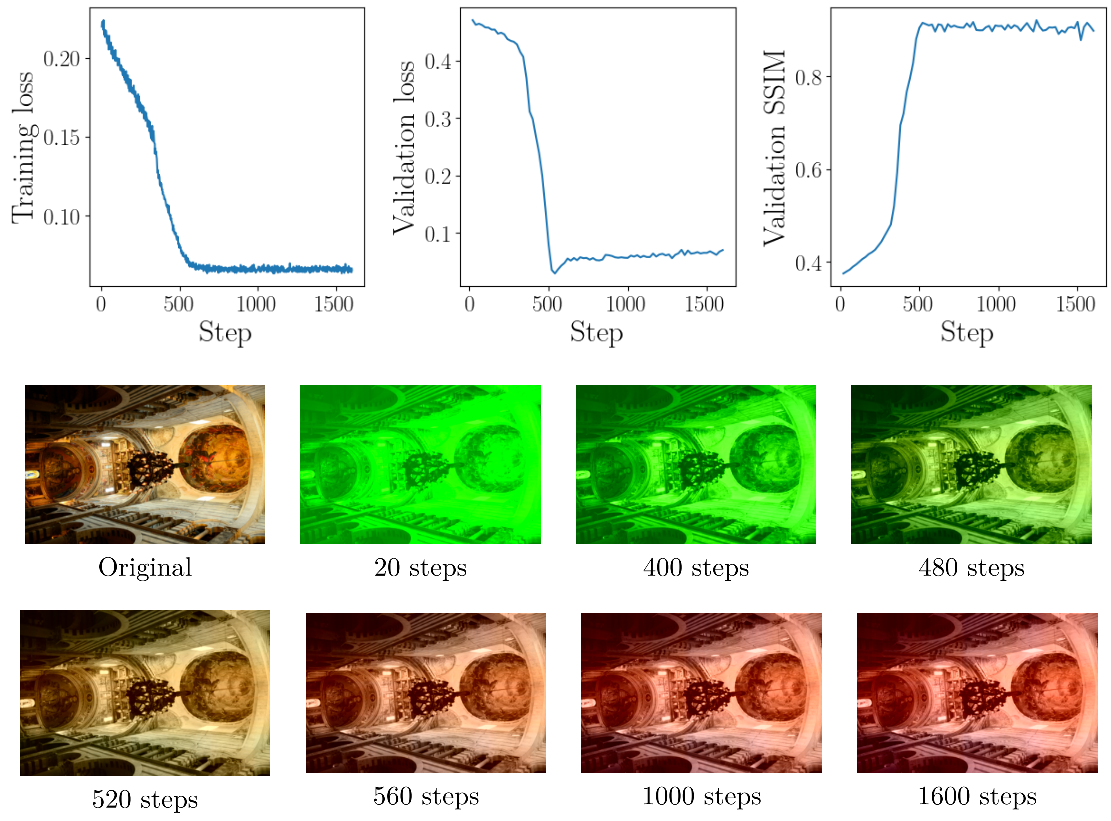

Figure 5.4 Top: three graphs showing the values of metrics in each step of the training. Bottom: a sample of the images generated during each validation step. A sharp elbow can be seen at step 520, in this step, the sample image also showed a great similarity to the original picture. After the model started to overfit, as indicated by the purple tint.

5.2 Comparison with other NMF initialization methods

Three other traditional NMF initialization methods were chosen for the evaluation of our trained NMFNet model. The random and NNDSVD initialization methods were used in Pikna's work and so they were chosen for the comparison. Additionally, the K-means initialization was chosen as a representative of the more sophisticated methods.

A random sample of 100 images was chosen for the evaluation. Each of the images was decomposed using the SED-MU algorithm with the various initializations listed above. The number of iterations was set to 150. For each iteration of the algorithm, the MSE, PSNR and SSIM metrics were tracked, also an image was generated from the decomposition, to subjectively track the quality of the decomposition. The results of the 100 decompositions were then averaged and analyzed.

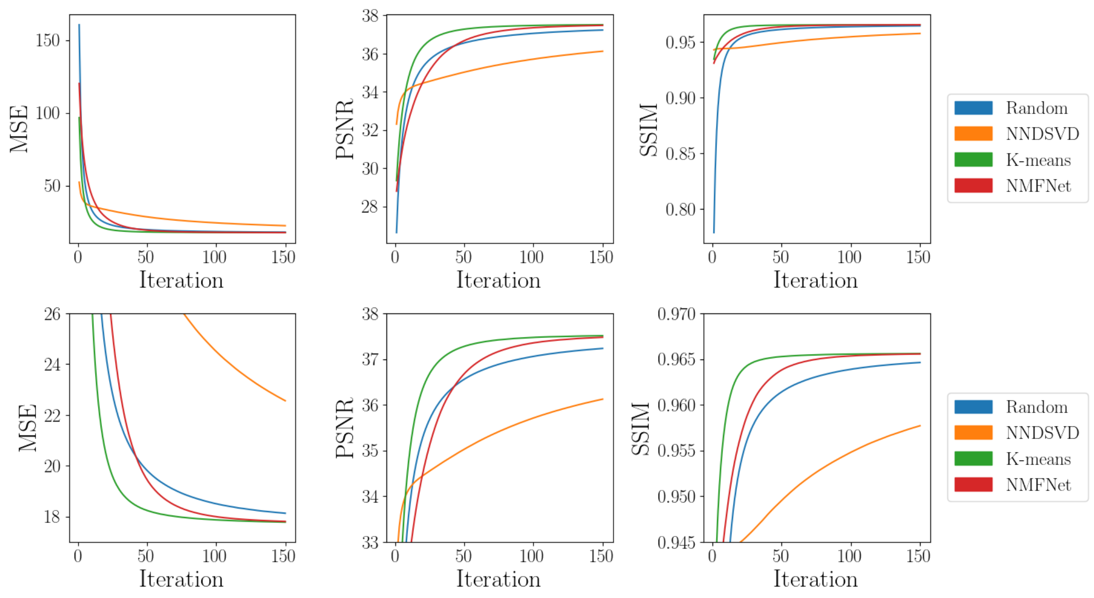

Figure 5.5 Top: graphs showing the averaged progress of all the initialization methods. Bottom: same as the top graphs, but with a different scale. For the graph showing the MSE metric, the lower value is better. For the PSNR and SSIM metrics, the higher value is better.

Fig. 5.5 summarises this analysis. We evaluated the initialization methods based on 3 criteria: absolute performance, speed of convergence and the quality of the initialization compared to the random initialization.

The absolute performance criterion compares the maximum value of each of the metrics reached during the decomposition. The best-performing initialization methods according to absolute performance are the K-means initialization and the NMFNet. Of the two, the K-means method performed slightly better. The next best was the random initialization, its performance was close to the K-means and NMFNet methods, but always strictly lower. The NNDSVD method suffers from very slow convergence and fails to catch up to the other methods.

The absolute performance of the NMFNet model compared to the other methods is also illustrated by Table 5.2. As can be seen, the model performed strictly better than the random and NNDSVD initialization methods most of the time. K-means initialization had a slight edge in the performance compared to the NMFNet but performed the same in most cases.

	Radnom	NNDSVD	K-means
MSE and PSNR			
Strictly better	97%	100%	98%
Within 0.001% or better	97%	100%	99%
SSIM			
Strictly better	90%	100%	25%
Within 0.001% or better	100%	100%	97%

Table 5.2 Percentage of the evaluation dataset on which the NMFNet model performed either strictly better than or within 0.001% of a given method. The top table shows performance for the MSE and PSNR metrics, the bottom table shows performance for the SSIM metric.

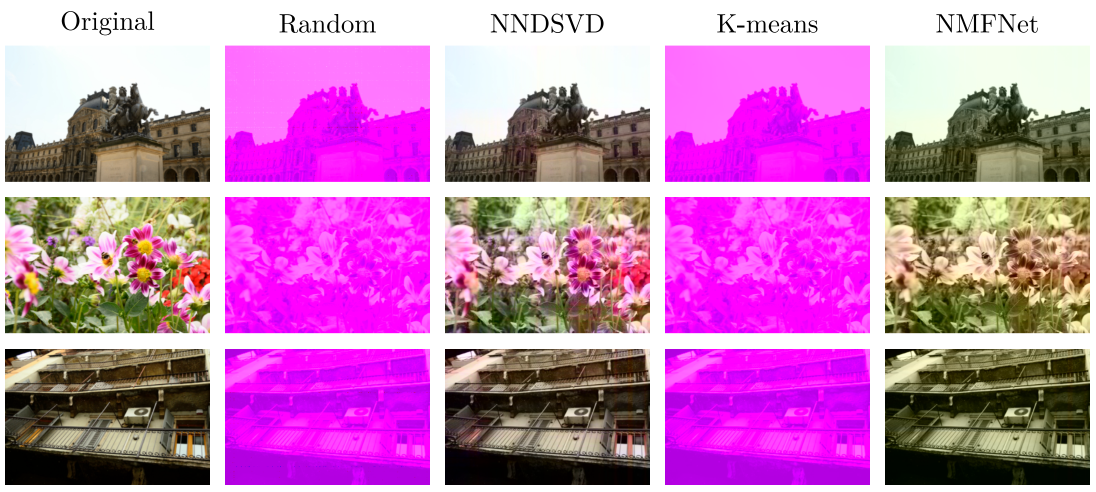

Figure 5.6 The images showed were reconstructed by multiplying the initial settings of the W and H matrices for each of the initialization methods.

The random initialization converges fastest, for the MSE and PSNR metrics it even outperforms the NMFNet initialization for a brief period. However, this is mainly because the initial setting of the decomposition is of poor quality compared to the other methods. Since the other methods generally initialize the MU algorithm close to the final solution, the gradient of the error is mild, and the rate of convergence is less. The K-means method also converges rapidly to the final solution. Although the random initialization converges faster at the start, the K-means initialization is never outperformed by it. The convergence of the NMFNet is also good, and after 60 iterations the performance generally reaches the K-means. NNDSVD converges by far the slowest and requires a very large number of iterations to reach the same performance as the other methods.

The quality of the initial setting of the NMF algorithm was determined by subjective analysis of images reconstructed from only the initial settings of the W and H matrices. A sample of the images can be seen in Fig. 5.6. The random and K-means methods did not approximate the image well. The purple color means, these values of the $\mathrm{C_B}$ and the $\mathrm{C_R}$ are large, near the upper limit allowed for the values. The NNDSVD initialization is clearly the best of the methods chosen. It preserves the colors of the images very well, even the small details. The NMFNet initialization produces a good approximation of the original image, however, the colors of the images are washed out and have a slight green tint.

The number of iterations of the Random initialization needed to improve to the level of each of the other methods' initial MSE, PSNR and SSIM values was also examined. It took on average 11 steps for the random initialization to reach the initial SSIM value of the NNDSVD initialization and on average 6 steps to reach the MSE and PSNR values. The NMFNet and the K-means initializations performed the same, on average 8 steps of the random initialization were needed to reach the SSIM value, and 2 to reach the MSE and PSNR values.

Conclusion and future work

In this work, the theory behind the Nonnegative Matrix Factorization was explored together with an examination of the state-of-the-art initialization techniques. The basics of Artificial Neural Network models were also surveyed and different components of the models described. After establishing the theoretical base, the NMFNet neural model for the initialization of the Nonnegative Matrix Factorization-based image compression scheme was developed, implemented, trained and evaluated. The model was found to perform well across multiple criteria. In 97 of the 100 decompositions evaluated the NMFNet model achieved SSIM performance within 0.001% or better than the K-means initialization, which was the best-performing method evaluated. Compared to random initialization, the NMFNet SSIM performance was strictly better 90% of the time and was within 0.001% or better in 100% of the cases. Furthermore, our model was able to skip on average 8 iterations of the NMF iterative algorithm compared to random initialization.

Our model finds a good initial setting by approximating the NMF decomposition. We believe that in the future our model could be generalized and be able to initialize the NMF iterative algorithms for any data. It also could be possible to replace the iterative algorithm outright.

Future work

The NMFNet model was shown to have good performance across the three criteria tested. We believe the results of the model can be improved with further investigation. For example, the NMFNet initialization was very slow, it took an average of 0.73 seconds, which is about the same as the K-means initialization. In comparison, the random initialization took an average of 0.026 seconds and the NNDSVD 0.12 seconds. After closer examination, a strong correlation was found between the number of layers the NMFNet model has and the inference time. With a hyperparameter configuration taking into account the inference time of the model, we could improve the initialization time while still having similar performance.

The model also has problems, when processing larger images. This issue can be possibly solved by inserting convolution and pooling layers between the sliding window and the semi-autoencoder, to further reduce the number of parameters of the model. Another option is to replace the sliding window altogether with a CNN.

More parameter configurations may too be explored. The model has shown increasing performance when increasing the gooseneck depth, it would be interesting the performance with a depth greater than 10. Increasing the number of layers is however in direct conflict with reducing the inference time and so may not be beneficial.

Also, more values of the parameter r should be explored. The performance of the NMFNet model may change when reducing r in comparison with the other initialization methods. In this work, the only value explored is 20, the reason being for each value of r, a new statistical model of the data must be developed, since the distributions of the W and H matrices change with it. The development of these models was time-consuming and since it was not the focus of this work, only one model was developed.

Bibliography

1. PIKNA, Marek. Evaluating performance of an image compression scheme based on non-negative matrix factorization. *Czech Technical Universiry Digital Library*. 2019. Available also from: `http://hdl.handle.net/10467/82324`.
2. LEE, Daniel D.; SEUNG, H. Sebastian. Learning the parts of objects by non-negative matrix factorization. *Nature*. 1999, vol. 401, no. 6755, pp. 788–791. Available from DOI: `10.1038/44565`.
3. LEE, Daniel; SEUNG, H. Sebastian. Algorithms for Non-negative Matrix Factorization. In: LEEN, T.; DIETTERICH, T.; TRESP, V. (eds.). *Advances in Neural Information Processing Systems*. MIT Press, 2000, vol. 13. Available also from: `https://proceedings.neurips.cc/paper/2000/file/f9d1152547c0bde01830b7e8bd60024c-Paper.pdf`.
4. WANG, Yu-Xiong; ZHANG, Yu-Jin. Nonnegative Matrix Factorization: A Comprehensive Review. *IEEE Transactions on Knowledge and Data Engineering*. 2013, vol. 25, no. 6, pp. 1336–1353. Available from DOI: `10.1109/TKDE.2012.51`.
5. DONOHO, David; STODDEN, Victoria. When Does Non-Negative Matrix Factorization Give Correct Decomposition into Parts? *Advances in neural information processing systems*. 2004, vol. 16.
6. VAVASIS, Stephen A. On the complexity of nonnegative matrix factorization. *SIAM Journal on Optimization*. 2010, vol. 20, no. 3, pp. 1364–1377. Available from DOI: `10.1137/070709967`.
7. HAFSHEJANI, Sajad Fathi; MOABERFARD, Zahra. Initialization for non-negative matrix factorization: a comprehensive review. *International Journal of Data Science and Analytics*. 2022. Available from DOI: `10.1007/s41060-022-00370-9`.
8. GONZALEZ, Edward F.; ZHANG, Yin. *Accelerating the Lee-Seung Algorithm for Nonnegative Matrix Factorization*. 2005. Tech. rep. Department of Computational and Applied Mathematics Rice University. Available also from: `https://hdl.handle.net/1911/102034`.
9. LIN, Chih-Jen. On the Convergence of Multiplicative Update Algorithms for Nonnegative Matrix Factorization. *IEEE Transactions on Neural Networks*. 2007, vol. 18, no. 6, pp. 1589–1596. Available from DOI: `10.1109/TNN.2007.895831`.

10. ZDUNEK, Rafal; CICHOCKI, Andrzej. Non-negative Matrix Factorization with Quasi-Newton Optimization. In: RUTKOWSKI, Leszek; TADEUSIEWICZ, Ryszard; ZADEH, Lotfi A.; ŻURADA, Jacek M. (eds.). *Artificial Intelligence and Soft Computing – ICAISC 2006*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 870–879. ISBN 978-3-540-35750-6.

11. DING, Chris; HE, Xiaofeng; SIMON, Horst D. On the equivalence of nonnegative matrix factorization and spectral clustering. In: *Proceedings of the 2005 SIAM international conference on data mining*. 2005, pp. 606–610.

12. ESPOSITO, Flavia. A Review on Initialization Methods for Nonnegative Matrix Factorization: Towards Omics Data Experiments. *Mathematics*. 2021, vol. 9, no. 9. ISSN 2227-7390. Available from DOI: 10.3390/math9091006.

13. LANGVILLE, Amy Nicole; MEYER, Carl Dean; ALBRIGHT, Russell; COX, James; DULING, David. Algorithms, Initializations, and Convergence for the Nonnegative Matrix Factorization. *CoRR*. 2014, vol. abs/1407.7299. Available from arXiv: 1407.7299.

14. SANDLER, Mark. On the Use of Linear Programming for Unsupervised Text Classification. In: *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*. Chicago, Illinois, USA: Association for Computing Machinery, 2005, pp. 256–264. KDD '05. ISBN 159593135X. Available from DOI: 10.1145/1081870.1081901.

15. ZHENG, Zhonglong; YANG, Jie; ZHU, Yitan. Initialization enhancer for non-negative matrix factorization. *Engineering Applications of Artificial Intelligence*. 2007, vol. 20, no. 1, pp. 101–110. ISSN 0952-1976. Available from DOI: https://doi.org/10.1016/j.engappai.2006.03.001.

16. LEE, Tai Sing. Image representation using 2D Gabor wavelets. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. 1996, vol. 18, no. 10, pp. 959–971. Available from DOI: 10.1109/34.541406.

17. KIM, Yong-Deok; CHOI, Seungjin. A Method of Initialization for Nonnegative Matrix Factorization. In: *2007 IEEE International Conference on Acoustics, Speech and Signal Processing - ICASSP '07*. 2007, vol. 2, pp. II-537-II–540. Available from DOI: 10.1109/ICASSP.2007.366291.

18. BOUTSIDIS, C.; GALLOPOULOS, E. SVD based initialization: A head start for nonnegative matrix factorization. *Pattern Recognition*. 2008, vol. 41, no. 4, pp. 1350–1362. ISSN 0031-3203. Available from DOI: https://doi.org/10.1016/j.patcog.2007.09.010.

19. ROSENBLATT, Frank. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*. 1958, vol. 65 6, pp. 386–408.

20. ROSENBLATT, Frank. Perceptron Simulation Experiments. *Proceedings of the IRE*. 1960, vol. 48, no. 3, pp. 301–309. Available from DOI: 10.1109/JRPROC.1960.287598.

21. LAGERCRANTZ, Hugo. The emergence of consciousness: Science and ethics. *Seminars in Fetal and Neonatal Medicine*. 2014, vol. 19, no. 5, pp. 300–305. ISSN 1744-165X. Available from DOI: https://doi.org/10.1016/j.siny.2014.08.003. The Interface Between Perinatology, Ethics And The Law.

22. GOODFELLOW, Ian; BENGIO, Yoshua; COURVILLE, Aaron. *Deep Learning*. MIT Press, 2016. Available also from: http://www.deeplearningbook.org. Book in preparation for MIT Press.

23. LEHMAN, Charlie. *Visualizing Softmax*. 2019. Available also from: https://charlielehman.github.io/post/visualizing-tempscaling/.

24. PLAUT, Elad. *From Principal Subspaces to Principal Components with Linear Autoencoders*. arXiv, 2018. Available from DOI: 10.48550/ARXIV.1804.10253.

25. LECUN, Y.; BOTTOU, L.; BENGIO, Y.; HAFFNER, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*. 1998, vol. 86, no. 11, pp. 2278–2324. Available from DOI: 10.1109/5.726791.

26. ALBAWI, Saad; MOHAMMED, Tareq Abed; AL-ZAWI, Saad. Understanding of a convolutional neural network. In: *2017 International Conference on Engineering and Technology (ICET)*. 2017, pp. 1–6. Available from DOI: 10.1109/ICEngTechnol.2017.8308186.

27. DANG-NGUYEN, Duc-Tien; PASQUINI, Cecilia; CONOTTER, Valentina; BOATO, Giulia. RAISE. In: *Proceedings of the 6th ACM Multimedia Systems Conference*. ACM, 2015. Available from DOI: 10.1145/2713168.2713194.

28. VOSKOGLOU, Christina. *What is the best programming language for Machine Learning?* [Online]. 2017 [visited on 2023-05-10]. Available from: https://towardsdatascience.com/what-is-the-best-programming-language-for-machine-learning-a745c156d6b7.

29. LION, David; CHIU, Adrian; STUMM, Michael; YUAN, Ding. Investigating Managed Language Runtime Performance: Why JavaScript and Python are 8x and 29x slower than C++, yet Java and Go can be Faster? In: *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. Carlsbad, CA: USENIX Association, 2022, pp. 835–852. ISBN 978-1-939133-29-40. Available also from: https://www.usenix.org/conference/atc22/presentation/lion.

30. JAIN, Arpan; AWAN, Ammar Ahmad; ANTHONY, Quentin; SUBRAMONI, Hari; PANDA, Dhableswar K. DK. Performance Characterization of DNN Training using TensorFlow and PyTorch on Modern Clusters. In: *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. 2019, pp. 1–11. Available from DOI: 10.1109/CLUSTER.2019.8891042.

31. HE, Horace. The State of Machine Learning Frameworks in 2019. *The Gradient* [online]. 2019 [visited on 2023-05-01]. Available from: https://thegradient.pub/state-of-ml-frameworks-2019-pytorch-dominates-research-tensorflow-dominates-industry/.

32. STACK EXCHANGE INC. *Stack Overflow Developer Survey 2022* [online]. 2022 [visited on 2023-05-10]. Available from: https://survey.stackoverflow.co/2022/#section-most-popular-technologies-other-frameworks-and-libraries.

Abbreviations

ANN	Artificial Neural Network
GD	Gradient Descent
GKLD	Generalized Kullback-Leibler Divergence
LU	Linear Unit
MAE	Mean Absolute Error
ML	Machine Learning
MSE	Mean Squared Error
MU	Multiplicative Updates
NMF	Non-negative Matrix Factorization
PCA	Principal Component Analysis
PSNR	Peak Signal-to-Noise Ratio
ReLU	Rectified Linear Unit
RGB	Red-Green-Blue color space
SED	Squared Euclidean Distance
SGD	Stochastic Gradient Descent
SSIM	Structural Similarity
SVD	Singular Value Decomposition

Definitions

r	$\in \mathbb{N}_+$; rank of the NMF decomposition
V	$\in \mathbb{R}_{0+}^{n\times m}$; data matrix to be decomposed by the NMF
W	$\in \mathbb{R}_{0+}^{n\times r}$; the left-hand factor matrix of the NMF decomposition
H	$\in \mathbb{R}_{0+}^{r\times m}$; the right-hand factor matrix of the NMF decomposition
$\mathcal{N}_W$	normal distribution of elements of the W matrix
μ_W	mean of the W matrix; a parameter of $\mathcal{N}_W$
σ_W	standard deviation of the W matrix; a parameter of $\mathcal{N}_W$
$\mathcal{N}_H$	normal distribution of elements of the H matrix
μ_H	mean of the H matrix; a parameter of $\mathcal{N}_H$
σ_H	standard deviation of the H matrix; a parameter of $\mathcal{N}_H$
α	$\in \{\mu_W, \sigma_W, \mu_H, \sigma_H\}$; a symbol representing any of the listed parameters
σ_α	standard deviation of parameter α
σ_α^c	correction coefficient of the parameter α