METHODOLOGY FOR ADAPTING MACHINE LEARNING AND LARGE-SCALE OPTIMIZATION PROBLEMS TO BE SOLVED BY GRADIENT-BASED ALGORITHMS

Rok Hribar

Doctoral Dissertation
Jožef Stefan International Postgraduate School
Ljubljana, Slovenia

Supervisor: Prof. Gregor Papa, Jožef Stefan Institute, Ljubljana, Slovenia
Co-Supervisor: Dr. Jurij Šilc, Jožef Stefan Institute, Ljubljana, Slovenia

Evaluation Board:
Prof. Bogdan Filipič, Chair, Jožef Stefan Institute, Ljubljana, Slovenia
Prof. Janez Povh, Member, University of Ljubljana, Ljubljana, Slovenia
Prof. Boris Naujoks, Member, TH Köln, Cologne, Germany

MEDNARODNA PODIPLOMSKA ŠOLA JOŽEFA STEFANA
JOŽEF STEFAN INTERNATIONAL POSTGRADUATE SCHOOL



Rok Hribar

METHODOLOGY FOR ADAPTING MACHINE LEARNING AND LARGE-SCALE OPTIMIZATION PROBLEMS TO BE SOLVED BY GRADIENT-BASED ALGORITHMS

Doctoral Dissertation

METODOLOGIJA PRILAGAJANJA PROBLEMOV STROJNEGA UČENJA IN OPTIMIZACIJE VISOKE ZAHTEVNOSTI ZA REŠEVANJE Z GRADIENTNIMI ALGORITMI

Doktorska disertacija

Supervisor: Prof. Gregor Papa

Co-Supervisor: Dr. Jurij Šilc

Ljubljana, Slovenia, February 2025

Acknowledgments

I would like to thank for the financial support received through the Slovenian Research and Innovation Agency (research core funding No. P2-0098 and young researcher grant) and through the Computer Systems Department at the Jožef Stefan Institute.

Abstract

The work presented in this thesis deals with the yet unrealized potential of gradient-based algorithms in the fields of computer science other than deep learning. Immense advances in deep learning have revealed the power of both automatic differentiation and gradient descent for solving large-scale optimization problems in a scalable and computationally efficient way. However, its use is still extremely limited due to the lack of methodologies to bring general optimization problems on a similar footing to that of deep learning, as well as due to a belief that deep learning problems are somehow special and that the methods employed there do not transfer well to other problems in computer science that include optimization. In this thesis, we challenge this belief by demonstrating the usefulness of gradient-based algorithms on a diverse set of real-world problems from machine learning and large-scale optimization. One of the goals of this thesis is to show the universality and usefulness of gradient-based algorithms even in the fields where it is not clearly obvious how to apply them. To this aim, we have set out to focus on cases where the applicability and efficiency of gradient-based algorithms seem especially surprising, such as discrete optimization, clustering, and problems that include simulations.

In this thesis, we develop methodological frameworks for applying gradient-based algorithms in machine learning and large-scale optimization by adapting (transforming) those problems to a form where generic gradient-based algorithms can be freely applied. These problem adaptations include developing viable ways to handle constraints, transforming problems by using a more suitable encoding, and rewriting problems to a form that allows gradient calculation (even for simulations and discrete problems). With the aim of comprehensibility, we tackle problems in all three branches of machine learning (supervised, unsupervised, and reinforcement learning) and both discrete and continuous large-scale optimization. We conclude the thesis with one of the open topics in gradient-based optimization, adaptive parameter control of gradient descent. We propose a novel, algorithm-agnostic approach to adapt gradient descent parameters in an online fashion.

Povzetek

V disertaciji predstavljeno delo se ukvarja s še nerealiziranim potencialom gradientnih algoritmov na področjih računalništva izven globokega učenja. Ogromen napredek na področju globokega učenja je razkril zmogljivost tako metode avtomatskega odvajanja kot gradientnega spusta za reševanje optimizacijskih problemov velikih dimenzij na skalabilen in računsko učinkovit način. Vendar je njuna uporaba še vedno zelo omejena zaradi pomanjkanja metodologij, s katerimi bi lahko splošne optimizacijske probleme prilagodili do oblike, znane iz globokega učenja, kot tudi zaradi prepričanja, da so problemi globokega učenja na nek način posebni in da se metode, ki se pri njih uporabljajo, ne prenašajo dobro na druge probleme v računalništvu, ki vključujejo optimizacijo. V tej disertaciji izpodbijamo to prepričanje s prikazom uporabnosti gradientnih algoritmov na raznolikem naboru realnih problemov iz strojnega učenja in optimizacije velikih dimenzij. Eden od ciljev te disertacije je pokazati univerzalnost in uporabnost gradientnih algoritmov tudi na področjih, na katerih ni očitno, na kakšen način jih uporabiti. V ta namen smo se osredotočili na primere, kjer se uporabnost in učinkovitost gradientnih algoritmov zdita še posebej presenetljivi, kot so diskretna optimizacija, gručenje in problemi, ki vključujejo simulacije.

V tem delu razvijamo metodološke okvire za uporabo gradientnih algoritmov v strojnem učenju in optimizaciji velikih dimenzij na način, da prilagodimo (transformiramo) te probleme v obliko, v kateri je mogoče prosto uporabiti splošne gradientne algoritme. Te prilagoditve problemov vključujejo razvoj izvedljivih načinov za obravnavanje omejitev, preoblikovanje problemov z uporabo primernejših načinov kodiranja in prepisovanje problemov v obliko, ki omogoča izračun gradienta (tudi pri simulacijah in diskretnih problemih). V želji po izčrpnosti smo se spopadli s problemi iz vseh treh vej strojnega učenja (nadzorovano, nenadzorovano in spodbujevano učenje) ter v diskretni in zvezni optimizaciji velikih dimenzij. Disertacijo zaključimo z eno od odprtih tem v gradientni optimizaciji, in sicer kontroli parametrov gradientnega spusta. Tu predlagamo novo, algoritemsko agnostično metodo za prilagajanje parametrov gradientnega spusta na sproten način, med optimizacijo.

Contents

List of Figures **xiii**

List of Tables **xix**

List of Algorithms **xxi**

1 Introduction **1**
1.1 Problem Definition . . . 5
1.2 Hypotheses . . . 6
1.3 Scientific Contributions . . . 7
1.4 Thesis Structure . . . 8

2 Gradient-Based Algorithms **9**

I Machine Learning **13**

3 Supervised Learning **15**
3.1 Problem of Forecasting Residential Natural Gas Demand in Ljubljana . . . 16
3.2 Physics-Informed Semi-Empirical Models . . . 18
3.3 Neural Network Models . . . 27
3.4 Discussion . . . 31

4 Unsupervised Learning **35**
4.1 Nonnegative Matrix Factorization and its Connection to Unsupervised Learning . . . 37
4.2 Soft Clustering on Graphs and Similarity Learning . . . 44
4.3 Hard Clustering on Graphs . . . 58
4.4 The `nmfgd` Library for Solving General NM(T)F Problems with Gradient-Based Methods . . . 71
4.5 Discussion . . . 75

5 Reinforcement Learning **77**
5.1 Deep Reinforcement Learning with Exact Gradient . . . 79
5.2 Application to a Protein Structure Problem . . . 83
5.3 Discussion . . . 88

II Large-Scale Optimization **91**

6 Continuous Optimization **93**
6.1 Optimization Methodology in Simulation-Based Shape Optimization . . . 95
6.2 Shape Optimization Problem for Eigenvalues of the Laplace Operator . . . 96

6.3 Methodology for General Shape Optimization with Gradient-Based Algorithm 99
6.4 Experiments 103
6.5 Discussion 107

7 Discrete Optimization **109**
7.1 General-Purpose Discrete Optimization with a Gradient-Based Algorithm . 111
7.2 Application to the Traveling Salesman Problem 116
7.3 Discussion 118

8 Parameter Control of Gradient-Based Algorithms **121**
8.1 Controlling Parameters of Gradient Descent with Gradient Descent 122
8.2 Application to Adam 125
8.3 Discussion 133

9 Conclusions **135**

References **141**

Bibliography **159**

Biography **163**

List of Figures

Figure 3.1: The problem of forecasting future gas demand. The shaded area is a 24-hour time range for the forecast, and the cross-shaped ticks are the desired output of the forecast model for each hour in that range. Reprinted from [55]. 17
Figure 3.2: Density of data points for different values of hourly gas consumption and temperature. The two branches correspond to daytime and nighttime gas consumption behavior. Reprinted from [58]. 19
Figure 3.3: Density of data points for different values of hourly gas consumption and hour of the week. The white line is the median gas consumption with respect to hour of the week. 19
Figure 3.4: Depiction of the general shape of cost of gas with respect to prediction error (difference between forecasted and consumed quantity of gas). The exact way the price is calculated is confidential. 20
Figure 3.5: Example of how $f(T)$ is calculated in TLM using a discrete convolution on past temperatures with a learned response function. 23
Figure 3.6: Example of how $f(T)$ is calculated in TNM using fixed convolutions and learned nonlinear functions to result in a model that includes both nonlinearities and temperature history. Symbol $\circ$ represents function composition. 24
Figure 3.7: Comparison of test error for models when trained by gradient-based Quasi-Newton method, gradient-free Nelder-Mead method, and differential evolution for different values of the forecasting horizon. Differential evolution was applied only to the smallest model due to its long runtime. The last graph compares the errors for all three models when the Quasi-Newton method is used, which was found to bring better accuracy for all models and forecasting horizons. 26
Figure 3.8: Depiction of feedforward artificial neural network model with two hidden layers where every step of model calculation is explicitly shown. W_i are matrices and σ is element-wise nonlinear function. [57] 27
Figure 3.9: Example of RNN model architecture where all layers and units are explicitly shown. Connections between layers are shown with arrows. Hidden layers consist of reccurrent units which need values from the previous time step. For this purpose, one step delay is represented with a black square. Reprinted from [57]. 28
Figure 3.10: Convergence of training and test error during optimization when using RMSProp algorithm. (a) An example of the standard training progress without overfitting where test error is slightly higher than training error. (b) An example of overfitting in the middle of training that was resolved after continuing optimization. 30

Figure 3.11: Test error of RNN models for different model architectures with respect to the number of layers. All models used have an overall number of parameters equal to 4000 and are therefore comparable. 31

Figure 4.1: A simple example of hard and soft clustering on a sample data set. Reprinted from [118]. 39
Figure 4.2: Depiction of matrix factorization with the meanings of rows and columns for each matrix. 40
Figure 4.3: Data fusion approach when data sets are available that measure the same quantities in different ways in case of symmetric NMTF. Reprinted from [127]. 43
Figure 4.4: Multi-type data fusion approach where several object types are clustered simultaneously using several data sets. Reprinted from [128]. 44
Figure 4.5: MSE with respect to the number of clusters k for different algorithms and problems. In case of the synthetic problems, k is measured in % of the actual number of clusters. Reprinted from [56]. 51
Figure 4.6: The plot shows which algorithm achieved the lowest MSE of all four algorithms considered with respect to the matrix dimension n and the chosen number of clusters k. Each marker position represents one symmetric NMTF problem, and the type of marker encodes which algorithm has found the best solution for that problem. The plot includes results for all three data collections considered. Reprinted from [56]. 53
Figure 4.7: Distributions of RSE values of individuals produced by crossover from parents with RSE from the shaded range. The crossovers considered are simulated binary crossover (SBX), simplex crossover (SPX), differential evolution crossover (DE), and choosing random columns (RC) with uniform probability out of both parents. The trivial solution is a candidate with all components of matrices G, S_i equal to zero. Reprinted from [63]. 54
Figure 4.8: Box plot of hypervolumes for the three algorithms considered. Hypervolume is one of the measures used to assess the quality of solutions found by multi-objective optimization algorithms. Higher hypervolume value indicates a higher-quality solution. M1 is a memetic algorithm when only mutations are applied, M2 when both mutation and crossover are applied, and NA is the naive approach where only Adam is applied from random points with different and predetermined values of k. Reprinted from [60]. 56
Figure 4.9: Progress of the Adam algorithm when initial points are individuals created by crossover compared to the random individuals. All solutions here have $k = 10$, and crossovers were performed using already descended individuals with $k = 3, 4, 5, 6, 7$ ($3 + 7$, $4 + 6$ and $5 + 5$). Full lines are the medians, and the shaded areas represent the range of the central 66% of runs. Reprinted from [63]. 57
Figure 4.10: Median progress of the Adam algorithm when initial points are random individuals compared to mutated individuals. All solutions here have $k = 10$, and mutations were performed using already descended individuals with $k = 4, 5, \ldots, 9$. Quantity Δk tells how many columns were added via mutation and represents the extent to which an individual was mutated. Reprinted from [63]. 58

Figure 4.11: Depiction of the three-stage scheme for using a gradient-based algorithm for finding orthogonal nonnegative solutions of arbitrary NMF problem. The color signifies an optimization landscape, with yellow tones indicating more optimal solutions. The orange line represents a subset of the search space where orthogonality is fulfilled (in addition to the nonnegativity). The three stages are depicted with black arrows where the first stage finds a good solution outside of the feasible space, the second projects the solution onto the feasible space by orthogonalization, and the third finds a good solution while restricted to the feasible space of orthogonal nonnegative matrices. 61
Figure 4.12: Comparison of the best objective values obtained with the three-stage Adam algorithm for different ratios of k/K, where k is the number of columns in G and K the rank of matrices R_i. For better visibility, the case $k = K$ is shown in greater scale on the right and, since the optimization finds extremely low values, using a logarithmic scale. . . . 65
Figure 4.13: Comparison of best objective values obtained with the three-stage Adam algorithm for different sizes of synthetic matrices R_i. 68
Figure 4.14: Example of the progress of optimization when the three-stage scheme is used in conjunction with the Adam algorithm. The blue line shows how objective value MSE changes in logarithmic scale and the red line how the orthogonality changes with the number of gradient descent steps. The shaded area represents the third stage of the scheme and the border between the white and the shaded area is where orthogonalization takes place (the second stage). The matrix being factorized is an instance from the synthetic data set collection with dimension of $n = 5000$ and rank $k = K = 20$. This means that the global minimum is known to be exactly zero. 68
Figure 4.15: Toy example of a weighted graph with one cluster that joins biomedical terms that are semantically more related among themselves than the rest of the terms in the graph. 69
Figure 4.16: A snippet from an example file `GNMTF.py` that shows a problem of simultaneously factorizing 9 matrices describing relations between 3 types of data objects. For some relations, many matrices are available, while for some, there are none available. This example demonstrates the flexibility of `nmfgd` with regards to problem formulation that can imply data fusion of many data sets. 72
Figure 4.17: Progress of optimization using the Adam algorithm to solve a multi-relational multi-type NMTF problem with 80% of missing values and using a flexible metric choice RAE. 74

Figure 5.1: Small-scale example of how network g receives the protein specification and how its output is interpreted. If a protein is shorter than the size of the network's input layer, zero is placed on sites where there are no amino acids. In this case some angles from the output are discarded (crossed out outputs). Reprinted from [59]. 84

Figure 5.2: The objective function value achieved by our models with respect to the number of hidden layers for two different magnitudes of initial weights w. The full line is the mean objective function value achieved by the models, the shaded area shows the range of the objective function for the central 66% of the models, and the dashed line is the objective function of the best model. To obtain these values, 100 runs of training were performed for each configuration. Reprinted from [59]. 86
Figure 5.3: The objective function value during training. The upper plot shows the progress of a usual gradient descent procedure, while the lower plot shows progress when the gradient norm has a predefined upper bound by using gradient norm clipping. Reprinted from [59]. 87
Figure 5.4: Progress of 30 runs of differential evolution by using a random initial population or initial population partially populated with solutions predicted by our neural network models. The full line is the mean over all runs, shaded area shows the range of the central 66% of runs, and the dashed line shows the best run. Reprinted from [59] 88

Figure 6.1: Illustration of how our shape model with an invertible neural network works on a two-dimensional example. The neural network maps vectors from the circle to general two-dimensional vectors. Examples of such points are depicted as color dots. Since a neural network is a continuous function, its codomain is a closed curve, so it maps to a legitimate boundary of a two-dimensional shape Ω. 100
Figure 6.2: Sensitivity of the algorithms to the value of the hyperparameters learning rate for Adam and mutation strength sigma for LM-CMA. Mean absolute percentage error (MAPE) was calculated with respect to the baseline of the best-known objective values found in [225]. The full line is the median, and the shaded area includes the central 66% runs. These values were obtained via linear quantile regression. 104
Figure 6.3: Optimization progress for Adam and LM-CMA when run on six different problems ($k = 2, 3, 4, 5, 6, 7$). The full line is the median over 30 runs of the algorithm, while the shaded area shows the central 66% of the runs. The dotted line is the best objective function value found. 105

Figure 7.1: Neural network g_ϑ representing a continuous probability distribution by transforming samples drawn from a standard normal distribution. . . . 113
Figure 7.2: Discretization D can be used to map a continuous probability distribution to a discrete probability distribution. 114
Figure 7.3: Architecture of how the composition of functions is performed to obtain the final objective values. Below each variable, we depict an example distribution that shows how samples of variables might be distributed. . 115
Figure 7.4: The best objective value observed during optimization with Adam with respect to the learning rate for four different problem instances. 118
Figure 7.5: Convergence of optimization with Adam and random search for four different problem instances. Full lines show the median objective value over 100 runs, the shaded area represents the central 66% of runs, and the dotted line shows the best objective value observed at each time step across all runs. The figure illustrates the expected statistical behavior of the optimization and the distribution of objective values over time rather than specific individual trajectories. The optimal objective function value is in all cases zero, in accordance to (7.11). 119

Figure 8.1: Application of a step of gradient descent with momentum from point x^i to x^{i+1} when applied to a maximization problem. Two parameters α and β determine the position of x^{i+1} given gradient g^i and momentum in the previous step m^{i-1}. Dashed lines show where x^{i+1} would end up if parameters α and β have been chosen differently. Based on the gradient g^{i+1} in the new point x^{i+1} we can deduce in which direction the parameters α and β would need to be changed so that x^{i+1} would end up in a region with a better objective function value. 124

Figure 8.2: Comparison of the Adam and cAdam algorithms for training a convolutional neural network on an MNIST data set. The full line is the median over 100 runs, and the shaded area includes the central 66% of runs. The bottom plot shows how objective value changes with respect to the number of steps of gradient descent in the log-log scale. The top two plots show how the parameters α, β_1, β_2, and ε of the cAdam algorithm change during the descent in the lin-log scale. 129

Figure 8.3: Comparison of the Adam and cAdam algorithms for training symmetric NMTF model for soft clustering on four real-world data sets. The full line is the median over 100 runs, and the shaded area includes the central 66% of runs. The plots show how objective value changes with respect to the number of steps of gradient descent in the log-lin scale. . 131

Figure 8.4: Comparison of the Adam and cAdam algorithms for training symmetric NMTF model for soft clustering on the Protein data set. The full line is the median over 100 runs, and the shaded area includes the central 66% of runs. The bottom plot shows how objective value changes with respect to the number of steps of gradient descent in the log-log scale. The top two plots show how the parameters α, β_1, β_2, and ε of the cAdam algorithm change during the descent in the lin-log scale. 132

List of Tables

Table 2.1: Three examples of v_i calculations used in modern gradient descent algorithm implementations. $\gamma \in [0, 1)$ is a fixed parameter (with usual value of around $\gamma = 0.999$). 10

Table 3.1: Variables used by the models in this work. Variables *ph*, *sh* and *hw* are boolean whereas the others are real. 18

Table 3.2: Training times for the three models when using the Quasi-Newton method, Nelder-Mead algorithm and Differential evolution. Differential evolution was applied only to the smallest model due to its long runtime. 26

Table 3.3: Training time of RNN models in hours for different unit types and for different hardware with respect to number of layers. All models used have an overall number of parameters equal to 4000 and are therefore comparable. One training comprises of 10,000 epochs or $\approx$ 70,000 gradient descent steps. 30

Table 3.4: Mean absolute error (MAE), mean absolute percentage error (MAPE), and training time of the tested models when hourly resolution is required (hour) and when daily resolution is required (day). All models were trained using gradient-based optimization and using a data set for a 24-hour forecasting horizon. 33

Table 4.1: Optimization results for the one- and three-stage Adam algorithm when applied to synthetically generated data sets. n is the number of objects in a data set, K is the actual number of clusters in a data set and k is our guess for the number of clusters. If $k = K$, the optimal MSE is known to be zero, while for $k < K$ the optimal MSE is not known. However, it is known that optimal MSE is a monotonously decreasing function of k. 66

Table 4.2: Objective values for the one- and three-stage Adam algorithm when applied to real-world data sets. 70

Table 5.1: Free energy calculated for three proteins found in nature by using structures predicted by our neural network model and by differential evolution (DE). 87

Table 6.1: Comparison of the best solutions found by Adam and in [225] for six different problems ($k = 2, 3, 4, 5, 6, 7$). Our solution for problem $k = 3$ is visually different because it was generated from a different initial mesh (a square instead of a disk) in order to obtain optimization difficulty similar to other k. 106

Table 7.1: Possible discretization functions D for different encodings commonly used in discrete optimization. 114

Table 8.1: Common learning rate schedules and their definitions [267]. In this formulation, the learning rate schedule has the form $k_0 + (k_1 - k_0)g(t)$ where t is the iteration number, $k_0 < k_1$ are the minimum and maximum learning rate values, and $g(t) \in [0, 1]$ determines how the learning rate changes with t. The schedules are classified into three categories: fixed, decaying and cyclic. 122

List of Algorithms

Algorithm 6.1: Code implementing gradient-based shape optimization. The only part of the code omitted here (imports in lines 2–3) are methods related to FEM (geometrical operations described in Section 6.2.1.1 and round mesh generation) and neural network construction. Here, we highlight how user-friendly gradient methods can be implementation-wise. For example, gradient calculation is achieved in a single line of code (line 33). To perform automatic differentiation, the TensorFlow library requires the user to only specify how the objective is calculated from the encoding (lines 19–29) and the computational graph for the gradient calculation is built automatically without any explicit code from the user. As seen in lines 24–27, the calculation of the objective involves complicated linear algebra operations which make the explicit derivation of formulas for the gradient calculation extremely difficult. However, if all intermediate steps during objective calculation are recorded (on `tape` in line 18), the gradient can be automatically calculated in the same computational complexity as the objective. Then, a gradient can be used to update the neural network weights iteratively using the Adam algorithm (line 35). . . 102

Algorithm 7.1: Our procedure for discrete optimization. In the line 10 we employ a generic "vanilla" gradient descent, however, this can be substituted for any other gradient-based optimization algorithm. 116

Algorithm 8.1: Adam. 126
Algorithm 8.2: Adam with dynamical parameters. 127
Algorithm 8.3: cAdam. 127

Chapter 1

Introduction

Gradient-based optimization has recently become an important research topic thanks to the triumph of deep learning. In the last decade, deep learning brought immense advances in the field of artificial intelligence. Examples include image classification [1], speech recognition [2], image caption generation [3], image generation from text [4], machine translation [5], game playing [6], natural language processing [7], drug discovery [8], and so on. These achievements are nowadays attributed to the following four circumstances: accessibility of large amounts of data, efficient GPU implementations, advances in neural network architectures, and efficient training techniques [9]. The last point is inextricably linked to the use of gradient-based optimization methods. Neural network training has been performed almost exclusively by some form of gradient descent in the last decades, with gradients being calculated using automatic differentiation. Even though gradient-based optimization is so efficient in deep learning setting, it is currently rarely seen in research focusing on optimization itself. Even though gradient-based optimization was prevalent in the early days of computer science, it slowly fell out of favor [10] starting with the first derivative-free methods in the 1960s [11].

There are several reasons for the shift in popularity from gradient-based to derivative-free optimization methods. The first stems from the belief that a usual objective function found in real-world optimization has a ubiquitous number of local minima with objective values much larger than that of the global minimum. Following this belief, a gradient descent would, due to its local behavior, on average, converge toward a local minimum with a high objective function value and would therefore be inadequate for real-world multi-modal optimization problems. Contrary, following the same train of thought, algorithms with more exploration power would be able to explore new regions of the search space, visit several local minima along the way, and converge toward a solution closer to global optimum [12]. This belief contributed to a shift from single-solution-based optimization algorithms, where there are fewer opportunities for exploration, toward population-based ones, where it might be possible to balance exploitation and exploration, which would be needed to search in multi-modal optimization landscapes. Another reason for shifting to population-based algorithms is also their parallelizability. In population-based algorithms, a whole population of candidates (typically around 100) need to be evaluated in order to generate a new population of candidates for the next iteration of the algorithm. This means that the evaluations of the objective function can be performed concurrently, for all candidates in a population. If this parallelization is utilized, the runtime of the optimization algorithm can be reduced by a factor equal to the population size. This is in contrast to single-solution-based algorithms, such as gradient-based ones, where only a single evaluation of the objective function needs to be performed to generate a new candidate. This scheme requires sequential evaluation of candidates and does not allow for parallelization

of this type.

The success of deep learning puts the belief about the existence of a high number of high-laying local minima in real-world optimization problems into question. Current evidence, in fact, indicates that these "problematic" local minima occur only when search space is low-dimensional and for objective functions that were specially designed to possess this property [13]–[15]. Theoretical findings from statistical physics and random matrix theory demonstrate that for randomly chosen objective functions, the likelihood of such "problematic" local minima fall exponentially with the dimension of the search space [13]–[15]. For a Gaussian field, it was shown that the average difference in objective function value between any local minimum and the global minimum falls exponentially with dimension [13]. In an earlier work [14], similar results were shown for a confined case (for random error functions superimposed on a quadratic error surface) by analyzing the distribution of critical points with respect to their type and objective function value. It was shown that the probability that a critical point is a local minimum (as opposed to a saddle point) increases when the objective function decreases. However, this switch between saddle points and local minima becomes more and more abrupt when dimension is increased (also with exponential dependency). In other words, for high-dimensional objective functions, all local minima are very likely to have an objective function value that is very close to that of the global minimum. Practically all other critical points with higher objective function values are, therefore, saddle points. This phenomenon was also empirically observed in several deep learning use cases [15].

We can also understand these results more intuitively. For a critical point to be a local minimum, the objective function should "increase" in all possible directions from that point. If there is only a single direction in which the function "decreases", such a point is a saddle point. When the dimension becomes very large, it becomes increasingly unlikely that objective function "increases" in all possible directions. Probabilistically speaking, if the space dimension is in the thousands or millions, it is quite likely that we can find at least one direction in which the function decreases. There is, however, a geometric obstruction that would make the local minima more likely, even in high-dimensional space. If the critical point has an objective function value that is very close to the global minimum, it is, due to geometry, difficult to find directions that would "decrease" the objective function. This is why we can observe local minima that are concentrated very close to the level of the global minimum while saddle points become abruptly and overwhelmingly predominant above this level.

If we suppose that real-world optimization problems follow a similar distribution as "random" functions, these findings indicate that reluctance toward gradient-based optimization methods is largely unjustified if the dimensionality of the search space is large enough. The well-known fact that gradient-based optimization algorithms converge toward a local minimum, in this setting, means that the algorithm converges toward a point with objective function value that is very close to the global minimum, which is the aim of the global optimization. However, the usefulness of gradient-based methods is yet to be verified in practice in several fields of machine learning and optimization.

It is important to note that the efficiency of optimization algorithms in the current age is determined by applying them to a set of benchmark functions, such as the ones provided by COCO platform [16]. These benchmarks usually consist of subsets with similar properties, such as separability, low/high conditioning, multi-modality, weak global structure, and so on. These functions are well established and were hand-crafted to possess these properties [17]. This means that the multi-modality of benchmark functions is enforced by design and does not necessarily reflect real-world situations. Objective functions such as Rastrigin, Weierstrass, Schaffer and Griewank-Rosenbrock can all be made to work on

high-dimensional search space and have a ubiquitous number of "problematic" local minima because they were designed in such a way. But this does not necessarily mean that they reflect characteristic objective functions from a real-world optimization. In some sense, the shift to evolutionary algorithms that has been seen in the last decades in the optimization research community might be simply due to their efficiency in these hand-crafted objective functions included in the benchmark sets. A similar claim was already expressed in the case of CMA-ES (its derivatives are among the most efficient ones on the established benchmark functions) that it works well due to its invariance to rotations and the fact that rotations are used in benchmark sets to generate different (non-separable) function instances [18].

The second reason for the shift in popularity from gradient-based to derivative-free optimization methods is the belief that gradient calculation is a highly nontrivial procedure for a general optimization problem which also fueled the development of derivative-free optimization algorithms in the last decades [19]. In order to apply gradient-based algorithms in optimization, one needs a method for calculating a gradient of the objective function with respect to all variables in the search space. The belief is that there are cases where the gradient of the objective function is unavailable, unreliable or impractical to obtain. With the immense advances in metaheuristic methods that work well on a very diverse set of problems, even those where gradient is "unavailable", the interest moved away from various methodologies for efficient gradient calculation and consequently left gradient-based optimization in the background.

This work will test this belief from two perspectives, that gradient is oftentimes "hard to obtain" and that it is "not practical to use". Several large-scale optimization problems will be analyzed from different fields of computer science and industry and put to the test how problematic is gradient calculation and whether it is useful. There are clues, however, that already show that these two beliefs stand on shaky ground. The first statement that the gradient is sometimes costly to produce (either computationally or from a programming point of view), with simulation-based optimization being a common example [19], can be refuted by considering automatic differentiation, which can produce a gradient with the same time complexity as objective function evaluation and can be implemented without extensive programming effort [20]. The space complexity can, however, be several times larger (compared to objective function evaluation) but this factor is limited by the logarithm of the runtime, which makes the method applicable to computational problems of virtually any size [21]. The second statement that the gradient is sometimes not useful, for example, when the objective function is very noisy or flat on a large proportion of the search space, can be challenged by the success of deep learning where both properties are common. Noisy function evaluation is standard in deep learning due to training neural networks in batches but is especially notorious in the case of deep reinforcement learning. Therefore, several methods emerged in the level of optimization algorithms that deal with such noise, including the use of momentum, gradient clipping, individual adaptive learning rates, and weight averaging [22]. Also, very flat regions of the optimization landscape can be traversed more easily by the use of momentum and individual adaptive rates learning as well as by plateau deflecting [15]. However, the current methodology does lack examples of use in cases where search space is flat almost everywhere. This case will be analyzed in this work in two different use cases, clustering and discrete optimization problems.

The prospect of gradient-based methods for large-scale optimization is further pronounced by the difficulties of evolutionary algorithms in this setting. Evolutionary algorithms are currently state of the art in optimization, however, their efficiency deteriorates when search space dimension increases [23]. Due to their exploration power, they suffer from the curse of dimensionality [24]. It is yet unclear how gradient-based methods

compare to evolutionary algorithms for real-world large-scale optimization problems.

There are some instances in the literature where gradient-based methods have been used in other machine learning fields outside of deep learning, especially in probabilistic modeling. These examples include the use of user-friendly libraries established by deep learning for performing general automatic differentiation, such as Theano [25], TensorFlow [26], Caffe [27], CNTK [28], PyTorch [29], and so on. More specifically, Theano was applied for Bayesian modeling by Markov chain Monte Carlo [30], TensorFlow for probabilistic modeling, inference, and assessing the model's goodness of fit (criticism) [31], PyTorch for universal probabilistic programming with support for recursion and control flow [32]. However, several fields of machine learning still remain untouched by gradient-based methods.

Automatic differentiation has also been used in simulation-based optimization. Examples include simulations in atmospheric sciences [33], engineering design [34], structural mechanics [35] and computational finance [36]. However, these papers left little impact and preceded more advanced methods that emerged with deep learning. Even though the mentioned works show that it is possible to calculate gradients, even when function evaluation is a result of a simulation it remains unclear how well modern gradient-based optimization algorithms perform in such simulation-based optimization.

An area of machine learning that routinely employs simulations is reinforcement learning which is becoming an increasingly important area of artificial intelligence. In reinforcement learning, however, simulations are not differentiated at all. In order to bypass differentiating simulation control flow, deep reinforcement learning adopted complicated mechanisms to generate gradients only from objective function values. Currently the most common approaches include policy gradient method [37], actor-critic method [38] and Q-learning [39]. They all suffer from stability problems and require extensive tuning. Several mechanisms are normally used to combat these issues, including experience replay [40], target networks [41] and batch normalization [42]. It remains unknown whether the use of simulation differentiation alleviates these issues.

An important aspect of evaluating algorithm performance is how results are compared and interpreted. In this thesis, we adopt specific methodological choices regarding the presentation and comparison of results. The evaluation of algorithm performance is presented primarily through visual and distributional summaries, such as medians and intervals containing two-thirds of the runs, rather than through classical statistical significance testing. This approach is motivated by several practical considerations. Statistical tests may highlight differences that are numerically small but practically irrelevant, and they rely on assumptions (independence, identical distributions, large sample sizes) that are often violated in realistic optimization experiments [43]–[45]. Moreover, single-score comparisons in machine learning can be misleading due to noise (introduced by splitting the data set into training and test set) and one can observe statistical differences even between identical approaches [46]. Information about statistical significance can thus give the reader the impression that one algorithm is definitively better than another, even when differences are practically irrelevant or the result of noise. Another aspect of single-score comparisons is that they fail to convey information about variability, robustness, and the range of likely outcomes for the algorithm [47]. This can give the reader the impression that two algorithms behave the same even when one is much more robust then the other.

In applied settings, practitioners such as engineers, computational biologists, or industry specialists are primarily concerned with whether an algorithm is likely to produce acceptable results within limited computational budgets, rather than with formal statistical significance. This makes distributional summaries directly informative since they allow users to assess the expected behavior, stability, and potential risks associated with each

algorithm. Evaluating the practical usefulness of an algorithm for specific tasks, such as protein structure prediction, finite-element simulation-based shape optimization, or other domain-specific problems, requires expert knowledge of the problem context, constraints, and acceptable tolerances. Accordingly, the thesis presents algorithmic behavior in a form that supports informed decisions by domain experts, while leaving the assessment of task-specific practical significance to the practitioners who operate within those fields.

1.1 Problem Definition

This work is intended to shed light on the usefulness of gradient-based optimization in a broader sense, not only in deep learning but also in other machine learning approaches and large-scale optimization in general.

The era of deep learning revived development of gradient-based optimization algorithms and gradient calculation methods and also produced state-of-the-art libraries that employ these approaches. From the standpoint of the optimization research community, the success of gradient-based optimization in deep learning is rather surprising given the fact that the optimization problem of training a neural network is known to be non-convex and multi-modal [48]. The question arises whether the optimization landscapes in deep learning have some special property that makes the success of gradient-based optimization possible, or would such methodology work elsewhere as well.

The purpose of this work is to test whether the optimization approach used in deep learning can be applied to other large-scale optimization problems and what kind of adaptations are needed to make that happen. Since evidence shows that gradient-based methods should work best when the dimension of the search space is large, our focus is on large-scale optimization problems. We especially look into large-scale optimization problems in one of the most favored areas in current computer science, machine learning. Gradient-based optimization will be studied in all three types of machine learning: supervised, unsupervised, and reinforcement learning. In supervised learning, we will focus on a specific forecasting problem and apply gradient-based optimization to custom physics-informed machine learning models. In unsupervised learning, this work will center on (soft) clustering problems and dimensionality reduction problems which reduce to various types of matrix or tensor decompositions. In reinforcement learning, full automatic differentiation will be demonstrated, which does not require unstable deep reinforcement learning techniques. The advantages and drawbacks of gradient-based approaches will be deduced using the mentioned use cases.

Another aim of this work is to prove that gradient-based methods may be applied in rather unexpected fields, including large-scale simulation-based optimization and discrete optimization. In simulation-based optimization, we will focus on shape optimization and show how gradient-based optimization can be used for problems with practically arbitrary continuous objective functions, even ones that include complicated programs, such as simulations. In discrete optimization, methods will be introduced that map discrete problems to continuous ones where gradient-based methods may be applied. This approach is based on modeling a discrete distribution using a continuous model and will be applied to scheduling problems often found in industry and engineering.

The last topic of this thesis is the problem of efficient parameter control of gradient descent algorithms which can also be tackled with gradient-based methodology by changing control parameters with accordance to derivatives of the objective function with respect to the control parameters.

1.2 Hypotheses

The hypotheses of the thesis relate to the success of the developed methods for adapting diverse set of problems in machine learning and large-scale optimization into a form that can be solved with gradient-based algorithms. We are interested in the effect that this adaptation has on convergence properties, computational efficiency, and stability of an optimization or training procedure.

Hypotheses **H1**–**H3** are connected to machine learning and address the potential of gradient-based methods in each of the three types of machine learning. All three types of machine learning have specific methodological gaps where gradient-based methods could be beneficial. In supervised machine learning, it is known that there is a lot of potential in using customized machine learning models to enhance their explainability and generalization. We can incorporate expert knowledge into the structure of the model, which is known to promote generalization and is especially relevant when only a small amount of data is available. However, it not known how exactly training should be performed on such specialized machine learning models. In unsupervised machine learning, gradient-based methods are currently used in deep learning to model distributions and perform dimensionality reduction. However, the task of clustering data instances to either overlapping or disjoint groups is difficult to achieve due to the discrete nature of the problem. In deep reinforcement learning, the training of the model is performed by circumventing the calculation of the gradient of the reward function by employing elaborate and approximate methods such as using a surrogate to differentiate the reward. This introduces instabilities when performing gradient descent.

H1: Gradient-based algorithms are applicable and, due to their local nature, less computationally expensive for training custom supervised machine learning models where there is no established training procedure and where general optimization methods are usually employed.

H2A: It is possible to formulate (soft) clustering problems using continuous encoding by imposing constraints and solve clustering problems posed in this form efficiently by using gradient-based algorithms without separating the search space into blocks.

H2B: The continuous formulation of (soft) clustering enables a more flexible definition of clustering and can in practice be used for data fusion and data imputation.

H3: Training instabilities observed in standard deep reinforcement learning can be eliminated by using exactly calculated gradients of the objective function, which can be obtained by automatic differentiation.

Hypotheses **H4**–**H6** relate to large-scale optimization and address potential of gradient-based methods to continuous and discrete optimization and parameter control. Continuous large-scale optimization and especially simulation-based optimization problems are currently being solved by derivative-free optimization algorithms. There are special cases where gradient-based algorithms have been established historically. However, there is no generality in the methodology in this cases. In discrete optimization, the objective function is flat almost everywhere in the search space, and there is no clear way to apply gradient-based optimization in this setting. In parameter control, we focus on gradient-based optimization algorithms, which include several control parameters that influence their efficiency depending on the characteristics of the local optimization landscape. During the descent, the optimization landscape changes, however, current practice does not take that into account but rather includes static parameter control that is oblivious of the geometric nature of the current candidate surroundings.

H4: Automatic differentiation can be used to calculate the gradient of any computable objective function (even those that include simulations) and this methodology can be used in practice in conjunction with gradient descent to produce efficient ways of solving general simulation-based continuous optimization problems.

H5: Gradient descent can be applied to discrete optimization problems if we substitute a discrete encoding for a (continuous) probabilistic one and adapt gradient calculation scheme.

H6: Adaptive parameter control scheme for gradient descent algorithms can be constructed in which change in parameter values is guided by the derivatives of the objective function with respect to the parameters of the gradient descent. This approach alleviates the need for parameter tuning, which is a usual part of the workflow that includes gradient-based optimization.

1.3 Scientific Contributions

Machine learning and large-scale optimization problems are at the core of many problems found in industry, engineering, and academia, which is why improving the efficiency of used algorithms is very relevant. The increase in data quantity and the tendency for higher resolution in computation directs research toward developing faster, more accurate, and also reconfigurable algorithms. This work sheds light on whether gradient-based optimization can be used to achieve this.

The main scientific contributions of this thesis can be condensed into the following seven points:

1. We challenge the paradigm that gradient-based optimization is primitive and applicable only to simple optimization problems with a special structure (those with a continuous objective function with a closed mathematical form).

2. We transfer the well-developed gradient-based algorithms from deep learning to novel contexts, most notably, physics-informed modeling, clustering, simulation-based optimization, and discrete optimization.

3. We develop novel adaptation methodologies that bridge the gap between deep learning problems and more general large-scale optimization problems. These adaptations address traits that are not common in deep learning such as constraints, discrete and mixed search spaces, and objective functions with computationally complicated control flow.

4. We demonstrate that gradient-based algorithms are competitive (and in some cases superior) compared to commonly used metaheuristic ones in several different problem settings that include large-scale optimization.

5. We develop an algorithm for soft and hard (co-)clustering that can accommodate data fusion and can work on data sets with (many) missing values.

6. We develop a novel algorithm for discrete optimization that utilizes a new method for encoding a distribution over discrete variables using deep neural networks.

7. We develop a novel adaptive parameter control strategy for gradient-based algorithms and apply it to the Adam algorithm resulting in a new cAdam algorithm which exhibits more favorable convergence.

A byproduct of this thesis are implementations of the developed methods, which are made publicly available under the "Unlicence" license (the software is dedicated to the public domain without restrictions on its use). With the exception of our work on gas demand prediction problem (Section 3) which is protected by the non-disclosure agreement, all results of this thesis can be reproduced by third parties using our implementations [49]–[54]. Because state-of-the-art libraries for automatic differentiation employed in our code allow for differentiation of any differentiable program, our implementations can be deployed for solving a wide range of large-scale optimization problems related to the ones considered in this work. This brings a certain degree of universality and if a user is willing to implement their own objective function using the same framework, our implementations can be used to run the proposed methods on arbitrary problems. A highlight of our code collection is the `nmfgd` library which includes extensive documentation and examples of its use. `nmfgd` can be used for (soft) (co-)clustering and dimensionality reduction and is highly reconfigurable, with support for data sets with missing values and for analyzing multiple related data sets at the same time (data fusion).

This work also contributed (in collaboration with co-authors) to three published journal papers [55]–[57], five published scientific conference contributions [58]–[62], two published monograph chapters [63], [64], and four published scientific conference contribution abstracts [65]–[68].

1.4 Thesis Structure

The thesis is composed of two introductory chapters, two three-chapter parts, each considering a different problem setting for the use of gradient-based optimization, and a conclusion. In Chapter 2, an introduction to gradient-based methods is presented, including methods for gradient calculation and optimization algorithms that require gradients. Following the introductory chapters, Part I is composed of three chapters that correspond to the three machine learning approaches: supervised, unsupervised, and reinforcement learning. Chapter 3 focuses on optimization methods for training supervised machine learning models. The methods are applied to a real-world use case of gas demand forecasting and include gradient-based optimization of custom machine learning models and recurrent neural network models. In Chapter 4, gradient-based optimization is applied to unsupervised machine learning problems, including (soft) clustering, dimensionality reduction, and similarity learning. The methodology is presented for transforming unsupervised problems to a representation in which a gradient-based method can be applied. Chapter 5 is focused on reinforcement learning, especially on the adaptation of the model training procedure so that the gradients are calculated by automatic differentiation. To demonstrate the applicability of gradient-based optimization in this setting, a protein structure prediction model is trained without a data set using a reinforcement learning approach. Part II consists of three chapters that deal with applying gradient-based methods in the three problem types in large-scale optimization: continuous, discrete, and parameter control. In Chapter 6, we are concerned with general large-scale optimization problems with continuous search spaces. We focus on problems where the objective function is calculated by complicated programs, such as in simulation-based optimization, and apply the developed methodology to a shape optimization problem where the objective function is calculated using the finite element method. In Chapter 7, a gradient-based optimization algorithm is developed for general discrete optimization problems and applied to the traveling salesman problem. Chapter 8 presents a novel parameter control method that can be used for gradient-based optimization algorithms and is applied to the Adam algorithm. Finally, in Chapter 9, conclusions are drawn.

Chapter 2

Gradient-Based Algorithms

In this work, gradient-based algorithms, as a term, encapsulates both optimization algorithms that use gradients in their search as well as algorithms used to calculate or approximate the gradient of the objective function. Both aspects are important for this work, and in this chapter, we provide a brief introduction to the existing methods.

Gradient-based optimization is any method that uses gradients to search for the minimum. The most basic example is "vanilla" gradient descent that in step i of the iteration finds the next candidate x_{i+1} by moving in the search space from the current candidate x_i in the opposite direction of the gradient of the objective function g_i.

$$x_{i+1} = x_i - \alpha g_i \tag{2.1}$$

Fixed positive parameter α determines the step size and is known in machine learning as the learning rate. This means that if α is infinitesimally small, each step of gradient descent moves in the search space in a direction that is expected to decrease the objective function the most. However, very small values of α lead to slow convergence because a large number of steps are needed to descend to regions with small objective function values. On the other hand, very large values of α can cause the algorithm to simply skip over such regions and cause the algorithm to diverge. Therefore, the value of α needs to be carefully chosen for a given problem. The basic form of gradient descent has some other notable drawbacks. If the optimization landscape has very narrow valleys (high conditioning), the algorithm will usually oscillate around the valley and take many steps to find the minimum. Likewise, if the optimal descent path is very curved, it will converge slowly for the same reason. Another possible reason that can slow the convergence of "vanilla" gradient descent is descending in the regions of the landscape where the magnitude of the gradient g_i is small (for example, near saddle points and plateaus) [69].

In the recent decade, several new gradient descent-inspired algorithms were developed and successfully used in deep learning. They build on the original gradient descent algorithm with only minute changes (which is why they are all sometimes referred to as gradient descent), however, they mended the drawbacks of the original algorithm to a large degree. The introduction of momentum reduces oscillations and stabilizes the descent by using

$$x_{i+1} = x_i - \alpha m_i, \tag{2.2}$$

where m_i is some type of average of the gradients seen in this and previous steps. The most usual version uses the exponential moving average

$$m_i = \beta m_{i-1} + (1 - \beta) g_i, \tag{2.3}$$

where $\beta \in [0, 1)$ is a fixed parameter (with usual value of around $\beta = 0.9$). The use of momentum also makes it harder for the algorithm to get stuck in shallow local minima and

Table 2.1: Three examples of v_i calculations used in modern gradient descent algorithm implementations. $\gamma \in [0, 1)$ is a fixed parameter (with usual value of around $\gamma = 0.999$).

Algorithm	v_i calculation rule	description of the rule
AdaGrad [71]	$v_i = \frac{(i-1)v_{i-1} + g_i^2}{i}$	full arithmetic mean
RMSProp [72]	$v_i = \gamma v_{i-1} + (1-\gamma) g_i^2$	exponential moving average
Adam [73]	$v_i = \frac{\gamma v_{i-1} + (1-\gamma) g_i^2}{1-\gamma^i}$	unbiased exponential moving average

makes it easier to traverse flat regions of the optimization landscape [70]. Since plateaus and saddles become more pronounced when the dimensionality of the problem increases, much effort was put into finding a way to appropriately scale step sizes made by the algorithm, which would be based on the current geometry of the landscape. One of the most established ways to achieve this is the adaptive per-parameter learning rate

$$x_{i+1} = x_i - \alpha \frac{g_i}{\sqrt{v_i}}, \tag{2.4}$$

where v_i is some type of average of g_i^2 over this and previous steps. With this method the step size made for each component of x is scaled in accordance with the magnitude of the previous gradients. Table 2.1 lists three algorithms that use such learning rate scaling alongside with their update rules for v_i.

Gradient descent algorithms have been extensively studied from theoretical point of view as well. For "vanilla" gradient descent (defined in (2.1)), it is known that the minimal magnitude of the gradient along the descent falls as $\mathcal{O}(1/\sqrt{N})$, where N is the number of steps [74]. This is true for L-smooth objective functions that are bounded from below if the learning rate $\alpha \leq \sqrt{3}/L$, where L is the Lipschitz constant. For adaptive variants of gradient descent the worst case guarantees are in general less favorable. For several adaptive variants (including Adam and RMSProp algorithms), the minimal magnitude of the gradient along the descent falls as $\mathcal{O}(\log(N)/\sqrt{N})$ under some reasonable conditions for the parameter values [75]. Note that it is possible that adaptive variants do not converge at all (even for convex objective functions) [76], but this can be avoided by an appropriate choice of parameters [75]. If we allow the learning rate α to change during descent (with a fixed decreasing schedule and some reasonable conditions on its magnitude), it is possible to prove that the adaptive variants of gradient descent match the convergence properties of "vanilla" gradient descent ($\mathcal{O}(1/\sqrt{N})$) and also that they converge almost surely [77]. Assuming a suitable learning rate schedule, it is even possible to prove the convergence of Adam-like algorithms for non-smooth (locally Lipschitz continuous) objective functions [78], which are often encountered in deep learning. In such setting Clarke subdifferential [79] is used instead of gradient to provide rigorous mathematical analysis. These results provide confidence that gradient descent algorithms converge relatively quickly, at least toward some critical point in the optimization landscape.

Gradient-based optimization algorithms are almost exclusively single-solution-based and, therefore, lack exploration power. There are, however, several hybrid versions of gradient-based optimization with evolutionary algorithms [80] and fall to the class of memetic algorithms meaning that the gradient method is only used to refine the individuals of the population and the evolutionary approach is used for global search rather than to use extensive gradient descent as the foundation.

If gradient-based optimization is to be applied to find an optimum of the objective function, one needs to evaluate both the objective value and the gradient for every candidate. There are several ways a gradient can be calculated, and which is the most efficient depends on the nature of the problem. We recognize three classes of gradient calculation methods:

1. symbolic differentiation,
2. numerical differentiation, and
3. automatic differentiation.

In symbolic differentiation, the computational graph for gradient calculation is constructed using standard differentiation rules, especially the chain rule, from the objective function definition. A computer algebra system and compiler can be used to construct this computational graph in case of complicated objective functions. Mathematica, Maxima, and Maple have efficient implementations for symbolic differentiation when the objective function has a closed mathematical form. It must be noted that symbolic differentiation can also be applied when the objective function is defined by a computer program, for example, including loops, conditionals, and computational behavior that depend on the input. Machine learning framework Theano is an example of such a library that can, for example, symbolically differentiate branching conditionals by lazy-evaluation scheme and loops of unknown length using symbolic scan loops [25]. The workflow of using Theano for gradient-based optimization consists of writing a program for objective function calculation using Theano programming constructs. Theano compiler then traverses the computational graph of this program and constructs another computational graph of how the gradient of the result of this program is calculated using symbolic differentiation rules. Both graphs are then compiled to `C++` code in a numerically stable way. The resulting code can then be freely run, taking a single candidate point in a search space as input and returning the objective function value alongside with the gradient in that point. This workflow can be used regardless of the complexity of the program that calculates the objective function and can be regarded as a general differentiation method for gradient-based optimization.

Numerical differentiation is the finite difference approximation of derivatives using values of the original function evaluated at some sample points. In its simplest form, it is based on the limit definition of a derivative:

$$\frac{\partial f(\mathbf{x})}{\partial x_i} \approx \frac{f(\mathbf{x} + h\mathbf{e}_i)}{h}, \tag{2.5}$$

where $\mathbf{e}_i$ is the i-th unit vector and $h > 0$ is a small step size. This has the advantage of being uncomplicated to implement and disadvantages of performing $\mathcal{O}(n)$ evaluations of f for a gradient in n dimensions and requiring careful consideration in selecting the step size h. Numerical approximations of derivatives are inherently ill-conditioned and unstable [81]. However, special schemes of numerical differentiation can be used in optimization setting when the objective function is especially noisy such as in using OpenAI ES for reinforcement learning [82].

Automatic differentiation (AD) is a derivation type where the intended form for expressing the objective function is a computer program. While symbolic differentiation can also be extended to work on programs, AD is designed for such description of functions. It exploits the fact that every computer program, no matter how complicated, executes a sequence of operations that are well understood from a mathematical point of view. By applying the chain rule repeatedly to these operations, derivatives of arbitrary order can be computed automatically, accurately to working precision, and using at most a small

constant factor more arithmetic operations than the original program. Two main types of AD are forward AD, where the chain rule is propagated in the direction from the inputs to the outputs of the program, and reverse AD, where the chain rule is propagated on the program computational graph in reverse, from the outputs to the inputs. Forward and reverse mode are just two (extreme) ways of traversing the chain rule. The problem of computing a full Jacobian with a minimum number of arithmetic operations is known as the optimal Jacobian accumulation problem, which is NP-complete [83]. It is known that reverse AD is more efficient when the number of inputs is much larger than the number of outputs, which is the setting of optimization, and forward AD is more efficient in the opposite case [84]. It must be noted that a recent proof shows that symbolic and forward AD are equivalent [85] and, in the modern view, the difference between the two boils down to the question of when the computational graph for the gradient is constructed. If it is constructed beforehand, at compilation time, we talk about symbolic differentiation, and if the computational graph for the gradient is constructed during the program execution, we talk about forward AD.

An objective function is usually represented as a computer program that gets a candidate (element of the search space) as input and returns the objective value(s). If the inputs are continuous variables, then any program that manipulates these values to produce objective value(s) can be differentiated using AD. Almost all programming language constructs (control flow, recursion, mutation) can be handled using AD, even some that seem discrete at first glance, such as finding a maximum of a list and index substitutions [86].

A real-world program has a finite number of branchings, for example, if clauses, which means that it is, mathematically speaking, a piecewise differentiable function. This means that it is possible that the gradient is not mathematically defined for some points of the search space. This fact fueled some reluctance toward using gradient-based methods in the past, even though the subset of the search space for which this is the case has measure zero. However, the gradient-based optimization becomes theoretically well-founded even for piecewise differentiable objective functions if we switch gradients for subgradients. With this, every program is differentiable on the entire domain. A subgradient is defined as any element of the convex hull of limits of gradients at nearby differentiable points [87]. Therefore, a subgradient can be chosen to be the gradient from any of the differential pieces the point borders to or their average [26]. A gradient-based optimization algorithm can be used without modification to work on subgradients instead of gradients, and in some cases, we also have proof of convergence in this setting [88].

This work centers around large-scale optimization problems (which include machine learning problems) because for those we have indications [13]–[15] that all local minima are very likely close to the global one. Given the nature of high-dimensional optimization problems, the central issue of gradient-based optimization is, therefore, plateaus near saddle points and not local minima. The deep learning community developed or rediscovered several tactics that improve the behavior of gradient-based optimization near saddle points and narrow valleys. In addition to advances in gradient-based optimization algorithms, deep learning also fueled the development of general differentiation libraries like Theano [25], TensorFlow [26], Caffe [27], CNTK [28], PyTorch [29], and so on. They implement either symbolic or reverse automatic differentiation of general and possibly very complex programs that may include condition-controlled loops, exceptions, conditional constructs, and so on. Even though these libraries were designed to differentiate neural network control flows, they are applicable for any program and, therefore, for optimization of any objective function described by a computer program.

Part I

Machine Learning

Chapter 3

Supervised Learning

A modeling task of finding a model where both the input and the output of the model are known is denoted as supervised learning. In this setting, a function is sought that when given known inputs as an argument returns values that are as close as possible to the known outputs. Below are four key ingredients that are needed for supervised machine learning:

1. Data set: A set of known inputs and corresponding outputs for the model.
2. Model: A class of functions with domain and codomain reflecting the properties of the data set (dimensionality and variable types of the inputs and outputs in the data set). The specific function that belongs to this class is determined by the model parameters which in turn determine the model behavior.
3. Metric: A (quasi)metric on the codomain space of the model that enumerates what "close" means for the specific problem in mind. For a given output of the model and the known, desired output from the data set it returns the error, cost, loss of the model.
4. Learning algorithm: An algorithm that sets the parameters of the model based on the given data set and the chosen metric. This is usually an optimization algorithm deployed to minimize the error of the model on the given data. This algorithm can be specialized to work well for a specific class of models or it can be a more general purpose optimization method.

The task of supervised machine learning is, therefore, a task of finding the parameters of the model that minimize the error calculated using the chosen metric on the given data set using the chosen learning algorithm.

The key parts of the supervised machine learning task mentioned above can be chosen to be rather generic and general or more customized in order to reflect more the underlying process that is modeled. For example, the chosen model can be a generic kernel machine or an artificial neural network which are known to work for almost every supervised machine learning task [89], [90] or more customized models that are structured specially for the given problem and include expert knowledge. It is known that customized models perform better than generic ones and require less data to generalize well [91] and can, depending on the design, bring superior explainability compared to generic ones. The learning algorithms deployed to train the model can also be more generic or more specialized. In the case of general-purpose machine learning models there is usually a set of commonly used and established optimization methods known to work well for a given model type. For example, greedy local search for training decision trees, interior-point method for training kernel

machines, and gradient descent for training artificial neural networks. However, for more customized models the optimization methods used for training are not that well explored, and, in many cases, more generic, metaheuristic optimization algorithms are deployed that are known to work satisfactorily for a wide range of optimization problems.

In this chapter we test whether gradient-based optimization methods can be used as a generic learning algorithm in supervised machine learning by using it on several different models ranging from highly customized, physics-informed models to generic neural network models. By this, we investigate if gradient-based optimization can be put by the side of common metaheuristic optimization algorithms usually deployed when training specialized models that are not well established. In the setting where models are customized for a specific task in mind common choices for training algorithms include differential evolution [92], covariance matrix adaptation evolution strategy [93], genetic algorithm [94], and so on. Gradient-based optimization, on the other hand, was not well studied as an option for a generic large-scale optimization in supervised machine learning in recent years. To test whether gradient-based optimization is a suitable option for a model training algorithm a real-world machine learning use case of forecasting residential natural gas demand was chosen since real industrial problems are the most eligible for assessing the practical usability of the methods.

It must be noted that supervised learning is methodologically the most appropriate setting for applying gradient-based optimization for model training. This is because machine learning models are usually functions with a closed mathematical form that are parameterized by numerical parameters. This means that the objective function used for training is well understood and can be easily differentiated. This is not the case in other chapters of this thesis where additional constructs were developed to rewrite the problems into a form that a problem from supervised learning already possesses. This is why it is also instructive to start with supervised learning (apart from it being the most common type of machine learning). There are, of course, also examples of supervised learning problems that do not fall under this broad category, however, other chapters of the thesis address these types of problems. Chapter 6 introduces methodology for dealing with models that do not have a closed mathematical form and Chapters 4 and 7 with models that are not parameterized by continuous variables.

A large part of the work reported in this chapter was also published in a conference paper dealing with the development and training of the physics-informed models [58], a journal paper where a comparison of a broader set of models for demand forecasting was performed [55], and a journal paper in which neural network models were more deeply explored [57]. All publications mentioned here were prepared in collaboration with Primož Potočnik from the Faculty of Mechanical Engineering at the University of Ljubljana, who contributed to data preparation and feature extraction. Our contributions were data analysis, model development and implementation, optimization (training of the models), and analysis of the results. In a review paper on advance computing approaches to support climate neutrality [57], we contributed to the energy forecasting use case, while the other two use cases were covered by Gašper Petelin and Vida Vukašinović from the Jožef Stefan Institute.

3.1 Problem of Forecasting Residential Natural Gas Demand in Ljubljana

In this section we present a real-world supervised machine learning problem of demand forecasting which will be used as a use case to assess the usefulness of gradient-based optimization for training both custom and more standard models in a supervised manner.

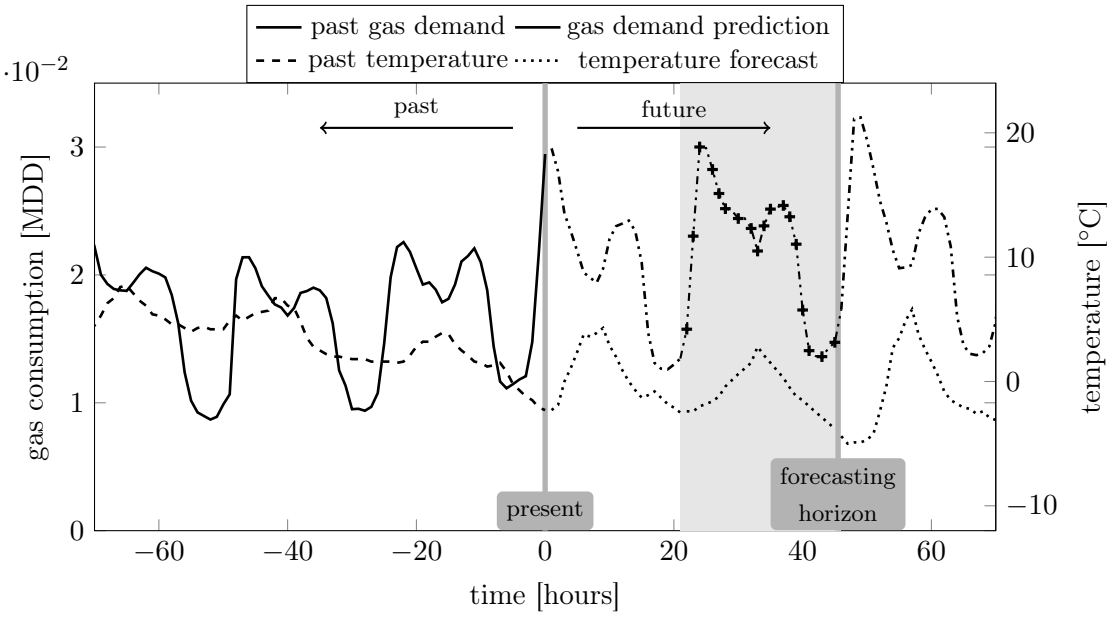


Figure 3.1: The problem of forecasting future gas demand. The shaded area is a 24-hour time range for the forecast, and the cross-shaped ticks are the desired output of the forecast model for each hour in that range. Reprinted from [55].

Forecasting natural gas demand is essential for efficient logistics in the energy sector especially for energy management and transportation. The issue of transporting natural gas via pipelines is a significant example of a situation where such predictions are required. Such transportation must be planned in accordance with the anticipated demand that will occur in a few days. While industrial gas demand can be planned in accordance with the timetable for each industrial client individually, this is not possible for the residential gas demand. This portion of gas demand must be forecasted using computational models. The task of modeling all of the factors that influence residential gas demand is extremely challenging. An accurate forecast model must take into account the physics of heat transfer and the accumulation of heat within buildings (because heating is the primary use of residential gas [95]) and socioeconomic factors which include the dynamics of the population's preferences for indoor temperature, hot water consumption, cooking, daily migrations, and so on.

We worked on the specific use case of forecasting residential natural gas demand in Ljubljana, Slovenia. To ensure the applicability of this work, the use case was handled as realistically as possible, considering characteristics of purchasing bulk natural gas in Slovenia for further distribution to the end users. In Slovenia, an order for a bulk gas purchase needs to be placed up to two days before its consumption, together with the quantity that will be consumed in the span of 24 hours. Since there are only minute gas reserves present in Slovenia, ordering an inaccurate quantity of gas results in considerable penalties. The aim of our modeling is to minimize the penalties considering the imposed forecasting scheme. The forecasting framework is illustrated in Figure 3.1 with temperature as an example input feature. The output of the forecast model is the gas demand within a 24-hour time range with an hourly resolution for a given forecasting horizon. The input of the model is a certain subset of the available time data, the past weather data, the forecasted weather data, and the past gas demand. The figure points out the important fact that the forecast model can only use measured variables from the past and forecasted

Table 3.1: Variables used by the models in this work. Variables *ph*, *sh* and *hw* are boolean whereas the others are real.

variable	meaning
P	gas demand
T	outdoor temperature
t_d	time of the day
t_w	time of the week
ph	presence of public holiday
sh	presence of school holiday
hw	day between public holiday and weekend

variables for the future (even though we may posses actually measured values, since we train on historical data). By using this forecasting framework, the error of the models is not underestimated and our setting is more in line with what is expected in the model deployment phase. Researchers working with this data set have even found that training on forecasted weather features leads to more accurate models in the validation phase [96]. They hypothesize that this is due to some unidentified biases in the weather forecast service that the model can correct for. This gives an additional reason for us using such a forecasting framework.

Details regarding the data set, feature extraction, and the metric suitable for this problem are described in [58]. Here we offer only a short summary. We operate with 8 winter seasons (spanning roughly from September to March) of gas demand data in an hourly resolution, which was provided by a major gas distribution company in Slovenia and includes values of total residential gas demand in the city of Ljubljana, Slovenia. Because the gas demand data is confidential, all figures and tables in this thesis present normalized gas demand values, expressed in units of maximum daily demand (MDD). Because of confidentiality we were also not allowed to publish the data set or the implementations of our models that used this data set for training. A comprehensive correlation analysis of gas, weather, and time data was made to extract the most relevant features. Outdoor temperature, together with time seasonalities, were found to be the most relevant features, see Figures 3.2 and 3.3. Full list of features used as the model input is in Table 3.1.

In this use case, the model's future gas demand predictions are not only evaluated for statistical accuracy but also for their direct economic impact. The gas distribution company orders gas several days ahead, and its costs are determined by how much the ordered quantity deviates from the actual consumption. This makes it possible to design and train our models using a metric that directly reflects the company's financial exposure, rather than relying solely on standard statistical metrics. However, since the exact pricing mechanism is confidential (we are allowed to only disclose the crude shape of it shown in Figure 3.4), we report results using the mean absolute error metric. Importantly, the true cost function is closely aligned with this metric in the regime of large deviations, which are precisely the errors that matter most in practice and where our model provides the greatest benefit.

3.2 Physics-Informed Semi-Empirical Models

It is known that customized models, such as those whose structure is specially tailored to the problem, perform better than generic ones and require less data to generalize well [91].

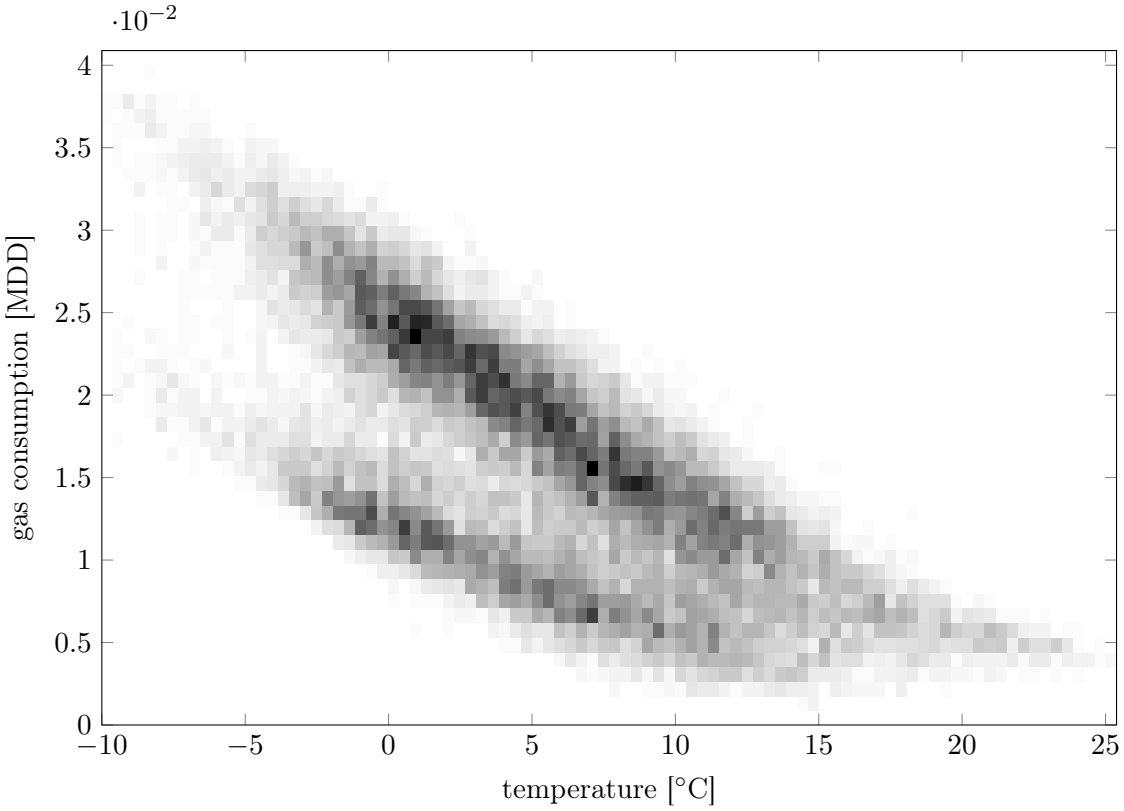


Figure 3.2: Density of data points for different values of hourly gas consumption and temperature. The two branches correspond to daytime and nighttime gas consumption behavior. Reprinted from [58].

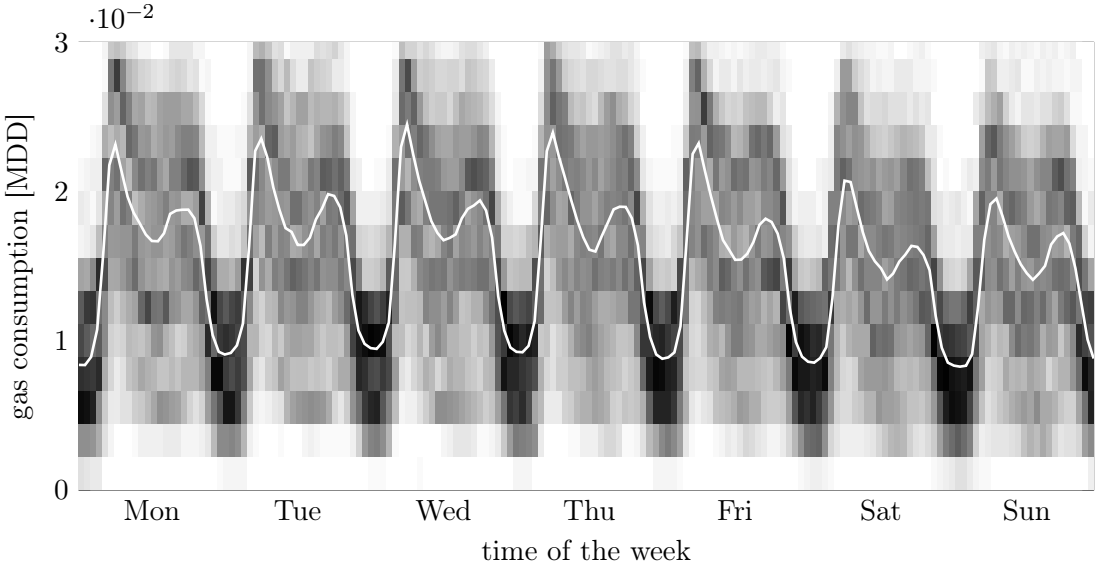


Figure 3.3: Density of data points for different values of hourly gas consumption and hour of the week. The white line is the median gas consumption with respect to hour of the week.

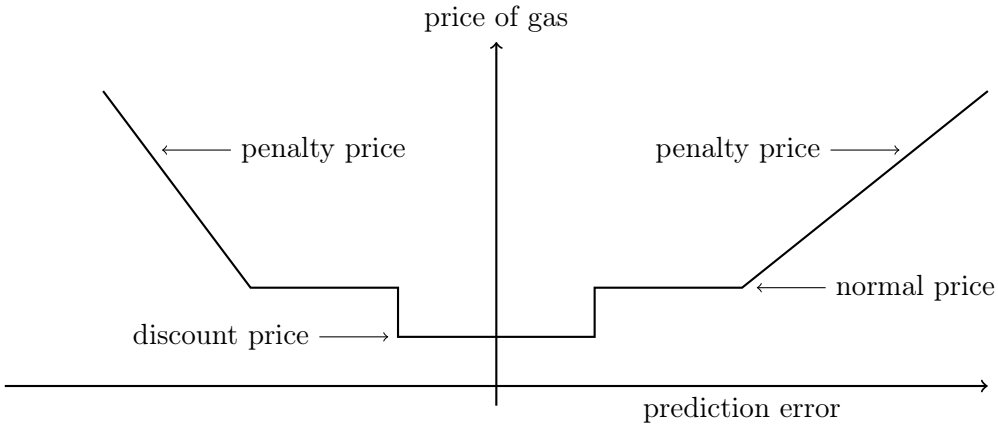


Figure 3.4: Depiction of the general shape of cost of gas with respect to prediction error (difference between forecasted and consumed quantity of gas). The exact way the price is calculated is confidential.

One way to construct such customized models is to include expert knowledge in its structure so that some properties of the system do not need to be learned by the model in the training phase but are already known, so fewer patterns need to be recognized. This difference in the number of patterns that need to be identified is the reason why less data is usually needed for training such models compared to generic machine learning models. Physics-informed model is a term designating a model that, in its structure already includes, to some degree, the underlying physics of the system in question. In this respect, the model cannot be trained in a way that violates the laws of physics, which restricts the parameter search space of the model and makes its optimization more straightforward. For example, in plasma turbulence control (for confined fusion reactors) it was observed that neural network models become considerably more accurate in predicting dangerous plasma breaches if a special scheme is used that constrains the predictions of the model so that they follow Braginskii two-fluid theory [97]. Another example comes from surface crack detection on metals, where it was found that the accuracy can be improved by enforcing acoustic wave equation into the neural network structure by using adaptive activation functions [98]. Physics-informed machine learning has been especially successful in solving inverse problems where it was applied in thermodynamics [99], quantum chemistry [100], and seismology [101].

The other possible aim of constructing a customized model is to make it more explainable. The importance of model explainability has been increasingly more recognized in recent years in the field of artificial intelligence because systems for which we can explain their decisions/predictions give higher confidence for their validity to the end users. It was shown that this confidence is a crucial factor in deciding whether to deploy machine learning in real-world applications as opposed to opting for a fully explainable expert system [102].

In this section, we present machine learning models whose structure stems from the physics of accumulation and transfer of heat in buildings (heating is the most considerable contribution to gas consumption) and from empirical observations about the data distribution of the Ljubljana gas demand data set. The models were constructed in a way that their parameters have a clear interpretation that humans can understand. This is in contrast to popular machine learning methods such as neural networks, tree ensembles, and kernel machines, which all belong to a black-box model category. The explainability of

different parts of the model brings opportunities for understanding how gas is consumed. Parameters of the trained model can tell us things like what are the statistical properties of heat transfer and its accumulation in buildings, how humans decide when to heat their homes, how they migrate daily, and many more.

Three types of models of increasing complexity were constructed, and, given their form, they do not belong to any established category of machine learning models. This means that, due to the lack of use in the literature, there is also no established training algorithm. Therefore, some general optimization method needs to be used to train the models, that is, to find model parameters that minimize the error of the model. We will test whether gradient-based optimization is a suitable choice for training such models by considering resulting model accuracy, training time, and efforts needed to implement gradient calculation which is an essential requirement in gradient-based optimization.

In this section, physics-informed semiempirical models (such that are based partially on theoretical and partially on empirical considerations) will be devised. Two aims will be followed to construct the gas demand models:

1. to include, in the structure of the model, the laws of accumulation and transfer of heat in buildings as they are related to space heating and, consequently, gas demand and

2. to structure the model in a way that would allow disentanglement of the contributions of different features, which increases model explainability.

The second point has to do with the troubles encountered with analyzing the Ljubljana gas demand data set, which was addressed in [58]. To more thoroughly understand the gas consumption behavior in a way understandable to humans, the dependency needs to be split into parts that depend on a small number of variables. For example, it was observed in [58] that the influence of temperature on gas consumption changes with time, and the dependence even has a different shape in daytime compared to nighttime. The aim is to form a model that would allow qualitative observations about the nature of what drives gas demand in Ljubljana as opposed to considering all possible features that were found to be entangled in a complicated manner.

3.2.1 Two-Reservoir Model

The following physics-informed semiempirical model was constructed by incorporating theoretical physics considerations described in [103] and empirical analysis of the Ljubljana gas demand data set conducted in [58]:

$$\begin{aligned} P &= n_w(t_w)n_d(t_d)f(T) + g(T) + u_w(t_w)u_d(t_d) \\ t_d &= t \pmod{1 \text{ day}} \\ t_w &= t \pmod{1 \text{ week}}, \end{aligned} \tag{3.1}$$

where P is gas consumption, t is time, T is outdoor temperature and n_i, f, g and u_i are unknown functions. These unknown functions have a clear interpretation which is outlined below. The structure of the model is based on the following observations:

1. the two temperature-dependent functions $f(T)$ and $g(T)$ were created because the influence of T on P is different at night than during the day (see Figure 3.2),

2. the general shape of the dependence of the hour of the day on P is similar regardless of the day of the week, therefore, the product form $n(t) = n_w(t_w)n_d(t_d)$ is a compact way to introduce the proper scaling due to day-of-the-week dependence, and

3. P is nonzero even at high temperatures, therefore, we introduce the temperature-independent term $u(t) = u_w(t_w)u_d(t_d)$.

The interpretation of the first two summands of Eq. (3.1) gives rise to the name of this model: the two-reservoir model (TRM). Each of the two temperature-dependent terms represents a heat reservoir with possibly different heat-transfer characteristics. The first reservoir is made up of buildings where only a percentage of them are heated, and the second of buildings with constant heating. The last term envelopes gas demand other than heating and is only time-dependent. Having a clear interpretation of unknown functions makes TRM more explainable.

The unknown functions n_i, f, g and u_i were parameterized as a linear combination of predetermined basis functions (differently positioned hard sigmoid and ramp functions). This means that Eq. (3.1) can be rewritten in an alternative form that is more useful for training so that all points in the data set are considered at once. In such form, all parameters of the model are aggregated in a single vector $\mathbf{p}$ and the summands of Eq. (3.1) are element-wise multiplications (marked with $\odot$) of matrix multiplications with $\mathbf{p}$:

$$\mathbf{P} = M_{n_w}\mathbf{p} \odot M_{n_d}\mathbf{p} \odot M_f\mathbf{p} + M_g\mathbf{p} + M_{u_w}\mathbf{p} \odot M_{u_d}\mathbf{p}. \tag{3.2}$$

Matrices M_i are constructed based on the temperature and time data and the choice of basis functions for parameterization of functions n_i, f, g, and u_i. This form is also more in accordance with the form in which most optimization problems are presented. To train the model, the following optimization problem needs to be solved:

$$\mathbf{p}_{\text{optimal}} = \arg\min_{\mathbf{p}} \left\| \mathbf{P}_{\text{real}} - M_{n_w}\mathbf{p} \odot M_{n_d}\mathbf{p} \odot M_f\mathbf{p} + M_g\mathbf{p} + M_{u_w}\mathbf{p} \odot M_{u_d}\mathbf{p} \right\|_1, \tag{3.3}$$

where $\mathbf{P}_{\text{real}}$ is the vector of measured gas consumptions (for our data set the characteristic dimension of $\mathbf{P}_{\text{real}}$ is $\sim 35,000$) and $\mathbf{p}_{\text{optimal}}$ hold the parameters of the ideal TRM that predicts gas consumption as close as possible to the measured ones (acknowledging the mean absolute error metric).

3.2.2 Two-Reservoir Model with Linear Memory

TRM can be upgraded by also considering heat accumulation in buildings. Following [103], the effects of heat accumulation in buildings (and also heat conduction through the ground) can be fully described by the use of a linear operator on the function of temperature dynamics to acquire heat demand. In discrete case, and acknowledging the time invariance of physical laws, this reduces to discrete convolution operation $*$ on a column of past temperatures with some unknown response function:

$$f(\mathbf{T}) = \mathbf{r}_f * \mathbf{T} \quad \text{and} \quad g(\mathbf{T}) = \mathbf{r}_g * \mathbf{T}, \tag{3.4}$$

where vector $\mathbf{r}_i$ holds the parameters of the model and is learned (see Figure 3.5 for visual depiction). Eq. (3.4) is equivalent to

$$(\mathbf{r}_f * \mathbf{T})(t) = r_f^0 T(t) + r_f^1 T(t-1) + r_f^2 T(t-2) + \dots, \tag{3.5}$$

where r_f^i are the components of vector $\mathbf{r}_f$ and constitute the new parameters of the model and $T(t)$ is the temperature in time t. These new calculations of functions f and g are still linear with respect to model parameters and, therefore, conform to the scheme presented in Eq. (3.2).

The model described with Eq. (3.1) where f and g are defined as in Eq. (3.4) will be referred to as two-reservoir model with linear memory (TLM). This model can model

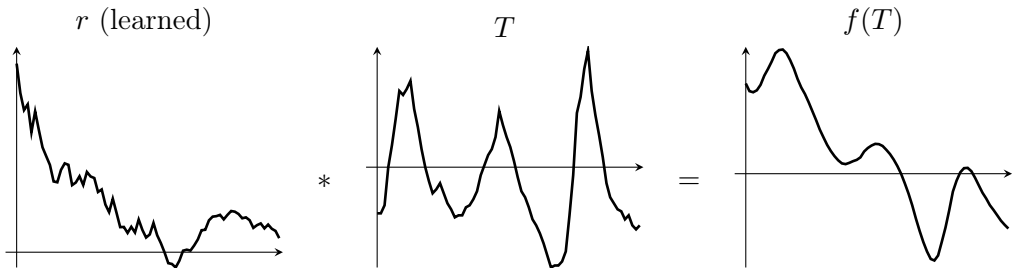


Figure 3.5: Example of how $f(T)$ is calculated in TLM using a discrete convolution on past temperatures with a learned response function.

memory effects that are known to be relevant [103]. However, it does so in a linear way and can not capture the nonlinear dependency seen in Figure 3.2. It is nonetheless a reasonable approximation and allows for explainable learning of memory effects in gas demand data. For further details on the arguments behind TLM structure, see [58], [103].

3.2.3 Two-Reservoir Model with Nonlinear Memory

To add nonlinearity to the temperature dependency of TLM while retaining memory effects, at the same time we need to compose nonlinearity application and convolution. In order to conform to the model structure set in Eq. (3.1), we can not parameterize both operations at the same time. We chose to have fixed convolution kernels while the nonlinearity is trainable:

$$f(\mathbf{T}) = \sum_i f_i(\mathbf{r}_i * \mathbf{T}), \tag{3.6}$$

where functions f_i are parameterized in the same way as in the TRM version and $\mathbf{r}_i$ are exponentially decreasing functions

$$r_i(t) = \exp(-t/\tau_i)/\tau_i. \tag{3.7}$$

See Figure 3.6 for a depiction of such a definition of the temperature-dependent part of the TNM. Because $\mathbf{r}_i * \mathbf{T}$ can be interpreted as temperature average on time scale τ_i, we can interpret Eq. (3.6) as a sum of nonlinearly mapped temperature averages on different time scales. The model employing functions f and g in this manner will be referred to as a two-reservoir model with nonlinear memory (TNM). For further details on the arguments behind TNM structure, see [58], [103].

3.2.4 Training the Custom Models

The three custom models for gas demand forecasting (TRM, TLM, and TNM) all have the same underlying structure. Their return value is calculated as a sum of element-wise products (marked with $\odot$) of matrix multiplication on their parameters:

$$\mathbf{P} = \sum_i \bigodot_j M_{ij}\mathbf{p}_{ij} = \sum_i M_{i1}\mathbf{p}_{i1} \odot M_{i2}\mathbf{p}_{i2} \odot M_{i3}\mathbf{p}_{i3} \odot \dots \tag{3.8}$$

Models of this type were not extensively studied, so there is no established algorithm for training them. Therefore, some generic optimization algorithm needs to be used to find the values of parameters $\mathbf{p}_{ij}$ in order for the resulting gas demand $\mathbf{P}$ to be as close as possible to the measured one.

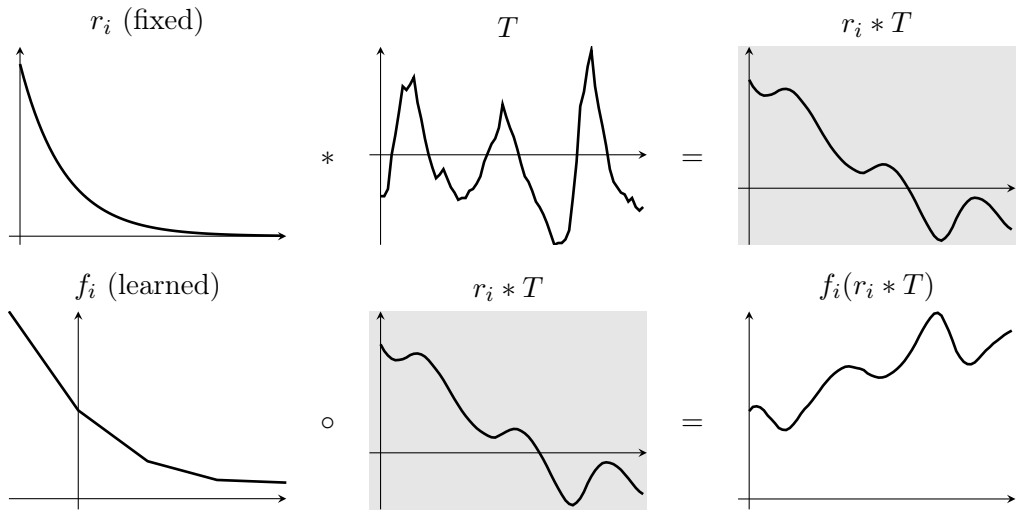


Figure 3.6: Example of how $f(T)$ is calculated in TNM using fixed convolutions and learned nonlinear functions to result in a model that includes both nonlinearities and temperature history. Symbol $\circ$ represents function composition.

With proper implementation, the calculation in Eq. (3.8) can be extremely fast, however, this calculation is nonlinear with respect to the parameters. Consequently, searching for the optimal values of the model parameters requires nonlinear optimization techniques, and we will test both derivative-free and gradient-based algorithms. The gradient of the error with respect to $\mathbf{p}$ can, due to the closed form of the problem, be calculated analytically using symbolic differentiation. An important practical aspect of this problem is that it is possible to construct a good initial point in the search space (as detailed in [58]). This is achieved by fixing different subsets of the parameters and, for each subset, solving a linear regression-type subproblem. Specifically, when the other parameters are fixed, the objective function becomes quadratic in the active subset of parameters, and the optimal solution can be obtained efficiently (by solving the corresponding linear system). Since our actual loss function is the mean absolute error, this quadratic treatment is an approximation, but it still provides a computationally inexpensive and effective way to generate high-quality initial points. The main workload of this initialization procedure is therefore solving these linear regression subproblems, which can be done very efficiently. From the initial point in parameter space we need to continue the optimization considering all parameters at the same time. Since the mean absolute error is used as a metric, the gradient is not defined everywhere. For this reason, a subgradient is used (mentioned in Chapter 2) with the definition $\mathrm{abs}'(0) = 0$. In this way, gradient-based optimization can be deployed in a mathematically rigorous way.

We tested three optimization algorithms for training our models:

1. Differential evolution (derivative-free, population-based),
2. Nelder-Mead algorithm (derivative-free, single-solution-based), and
3. Quasi-Newton method (gradient-based, single-solution-based).

We chose those specific algorithms in order to cover all the major categories of the popular metaheuristic algorithms. Differential evolution is a population-based algorithm, while the other two are single-solution-based. Nelder-Mead algorithm is derivative-free, while

the Quasi-Newton method is gradient-based. We used default configurations for these algorithms as implemented in Matlab. In this way, without any parameter tuning, the comparison is fair from the perspective of a user from industry. For all three algorithms, we used approximation from the previous paragraph as an initial point. The initial population for Differential evolution was sampled from the isotropic normal distribution centered around this initial point (the point itself was also directly added to the population). For the Nelder-Mead algorithm the initial simplex was chosen to also be centered around that initial point. All three algorithms were run on hardware with an Intel i7-4500U processor and with 8GB of RAM. Further details on the implementation of the experiments are described in [58].

Unlike the other two methods, the Quasi-Newton method requires the gradient of the objective function with respect to the model parameters to be supplied in each function evaluation. The gradient for the Quasi-Newton method was calculated using symbolic differentiation. The gradient calculation brings some overhead to the function evaluation, however, the computation time of the Quasi-Newton algorithm is, nonetheless, due to faster convergence, orders of magnitude smaller compared to Nelder-Mead and Differential evolution. Table 3.2 reports the training times. We can see that the Quasi-Newton method requires only seconds before converging while Nelder-Mead requires minutes. Since Nelder-Mead, to some degree, follows gradient descent logic but in a derivative-free way, we can see here the power of having an efficient way for gradient calculation. Differential evolution, on the other hand, has more capacity for exploration, however, in higher dimensions this can bring a high penalty in runtime.

Each model was tested on forecasted weather data. In this way, the models were tested as they would be used in a realistic setting, as depicted in Figure 3.1. In this study, each season is treated as an independent observational unit, with summer months representing a natural interruption in gas demand patterns. Consequently, classical k-fold cross-validation is applied across the eight seasons, training on seven and testing on the remaining one in each fold. This ensures that all folds have an identical training set size, mitigating bias associated with varying data volumes [104] and providing a more consistent and robust estimate of forecasting error. Moreover, due to the seasonal interruption, the temporal ordering of the test set relative to the training set is less critical and each winter season can be considered approximately independent. As a result, the test set does not necessarily need to follow the training data chronologically, since the summer gap effectively resets the system, reducing temporal dependencies between folds and justifying the use of non-sequential cross-validation. This type of error estimation should be close to the error of the model if it were used in real life.

The Quasi-Newton method does not only converge faster, but it also produces the most accurate models (with respect to both training and test error). Figure 3.7 shows the model errors calculated with cross-validation for the three models considering different forecasting horizons in accordance to the modeling framework shown in Figure 3.1 for all three training methods. Differential evolution was tested only on the TRM due to very long training time[1]. This shows that gradient-based optimization found parameters for the model that is accurate in practically negligible computation time compared to other algorithms we tested. By this fact we can confirm hypothesis **H1** for the specific case of the three custom models on the given data set.

[1]To produce Figure 3.7, the model needs to be trained 312 times – for 38 different forecasting horizons and for 8 different cross-validation folds.

Table 3.2: Training times for the three models when using the Quasi-Newton method, Nelder-Mead algorithm and Differential evolution. Differential evolution was applied only to the smallest model due to its long runtime.

model	#parameters	mean training time [s]		
		Quasi-Newton	Nelder-Mead	Differential evolution
TRM	81	1.1	36	2665
TLM	209	1.6	426	–
TNM	177	3.6	534	–

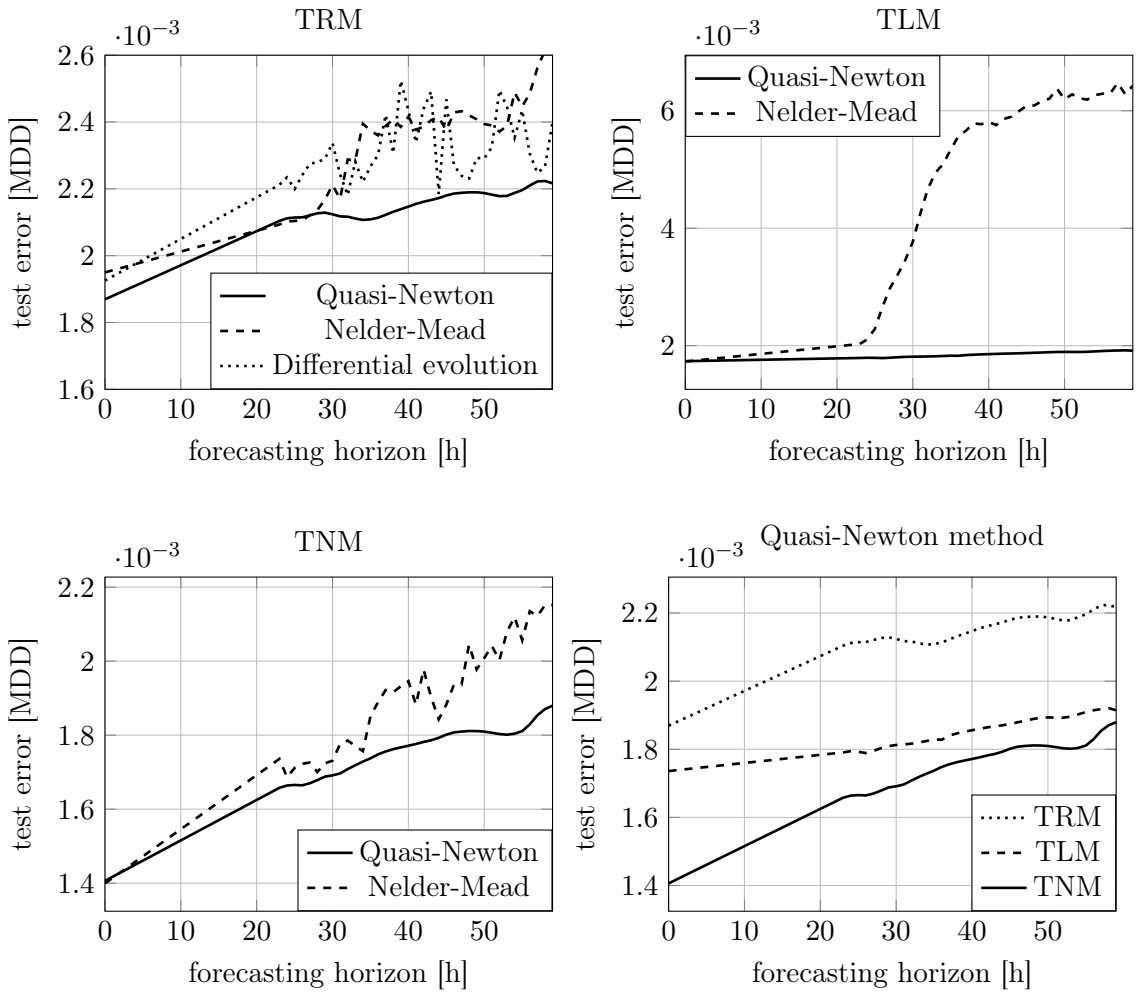


Figure 3.7: Comparison of test error for models when trained by gradient-based Quasi-Newton method, gradient-free Nelder-Mead method, and differential evolution for different values of the forecasting horizon. Differential evolution was applied only to the smallest model due to its long runtime. The last graph compares the errors for all three models when the Quasi-Newton method is used, which was found to bring better accuracy for all models and forecasting horizons.

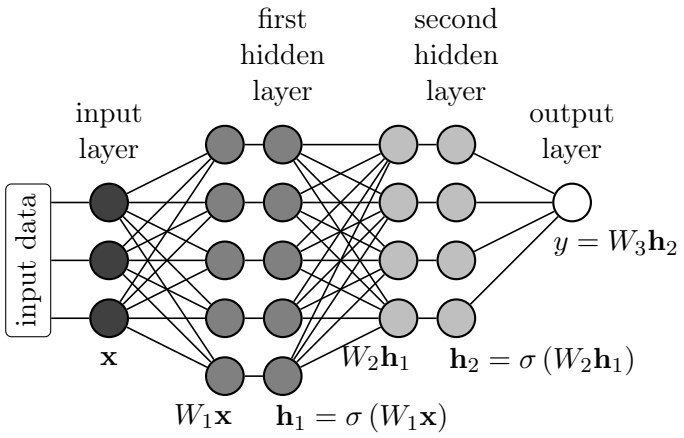


Figure 3.8: Depiction of feedforward artificial neural network model with two hidden layers where every step of model calculation is explicitly shown. W_i are matrices and σ is element-wise nonlinear function. [57]

3.3 Neural Network Models

The second class of models we apply to the gas demand forecasting problem are artificial neural networks. They belong to the black-box category of machine learning and, therefore, do not offer explainability. They are, however, known to be of extremely high capacity, and their training usually results in very accurate models. It is known there is a trade-off between model accuracy and explainability [105]. If in the previous section we were dealing with more explainable and less accurate models, we now move toward less explainable and more accurate models. Neural networks are in the last decade trained almost exclusively by gradient-based optimization which makes them a good choice for this thesis. Modern training implementations use automatic differentiation for gradient calculation which is in this case more efficient compared to symbolic differentiation especially when recurrent neural networks are deployed, which will be the case in this section.

The work presented in this section was also reported in [55] and [57]. It must be noted that this section is more interesting from the machine learning point of view and less from the optimization point of view. Since gradient-based optimization is already a well-established training method for neural networks, it is not surprising that gradient descent algorithms work well in this setting. It is nonetheless an example of gradient-based optimization in supervised machine learning and neural network models complement the physics-informed ones from a machine-learning point of view. Here we offer only a brief summary of results that outline how different types of neural network models are trained with gradient-based optimization using the same gas consumption forecasting use case.

Artificial neural network (ANN) is a machine learning model that consists of a composition of parameterized linear maps and fixed element-wise nonlinear maps interchangeably. An example of one such model is depicted in Figure 3.8. It is known that ANN can approximate arbitrary functions to an arbitrary precision [90]. This means that ANN can capture any function if ANN is large enough. If there exists a function that takes available data as input and returns gas consumption, a big enough ANN can approximate this function. It is only a question of our ability to find those good ANNs using optimization techniques. If ANN model consists of several applications of linear and nonlinear maps, we talk about a deep neural network. The depth of ANN significantly improves the capacity of the model, which means that deeper ANNs are able to capture a wider range of functions [106].

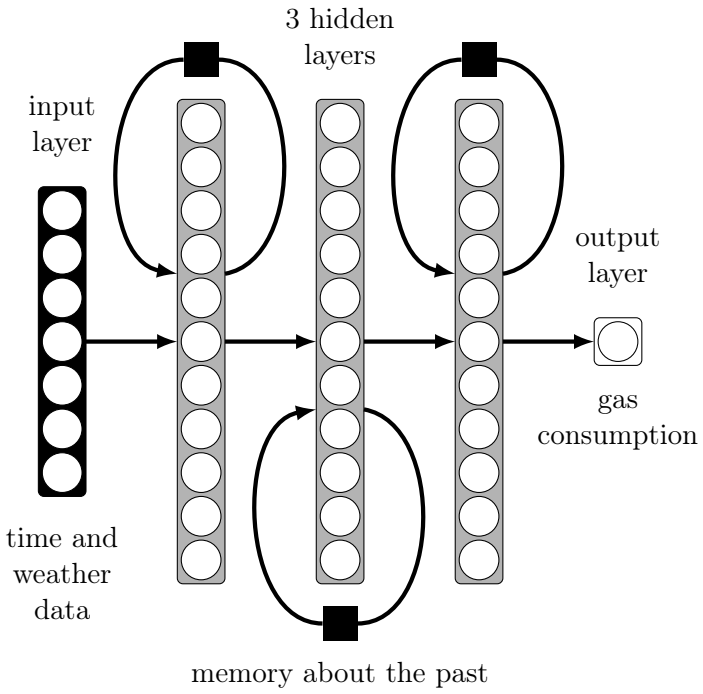


Figure 3.9: Example of RNN model architecture where all layers and units are explicitly shown. Connections between layers are shown with arrows. Hidden layers consist of reccurrent units which need values from the previous time step. For this purpose, one step delay is represented with a black square. Reprinted from [57].

For the forecasting use case studied here, the ANN model estimates gas consumption from several days of past weather data (accounting for heat accumulation [50]) as well as other relevant features, such as temporal and calendar information. Consequently, a sufficiently wide ANN is required to process multiple days of input simultaneously. However, since the number of ANN parameters grows quadratically with width, increasing the width too much can quickly lead to overfitting. There is, however, another way for ANN to know of past data. A recurrent neural network (RNN) can be used, an example of which is depicted in Figure 3.9. Such ANN includes feedback loops that allow the network to keep a memory of previous inputs. Therefore RNN can be fed one weather data instance at a time, one for each time step in a season. This way, ANN with a small width (and therefore a manageable number of model parameters) can be used that still takes past data into account. Therefore, the RNN type ANN is used in this section since it is the most suited for this problem. We tested three different RNN architectures: simple recurrent network (SRN), long short-term memory (LSTM) and gated recurrent unit (GRU). More information on the models and the comparison can be found in [57]. Unlike the models presented earlier, our RNN also incorporates past gas consumption as an input feature. Previous models were manually designed based on physical and empirical knowledge, making it difficult to incorporate past gas demand features, whereas the RNN can account for historical usage and, if necessary, adjust its forecasts when past demand deviates from the model's predictions.

3.3.1 Training the Neural Network Models

Neural network models were trained using the root mean square propagation (RMSProp) algorithm [72] which is a gradient descent variant that employs per-parameter learning rate adaptation and is currently the default algorithm in many deep learning libraries. The gradient of the error with respect to network parameters was calculated using symbolic differentiation with the Theano library [25]. Theano is a Python library that allows to define, optimize, and evaluate computational graphs involving multi-dimensional arrays efficiently. It is essentially a mathematical compiler (CPU and GPU) for the generation of efficient and stable code for gradient calculation by symbolic differentiation technique. Using Theano, the user needs to provide only the code for objective function calculation while the code for gradient calculation (and optimization) is generated automatically. In this case, we provided the code for the structure of the model (how the model output is calculated) and how to calculate the error of the model. Theano took care of the differentiation and optimization, even taking care of graph unfolding since the data instances are time series of unequal length. This means that the gradient calculation graph was dynamic, for example, involving a while loop which accumulates contributions to the gradient for each time step until the time series instance runs out of time steps.

RNN models are known to be more difficult to train compared to classical feed-forward ANN models [107]. This is in part due to more computationally expensive objective function evaluation (a single data instance is one season of gas consumption) and in part due to the fact that RNN's behavior is more sensitive to changes in network parameters (optimization landscape can change abruptly). This is why RNNs are usually trained for much longer than classical ANNs. In our case, we found that training them for 10,000 epochs was sufficient for convergence. Due to such computationally extensive training, the optimization was run on a cluster with 2 Intel Xeon E5-2680 v3 processors and 1 TB of RAM with available GPU Tesla K80. Training times are reported in Table 3.3. Because each training can last for hours, we restricted our analysis to only a 24-hour forecasting horizon. Otherwise, the evaluation methodology is identical to the one reported in Section 3.2. It is interesting that the implementation run on GPU was, in fact, slower than the one run on CPU. We suspect that this is due to sequential nature of RNN operations.

From an optimization point of view, we observed one interesting phenomenon concerned with the convergence of the RMSProp algorithm. It seems that RMSProp is drawn to minima that not only have a low training error but low test error as well. In other words RMSProp is inclined to converge toward wide minima rather than toward narrow ones. In machine learning, there is an empirically well-studied observation that wide minima generalize better (have low test error) compared to narrow minima in which overfitting is likely to occur [108]. Figure 3.10 shows two characteristic convergence paths of the RMSProp algorithm for our training use case. The first is standard convergence we observe when the model trains without overfitting (test error is slightly higher than training error but they both have the same general shape). The second, however, shows how RMSProp guides the model toward overfitting. In the middle of optimization, the test error starts to increase, meaning that RMSProp is exploiting a narrow minimum representing a model that is very accurate on the training set but gets increasingly inaccurate on the test set. However, RMSProp does not get stuck in this narrow minimum. Further optimization moves the model away from this area of overfitting, and RMSProp starts to converge toward a region with a wider landscape resulting in much better model generalization and consequently lower test error. It must be noted that in all RMSProp optimization runs performed in this use case (240 in total) there were no instances of overfitted models that were not resolved by the end of training. This means that RMSProp found high-quality wide minima with an extremely high success rate.

Table 3.3: Training time of RNN models in hours for different unit types and for different hardware with respect to number of layers. All models used have an overall number of parameters equal to 4000 and are therefore comparable. One training comprises of 10,000 epochs or $\approx$ 70,000 gradient descent steps.

#layers	on CPU			on GPU		
	SRN	LSTM	GRU	SRN	LSTM	GRU
1	1.1	2.0	2.2	6.0	11.3	16.4
2	1.6	2.9	3.5	9.9	20.6	30.2
3	1.9	4.0	4.7	13.8	28.6	42.8
4	2.3	5.0	5.9	17.7	36.5	54.5
5	2.6	6.0	7.0	21.6	44.7	68.3

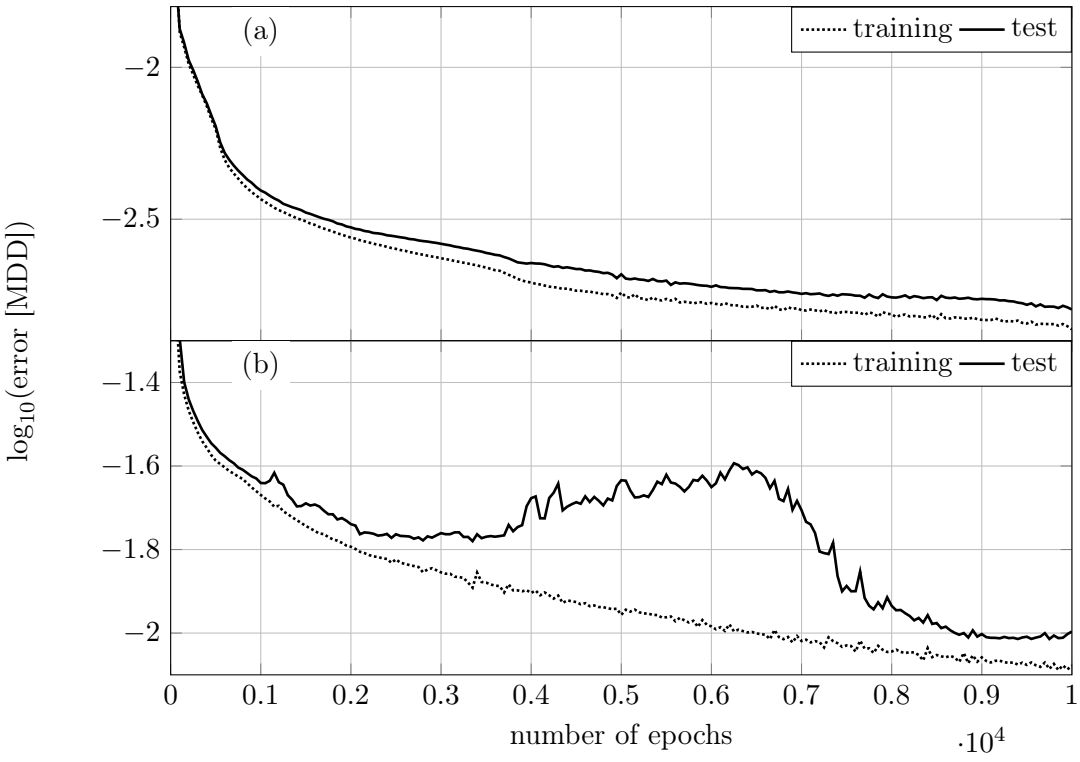


Figure 3.10: Convergence of training and test error during optimization when using RMSProp algorithm. (a) An example of the standard training progress without overfitting where test error is slightly higher than training error. (b) An example of overfitting in the middle of training that was resolved after continuing optimization.

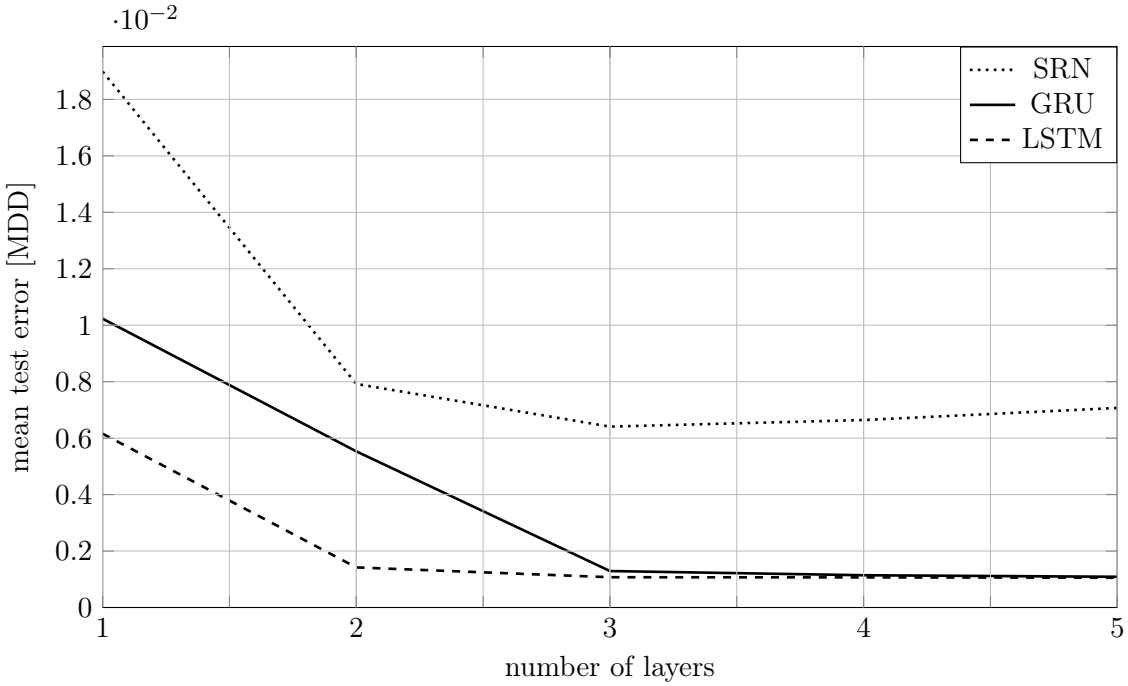


Figure 3.11: Test error of RNN models for different model architectures with respect to the number of layers. All models used have an overall number of parameters equal to 4000 and are therefore comparable.

To compare the accuracies of different model architectures after training (optimization), it is fair to compare models with the same number of parameters (same search space dimension). Because different unit types (SRN, LSTM, and GRU) have quite different structure this is extremely difficult to guarantee, especially if we are interested in how the depth of the model influences its accuracy. To accomplish a fair comparison for each unit type and layer count, two models were constructed with different layer widths. The two models had parameter counts just below 4000 and just above 4000. So, a theoretical error of the model with 4000 parameters was approximated to be a weighted sum of the two models whose parameter count is closest to 4000. In this way the model instances are really comparable. The accuracies of models for different unit types and different numbers of layers are depicted in Figure 3.11. As expected, the SRN models were found to be the least accurate. This can be attributed to the inability of SRN to account for long-range dependencies. For a low number of layers, the LSTM unit is clearly superior, while for deeper architectures, the LSTM and GRU units are comparable.

3.4 Discussion

In this chapter, we used gradient-based optimization to train different types of supervised machine learning models. In all cases, the search space was high-dimensional (ranging from 80 to 4000 model parameters). In this setting, we observed that gradient-based algorithms indeed did not get stuck in "problematic" local minima as was expected based on theoretical considerations [13]–[15]. This is indicated by models acquiring low training error. It also turned out that training with gradient-based optimization guided models toward regions of the search space with low test error. This is extremely important for supervised machine learning use cases since it prevents model overfitting. From the point of

view of the optimization landscape the gradient-based algorithms are much more likely to converge toward wider minima compared to narrower ones. This is not only supported by the fact that trained models acquired low test error but also by the fact that overfitting of the models was in fact observed in our experiments during training and that it was resolved by further optimization (see Figure 3.10). This means that in cases where gradient descent led models to narrow valleys it also guided them out of those valleys, to wider regions of the landscape (and this by only going downhill).

Optimization algorithms used for training supervised machine learning models are diverse and usually specialized for a specific class of models. When a new type of model is designed (that incorporates expert knowledge of the underlying problem) there is no clearly evident choice for an algorithm that would efficiently train those models. We show in this chapter that gradient-based optimization algorithms can be regarded as a generic general-purpose training algorithm as long as the search space is large enough. This would put them alongside other general-purpose optimization algorithms used for training this type of models such as differential evolution [92], covariance matrix adaptation evolution strategy [93], genetic algorithm [94], and so on. Our custom physics-informed models which were specially designed to offer better qualitative explanations for the patterns of gas consumption were trained most efficiently by gradient-based algorithm. When considering the gas demand use case we can, without a doubt, confirm hypothesis **H1**. Compared to other general-purpose optimization algorithms gradient-based one was orders of magnitude faster (see Table 3.2) and even found the most accurate models (see Figure 3.7).

This chapter is methodologically speaking quite undemanding because objective functions used for training are well-behaved and have a closed mathematical form. There was no need to develop any novel constructs to deal with discrete variables, constraints, or gradient calculation. Symbolic differentiation was used to acquire the gradient[2] of the objective function and standard gradient-based optimization algorithms were run to optimize the error of the models. Since all the encountered objective functions had a closed mathematical form symbolic differentiation has a similar time complexity compared to automatic differentiation. It is also an option that we are usually more accustomed to and leads to less overhead when conceptually designing the implementation. Implementing gradient calculation can be laborious and error-prone if done by hand, however, using a symbolic differentiation engine, such as Theano, can completely dissipate any implementation overhead. Theano can generate stable and efficient code (with built-in parallelization and GPU support) for gradient calculation without any intervention by the user. Therefore, to use gradient-based optimization in this setting the user only needs to invest in learning how to use such automatic code generation frameworks. As we have seen, when search space is large enough, this investment can bring superior models and a reduction of training time compared to other general-purpose algorithms. We believe that current state-of-the-art evolutionary algorithms are a good choice for supervised learning, however, only if the number of model parameters is small enough. When the number of model parameters comes close to ≈ 100 we believe gradient-based optimization should be considered as a legitimate option. In this sense the two optimization methodologies are complementary and each should be used in their own specific setting.

This chapter also presents interesting machine-learning results that are of great importance in energy demand forecasting. In Table 3.4 we can see how different types of models compare with respect to accuracy and training time. The custom explainable models (TRM, TLM, TNM) achieve reasonable accuracy but are not as accurate as the generic state-of-the-art model (RNN). They are, however, much easier to train and offer

[2]Strictly speaking, a subgradient was used, since the metric used is the mean absolute error, which is not differentiable at zero. The subgradient is computed using the convention $\text{abs}'(0) = 0$.

Table 3.4: Mean absolute error (MAE), mean absolute percentage error (MAPE), and training time of the tested models when hourly resolution is required (hour) and when daily resolution is required (day). All models were trained using gradient-based optimization and using a data set for a 24-hour forecasting horizon.

model	MAE [10^{-3} MDD]		MAPE [%]		training time [s]
	hour	day	hour	day	
TRM	2.17	32.9	19.8	11.2	1.1
TLM	1.84	32.5	15.4	10.5	1.6
TNM	1.66	25.2	15.8	8.6	3.6
RNN	1.06	18.3	9.3	6.8	21,545.3

much greater level of explainability. Observation about the tradeoff between accuracy and model explainability [105] holds for our forecasting models as well. It is also important to note that the difference in training time is very large. This means that abandoning explainability for a small improvement in accuracy results in much more extensive training which might not be practical in many supervised machine learning use cases.

Chapter 4

Unsupervised Learning

A modeling task where unlabeled data is processed with the aim of acquiring a more compact representation of the data distribution is termed unsupervised learning. Loosely speaking, in unsupervised learning, we are mining the data in order to uncover its properties which in turn offers us a better understanding of the data we are dealing with. In this setting, we are working with untagged, unlabeled data, and we are trying to detect patterns or more general properties attributed to this data set. The use of unsupervised learning usually falls into either of the three categories:

1. modeling probability distribution of the data set,
2. dimensionality reduction, and
3. clustering.

Nowadays, tasks under the first two points are oftentimes solved by deep learning [109] which uses gradient-based optimization for model training by default. Clustering, on the other hand, has been largely untouched by gradient-based methods and will be the main focus of this chapter.

It is important to note that all three categories of unsupervised learning are, nonetheless, related among themselves. For this thesis, the connection between dimensionality reduction and clustering is especially important, which is why our introduction covers both of them. Dimensionality reduction is a class of methods that attempt to reduce the number of relevant features in the data set. For example, if we have a data set where each data instance is described by 100 features, the aim is to find a data representation where data instances can be effectively described by a smaller number of features, say 20 instead of 100. If we succeed in this, we have found a 20-dimensional subspace embedded in a 100-dimensional feature space and a large majority of data points reside near this embedded subspace. Now, instead of describing a data instance using 100 features, we can use the 20 coordinates defined on the embedded subspace with, if the model is very accurate, negligible information loss[1]. The most basic example of such a method is principal component analysis (PCA) which can find linear subspaces in the full feature space [111]. However, in real-world data sets, relevant subspaces are usually not linear and can even be topologically complicated manifolds [112]. This is why deep learning is oftentimes used in such cases [113], [114], because deep neural networks are able to capture highly complicated geometrical patterns. Somewhere between PCA and deep learning is the nonnegative matrix tri-factorization (NMTF) method which is not linear, however, not as strong as deep learning (or as time consuming). NMTF and its variants will be pivotal for our work with clustering.

[1] Models can also be designed so that this information loss is useful, for example, to denoise data [110].

The term clustering, on the other hand, refers to the grouping of data instances. The goal of clustering is to assign each data instance to clusters in a way that similarity among instances that belong to the same cluster is high compared to the similarity between instances that belong to different clusters. Clustering is distinct from supervised classification that also groups data instances into classes, however, it does so based on the class labels that are included in the data set. In clustering, however, the data set is assumed to be unlabeled, meaning that the ideal grouping of the instances is not known beforehand and must be found by other means. In other words, clustering groups unlabeled data to ensure higher intra-cluster similarity and lower inter-cluster similarity [115]. It must be noted that the similarity between two data instances is not a universally well-defined concept. What can be considered a similarity measure for a given data set strongly depends on the problem formulation, structure of the feature space, and the underlying data distribution. For numerical features, similarity measures are usually related to spatial distance between the instances in the feature space, represented as vectors. Common examples include Euclidean, cosine, Levenshtein and Manhattan distance [116].

In this chapter, we make an effort toward transforming clustering problems using a formulation where gradient-based methods can be used for model training. This is not a trivial task since clustering problems involve decisions of whether a given data point belongs to a given cluster and is, in its foundation, a discrete optimization problem. This discrete nature makes the optimization landscape for training clustering models non-differentiable (locally flat with jumps). Therefore, in this setting, gradient-based optimization is an unusual choice, to say the least. However, there is an alternative formulation of clustering problems: as a limiting case of dimensionality reduction with additional constraints. When one thinks in this fashion, clustering is really an extreme case of dimensionality reduction where data instances are reduced in dimension to such a degree that their values are simply a measure of how much a data instance belongs to a given cluster. If the cluster centroids are well separated, we end up with a standard (hard) clustering. However, if clustering centroids are overlapping, we end up with soft clustering where, unlike in hard clustering, we allow a given data instance to belong to more than one cluster at the same time. Soft clustering is not just of theoretical interest, as a mathematical tool, but has real-world applications. For example, when clustering genetic data, it is quite usual that one gene contributes to several different processes in a living organism and not only to a single one. A more everyday example is also the use of soft clustering in recommender systems where a single person might like both action movies and romantic comedies and, therefore, strict separation of clusters is not reasonable.

Another machine learning task connected to clustering, that falls under supervised learning, is also similarity learning. This is a task of finding a model able to predict how similar two data instances are. However, in similarity learning, the similarity values for each pair of data instances are included in the data set. Therefore, a model can be trained in a supervised fashion to predict similarities included in the data set as accurately as possible. Connection to clustering comes from the fact that clustering, in its loose definition, is to partition a data set to instances that are similar among themselves. A good model for measuring similarity among data instances allows us to partition a data set into groups of similar instances which is equivalent to clustering. This loose definition of clustering allows us to apply this methodology in areas where clustering is not that well defined, such as clustering the nodes on a graph, which is becoming very important with the ever-increasing availability of data with graph-like properties. In this chapter, we will touch on several interconnected topics in unsupervised learning mentioned earlier and test how model training can be achieved using gradient-based optimization and especially what kind of adaptations and tricks need to be employed to make that happen.

A large part of the work reported in this chapter was also published in a journal paper dealing with the development of algorithms for optimizing NMTF objective function [56] and in the following series of conference papers/abstracts and book chapters on: multi-objective NMTF [60], [63], [66], development of algorithms for multi-type NMTF [67], development of algorithms for hard clustering by NMTF [61] and clustering scientific bibliographic data [68]. All publications mentioned here were prepared in collaboration with co-authors. In [56], [67] we contributed to the data preparation of (one of three) data sets, the development and implementation of (two of four) algorithms, and the preparation and analysis of results, Janez Povh from the {Rudolfovo – Science and Technology Centre Novo mesto made contributions of equal scope across these tasks, Gašper Petelin from the Jožef Stefan Institute contributed to the implementation of (one of four) algorithms, Vida Vukašinović from the Jožef Stefan Institute contributed to the analysis of the results, and Nataša Pržulj from the MBZUAI contributed to the preparation of (one of three) data sets. In [61], [68] we contributed to the development and implementation of (one of two) algorithms, and to the preparation of results, Janez Povh made contributions of equal scope across these tasks, and Andrej Kastrin from the Faculty of Medicine at the University of Ljubljana contributed to the preparation of the data sets, and the analysis of the results. In [60], [63], [66] we contributed to the development and implementation of algorithms, and to the preparation and analysis of results, while Gašper Petelin and Vida Vukašinović contributed to the initial investigation of multi-objective algorithms for the problem and to the analysis of the results.

4.1 Nonnegative Matrix Factorization and its Connection to Unsupervised Learning

Suppose we have a data set with n instances, each described by m numerical features, and the task is to better understand the data set by finding out how the data is distributed in the feature space, whether some data instances are clustered to smaller regions of the feature space and where are those regions. We can pack the data set in a $n \times m$ matrix R whose rows are data instances and columns represent different features. It is worth noting that practically all data sets can be represented in such a fashion. For example, if the data set includes nonnumerical, discrete features we can transform them to numerical ones using, for example, one-hot encoding. Also, if data instances do not have the same number of features, we can use missing values to fix the number of features. Such matrix encoding may make the data set less space efficient, however, it does not reduce its information content.

One of the approaches to better understand our data set R is to reduce its feature dimensionality m using dimensionality reduction and by this transform the existing features to new ones that have as little correlation among them as possible. PCA is a classical method to be employed for such a task, and many machine learning pipelines employ PCA as the first data processing step that extracts the most relevant features and minimizes correlations among them (feature extraction). It works by employing low-rank approximation by decomposing the matrix R, usually using singular value decomposition (SVD), and dropping the columns/rows in the decomposition that are less important (have small singular values):

$$R \approx GH^\top, \qquad \begin{aligned} &R \in \mathbb{R}^{n\times m} \\ &G \in \mathbb{R}^{n\times k} \\ &H \in \mathbb{R}^{m\times k}. \end{aligned} \tag{4.1}$$

The point behind this is that if the approximation is close to equality and $k \ll m$, matrix G

holds the same information as matrix R and we can use matrix G as our data set instead of R and, by this, reduce the number of features from m to k. SVD can be efficiently performed using state-of-the-art methods such as the Augmented Lanczos bidiagonalization algorithm, especially when the chosen k is small. The implicitly restarted version of the algorithm can also handle sparse R matrices in an efficient manner, however, there is no specialized algorithm for SVD on R with missing values. In this case, some type of imputation is usually employed.

A special case of such dimensionality reduction is a nonnegative matrix factorization (NMF) where nonnegativity constraint is enforced on one or both of the factors of the decomposition:

$$\begin{aligned} R &\approx G_+ H^\top & G_+ &\geq 0 \\ R_+ &\approx G_+ H_+^\top & R_+, G_+, H_+ &\geq 0. \end{aligned} \tag{4.2}$$

In the first case in Eq. (4.2), we have an arbitrary data set R that can have both positive and negative values but we decompose it in a way that the new, reduced version of the data set G_+ contains only nonnegative values. As will be shown in the next section, such constraint $G_+ \geq 0$ adds additional interpretability to the decomposition that is not present in Eq. (4.1) where only a reduction of the number of features is achieved. The second case in Eq. (4.2) applies when the data set R only has nonnegative values by definition (quantities in R have no meaning if negative). Examples of nonnegative data sets include count data (text, recommender systems, genetics), image data (medical imaging and spectroscopic data in astronomy), similarity data (graph analysis), and so on. However, nonnegativity constraint on any of the factors also makes the use of efficient linear algebra algorithms for decomposition useless, and such decomposition is currently only performed using some type of optimization algorithm.

The second special case is nonnegative matrix tri-factorization (NMTF):

$$\begin{aligned} R_+ &\approx G_+ S_+ H_+^\top & R_+, G_+, S_+, H_+ &\geq 0 \\ R_+ &\approx G_+ S_+ G_+^\top & R_+, G_+, S_+ &\geq 0. \end{aligned} \tag{4.3}$$

In the first case of Eq. (4.3), the middle matrix S_+ allows us to have G_+ and H_+ with a different number of columns. Also, it was shown that NMTF has a more favorable optimization landscape compared to NMF [117] even though the two decompositions are mathematically equivalent if S_+ is square. The second case in Eq. (4.3) applies when R_+ is a symmetric matrix as is the case in graph and similarity data due to $R_+ = R_+^\top \implies G_+ = H_+$.

It seems that by imposing a nonnegativity constraint on the factors, we only lost the benefit of using efficient linear algebra algorithms for factorization, however, as we will see in subsequent sections, this constraint brings important interpretation to the components of the factors that are related to the clustering task. Nonetheless, the necessity of using general optimization for such factorizations brings some flexibility that we cannot afford when using linear algebra algorithms. As we will see in subsequent sections, we can afford to have (a lot of) missing values in R and even perform multiple factorizations simultaneously and, in this way, fuse information from several data sets into a single representation.

4.1.1 Soft Clustering from Nonnegativity Constraint

For many data sets, there are groups of data points that are more similar to each other than to other points. We call such groups clusters. In some cases these clusters are well-separated in feature space. For example, in optical character recognition, images of the same character will be very similar to one another, and we would observe the same

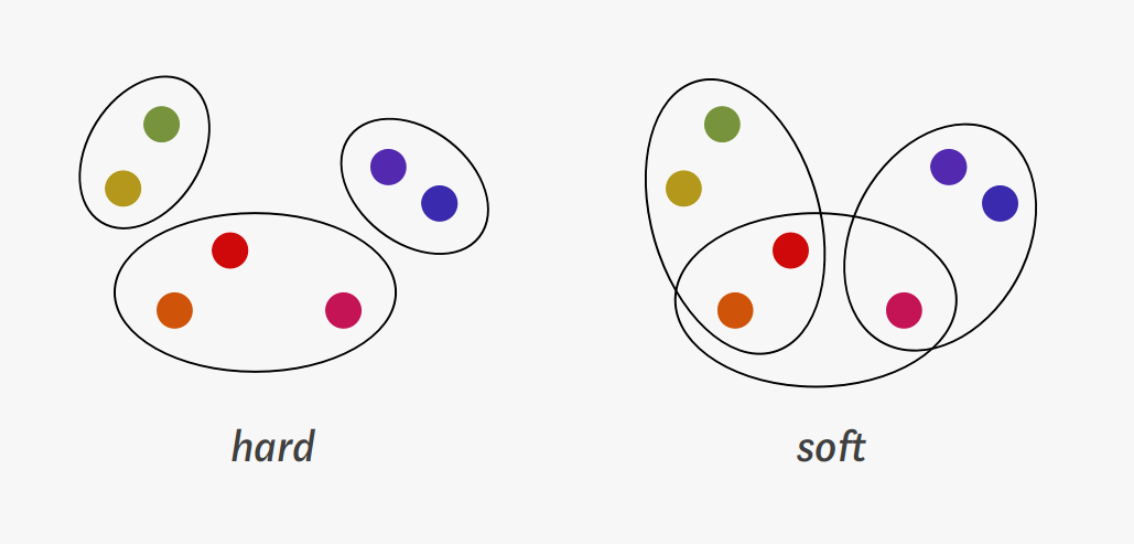


Figure 4.1: A simple example of hard and soft clustering on a sample data set. Reprinted from [118].

number of clusters as there are different characters in our data set. But (possibly even more commonly) clusters may overlap, so that a single data instance can belong to several clusters at the same time. Examples include genetic data (one gene can belong to several groups of genes with related expression patterns), text data (a document may belong to several topic clusters), medical data (a person can have several diagnoses, that is, groups of symptoms), recommender systems (a user may like several groups of items), and so on. With respect to this we distinguish between two types of clustering: soft clustering, which allows cluster overlaps, and hard clustering, which does not. Figure 4.1 depicts a simple example of soft and hard clustering on the same data set.

We will now show that nonnegativity constraints in Eq. (4.2) and (4.3) come from the perspective of soft clustering of data set R. Let us begin with the most general case where matrix R is unconstrained (it may have mixed signs). Suppose we use a k-means algorithm for clustering our data set R, meaning that we cluster data instances R_i (rows of R) and obtain cluster centroids in the feature space $H_j^\top$ (columns of H). Let G_i denote the cluster indicators for data instance R_i, that is, $G_{ij} = 1$ if data instance R_i belongs to cluster j (defined by centroid $H_j^\top$), and $G_{ij} = 0$, otherwise. It was shown in [119] that the k-means clustering (more formally minimum sum-of-squares clustering) objective function is equivalent to

$$\sum_i \sum_j G_{ij} \|R_i - H_j^\top\|^2 = \|R - GH^\top\|^2. \tag{4.4}$$

We see that the k-means clustering objective can be alternatively viewed as an objective function for matrix approximation with a constraint on G that each row of G has one component equal to 1 and all others are 0. Moreover, this k-means clustering will generally be softer if we relax the constraint by allowing G_{ij} to range over values in $[0, 1]$ or values in $[0, \infty)$. Figure 4.2 shows the same point in a more visual manner, and we can see how rows of $H^\top$ hold centroids in the feature space and rows of G how much each instance belongs to the given cluster.

The values of G, if interpreted as cluster affiliations, cannot be negative. The nonnegative values of G_i place the data instance R_i to a feature space by encoding its position

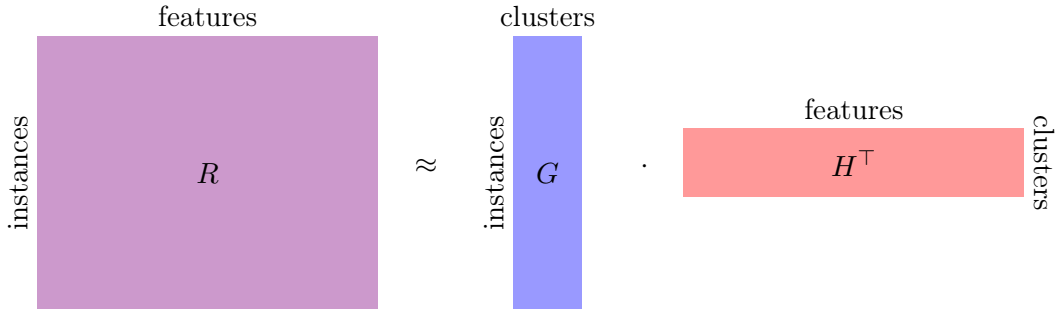


Figure 4.2: Depiction of matrix factorization with the meanings of rows and columns for each matrix.

with linear combination of centroids $H_j^\top$ in the following way:

$$R_i \approx \sum_j G_{ij} H_j^\top = G_{i1} H_1^\top + G_{i2} H_2^\top + \cdots + G_{ik} H_k^\top . \tag{4.5}$$

Therefore, nonnegativity ensures that instances are localized to a hyperrectangle with centroids defining its vertices. This holds when $0 \leq G_{ij} \leq 1$, however, if only nonnegativity is enforced on G it is always possible to rescale G and H so the interpretation presented holds also for $G \geq 0$.

In case the data set R_+ is nonnegative by definition, the centroids of the clustering must also be nonnegative which gives us $R_+ \approx G_+ H_+^\top$ factorization with the same interpretation as stated before, matrix G_+ as an indicator matrix and H_+ as the matrix holding centroids for soft clustering [117]. However, if tri-factorization $R_+ \approx G_+ S_+ H_+^\top$ is used, the scale factors for centroids can be absorbed by the S_+ and both G_+ and H_+ can be interpreted as indicator matrices, and such factorization produces simultaneous soft clustering of both the rows and columns of R_+ at the same time [120]. By this, we see that tri-factorization enables co-clustering and allows, by choosing S_+ to be nonsquare, clustering rows and columns of R_+ so that row space and column space have different number of centroids, which is beneficial because, in general, there is no guarantee that the number of clusters in the data set to be equal for both data types. Such type of clustering is, in turn, equivalent to bipartite graph clustering [121]. For example, consider a matrix of gene expressions for a number of subjects. Using NMTF, both subjects and genes are soft-clustered, and G_+ holds the clusters of subjects and H_+ the clusters of genes, while S_+ measures how each cluster of subjects is connected to each gene cluster. In this example, it is clear that the number of gene and subject clusters will not be the same.

A relevant special case is when $R_+ = R_+^\top$, which implies $G_+ = H_+$. Examples of such data sets fall under the category of similarity data with use cases from graph analysis, text mining, gene co-expression, protein interaction, chemical reactions, and many more. Soft clustering can be performed in the same manner as mentioned in the previous paragraph, but now only one indicator matrix is present:

$$R_+ \approx G_+ S_+ G_+^\top . \tag{4.6}$$

In this case, it is necessary to use tri-factorization because we need a matrix S_+ to absorb the scale factors, and the components of S_+ represent how clusters are interconnected among themselves. For example, in text mining, if R_+ is a co-occurrence matrix (value

R_{ij} is the number of documents which contain both word i and word j), NMTF can be performed on R_+ and G_+ would define word clusters, and S_+ would measure how frequently any pair of clusters of words occur in the same document.

Soft clustering can, therefore, be achieved by performing matrix factorization where nonnegativity is enforced. Nonnegativity constraint upgrades dimensionality reduction via factorization into soft clustering.

4.1.2 Hard Clustering from Orthogonality Constraint

We have seen in the previous section that the nonnegativity of a factor in factorization implies soft clustering. We will now show that if we add the orthogonality constraint to that factor, it implies hard clustering. In connection with Eq. (4.4), we already mentioned that if all rows of G_+ have one component equal to 1 and all others equal to 0, we get hard k-means clustering. We will now show that a more relaxed requirement, orthogonality of columns of G_+, implies the same meaning. Our treatment will follow [120] and [122].

Let us consider the basic NMF objective function J for a general data set R (with negative and nonnegative values) and, therefore, we impose the nonnegativity constraint on only one factor:

$$J = \|R - G_+ H^\top\|^2. \tag{4.7}$$

We can rewrite this expression using a trace of a matrix by employing well-known trace properties [123] in the following way:

$$J = \mathrm{Tr}(R^\top R) - 2\,\mathrm{Tr}(G_+^\top R H) + \mathrm{Tr}(G_+^\top G_+ H^\top H). \tag{4.8}$$

Using zero gradient condition $\partial J/\partial H = 0$, we can express H applying differentiation rules for traces [123] and obtain:

$$\begin{array}{rcl} H & = & R^\top G_+ (G_+^\top G_+)^{-1} \\ H^\top & = & (G_+^\top G_+)^{-1} G_+^\top R. \end{array} \tag{4.9}$$

We can now plug these expressions into Eq. (4.8) while ignoring the constant term that depends only on R. By using the cyclic property of trace [123], we can reorder the result to

$$J = -\mathrm{Tr}((G_+^\top G_+)^{-1} G_+^\top R R^\top G_+). \tag{4.10}$$

On the other hand, the k-means clustering optimization problem was proven to be equivalent to [124]:

$$\max\ \mathrm{Tr}(G_+^\top R R^\top G_+). \tag{4.11}$$

The two objectives would be equivalent if $G_+^\top G_+ = I$. Since H can be freely rescaled, this condition can be further relaxed to

$$R \approx G_+ H^\top \quad \text{with} \quad G_+ \geq 0 \quad \text{and} \quad G^\top G \quad \text{diagonal}. \tag{4.12}$$

Therefore, if G is nonnegative with orthogonal columns, NMF performs hard clustering.

In the same manner, this can be proven for other variants of nonnegative matrix factorization [120], when orthogonality is imposed in addition to nonnegativity on any of the factors, the soft clustering is constrained to hard clustering. Below we list all the variants of NMF and NMTF problems that are equivalent to hard k-means clustering of unconstrained data set, nonnegative data set, on a bipartite graph (simultaneous co-clustering on

nonnegative data set) and on a general graph (clustering on similarity data), respectively:

$$\begin{aligned}
R &\approx G_+ H^\top && G_+ \geq 0 && \text{and} \quad G^\top G \quad \text{diagonal} \\
R_+ &\approx G_+ H_+^\top && R_+, G_+, H_+ \geq 0 && \text{and} \quad G^\top G, H^\top H \quad \text{diagonal} \\
R_+ &\approx G_+ S_+ H_+^\top && R_+, G_+, S_+, H_+ \geq 0 && \text{and} \quad G^\top G, H^\top H \quad \text{diagonal} \\
R_+ &\approx G_+ S_+ G_+^\top && R_+, G_+, S_+ \geq 0 && \text{and} \quad G^\top G \quad \text{diagonal.}
\end{aligned} \tag{4.13}$$

4.1.3 Connection to Data Imputation and Data Fusion

In many practical applications of clustering based on matrix factorization, the available data departs from the idealized setting of a single, fully observed matrix of data values. Two important situations arise frequently:

1. matrix to be factored includes (a great amount of) missing values and
2. there are several data sets (matrices) available related to the same system that is being modeled which could be factored simultaneously in order to perform a more thorough unsupervised learning benefiting from all data sets not just a single one.

The first point can be achieved by slightly altering the objective function of NM(T)F. We will demonstrate this technique on an NMF problem, however, it can be applied to any nonnegative matrix factorization version listed in Eq. (4.13). Suppose we have a matrix R with missing values. We compose a matrix W of the same shape whose values are 1 on positions of matrix R where data values are available and 0 on positions where the values are missing. The objective function is then modified to include contributions to the error only for components of R that are available, while the error is not calculated for values that are missing:

$$\left\| W \odot (R - G_+ H^\top) \right\|^2. \tag{4.14}$$

In Eq. (4.14), we see that the errors where R components are missing get multiplied by 0 and errors where R components are known get multiplied by 1. In this way, only available data in R contribute to error calculation. This means that imputation does not need to be performed on R, which is of great advantage since there is no clear theory that would tell which imputation method would produce good results on a given data set [125].

The capacity of NM(T)F to factor R with missing values to a product $G_+ H^\top$ allows for NM(T)F itself to be used as a data imputation method. By multiplying the factors, we get an approximation of the original matrix $G_+ H^\top = \tilde{R}$ that does not have any missing data. By substituting missing values from R for the ones in $\tilde{R}$, we get a data imputation method for our data set which also comes with feature dimensionality reduction and soft clustering, three important unsupervised learning tasks with one method. Data imputation can also be understood here as a method for prediction, as is used in the scope of supervised learning, for example, as link prediction in graphs [126].

The formalism of NM(T)F also enables data fusion. We can easily compose several factorizations in the same objective function and perform several related factorizations at the same time. We will highlight two not mutually exclusive schemes for data fusion by NM(T)F, both of which have been used to perform data fusion in real-world cases of very large data sets with enormous performance gains compared to processing each data set individually.

The first example is from [127], where genetic data was clustered using three different data sets. The task was to find groups of human genes that contribute to four types of cancer by clustering human genes based on three different similarity measures at the same

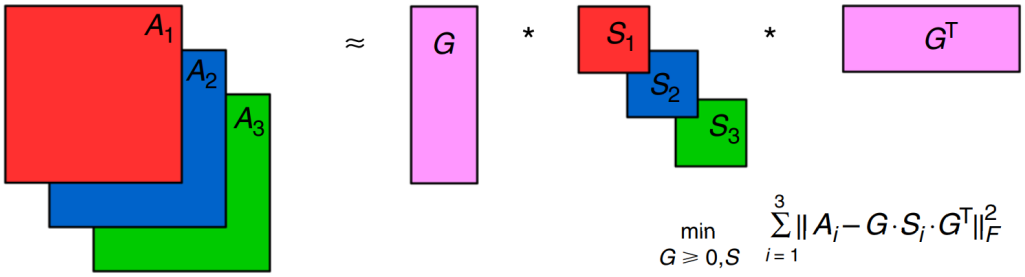


Figure 4.3: Data fusion approach when data sets are available that measure the same quantities in different ways in case of symmetric NMTF. Reprinted from [127].

time. The rationale behind this is that each gene similarity measure has its advantages and drawbacks, and with missing values distributed differently in each of them. Clustering by considering all three of them at the same time would allow to use their joint information. Three gene similarity data sets were used: protein-protein interaction A_1, gene co-expression A_2, and genetic interaction A_3. jk components of matrices A_i measure how much gene j is similar to gene k by its function, expression and interaction. Since A_i are similarity matrices, they are nonnegative and symmetric, so NMTF was applied to perform soft clustering using the following objective function that enables data fusion of those three data sets in the following way:

$$\sum_{i=1}^{3} \|A_i - G_+ S_i G_+^\top\|^2. \tag{4.15}$$

See Figure 4.3 for a more visual depiction of the way the factorization model is structured. This use case presents an example of indicator matrices G_+ being shared among three different factorizations with different S matrices. Therefore, the clustering of genes specified by G_+ is the same for all data sets, however, how these clusters are connected among themselves, which is specified by matrices S_i, are different for the three data sets.

The second example is to perform data fusion by sharing indicator matrices among factorizations that only have one data type in common, not two, as was the case in the former paragraph. The concrete example was taken from [128] and deals with web search data. Four types of objects are considered, which are to be clustered:

1. users (G_1),
2. web pages (G_2),
3. search queries (G_3), and
4. words (G_4).

Five data sets are available that help us cluster these objects:

1. web pages viewed by the users (R_{12}),
2. search queries done by the users (R_{13}),
3. web pages referenced by the queries (R_{23}),
4. words contained in web pages (R_{24}), and
5. words contained in search queries (R_{34}).

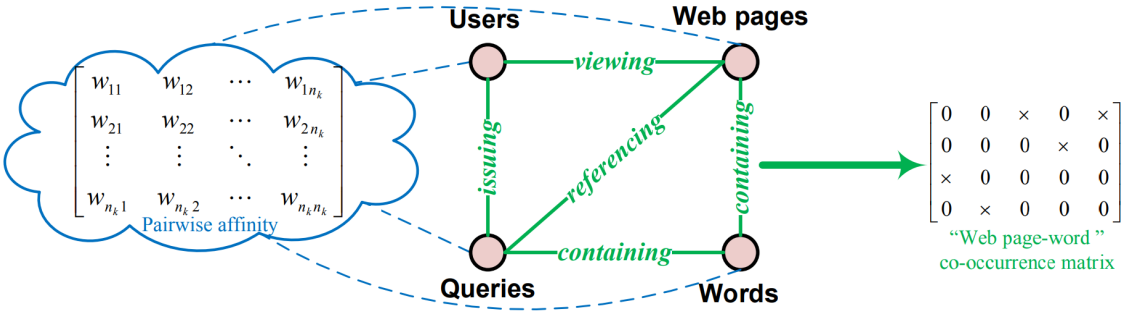


Figure 4.4: Multi-type data fusion approach where several object types are clustered simultaneously using several data sets. Reprinted from [128].

See Figure 4.4 for a more visual depiction of the way this factorization model is structured. Clustering of each of the four objects by using only the available data can be performed using the following objective function:

$$\sum_{i,j} \|R_{ij} - G_i S_{ij} G_j^\top\|^2 = \|R_{12} - G_1 S_{12} G_2^\top\|^2 + \|R_{13} - G_1 S_{13} G_3^\top\|^2 + \\ \|R_{23} - G_2 S_{23} G_3^\top\|^2 + \|R_{24} - G_2 S_{24} G_4^\top\|^2 + \|R_{34} - G_3 S_{34} G_4^\top\|^2. \tag{4.16}$$

We could, of course, cluster each of the five matrices individually using standard NMTF, however, mutual information contained in all five data sets is much more informative for quality clustering. Even if only one object type needs to be clustered, it is more efficient to implement such a data fusion approach despite higher computational costs [128].

We conclude this section with another benefit of before mentioned flexibility that NM(T)F offers. Up until now we wrote the objective function as a square of Frobenius norm of the difference between real R and the predicted one. This is customary for SVD and other standard matrix factorization from linear algebra because it allows linearization and efficient linear algebra algorithms to be deployed for performing factorization. However, since optimization will be used to perform NM(T)F, we are free to use a metric that is more in accordance with the modeling problem at hand. For example, absolute error can be more meaningful than square error for assessing how well $G_+ H^\top$ approximates R_+ or even divergence instead of the norm could be used, such as Kullback–Leibler divergence for positive matrices [129], Bregman divergence [130], and total variation norm that was found to be superior for image data [131].

4.2 Soft Clustering on Graphs and Similarity Learning

In this section, we consider the problem of soft clustering on graphs or, equivalently, the symmetric NMTF problem. We present a methodology for using gradient-based methods to optimize the encountered objective function and, in turn, train the clustering model. A more comprehensive comparison of methods for this type of optimization problems is presented in a journal paper [56], and in this section, we offer a summary of that paper. Additionally, we present our results for solving a multi-objective version of the symmetric NMTF problem where both the accuracy of the model and the number of clusters are being minimized. Full treatment of this effort is reported in [60], [63], [66].

The symmetric NMTF problem is an important variant of NMTF that is specialized

for decomposing symmetric nonnegative matrices:

$$R \approx GSG^\top \qquad \text{with} \qquad R, G, S \geq 0 \qquad \text{and} \qquad R = R^\top \tag{4.17}$$

This problem naturally encodes the clustering of nodes of a graph where components of matrix R are interpreted as similarities between each pair of nodes. Component R_{ij} is seen as a similarity between node i and node j. The decomposition $GSG^\top$, therefore, encodes in what way the nodes of the graph are soft clustered. Component G_{ik} measures to what degree node i belongs to cluster k, and component S_{kl} measures to what degree cluster k is similar to cluster l. However, this problem formulation has been used for a much broader category of clustering problems by defining matrix R as a block matrix. For example, in [132] multi-type clustering (similar to the problem shown in Figure 4.4) of 6 different types of graph nodes was performed using 11 data sets that measure similarities between the nodes. Formulation (4.17) was used for acquiring the clusters by using the 11 data sets as blocks of R form (4.17) while keeping the other components of R zero. We can, therefore, view (4.17) as a general formulation for soft clustering, even for problems with several types of objects to cluster (multi-type clustering). However, formulation (4.17) still does not include the possibility for data fusion in its full generality. To achieve this, we need to allow several matrices R to be factored simultaneously (as in Figure 4.3) using the same factor G. Therefore, in this thesis, we consider the optimization problem that allows both of those features:

$$\min \Big\{ \sum_{i=1}^{N} \|R_i - GS_iG^\top\|^2 : \;\; G, S_i \geq 0, \;\; S_i = S_i^\top \Big\}. \tag{4.18}$$

We use two measures for the quality of our optimization (4.18): the objective function which is the square error (SE) of the factorization and mean square error (MSE), which is a relative value of SE and is used for analysis:

$$\text{SE} = \sum_i \|R_i - GS_iG^\top\|^2 \tag{4.19}$$

$$\text{MSE} = \frac{\sum_i \|R_i - GS_iG^\top\|^2}{\sum_i \|R_i\|^2}. \tag{4.20}$$

4.2.1 Handling Nonnegativity Constraint with Search Space Transformation

One of the major obstacles to applying gradient-based optimization to problem (4.18) is the nonnegativity constraint. The large majority of gradient-based optimization algorithms were designed for searching in the unbounded real space without considering either bounds or constraints. This means that if we want to use generic gradient-based optimization algorithms, we need to handle the constraints in (4.18) in some other way. In our case, we need to ensure that the path of the descent during optimization is localized only to the feasible region. One common way to achieve this is to iteratively project a candidate to the boundary of the feasible region [133]–[135], however, by this, we disturb the descent and impose steps that are not necessarily beneficial. To circumvent this, we developed a different method that is based on the so-called space transformation where we replace the nonnegativity constraints for G and S_i by transforming objective function SE. By moving constraints to the objective function, we are free to apply generic gradient-based optimization. We can transform the feasible set for G and S_i from the nonnegative part of

the search space to the entire real domain by a simple variable substitution

$$G = f(\tilde{G}) \tag{4.21}$$

$$S_i = f(\tilde{S}_i), \tag{4.22}$$

where G, S_i are nonnegative matrices, $\tilde{G}, \tilde{S}_i$ are real matrices and f is an appropriate transformation function that is applied element-wise.

The transformed optimization problem, with $\tilde{G}, \tilde{S}_i$ as new variables, is an unconstrained optimization problem, however, this gain is traded for a more complicated objective function

$$\text{SE} = \sum_i \|R_i - f(\tilde{G}) f(\tilde{S}_i) f(\tilde{G})^\top\|^2. \tag{4.23}$$

The advantage of this approach is that common local search algorithms developed for unconstrained optimization problems can be applied without modifications. This technique has a long history in non-linear optimization and is usually employed in order to transform bounded search space variables to unbounded ones. Examples of transformations used in the literature include smooth transformation of open hyper-rectangle to $\mathbb{R}^m$ [136], continuous transformation of closed hyper-rectangle to $\mathbb{R}^m$ [137] and even continuous transformation of closed hyper-rectangle to search space with toroidal topology [138].

In this work, we consider two transformation functions:

1. element-wise absolute value (search space boundary extension by mirroring [137] adapted for nonnegativity)
$$f_1(X) = |X| \tag{4.24}$$
and

2. element-wise square
$$f_2(X) = X \odot X. \tag{4.25}$$

We can apply f_1 and f_2 to any matrix, regardless of its dimension. The benefit of using absolute value is that this transformation does not change the characteristics of the optimization landscape in any way. However, by using absolute value, the objective function ceases to be a polynomial. Element-wise square, on the other hand, changes the optimization landscape, however, the objective function remains a polynomial, but with doubled degree.

Function f may also be only piecewise differentiable. This is the case for $f_1(X) = |X|$ from above, which is non-differentiable at any component of X equal to 0. In such a situation, we use sub-derivatives instead of derivatives in the non-differentiable points. In the case of f_1, the set of all possible sub-derivatives is $[-1, 1]$. Therefore, we are free to make a choice $f_1'(0) = 0$, which is also in accordance with the statement of Fermat's theorem on stationary points, which says that every local extreme is a stationary point (the function derivative is zero at that point). This is a very common approach in gradient-based optimization, especially in deep learning. However, the proof of mathematical validity of using subgradients in stochastic optimization is, despite its wide use, remarkably recent [88].

4.2.2 Adaptive Moment Estimation Algorithm for Solving Symmetric NMTF Problem

In this section, we show how to apply an established gradient descent algorithm to the symmetric NMTF problem. The adaptive moment estimation (Adam) [73] is a gradient-based optimization algorithm and is currently the state-of-the-art algorithm for training

deep neural networks. It was designed for problems with high-dimensional search space, noisy objective function, and sparse gradients. Adam cannot be straightforwardly applied for solving (4.18) due to nonnegativity constraint. However, this limitation can be circumvented by using a transformed formulation of (4.18) presented in (4.23), which releases the nonnegativity constraint.

In order to apply Adam to (4.18), we chose to transform search space by using element-wise absolute value, that is, $f = f_1$. We decided to use absolute value because this operation is the most computationally efficient way to guarantee nonnegativity since it can be implemented with a conditional clause and no other operations. Also, this transformation leaves the optimization landscape unchanged. The use of absolute value implies the use of sub-derivatives, as explained in the previous section.

In the Adam algorithm, steps are not made in the direction of the gradient but in the direction of the moving average of the gradient. Another trait is step size scaling which is adapted for each matrix element independently. The update rule for both G and S_i are the same, therefore we only state the one for G, which is updated using the following formulas:

$$M_{\tilde{G}} \leftarrow \beta_1 M_{\tilde{G}} + (1 - \beta_1) \nabla_{\tilde{G}} \mathrm{SE} \tag{4.26}$$

$$V_{\tilde{G}} \leftarrow \beta_2 V_{\tilde{G}} + (1 - \beta_2) \nabla_{\tilde{G}} \mathrm{SE} \odot \nabla_{\tilde{G}} \mathrm{SE} \tag{4.27}$$

$$\tilde{G} \leftarrow \tilde{G} - \alpha\eta M_{\tilde{G}} \oslash (\sqrt{V_{\tilde{G}}} + \varepsilon), \tag{4.28}$$

where $\oslash$ denotes element-wise division of two matrices and $\sqrt{\ }$ and adding ε are applied element-wise. The Greek symbols are scalars, and $M_{\tilde{G}}$ and $V_{\tilde{G}}$ are the matrices of the same size as $\tilde{G}$ and contain the information about the mean and the variance of gradient $\nabla_{\tilde{G}} \mathrm{SE}$, respectively. Parameter η changes with a predetermined schedule as statedin [73].

The Adam algorithm is controlled by four parameters (α, β_1, β_2 and ε) which influence its efficiency. The most important one is the so-called learning rate α. The usual magnitude of the learning rate is $\alpha \sim 10^{-3}$, however, due to descent direction scaling by using second moment V, algorithm efficiency is intended to be robust with respect to this choice. Another important parameter is β_1, which should be chosen in accordance with the noise level and gradient sparsity level. However, there are no theoretically founded ways to assess appropriate parameter values based on the optimization problem at hand. Therefore some degree of parameter tuning is normally used. In our experiments we used $\alpha = 0.002$, $\beta_1 = 0.95$, $\beta_2 = 0.995$ and $\varepsilon = 10^{-8}$. More comprehensive analysis showed that the performance of Adam is not very sensitive to changes in control parameters for problem (4.23) and a large proportion of the parameter space results in comparable performance [56].

The gradient calculation for objective function (4.23) with respect to $\tilde{G}$ and $\tilde{S}_i$ was performed by automatic differentiation. For this, the TensorFlow library [26] was used, which is one of the state-of-the-art frameworks for deep learning applications. One of the major advantages of TensorFlow is its highly efficient compiler which results in optimized and stable computational graph for gradient calculation with built-in parallelisation. In this regard, it is able to construct a graph for gradient calculation with respect to both $\tilde{G}$ and $\tilde{S}_i$ simultaneously. This is in contrast to more usual gradient-based algorithms for NMTF that split the search space into blocks and iteratively optimize each block separately [56]. To use the Adam algorithm inside the TensorFlow library in order to solve (4.23) we only need to implement the code for the calculation of the objective function (4.23). The computational graph for gradient calculation is generated automatically by TensorFlow from the code we supplied, and we can directly use the built-in implementation of the Adam algorithm. We chose this method because it is computationally efficient and requires less programming effort and is therefore less error prone.

Empirically, Adam has proven to be very efficient, especially in deep learning, and with a superior convergence rate compared to other gradient-based optimization algorithms [73]. However, contrary to these findings, there are no guarantees that the algorithm converges. In [76], an explicit example of a simple convex optimization problem was provided where Adam does not converge to the optimal solution. In practice, however, such convergence problems are seldomly encountered and can be mended by changing the initial point and/or algorithm parameter values that influence the algorithm's behavior.

4.2.3 Exact Line Search Algorithm for Solving Symmetric NMTF Problem

Using automatic or symbolic differentiation, the problem (4.23) can be solved by any generic gradient-based algorithm. However, our problem (4.23) has a special form that allows us to devise a more specialized algorithm. If we take f form (4.23) to be an element-wise square $f(X) = X \odot X$, our objective function SE is a polynomial of degree 12. This property allows us to devise a gradient descent algorithm that makes optimal-size steps in the direction of the gradient. This means that the learning rate of gradient descent does not need to be preset but is calculated for each step separately so that the step minimizes SE the most. Such an algorithm would, therefore, have no tunable parameters.

To devise our algorithm, we follow the design of a basic line search algorithm where we establish the descent direction to point in the direction opposite to the gradient of the objective function SE that we are minimizing:

$$\tilde{G} \leftarrow \tilde{G} - t\,\nabla_{\tilde{G}}\mathrm{SE} \tag{4.29}$$

$$\tilde{S}_i \leftarrow \tilde{S}_i - t\,\nabla_{\tilde{S}_i}\mathrm{SE} \tag{4.30}$$

$$t = \arg\min_t \mathrm{SE}(\tilde{G} - t\,\nabla_{\tilde{G}}\mathrm{SE},\ \tilde{S}_i - t\,\nabla_{\tilde{S}_i}\mathrm{SE}). \tag{4.31}$$

In general, the descent direction in line search need not be parallel to the gradient. This is the case in the nonlinear conjugate gradient method [139], Gauss-Newton algorithm [140] or Levenberg–Marquardt algorithm [141]. The step size t is calculated independently for each specific step made so that the step taken minimizes SE. Optimization problem (4.31) can be solved either exactly or inexactly. In this regard, we distinguish between exact and inexact line search methods. In our case, minimization (4.31) is a uni-variate minimization of a polynomial of order 12, which can be solved exactly (up to numerical errors). We call the algorithm employing this technique gradient method with exact line search (GM-ELS).

By choosing $f = f_2$, the objective function becomes

$$\mathrm{SE} = \sum_i \left\| R_i - \tilde{G}^2 \tilde{S}_i^2 \tilde{G}^{2\top} \right\|^2, \tag{4.32}$$

where squares act element-wise. To find out how SE changes when moving in the direction of the gradient we substitute $\tilde{G}$ and $\tilde{S}_i$ in (4.32) with right-hand sides of (4.29) and (4.30). We observe that SE changes with t as a polynomial

$$p(t) = \sum_{i=0}^{12} c_i t^i. \tag{4.33}$$

By finding t that minimizes $p(t)$, we find how large of a step to make in the direction of the gradient so that SE is minimal. This polynomial tells how SE changes with respect to step size t when making a move in the direction of the gradient.

Due to exact calculation of optimum t, SE monotonically decreases during iterations, hence it converges to some nonnegative value. This means that at some iteration, we have

$$\min p(t) \approx p(0). \tag{4.34}$$

The equality can (approximately) hold only if the polynomial $p(t)$ has a minimum at point $t = 0$, which indicates that the gradient of SE is zero and proves convergence of GM-ELS toward a point with zero gradient.

A full mathematical description of how polynomial $p(t)$ is calculated is presented in the appendix of [56]. It is important to note that the calculation of coefficients c_j requires a far higher number of matrix multiplications compared to the number needed for gradient calculation. In our implementation, these computations are left to the TensorFlow library, which manages to do them efficiently due to its compiler optimizations, such as the elimination of common subexpressions and simplification of arithmetic statements. However, the time needed for this step was still non-neglectable (approximately 7 times longer compared to gradient calculation). We also tried to reconstruct coefficients of $p(t)$ by polynomial curve fitting methods using evaluations on ≥ 13 points, but the step lengths obtained this way yielded worse performance. For the actual minimization of polynomial $p(t)$, we used analytical methods. Minimization was performed by finding all roots of the derivative of $p(t)$. We used NumPy's function `roots` that relies on computing the eigenvalues of the companion matrix.

4.2.4 Experiments on Synthetic and Real-World Similarity Data Sets

In this section, we use the developed methodology and apply our algorithms to perform soft clustering on several collections of data sets from various domains. In order to assess the performance of Adam and GM-ELS algorithms, we compare them to two other algorithms for solving (4.18) that were developed by the co-authors of [56]. The two algorithms used for comparison are the fixed point method (FPM) and the block coordinate descent with projected gradient (BCD). It is important to highlight some differences in design methodology between the two groups of algorithms. While Adam and GM-ELS use search space transformation to handle constraints, FPM and BCD do this by addressing the Karush-Kuhn-Tucker conditions [56]. Both FPM and BCD optimize G and S_i alternately which means that they are variants of the block-coordinate descent method since, in each iteration, an update is made on one block of variables while keeping the other fixed. Adam and GM-ELS, on the other hand, update the entire candidate in a single step without blocking. The full description of FPM and BCD algorithms as applied to symmetric NMTF problem can be found in [56].

4.2.4.1 Data and Implementation

We benchmark our methods on three data set collections: on a set of synthetic random matrices generated by [56], real-world similarity matrices gathered by [142] and biological network data compiled during the study [127]. A full description of the data set can be found in [56]. All three data collections are also included in the publicly available GitLab repository [49] that contains the code for our experiments.

The set of synthetic random matrices consists of 30 instances of 5-tuples of nonnegative symmetric matrices of dimension $n \in \{100, 200, 500, 1000, 2000, 5000\}$. For each n, we have five matrices with rank 50, meaning that it is possible to factor them using G, S_i with 50 columns. The second data set consists of five symmetric nonnegative matrices (named Aural, Protein, Voting, FaceRec and Zongker) which are commonly used to benchmark similarity learning algorithms. They are real-world similarity or proximity

matrices from different fields of data science and have dimensions $n = 100, 213, 435, 945$ and 2000. The third data set is a biological molecular network data for five different species (fission yeast, baker's yeast, fruit fly, nematode worm, and human). For each species, we have a 4-tuple of nonnegative symmetric matrices that encode different ways to measure genetic similarity (protein-protein interaction, gene co-expression, genetic interaction, and sequence similarity). The matrices in this collection are quite large and have dimensions $n = 4979, 5890, 13190, 17519$ and 20967, respectively. These networks represent real-world use cases where several matrices need to be factored simultaneously in order to integrate multiple data sources and produce more consistent, accurate, and useful information.

Before we run the algorithms, we need to decide on the number of clusters we assume are present in our data. In symmetric NMTF clustering formulation, the number of clusters is equivalent to the number of columns k that matrices G, S_i have. Currently, there is no streamlined way to approximate k, and the usual approach is to simply run clustering algorithms for several different k. In this way, we can observe how the quality of clustering changes with k and decide on the value of k that is a good trade-off between the clustering quality and our ability to analyze the clustering with this many clusters. For synthetic data collection, we know that the ideal number of clusters is 50 [56], therefore, we run the algorithms for $k \in \{10, 20, 30, 40, 50, 60\}$. However, in this case, we know that for $k = 50$ and $k = 60$, by construction, the optimum $\text{MSE}_{opt} = 0$, and our results demonstrate the capability of the methods we used to reach a global optimum. For real-world data sets, on the other hand, we evaluated the methods for five different numbers of clusters $k = [\sqrt{n}/5], 2[\sqrt{n}/5], \ldots, 5[\sqrt{n}/5]$, where $[\cdot]$ denotes rounding to the nearest integer. This choice is in accordance with a commonly used rule of thumb for the expected number of clusters in data sets [143]. However, for some real-world similarity matrices gathered by [142], we found that these values were too large and adjusted them accordingly (for Voting, FaceRec, and Zongker $k \in \{1, 2, 3, 4, 5\}$).

All four algorithms that are compared were run on the same hardware and using a unified methodology for initial point generation and termination criterion. Full implementation details are reported in [56]. For initial point generation, we tested both random initialization and deterministic initialization by using eigenvalue decomposition [144]. Our analysis shows that random initialization on average results in slightly worse objective values compared to deterministic initialization, which is why we use the latter. This means that all algorithms reported here are completely deterministic. As advice to the reader, we suggest using random initialization only when matrices are small and when small differences in objective value matter greatly to the user. In this case, many runs can be performed, and there is a reasonable probability that some runs using random initialization will end up lower in objective value than the one from a deterministic point. It must be noted that this procedure is orders of magnitude more time-consuming and, therefore, not suitable for large matrices. The stopping criteria for all methods were the maximum number of iterations (4000 for FPM, 300 for BCD, 1000 for GM-ELS, and 3000 for Adam) or if the MSE falls below 10^{-2} or if the absolute difference between the MSE in given iteration and the MSE in the previous iteration is below 10^{-10}. All four algorithms were run on the same type of node that is part of high-performance cluster at the University of Ljubljana, Faculty of Mechanical Engineering (E5-2680 V3 (1008 hyper-cores) DP cluster, with IB QDR interconnection, 164 TB of LUSTRE storage, 4.6 TB RAM).

4.2.4.2 Optimization Results

Results of the four algorithms, when applied to all 16 data sets, are shown in Figure 4.5. The figure shows how MSE changes with number of clusters k. We can clearly see that the MSE decreases with an increasing number of clusters k, the only exception is the Zonker

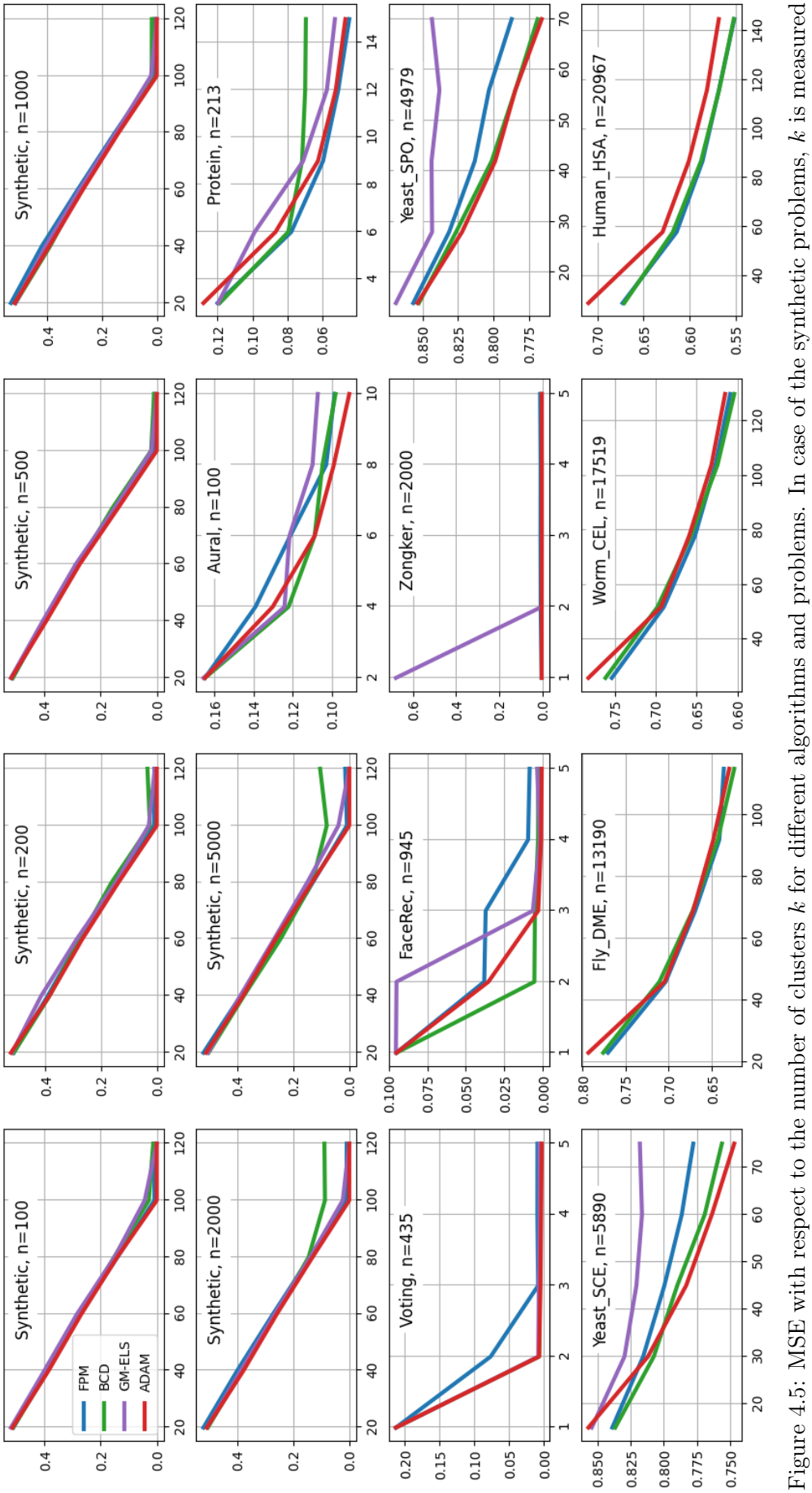


Figure 4.5: MSE with respect to the number of clusters k for different algorithms and problems. In case of the synthetic problems, k is measured in % of the actual number of clusters. Reprinted from [56].

data set, where MSE values are very small at the very beginning (below 10^{-2}) and due to the stopping criteria MSE does not decrease any further, actually it even slightly increases for the case of FPM, but remains below 10^{-2}.

On the synthetic data collection, FPM, GM-ELS, and Adam are very competitive regarding the MSE, while BCD yields worse MSE for larger values of n. A deeper analysis shows that the worse performance of BCD is due to a small number of iterations (it was limited to 300). When we allow 1000 iterations, the resulting MSE was of the same order of magnitude, that is, below 10^{-2} as for the other three methods.

When the algorithms are run on real-world similarity matrices, we can see that the mean values of MSE for `Voting, FaceRec`, and `Zonker` are falling rapidly towards 0, which demonstrate that all four methods are capable of finding the optimum decompositions when k is increased. However, for `Aural` and `Protein`, the MSE does not come very close to 0. This can mean that those two matrices cannot be factored in the proposed way due to the noise, or the inner dimension was not correctly identified, or some other hidden reason might be present. For this data set, we observe no clear pattern that would declare which algorithms are systematically better than the others. For large k, all algorithms obtain similar MSE values (with the exception of `Protein` where BCD is clearly not as good as the others). In the case of smaller k, the ranking of the algorithms changes with respect to problem instance and inner dimension k.

Applying the four algorithms on biological network data instances was very time-consuming and memory-consuming. GM-ELS even ran out of memory for the largest three tuples of matrices, while the other three methods always terminated regularly, according to our stopping criteria. Regarding the MSE we observe that FPM, BCD, and Adam perform similarly. For the largest instance (Human_HSA), Adam returns slightly worse MSE, but the difference is quite small. A similar situation can be noticed in the case of Yeast_SPO and Yeast_SCE, where MSEs for the FPM algorithm are systematically slightly higher. GM-ELS, on the other hand, is consistently worse than others or, for larger data sets, even runs out of memory. This indicates that GM-ELS is not an appropriate choice when factoring large data sets. This is a result of considerable resources needed for the calculation of the 12-th order polynomial used for determining the ideal step size for GM-ELS.

To provide a better insight into which algorithms perform best on a given instance (regarding the MSE) we show the winners in Figure 4.6. More precisely, for each synthetic instance, each real-world similarity instance, and each biological network instance, we plot at position $(\log(n), \log(k))$ the symbol denoting which one out of the four algorithms gives the lowest MSE for a given instance within this numerical evaluation. The more signs appear in this plot, the more often the corresponding algorithm returns the lowest MSE. We can observe that Adam the most often (in approximately 50% of the cases) gives the best MSE, followed by BCD and FPM, each winning in approximately 24% of the instances. The GM-ELS results with best MSE only in two cases. Additionally, Adam outperforms the other algorithms for larger values of the inner dimension k, while for the smaller values of k the winners are usually FPM and BCD.

It is also important to consider the time efficiency of the compared algorithms. Depiction of the characteristic convergence curves of the algorithms is shown in [56] and here we offer a short summary of our findings. Results indicate that the algorithms that end with the lowest MSE in Figure 4.5 also spend much more time to reach these results compared to the others. If we limit the time to a quite modest range, then the FPM outperforms the others since it demonstrates the best convergence at the beginning and computes the lowest MSE in a given (short) time frame. Otherwise, the overall ranking of algorithms FPM, BCD, and Adam are quite uniformly distributed when considering their entire runtime.

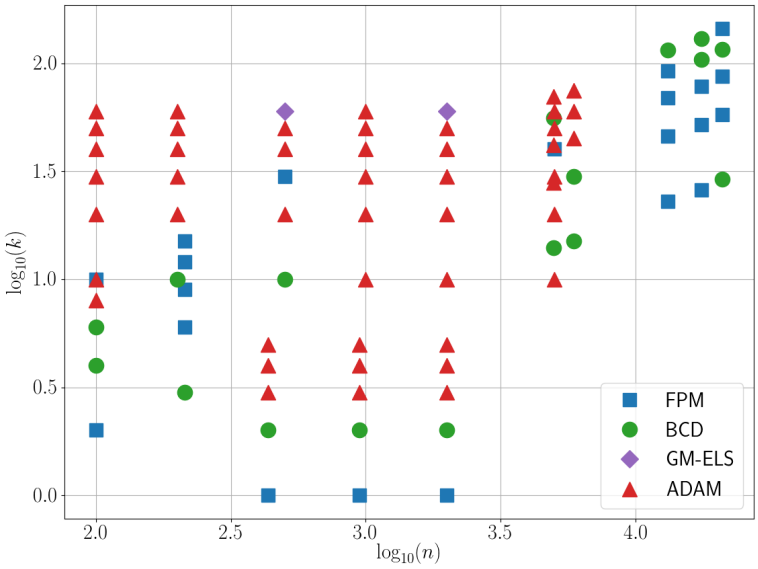


Figure 4.6: The plot shows which algorithm achieved the lowest MSE of all four algorithms considered with respect to the matrix dimension n and the chosen number of clusters k. Each marker position represents one symmetric NMTF problem, and the type of marker encodes which algorithm has found the best solution for that problem. The plot includes results for all three data collections considered. Reprinted from [56].

The exception is GM-ELS which systematically lags behind the others and oftentimes does not converge to a similar MSE level as others do in the given computational budget.

Considering the results, we can confirm hypothesis **H2A** for the soft clustering case since we have shown in practice that the Adam algorithm can be used to efficiently soft cluster data sets. General-purpose gradient-based algorithms are applicable to clustering problems and, in the case of Adam can even find superior solutions with high probability. We have tested this claim on a diverse set of data sets by reformulating soft clustering problems and using automatic differentiation and gradient descent for soft clustering. For large data sets, our methodology can converge slower compared to the current state-of-the-art approach (FPM), however, this is in large part due to FPM exploiting matrix sparsity. TensorFlow, on the other hand, did not have support for sparse matrices at the time of this study, however, this has changed in the meantime with the introduction of module `tensorflow.sparse`. This means that we can expect even faster convergence of gradient-based methods for clustering in the future.

4.2.5 Multi-Objective Approach to Soft Clustering

In the previous section, we have seen that soft clustering of the same data set was performed several times for different number of clusters k. The reason for this is the absence of a methodology for determining the number of clusters during the clustering process itself. The problem behind it is that when we increase the number of clusters k, the quality of clustering also increases. But, on the other hand, working with an exceptionally large number of clusters defeats the purpose of clustering, which is to better understand the data. This is why, in practice, we want to find clustering with manageable number of clusters that still has a sufficient quality to be meaningful. Looking for this trade-off between the number of clusters k, and the clustering quality naturally leads us to multi-objective optimization. In this setting, we would like to minimize both the number of clusters k and

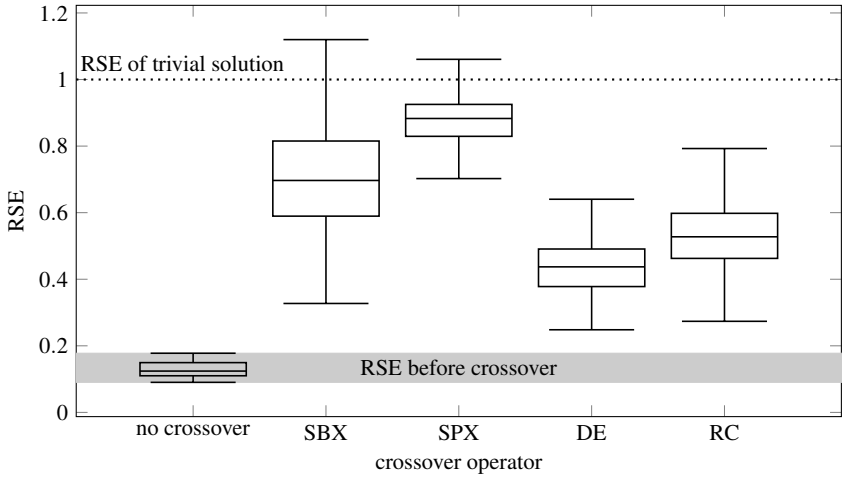


Figure 4.7: Distributions of RSE values of individuals produced by crossover from parents with RSE from the shaded range. The crossovers considered are simulated binary crossover (SBX), simplex crossover (SPX), differential evolution crossover (DE), and choosing random columns (RC) with uniform probability out of both parents. The trivial solution is a candidate with all components of matrices G, S_i equal to zero. Reprinted from [63].

the clustering error RSE:

$$\begin{array}{lrcll} \text{minimize} & \text{RSE} & = & \dfrac{\sum_i \|R_i - GS_iG^\top\|^2}{\sum_i \|R_i\|^2} & \text{with respect to} \quad G, S_i \\ & k & = & \dim S_i & \text{and subject to} \quad G, S_i \geq 0,\ S_i^\top = S_i. \end{array} \tag{4.35}$$

By solving problem (4.35), we would acquire a set of clustering solutions for different number of clusters (similar to the plots in Figure 4.5) and by this, avoid running optimization several times on the predetermined set of values k. For more information on the nature of multi-objective optimization and methods usually employed in this setting, see, for example, books [145], [146].

In this section, we provide a short summary of the work reported in [60], [63], [66]. It represents an effort toward finding a way to efficiently solve optimization problem (4.35) either by evolutionary algorithms or by hybridization of gradient descent with evolutionary algorithms. The necessity to turn toward evolutionary algorithms comes from the fact that the current methodology in the literature heavily relies on population-based methods for solving multi-objective problems [147] even for large-scale problems [148]. However, it turns out that problems of the NMF type have not been well-studied using evolutionary approaches. It is not known what kind of recombination and mutation operators work well in this setting (when candidates encode matrices). To identify possibly useful operators, we first focused on a single-objective case (4.18) and tested several evolutionary operators known from the literature to work well in continuous search spaces. We have found that none of them was able to produce offspring that would have comparable fitness in comparison to their parents (see Figure 4.7).

Based on the results, we see that classical crossover operators do diversify the population, which is also their aim, however, to such a degree that an offspring appears random compared to its parents. In our experiments, we have observed that, when pure evolutionary algorithms are run to solve (4.18), the population of candidates remain on similar fitness level throughout the optimization even when the algorithms are run for exceptionally large

number of iterations[2]. Our interpretation is that in the absence of fitness inheritance, the evolutionary algorithms when applied agnostically to problem (4.18) search in a similar manner than random search does. It is known from the literature that inheritance is a crucial property for this type of algorithms with a guideline that crossover in an idealized setting should create instances with higher fitness [149].

The lack of fitness inheritance when using classical evolutionary operators leads us either towards finding a more meaningful encoding for the individuals or towards devising new evolutionary operators that would be more in line with the nature of our optimization problem (4.35). In [60], we have devised a novel mutation and crossover operators that treat columns of G (and associated components of S_i) as the atomic elements for the operators. Using the clustering interpretation of problem (4.35) means that these operators can only lose or attain an entire cluster definition from its parent and do not have the capacity to merge cluster definitions (and consequentially corrupt them). By leaving each individual cluster inside a candidate intact, we can guarantee the inheritance of fitness from parents to offspring. In the case of lossless crossover, which was devised specifically for the multi-objective case, we can prove that the RSE objective value of the offspring is less than or equal to the average of RSE objective values of the parents [63]. However, since these operators can only delete or introduce new clusters, they are unable to change the magnitudes of how much each data instance belongs to each cluster. To change those magnitudes, we would need some type of cluster merging crossover, and all operators of this type that we tested were found to be detrimental to the convergence (see Figure 4.7).

One way to circumvent this problem is to search for appropriate magnitudes using some other algorithm that is not native to the evolutionary setting and, by this, develop a hybrid approach. Since we already know that gradient descent algorithms work well for a single-objective variant of our optimization problem, we opted to look in the direction of memetic algorithms that attempt to join the strengths of population-based and single-solution-based optimization algorithms. For more information on the common hybridization design patterns for memetic algorithms, see the review [150]. Informally, the memetic algorithm uses a population-based global search technique to broadly locate good areas of the search space while repeated usage of some local search heuristic is employed to improve the quality of offspring as far as possible. The local search heuristic is used to refine individuals and correspond, in the original metaphor of memetic algorithms, to the lifetime learning of the individuals.

Our memetic algorithm is specifically tailored to solve (4.35). For the local search part of the algorithm, we decided to use Adam which was already proven to work well in the previous section. It is important to note that, when solving (4.35), Adam can only improve the first objective RSE while keeping the second objective k constant. This is why the population-based part of the hybrid is required to change the number of columns in matrices G, S_i in order to explore different values of k for our second objective. This is achieved by exchanging columns among the population using column-based evolutionary operators from [60]. Informally, the evolutionary part of the proposed memetic algorithm is focused more on the "discrete" part of the search space (how many clusters and which data instance belongs to which cluster) while Adam is more focused on the "continuous" part of the search space (to what degree a given data instance belongs to each cluster).

We tested several variants of a memetic algorithm to solve (4.35). The main difference between them was how column-based evolutionary operators are implemented. We observed that column-based crossover works best when the number of columns that are discarded is as small as possible. Inheritance of the RSE objective is maximal if all columns

[2]We have tested this on small matrices where we could afford trillions of evaluations. For large matrices, on the other hand, computational resources limit us to $\sim 10^4$ evaluations.

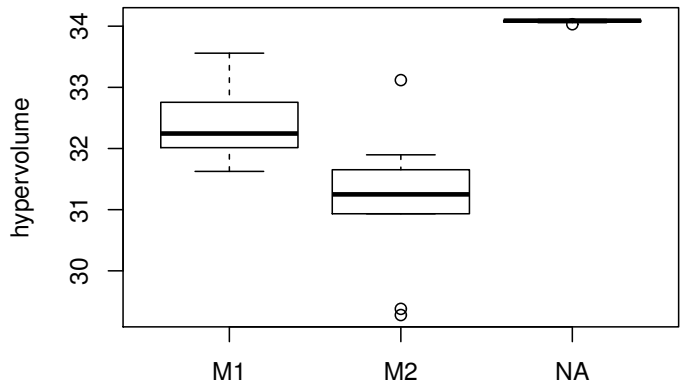


Figure 4.8: Box plot of hypervolumes for the three algorithms considered. Hypervolume is one of the measures used to assess the quality of solutions found by multi-objective optimization algorithms. Higher hypervolume value indicates a higher-quality solution. M1 is a memetic algorithm when only mutations are applied, M2 when both mutation and crossover are applied, and NA is the naive approach where only Adam is applied from random points with different and predetermined values of k. Reprinted from [60].

of the parents end up in the offspring. This means that the crossover can only increase the second objective k (number of columns), however, it does provide an offspring with an inherited low RSE objective that is used as an initial point for Adam. For example, consider two parents that were produced by Adam in the objective space (RSE, k). Crossover between $(0.42, 2)$ and $(0.29, 3)$ would produce an offspring $(0.35, 5)$. Compared to a randomly generated individual, which is, for example, $(1.04, 5)$, this offspring has a much smaller RSE objective and should be a better starting point for Adam. The mutation operators considered either adding or deleting clusters from an individual. This mutation works best if the added columns are populated by small numbers so that the RSE value of the offspring is reasonably close to the parent. For the column deletion, we found that the convergence of the memetic algorithm is more favorable if the deleted columns are the ones that contribute the least to the $\sum_i \|GS_iG^\top\|^2$. These experimental results led us to the final version of evolutionary operators which can change objective k while minimally altering objective RSE. The full design and implementation of our algorithms is reported in [60].

Figure 4.8 shows how our memetic algorithm compares with respect to solution quality if only mutation is applied (M1) or if both mutation and crossover are applied (M2) to the naive approach (NA), where we apply Adam on random points with different and predetermined values of k. We can see that NA finds the best approximation of the Pareto front. This is a surprising result since we know from the previous section that when Adam is started from an initial point with lower RSE, it produces better-quality solutions. This is why the SVD decomposition was used to generate initial points for Adam in the previous section. Using evolutionary operators to generate initial points with low RSE, however, does not bring the same benefits. In order to understand this discrepancy, it is instructive to look at the convergence of Adam from the initial points that were generated by evolutionary operators. Figure 4.9 shows how Adam converges when applied to a crossover offspring compared to the convergence from a random individual with the same k. We see that the initial RSE of the offspring is indeed smaller compared to the random point, however, the convergence curves look quite different. When Adam is applied to individuals created by crossover, RSE steeply falls, but shortly after, the convergence becomes very slow.

Comparing M1 and M2 in Figure 4.8, we see that the use of the column-based mutation

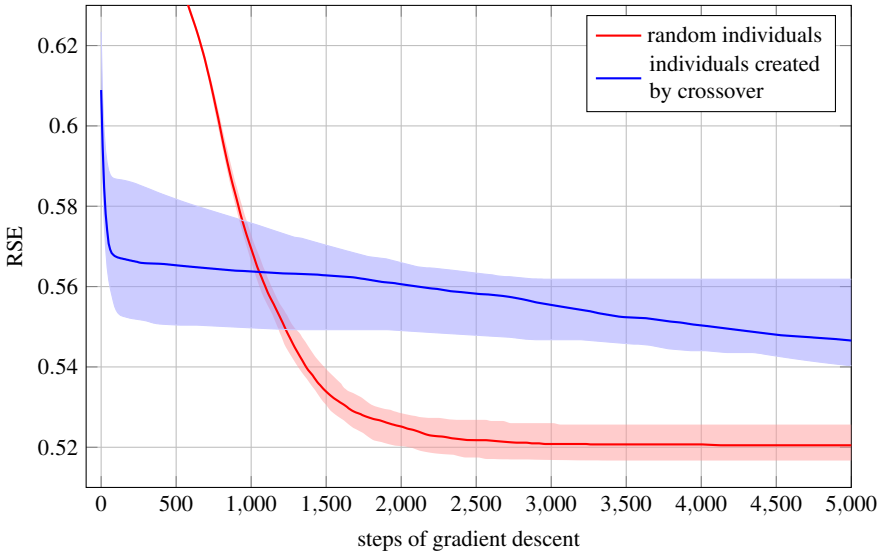


Figure 4.9: Progress of the Adam algorithm when initial points are individuals created by crossover compared to the random individuals. All solutions here have $k = 10$, and crossovers were performed using already descended individuals with $k = 3, 4, 5, 6, 7$ ($3 + 7$, $4 + 6$ and $5 + 5$). Full lines are the medians, and the shaded areas represent the range of the central 66% of runs. Reprinted from [63].

is less problematic compared to the crossover. It turns out that the M1 algorithm also required the least amount of evaluations out of the three algorithms [60]. This means that our mutations supplied individuals in the population on which Adam converged faster than when started from random points but to RSE values that are worse compared to NA. If we look at the convergence of Adam started from different mutated individuals (see Figure 4.10), we can see why this is so. If mutation adds only one column, the convergence of Adam is approximately three times faster compared to Adam started from a random individual. The best RSE found is worse for the mutated individual, however, the difference is quite small. This result can guide us to an improved version of the M1 algorithm, which would iteratively: (1) add a single random column to an optimized individual and (2) apply Adam to it. Such an algorithm could start with $k = 1$ and use Adam and mutations to fill in the Pareto front for each k (by iteratively increasing it by one) up to the desired maximum value. However, we did not decide to implement this because NA has two clear advantages over such an algorithm. The first is that RSE values found by NA are better, and the second is that NA allows the user to increase k in steps larger than one. This way, the user can first explore the Pareto front with bigger steps and use smaller steps in the region of the Pareto front that is of bigger interest. On the other hand, mutations that would be needed for increasing k in bigger steps were found to be more detrimental to the final quality of the Pareto front approximation (see Figure 4.8).

The results presented in this section show that the difficulty of determining the appropriate number of clusters for a given data set persists even if we use NMF formulation of clustering. Machine learning and data science communities have been working on this problem using the classical formulation of clustering for decades with very limited success [151]. In this section, we can observe how these difficulties manifest in the NMF formulation of clustering. We have found that, even for NMF formulation of clustering, the most efficient way to find the correct number of clusters is to run a clustering algorithm (Adam) several times, once for each number of clusters k (NA algorithm), which

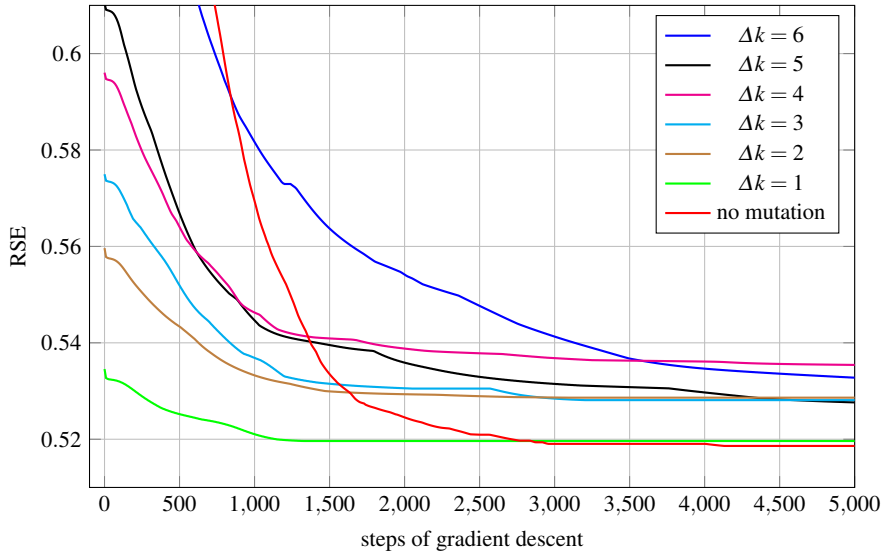


Figure 4.10: Median progress of the Adam algorithm when initial points are random individuals compared to mutated individuals. All solutions here have $k = 10$, and mutations were performed using already descended individuals with $k = 4, 5, \ldots, 9$. Quantity Δk tells how many columns were added via mutation and represents the extent to which an individual was mutated. Reprinted from [63].

is also the guideline in classical clustering. From the perspective of memetic algorithms, when applied to NMF-type problems, we have shown that fitness inheritance is a crucial property of evolutionary operators, but it is not the whole story. If fitness inheritance is low, we get an algorithm analogous to random search, but if fitness inheritance is high, the algorithm can concentrate on degenerate parts of the search space where convergence of the local search is very slow. We deduce that fitness might not be the most important trait to be inherited but rather the ability of Adam to converge to optimum with a small number of steps. However, finding a meaningful mathematical description of such a trait and designing operators that would inherit such a trait requires a more involved treatment of problem (4.35), both theoretically and experimentally.

4.3 Hard Clustering on Graphs

The current methodology presented in Section 4.2 allows us to perform soft clustering on a data set using gradient-based optimization algorithms. In this section, we augment this methodology so that we are able to perform hard clustering as well. The task of hard clustering is much more prevalent in the literature and is regarded as a more classical formulation of clustering, even though soft clustering is gaining an increasing amount of attention in the last decades [152]. In hard clustering, data instances cannot belong to several clusters at the same time but are clustered in a way that a single data instance can only belong to a single cluster. This is an important distinction because this effectively moves us into discrete optimization domain. When performing soft clustering, the natural encoding is continuous (for each data instance and each cluster, we have a nonnegative real number that conveys to what degree an instance belongs to that cluster), but when performing hard clustering, the natural encoding is categorical (for each data instance we have an integer that conveys to which cluster the instance belongs). This fact makes the applicability of gradient-based algorithms quite surprising in this setting. To achieve hard

clustering, our treatment will follow theoretical considerations of Section 4.1.2 where we retained the continuous encoding of soft clustering and acquired hard clustering interpretation of the solution by imposing an additional constraint, orthogonality. In this way, we do not rely on the natural discrete encoding of hard clustering at all. This allows the application of gradient-based optimization and represents one way to circumvent the discrete nature of the optimization problem in question. The results presented in this section were also partially reported in [61], [68].

As with soft clustering, we will restrict our treatment to a multi-type graph clustering problem which is a generalization of a common clustering task and includes many clustering schemes as a special case[3]. The problem formulation is an extension of (4.18) adding an orthogonality constraint to matrix G. This problem is now called symmetric multi-type orthogonal nonnegative matrix tri-factorization (SONMTF) problem. The data to be clustered is, in this setting, an N-tuple of symmetric nonnegative matrices $(R_1, \ldots, R_N)$, which we want to factorize simultaneously by solving

$$\min \Big\{ \sum_{i=1}^{N} \|R_i - G S_i G^\top\|^2 : \;\; G, S_i \geq 0, \;\; S_i = S_i^\top, \;\; G^\top G = I \Big\}, \tag{4.36}$$

where I is the identity matrix. It is important to note that properties "G is orthogonal" ($G^\top G$ is diagonal) and "G is orthonormal" ($G^\top G = I$) are equivalent for problem (4.36), since the norms of columns of G can always be absorbed into matrices S_i. This is why in the common literature, constraint $G^\top G = I$ is usually called orthogonality even though it should be called orthonormality from a mathematical point of view.

4.3.1 Imposing Orthogonality Constraint while Using Gradient-Based Algorithms

Compared to the nonnegativity constraint that was handled using search space transformations (see Section 4.2.1), the orthogonality cannot be imposed in such a manner due to the complicated geometry and topology of the feasible region. The subspace of the search space on which orthonormality is satisfied is known as the Stiefel manifold, which is a compact and, in general, not simply connected manifold [153]. This means that the feasible space is, in this case, curved and twisted in a non-trivial fashion. This is in contrast to the nonnegative subspace (from Section 4.2.1), which is simply connected and non-compact, which makes it quite similar to the unconstrained search space with regards to geometrical and topological properties. This is why nonnegativity could be imposed by search space transformations from the unconstrained search space onto the feasible region of the search space. But when imposing orthogonality, such transformations onto the Stiefel manifold are non-trivial to calculate as a function. This is true even if an algorithm is designed for a specific matrix shape (n, k) on which orthogonality is imposed [153]. There is no known computationally efficient way to perform a coordinate transformation that would work for any matrix shape from the search space onto the Stiefel manifold. This means that orthogonality will need to be handled in a different way, without search space transformations.

A common approach in the literature for dealing with orthogonality constraint when solving orthogonal NMF-type problems is penalization [154]–[158]. By this technique, a penalty is added to the objective function, which increases when the solution deviates away from the Stiefel manifold. Usually, constraint $G^\top G = I$ is imposed by turning the objective function being minimized into

$$\|R - GH\|^2 + \lambda \|G^\top G - I\|^2. \tag{4.37}$$

[3] See the paragraph leading to (4.18) for examples.

We tested various versions of such penalization, including adding $|G^\top G - I|^p$ for different values of p, adding the sum of cosines of angles between all pairs of columns (also using various powers of p) and adding measures that penalize linear dependence, such as Pearson correlation among columns of the matrix. We found that none of these methods work well in conjunction with the gradient descent algorithm without extensive tuning of the penalization constant λ. We saw that the success of the descent was quite sensitive to λ and that the value for "optimal" λ strongly depends on the shapes of matrices as well as on the data distribution itself (statistical properties of components of matrix R). It seems that when both orthogonality and nonnegativity are imposed, the feasible space splits into regions that are weakly connected among themselves. If an algorithm is only allowed to traverse over the feasible region, it is very difficult for it to jump from one region to another. We can back this claim with a consideration on the nature of the problem that orthogonal NMF is solving. In hard clustering interpretation of NMF, we are solving a mixed optimization problem with a discrete part that tells to which cluster a given data instance belongs and a continuous part that tells to what degree it belongs to that cluster. The discrete part of this optimization results in the splitting of the feasible space to regions that are not well connected among themselves. Because gradient descent follows a path in a search space and because feasible space is, in this case, not well path-connected, we need to ensure that the gradient descent is allowed to leave the feasible space and can follow the path that lowers the objective. At least during the first stage of optimization, the penalization due to constraints needs to be weak compared to the objective.

Because it is difficult to assess beforehand the span of values and gradients of the objective for a given matrix, it is also difficult to select an appropriate penalization constant λ. If taken too large, the descent will be restricted to a feasible region closest to its initial value and will not be able to leave it. If, on the other hand, λ is chosen too small, the descent will practically ignore the constraint and can end up in a minimum that is far away from the feasible region. In our tests, we found that it is beneficial to employ a strategy of using a small penalization in the first stage of the optimization and increasing it in the last stages. In this regard, the gradient descent moves towards a region with low objective values in the first stages and then, with a sharp increase in penalization, it jumps onto the feasible region and then traverses in that region to a minimum that is close in value as before the penalization increase, but now lying on a feasible set as desired. Looking at this strategy from the hard clustering point of view, this means that we first find a soft clustering for the data, allowing data instances to be in several clusters at once, and then, for a given data instance, find out to which cluster it belongs the most and then adjust the degree of belonging to that cluster until a minimum is found.

4.3.2 Three Stage Scheme for Solving General Orthogonal NMF Problems

As mentioned in the previous section, it is beneficial to increase the penalization constant λ during the descent. We now explain how building on this idea results in a simpler three-stage scheme for solving orthogonal NMF problems that do not include penalization at all. When experimenting with gradient descent with increasing penalization in time, we found it difficult to assess the penalization degree in the first stage of the descent because the optimization landscape can be quite flat close to the initial point, and even the slightest value for λ can pull the descent to unpromising regions from where gradient descent may converge with a very slow pace. In addition, it is also difficult to decide beforehand when the increase in penalization should happen, since it is not known at what scale the optimum objective is. This led us to completely dismiss the penalization in the first stage of the optimization and run the descent until the convergence criterion is

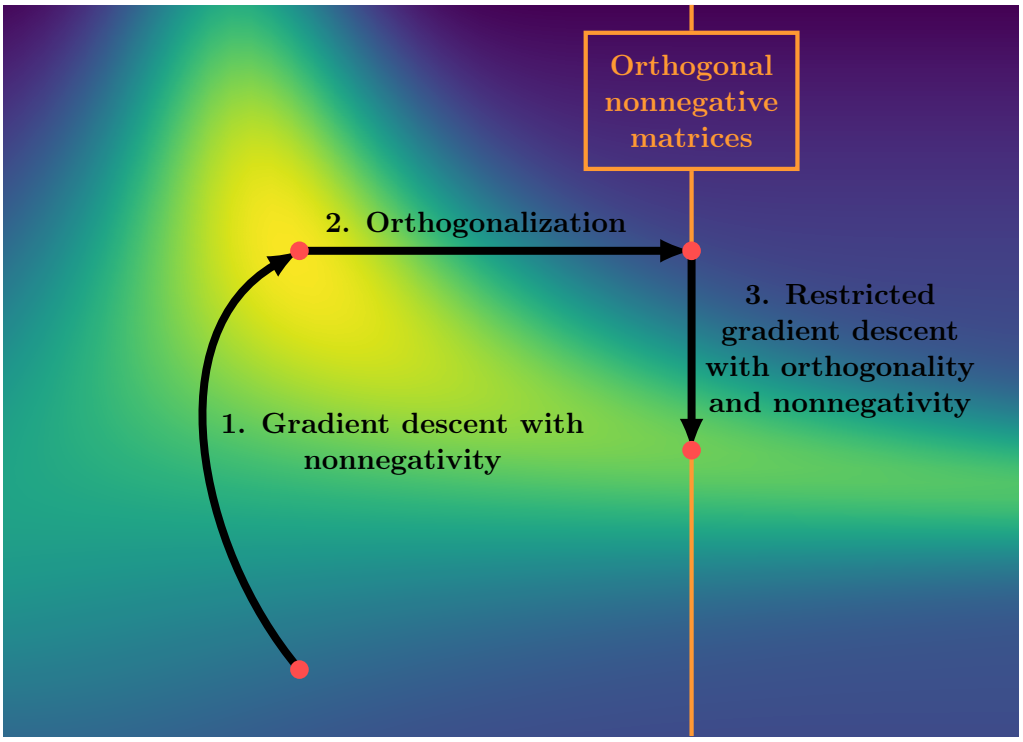


Figure 4.11: Depiction of the three-stage scheme for using a gradient-based algorithm for finding orthogonal nonnegative solutions of arbitrary NMF problem. The color signifies an optimization landscape, with yellow tones indicating more optimal solutions. The orange line represents a subset of the search space where orthogonality is fulfilled (in addition to the nonnegativity). The three stages are depicted with black arrows where the first stage finds a good solution outside of the feasible space, the second projects the solution onto the feasible space by orthogonalization, and the third finds a good solution while restricted to the feasible space of orthogonal nonnegative matrices.

satisfied without any regard to the orthogonality constraint. At this point, a descent with strong penalization should be started, however, we found that even better results can be obtained with a simplified version for the second stage. Instead of using penalization to first find a nearby feasible region and then traverse through it while following the gradient, we can simply project the current point in the search space onto the Stiefel manifold in one step. Such a projection is mathematically viable and can be performed in an efficient manner. By this, we save a number of gradient steps needed from the current point to the feasible region. Now highly penalized version should commence ensuring that gradient descent cannot leave the feasible region, however, it is computationally more efficient to use a restricted version of the gradient descent without any penalization. Such restricted descent is only able to move in a current feasible region and find a minimum that is also feasible. This is more computationally efficient because gradient descent inside the Stiefel manifold can be performed in much smaller space compared to the unconstrained search space (dimensionality of the feasible space is k-times smaller with k being the number of columns of the matrix on which the orthogonality is imposed).

This three-stage scheme is applicable for solving any variation of the NMF problem that includes orthogonality constraints. Also, any version of a gradient-based algorithm (such as Adam or GM-ELS) can be used for the gradient descent part. A crude depiction of the scheme is shown in Figure 4.11, showing all three stages together with an example

optimization landscape and feasible region of orthogonal matrices. The advantage of this three-stage scheme is also simplicity. We simply ignore orthogonality to find a good solution, then force the matrices to be orthogonal, and then use gradient descent again, now in a much smaller (feasible) space to find a good feasible solution.

In the following sections, we provide a full description of the three-stage scheme.

4.3.2.1 First Stage: Gradient Descent with Nonnegativity Constraint

In the first stage, the same method is employed as when performing soft clustering (described in Section 4.2.1). Some type of gradient descent is used to minimize an objective that was modified using the search space transformation method (to ensure that the solutions fulfill the nonnegativity constraint). In summary, variables with nonnegativity constraint are in the objective function substituted for their absolute value or element-wise square or similar transformation function. For example, by using the absolute value for symmetric NMTF problem:

$$\begin{array}{lccc} \text{Objective function:} & \|R - GSG^\top\|^2 & \rightarrow & \|R - |G||S||G|^\top\|^2 \\ \text{Constraints:} & G \geq 0,\, S \geq 0 & \rightarrow & \text{none} \end{array} .$$

The transformed problem can now be treated as unconstrained, and any gradient-based algorithm can be applied to it. The benefit of using the absolute value is that this transformation does not change characteristics of the optimization landscape in any way. Using this transformation, we can now apply any gradient descent algorithm such as Adam, RMSprop or GM-ELS, however, with GM-ELS it is more prudent to use the square as the transformation function as explained in Section 4.2.3.

This stage is performed until the gradient descent converges. This can mean until a satisfactory objective value is reached, or when the computational time budget is expended, or when consecutive improvements in the objective value fall below some threshold. In our usage of this method, we employ the same termination criteria as already described in Section 4.2.4.1.

4.3.2.2 Second Stage: Orthogonalization

The second stage of the proposed three-stage algorithm is orthogonalization, where a nonnegative solution is mapped to the closest possible orthogonal nonnegative solution. We will demonstrate this procedure on the SONMTF problem where orthogonality is imposed on matrix G, however, this can be easily expanded for other types of NMF problems.

It is important to understand that when matrix G is both orthogonal and nonnegative, it will have at most one nonzero component in a row. To prove this, let us denote the i-th column of G as g_i and let us take two columns g_a and g_b of matrix G. If $a \neq b$, then it follows from orthogonality that $g_a \cdot g_b = 0$ and because all components of G are nonnegative, this means that all summands in the scalar product $g_a \cdot g_b$ are zero as well. In other words, $G_{ia}G_{ib} = 0$, and this holds for all rows i and for arbitrary $a \neq b$. Now suppose that component G_{ic} is nonzero. From $G_{ia}G_{ib} = 0$ for $a \neq b$, it follows that all components in row i are zero except for G_{ic}. This means that at most one component is nonzero in a given row of G if G is both orthogonal and nonnegative.

Now let us consider how to turn an arbitrary nonnegative matrix G into an orthogonal nonnegative matrix so that the result will be as close as possible to the original matrix G, a task we call an orthogonalization. The naive way to perform orthogonalization of nonnegative matrix G is to set all but the maximal component of every row of G to zero. This is the minimal alteration of G with respect to any p-norm $\|G\|_p$ and produces orthogonal nonnegative G. However, this is not necessarily the best way when considering

how this matrix G is used in our model. More ideally, we should perform orthogonalization in such a way that would least change the objective value MSE when moving from the current G to its orthogonal version. With respect to our optimization problem, not all columns of G are equally important. For example, if $S_{ai} = S_{ia} = 0$ for all i, then column g_a does not contribute to the MSE at all. In this case, column g_a is utterly unimportant and should be set to zero when performing orthogonalization, no matter how high the values in that column are. This indicates that values of matrix S should also be considered when performing orthogonalization.

Given the nature of the underlying factorization problem, the more relevant measure of how much objective value MSE changes when G changes is $\|GSG^\top\|$. When G is orthogonalized, the difference between orthogonal and non-orthogonal version should be small with respect to this measure. In this way, the values of matrices S also contribute as opposed to the naive way outlined in the previous paragraph. One way to achieve this is by scaling G and S in the following way:

$$\bar{G}_{ik} = G_{ik}h_k, \qquad \bar{S}_{kl} = \frac{S_{kl}}{h_k h_l}, \qquad \text{where} \qquad h_k = \sum_{jl} S_{kl}G_{jl} = \sum_{l} S_{kl} \sum_{j} G_{jl}. \tag{4.38}$$

Vector h can be interpreted as the column sum vector of matrix G multiplied by S. Such scaling has no effect on our model, that is, $GSG^\top = \bar{G}\bar{S}\bar{G}^\top$. We can easily see that this is true if we look at this scaling in another way, $\bar{G} = GD$ and $\bar{S} = D^{-1}SD^{-1}$, where $D = \text{diag}(h)$. Let us now look at the norm of $\bar{G}\bar{S}\bar{G}^\top$ since we wish that our orthogonalization of $\bar{G}$ has a small impact on this product. It turns out that such calculation is especially simple if, instead of Frobenius norm, we take 1-norm:

$$\left\| \bar{G}\bar{S}\bar{G}^\top \right\|_1 = \sum_{ij} \left| \bar{G}\bar{S}\bar{G}^\top \right|_{ij} = \sum_{ij} \left| \sum_{kl} \bar{G}_{ik}\bar{S}_{kl}\bar{G}_{jl} \right| = \sum_{ik} \bar{G}_{ik} \sum_{jl} \bar{S}_{kl}\bar{G}_{jl} = \sum_{ik} \bar{G}_{ik} = \left\|\bar{G}\right\|_1. \tag{4.39}$$

In this calculation, we used the fact that $\bar{G}$ and $\bar{S}$ are nonnegative and that

$$\sum_{jl} \bar{S}_{kl}\bar{G}_{jl} = \sum_{jl} \frac{S_{kl}}{h_k h_l}(G_{jl}h_l) = \frac{1}{h_k} \sum_{jl} S_{kl}G_{jl} = 1, \tag{4.40}$$

where we used the definition of h_k. This means that with such scaling we achieved that the 1-norm of $\bar{G}\bar{S}\bar{G}^T$ directly corresponds to the 1-norm of $\bar{G}$. Therefore, we can now use the naive approach of orthogonalization where we, in each row of $\bar{G}$, set all but the maximal component of every row of $\bar{G}$ to zero. Intuitively, our scaling weighted the columns of G in such a way that the actual values of components of $\bar{G}$ correspond to the importance of those components to our final model result $\bar{G}\bar{S}\bar{G}^\top$, or more precisely to its 1-norm.

This procedure can be extended to other variations of NM(T)F problems, such as, multi-relational ($R_i \approx GS_iG^\top$), multi-type ($R_{ij} \approx G_iS_{ij}G_j^\top$), or even multi-relational multi-type problems ($R_{ijk} \approx G_iS_{ijk}G_j$). The difference here is only that vectors h are acquired by summing several S matrices (in multi-relational case) or that each matrix G_i has its own vector h_i that is used for scaling $\bar{G}_i = G_i \odot h_i$. But it must be noted that the proposed procedure of orthogonalization is only an approximation in cases where orthogonalization is required on both G_i and G_j matrices for problems $R_{ijk} \approx G_iS_{ijk}G_j$. This is also the case when G_i and G_j are the same matrix, as is the case with symmetric NMTF from our treatment above $R \approx GSG^\top$. This is because vector h is calculated from G itself and each alteration of G, such as zeroing out components, alters the value of h. This means that Eq. (4.39) does not hold any more if we alter components of $\bar{G}$. However, this approximation is accurate in practice, since our orthogonalization procedure eliminates only those components that contribute the least to the 1-norm $\|\bar{G}\|_1$, or equivalently,

to $\|\bar{G}\bar{S}\bar{G}^\top\|_1$. Moreover, our experiments (see Section 4.3.3) show that the matrices G subject to orthogonalization are typically already close to being orthogonal. Consequently, the components removed during orthogonalization are generally small compared to the dominant ones that remain unchanged.

4.3.2.3 Third Stage: Restricted Gradient Descent with Orthogonality and Nonnegativity Constraint

The third stage of the proposed three-stage algorithm is the refinement stage, where gradient descent with nonnegativity is executed again but with a restriction which ensures that orthogonality of the solution is fulfilled throughout the descent. Matrix G generated in the second stage (orthogonalization) has one nonzero value in every row, and during the third stage, the position of this nonzero component will not change anymore. The gradient descent is, in this stage, allowed to change only a single component in a row while leaving others at zero value. This means that the search space dimension is reduced substantially. Implementation-wise, this is achieved by masking the gradients with a pattern that was found during orthogonalization so that the masked gradients never point outside of the feasible region[4]. Using masked gradients, any gradient-based algorithm can be used to refine the values of the nonzero components of G while leaving the zero components constant throughout the descent. The S matrix is also refined during this stage, however, no restriction is made on this part of the search space because the orthogonality constraint only applies to the G matrix and not S. This stage is run until a satisfactory solution is found or some other termination criterion is fulfilled. In our implementation, we set the maximum number of steps equal to the fifth of what is set as the maximum in the first stage of this scheme.

4.3.3 Results on Synthetic and Real-World Similarity Data

In this section, we present empirical results for our three-stage scheme when applied to SONMTF problems. To test our methodology, we used synthetic and real-world similarity data. We tested the three-stage scheme using the Adam algorithm for the gradient descent part since it was found to be most efficient for soft clustering in Section 4.2. Configuration of Adam (initialization, hyperparameters, stopping criterion, etc.) is identical to the one already described in Section 4.2.4.1 with the only difference that in the third stage Adam is run for a smaller number of steps. In our experiments we found that Adam converges approximately five times faster in the third stage compared to the first stage. This is why the maximum number of steps was set to 3000 in the first stage and 600 in the second stage.

We use SE for the objective function and MSE for analysis, as described in Section 4.2 with Eq. (4.19) and (4.20). For easier analysis, we define a measure for orthogonality of matrix G of shape (n, k):

$$\text{orthogonality} = 1 - \frac{\|G^\top G - I\|^2}{k(k-1)}. \tag{4.41}$$

In this definition, we assume that columns of G are normalized $\sum_j G_{jl}^2 = 1$ for all l. This is not true in general, however, we can always normalize G by absorbing norms of columns of G into matrices S_i. Orthogonality measures a degree to which G is orthogonal, and it

[4]More precisely, let G_{orth} be the G matrix acquired by the orthogonalization, which has one nonzero component per row. We define a mask $M = \text{sign}(G_{\text{orth}})$ that has ones where G_{orth} is nonzero and zeros everywhere else. For descent direction, we now use $M \odot \nabla_G \text{MSE}$ instead of gradient $\nabla_G \text{MSE}$. This guarantees that during the descent we never leave the feasible set.

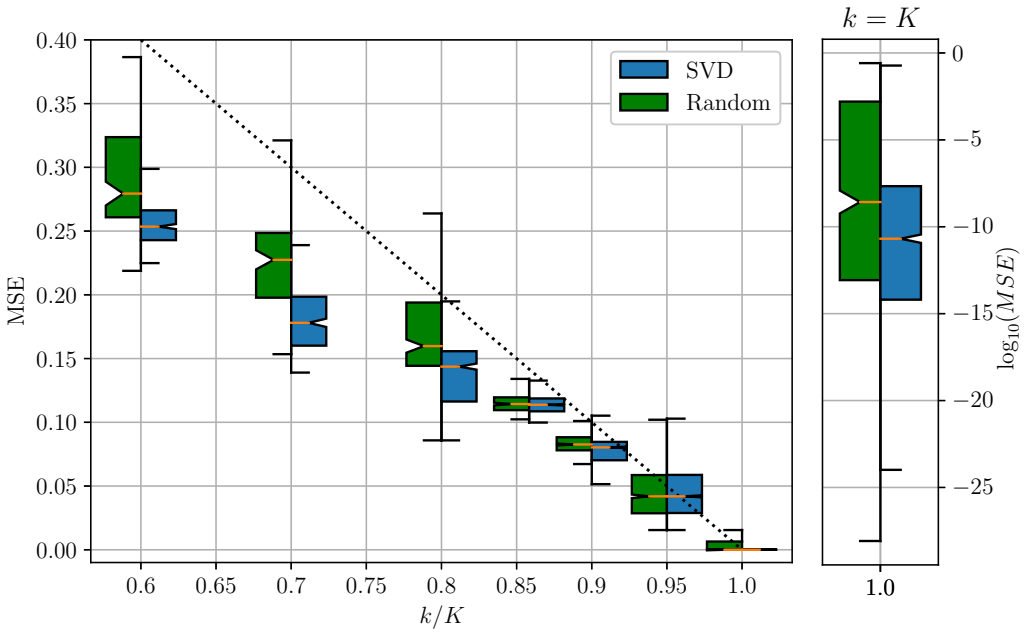


Figure 4.12: Comparison of the best objective values obtained with the three-stage Adam algorithm for different ratios of k/K, where k is the number of columns in G and K the rank of matrices R_i. For better visibility, the case $k = K$ is shown in greater scale on the right and, since the optimization finds extremely low values, using a logarithmic scale.

can be between 0 and 1. Orthogonality is 1 when G is orthogonal and zero when all of the columns of G are identical (for example, if G is an equi-valued matrix $G_{jl} = 1/\sqrt{n}$). Denominator in (4.41) ensures that the orthogonality is always between 0 and 1, regardless of the shape of G.

We first present results for synthetic data sets described in Section 4.2.4.2. This is a collection of 5-tuples of symmetric matrices of dimension spanning from 100 to 5000 of rank spanning from 10 to 50. This data set collection was constructed using orthogonal G matrices, therefore, it is suitable for testing the three-stage scheme as well. Because this collection includes a large number of data sets, we have sufficient statistics to study how the initial point influences the optimization when orthogonality is considered. As mentioned in Section 4.2.4.2, we can use random or SVD-based initialization. SVD produced slightly better results for the nonorthogonal case, however, the improvement in MSE was small (below 0.01). For SONMTF problems, we found that this difference is more pronounced with SVD-based initialization being substantially better in some cases or equally good in others. Figure 4.12 shows aggregated results for all problem instances in the synthetic data collection and shows how random initialization compares to SVD decomposition method.

Based on the findings, we focus on optimization performed from SVD-based initial points. This makes our optimization completely deterministic and makes our experiments less computationally expensive since we require only a single run of the algorithm for each data set in the collection. In Table 4.1 we report the main performance metrics of our three-stage method with Adam compared to one-stage Adam (method from Section 4.2 that ignores orthogonality constraint) for different dimensions of R_i and G when applied to synthetic data sets. To make the analysis as realistic as possible, we pretend not to know the ideal number of clusters K (rank of matrices R_i) and run the algorithms for several different number of clusters k (number of columns in matrix G). We can observe that one-

Table 4.1: Optimization results for the one- and three-stage Adam algorithm when applied to synthetically generated data sets. n is the number of objects in a data set, K is the actual number of clusters in a data set and k is our guess for the number of clusters. If $k = K$, the optimal MSE is known to be zero, while for $k < K$ the optimal MSE is not known. However, it is known that optimal MSE is a monotonously decreasing function of k.

			one-stage Adam			three-stage Adam		
n	K	k	MSE	orthogonality	time [min]	MSE	orthogonality	time [min]
100	30	26	0.0986	0.9935	0.37	0.1112	1.0	0.43
100	30	27	0.0757	0.9953	0.37	0.0852	1.0	0.35
100	30	28	0.0496	0.9970	0.38	0.0556	1.0	0.38
100	30	29	0.0255	0.9985	0.38	0.0274	1.0	0.36
100	30	30	0.0000	0.9999	0.18	0.0000	1.0	0.26
200	30	26	0.1039	0.9931	0.55	0.1154	1.0	0.59
200	30	27	0.0786	0.9952	0.56	0.0877	1.0	0.56
200	30	28	0.0528	0.9969	0.56	0.0593	1.0	0.56
200	30	29	0.0263	0.9982	0.57	0.0301	1.0	0.61
200	30	30	0.0000	0.9999	0.24	0.0000	1.0	0.36
500	30	26	0.0994	0.9936	0.58	0.1103	1.0	0.70
500	30	27	0.0719	0.9954	0.59	0.0806	1.0	0.71
500	30	28	0.0497	0.9967	0.59	0.0544	1.0	0.71
500	30	29	0.0247	0.9981	0.59	0.0268	1.0	0.72
500	30	30	0.0003	0.9996	0.46	0.0000	1.0	0.59
1000	30	26	0.0971	0.9934	2.18	0.1087	1.0	2.65
1000	30	27	0.0740	0.9952	2.18	0.0834	1.0	2.65
1000	30	28	0.0506	0.9967	2.20	0.0556	1.0	2.67
1000	30	29	0.0252	0.9982	2.19	0.0281	1.0	2.66
1000	30	30	0.0001	0.9996	1.73	0.0000	1.0	2.22
2000	30	26	0.0968	0.9931	10.02	0.1076	1.0	12.12
2000	30	27	0.0722	0.9949	10.03	0.0810	1.0	12.14
2000	30	28	0.0485	0.9965	10.06	0.0543	1.0	12.17
2000	30	29	0.0242	0.9980	10.01	0.0267	1.0	12.11
2000	30	30	0.0002	0.9995	6.79	0.0000	1.0	8.89
5000	30	26	0.0921	0.9931	69.13	0.1085	1.0	84.02
5000	30	27	0.0690	0.9950	69.68	0.0803	1.0	84.63
5000	30	28	0.0442	0.9964	69.55	0.0534	1.0	84.60
5000	30	29	0.0246	0.9977	70.42	0.0279	1.0	85.47
5000	30	30	0.0003	0.9994	51.20	0.0000	1.0	66.18

stage Adam finds solutions that are very close to the feasible space (orthogonality ≈ 1). This makes sense for $k \geq K$ since a good algorithm should come close to an orthogonal solution because an optimal solution is known to be orthogonal (this is how synthetic problems were constructed). But when the number of clusters is underestimated ($k < K$), which is more aligned with real-world use of clustering[5], this result is rather unexpected.

When we have a data set that was synthetically constructed to have K clusters and we attempt to cluster this data with a model that allows a strictly smaller number of clusters $k < K$, we would expect that the model would compensate by overlapping clusters for some instances (making clustering softer instead of hard) in order to acquire a more accurate clustering model. This should cause a deviation of the final solution away from orthogonality. This does not happen, though, and it seems that the gradient descent favors solutions that are close to orthogonal subspace at the expense of higher objective value[6]. In other words, the gradient descent is naturally drawn to the region of the search space where the separation of the clusters is large, even if the guessed number of clusters is insufficiently small. This is important because it provides further arguments for our three-stage approach, because it seems that the first stage is unlikely to end up far away from the feasible set. Therefore, our three-stage optimization is less likely to encounter large jumps in the objective value during the second stage.

If we compare the three-stage Adam with the one-stage variant, we can observe (in Table 4.1) that enforcing orthogonality slightly worsens the result when the number of columns of G is smaller than the rank of matrices R_i. This is expected since hard clustering is a more difficult problem compared to soft clustering. Also, one would expect a decrease in model accuracy when additional constraint is enforced. However, if G is chosen to be of suitable size, the orthogonalization improves the quality of the solution. This is less expected, and we speculate that this has to do with the fact that orthogonality also implies sparsity, and it is known from the machine learning literature that sparse models can bring improvements in accuracy [159]. We can also take a look at how well the three-stage method works specifically in the case $k = K$. Figure 4.13 shows the results averaged over rank of R_i for different dimensions of matrices R_i. There we see that, on average, the objective values obtained using the three-stage method are well below any reasonable threshold needed for practical clustering. We can also observe that the best MSE found by the algorithm increases with the size of matrices R_i, however, the average values are extremely small, even for larger matrices.

To better understand the three-stage scheme, it is illustrative to present an example of the optimization progress when solving a SONMTF problem. This is shown in Figure 4.14 for the three-stage Adam algorithm. We can observe that, in the first stage, MSE gradually decreases while orthogonalization increases. In the second stage, MSE drops from MSE $\approx 10^{-3}$ to MSE $\approx 10^{-4}$, while orthogonality jumps to exactly 1. In the third stage, MSE rapidly decreases and in the end we achieve MSE $\approx 10^{-15}$ which is much better than actually needed in practice.

To test our methodology on a real-world use case, we applied it to a problem of clustering scientific bibliographic data. The data set originates from a work performed in [61] and contains co-occurrences of biomedical terms present in the scientific literature. We can understand this problem as a clustering of a weighted graph. This graph has biomedical

[5]The aim of clustering is to better understand the data, therefore, we want a good tradeoff between the accuracy of the clustering model and the number of clusters (see Section 4.2.5 for more information on multi-objective treatment of clustering problems).

[6]Ideal orthogonal solution with $k < K$ necessarily has larger MSE compared to non-orthogonal one, because $k < K$ implies that some clusters were merged, meaning that some instances are misclassified resulting in larger MSE. In non-orthogonal case (soft clustering), misclassification error is not so pronounced due to the possibility of cluster overlaps.

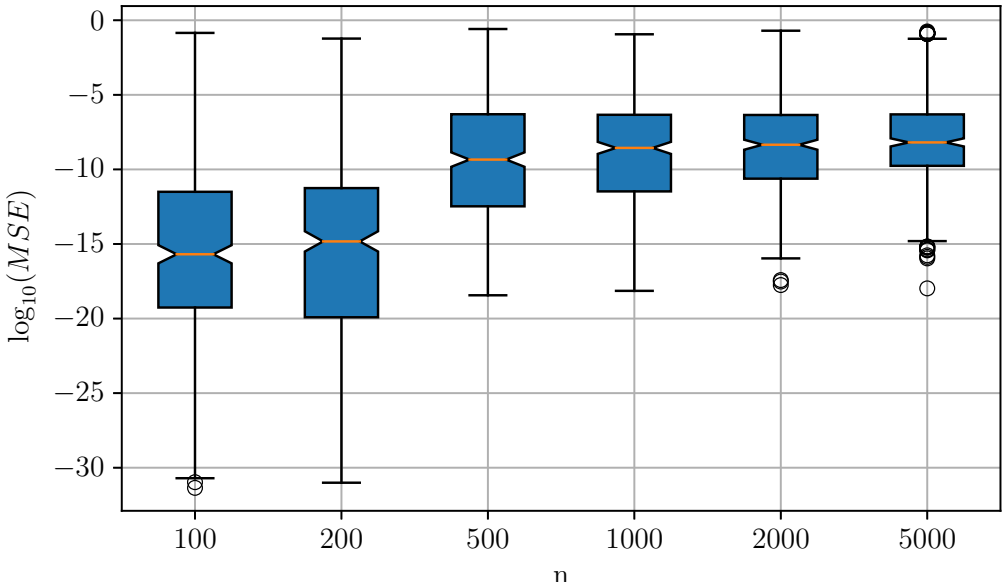


Figure 4.13: Comparison of best objective values obtained with the three-stage Adam algorithm for different sizes of synthetic matrices R_i.

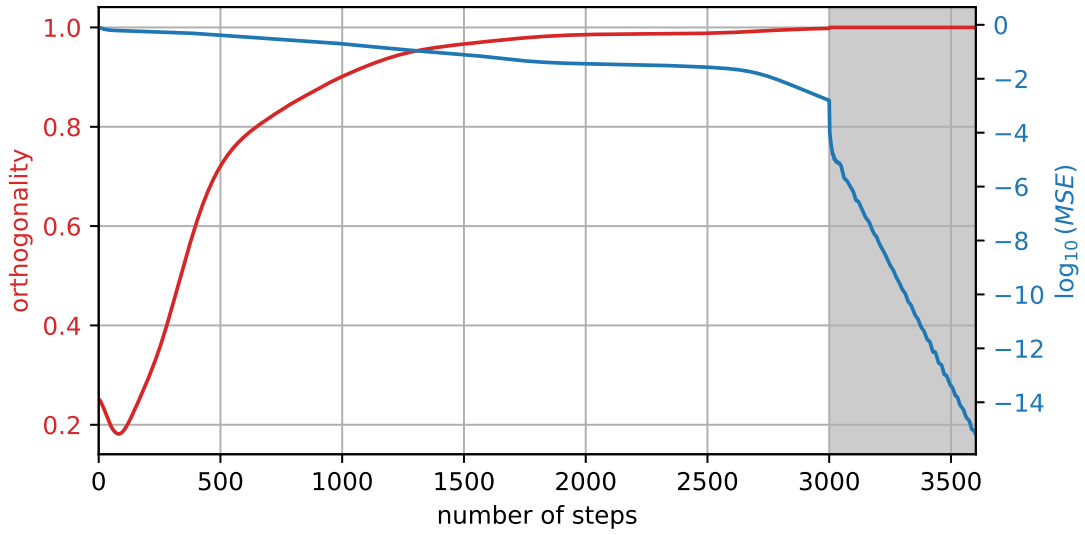


Figure 4.14: Example of the progress of optimization when the three-stage scheme is used in conjunction with the Adam algorithm. The blue line shows how objective value MSE changes in logarithmic scale and the red line how the orthogonality changes with the number of gradient descent steps. The shaded area represents the third stage of the scheme and the border between the white and the shaded area is where orthogonalization takes place (the second stage). The matrix being factorized is an instance from the synthetic data set collection with dimension of $n = 5000$ and rank $k = K = 20$. This means that the global minimum is known to be exactly zero.

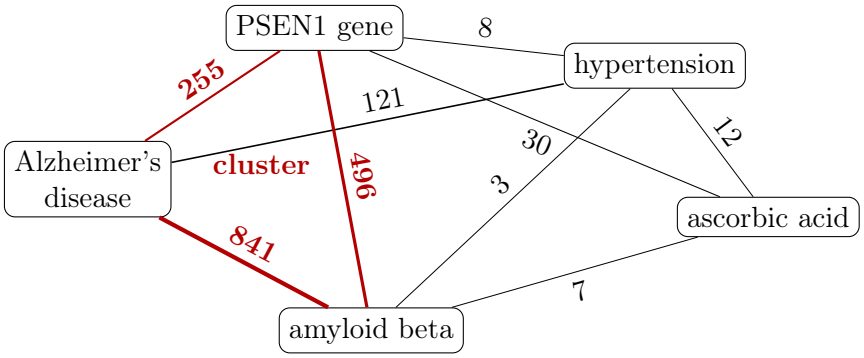


Figure 4.15: Toy example of a weighted graph with one cluster that joins biomedical terms that are semantically more related among themselves than the rest of the terms in the graph.

terms (such as diseases, drugs, genes) for vertices and edges that measure how many times a pair of terms was used in the scientific literature. Figure 4.15 depicts a very simple example of such a graph. In this toy example, there are vertices that are weakly connected and vertices that are strongly connected, such as Alzheimer's disease and amyloid beta. Those two are connected by an edge with a weight of 841, which means that 841 scientific papers included both of those terms in their content. We can observe a cluster that includes terms that are oftentimes used simultaneously in the literature. The specific cluster marked in red is well-connected because of the fact that PSEN1 gene encodes a protein called amyloid beta, and because it was found that amyloid beta plays a role in the onset of Alzheimer's disease. Clustering such data is worthwhile because it can reveal novel connections among biomedical terms that were not discovered yet directly. In this way, such clustering can provide guidelines for future research in the biomedical community.

The real-world data set consists of four matrices. The first three include co-occurrences for 28,693 general biomedical terms while considering the scientific literature in three different time spans (1966-2010, 1966-2015, and 1966-2020). Factorizing different time spans is useful for evaluation purposes. Suppose that we obtain a factorization $R \approx GSG^\top$ of the co-occurrence matrix R for the time span 1996-2010. Now suppose that component $(GSG^\top)_{ij}$ is large while the same component in the data R_{ij} is small. This means that our model predicts that concepts i and j are strongly related while in the data from 1966-2010 indicates that they are weakly related. But it might be that this relation between the concepts had not been discovered yet (as of 2010). If we can show that the scientific literature in 2010-2020 found a strong connection between those concepts (R_{ij} for time span 1966-2020 is large), we would have proven that our model can predict relations between concepts that have not been discovered yet. The full data set (1966-2020) includes 944,878,660 overall co-occurrences with 34,876,520 unique pairs of terms found in the literature. This means that the three matrices considered are sparse with $\approx 4\%$ nonzero components. The fourth data set is a filtered version of 1966-2020 data limited to terms related to COVID-19 and includes 1880 terms with a matrix with 7960 nonzero components (0.2% sparsity).

In this section, we do not focus on the clustering analysis since this entails expert knowledge for proper interpretation of results but rather focus on the aspect of optimization. Some rudimentary results on clustering and interpretation of this real-world data set can be found in [61]. We applied the three-stage Adam algorithm to all four matrices using the same settings as with synthetic data. Table 4.2 reports the main metrics of this

Table 4.2: Objective values for the one- and three-stage Adam algorithm when applied to real-world data sets.

instance	k	one-stage Adam			three-stage Adam		
		MSE	orthogonality	time [h]	MSE	orthogonality	time [h]
1966-2010	5	0.1070	0.9886	25.41	0.1056	1.0	29.68
1966-2010	10	0.0606	0.8627	24.79	0.0572	1.0	29.06
1966-2010	15	0.0536	0.7315	22.01	0.0536	1.0	26.15
1966-2010	30	0.0454	0.5618	25.85	0.0448	1.0	30.39
1966-2010	45	0.0355	0.5855	21.86	0.0366	1.0	26.48
1966-2015	5	0.2034	0.8999	23.70	0.1788	1.0	28.09
1966-2015	10	0.0969	0.8645	22.53	0.0900	1.0	26.79
1966-2015	15	0.0801	0.7984	22.57	0.0787	1.0	26.84
1966-2015	30	0.0620	0.6054	22.40	0.0606	1.0	26.75
1966-2015	45	0.0466	0.6107	22.66	0.0471	1.0	27.65
1966-2020	5	0.2144	0.9921	22.91	0.2137	1.0	27.34
1966-2020	10	0.1179	0.9737	23.69	0.1179	1.0	27.89
1966-2020	15	0.1024	0.8962	28.39	0.1004	1.0	33.48
1966-2020	30	0.0732	0.7578	25.69	0.0732	1.0	30.38
1966-2020	45	0.0561	0.7201	26.02	0.0567	1.0	30.88
covid	5	0.3533	0.8888	0.06	0.4201	1.0	0.13
covid	10	0.1906	0.9090	0.14	0.2511	1.0	0.32
covid	15	0.1385	0.9311	0.15	0.2090	1.0	0.32
covid	20	0.1080	0.9360	0.16	0.1732	1.0	0.36
covid	25	0.0940	0.9442	0.15	0.1549	1.0	0.34
covid	30	0.0848	0.9444	0.19	0.1500	1.0	0.40
covid	35	0.0758	0.9507	0.17	0.1398	1.0	0.38
covid	40	0.0736	0.9439	0.17	0.1449	1.0	0.37
covid	45	0.0666	0.9560	0.19	0.1282	1.0	0.43
covid	50	0.0585	0.9596	0.18	0.1146	1.0	0.39

optimization. We can see that, for the first three data sets, the results are satisfactory with an error of ~ 0.05 of the full matrix norm. What is interesting is, contrary to what was observed on synthetic data, that orthogonality of the best solution decreases with increasing k. For synthetic data, orthogonality was practically independent of k. What we see here is that for low k, the clusters found by the first stage are well separated, but when we allow a larger number of clusters k, clusters found by Adam are starting to overlap which results in decreasing orthogonality. However, after the second and third stage, solutions are found with practically identical (or better) MSE but now orthogonal. This shows that orthogonal solutions are possible, but without the three-stage method, such solutions are not found for larger k by one-stage Adam. This behavior shows that the additional two stages we introduced can divert optimization toward feasible regions without severely impacting the final objective value. For smaller number of k, we can see that the additional two stages, in fact, improve the objective value despite imposing additional constraints.

The COVID-19 matrix shows an opposite behavior with regard to orthogonality for one-stage Adam, it increases when k increases. However, it must be noted that this matrix is much more sparse compared to what we encountered so far. In fact, when $k = 5$,

the non-orthogonal model has 9415 parameters (number of independent components in G and S) while the matrix R holds only 7960 nonzero components. When orthogonality is imposed, the number of model parameters decreases to 1895 (because G than has only one nonzero component per row while the others are constrained to zero). Therefore, this data set is an extreme case, and optimizing our model might result in some overfitting even for small k. However, without more involved cluster analysis, this is impossible to know, and it nonetheless represents an interesting example with regard to optimization. Here MSE falls more gradually with k when comparing it to the first three cases, however, a similar best value is achieved in the scale of ~ 0.05 of the full matrix norm. But contrary to the first three matrices, here, the additional two stages universally increase MSE for about ~ 0.05, independently of k.

At this point, we can confirm hypothesis **H2A** for the hard clustering as well. We have demonstrated that the three-stage methodology developed, in this case used in conjunction with Adam gradient descent, is able to efficiently factor even very large matrices in a way that the solution can be interpreted as hard clustering. Additionally, the three-stage approach showed remarkable versatility that was not initially expected. We have seen that for data sets that include well-separated clusters (synthetic data), the first stage is naturally drawn to the feasible subspace and additional two stages reach exceptionally small error. For data sets where separability is not guaranteed (real-world data) and, therefore, the first stage finds a solution not that close to the feasible subspace, the subsequent two stages are very effective in guiding descent onto the feasible subspace while retaining the low error or even in some cases reducing it.

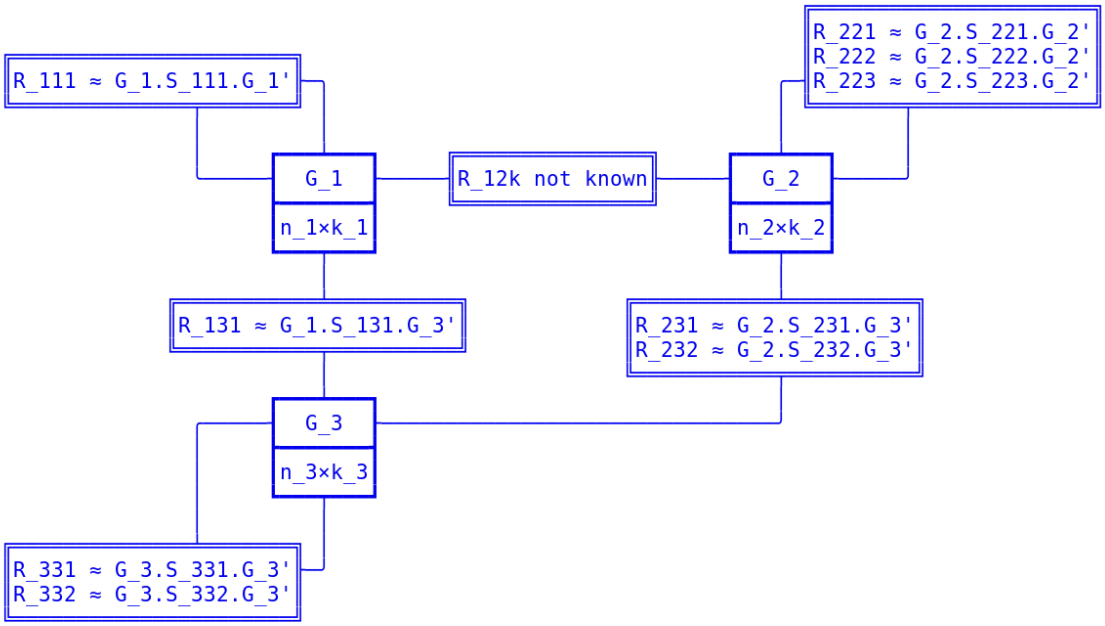
4.4 The `nmfgd` Library for Solving General NM(T)F Problems with Gradient-Based Methods

In this section, we present a general-purpose library we developed for solving all types of NM(T)F problems with gradient-based methods and show the flexibility of this formulation in order to confirm the hypothesis **H2B**. This hypothesis claims that the formulation of (soft) clustering by (orthogonal) NM(T)F brings additional flexibility and allows clustering data from different data sets at the same time, resulting in data fusion and enabling clustering of data sets with (lot of) missing values. The last point is also connected to data imputation because after factorizing a data set with missing values, the factor matrices can be multiplied to acquire a reconstruction of the original data set that, in this form, does not include any missing values. This method can be used to fill out the values that are missing, and such an imputed data set can be used in further machine-learning piplines. Therefore, the NMF method can be used as a stand-alone imputation method. Both data fusion and the ability to work with missing values allow better utilization of the available data, because there is no need to drop any data instances or work on separate data sets individually and, in effect, losing mutual information.

All optimization methods presented in this chapter that employ gradient-based optimization using the full gradient with respect to the entire search space are gathered in a general-purpose library for NM(T)F called `nmfgd` [50]. This library implements the GMELS and the Adam algorithms for all types of nonnegative factorizations mentioned so far. It exploits the automatic differentiation of the TensorFlow library to calculate gradients of any type of factorization scheme, data fusion scheme, constraint, accuracy norm selected, has support for missing values, implements several regularizations to impose factor sparsity and many more features. It does, however, currently lack the support for efficient factorization of sparse matrices.

`nmfgd` software relies on the TensorFlow library to perform gradient calculation by

```
In this example we will be solving a problem with the structure shown below,
where thick rectangles represent all G matrices with their shape and for
each possible pair of G matrices there is a double line rectangle with a
possible data matrices that connects that pair of data types.
```




```
The shape of matrices R_ijk is n_i×n_j and S_ijk is k_1×k_2. The matrices R
are supplied in variable called data which is a two-level nested list and its
structure defines the graph shown above. In this case

data = [ [R_111],               [],             [R_131],
         [R_221, R_222, R_223], [R_231, R_232],
         [R_331, R_332] ].
```

Figure 4.16: A snippet from an example file `GNMTF.py` that shows a problem of simultaneously factorizing 9 matrices describing relations between 3 types of data objects. For some relations, many matrices are available, while for some, there are none available. This example demonstrates the flexibility of `nmfgd` with regards to problem formulation that can imply data fusion of many data sets.

automatic differentiation and to allow parallelization inside one node or execution on GPU. Numpy is the library used for some supplementary matrix manipulations. For reading large text files efficiently, the Pandas library is used (NumPy's loadtxt is extremely slow in comparison for large data sets). There is one optional dependency for performing parameter tuning of gradient-based algorithms for which the `sobol_seq` library is needed. The `nmfgd` library supports a very flexible problem formulation. In its most general case it can factorize problems of form

$$R_{ijk} \approx G_i S_{ijk} G_j^\top, \tag{4.42}$$

where i, j, k indices index different data sets (matrices). In Figure 4.16, we can see a diagram taken from one of the examples of `nmfgd` use that demonstrates one possible way to use `nmfgd` for a more general factorization problem that employs data fusion of many data sets. `nmfgd` can solve both NMF and NMTF problems, that is, problems with or without S_{ijk} matrices. Additionally, one can set if some of the matrices need not be nonnegative. This is useful for cases where some part of the desired factorization scheme does not need to possess clustering interpretation (see [127] for example). The metric that

defines how to calculate the error from the deviation between R_{ijk} and $G_i S_{ijk} G_j^\top$ can be chosen freely with the following four options offered in a more user-friendly way:

	sum	mean
abs	$\sum_{ijk} \lvert R_{ijk} - G_i S_{ijk} G_j^\top \rvert$	$\sum_{ijk} \frac{\lvert R_{ijk} - G_i S_{ijk} G_j^\top \rvert}{\lvert R_{ijk} \rvert}$
square	$\sum_{ijk} \lvert R_{ijk} - G_i S_{ijk} G_j^\top \rvert^2$	$\sum_{ijk} \frac{\lvert R_{ijk} - G_i S_{ijk} G_j^\top \rvert^2}{\lvert R_{ijk} \rvert^2}$

.

Matrices to be factorized can include missing values, in which case missing value masks W_{ijk} are supplied that have zeros on spots where associated R_{ijk} matrices have missing values and ones elsewhere. Implementation-wise the gradient is calculated only on errors that result from known values and ignoring the missing ones. This is achieved by defining objective function in the following way:

$$\sum_{ijk} \left\| W_{ijk} \odot \left(R_{ijk} - G_i S_{ijk} G_j^\top \right) \right\|. \tag{4.43}$$

Three gradient descent algorithms were implemented: vanilla gradient descent, Adam and line search (GM-ELS).

The library implements several penalty functions, including $|G_i^\top G_i - I|^p$ for different values of p, the sum of cosines of angles between all pairs of columns of G_i (also for different powers p), measures that penalize linear dependence of columns of G_i using Pearson correlation among columns of the matrix and $|S_{ijk}|^p$ to induce sparsity (used in [127], for example).

Initial points for G_i and S_{ijk} can be constructed easily using well-scaled random matrices or using SVD decomposition. A proper scaling of the random initial point generation is based on probability theory. Initial matrices G_i and S_{ijk} are randomly generated but scaled in a way that $\|R_{ijk}\| \approx \|G_i S_{ijk} G_j^\top\|$. This calculation can be performed approximately if using 1-norm, and is analogous to the one shown in Section 4.3.2.2.

The three-stage scheme is also implemented and can be used in conjunction with ordinary gradient descent, Adam or line search algorithm. The implementation of the three-stage scheme is agnostic to the form of the factorization problem and can be used for any NMF variant. From an implementation point of view, it is also important to point out that in the orthogonalization stage, S_{ijk} can be adapted in addition to G_i to minimize any changes in objective value. An approximate projection method for S_{ijk} was developed as well, however, it was not thoroughly tested. For now, we suggest to not use any projection operation of S_{ijk}.

4.4.1 Solving Multi-Relational Multi-Type NMTF with nmfgd

In this section, we demonstrate the power of the nmfgd library on a very general example that is shown in Figure 4.16 and is one of the examples for using the library. In order to show the flexibility of nmfgd, we generated a random synthetic problem of soft clustering 3 types of data objects whose clusterings will be held in $G_{i=1,2,3}$ and the relations among these data objects are stored in 9 matrices R_{ijk} in the following way

$$\begin{aligned}
&R_{111} \approx G_1 S_{111} G_1^\top && R_{221} \approx G_2 S_{221} G_2^\top && R_{222} \approx G_2 S_{222} G_2^\top \\
&R_{223} \approx G_2 S_{223} G_2^\top && R_{231} \approx G_2 S_{231} G_3^\top && R_{232} \approx G_2 S_{232} G_3^\top \\
&R_{331} \approx G_3 S_{331} G_3^\top && R_{332} \approx G_3 S_{332} G_3^\top && R_{131} \approx G_1 S_{131} G_3^\top
\end{aligned}$$

as shown in Figure 4.16. To demonstrate that gradient-based algorithms can easily handle missing values, we set 80% of values in R_{ijk} to nan. The problem in its full scope is

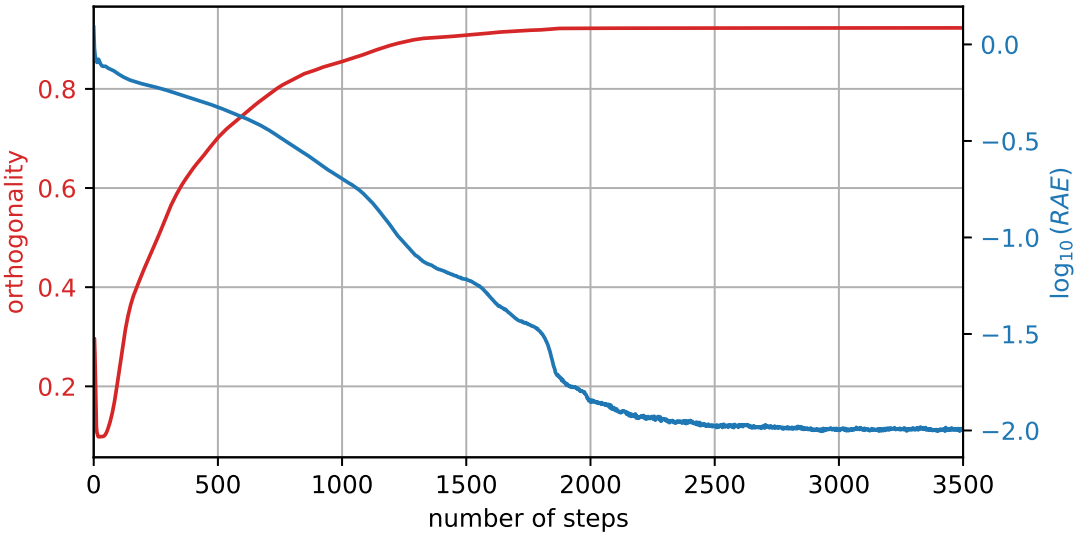


Figure 4.17: Progress of optimization using the Adam algorithm to solve a multi-relational multi-type NMTF problem with 80% of missing values and using a flexible metric choice RAE.

mathematically formulated below. To show that MSE is not the only applicable metric for use with gradient-based algorithms, we chose to minimize objective function (4.44) that uses 1-norm[7] and scales each summand individually with appropriate R_{ijk}. We call this metric relative absolute error (RAE).

With $I = \{(1,1,1),(1,3,1),(2,2,1),(2,2,2),(2,2,3),(2,3,1),(2,3,2),(3,3,1),(3,3,2)\}$ the objective function to be optimized is

$$\text{RAE} = \sum_{(i,j,k)\in I} \frac{\left\| W_{ijk} \odot \left(R_{ijk} - G_i S_{ijk} G_j^\top \right) \right\|_1}{\left\| W_{ijk} \odot R_{ijk} \right\|_1}, \tag{4.44}$$

where $R_{ijk}, W_{ijk} \in \mathbb{R}_+^{n_i \times n_j}$ are constant and $G_i \in \mathbb{R}_+^{n_i \times k_i}$ and $S_{ijk} \in \mathbb{R}_+^{k_i \times k_j}$ are the variables of the optimization[8]. We have chosen $n_1 = 2100$, $n_2 = 3020$, $n_3 = 1803$, $k_1 = 12$, $k_2 = 33$ and $k_3 = 19$. The matrices W_{ijk} have 80% components equal to zero, which means that the model treats those components as missing values.

For optimization the Adam algorithm was used with identical settings as in previous sections. After 3500 gradient descent steps, the objective value reached $\text{RAE} = 9.27 \cdot 10^{-3}$. When the model was applied to reconstruct the missing values of R_{ijk} that was not used during optimization, this error was only slightly higher $\text{RAE} = 9.61 \cdot 10^{-3}$. This proves an exceptional ability for data imputation, even when as much as 80% of data instances are missing. Figure 4.17 shows the progress of the optimization and is in shape quite similar to the ones seen in simpler SNMTF problems.

Here we have shown that the methodology developed in this chapter can be used in a much wider context than we are used to in mainstream unsupervised learning (compared to methods such as k-means, DBSCAN, BIRCH, and so on). We have demonstrated that

[7] 2-norm (MSE) is more prevalent in the literature because it can simplify expressions for some types of algorithms. Up until now, we used MSE solely for easier comparison with other algorithms that are limited to this type of metric.

[8] $\mathbb{R}_+ = \{x \in \mathbb{R} : x \geq 0\}$

our methodology works well on a very complicated model that includes data fusion of many different data sets, all with a large proportion of missing values. We have also demonstrated that our methodology can be used for data imputation (prediction of values on previously unseen data was found to be extremely accurate with RAE $\approx 10^{-2}$). This is especially relevant because 80% of values were unknown, which is a substantial proportion. This also means that our methodology can be used as a part of a preprocessing pipeline in machine learning by using it to fill in the missing data and use such data in other machine learning pipelines that do not naturally support missing values. By these two observations (the ability of data fusion and data imputation), we have confirmed the hypothesis **H2B**. It is worth noting that specialized optimization methods that were specially developed for solving some variants of NMF problems cannot be so easily adapted to employ all these options, even handling orthogonality is quite troublesome for many methods. In this way, we have shown that using our methodology coupled with general-purpose gradient-based algorithms brings additional flexibility, which is an extremely important property in the fast-paced field of machine learning.

4.5 Discussion

In this chapter, we used gradient-based optimization to train different types of unsupervised learning models primarily used for clustering. Since clustering problems are inherently discrete in nature, a special methodology was developed to formulate these problems in a more continuous form which allows the usage of general-purpose gradient-based optimization algorithms. This methodology consisted of rewriting clustering problems into a more general form (as a nonnegative matrix factorization problem) and imposing constraints (that allow clustering interpretation) in a way compatible with gradient-based optimization. By introducing search space transformation (for nonnegativity constraint) and a three-stage scheme (for orthogonality constraint), we developed a framework where one can use any gradient-based optimization algorithm to train clustering models. We found the Adam algorithm to perform especially well. It found clustering models that were even orders of magnitude more accurate than required in practice. By this, we confirmed hypothesis **H2A**, which states that both soft and hard clustering problems can be reformulated into continuous form and solved by gradient-based optimization.

An important part of using gradient-based optimization for training clustering models is a way to calculate the gradient of the objective with respect to search space variables. Most current algorithms in this field that rely on derivatives (such as block coordinate descent) calculate partial derivatives for only a part of the search space at the time and do not consider the full gradient. This simplifies some operations for this specific type of objective functions. We, on the other hand, opted for full gradient calculation in the same way that training is performed in deep learning. Because our aim is to use general-purpose gradient descent algorithms, partial gradients are not directly useful. To calculate the full gradient, we used automatic differentiation as opposed to symbolic differentiation that was used in Chapter 3. Automatic differentiation offered more flexibility and allowed gradient calculation code to be generated without any explicit coding. To perform automatic differentiation, the TensorFlow library was used, which requires the user to define only how the objective function is calculated while the library builds the code for calculating the gradient of that objective. In this way, the process of writing code for the training routine is quite universal for both gradient calculation and optimization. The only exception to this universality is that we require a way of dealing with constraints, which was addressed by the methodology presented in this chapter.

Gradient-based optimization makes sense in this setting because the search space is

very large (its dimensionality scales linearly with the number of data instances). This aligns well with theoretical [13], [14] and empirical [15] results which show gradient-based optimization to be the most effective in large-scale setting. Additionally, gradient-based optimization tend to converge toward wide minima [108], which is beneficial for machine learning applications since it prevents overfitting. We have, up to now, observed this property both in supervised (see Figure 3.10) and unsupervised learning (Table 4.2). More specifically for clustering, we have observed that gradient descent tends toward minima where clusters are well separated rather than overlapping, even in the absence of constraints that impose separation. Furthermore, using our three-stage scheme we have found that explicitly imposing cluster separation oftentimes results in solutions of very a similar objective value than before imposing separation. This indicates that the optimization landscape includes several minima with a similar objective value, with some having larger cluster separation than others. In this manner, gradient descent, out of many possible minima, tends to converge toward the ones that are less sensitive to changes in the data (due to the width of the minimum) and that are easier to interpret (due to higher cluster separation). These properties show that gradient-based optimization should be considered as a legitimate option for training unsupervised learning models.

In unsupervised learning, gradient-based optimization is currently becoming a prevalent method of training dimensionality reduction models [160], while for clustering, training is still largely performed by specialized algorithms [161]. We have seen in our experiments that specialized training algorithms (such as the fixed point method) produce clustering models of similar accuracy as gradient-based optimization algorithms. However, specialized algorithms are not as flexible since they were devised for a specific model in mind and for a specific setting. In many cases, it is difficult to tailor model architecture, loss function, and other attributes to our needs. This is not the case with gradient-based optimization, and, in this chapter, we showed that we are able to train custom models that enable data fusion, that can handle missing values, and allow custom loss functions (not solely mean square error). By this, we confirmed hypothesis **H2B**, which states that continuous formulation of clustering offers more flexibility and enables data fusion and data imputation. This flexibility also opens novel possibilities for the use of gradient-based optimization for training semi-supervised learning models. Since we know that the same training algorithm can be used for both supervised and unsupervised learning, it should also be possible to train specialized models that perform both of those tasks at the same time (semi-supervised learning). This is becoming an increasingly important topic in machine learning since it can circumvent labor-intensive data annotation [162].

When considering the drawbacks of using gradient-based optimization in unsupervised setting, we should mention that the implementation of training routines requires some expertise and experience with frameworks that allow gradient calculation. For the specific case of clustering, however, we developed a user-friendly library `nmfgd` that allows users to deploy gradient-based training with no prior knowledge of automatic differentiation frameworks. It must also be noted that our implementation was, in most cases, more computationally demanding than specialized algorithms. However, we can attribute this to the lack of support for sparse matrices in our implementation. With further effort on the implementation part, we expect this computational burden to reduce considerably and become competitive with specialized clustering algorithms. We believe this to be the case because the TensorFlow library also has excellent support for hardware acceleration (such as computation on GPU) which is currently absent for most implementations of specialized clustering algorithms.

Chapter 5

Reinforcement Learning

A modeling task where the outputs of the model are not known, but there is a way to calculate how well the model performs, is termed reinforcement learning. The archetypal example is the control of a mobile robot where the model receives environmental sensor data as input and returns actions that result in robot movement. There is no way to say what is the optimal action to take, however, there is a way to assess how good the robot's actions were with regard to the goal behind the robot's movement. In reinforcement learning terminology, this quantity is called a reward and during model training the reward is being maximized. The difficulty of reinforcement learning comes from two main sources:

1. delayed reward calculation with respect to performed actions and
2. black-box nature of reward calculation.

The first point means that the reward may not be known instantly after an action is performed by the model (contrary to supervised learning). Instead, reward may be accessible only after a multitude of actions have been performed by the model and this single reward evaluates all performed actions collectively. If the possibility of reward calculation is sparse in time, this slows down the convergence of training since many model actions are needed to achieve only a single objective function evaluation. One way to remedy this is to use a surrogate model for calculating the reward. Such a surrogate is able to approximate the reward not only for some special states but for all states of the environment. In this way, the model that performs actions can be updated for every action made by using the reward calculated by the surrogate model. The surrogate model itself is, on the other hand, updated every time a state is reached which allows real reward calculation [163]. This can speed up the convergence of the model if implemented correctly, however, it can bring additional training instabilities that result from maximizing the surrogate reward instead of the real reward.

The second point will be addressed in this chapter. Contrary to supervised learning, where the objective function usually has a closed mathematical form, in reinforcement learning the objective function (reward) is calculated by a black-box program. This is a limiting factor because many machine learning models have their own preferred training methods that are known to work well and that usually have some requirements and assumptions on the nature of the objective function. However, if reward calculation is a black-box program, these requirements are not usually met. Therefore, when it comes to training, the reinforcement learning community employs a much broader set of optimization algorithms. In contrast with other branches of machine learning where deviations from well-established training procedures are rare in the literature, in the reinforcement learning literature it is not rare to apply various metaheuristics that are not limited by the

black-box nature of the objective function [164]. In this chapter, we will show that, with some caveats, gradient-based optimization is also not limited by this black-box nature and could be applied in a generic way, much like their metaheuristic counterparts.

It must be noted that using metaheuristic optimization is not the most common way of dealing with the black-box nature of the reward calculation in the literature. In fact, the reinforcement learning community developed several constructions that are able to circumvent the black-box limitation while still using the preferred training algorithm for optimization. One reason these constructions might be preferred is that they usually address several limitations of reinforcement learning at the same time [165]. For example, if we use a surrogate model for the reward calculation (due to the problem of delayed rewards), then the objective function values are just outputs of a machine learning model. If our preferred and well-established training algorithm requires an objective function with specific properties, we are able to tailor the surrogate model to this aim [163].

In this chapter, we will focus on deep reinforcement learning which has become the leading methodology in the field with major achievements in autonomous driving [166], game playing [167], robotics [168], control [169], and so on. In deep reinforcement learning, the model is some type of neural network that gets environment variables as input and returns actions. Based on those actions, the reward is calculated and the weights of the model are adjusted in order to maximize the reward. The preferred way to maximize the reward in the literature is by gradient descent [170]. In order to calculate the gradient of the reward with respect to model parameters, one would need to differentiate both the reward calculation and action calculation. In other words, to know how reward changes when we perturb model parameters, we need to know how actions of the model change with respect to this perturbation and how this change in actions affects the final reward. The state-of-the-art methodology is to only differentiate the model, use derivatives of actions with respect to model parameters and reward values to approximate the gradient of the reward. The most common method to construct this approximation is to use a surrogate reward [163], as we mentioned before, or to accumulate the gradients of actions and then update the model by weighted sum of gradients with rewards as weights [171]. The second option is, in its basic form, applicable only to models that return a discrete set of actions of small cardinality. For continuous actions, additional constructs need to be added [37]. However, because the gradient of the reward is, in all cases, calculated only approximately, the training procedure is usually quite sensitive and unstable [170].

In this chapter, we will show that the gradient of the reward can also be calculated exactly without any approximations. We will achieve this by using automatic differentiation to fully differentiate all parts of the objective function calculation which entails differentiation of both the model and the program for reward calculation in a single sweep. We expect that substituting approximative gradient calculation for an exact one would reduce training instabilities and result in faster convergence of the training stage. This methodology is suitable for reinforcement learning problems where reward calculation is made by a known computer program, such as a simulation. This is not, however, universally true in reinforcement learning. In general, the reward can be tied to some measurements made in the physical world. For example, suppose we are training a model for controlling plasma in a fusion reactor by influencing magnetic fields in the chamber [169]. In this case, the reward would be calculated from sensor readings and should reflect how stable the plasma is at any given time. However, training such a reinforcement learning model in its entirety using an actual physical reactor would take a very long time and could also harm the reactor. This is why training is in practice performed using simulations of the environment while the actual physical environment is used only to fine-tune the model. Therefore, reward calculation based on simulation is a widespread setting in reinforcement

learning. Our methodology of differentiating reward calculation is, consequently, relevant for many real-world reinforcement learning use cases.

Part of the work reported in this chapter was also published as a conference paper [59] and a conference abstract [65] in which we explored the problem of protein structure prediction using reinforcement learning.

5.1 Deep Reinforcement Learning with Exact Gradient

Reinforcement learning problems have various equivalent forms alongside with many forms that are more specific or more general [172]. Different ways of looking at these problems can help us to relate methods from other fields and make such methods useful in a wider scope. Probably the most prevalent way of looking at a reinforcement learning problem is that of an agent (controlled by a machine learning model) that performs actions in reaction to its environment with the aim to maximize the reward [165]. An alternative interpretation, and relevant to this thesis, is that of optimal decision-making. In this interpretation, an algorithm searches for an optimal action given the current environment [172]. Such an interpretation makes sense if reward is accessible at all times, which is the case when reward calculation is a computer program, for example, a simulation. In this sense an algorithm is solving an optimization problem every time a new action is performed and ideally finds the action that maximizes the reward. The objective function (reward) is dependent on the environment and, if actions are sequential in nature, this naturally gives a parallel to dynamic optimization problems [173]. In optimal decision-making, we have an objective function (reward) that changes (as the environment changes) and we require an algorithm that can efficiently search and follow the optimum.

If reward is always accessible, we can unify these views into a single formulation that will be the basis for our reinforcement learning methodology. Let the state of the environment be encoded as a vector $s \in \mathcal{S}$ where $\mathcal{S}$ is a set that holds all possible states the environment can be in. We allow s to be any type of object (discrete, ordinal, continuous, mixed) and our only requirement is that we can encode s as a numerical vector[1]. Vector s can also hold information about the past environment (and actions) so that we can also accommodate reinforcement learning problems with sequential nature (partially observable Markov decision processes) as done in [175]. The reinforcement learning model g, therefore, receives environment state s as input and returns an action $x \in \mathcal{X}$ where $\mathcal{X}$ is a set of all possible actions:

$$\begin{aligned} g &: \mathcal{S} \to \mathcal{X} \\ x &= g(s). \end{aligned} \tag{5.1}$$

How well the model g performs is measured by a reward function f that evaluates an action x given the current environment s. Therefore, the reward function f takes both s and x as an argument and returns reward r which is a real number:

$$\begin{aligned} f &: (\mathcal{S}, \mathcal{X}) \to \mathbb{R} \\ r &= f_s(x). \end{aligned} \tag{5.2}$$

To simplify later discussion, we adopt the notation $f_s(x)$ instead of $f(s, x)$, since this allows us to naturally treat $f_s : \mathcal{X} \to \mathbb{R}$ as a function in its own right and reason about its properties without repeatedly referring to the partially applied function $f(s, \cdot)$. An action x_s^{optimal} is optimal in environment s when the reward is maximal:

$$x_s^{\text{optimal}} = \arg\max_{x \in \mathcal{X}} f_s(x) \tag{5.3}$$

[1]The deep learning literature offers many ways to encode highly diverse objects as numerical vectors, such as one-hot encoding, embeddings, positional encoding, and so on [174].

An optimal reinforcement learning model g^{optimal} is such a model that always returns an optimal action:

$$g^{\text{optimal}}(s) = \arg\max_{x \in \mathcal{X}} f_s(x) = x_s^{\text{optimal}}. \tag{5.4}$$

We see that the model g^{optimal} returns a solution to the problem of maximizing $f_s(x)$.

From an optimal decision-making perspective, f_s represents an objective function defined on a search space $\mathcal{X}$, and the ideal model g^{optimal} returns the solution to that optimization problem. Here we can make a connection to the field of specialized heuristic optimization algorithms that are specially developed for a specific class of optimization problems. For example, there are specialized heuristics known to work well for vehicle routing [176], production scheduling [177], protein folding [178], and so on. Heuristics are especially useful if similar problems need to be solved repeatedly. In this regard, a set of functions $\{f_s : s \in \mathcal{S}\}$ is a special class of objective functions and s parametrizes that space of functions. For example, in production scheduling, s would encode quantities and deadlines for products that need to be produced and x would encode a schedule for producing those products. To find the optimal schedule x^{optimal}, one needs to minimize the cost function f_s, which assigns a cost to each schedule x. In practice, if many such minimization problems for different s need to be solved frequently, a tradeoff between computational costs and solution quality needs to be addressed. Where we set this tradeoff is probably the most meaningful measure for separating the concepts of reinforcement learning and optimal decision-making. In reinforcement learning, the authors tend to prioritize the speed of the algorithm that generates actions because the actions are needed very frequently (for autonomous driving it is of the order of 10 ms [179]). In optimal decision-making, on the other hand, more computational resources can be invested in order to get closer to the optimal action (in production scheduling the schedule is usually generated overnight or over the weekend [180]).

In this chapter, we focus on deep reinforcement learning which means that g is some type of neural network that, based on environment vector s, returns an action $x = g(s)$. Because neural networks are known to be very fast in the prediction phase[2], our methodology is more appropriate for problems where quickly acquired approximate solutions of optimization problems (that belong to a given class of problems $\{f_s : s \in \mathcal{S}\}$) are required. It must be noted that approximately solving optimization problems quickly is not an easy task and for our neural network model g we require that it can do so for all problem specifications $s \in \mathcal{S}$. In general, $\mathcal{S}$ can be extremely large and the neural network would need to be trained so that each problem specification s is accurately mapped to the optimal solution x_s^{optimal}. It is known that a neural network can approximate an arbitrary function to an arbitrary precision [90], therefore, g^{optimal} can be approximated well using a neural network. However, it is not known how large such a network would be and whether it is possible to find it using the established training techniques.

In order to find neural network g that would accurately approximate function g^{optimal}, we can use different training strategies. The most straightforward is to follow supervised learning methodology and train model g by minimizing an objective function

$$J = \sum_i \left\| g(s_i) - x_{s_i}^{\text{optimal}} \right\|, \tag{5.5}$$

where $\| \cdot \|$ is some metric that is meaningful for our class of problems $\{f_s : s \in \mathcal{S}\}$.

[2]Deep neural network training is usually quite computationally demanding. However, when the network is trained, using this model to generate predictions is very fast. This is not universally true for all machine learning models. For example, models that belong to the class of kernel methods (support vector machines, Gaussian processes, kriging, and so on) are examples for which predictions can also be quite computationally expansive [55].

Objective functions of these type are routinely encountered in supervised deep learning and the optimization method of choice is gradient descent applied to minimize J with respect to parameters of network g [9]. Such training can be easily implemented and the gradient calculation is, due to J having a closed mathematical form, trivial to perform. It is important to note that the optimization landscape for finding g is not similar to f_s landscape. Network parameters encode a strategy for predicting x_s^{optimal} from s, so optimization is not performed on a single problem encoded by s, but for all possible s at once.

However, the supervised learning approach has several critical drawbacks. To minimize J, we need to have access to optimal solutions $x_{s_i}^{\text{optimal}}$. These are not known in general. They can be generated before training by using some external optimization algorithm that finds $x_{s_i}^{\text{optimal}}$ for many different s_i. In this way, we can build a large data set of pairs $(s_i, x_{s_i}^{\text{optimal}})$ that is used during the minimization of J but this can lead to substantial computational costs. Also, solutions found in this way may only be approximately close to optimal ones and poor data quality usually results in a poor model g after supervised training. Another important drawback is that this approach requires the user to define a meaningful metric in the action space $\mathcal{X}$. Norm $\|\cdot\|$ in (5.5) is measuring a difference between an ideal action and the action generated by our model. From an optimal decision-making perspective, this means that we require a concept of a distance in the search space. However, the search space could be populated by practically anything (permutations, graphs, ordinal variables, mixed). This is why the supervised approach, in general, requires some extra thought to base this distance metric on expert knowledge related to the problem class $\{f_s : s \in \mathcal{S}\}$. Nonetheless, there are examples of this approach in the literature with data instances generated via optimization [181] and also by using $x_{s_i}^{\text{optimal}}$ measured in nature [182].

A different approach is to employ reinforcement learning which circumvents the need to know the optimal solutions $x_{s_i}^{\text{optimal}}$ during training. Instead of minimizing the "distances" between optimal solutions and model actions $g(s_i)$ we maximize the reward

$$E = \sum_i f_{s_i}(g(s_i)). \tag{5.6}$$

This is much less convoluted from a theoretical point of view because reward is the natural measure to assess the quality of the model g. However, in this formulation, the gradient needed for training g with gradient descent is no longer trivial to calculate. To calculate the gradient of E with respect to parameters of g, we require the Jacobian matrix of reward function f_s. Since f_s is most commonly a simulation, the implementation of gradient calculation becomes more involved.

In the literature, two common approaches are used to circumvent differentiating through a simulation. The first is Q-learning, where a second neural network Q is introduced to approximate the reward function f and is trained alongside the model g [163]. The training procedure involves solving two optimization problems simultaneously:

$$\begin{aligned} &\text{maximize} \quad \sum_i Q_{s_i}(g(s_i)), \\ &\text{minimize} \quad \sum_i \|Q_{s_i}(x_i) - f_{s_i}(x_i)\|. \end{aligned} \tag{5.7}$$

This approach is generally considered easier in terms of gradient computation, at the expense of having to train two models, g and Q, in parallel. With regard to gradient calculation, the differentiation of f_s is now substituted for differentiation of Q_s. Since Q_s is a neural network, gradient can be calculated trivially using the backpropagation algorithm.

The second method commonly used in the literature to circumvent the differentiation of f_s is the policy gradient method. This method requires g to be a probabilistic model [171]. This means that g does not return a single action but a probability distribution over actions. In this case, the reward function is

$$F = \sum_i \mathbb{E}_{x\sim g(s_i)} \left[f_{s_i}(x) \right]. \tag{5.8}$$

Here, the expectation is taken over x sampled from the distribution $g(s_i)$, that is, x is a random variable with probability density $g(s_i) = \mathrm{d}P/\mathrm{d}x$. In this formulation, when $g(s)$ is a distribution, the gradient of the reward (with respect to parameters of g) can be written as [171]

$$\nabla F = \sum_i \mathbb{E}_{x\sim g(s_i)} \left[f_{s_i}(x_i) \nabla \log g(s_i) \right]. \tag{5.9}$$

The advantage of such gradient calculation is that it does not involve differentiation of f_s. The drawback, however, is that the expectation calculation is required which involves sampling over all possible actions. This method is straightforward to implement only if the action space $\mathcal{X}$ is discrete and small because, in this case, the expectation $\mathbb{E}$ becomes a summation over a small set. The policy gradient method requires additional constructions for problems where the space of actions $\mathcal{X}$ is continuous, and when strictly deterministic models are needed [37]. Both of the methods mentioned above have many variations and further subtleties. The equations above are meant to illustrate the basic ideas of those methods.

Our methodology in this chapter is to employ exact gradient calculation. We will maximize E from (5.6) via gradient descent using exactly calculated gradients, without approximations using surrogates (as in (5.7)) or sampling for expectation calculation (as in (5.9)). Fortunately, derivation of f_s can also be performed using automatic differentiation, that is, applying chain rule coupled with dynamic programming. There are many libraries that perform such differentiation, for example, Theano [25], TensorFlow [26] and CNTK [28]. In this work, Theano was used to write expressions for the calculation of E. These expressions are then transformed into a computational graph for the calculation of E which can be used to build a computational graph for gradient calculation ∇E using the chain rule. In this respect, procedure is returned for analytical gradient calculation ∇E without any assistance from the user. Computational graphs for E and ∇E calculation can be compiled to `C++`, `CUDA` or `OpenCL`, which brings multi-processor support and can easily be accelerated on GPU or even FPGA [183]. Therefore, the use of Theano is beneficial even if just a calculation of E is needed.

Therefore, by using automatic differentiation with Theano, the neural network g can be trained using gradient descent so that E from Eq. (5.6) is maximized without any additional mathematical constructions. The user needs to supply only the code for the calculation of E and the gradient calculation ∇E is generated by the library. This does bring some further work with regard to implementation since the code for the calculation of E needs to be written in a way that is supported by an automatic differentiation library. Also, there is a learning curve for using automatic differentiation libraries because they usually employ a function programming style which is not the most common paradigm in programming. However, automatic differentiation libraries are numerous, and there is support for differentiating programs in many different languages, for example, C++ [184], Fortran [185], Java [186], and many more. If our reward calculation is written in several different languages, the code segments need to be differentiated separately which makes implementation more involved.

Compared to the supervised learning approach, our method is not limited by the amount of training samples. They can be drawn from a desired distribution over s_i without

major computational effort. Comparing J from (5.5) and E from (5.6) we see that one training sample for J costs us one run of some optimization algorithm while one training sample for E costs us only one objective function (reward) evaluation[3]. These two costs are orders of magnitude apart. It is true, though, that the training samples for J can be reused during training (the number of times a sample is used during training is called the number of epochs). However, the number of epochs in neural network training is typically ~ 10, while the number of function evaluations in a well-designed heuristic optimization algorithm typically goes into thousands or millions. This means that our methodology should be orders of magnitude faster compared to the supervised approach. Additionally, using E instead of J, there is no overhead of implementing and testing efficient optimization algorithms to generate high-quality training samples or the need for samples to be acquired by experimental measurement.

Compared to Q-learning and policy gradient method, our method has no approximations which should be beneficial for training convergence. In addition, the computational complexity of one iteration should be slightly smaller since there are no additional constructs, such as sampling or additional network training. However, both of those methods are well established and there are good implementations available, while our method requires more programming, especially if the user is new to automatic differentiation. Nonetheless, the approximative nature of Q-learning and policy gradient method make training more sensitive and less stable. It is customary to invest a fair amount of time into fine-tuning the training method and to find specific variation of those methods in order to get a satisfactory convergence of gradient descent. Our methodology is aimed at reducing this burden at the expense of implementing reward in a way applicable to automatic differentiation.

5.2 Application to a Protein Structure Problem

To demonstrate the usefulness of our methodology (presented in Section 5.1), we will apply it to a protein structure problem. This is an optimization problem that needs to be solved frequently as a part of several computational pipelines used to, for example, discover new drugs, catalysts, new imaging reagents, molecular motors, and so on [187]. In this respect, the protein structure problem belongs to the category of optimal decision-making problems as mentioned in Section 5.1. It is an optimization problem of determining the 3D structure of a protein from a sequence of its residues. A protein is a chain of amino acids, and the interaction among its constituents and the solvent determine its 3D structure, which in turn determines how the protein interacts with other proteins, DNA/RNA, and small molecules. The native 3D structure of a protein is achieved when the free energy of the structure is minimal. Therefore, using notation from Section 5.1, we have

$$x_s^{\text{optimal}} = \arg\min_{x \in \mathcal{X}} f_s(x), \tag{5.10}$$

where s is the sequence of amino acids, x^{optimal} is the 3D structure (such as position and orientation for each amino acid in the sequence s), and f_s is a function that returns the free energy of a protein based on its 3D structure x^{optimal} and the sequence of its constituents s. In general, function f_s is a simulation that involves electrostatic field calculations of how amino acids in a given spatial configuration influence the polarity of themselves and the solvent (usually water) and how the solvent influences them in return [188].

[3]More precisely, to calculate the gradient of E using automatic differentiation, the cost is approximately two times the cost of function evaluation f_s.

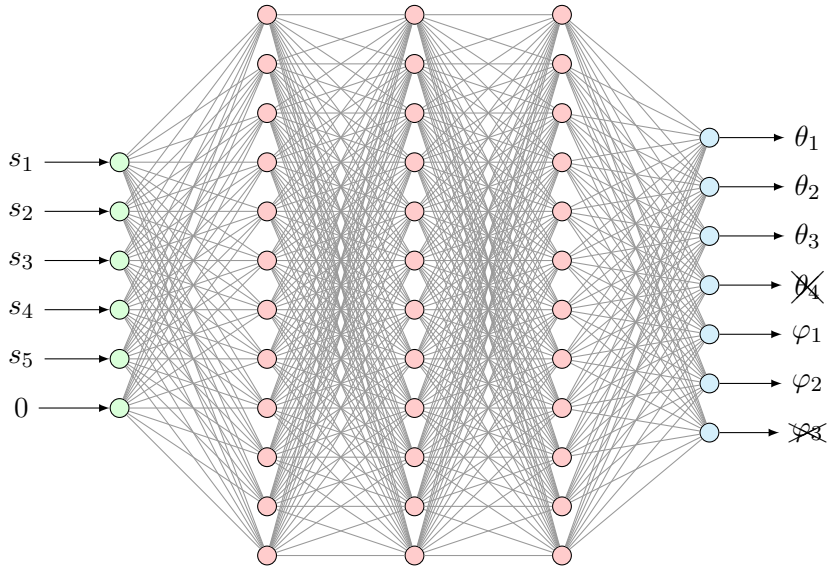


Figure 5.1: Small-scale example of how network g receives the protein specification and how its output is interpreted. If a protein is shorter than the size of the network's input layer, zero is placed on sites where there are no amino acids. In this case some angles from the output are discarded (crossed out outputs). Reprinted from [59].

Comparing optimization problems (5.10) with (5.3) from Section 5.1 we see that the protein structure problem is already in the right form to apply our methodology of deep reinforcement learning with exact gradient. What remains are details connected to implementation. For free energy calculation function f we used generalized three-dimensional AB off-lattice protein model detailed in [59]. In this model, each amino acid in s is described using hydrophilicity which is a single real number that conveys to what degree an amino acid binds to water. This means that s is a real vector with a length equal to the number of amino acids in a protein n. The 3D protein structure x_s was encoded using angles θ_i and φ_i with some of them fixed in order to eliminate rotational symmetry (see [59] for more details). The number of angles needed to fully describe the 3D structure in this encoding is equal to $2n - 5$, making x_s a real vector of length $2n - 5$.

We chose a neural network with dense layers to be the machine learning model g that generates x_s out of s. Since both s and x_s are sequences, it might be tempting to use architectures from time series modeling such as recurrent or convolutional networks, however, it turns out that parts of the protein that are far apart in s can be very close in x_s. This means that interactions between all pairs of amino acids in a protein need to be considered, not only between the ones that are close with regard to their position in the chain. This makes the use of network architectures developed for time series modeling unsuitable for this problem. The most standard network architecture that allows modeling correlations between any pair of input features is a dense network. However, the size of such a model grows quadratically with the size of the input. This is why we have set a limit on the length of the input for model g so that it can only generate 3D structures for proteins with up to $n = 100$ amino acids. This means that the input layer of the model has a static size of 100. Following the input layer are hidden layers with a size of 200 and the output layer with a size of 195, which is aligned with the size of x_s when $n = 100$. The network was implemented in such a way that s of a shorter length than 100 is also a valid input by using zero padding. In this case, the redundant angles in x_s are also ignored in the structure of the protein. Depiction of how this works is shown in Figure 5.1. Rectified

linear units were used as activation functions on hidden layers and tanh on the output layer to ensure that $\theta_i, \varphi_i \in [-\pi, \pi]$.

In accordance with our methodology, we trained the model by minimizing

$$E = \sum_{i=1}^{B} f_{s_i}(g(s_i)), \tag{5.11}$$

where s_i are proteins in a given batch of size B. In each step of training, a batch of proteins s_i of random length was randomly selected. Distribution over length was uniform, and over hydrophilicity, a mixture of two Gaussians with mean at ± 1 and standard deviation 0.15 (see [59] for more details about this choice). By experimenting with different batch sizes B, a good balance between speed and accuracy was found at $B = 512$. For gradient-based optimization, we used the Adam algorithm [73]. Informally, model training works in such a way that in each step, the network g tries to solve 512 random protein optimization problems and gets updated in the direction that would improve its solving capabilities for those 512 proteins. Because network g gets a different batch of random proteins in each step, it converges to a state that is able to solve all protein optimization problems equally well. Picking training batches randomly also ensures that the network g cannot overfit since duplicates in the training data are extremely unlikely. Therefore, the quality of our models during training is not a biased estimate of its quality, and the test quality is, in fact, calculated equivalently to the training quality.

To calculate the gradient of E with respect to network parameters, we used the Theano library. This means that we implemented the calculation of E (including free energy calculation f) using constructs supplied by Theano and then used the Theano differentiation engine to construct the code for the calculation of the gradient ∇E. Implementing the generation of random proteins s_i as well as operations connected to the neural network g is rather rudimentary in Theano. However, implementing f in Theano is more involved. Porting simulations to frameworks such as Theano means reimplementing them in a functional programming style (to some degree). This is beneficial because it allows the compiler to find more ways for parallelization, however, it is also time-consuming because most code in scientific programming is still written in imperative style. This can make translations from one style to another more time-consuming.

Before training, the initial weights of our neural network g need to be set. Initial weights were generated randomly using Glorot initialization [189]. However, we observed that this initialization sometimes causes very large initial gradients. This has to do with the fact that initial g can predict very densely packed structures x where some amino acids can be very close to one another. Since free energy function f has singularities in such cases the gradient can also be extremely large. Therefore, a prefactor w was added to initialization, and neural network models with $w = 0.1$ and $w = 1.0$ were trained. To mitigate the effect of these singularities during training, we used gradient norm clipping. In other words, if the gradient length exceeds a given threshold, the gradient is clipped so that its length is equal to the threshold.

The quality of our neural network g depends on the number of layers. This dependence is usually monotonic [9], but for this problem, this is not the case (see Figure 5.2). This can be attributed to the sensitivity of training to the selection of the initial weights. Neural networks were initialized using random matrices with small components. Therefore, the magnitude of neural network output falls exponentially with the number of layers. In other words, an initial neural network with a small number of layers predicts very folded structures, while an initial neural network with a high number of layers predicts practically straight structures. Figure 5.2, therefore, also conveys that initial neural network predictions should not be very folded nor very straight. The best initial models are somewhere in

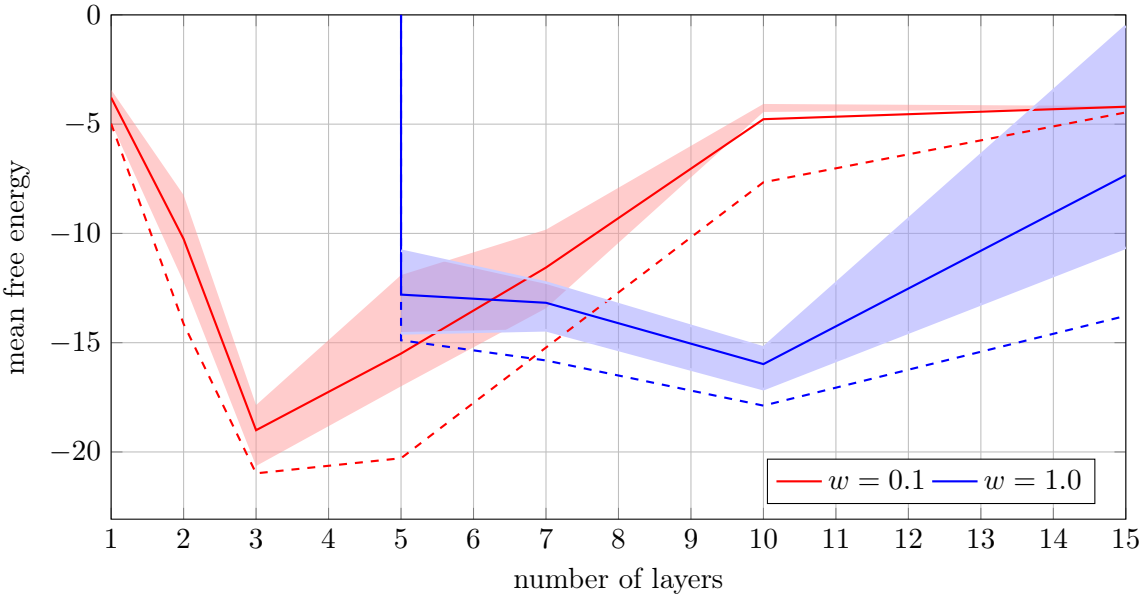


Figure 5.2: The objective function value achieved by our models with respect to the number of hidden layers for two different magnitudes of initial weights w. The full line is the mean objective function value achieved by the models, the shaded area shows the range of the objective function for the central 66% of the models, and the dashed line is the objective function of the best model. To obtain these values, 100 runs of training were performed for each configuration. Reprinted from [59].

between. The best neural network models found have three layers. Given that our models are able to partially solve an optimization problem this is a surprisingly shallow neural network architecture.

We have observed that gradient norm clipping was very beneficial for model training. Figure 5.3 shows the progress of training with and without the use of gradient norm clipping. Occurrence of very high gradients is rare, however, we observed that they substantially disrupt the training progress. In this regard, we cannot fully confirm hypothesis **H3** because we can still observe training instabilities that are not so pronounced in other branches of machine learning. It seems that the instabilities are not the sole results of non-ideal gradient calculation, which is otherwise characteristic of reinforcement learning. In our work, the gradient was exactly calculated through the entire pipeline, and the training still experienced unusual jumps during optimization. It is, however, true that the instabilities were not so severe. We were able to remedy them with simple gradient clipping without resorting to more advanced techniques characteristic of standard reinforcement learning.

We now have trained neural network models that are able to predict a solution to an optimization problem $f_s(x) = \min$. But to fully assess the quality of these predictions, we need to compare them to the actual optimal solutions. The exact solutions are not known for this specific class of objective functions, however, we can find good approximations to them using established optimization algorithms that were found to be especially successful for this class of problems. A comparison study of optimization algorithms performed for this specific objective function class revealed that differential evolution is especially suitable for this problem [190]. Therefore, we can compare the objective function obtained by differential evolution to the objective function of the prediction by our model. Specifications of the implemented differential evolution algorithm were taken from [190], but

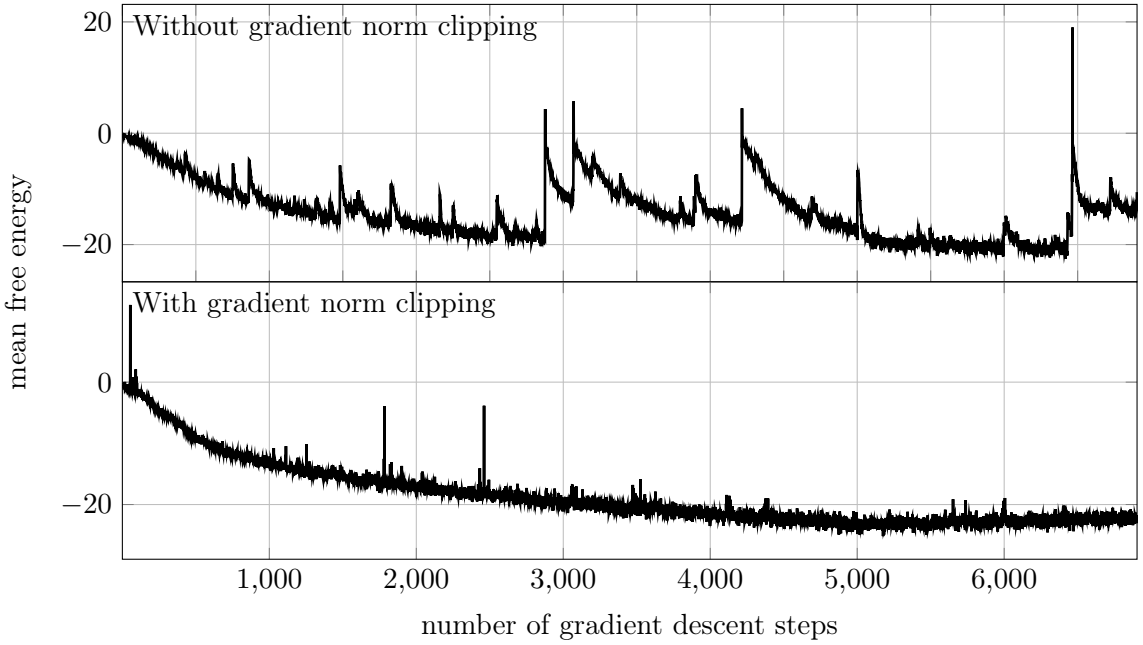


Figure 5.3: The objective function value during training. The upper plot shows the progress of a usual gradient descent procedure, while the lower plot shows progress when the gradient norm has a predefined upper bound by using gradient norm clipping. Reprinted from [59].

Table 5.1: Free energy calculated for three proteins found in nature by using structures predicted by our neural network model and by differential evolution (DE).

protein	length	our best	DE mean	DE best
1CB3	13	−4.6235	−2.1513	−6.7700
1CRN	46	−42.765	−64.948	−79.906
2EWH	98	−65.163	−148.32	−170.47

without parameter control. We used the `best/1/bin` type with a population size of 100. Mutation with dithering was employed with a mutation constant taken between 0.1 and 1, and the recombination constant was 0.9. For comparison, we have chosen three proteins found in nature of length 13, 46, and 98. Free energy of structures predicted by our neural network model and by differential evolution is shown in Table 5.1. The differential evolution algorithm was run 30 times to get a more reliable estimate for the minimum free energy value. For the sense of scale, Table 5.1 also reports the mean objective value obtained over 30 runs of differential evolution algorithm.

We can observe that the quality of structures predicted by our neural network model degrades when protein length increases. This is expected since the complexity of the optimization problems also grows (exponentially) with protein length. Based on Table 5.1, our neural network model predictions are never especially close to the best known solution, however, they do provide less than a trivial approximation of the protein structure that can be useful in some other way. In a setting where this type of optimization problems need to be solved repeatedly, we can benefit from these predictions by using them as an initial

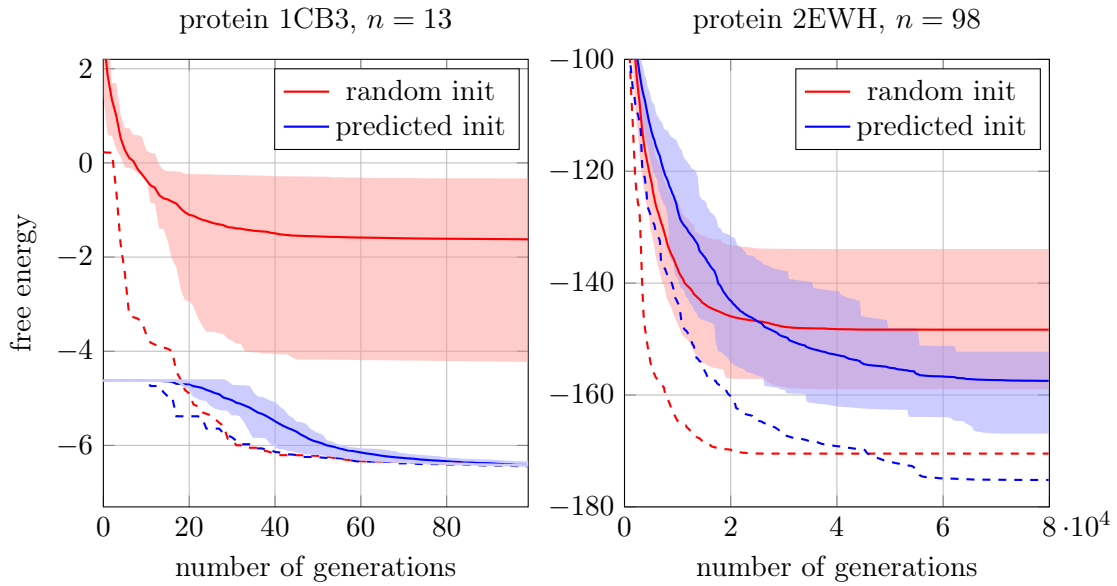


Figure 5.4: Progress of 30 runs of differential evolution by using a random initial population or initial population partially populated with solutions predicted by our neural network models. The full line is the mean over all runs, shaded area shows the range of the central 66% of runs, and the dashed line shows the best run. Reprinted from [59]

point for a more reliable optimization algorithm and possibly accelerate its convergence. We tested this strategy using an already implemented differential evolution algorithm. We performed another 30 runs of differential evolution algorithm for the three proteins from Table 5.1 with initial population where 50% of candidates were predicted by our neural network models. Since we performed 100 training runs, we are able to generate up to 100 different initial solutions. We found this strategy to be beneficial. When using the predicted initial population, differential evolution converges to lower values of free energy (see Figure 5.4). But the convergence is slightly less rapid for the predicted population, which could indicate that the population is less diverse. On the other hand, differential evolution progress is less dispersed among runs which means that fewer runs are needed to find a satisfactory solution. In the case of the 1CB3 protein, the predicted structures are, in fact, so good that all runs converge to the best known solution in just 100 generations. For larger proteins the predicted solution are not as good, however, this strategy performs considerably better when using the predicted population compared to the random one.

5.3 Discussion

In this chapter, we demonstrated that it is possible to train deep reinforcement learning models using gradient descent without resorting to any approximative gradient calculation methods. We have shown that, in this setting, automatic differentiation is sufficient to differentiate the objective function in an exact manner even if it cannot be formulated in a closed mathematical form. Our methodology consists of reimplementing reward calculation into a programming framework that allows off-the-shelf automatic differentiation (for example Theano). This allowed us to use standard gradient descent training techniques characteristic of supervised learning without excessive fine-tuning. In this sense, we substitute the burden of fine-tuning additional reinforcement learning hyperparameters (a

process that is also usually plagued by high levels of noise) for the burden of reimplementing reward calculation code. In order to appreciate this substitution, it is important to note that, in deep reinforcement learning, there are, in addition to standard supervised deep learning hyperparameters, dozens of additional hyperparameters related to reinforcement learning methodology. For example, the soft actor-critic method has over 20 hyperparameters [191], many without an intuitive default setting. In this regard, we offer a methodology for those who are well versed in the use of modern deep learning libraries (all of which use an automatic differentiation engine) and supervised learning, but are not highly skilled in deep reinforcement learning techniques.

In our use case of protein structure prediction, we have trained deep learning models that are able to approximately solve protein structure optimization problems. The training procedure was not completely straightforward since it required gradient norm clipping and smaller initial weights than in standard supervised learning. We attribute this to the properties of the objective function which is not a completely well-behaved function (it even includes singularities). Comparing our objective function (free energy calculation) to a loss function from supervised learning (which includes absolute value, square, categorical cross-entropy) it makes sense that, in our setting, more care is needed to find a configuration of gradient descent that would work well. In this regard, we have wrongfully assumed that the larger effort needed to train deep reinforcement learning models is primarily due to inexact gradient calculation. In our case, the gradient was calculated exactly, but we still observed moderate difficulties during training. For this reason, we can reject hypothesis **H3** that claimed that training instabilities can be eliminated by exact gradient calculation. It seems that approximative gradient is only one source of training instabilities in reinforcement learning and that possibly complicated geometry of the reward function also contributes to this effect. However, we believe that fine-tuning our training procedure was still easier compared to orthodox reinforcement learning since it included techniques that are also known to supervised deep learning users, especially those that have dealt with recurrent neural networks.

The accuracy of the models trained by our methodology for protein structure prediction might seem quite unexceptional at first glance. But given the computational complexity of applying our prediction model, its accuracy is quite surprising. Our model produces a solution for the price of just 4 matrix-vector multiplications (of size 200), which is of similar order as a single objective function evaluation. That predicted solution is with respect to objective function value as good as the best solution found by differential evolution after hundreds or even thousands of generations. This means that our models have the potential to speed up such optimizations. We have proven that using our predicted solutions as initial points for differential evolution substantially increases the probability that a given optimization run finds the best-known solution. This shows the potential of using reinforcement learning in this way even if the models are unable to predict solutions very close to the optimal ones.

Part II

Large-Scale Optimization

Chapter 6

Continuous Optimization

Continuous optimization is a process of finding the maximum or minimum of a function over a continuous domain, such as open subsets of real numbers. It is a powerful tool for solving complex problems in various scientific and engineering fields. Continuous optimization problems are frequently encountered in, for example, operations research, energy, chemical, electrical, mechanical, civil, and nuclear engineering [192]. Continuous search spaces represent a special category inside all possible objective function domains because they possess a topological and geometric structure with notions such as distance, angle, projection, and so on. This can help with the algorithm design because functions of real variables are studied extensively, and the resulting geometrical knowledge and tools can be used to devise good strategies for efficiently sifting the search space.

One of the most useful notions in continuous optimization is the gradient. Fermat's theorem tells us that the gradient in all (local) extrema of a differentiable function is zero. This gives us an option to even solve optimization problems analytically in some special cases (for example, unconstrained quadratic optimization problems). However, in real-world applications, the objective function usually has many local extrema[1] and inspecting all stationary points in an exhaustive manner becomes intractable. For this reason, general-purpose optimization algorithms usually employ other, approximate search strategies to search for global extremes.

In recent decades, the design of continuous optimization algorithms has been centered on derivative-free methods. General-purpose optimization algorithms are designed to work on any reasonable objective function without assuming the availability of the gradient. Their search logic is usually founded on some geometric considerations that stem from properties of the continuous search space. For example, the covariance matrix adaptation evolution strategy (CMA-ES) keeps track of how the objective function changes in different directions of the search space by learning a second-order model of the underlying objective function similar to the approximation of the inverse Hessian matrix in the quasi-Newton method in gradient-based optimization. Its current state is a distribution from which new candidates are drawn, and the update of the distribution parameters resembles the descent in the direction of a sampled natural gradient of the expected objective function value [193]. This method works very well for objective functions with high conditioning and is a popular general-purpose optimization method.

In CMA-ES, a lot of computational effort is put into finding out how the objective function looks like geometrically. It takes several function evaluations to attain enough information to reasonably approximate the slope and curvature of the given region of the search space. In higher-dimensional search spaces, this fact becomes especially pro-

[1] Alongside local extrema, there might also exist saddle points in the landscape. In these points, which are of no interest in optimization, the gradient is also zero.

nounced since the number of function evaluations needed for this scales between $\mathcal{O}(n)$ and $\mathcal{O}(n^2)$, where n is the search space dimension [194]. This curse of dimensionality plagues many derivative-free metaheuristic algorithms and makes them less suitable for large-scale optimization [24]. To combat these issues, many large-scale variants of metaheuristic algorithms were devised over the years. In case of CMA-ES, we have sep-CMA-ES [195], VkD-CMA-ES [196], LM-CMA [197] and RmES [198], to name some. Even though these variants employ special techniques to infer the needed geometric information using as few function evaluations as possible, they converge linearly at best [199]. In other words, the number of function evaluations needed for convergence of the algorithm scales, at best, as $\mathcal{O}(n)$.

Unlike derivative-free methods, gradient-based ones seem to converge in the number of steps that is independent of the search space dimension n, that is, the number of steps needed scales as $\mathcal{O}(1)$. This property was observed empirically in deep learning, where it was shown that the number of iterations needed for gradient descent to converge scales linearly with respect to the size of training data while it is independent of the size of the model [200]. In other words, gradient descent in deep learning converges in a constant number of steps. In fact, for large-scale models, the number of training epochs rarely exceeds 100 epochs [201] and there are known guidelines in deep learning that training for only a single epoch is sufficient for convergence [202].

This discrepancy in convergence properties between gradient-based ($\mathcal{O}(1)$) and derivative-free ($\mathcal{O}(n)$) optimization algorithms indicates that gradient methods perform search in a space that is effectively "smaller" in dimension. Our interpretation is that gradient methods, since they move in the search space by loosely following the gradients, concentrate their search on a subspace of the search space that is spanned by the previously observed gradients. This subspace is low-dimensional, and its size is dependent on the choice of algorithm parameters[2]. In other words, a gradient-based optimization algorithm ignores the part of the search space in which the objective function does not change[3]. On the other hand, derivative-free optimization requires several function evaluations (typically $\mathcal{O}(n)$ [203]) to acquire information about which directions are more or less promising.

It is important to note that the interpretation presented above only considers the number of function evaluations until convergence and says nothing about the quality of the region of the landscape to which the algorithm converges. Modern metaheuristic algorithms are designed to converge toward the global extreme, and for state-of-the-art ones, we have empirical evidence on a wide variety of low-dimensional functions that they converge toward a global extreme on average if run for a long enough time. However, by analyzing COCO post-processed data archive [204], we can observe a clear deterioration when dimension increases[4]. It is important to also note that this holds for hand-crafted objective functions that do not necessarily reflect the properties of objective functions encountered in real-world problems.

For gradient-based optimization algorithms, there is no extensive evidence for how likely they converge toward the global extreme. However, there are implicit indications that they are likely to converge to a level close to the global extreme for large-scale optimization problems. Firstly, there are theoretical proofs that a characteristic high-dimensional function

[2]For example, gradient descent with momentum has a parameter β, that controls how many formerly observed gradients contribute to the next move, which in this sense can be interpreted as the parameter that controls the effective dimensionality of the search space where the search is being performed.

[3]In directions orthogonal to the gradient, the directional derivative is zero.

[4]Using data gathered by [204], the best algorithms solve 100% of the problems up to $n \approx 20$ and than the success rate decreases to 50% when dimension approaches $n \approx 100$.

has all local minima very close in value to the global minimum[5] [13]–[15]. This means that if an algorithm can efficiently cross saddle points, a gradient-based algorithm will converge on average to a point very close in value to the global minimum. The second indication is the empirical success of gradient-based optimization in deep learning [9]. This is only an indication because there is no evidence that the optimization landscapes encountered in neural network training are rich enough to transfer these findings to a general optimization setting.

The work reported in this chapter was also published as a conference paper [62] in which we made a case for treating gradient-based optimization algorithms as a proper metaheuristic (via the use of automatic differentiation).

6.1 Optimization Methodology in Simulation-Based Shape Optimization

To evaluate the usefulness and applicability of gradient methods in general continuous setting, we focus on simulation-based shape optimization problems. This is a class of problems where, for each function evaluation, a simulation needs to be run and where candidates are shapes in two- or three-dimensional space. Here, we make a distinction between shape and size optimization as in [205], for example. In size optimization, the general shape is already set, and only the sizing of the different parts of the shape is being optimized, such as distances between components, thicknesses, cross-sections, and so on. Shape optimization, on the other hand, modifies the design geometry more broadly to meet the required objective, changing the curvature and position of the boundary.

We chose the class of simulation-based shape optimization problems because they are typically large-scale problems, where gradient-based optimization should be efficient, and because the objective function does not have a closed mathematical form, which means that gradient calculation cannot be performed by simple symbolic differentiation. Our review of the current literature shows that in real-world engineering applications, if the objective function has a closed mathematical form, it is oftentimes differentiated using symbolic differentiation, and a gradient-based optimization algorithm is used [206]–[212]. The symbolic differentiation approach is a well-known method of obtaining the gradient, which makes it widely used, even though it is not necessarily the most efficient [84]. On the other hand, if the objective function does not have a closed form, as is the case with simulation-based optimization, derivative-free optimization methods are used almost exclusively [213]–[219]. Given that the use of gradient methods is largely absent in simulation-based optimization, we focus on those problems.

The prevailing methodology for solving simulation-based shape optimization problems in the current literature is to use an evolutionary computation algorithm coupled with some type of surrogate model and/or multi-fidelity to reduce the cost of function evaluations. For example, in [213], aerodynamic shape optimization of an airfoil in transonic flows is considered and solved by a genetic algorithm and using the kriging model for a surrogate and multi-fidelity. Similarly, the aerodynamic optimization process of the reentry vehicle is considered in [214] with the same general methodology, however, a neural network is used for a surrogate instead of kriging. Aerodynamic shape optimization of an airfoil is also considered in [215] and solved by particle swarm optimization, also with deep neural network surrogate and multi-fidelity. In [216], shape optimization of a vehicle suspension arm is solved by the butterfly optimization algorithm with kriging surrogate and

[5]On average, the gap in the objective value between any local minimum and the global one falls exponentially with the dimension of the search space.

without multi-fidelity. There are also many instances of applying derivative-free methods without considering mechanisms for reducing the function evaluation costs. For example, in [217], several metaheuristic algorithms were compared (sine-cosine algorithm, swarm algorithm, grasshopper algorithm) on a shape optimization problem of a vehicle clutch lever. Nano-optical optimization problems are considered in [218] and solved with particle swarm optimization, differential evolution, Bayesian optimization, and a multistart version of the downhill simplex algorithm. The golden jackal algorithm combined with opposition-based learning was introduced in [219] to solve several engineering shape optimization problems.

There are also some examples in the literature of gradient-based algorithms being used in simulation-based shape optimization. They either apply some specialized and problem-specific method to calculate gradients symbolically, or they acquire gradients in an approximate manner by differentiating a surrogate of the objective function (usually a neural network). In the first case, the optimization problem is usually written in the Lagrangian formalism, which allows for the explicit derivation of formulas for the gradient symbolically as in [206], however, such calculations can involve singular integrals over the shape, which requires special coordinate transformations to be stable [207]. There are also other approaches of this kind, for example, by introducing additional variables [208] or restricting gradient directions to a predefined mesh [209]. In all cases mentioned, the gradient calculation is problem-specific and requires substantial additional calculations. Also, many problems cannot be formulated in the Lagrangian formalism, such as aerodynamic and thermodynamic problems. The second methodology for gradient calculation is similar to the methods found in deep reinforcement learning [210]. Instead of differentiating the result of a simulation with respect to the shape, they differentiate a surrogate model, a neural network model that is trained to approximate the objective function. This method was applied for transonic aircraft wing design [211] and for general problems based on the Reynolds-averaged Navier-Stokes equations [212]. Both use cases are from aerodynamics where problems cannot be posed in the Lagrangian formalism.

Therefore, the current literature lacks a direct application of gradient-based algorithms to general simulation-based shape optimization problems. By direct, we mean calculating the gradient of the simulation result exactly in a general setting without assumptions about the problem. One way to achieve this is by reverse automatic differentiation, which is proven to be the most computationally efficient way to differentiate a program that has many inputs and few outputs [21], which is the case in large-scale optimization.

6.2 Shape Optimization Problem for Eigenvalues of the Laplace Operator

To test our methodology, we chose a well-studied shape optimization problem from acoustics, the minimization of resonance frequencies of a two-dimensional shape. This problem is closely related to many design problems from acoustics, such as development of acoustic structures that are tailored to minimize or maximize certain acoustic qualities. This is useful in architectural acoustics, such as in concert halls, and the design of musical instruments where shape can be optimized to achieve a desired reverberation time or sound clarity. Additionally, the problem is also related to design problems from civil and mechanical engineering of finding a shape that reduces unwanted resonances and, therefore, the likelihood of resonance disaster in machines and buildings.

6.2.1 Theoretical Background

To describe acoustic phenomena in general, we need to solve the wave equation. In our treatment, we consider how sound propagates throughout a domain of shape $\Omega \subset \mathbb{R}^m$ with boundary $\partial\Omega$. In this case, the wave equation has the form

$$\begin{aligned} c^2\nabla^2 u(\mathbf{r},t) = \frac{\partial^2 u(\mathbf{r},t)}{\partial t^2} \qquad &\text{for} \quad \mathbf{r} \in \Omega \qquad \text{and} \quad t \in \mathbb{R} \\ u(\mathbf{r},t) = 0 \qquad &\text{for} \quad \mathbf{r} \in \partial\Omega \qquad \text{and} \quad t \in \mathbb{R} \end{aligned}, \tag{6.1}$$

where $u(\mathbf{r},t)$ is the sound amplitude at point $\mathbf{r}$ and time t, and c is the speed of sound[6] in the material inside Ω. It is known [220], [221] that all solutions of Eq. (6.1) can be expressed as a sum of waves with distinct frequencies, if Ω is bounded:

$$u(\mathbf{r},t) = \sum_{k=1}^{\infty} a_k s_k(\mathbf{r}) \sin(\omega_k t - \varphi_k), \tag{6.2}$$

where ω_k is the resonance frequency of the domain in question and a_k and φ_k its corresponding amplitude and phase, respectively. Functions $s_k(\mathbf{r})$ are eigenfunctions that depend only on space and determine how vibrations of different frequencies look like spatially. Parameters a_k and φ_k can be determined from the initial condition, while ω_k and $s_k(\mathbf{r})$ need more involved treatment.

Using decomposition (6.2), we can fully describe all lossless acoustic phenomena by only considering how the domain Ω resonates on discrete set of frequencies ω_k. The frequencies ω_k that are possible inside the object Ω are determined by

$$\begin{aligned} \nabla^2 s_k(\mathbf{r}) + \frac{\omega_k^2}{c^2} s_k(\mathbf{r}) = 0 \qquad &\text{for} \quad \mathbf{r} \in \Omega \quad \text{and} \quad k = 1,2,3,\dots \\ s_k(\mathbf{r}) = 0 \qquad &\text{for} \quad \mathbf{r} \in \partial\Omega, \end{aligned} \tag{6.3}$$

which is the result of plugging (6.2) into (6.1). Solving this equation brings us a full description of sound propagation inside Ω. Therefore, by changing the shape of Ω we change the values of ω_k which in turn changes the properties of sound propagation inside Ω. This means that to acquire a shape with given acoustic qualities, we need to devise an objective function that depends on ω_k and has an extreme where those qualities are present. For example, consider we are designing a machine that has a part that rotates with frequency $\omega \in [f_1, f_2]$, and we want to reduce unwanted vibrations. In this case, we maximize an objective function that takes the shape of Ω as input and returns the sum of distances of ω_k from the interval $[f_1, f_2]$.

6.2.1.1 Solving the Laplace Eigenvalue Problem with the Finite Element Method

Eq. (6.3) is a linear partial differential equation that can be solved with the finite element method. The first step is to choose a basis of functions $\phi_i(\mathbf{r})$ that respect the boundary condition $\phi_i(\mathbf{r}) = 0$ for $\mathbf{r} \in \partial\Omega$. A common choice for basis functions $\phi_i(\mathbf{r})$ in the finite element method are piecewise linear functions on a triangular mesh that approximately covers domain Ω. This basis is used to approximate eigenmode solutions $s_k(\mathbf{r})$ from (6.3) as a linear combination

$$s_k(\mathbf{r}) \approx \sum_{i=1}^{n} v_{ki} \phi_i(\mathbf{r}). \tag{6.4}$$

[6] For general isotropic elastic medium, there can be two different speeds of sound, one for transverse and the other for longitudinal waves, which means that two wave equations need to be solved, one for each speed of sound. In fluids and gases, on the other hand, only a single speed of sound is possible.

When we plug in this approximation to Eq. (6.3) we transformed problem (6.3) to a generalized eigenvalue problem from linear algebra:

$$A\mathbf{v}_k = \lambda_k B\mathbf{v}_k, \tag{6.5}$$

where $\lambda_k = \omega_k^2/c^2$ and matrices A and B and vector $\mathbf{v}_k$ are

$$\begin{aligned} [\mathbf{v}_k]_i &= v_{ki} \\ [A]_{ij} &= \int_\Omega \nabla\phi_i(\mathbf{r}) \cdot \nabla\phi_j(\mathbf{r})\mathrm{d}\mathbf{r} \\ [B]_{ij} &= \int_\Omega \phi_i(\mathbf{r})\phi_j(\mathbf{r})\mathrm{d}\mathbf{r} \end{aligned} \quad . \tag{6.6}$$

The most efficient way to solve problem (6.5) is by transforming it to the standard eigenvalue problem using the Cholesky decomposition and two triangular linear system solves [222] (this process is detailed in Algorithm 6.1).

Therefore, given a shape Ω, we can find ω_k and $s_k(\mathbf{r})$, and from this all properties of sound propagating through Ω, by the following procedure:

1. Create a triangular mesh on domain Ω and fix a basis of functions $\phi_i(\mathrm{r})$ that are linear on the simplices of the mesh.
2. With $\phi_i(\mathrm{r})$ fixed, analytically solve integrals in (6.6) to acquire matrices A and B.
3. Obtain λ_k and $\mathbf{v}_k$ by solving (6.5) which determine $\omega_k = c\sqrt{\lambda_k}$ and $s_k(\mathbf{r})$ by (6.4).

6.2.2 Optimization Problem

Many design problems in acoustics can be reduced to finding the extreme of a simple function of values of ω_k with respect to the shape of Ω. An important special case of this problem, which is used as a benchmark and has been well-studied [223]–[229], is finding a two-dimensional shape Ω^* that has the smallest possible value of ω_k for some fixed k. More formally,

$$\Omega^* = \arg\min_{\Omega\subset\mathbb{R}^2}\{\lambda_k(\Omega) \ : \quad |\Omega| = 1\}, \tag{6.7}$$

where $\lambda_k(\Omega)$ is the k-th eigenvalue of the Laplace operator $(-\nabla^2)$ on domain Ω with area $|\Omega|$. In acoustics, $\lambda_k(\Omega)$ is related to the frequency of the k-th resonant mode (overtone), more precisely $\lambda_k = \omega_k^2/c^2$. The area constraint of $|\Omega| = 1$ can be eliminated by observing that $\lambda_k(c\,\Omega) = \lambda_k(\Omega)/c^2$ for any scalar $c > 0$. From this it follows [224] that problem (6.7) is equivalent to

$$\Omega^* = \arg\min_{\Omega\subset\mathbb{R}^2}\{|\Omega|\lambda_k(\Omega)\}. \tag{6.8}$$

This is the standard form of this optimization problem and is solved in this chapter to investigate the usefulness of our methodology in a simulation-based shape optimization setting. As a side note, there are further constraints that are implicitly imposed on Ω, such as being two-dimensional and simply connected. The classical form of this problem, which we will use, is that Ω is topologically equivalent to a disk. However, there are other variations that are more restricting, such as convex Ω [229], or less restricting, such as using topology optimization to study disconnected Ω [225].

To illustrate the difficulty of this optimization problem, consider that for each function evaluation of a candidate Ω one needs to triangulate Ω and create a triangular mesh with r inner vertices[7], then, from distances between mesh vertices and areas of the mesh simplices,

[7] This step can be avoided in some cases by deforming previously seen meshes.

matrices A and B, both of size $r \times r$, are calculated. Finally, the Cholesky decomposition, two triangular linear solves, and eigenvalue decomposition must be performed on matrices of size $r \times r$ to acquire the objective function value ω_k. Therefore, this is a large-scale optimization problem with an expensive objective function that does not allow simple symbolic differentiation to obtain the gradient. This is because the routines involved in the underlying simulation are complicated, and their computational graph is not static since, for example, linear algebra routines branch in accordance with the structure of the matrix that is supplied.

6.3 Methodology for General Shape Optimization with Gradient-Based Algorithm

In this section, we present a general method for large-scale shape optimization using gradient-based algorithms. The two basic ingredients are the differentiable encoding for the problem and the gradient calculation scheme using reverse automatic differentiation. These ingredients give us the means to calculate the gradient of the objective function with respect to shape Ω and allow us to use general-purpose gradient-based optimization algorithms to search for a global extreme. This method should be most competitive for problems where gradient cannot be obtained by symbolic differentiation, such as in simulation-based optimization, where derivative-free algorithms are practically the only way to go and which have drawbacks in large-scale optimization setting. For other cases, there usually already exist specialized libraries that can perform gradient-based optimization efficiently, however, this is true only for well-studied problems, and customizing these methods to related problems can be troublesome and labor intensive. On the other hand, the reverse automatic differentiation procedure is applicable to all problems where the code for the objective function calculation is available. Even if symbolic differentiation is possible to perform, the use of automatic differentiation can reduce gradient calculation complexity. Furthermore, automatic differentiation is easier to implement since it requires no direct coding of gradient calculation and requires no considerations about stability and the optimal order of operations. For automatic differentiation to work, we only need an implementation of the objective function calculation, and the gradient calculation code is constructed automatically from it. Therefore, this methodology is applicable to all optimization problems with continuous encoding.

The first ingredient is the shape encoding which should be continuous and easily differentiable to reduce complexity. There are many ways to continuously encode shape Ω. All shape models used in the literature encode the boundary $\partial\Omega$ of shape Ω since this completely determines the shape and requires fewer variables. One popular shape encoding is an isogeometric model [230] that uses B-splines to describe the shape of the boundary. Its main advantage is that this description is directly compatible with the computer-aided design environments used in industry. However, this model requires a preset number of points for the nodes on the boundary and requires remeshing of the shape after each objective function evaluation [227]. We, therefore, opted for a more modern model to encode the shape, a deep neural network. Even small networks have quite large capacity and can model very complicated shapes, which is why they are used in artificial intelligence[8]. Also, using our model, it is possible to construct the mesh once and deform it for each candidate shape, thus avoiding remeshing.

[8]For example, in binary classification problems, a neural network needs to find a boundary that separates the two classes in the feature space. This boundary is, for hard artificial intelligence problems, very complicated. Therefore, a neural network is known to be very capable of capturing possibly very complicated shapes.

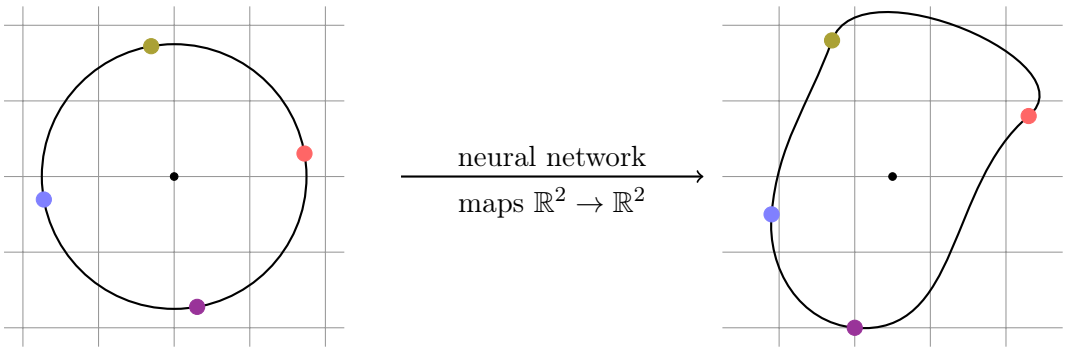


Figure 6.1: Illustration of how our shape model with an invertible neural network works on a two-dimensional example. The neural network maps vectors from the circle to general two-dimensional vectors. Examples of such points are depicted as color dots. Since a neural network is a continuous function, its codomain is a closed curve, so it maps to a legitimate boundary of a two-dimensional shape Ω.

The shape model developed in this work uses an invertible neural network that maps points from a preset domain to points on the boundary $\partial\Omega$ of shape Ω. See Figure 6.1 for an illustration of a two-dimensional example. We use invertible neural networks in order to prevent intersections of the boundary $\partial\Omega$ with itself and thus guarantee that the resulting shape Ω is more aligned with the engineering applications of shape optimization. Invertible neural networks, such as the ones based on coupling flows, have also recently been shown to be universal approximators [231]. This means that the restriction imposed on neural network structure for it to be invertible does not impose any restrictions on its representation power. Because a neural network is also a continuous function, its domain should be topologically equivalent to the shape boundary $\partial\Omega$ we are modeling. This means that this shape model has a preset topology and cannot produce shapes with a topology different from the domain of the neural network. Therefore, this encoding is appropriate for shape optimization, not topology optimization. This is aligned with the optimization problem described in Section 6.2.2 where only shapes Ω that are topologically equivalent to the disk are considered. In this case, the domain of the neural network should be a circle, which is shown in Figure 6.1.

More formally, the boundary of a candidate shape $\partial\Omega$ is determined by the parameters ϑ of an invertible neural network g and by a predefined domain $\partial\Delta$ on which g acts on.

$$\begin{aligned} x &\in \partial\Delta \quad \subset \mathbb{R}^n \\ y &\in \partial\Omega \quad \subset \mathbb{R}^n \end{aligned} \qquad x \overset{\text{neural network } g_\vartheta}{\longmapsto} y \tag{6.9}$$

Due to the properties of g, $\partial\Omega$ is topologically equivalent to $\partial\Delta$. In many cases of shape optimization, due to the nature of objective function calculation, the candidate shape boundary $\partial\Omega$ needs to be supplied in the form of a mesh, that is, a set of vertices y_i together with information on how those vertices form edges, faces, cells, and so on. To obtain such a mesh on $\partial\Omega$, we can apply one of the mesh generation algorithms on a sample of points $y_i \in \partial\Omega$, but we can also circumvent this step. If we instead precompute a mesh on a sample of points $x_i \in \partial\Delta$, we can use g_ϑ to transform this mesh on $\partial\Delta$ to a one on

$\partial\Omega$ in a following way:

$$\begin{array}{c}\text{Mesh on } \partial\Delta \\ \text{vertices} \quad x_i \in \partial\Delta \\ \text{for } i = 1, 2, \ldots, m \text{ with} \\ \text{edges } E, \text{faces } F, \text{cells } C, \ldots\end{array} \quad \Longrightarrow \quad \begin{array}{c}\text{Mesh on } \partial\Omega \\ \text{vertices} \quad y_i = g_\vartheta(x_i) \in \partial\Omega \\ \text{for } i = 1, 2, \ldots, m \text{ with} \\ \text{edges } E, \text{faces } F, \text{cells } C, \ldots\end{array} \tag{6.10}$$

In this way, to get a mesh on $\partial\Omega$, we only need to apply our neural network model g_ϑ on m points while keeping $E, F, C, \ldots$ unchanged. Such a mesh on $\partial\Omega$ can then be used for the objective function calculation. Because g_ϑ is continuous and invertible, this mesh transformation will not cause any illegal deformations of the mesh elements. The method described so far works most naturally for objective functions that directly require shape boundary, such as simulations employing analytic or boundary element method, for example. But for other types of simulations, such as the finite element method, it is convenient to directly generate a mesh on the full shape Ω, not only on the boundary $\partial\Omega$. It is true that we can always generate a mesh on Ω if given $\partial\Omega$, but this brings additional computational burden. Luckily, we can easily transfer our shape model from capturing only the boundary $\partial\Omega$ to one that captures the entire shape Ω.

$$\begin{array}{ll} x \in \Delta & \subset \mathbb{R}^n \\ y \in \Omega & \subset \mathbb{R}^n \end{array} \qquad x \xmapsto{\text{neural network } g_\vartheta} y \tag{6.11}$$

In this formulation, all arguments about mesh generation also apply. If we precompute a mesh on Δ, we can transfer it to a legitimate mesh on Ω using continuous and invertible g_ϑ.

$$\begin{array}{c}\text{Mesh on } \Delta \\ \text{vertices} \quad x_i \in \Delta \\ \text{for } i = 1, 2, \ldots, m \text{ with} \\ \text{edges } E, \text{faces } F, \text{cells } C, \ldots\end{array} \quad \Longrightarrow \quad \begin{array}{c}\text{Mesh on } \Omega \\ \text{vertices} \quad y_i = g_\vartheta(x_i) \in \Omega \\ \text{for } i = 1, 2, \ldots, m \text{ with} \\ \text{edges } E, \text{faces } F, \text{cells } C, \ldots\end{array} \tag{6.12}$$

An important property of our shape encoding is that ϑ, which is changed during optimization, encodes the shape directly in the function space, and not using coordinates on the boundary as many other methods do. This means that the number of points on the boundary is completely independent of the size of the search space (the number of parameters in our neural network). The first has to do with how accurate the objective function calculation is and the second with how rich the shape search space is. This means that we can freely change the resolution of our candidate shape even during optimization without altering the search space dimension since this does not affect the neural network model at all. The only thing we need to do to change the resolution is to generate a different mesh with different number of vertices. Furthermore, we can freely choose to which class of functions the encoded shapes belong to by selecting the appropriate activation function of the neural network. For example, if we desire the shape to be smooth, we choose a smooth activation, such as `softplus`, or if we desire the shape to be piecewise linear, we choose a piecewise linear activation, such as `ReLU`.

The second ingredient needed for gradient-based optimization, apart from continuous encoding, is a way to calculate the gradient of the objective function. For this, we will use reverse automatic differentiation. For a clearer depiction of how this is done (using the TensorFlow library) we provide full code that implements the automatic differentiation part in Algorithm 6.1 for objective function (Eq. (6.8)) and an approximative procedure to evaluate the objective function (described in Section 6.2.1.1). By using automatic dif-

Algorithm 6.1: Code implementing gradient-based shape optimization. The only part of the code omitted here (imports in lines 2–3) are methods related to FEM (geometrical operations described in Section 6.2.1.1 and round mesh generation) and neural network construction. Here, we highlight how user-friendly gradient methods can be implementation-wise. For example, gradient calculation is achieved in a single line of code (line 33). To perform automatic differentiation, the TensorFlow library requires the user to only specify how the objective is calculated from the encoding (lines 19–29) and the computational graph for the gradient calculation is built automatically without any explicit code from the user. As seen in lines 24–27, the calculation of the objective involves complicated linear algebra operations which make the explicit derivation of formulas for the gradient calculation extremely difficult. However, if all intermediate steps during objective calculation are recorded (on `tape` in line 18), the gradient can be automatically calculated in the same computational complexity as the objective. Then, a gradient can be used to update the neural network weights iteratively using the Adam algorithm (line 35).

```
1  import tensorflow as tf
2  from fem import get_matrices, get_round_mesh
3  from shape_models import construct_neural_network_model
4
5  # Choose Adam for our gradient-based optimization method.
6  optimizer = tf.optimizers.Adam()
7
8  # Precompute a round triangular mesh that will be deformed by our
9  # neural network shape model in order to search for optimal shape.
10 round_mesh = get_round_mesh()
11
12 # Build an invertible neural network shape model as in Figure 6.1.
13 model = construct_neural_network_model()
14
15 # Perform num_steps iterations of gradient descent using Adam algorithm.
16 for step in range(num_steps):
17     # Record all intermediate calculations inside the context to "tape".
18     with tf.GradientTape() as tape:
19         # Deform a round triangular mesh using neural network model.
20         mesh = model(round_mesh)
21         # Based on the mesh, find matrices A, B in accordance with FEM.
22         A, B = get_matrices(mesh)
23         # Solve generalized eigenvalue problem A.v_i = vals_i B.v_i.
24         C = tf.linalg.cholesky(B)
25         D = tf.linalg.triangular_solve(C, A)
26         E = tf.linalg.triangular_solve(C, tf.transpose(D))
27         vals, V = tf.linalg.eigh(E)
28         # Objective function value is the k-th eigenvalue.
29         objective = vals[k]
30     # Calculate gradient using reverse automatic differentiation by
31     # traversing computational graph saved on "tape" in reverse
32     # and calculating the gradient along the way.
33     gradient = tape.gradient(objective, model.trainable_variables)
34     # Perform gradient descent step using Adam.
35     optimizer.apply_gradients(zip(gradient, model.trainable_variables))
```

ferentiation, we only need to supply a computational graph for how the objective function value is calculated, and the computational graph for calculating the gradient is constructed automatically. This is especially important for simulation-based optimization, where the derivative of a simulation result with respect to the simulation settings is, in general, infeasible to derive and code. In Algorithm 6.1 we can see the full workflow of our gradient-based optimization approach. In line 20, we (in accordance with Eq. (6.12)) deform a round mesh using a neural network shape model into a new mesh that represents a candidate shape for the optimization (as in Figure 6.1). In line 22, we find matrices A, B using Eq. (6.6) and in lines 24–29, find the k-th eigenvalue, which is the objective we are minimizing. Computations in lines 19–29 are then automatically differentiated to acquire the gradient of the k-th eigenvalue with respect to the weights of the neural network shape model. This gradient is then used to update the neural network weights using the Adam algorithm in line 35.

6.4 Experiments

To demonstrate the applicability of our methodology for shape optimization using gradient-based algorithms, we will utilize it for the problem of finding a two-dimensional shape with the smallest Laplace eigenvalue λ_k for some fixed k (as formalized in Section 6.2.2). A large part of our implementation is already explained in Algorithm 6.1 and in the text explaining it. Here, we detail other implementation considerations related to the shape model and mesh generation for the finite element method. We implemented a two-dimensional neural network shape model (as shown in Figure 6.1). To assure that the neural network model g_ϑ is an invertible function, we used normalizing flow architecture [232]. For the network activation, we used `ReLU` function, which makes our shape model piecewise linear. This is aligned with the fact that the mesh used for objective function calculation is also piecewise linear as a shape. Manual architecture tuning showed that a network with three layers and eight units per layer is sufficient for this problem, which is the configuration we used. This makes our search space 186-dimensional (the number of neural network parameters in ϑ). The initial value for ϑ was generated randomly using Glorot initialization [189]. To generate a mesh for a candidate shape Ω, g_ϑ is applied on a pre-computed mesh with the shape of a disk Δ (as shown in Figure 6.1). This mesh was generated using Delaunay triangulation and has 7225 vertices x_i. For each objective function evaluation (6.8), the shape model g_ϑ is applied on vertices x_i and returns vertices y_i that represent candidate shape Ω. Vertices y_i, alongside with the precomputed triangulation for the mesh, are used to calculate matrices A, B using Eq. (6.6). The implementation of this calculation in TensorFlow was slightly more labor intensive since it involves geometric calculations relating to triangles, and there is a lack of prior TensorFlow code on this topic. All remaining steps to calculate the objective function and the gradient are already well implemented and are explicitly shown in Algorithm 6.1.

For the gradient-based optimization algorithm, we chose Adam [73] as implemented in `tensorflow.keras`. But for comparison, we also used the Limited-Memory Covariance Matrix Adaptation (LM-CMA) algorithm as implemented in PyPop7 [233]. LM-CMA is a special version of the CMA-ES algorithm, which, unlike the standard version of CMA-ES, is known to work well in large-scale optimization setting [234]. For both Adam and LM-CMA, we used the default configuration except for the learning rate for Adam and mutation strength for LM-CMA. For those two hyperparameters we performed tuning with random search. For Adam, the learning rate was chosen randomly between 0.001 and 0.1, and for LM-CMA, the mutation strength between 0.1 and 2 times the default value. We performed 30 runs on 6 different problems ($k = 2, 3, 4, 5, 6, 7$). For problem with $k = 3$ the solution is

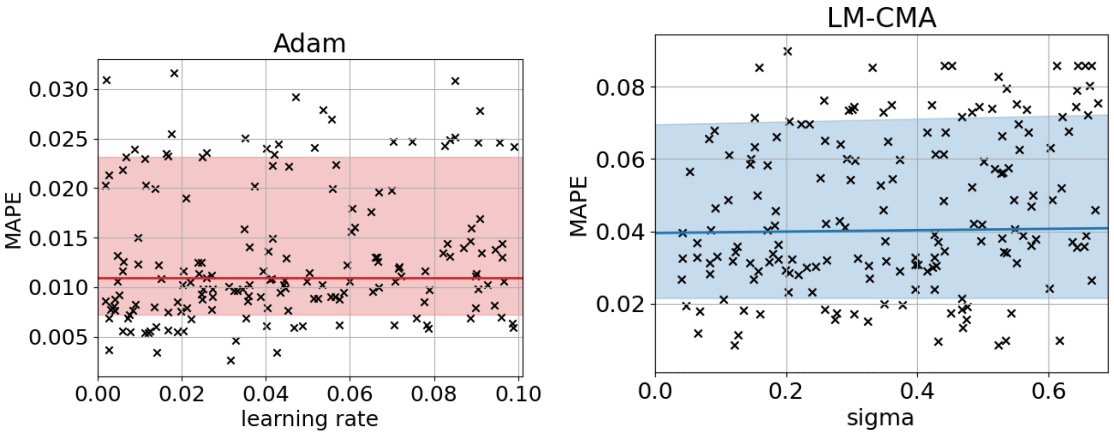


Figure 6.2: Sensitivity of the algorithms to the value of the hyperparameters learning rate for Adam and mutation strength sigma for LM-CMA. Mean absolute percentage error (MAPE) was calculated with respect to the baseline of the best-known objective values found in [225]. The full line is the median, and the shaded area includes the central 66% runs. These values were obtained via linear quantile regression.

known to be a disk [225] which is very close to the initial candidates. For this reason, we used a square mesh Δ for $k = 3$ to get more reasonable results in comparison to problems with other k. For both algorithms, we limited the run time of a single optimization run to one hour. This was in order to guarantee a fair comparison due to gradient calculation which is needed for Adam but not for LM-CMA. The calculation of the objective function (and gradient) was, in both cases, performed by TensorFlow, compiled to `CUDA` code, and run on a Tesla K80 GPU.

It is important to note that our implementation of LM-CMA does not include features like surrogate modeling or multi-fidelity, which are common in large-scale simulation-based optimization [213]–[216] and are known to reduce the optimization time. Both Adam and LM-CMA could be implemented to include those features, however, this would require extensive fine-tuning of several components in order to fairly compare gradient-based and derivative-free methods. To reduce both the computational and programming burden, we opted to compare the most basic implementations of each algorithm with minimal hyperparameter tuning (one parameter believed to be the most important for each algorithm).

For both algorithms, the hyperparameters we tuned had no noticeable effect on performance. Figure 6.2 shows the mean absolute percentage error (where the best-known solutions from [225] were taken as the baseline) for different hyperparameter values. We used linear quantile regression to uncover possible trends in performance with respect to hyperparameter values, but we found that the performance is, in fact, constant, as depicted in Figure 6.2. Figure 6.3 shows the convergence of both algorithms for the six problems and Table 6.1 the best solutions found by the Adam algorithm compared to the best solutions found in [225]. We can observe that Adam has a more favorable convergence and attains lower overall objective function values. However, comparing the results of Adam to the best-known solutions from the literature shows that there is still some room for improvement. Shapes themselves seem quite similar to the ones found in [225], however, the objective function values are higher by around 0.5%. We suspect that more involved hyperparameter fine-tuning, of both the shape model architecture and optimization procedure, would further improve our results. Also, as mentioned above, surrogate modeling and multi-fidelity would probably be of benefit as well.

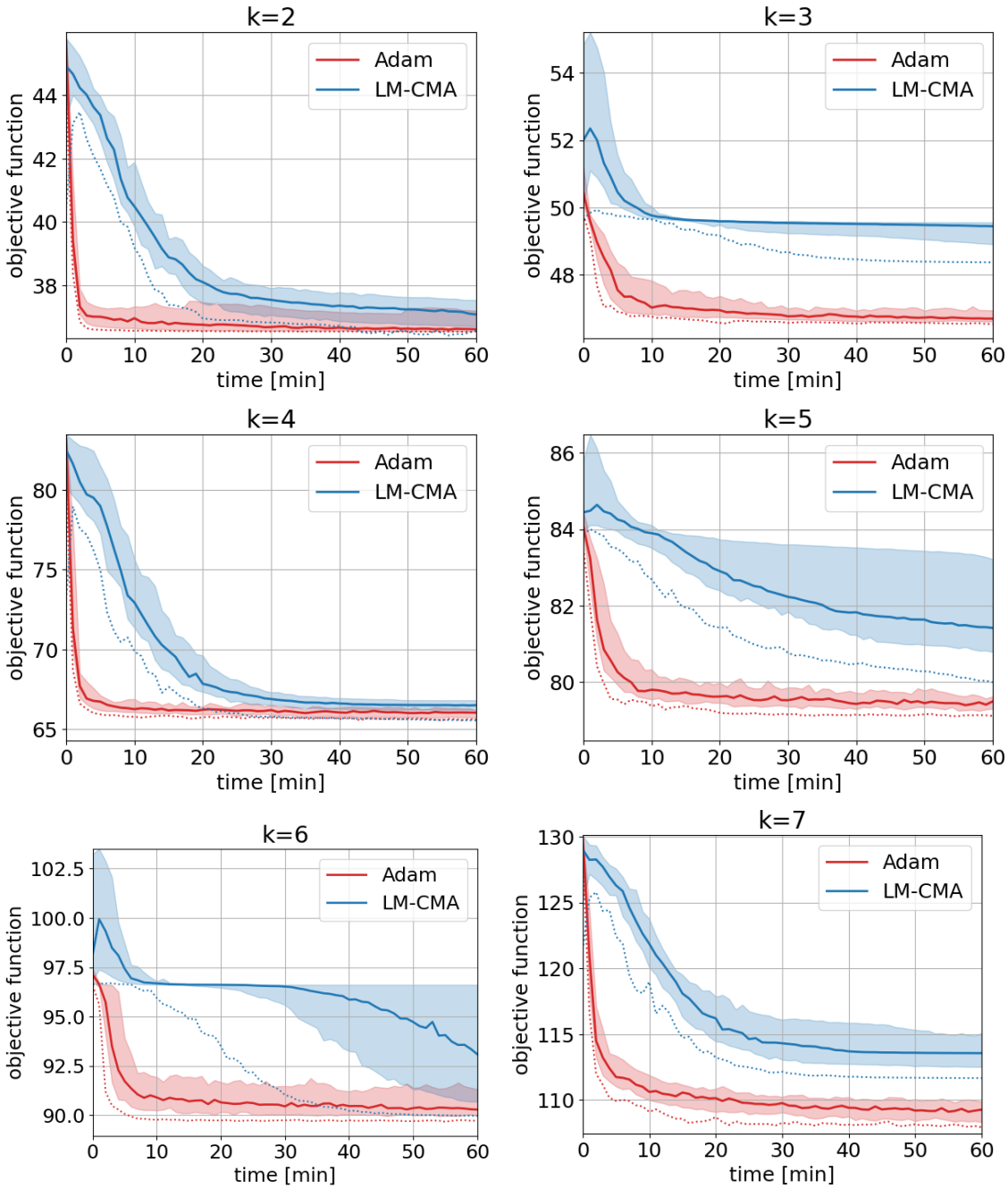
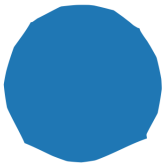


Figure 6.3: Optimization progress for Adam and LM-CMA when run on six different problems ($k = 2, 3, 4, 5, 6, 7$). The full line is the median over 30 runs of the algorithm, while the shaded area shows the central 66% of the runs. The dotted line is the best objective function value found.

Table 6.1: Comparison of the best solutions found by Adam and in [225] for six different problems ($k = 2, 3, 4, 5, 6, 7$). Our solution for problem $k = 3$ is visually different because it was generated from a different initial mesh (a square instead of a disk) in order to obtain optimization difficulty similar to other k.

	Our result with Adam		[225]	
k	objective value	solution	objective value	solution
2	36.389		36.337	
3	46.373		46.126	
4	64.790		64.306	
5	78.569		78.166	
6	89.218		88.502	
7	106.983		106.211	

6.5 Discussion

In this chapter, we demonstrated that gradient-based optimization algorithms can be used to solve simulation-based continuous optimization problems. Our methodology did not employ any problem-specific or approximative techniques to calculate the gradient but rather automatic differentiation, which is problem-agnostic and exact. This means that our methodology puts gradient-based optimization on the same level as general-purpose metaheuristics (such as evolutionary algorithms). Simulation-based optimization is oftentimes presented as an example of an optimization problem for which general-purpose derivative-free metaheuristic algorithms are the only option for solving them [19]. This is not the case, though. Given that many metaheuristics struggle in large-scale optimization setting, gradient-based algorithms can, in fact, be more suitable. In our use case of finding two-dimensional shape with minimal Laplace eigenvalues this is exactly what we observed. Gradient-based algorithm Adam converged faster and obtained better overall solutions compared to commonly used derivative-free LM-CMA. Our explanation for this is that derivative-free metaheuristics require more objective function evaluations to acquire the same amount of information about the optimization landscape as gradient-based algorithms get with a single gradient calculation[9]. Since the computational complexity of gradient calculation is twice the complexity of objective function evaluation these additional evaluations result in slower convergence for derivative-free methods compared to gradient-based ones.

In this regard, we can confirm hypothesis **H4** that states that automatic differentiation can be applied to general objective functions, even the ones including simulations, and that, when coupled with a gradient-based algorithm, it can be used as a general methodology for solving large-scale continuous optimization problems. We have shown this for a use case where the objective function included several matrix factorizations and inversions in a row as a part of a simulation using the finite element method on a triangular mesh. Without automatic differentiation, the calculation of the gradient of such an objective function would be intractable. In our use case, the gradient-based method was even superior to minimally tuned metaheuristic LM-CMA specialized for large-scale optimization. The implementation of gradient calculation still requires some effort, however, our use case shows that this effort is also accompanied by faster convergence. As a part of this use case, we also developed a general shape model that employs invertible neural networks. This shape model can be used for various shape optimization problems but it is especially suitable for ones that work with meshes such as finite or boundary value methods. This methodology is computationally efficient because a neural network only deforms an existing mesh without the need to generate a new mesh for each shape during optimization.

[9]For illustration, the characteristic number of objective function evaluations needed to obtain a reasonable approximation of the gradient with derivative-free methods scales linearly with the dimension of the search space [235].

Chapter 7

Discrete Optimization

Discrete optimization is a process of finding the maximum or minimum of a function defined over a domain that includes discrete variables, such as graphs, permutations, strings, integers, and so on. In real-world use cases, the discrete part of the search space usually consists of a finite but exceptionally large number of feasible candidates, making exhaustive search impractical and the use of approximative algorithms extremely common [236]. In this chapter, we will be interested in general-purpose algorithms, also termed metaheuristic algorithms, and will not touch problem-specific ones that were designed for a given special class of discrete optimization problems. This is because the aim of this chapter is to explore the possibilities of using gradient-based optimization as a general-purpose the optimizer in the discrete setting. General-purpose algorithms should work reasonably well on a wide range of different discrete optimization problems. But designing such algorithms requires different techniques compared to the continuous case because discrete search spaces lack a general geometric structure. In contrast to continuous optimization, there are no general concepts like distance, angle, or projection in discrete search spaces. This is why discrete optimization is a special category in optimization and considerations put into algorithm design are quite different from what we are accustomed to from continuous optimization.

Consider a discrete optimization problem of finding a candidate x^* for which function ϕ achieves a global minimum with respect to all other $x \in \mathcal{S}$ on which ϕ is defined on:

$$x^* = \arg\min_{x \in \mathcal{S}} \{\phi(x)\}. \tag{7.1}$$

One essential ingredient that all algorithms require is an encoding for candidates $x \in \mathcal{S}$. An encoding is (not necessarily unique [237]) representation of candidates that include all relevant information about the candidates that are needed for the objective function ϕ evaluation. For example, if $\mathcal{S}$ is a set of graphs with n nodes, a candidate graph x can be encoded using its adjacency matrix, which determines which nodes of the graph x are connected with edges. Using this encoding (and discarding symmetric and diagonal part of the matrix), we can encode a candidate x using a bit string of length $n(n-1)/2$. An encoding allows us to enumerate feasible candidates $x \in \mathcal{S}$ and to use most general optimization algorithms such as exhaustive search. To use more advanced general-purpose optimizers, however, we require an additional structure on the search space called the neighborhood.

Most state-of-the-art metaheuristic algorithms assume that it is possible to generate candidates that are in some sense similar/close to the given candidate $x \in \mathcal{S}$. For example, if we flip a single bit in the encoding of candidate x, we may get a candidate that encodes a solution that is conceptually quite similar to the candidate x. This notion is formalized by a neighborhood structure, which is a function that returns, for each candidate x, a set of its neighbors. Neighborhood structure allows us to use local search algorithms that, to some

degree, follow the principle behind gradient-based algorithms. This class of algorithms iteratively improves the current candidate by searching through its neighbors, and in some sense, the algorithm follows a trajectory of steepest descent. There are many variations of this algorithm that provide mechanisms for more exploration, such as Simulated annealing [238], Tabu search [239], Guided local search [240], and so on. It is important to note that the choice of neighborhood structure can substantially influence the performance of algorithms [241]. For example, a neighborhood of x might consist of candidates that are "close" to x with respect to the encoding, but their objective function value may not be close to $\phi(x)$ defeating the purpose behind local search assumptions. This is why neighborhood structure is usually aligned with our knowledge about the objective function, for example, neighborhood structure for maximum clique problems would be different from satisfiability problems even if the encoding is a binary string in both cases [242]. Nonetheless, metaheuristic algorithms work reasonably well even with a generic neighborhood structure, such that is inherited by encoding, without extensive fine tuning [243].

Another class of metaheuristic algorithms are population-based ones, such as genetic algorithms [244] and ant colony optimization [245] and their variants. This class of algorithms maintains a population of candidates rather than a single solution and provide a natural way for the exploration of the search space. In the case of genetic algorithms, two operators (recombination and mutation) are used to generate new candidates in a population, while the selection mechanism maintains the expected population size and guides the population toward more promising candidates. The selection mechanism is problem-independent, while recombination and mutation are usually chosen based on the encoding and the properties of the objective function. The mutation operator is usually tied to a neighborhood structure while the recombination to the encoding. There are several generic recombination operators suited for binary, integer, permutation, and string encoding, with many more specialized for specific problem classes [246]. There is an extensive amount of literature available on population-based algorithms with guidelines that allow us to apply the general principles behind these algorithms to almost any discrete optimization setting, which makes them general-purpose metaheuristics [241].

The goal of this chapter is to develop a methodology that would allow gradient-based algorithms to be used in discrete optimization. Since this thesis aims toward presenting gradient-based algorithms at the same level of generality as metaheuristic ones, our requirement is that the methodology works for any discrete optimization problem. In this regard, we will try to distance our methodology from problem-specific manual user interventions such as choosing a neighborhood structure or other problem-specific operators. In Section 4.3, we already developed a three-stage algorithm for a mixed discrete-continuous optimization problem. However, this methodology was developed specifically for matrix factorization problems and is not general enough for general discrete optimization. In that special case, we had an objective function that had a natural extension from a mixed discrete-continuous domain (hard clustering) onto a continuous one (soft clustering). This is not the case in general discrete optimization. For example, we can extend the discrete space of graphs (encoded as bit strings) to a continuous space of weighted graphs (encoded as real vectors). But, even if our encoding is now continuous, our objective function may not have any meaning on weighted graphs, which makes such an extension useless. For some other discrete objects, like permutations, even extending the encoding to continuous space can be troublesome. This is why, in this chapter, we tackle discrete optimization from a different angle. We will not encode a discrete candidate in a continuous space but rather a distribution over discrete candidates and use gradient descent to move this distribution toward promising candidates.

7.1 Methodology for General-Purpose Discrete Optimization with a Gradient-Based Algorithm

Modern gradient-based optimization can be treated as a global optimization technique only in a large-scale continuous setting [13]–[15]. Otherwise, it is prone to getting stuck in low-quality local minima. Since gradient-based optimization moves from one candidate to another in steps, we might be tempted to translate this process to a discrete setting by using a scheme where the algorithm iteratively moves from the current candidate to its fittest neighbor. Such an algorithm would belong to the class of local search algorithms mentioned above. However, algorithms of this type do not retain the beneficial properties of gradient descent. In a discrete setting, we cannot efficiently calculate in which direction to move to find the fittest neighboring candidate, but rather, we need to evaluate many candidates in the neighborhood to find the fittest one. This is a substantial drawback because, in the continuous case, the gradient can be calculated with the same time complexity as a single objective function evaluation while in this case, it would be orders of magnitude more expensive[1]. Additionally, for a discrete local search algorithm to resemble a gradient-based one, we require a candidate encoding with a high-quality neighborhood structure so that objective function ϕ is "continuous" when moving between neighbors. Finding neighborhood structures of this type requires extensive experiments on a specific class of optimization problems, which draws us away from our aim of finding a general-purpose methodology.

However, specialized state-of-the-art algorithms are known to be extremely successful in the setting they were designed for [247]–[249]. In this way, they are able to leverage the known properties of the search space $\mathcal{S}$ and the objective function ϕ and use it to find good solutions in a reasonable time. In our view, they utilize commonly encountered relations and correlations between encoding and objective value in their design to make the search efficient. Our aim is to also use such information in our algorithm, however, not by hard-coding it in the algorithm design but rather by learning this knowledge during optimization through the encountered candidates and corresponding objective values. During the run of any optimization algorithm, a lot of data is generated that can be used to gain knowledge about the structure of $(\mathcal{S}, \phi)$ and what kind of relations and correlations are present in that data. Machine learning has a long history of finding knowledge in complex data with discrete variables. Therefore, it is a sound assumption that machine learning is able to find some structure and patterns when presented with $(x_i, \phi(x_i))$ data. The advantage of this design is that the gained knowledge is specific to the problem being solved, not generic, and to some degree associated with an "average" problem with this specific $\mathcal{S}$. In this regard, our idea is to use a machine learning model to generate candidates x and train the model in a way that guides it to generate fitter candidates. In this way, the model has information about the encoding (since it generates candidates) and the concept of fitness of those candidates (since it is penalized for generating poor candidates). Our intention is that the underlying patterns in this information the model is exposed to contributes to the model finding a problem-specific strategy for solving that problem even though such an algorithm is inherently general-purpose.

The core idea of our methodology is to use a model g_ϑ, parameterized by a real vector ϑ, to represent a probability distribution over discrete candidate solutions. Candidates are then sampled from this distribution and evaluated using the objective function ϕ. We then

[1] Gradient in continuous optimization can be calculated for the price of two objective function evaluations (one for the objective function value and one for the gradient). On the other hand, vanilla local search in discrete optimization requires the evaluation of some proportion candidates in the neighborhood of the current candidate in each iteration. The number of evaluations per iteration is, therefore, approximately equal to the size of the encoding.

search for a distribution g_ϑ (continuous optimization) so that the expected objective value of the drawn candidates $x \sim g_\vartheta$ is minimal. In this respect, we transformed a discrete optimization problem into a continuous one:

$$\min_{x \in \mathcal{S}} \{\phi(x)\} \qquad \Longrightarrow \qquad \min_{\vartheta} \mathbb{E}_{x \sim g_\vartheta}[\phi(x)]. \tag{7.2}$$

The ideal solution to the continuous version of the problem is a delta distribution with a peak at the position of the candidate that is the solution to the discrete optimization problem. This method has similarities to the estimation of distribution algorithms [250] where discrete candidates are decomposed to components $x = [x_1, x_2, \ldots, x_m]$ and usually a graphical probabilistic model is used to draw values for components x_i and generate new candidates. Graphical models, such as probabilistic trees, Bayesian networks, or Markov models, are used to model probabilities over components x_i and, to some degree, model dependencies between them. Estimation of distribution algorithms also have some similarities to genetic algorithms since they employ a selection mechanism to update the probabilistic model. In each generation, more promising candidates from the population are used to rebuild the probabilistic model, therefore, the probability distribution over candidates shifts toward candidates with lower objective values.

In our approach, we will use a probabilistic model g_ϑ that has a fully continuous encoding (unlike graphical models) so that gradient-based algorithms can be used for minimization instead of a selection mechanism. Additionally, we would like a probabilistic model g_ϑ with computationally efficient sampling of new candidates $x \sim g_\vartheta$ (unlike Markov models) and that g_ϑ is able to account for arbitrary dependencies between x_i (unlike graphical models). Current research on modeling complicated probability distributions in AI is presently centered around generative adversarial networks and flow-based generative models [251]. Since generative adversarial networks are known to be more difficult to train [252], we opted for flow-based models. These probabilistic models have been used to model exceptionally complicated distributions and were able to capture data distributions for audio [253], images [254], video [255], molecular graphs [256], and so on. A flow-based model is essentially an invertible neural network (already mentioned in Section 6.3) that is specially tailored so that the Jacobian determinant can be calculated efficiently (which we will see is an important advantage).

In probabilistic modeling, there are two approaches for modeling a distribution with a machine learning model g_ϑ. The first approach are density estimation models where a model g_ϑ returns a probability (density) when presented with a candidate x (graphical models belong to this class). The other class are generative models where g_ϑ does not return a probability (density) but rather a sample x drawn from the distribution that it is modeling. In other words, when g_ϑ is given standard random (for example, normal) sample r as input, it returns a sample x from a distribution it is modeling. In this way, g_ϑ encodes a transformation from a fixed standard (for example, normal) distribution to another distribution we are modeling:

$$r \sim N(0, I) \qquad x = g_\vartheta(r) \qquad x \sim \text{complicated distribution over } x. \tag{7.3}$$

See Figure 7.1 for a visual depiction of this idea. We see that sampling x from distribution encoded by ϑ is extremely simple, it is just an application of g_ϑ. But to get a probability density for x being sampled from this distribution is not quite as streamlined. In our case, the probability density is also needed because we need to calculate an expectation $\mathbb{E}[\phi]$. For this, we require the Jacobian determinant of $g_\vartheta^{-1}(x)$. However, flow-based models are specifically designed for this calculation to be simple. Let us look at the real-valued non-volume preserving (Real NVP) model [257], that will be used in this work, to understand

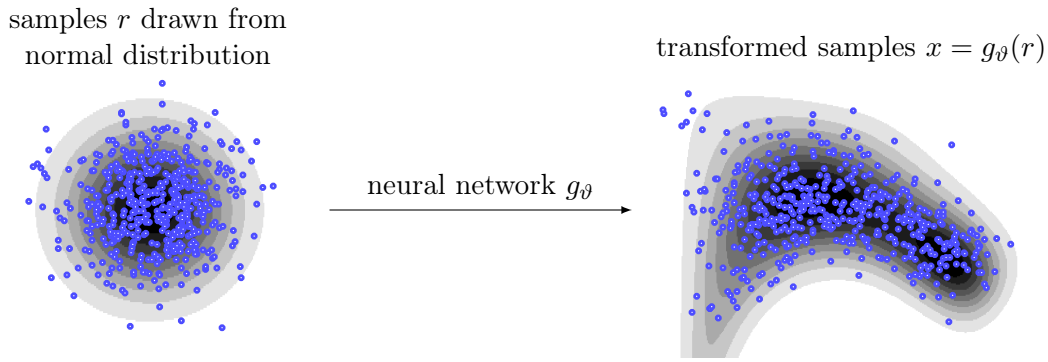


Figure 7.1: Neural network g_ϑ representing a continuous probability distribution by transforming samples drawn from a standard normal distribution.

this property. A single layer of the Real NVP model is

$$\begin{aligned} x_{I_1} &= r_{I_1} \\ x_{I_2} &= r_{I_2} \odot \exp(s(r_{I_1})) + t(r_{I_1}), \end{aligned} \tag{7.4}$$

where I_1 and I_2 are complementary sets of indices and s and t are arbitrary functions (like neural networks). For such a layer, the logarithm of the Jacobian determinant can be shown [257] to be

$$\log \left| \det \left(\frac{\partial x}{\partial r} \right) \right| = \sum_i s(x_{I_1})_i, \tag{7.5}$$

which is just a sum of activations that are an intermediate calculation needed to sample x. This is true because the Jacobian of the transformation (7.4) is triangular[2] and its determinant is simply the product of the diagonal terms, which reduces to a sum in the logarithmic form. If several layers (7.4) are used, the logarithm of the Jacobian determinant is just a sum of all activations s in each layer due to the properties of the logarithm and Jacobians for the composition of functions [257]. Therefore, using the Real NVP model for g_ϑ allows us to easily sample candidates x and easily calculate the expectation of functions defined on x such as $\mathbb{E}[\phi]$.

However, the Real NVP model can only model a continuous probability distribution in a natural way. To model a probability distribution over a discrete set `softmax` output layer is usually used in deep learning. But doing so in our case, where the size of the search space $\mathcal{S}$ is enormous, would make the training of such an architecture intractable. For this reason, discretization D is used in this study which maps the continuous probability distribution to a discrete one. Function D simply maps a real vector to a discrete object (see Figure 7.2 for an illustration). This choice of structure of D is in correspondence with how g_ϑ is also just a mapping of samples from one distribution to another. Table 7.1 lists possible options for D for common search spaces $\mathcal{S}$ found in discrete optimization. We choose a discretization function D in the same manner we choose an encoding in discrete optimization. For some optimization problems that are more constrained, function D is more challenging to find, but this is also the case for encoding in other metaheuristic algorithms [258].

For a more transparent mathematical treatment, let us rename the output of the neural network g_ϑ as $y = g_\vartheta(r)$ to distinguish it from a discrete object $x = D(y)$. The full

[2] x_{I_1} depends only on r_{I_1}, making the Jacobian upper-right block zero, and x_{I_2} depends on both r_{I_1} and r_{I_2} in a way that gives a diagonal lower-right block, so the determinant of the Jacobian reduces to the product of the diagonal entries.

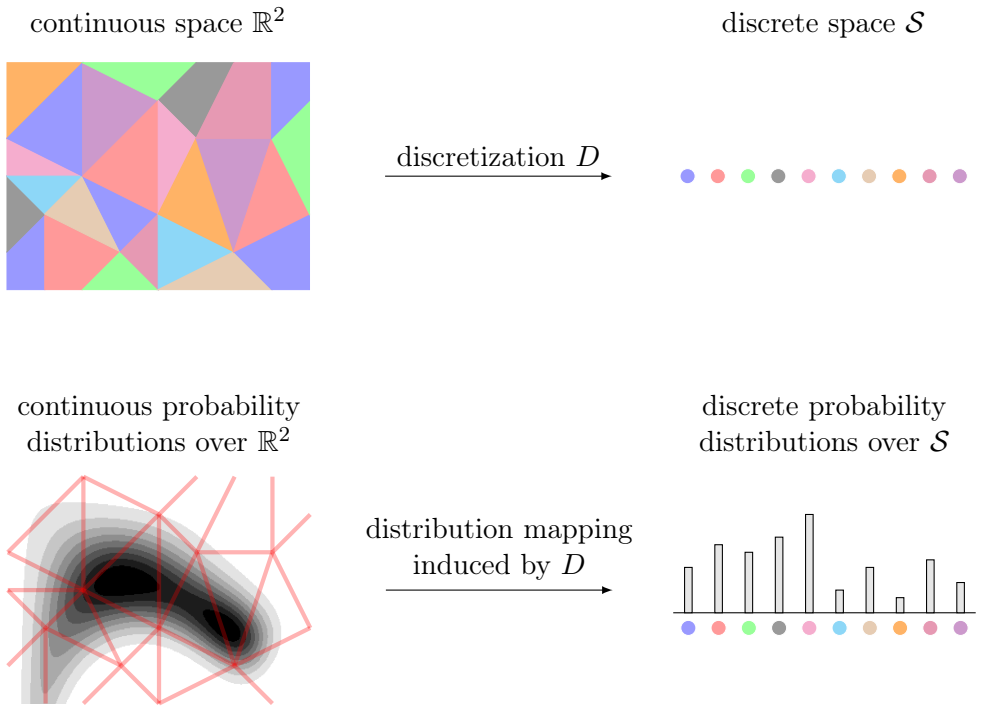


Figure 7.2: Discretization D can be used to map a continuous probability distribution to a discrete probability distribution.

architecture of how our candidates x are sampled together with their objective values is depicted in Figure 7.3. We now have all the ingredients and what remains for discrete optimization problem to be solved by a gradient-based algorithm is to express the gradient of the expected objective value $\mathbb{E}[\phi]$ with respect to the model parameters ϑ. At first glance, we might think that our expectation $\mathbb{E}[\phi]$ will be very difficult to express since such calculation involves integrating the probability distribution on segments where D is constant (triangles depicted in the left part of Figure 7.2). However, since D is a locally constant function and Jacobian of g_ϑ is easy to calculate, this is not the case. Let us denote the probability densities for r and y as f_r and f_y, respectively. Then, the expectation of the objective value, when candidates are sampled, as depicted in Figure 7.3, is

$$\mathbb{E}_{y\sim f_y}[\phi(D(y))] = \int \phi(D(y)) f_y(y)\,\mathrm{d}y. \tag{7.6}$$

Because $\phi(D(y))$ is locally constant, we can express the gradient of this expression by using

Table 7.1: Possible discretization functions D for different encodings commonly used in discrete optimization.

optimization problem	encoding	discretization function	example of x
knapsack problem	binary string	unit step function	01101001
scheduling problem	permutation	argsort	62138547
assignment problem	integer vector	round $\circ$ ReLU	31225013

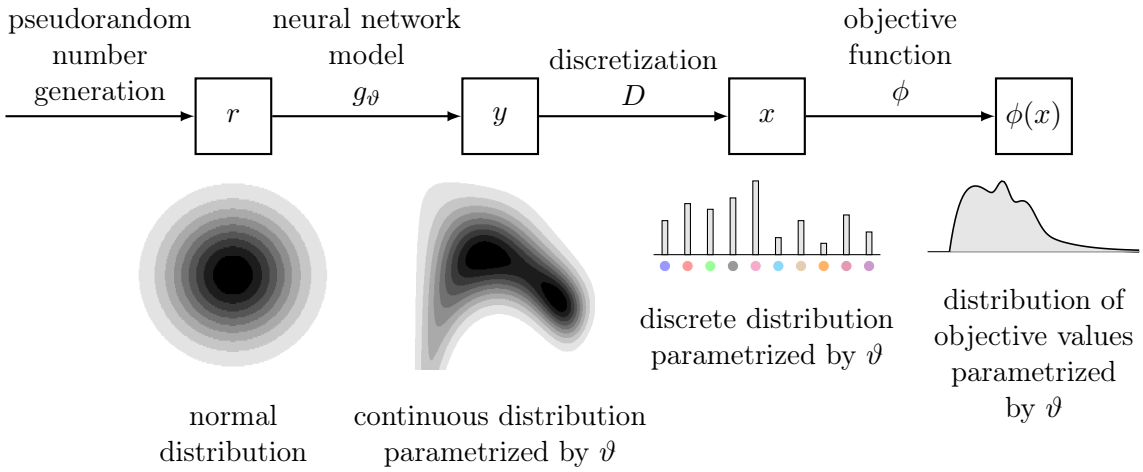


Figure 7.3: Architecture of how the composition of functions is performed to obtain the final objective values. Below each variable, we depict an example distribution that shows how samples of variables might be distributed.

the well-known log-derivative trick from reinforcement learning [259]:

$$\begin{aligned}\nabla_\vartheta \mathbb{E}_{y\sim f_y}[\phi(D(y))] &= \int \phi(D(y))\nabla_\vartheta f_y(y)\,\mathrm{d}y = \int \phi(D(y))\frac{\nabla_\vartheta f_y(y)}{f_y(y)} f_y(y)\,\mathrm{d}y \\ &= \int \phi(D(y))(\nabla_\vartheta \log(f_y(y)))f_y(y)\,\mathrm{d}y \\ &= \mathbb{E}_{y\sim f_y}[\phi(D(y))\nabla_\vartheta \log(f_y(y))].\end{aligned} \tag{7.7}$$

This means that the only quantity we need to differentiate to get the gradient is the logarithm of the probability density at sampled points y. For this, we can use the change of variable formula that is central to flow-based models [257]:

$$f_y(y) = f_r\left(g_\vartheta^{-1}(y)\right)\left|\det\left(\frac{\partial g_\vartheta^{-1}(y)}{\partial y}\right)\right|. \tag{7.8}$$

Since g_ϑ is invertible, we can use the inverse function theorem and express the logarithm as

$$\log f_y(y) = \log f_r\left(g_\vartheta^{-1}(y)\right) - \log\left|\det\left(\frac{\partial g_\vartheta(r)}{\partial r}\right)\Big|_{r=g_\vartheta^{-1}(y)}\right|. \tag{7.9}$$

We can further simplify this expression by choosing f_r to be a uniform distribution which means that the gradient of first summand in (7.9) is zero. This gives us an expression for the gradient:

$$\nabla_\vartheta \mathbb{E}_{r\sim\,\mathrm{uniform}}[\phi(D(g_\vartheta(r)))] = -\mathbb{E}\left[\phi(D(g_\vartheta(r)))\nabla_\vartheta \log\left|\det\left(\frac{\partial g_\vartheta(r)}{\partial r}\right)\right|\right]. \tag{7.10}$$

By using Real NVP for g_ϑ, the derivatives in the expectation are easy to calculate using Eq. (7.5) and standard automatic differentiation technique. In this way we have a formulation of a general discrete optimization problem that can be solved by gradient-based optimization. With respect to the optimization problem, we only need to choose a discretization D that maps an arbitrary real vector into a candidate encoding required by objective function ϕ. The user also needs to choose a generative probabilistic model g_ϑ

that is invertible and preferably has an efficient way of calculating the logarithm of the Jacobian determinant. We found Real NVP to be a suitable choice for such a model. Since g_ϑ is invertible, both the input and the output layer of g_ϑ need to be of the same size as the dimensionality of the domain of D. The authors of [257] advise using at least three layers for a Real NVP model. With this, we have all the ingredients to apply any gradient-based optimization algorithm to solve a discrete optimization problem. Our approach is outlined in Algorithm 7.1 where we used Real NVP for the probabilistic model and "vanilla" gradient descent for the optimization algorithm.

Algorithm 7.1: Our procedure for discrete optimization. In line 10 we employ a generic "vanilla" gradient descent, however, this can be substituted for any other gradient-based optimization algorithm.

Objective function: $\phi : \mathcal{S} \to \mathbb{R}$
Discretization: $D : \mathbb{R}^n \to \mathcal{S}$
Real NVP model: $g_\vartheta : \mathbb{R}^n \to \mathbb{R}^n$
Parameters: model parameters $\vartheta \in \mathbb{R}^p$, population size $m \in \mathbb{N}$, problem encoding size $n \in \mathbb{N}$, learning rate $\alpha > 0$

1 **while** *stopping criterion not met* **do**
2 Initialize grad $\leftarrow 0$
3 **for** $i = 1$ **to** m **do**
4 Generate vector $r_i \in \mathbb{R}^n$ with components drawn from a uniform distribution
5 Compute $y_i = g_\vartheta(r_i)$ and (during this calculation) the gradient of log-determinant $\nabla_\vartheta \log|\det(J)|$ using Eq. (7.5)
6 Discretize y_i to obtain a discrete object $x_i = D(y_i)$
7 Evaluate objective function $\phi_i = \phi(x_i)$
8 Accumulate the gradient: grad $\leftarrow$ grad $- \phi_i \cdot \nabla_\vartheta \log|\det(J)|$ in accordance to Eq. (7.10)
9 **end**
10 Update parameters: $\vartheta \leftarrow \vartheta - \alpha \nabla_\vartheta \Phi(\vartheta)$
11 **end**

7.2 Application to the Traveling Salesman Problem

To assess the effectiveness of the proposed algorithm (outlined in Algorithm 7.1) we consider its application to the traveling salesman problem using a set of benchmark instances from TSPLIB problem library [260]. We chose to solve symmetric problems that have known optimal tours (optimal solutions are known) of sizes 29, 48, 76 and 120. These problem sizes provide a suitable testing ground because they are large enough to exhibit the combinatorial complexity of the task while remaining small enough to allow systematic experimentation and analysis. The goal of this study is not to compete with specialized solvers on large scale instances but to demonstrate that the algorithm is able to adapt its sampling distribution g_ϑ in a way that consistently favors regions of the search space containing high quality tours. For this purpose, the method is compared with random search which serves as a neutral baseline and makes it possible to observe directly how the distribution shifts toward better solutions.

As it is common in the literature [261], we transformed the objective function ϕ for

easier comparison among instances

$$\phi \longleftarrow \frac{\phi - \phi_{\text{optimal}}}{\phi_{\text{random}}}, \tag{7.11}$$

where ϕ_{optimal} is a known optimal value and ϕ_{random} is an average objective value when permutations are sampled uniformly. In this way we can easily place the quality of a solution independently of the problem instance and the number of cities. The encoding chosen for the input of ϕ is a permutation encoded as a vector of distinct integers smaller than the number of cities. To align the encoding to the symmetric problem, we implemented ϕ to add the final link between the first and last node.

For discretization function D, we used `argsort` function that takes a vector of real values and returns the indices that would sort the vector. The output of D can, therefore, be interpreted as a permutation since duplications of real random numbers are extremely unlikely. Nonetheless, we established a safeguard for this occurrence in the implementation of the objective function so that ϕ returns a very large value in case of invalid input x. For the probabilistic model g_ϑ, we used the Real NVP model with four layers. With accordance to suggestions in [257], we used a hyperbolic tangent function multiplied by a learned scale for functions s in Eq. (7.4) and for functions t a two-layered dense networks of twice the width of the input and `ReLU` activations on hidden layers. The initial point for parameters ϑ was chosen randomly in accordance with Glorot initialization [189]. The split of the indices to I_1 and I_2 from Eq. (7.4) was done by halving the vectors and swapping the roles of I_1 and I_2 with each next layer (as is standard for all coupling layer models [231]). Pseudorandom numbers are used as input to g_ϑ and were drawn from a uniform distribution with support $[-1, 1]^n$ where n is the number of cities in a traveling salesman problem being solved. It must be noted that the initial distribution of x is not uniform in the permutation space, however, it was found to be wide enough to cover all possible solutions when testing on small problems.

For optimization, we used the Adam algorithm [73] with a population size (batch size in deep learning terminology) of 1024. This means that 1024 candidates were sampled from our distribution and evaluated for each gradient descent step. The optimization was run for 1000 steps, meaning that our optimization run is approximately equivalent to $2 \cdot 10^6$ objective function evaluations. For each instance, we performed 100 runs with randomly chosen learning rates, while for other parameters, using the default parameter values as implemented in TensorFlow ($\beta_1 = 0.9, \beta_2 = 0.999, \varepsilon = 10^{-7}$) which are widely used and considered robust [262]. Learning rate is known to have the strongest influence on performance [263]. The distribution of the learning rate was chosen in a way that it is uniform when depicted in logarithmic scale and has support $[10^{-7}, 0.1]$. Figure 7.4 shows the best objective value found with respect to learning rate for the four problem instances. We can observe that for all instances there is a clear minimum at $\approx 3 \cdot 10^{-4}$. This is close, but slightly smaller in value, to the usually used learning rate of 10^{-3}.

We additionally performed another 100 runs with the best found learning rate. The convergence of our algorithm for those 100 runs is shown in Figure 7.5. We can see that, in all four cases, our algorithm does not converge toward optimal values. However, in relation to random search, we see that probabilistic model g_ϑ does learn some structure on $(\mathcal{S}, \phi)$, allowing it to converge faster than random search. The convergence of our algorithm and random search is very similar in the earliest stage of the optimization, but then the distribution of candidates starts to shift to fitter ones. But state-of-the-art algorithms for traveling salesman problem achieve close to optimal solutions in about a third of our computational budget [261]. An additional analysis shows that the distribution associated with g_ϑ, after convergence, has one very narrow peak and nearly zero probability around that peak. Furthermore, the final g_ϑ is close to not being invertible anymore with the

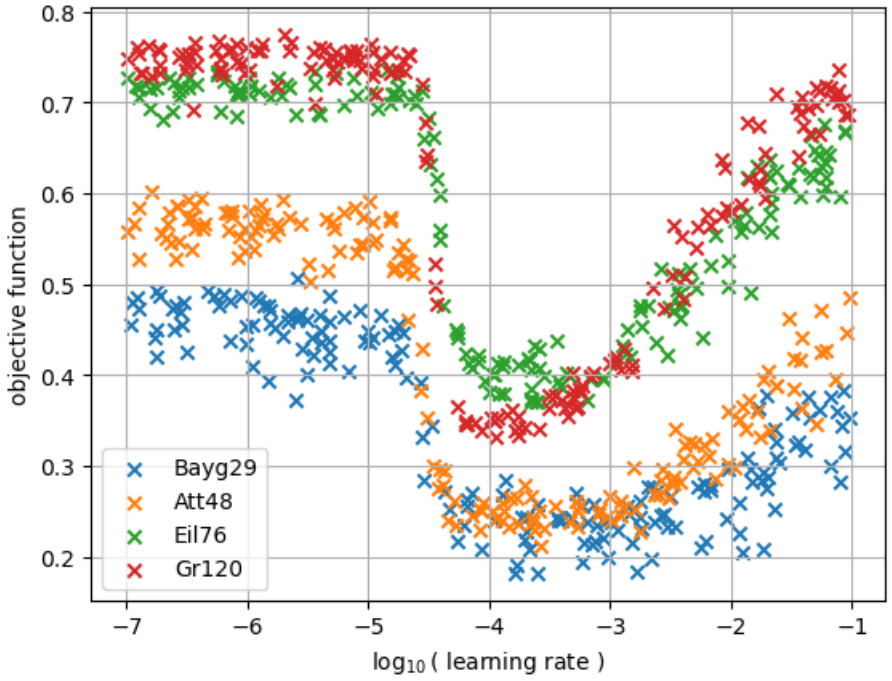


Figure 7.4: The best objective value observed during optimization with Adam with respect to the learning rate for four different problem instances.

condition number of its Jacobian in the order of 10^{-10}. This indicates that our algorithm at first samples diverse candidates, but during optimization the distribution prematurely concentrates close to the best candidates seen so far to such a degree that the algorithm performs no exploration anymore. This phenomenon is known as a mode collapse and is more characteristic of generative adversarial models where it has been well documented and studied [264]. Mode collapse is, on the other hand, not that common for the flow-based models that we used. However, it has been observed in applications connected to Monte-Carlo sampling [265], [266], a use case that has similarities with our methodology. There are several ways to mitigate mode collapse, but in our application, we believe exploration strategies (similar to the ones characteristic for reinforcement learning) to be the most applicable. By this method, we would not sample only from the distribution encoded by ϑ but, with some probability, also from the uniform distribution. This method requires an additional pass of samples through the model and makes the expectation calculation less elegant. This upgrade will be part of our future work.

7.3 Discussion

In this chapter we presented a new general-purpose algorithm for discrete optimization. Given that its usage does not require any other configuration from the user other than the choice of an encoding and a corresponding discretization function, it belongs to the class of metaheuristic algorithms. The key part of the methodology developed here is a mathematically tractable way of transforming a discrete optimization problem into a continuous one without any assumptions about the encoding. Such transformations are otherwise possible to perform but only for a specific class of problems because there are considerations about the encoding and the neighborhood structure that need to be addressed for an objective function to be globally defined. We circumvented this problem by not encoding only a

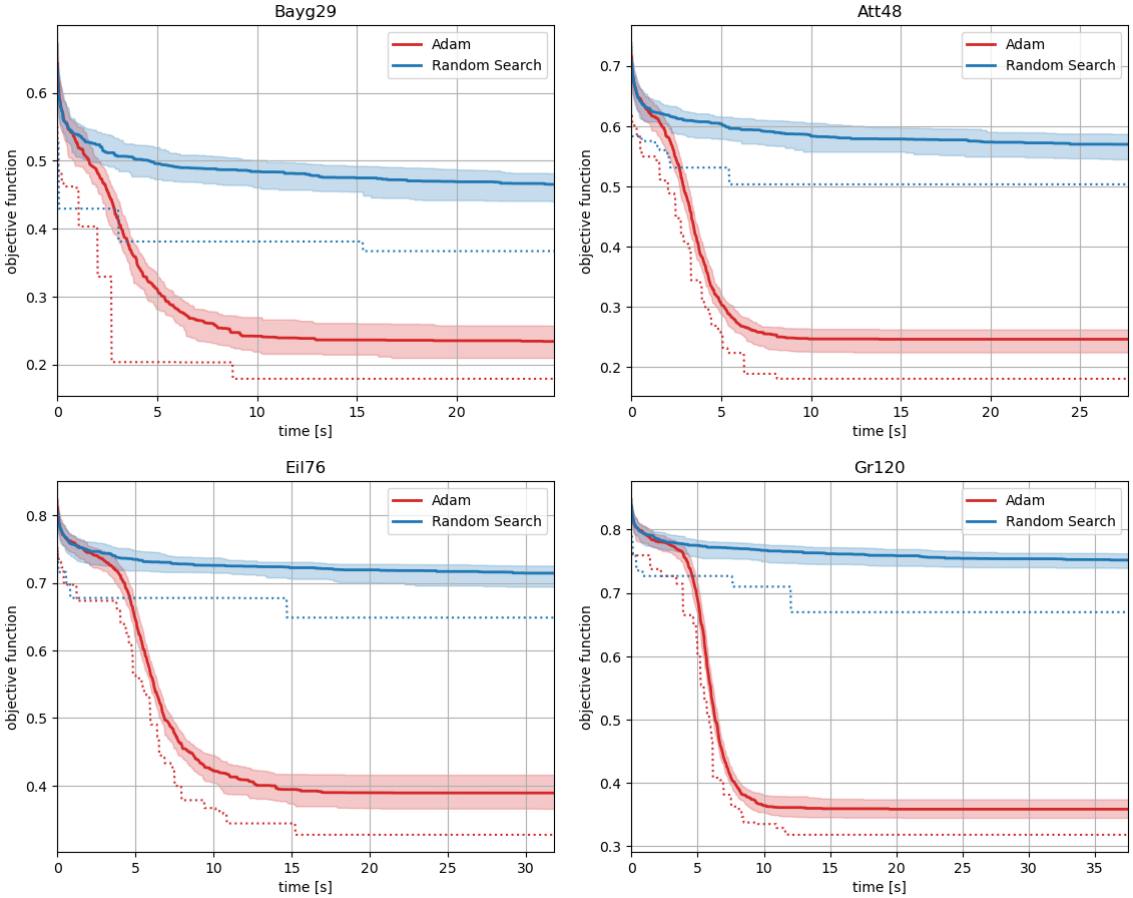


Figure 7.5: Convergence of optimization with Adam and random search for four different problem instances. Full lines show the median objective value over 100 runs, the shaded area represents the central 66% of runs, and the dotted line shows the best objective value observed at each time step across all runs. The figure illustrates the expected statistical behavior of the optimization and the distribution of objective values over time rather than specific individual trajectories. The optimal objective function value is in all cases zero, in accordance to (7.11).

single candidate in a continuous space but rather a probability distribution over discrete candidates. Most surprisingly, we found a way to elegantly calculate the gradient of the transformed objective function which allows the usage of a gradient-based algorithm. This is relevant because the technique of analytical differentiation is an extremely rare sight in discrete optimization. Since gradient calculation is a computationally highly efficient way to predict improvements in the objective value, this brings a new advantage to discrete optimization that was previously absent. In this respect, we can confirm hypothesis **H5**, which states that it is possible to transform discrete optimization problems into a form where generic gradient-based algorithms can be used for optimization.

Our experiments on the traveling salesman problem show that the algorithm works as intended, by shifting and transforming the distribution toward fitter candidates. Comparison with random search shows that our probabilistic model does indeed learn some patterns in the discrete search space that allows it to exploit more promising parts of the search space. However, we have found that convergence of the algorithm, as currently implemented, is premature, and the algorithm does not find candidates that are close to optimal ones. Our analysis shows that this is due to mode collapse that plagues several branches of probabilistic and generative modeling. In order to bring the accuracy of our algorithm to the level of other metaheuristic algorithms, a solution to the mode collapse problem will need to be found. Luckily, this problem is well studied for its occurrence in generative adversarial networks and several mitigation strategies exist. Additional study is needed to test which of these strategies are applicable to our methodology and which of them are successful in discrete optimization setting.

Chapter 8

Parameter Control of Gradient-Based Algorithms

Most optimization algorithms have tunable hyperparameters that influence the behavior of the algorithm and, in turn, its performance. Most commonly, these hyperparameters are constant throughout the run of the algorithm, and usually, the user tests several values for the most important hyperparameters to find an algorithm setting most aligned with the optimization landscape of the problem at hand. This process is termed parameter tuning [64]. But because the optimization landscape changes when algorithms search through the search space, it makes sense to also change the hyperparameters when optimization progresses. This process is termed parameter control [64].

Let us illustrate the importance of parameter control in gradient-based optimization by looking into a hyperparameter that is common to almost all gradient-based optimization algorithms, the so-called learning rate α. This parameter controls the size of steps performed during the descent. It is evident that the value of α needs to be carefully selected so that the steps are not too large, which would skip promising regions in the optimization landscape, or too small, which would result in slow convergence of the algorithm. It is also evident that one single value of α will usually not be ideal for the entire course of the descent because local optimization landscape characteristics change during the descent. For this reason, several methods arose in the literature in the last decade for changing α during optimization, especially in the scope of deep learning, which fueled the interest in gradient-based optimization.

One common way is to change the α using a predefined schedule. Most state-of-the-art models in AI employ this strategy during training by either decaying α or changing it cyclically (see common options in Table 8.1). However, it was found that the choice of an ideal schedule and its parameters strongly depend on the objective function we optimize [267], and there is no clear winner in Table 8.1. Another common method is to change α in response to the magnitude of the gradient. Two common algorithms that employ such parameter control are RMSprop and Adam. In both cases some type of average of gradient magnitude is maintained and used to scale the learning rate for each parameter of the search space separately. This technique brings much faster convergence of the algorithms in real-world setting [73], however, it enlarges the number of hyperparameters and breaks the convergence guarantees that ordinary gradient descent without parameter control enjoys.

In this chapter, we introduce a methodology for adaptive parameter control of the hyperparameters of an arbitrary gradient-based algorithm. Using this approach, changes in hyperparameters are based on the local geometry of the landscape and not only on the magnitude of the gradients or in a predefined schedule fashion, as is the case with current

Table 8.1: Common learning rate schedules and their definitions [267]. In this formulation, the learning rate schedule has the form $k_0 + (k_1 - k_0)g(t)$ where t is the iteration number, $k_0 < k_1$ are the minimum and maximum learning rate values, and $g(t) \in [0, 1]$ determines how the learning rate changes with t. The schedules are classified into three categories: fixed, decaying and cyclic.

Learning Rates		abbr.	k_0	k_1	$g(t)$
Fixed		FIX	k_0	k_0	N/A
Decaying LR	fixed step size	STEP	k_0	0	$\gamma^{floor(t/l)}$
	variable step sizes	NSTEP	k_0	0	Given n step sizes: $l_0, l_1, l_2, ..., l_{n-1}$, $\gamma^i,\ i \in \mathbb{N}\ s.t.\ l_{i-1} \leq t < l_i$
	exponential function	EXP	k_0	0	γ^t
	inverse time	INV	k_0	0	$\frac{1}{(1+t\gamma)^p}$
	polynomial function	POLY	k_0	0	$(1 - \frac{t}{l})^p$
CLR	triangular	TRI	k_0	k_1	$TRI(t) = \frac{2}{\pi}\lvert arcsin(sin(\frac{\pi t}{2l}))\rvert$
	triangular2	TRI2	k_0	k_1	$\frac{1}{2^{floor(\frac{t}{2l})}} TRI(t)$
	triangular_exp	TRIEXP	k_0	k_1	$\gamma^t TRI(t)$
	sin	SIN	k_0	k_1	$SIN(t) = \lvert sin(\pi \frac{t}{2l})\rvert$
	sin2	SIN2	k_0	k_1	$\frac{1}{2^{floor(\frac{t}{2l})}} SIN(t)$
	sin_exp	SINEXP	k_0	k_1	$\gamma^t SIN(t)$
	cosine	COS	k_0	k_1	$COS(t) = \frac{1}{2}(1 + cos(\pi \frac{2t}{l}))$

state-of-the-art methodology in deep learning. During the descent of some gradient-based optimization algorithms, a series of objective value and gradient pairs are calculated but are commonly not used to its full potential to align the behavior of gradient descent to an ever-changing optimization landscape. To better use this information, our parameter control approach uses the same methodology that was employed throughout this thesis and, in fact, uses gradient-based algorithms for changing the hyperparameters of the gradient-based algorithm. To put it differently, it uses the derivatives of the objective function with respect to hyperparameters of the algorithm to know how to change them in an "optimal" way. This strategy of parameter control is not specific to gradient-based algorithms. We could apply it to any optimization algorithm, however, since it requires a gradient of the objective function in its workflow, it is most suitable for gradient-based algorithms where gradients are calculated either way so that parameter control strategy does not bring additional overhead.

This chapter is composed of two sections. The first describes the general methodology we developed for parameter control of gradient-based optimization algorithms, and in the second, we apply this methodology to the Adam algorithm and test it on use cases from deep learning and nonnegative matrix factorization problems.

8.1 Controlling Parameters of Gradient Descent with Gradient Descent

A general gradient-based optimization algorithm has the following form

$$\begin{aligned} x^{i+1} &= f(x^i, g^i, d^i, p^i) \\ d^{i+1} &= g(x^i, g^i, d^i, p^i) \end{aligned}, \tag{8.1}$$

where x^i is a point in the search space, g^i the gradient of the objective function E in point x^i, d^i holds any additional statistics maintained by the algorithm[1] and p^i are the parameters of the algorithm in step i. The parameters p^i may change when gradient descent traverses through the landscape and in the current state-of-the-art algorithms, p^i are either left static or are changed with a predefined schedule, such as $p^{i+1} = \gamma p^i$. Here, however, we will devise a rule for changing p^i that takes E^i and g^i of the previous steps into account. Acknowledging the success of the gradient-based algorithm from previous chapters, we can start with calculating the derivatives of the objective function E with respect to parameters and attempt to devise a rule for p^i that would change p^i using gradient descent.

Objective function E^{i+1} is calculated in point x^{i+1} and this calculation involves parameters p^i, therefore, we can only calculate how p^i influences E^{i+1} for the step that was already done and not for the one that will come next. However, this is still a quite useful information. If we assume that the optimization landscape features do not change too abruptly, then finding out what parameter values would be more useful in the previous step can help us in the next step as well. The derivative of E^{i+1} with respect to p^i is:

$$\frac{\partial E^{i+1}}{\partial p_k^i} = \sum_j \frac{\partial E(x^{i+1})}{\partial x_j^{i+1}} \frac{\partial x_j^{i+1}}{\partial p_k^i} \qquad \text{chain rule for differentiation} \tag{8.2}$$

$$= \sum_j g_j^{i+1} \frac{\partial x_j^{i+1}}{\partial p_k} \qquad \text{definition of the gradient} \tag{8.3}$$

$$= \sum_j g_j^{i+1} \frac{\partial f(x^i, g^i, d^i, p^i)_j}{\partial p_k^i} \qquad \text{definition of the update rule.} \tag{8.4}$$

If we now define a matrix J, which is a Jacobian matrix of f, the update rule of our gradient-based algorithm, with respect to its parameters p^i:

$$J_{jk}^i = \frac{\partial f(x^i, g^i, d^i, p^i)_j}{\partial p_k^i}, \tag{8.5}$$

we can write

$$\frac{\partial E^{i+1}}{\partial p_k^i} = \sum_j g_j^{i+1} J_{jk}^i \tag{8.6}$$

$$\frac{\partial E^{i+1}}{\partial p^i} = g^{i+1} \cdot J^i, \tag{8.7}$$

where $\cdot$ is matrix multiplication. Matrix J^i measures in which direction x^{i+1} would move if p^i changed. Each column of matrix J^i corresponds to a specific parameter p_k and is a vector in the search space that points in the direction in which x_{i+1} would move if p_k was larger. By multiplying J^i with the gradient g^{i+1} we get how E^{i+1} would change if parameters p^i changed.

We can also look at Eq. (8.7) from a more geometric perspective, which offers us a more visual explanation of how this works. To better understand the principle behind this equation, let us look at a simple example of ordinary gradient descent with momentum. Such an algorithm has two parameters. One is the learning rate α and the other is β, which is an exponential decay factor between 0 and 1 that determines the relative contribution of the current gradient and earlier gradients to the step. Figure 8.1 shows a single step of such

[1] such as running averages of the gradient (if momentum is used) or average scale of the gradient magnitude (such as in AdaGrad)

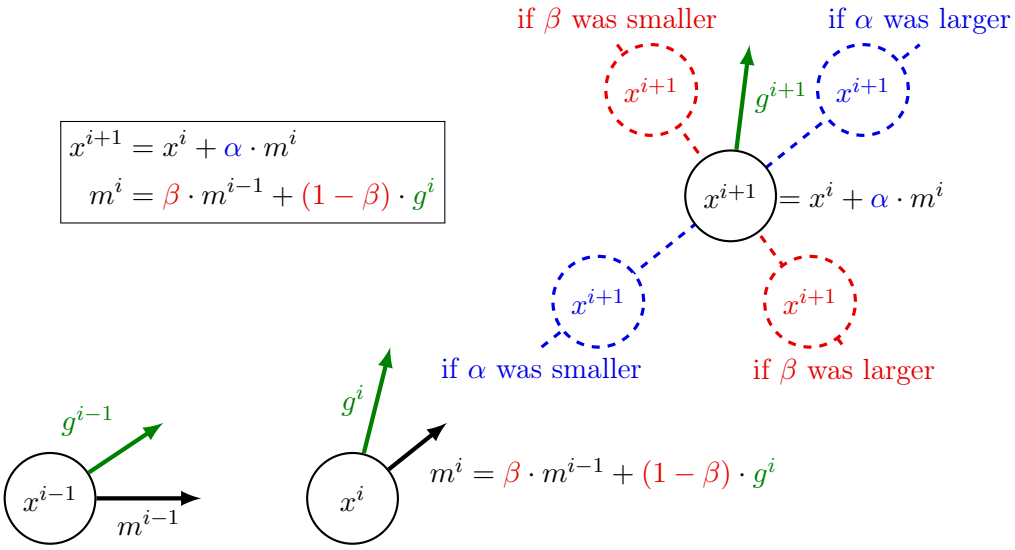


Figure 8.1: Application of a step of gradient descent with momentum from point x^i to x^{i+1} when applied to a maximization problem. Two parameters α and β determine the position of x^{i+1} given gradient g^i and momentum in the previous step m^{i-1}. Dashed lines show where x^{i+1} would end up if parameters α and β have been chosen differently. Based on the gradient g^{i+1} in the new point x^{i+1} we can deduce in which direction the parameters α and β would need to be changed so that x^{i+1} would end up in a region with a better objective function value.

gradient descent when applied to a maximization problem[2]. In the figure, we see all the relevant information needed to perform a step and the directions to which the columns of J are pointing at (dashed lines in Figure 8.1) which tell how the position of x_{i+1} would change if α or β was changed (prior to the gradient descent step from x_i to x_{i+1}). Comparing these directions to the direction of the gradient g^{i+1} in point x^{i+1} we can conclude (for the specific case shown in Figure 8.1) that the objective value would be more favorable when moving from x_i to x_{i+1} if we chose α to be larger and β to be smaller. Intuitively, this is because the gradient g^{i+1} points in a direction with more favorable objective values. This means that if α and β is adjusted in a way that x^{i+1} is moved in the direction of g^{i+1} our gradient descent step from x^i to x^{i+1} is expected to yield a larger improvement in the objective value. More formally, Eq. (8.7) expresses that the derivative of the objective value attained after the gradient descent step with respect to a given parameter (such as α) is the scalar product of the corresponding column in J^i with the gradient g^{i+1}. This is the principle behind the parameter control strategy we propose in this chapter.

We can now take an arbitrary gradient-based optimization algorithm defined by functions f_1 and g_1 with the update rule

$$\begin{aligned} x^{i+1} &= f_1(x^i, g^i, d^i, p^i) \\ d^{i+1} &= g_1(x^i, g^i, d^i, p^i) \end{aligned} \tag{8.8}$$

[2]We chose a maximization problem because, in this case, gradient descent moves in the direction of the gradient and not away from it, which provides a more intuitive visual depiction.

and apply our parameter control methodology to it in the following way:

$$\begin{aligned} x^{i+1} &= f_1(x^i, g^i, d^i, p^i) \\ d^{i+1} &= g_1(x^i, g^i, d^i, p^i) \\ p^{i+1} &= p^i \pm \mu \; g^{i+1} \cdot J^i, \end{aligned} \tag{8.9}$$

where $(+)$ applies to maximization and $(-)$ to minimization problems. Here, we took the most primitive strategy, ordinary gradient descent, for changing p^i in an adaptive manner. In (8.9) we see that p^i is iteratively adapted in a direction of $\partial E/\partial p$ which is the direction that is expected to result in the biggest change in E per given step of the algorithm.

However, we can choose any gradient-based optimization algorithm to change p^i during the descent. In general, our parameter control strategy that adaptively controls the algorithm defined by f_1 and g_1 is

$$\begin{aligned} x^{i+1} &= f_1(x^i, g^i, d^i, p^i) \\ d^{i+1} &= g_1(x^i, g^i, d^i, p^i) \\ p^{i+1} &= f_2(p^i, g^{i+1} \cdot J^i, h^i, r^i) \\ h^{i+1} &= g_2(p^i, g^{i+1} \cdot J^i, h^i, r^i), \end{aligned} \tag{8.10}$$

where we use a gradient-based optimization algorithm defined by f_2 and g_2 for adapting p^i (this algorithm has its own parameters r^i and maintained statistical information held in h^i).

8.2 Application to Adam

In this section, we apply the developed parameter control strategy on Adam, one of the most used gradient-based optimization algorithms. As we can see in Algorithm 8.1, Adam has four tunable parameters that control its behavior. There is the learning rate α that controls the step size, two forgetting factors β_1 and β_2 that control how far-reaching are the running averages of first- and second-order gradient information and a positive dampening factor ε that prevents divisions with very small numbers. In modern implementations, only the learning rate α is changed during the descent (using a predefined schedule) while other parameters are held at fixed values.

Before applying our parameter control, we need to address the initialization bias correction, which can be seen in line 9 of Algorithm 8.1. This part is designed for fixed β_1 and β_2 and would have a different form if we allow β_1 and β_2 to change during the descent. Therefore, we need to generalize the Adam algorithm to allow dynamical parameters before we introduce the parameter control. Adam uses both momentum and per-parameter learning rate adaptation. For this, it needs to maintain a running exponential average of both first- and second-order gradient information. One of the important contributions of Adam is that it does this in an unbiased way with respect to the number of steps already done. Line 9 of Algorithm 8.1 takes care of underestimation of the moments that occur in other algorithms (for example, RMSprop) during the first stages of the descent.

Let us take the running average m, for example. Its update rule in Adam is

$$m_i = \beta_1 m_{i-1} + (1-\beta_1) g_i = (1-\beta_1) \sum_{j=1}^{i} \beta_1^{i-j} g_j, \tag{8.11}$$

where we expanded the recurrence by repeated substitution. Taking expectations and

Algorithm 8.1: Adam.

Objective function: $E(x)$
Parameters: $\alpha > 0, \quad \beta_1 \in [0,1), \quad \beta_2 \in [0,1), \quad \varepsilon > 0$

1 $x \leftarrow \text{rand}$
2 $m \leftarrow 0 \cdot x$
3 $v \leftarrow 0 \cdot x$
4 $i \leftarrow 1$
5 **while** *stopping criterion not met* **do**
6 $\quad g \leftarrow \nabla_x E(x)$
7 $\quad m \leftarrow \beta_1 \cdot m + (1-\beta_1) \cdot g$
8 $\quad v \leftarrow \beta_2 \cdot v + (1-\beta_2) \cdot g^2$
9 $\quad \eta \leftarrow \alpha \frac{\sqrt{1-\beta_2^i}}{1-\beta_1^i}$
10 $\quad x \leftarrow x - \eta \frac{m}{\sqrt{v}+\varepsilon}$
11 $\quad i \leftarrow i + 1$
12 **end**

assuming that g_j are drawn from a stationary distribution, we obtain

$$\mathbb{E}[m_i] = (1-\beta_1)\mathbb{E}[g] \sum_{j=1}^{i} \beta_1^{i-j} = (1-\beta_1^i)\mathbb{E}[g], \tag{8.12}$$

where the last equality follows from the formula for the sum of a geometric series. This shows that m_i is biased toward zero due to initialization, which motivates the bias correction step in line 9 of Algorithm 8.1. If, on the other hand, β_1 changes during the descent, this scaling needs to be performed differently for the average to be unbiased. We can achieve this by introducing a new variable c_m, which measures to what degree the average is underestimated:

$$\mathbb{E}[m_i] = \mathbb{E}[g] \sum_{j=1}^{i} (1-\beta_{1j}) \prod_{k=j+1}^{i} \beta_{1k} = c_m \mathbb{E}[g]. \tag{8.13}$$

Variable c_m can be calculated more easily by using recursion, as can be seen in line 11 of Algorithm 8.2, which is an expanded form of the standard Adam algorithm that allows β_1 and β_2 to change during the descent. It includes two additional variables c_m and c_v, for each running average m and v, respectively. If β_1 and β_2 are left constant, we get the standard Adam algorithm with

$$\begin{aligned} c_m &= 1 - \beta_1^i \\ c_v &= 1 - \beta_2^i \end{aligned}. \tag{8.14}$$

We can now implement our parameter control in Algorithm 8.2. We will call this algorithm cAdam for controlled Adam, and its pseudocode is shown in Algorithm 8.3. This algorithm has two additional steps. The first additional step is the calculation of the Jacobian matrix J, which is performed after a gradient descent step to a new point (line 15). J holds the derivatives of the move just made with respect to the four parameters. The second additional step is to apply the standard Adam algorithm to change the four parameters when the new gradient becomes available (line 8). The change in four parameters

Algorithm 8.2: Adam with dynamical parameters.

Objective function: $E(x)$
Sequences: $\alpha_i > 0, \quad \beta_{1i} \in [0,1), \quad \beta_{2i} \in [0,1), \quad \varepsilon_i > 0; \quad i = 1, 2, \ldots$

$x \leftarrow \text{rand}$
$m \leftarrow 0 \cdot x$
$v \leftarrow 0 \cdot x$
$c_m \leftarrow 0$
$c_v \leftarrow 0$
$i \leftarrow 1$
while *stopping criterion not met* **do**
  $g \leftarrow \nabla_x E(x)$
  $m \leftarrow \beta_{1i} \cdot m + (1 - \beta_{1i}) \cdot g$
  $v \leftarrow \beta_{2i} \cdot v + (1 - \beta_{2i}) \cdot g^2$
  $c_m \leftarrow \beta_{1i} \cdot c_m + (1 - \beta_{1i})$
  $c_v \leftarrow \beta_{2i} \cdot v_m + (1 - \beta_{2i})$
  $\eta \leftarrow \alpha_i \dfrac{\sqrt{c_v}}{c_m}$
  $x \leftarrow x - \eta \dfrac{m}{\sqrt{v} + \varepsilon_i}$
  $i \leftarrow i + 1$
end

Algorithm 8.3: cAdam.

Objective function: $E(x)$
Initial Values for Parameters: $\alpha > 0, \quad \beta_1 \in [0,1), \quad \beta_2 \in [0,1), \quad \varepsilon > 0$

$x \leftarrow \text{rand}$
$m \leftarrow 0 \cdot x$
$v \leftarrow 0 \cdot x$
$c_m \leftarrow 0$
$c_v \leftarrow 0$
while *stopping criterion not met* **do**
  $g \leftarrow \nabla_x E(x)$
  update $(\alpha, \beta_1, \beta_2, \varepsilon)$ with Adam using derivatives $g \cdot J$ (Eq. (8.7))
  $m \leftarrow \beta_1 \cdot m + (1 - \beta_1) \cdot g$
  $v \leftarrow \beta_2 \cdot v + (1 - \beta_2) \cdot g^2$
  $c_m \leftarrow \beta_1 \cdot c_m + (1 - \beta_1)$
  $c_v \leftarrow \beta_2 \cdot v_m + (1 - \beta_2)$
  $\eta \leftarrow \alpha_i \dfrac{\sqrt{c_v}}{c_m}$
  $x \leftarrow x - \eta \dfrac{m}{\sqrt{v} + \varepsilon}$
  calculate J in accordance with Eq. (8.5)
end

is made in a direction that would reduce the objective function the most in previous step considering only the space $(\alpha, \beta_1, \beta_2, \varepsilon)$.

The cAdam algorithm requires us to set four initial values for parameters, however, they will change during the descent toward values that are appropriate for the observed optimization landscape. It makes sense starting with values that have been found useful for the standard Adam algorithm over the years. This should be a good starting point from where cAdam is able to change the four parameters to more useful values. The default values for the Adam algorithm in the TensorFlow library are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\varepsilon = 10^{-7}$. These are the initial values used in our work.

For updating parameters $(\alpha, \beta_1, \beta_2, \varepsilon)$, we use the standard Adam algorithm (line 8). We could use any gradient-based algorithm for these updates, however, we chose Adam because it works well for both deterministic and stochastic objective functions. This brings robustness and should make cAdam useful in both cases without the need for additional modifications. The standard Adam algorithm that performs parameter updates requires its own parameters as well, for which we take the default values mentioned in the previous paragraph.

8.2.1 Experiments

We tested the cAdam algorithm in two optimization settings, with stochastic and deterministic objective functions. In the first case, we apply it to the task of training a convolutional neural network in a supervised machine-learning setting. In this case the objective function is stochastic due to the network training in minibatches. In the second case, we apply it to training a symmetric NMTF model in a unsupervised machine learning setting. In this case, the soft clustering is performed on data using the entire data set in each step, which makes this objective function deterministic. In both cases, we use the implementation of cAdam in the TensorFlow library in which both models were implemented. The gradients of the objective function are calculated by automatic differentiation using the native TensorFlow constructs while derivatives contained in matrix J (see Eq. (8.5)) are calculated by symbolic differentiation and hardcoded in the cAdam algorithm.

8.2.1.1 Neural Network Training

For testing cAdam in a deep learning setting, we selected a widely used benchmark problem from computer vision, the recognition of handwritten digits. This classification task requires the neural network to map an input image to probabilities over ten possible classes (digits from 0 to 9). As a model, we adopted a simple convolutional architecture consisting of one convolutional layer with 28 channels and kernel size 3, followed by max-pooling of size 2 and a dense hidden layer of size 128 with ReLU activation. Dropout with probability 0.2 was applied to the hidden dense layer, and the final class probabilities were obtained with a dense output layer of size 10 and softmax activation. The model was trained on the MNIST dataset [268] with categorical cross-entropy loss and mini-batches of size 128. This architecture was chosen for its simplicity and computational efficiency, which makes it suitable for clearly illustrating the behavior of cAdam without being influenced by architectural complexity or extensive hyperparameter tuning. The goal of this experiment is not to achieve state-of-the-art performance, but to demonstrate the application of cAdam on a standard deep learning task.

Both Adam and cAdam were applied 100 times from different random starting points and with different random orders of feeding the data. For both Adam and cAdam, default parameter values for $(\alpha, \beta_1, \beta_2, \varepsilon)$ were used. Comparison in performance of the algorithms is depicted in Figure 8.2 with full lines being the median over 100 runs and shaded areas

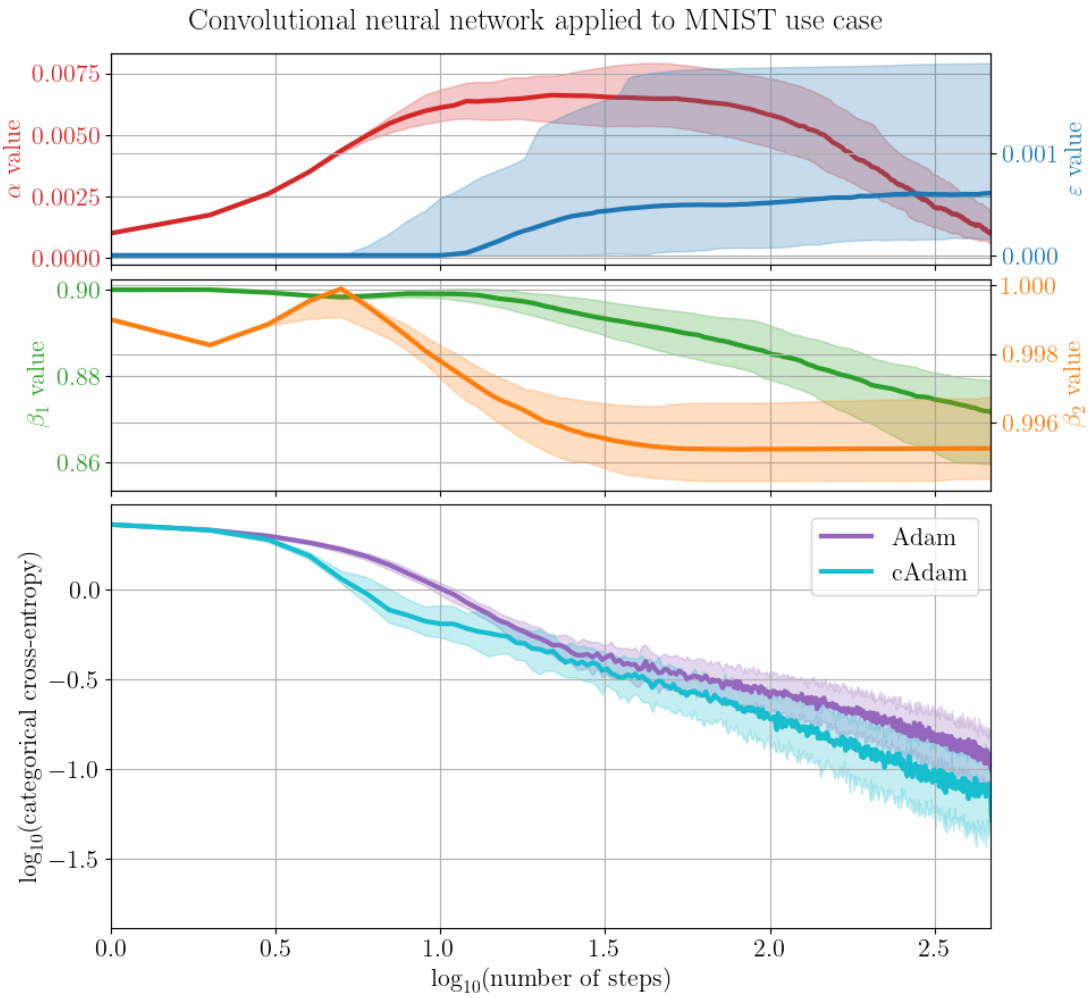


Figure 8.2: Comparison of the Adam and cAdam algorithms for training a convolutional neural network on an MNIST data set. The full line is the median over 100 runs, and the shaded area includes the central 66% of runs. The bottom plot shows how objective value changes with respect to the number of steps of gradient descent in the log-log scale. The top two plots show how the parameters α, β_1, β_2, and ε of the cAdam algorithm change during the descent in the lin-log scale.

containing the central 66% of the runs. Bottom part of Figure 8.2 shows how Adam and cAdam converge (objective function with respect to the number of function evaluations). The top part of Figure 8.2 shows how parameters $(\alpha, \beta_1, \beta_2, \varepsilon)$ used by cAdam change during the descent.

In Figure 8.2, we can see that cAdam indeed changes parameters $(\alpha, \beta_1, \beta_2, \varepsilon)$ significantly during the descent. These changes in parameters result in cAdam converging approximately one standard deviation below the Adam algorithm, whose parameters are left constant at default values. For comparison, the values of Adam's parameters are identical to the initial values of cAdam's parameters. The progress of the learning rate α for cAdam, which can be seen in the top part of Figure 8.2, can be compared to deterministic

schedules used for Adam. In state-of-the-art deep learning applications, the learning rate is usually increased from a small value (warm-up stage) to a peak value and then decayed back to smaller values [269]. This is exactly what we see in Figure 8.2 which means that cAdam has uncovered a schedule for α that is similar in shape to the ones found by extensive experimentation by the deep learning community over the past decade. However, in the case of cAdam, this schedule is tailored to the specific problem (geometrical properties of the current optimization landscape) and requires no tuning.

Unlike the learning rate α, the progress of β_1, β_2, and ε seen in Figure 8.2 can not be compared to any established schedules in deep learning since those parameters are left constant in the state-of-the-art applications. We can see that β_1, β_2, and ε are close to constant in the first stage of the optimization, however, all three start changing in value at a similar point in time. Parameters β_1 and β_2 start decreasing while ε starts increasing. This means that in the second stage of the optimization, a shorter averaging window is needed, which is indicated by smaller β_1 and β_2. Increase in ε is much harder to interpret since this parameter is meant only to prevent division with very small numbers. There are indications that an increase in ε reduces the effect of Adam's per-parameter learning rate adaptation [270] as well as its variance [271]. Both arguments support an explanation that in the part of the optimization where ε is not negligible, per-parameter learning-rate adaptation scheme is not a good approximation anymore. Limitations of this approximation were also pointed out in [272], where the authors shed doubt on the validity of this approximation and even on the necessity of the square root in the denominator of the per-parameter learning-rate adaptation scheme.

8.2.1.2 Performing Soft Clustering with Symmetric NMTF

The second setting we tested cAdam on is for training an unsupervised machine learning model. For this, we chose a soft clustering task where we operate on the entire data, which makes the objective function deterministic (unlike the deep learning use case from the previous section). For soft clustering, we use a symmetric NMTF model as described in Chapter 4. We already know that standard Adam works well in this setting and we are interested if our parameter control brings any benefits. Using cAdam, we perform soft clustering on four real-world data sets described in Section 4.2.4.1. Based on the results from Figure 4.5 in Section 4.2.4.2, we already know the appropriate number of clusters (latent dimension) k needed for sufficient clustering. We used $k = 10$ for Aural, $k = 3$ for Facerec, $k = 12$ for Protein, and $k = 3$ for Zongker. The nonnegativity constraint was handled by using the absolute value transformation. The initial points were generated by adding small noise to solutions of singular value decomposition and projecting them to nonnegative space. We used the mean square error as a metric and trained 100 models for each data set using the standard Adam and cAdam algorithms. In Figure 8.3, the convergence of the two algorithms is depicted for all four data sets. The full line is the median over 100 runs, and the shaded area contains the central 66% of all runs. We can see that the difference in performance between Adam and cAdam is much more pronounced compared to the deep learning use case. For the Facerec data set, cAdam performs slightly better compared to Adam, while for the other three data sets, cAdam converges much faster. Figure 8.4 shows how the four parameters of cAdam change during the descent for the Protein data set. The progress is similar in shape to the one observed for the deep learning use case. This indicates that similar schedules apply for both deterministic and stochastic objective functions and that the experience of the deep learning community can be transferred to other optimization problems. As is the case with the deep learning use case, it is beneficial that the learning rate α first increases (warm-up stage), is then left constant, and finally decreases when approaching the minimum (learning rate decay stage).

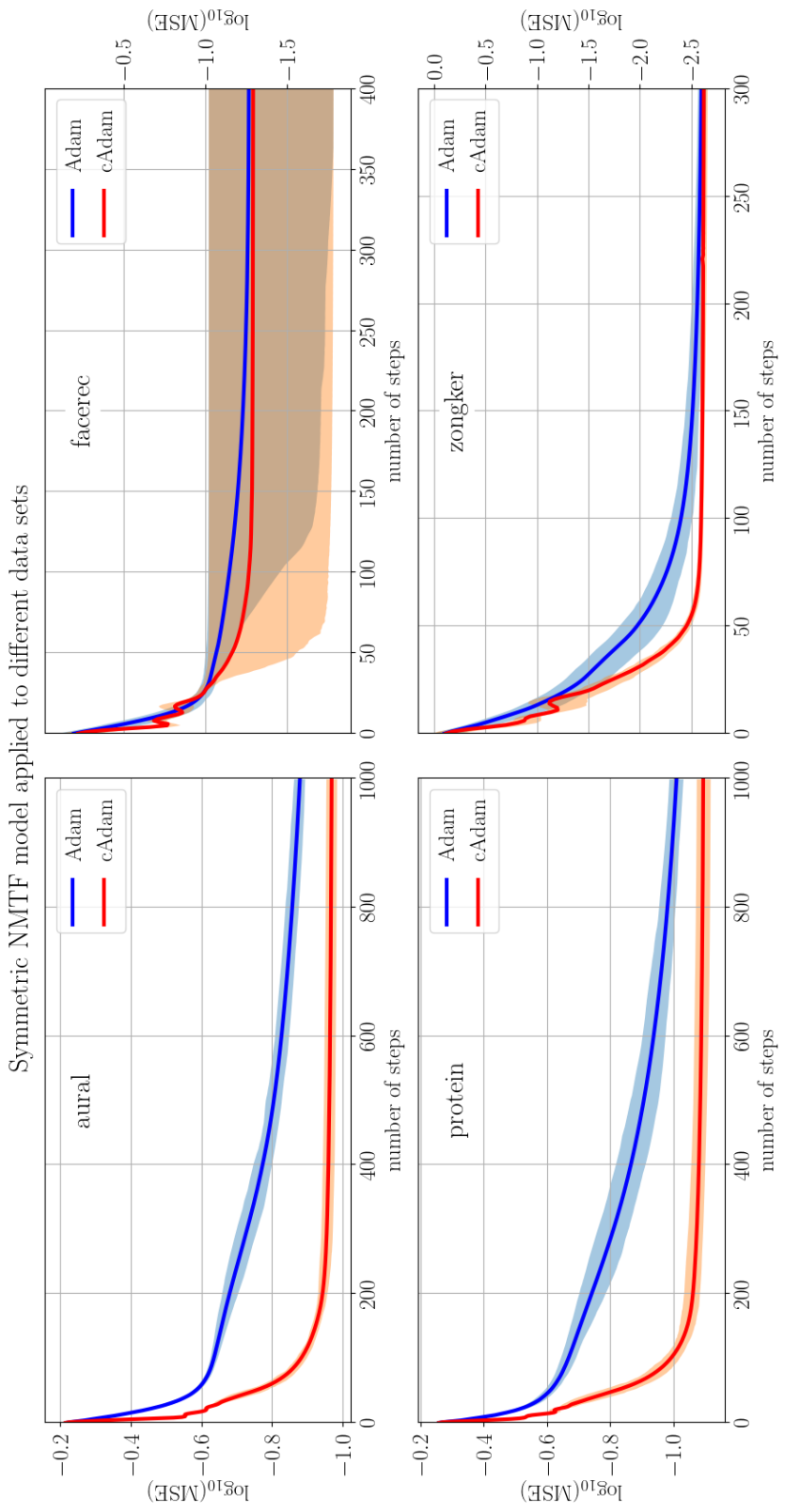


Figure 8.3: Comparison of the Adam and cAdam algorithms for training symmetric NMTF model for soft clustering on four real-world data sets. The full line is the median over 100 runs, and the shaded area includes the central 66% of runs. The plots show how objective value changes with respect to the number of steps of gradient descent in the log-lin scale.

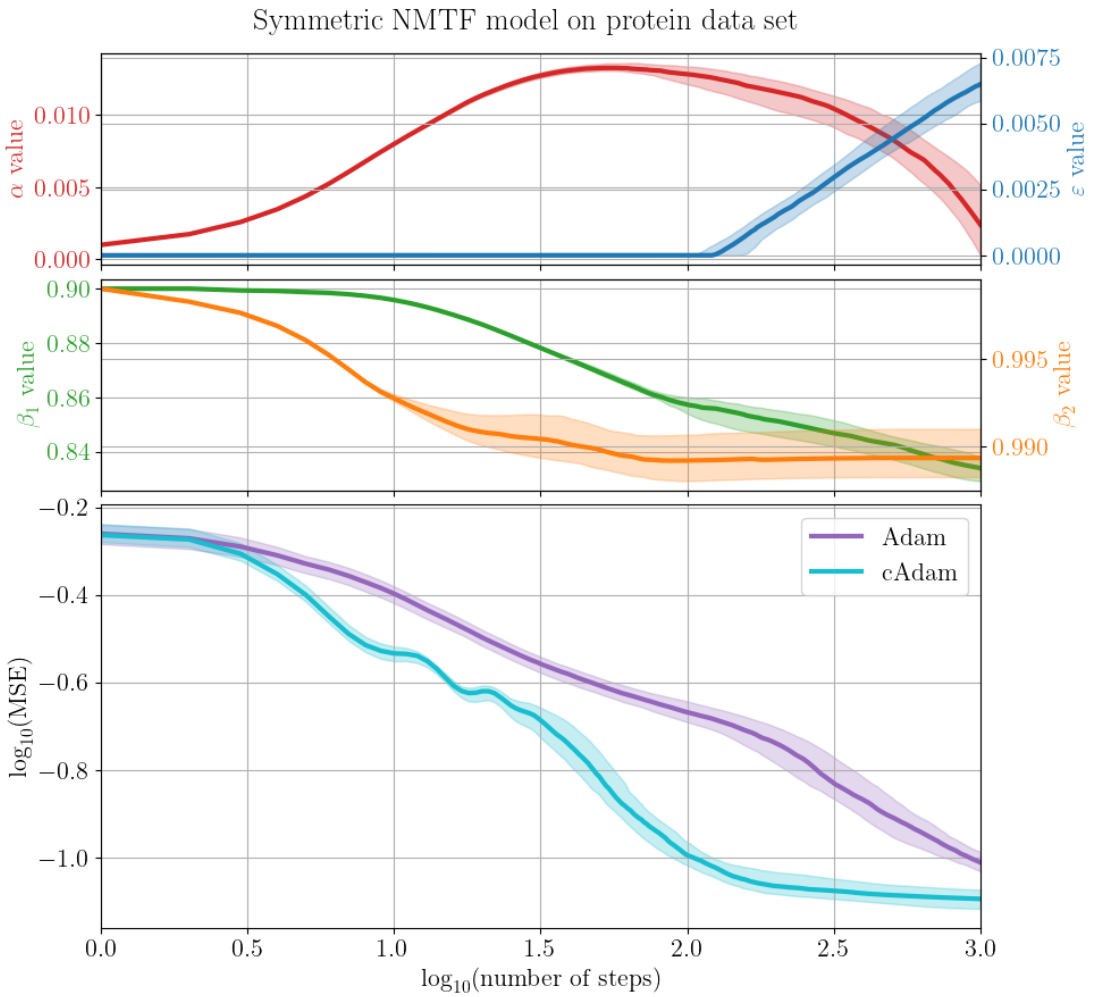


Figure 8.4: Comparison of the Adam and cAdam algorithms for training symmetric NMTF model for soft clustering on the Protein data set. The full line is the median over 100 runs, and the shaded area includes the central 66% of runs. The bottom plot shows how objective value changes with respect to the number of steps of gradient descent in the log-log scale. The top two plots show how the parameters α, β_1, β_2, and ε of the cAdam algorithm change during the descent in the lin-log scale.

Parameters β_1 and β_2 decrease during the descent, indicating that less averaging is needed as the optimization progresses. Parameter ε is left close to zero for the most part of the descent and then linearly rises when learning rate α starts to decay. Our interpretation is that when optimization closes on the minimum (when a lower learning rate is needed), the per-parameter learning rate adaptation that Adam uses becomes less and less precise in approximating the correct scaling factors.

8.3 Discussion

In this chapter, we developed a parameter control strategy applicable to any gradient-based optimization algorithms. We applied this strategy to a popular gradient-based algorithm Adam that resulted in a new algorithm we call cAdam. We tested cAdam on two use cases, one with deterministic objective function evaluation and one with stochastic one. Based on the results, we can deduce that cAdam works better with deterministic objective functions compared to stochastic ones. In both cases, we can see an improvement compared to Adam, however, the difference between them is much more pronounced when applied to a deterministic optimization setting. This was, in fact, expected since our parameter control method works by adjusting parameter values in small steps. Any noise should, therefore, cause the parameters to converge to ideal values more slowly. When there is no noise present, the changes in parameter values are more coherent and our parameter control can follow ideal parameter values faster. To combat this noise, we chose to use the Adam algorithm for changing the parameter values, which is known to work well in stochastic setting. However, we believe that there is space to improve cAdam in a stochastic setting by looking into other algorithms to change the parameter values from its derivatives, possibly from control theory and of higher order (for example, Gauss-Newton and AdaHessian). Also, the current theoretical derivation of the parameter control was performed for a deterministic objective function, and by expanding this treatment to probabilistic one might offer new insights into further possibilities.

Given the results of the presented experiments, we can fully confirm hypothesis **H6**. It is indeed possible to adaptively guide the parameters of gradient descent simply by following the derivatives of the objective function with respect to parameters. Furthermore using this approach leads to equal or better convergence properties of the algorithm. Given that no parameter tuning is required when using cAdam, this approach simplifies the optimization or training workflow. This reduces both the computational costs and the final optimization score.

As mentioned in previous sections, the observed progress of adapting learning rate with cAdam is similar in shape to the state-of-the-art schedules used in deep learning. However, using such schedules requires a user to preset for how long and to what value each stage of the schedule goes to. By using cAdam, none of this is required, and the "schedule" is automatically adapted based on the geometry of the optimization landscape. It is interesting that we found that changes in other parameters are also beneficial. In the current literature, not much effort was put into scheduling β_1, β_2, and ε parameters since learning rate α was recognized to be the most important of the four parameters. However, our results indicate that decaying β_1 and β_2 are beneficial and that in the final stage of the descent ε should be risen to nonnegligible value. Based on the current results, we can not definitely conclude that the changes in β_1, β_2, and ε are crucial to achieve the observed superiority of cAdam. For this, further experiments are required where different subsets of parameters are adaptively changed while the others are left fixed.

Chapter 9

Conclusions

In this thesis, we aimed toward showing that gradient-based algorithms can be viewed as general-purpose optimization algorithms if the search space dimension is large enough (large-scale optimization setting). As with other metaheuristic algorithms, when applied to settings that are different from their original use cases, there is some level of adjustment that needs to be performed for an algorithm to be applicable. In the case of gradient-based optimization, we developed several methods to achieve this. We have shown that these methods allow gradient-based optimization to be applicable for optimization use cases from supervised, unsupervised, and reinforcement learning, as well as for use cases from simulation-based and discrete optimization, and optimization parameter control. In this regard, we strived toward being comprehensive in our use cases and to cover currently the most important classes of optimization problems. In some cases, we achieved performance that is comparable to, or even better than, the one achieved by commonly applied metaheuristic algorithms. In some other cases, our results were not quite state-of-the-art, but we have shown that gradient-based optimization is applicable to the problems considered using our methodology. For these cases, more study is needed to address some problems we identified.

Our experiments with gradient-based optimization were also in agreement with many theoretical and empirical claims about their efficiency in large-scale optimization. We have observed that they can find fitter candidates in a given computational budget compared to some other metaheuristic algorithms (see Chapters 3 and 6). We attribute this rapid convergence to a computationally efficient way of determining promising descent directions during optimization. While derivative-free algorithms require many objective function evaluations (dependence is at best linear in search space dimension) to find promising descent directions, gradient-based algorithms can acquire this information through the gradient, which can be computed for the cost equivalent to just one additional objective function evaluation. In agreement with [13]–[15], we did not observe any problems due to local minima. It seems that our use cases also share the property, observed both theoretically and empirically in [13]–[15], that local minima are, in large-scale setting, concentrated very close to the global minimum[1]. Also, we did not observe any detrimental effects due to the presence of saddle points. This is also in line with the current literature since most modern algorithms have mechanisms that enable them to cross flat regions (such as saddles) efficiently[2]. In Chapter 7, we did encounter premature convergence due to mode collapse,

[1] On average, the gap between objective values of local and global minimum falls exponentially with the dimension of the search space [13], [14].

[2] Saddle points are the most prominent obstacle in gradient-based optimization since the absence of local minima is known to be at the expense of more abundant saddle points, which is why much effort has been devoted to developing strategies that make gradient-based optimization resistant to the presence of saddle points [15].

which to some degree resembles local minima or saddle point problems. However, mode collapse has its origin in approximations that are part of probabilistic modeling, especially due to calculating the expectation by Monte Carlo sampling. We have also observed that gradient descent tends to converge toward wider minima rather than toward narrower ones (see Chapter 3). This is especially important for applications to machine learning because the robustness of the solution loosely corresponds to the generalizability of the model. In this way, some overfitting problems can be resolved by optimization procedure alone rather than by (manual) model and data fine-tuning. This means that by opting for gradient-based training, we can afford to work with bigger (higher capacity) models and tailor them more to the problem at hand and less to the data quantity, which can oftentimes be a limiting factor.

When deciding on the generality of gradient-based optimization, the availability of gradient is an important factor. In this thesis, we used both symbolic (using the Theano library) and automatic differentiation (using the TensorFlow library) to calculate gradients. In both cases, we have shown that the gradient of the objective function can be obtained without providing explicit code for its calculation. Rather, the libraries automatically generate the code for gradient calculation from the code for the objective function calculation. This method makes gradient-based optimization far more accessible to the community because it does not bring any additional overhead with regard to implementation and it works regardless of the complexity of the objective function. In Chapter 5 and 6 our objective calculation even involved simulations that included conditionals, loops, complicated geometrical expressions, matrix decomposition procedures, solving of linear systems, and eigendecompositions. In our case we used Theano and TensorFlow and implemented objective functions in Python but this is far from being the only option. There are well maintained libraries that support differentiable programming for different languages (C++ [184], Fortran [185], Java [186], Python [25], [26], [29], Julia [273]) that can be applied to perform such differentiation to existing code without excessive adjustments from the user. Since differentiable programming libraries require access to the entire computational graph to generate code for the gradient, they usually include powerful compilers that, as a bonus, also apply mathematical simplifications, compute operations in place, apply numerical stability optimization, and even offer parallelized execution and acceleration (GPU and TPU) [25], [26]. Differentiable programming is, therefore, a crucial part of our methodology that place gradient-based optimization on a level of generality that is close to the level of metaheuristic optimization.

Neural network models can be encountered as a part of our methodology throughout this thesis. This is not because they would be an especially crucial part for gradient-based optimization to work but rather because this class of models offered a high level of flexibility. To transform an optimization problem to be solved by a gradient-based algorithm we require an intermediate mathematical construct that brings the two settings together and neural network allowed enough flexibility to make that happen to a level of generality we desired. Presently, neural networks are extremely well-studied class of models with countless varieties with different properties. In this way, opting for neural networks gives us the option to pick from the vast literature a very specific model that has the required properties, such as being locally linear, smooth, invertible, equivariant to a specific transformation (for example, volume-preserving), causal, with tractable Jacobian calculation, and so on. Another factor for our use of neural networks is that, for many of our use cases, they are known to be state-of-the-art models and, therefore, our model of choice (see Chapters 3, 5 and 7). Additionally, a neural network is naturally encoded by a high-dimensional real vector which puts us, by default, in a large-scale continuous

optimization setting[3] needed for gradient-based optimization. Even if our problem is low-dimensional, as can be the case in 2D shape optimization if using Möbius transformations with only 4 parameters, the use of neural networks moves us into large-scale optimization setting.

Let us now outline the basic characteristics of the methodology that was developed in the scope of this thesis. In general, our methodology for adapting optimization problems to be solved by gradient-based optimization consisting of:

1. reformulating objective function and candidate(s) into a continuous form and

2. using symbolic or automatic differentiation to acquire the gradient of this objective

in order to apply some generic gradient-based optimization algorithm. Point 1. could even be skipped in use cases from Chapter 3 because the objective function was already in an appropriate form. Reformulation of the problem into a continuous form might seem to bring, as a side effect, further restrictions and constraints on how the problem can be posed. However, in our use cases we have observed that, in fact, the opposite is true. In general, this reformulation into a continuous form brought more flexibility and further possibilities for customization. For example, in Chapter 4 we saw that reformulation of a clustering problem into a continuous form brought possibilities for clustering several data sets at the same time (data fusion), for choosing nonstandard loss functions (metrics), for co-clustering both instances and features at the same time and for handling missing values in the data set (data imputation). Likewise, in Chapter 3, we could afford model customization by embedding expert knowledge about the problem and by bringing more explainability to the models. By our claim, gradient-based optimization can be regarded as general-purpose, which means that it can handle any alterations (customizations) of the problem as long as it is continuous. This is not the case for state-of-the-art optimization algorithms used in machine learning which are designed for a specific purpose, model structure, loss function, and so on.

Looking at our methodology for reformulating objective function and candidate(s) into a continuous form, we can observe that the methods applied differ among use cases in this thesis. This is in contrast to our methodology for acquiring gradient and performing optimization, which was the same throughout the thesis (using automatic or symbolic differentiation and some type of gradient descent). Let us, therefore, address each use case separately and address our findings, especially in relation to the hypotheses posed in the introduction.

In supervised learning use case (Chapter 3) we used a gradient-based algorithm to train a custom model (without an established training algorithm) and showed that it was superior in performance compared to some other general-purpose optimization algorithms. Because our custom model was already posed in a continuous form there was no need for any additional methodology other than gradient calculation and applying a gradient-based algorithm. It is known that there is a lot of potential in using customized machine learning models to enhance their explainability and generalization. We can incorporate expert knowledge into the structure of the model, which is known to help generalization and is especially relevant when only a small amount of data is available. In this way, we have shown that gradient-based algorithms are a sound choice for a training algorithm in the setting of customized machine learning for this specific use case and consequently confirmed hypothesis **H1**. In addition, we also used gradient-based optimization to train a neural network model (that is generic) and have observed that such training is resistant

[3]In real-world applications it is difficult to construct a neural network that would have a number of parameters any smaller than in the order of 10^2 or 10^3.

to overfitting. We connected these observations with other similar findings and concluded that gradient-based optimization prefers to converge toward wide minima resulting in more robust solutions (which corresponds to better generalizability).

In unsupervised learning use case (Chapter 4) we developed a framework for using gradient-based optimization to solve problems related to clustering. There, the encoding is not naturally continuous, which requires a reformulation of the problem. This was achieved by generalizing the clustering objective function from mixed form (continuous positions of clusters and discrete quantities that convey which instance belongs to which cluster) to a fully continuous form (continuous positions of clusters and continuous quantities that measure to what degree each instance belongs to each cluster). This is a more general form of clustering called soft clustering. Because gradient-based optimization was used for training, we could afford further generalizations that allowed co-clustering, data fusion, data imputation and use of custom metrics by which we confirmed hypothesis **H2B** that addresses the flexibility of clustering models in continuous form. We found that soft clustering models can be efficiently trained by gradient-based optimization algorithms, confirming the first part of hypothesis **H2A**. Our algorithms were found to be competitive compared to other algorithms for soft clustering and, for some data sets, even superior. However, to obtain classical (hard) clustering, an orthogonality constraint needed to be imposed. Since generic gradient-based optimization does not support constraint handling, we developed a three-stage scheme that worked well for our specific use case. Our three-stage algorithm was also found to be competitive compared to other algorithms by which we have confirmed the second part of hypothesis **H2A**. In summary, the strategy behind our methodology in this work was to generalize the problem to the degree that it has a continuous form and then handle constraints needed to make the generalized version of the problem meaningful (with the same interpretation as the original problem).

In reinforcement learning use case (Chapter 5) we trained a deep reinforcement learning model without resorting to any approximative gradient calculation methods. In deep reinforcement learning, problems are usually already posed in a continuous form, however, methodology in this field does not take advantage of this fact to a full degree. Our reformulation of the deep reinforcement learning problem was to simply drop the assumption that the gradient of reward is unavailable which simplifies the way how gradient-based optimization can be applied. For problems posed in such a way, we then require a way to calculate the gradient of the reward for which we used automatic differentiation. In this way, we were able to train a reinforcement learning model for protein structure prediction by generic gradient descent without the need for Monte-Carlo sampling (as with policy gradient methods) or surrogate modeling (as with Q-learning methods). We suspected in hypothesis **H3** that our approach would, due to exact gradient calculation, eliminate convergence instabilities otherwise encountered in reinforcement learning. We cannot fully confirm **H3** because instabilities were still present with our approach, however, we were able to stabilize the descent by using gradient clipping, a relatively common technique in deep learning. Our takeaway is that instabilities in training convergence encountered in reinforcement learning are both due to noisy gradients as well as due to the complicated geometry of the reward function (in comparison to more tame loss functions encountered in (un)supervised learning). The final protein structure prediction models that we trained were not able to predict structures that are very close to optimal ones, however, we were able to show that our models can be used for initialization. We showed that differential evolution is able to find better solutions and converges faster when the initial population is partially populated by predictions from our models.

In the continuous optimization use case (Chapter 6), we developed a framework for using gradient-based optimization to solve simulation-based shape optimization problems.

The main contribution there was the development of a shape model that has a large number of parameters (large-scale optimization) and is flexible enough to capture a rich enough set of shapes needed for the problem. We used an invertible neural network for that. In this formulation, we were able to apply automatic differentiation of the objective function to calculate gradients and use gradient descent to find optimal shapes. Given that this approach of optimization was able to find better solutions and had more favorable convergence compared to a comparable derivative-free algorithm, we were able to confirm hypothesis **H4**. An important takeaway of this use case is also that automatic differentiation can, indeed, be used for very complicated objective functions meaning that gradient-based algorithms are applicable in general simulation-based optimization.

In the discrete optimization use case (Chapter 7), we reformulated a completely general discrete optimization problem into a continuous form. In contrast to our unsupervised learning use case, we did not expand the definition of a candidate (and the corresponding objective function) but rather transformed the optimization problem so that the search space now consists of distributions over candidates rather than candidates themselves. This approach provided more generality since not all discrete objective functions allow for a meaningful generalization to a continuous domain. We used a probabilistic model based on invertible neural networks to encode distribution over discrete candidates which has a natural continuous form. We used automatic differentiation to acquire gradients of objective in this new formulation and gradient descent to optimize it. We tested our algorithm on a traveling salesman problem. The experiments showed that the distribution indeed does shift towards fitter candidates and converges faster than random search by which we confirmed hypothesis **H5**. We also observed that our current algorithm converges prematurely and lacks sufficient exploration due to mode collapse, a common problem in generative modeling. The methodology deployed in this chapter still represents an important step in using gradient-based algorithms in this setting, especially since the objective function is flat almost everywhere in the search space.

In optimization parameter control use case (Chapter 8) we developed a strategy for adapting optimization parameters during optimization using gradient-based algorithms. This parameter control strategy works by changing optimization parameter values in the direction that would improve the objective value most (a derivative of the objective with respect to parameter values). For this to work we require a way to differentiate the optimization algorithm update rule as well as the objective function. We focused on parameter control of gradient-based algorithms, especially because the gradient of the objective function (required for parameter control) is already available without additional overhead. We applied this strategy to the Adam algorithm and showed that our parameter control improves convergence by which we confirmed hypothesis **H6**. Parameter adaptations observed during our experiments were aligned with deterministic state-of-the-art parameter control schedules described in the literature, with the exception that our strategy found those schedules automatically without manual intervention. This use case is the only one in the thesis where gradient descent was used in a low-dimensional search space. The main optimization algorithm is solving a large-scale optimization problem, while the optimization of parameters for the Adam algorithm, for example, is done in a space with only four parameters. However, for modern gradient-based algorithms, we expect them not to be extremely sensitive to the values of their parameters, meaning that the optimization landscape in (hyper)parameter space should be quite simple. Our experiments also give support for this assumption. It is important to note the importance of this work since it alleviates the need for parameter tuning, which is a usual (and expensive) part of the workflow that includes gradient-based optimization (especially in deep learning).

There are many directions for future work on the topic of gradient-based optimization

in a general setting. At the level of gradient-based algorithm design, which was only addressed in Chapter 8 of this thesis, there are several ways to upgrade them so that they are able to handle settings that are otherwise commonplace in real-world optimization. For example, constraint handling is not well developed in current gradient-based algorithms and penalization is practically the only choice (apart from special cases like dealing with orthogonality constraint in Chapter 4). Another common situation is multi-objective optimization, which is also not well-developed in current gradient-based algorithms and scalarization methods and hybridization with other algorithms are the only usual approaches. With regards to our methodology for reformulating problems into a continuous form there are also several directions for future work. Opportunities lie especially in discrete and mixed optimization setting. Specifically for clustering, there are ways to expand our current clustering model from a machine-learning point of view. Our clustering model in Chapter 4 uses an inner product of (nonnegative) representations to generate a prediction, however, this is not the only option. We are free to use more nonlinear binary operations instead of an inner product, possibly even a tunable one such as a machine learning model. This would possibly allow the clustering model to capture more complicated cluster structures and bring clustering closer in power to manifold learning. A fruitful future work would also be to test existing techniques for mitigating mode collapse in our formulation of discrete optimization. With regards to the differentiation of programs, one way to expand on this thesis is to more thoroughly test libraries for automatic differentiation in other languages, especially for existing programs that are possibly written in several different languages and are using MPI (message passing interface).

Even though this thesis has left several questions unanswered (and has also opened some new ones), we feel that it has provided a valuable step toward showing that gradient-based optimization is not only applicable to problems with a special mathematical form. Rather, it can be viewed as a general-purpose algorithm that can even bring superior performance in large-scale setting in comparison to derivative-free metaheuristics. We believe that much of its potential is still unrealized, especially in expensive, simulation-based, and discrete optimization. Our thesis, in this regard, offers a way to seize the advantages of gradient-based optimization in these fields.

References

[1] G. Algan and I. Ulusoy, "Image classification with deep learning in the presence of noisy labels: A survey," *Knowledge-Based Systems*, vol. 215, pp. 1–16, Article ID 106771, 2021.

[2] J. Li, "Recent advances in end-to-end automatic speech recognition," *APSIPA Transactions on Signal and Information Processing*, vol. 11, 2022.

[3] H. Wang, Y. Zhang, and X. Yu, "An overview of image caption generation methods," *Computational Intelligence and Neuroscience*, vol. 2020, pp. 1–13, Article ID 3062706, 2020.

[4] A. Ramesh, M. Pavlov, G. Goh, S. Gray, C. Voss, A. Radford, M. Chen, and I. Sutskever, "Zero-shot text-to-image generation," in *International Conference on Machine Learning*, M. Meila and T. Zhang, Eds., vol. 139, 2021, pp. 8821–8831.

[5] R. Dabre, C. Chu, and A. Kunchukuttan, "A survey of multilingual neural machine translation," *ACM Computing Surveys (CSUR)*, vol. 53, no. 5, pp. 1–38, Article no. 99, 2020.

[6] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," *arXiv preprint arXiv:1712.01815*, 2017.

[7] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[8] A. Zhavoronkov, Y. A. Ivanenkov, A. Aliper, M. S. Veselov, V. A. Aladinskiy, A. V. Aladinskaya, V. A. Terentiev, D. A. Polykovskiy, M. D. Kuznetsov, A. Asadulaev, *et al.*, "Deep learning enables rapid identification of potent DDR1 kinase inhibitors," *Nature Biotechnology*, vol. 37, no. 9, pp. 1038–1040, 2019.

[9] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[10] A. R. Conn, K. Scheinberg, and L. N. Vicente, *Introduction to Derivative-Free Optimization*. SIAM, 2009.

[11] H. H. Rosenbrock, "An automatic method for finding the greatest or least value of a function," *The Computer Journal*, vol. 3, no. 3, pp. 175–184, 1960.

[12] M. Črepinšek, S.-H. Liu, and M. Mernik, "Exploration and exploitation in evolutionary algorithms: A survey," *ACM Computing Surveys (CSUR)*, vol. 45, no. 3, pp. 1–33, Article no. 35, 2013.

[13] A. J. Bray and D. S. Dean, "Statistics of critical points of Gaussian fields on large-dimensional spaces," *Physical Review Letters*, vol. 98, no. 15, pp. 1–4, Article ID 150201, 2007.

[14] Y. V. Fyodorov and I. Williams, "Replica symmetry breaking condition exposed by random matrix calculation of landscape complexity," *Journal of Statistical Physics*, vol. 129, no. 5-6, pp. 1081–1116, 2007.

[15] Y. N. Dauphin, R. Pascanu, C. Gulcehre, K. Cho, S. Ganguli, and Y. Bengio, "Identifying and attacking the saddle point problem in high-dimensional non-convex optimization," in *Advances in Neural Information Processing Systems*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, Eds., 2014, pp. 2933–2941.

[16] N. Hansen, A. Auger, R. Ros, O. Mersmann, T. Tušar, and D. Brockhoff, "COCO: A platform for comparing continuous optimizers in a black-box setting," *Optimization Methods and Software*, vol. 36, pp. 114–144, 2021.

[17] G. Cenikj, R. D. Lang, A. P. Engelbrecht, C. Doerr, P. Korošec, and T. Eftimov, "Selector: Selecting a representative benchmark suite for reproducible statistical comparison," in *GECCO'22: Proceedings of the Genetic and Evolutionary Computation Conference*, J. E. Fieldsend, Ed., 2022, pp. 620–629.

[18] N. Hansen, R. Ros, N. Mauny, M. Schoenauer, and A. Auger, "Impacts of invariance in search: When CMA-ES and PSO face ill-conditioned and non-separable problems," *Applied Soft Computing*, vol. 11, no. 8, pp. 5755–5769, 2011.

[19] J. Larson, M. Menickelly, and S. M. Wild, "Derivative-free optimization methods," *Acta Numerica*, vol. 28, pp. 287–404, 2019.

[20] C. C. Margossian, "A review of automatic differentiation and its efficient implementation," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 9, no. 4, pp. 1–19, Article ID 1305, 2019.

[21] A. Griewank, "Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation," *Optimization Methods and Software*, vol. 1, no. 1, pp. 35–54, 1992.

[22] E. Nikishin, P. Izmailov, B. Athiwaratkun, D. Podoprikhin, T. Garipov, P. Shvechikov, D. Vetrov, and A. G. Wilson, "Improving stability in deep reinforcement learning with weight averaging," in *Uncertainty in Artificial Intelligence Workshop on Uncertainty in Deep Learning*, J. Warrell and M. B. Gerstein, Eds., 2018, pp. 1–5.

[23] M. N. Omidvar, X. Li, and K. Tang, "Designing benchmark problems for large-scale continuous optimization," *Information Sciences*, vol. 316, pp. 419–436, 2015.

[24] S. Mahdavi, M. E. Shiri, and S. Rahnamayan, "Metaheuristics in large-scale global continues optimization: A survey," *Information Sciences*, vol. 295, pp. 407–428, 2015.

[25] R. Al-Rfou *et al.*, "Theano: A Python framework for fast computation of mathematical expressions," *arXiv preprint arXiv:1605.02688*, 2016.

[26] Martín Abadi *et al.*, *TensorFlow: Large-scale machine learning on heterogeneous systems*, Software available from tensorflow.org, 2015. [Online]. Available: `https://www.tensorflow.org/`.

[27] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014.

[28] F. Seide and A. Agarwal, "CNTK: Microsoft's open-source deep-learning toolkit," in *KDD'16: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, B. Krishnapuram, M. Shah, A. Smola, C. Aggarwal, D. Shen, and R. Rastogi, Eds., ACM, 2016, pp. 2135–2135.

[29] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in PyTorch," in *NIPS'17: Proceedings of the 31st International Conference on Neural Information Processing Systems*, U. von Luxburg, I. Guyon, S. Bengio, H. Wallach, and R. Fergus, Eds., 2017.

[30] J. Salvatier, T. V. Wiecki, and C. Fonnesbeck, "Probabilistic programming in Python using PyMC3," *PeerJ Computer Science*, vol. 2, pp. 1–24, Article no. 55, 2016.

[31] D. Tran, A. Kucukelbir, A. B. Dieng, M. Rudolph, D. Liang, and D. M. Blei, "Edward: A library for probabilistic modeling, inference, and criticism," *arXiv preprint arXiv:1610.09787*, 2016.

[32] N. Siddharth, B. Paige, J.-W. Van de Meent, A. Desmaison, N. Goodman, P. Kohli, F. Wood, and P. Torr, "Learning disentangled representations with semi-supervised deep generative models," in *Advances in Neural Information Processing Systems*, 2017, pp. 5925–5935.

[33] G. R. Carmichael, A. Sandu, and F. A. Potra, "Sensitivity analysis for atmospheric chemistry models via automatic differentiation," *Atmospheric Environment*, vol. 31, no. 3, pp. 475–489, 1997.

[34] S. A. Forth and T. P. Evans, "Aerofoil optimisation via AD of a multigrid cell-vertex euler flow solver," in *Automatic Differentiation of Algorithms*, G. Corliss, C. Faure, A. Griewank, L. Hascoet, and U. Naumann, Eds., Springer, 2002, pp. 153–160.

[35] G. Haase, U. Langer, E. Lindner, and W. Mühlhuber, "Optimal sizing of industrial structural mechanics problems using AD," in *Automatic Differentiation of Algorithms*, G. Corliss, C. Faure, A. Griewank, L. Hascoet, and U. Naumann, Eds., Springer, 2002, pp. 181–188.

[36] C. H. Bischof, H. M. Bücker, and B. Lang, "Automatic differentiation for computational finance," in *Computational Methods in Decision-Making, Economics and Finance*, E. J. Kontoghiorghes, B. Rustem, and S. Siokos, Eds., Springer, 2002, pp. 297–310.

[37] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic policy gradient algorithms," in *Proceedings of the 31st International Conference on Machine Learning*, E. P. Xing and T. Jebara, Eds., 2014, pp. 387–395.

[38] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[39] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double Q-learning," *arXiv preprint arXiv:1509.06461*, 2015.

[40] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *arXiv preprint arXiv:1511.05952*, 2015.

[41] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *Proceedings of The 33rd International Conference on Machine Learning*, M. F. Balcan and K. Q. Weinberger, Eds., 2016, pp. 1928–1937.

[42] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv preprint arXiv:1502.03167*, 2015.

[43] L. Wasserman, "The role of assumptions in machine learning and statistics: Dont drink the koolaid," Technical report, Carnegie Mellon University, Tech. Rep., 2015.

[44] H. Lee, R. Gramacy, C. Linkletter, and G. Gray, "Optimization subject to hidden constraints via statistical emulation," *Pacific Journal of Optimization*, vol. 7, no. 3, pp. 467–478, 2011.

[45] W. A. Yousef, "Prudence when assuming normality: An advice for machine learning practitioners," *Pattern Recognition Letters*, vol. 138, pp. 44–50, 2020.

[46] N. Reimers and I. Gurevych, "Why comparing single performance scores does not allow to draw conclusions about machine learning approaches," *arXiv preprint arXiv: 1803.09578*, 2018.

[47] R. Torkar, C. A. Furia, R. Feldt, F. G. de Oliveira Neto, L. Gren, P. Lenberg, and N. A. Ernst, "A method to assess and argue for practical significance in software engineering," *IEEE Transactions on Software Engineering*, vol. 48, no. 6, pp. 2053–2065, 2021.

[48] R. Sun, D. Li, S. Liang, T. Ding, and R. Srikant, "The global landscape of neural networks: An overview," *IEEE Signal Processing Magazine*, vol. 37, no. 5, pp. 95–108, 2020.

[49] T. Hrga, R. Hribar, and J. Povh. "Symmetric NMTF." (2020), [Online]. Available: `https://repo.ijs.si/hribarr/symmetric-nmtf`.

[50] R. Hribar and G. Petelin. "nmfgd Library." (2019), [Online]. Available: `https://repo.ijs.si/hribarr/nmfgd`.

[51] R. Hribar. "Protein Structure Prediction with Reinforcement Learning." (2018), [Online]. Available: `https://repo.ijs.si/hribarr/protein-structure-prediction-with-reinforcement-learning.git`.

[52] ——, "FEM gradient descent." (2023), [Online]. Available: `https://repo.ijs.si/hribarr/fem-gradient-descent`.

[53] ——, "Discrete Optimization with Gradient Descent." (2025), [Online]. Available: `https://repo.ijs.si/hribarr/discrete-optimization-with-gradient-descent`.

[54] ——, "Parameter Control for Gradient Descent." (2025), [Online]. Available: `https://repo.ijs.si/hribarr/parameter-control-for-gradient-descent`.

[55] R. Hribar, P. Potočnik, J. Šilc, and G. Papa, "A comparison of models for forecasting the residential natural gas demand of an urban area," *Energy*, vol. 167, pp. 511–522, 2019.

[56] R. Hribar, T. Hrga, G. Papa, G. Petelin, J. Povh, N. Pržulj, and V. Vukašinović, "Four algorithms to solve symmetric multi-type non-negative matrix tri-factorization problem," *Journal of Global Optimization*, vol. 82, pp. 283–312, 2021.

[57] G. Papa, R. Hribar, G. Petelin, and V. Vukašinović, "Advanced computing to support urban climate neutrality," *Energy, Sustainability and Society*, vol. 15, pp. 1–16, Article no. 16, 2025.

[58] R. Hribar, G. Papa, and J. Šilc, "Prediction of natural gas consumption using empirical models," in *Proceedings of International Conference on Smart Systems and Technologies 2017*, D. Žagar, Ed., IEEE, 2017, pp. 31–36.

[59] R. Hribar, J. Šilc, and G. Papa, "Construction of heuristic for protein structure optimization using deep reinforcement learning," in *Bioinspired Optimization Methods and Their Applications*, P. Korošec, N. Melab, and E.-G. Talbi, Eds., Springer, 2018, pp. 151–162.

[60] R. Hribar, G. Petelin, J. Šilc, G. Papa, and V. Vukašinović, "Evolutionary operators in memetic algorithm for matrix tri-factorization problem," in *Proceedings of META'2018*, E.-G. Talbi, Ed., Université de Lille, 2018, pp. 426–434.

[61] A. Kastrin, R. Hribar, G. Papa, and J. Povh, "Bibliographic data clustering based on symmetric non-negative matrix tri-factorization," in *SOR'21 Proceedings: The 16th International Symposium on Operational Research in Slovenia*, S. Drobne, L. Zadnik Stirn, M. Kljajić Borštnar, J. Povh, and J. Žerovnik, Eds., Slovenian Society Informatika, 2021, pp. 194–199.

[62] R. Hribar and G. Papa, "Gradient descent as a metaheuristic: Simulation-based shape optimization use case," in *Proceedings OLA'2025: International Conference on Optimization and Learning*, E.-G. Talbi, Ed., 2025, pp. 128–139.

[63] R. Hribar, G. Petelin, J. Šilc, G. Papa, and V. Vukašinović, "Memetic algorithm and evolutionary operators for multi-objective matrix tri-factorization problem," in *Heuristics for Optimization and Learning*, F. Yalaoui, L. Amodeo, and E.-G. Talbi, Eds., Springer, 2021, pp. 286–298.

[64] M. Antoniou, R. Hribar, and G. Papa, "Parameter control in evolutionary optimisation," in *Optimization Under Uncertainty with Applications to Aerospace Engineering*, M. Vasile, Ed., Springer, 2021, pp. 357–385.

[65] R. Hribar, "Protein structure prediction using deep reinforcement learning," in *Zbornik povzetkov slovenskih delavnic "Algoritmi po vzorih iz narave" v študijskem letu 2017/2018*, B. Filipič, M. Mernik, and G. Papa, Eds., Institut "Jožef Stefan", 2018, p. 8.

[66] ——, "Evolutionary operators for memetic algorithm for solving matrix tri-factorization problem," in *Zbornik povzetkov slovenskih delavnic "Algoritmi po vzorih iz narave" v študijskem letu 2017/2018*, B. Filipič, M. Mernik, and G. Papa, Eds., (in Slovene), Institut "Jožef Stefan", 2018, p. 16.

[67] J. Povh, R. Hribar, and G. Papa, "Four algorithms to solve symmetric multi-type non-negative matrix tri-factorization problem," in *Book of Abstracts: 18th International Conference on Operational Research*, A. Čeh Časni and J. Arnerić, Eds., Croatian Operational Research Society, 2020, pp. 30–31.

[68] R. Hribar, A. Kastrin, G. Papa, J. Povh, and V. Vukašinović, "Clustering scientific bibliographic data with symmetric orthogonal matrix tri-factorization," in *Austrian-Slovenian HPC Meeting 2021*, J. Krek, Ed., University of Ljubljana, 2021, p. 36.

[69] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv preprint arXiv:1609.04747*, 2016.

[70] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning," in *International Conference on Machine Learning*, S. Dasgupta and D. McAllester, Eds., 2013, pp. 1139–1147.

[71] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization.," *Journal of Machine Learning Research*, vol. 12, no. 7, pp. 2121–2159, 2011.

[72] G. Hinton, N. Srivastava, and K. Swersky, *Neural networks for machine learning*, `https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`, 2012.

[73] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[74] H. Abbaszadehpeivasti, E. de Klerk, and M. Zamani, "The exact worst-case convergence rate of the gradient method with fixed step lengths for L-smooth functions," *Optimization Letters*, vol. 16, no. 6, pp. 1649–1661, 2022.

[75] X. Chen, S. Liu, R. Sun, and M. Hong, "On the convergence of a class of Adam-type algorithms for non-convex optimization," *arXiv preprint arXiv:1808.02941*, 2018.

[76] S. J. Reddi, S. Kale, and S. Kumar, "On the convergence of Adam and beyond," *arXiv preprint arXiv:1904.09237*, 2019.

[77] Y. Liang, M. He, J. Liu, and D. Xu, "Convergence of Adam for non-convex objectives: Relaxed hyperparameters and non-ergodic case," *Machine Learning*, vol. 114, no. 3, p. 75, 2025.

[78] N. Xiao, X. Hu, X. Liu, and K.-C. Toh, "Adam-family methods for nonsmooth optimization with convergence guarantees," *Journal of Machine Learning Research*, vol. 25, no. 48, pp. 1–53, 2024.

[79] F. H. Clarke, *Optimization and Nonsmooth Analysis*. SIAM, 1990.

[80] G. Friedl and M. Kuczmann, "Population and gradient based optimization techniques, a theoretical overview," *Acta Technica Jaurinensis*, vol. 7, no. 4, pp. 378–387, 2014.

[81] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, "Automatic differentiation in machine learning: A survey," *Journal of Marchine Learning Research*, vol. 18, pp. 1–43, 2018.

[82] T. Salimans, J. Ho, X. Chen, S. Sidor, and I. Sutskever, "Evolution strategies as a scalable alternative to reinforcement learning," *arXiv preprint arXiv:1703.03864*, 2017.

[83] U. Naumann, "Optimal Jacobian accumulation is NP-complete," *Mathematical Programming*, vol. 112, no. 2, pp. 427–441, 2008.

[84] M. Bartholomew-Biggs, S. Brown, B. Christianson, and L. Dixon, "Automatic differentiation of algorithms," *Journal of Computational and Applied Mathematics*, vol. 124, no. 1–2, pp. 171–190, 2000.

[85] S. Laue, "On the equivalence of forward mode automatic differentiation and symbolic differentiation," *arXiv preprint arXiv:1904.02990*, 2019.

[86] M. Innes, A. Edelman, K. Fischer, C. Rackauckas, E. Saba, V. B. Shah, and W. Tebbutt, "A differentiable programming system to bridge machine learning and scientific computing," *arXiv preprint arXiv:1907.07587*, 2019.

[87] F. H. Clarke, "Generalized gradients and applications," *Transactions of the American Mathematical Society*, vol. 205, pp. 247–262, 1975.

[88] D. Davis, D. Drusvyatskiy, S. Kakade, and J. D. Lee, "Stochastic subgradient method converges on tame functions," *Foundations of Computational Mathematics*, vol. 20, no. 1, pp. 119–154, 2020.

[89] B. Schölkopf, R. Herbrich, and A. J. Smola, "A generalized representer theorem," in *Computational Learning Theory*, D. Helmbold and B. Williamson, Eds., Springer, 2001, pp. 416–426.

[90] K. Hornik, "Approximation capabilities of multilayer feedforward networks," *Neural Networks*, vol. 4, no. 2, pp. 251–257, 1991.

[91] R. Rai and C. K. Sahu, "Driven by data or derived through physics? A review of hybrid physics guided machine learning techniques with cyber-physical system (CPS) focus," *IEEE Access*, vol. 8, pp. 71 050–71 073, 2020.

[92] R. R. Karri, J. Sahu, and N. Jayakumar, "Optimal isotherm parameters for phenol adsorption from aqueous solutions onto coconut shell based activated carbon: Error analysis of linear and non-linear methods," *Journal of the Taiwan Institute of Chemical Engineers*, vol. 80, pp. 472–487, 2017.

[93] Z. Jedrzejewski-Szmek, K. P. Abrahao, J. Jedrzejewska-Szmek, D. M. Lovinger, and K. T. Blackwell, "Parameter optimization using covariance matrix adaptation evolutionary strategy (CMA-ES), an approach to investigate differences in channel properties between neuron subtypes," *Frontiers in Neuroinformatics*, vol. 12, pp. 1–20, Article no. 47, 2018.

[94] F. Wang, F. Liao, Y. Li, and H. Wang, "A new prediction strategy for dynamic multi-objective optimization using Gaussian mixture model," *Information Sciences*, vol. 580, pp. 331–351, 2021.

[95] Eurostat. "Energy consumption in households." (2018), [Online]. Available: `http://ec.europa.eu/eurostat/statistics-explained/index.php?title=Energy_consumption_in_households` (visited on 05/09/2018).

[96] P. Potočnik, Faculty of Mechanical Engineering, University of Ljubljana, Personal communication, 2018.

[97] A. Mathews, M. Francisquez, J. W. Hughes, D. R. Hatch, B. Zhu, and B. N. Rogers, "Uncovering turbulent plasma dynamics via deep learning from partial observations," *Physical Review E*, vol. 104, pp. 1–11, Article ID 025205, 2021.

[98] K. Shukla, P. C. Di Leoni, J. Blackshire, D. Sparkman, and G. E. Karniadakis, "Physics-informed neural network for ultrasound nondestructive quantification of surface breaking cracks," *Journal of Nondestructive Evaluation*, vol. 39, pp. 1–20, Article no. 61, 2020.

[99] R. G. Patel, I. Manickam, N. A. Trask, M. A. Wood, M. Lee, I. Tomas, and E. C. Cyr, "Thermodynamically consistent physics-informed neural networks for hyperbolic systems," *Journal of Computational Physics*, vol. 449, pp. 1–24, Article ID 110754, 2022.

[100] D. Pfau, J. S. Spencer, A. G. D. G. Matthews, and W. M. C. Foulkes, "Ab initio solution of the many-electron Schrödinger equation with deep neural networks," *Physical Review Research*, vol. 2, pp. 1–20, Article ID 033429, 2020.

[101] W. Zhu, K. Xu, E. Darve, and G. C. Beroza, "A general approach to seismic inversion with automatic differentiation," *Computers & Geosciences*, vol. 151, pp. 1–10, Article ID 104751, 2021.

[102] M. Chromik and A. Butz, "Human-XAI interaction: A review and design principles for explanation user interfaces," in *Human-Computer Interaction–INTERACT 2021*, C. Ardito, R. Lanzilotti, A. Malizia, H. Petrie, A. Piccinno, G. Desolda, and K. Inkpen, Eds., Springer, 2021, pp. 619–640.

[103] R. Hribar, "Development of models for heat loss in buildings with aim to predict natural gas consumption in Ljubljana," (in Slovene), M.Sc. thesis, University of Ljubljana, Faculty of Mathematics and Physics, 2016. [Online]. Available: `https://repozitorij.uni-lj.si/IzpisGradiva.php?id=97521`.

[104] V. Cerqueira, L. Torgo, and C. Soares, "Machine learning vs statistical methods for time series forecasting: Size matters," *arXiv preprint arXiv:1909.13316*, 2019.

[105] A. Duval, *Explainable Artificial Intelligence (XAI)*. University of Warwick, 2019.

[106] G. F. Montufar, R. Pascanu, K. Cho, and Y. Bengio, "On the number of linear regions of deep neural networks," in *Advances in Neural Information Processing Systems 27 (NIPS 2014)*, Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Weinberger, Eds., 2014, pp. 2924–2932.

[107] Z. C. Lipton, J. Berkowitz, and C. Elkan, "A critical review of recurrent neural networks for sequence learning," *arXiv preprint arXiv:1506.00019*, 2015.

[108] N. Iyer, V. Thejas, N. Kwatra, R. Ramjee, and M. Sivathanu, "Wide-minima density hypothesis and the explore-exploit learning rate schedule," *Journal of Machine Learning Research*, vol. 24, no. 65, pp. 1–37, 2023.

[109] X. Wu, X. Liu, and Y. Zhou, "Review of unsupervised learning techniques," in *Proceedings of 2021 Chinese Intelligent Systems Conference: Volume II*, Y. Jia, W. Zhang, Y. Fu, Z. Yu, and S. Zheng, Eds., Springer, 2022, pp. 576–590.

[110] H.-T. Chiang, Y.-Y. Hsieh, S.-W. Fu, K.-H. Hung, Y. Tsao, and S.-Y. Chien, "Noise reduction in ECG signals using fully convolutional denoising autoencoders," *IEEE Access*, vol. 7, pp. 60 806–60 813, 2019.

[111] K. Pearson, "LIII. On lines and planes of closest fit to systems of points in space," *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, vol. 2, no. 11, pp. 559–572, 1901.

[112] G. Magai and A. Ayzenberg, "Topology and geometry of data manifold in deep learning," *arXiv preprint arXiv:2204.08624*, 2022.

[113] R. Patil, S. Boit, V. Gudivada, and J. Nandigam, "A survey of text representation and embedding techniques in NLP," *IEEE Access*, vol. 11, pp. 36 120–36 146, 2023.

[114] W. L. Hamilton, R. Ying, and J. Leskovec, "Representation learning on graphs: Methods and applications," *arXiv preprint arXiv:1709.05584*, 2017.

[115] A. M. Ikotun, A. E. Ezugwu, L. Abualigah, B. Abuhaija, and J. Heming, "k-means clustering algorithms: A comprehensive review, variants analysis, and advances in the era of big data," *Information Sciences*, vol. 622, pp. 178–210, 2023.

[116] V. Shikhman and D. Müller, "Clustering," in *Mathematical Foundations of Big Data Analytics*, V. Shikhman and D. Müller, Eds., Springer, 2021, pp. 87–105.

[117] T. Li and C. Ding, "The relationships among various nonnegative matrix factorization methods for clustering," in *Sixth IEEE International Conference on Data Mining (ICDM'06)*, C. W. Clifton, N. Zhong, J. Liu, B. W. Wah, and X. Wu, Eds., IEEE, 2006, pp. 362–371.

[118] M. Obiedat, A. Al-yousef, A. Khasawneh, N. Hamadneh, and A. Aljammal, "Using fuzzy c-means for weighting different fuzzy cognitive maps," *International Journal of Advanced Computer Science and Applications*, vol. 11, no. 5, pp. 545–551, 2020.

[119] C. H. Ding, T. Li, and M. I. Jordan, "Convex and semi-nonnegative matrix factorizations," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 32, no. 1, pp. 45–55, 2008.

[120] C. Ding, T. Li, W. Peng, and H. Park, "Orthogonal nonnegative matrix t-factorizations for clustering," in *KDD'06: Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, T. Eliassi-Rad, L. Ungar, M. Craven, and D. Gunopulos, Eds., 2006, pp. 126–135.

[121] I. S. Dhillon, "Co-clustering documents and words using bipartite spectral graph partitioning," in *KDD'01: Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, D. Lee, M. Schkolnick, F. Provost, and R. Srikant, Eds., 2001, pp. 269–274.

[122] C. Ding, X. He, and H. D. Simon, "On the equivalence of nonnegative matrix factorization and spectral clustering," in *Proceedings of the 2005 SIAM International Conference on Data Mining*, H. Kargupta, J. Srivastava, C. Kamath, and A. Goodman, Eds., SIAM, 2005, pp. 606–610.

[123] K. B. Petersen and M. S. Pedersen, *The Matrix Cookbook*. Technical University of Denmark, 2008.

[124] H. Zha, X. He, C. Ding, M. Gu, and H. Simon, "Spectral relaxation for k-means clustering," in *Advances in Neural Information Processing Systems 14 (NIPS 2001)*, T. Dietterich, S. Becker, and Z. Ghahramani, Eds., 2001, pp. 1057–1064.

[125] M. A. Ghazanfar and A. Prugel, "The advantage of careful imputation sources in sparse data-environment of recommender systems: Generating improved SVD-based recommendations," *Informatica*, vol. 37, pp. 61–92, 2013.

[126] Y.-D. Kim and S. Choi, "Weighted nonnegative matrix factorization," in *2009 IEEE International Conference on Acoustics, Speech and Signal Processing*, L.-s. Lee, Ed., IEEE, 2009, pp. 1541–1544.

[127] N. Malod-Dognin, J. Petschnigg, S. F. Windels, J. Povh, H. Hemingway, R. Ketteler, and N. Pržulj, "Towards a data-integrated cell," *Nature Communications*, vol. 10, pp. 1–13, Article no. 805, 2019.

[128] H. Wang, H. Huang, and C. Ding, "Simultaneous clustering of multi-type relational data via symmetric nonnegative matrix tri-factorization," in *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, B. Berendt, A. de Vries, W. Fan, C. Macdonald, I. Ounis, and I. Ruthven, Eds., 2011, pp. 279–284.

[129] D. D. Lee and H. S. Seung, "Algorithms for non-negative matrix factorization," in *Advances in Neural Information Processing Systems 13 (NIPS 2000)*, T. Leen, T. Dietterich, and V. Tresp, Eds., 2000, pp. 556–562.

[130] S. Sra and I. Dhillon, "Generalized nonnegative matrix approximations with Bregman divergences," in *Advances in Neural Information Processing Systems 18 (NIPS 2005)*, Y. Weiss, B. Schölkopf, and J. Platt, Eds., 2005, pp. 283–290.

[131] T. Zhang, B. Fang, W. Liu, Y. Y. Tang, G. He, and J. Wen, "Total variation norm-based nonnegative matrix factorization for identifying discriminant representation of image patterns," *Neurocomputing*, vol. 71, no. 10, pp. 1824–1831, 2008.

[132] M. Žitnik and B. Zupan, "Data fusion by matrix factorization," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 37, no. 1, pp. 41–53, 2014.

[133] R. Zdunek and A. Cichocki, "Fast nonnegative matrix factorization algorithms using projected gradient approaches for large-scale problems," *Computational Intelligence and Neuroscience*, vol. 2008, pp. 1–13, Article ID 939567, 2008.

[134] Z. Liang, Y. Li, and T. Zhao, "Projected gradient method for kernel discriminant nonnegative matrix factorization and the applications," *Signal Processing*, vol. 90, no. 7, pp. 2150–2163, 2010.

[135] L. Lin and Z.-Y. Liu, "An alternating projected gradient algorithm for nonnegative matrix factorization," *Applied Mathematics and Computation*, vol. 217, no. 24, pp. 9997–10 002, 2011.

[136] G. R. Atwood and W. W. Foster, "Transformation of bounded variables in simplex optimization techniques," *Industrial & Engineering Chemistry Process Design and Development*, vol. 12, no. 4, pp. 485–486, 1973.

[137] S. Tsutsui, "Multi-parent recombination in genetic algorithms with search space boundary extension by mirroring," in *Parallel Problem Solving from Nature, PPSN V*, A. E. Eiben, T. Bäck, M. Schoenauer, and H.-P. Schwefel, Eds., Springer, 1998, pp. 428–437.

[138] H. Someya and M. Yamamura, "A robust real-coded evolutionary algorithm with toroidal search space conversion," *Soft Computing*, vol. 9, no. 4, pp. 254–269, 2005.

[139] Y.-H. Dai and C.-X. Kou, "A nonlinear conjugate gradient algorithm with an optimal property and an improved Wolfe line search," *SIAM Journal on Optimization*, vol. 23, no. 1, pp. 296–320, 2013.

[140] Y. Wang, "Gauss–Newton method," *Wiley Interdisciplinary Reviews: Computational Statistics*, vol. 4, no. 4, pp. 415–420, 2012.

[141] L. Chen and Y. Ma, "Shamanskii-like Levenberg-Marquardt method with a new line search for systems of nonlinear equations," *Journal of Systems Science and Complexity*, vol. 33, no. 5, pp. 1694–1707, 2020.

[142] F. Schleif and A. Gisbrecht, "Data analysis of (non-)metric proximities at linear costs," in *Similarity-Based Pattern Recognition*, E. R. Hancock and M. Pelillo, Eds., Springer, 2013, pp. 59–74.

[143] T. S. Madhulatha, "An overview on clustering methods," *arXiv preprint arXiv: 1205.1117*, 2012.

[144] C. Boutsidis and E. Gallopoulos, "SVD based initialization: A head start for nonnegative matrix factorization," *Pattern Recognition*, vol. 41, no. 4, pp. 1350–1362, 2008.

[145] J. Branke, K. Deb, K. Miettinen, and R. Slowinski, *Multiobjective Optimization: Interactive and Evolutionary Approaches*. Springer, 2008.

[146] N. Nedjah and L. de Macedo Mourelle, *Real-World Multi-Objective System Engineering*. Nova Publishers, 2005.

[147] S. Sharma and V. Kumar, "A comprehensive review on multi-objective optimization techniques: Past, present and future," *Archives of Computational Methods in Engineering*, vol. 29, no. 7, pp. 5605–5633, 2022.

[148] Y. Tian, L. Si, X. Zhang, R. Cheng, C. He, K. C. Tan, and Y. Jin, "Evolutionary large-scale multi-objective optimization: A survey," *ACM Computing Surveys (CSUR)*, vol. 54, no. 8, pp. 1–34, Article no. 174, 2021.

[149] M. Mitchell and J. H. Holland, "When will a genetic algorithm outperform hill climbing?" In *Advances in Neural Information Processing Systems 6 (NIPS 1993)*, J. Cowan, G. Tesauro, and J. Alspector, Eds., 1993, pp. 51–58.

[150] F. Neri and C. Cotta, "Memetic algorithms and memetic computing optimization: A literature review," *Swarm and Evolutionary Computation*, vol. 2, pp. 1–14, 2012.

[151] D. M. Saputra, D. Saputra, and L. D. Oswari, "Effect of distance metrics in determining k-value in k-means clustering using elbow and silhouette method," in *Proceedings of the Sriwijaya International Conference on Information Technology and Its Applications (SICONIAN 2019)*, T. Sutikno, D. Stiawan, and H. Satria, Eds., Atlantis Press, 2020, pp. 341–346.

[152] M. B. Ferraro and P. Giordani, "Soft clustering," *Wiley Interdisciplinary Reviews: Computational Statistics*, vol. 12, pp. 1–12, Article no. 1480, 2020.

[153] I. M. James, *The Topology of Stiefel Manifolds*. Cambridge University Press, 1976.

[154] S. Peng, W. Ser, B. Chen, and Z. Lin, "Robust orthogonal nonnegative matrix tri-factorization for data representation," *Knowledge-Based Systems*, vol. 201–202, pp. 1–14, Article ID 106054, 2020.

[155] M. Yang and S. Xu, "Orthogonal nonnegative matrix factorization using a novel deep autoencoder network," *Knowledge-Based Systems*, vol. 227, pp. 1–20, Article ID 107236, 2021.

[156] A. Rahiche and M. Cheriet, "Forgery detection in hyperspectral document images using graph orthogonal nonnegative matrix factorization," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, 2020, pp. 662–663.

[157] W. Li, J. Li, X. Liu, and L. Dong, "Two fast vector-wise update algorithms for orthogonal nonnegative matrix factorization with sparsity constraint," *Journal of Computational and Applied Mathematics*, vol. 375, pp. 1–21, Article ID 112785, 2020.

[158] S. Asadi and J. Povh, "A block coordinate descent-based projected gradient algorithm for orthogonal non-negative matrix factorization," *Mathematics*, vol. 9, no. 5, pp. 1–22, Article no. 540, 2021.

[159] Z. Zhang, Y. Xu, J. Yang, X. Li, and D. Zhang, "A survey of sparse representation: Algorithms and applications," *IEEE Access*, vol. 3, pp. 490–530, 2015.

[160] F. Chen, Y.-C. Wang, B. Wang, and C.-C. J. Kuo, "Graph representation learning: A survey," *APSIPA Transactions on Signal and Information Processing*, vol. 9, pp. 1–21, Article no. 13, 2020.

[161] A. E. Ezugwu, A. M. Ikotun, O. O. Oyelade, L. Abualigah, J. O. Agushaka, C. I. Eke, and A. A. Akinyelu, "A comprehensive survey of clustering algorithms: State-of-the-art machine learning applications, taxonomy, challenges, and future research prospects," *Engineering Applications of Artificial Intelligence*, vol. 110, pp. 1–43, Article ID 104743, 2022.

[162] Y. Ouali, C. Hudelot, and M. Tami, "An overview of deep semi-supervised learning," *arXiv preprint arXiv:2006.05278*, 2020.

[163] J. Fan, Z. Wang, Y. Xie, and Z. Yang, "A theoretical analysis of deep Q-learning," in *Learning for Dynamics and Control*, A. M. Bayen, A. Jadbabaie, G. Pappas, P. A. Parrilo, B. Recht, C. Tomlin, and M. Zeilinger, Eds., 2020, pp. 486–489.

[164] A. Telikani, A. Tahmassebi, W. Banzhaf, and A. H. Gandomi, "Evolutionary machine learning: A survey," *ACM Computing Surveys (CSUR)*, vol. 54, no. 8, pp. 1–35, Article no. 161, 2021.

[165] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. MIT press, 2018.

[166] B. B. Elallid, N. Benamar, A. S. Hafid, T. Rachidi, and N. Mrani, "A comprehensive survey on the application of deep and reinforcement learning approaches in autonomous driving," *Journal of King Saud University-Computer and Information Sciences*, vol. 34, no. 9, pp. 7366–7390, 2022.

[167] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, *et al.*, "Mastering the game of Go without human knowledge," *Nature*, vol. 550, pp. 354–359, 2017.

[168] X. B. Peng, P. Abbeel, S. Levine, and M. Van de Panne, "Deepmimic: Example-guided deep reinforcement learning of physics-based character skills," *ACM Transactions On Graphics (TOG)*, vol. 37, no. 4, pp. 1–14, Article no. 143, 2018.

[169] J. Degrave, F. Felici, J. Buchli, M. Neunert, B. Tracey, F. Carpanese, T. Ewalds, R. Hafner, A. Abdolmaleki, D. de Las Casas, *et al.*, "Magnetic control of tokamak plasmas through deep reinforcement learning," *Nature*, vol. 602, pp. 414–419, 2022.

[170] X. Wang, S. Wang, X. Liang, D. Zhao, J. Huang, X. Xu, B. Dai, and Q. Miao, "Deep reinforcement learning: A survey," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 35, no. 4, pp. 5064–5078, 2024.

[171] C. Daskalakis, D. J. Foster, and N. Golowich, "Independent policy gradient methods for competitive reinforcement learning," *Advances in Neural Information Processing Systems*, vol. 33, pp. 5527–5540, 2020.

[172] W. B. Powell, "From reinforcement learning to optimal control: A unified framework for sequential decisions," in *Handbook of Reinforcement Learning and Control*, K. G. Vamvoudakis, Y. Wan, F. L. Lewis, and D. Cansever, Eds., Springer, 2021, pp. 29–74.

[173] A. Boulesnane and S. Meshoul, "Reinforcement learning for dynamic optimization problems," in *GECCO'21: Proceedings of the Genetic and Evolutionary Computation Conference Companion*, F. Chicano, Ed., 2021, pp. 201–202.

[174] J. T. Hancock and T. M. Khoshgoftaar, "Survey on categorical data for neural networks," *Journal of Big Data*, vol. 7, pp. 1–41, Article no. 28, 2020.

[175] S. Xie, Z. Zhang, H. Yu, and X. Luo, "Recurrent prediction model for partially observable MDPs," *Information Sciences*, vol. 620, pp. 125–141, 2023.

[176] T. Vidal, T. G. Crainic, M. Gendreau, and C. Prins, "Heuristics for multi-attribute vehicle routing problems: A survey and synthesis," *European Journal of Operational Research*, vol. 231, no. 1, pp. 1–21, 2013.

[177] J. Branke, S. Nguyen, C. W. Pickardt, and M. Zhang, "Automated design of production scheduling heuristics: A review," *IEEE Transactions on Evolutionary Computation*, vol. 20, no. 1, pp. 110–124, 2016.

[178] A. Perez, J. MacCallum, and K. A. Dill, "Using physics and heuristics in protein structure prediction," *Biophysical Journal*, vol. 108, no. 2, 210a, 2015.

[179] L. Liu, S. Lu, R. Zhong, B. Wu, Y. Yao, Q. Zhang, and W. Shi, "Computing systems for autonomous driving: State of the art and challenges," *IEEE Internet of Things Journal*, vol. 8, no. 8, pp. 6469–6486, 2020.

[180] P. Korošec, U. Bole, and G. Papa, "A multi-objective approach to the application of real-world production scheduling," *Expert Systems with Applications*, vol. 40, no. 15, pp. 5839–5853, 2013.

[181] S. Ganscha, V. Fortuin, M. Horn, E. Arvaniti, and M. Claassen, "Supervised learning on synthetic data for reverse engineering gene regulatory networks from experimental time-series," *bioRxiv preprint*, 2018. DOI: `https://doi.org/10.1101/356477`.

[182] W. Wang, Z. Peng, and J. Yang, "Single-sequence protein structure prediction using supervised transformer protein language models," *Nature Computational Science*, vol. 2, no. 12, pp. 804–814, 2022.

[183] A. C. Ling, U. Aydonat, S. O'Connell, D. Capalija, and G. R. Chiu, "Creating high performance applications with Intel's FPGA OpenCL SDK," in *IWOCL'17: Proceedings of the 5th International Workshop on OpenCL*, ACM, 2017, p. 1, Article no. 11.

[184] R. J. Hogan, "Fast reverse-mode automatic differentiation using expression templates in C++," *ACM Transactions on Mathematical Software (TOMS)*, vol. 40, no. 4, pp. 1–16, Article no. 26, 2014.

[185] S. Stamatiadis and S. C. Farantos, "AUTO_DERIV: Tool for automatic differentiation of a FORTRAN code," *Computer Physics Communications*, vol. 181, pp. 1818–1819, 2010.

[186] E.-I. Slusanschi, "Algorithmic differentiation of Java programs," Ph.D. dissertation, Rheinisch-Westfälische Technische Hochschule Aachen, 2008.

[187] B. Kuhlman and P. Bradley, "Advances in protein structure prediction and design," *Nature Reviews Molecular Cell Biology*, vol. 20, no. 11, pp. 681–697, 2019.

[188] T. Siebenmorgen and M. Zacharias, "Evaluation of predicted protein–protein complexes by binding free energy simulations," *Journal of Chemical Theory and Computation*, vol. 15, no. 3, pp. 2071–2086, 2019.

[189] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, Y. W. Teh and M. Titterington, Eds., vol. 9, 2010, pp. 249–256.

[190] B. Bošković and J. Brest, "Differential evolution for protein folding optimization based on a three-dimensional AB off-lattice model," *Journal of Molecular Modeling*, vol. 22, no. 10, pp. 1–15, Article no. 252, 2016.

[191] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann. "Implementation of off-policy maximum entropy deep reinforcement learning with a stochastic actor." (2023), [Online]. Available: `https://github.com/DLR-RM/stable-baselines3/blob/master/stable_baselines3/sac/sac.py` (visited on 09/07/2024).

[192] N. Andrei, *Continuous Nonlinear Optimization for Engineering Applications in GAMS Technology*. Springer, 2017.

[193] Y. Akimoto, Y. Nagata, I. Ono, and S. Kobayashi, "Bidirectional relation between CMA evolution strategies and natural evolution strategies," in *Parallel Problem Solving from Nature, PPSN XI*, R. Schaefer, C. Cotta, J. Kolodziej, and G. Rudolph, Eds., Springer, 2010, pp. 154–163.

[194] N. Hansen and A. Ostermeier, "Completely derandomized self-adaptation in evolution strategies," *Evolutionary Computation*, vol. 9, no. 2, pp. 159–195, 2001.

[195] R. Ros and N. Hansen, "A simple modification in CMA-ES achieving linear time and space complexity," in *International Conference on Parallel Problem Solving from Nature, PPSN X*, G. Rudolph, T. Jansen, N. Beume, S. Lucas, and C. Poloni, Eds., Springer, 2008, pp. 296–305.

[196] Y. Akimoto and N. Hansen, "Projection-based restricted covariance matrix adaptation for high dimension," in *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, 2016, pp. 197–204.

[197] I. Loshchilov, "LM-CMA: An alternative to L-BFGS for large-scale black box optimization," *Evolutionary Computation*, vol. 25, no. 1, pp. 143–171, 2017.

[198] Z. Li and Q. Zhang, "A simple yet efficient evolution strategy for large-scale black-box optimization," *IEEE Transactions on Evolutionary Computation*, vol. 22, no. 5, pp. 637–646, 2017.

[199] K. Varelas, A. Auger, D. Brockhoff, N. Hansen, O. A. ElHara, Y. Semet, R. Kassab, and F. Barbaresco, "A comparative study of large-scale variants of CMA-ES," in *Parallel Problem Solving from Nature, PPSN XV*, A. Auger, C. M. Fonseca, N. Lourenco, P. Machado, L. Paquete, and D. Whitley, Eds., Springer, 2018, pp. 3–15.

[200] J. Hestness, S. Narang, N. Ardalani, G. Diamos, H. Jun, H. Kianinejad, M. M. A. Patwary, Y. Yang, and Y. Zhou, "Deep learning scaling is predictable, empirically," *arXiv preprint arXiv:1712.00409*, 2017.

[201] Y. You, Z. Zhang, C. Hsieh, J. Demmel, and K. Keutzer, "100-epoch imagenet training with alexnet in 24 minutes," *arXiv preprint arXiv:1709.05011*, 2017.

[202] A. Komatsuzaki, "One epoch is all you need," *arXiv preprint arXiv:1906.06669*, 2019.

[203] K. Varelas, O. A. El Hara, D. Brockhoff, N. Hansen, D. M. Nguyen, T. Tušar, and A. Auger, "Benchmarking large-scale continuous optimizers: The BBOB-largescale testbed, a COCO software guide and beyond," *Applied Soft Computing*, vol. 97, pp. 1–13, Article no. 106737, 2020.

[204] N. Hansen, A. Auger, R. Ros, O. Mersmann, T. Tušar, and D. Brockhoff, *COCO postprocessed data archive*. [Online]. Available: `https://numbbo.github.io/ppdata-archive` (visited on 08/03/2023).

[205] J. Gao, M. Xiao, Y. Zhang, and L. Gao, "A comprehensive review of isogeometric topology optimization: Methods, applications and prospects," *Chinese Journal of Mechanical Engineering*, vol. 33, pp. 1–14, Article no. 87, 2020.

[206] H. Arata, M. Kishida, and T. Kurahashi, "Texture shape optimization analysis using a new acceleration gradient method based on the Taylor expansion and conjugate direction," *Journal of Fluid Science and Technology*, vol. 17, no. 4, pp. 1–13, Article no. 359, 2022.

[207] L. Chen, H. Lian, Z. Liu, H. Chen, E. Atroshchenko, and S. Bordas, "Structural shape optimization of three dimensional acoustic problems with isogeometric boundary element methods," *Computer Methods in Applied Mechanics and Engineering*, vol. 355, pp. 926–951, 2019.

[208] M.-P. Schmidt, P. Clausen, C. B. Pedersen, and P. Hebrard, "Semi-analytical gradient-based optimization of exact CAD models using intermediate field representations," *Structural and Multidisciplinary Optimization*, vol. 66, no. 6, pp. 1–18, Article no. 138, 2023.

[209] O. T. Doganay, H. Gottschalk, C. Hahn, K. Klamroth, J. Schultes, and M. Stiglmayr, "Gradient based biobjective shape optimization to improve reliability and cost of ceramic components," *Optimization and Engineering*, vol. 21, pp. 1359–1387, 2020.

[210] J. Viquerat, J. Rabault, A. Kuhnle, H. Ghraieb, A. Larcher, and E. Hachem, "Direct shape optimization through deep reinforcement learning," *Journal of Computational Physics*, vol. 428, pp. 1–12, Article ID 110080, 2021.

[211] J. Li and J. Cai, "Massively multipoint aerodynamic shape design via surrogate-assisted gradient-based optimization," *AIAA Journal*, vol. 58, no. 5, pp. 1949–1963, 2020.

[212] L. Jofre and A. Doostan, "Rapid aerodynamic shape optimization under parametric and turbulence model uncertainty: A stochastic gradient approach," *arXiv preprint arXiv:2105.01048*, 2021.

[213] H. Zhonghua, X. Chenzhou, L. Zhang, Y. Zhang, K. Zhang, and S. Wenping, "Efficient aerodynamic shape optimization using variable-fidelity surrogate models and multilevel computational grids," *Chinese Journal of Aeronautics*, vol. 33, no. 1, pp. 31–47, 2020.

[214] H. Zhu, J. Sun, H. Guo, D. Xu, and G. Cai, "Research on aerodynamic shape optimization of reentry vehicle based on hybrid scale multi-fidelity neural network model," *Aerospace Science and Technology*, pp. 1–17, Article ID 108619, 2023.

[215] X. Zhang, F. Xie, T. Ji, Z. Zhu, and Y. Zheng, "Multi-fidelity deep neural network surrogate model for aerodynamic shape optimization," *Computer Methods in Applied Mechanics and Engineering*, vol. 373, pp. 1–25, Article ID 113485, 2021.

[216] B. S. Yıldız, A. R. Yıldız, E. İ. Albak, H. Abderazek, S. M. Sait, and S. Bureerat, "Butterfly optimization algorithm for optimum shape design of automobile suspension components," *Materials Testing*, vol. 62, no. 4, pp. 365–370, 2020.

[217] B. S. Yıldız, N. Pholdee, S. Bureerat, A. R. Yıldız, and S. M. Sait, "Sine-cosine optimization algorithm for the conceptual design of automobile components," *Materials Testing*, vol. 62, no. 7, pp. 744–748, 2020.

[218] P.-I. Schneider, X. Garcia Santiago, V. Soltwisch, M. Hammerschmidt, S. Burger, and C. Rockstuhl, "Benchmarking five global optimization approaches for nano-optical shape optimization and parameter reconstruction," *ACS Photonics*, vol. 6, no. 11, pp. 2726–2733, 2019.

[219] G. Hu, L. Chen, and G. Wei, "Enhanced golden jackal optimizer-based shape optimization of complex CSGC-Ball surfaces," *Artificial Intelligence Review*, vol. 56, pp. 2407–2475, 2023.

[220] L. D. Landau and E. M. Lifshitz, *Fluid Mechanics: Volume 6 of Course of Theoretical Physics*. Pergamon Press, 1987.

[221] L. D. Landau, E. M. Lifshitz, A. M. Kosevich, and L. P. Pitaevskii, *Theory of Elasticity: Volume 7 of Course of Theoretical Physics*. Elsevier, 1986.

[222] B. N. Parlett, *The Symmetric Eigenvalue Problem*. SIAM, 1998.

[223] M. Qian and S. Zhu, "A level set method for Laplacian eigenvalue optimization subject to geometric constraints," *Computational Optimization and Applications*, vol. 82, no. 2, pp. 499–524, 2022.

[224] P. R. Antunes and P. Freitas, "Optimisation of eigenvalues of the Dirichlet Laplacian with a surface area restriction," *Applied Mathematics & Optimization*, vol. 73, no. 2, pp. 313–328, 2016.

[225] A. Berger, "The eigenvalues of the Laplacian with Dirichlet boundary condition in $\mathbb{R}^2$ are almost never minimized by disks," *Annals of Global Analysis and Geometry*, vol. 47, no. 3, pp. 285–304, 2015.

[226] P. R. Antunes and P. Freitas, "Numerical optimization of low eigenvalues of the Dirichlet and Neumann Laplacians," *Journal of Optimization Theory and Applications*, vol. 154, pp. 235–257, 2012.

[227] N. D. Manh, A. Evgrafov, A. R. Gersborg, and J. Gravesen, "Isogeometric shape optimization of vibrating membranes," *Computer Methods in Applied Mechanics and Engineering*, vol. 200, no. 13-16, pp. 1343–1353, 2011.

[228] Z. Belhachmi, D. Bucur, G. Buttazzo, and J.-M. Sac-Epée, "Shape optimization problems for eigenvalues of elliptic operators," *ZAMM–Journal of Applied Mathematics and Mechanics*, vol. 86, no. 3, pp. 171–184, 2006.

[229] É. Oudet, "Numerical minimization of eigenmodes of a membrane with respect to the domain," *ESAIM: Control, Optimisation and Calculus of Variations*, vol. 10, no. 3, pp. 315–330, 2004.

[230] T. J. Hughes, J. A. Cottrell, and Y. Bazilevs, "Isogeometric analysis: CAD, finite elements, NURBS, exact geometry and mesh refinement," *Computer Methods in Applied Mechanics and Engineering*, vol. 194, no. 39-41, pp. 4135–4195, 2005.

[231] I. Ishikawa, T. Teshima, K. Tojo, K. Oono, M. Ikeda, and M. Sugiyama, "Universal approximation property of invertible neural networks," *Journal of Machine Learning Research*, vol. 24, no. 287, pp. 1–68, 2023.

[232] G. Papamakarios, E. Nalisnick, D. J. Rezende, S. Mohamed, and B. Lakshminarayanan, "Normalizing flows for probabilistic modeling and inference," *Journal of Machine Learning Research*, vol. 22, no. 57, pp. 1–64, 2021.

[233] Q. Duan, G. Zhou, C. Shao, Z. Wang, M. Feng, Y. Yang, Q. Zhao, and Y. Shi, "PyPop7: A pure-python library for population-based black-box optimization," *arXiv preprint arXiv:2212.05652*, 2022.

[234] I. Loshchilov, "A computationally efficient limited memory CMA-ES for large scale optimization," in *GECCO'14: Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, C. Igel, Ed., 2014, pp. 397–404.

[235] Y. Nesterov and V. Spokoiny, "Random gradient-free minimization of convex functions," *Foundations of Computational Mathematics*, vol. 17, no. 2, pp. 527–566, 2017.

[236] E.-G. Talbi, *Metaheuristics: From Design to Implementation*. John Wiley & Sons, 2009.

[237] M. B. Correia, "A study of redundancy and neutrality in evolutionary optimization," *Evolutionary Computation*, vol. 21, no. 3, pp. 413–443, 2013.

[238] S. Kirkpatrick, C. D. Gelatt Jr, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[239] F. Glover and M. Laguna, *Tabu Search*. Springer, 1998.

[240] C. Voudouris and E. P. Tsang, "Guided local search joins the elite in discrete optimisation," in *Constraint Programming and Large Scale Discrete Optimization*, E. C. Freuder and R. J. Wallace, Eds., 1998, pp. 29–39.

[241] C. Blum and A. Roli, "Metaheuristics in combinatorial optimization: Overview and conceptual comparison," *ACM Computing Surveys (CSUR)*, vol. 35, no. 3, pp. 268–308, 2003.

[242] Q. Wu and J.-K. Hao, "A review on algorithms for maximum clique problems," *European Journal of Operational Research*, vol. 242, no. 3, pp. 693–709, 2015.

[243] M. Harman and P. McMinn, "A theoretical and empirical study of search-based testing: Local, global, and hybrid search," *IEEE Transactions on Software Engineering*, vol. 36, no. 2, pp. 226–247, 2009.

[244] J. H. Holland, *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. MIT Press, 1992.

[245] M. Dorigo and G. Di Caro, "Ant colony optimization: A new meta-heuristic," in *Proceedings of the 1999 Congress on Evolutionary Computation, CEC99*, IEEE, vol. 2, 1999, pp. 1470–1477.

[246] G. Pavai and T. Geetha, "A survey on crossover operators," *ACM Computing Surveys (CSUR)*, vol. 49, no. 4, pp. 1–43, Article no. 72, 2016.

[247] O. Cheikhrouhou and I. Khoufi, "A comprehensive survey on the multiple traveling salesman problem: Applications, approaches and taxonomy," *Computer Science Review*, vol. 40, pp. 1–19, Article ID 100369, 2021.

[248] C. Basnet, "Heuristics for the multiple knapsack problem with conflicts," *International Journal of Operational Research*, vol. 32, no. 4, pp. 514–525, 2018.

[249] C. P. Gomes, H. Kautz, A. Sabharwal, and B. Selman, "Satisfiability solvers," *Foundations of Artificial Intelligence*, vol. 3, pp. 89–134, 2008.

[250] M. Hauschild and M. Pelikan, "An introduction and survey of estimation of distribution algorithms," *Swarm and Evolutionary Computation*, vol. 1, no. 3, pp. 111–128, 2011.

[251] S. Bond-Taylor, A. Leach, Y. Long, and C. G. Willcocks, "Deep generative modelling: A comparative review of VAEs, GANs, normalizing flows, energy-based and autoregressive models," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 44, no. 11, pp. 7327–7347, 2021.

[252] D. Saxena and J. Cao, "Generative adversarial networks (GANs) challenges, solutions, and future directions," *ACM Computing Surveys (CSUR)*, vol. 54, no. 3, pp. 1–42, Article no. 63, 2021.

[253] W. Ping, K. Peng, K. Zhao, and Z. Song, "Waveflow: A compact flow-based model for raw audio," in *International Conference on Machine Learning*, H. Daumé III and A. Singh, Eds., 2020, pp. 7706–7716.

[254] D. P. Kingma and P. Dhariwal, "Glow: Generative flow with invertible 1x1 convolutions," in *Advances in Neural Information Processing Systems 31 (NeurIPS 2018)*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., 2018, pp. 10 215–10 224.

[255] M. Kumar, M. Babaeizadeh, D. Erhan, C. Finn, S. Levine, L. Dinh, and D. Kingma, "Videoflow: A conditional flow-based model for stochastic video generation," *arXiv preprint arXiv:1903.01434*, 2019.

[256] C. Shi, M. Xu, Z. Zhu, W. Zhang, M. Zhang, and J. Tang, "Graphaf: A flow-based autoregressive model for molecular graph generation," *arXiv preprint arXiv: 2001.09382*, 2020.

[257] L. Dinh, J. Sohl-Dickstein, and S. Bengio, "Density estimation using Real NVP," *arXiv preprint arXiv:1605.08803*, 2016.

[258] H. J. Barbosa, A. C. Lemonge, and C. C. Borges, "A genetic algorithm encoding for cardinality constraints and automatic variable linking in structural optimization," *Engineering Structures*, vol. 30, no. 12, pp. 3708–3723, 2008.

[259] S. Mohamed, M. Rosca, M. Figurnov, and A. Mnih, "Monte Carlo gradient estimation in machine learning," *Journal of Machine Learning Research*, vol. 21, no. 132, pp. 1–62, 2020.

[260] G. Reinelt, "TSPLIB–A traveling salesman problem library," *ORSA Journal on Computing*, vol. 3, no. 4, pp. 376–384, 1991.

[261] A. H. Ismail, N. Hartono, S. Zeybek, and D. T. Pham, "Using the bees algorithm to solve combinatorial optimisation problems for TSPLIB," in *IOP Conference Series: Materials Science and Engineering*, IOP Publishing, vol. 847, 2020, pp. 1–9, Article ID 012027.

[262] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," *arXiv preprint arXiv:1711.05101*, 2017.

[263] L. N. Smith, "Cyclical learning rates for training neural networks," in *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*, L. O'Conner, Ed., IEEE, 2017, pp. 464–472.

[264] S. Tomar and A. Gupta, "A review on mode collapse reducing GANs with GAN's algorithm and theory," in *GANs for Data Augmentation in Healthcare*, A. Solanki and M. Naved, Eds., Springer, 2023, pp. 21–40.

[265] K. A. Nicoli, C. Anders, L. Funcke, T. Hartung, K. Jansen, P. Kessel, S. Nakajima, and P. Stornati, "Machine learning of thermodynamic observables in the presence of mode collapse," *arXiv preprint arXiv:2111.11303*, 2021.

[266] K. A. Nicoli, C. J. Anders, T. Hartung, K. Jansen, P. Kessel, and S. Nakajima, "Detecting and mitigating mode-collapse for flow-based sampling of lattice field theories," *Physical Review D*, vol. 108, no. 11, pp. 1–17, Article ID 114501, 2023.

[267] Y. Wu, L. Liu, J. Bae, K.-H. Chow, A. Iyengar, C. Pu, W. Wei, L. Yu, and Q. Zhang, "Demystifying learning rate policies for high accuracy training of deep neural networks," in *2019 IEEE International Conference on Big Data (Big Data)*, C. Baru, J. Huan, L. Khan, X. Hu, R. Ak, Y. Tian, R. Barga, C. Zaniolo, K. Lee, and Y. F. Ye, Eds., IEEE, 2019, pp. 1971–1980.

[268] Y. LeCun and C. Cortes. "MNIST handwritten digit database." (2010), [Online]. Available: `http://yann.lecun.com/exdb/mnist/` (visited on 06/09/2023).

[269] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized BERT pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.

[270] M. Zaheer, S. Reddi, D. Sachan, S. Kale, and S. Kumar, "Adaptive methods for nonconvex optimization," *Advances in Neural Information Processing Systems 31 (NeurIPS 2018)*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., pp. 9793–9803, 2018.

[271] L. Liu, H. Jiang, P. He, W. Chen, X. Liu, J. Gao, and J. Han, "On the variance of the adaptive learning rate and beyond," *arXiv preprint arXiv:1908.03265*, 2019.

[272] Y. Hu, L. Lin, and S. Tang, "Second-order information in first-order optimization methods," *arXiv preprint arXiv:1912.09926*, 2019.

[273] M. Innes, "Don't unroll adjoint: Differentiating SSA-form programs," *arXiv preprint arXiv:1810.07951*, 2018.

Bibliography

Publications Related to the Thesis

Original Scientific Article

R. Hribar, T. Hrga, G. Papa, G. Petelin, J. Povh, N. Pržulj, and V. Vukašinović, "Four algorithms to solve symmetric multi-type non-negative matrix tri-factorization problem," *Journal of Global Optimization*, vol. 82, pp. 283–312, 2021.

R. Hribar, P. Potočnik, J. Šilc, and G. Papa, "A comparison of models for forecasting the residential natural gas demand of an urban area," *Energy*, vol. 167, pp. 511–522, 2019.

G. Papa, R. Hribar, G. Petelin, and V. Vukašinović, "Advanced computing to support urban climate neutrality," *Energy, Sustainability and Society*, vol. 15, pp. 1–16, Article no. 16, 2025.

Published Scientific Conference Contribution

A. Kastrin, R. Hribar, G. Papa, and J. Povh, "Bibliographic data clustering based on symmetric non-negative matrix tri-factorization," in *SOR'21 Proceedings: The 16th International Symposium on Operational Research in Slovenia*, S. Drobne, L. Zadnik Stirn, M. Kljajić Borštnar, J. Povh, and J. Žerovnik, Eds., Slovenian Society Informatika, 2021, pp. 194–199.

R. Hribar, J. Šilc, and G. Papa, "Construction of heuristic for protein structure optimization using deep reinforcement learning," in *Bioinspired Optimization Methods and Their Applications*, P. Korošec, N. Melab, and E.-G. Talbi, Eds., Springer, 2018, pp. 151–162.

R. Hribar, G. Petelin, J. Šilc, G. Papa, and V. Vukašinović, "Evolutionary operators in memetic algorithm for matrix tri-factorization problem," in *Proceedings of META'2018*, E.-G. Talbi, Ed., Université de Lille, 2018, pp. 426–434.

R. Hribar, G. Papa, and J. Šilc, "Prediction of natural gas consumption using empirical models," in *Proceedings of International Conference on Smart Systems and Technologies 2017*, D. Žagar, Ed., IEEE, 2017, pp. 31–36.

R. Hribar and G. Papa, "Gradient descent as a metaheuristic: Simulation-based shape optimization use case," in *Proceedings OLA'2025: International Conference on Optimization and Learning*, E.-G. Talbi, Ed., 2025, pp. 128–139.

Independent Scientific Component Part or a Chapter in a Monograph

R. Hribar, G. Petelin, J. Šilc, G. Papa, and V. Vukašinović, "Memetic algorithm and evolutionary operators for multi-objective matrix tri-factorization problem," in *Heuristics for Optimization and Learning*, F. Yalaoui, L. Amodeo, and E.-G. Talbi, Eds., Springer, 2021, pp. 286–298.

M. Antoniou, R. Hribar, and G. Papa, "Parameter control in evolutionary optimisation," in *Optimization Under Uncertainty with Applications to Aerospace Engineering*, M. Vasile, Ed., Springer, 2021, pp. 357–385.

Published Scientific Conference Contribution Abstract

J. Povh, R. Hribar, and G. Papa, "Four algorithms to solve symmetric multi-type non-negative matrix tri-factorization problem," in *Book of Abstracts: 18th International Conference on Operational Research*, A. Čeh Časni and J. Arnerić, Eds., Croatian Operational Research Society, 2020, pp. 30–31.

R. Hribar, A. Kastrin, G. Papa, J. Povh, and V. Vukašinović, "Clustering scientific bibliographic data with symmetric orthogonal matrix tri-factorization," in *Austrian-Slovenian HPC Meeting 2021*, J. Krek, Ed., University of Ljubljana, 2021, p. 36.

R. Hribar, "Evolutionary operators for memetic algorithm for solving matrix tri-factorization problem," in *Zbornik povzetkov slovenskih delavnic "Algoritmi po vzorih iz narave" v študijskem letu 2017/2018*, B. Filipič, M. Mernik, and G. Papa, Eds., (in Slovene), Institut "Jožef Stefan", 2018, p. 16.

——, "Protein structure prediction using deep reinforcement learning," in *Zbornik povzetkov slovenskih delavnic "Algoritmi po vzorih iz narave" v študijskem letu 2017/2018*, B. Filipič, M. Mernik, and G. Papa, Eds., Institut "Jožef Stefan", 2018, p. 8.

Other Publications

Original Scientific Article

G. Petelin, R. Hribar, and G. Papa, "Models for forecasting the traffic flow within the city of Ljubljana," *European Transport Research Review*, vol. 15, pp. 1–20, Article no. 30, 2023.

Published Scientific Conference Contribution

T. Eftimov, G. Petelin, R. Hribar, G. Popovski, U. Škvorc, and P. Korošec, "Deep statistics: More robust performance statistics for single-objective optimization benchmarking," in *GECCO'20 Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, C. A. Coello Coello, Ed., ACM, 2020, pp. 5–6.

T. Eftimov, R. Hribar, U. Škvorc, G. Popovski, G. Petelin, and P. Korošec, "Performviz: A machine learning approach to visualize and understand the performance of single-objective optimization algorithms," in *GECCO'20 Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, C. A. Coello Coello, Ed., ACM, 2020, pp. 7–8.

R. Hribar, G. Petelin, M. Antoniou, A. Biasizzo, S. Ciglarič, and G. Papa, "On suitability of the customized measuring device for electric motor," in *IECON 2022, 48th Annual Conference of the IEEE Industrial Electronics Society*, 2022, pp. 1–6, Article ID 9968876.

Published Scientific Conference Contribution Abstract

T. Eftimov, R. Hribar, and P. Korošec, "Deep statistics: Next generation statistics for optimization algorithms," in *2019 IEEE Symposium Series on Computational Intelligence (SSCI)*, Q. Liu and N. Zhang, Eds., IEEE, 2019, p. 35.

Independent Scientific Component Part or a Chapter in a Monograph

R. Hribar and D. Torkar, "Explainable anomaly detection of 12-lead ECG signals using denoising autoencoder," in *Intelligent Secure Trustable Things*. Springer, 2024, pp. 127–140.

R. Hribar, M. Antoniou, and G. Papa, "A framework for applying data-driven AI/ML models in reliability," in *Recent Advances in Microelectronics Reliability: Contributions from the European ECSEL JU Project IRel40*. Springer Nature, 2024, pp. 323–337.

G. Petelin, R. Hribar, S. Ciglarič, J. Herman, A. Biasizzo, P. Korošec, and G. Papa, "Reliability improvements for in-wheel motor," in *Recent Advances in Microelectronics Reliability: Contributions from the European ECSEL JU Project IRel40*. Springer Nature, 2024, pp. 197–212.

Biography

Rok Hribar was born on December 4, 1988 in Celje, Slovenia. In 2016, he received a Bachelor of Science in Physics from the Faculty of Mathematics and Physics of the University of Ljubljana, Slovenia. From 2014 to 2017, he held a teaching position at the private school LANDIS Education and Business Consulting, where he led courses Physics and Physics in Healthcare. In 2016, he began his doctoral studies at the Jožef Stefan International Postgraduate School in Ljubljana, Slovenia, under the supervision of Prof. Dr. Gregor Papa. For his studies, he received funding from Young Researchers grant and Computer Structures and Systems program, both funded by the Slovenian Research and Innovation Agency.

Since 2016, he has held a research position at the Computer Systems Department of the Jožef Stefan Institute, Slovenia. There he took part in several projects, including, SYNERGY (H2020–Twinning), Biomedical Data Fusion (ARIS), InSecTT, iRel40 (ECSEL/H2020), CONDUCTOR (HE) and EXOWORLD (HE MSCA), where he also took on the role of leader of the Advanced Computational Tools workpackage. In 2022, he also held a researcher position at the Institute for Biostatistics and Medical Informatics, Faculty of Medicine, University of Ljubljana, Slovenia. There he was involved in two projects funded by the Slovenian Research and Innovation Agency related to statistics and knowledge discovery in medicine.

Between 2019 and 2024, he was invited to lead several workshops on deep learning and gradient-based methods for both national and international audiences. These included three full-day invited talks at the PRACE Autumn School in Ljubljana (2019–2021), two tutorials at the Croatian AI2FUTURE22 conference and the SSDS 2021 summer school, three tutorial workshops in Ljubljana organized by the TETRAMAX project in 2021, two two-day lectures and workshops in Ljubljana organized by the EuroCC and EuroHPC projects (2021–2022), and a two-day workshop in Berlin organized by the EXOWORLD project in 2024.

His main areas of research are machine learning, optimization, and data mining. He is particularly interested in deep learning, gradient-based optimization, physics-informed machine learning, probabilistic modeling, representation learning, time series modeling and forecasting, knowledge discovery, anomaly detection, automatic differentiation, manifold learning, parameter control in optimization, reinforcement learning, nonnegative matrix factorization, simulation-based optimization, multi-objective optimization, and evolutionary algorithms. He has presented his work at several national and international conferences and published papers in scientific journals with an impact factor.