Fachhochschule Aachen
Campus Jülich
Fachbereich 09 – Medizintechnik und Technomathematik
Studiengang Angewandte Mathematik und Informatik M.Sc.

FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

Investigating self-supervised learning for diffusion MRI-based microstructure imaging

Masterarbeit von Daniel Todt
Matrikelnummer 3110512
Jülich, 13. September 2021

Erklärung

Diese Arbeit ist von mir selbstständig angefertigt und verfasst. Es sind keine anderen als die angegebenen Quellen und Hilfsmittel benutzt worden.

Ort, Datum Daniel Todt

Diese Arbeit wurde betreut von:
1. Prüfer: Prof. Dr. Martin Reißel
2. Prüfer: Dr. Kai Krajsek

Contents

Abstract

Diffusion magnetic resonance imaging (diffusion MRI or dMRI) offers insights into the structure and the connectivity of the brain by observing particle displacements of water molecules in brain tissue. This helps to study the microstructural organization and morphology of the brain. Diffusion MRI can also be used as a diagnostic tool in medicine, e. g. for the detection of tumors. Microstructure models are used to describe dMRI signals depending on the geometry of the underlying tissue. They are based on diffusion characteristics of water molecules in biological tissue. The idea in microstructure imaging is to find a parameter setting for a microstructure model based on dMRI measurements which describes the resulting dMRI signals best. This is an inverse problem which needs to be solved efficiently.

In this thesis, microstructure imaging is performed for the Neurite Orientation Dispersion and Density Imaging (NODDI) model using different approaches. Apart from Accelerated Microstructure Imaging via Convex Optimization (AMICO) as the baseline, successful supervised deep learning architectures for microstructure imaging are employed. As a new approach, self-supervised models which originate from deep inverse model designs are used for microstructure imaging as well. Their performance on the estimation of NODDI parameters is evaluated and compared using a synthetic dataset. A particular advantage of self-supervised models is that they can be directly applied to dMRI measurements without the need for a dataset with a ground truth.

The thesis is organized as follows: first, the thematic background regarding microstructure imaging and microstructure models is described. After that, basics for machine learning are explained and the methods used in this thesis are presented, including implementation details. Finally, the results are discussed and an outlook is given.

List of Abbreviations

AMICO	Accelerated Microstructure Imaging via Convex Optimization
CPU	Central Processing Unit
CSF	Cerebrospinal Fluid
dMRI	Diffusion Magnetic Resonance Imaging
FOD	Fiber Orientation Distribution
GPU	Graphics Processing Unit
HAF	Helmholtz Analytics Framework
LSTM	Long Short-Term Memory
MAE	Mean Absolute Error
MESC	Microstructure Estimation with Sparse Coding
MLP	Multilayer Perceptron
M-LSTM	Modified Long Short-Term Memory
MR	Magnetic Resonance
MRE	Mean Relative Error
MRI	Magnetic Resonance Imaging
MSE	Mean Squared Error
NA	Neural Adjoint
NN	Neural Network
NODDI	Neurite Orientation Dispersion and Density Imaging
ODI	Orientation Disperson Index
PGSE	Pulsed-Gradient Spin-Echo
ReLU	Rectified Linear Unit
RNN	Recurrent Neural Network
SGD	Stochastic Gradient Descent
SNR	Signal-to-Noise Ratio

1 Introduction

Diffusion magnetic resonance imaging (diffusion MRI or dMRI) is a non-invasive imaging technique which allows to observe molecule displacements in biological tissue microstructure. Usually the movement of water molecules in brain tissue is measured along specific orientations. This helps analyzing the connectivity and organization of the brain, including brain structure and function. The examination of brain structures is a current research topic. Besides, dMRI helps understanding brain disorders like Alzheimer's disease or multiple sclerosis and can be used as a diagnostic tool to detect tumors in clinical settings.

To analyze brain tissue using dMRI, biophysical models are used to represent tissue microstructure. Such microstructure models map the tissue geometry to dMRI signals by considering the diffusion behavior of water molecules inside the tissue. The models are based on diffusion characteristics of water molecules in specific types of tissue. However, dMRI signals arise from water molecules in all different kinds of brain tissue and cells. Therefore, multi-compartment models provide individual models for each water movement pattern which arise from different tissue regions. The signals from each compartment are then added up to yield the final dMRI signal. Now the idea is to make an association between dMRI measurements and the properties of the tissue microstructure. In microstructure imaging, the task is to find parameters for the microstructure model that represent the tissue microstructure based on dMRI measurements. This is an inverse problem which can be addressed with different approaches.

The research questions addressed in this thesis were motivated from the Helmholtz Analytics Framework (HAF) [18], which is a pilot project in data science funded by the Helmholtz Initiative and Networking Fund. The idea is to develop techniques for data analysis following a co-design approach, which encourages the exchange between experts for particular domains and data scientists. Microstructure imaging is a part of one of the use cases in the HAF project. The most relevant microstructure model used in the project is the Neurite Orientation Dispersion and Density Imaging (NODDI) model [Zha+12], which will be the microstructure model examined in this thesis.

The interesting question discussed in this thesis is how the model parameters from a microstructure model can be obtained from dMRI measurements. Early and traditional methods often need a long time to compute or may provide inaccurate results. The computation time is shortened drastically with the Accelerated Microstructure Imaging via Convex Optimization (AMICO) [Dad+15] approach. This method will be the baseline in this work. Recent publications like [Gol+16; YLC19] show that artificial neural networks are also capable to estimate microstructure quantities. The network architectures are sometimes customized to fit the microstructure imaging task. All the presented methods work in a supervised manner. In this thesis, the idea is to use a

self-supervised approach for microstructure imaging and compare the results to current supervised methods. The self-supervised approach can be advantageous, since the resulting methods can be directly applied to real measurements without the need for a dataset with a ground truth.

To examine different methods for microstructure imaging using the NODDI model, a synthetic dataset will be generated using the Python [Tea15] library DmiPy [Fic+18]. Then, the different methods will be applied to the dataset. Alongside the AMICO approach as the baseline, a multilayer perceptron (MLP) [Gol+16] as well as the Microstructure Estimation with Sparse Coding (MESC) [YLC19] architecture are employed. As new techniques for microstructure imaging following a self-supervised approach, the Tandem model [Liu+18] and Neural Adjoint [RPM21] are introduced. Both architectures originate from deep inverse model designs. At the end, the performance of the methods on estimating NODDI parameters will be compared.

The thesis is organized as follows: In chapter 2, the thematic background of this thesis, i. e. diffusion MRI and microstructure imaging in general, is described in detail. After that, basics for microstructure models, especially multi-compartment models, are explained and the NODDI model is presented. In chapter 4, basics of machine learning and deep learning and relevant details for this work are summarized. The following chapter presents the algorithms and methods used in this work, including the AMICO approach among others. Chapter 6 outlines implementation details for the experiment, i. e. data generation and the execution of the algorithms and methods presented before. Finally, the results will be discussed and an outlook will be given.

2 Thematic Background

To get a better understanding about the thematic background of diffusion MRI-based microstructure imaging, this chapter will introduce the basics that are needed to follow the rest of this thesis. It is organized as follows: first, basic terms about brain tissue and biological neurons will be explained. Then, diffusion MRI will be described including the PGSE sequence and the structure of acquisition schemes. Finally, the idea of microstructure imaging will be explained.

2.1 Basic Terms

Before describing diffusion MRI and microstructure models in detail, some basics about brain tissue and biological neurons will be explained. It is necessary to know a few basic terms about neurons and brain tissue to understand the following chapters.

As described in [RCC07], the brain consists basically of cells, which are contained in grey matter and white matter. In brain slices, grey matter refers to the darker areas of the brain, whereas white matter denotes the paler areas. White matter transmits information between different brain regions of grey matter. The cells in the brain can be divided into neurons and glial cells. The neurons, the first type of brain cells, make it possible to master complex computational tasks by carrying an electric signal from one neuron to another. Their structure is shown in fig. 2.1. The signal from other neurons is received by the dendrites. The inputs from the dendrites are then integrated together and processed in the cell body called soma. Subsequently, the new signal is conducted by the axon to other neurons. The dendrites and axons are also referred to as neurites. Glial cells, the other type of brain cells, have a more supportive character and have different functions, which help the neurons to process and communicate their signals. One type of glial cells facilitate the signal processing, which makes it possible for neurons to carry a signal over a long distance. Roughly said, they wrap their membranes around the axons, which causes the processed signal in the axons to speed up. Other glial cells try to obtain a good environment for neurons, e. g. by absorbing ions which were released by the neurons. Some regions in the brain are filled with cerebrospinal fluid (CSF). CSF protects the brain from shear forces which may occur in the skull. The main spaces where CSF can be found are called the ventricles. More details about brain tissue and structure can be found in [RCC07].

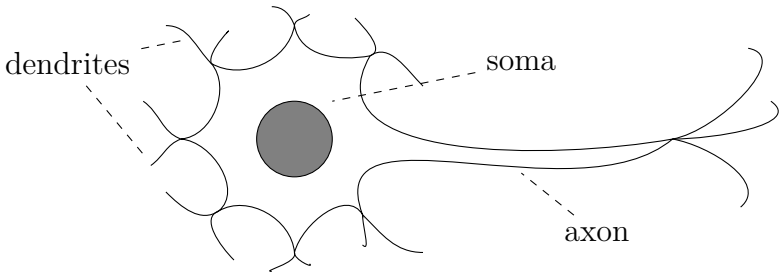


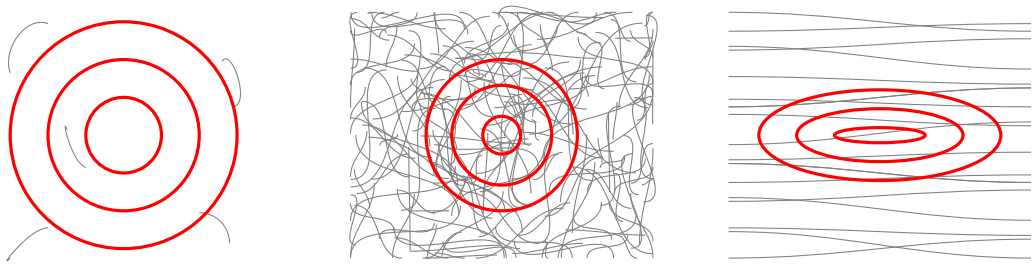
Figure 2.1: Simplified visualization of a biological neuron

2.2 Diffusion MRI

Diffusion magnetic resonance imaging (diffusion MRI or dMRI) is based on the nuclear magnetic resonance (NMR) phenomenon. Diffusion MRI serves the observation of particle displacements in tissue microstructure. To be more precise, it is a measure of the probability density function p which describes the displacements of specific particles x over a fixed time t. The tissue microstructure determines the motion or displacements of the particles, i. e. it thus defines the probability density function p. The content in this section is based on [Gos+10], [Ale06] and [Fic+18].

In biomedicine, the particles that people are interested in often are water molecules. Water is ubiquitous in biological tissue. For example, the human body consists of 70% water (by weight). Water molecules move randomly in biological tissue because of thermal fluctuations. Some barriers inside the tissue however limit their motion. The most common application for diffusion MRI is brain imaging or microstructure imaging. Diffusion MRI makes the non-invasive evaluation of brain tissue possible without ionizing radiation. This enables researchers to investigate the connectivity of the brain. Besides, diffusion MRI became a diagnostic tool in medicine, e. g. for the detection of tumors or the conduction of a brain fiber analysis. Today, the resolution of diffusion MRI is below a millimeter and is able to generate three-dimensional images, where each voxel contains multiple measurements.

The brain structure is complex, as described in section 2.1. It roughly consists of grey matter, which is connected by white matter fibers. Moreover, there are some areas of the brain filled with cerebro-spinal fluid (CSF), e. g. in the ventricles. In CSF, there are nearly no barriers which limit the particle displacements of water molecules. Therefore, the distribution p is isotropic in these regions of the brain as visualized in fig. 2.2. Grey matter is dense and contains many barriers which limit the mobility of water molecules. Such barriers can be cell walls or membranes. These barriers often have no major orientation, therefore they hinder the diffusion equally in all directions. Hence, p again is isotropic, but this time is less spread out than in CSF, i. e. the average movement is smaller. In contrast, white matter consists for example of axon fiber bundles which connect the regions in the brain to each other. The barriers now have a preferred orientation and have much more consistency than in grey matter. The movement is now hindered in directions which are perpendicular to the fiber orientation, but is not



cerebrospinal fluid grey matter white matter

Figure 2.2: Simplified visualization of different brain tissue and the corresponding probability distribution. The grey lines refer to barriers in tissue microstructure and the red lines indicate the isolines of the probability density function p.

hindered along the fiber orientation. This leads to larger displacements along the fiber axis, so p is not isotropic in this case. If the orientation of diffusion (and with that the fiber orientation) can be estimated for each voxel in the 3D image of a brain, the fibers can be followed through the brain for better insights into the connectivity. This is called tractography.

2.2.1 PGSE Sequence

This subsection will describe the acquisition procedure of dMRI measurements. With dMRI, the self-diffusion of water molecules can be measured along defined orientations. To achieve this, the standard MRI spin-echo sequence is extended by magnetic gradients for dMRI measurements. The resulting sequence presented in this subsection is called pulsed-gradient spin-echo (PGSE) sequence, which is the most common sequence for diffusion MRI. Its procedure is visualized in fig. 2.3.

Water molecules have an intrinsic spin that leads to a magnetic moment. In dMRI, the tissue of interest is put in a constant and homogeneous magnetic field B_0. This external megnetic field causes the spins to line up with B_0, i. e. aligning their rotation axis with the magnetic field. More spins align along the external magnetic field (spin up state) than oppositely (spin down state), as this configuration has a lower energy. As a consequence of this disequilibrium, the tissue has a net magnetization. The magnetic moments of the water molecules add up to a magnetic moment vector that has a vector component alongside the external field. At the beginning of the PGSE sequence, a 90° radio frequency pulse (RF) P90 makes the spins to tip over into the transverse plane, which is the plane perpendicular to the field B_0. A radio frequency pulse in general can be used to flip the magnetic moment of a molecule by a particular angle (in this case 90°). The pulse is in fact another magnetic field, but this time it is oscillating instead of being static like B_0. The pulse is only effective at the correct frequency, thus this is a resonance phenomenon. The frequency is called Larmor frequency and is dependent on B_0 and on the respective molecule. The spins are now precessing at Larmor frequency around B_0. Besides, the P90 pulse makes the spins to precess in phase.

After the P90 pulse, two types of relaxation occur. The first type is T1 relaxation, which occurs when the molecule flips from spin up state to spin down state due to energy difference. The second one, T2 relaxation, occurs when the spins begin to get out of phase gradually due to inhomogeneities of B_0. Therefore, the net magnetization decays. During that relaxation, a second RF pulse P180 (now inducing a flip by 180°) is inserted at half the echo time $\frac{TE}{2}$. The echo time TE is the time that elapses between the beginning of the sequence and when measurement is taken. The P180 pulse causes the spins to negate their phase and thus reverse the relaxation process they experienced after the P90 pulse. The negation of the phase leads on the one hand to a delay for the faster magnetic moments and on the other hand to an advance for the slower ones. This however makes the spins to be in phase again at echo time TE and therefore refocus the magnetic moments when the measurement is taken. Hence, without additional magnetic gradient pulses, the net magnetization is recovered at TE. This magnetization is then measured being the resulting MR signal.

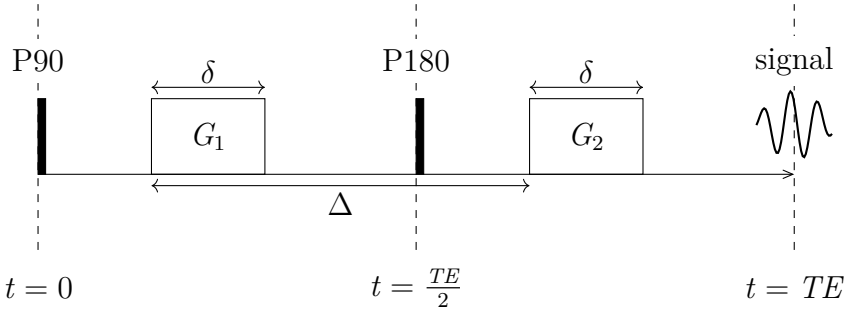


Figure 2.3: Schematic visualization of the PGSE sequence

The sequence described above is a standard MRI spin-echo sequence. To extend it to the PGSE sequence for diffusion MRI, two additional gradient pulses G_1 and G_2 are integrated into the sequence. The two magnetic gradient pulses added to the sequence have a duration of δ for each one and a separation time of Δ. One pulse (G_1) is conducted after the P90 pulse, the second one (G_2) after the P180 pulse. The gradient pulses are constant at each position over the time δ. The gradient of the magnetic fields is applied along a specific orientation q, i. e. the magnetic field increases linearly along q. If a stationary spin is considered, the first gradient pulse leads to a spin offset depending on its position along q. The second gradient pulse reverses this effect (the spin offset) of the first pulse, since the P180 pulse applied in between the gradient pulses negates the spins. The key for assessing the particle displacement now is that when a spin moves along q during the sequence (between the gradient pulses), it undergoes a different magnetic field during the second gradient pulse since the magnetization is dependent on the position. This causes the spin to have a phase offset at echo time TE compared to stationary spins that did not move. This results in a different magnetization that will be measured.

MRI measurements are in general complex valued, because the magnetization is measured in magnitude and phase. Since the phase measurements can be unstable due to the movement of the probed tissue or to inhomogeneities in B_0, the phase often is discarded.

The modulus of the summed magnetization of all spins is taken as the measurement in practice. It is common to consider an additive Gaussian noise for MRI measurements. This means the real and imaginary part of the error is independent and identically distributed (iid) with $\mathcal{N}(0, \sigma^2)$. Interpreted on the resulting modulus signal, this is called a Rician noise. Rician noise is described in more detail in section 6.1.

2.2.2 Acquisition Schemes

The information of different microstructure tissue properties can be obtained by repeatedly applying a PGSE sequence with multiple parameter combinations. The full description of an acquisition procedure is then called an acquisition scheme. It describes the different combinations used for examining the tissue. The parameters contained in an acquisition scheme are the gradient strength G, the pulse duration δ and separation time Δ, the echo time TE and the different orientations q of the gradient pulse. Besides, there is a b value which is calculated as

$$b = G^2\delta^2\gamma^2\left(\Delta - \frac{\delta}{3}\right) \left[\mathrm{s/m^2}\right] \tag{2.1}$$

with γ as the nuclear gyromagnetic ratio. The b value is also common to be used in acquisition schemes for clinical settings. In general, acquisition schemes are organized in so-called shells. Single-shell acquisition schemes only vary the orientation q of gradient pulses. A common number of gradient pulses is 90. Multi-shell acquisition schemes are also varying the gradient strength G. One shell is then the application of all the gradient orientations q with a single gradient strength G. An example of a multi-shell scheme is shown in listing 2.1, the WU-Minn HCP acquisition scheme. Multi-diffusion time shell acquisition schemes also vary the duration δ and separation time Δ. This can be beneficial to gain a better insight into some microstructure tissue properties. Since multi-shell acquisition schemes vary the orientation q and the gradient strength G, the measurements can be located in the q-space, which is spun by the orientation and gradient strength.

Listing 2.1: WU-Minn HCP acquisition scheme

```
>>> from dmipy.data import saved_acquisition_schemes
>>> q=saved_acquisition_schemes.wu_minn_hcp_acquisition_scheme()
>>> q.print_acquisition_info
'''Acquisition scheme summary

total number of measurements: 288
number of b0 measurements: 18
number of DWI shells: 3

shell_index |# of DWIs |bvalue [s/mm^2] |gradient strength [mT/m] |delta [ms] |Delta[ms] |TE[
    ↪ ms]
0           |18        |0               |0                        |10.6       |43.1      |N/A
1           |90        |1000            |56                       |10.6       |43.1      |N/A
2           |90        |2000            |79                       |10.6       |43.1      |N/A
3           |90        |3000            |97                       |10.6       |43.1      |N/A
    ↪ '''
```

2.3 Microstructure Imaging

Diffusion MRI enables the non-invasive examination of the morphology of biological tissue. This can help to study the microstructural organization of the brain, like the connectivity [Pan+12]. Exploring the neurite morphology allows to better understand brain structure and brain functions. To get a better understanding about axon bundles (diameter, dispersion), extra-axonal diffusivity or in tumor composition, microstructure imaging is performed [Fic+18].

In microstructure imaging, biophysical models are used to represent tissue microstructure [Pan+12]. They are based on the diffusion characteristics of water molecules in different brain tissue and cells. The idea of microstructure imaging is to make an association between the dMRI measurements and the properties of the tissue microstructure. The models must be designed carefully to be able to describe the water diffusion in brain tissue accurately. They must be appropriate to represent the tissue microstructure and its diffusion properties [Fic+18]. The models map parameters or features of tissue microstructure to the resulting dMRI signals (details about microstructure models will be presented in chapter 3). Microstructure imaging now turns this procedure around (as indicated in fig. 2.4): it describes the process of finding good model parameters to represent the tissue microstructure based on dMRI measurements. In other words, microstructure imaging maps the resulting signals back to the tissue microstructure. The parameter setting is meant to produce a signal as close to the measurements as possible in theory.

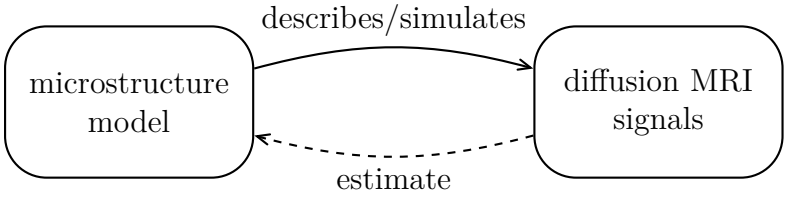


Figure 2.4: Visualization of microstructure imaging

Typical approaches for microstructure imaging make use of non-linear optimization procedures, which implies a huge computational effort. Newer techniques overcome this problem, e. g. by linearizing the optimization problem as in [Dad+15]. Also deep learning approaches were developed having promising results [YLC19]. Especially, they have a potential to make accurate predictions of microstructure quantities/features even with a reduced number of diffusion gradients used in the acquisition process. Methods used for microstructure imaging are described in detail in chapter 5.

3 Microstructure Models

Diffusion MRI is a tool for the non-invasive examination of brain tissue. To get deeper insights into the connectivity of the brain, microstructure models are used to bring the diffusion MRI measurements in a more interpretable form. They model the brain tissue such as white matter, grey matter, CSF and other parts of the brain. The diffusion MRI signals are affected by the structure of the tissue as described in section 2.2. Therefore the parameters of the microstructure model are estimated based on diffusion MRI signals in microstructure imaging. It is important that a microstructure model includes all the relevant physiological parameters which are needed to describe the tissue and the resulting signal. Otherwise no sensible parameter setting can be obtained that describes the tissue reasonably well. A common type of microstructure models is the multi-compartment model, which is described in the next section. After the introduction to multi-compartment models, the NODDI model will be presented, which is the model used for this work.

3.1 Multi-Compartment Models

In a multi-compartment model, the diffusion signal of dMRI is interpreted as a linear combination of separate signal models describing distinct types of brain tissue and their corresponding diffusion characteristics. In other words, the total diffusion signal is modeled as a sum over the signals that arise from different tissue structures. The water can be inside as well as outside of the structures of interest (e. g. axons), separated by impermeable walls. Hence, a multi-compartment model typically distinguishes between different kinds of impact that the water in each region has on the total diffusion signal: intra-axonal, extra-axonal and other compartments like CSF. These compartments are described in the following. The content of this section is based on [Pan+12].

To investigate the connectivity of the brain, white matter is the tissue of interest, since it contains the axon bundles which connect the brain regions to each other. Besides axon bundles, there are also other structures in white matter that influence the total diffusion signal, since water is found everywhere in the tissue, not only inside the axons. Therefore, the signal is usually split into contributions of an intra-axonal, extra-axonal and an isotropic compartment. The intra-axonal compartment describes the water moving inside the axons, whereas the extra-axonal compartment refers to the water outside the axons. The isotropic compartment describes isotropic restriction of water diffusion, which is found in other cellular structures besides axons, e. g. glial cells, trapped water in membranes or non-parallel fibers. Having the diffusion signals S_1, S_2 and S_3 for the intra-, extra-axonal and isotropic compartment, respectively, the total

normalized diffusion signal is then calculated as

$$S = \sum_{i=1}^{3} f_i S_i \quad \text{with} \quad \sum_{i=1}^{3} f_i = 1 \tag{3.1}$$

and $0 \leq f_i \leq 1$ being the proportion of each compartment signal from water in the corresponding compartment. How the signal for each compartment can be modeled is described in the following paragraphs.

For the intra-axonal compartment there are various models that can be used to describe the signal. One of these models is the stick model. In the stick model, the ideal diffusion in a cylinder is described, but the radius of the cylinder is zero (hence the name "stick"). The parameters of the stick model are the fiber direction μ and the parallel diffusivity d. The signal is then modeled as

$$S_1 = e^{-bd(\mu \cdot q)^2} \tag{3.2}$$

with b being the b value of eq. (2.1) described in section 2.2.2 and q being the gradient vector. The stick can be generalized to a cylinder model with non-zero radius. This results in an additional parameter r, which represents the radius of a single axon. More extensions of the stick and cylinder model exist, e. g. the radius can be modeled to be gamma-distributed to consider more than a single axon radius which occur in the examined voxel of the dMRI scan.

The extra-axonal compartment assumes hindered diffusion outside the axons, because the water is expected to be surrounded by cellular membranes (such as glial cells and somas). In hindered diffusion, the diffusion of water is modeled with a Gaussian displacement pattern. The diffusion signal arises from a diffusion tensor D with different constraints and can be calculated as

$$S_2 = e^{-bq^T Dq} \tag{3.3}$$

with q again as the gradient direction. The extra-axonal compartment can now be modeled for example via the ball model. The ball model is isotropic and only has a single parameter, the isotropic diffusivity d. The diffusion tensor for the ball model is then simply computed as

$$D = dI \tag{3.4}$$

where I is the identity matrix. Another common model is the zeppelin model. It is anisotropic, contrary to the ball model, but is cylindrically symmetric. The zeppelin model has three parameters: the principal direction μ, the parallel diffusivity $d_\parallel$ and the perpendicular diffusivity $d_\perp$. The diffusion tensor is

$$D = \alpha \mu \mu^T + \beta I \tag{3.5}$$

with $d_\parallel = \alpha + \beta$ and $d_\perp = \beta$. There are more models that can be used to describe the extra-axonal compartment, e. g. a full tensor having additional degrees of freedom.

Other cellular structures are usually consulted in the third compartment, which is used for isotropic restriction. Restricted diffusion relates to the diffusion of water which is trapped inside geometries. The displacement now follows a non-Gaussian pattern. It describes intra-cellular water, but which is bounded for example by membranes (axonal or dendritic). The two models presented below assume restrictions caused by spherical boundaries. The first model presented here is the sphere model. It characterizes diffusion inside impermeable spherical boundaries at a particular radius. The signal considered in the sphere model usually comes from water inside spherical glial cells. The second model is the dot model, which is basically the sphere model with zero radius. It is used for particles that do not move. This is the case for molecules stuck in glial cells or trapped in cellular membranes. Static molecules do not influence the diffusion signal, so the signal for the dot model is simply

$$S_3 = 1. \tag{3.6}$$

For the isotropic compartment, also other models, which are not presented here, can be used to describe the diffusion.

3.2 NODDI

The motivation of the Neurite Orientation Dispersion and Density Imaging (NODDI) model [Zha+12] is to use clinical MRI scans for the estimation of the microstructural complexity of neurites (which are dendrites and axons) in vivo. The structure of neurites relate to brain tissue mirostructure more directly than other characteristics captured by (earlier) standard imaging techniques. Examining the neurite structure can lead to better insights to specific tissue properties which concern the structural basis of brain functions. This can help to better understand brain development (e. g. in neonatal imaging) and brain disorders (such as dementia). The dendritic density measures the branching complexity of dendrites and dendritic trees. The density can represent their function: low density often occurs in an early stage of the information processing in the brain, high density (or complex structures) tend to appear at a later stage. This is also related to aging and brain development. Brain development is measurable by the increase of neurite orientation dispersion. In contrast, aging is associated with a reduction of the dendritic density. Such changes in the neurite morphology are also often linked to diseases like Alzheimer's disease or multiple sclerosis. This is why NODDI is intended to be a non-invasive technique to quantify orientation dispersion and neurite density in vivo.

The NODDI model is a three-compartment model designed to be sensitive to neurite density and their orientation dispersion. The compartments of the NODDI model are an intra-cellular, extra-cellular and an isotropic compartment (for CSF). Each compartment models the water diffusion in a different environment, resulting in a different contribution to the total diffusion signal. By distinguishing the intra- and extra-cellular compartments, the neurite morphology becomes measurable by diffusion MRI. While orientation dispersion can be investigated using a single-shell acquisition scheme, two

shells are required for neurite density. The total normalized signal of the NODDI model is defined as

$$S = (1 - v_{\text{iso}}) \left(v_{\text{ic}} S_{\text{ic}} + (1 - v_{\text{ic}}) S_{\text{ec}} \right) + v_{\text{iso}} S_{\text{iso}} \tag{3.7}$$

with S_{ic}, S_{ec} and S_{iso} as the separated normalized signals for the intra-, extra-cellular and isotropic compartment and $v_{\text{ic}}, v_{\text{iso}} \in [0; 1]$ being the intra-cellular volume fraction and isotropic volume fraction, respectively. The next paragraphs will separately describe each compartment in detail.

The water bounded by membranes of neurites is considered in the intra-cellular compartment. NODDI uses a set of sticks (cylinders of zero radius, see section 3.1) to model the intra-cellular space. This set of sticks is meant to represent the unhindered diffusion in the direction of neurites and the diffusion restriction perpendicular to the neurites. The set of sticks is modeled as a distribution where the sticks can be expressed to be highly parallel for example, but it is also possible to model them quite dispersed. The distribution of sticks covers a range of patterns that are usually observed in tissue microstructure, e. g. white matter tissue with axon bundles which are coherently orientated. Besides, it captures white matter where the axons bend or run apart. It can also describe grey matter where dendrites sprawl widely in all directions. The intra-axonal signal is calculated as

$$S_{\text{ic}} = \int_{\mathbb{S}^2} f(n) e^{-bd(q \cdot n)^2} dn \tag{3.8}$$

where $\mathbb{S}^2$ is the unit sphere, f is the probability density function for the presence of sticks in orientation n and the latter part of the integral is the general signal of a stick already presented in eq. (3.2) (section 3.1). The distribution of f is modeled with a Watson distribution, so f is defined as

$$f(n) = \mathrm{M}\left(\frac{1}{2}, \frac{3}{2}, \kappa\right)^{-1} e^{\kappa(\mu \cdot n)^2} \tag{3.9}$$

with M as a confluent hypergeometric function, μ as the mean orientation of the sticks and κ as the concentration parameter which controls the dispersion of stick orientations around μ. The Watson distribution is the simplest distribution for modeling orientation dispersion in this case. It can express not only low dispersion as it is often observed in white matter, but also high dispersion mostly observed in grey matter.

The water diffusion around the neurites is captured in the extra-cellular compartment. In this space, glial cells and somas can be found. The diffusion of water is hindered due to the neurites. Since the diffusion is hindered and not restricted, a Gaussian anisotropic diffusion is applied here. The signal is modeled as

$$\log S_{\text{ec}} = -b q^T \left(\int_{\mathbb{S}^2} f(n) D(n) dn \right) q \tag{3.10}$$

which adapts the extra-axonal zeppelin model presented in eqs. (3.3) and (3.5) to the orientation dispersion distribution. The perpendicular diffusivity $d_\perp$ of the zeppelin model is computed following a tortuosity model:

$$d_\perp = d_\parallel \left(1 - v_{\text{ic}}\right). \tag{3.11}$$

It is important to mention that both, the intra- and extra-cellular compartment, are dependent on the distribution of f. This means that they are affected by each other.

The last compartment considers regions in the brain flooded with CSF. It is modeled using isotropic Gaussian diffusion with a specific isotropic diffusivity d_{iso}. This concept corresponds to the ball model described in eqs. (3.3) and (3.4) in section 3.1.

Besides diffusivities, which are usually set beforehand, the parameters to be estimated for the NODDI model are the intra-cellular volume fraction v_{ic}, the isotropic volume fraction v_{iso} and the distribution parameter κ. Instead of using κ, the authors of [Zha+12] propose the orientation dispersion index

$$\text{ODI} = \frac{2}{\pi} \arctan\left(\frac{1}{\kappa}\right) \tag{3.12}$$

to be used as the parameter to be estimated for NODDI. They suggest this, because the ODI, unlike κ, maps high dispersion to high values of ODI and vice versa. This is more intuitive and easier to interpret. Besides, the ODI only has values in the range 0 to 1, which leads to better visualization of ODI compared to κ.

4 Machine Learning Basics

This chapter will give an introduction into concepts of machine learning. First, general terms used in machine learning will be explained. After that the bias variance decomposition and the idea of validation will be described. This is followed by an introduction to artificial neural networks. The chapter is based on [GBC16] and [Alp10].

Machine learning is the research field investigating methods that improve performance of tasks from experience. The goal is to find a solution to the considered problem by finding functional dependencies as well as patterns and regularities in the given dataset. Machine learning approaches can be roughly divided into supervised, unsupervised and reinforcement learning. This work focuses on the use of supervised and self-supervised learning, where the latter is an intermediate form of supervised and unsupervised learning. The main difference between supervised and unsupervised learning concerns the nature of the dataset. A dataset for supervised learning consists of input data $\mathbb{X}$ and target data $\mathbb{Y}$, whereas unsupervised learning only uses input data $\mathbb{X}$ without any target values. This fact affects the types of problems that are typically solved by each learning form.

A learning algorithm in supervised learning is supposed to find a mapping between the input data $\mathbb{X}$ and the true output values $\mathbb{Y}$, whereat the quality of that mapping can be verified during the training process. The dataset $\mathbb{D}$ contains data points (also called samples) as tuples $(x_i, y_i) \in \mathbb{D}$ which assign a true output value $y_i \in \mathbb{Y}$ (also called label) to each input $x_i \in \mathbb{X}$. The dataset is often assumed to be randomly drawn from a multivariate probability distribution $d_{\text{data}}(x, y)$. The samples are considered to be independent and identically distributed. Since the dataset $\mathbb{D}$ is sampled from a distribution, it usually doesn't cover up all possible inputs and labels, but rather a subset from $\mathbb{X}$ and $\mathbb{Y}$.

Since the connections between the inputs x_i and targets y_i are often complex and depend on various properties and characteristics of the considered problem, several variables are typically used as predictors in the learning algorithms. These are called features. A feature can either be the input data or a transformation of them. An input with m features can be expressed as a vector $x_i \in \mathbb{R}^m$, where every component represents the value of a certain feature.

It is often assumed that there is a function $f : \mathbb{X} \to \mathbb{Y}$ which was used to generate the dataset. A learning algorithm aims to approximate the function f with another function $g : \mathbb{X} \to \mathbb{Y}$ as good as possible. A set of candidate functions $\mathbb{H}$ is provided to the algorithm. This set typically results from the structure of the algorithm and is called hypothesis set or hypothesis space. For instance, a simple linear regression without any transformation of the input data can only find hypotheses g in form of a linear function, not a parabola or exponential function. The learning algorithm usually considers several

hypotheses and tries to select a final hypothesis $g \in \mathbb{H}$ that behaves most likely like the target function f, meaning $g(x_i) \approx f(x_i)\ \forall x_i \in \mathbb{X}$.

Supervised learning distinguishes between two types of problems: classification and regression. In a classification task the model is meant to specify which category a sample belongs to, whereas in a regression task the model should predict a continuous value. For instance, microstructure imaging, the problem investigated in section 2.3, is a regression task. In this case the codomain of the target function f are real numbers, which need to be approximated. The performance measure of a hypothesis g on a dataset is done via a loss function that calculates the error between a prediction of a learning algorithm $g(x_i)$ and the associated label. For a simple regression task a typical choice of a loss function is the squared error.

The predictions of a learning algorithm can often be represented by a conditional probability distribution $p_\vartheta(y \mid x)$. The structure and settings of the model (e. g. the hypothesis set) define a family of possible probability distributions parameterized by ϑ. Every single possible value of the parameters ϑ result in a different final hypothesis g. To make the best possible predictions on unknown data, the model's probability distribution p_ϑ should be as close to the conditional data-generating probability distribution $p_{\text{data}}(y \mid x)$ as possible. The modeled distribution p_ϑ as well as the hypothesis set $\mathbb{H}$ are closely related to the model complexity. The model complexity describes the capability of the model to approximate the data with appropriate hypotheses. The model complexity is affected in particular by adaptions to the hypothesis set. A model obtains the best results, if the model complexity matches the complexity of the problem and the size of the dataset.

The process of finding a final hypothesis g which minimizes the error on the given dataset, is called training. The error on the data used in the training process is called training error, empirical risk or in-sample error E_{in}. It is calculated as

$$E_{\text{in}} = \frac{1}{n} \sum_{i=1}^{n} l(y_i, g(x_i)) \tag{4.1}$$

using a specially chosen loss function l which compares the model predictions $g(x_i)$ to the labels y_i. The term empirical risk is due to the fact that the error can only be measured on a subset of $\mathbb{X}$ and $\mathbb{Y}$ but not on the entire domain. The error one would expect on all imaginable data for the specific problem is called generalization error, expected risk or out-of-sample error E_{out}. Since the generalization error is intractable to compute for most problems, it is estimated using a special dataset. This dataset contains data points that were not used during training, i. e. they were not used to find the final hypothesis g. The out-of-sample error E_{out} is estimated using new and unknown data, which has not been seen by the learning algorithm before. These unseen data points are often drawn from the given dataset at the beginning of the machine learning project. To ensure that these data points are not used by the learning algorithm, the dataset $\mathbb{D}$ is separated into two datasets: the training set (which contains the training data used during training) and the test set. The test set is only used for estimating the out-of-sample error E_{out} for the final performance measure of the method.

A few things must be kept in mind when trying to obtain good results with a learning algorithm. First, it is very important not to make any decisions based on the test set. The test is only used for approximating the out-of-sample error E_{out}. Every decision made using the test set can lead to a biased approximation of E_{out}. Hence, for instance when scaling the data before training, it is important to consider the training data only. A leakage of test data into the training process is called data snooping. Another important point is the selection of the data. A systematical exclusion of data points from the data sets may lead to a sampling bias. In this case the probability distribution which the training data was drawn from does not match the distribution the data occurs in reality. Hence, running into a sample bias can result in a worse performance on previously unobserved data.

4.1 Bias Variance Decomposition

According to the statistical learning theory it is possible to make statements about the observed data in a probabilistic way. In the first step a final hypothesis g is used to approximate the training data. The goal is to have the in-sample error E_{in} as low as possible. The second step is to make the final hypothesis g produce equally as good predictions on new and unknown data as on the training set. This idea is called generalization and it can be observed when the out-of-sample error E_{out} is approximately as low as the in-sample error E_{in}, i. e. $E_{\text{out}} \approx E_{\text{in}}$. Since all data points are drawn from the same data-generating distribution p_{data}, the errors E_{in} and E_{out} can be expected to be about the same magnitude in the case of perfect adaption of the model distribution p_ϑ. Approximation and generalization are in conflict with each other. Perfect approximation on the training set but at the same time bad generalization on the test set can often be observed in practice. Therefore both quantities must be weight up against each other. Important influencing factors are the amount of data and the model complexity. The more data is used for the training process, the smaller the difference between empirical and expected risk, and therefore the lower the tendency to overfitting.

Directly connected to the approximation and generalization trade-off is the bias and variance of a learning model. The bias describes the difference between the target function f and the mean final hypothesis g over every possible dataset. It can be interpreted as the principal difference between the target function and final hypothesis. A low model complexity and the resulting small hypothesis set $\mathbb{H}$ leads often to a high bias. The variance however measures the spread of the final hypothesis based on a single concrete dataset around the mean final hypothesis. It can be characterized as the instability of the model. A high instability can be observed when small changes in the dataset lead to a high variation of the final hypothesis. When the model complexity is rather high and therefore the hypothesis set $\mathbb{H}$ large, a low bias but a high variance can be observed. In addition the variance often decreases by the enlargement of the dataset. The relation between bias, variance and out-of-sample error E_{out} in dependence on the model complexity is shown in fig. 4.1.

During the training of a machine learning model overfitting or underfitting may oc-

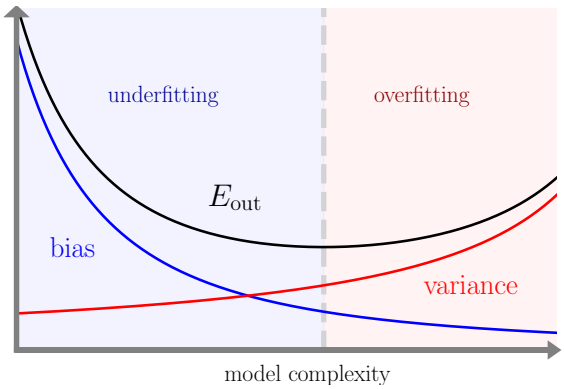


Figure 4.1: Bias, variance and out-of-sample error dependent on the model complexity

cur. Overfitting is the process of choosing final hypotheses which lead on the one hand to decreasing in-sample errors E_{in}, but on the other hand to increasing out-of-sample errors E_{out}. In this case the training data is approximated quite well, whereas unknown data like the test set is not. This phenomenon results in a large difference between E_{out} and E_{in}. Therefore the model generalizes badly. An essential influencing factor regarding overfitting is how the quality and quantity of the given data fit to the chosen model complexity. If the model complexity is too high, the model will be able to learn many insignificant characteristics and noise in the data. This has a negative impact on the generalization. Overfitting can be reduced for instance by using larger datasets, adapting the model complexity or applying regularization, which is actually a way to reduce the model complexity. In the case of underfitting, the training set is not approximated reasonably good, since the model complexity is not sufficient to describe the mapping between the input and output. Therefore underfitting represents the opposite of overfitting. The error both on the training and the test set remains high. The data is approximated badly and the model has a high bias. Figure 4.2 shows the effect of under- and overfitting in the case of approximating the function $f(x) = -\frac{1}{2}x^2 + 2x + 2$.

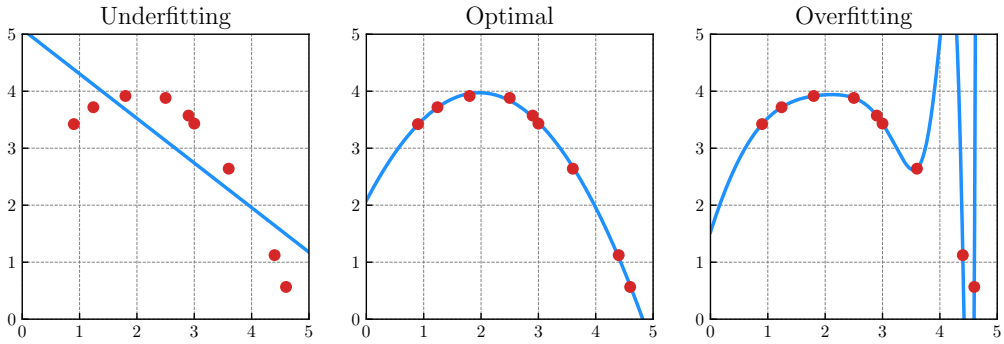


Figure 4.2: Underfitting, overfitting and optimal fitting during the approximation of the function $f(x) = -\frac{1}{2}x^2 + 2x + 2$. The red dots represent the training data and the blue curves are the final hypotheses.

4.2 Hyperparameters and Validation

There are many adjustments that can be made to the model to influence the model complexity and the hypothesis set. The adjustable parameters of a model which are not optimized during the training process are called hyperparameters. It is not recommended to learn hyperparameters as free parameters during training, because this would lead to an increase of model complexity until the in-sample error E_{in} is extremely small. This most probably leads to overfitting. Besides, for some hyperparameters it can be difficult to optimize them as a part of the algorithm. Therefore they must be set before training. It is important to set hyperparameters to appropriate values, since they can influence the performance significantly.

Before starting the learning process, another dataset is separated from the training data, the validation set. The remaining training data is the dataset which will be used during the training process to find a final hypothesis g. The validation set however serves for the approximation of the out-of-sample error E_{out} during the training process. The idea is to make decisions regarding the model (especially hyperparameters) based on the approximations of E_{out} without using the test set. Usually one would choose the model which got the lowest error on the validation set. This process is called model selection. It is common to systematically try out different combinations of hyperparameters, e. g. in the form of a grid search which is validated on the validation set. Meanwhile there are also other methods and strategies to find optimal parameters, see [Sha+16] and [DMC]. The process of selecting good hyperparameters is called hyperparameter optimization or hyperparameter tuning.

4.3 Artificial Neural Networks

Artificial neural networks (short: neural networks, NN) are a popular research topic in the field of machine learning. They are deployed for example in tasks of pattern recognition (e. g. spoken and written language), error detection or time series analysis.

This section will first explain the basics of neural networks as well as the mathematical background, including the feed-forward architecture. Moreover, the optimzation for neural networks and the backpropagation algorithm will be described.

4.3.1 Feed-forward Architecture

Neural networks in general are supposed to learn functional dependencies between the input $\mathbb{X}$ and the labels $\mathbb{Y}$. An input sample is commonly processed by applying linear affine transformations and scalar nonlinear functions. The free parameters ϑ of the network must be adjusted accordingly to reach a state of good generalization.

The most common type of a neural network is a multilayer perceptron (MLP), which is a feed-forward network. It consists of several units called artificial neurons, which are connected to each other. The artificial neuron is motivated by the biological neuron,

as it processes various input signals to a single output signal. This output signal is communicated to other neurons through the connections between the neurons.

The signal processing between neurons is modeled in a so-called perceptron. The activation of a neuron is represented by a real number. The connections between the neurons are weighted differently and thus they determine how the activation will be transmitted to the following neurons. Figure 4.3 shows how activations are computed. First, the activations x_i of the preceding neurons are multiplied with the weights w_i of the corresponding connections. These products are summed up and a bias b is added. Lastly, an activation function f is applied to the resulting sum. In summary, the activation h of a neuron is computed as

$$h = f\left(\sum_{i=1}^{n} w_i x_i + b\right). \tag{4.2}$$

Typically the activation function is a nonlinear function $f : \mathbb{R} \to \mathbb{R}$. In this work especially the rectified linear unit activation function (ReLU)

$$f(z) = \max\{0, z\} \tag{4.3}$$

is used. The ReLU activation function has no saturation effects in contrast to other activation functions (e. g. sigmoid, section 4.3.5) and it is well applicable for neural networks, because its derivative[1] is quite simple.

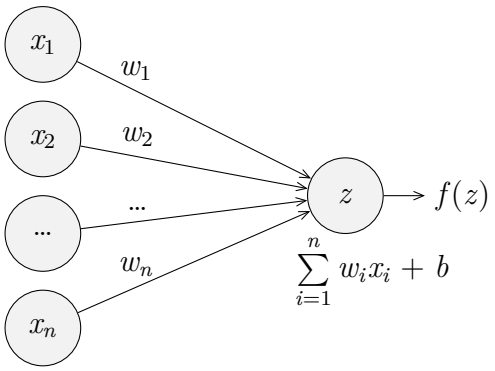


Figure 4.3: Computation of the activation of a neuron

In a feed-forward network the neurons are arranged in layers. Figure 4.4 shows an example of a feed-forward network architecture. In the case of fully connected layers, every neuron in a layer is connected to every neuron of the following layer. Thus, the input is processed only in one direction and the activations are not fed to previous neurons, hence the name feed-forward. If there are any connections within a layer or to a preceding layer, one is talking about recurrent neural networks. This network type is described in more detail in section 4.3.5 and [GBC16]. The input layer defines the beginning of a feed-forward network. Every feature in the input data corresponds to one

[1] derivative in terms of subgradient

neuron in the input layer. At the end of the network is the output layer. The amount of neurons in the output layer depends on the kind of problem to be solved. It often matches the dimensionality of the labels to make predictions directly comparable to the labels. Between the input and output layer there can be any number of additional layers. These are called hidden layers, since they are not accessed or observed from outside of the network. If there is more than one hidden layer it is usually called a deep neural network, even though there is no strict definition for networks to be considered deep.

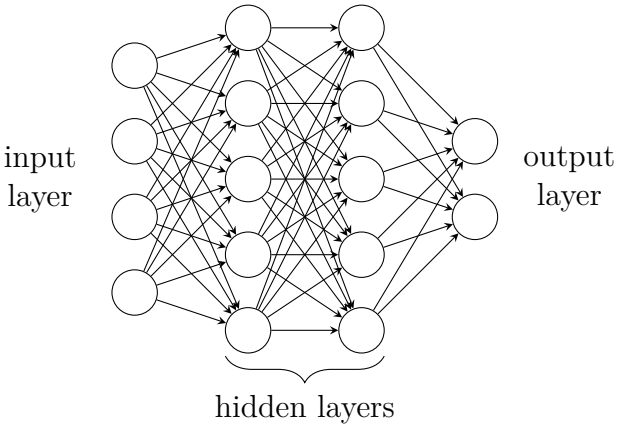


Figure 4.4: Example of a feed-forward network architecture

The computation of a prediction of a feed-forward network can be expressed using matrices and vectors. The activations of the n neurons in a layer are modeled as a vector $h \in \mathbb{R}^n$. The weights of the connections between a layer with n neurons and a layer with m neurons are gathered in a matrix $W \in \mathbb{R}^{m \times n}$. The element w_{ij} corresponds to the weight of the connection between neuron j in the preceding layer to neuron i in the following layer. The matrix-vector product Wh automatically multiplies the activations with the correct weights and sums them up for the neurons of the following layer. The bias can also be expressed as a vector $b \in \mathbb{R}^m$ and is added to the resulting activations. Finally, the activation function is applied element-wise on the resulting vector. The transition from layer i to layer $i+1$ can therefore be described as

$$h_{i+1} = f_i(\, W_i h_i + b_i). \tag{4.4}$$

If the transition is defined as a function $S_i(h_i)$, the calculation rule for the prediction $\hat{y} = h_n$ to an input $x = h_0$ of a feed-forward network with n layers will be

$$\hat{y} = h_n = S_{n-1}(S_{n-2}(\ldots S_1(S_0(x))\ldots)). \tag{4.5}$$

4.3.2 Optimization

The weights in a neural network must be adapted in such a way that the prediction based on an input sample is as close as possible to the label in terms of the cost function $J(\vartheta)$. The cost function is often defined as the in sample error E_{in}, but it can also have

additional terms (e. g. for regularization). All the weights and free parameters in a network are represented in the variable ϑ. In the case of a feed-forward network the parameters consist of weight matrices W_i and bias vectors b_i. Neural networks differ in their structure and their amount of free parameters. Every change in the parameter values in ϑ lead to a different probability distribution $p_\vartheta(y \mid x)$ that the neural network represents.

During the training process the neural network provides predictions for the training data, which can be interpreted as estimations for the corresponding labels. By considering the true labels, the in-sample error E_{in} can be calculated on the training set. Then the error can be traced back to the weights in the neural network. Specific adjustments to these weights should reduce the error. The aim of the training process is to minimize the cost function $J(\vartheta)$ and with that the in-sample error E_{in} measured as the mean loss. This is usually achieved using a gradient descent algorithm.

In every step of a gradient descent algorithm, the gradient of the cost function $\nabla_\vartheta J(\vartheta)$ must be computed with the current parameter configuration. The parameters are then updated in the opposite direction of the gradient, so

$$\vartheta_{i+1} = \vartheta_i - \eta \nabla_\vartheta J(\vartheta_i). \tag{4.6}$$

The learning rate $\eta \in \mathbb{R}^+$ represents the step size, which is a typical hyperparameter in the training of neural networks.

The computation of a step for the gradient descent based on the entire training set is often very expensive. Therefore the dataset is randomly split into many small but same sized subsets, so-called mini-batches, where the size of a mini-batch is another hyperparameter. Every gradient descent step is then calculated only using one such mini-batch. This is possible, because the total loss is a sum over pointwise losses. One of these steps is referred to as an iteration. Iterating over the whole training set once is called an epoch. The gradient that is computed based on a single mini-batch is only an estimate of the real gradient based on the entire training set. That is why this method is called stochastic gradient descent (SGD). The SGD can lead to a better performance and generalization of the network than vanilla gradient descent as described above. For SGD, the learning rate is usually set before training and is reduced during the training process. On the one hand, the reduction is supposed to prevent jumping across minima because of a rather large step size. On the other hand, a constant small step size would lead to slow convergence, because reaching the minimum is only possible doing small steps.

Since the gradient is estimated only using a single mini-batch, the error might increase after an iteration and the gradient directions may vary greatly, leading to oscillations. To gain faster convergence and avoid oscillations, a momentum term is usually incorporated in the parameter update. The actual direction of a descent step is then not only based on the direction of the current gradient, but also on the direction of previous gradients. This helps to accelerate in the major descent direction. Occasional, strongly deviating gradients thus have only a small influence on the training process.

The learning rate η has a significant impact on the learning process, so it is important to set it to an appropriate value before training. Besides, every single parameter in

ϑ influences the cost function in a different way. Adaptive methods, which provide an individual learning rate for every parameter and adapt them during the training process, are more efficient than the SGD. The most common adaptive optimization algorithms are summarized in [Rud16] and [GBC16]. The AdaGrad algorithm (*adaptive gradients*, [DHS11]) sums up squared gradients during the training and scales the learning rate anti-proportional to the square root of this sum. The problem with this algorithm is, that summing up all the gradients from the very beginning leads to an early decrease of the effective learning rate. The RMSprop algorithm [Hin12] is closely related to AdaGrad. In this algorithm the previous gradients are not summed up over the whole training process, instead an exponentially decaying average is used to enhance the algorithm to find minima that AdaGrad would not have found due to its diminishing learning rate towards the end of the training process. The Adam algorithm (*adaptive moments*, [KB14]) additionally uses estimations of the first and second moment (m_i and v_i) of the gradient to find a reasonable step direction and step size. As described in [Rud16], they are incorporated as an decaying average

$$m_i = \beta_1 m_{i-1} + (1 - \beta_1) g_i \tag{4.7}$$

$$v_i = \beta_2 v_{i-1} + (1 - \beta_2) g_i^2 \tag{4.8}$$

with $\beta_1, \beta_2 \in \mathbb{R}$ and $g_i = \nabla_\vartheta J(\vartheta_i)$. At the beginning of the training process, m_i and v_i are often biased towards zero. Therefore, the authors decided to use bias-corrected estimates of the first and second moments ($\hat{m}_i$ and $\hat{v}_i$), which are computed as

$$\hat{m}_i = \frac{m_i}{1 - \beta_1^i} \quad \text{and} \quad \hat{v}_i = \frac{v_i}{1 - \beta_2^i}. \tag{4.9}$$

The final update rule for Adam is

$$\vartheta_{i+1} = \vartheta_i - \frac{\eta}{\sqrt{\hat{v}_i} + \varepsilon} \hat{m}_i \tag{4.10}$$

with ε being a small positive constant preventing division by zero. Since $\hat{v}_i$ is a vector, the operations in eq. (4.10) are intended to be applied element-wise. The authors recommend $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\varepsilon = 10^{-8}$ as default values to be used for Adam. Adam is advantageous because of its robustness regarding hyperparameters (including the learning rate η) in the optimization process. Nevertheless the learning rate might be set manually for specific applications. This work uses the Adam algorithm for the optimization process of the neural networks.

To monitor and examine the learning process of the neural network, learning curves are usually taken into account. They show the progress of the error on the training and validation set dependent on the epoch in a plot. The training error should be decreasing with increasing number of epochs. Ideally the validation error also decreases at the beginning and flattens towards the end the more the network is saturated. If the validation error is as low as the training error, a good generalization is observed. An increase in the validation error indicates overfitting.

4.3.3 Backpropagation

To perform a gradient descent step, the gradient of the cost function $\nabla_\vartheta J(\vartheta)$ must be calculated. The algorithm used for this purpose is called backpropagation algorithm [RHW86]. The gradient for the parameters in ϑ can be calculated analytically. However, the numerical evaluation of this analytic gradient is computationally expensive, since many subexpressions must repeatedly be evaluated. Depending on the implementation, these can either be computed multiple times, which is more computational expensive and time consuming, or cached, which is only possible with larger memory requirements. The backpropagation algorithm supplies a very efficient computation method, which is based on the principle of dynamic programming. The area of application of this algorithm is not only limited to the training of neural networks.

The computation of the prediction of a neural network can be represented in a computational graph, so that each individual operation applied sequentially to the input of the network can be viewed as an independent function. For a perceptron, this is shown in fig. 4.5. Accordingly, the computation of a perceptron can be represented as a concatenation of many simple functions. Therefore, the partial derivatives which need to be calculated for the gradient can be computed using the chain rule. For a composition $f \circ g$ with $f : \mathbb{R}^n \to \mathbb{R},\ y \mapsto z$ and $g : \mathbb{R}^m \to \mathbb{R}^n,\ x \mapsto y$, according to the chain rule, the following applies

$$\frac{\partial z}{\partial x_i} = \sum_{j=1}^{n} \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}. \tag{4.11}$$

Rewritten in vector notation this results in

$$\nabla_x z = \left(\frac{\partial y}{\partial x}\right)^\top \nabla_y z \tag{4.12}$$

with $\frac{\partial y}{\partial x} \in \mathbb{R}^{n \times m}$ being the Jacobian of g. Thus, it is clear that the gradient of z with respect to x can be calculated using the matrix-vector product of the Jacobian $\frac{\partial y}{\partial x}$ and the gradient $\nabla_y z$. The backpropagation algorithm basically only consists of performing such matrix-vector products. Although the parameters in ϑ can have arbitrary dimensions, the above chain rule suffices. Indeed, matrices and tensors can be transformed into a vector by rearranging their elements. For example, a $n \times m$ matrix can be transformed row-wise into a vector of length $n \cdot m$.

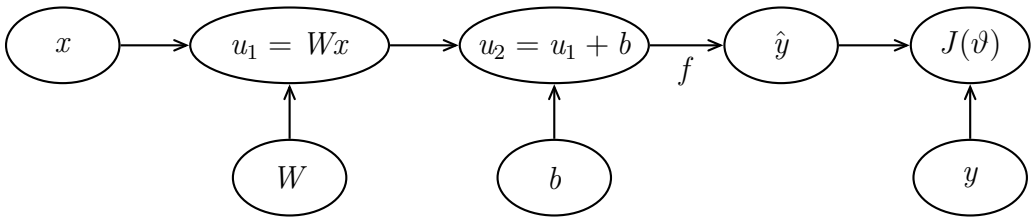


Figure 4.5: Computational graph of a perceptron

In order to describe the procedure of the backpropagation algorithm, a computational graph is considered in which each scalar variable is represented by its own node. Accordingly, each individual element of a matrix or vector is described by a separate node. Also the parameters from ϑ are contained in the computational graph. The indexing of the individual nodes $u^{(i)}$ is chosen in such a way that the sequential evaluation from $u^{(1)}$ to $u^{(n)} = J(\vartheta)$ allows the entire forward computation of the neural network including the application of the loss function. To perform backpropagation the algorithm starts at the last node $u^{(n)}$ of the graph and computes the derivatives $\frac{\partial J}{\partial u^{(i)}}$ using the chain rule as described above in reverse order up to the first node $u^{(1)}$. The clever indexing of the nodes makes it possible to store intermediate results and reuse them when applying the chain rule. A simplified description of the algorithm is shown in fig. 4.6. There, the derivative $\frac{\partial J}{\partial u^{(i)}}$ is always stored in grad$[i]$. After the execution of the algorithm is completed, the required gradient $\nabla_{\vartheta} J(\vartheta)$ is therefore available.

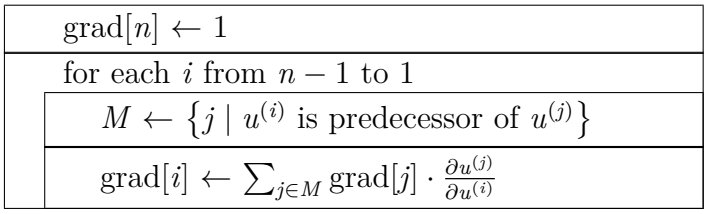


Figure 4.6: Simplified notation of the backpropagation algorithm as a Nassi–Shneiderman diagram

In practice, the nodes in the computational graph do not represent scalar variables, but instead the vectors, matrices and tensors processed and contained in the neural network. Then the gradient of $u^{(n)} = J(\vartheta)$ to every preceding node $u^{(i)}$ can be easily calculated by considering the direct successor $u^{(j)}$ of $u^{(i)}$. To do this, simply multiply the gradient $\nabla_{u^{(j)}} u^{(n)}$, which is already obtained and stored in an earlier step of the algorithm, by the Jacobian of the operation that generated $u^{(j)}$ from $u^{(i)}$. In this way, for each previous operation of $u^{(n)}$, the respective Jacobians are multiplied up until all parts of the required gradient $\nabla_{\vartheta} J(\vartheta)$ have been determined. If $u^{(n)}$ can be reached from $u^{(i)}$ via multiple paths, the gradients of the paths arriving at $u^{(i)}$ are summed up according to the chain rule.

Many libraries add additional nodes to the computational graph when running the backpropagation algorithm, which compute the individual derivatives of the nodes symbolically (see fig. 4.7). The calculation of the gradients with concrete numerical values can then be done at a later time. The advantage of this concept is that a repeated execution of the backpropagation algorithm on the graph allows the computation of higher derivatives. In order to be able to insert the derivatives as nodes into the graph, libraries contain special functions for each tensor operation, which calculate the product of the Jacobian and the desired vector according to chain rule. In this way, the backpropagation algorithm itself does not need to know a single derivative at any time. The algorithm must only call the additional gradient function of the individual operations.

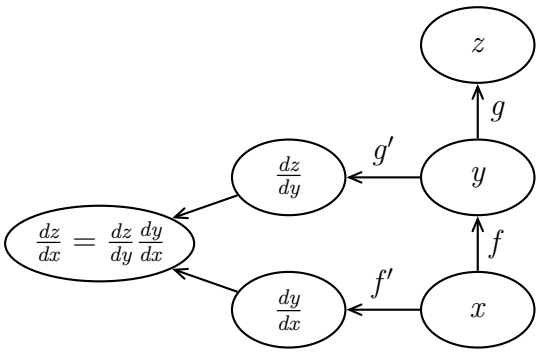


Figure 4.7: Example of a computational graph including derivatives

4.3.4 Convolutional layers

Besides fully-connected layers, another common type of layers in a neural network are convolutional layers. They compute convolution operations on the input. In this work, discrete one dimensional convolution is employed in convolutional layers. The convolution operation needs an input vector a and a kernel $k \in \mathbb{R}^n$, which is typically smaller than a with odd n. The kernel is the free parameter of a convolutional layer to be learned during the training process. An element of the output vector b is then computed

$$b(i) = (k * a)(i) = \sum_{l=1}^{n} a(i + m - l)k(l) \tag{4.13}$$

with $m = \frac{n+1}{2}$. The vectors a, b and k map to their vector elements and map invalid indices (which may occur in the sum in eq. (4.13)) to zero. In a convolutional layer, the convolution operation is followed by the application of an activation function, which is applied element-wise on the output vector b. The description of convolutional layers in this work is limited to the one dimensional case, but they can also be applied to higher dimensional data. Implementations of convolutional layers tend to implement a cross-correlation instead of a convolution. Cross-correlation is the same as convolution, but the kernel is flipped. Since the kernel is the free parameter learned during the training process, flipping is not important and can be ignored.

In cross-correlation, the output can be computed as follows: The kernel slides across the input vector and in every step, the kernel is multiplied element-wise with the corresponding input patch. The products are then summed up and are written into the output vector. The hyperparameters in a convolutional layer are usually the kernel size n and the stride, which describes the step size used for sliding the kernel. Besides, the kernel can have several layers (if the kernel is applied to multiple input layers), which allows the extraction of multiple information from the input at once. These layers are called filters. Also the output can have multiple layers, which are called channels and represent multiple features. The number of filters and channels are also hyperparameters in convolutional layers.

An advantage of convolutional layers is the reduced number of free parameters which have to be learned in training compared to fully-connected layers. The kernel in a

convolutional layer is usually much smaller than a weight matrix in a fully-connected layer. This results in a more efficient training and less memory usage. The kernel is applied to the entire input, so the weights in the kernel are shared among the input values (parameter sharing). This leads to a property called equivariance to translation, which describes the fact that a change in the input data causes the output to change the same way. If the values in the input vectors are shifted, the output will be shifted the same way. This stands in contrast to fully-connected layers, where this is in general not the case.

Convolutional layers can be represented in a computational graph as well. Therefore backpropagation and the optimization methods described in section 4.3.2 can be applied to convolutional layers too.

4.3.5 LSTM units

The long short-term memory (LSTM) architecture [HS97] belongs to the class of recurrent neural networks (RNNs). RNNs in general are used to process sequences, which is crucial for applications in language processing for instance. A sequence is usually composed of time steps, where each time step provides an input vector x^t. They are fed one by one into the RNN. The idea of RNNs now is to keep information gained from earlier time steps and use it for a prediction. To achieve this, a RNN typically stores a hidden state h^t which contains historical information and which is manipulated in each time step t in the sequence. The network passes the hidden state to the next time step, so it can be considered to be an additional input to the network (apart from x^t), which is automatically generated and passed by the network. This means when processing the vector input x^t from a sequence, the hidden state from the previous time step h^{t-1} will be also processed in the network.

In a simple RNN architecture, the previous hidden state h^{t-1} and the new input x^t are concatenated to form a single input vector. This vector holds historical and new information from the sequence. It is usually processed through a fully-connected layer using the activation function

$$\tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^x + e^{-x}}{e^x - e^{-x}}. \tag{4.14}$$

The tanh activation ensures that the output values from that layer range from -1 to 1. This prevents the output values from exploding, when they are processed through many time steps in the RNN. A simple RNN architecture is shown in fig. 4.8.

RNNs as described above can only keep information over a short distance of time. In other words, the information gained from the beginning of a sequence may already be discarded towards the end of the sequence, even though the information may be important to keep. Simple RNNs cannot access information which was fed into the network much earlier. This phenomenon is referred to as short-term memory and can be disadvantageous when processing texts. Another problem with RNNs is the training based on backpropagation. The gradients computed during training can vanish or explode when

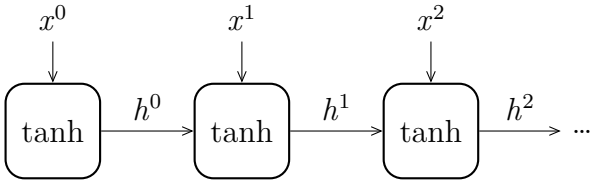


Figure 4.8: Visualization of a simple RNN architecture. The nodes refer to a fully-connected layer with tanh activation.

they are propagated back through time. This can lead to unstable training and can prevent convergence.

LSTM can be a solution to the problem regarding short-term memory pointed out above. The main idea of LSTM is that gates regulate the information processing flow in the network architecture. The gates are meant to learn which data in a sequence is relevant and which is not. Thus, the LSTM can keep the important information only.

The LSTM is designed as a cell. A simplified visualization of the LSTM cell is shown in fig. 4.9. It consists of a cell state c^t, a hidden state h^t and the gates mentioned before. The cell state is propagated to the next step in the sequence, just as the hidden state. It holds the information over a long distance, it works as the memory of the LSTM cell. The gates manipulate the cell state: they decide what information to forget, add or give as the output. For each of this tasks, the LSTM cell has a specific gate: the forget gate f^t, the input gate i^t and the output gate o^t. The cell is calculated as

$$f^t = \sigma\left(W_f \cdot \left[h^{t-1}, x^t\right] + b_f\right) \tag{4.15}$$

$$i^t = \sigma\left(W_i \cdot \left[h^{t-1}, x^t\right] + b_i\right) \tag{4.16}$$

$$o^t = \sigma\left(W_o \cdot \left[h^{t-1}, x^t\right] + b_o\right) \tag{4.17}$$

$$\tilde{c}^t = \tanh\left(W_c \cdot \left[h^{t-1}, x^t\right] + b_c\right) \tag{4.18}$$

$$c^t = f^t \circ c^{t-1} + i^t \circ \tilde{c}^t \tag{4.19}$$

$$h^t = o^t \circ \tanh\left(c^t\right) \tag{4.20}$$

with weight matrices W and bias vectors b, the sigmoid function σ and the operator $\circ$ representing element-wise multiplication. The sigmoid function is defined as

$$\sigma(a) = \frac{1}{1 + e^{-a}} \tag{4.21}$$

and outputs values between 0 and 1. It is applied element-wise. Each gate and its computation will be described in detail in the following paragraphs.

The forget gate f^t decides which information is kept and which information will be discarded in a step. As defined in eq. (4.15), it takes the hidden state h^{t-1} and the input x^t and processes it through a fully-connected layer having a sigmoid activation. The resulting vector (having values between 0 and 1) is then multiplied with the cell state c^{t-1} in the first part of eq. (4.19). This makes the cell state to forget parts of the stored information where the sigmoid outputs are close to 0. The information in the cell state multiplied with values near 1 will be kept.

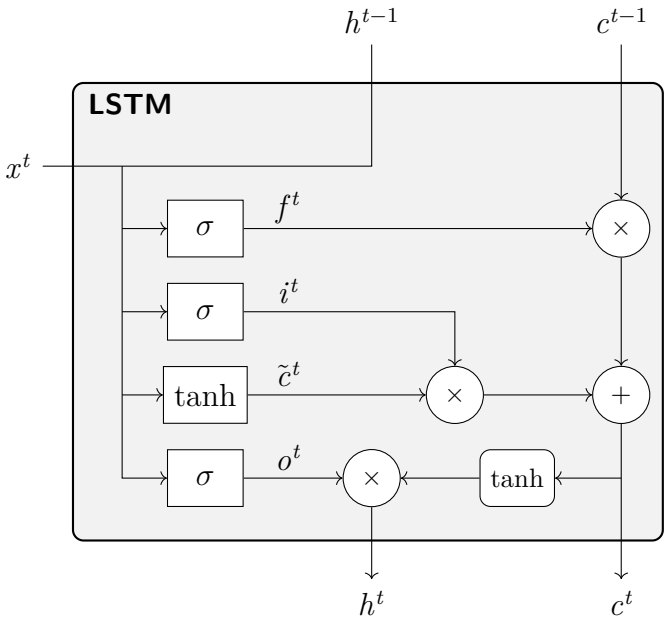


Figure 4.9: Visualization of a LSTM cell. The rectangular nodes (on the left) represent a fully-connected layer, whereas the rounded shapes (on the right) represent operators and function calls.

The input gate i^t puts new information into the cell state. This is done by using the hidden state h^{t-1} and new input x^t again which is fed into a fully-connected layer with sigmoid activation (as shown in eq. (4.16)). The resulting vector decides which new information will be added to the cell state. The new information for the cell state $\tilde{c}^t$ is calculated from the hidden state and input using a fully-connected layer with tanh activation (eq. (4.18)). Both vectors are multiplied element-wise in the latter part of eq. (4.19), so the output of the sigmoid layer decides which information of the tanh layer will be adopted into the cell state. The final vector is then added to the cell state in eq. (4.19). This defines the new cell state which is fed to the next step.

The last gate, the output gate o^t, defines the hidden state h^t which will be processed to the next step. The old hidden state h^{t-1} and the input x^t is again processed through a fully-connected layer with sigmoid activation (eq. (4.17)). This is then multiplied element-wise with the tanh of the cell state in eq. (4.20). The hidden state h^t also defines the output of the cell which can be used as a prediction.

All in all, the LSTM cell is used for recurrent architectures and overcomes problems of standard RNN networks. This is achieved by introducing a cell state which can be interpreted to be the cell's memory and is manipulated via different gates. This section introduced the standard LSTM cell, but there are many other advanced models and modified cells based on LSTM. A modified version of the LSTM cell for solving the microstructure imaging task will be presented in section 5.5.

5 Reference Methods

5.1 AMICO

Accelerated Microstructure Imaging via Convex Optimization (AMICO) [Dad+15] is a framework accelerating calculations for fitting microstructure models like NODDI to dMRI measurements. The general idea is to reformulate existing techniques in terms of linearization, i. e. approximate classical microstructure imaging techniques by a system of linear equations which can then be solved efficiently using convex optimization algorithms.

Before presenting the AMICO approach itself, some basics of spherical deconvolution and fiber orientation reconstruction need to be explained. The dMRI signal $S(q)$ (dependent on the gradient vector q) can be modeled as a convolution between a fiber orientation distribution (FOD) function $f : \mathbb{S}^2 \to \mathbb{R}^+$ and a response function $K(\cdot, u)$ that gives the signal attenuation depending on the orientation $u \in \mathbb{S}^2$ of a single fiber.

$$S(q) = S_0 \int_{\mathbb{S}^2} K(q, u) f(u) du \tag{5.1}$$

S_0 describes the signal without diffusion weighting and $\mathbb{S}^2$ is the unit sphere. The FOD can be written as a linear combination of so-called atoms, i.e. N different basis functions, which gives

$$f(u) = \sum_{i=1}^{N} w_i f_i(u). \tag{5.2}$$

If the response functions $K(\cdot, u)$ can be estimated beforehand, eq. (5.1) can be formulated as a system of linear equations:

$$y = \Phi x + \eta, \tag{5.3}$$

where $y \in \mathbb{R}_+^M$ is a vector of normalized q-space measurements, $\Phi \in \mathbb{R}^{M \times N}$ is the linear operator modeling the convolution operation, $x = (w_1, w_2, \ldots)^T$ are the FOD coefficients which need to be estimated and η describes the acquisition noise. The operator Φ is also called dictionary and consists of elements $\phi_{ij} = \int_{\mathbb{S}^2} K(q_i, u) f_j(u) du$. The linear problem in eq. (5.3) can now efficiently be solved using convex optimization techniques which try to solve the regularized optimization problem

$$\underset{x \geq 0}{\arg\min} \quad \frac{1}{2} \|\Phi x - y\|_2^2 + \lambda \Psi(x). \tag{5.4}$$

Regularization is controlled via a regularization function $\Psi(x)$ (e. g. L2 regularization $\Psi(x) = \|x\|_2^2$) and a regularization parameter $\lambda > 0$. The positivity for x must be ensured due to the interpretation of its coefficients as volume fractions. Most deconvolution models have ill-conditioned linear systems or the dictionary is under-determined due to the reduction of scan time. Therefore regularization is necessary in most cases to increase the stability of reconstruction or to put in prior knowledge about the solution. L1 regularization promotes sparsity, whereas L2 regularization accounts for the reduction of ill-conditioning in Φ.

The first step of the AMICO method is the estimation of the major fiber orientation $\mu \in \mathbb{S}^2$ in a voxel. AMICO uses standard algorithms like DTI (diffusion tensor imaging, [BML94]) for this task, since they give accurate and robust results. After μ is computed, the next step is to build a dictionary Φ depending on that specific orientation μ. To achieve this, the linear operator Φ is calculated and stored for a canonical orientation beforehand. For a particular orientation μ, the atoms in Φ need to be rotated to match the actual orientation μ. This is done via a rotation operation $\mathcal{R}_\mu$, so the final dictionary is computed as $\tilde{\Phi} = \mathcal{R}_\mu(\Phi)$. The dictionary $\tilde{\Phi}$ is then plugged in the least squares formulation in eq. (5.4), which finally needs to be solved. To save time during the construction of the dictionaries for each voxel, the implementation of AMICO not only stores precomputed dictionaries in a canonical orientation, but also rotated versions of the dictionaries with an angular resolution of 1°, which can be accessed via lookup tables.

In the following the construction and evaluation of the NODDI model in AMICO will be described. The NODDI model consists of three compartments: the intra-cellular, extra-cellular and the isotropic compartment (see section 3.2). The general idea in AMICO is to separate the dictionary Φ into distinct parts for each compartment. However, the extra-cellular compartment depends on the intra-cellular volume fraction v_{ic} and the Watson distribution parameter κ, which are parameters from the intra-cellular compartment. Therefore the intra-cellular and extra-cellular compartment must be considered combined, i. e. as a single compartment. This leads to a partitioning of the dictionary into two parts:

$$\Phi = \left(\Phi^t \middle| \Phi^i\right) \tag{5.5}$$

The first submatrix Φ^t contains the coupled intra-cellular and extra-cellular compartment. Each column contains the signal attenuation for a particular parameter configuration of v_{ic} and κ. In the standard NODDI implementation in AMICO, 12 values for v_{ic} and 20 values for κ are considered, which leads to a total of 144 columns for Φ^t. The second part Φ^i models the isotropic contribution to the signal. Since the isotropic diffusivity is estimated with a fixed value, the submatrix only consists of a single column.

With the dictionary described in eq. (5.5) the convex optimization problem for NODDI results in

$$\underset{x \geq 0}{\arg\min} \quad \frac{1}{2} \left\| \tilde{\Phi} x - y \right\|_2^2 \; + \; \lambda_1 \|x\|_1 \; + \; \frac{1}{2} \lambda_2 \|x\|_2^2 \tag{5.6}$$

where there is both L1 and L2 regularization applied. Even though L1 regularization is used here to promote sparsity in x, it is important that the solution is only sparse in

the first part of x corresponding to Φ^t, since the isotropic contribution must be able to take any value without restrictions. Since standard solvers (just like the one used for AMICO) only offer L1 regularization on the entire vector x, the following strategy was implemented in AMICO to overcome this problem:

1. Since the isotropic compartment should not be affected by sparsity, eq. (5.6) is solved without regularization ($\lambda_1 = \lambda_2 = 0$) to estimate the isotropic volume fraction v_{iso}.

2. The isotropic contribution is then removed from the signal by computing $y = y - \tilde{\Phi}^i v_{\text{iso}}$. Equation (5.6) is now solved again using the sparsity prior to identify the support of the solution, which means the subset of atoms which can explain the signal in y. The solution in x is biased, since the L1 norm underestimates the true values.

3. Finally, eq. (5.6) is then solved for the last time, without the sparsity prior and only on the support of the solution that was obtained in step 2.

Similar to eq. (5.5), the solution vector x can be partitioned into $(x^t|x^i)$. The computation of the parameters of the NODDI model then results in

$$v_{\text{ic}} = \frac{\sum_j f_j x_j^t}{\sum_j x_j^t}; \qquad \kappa = \frac{\sum_j k_j x_j^t}{\sum_j x_j^t}; \qquad v_{\text{iso}} = \sum_j x_j^i \tag{5.7}$$

with f_j and k_j as the intra-cellular volume fraction and Watson distribution parameter, respectively, of the j-th atom used in Φ^t.

The AMICO method shows a significant reduction of computation time compared to existing earlier microstructure imaging techniques (e. g. the original NODDI [Zha+12]) and achieves the same accuracy and robustness. The AMICO approach is flexible and therefore can easily be adapted to many other microstructure models apart from NODDI.

5.2 Multilayer Perceptron

Classical data processing approaches for microstructure imaging are disadvantageous because of the long scan time that is needed to produce enough data to get reliable results. In [Gol+16] multilayer perceptrons (MLPs, section 4.3.1) were proposed to overcome this problem. The authors showed that MLPs are not only capable of estimating microstructure model parameters based on dMRI measurements, but also are able to do this with less data points and therefore reduced scanning time.

The idea of this approach is that a machine learning algorithm in theory is able to represent any mapping between input and output if the relationship exists. The use of deep learning in the context of microstructure imaging is called q-space deep learning. Each voxel in a dMRI measurement is treated as an individual sample. The MLP is meant to predict the parameters of the microstructure model directly from dMRI measurements. So the measurements are fed into the network as input and the output

represents the scalar quantities of the microstructure model which need to be estimated. Training such a network will try to solve the inverse problem directly. Figure 5.1 shows the architecture of a MLP used for q-space deep learning on the NODDI model. There are six output neurons: the first three predict the orientation μ in cartesian coordinates, the remaining neurons predict ODI, v_{ic} and v_{iso}. The number of input neurons depends on the acquisition scheme that was used in the data generating process. This implies that for a different sampling scheme a new network must be trained. During the training process, the weights in the network will be adjusted so that the outputs of the network approximate the labels in the training set (see section 4.3.2). Since estimating the parameters of a microstructure model is a regression task, the mean squared error is used as a loss function. The activation function in the hidden layers is ReLU.

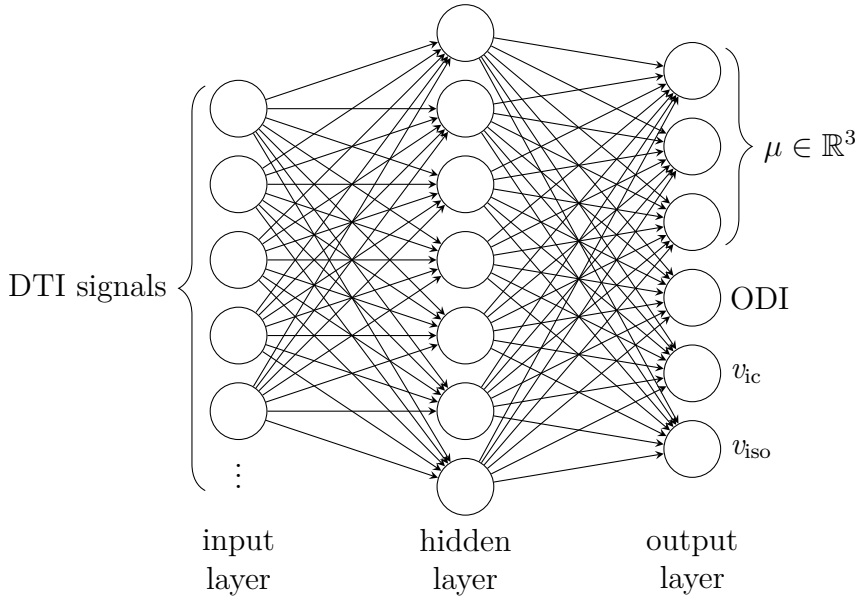


Figure 5.1: Example of a MLP architecture for the NODDI model

Before training, the data is scaled to have a mean of 0 and a variance of 1 for each feature. Both input and output data is scaled independently. Scaling can be benificial for the training of a neural network. Since each voxel is treated as an individual sample in this setting, the location of a voxel in the image is unknown. Nevertheless, a layer of the MLP can be interpreted as a convolution with a kernel size of 1×1 on a whole dMRI image, ignoring non-brain voxels. Making predictions with the network after training is deterministic and fast compared to other imaging techniques. Another advantage is that, in contrast to other methods, no information is discarded between steps of the process pipeline, since the MLP directly predicts the desired quantities from the measurements.

5.3 Tandem

Another architecture for deep inverse modeling is the tandem model [Liu+18]. It addresses problems of traditional inverse models. A major problem is the non-uniqueness of the data in inverse problems. This means that in datasets for inverse problems, one single input can lead to different outputs. The performance of standard neural networks suffers from this ambiguity. This also affects deep inverse models like the MLP in section 5.2. Besides, traditional methods are performed iteratively, computing simulations in every single iteration step. The new simulations are compared to the input and the algorithm adjusts its prediction calculation rule. This procedure causes the need to compute thousands of simulations, which may be very computationally expensive. Thus, traditional approaches can be very slow, especially for increasing sizes of datasets.

Even though standard neural networks cannot overcome the problem of non-uniqueness in the inverse setting, they can help to make simulations faster. Neural networks can be used to replace a simulator cheaply. A network which is trained to approximate the simulator is referred to as a forward model. A forward model can often be trained without having to worry about non-uniqueness in the dataset, since the problem mostly appears when training inverse models. The training set used for the forward model has parameters as input values and simulation results as output values. Every set of parameters has its unique simulation result. Admittedly, the generation of the dataset includes many simulations that need to be computed, but this is a one-time cost. Once generated, the dataset can be reused for the training of different architectures and each trained network can produce simulations at nearly zero cost. There are no repeated computations of simulations needed, which makes traditional methods using a neural network as a forward model much faster.

However, neural networks struggle with the non-uniqueness when considering the inverse problem. This is due to the fact that most neural networks are deterministic and thus cannot produce two different outputs to a single input. When training deep inverse models, such conflicting samples in the dataset can lead to a poor performance. A simple idea to address this problem is to split the dataset into distinct groups, which do not contain samples with the same input values. This has been investigated in [Kab+08] and led to limited success on small datasets. Removing conflicting training samples from the dataset can be disadvantageous, since this may lead to inconsistency. The tandem approach in contrast tries to combine an inverse design with a forward model to overcome this limitation of non-uniqueness.

The tandem architecture consists of two neural networks. One of these networks is a forward model as described above. The other one is a network in the traditional inverse design (like MLP in section 5.2), which is prepended to the forward model. This inverse network is referred to as a backward model. In fig. 5.2 the tandem architecture is visualized. The tandem model is a design for solving inverse problems, so the input y will be dMRI signals in the case of microstructure imaging. In the intermediate layer, just in between the backward and the forward model, the neurons x represent the parameters of the microstructure model that need to be estimated. The number of neurons in this layer matches the number of scalar quantities to be estimated. The overall output of

the network $\hat{y}$ are dMRI signals again, but this time they are not simply given by the input but rather an estimation of signals based on the quantities in the intermediate layer. The amount of neurons in the input and output layer must match, since they both describe the same kind of dMRI signals in principle, but one being the given input and the other being a resimulation. The training of a tandem model is not the same as in standard neural networks, instead it must be divided into two steps. The first step is to train the forward model separately, i. e. isolated from the backward model. After the training is completed, the weights of the forward model are fixed to their current values. Now the second step is to train the backward model in an end-to-end manner. This means the parameters that come out of the backward model in the intermediate layer are not compared to the training data, but instead they are processed through the forward model to produce estimated dMRI signals. These estimated signals based on the prediction of the desired quantities in the intermediate layer are compared to the input that was given into the backward model, so the loss function used for backward training usually is

$$l(y) = (\hat{y} - y)^2 . \tag{5.8}$$

This in fact is the key factor of the tandem model that helps to overcome the problem of ambiguity: the conflicting output values of the backward model are not compared to the training data, just their simulations. By only comparing the simulated signals to the original signals the tandem model tends to produce parameter predictions that lead to a similar simulation output instead of similar model quantities. The predictions in the intermediate layer of the network are not accessed directly during the backward training. After training is completed, the parameter predictions of the tandem model can be obtained from the intermediate layer. Since scaling is again beneficial for the training this network, the input and output data is scaled to have a mean of 0 and a variance of 1.

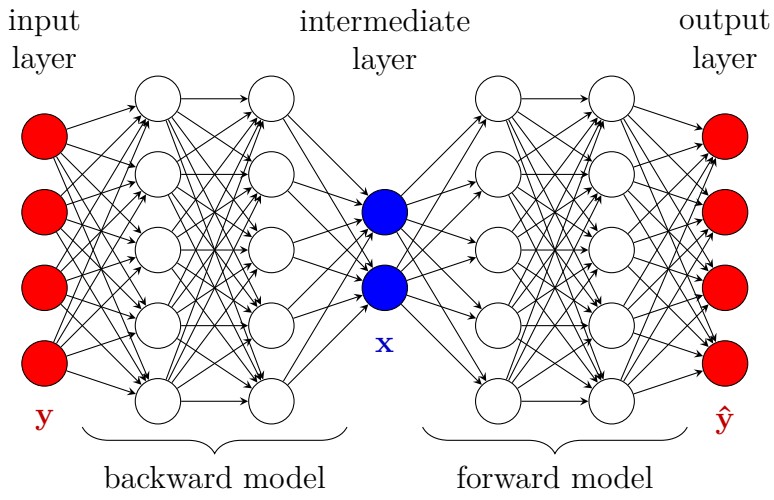


Figure 5.2: Example of a tandem model architecture

The tandem network design can be interpreted as a self-supervised learning approach. The reason is that it uses the forward model as a surrogate for a real simulator and therefore can train the backward model in an end-to-end manner. If a forward model of a problem is given, the tandem approach only needs to train the backward model. The backward training however is done only using the output values of the dataset. No input parameters are needed. In principle, they can be ignored during the backward training process. They are only needed if a forward model needs to be trained beforehand, but they do not directly influence the inverse part of the network.

5.4 Neural Adjoint

The Neural Adjoint (NA) method [RPM21] is another method to find inverse solutions using neural networks. In [RPM21] it was shown in a benchmark that NA outperforms other deep inverse models like the Tandem model discussed in section 5.3. It was also observed that NA can more accurately localize inverse solutions, because NA more tends to search through the whole x space compared to other deep inverse models. The underlying idea of NA is based on the classical Adjoint method for inverse problems.

The key factor of Adjoint based methods is to find an analytical gradient of the real forward model or simulator with respect to the input parameters. Having this gradient, the next step is to iteratively find an inverse solution adjusting the input using this gradient. The goal is to find a locally optimal inverse solution to the given problem. The challenge here is that the desired gradient may be difficult to find. The NA method also uses the gradient of a forward model, but the forward model now is a neural network that approximates the real simulator and works as a surrogate (similar to the forward model of the Tandem model in section 5.3). Since the forward model in NA is now a neural network, the gradient with reference to the input is easy to compute. There is no need to derive an analytical gradient, because the backpropagation algorithm (section 4.3.3) provides the numerical gradient at relatively low cost. Modern deep learning libraries, which can be used to implement NA, estimate the desired gradients efficiently.

The NA method can be described in two steps: the first step is to train a forward model and the second step is the inference of an inverse solution. In the first step the forward model, a generic neural network, is trained on the training set to approximate the forward process of the considered problem. This is similar to the training of a forward model for the Tandem model in section 5.3. The forward model must only be trained once, so the training cost is again a one-time cost.

In the second step NA tries to find an inverse solution $\hat{x}$ given a particular output y. This is done using a gradient descent algorithm. NA starts from different random locations in x space and then iteratively updates the input $\hat{x}$ to find a locally optimal value that produces an output $\hat{y}$ close to y. In other words, the input $\hat{x}$ is optimized to lead to a prediction $\hat{y}$ that corresponds to the desired output y. The update rule is

$$\hat{x}^{i+1} = \hat{x}^i - \alpha \left. \frac{\partial l(\hat{y}(\hat{x}^i), y)}{\partial x} \right|_{x=\hat{x}^i}, \tag{5.9}$$

where l is a loss function comparing the true y and the prediction $\hat{y}$, which is dependent on $\hat{x}^i$, and α describes a learning rate. The gradient can be calculated using the backpropagation alogorithm (section 4.3.3). To make the learning rate adaptive, conventional methods like Adam (section 4.3.2) can be used. It is important to understand that the iteration adjusts the input instead of the weights of the neural network. So after the training of the forward model is finished in the first step of NA, the weights are fixed for the second step. The initial value $\hat{x}^0$ for the iteration is drawn from a probability distribution Γ. The distribution Γ could be a Gaussian or uniform distribution, but ideally the distribution Γ should match the original distribution of x in the dataset or data generating process. However, this distribution is not always known. The iteration stops after convergence is reached. One completed iteration following the update scheme in eq. (5.9) represents one single estimation of an inverse solution. For every sample of y, a separate iteration process must be calculated until convergence. Therefore NA results in high computation costs and appears to be a time consuming algorithm.

However, there are challenges with the NA approach as described above. One challenge is that the algorithm is sensitive to the initialization $\hat{x}^0$. Some initializations result in a poor minimum after the iteration or they do not even converge. To overcome this problem, there are thousands of solutions extracted at once by the algorithm. This means that NA calculates multiple iteration processes for a single sample y starting at different initial values $\hat{x}^0$. At the end, the best solution is chosen considering the output of the forward model. The solution which produces the output closest to y (in terms of the loss function l) will be chosen to be the inverse solution, the results of the other iterations will be discarded. This procedure does not necessarily significantly increase the computation time for the inference of a solution, since the calculations can be parallelized on the GPU. Despite parallelization, NA is still a computationally expensive algorithm compared to other deep inverse models.

Another challenge is that the inverse solutions $\hat{x}$ may lie outside of the training sampling domain. This is due to the fact that the predictions of the forward model become highly inaccurate when x is not contained in the training data range. NA just tries to find an optimal solution $\hat{x}$, without any restrictions, which produces the desired output y. This however leads to high errors in x. The authors of [RPM21] address this problem by adding a boundary loss to the total loss, which is meant to penalize x values outside the training sampling domain. In other words, the boundary loss makes the solution to stay within the domain. The boundary loss can improve the reliability and accuracy of NA substantially. It is defined as

$$l_{\text{bnd}}(\hat{x}) = \text{ReLU}(|\hat{x} - \mu_x| - 2\sigma_x), \tag{5.10}$$

where μ_x is the mean input value of the training set and σ_x the standard deviation respectively. ReLU is the activation function of eq. (4.3) described in section 5.2. The boundary loss penalizes x which deviate more than $2\sigma_x$ from μ_x. It is important to mention that the boundary loss is only added in the second step of NA, the inference of an inverse solution, which can be seen in the visualization of the NA method in fig. 5.3. The boundary loss is not used for the training of the forward model. There are also some

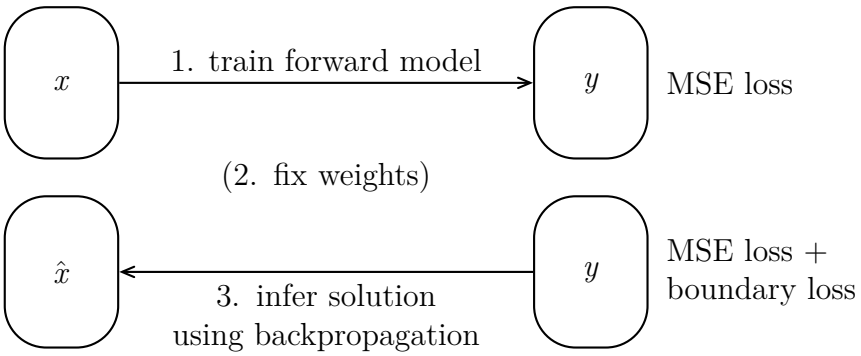


Figure 5.3: Visualization of NA method

limitations with the boundary loss that must be kept in mind. The boundary loss in eq. (5.10) assumes the data domain to be approximately a hypercube. If this assumption is not true, the boundary loss can be less effective. The boundary loss also assumes that the forward model is equally accurate for x inside the domain and this accuracy drops uniformly in all directions outside the domain. These assumptions will in general not be met by the data and the forward model.

In summary, it can be stated that NA is able to produce very accurate and reliable inverse solutions at the cost of higher computation time. The computational effort arises because of the optimization problem that must be solved for every single sample. Therefore NA is not recommended to be used for time sensitive applications. With the forward model being given (like in Tandem, section 5.3), the NA approach can be interpreted as a self-supervised method, since NA tries to minimize the error between the prediction $\hat{y}$ and the desired output y for the inference of solutions, ignoring x.

5.5 MESC-Net

[YLC19] presents a deep neural network architecture called MESC-Net (Microstructure Estimation with Sparse Coding) which is motivated by iterative optimization where historical information is incorporated. The architecture is specialized for its application in microstructure imaging. Iterative optimization strategies used for microstructure imaging often have fixed update schedules based only on the previous iteration. The idea of MESC-Net now is to unfold the iteration and incorporate historical information adaptively, i. e. outcomes from earlier iteration steps than just the previous step. This can improve convergence as well as performance compared to methods using fixed update rules. The aim of MESC-Net was to design a generic neural network architecture which can be adapted easily to different microstructure models and provides reliable estimations of tissue microstructure, even when using a reduced number of diffusion gradients.

The approach of MESC-Net underlies two assumptions: first, diffusion signals can be sparsely represented using a dictionary (for spatial and angular domain), and second, estimation of tissue microstructure is possible using that sparse representation. MESC-

Net tries to imitate sparse coding as how it is done in AMICO (see section 5.1). The main idea in sparse coding is to find a dictionary so that the diffusion signals can approximately be expressed as

$$y = \Phi x \tag{5.11}$$

at each voxel separately, with Φ being the dictionary, x the sparse vector representation and y the diffusion signals. The design of the dictionary by hand (e. g. as in AMICO) implicitly provides a rule how to compute microstructure quantities from the sparse vector x. This linearization using a dictionary can also be generalized to image patches. Then, the diffusion signals of the voxels in the image patch are concatenated in a single vector. The design of a proper dictionary now is not straightforward as the difficulty of dictionary design is increased when using image patches. Therefore one goal of MESC-Net is to find a sparse coding without the need to explicitly design dictionaries by hand. The architecture should also be generic, so that MESC-Net can easily be applied to different microstructure models.

The MESC-Net consists of two stages. The stages will be presented shortly in this paragraph to get an overview of the model. They are described in more detail in the following paragraphs. The first stage of MESC-Net takes an image patch as an input and computes the spatial-angular sparse representation for each input using learned weights. Since the MESC-Net will be applied to synthetic data in this work, a single voxel is processed instead of an image patch. The input to MESC-Net are the diffusion signals obtained from dMRI. In this stage, historical information is adaptively incorporated in a way which is comparable to modified LSTM units (section 4.3.5). In the second stage, the sparse representation is mapped to the desired microstructure quantities. Since the goal is to have a generic network architecture, the interpretation of the sparse representation is unknown. Therefore, a bunch of fully-connected layers is used to determine the scalar microstructure measures. This procedure can be compared to convolutional neural networks: after the convolutional layers extracted features in an image, fully-connected layers are used to interpret these and transform them into the desired output. The separation of MESC-Net into two stages is only for understanding, the stages are learned jointly minimizing the mean squared error of the microstructure quantities at the end of the network.

For a deeper insight into the first stage of MESC-Net, the assumption that diffusion signals can be sparsely represented must be revisited. Given the dictionary, the coefficients of the sparse representation can be obtained from solving the L1 regularized least squares problem (see AMICO, section 5.1)

$$\hat{x} = \underset{x \geq 0}{\arg\min} \ \|\Phi x - y\|_2^2 + \gamma \|x\|_1 \tag{5.12}$$

where sparsity is controllable by adapting the parameter γ. This minimization problem can iteratively be solved using IHT (Iterative Hard Thresholding) [BD08]. The update scheme for the sparse representation x is

$$x^t = h_\lambda \left(Wy + Sx^{t-1} \right) \tag{5.13}$$

with the iteration index t, matrices W and S (which can be obtained using the dictionary Φ) and the thresholding function

$$[h_\lambda(a)]_i = \begin{cases} 0 & \text{if } |a_i| < \lambda \\ a_i & \text{if } |a_i| \geq \lambda \end{cases} \tag{5.14}$$

where λ again implicitly controls sparsity. The iteration in eq. (5.13) is a nonadaptive strategy with a fixed update rule. MESC-Net now unfolds this iteration and defines the matrices W and S as free model parameters which are additionally shared among the layers of MESC-Net. That means the matrices will be learned during the training process instead of being predefined. Furthermore, MESC-Net incorporates historical information adaptively, as already mentioned before. To achieve this, two additional intermediate variables are introduced: c and $\tilde{c}$. Their update scheme is as follows:

$$\tilde{c}^t = Wy + Sx^{t-1} \tag{5.15}$$
$$c^t = f^t \circ c^{t-1} + i^t \circ \tilde{c}^t \tag{5.16}$$
$$x^t = h_\lambda(c^t) \tag{5.17}$$

The weight vectors f^t and i^t determine the influence of information from the previous and and current layer, the operator $\circ$ represents the element-wise product. The variable c^{t-1} contains historical information, i. e. the outputs from previous layers, and $\tilde{c}^t$ holds the information from the current layer. Both are linearly combined, which allows the interpretation as a momentum term. The variables f^t and i^t are based on the diffusion signals y and the previous sparse encoding x^{t-1}, but the exact dependency is learned during the training process. They are updated as follows:

$$f^t = \sigma(\, W_{fx}x^{t-1} + \, W_{fy}y) \tag{5.18}$$
$$i^t = \sigma(\, W_{ix}x^{t-1} + \, W_{iy}y) \tag{5.19}$$

with σ being the sigmoid function defined in eq. (4.21). The matrices W_{fx}, W_{fy}, W_{ix} and W_{iy} are again learned from data. The updates in eqs. (5.18) and (5.19) are equivalent to a modified version of a LSTM unit, where i^t can be interpreted as an input gate, f^t as a forget gate and the output gate is missing. This modified LSTM cells is referred to as a M-LSTM unit. Besides, there is the constraint $x \geq 0$ added to ensure the sparse interpretation. Therefore, h_λ is simplified to a thresholded ReLU function. The M-LSTM unit is visualized in fig. 5.4. It is important to mention again that all the matrices of the M-LSTM update scheme described above are learned during the training process and do not require to build a dictionary by hand, unlike AMICO in section 5.1. For the matrices holds W, W_{fx}, $W_{fy} \in \mathbb{R}^{N\times M}$ and S, W_{ix}, $W_{iy} \in \mathbb{R}^{N\times N}$, where M is defined from the input size and N describes the dictionary size. The size of the dictionary is a hyperparameter, but $N = 301$ is recommended. The first stage of MESC-Net consists of 8 M-LSTM cells as described in eqs. (5.15)–(5.19). All the weight matrices are shared among the layers. The authors of MESC-Net recommend the initialization $c^0 = 0$ and $x^0 = 0$.

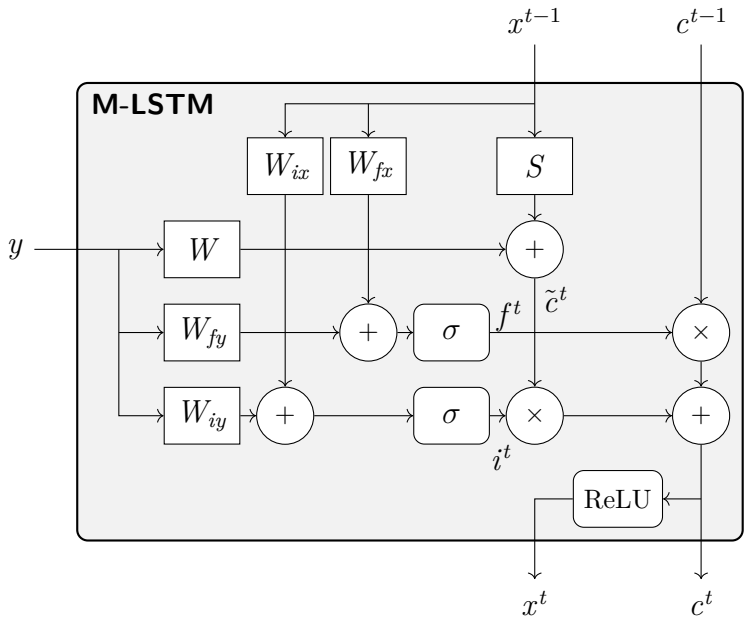


Figure 5.4: Visualization of M-LSTM unit

Due to the learned matrices in the first stage, the mapping from the sparse representation to the microstructure quantities is unknown. Therefore fully connected layers using the ReLU activation function are appended to the first stage. They define the second stage. The authors describe the analogy of the second stage to the fully connected layers at the end of a convolutional network. There, they define the mapping from the features which were extracted by the convolutional layers to the final prediction. The authors used three fully connected layers with 75 neurons each. A full visualization of the MESC-Net is shown in fig. 5.5.

Just as in all the other deep models presented in this chapter, the data is scaled to have a mean of 0 and a variance of 1. The Adam algorithm is used for the optimization of the MESC-Net. It is worth to note again that the two stages of MESC-Net are learned jointly to minimize the mean squared error between the predictions and the true

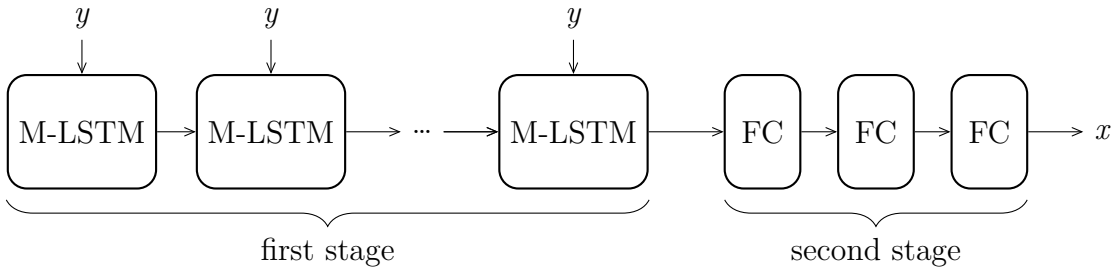


Figure 5.5: Visualization of MESC-Net architecture. M-LSTM refers to the modified LSTM unit and FC to a fully connected layer.

microstructure quantities.

All in all, MESC-Net learns a sparse representation for the diffusion signals and subsequently maps it to tissue microstructure measures. The network architecture is generic, what makes MESC-Net easily adaptive to various microstructure models. Essentially the MESC-Net consists of a modified version of LSTM cells called M-LSTM which enable the MESC-Net to consider historical information in its unfolded iterative updates.

6 Implementation Details

In this chapter, the implementation of the data simulation and the implementation and application of the methods presented in chapter 5 is described. In general, Python [Tea15] was used as the programming language. Deep learning models were implemented using PyTorch [Pas+19]. PyTorch is a library for deep learning in Python, which has a simple and flexible API and brings the numerical computations to GPUs with ease, having efficient memory usage. The computational graphs for neural networks are built dynamically and offer automatic differentiation. This is advantageous for implementing new or complex network architectures. For building microstructure models and simulating dMRI signals, the Python library DmiPy [Fic+18] was used. It comprises many common microstructure compartments (as described in section 3.1) which can easily be put together to form a multi-compartment model. Besides, it allows to formulate dependencies between different parameters in the microstructure model, which helps to implement common models like NODDI (section 3.2).

All the algorithms were run on a machine specialized for machine learning and deep learning for the HAF project. It has Intel(R) Xeon(R) Gold 6126 CPUs and Tesla V100 GPUs (PCIe, 32GB). All the neural networks were run on the GPUs, just AMICO was executed on the CPU only.

6.1 Data Simulation

To generate a dataset containing simulations of a microstructure model, the NODDI model was implemented using DmiPy. The implementation is shown in listing 6.1. As already described in section 3.2, the NODDI model consists of an intra-cellular, extra-cellular and an isotropic compartment. The stick model (section 3.1) can be used to implement the intra-cellular compartment, which is also available in DmiPy under the name `C1Stick` (line 5). The zeppelin model, which is used for the extra-cellular compartment, can be accessed as `G2Zeppelin`, which assumes hindered diffusion outside the axons. They both must be added to a bundle, since they will both be affected by a Watson distribution. To enforce the zeppelin model to use the tortuosity model of eq. (3.11) in section 3.2, the affected parameters (intra-cellular volume fraction, parallel and perpendicular diffusivity) must be explicitly named in the function call in line 8. The parameter names ending in `lambda_par` always refer to a parallel diffusivity, whereas parameters ending in `lambda_perp` refer to a perpendicular diffusivity. Finally, the isotropic compartment is realized simply via a ball model named `G1Ball` (line 4), which is added to the final multi-compartment model in line 9.

Listing 6.1: Implementation of NODDI model

```
class NoddiModel():
  def __init__(self, acq_scheme):
    self.acq_scheme = acq_scheme
    self.ball = gaussian_models.G1Ball()
    self.stick = cylinder_models.C1Stick()
    self.zeppelin = gaussian_models.G2Zeppelin()
    self.watson_bundle = distribute_models.SD1WatsonDistributed(models=[self.stick, self.
    ↪ zeppelin])
    self.watson_bundle.set_tortuous_parameter('G2Zeppelin_1_lambda_perp','
    ↪ C1Stick_1_lambda_par','partial_volume_0')
    self.model = MultiCompartmentModel(models=[self.ball, self.watson_bundle])

  def simulate(self, ball_lambda_iso, lambda_par, mu, odi, icvf, isovf):

    parameter_vector = self.model.parameters_to_parameter_vector(
      G1Ball_1_lambda_iso = ball_lambda_iso,
      SD1WatsonDistributed_1_C1Stick_1_lambda_par = lambda_par,
      SD1WatsonDistributed_1_G2Zeppelin_1_lambda_par = lambda_par,
      SD1WatsonDistributed_1_SD1Watson_1_mu = mu,
      SD1WatsonDistributed_1_SD1Watson_1_odi = odi,
      SD1WatsonDistributed_1_partial_volume_0 = icvf,
      partial_volume_0 = isovf,
      partial_volume_1 = 1-isovf
    )

    return self.model.simulate_signal(self.acq_scheme, parameter_vector)
```

The resulting NODDI model can be used to simulate dMRI signals by calling the function `simulate` defined in line 11. The parameters `ball_lambda_iso` and `lambda_par` refer to the isotropic diffusivity and the parallel diffusivity (for both stick and zeppelin), respectively. The parameter `mu` is the desired principal direction of diffusion given as a vector in spherical coordinates on unit sphere (i. e. two angles). The rest of the parameters, `odi`, `icvf` and `isovf` are the particular parameters of the NODDI model: orientation dispersion index ODI, intra-cellular volume fraction v_{ic} and isotropic volume fraction v_{iso}. Given all these parameters, a simulation of the NODDI model can be calculated. To achieve this, a parameter vector is prepared in line 13, assigning the given parameters to the (automatically generated) model parameters. The actual simulation is done by DmiPy via a simple function call in line 24, using the parameter vector as well as an acquisition scheme. The acquisition scheme used for the NODDI model is the WU-Minn HCP acquisition scheme already shown in listing 2.1 in section 2.2.2. It can be directly accessed and loaded via DmiPy.

For generating a dataset based on the NODDI model, the NODDI-specific parameters μ, ODI, v_{ic} and v_{iso} are drawn randomly from a uniform distribution. The orientation μ covers up the entire unit sphere. For ODI holds $0 \leq \text{ODI} \leq 1$ in theory, but it is drawn between 0.03 and 1, because smaller values than 0.03 led to NaN values in the simulation. This happened, because according to eq. (3.12) a small ODI results in a large value for κ (the parameter in the Watson distribution, see eq. (3.9)). This however leads to an overflow in the exponential function in eqs. (3.8) and (3.10). Both volume fractions v_{ic} and v_{iso} can vary between 0 and 1 in theory, but more commonly range from 0.1 to 0.9, so random values between 0.1 and 0.9 were used. After drawing the random parameter setting for the NODDI model, a simulation was computed. Then, noise was added to the signals following a Rician noise model as it is implemented in DmiPy. As

mentioned in section 2.2.1, the Rician noise arises from the the modulus of an additive Gaussian noise model. Rician noise is characterized via the signal-to-noise ratio (SNR). The noisy signal S_{Ric} following Rician noise from the original signal S based on SNR and a corresponding reference signal S_0 is calculated as

$$S_{\text{Ric}} = |S + \eta_1 + i\eta_2| = \sqrt{(S + \eta_1)^2 + \eta_2^2} \tag{6.1}$$

with the imaginary unit i and $\eta_1, \eta_2 \sim \mathcal{N}(0, \sigma^2)$ with $\sigma = \frac{S_0}{\text{SNR}}$. This procedure of randomly drawing parameter values, simulating the signal and adding noise was repeated until the desired number of samples was reached. The data was saved as NumPy [Oli06] arrays to disk. There were several datasets generated: two training sets (one containing 10^5 and the other containing 10^7 samples), a validation and a test set (each containing $2 \cdot 10^4$ samples). The smaller training set was used for hyperparameter tuning, since this was feasible in an appropriate time compared to the larger dataset. The best model was then trained on the larger training set again, which led to an even better performance of each model.

6.2 AMICO

AMICO is the baseline model of this work. It was published in [Dad+15] including an implementation in Python using the toolbox SPAMS (Sparse Modeling Software, [Jen+10]). The application of AMICO to NODDI using Python is shown in listing 6.2. In the implementation of AMICO, the datasets saved as NumPy arrays cannot be directly accessed. Besides, the acquisition scheme must be available as a file in a specific file format. This is due to the fact that AMICO loads the data automatically from file paths instead of taking pre-loaded arrays. Therefore, the simulated dataset must be restored in a different file format (NIfTI1 to be precise) and the acquisition scheme must be saved in a schemefile format. Both file paths must then be given into the AMICO load function. This procedure can be seen in lines 10 to 29. After that (in lines 35–37), the model is set to the NODDI model and the dictionaries are computed (or loaded if they have already been computed before).

Listing 6.2: Application of AMICO to the NODDI model

```
"""
Script for fitting AMICO to simulated NODDI data
"""

import amico
import numpy as np
from dmipy.data import saved_acquisition_schemes
import nibabel

amico.core.setup()

##############
# Load data #
##############

ae = amico.Evaluation(".", ".")
```

```
# load scheme from dmipy and save to disk
acq_scheme = saved_acquisition_schemes.wu_minn_hcp_acquisition_scheme()
acq_scheme.to_schemefile('scheme.txt')

# load data from numpy and save as nii
data = np.load('y_val.npy')[..., 1] # select a level of noise (snr)
data = data.reshape((len(data), 1, 1, 288)) # dummy dimensions due to 3D image
img = nibabel.nifti1.Nifti1Image(data, None)
nibabel.save(img, 'data.nii')

# load data and scheme in AMICO
ae.load_data(dwi_filename='data.nii', scheme_filename='scheme.txt')

##############################
# Compute response functions #
##############################

ae.set_model('NODDI')
ae.generate_kernels()
ae.load_kernels()

##############
# Model fit #
##############

lambda1 = 0.2
lambda2 = 0
ae.set_solver(lambda1=lambda1, lambda2=lambda2)
ae.fit()

ae.save_results()
```

For the NODDI model, AMICO provides two regularization parameters to be set. As described in section 5.1, one is for L1 and the other one for L2 regularization. These are set and used for solving in lines 43–46. For datasets corrupted with noise, both parameters must be set jointly to an appropriate configuration. To ensure a good parameter setting, the AMICO method is computed on the validation set for many different parameter combinations of λ_1 and λ_2. The values for both parameters are systematically varied from the parameter set $\lambda_1, \lambda_2 \in \{0, 0.001, 0.002, 0.005, 0.01, 0.02, 0.05, 0.1, 0.2, 0.5\}$. Since these are in total 10 values for each parameter, it ends up in 100 parameter combinations which need to be considered. This means computing the AMICO method for the validation set a hundred times. One fit on the validation set takes roughly 10–20 minutes. The results for AMICO on the validation set with noise level SNR=100 are shown in fig. 6.1. In general, the performance for the prediction of each parameter was measured in terms of mean absolute error (MAE), mean squared error (MSE) and mean relative error (MRE). They are computed as

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|\hat{x}_i - x_i|\,; \quad \text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(\hat{x}_i - x_i)^2\,; \quad \text{MRE} = \frac{1}{n}\sum_{i=1}^{n}\frac{|\hat{x}_i - x_i|}{x_i} \tag{6.2}$$

with $\hat{x}_i$ being the prediction of the true x_i. However, the orientation μ is a vector of length 1 (unit sphere) indicating the direction of diffusion, but the opposite direction $-\mu$ indicates the same orientation. Hence, other performance measures were used for evaluating μ: the mean distance (euclidean distance) and the mean spherical distance (the

distance on the surface of the unit sphere), both interpreted orientationally invariant. They are calculated as

$$\text{mean distance} = \frac{1}{n}\sum_{i=1}^{n} \min\left\{\left\|\hat{\mu}_i - \mu_i\right\|_2, \left\|\hat{\mu}_i + \mu_i\right\|_2\right\}; \tag{6.3}$$

$$\text{mean spherical distance} = \frac{1}{n}\sum_{i=1}^{n} \arccos\left|\hat{\mu}_i \cdot \mu_i\right| \tag{6.4}$$

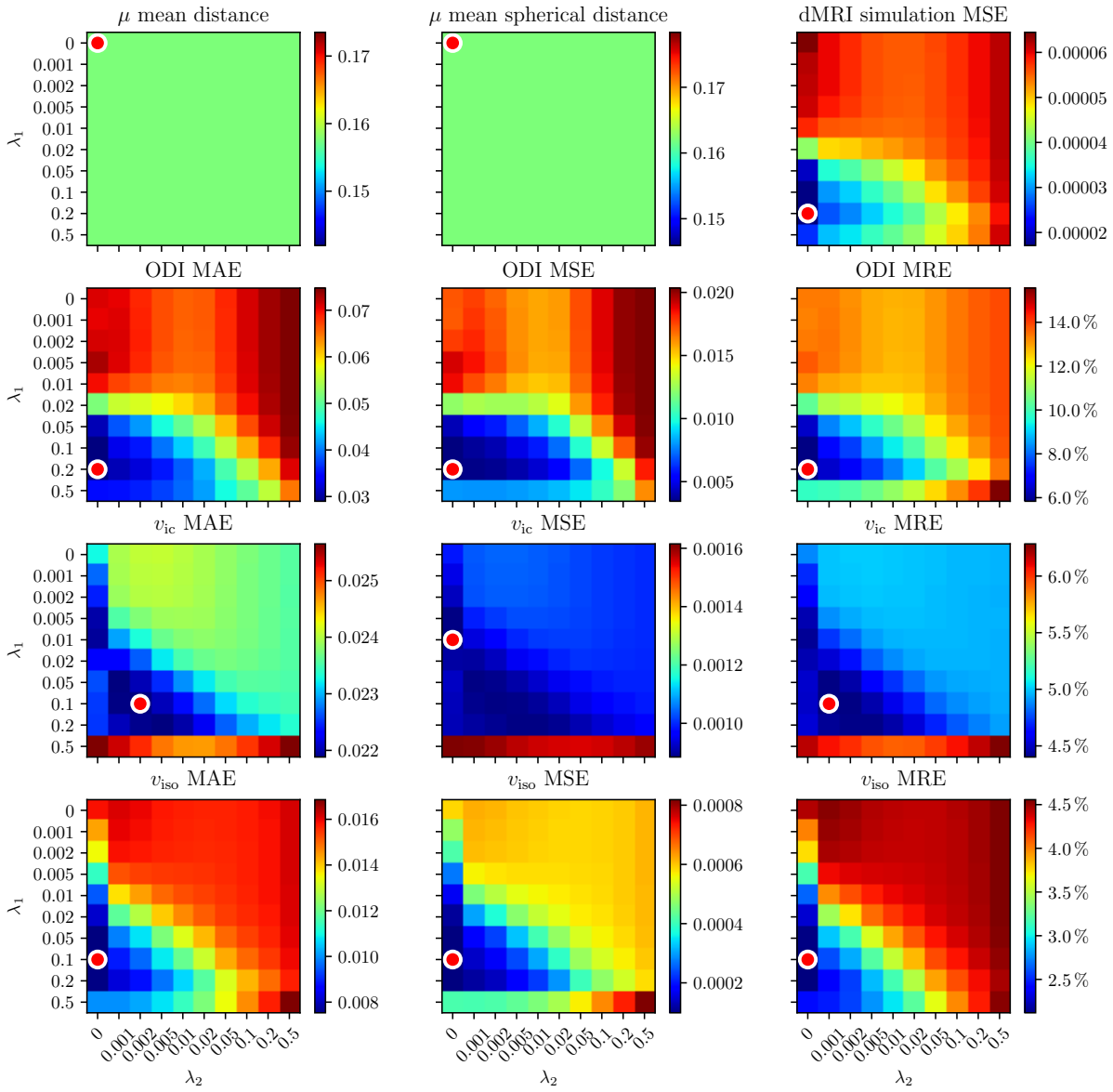


Figure 6.1: AMICO results for NODDI on validation set at SNR=100. The red dots indicate the lowest value in each subplot.

with $\hat{\mu}_i$ being the prediction of the true μ_i. The results for μ are not affected by regularization, since μ is not estimated in the minimization problem of eq. (5.5) but instead computed by a standard algorithm as described in section 5.1. The best performing parameter combination is not indisputable to find. Here, the parameter combination $\lambda_1 = 0.2; \lambda_2 = 0$ was selected to be compared to the other methods in the final results, because it achieved very low results on the validation set for all estimated parameters and metrics.

6.3 Multilayer Perceptron

The multilayer perceptron was implemented using PyTorch. For hyperparameter optimization, the number of layers and neurons was varied. In total, 3, 5, 7, …, 17 hidden layers with 150, 500 and 1000 neurons each were tested. The best performing one had 13 layers with 150 neurons in each layer for the case without noise, and it had 9 layers with 150 neurons each for SNR=100. The training process on the large training set takes about 2 hours and 15 minutes for 65 epochs.

In the standard MLP model, it is noticeable that the predictions of the orientation μ had a very high error. So the idea arose to give the orientation μ a different weighting in the loss function. Different weights for μ were tested, e. g. 2, 5, 10 and even 100 times higher than the other parameter weightings. However, the error could not be reduced. Instead, the errors for the other variables began to increase. As an extreme, only μ was trained with the MLP and the other variables were excluded. Even in this case, the MLP had only a poor performance. Hence, the MLP seems inappropriate for the estimation of the fiber orientation μ. In practice, other standard methods for the estimation of μ could be used, just as in AMICO.

6.4 Deep Inverse Models

For the deep inverse models in this work, the implementation given in [RPM21] was used. The models were implemented using PyTorch. To adapt the models to the microstructure imaging task, some changes in data reading and model parameters were made. Apart from that, scripts for evaluation and hyperparameter tuning must be provided.

6.4.1 Tandem

For the tandem model, different hyperparameters were tested. The number of fully connected layers, both for the forward and the backward model, was varied between 3, 5, 7 and 9 with 1000 neurons in each layer. Besides, for a specific number of fully-connected layers convolutional layers were added and varied. In this case the forward model consists of 4 hidden layers containing 1000 neurons and the backward model has 5 of such layers. For tuning, 2 and 3 convolutional layers were considered. Also, the number of channels was tested to be 8 or 16 and the kernel size was varied between 3 and 5. The tandem model with 2 convolutional layers, 16 channels each and a kernel

size of 3 performed best on the validation data with and without noise. The forward training took 2 hours and 20 minutes (60 epochs), whereas the backward training took 4 hours and 40 minutes (60 epochs).

In the case of noisy data, three different settings for the forward model were tested. In one setting, the forward model was trained on the noisy data. This however led to overfitting for the forward model and, as a result, in a bad overall performance. Hence in the second setting, the forward model was trained on data without noise to prevent overfitting and to keep the predictions of the forward model accurate. This setting led to a good performace, which is why this was considered in the final results. In the last setting, the forward model was again trained on data without noise, but this time random noise was added to the predictions of the forward model. Since the tandem model was trained in an end-to-end manner, the noise in the predictions must be realized in such a way that it was still possible to compute the backpropagation algorithm. However, this setting resulted in a bad performance, so the idea was discarded.

6.4.2 Neural Adjoint

The most important point of the implementation of NA to bring up is that the evaluation of NA takes a lot of time. As described in section 5.4, every prediction requires an optimization problem to be solved. In fact, there are 2048 optimization problems solved for a single sample at once, since there are problems with the NA approach regarding convergence (see section 5.4). From these 2048 problem solutions, the best performing one will be chosen. Since this evaluation process is very time-consuming, the number of samples considered in the evaluation of the NA model is limited to 1000 samples in this work. This means, the evaluation for NA is done on a subset of the validation/test set compared to the other models. It is important to keep this in mind when comparing the results. The evaluation of 1000 samples takes about two hours.

For hyperparameter tuning, 3, 5, and 7 layers were tested with 150, 500 and 1000 neurons in each layer. The best performing combination was 5 layers with 500 neurons each for the data without noise. For SNR=100, 5 layers with 1000 neurons obtained the best results on the validation set. The training process took in both cases about 2 hours and 25 minutes for 65 epochs. Similar to Tandem, the performance of NA with noisy data was poor. Just like the forward model in Tandem was replaced, NA was run using the model trained on data without noise too, but this time using noisy data for evaluation, which led to a much better performance.

6.5 MESC-Net

An implementation of MESC-Net was not available, so it had to be implemented following the description in [YLC19] (see section 5.5). It was implemented in PyTorch. Since the simulated dataset contains single independent voxels, the MESC-Net implemented for this work does not process image patches, but instead single voxels. The interesting part of the implementation is the iterative computation of modified LSTM units, which

is shown in listing 6.3. First, the variables x and c are initialized with zeros (lines 3–4). After that, the M-LSTM units are calculated. Since they are described as an iterative procedure, this can be realized in a loop. The iterative computation from section 5.5 is simply transferred into Python code (lines 8–12). The matrices W, S, W_{fx}, W_{fy}, W_{ix} and W_{iy} are the heart of the M-LSTM cell. They are shared among the layers, so they are used in every iteration in the implementation. Besides, they are realized here via a linear fully-connected layer without bias. After computing the M-LSTM cells, x is fed into some fully-connected layers, which generate the final output of the forward process.

For the MESC-Net, most hyperparameters like the number of layers and neurons were adopted from the original paper [YLC19] (8 M-LSTM cells followed by 3 fully-connected layers with 75 neurons each), since the MESC-Net was presented having this particular architecture. In [YLC19], the dictionary size N was varied instead. This was also done for this work. The dictionary sizes 100, 200 and 301 were considered, with $N = 301$ achieving the best results on the validation set for both cases with and without noise. The networks were trained for 30 epochs, which took about 4 hours and 30 minutes.

Listing 6.3: Implementation of MESC-Net forward computation

```
    def forward(self, y):
        # initialization for iterative computation of LSTM cells
        c = torch.zeros((len(y), self.N)).cuda()
        x = torch.zeros((len(y), self.N)).cuda()

        # modified LSTM cells
        for i in range(self.num_lstm):
            c_tilde = self.W(y) + self.S(x)
            f = torch.sigmoid(self.Wfx(x) + self.Wfy(y))
            i = torch.sigmoid(self.Wix(x) + self.Wiy(y))
            c = f*c + i*c_tilde
            x = nn.functional.threshold(c, self.lambda_thr, 0)

        # fully connected part
        x = self.layers(x)
        return x
```

7 Results

The results for the NODDI model on the test set without noise is shown in fig. 7.1. In the first row, the results for the orientation μ are compared using mean (euclidean) distance and mean spherical distance (as introduced in section 6.2). The plot on the top right shows the MSE for the re-simulated signal from the parameter estimations of each algorithm. For the estimation of μ, the MLP and the MESC-Net perform much worse than the other algorithms. The bad performance for μ leads to a high re-simulation error of dMRI signals. This indicates that the MLP and MESC-Net architectures are not appropriate to estimate the orientation μ. As described in section 6.3, higher weighting of μ in the loss function for the MLP cannot reduce the error. The MESC-Net architecture however is based on sparse coding just like AMICO. This approach might be inappropriate for estimating μ, which is why AMICO uses a standard method for this task instead (as described in section 5.1). Both self-supervised methods, Tandem and NA, perform similarly as good as AMICO when considering μ, with Tandem being slightly worse and NA being slightly better than AMICO. For the re-simulation error, the bars for AMICO, NA and Tandem are barely visible. AMICO (MSE: $1.59 \cdot 10^{-5}$) performs worse than Tandem (MSE: $3.40 \cdot 10^{-5}$) and NA (MSE: $1.13 \cdot 10^{-6}$).

In the following rows of fig. 7.1, the MAE, MSE and MRE of the orientation dispersion index ODI, the intra-cellular volume fraction v_{ic} and isotropic volume fraction v_{iso} are visualized. The MLP performs best, followed by MESC-Net. NA performs better than AMICO for v_{ic} and v_{iso}, but worse for ODI. The Tandem model outperforms AMICO only for v_{ic}. In principle, the Tandem model is the extension of the MLP by a forward model. This enables the Tandem model to be trained in an end-to-end manner minimizing the re-simulation error. The extension by a forward model on the one hand seems to overcome the limitation of the MLP in estimating μ, but on the other hand results in a worse performance of the other parameters. In general, since NA and Tandem are trained to minimize the MSE of the simulated dMRI signals and instead of the MSE of the parameter estimates, they might tend to find other parameter settings which lead to very similar dMRI signals. This phenomenon of non-uniqueness in inverse problems was described in section 5.3 and must be kept in mind when evaluating the results. Also, NA is only evaluated on a subset of the test set (due to the time-consuming evaluation process described in section 6.4.2), so the results must be compared carefully.

The results on the dataset with noise (SNR=100) are shown in fig. 7.2. Again, MLP and MESC-Net are unable to estimate the fiber orientation μ, which leads to high errors for the re-simulated signals. Tandem and NA can estimate μ a bit worse than AMICO, with Tandem being closer to the AMICO result. Also the MSE of simulated signals shows AMICO achieving the best results, followed by Tandem and NA. The MSE of dMRI signals however reflects the combination of all the estimated parameters, which

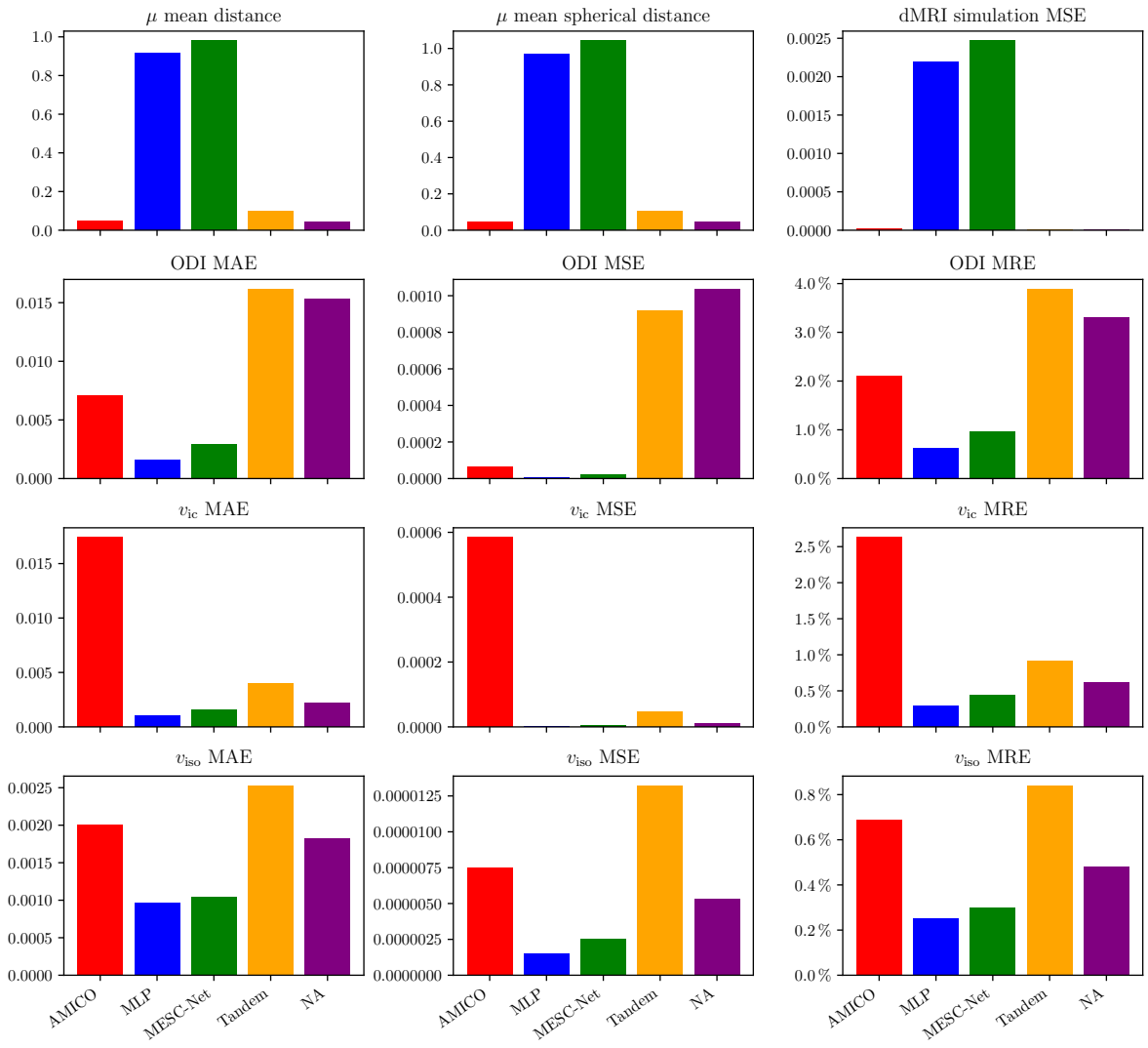


Figure 7.1: Results for the NODDI model on the test set without noise

are evaluated individually in the plots below. For ODI, MLP, MESC-Net and Tandem are close to AMICO, with Tandem being slightly worse and the others being a little better than AMICO. NA can only achieve a high error for ODI. For both of the volume fractions, v_{ic} and v_{iso}, all the neural network approaches can outperform AMICO, with MLP and MESC-Net having the smallest errors followed by Tandem and lastly NA.

To conclude, for the dataset without noise the parameter estimates of MLP and MESC-Net are best except for the orientation μ. Because of the bad estimates for μ, the re-simulation for MLP and MESC-Net has a high error. The self-supervised methods Tandem and NA have worse parameter estimates than MLP and MESC (except for μ), but achieve good re-simulation errors, even better than AMICO. This is due to the fact that both approaches minimize the re-simulation error during training. In the case of SNR=100, μ and the simulated dMRI signals are approximated best by AMICO, followed by Tandem and NA. MLP and MESC-Net show high errors for μ again. When considering ODI, all the methods have a similar performance except for NA, which has a very high error. In contrast, all the neural network approaches can outperform AMICO when considering both of the volume fractions.

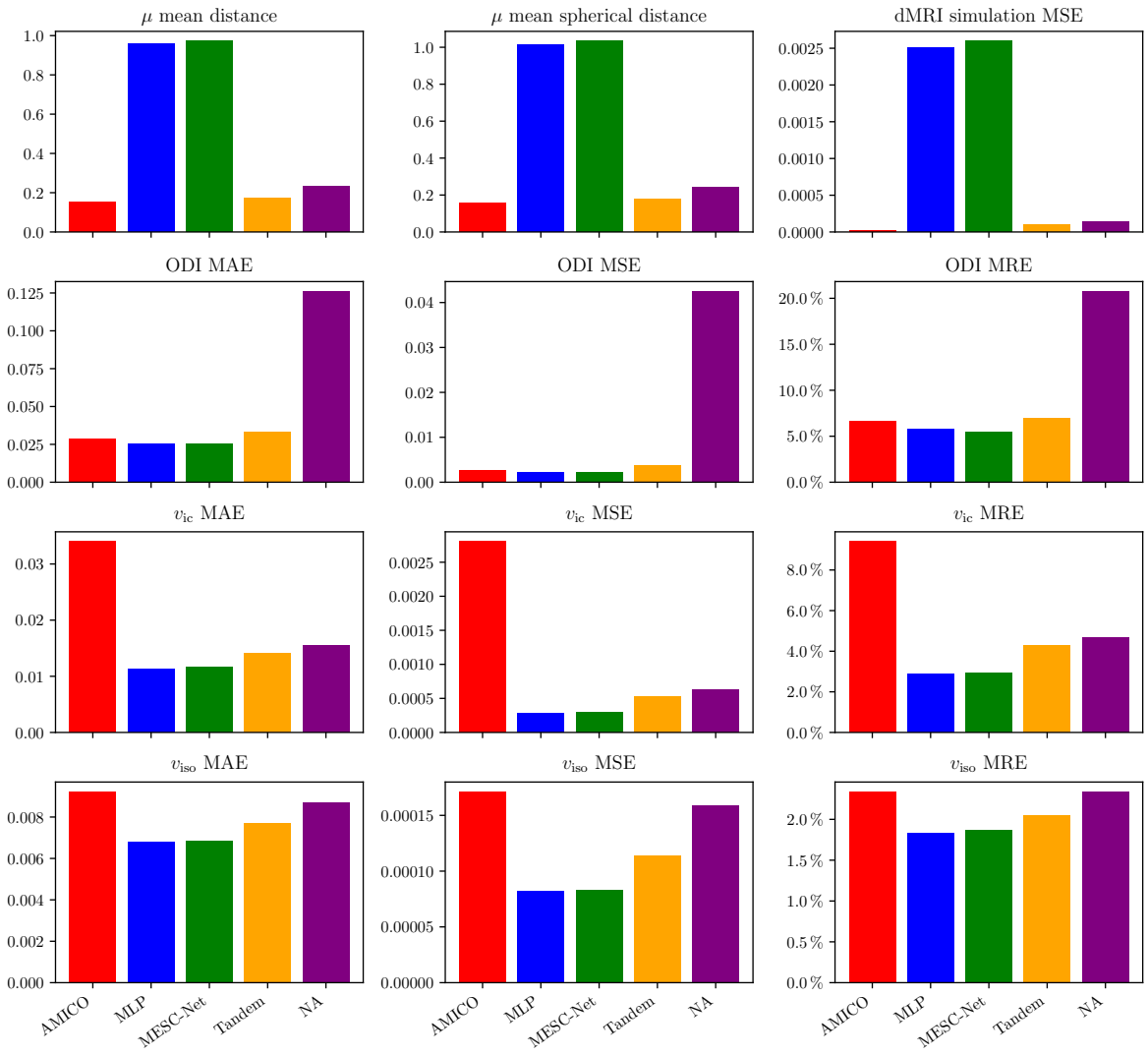


Figure 7.2: Results for the NODDI model on the test set with SNR=100

8 Conclusion

None of the algorithms used in this work outperforms all the others with its overall performance. The MLP and the MESC-Net had high errors for the estimation of the fiber orientation μ and, as a result, also a high error for the re-simulated dMRI signals. But for the other NODDI-specific parameters, ODI, v_{ic} and v_{iso}, they could achieve the best results, with MLP being slightly the best. The self-supervised methods Tandem and NA performed worse than MLP and MESC-Net on the NODDI parameters, sometimes even worse than AMICO. However, their estimations for μ are much better than MLP and MESC-Net and they achieve the best results for the re-simulated dMRI signals for the case without noise, since they are trained to minimize this error during training. This however does not hold for the noisy data.

The advantage of self-supervised methods is that they can be directly applied to real measurements. They do not require a dataset with true (or estimated) parameter configurations but only the dMRI measurements. Their design to minimize the re-simulation error of dMRI signals on the one hand leads to a good performance in terms of dMRI signals, but on the other hand can produce wrong parameter estimates, since inverse problems like the one in microstructure imaging can suffer from non-uniqueness of solutions.

The next step would be to apply self-supervised methods to real dMRI measurements and to evaluate their performance. The Tandem and the NA approach could be improved or new architectures could be developed. An idea for the development of new architectures is to use the simulator implemented in DmiPy as a forward model appended to an arbitrary deep learning model (similar to the Tandem approach). A training using a gradient descent algorithm however requires the gradient of the simulator to be computed. This could be achieved porting the simulator to a deep learning library like PyTorch.

Bibliography

[18] *Helmholtz Analytics Framework*. Mar. 6, 2018. URL: http://www.helmholtz-analytics.de/helmholtz_analytics/EN/Project/_node.html (visited on 09/06/2021).

[Ale06] Daniel Alexander. "An Introduction to Computational Diffusion MRI: the Diffusion Tensor and Beyond". In: Jan. 2006, pp. 83–106. ISBN: 978-3-540-25032-6. DOI: 10.1007/3-540-31272-2_5.

[Alp10] Ethem Alpaydin. *Introduction to Machine Learning*. 2nd ed. The MIT Press, 2010. ISBN: 978-0-26-201243-0.

[BD08] Thomas Blumensath and Mike E. Davies. *Iterative Hard Thresholding for Compressed Sensing*. 2008. arXiv: 0805.0510 [cs.IT].

[BML94] P. Basser, J. Mattiello, and D. LeBihan. "MR diffusion tensor spectroscopy and imaging." In: *Biophysical journal* 66 1 (1994), pp. 259–67.

[Dad+15] Alessandro Daducci et al. "Accelerated Microstructure Imaging via Convex Optimization (AMICO) from diffusion MRI data". In: *NeuroImage* 105 (2015), pp. 32–44. ISSN: 1053-8119. DOI: https://doi.org/10.1016/j.neuroimage.2014.10.026. URL: https://www.sciencedirect.com/science/article/pii/S1053811914008519.

[DHS11] John Duchi, Elad Hazan, and Yoram Singer. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization". In: *Journal of Machine Learning Research* 12 (2011), pp. 2121–2159. ISSN: 1532-4435. URL: http://www.jmlr.org/papers/volume12/duchi11a/duchi11a.pdf (visited on 09/06/2021).

[DMC] Ian Dewancker, Michael McCourt, and Scott Clark. *Bayesian Optimization Primer*. URL: https://app.sigopt.com/static/pdf/SigOpt_Bayesian_Optimization_Primer.pdf (visited on 09/06/2021).

[Fic+18] Rutger Fick et al. "Dmipy, a Diffusion Microstructure Imaging toolbox in Python to improve research reproducibility". In: Sept. 2018.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. URL: http://www.deeplearningbook.org.

[Gol+16] V. Golkov et al. "q-Space Deep Learning: Twelve-Fold Shorter and Model-Free Diffusion MRI Scans". In: *IEEE Transactions on Medical Imaging* 35.5 (2016). Special Issue on Deep Learning. URL: http://lmb.informatik.uni-freiburg.de/Publications/2016/DB16f.

[Gos+10] Yves Gossuin et al. “Physics of magnetic resonance imaging: From spin to pixel”. In: *Journal of Physics D Applied Physics* 43 (June 2010), p. 213001. DOI: 10.1088/0022-3727/43/21/213001.

[Hin12] Geoffrey Hinton. *Neural Networks for Machine Learning*. Lecture slides. 2012. URL: https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf (visited on 09/06/2021).

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. “Long Short-term Memory”. In: *Neural computation* 9 (Dec. 1997), pp. 1735–80. DOI: 10.1162/neco.1997.9.8.1735.

[Jen+10] Rodolphe Jenatton et al. “Proximal Methods for Sparse Hierarchical Dictionary Learning”. In: International Conference on Machine Learning. 2010.

[Kab+08] Humayun Kabir et al. “Neural Network Inverse Modeling and Applications to Microwave Filter Design”. In: *Microwave Theory and Techniques, IEEE Transactions on* 56 (May 2008), pp. 867–879. DOI: 10.1109/TMTT.2008.919078.

[KB14] Diederik P. Kingma and Jimmy Lei Ba. *Adam: A Method for Stochastic Optimization*. 2014. URL: https://arxiv.org/pdf/1412.6980v8.pdf (visited on 09/06/2021).

[Liu+18] Dianjing Liu et al. “Training Deep Neural Networks for the Inverse Design of Nanophotonic Structures”. In: *ACS Photonics* 5.4 (Feb. 2018), pp. 1365–1369. ISSN: 2330-4022. DOI: 10.1021/acsphotonics.7b01377. URL: http://dx.doi.org/10.1021/acsphotonics.7b01377.

[Oli06] Travis E Oliphant. *A guide to NumPy*. Vol. 1. Trelgol Publishing USA, 2006. URL: https://numpy.org/ (visited on 09/06/2021).

[Pan+12] Eleftheria Panagiotaki et al. “Compartment models of the diffusion MR signal in brain white matter: A taxonomy and comparison”. In: *NeuroImage* 59 (2012), pp. 2241–2254.

[Pas+19] Adam Paszke et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[RCC07] Doug Richards, Tom Clark, and Carl Clarke. *The Human Brain and Its Disorders*. Oxford University Press, 2007. URL: https://books.google.de/books?id=dr7bQBtiMI4C.

[RHW86] David E. Rumelhart, Geoffrey Hinton, and Ronald J. Williams. “Learning representations by back-propagating errors”. In: *Nature* 323 (Oct. 9, 1986), pp. 533–536. URL: https://www.iro.umontreal.ca/~vincentp/ift3395/lectures/backprop_old.pdf (visited on 09/06/2021).

[RPM21] Simiao Ren, Willie Padilla, and Jordan Malof. *Benchmarking deep inverse models over time, and the neural-adjoint method*. 2021. arXiv: 2009.12919 [cs.LG].

[Rud16] Sebastian Ruder. *An overview of gradient descent optimization algorithms*. 2016. arXiv: 1609.04747 [cs.LG].

[Sha+16] Bobak Shahriari et al. "Taking the Human Out of the Loop: A Review of Bayesian Optimization". In: *Proceedings of the IEEE* 104 (1 Jan. 2016). DOI: 10.1109/JPROC.2015.2494218.

[Tea15] Python Core Team, ed. *Python. A dynamic, open source programming language*. Python Software Foundation. 2015. URL: https://www.python.org/ (visited on 09/06/2021).

[YLC19] Chuyang Ye, Xiuli Li, and Jingnan Chen. "A deep network for tissue microstructure estimation using modified LSTM units". In: *Medical image analysis* 55 (2019), pp. 49–64.

[Zha+12] Hui Zhang et al. "NODDI: Practical in vivo neurite orientation dispersion and density imaging of the human brain". In: *NeuroImage* 61.4 (2012), pp. 1000–1016. ISSN: 1053-8119. DOI: https://doi.org/10.1016/j.neuroimage.2012.03.072. URL: https://www.sciencedirect.com/science/article/pii/S1053811912003539.