Design and Interpretation of a Convolutional Neural Network Architecture for Imaging Mass Spectrometry Data

Ioanna Chanopoulou

Master of Science Thesis

Delft Center for Systems and Control

Design and Interpretation of a Convolutional Neural Network Architecture for Imaging Mass Spectrometry Data

Master of Science Thesis

For the degree of Master of Science in Systems and Control at Delft University of Technology

Ioanna Chanopoulou

July 4, 2023

Faculty of Mechanical, Maritime and Materials Engineering (3mE) · Delft University of Technology

VANDERBILT
UNIVERSITY

The datasets used in this thesis were provided by the Mass Spectrometry Research Center of Vanderbilt University in Nashville, Tennessee, United States of America. Their contribution is hereby gratefully acknowledged.

TU Delft Delft University of Technology

DCSC

Delft University of Technology
Department of
Delft Center for Systems and Control (dcsc)

The undersigned hereby certify that they have read and recommend to the Faculty of Mechanical, Maritime and Materials Engineering (3mE) for acceptance a thesis entitled

Design and Interpretation of a Convolutional Neural Network Architecture for Imaging Mass Spectrometry Data

by

Ioanna Chanopoulou

in partial fulfillment of the requirements for the degree of

Master of Science Systems and Control

Dated: July 4, 2023

Supervisor(s): ____________________
Dr. ing. R. Van de Plas

Reader(s): ____________________
Dr. ing. R. Van de Plas

____
Dr. Oleg Soloviev

____
Léonore Tideman

Abstract

Convolutional Neural Networks (CNNs) have emerged primarily from research focusing on image classification tasks and as a result, most of the well-motivated design choices found in literature are relevant to computer vision applications. CNNs' application on Imaging Mass Spectrometry (IMS) data is quite recent and involves new challenges, such as taking into account their unique structure (e.g. both spatial and spectral dimensions).

In this thesis, we suggest a 1-D CNN architecture that extracts local features along the spectral dimension. The aim is to investigate if CNNs improve the classification accuracy compared to other classic Machine Learning (ML) methods such as linear models. Furthermore, we explore Neural Networks (NNs) that employ the novel Sharpened Cosine Similarity (SCS) as a feature extraction method opposed to convolution. We call those networks SCS-NN in correspondence to the Convolutional-NN (CNN). To evaluate these methods, we implement our pipeline for various IMS datasets, with different characteristics and classification tasks, using several performance metrics such as balanced accuracy and F1 score.

Moreover, we provide a detailed description of the methodology pipeline used for the CNN architecture design. The suggested methodology is the Tree-structured Parzen Estimator (TPE) algorithm, a Bayesian optimization technique for automated architecture selection. By implementing TPE, we manage to explore and exploit efficiently a complex and large hyperparameter configuration space and automatically select optimal hyperparameters (such as number of convolutional layers, kernel size, strides, learning rates etc.). This automated approach reduces time consumption, errors, and the need for specialized knowledge in biology and biochemistry that would be associated with manual design. In addition to developing a pipeline for designing, training and evaluating a CNN for IMS data classification, we also apply a model agnostic interpretation methodology based on SHapley Additive exPlanations (SHAP) and provide SHAP score maps that visualize the importance of features in the spatial dimension of the IMS datacube.

To sum up, in this thesis, we present and analyse the automated selection of 1-D CNN architectures for IMS data classification based on TPE. Furthermore, we investigate a novel alternative to convolution, SCS, and evaluate its strengths and weaknesses in IMS data classification. The experimental results show that the TPE-generated CNN architectures outperform all the other applied classifiers. Finally, our interpretation of the CNN models reveals that accuracy performance alone might not be a sufficient criterion to trust the model's output.

Table of Contents

Acknowledgements

The process of working on a final thesis for a Master of Science degree can be challenging. Throughout this challenging journey, I had several allies to whom I would like to express my gratitude.

I would like to thank my supervisor Dr. ing. R. Van de Plas for his guidance and support both in an academical and personal level, during the writing of this thesis. I would also like to express my special gratitude to Léonore Tideman, my weekly supervisor, for her invaluable suggestions, support and guidance throughout this journey.

Last but not least, I would like to thank my family and friends for their love and support throughout my academic studies. Μαμά, σ'ευχαριστώ για την υποστήριξη και την αφοσίωσή σου όλα αυτά τα χρόνια.

Delft, University of Technology
July 4, 2023

Ioanna Chanopoulou

“To the memory of my father and to my mother”

Chapter 1

Introduction

In chapter 1, we provide an introduction to the topic and motivation of this thesis. In section 1-1, an introduction to the need for the use of Machine Learning (ML) techniques in the context of Imaging Mass Spectrometry (IMS) data classification is presented along with the existing approaches. Next, in section 1-2, the problem statement addressed within this thesis, the motivation for the selected approach and the main objectives of the thesis are discussed. Finally, the contributions this thesis attempts to offer are described in section 1-3.

1-1 Existing Work for IMS Data Classification

The field of IMS has rapidly advanced in recent years, providing high-resolution imaging of molecules in tissues, cells, and organisms. However, the complex and large-scale nature of IMS data presents a significant challenge for accurate and efficient analysis [13]. Analysis of the data by a trained expert would require the human visual inspection of each ion distribution on the tissue and the manual evaluation of the molecular alterations. Nevertheless, given the large number of features measured per pixel in IMS data, the manual data analysis would be very time consuming and highly prone to errors. Moreover, traditional processing - such as statistical analysis - of high-dimensional IMS data requires a big amount of computational resources. Sequentially, there is a need for adopting a more automated and efficient method in IMS data analysis and classification.

Supervised ML algorithms are established as powerful tools for the efficient classification of IMS data. Both linear and non-linear algorithms have been reported in literature. For example, in [23], Linear Discriminant Analysis (LDA) is used to classify epithelian ovarian cancer histotypes and in [29], Random Forest (RF) and Support Vector Machine (SVM) are used for classifying six cancer types in different organs. Furthermore, Deep Learning (DL) has also been applied by researchers. One significant difference between the traditional supervised ML and DL techniques lies in the fact that the first require manual feature extraction prior to the training of the classification model, while the latter learn the features automatically from the raw input data. On the other hand, DL techniques require a larger amount of training data and computational resources than classic ML techniques.

One common DL methodology is the Convolutional Neural Network (CNN). For instance, in [4], the authors design and interpret a CNN architecture by adapting an image classification model, ResNet, to the characteristics of IMS data, while, in [36], the researchers base their architecture on [4] and they are introducing the use of dilated CNNs, which aims to deal with the problem of having a lot of hyperparameters and to reduce the computational complexity. Furthermore, in [30], the authors are building a 1-D CNN to classify colorectal tumor versus normal tissue, while they mention that they haven't exhausted hyper-parameter optimization for the model.

In conclusion, there has been an extensive use of traditional ML and DL techniques - CNNs - for the task of IMS data classification. Some of the weaknesses of the relative work for CNN design seem to be that some of the research is not providing detailed motivation for each choice of the used architecture. Additionally, in cases such as [4], the CNN design choices are based on some feature engineering output, which counteracts the independence of DL approaches from extensive human pre-processing techniques. Moreover, some of the research is lacking interpretability of the proposed model.

1-2 Research Motivation and Objectives

Linear methods might be simpler and transparent but they have limitations in handling the complex underlying relationships between the spectral features of IMS data. Hence, the use of non-linear methods such as CNNs, which have demonstrated remarkable success in image classification tasks, are also worth to be adopted in IMS data classification. The subsequent question that arises concerns the rationale behind the adoption of a CNN in IMS data classification. The most fundamental reasons for using CNNs for IMS data classification are:

- CNNs were developed for image data classification. The fact that spectral data share similar characteristics with image data make CNNs a suitable method.

Table 1-1: CNN suitability for spectral and image data classification

Spectral Data	**Image Data**
large amount of mass spectra	large amount of pixels
convolution: correlation between neighbouring m/z bins	convolution: correlation between neighbouring pixels
layered architecture: 1st - spectrum peaks 2nd - feature paterns (e.g. isotopes) 3rd - more complex patterns (e.g. proteins, lipids)	layered architecture: 1st - image edges 2nd - image shapes 3rd - entire structures

In Table 1-1, the aspects that make IMS data image-like are reported and the suitability of the CNN usage is justified. First of all, CNNs can handle large amounts of pixel data which both images and IMS data have. The pixel locations in an IMS dataset can be hundreds of thousands. Secondly, the operation of convolution requires the existence of correlation between neighbouring datapoints, which is also true for neighbouring *m/z* bins. Last but not least, CNNs process data in a hierarchical manner. Therefore, lower layers of the network learn low-level features such as edges in images and spectrum peaks in mass spectra, and higher layers learn more complex features such as entire objects in images and for instance, protein patterns in mass spectra.

- CNNs can be trained on multiple GPUs [24] simultaneously, which makes them powerful tools in high dimensional datasets such as IMS data. By using multiple GPUs, the computational workload can be distributed across the GPUs, allowing for parallel processing. This can alleviate the memory constraints that might arise as the number of features increases.
- CNNs automatically learn features of the raw input data without requiring manual feature engineering techniques prior to training.
- CNNs hold the highest performance in image classification and therefore, it's an attractive method to be applied as well in IMS data.

Nevertheless, the methodology of CNNs can also have several drawbacks. First of all, if the available dataset size is small it may not be possible to train a CNN model effectively. Additionally, if regularization techniques are not employed, overfitting can pose a problem. Furthermore, the CNN model's trustworthiness relies on its interpretability, as it may be perceived as a black box. Along with the above mentioned issues, due to their complexity and the large number of computations involved in their operations, CNNs require high computational power, which is expensive to have. Moreover, hyperparameter optimization of a CNN can be a cumbersome task.

This thesis aims to deliver a detailed design of a CNN architecture for IMS data classification. The objectives of this thesis can be summarized in two main research questions:

1. How can one design an optimal architecture for a CNN model for accurate classification of IMS data?
2. How can this model's predictions be interpreted efficiently?

Several subquestions also addressed in this thesis include a more efficient hyperparameter optimization technique, how robust the hyperparameter tuning techniques can be and the assessment of the contribution of CNNs in IMS data classification as opposed to linear models. Finally, this thesis also aims to answer if a novel alternative to the convolution operation, Sharpened Cosine Similarity (SCS), can be more efficient for IMS data classification.

1-3 Contributions

This thesis aims to make contributions in the following topics:

- Implementation of Tree-structured Parzen Estimator (TPE) algorithm as a design method of the CNN architecture tailored for the task of IMS data classification
- Provide an easily understandable interpretation of the designed CNN models using visualizations of features' importance in the spatial dimension
- Implementation for the first time, to our knowledge, of SCS for feature extraction in IMS data classification and evaluation of its contribution opposed to convolutional layers
- Evaluation of the benefits of using CNNs in a variety of IMS data classification tasks
- Evaluation of the impact of IMS data pre-processing techniques on the classification performance of CNNs

Chapter 2

Exploring the Characteristics of IMS Data

The field of Imaging Mass Spectrometry (IMS) has recently seen a significant development due to its ability to provide high-resolution images of the distribution of biomolecules in tissue samples, which facilitates the connection of the detected ions and the histological structure of the sample. IMS is an imaging technique based on Mass Spectrometry (MS) that enables the spatially localized biochemical analysis of a sample. Therefore, in section 2-1, we begin with defining the methodology and instrumentation employed in MS and expand to the IMS methodology in section 2-2 and to the data structure in section 2-3.

2-1 Mass Spectrometry

MS is a technique for analyzing the components of inorganic or organic substances or biological samples. This method is characterized by a series of steps [16]. After the sample is prepared and put into the sample inlet, gas-phase ions are generated from the analyte - the sample being analyzed - through a process of ionization. Subsequently, these ions are sorted, under vacuum, according to their mass-to-charge (m/z) ratio using mass analyzers. Finally, the produced ions (either positively or negatively charged, depending on the imaging modality) are detected on the detector and their abundance/intensity is measured.

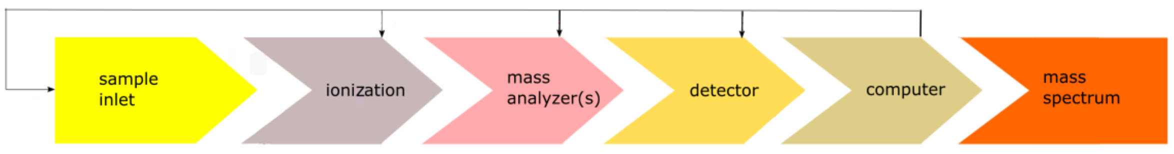


Figure 2-1: A general illustartion of the mass spectrometer's pipeline, based on [16]

The instrument with which the above technique is applied is called a mass spectrometer. The fundamental components of a mass spectrometer are: a sample inlet, where the preprocessed sample is inserted, an ion source, where ions are produced from the sample surface, mass analyzer(s), where the ions are distinguished based on their mass-to-charge ratio (m/z), an ion detector, a data processing system (computer), which coordinates the process and receives the output and a vacuum pump, which takes care of maintaining the vacuum conditions [11, 16]. The mass spectrometer components are outlined in Figure 2-1.

The output of the mass spectrometer is the mass spectrum. The mass spectrum is a 2-D visualisation of the intensity of ions in a sample as a function of their mass-to-charge ratio (m/z). It can be presented as either a 2-D plot or a table. An example is depicted in Figure 2-2.

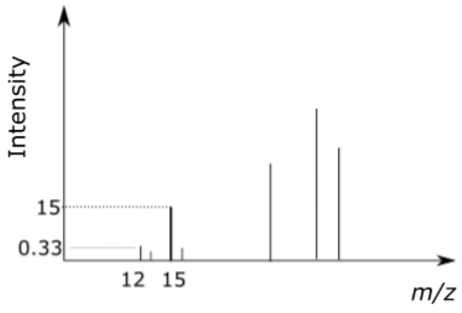


Figure 2-2: Example graph of a mass spectrum, the output of a mass spectrometer

In Figure 2-2, the x-axis represents the mass-to-charge ratio (m/z) and the y-axis the number of detected ions, termed as the ion intensity. Usually, the intensity is normalized according to total ion count.

The artificial scalar, m/z, used in MS is a combination of two parameters: m, which represents the relative atomic mass of the ion, and z, which indicates its number of charges. This leads to the units that m/z can have, although it can also be viewed as dimensionless. In general, the choice of unit for m/z depends on the context of the analysis. Nevertheless, a commonly used dimension for the mass, m is either the *atomic mass units*, u or the *dalton*, Da [11]. Both are equivalent and it holds:

$$1Da = 1u = 1.660540 \times 10^{-27} kg$$

Moreover, the charge unit is expressed as a multitple of the elementary charge, namely, the charge of one electron, *1e*. Consequently, the proposed unit for m/z is u/e and is called Thompson [$1\,Th$], although it is not an SI unit [11, 16].

2-2 The IMS Experiment

IMS is the technique which offers simultaneously the visualization of both the intensity and the spatial distribution of all the detected ions by mass spectrometry, on the actual surface of the analyzed sample [28]. The visualizations of the 2-D spatial distribution and abundance of the detected ions are called ion images. You can see an example of some ion images in Figure 2-3.

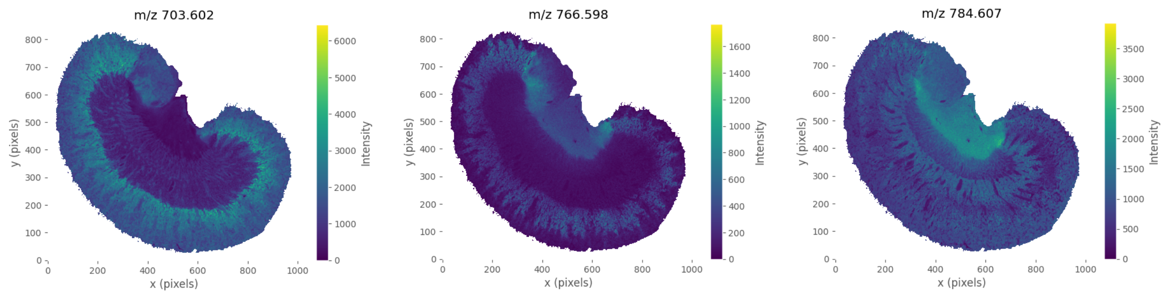


Figure 2-3: Examples of ion images on a rat kidney sample

In Figure 2-3, we can see the ion images of three different ions (m/z 703.602, m/z 766.598, m/z 784.607) on a rat kidney tissue surface. It is observed that the ion images are comprised of thousands of pixels, the total amount is almost 800,000 in this example. On these ion images we can distinguish the spatial location of the ions as well as their intensity/abundance at each pixel position, provided by the colourmap, the lighter the colour the higher the ion intensity. The ion images are one aspect of the IMS experiment datacube.

Using such a tool as IMS offers ion distribution information across a biomedical sample (e.g. protein expression on a tissue section) and it can be used to indicate alterations in protein -or other substances like lipids, metabolites, drugs- expression levels and connect them to specific effects (disease/ drug evolution) [9].

Furthermore, it is important to explain the processes that take place so that the IMS datacube is obtained. A common IMS experiment is comprised of the sample preparation, then the sample is positioned in the mass spectrometer and analyzed as already mentioned in the steps of MS with the addition of the spectral features visualization.

2-2-1 Ionization Techniques

After the sample is placed in the mass spectrometer as described in Figure 2-1 the first step is ionization. The selection of ionization techniques for MS analysis depends on the type of compound or sample that needs to be analyzed, with certain techniques being better suited for specific compounds than others. There are gas-phase, liquid-phase and solid-phase ionization techniques [11]. For the purpose of IMS [28], three main ionization methods are currently used:

1. Secondary Ion Mass Spectrometry (SIMS)
2. Matrix-Assisted Laser Desorption Ionization (MALDI)
3. Desorption Electrospray Ionization (DESI)

Since the data that are used throughout this thesis are MALDI IMS images, we are going to spend the following lines to explore just the MALDI IMS experiment.

MALDI

High mass capabilities of MALDI makes it a standard method to be applied for imaging protein distributions within samples of tissue sections [28]. Furthermore, The MALDI method can be described by the following steps [21].

Step 1 In the initial step, the acquired tissue section is prepared and placed on a MALDI plate.

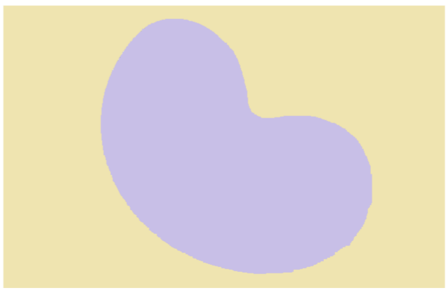


Figure 2-4: Tissue section placed onto glass plate for MALDI analysis

Step 2 Next, the matrix is deposited on the sample and left to dry (Figure 2-5). The matrix is a substance comprised of small organic molecules (e.g. organic acids) in a suitable solvent and its contribution is firstly, to absorb most of the laser energy minimizing the sample damage and secondly, to make the energy transfer more efficient. The sample (tissue section) is mixed or coated with the matrix (in an ordered array) which crystalizes and 'traps' the analytes within the crystals.

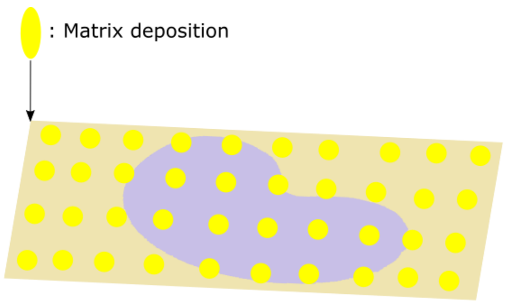


Figure 2-5: Local deposition of matrix solution across the tissue section sample

Step 3 In vacuum, removal of portions of this dry crystallized sample takes place by laser pulses over short time-duration (usually of UV wavelengths) leading to ion generation[11].

The advantages of the MALDI technique are that it is flexible, fast and capable of analyzing analytes with large molecular mass, such as proteins. In addition, MALDI usually produces single charged ions [11].

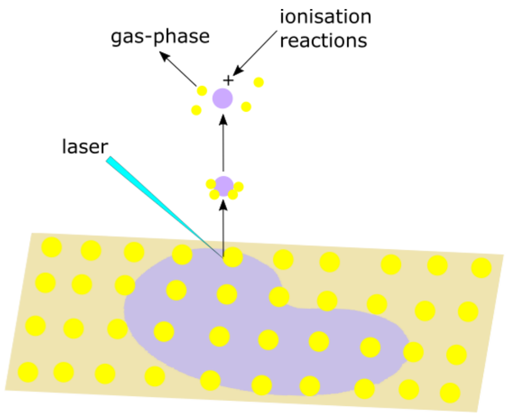


Figure 2-6: Pulsed ultraviolet laser irradiation of the matrix crystals for the production of ions that will further be mass analyzed

2-2-2 Mass Analyzers

Mass-over-charge (m/z) analyzers or mass analyzers are used to separate the produced ions, which are electrically charged particles, based on their m/z value in relation to their response to electric or magnetic fields [40]. The data used throughout this thesis are derived from *Time-of-flight (TOF)* and *Fourier Transform Ion Cyclotron Resonance (FTICR)* IMS. Hence, we are going to describe those two methods.

TOF analyzer

The operating principle of TOF analyzers is measuring the time that an ion needs to cross from the ion source to the detector Figure 2-1. TOF analyzers are mostly used in MALDI instrumentation setups and they can be found in both linear and orthogonal geometry.

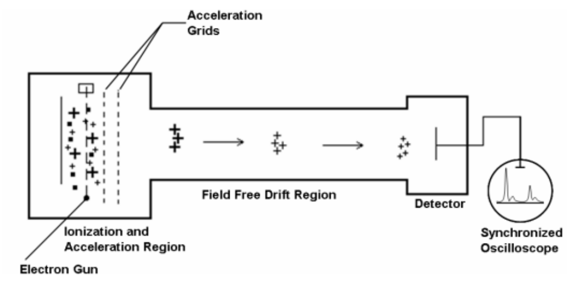


Figure 2-7: Linear TOF analyzer [Image taken from J. WATSON and O. SPARKMAN, Copyright ©2007, [40]]

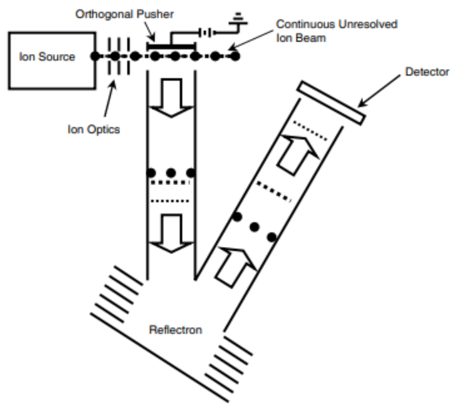


Figure 2-8: TOF with orthogonal acceleration [Image taken from J. WATSON and O. SPARKMAN, Copyright ©2007, [40]]

Linear Groups of ions get a specific kinetic energy and then they sequentially travel across the linear mass analyzer's field free drift region, hitting the detector as presented in Figure 2-7. A smaller m/z value results in getting to the detector faster than having a larger m/z ratio. The needed time is a function of the ions masses, thus the mass can be found. The acceleration is given to the ions by electromagnetic plates and their kinetic energy is equal to the potential energy as follows:

$$\begin{aligned} \frac{1}{2}mu^2 &= zV \\ u &= \sqrt{\frac{2zV}{m}} \end{aligned} \tag{2-1}$$

where u is the ion velocity, m the mass, V the acceleration potential and z, represents the total charge, which is the product between the charge amount and the elementary charge of an electron. In Equation 2-1, we can see the inverse relation between the ion velocity and its mass [40]. Calculating the velocity of the ion straightforward is not practical, thus, the TOF experiment uses the distance of the ion source from the detector, denoted by l, and then, the time of flight is calculated as:

$$TOF = \frac{l}{u} = l\sqrt{\frac{m}{2zV}} \tag{2-2}$$

Then, an ion's m/z value is calculated based on its TOF to the detector. The errors in linear TOF analyzers are compensated using the "reflectron", or alternatively, ion mirror.

Orthogonal TOF analyzers operating with orthogonal acceleration send the produced (from the ion source) ions into the analyzer area which applies mass analysis in an orthogonal direction with respect to the ion extraction direction. An illustration of this method can be seen in Figure 2-8. This methodology has the contributory factor of the independence of the resolution and accuracy from the ionization step [28].

FTICR analyzer

The operating principle of an FTICR analyzer is based on the effect that a magnetic field can have on the motion of a charged particle. Specifically, an ion which is travelling with a low velocity inside an intense magnetic field can be trapped within a circular trajectory, with radius r. Mathematically, this is motivated as follows [11]. If an ion, with total charge, z, travelling with velocity, u, is injected into a magnetic field with intensity B, then it is exposed to two forces, the centripetal and the centrifugal force, which are, respectively, equal to:

$$\begin{aligned} F_{\text{centripetal}} &= Bzu \\ F_{\text{centrifugal}} &= \frac{mu^2}{r} \end{aligned} \tag{2-3}$$

Then, the ion movement is based on the equalisation of those two forces, namely:

$$F_1 = F_2 \Rightarrow Bz = \frac{mu}{r} \Rightarrow u = \frac{Bzr}{m} \tag{2-4}$$

thus, the ion movement has a frequency of $f = \frac{u}{2\pi r}$, if we take $2\pi r$ as the circular trajectory. The ion's angular velocity is calculated by:

$$\omega = 2\pi f = \frac{1}{m/z} B \tag{2-5}$$

Hence, its frequency, f, depends on the ion's mass-to-charge ratio, which means that in order to determine the mass we need to determine the frequency. What practically happens is that ions get injected inside a box with an intense magnetic field, generated by a superconducting magnet $(3 - 9.4T)$, [11], and they get trapped into circular motion. Moreover, each ion has a unique cyclotron frequency. Ions with the same frequency, excited for the same period of time with the same energy, will have a circular trajectory of the same radius, grouping the ions into packets, according to their mass. But just the cyclotron movement is not beneficial, thus, a spatially uniform electric field is applied by the *excitation plates.* Then, ions react to their specific cyclotron frequency and start a spiral movement, close enough to the *detection plates.* Now, instead of hitting the detector, the circular movement of ion-groups causes a stream of electrons in the detector circuit, which tries to match the frequency of the moving ions. This creates an image current that can be measured and transformed into the frequency domain by Fourier Transform, so that the mass spectrum is obtained. FTICR analyzers have the highest mass resolution, enabling to distinguish ions that have very small mass differences. However, to reach big resolutions very high vacuum conditions are needed inside the box.

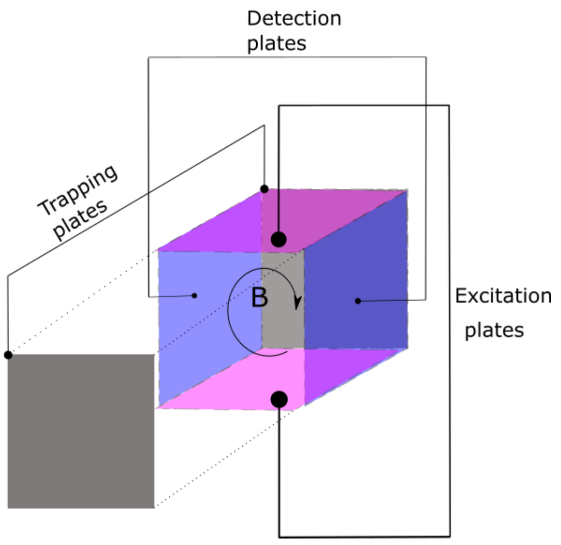


Figure 2-9: Illustration of an ion cyclotron resonance instrument inspired by [11] and [27]. A "trapping" voltage is applied on the electrostatic trapping plates, such that the ions cyclic movement is focused on a specific axis. Then the excitation plates apply a spatially uniform electric field, so that the ions can reach the detector since the trajectory radius increases or to avoid ion collisions or to get expelled from the instrument. Finally, the detection plates measure the induced image current by the ions.

2-3 The IMS Data Structure

In MALDI IMS, the data is captured as follows. After preparing the tissue section and placing it into the mass spectrometer sample inlet, it is analysed (through the ionization and mass analyzer phases) based on a defined spatial (x, y) position array. The measured data are mass spectra where each one is related to each pixel of the spatial dimension, as illustrated in Figure 2-10.

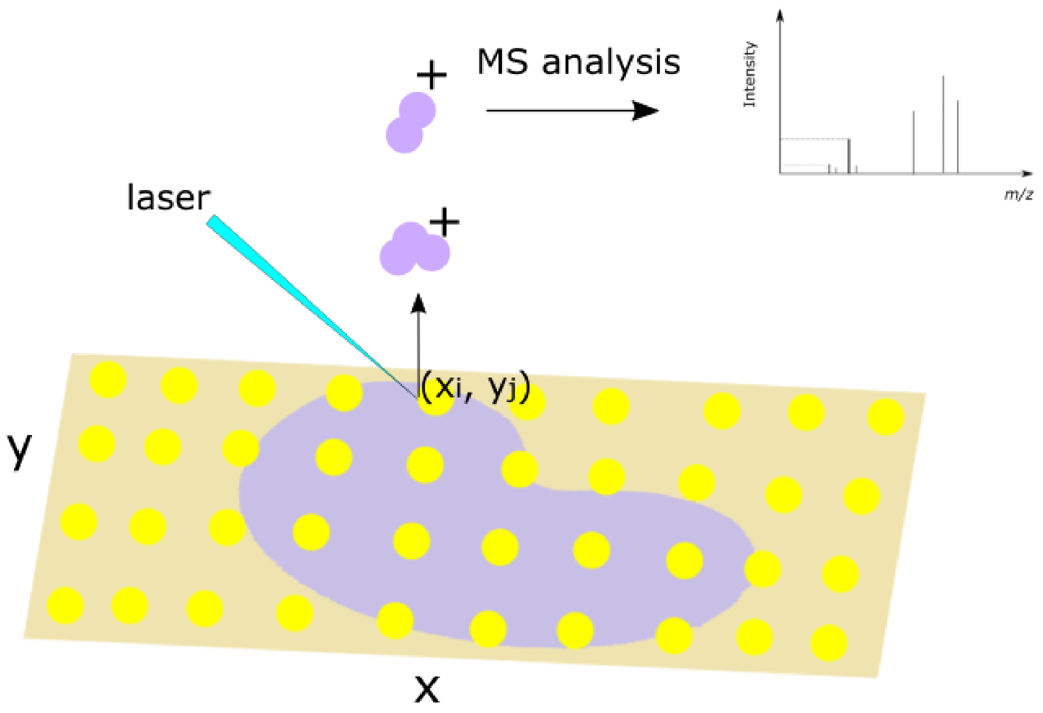


Figure 2-10: IMS experiment recording data, namely a mass spectrum, for one pixel at position (x_i, y_j). If we repeat for every location in the position array we derive the IMS datacube.

By repeating the MS analysis per pixel position (x_i, y_j), the data ends up to be a 2-D matrix where its rows are the number of pixels (or amount of mass spectra) and its columns are the m/z values. Therefore, IMS data follow a 2-D matrix format of size $N \times s$, where N the number of observations (pixel positions) and s, the number of features (m/z ratio with total amount symbolised by s), which is a suitable form for most of the supervised Machine Learning (ML) methods, as depicted in Table 2-1 where the $*$ represents each intensity value corresponding to every m/z value in the mass spectrum. Each column represents a feature.

Table 2-1: 2-D matrix format of IMS data

Pixel position	m/z_1	...	m/z_s
1	$*$	...	$*$
...	...	...	...
N	$*$	...	$*$

Consequently, the IMS data are big size, high dimensional data often comprised of observations $N > 100,000$ of pixels, each of which is characterized by a mass spectrum of length s.

Furthermore, acquiring mass spectra includes noise in the measurements, hence, mass spectra need a detailed cleaning procedure before data analysis, comprised of: baseline correction, smoothing and denoising, spectral centroiding and spectral normalization. Nevertheless, it is important to consider two more characteristics [21] that are connected to IMS data, namely mass resolution, which indicates how well a mass spectrometer differentiates between ions of various m/z ratios and spatial resolution, which indicates the size of the smallest feature that is able to be distinguished. Higher mass resolution means greater m/z precision and higher spatial resolution means smaller pixels and therefore a bigger dataset.

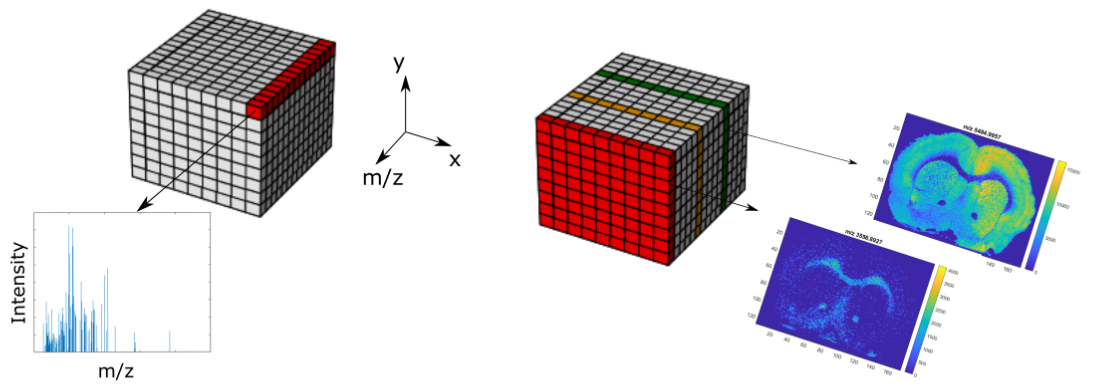


Figure 2-11: A 3-D datacube generated from an IMS experiment

In Figure 2-11, the IMS experiment datacube is displayed, which is comprised of two dimensions, the spatial one and the spectral one. On the left, we highlight the spatial dimension, where each pixel has different spatial coordinates (x,y) and corresponds to a different mass spectrum. On the right, we can see the spectral side of the IMS data, where each m/z bin corresponds to a different detected ion and its distribution on the spatial dimension gives a different ion image. The colorbar scale indicates the intensity of the specific ion at each spatial location.

Chapter 3

Theoretical Background of Convolutional Neural Networks

This chapter will introduce an explanation and definition of the Multilayer Perceptron (MLP), in section 3-1 and describe how we go from fully connected layers to convolutional layers in section 3-2 and their common fundamentals.

3-1 Introduction to Deep Learning and the Deep Neural Network (DNN)

Deep neural networks or alternatively named multilayer perceptrons are Deep Learning (DL) models that can be considered a subcategory of Artificial Intelligence (AI) and Machine Learning (ML) approaches. The most frequently used type of ML, either DL or not, is supervised ML. The aspect that differentiates DL from conventional supervised ML is the fact that the latter needs a meticulous and highly specialized feature engineering implementation. Hence, in conventional ML, feature engineering is needed in order to change the raw data into an appropriate representation (e.g. image-pixel intensities), from one level to another, internally, or else to extract features from raw data, which are then used to train the model and enable it to indicate or classify motifs in the input dataset. On the other hand, DL uses representation learning to fulfill this task. It, mainly, combines feature extraction and classification in a single step, as visualised in Figure 3-1 on the right. This is very useful for cases where feature extraction is very difficult to define manually.

Representation learning is based on the extraction of representations automatically. A representation is actually the set of features which describe the data [15]. End-to-end learning (E2E) concerns training by applying learning to the system in total, using the end target-output to learn features from the initial input based on gradient descent.

Before describing the fundamentals of Convolutional Neural Networks(CNNs), the intention is

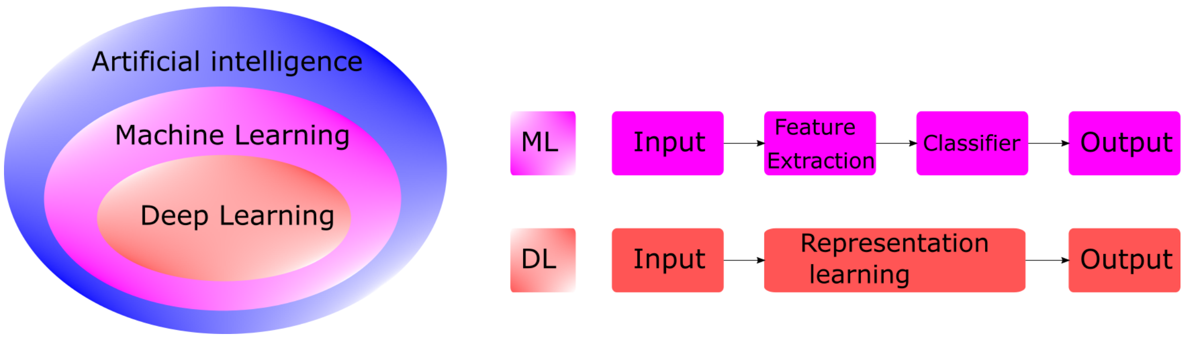


Figure 3-1: Illustration of interconnection between AI, conventional ML and DL. The feature extraction and classification steps are replaced by the single step of representation learning in DL.

to describe the fundamental operations in a DNN, also called a multilayer perceptron (MLP). A DNN gives a map representation, $f(X;\theta)$, with X the input and θ corresponding to the weights and the biases, and updates the θ parameters that lead to the most accurate approximation of the function f [15]. A representative illustration of how a DNN works can be seen in the following figure.

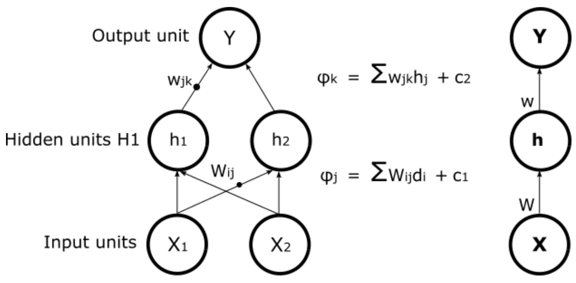


Figure 3-2: Basic illustration of a DNN with one hidden layer [Image inspired from I.Goodfellow et al., Copyright ©2016, [15]]

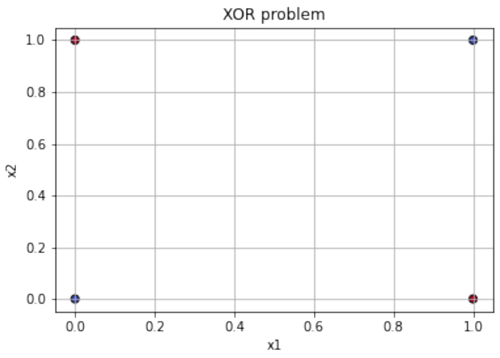


Figure 3-3: XOR toy-example visualization. The blue dots stand for "0-label" (negative class) and the red dots stand for "1-label" (positive class).

In Figure 3-2, we can see an example of a feedforward DNN with just one hidden layer H1, X_1, X_2, the input features, Y, the output feature and h_1, h_2 the hidden neurons that are computed internally. Additionally, for each neuron connection (arrow) there is a corresponding weight, w. For the hidden layer, the weights are W_{ij} with $i \in$ Input and $j \in$ Hidden Output, corresponding to $i = 1, 2$ and $j = 1, 2$. For the output layer, the weights are w_{jk} with $j \in$ Hidden Output and $k \in$ Output, corresponding to $j = 1, 2$ and $k = 1$. Additionally, for each layer a bias c is added to enhance the capacity of the model. Therefore, in this example, there are 9 parameters, every weight on the neuron connections (6) and each bias on each layer (3). The right side of the graph is the same as the left one with the only difference that the right side is written in matrix notation. More specifically, the complete network is determined as, with g denoting the activation function:

$$g(X_1W_{11} + X_2W_{21} + c_1)w_{11} + g(X_1W_{12} + X_2W_{22} + c_2)w_{21} + b = \mathbf{w}^T g(\mathbf{W}^T\mathbf{X} + \mathbf{c}) + b \tag{3-1}$$

The left hand side and the right hand side of Equation 3-1, correspond to the feedforward notation on the left and on the right of Figure 3-2, respectively.

Nevertheless, g, the activation function can be either a linear or a non-linear function. Non-linear transformation of the input features is more flexible and can capture more complex data distributions.

To indicate the need for the non-linear activation functions a toy-example is going to be used, namely, the XOR/"exclusive or" problem [15]. The one-hot encoded dataset for the XOR problem is shown in Table 3-1, which is visualized in Figure 3-3.

Input		**Label**
X_1	X_2	**t**
0	0	0
0	1	1
1	0	1
1	1	0

Table 3-1: One-hot encoded dataset of the XOR problem

If we had a single-layer, linear perceptron (g a linear activation) with, **c=0**: $\mathbf{X}^T\mathbf{W} + b$, then the network wouldn't be able to learn the data structure for the XOR problem. For example, if *b=1* and *W =0* then the output would be 1 for any input. The conclusion is that a single linear layer network cannot be used and a hidden layer should be added to the model, which must be non-linear. The most common used non-linear activation functions are the Rectified Linear Unit (ReLU) and the sigmoid [43].

3-1-1 Non-linear Activation Functions

ReLU The ReLU function is defined as:

$$y_{\text{ReLU}} = \max(0, X) \tag{3-2}$$

where X is the input tensor and y_{ReLU} is the output tensor. With this function we can define a forward pass of ReLU activation layer. In Figure 3-4, we can see the plot of the ReLU function. Additionally, if the input is less than zero, then the gradient of the ReLU function is equal to 0. On the contrary, if the input is exactly equal to 0, then the function is not differentiable and in this condition, 0 value is also taken. If the input is larger than 0, then the function's derivative is equal to 1 as illustrated in Figure 3-6. In conclusion, the derivatives of the ReLU function make it so popular in use because they either extinguish the value or they keep it the same, which alleviates the vanishing gradient problem and helps the optimization process [43].

Sigmoid The sigmoid function is defined as:

$$y_{\text{sigmoid}} = \frac{1}{1 + \exp(-X)} \tag{3-3}$$

a plot of this function is visible in Figure 3-5 created with matplotlib in Python. Sigmoid functions are smooth and differentiable (Figure 3-7) and are mainly used as activation functions on the output layers, mostly for classification problems, since it is a special case of softmax with number of classes $\nu = 2$. The softmax function is mainly used for multiple output classification probelms and is defined as [15]:

$$\text{softmax}(X_i) = \frac{e^{X_i}}{\sum_{j=1}^{\nu} e^{X_j}} \tag{3-4}$$

On the other hand, in hidden layers it is more typical to use ReLU activation functions, which are simpler and easier to train [43].

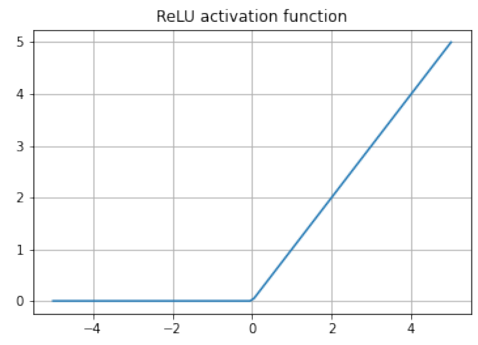


Figure 3-4: ReLU non-linear activation function

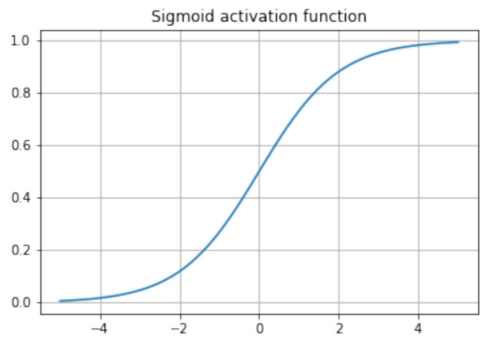


Figure 3-5: Sigmoid non-linear activation function

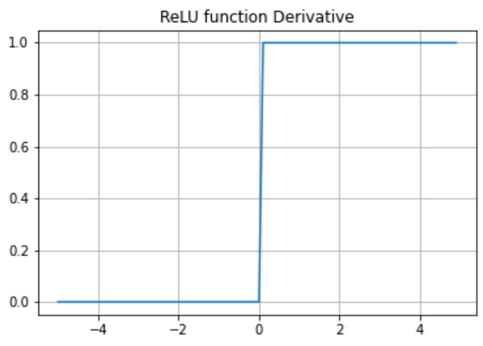


Figure 3-6: ReLU activation function derivative plot

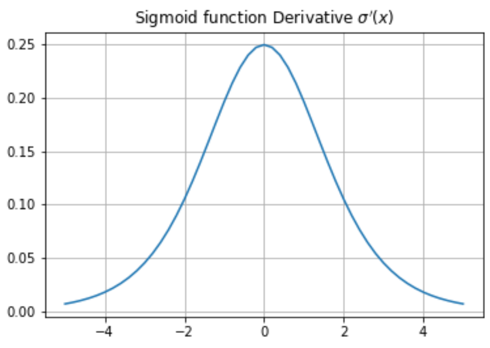


Figure 3-7: Sigmoid activation function derivative plot

Hence, while designing deep neural networks we get the design choice of activation functions.

In general, the training loop of a network is comprised of:

- presenting a training dataset - noisy, not exact examples of the ground truth function- and getting output predictions (forward pass)
- comparing the predicted output with the ground truth (loss computation)

- updating the hyperparameters accordingly in order to reach the best approximation of the function to be learned (backward pass and optmization step)

The comparison between output and ground truth is acquired by computing the loss, which is defined by the *objective/cost/loss/error* function, which is also a design choice, and then, the parameters of the network are updated using the back-propagation step.

3-1-2 Back-propagation

In section 6.5 of [15] and 4.7.3 of [43], one can dive into the back-propagation algorithm, which enables the information from the loss to move from the output layer backwards to the input layer, so that the gradient with respect to each parameter is calculated. The back-propagation step is actually an algorithm that efficiently calculates gradients to reduce the computational cost of this procedure. In other words, back-propagation is a smart way to apply the chain rule of calculus. Additionally, it is often misinterpreted that the back-propagation algorithm is merely referring to multi-layer neural networks, whereas it can actually be used to calculate derivatives for any function. A deeper insight on the back propagation step is given by using computational graphs.

Computational graphs: Every square represents a variable, which can be a vector, matrix, tensor, a scalar or other and every circle denotes an operation as in Figure 3-8. An operation is just a function applied on the variables. Furthermore, an operation gives back just one output variable. Finally, the topological ordering in the graph starts from the lower-left node and moves upwards and so on so forth. An example of a computational graph and the involved calculations is following, inspired by [43] in section 4.7.2. For simplicity, let's assume that we have an input $\chi \in R^s$ and a single hidden layer, with dimension h that doesn't include a bias term.

Then, if we perform the forward pass, it gives the intermediate variable $\phi = w\chi$, with $w \in R^{h \times s}$, the weight parameter of the layer.

Via the sigmoidal activation function σ we obtain the output $y_\sigma = \sigma(\phi)$.

Next, with t denoting the truth value vector and assuming that we have a mean squared error loss function, we get the loss term as: $L = \frac{1}{2}(y_\sigma - t)^2$.

Adding a regularization term R, with known hyperparameter λ, we have: $R = \frac{1}{2}w^2$. Finally, the regularized objective/loss/cost/error function is then: $J = L + \lambda R$

The above described simplistic network is depicted in Figure 3-8 and it shows the interconnections between the calculations. Based on this, we are, next, going to perform the *back-propagation* step, in order to present an example of how it works.

1. The initial step of the back-propagation algorithm is to calculate the gradient of the regularized function J with respect to the loss L and the regularization term R:

$$\begin{aligned} \frac{\partial J}{\partial L} &= 1 \\ \frac{\partial J}{\partial R} &= \lambda \end{aligned} \tag{3-5}$$

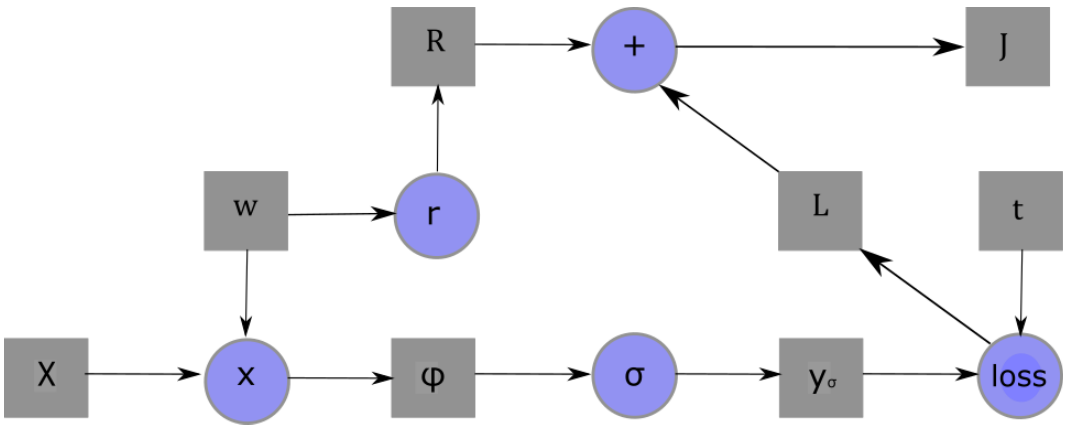


Figure 3-8: Computational graph for the above presented simple network - Grey squares denote variables and blue circles denote operations, based on an example in [43]

2. Next, the gradient of the regularized function J with respect to the output variable y needs to be computed, based on the chain rule of calculus:

$$\frac{\partial J}{\partial y_\sigma} = \frac{\partial J}{\partial L}\frac{\partial L}{\partial y_\sigma} = \frac{\partial L}{\partial y_\sigma} = y_\sigma - t \tag{3-6}$$

3. Following, the gradient of the regularized function J with respect to variable ϕ can be calculated with the chain rule and substituting already computed gradients:

$$\frac{\partial J}{\partial \phi} = \frac{\partial J}{\partial y_\sigma}\frac{\partial y_\sigma}{\partial \phi} = \frac{\partial J}{\partial y_\sigma}\sigma^{'}(\phi) \tag{3-7}$$

   where $\frac{\partial J}{\partial y_\sigma}$ has already been calculated in Equation 3-6.

4. Finally, the derivative of the regularized function J with respect to the weight can be computed as follows, where $\frac{\partial J}{\partial \phi}$ is calculated from Equation 3-7:

$$\frac{\partial J}{\partial w} = \frac{\partial J}{\partial \phi}\frac{\partial \phi}{\partial w} + \frac{\partial J}{\partial R}\frac{\partial R}{\partial w} = \frac{\partial J}{\partial \phi}\chi^T + \lambda w \tag{3-8}$$

The above steps conclude the application of the back-propagation algorithm for the calculation of the derivative with respect to the weight, on this simple network used as a toy-example.

3-1-3 Gradient-based Learning

After the gradients with respect to the trainable parameters are computed, we can minimize the objective function, namely the loss, by a method named *gradient descent*, as described in section 4.3 of [15] and [25]. This method suggests an update of the parameter θ that leads to a smaller loss function value based on:

$$\theta^{'} = \theta - \epsilon\nabla_\theta J(\theta) \tag{3-9}$$

where ϵ is the *learning rate*, namely the step size of the optimization.

3-1-4 Optimization Techniques for Training DL Models

There are various optimization methods employed in deep learning, we will focus on the method that has been chosen in this thesis.

Stochastic Gradient Descent (SGD) It is probably one of the most applied optimization algorithms in DL. In section 8.3.1 of [15] and 11.4 of [43] the authors give a good description of this method. The optimization algorithm for SGD collects a number of samples M, from the total training set, then the gradient is estimated:

$$\hat{g} = \frac{1}{\mathrm{M}} \nabla_\theta \sum_{i=1}^{M} J_i(\theta) \tag{3-10}$$

then, based on that estimate and the selected learning rate, the parameter is updated as in Equation 3-9, until a stopping criterion is met. The stochasticity lays in the fact that not the total amount of the training data is used for the estimation of the gradient, in order to reduce computational cost but this leads to a noisy output.

SGD with Momentum The objective of the momentum learning rule is to address the poor conditioning on the Hessian. In addition, it aims to deal with the varying output of SGD [15]. Eventually, momentum is used to accelerate convergence and stabilize the training process. In this method, a variable v is introduced, which is analogous to physical velocity. It controls the direction and speed of the parameters in the parameter space during learning. The variable v is defined as an exponentially reducing value. Moreover, a hyperparameter $\alpha \in [0, 1)$ is introduced, which controls how fast the previous gradients decay. Finally, the parameter θ value is updated according to v. This is described by the following set of equations:

$$\begin{aligned} v &= \alpha v - \epsilon \hat{g} \\ \theta' &= \theta + v \end{aligned} \tag{3-11}$$

Root Mean Squared Propagation (RMSProp) This is a method of adaptive learning rates [15]. The higher the partial derivative of the loss function with respect to parameter θ, the more rapid the decrease of θ's learning rate. On the contrary, the lower the partial derivative corresponding to θ, the smaller the decrease in its learning rate. The RMSProp keeps an exponentially decaying moving average, d of the squared gradient, $\hat{g}^2$, for each parameter. Decay rate ρ is a parameter between 0 and 1 that determines how much the past gradients contribute to the moving average. It also uses a global learning rate ϵ. Scaling the gradient by the root mean square benefits the learning. The parameter update rule for this method is described by the following equations:

$$\begin{aligned} d &= \rho d + (1 - \rho)\hat{g}^2 \\ \theta' &= \theta - \frac{\epsilon}{\sqrt{\delta + d}} \hat{g} \end{aligned} \tag{3-12}$$

where δ is typically chosen as 10^{-6} and a new hyperparameter is added, namely ρ, which controls the length scale of the moving average.

Adam *Ada*-ptive *m*-oments is another adaptive learning rate optimization algorithm [15], [22]. It can be viewed as the combination of momentum technique and RMSProp. Thus, the formulas including bias corrections for the weight update with Adam optimizer are a combination of Equation 3-11 and Equation 3-12:

$$\begin{aligned} v &= \alpha v - \epsilon \hat{g} \\ \hat{v}_\tau &= \frac{v}{1-\alpha^\tau} \\ d &= \rho d + (1-\rho)\hat{g}^2 \\ \hat{d}_\tau &= \frac{d}{1-\rho^\tau} \\ \theta' &= \theta - \epsilon \frac{\hat{v}_\tau}{\sqrt{\hat{d}_\tau + \delta}} \end{aligned} \tag{3-13}$$

In conclusion, Adaptive Moments (Adam) is considered better than plain SGD since it incorporates the momentum as well as the RMSProp advantages, [22]. Choosing an optimizer is going to be one of our design choices in the next sections.

3-2 Convolutional Neural Networks

Previously, the design and implementation of DNNs with fully connected layers, in which every neuron from each layer is connected to all neurons in the next layer has been presented. In [25] as well as in chapter 9 of [15], a good insight in CNNs operation principles is given.

3-2-1 Convolution

In order to explain convolutional neural networks we will start with the convolutional operation. First of all, it can be multi-dimensional; 1D for data such as signals/sequences, spectra or natural language processing, 2D for data like images and 3D for video data or volumetric images [15]. In general, the definition of the convolution operation, denoted by $*$, between two functions in discrete time, is given as [15]:

$$s(n) = X(n) * w(n) = \sum_{k=-\infty}^{\infty} X(k)w(n-k) \tag{3-14}$$

In CNNs, w is referred to as the kernel, X as the input and the output of the convolution operation, s, is referred to as the feature map. The input and the kernel are arrays that are used as tensors in PyTorch. Additionally, we calculate the convolution sum for a finite number of input and kernel array positions where we store their values, while everywhere else it is assumed that they have a value of zero.

2D-Convolution Operation

In chapter 9 of [15] and in chapter 6.2.1 in [43], the 2D convolution operations are described. An illustration on a 2D input is shown in Figure 3-9, inspired by [43].

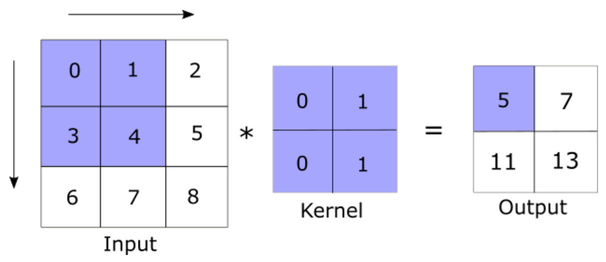


Figure 3-9: Illustration of 2D convolution operation. The blue-shaded elements are participating in the computations to give the top left corner element of the output

In Figure 3-9, the output element on the top left corner is derived from the application of the convolution operation which is:

$$\text{Output} = 0 \times 0 + 1 \times 1 + 3 \times 0 + 4 \times 1 = 5 \tag{3-15}$$

To obtain the rest of the output values the kernel "window" is slid across the input on both horizontal and vertical directions. Since the edge elements of the input cannot be covered by a 2×2 kernel the output has a reduced dimension, as we can see in Figure 3-9, the input is reduced from 3×3 to 2×2. Furthermore, the 2D convolution operation integrates the spatial dimensions of the data into the network's layers, thus, it is used successfully in image-like data.

1D-Convolution Operation An illustration of this operation in Figure 3-10 is presented next.

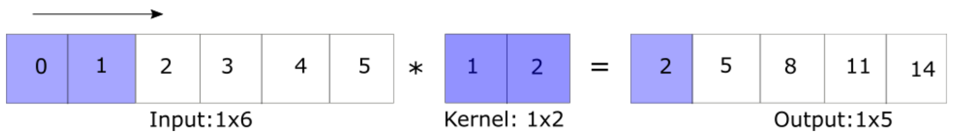


Figure 3-10: Illustration of 1D convolution operation. The output value is computed as: $0 \times 1 + 1 \times 2 = 2$

1D CNNs are beneficial, when 1D data are concerned, compared to their 2D counterparts because of the next facts:

- less computational complexity
- fewer hyperparameters to be tuned
- low-cost and suitable for real-time applications

Receptive Field

The area of the input layer that the hidden layer is able to "see" is called the *receptive field*. Based on chapter 7.2.6 of [43], the receptive field can be presented with the assistance of the illustration in Figure 3-9. In Figure 3-9, the output node on the top-left is affected by the four elements with the blue shadow in the input, given the 2×2 convolution kernel and this is defined as the receptive field. Additionally, in CNNs, the receptive field of elements in deeper layers is larger than the shallower layers one [15]. Again, this can be demonstrated with the help of the example illustrated in Figure 3-9, with the addition of one more kernel of dimension 2×2 as shown in Figure 3-11 (top).

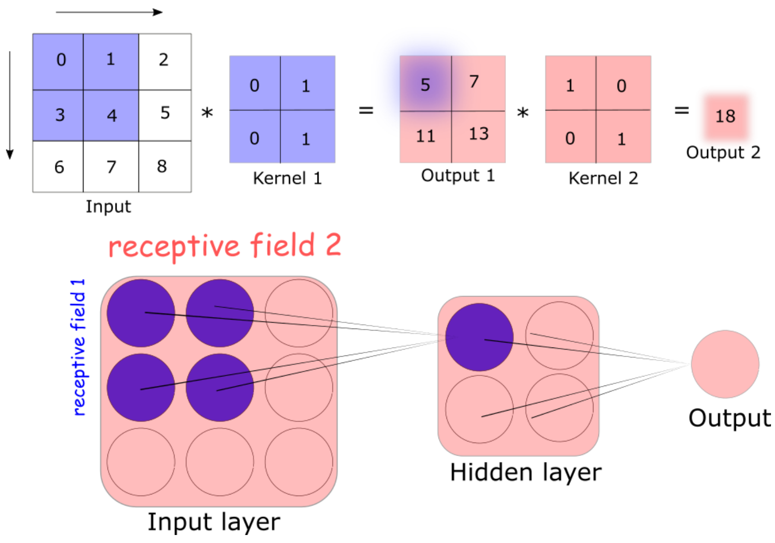


Figure 3-11: Receptive field illustration for a toy-example of two layers. On the top, the participating neurons in the calculation of the corresponding outputs are indicated. On the bottom, *receptive field 1*, corresponds to the region that affects the output in the top left neuron of the hidden layer. On the contrary, *receptive field 2*, corresponds to the region that affects the final output. The deeper the layer the larger the receptive field, as the final output "sees" all of the input layer neurons.

In Figure 3-11, we can see the receptive field of two different neurons. As already mentioned, the first one - the top left blue one in the hidden layer - only "sees" the four blue nodes from the input layer that were used for its calculation. The second one - the final output in pink colour - sees the whole input layer, since its calculation is affected by all of the neurons. Hence, whenever input features lie across a wider region, we should construct a deeper CNN, so that these features are detected from the hidden layers.

Padding

As already seen in Figure 3-9, the dimension of the input gets reduced after the convolution operation application (from 3×3 to 2×2, when a 2×2 kernel is applied). This effect is due to the fact that we lose pixels that are surrounding the 2D input and thus, they cannot be covered by the convolutional operation. In order to deal with this loss, extra pixels around the perimeter of the 2D image data are added, assigned the value of zero [43]. This procedure is called *padding*.

Striding

While applying the convolution operation, we begin with the kernel window at the top-left position (as shown in the blue shadow in Figure 3-9) of the input tensor and then slide the window around each position horizontally as well as vertically to cover the whole region. In the toy-example of Figure 3-9 the window was slid one step at a time, although, intermediate positions are skipped, so that computational cost is reduced and downsampling is accomplished. This act is called striding and when designing a CNN we refer to the amount of skipped steps as *stride* and it is a parameter that needs to be defined by the designer of the architecture [43].

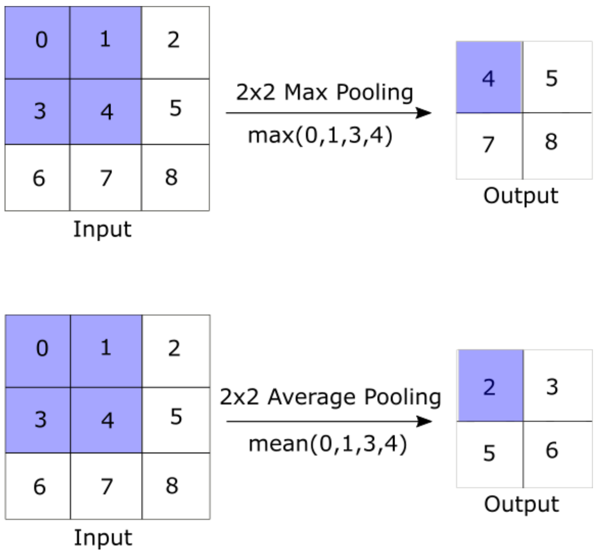


Figure 3-12: Illustration of maximum and average pooling

Pooling

Another important component of CNN architectures is pooling.Typical architectures employ *pooling* layers right after convolutional layers that basically summarize the result around a region by summing information in order to achieve a bigger receptive field of each neuron, up to the input layer, as we go further into the depth of the network [15] and [43]. This is particularly useful for cases where a general question is addressed, such as a question of image classification. For example, if the classification question is whether the image contains a dog, then, the nodes of our output layer should "have access"/be affected by the whole input image. Summing up, pooling layers aim at alleviating the shifts of location after convolutional layers and at downsampling detected features, chapter 6.5 [43]. Typically used pooling layers in literature are maximum and average pooling which are described visually in Figure 3-12.

In Figure 3-12, we can see that the maximum pooling leads to a downsampling of the input by using the computation of the function, $max()$ that extracts the maximum value out of each location, whereas the average pooling calculates the average value between the elements in each location. Input size from 3×3 is reduced to 2×2 enabling memory saving.

Summing up, a visual comparison to highlight the differences between a DNN model and a CNN model is presented in Figure 3-13. Convolutional neural networks are comprised of convolutional layers which learn automatically representations and fully connected layers which output the classification result. The trainable parameters of a Convolutional Neural Network (CNN) are the weights and biases of both the Fully Connected (FC) units and the convolutional kernels.

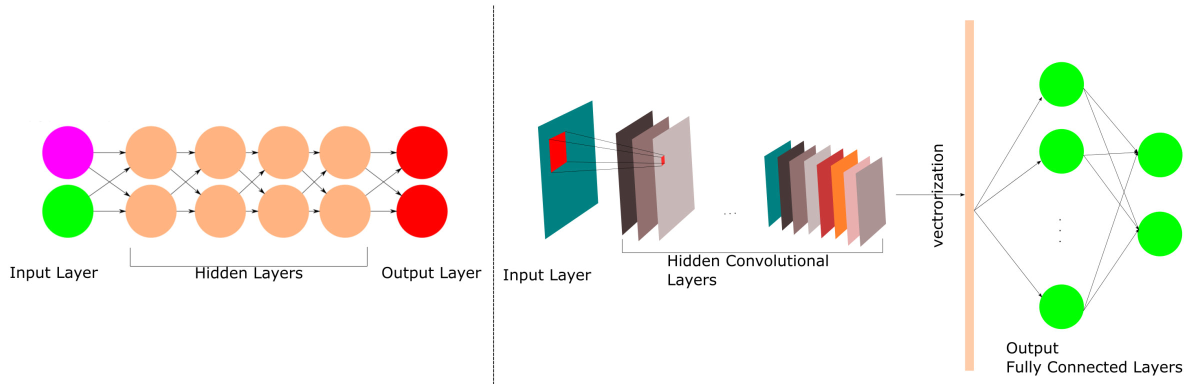


Figure 3-13: DNN vs CNN. (Left) Deep neural network with FC architecture.(Right) 2-D CNN architecture.

3-2-2 Regularization Techniques

In [15], the definition of regularization techniques is given as a reduction of the test error with the possibility to raise the error of the training set. On the other hand, regularization techniques can also be defined as techniques to alleviate the overfitting phenomenon. Some of the most popular techniques used nowadays are the following.

L2 Regularization L2 regulatization can also be called weight decay and is one of the most commonly used regularization techniques. The intuition behind this method is the measurement of a function's distance to zero. Therefore, the L_2 norm of the weights is introduced, which is equal to, [43] section 2.3.10:

$$||w||_2 = \sqrt{\sum_{i=1}^{n} w_i^2} \tag{3-16}$$

The term $||w||_2$ is added as a penalty to the loss function $L(w, b)$. Hence, now the optimization problem becomes the problem of minimizing the sum:

$$L(w, b) + \frac{\lambda}{2}||w||_2 \tag{3-17}$$

where w, b represent the weights and biases and λ is a regularization constant divided by 2 due to the convention of making the output look simpler after calculating the gradient. For a positive λ, the value of w is restricted.

Early Stopping Stop the training of the model as soon as it starts to overfit on the training dataset, instead of training the network for a standard epoch number.

Dropout Some neurons are actually dropped out during the training process. On every iteration while forward propagating, what happens is that part of the nodes get zero values before moving on to the next layer, in this way overfitting is alleviated.

3-2-3 Performance Evaluation

After training a CNN model it is important to evaluate its performance on a testing dataset, based on some metrics. In literature, several metrics for evaluating performance are encountered for IMS classifiers, some of them are presented next, where True Positives (TP), True Negatives (TN), False Positives (FP) and False Negatives (FN).

To start with, the classification performance of the models will be calculated and presented based on the confusion matrix, which is defined as follows.

Table 3-2: Illustration of confusion matrix for binary classification

	Predicted as Negative	**Predicted as Positive**
Truely Negative	TN	FP
Truely Positive	FN	TP

In Table 3-2, we see the confusion matrix for the case of binary classification. By analyzing the values on the confusion matrix we can determine our model's ability to properly classify different classes and we can calculate several performance metrics which are defined based on TN (True Negatives), FN (False Negatives), TP (True Positives) and FP (False Positives). It is easily understandable that the TN represent the data that actually belong to the negative class and are correctly predicted by the model to belong to the negative class. Additionally, the FN, correspond to data that actually belong to the positive class but are misclassified as belonging to the negative class. Alternatively, the model fails to recognise those observations as positive and incorrectly assigns them to the negative class. The same logic applies for the TP and FP in the oppoisite class.

Accuracy

The metric of overall accuracy, which is defined as:

$$\text{Accuracy} = \frac{TP + TN}{TP + FP + TN + FN} \tag{3-18}$$

Balanced Accuracy

In [4], balanced accuracy is used as an evaluation metric to compensate for the imbalanced classes, which is defined as:

$$\text{Balanced Accuracy} = \frac{1}{2}(\frac{TP}{P} + \frac{TN}{N}) \tag{3-19}$$

where P is the number of actual positives and N the number of actual negatives. This metric is not biased by the data distribution compared to the accuracy metric presented above. During cross-validation the median value of the balanced accuracy is calculated in [4].

Precision

The metric of precision is given as:

$$\text{Precision} = \frac{TP}{TP + FP} \tag{3-20}$$

Recall

It is the same as sensitivity or else it can be defined as the true positive rate. It measures the amount of the positive category (e.g. diseased tissue) that is correctly classified. Furthermore, it has the following formula:

$$\text{Recall} = \frac{TP}{TP + FN} \tag{3-21}$$

F1 score

The F1 score is a commonly used metric for evaluating the performance of a classification model, particularly in situations where the classes are imbalanced. It combines precision and recall into a single value that represents the overall effectiveness of the model.

$$\text{F1 score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{3-22}$$

Nonetheless, the denominators of the above given relations need to be different from zero so that they can be defined.

Chapter 4

Theoretical Background of Techniques for Hyperparameter Tuning, Classification and Interpretation

First of all, in this chapter, we describe the theoretical background of the rest of the applied methodologies for the purpose of this thesis. In section 4-1, we talk about the applied automated methodology for the hyperparameter tuning and consequently, the Convolutional Neural Network (CNN) architecture design, namely Tree-structured Parzen Estimator (TPE). In section 4-2, we define the theoretical models of the used linear baseline methods and we describe the alternative feature extraction method used instead of the convolutional layers. Finally, in section 4-3, we define the used methodology for finding the feature importances that aid the interpretation of the designed CNN models.

4-1 Hyperparameter Tuning

Searching for the best hyperparameters is an optimization problem. In this section, we explore an automated Bayesian hyperparameter optimization and CNN architecture design methodology. Bayesian optimization [14] isn't a specific methodology but rather refers to the category of optimization algorithms that are based on Bayes' rule. The algorithms belonging to the Bayesian optimization category don't require gradient computation of the objective function. The objective function is allowed to be a mere "black box", meaning that a closed-form expression of the objective function is not necessary. The relaxation on the need for gradient computation is valuable in hyperparameter optimization problems due to the fact that the gradient of the objective function does not always exist, for instance, when the hyperparameters are discrete.

Additionally, Bayesian optimization selects hyperparameter observations indirectly, based on acquisition functions and the evaluation of those locations. An acquisition function should be chosen carefully, so that exploration and exploitation of the search space is accomplished. Exploitation refers to collecting observations in regions where the objective function most likely is optimized, while exploration refers to collecting observations from new regions, where we are not sure about the objective function.

4-1-1 The Tree-structured Parzen Estimator (TPE) Approach

A method that belongs to the Bayesian optimization category is the Tree-structured Parzen Estimator (TPE) algorithm. Tree-structured refers to the conditional form that the configuration search space can take. For instance, in a CNN, if we use two hidden convolutional layers then we should determine the hyperparameters for both the first and the second hidden layer independently. However, if the option is to use just one hidden layer there is no need to define the hyperparameters for the second layer. This means that some hyperparameters are conditional and lead to a tree-structured search space. On the other hand, Parzen estimators or Kernel Density Estimators (KDEs) refer to the method of estimation of the probability density functions. The TPE strategy models $p(\eta|\psi)$, namely the probability of a hyperparameter observation $\{\eta^{(1)}, \eta^{(2)}, ..., \eta^{(N)}\}$ from the configuration space H, given a certain loss ψ, which is used for the evaluation of the acquisition function that determines the search direction. TPE determines $p(\eta|\psi)$ based on densities across the search space H, estimated using Parzen estimators; alternatively called KDEs [8, 38]:

$$p(\eta|\psi) = \begin{cases} l(\eta) \text{ if } \psi < \psi^* \\ \\ g(\eta) \text{ if } \psi \geq \psi^* \end{cases} \tag{4-1}$$

In Equation 4-1, $l(\eta)$ is the probability density function (or just density) resulting from using the hyperparameter observations $\{\eta^{(i)}\}$ that correspond to loss $\psi = f(\eta^{(i)})$ that is less than ψ^*. The ψ^* is defined as a threshold, a quantile γ of the observed losses ψ, to distinguish between "good" and "bad" observations. On the other hand, $g(\eta)$ is the density function of the remaining observations that correspond to a loss ψ that is larger than ψ^*.

In TPE, the acquisition function that is used to decide the next best location to sample is the Expected Improvement (EI) criterion, which is defined as [8] - if we parameterize $p(\eta, \psi) = p(\eta|\psi)p(\psi)$:

$$EI(\eta) = \int_{-\infty}^{\psi^*} (\psi^* - \psi)p(\psi|\eta)d\psi = \int_{-\infty}^{\psi^*} (\psi^* - \psi)\frac{p(\eta|\psi)p(\psi)}{p(\eta)}d\psi \tag{4-2}$$

We define the percentile split threshold as $\gamma = p(\psi < \psi^*) = \int_{-\infty}^{\psi^*} p(\psi)d\psi$ hence TPE models $p(\eta)$ as follows, $p(\eta) = \int p(\eta|\psi)p(\psi)d\psi = \gamma l(\eta) + (1-\gamma)g(\eta)$, which after substitution in Equation 4-2 leads to the relationship

$$EI(\eta) = \frac{\gamma\psi^* l(\eta) - l(\eta)\int_{-\infty}^{\psi^*} p(\psi)d\psi}{\gamma l(\eta) + (1-\gamma)g(\eta)} \propto (\gamma + \frac{g(\eta)}{l(\eta)}(1-\gamma))^{-1} \propto \frac{l(\eta)}{g(\eta)}$$

This relationship indicates that in order to maximize the expected improvement we desire to have hyperparameters $\{\eta^{(i)}\}$ with high probability $l(\eta)$ and low probability $g(\eta)$. For every iteration, the TPE algorithm outputs the next observation η^* that achieves the highest EI. This is illustrated in the following figure.

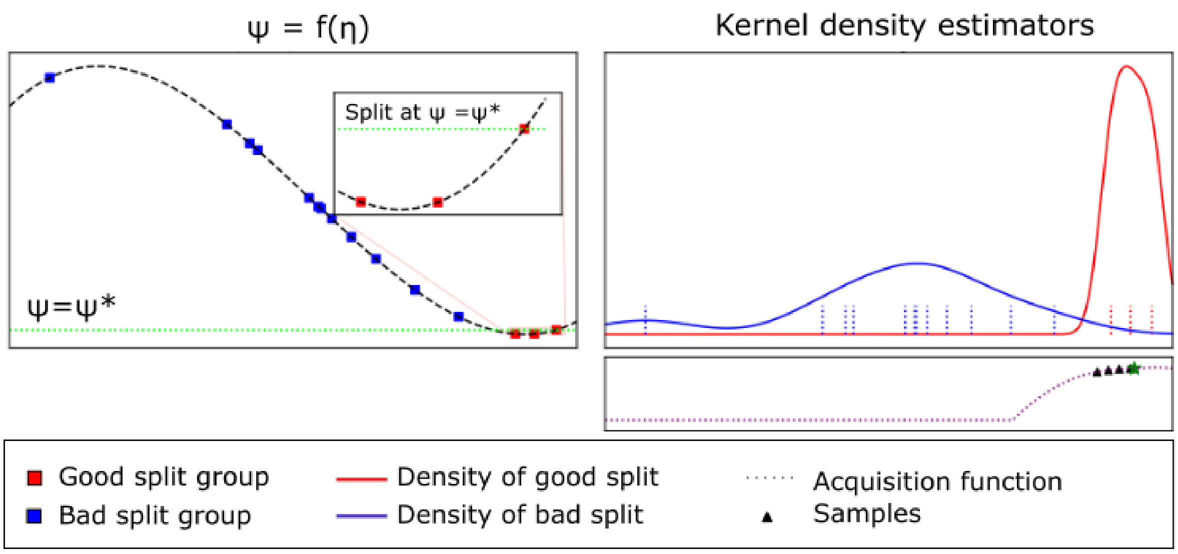


Figure 4-1: Three figures illustrating the fundamental concept of TPE algorithm. (Left) Observations of the objective function $\psi = f(\eta)$ split in two sample groups. The red samples correspond to the "good" split and the blue samples to the "bad" split. The green dashed line is the split threshold ψ^*. (Top right) The densities calculated using KDEs for the "good" (red) and "bad" samples (Bottom right) The $\frac{l(\eta)}{g(\eta)}$ ratio in the acquisition function that determines the highest EI. The configuration with the highest EI is chosen for the next iteration denoted by a star among the triangle-shaped samples. [Image taken from Watanabe ©2023 [38]]

In Figure 4-1, we observe the collection of 14 samples, and their corresponding loss ψ. The 3 out of the 14 are recorded in the group of the "good" and the rest 11 are recorded in the group of "bad" based on a top-quantile threshold ψ^* or else γ. Furthermore, on the top right figure, the probability density functions of the "good" split group and the "bad" split group are approximated using Parzen estimators/KDEs. The basic idea behind KDEs is to approximate the density by constructing a weighted sum of kernel functions centered around each hyperparameter value $\{\eta^{(i)}\}$ for a number of observations, N [38]:

$$p(\eta|\psi) = w_0 p_0(\eta) + \sum_{i=1}^{N} w_i K(\eta, \eta_i|\sigma) \tag{4-3}$$

Equation 4-3 estimates the densities where w the weights, K is the kernel function, σ is the bandwidth and p_0 is the prior. Typically, a kernel function K that is used in TPE for numerical hyperparameters is the Gaussian kernel, which is defined by the standard normal distribution [8, 38]. Furthermore, in Figure 4-2 and Figure 4-3, we illustrate how the Gaussian KDEs look for an amount of hyperparameter observations from one up to six samples. Through those figures, we show the main idea that each hyperparameter observation determines a normal distribution with mean equal to the observation and a bandwidth, equal to the standard deviation, which is set based on the greatest distance between neighbouring samples.

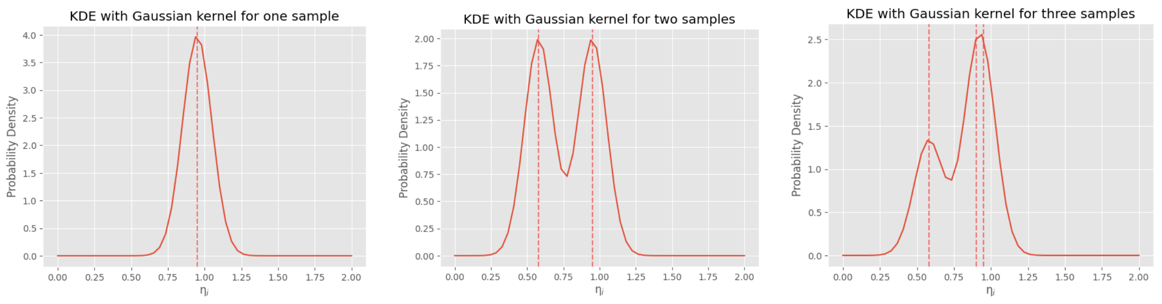


Figure 4-2: Kernel density estimators with Gaussian kernel for one, two and three hyperparameter samples respectively

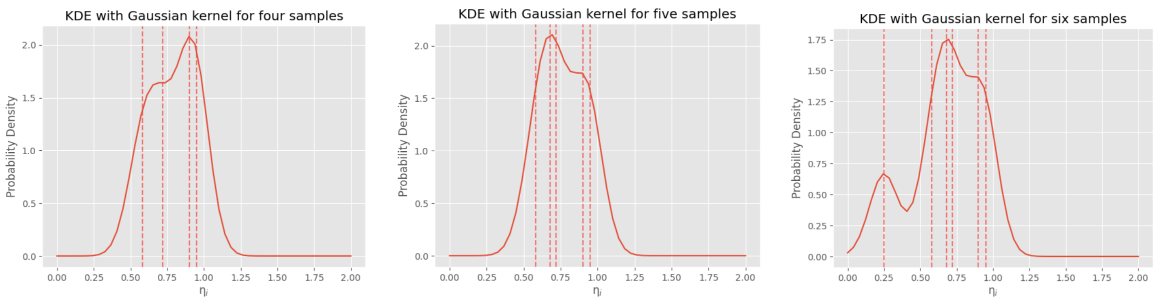


Figure 4-3: Kernel density estimators with Gaussian kernel for four, five and six hyperparameter samples respectively

To sum up, the first step for the Tree-structured Parzen Estimator (TPE) is to gather initial data. One common and straightforward approach is to initially perform a few iterations of Random Search. Next, we split the collected observation pairs (η_i, ψ_i) into two groups - based on the splitting threshold ψ^* - the "good", corresponding to the ones that gave low losses after evaluation and the "bad", corresponding to the rest observations. Then, the likelihood probability of belonging to one of these groups is modeled using KDEs and Gaussian kernel functions for numerical hyperparameters. Then, the improvement criterion is maximized according to the ratio of good over bad density and the next location is taken for exploration. The algorithm stops when the user defined maximum number of trials has been reached or if the computational resources are fully utilized. Finally, TPE is a Bayesian optimization technique that efficiently searches the hyperparameter configuration space H(e.g. number of convolutional layers, learning rate e.t.c.) to determine a configuration that optimizes our model. Subsequently, we can use it to select the optimal architecture for our 1D CNN models.

4-2 Approaches to IMS Data Classification apart from CNNs

4-2-1 Linear Models as Reference Methods

Linear Discriminant Analysis (LDA)

Assume we have a random input vector $X \in \mathbb{R}^S$, where S the total number of features, and $Y \in \mathbb{R}$ a random output which is the target value of the annotated input. Let's assume that

k denotes the classes, then $f_k(X)$ is the class conditional probability density function of X in class k and π_k is the prior probability of k [17]. The posterior probability is yielded by the Bayes theorem:

$$p(Y = k|X = x) = \frac{\pi_k f_k(x)}{\sum_{l=1}^{\nu} f_l(x)\pi_l} \tag{4-4}$$

Therefore, if we get an estimate of the density we can calculate the quantity $p(Y = k|X = x)$ and then assign X to the class with the highest posterior probability. Linear Discriminant Analysis (LDA) method fundamentally models each class density as a multivariate Gaussian:

$$f_k(x) = \frac{1}{(2\pi)^{p/2}|\mathbf{\Sigma}_k|^{1/2}} e^{-\frac{1}{2}(x-\mu_k)^T \mathbf{\Sigma_k}^{-1}(x-\mu_k)} \tag{4-5}$$

Additionally, LDA assumes that the covariance matrix is shared among the classes $\mathbf{\Sigma_k} = \mathbf{\Sigma}\ \forall k$. If we compare class l with class k, with respect to their log odds ratio, Equation 4-6, we get a relation that is linear in x.

$$\begin{aligned} \log \frac{p(Y = k|X = x)}{p(Y = l|X = x)} &= \log \frac{f_k(x)}{f_l(x)} + \log \frac{\pi_k}{\pi_l} \\ &= \log \frac{\pi_k}{\pi_l} - \frac{1}{2}(\mu_k + \mu_l)^T \mathbf{\Sigma}^{-1}(\mu_k - \mu_l) \\ &+ x^T \mathbf{\Sigma}^{-1}(\mu_k - \mu_l) \end{aligned} \tag{4-6}$$

In the equation above, the normalization factors $(2\pi)^{p/2}|\mathbf{\Sigma}_k|^{1/2}$ and the quadratic factors in the exponents are cancelled out due to the equality of the covariance matrices. From Equation 4-6, we conclude that the decision boundaries of the classes is linear in x for the LDA method and in n-dimensions it is a hyperplane. Practically, the Gaussian distribution parameters π_k, μ_k and $\mathbf{\Sigma}$are unknown and estimated via the training dataset.

Logistic Regression

Logistic regression is a widely used supervised Machine Learning (ML) algorithm for binary classification tasks, such as classifying diseased or not diseased tissue. It is based on the sigmoid function which is defined as in Equation 3-3 and Figure 3-5, which maps input values to a value between 0 and 1. The algorithm fundamentally assumes that the relationship between the input features and the probability of the positive class (e.g. diseased tissue) can be described by a linear equation and it estimates the coefficients of this linear equation.

Let's assume that the feature's vector is $X \in \mathbb{R}^s$, where s the total number of features and the target variable is Y, with $Y = 1$ corresponding to the positive class and $Y = 0$ corresponding to the negative class. Then the logistic model models the probability that the positive class occurs given a feature value $p(Y = 1|X_i)$. The assumption is that there exists a linear relationship between the features and the logarithm of the odds that $Y = 1$ or $Y = 0$ occurs. Next, we take the case of the class $Y = 1$ [17]:

$$\log \frac{p(Y=1|X_i)}{1-p(Y=1|X_i)} = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + ... + \beta_n X_n \tag{4-7}$$

with β the coefficients are on a logarithmic scale. The log of the odds is named the logit transformation and from Equation 4-7, the logistic probability function can be determined as:

$$\begin{aligned} p(Y=1|X_i) &= \frac{1}{1+\exp[-(\beta_0 + \beta_1 X_1 + ... + \beta_n X_n)]} \\ &= \frac{\exp[(\beta_0 + \beta_1 X_1 + ... + \beta_n X_n)]}{1+\exp[(\beta_0 + \beta_1 X_1 + ... + \beta_n X_n)]} \end{aligned} \tag{4-8}$$

The logistic regression model during training aims to learn the optimal values for the coefficients β in the linear equation by using the maximum likelihood. The log-likelihood for N observations is defined as:

$$L(\beta) = \sum_{i=1}^{N} \log p_{\beta_i}(X_i;\beta) \tag{4-9}$$

with $p_{\beta_i}(X_i;\beta) = p(Y=1|X_i)$. If Y_i corresponds to one of the classes output then $1-Y_i$ corresponds to the other and the log likelihood can be rewritten as:

$$\begin{aligned} L(\beta) &= \sum_{i=1}^{N} \{Y_i \log p(X_i;\beta) + (1-Y_i)\log(1-p(X_i;\beta))\} \\ &= \sum_{i=1}^{N} \left\{Y_i \beta^T X_i - \log(1+\exp[\beta^T X_i]\right\} \end{aligned} \tag{4-10}$$

with β the vector of the coefficients and X_i the vector of inputs containing the constant term 1. Hence, in order to maximize the log-likelihood, we equate its derivatives to zero:

$$\frac{\partial L(\beta)}{\partial \beta} = \sum_{i=1}^{N} X_i(Y_i - p(X_i,\beta) = 0 \tag{4-11}$$

the solution to these equations, that are non-linear in β is found using a numerical method such as the Newton-Raphston algorithm.

4-2-2 Sharpened Cosine Similarity (SCS)

Recently, Sharpened Cosine Similarity (SCS) has been proposed as an alternative to convolution for feature extraction [41]. Replacing convolutional layers with Sharpened Cosine Similarity layers is an innovative approach that introduces a novel perspective in the field of deep learning. SCS is a variant of Cosine Similarity (CS) that includes a sharpening factor, which enhances the influence of common elements in the compared vectors, for example.

According to [41], the mathematical description of Cosine Similarity (CS) between a weight kernel w and an input signal X is:

$$\mathrm{CS}(X, w) = \frac{X \cdot w}{||X|| \, ||w||} \tag{4-12}$$

It has been proven that cosine similarity can be enhanced/sharpened so that it distinguishes better whether two matrices are similar via applying a power p and maintaining the sign of the cosine value. The mathematical relationship between the weight kernel and the input signal that describes Sharpened Cosine Similarity (SCS) is [41]:

$$\mathrm{SCS}(X, w) = \mathrm{sign}(X \cdot w) \left| \frac{X \cdot w}{(||X|| + q)||w||} \right|^p \tag{4-13}$$

with q a small positive constant used to avoid numerical instability due to division by zero. In this thesis, we will explore the alternative of using Sharpened Cosine Similarity (SCS) instead of convolution and compare them in terms of performance.

As far as the implementation of this alternative operation is concerned, due to the lack of a formally developed library that incorporates it, we used the implementation from [2] with several adjustments to match our specific data structure.

4-3 CNN Model Interpretation

Deep learning applications can achieve high performances in several tasks. On the other hand, they also pose an important question to their users which is how to evaluate and trust the output of the trained model. This question is addressed through interpretability methods that allow us to understand how complex models work. Gaining insight in the model prediction process can be even more important than having a model with a high accuracy. In general, interpretability of models can be divided in two categories, transparency and post-hoc interpretation. A model can be called transparent when all of its parts and its learning process can be completely inspected and understood, for example, in linear models. On the other hand, post-hoc methods enable interpreting models after they have made their predictions and which are not transparent, such as convolutional neural networks which are viewed as "black-boxes".

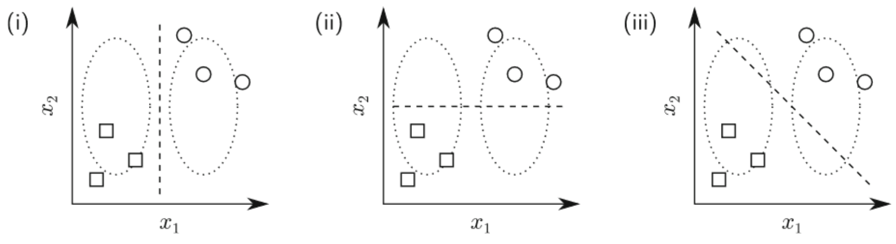


Figure 4-4: Three different learned boundaries leading to the same classification results [Image taken from Montavon et al.,Copyright ©2019 [31]]

In Figure 4-4, we can see how important it is to gain better understanding of how a model learns. It is obvious that in all three cases we get correct classification results, however, the correct data distribution is only learnt by the first model (i), whereas models (ii) and (iii) indeed predict correct classifications but without capturing the correct distribution. This is typical for high-dimensional data. Thus, we need interpretability methods in order to develop user trust, gain knowledge on how to make the model better by solving bugs, understand how the network layers interact with each other to result in the output and ultimately, obtain better applications.

A recently developed framework for model interpretability is SHapley Additive exPlanations (SHAP) [26]. SHAP method attributes importances to every feature based on the Shapley value. Lloyd Shapley (1923 - 2016) was an American mathematician and Nobel Prize-winner in economic sciences. He contributed especially in game theory and introduced the Shapley value. SHAP is a model agnostic methodology but it also introduces some optimizations especially for different types of models such as linear, trees and deep learning models. Since we are designing CNNs, we are going to use the deep learning SHAP explainer known as DeepSHAP.

4-3-1 The Shapley Value

The Shapley value [33], introduced by Lloyd Shapley, originates from cooperative game theory. If we have a game g_1 with cooperating players C, then the Shapley value $\phi(g_1)$ stands for the measure of each player's $i \in U$ contribution in the game with respect to all permutations on the set of players U. Due to the possibility of various contributions between player sets and outcomes, determining how to allocate individual importance can be ambiguous. The Shapley values provide one method of assigning importance, and more importantly they are uniquely defined in terms of satisfying fundamental and desirable criteria [26]. According to [33], the Shapley value $\phi_i(g_1)$ satisfies the following axioms. We define $\Pi(U)$ the set of player permutations and if $\pi \in \Pi(U)$:

- Axiom I. $\forall \pi \in \Pi(U)$ it holds $\phi_{\pi i}(\pi g_1) = \phi_i(g_1)$, $i \in U$
- Axiom II. $\sum_{i=1}^{C} \phi_i(g_1) = v(C)$
- Axiom III. For two games g_1, g_2 it holds $\phi(g_1 + w) = \phi(g_1) + \phi(g_2)$

The first axiom or else "symmetry" shows that the Shapley value is a property of the game v, the second axiom refers to "efficiency" and shows that the Shapley value distributes the total payoff of the game to every player, lastly, the thrid axiom or else "law of aggregation", shows that when two independent games v, w are combined then their values must be added.

4-3-2 Additive Explanations

While for a simple, transparent model the best interpretation is the model itself, for a complex model, such as CNNs, it is not possible to use the original model for interpretation. Therefore, a simpler, surrogate, explanation model should be used, which approximates the original one. This approach is not novel since it is also used in other existing methods such as LIME, DeepLIFT and Layer-Wise Relevance Propagation. SHAP is a unification of those methods.

Let's denote the original model as f, the explanation model as e and the input as X. The explanation model uses simplified inputs X' that are connected to the original inputs with a mapping function $X = h_X(X')$. Local methods satisfy $e(z') = f(h_X(z'))$ where $z' \approx X'$. The way the explanation model approximates the original model in additive feature attribution methods is depicted in Figure 4-5.

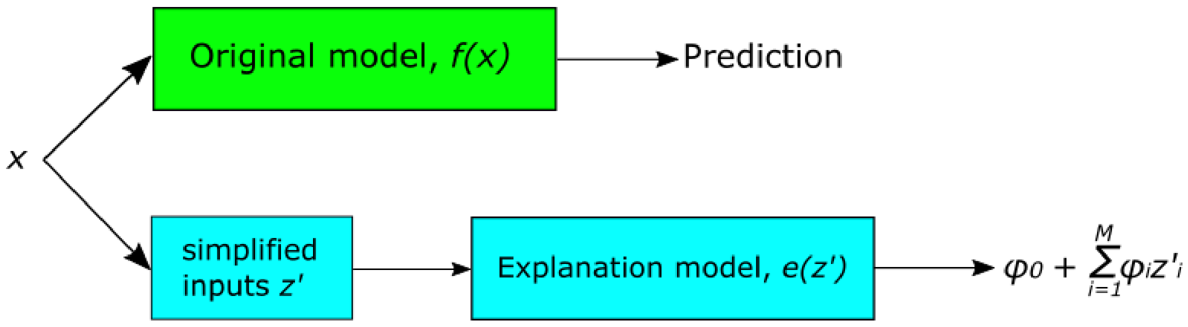


Figure 4-5: The form of the explanation model e of an original model f in additive feature attribution methods, where $z' \approx X'$ and $X = h_X(X')$

Therefore, in additive feature attribution methods we have a surrogate/explanation model that is locally a linear function of simplified inputs $z' \in \{0,1\}^M$, namely binary variables, with M the total number of simplified input features [26]:

$$e(z') = \phi_0 + \sum_{i=1}^{M} \phi_i z_i^{'} \tag{4-14}$$

All the methods that use Equation 4-14 as explanation models give an attribution $\phi_i \in \mathbb{R}$ to every feature and by adding all feature attributions results in the original model approximation. For SHAP this attribution ϕ_i is the Shapley value.

4-3-3 SHAP

The integration of the Shapley value from cooperative game theory into model interpretation is accomplished by treating each feature as a player, then the Shapley value is the measure of contribution of each feature to the final prediction. The introduction of the Shapley value as a measure of feature contribution is important because it is a unique solution as it follows from the game theory findings. The properties that make the attribution unique are similar to the ones presented for the Shapley value in game theory and are namely the property of local accuracy, missingness and consistency [26].

Local Accuracy The output of the original model f for a particular input X (locally) should align with the output of the explanation model e for the corresponding simplified input X':

$$f(X) = e(X') = \phi_0 + \sum_{i=1}^{M} \phi_i X'_i \tag{4-15}$$

where $\phi_0 = f(h_X(0))$ corresponds to all simplified inputs missing.

Missingness When features are missing from X, then it is required that those features have zero influence. All of the additive feature atribution methods follow this property.

$$X_i^{'} = 0 \Rightarrow \phi_i = 0 \tag{4-16}$$

Hence, missingness ensures that absent features have attribution equal to zero.

Consistency If $f_X(z') = f(h_X(z'))$ and $z'\backslash i$ denotes $z_i^{'} = 0$, then for two models f, $f^{'}$:

$$f'_X(z') - f'_X(z'\backslash i) \geq f_X(z') - f_X(z'\backslash i), \forall z' \in 0, 1^M \Rightarrow \phi_i(f', X) \geq \phi_i(f, X) \tag{4-17}$$

Combining the game theory findings it has been proven that there is a unique set of expressions that satisfy those three properties and this is the Shapley value. The Shapley values are calculated based on the following theorem [26]:

$$\phi_i = \sum_{z' \in \{0,1\}^M} \frac{|z'|!(M - |z'| - 1)!}{M!} [f_X(z') - f_X(z'\backslash i)] \tag{4-18}$$

Thus, in order to calculate the importance of an input feature X_i, which corresponds to the Shapley value in SHAP, we get all the permutations of feature subsets that do not contain $z'_i := z'\backslash i$, its simplified version, and then contain z'_i to all those subsets and calculate the effect on the output using their difference. The mean of those differences is the Shapley value of that feature. This can be computationally expensive and this is why SHAP has developed specific model type versions, so that it optimizes computational performance.

DeepSHAP

SHAP is a model agnostic interpretability method of complex models. DeepSHAP is based on the fundamentals of DeepLIFT and SHAP to boost the computational efficiency for deep learning models. Basically, DeepLIFT is adapted so that it approximates the SHAP values for deep models. DeepLIFT is an additive feature attribution method that assigns an attribute $C_{\Delta X_i \Delta Y}$ - with Y the model output - to every feature X_i. The simplified inputs are binary variables where 1 represents that X' takes its original value and 0 corresponds to a non-informative reference value.

Chapter 5

Methodology

This chapter will describe the use of an 1-D Convolutional Neural Network (CNN) for the Imaging Mass Spectrometry (IMS) data classification (section 5-1) and the motivation for using Tree-structured Parzen Estimator (TPE) methodology used to automatically select the architecture (section 5-2). Finally, the network architecture design implementation is presented for the final model selection per IMS data classification task.

Choosing the right architecture of a CNN for the task of IMS data classification is crucial for accomplishing high performance and accuracy. Classification of IMS data can be a challenging task due to its high dimensionality and complexity and the existence of noise within the data. CNNs have already been applied for this task demonstrating highly promising results. The selection of an appropriate architecture design methodology includes choosing between 2-D and 1-D CNN structures and deciding whether to tailor manually the network architecture or to use an automated architecture search.

5-1 1-D CNN Model

As previously discussed in section 1-2, the analysis of IMS data can benefit from the use of CNNs, since they share common characteristics with image data [4], for which CNNs were originally designed. IMS data can be viewed as a form of 1-D image data with the following analogy. An image is a 2-D matrix of pixels with intensities and a mass spectrum is an 1-D array of m/z values with intensities. Thus, a whole image can be seen as analogous to a mass spectrum. Each image-pixel is analogous to each m/z value of a mass spectrum and each image-pixel intensity can be viewed as analogous to each m/z peak intensity. These analogies are illustrated in Figure 5-1. Therefore, we select to apply an 1-D CNN model.

5-2 Selection of Architecture Design Approach

Regarding the design approaches of the CNN architecture there are two options, either design it manually or automatically. In general, neural networks and CNNs often keep on being de-

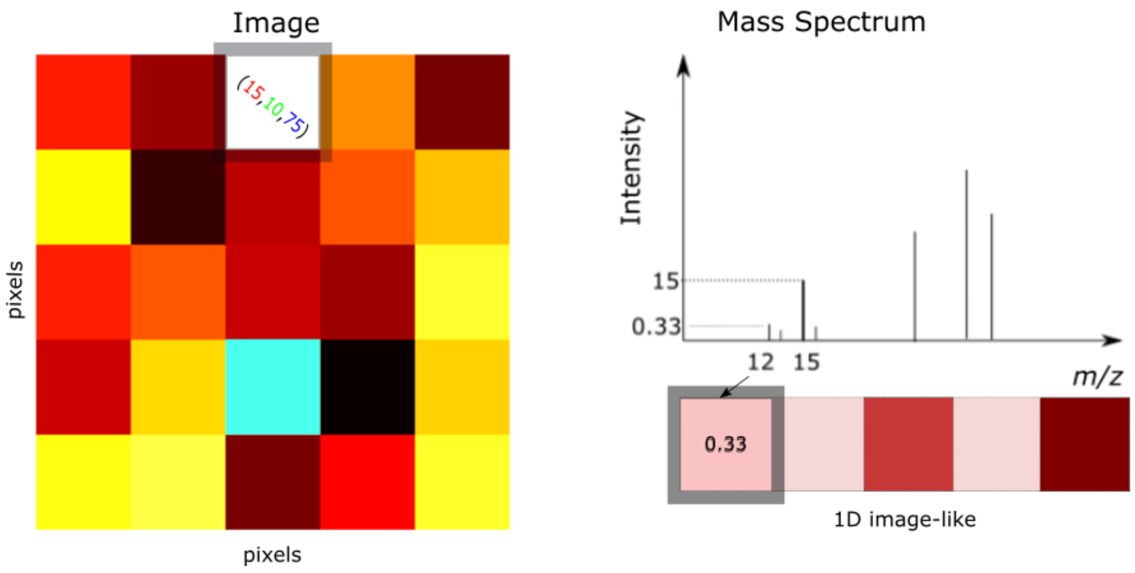


Figure 5-1: Image-like concept of a mass spectrum. A pixel with RGB intensities (e.g. [15,10,75]) is viewed analogous to an m/z value with 1-channel molecular intensity 0.33 and the whole mass spectrum can be viewed as a 1-D "image" with molecular intensities per m/z bin.

signed without much justification for every hyperparameter selection, which makes the design quite arbitrary and the hyperparameter selection is often not reported. Furthermore, designing a CNN architecture specifically for IMS data classification manually [4], with IMS data being high-dimensional, noisy, and characterized by complex spatial and spectral features, is very challenging. One needs to effectively capture these features. Thus, we focus on an efficient, systematic and automated methodology for designing the architecture of the 1-D CNN classifier for IMS. This can be accomplished by taking advantage of the fact that if one considers architectural design decisions as categorical hyperparameters then they can be optimized using conventional hyperparameter optimization techniques.

The indicators of an efficiently designed model extend along three directions based on [15]:

1. The ability of the model to represent and capture the underlying data patterns and structure, namely, the representational capacity
2. The ability of the model to find the lowest possible value of the cost function employed to train it
3. The generalization ability of the model

The optimal capacity of a well designed network can be illustrated as follows in Figure 5-2. A high representational capacity doesn't guarantee a high performance since it may lead to the issue of overfitting. Additionally, the ability to achieve low cost on the training data also runs the same risk. Finally, the extent to which regularization techniques are implemented can greatly influence the performance of the model. If the regularization is too weak, the model may overfit to the training data, while if it is too severe, the model may underfit. Therefore,

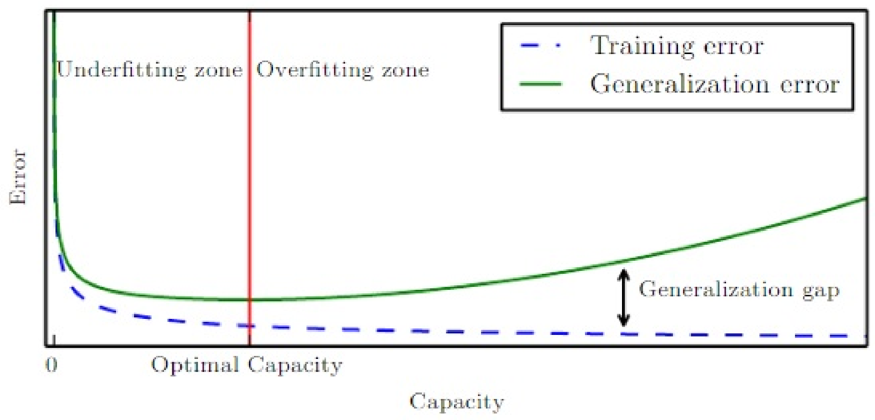


Figure 5-2: Generalization and training error cases. On the left, where both the training and generalization-testing error is high we have the underfitting region. On the right, where the generalization error has a gap from the training one, we enter the overfitting zone. [Image taken from Goodfellow et al., Copyright ©2016, [15]]

finding the right balance between model complexity and generalization is vital for achieving optimal performance.

The appropriate balance between model complexity and generalization can be accomplished via the optimal selection of the model's architecture/hyperparameters, which can be designed automatically by using conventional hyperparameter optimization techniques. Hyperparameter optimization can be automated by traditional methods such as Grid Search or Random search or more complex methods that are more efficient. For instance, in image classification, automated hyperparameter optimization with the Bayesian TPE algorithm has been proven more efficient than manual or random search processes [7], [8], trials on a grid are proven more effective in search spaces of smaller dimensions [6], while, models such as CNNs can have dozens of hyperparameters whose tuning can have a major impact on the model's peroformance. Furthermore, the TPE algorithm has been distinguished in competitions and has achieved state of the art performance in Deep Learning (DL), according to [39], which won the AutoML 2022 competition on “Multiobjective Hyperparameter Optimization for Transformers”. Thus, TPE is made an attractive approach.

An automated methodology for designing the CNN architecture for IMS data classification is viewed beneficial because of the following reasons. The performance of a CNN model massively depends on its hyperparameters and a systematic methodology can contribute to exploring a larger hyperparameter search space. Additionally, it can aleviate the time consumption and effort needed to design manually an effective CNN architecture as well as reduce error prone human tactics [19]. Furthermore, the design of a CNN depends heavily on the input data, the classification task and the available computational resources. Consequently, following an automated process can help in designing a model architecture that is effectively capturing the important features for the specific classification task. This, makes the design of the model effective without the need of incorporating specialized domain knowledge to make design choices and enhances its adaptability to different tasks and input data. However, there is a need for large amounts of computational resources in order to implement an automated design approach, especially when the hyperparameter search space is too broad.

5-2-1 Comparing TPE to Grid Search and Random Search

Automated network architecture search can be accomplished with various methodologies. In this thesis, we investigate the suitability and eventually, choose among two traditional methods, Grid Search and Random Search and one more sophisticated method, the Bayesian TPE algorithm. For the purpose of comparing the efficiency of the hyperparameter tuning methods, the rat kidney dataset is used for the task of inner medulla classification section 6-2.

Rejecting Grid Search The search space used for our trials is comprised of numerous hyperparameters in broad ranges (Table 5-2). The number of hyperparameter combinations required for Grid Search would be overwhelmingly large, making it impractical to execute with the available computational resources. The total amount of trials for Grid Search is calculated by multiplying the amount of options for each hyperparameter. Hence, there are almost 54 million combinations for Grid Search to explore and it is easily understood that it is impractical to implement such a method for hyperparameter tuning. Therefore, the Grid Search approach is considered infeasible for exploring our hyperparameter space in practice and no results are going to be compared about it.

Exploring optimal architectures with TPE and Random Search After disapproving of the vanilla Grid Search approach, we compare the efficiency of the rest of our options, TPE and Random Search, for finding the best performing hyperparameters. The comparison is based on a fixed number of trials, namely, 100. Then, we reduce them to 50 and then, to 20. In that way, we compare in how many trials each method is able to find a good performing hyperparameter set. Each trial requires roughly one and a half minute to complete. Additionally, the runs refer to three different random seeds, in order to check the method's robustness.

Table 5-1: Comparison between TPE and Random Search

	Trials	**Best performance (F1 score)**	**Mean (across trials)**
TPE	20	0.9967	0.77
	50	1.0	0.46
	100	0.9997	0.65
Random Search	20	0.9997	0.1530
	50	0.9993	0.16
	100	0.9989	0.17

The results for each number of trials ran by each methodology are reported in Table 5-1. It is observed that both methodologies achieve to find configurations that give relatively high performances. Nevertheless, the highest f1-score among all number of trials is achieved by TPE that reaches the perfect score of 1.0 in 50 trials, while for Random Search, the highest is 0.9997 found within 20 trials. On the contrary, TPE doesn't find a better hyperparameter set within 20 trials because of the amount of initial trials we have set it to search, namely 10. On the other hand, TPE achieves better results than Random Search in the other two cases which perform more trials. Furthermore, in Table 5-1, we report the mean values of the non-pruned trial performances and we see that Random Search has a very low mean value across its trials. This is further illustrated in the following figures.

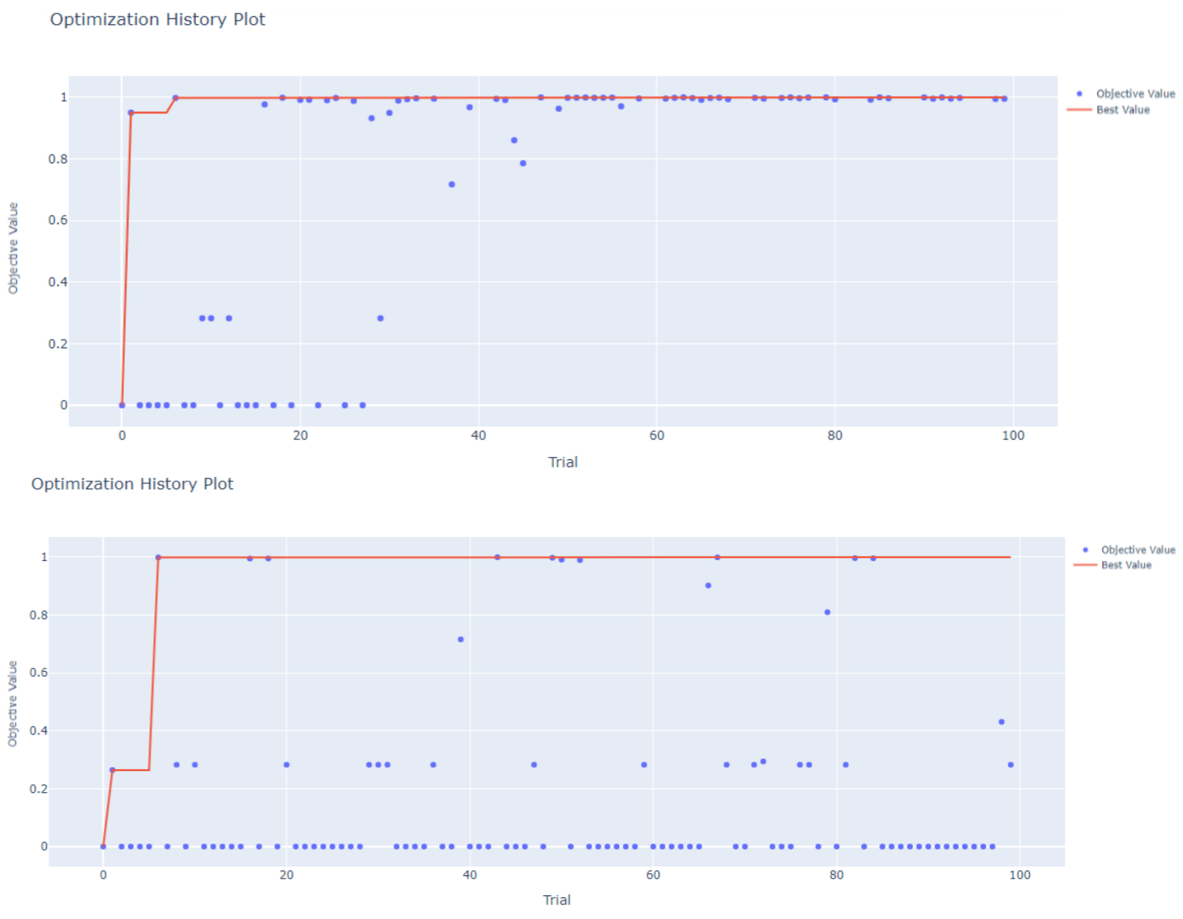


Figure 5-3: Performances along 100 trials of hyperparameter configurations with TPE (top figure) and Random Search (bottom figure). The red line demonstrates the best performance achieved over all trials.

In Figure 5-3, we observe the hyperparameter performances achieved by each trial - not pruned - by TPE and Random Search. The red line indicates the best performance among all trials. As expected, the TPE search, after the initial random samples are collected (10), it starts to converge to a really high performance for all the next hyperparameter trial sets, whereas the Random Search randomly peaks hyperparameter configurations from the search space and those achieve high performances occasionally. For example, around trials 18 and 50 the peacked sets are successful with high performances while most of the rest random hyperparameter sets achieve 0.0 F1 score. Therefore, we conclude that Random Search certainly, doesn't find the optimal solution but rather a randomly selected hyperparameter configuration performing quite well.

Conclusion In order to select one of the two methods, the traditional Random Search versus the sophisticated TPE - TPE uses two sets of probability density functions, one to model the distribution of hyperparameters that have performed well in the past, and one to model the distribution of hyperparameters that have not performed well and focuses on the well performing ones - we compare their efficiency based on several criteria.

- The model resulting from TPE exploration of the search space reaches a lower classification error and higher F1 score than Random Search (Table 5-1), even though it is not a significant difference.
- Random search generates configurations from the search space at random while TPE bases its future configurations on past observations according to a probabilistic model.
- TPE configurations converge faster since it focuses the search on promising hyperparameter regions compared to Random Search, which can take longer to converge and find near-optimal configurations. Due to that, the mean performance across the trials by TPE is much higher than Random search (see Table 5-1, Figure 5-3).
- Random search is simple to implement whereas TPE is more sophisticated but can be more efficient for high-dimensional search spaces.

To sum up, Random search is a simpler method to explore hyperparameter search spaces, however it may need more time and more trials to converge to a configuration that might or not be optimal. On the other hand, TPE models the probability of performance improvement and explores configurations based on the past observations. Therefore, we select the TPE algorithm as the automated architecture search method, since it is demonstrating more efficient characteristics and it adapts its search based on a probabilistic model which enables it to converge faster, in less trials and more efficiently.

Since we have selected the architecture search algorithm the design process can begin. In this section, we are going to dive into the pipeline of the implementation of the chosen automated architecture design methodology. The methodology starts with defining a search space, the set of possible network architectures that the algorithm can explore. Then a performance metric is selected to evaluate the performance of the architecture on a specific task and next, the TPE algorithm is used to explore iteratively the search space and optimize the network architecture based on that performance metric. We run the algorithm for several random seeds to check its robustness and finally, we evaluate the hyperparameter configurations on a validation set to do model selection. To sum up, in this section we will explore the implementation pipeline of the automated architecture search methodology, namely the TPE algorithm, for designing 1-D CNN architectures for the classification of the given IMS data.

5-3 Hyperparameter Search Space Design

Here, we will talk about the decisions we made regarding all relative architecture hyperparameters. To start with, the choices that someone comes across while designing a CNN model include the following hyperparameters:

- Total number of convolutional layers
- Number of filters in the hidden layers
- Kernel size
- Stride offset

- Pooling kernel size
- Pooling stride offset
- Batch size
- Learning rate
- Learning rate policy
- Number of Fully Connected (FC) layers
- Number of nodes in the FC layer
- Type of optimizer
- Batch normalization layer
- Type of pooling layers
- Dropout layer
- Layer activation function
- Weight initialization method

These hyperparameters are quite a few in number which would be really time consuming and challenging to be set manually by a designer in an efficient way. Thus, we create a search space to explore some of those, while some of the design choices are made based on the state of the art and empirical approaches in CNN architectures. More specifically, the hyperparameters which are set based on state of the art approaches and experience are the following: the layer activation type, the type of the optimizer and the use of batch normalization layers. The design choices for those can be motivated as follows:

- **Layer Activation Function:** The Rectified Linear Unit (ReLU) acivation function is applied after each convolutional and FC layer. In [24], the researchers use ReLUs to achieve better computational efficiency and generally, it is a widely used activation function. Additionally, ReLU activation is useful in alleviating the vanishing gradient problem occuring in DL, since it has a constant gradient for positive values, which helps prevent the gradients from becoming too small.
- **Optimizer:** In [42], Adaptive Moments (Adam) is reported as the state of the art optimizer for deep learning. Additionally, Adam combines the benefits of RMSprop and momentum [32], [22], which allows it to outperform them. Hence, we choose Adam as our optimizer type.
- **Batch Normalization Layer:** According to [20], batch normalization contributes to enabling the use of larger learning rates and reduces dependency on weight initialization. In [18], they use batch normalization exactly after each convolutional layer and before the activation. So, we do use batch normalization in the exact same manner and test the scenario of using it vs. not using it. The conclusion is that the addition of the batch normalization layer improved significantly the model's performance, namely it showed up to 40% accuracy increase.

- **Weight Initialization:** There are several weight initialization approaches implemented in contemporary CNN architectures. We use the one specifically designed for ReLUs as in [18].

For this purpose, we create a hyperparameter search space as the one provided in Table 5-2. When defining the search space general guidelines are taken into account [5], [15]. First of all, the fact that the range should be wide enough in order to discover better hyperparameters. Secondly, the availability of computational resources - a very wide search space is not manageable. Lastly, a satisfactory diversity of the values for a better exploration. Additionally, image classifier recommendations [25] about the fundamental components of a CNN are considered. Therefore, we define the following hyperparameter configuration space.

- **Kernel Size:** We preserve the range of the kernel size in a small area because smaller kernel sizes enable the detection of more local features. For example, some image classifiers [34] based on CNNs use kernel sizes of 3×3, which is the smallest possible size of a kernel for detecting information about up-down and left-right. In a similar manner, we define 3 as the minimum kernel value so that it can capture left-right information around at least one peak.

- **Stride offsets:** Similarly, the stride offsets should be contained in small values so that valuable information from the input is not missed. However, a larger stride size can be useful for reducing the computational cost. In [34] a convolutional stride of 1 and pooling stride of 2 is used and in [18] an offset of 2 both for convolutional and pooling layers, whereas in [24], the convolutional stride offset is 4 and the pooling is 2. Based on the existing architectures and on the notion that we should keep small enough stride offsets we explore their values from 1 to 4.

- **Pooling Kernel Size:** Pooling is used to downsample the feature maps produced by the convolutional layer, reducing the spatial dimensions while retaining important features. A large pooling kernel size can reduce the computational cost by greater downsampling. However, it might lead to the network failing to learn important details. In [34], the used size is 2 and in [24],a pooling kernel of size 3 is used and so we also explore values with maximum size for the pooling kernel as 5 and minimum size of 2.

- **Type of Pooling Layers:** We explore the usage of two basic categories of pooling namely, average and mean pooling layers.

- **Batch Size:** Theoretically, this hyperparameter should have an impact on the training time duration rather than the performance. We set the batch size as a power of 2 [42]. The model can run faster in that way because of the architecture of the Central Processing Unit (CPU)/Graphics Processing Unit (GPU) memory. In general, a default value of 32 is considered good and is a common practice [5]. Thus, we explore its value up to a few hundreds in a power of 2 manner, starting from 32.

- **Learning Rate:** The learning rate is one of the most crucial hyperparameters to tune for the Adam optimizer, since it controls the step size taken by the optimizer during each update of the weights. The learning rate should be chosen carefully because if it reduces severely, convergence is hindered, whereas if it's reduced leniently, the best solution cannot be found. Common search space values lie in the range of $[10^{-6}, 1]$ [5]. An alternative to having a fixed learning rate is its adaptation during training.

- **Learning Rate Policy:** The learning rate policy concerns its adjustment based on the training performance. There are several policies which can be followed. We explore the constantly fixed learning rate, the exponential decay, the cyclic scheduler and the step decay. In addition, log scales are suggested in [42].
- **Number of Layers:** The search space for the total number of convolutional layers is set from 1 to 5 and for the stacked convolutional layers is set from 0 to 5, via trial and error. The first choice is tuned based also on trial and error by checking the achieved performance and enlarging the model capacity until the balance between performance and generalization is disturbed. In this way, the maximum depth of 5 convolutional layers is explored. This procedure needs to be repeated for each dataset and each classification task. Furthermore, the amount of FC layers is explored based on existing architectures that use more than one FC layers, in [34], [24] they use 3, thus we explore up to this number each time by reducing the amount of nodes of the initial FC layer by two.
- **Number of Filters per Hidden Layer:** In a single feature map, all its units are convolved with the same filter/kernel values so that the same feature is detected on all possible regions of the input data [25]. In another feature map of the same hidden layer, another set of filter values is utilized so that a different kinds of local feature is learned. Hence, on every input position, different kinds of local features are learned, as many as the number of filters used. Thus, we want to explore the amount of representations/features present in the input data. This is based on the type of the dataset we are dealing with each time. Since we haven't applied any feature extraction technique, we try to set the range of possible values within hundreds and let the algorithm decide upon the most efficient value. We tune the number of filters of the first hidden layer and then double them for each next layer.
- **Number of Nodes in FC Layer:** Selecting too few nodes might result in underfitting, while [5] suggests that it is better to use a large enough number of nodes in the FC layer instead of less than the optimal number, since regularization techniques can aleviate the case of overfitting. However, this increases the number of trainable parameters and the computational cost. Nonetheless, we explore using hundreds to thousands of nodes.

The dropout layers were tuned manually based on trial and error and we set zero padding so that the output size remains the same. After having described the most important hyperparameter range definition, the resulting hyperparameter search space is explored by the TPE algorithm. In this section, we will not talk about the number of trials used or other tuning parameters of the algorithm. We set out to explore the search space and report hyperparameter importances.

In Figure 5-4, we observe the hyperparameter importances, where the highest importance is significantly attributed to the number of convolutional layers in the first convolutional stack. On the other hand, the type of pooling method takes up a very small importance thus, we decide to exclude it from our hyperparameter search space. We use the max pooling which is mainly applied in image classifiers. Nevertheless, more than just the pooling type are given

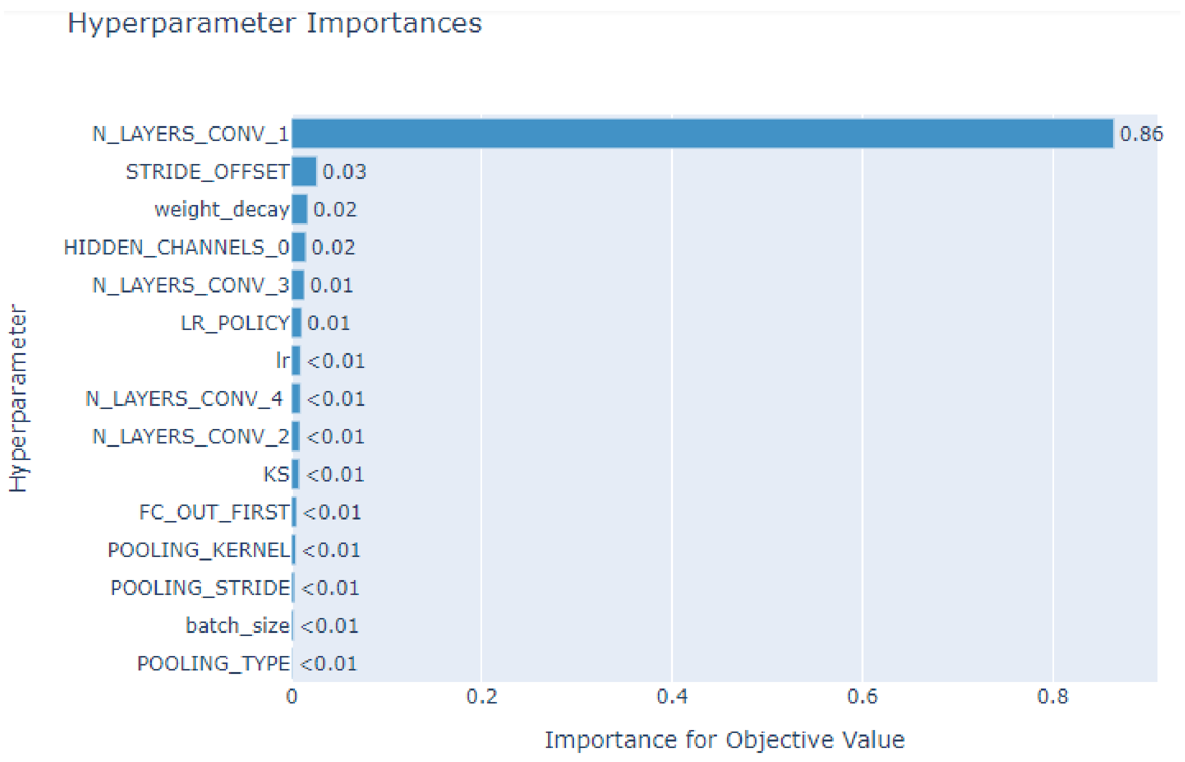


Figure 5-4: Hyperparameter Importances

minor importances however, comparing the hyperparameter importances among different runs it seems that the pooling type remains as the least influential parameter while the sorting of the others is partly redistributed. A different run of hyperparameter importance results can be viewed in section A-1.

Therefore, the finalized version of the search space we design for hyperparameter exploration is depicted in Table 5-2.

5-4 Automated Architecture Search by TPE

Once the hyperparameter configuration space is designed, we can start the exploration of the optimal CNN architecture using the TPE algorithm [8]. For the implementation, the tools we use are Google Colab, Optuna, an open-source framework for efficient automated hyper-parameter optimization (see Appendix) and PyTorch within Python.

First of all, TPE requires initialization. During the initial trials, Optuna randomly samples the hyperparameters to explore the search space before switching to TPE. The initial trials constitute a hyperparameter in Optuna which is called: *n_initial_trials*. In order to set the proper amount of initial trials we make a grid of possible values and choose the most efficient one. This means that we try to keep the initial trials at a low level so that we can reach optimal architecture outputs within 100 trials. We test with Grid Search among [10, 20, 30] initial trials and choose the one that achieves the best performance. Furthermore, the

Table 5-2: Hyperparameters Search Space

#	**Hyperparameter**	**Type**	**Range**	**Conditional**
1	no. of Convolutional layers	integer	[1, 5]	-
2	no. of FC layers	integer	[1, 3]	-
3	no. of Filters for 1st hidden layer	integer	[5, 60] (step=5)	-
4	no. of Nodes for FC layer	integer	[100, 1000] (step=100)	-
5	Learning rate policy	categorical	['Fixed', 'Step', 'Exp','Cyclic']	-
6	Exp. decay rate	float	[0.0, 1.0]	+
7	γ	float	[0.1, 0.9]	+
8	Step size	integer	[1, 10]	+
9	Learning rate	float	[1e-5, 1e-1] (log-scale)	-
10	Batch size	integer	[32, 64, 128, 256]	-
11	Kernel size	integer	[3,20]	-
12	Stride offset	integer	[1, 4]	-
13	Pooling kernel	integer	[2, 5]	-
14	Pooling stride	integer	[1, 4]	-
15	L2 regularization	float	[1e-7, 1e-2]	-

process that is followed when searching for the optimal CNN architecture, is the following. For each IMS dataset, we run the TPE algorithm on the specified search space, for three different random seeds and compare the resulting optimal hyperparameter configurations, as described in the next step, in order to evaluate and select the best architecture for each classification task.

During the iterative process of updating the hyperparameter configurations and their performance, the evaluation metric that is used is the F1 score. The hyperparameters are chosen so that this metric gets maximized. This metric is chosen because it is a good trade-off between precision and recall.

Some of our IMS datasets are imbalanced. Accuracy alone can be misleading as it can be biased by the majority class. Therefore, F1 score is a good metric when we have imbalanced classes since it is a combination of precision and recall which are important metrics when it comes to imbalanced datasets.

Additionally, not all of the trials are given equal chances to be trained. Hence, a pruning algorithm is used to judge whether a trial has a low chance of improving the performance based on already reported values and it should be pruned. The pruning algorithm used is the Median pruner which uses the median stopping rule and works as follows. It computes the median of the observed trial scores and prunes if the trial's best intermediate result is worse than the computed median of intermediate results of previous trials at the same step (same number of iterations or epochs).

In more detail, during each hyperparameter optimization trial a sequence of iterations are conducted where the intermediate F1 scores are recorded. When the trial reaches a certain amount of iterations, the pruning algorithm compares the median of all the previous trials at the same step with the best intermediate result of the current trial. If it is better than the current the trial is stopped prematurely (pruned), otherwise it continues. This process is repeated until the maximum number of iterations or epochs are reached. This method is suitable for most optimization problems and can reduce the optimization time significantly. Moreover, it has some hyperparameters to be set, namely:

- *n_startup_trials*: number of trials at the beggining of the optimization process without any pruning. Its default value is set to 5.
- *n_warmup_steps*: number of intermediate results(steps) before pruning starts. Its default value is set to 0.
- *interval_steps*: frequency at which the intermediate results are checked by the pruner. Its default value is set to 1.

We tune those values as follows. We don't change the default for the number of initial trials that pruning is disabled, we increase the number of the warm-up steps to 30, in order to allow the generation of sufficient intermediate results, and we make the pruner check every 10 interval steps.

5-5 Architecture Selection

Model selection is an important factor when we want to build an efficient machine learning model. As already mentioned, we will run the search space exploration for three different random seeds. The next goal is to select the best one out of the resulting configurations.

In order to achieve that, we use *k*-fold cross validation. The dataset is split into training and testing sets. We further split the training set into *k* folds and train the model on *k-1* folds and use the remaining fold as a validation set. This process will be repeated until all of the folds have been used as a validation set. Then, we are using the mean F1 score across all folds to select the best hyperparameter configuration. This concludes the final selection of the optimal architecture. The testing data is then used to evaluate the performance of the final model with the selected hyperparameters.

Chapter 6

Experimental Results

In this chapter, we present the classification outputs of the designed Convolutional Neural Network (CNN) models in comparison with linear Machine Learning (ML) models and with a Sharpened Cosine Similarity (SCS)-NN model applied on three different Imaging Mass Spectrometry (IMS) datasets, acquired from a rat brain sample in section 6-1, a rat kidney sample in section 6-2 and a mouse brain sample of the region of the hippocampus in section 6-3.

6-1 Dataset no.1: Rat brain

The first case study is based on a dataset obtained from a coronal section of the brain from a rat [37].

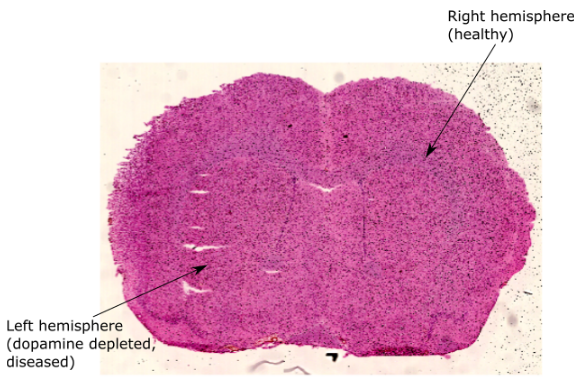


Figure 6-1: Microscopy image of the tissue sample from the brain of a rat

The Matrix-Assisted Laser Desorption Ionization (MALDI) IMS data, with a 2,5 - dihydroxyacetphenone matrix applied by a TM Sprayer (HTX Technologies, Carrboro, NC, USA), has been obtained using a 15 T Fourier Transform Ion Cyclotron Resonance (FTICR) mass spectrometer (Bruker Daltonics, Billerica, MA, USA), with a pixel size of $75\mu m$ and mass resolution of 50,000 m/Full Width at Half Maximum at m/z 5000. Furtheremore, the data have been normalized according to TIC (Total Ion Count) and peak picked resulting in their final form.

This sample has been specially treated to model Parkinson's disease. More specifically, the nigrostriatal dopaminergic neurons were artificially destroyed, since their disfunction is resembling Parkinson's disease, which has symptoms such as slowness of movement or tremors. Moreover, the left brain hemisphere is dopamine depleted and thus, diseased, whereas the right brain hemisphere is left intact and thus, healthy, (see Figure 6-1).

1. Left hemisphere (diseased - positive class - 1)
2. Right hemisphere (healthy -negative class - 0)

Therefore, in this case study, we have a binary classification problem with the objective to differentiate among the diseased (left hemisphere) and the healthy (right hemisphere) class.

6-1-1 The Data

The size of the IMS data is comprised of totally 17,964 pixels while the detected ions lie within a range from m/z 1300 to 23,000 and the total number of the ion images - alternatively peaks - used is 6638.

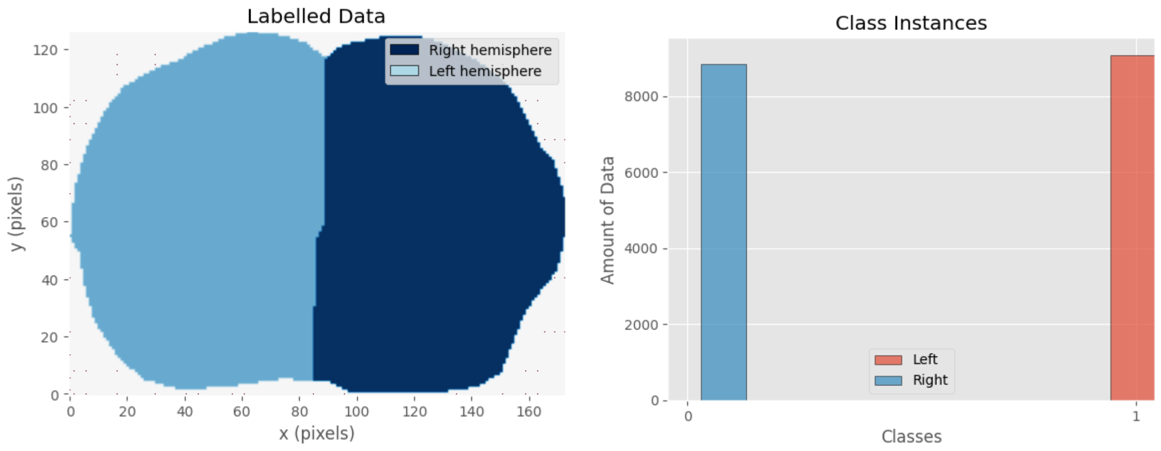


Figure 6-2: (Left - image) Pixelwise annotation of the rat brain, the left hemisphere is dopamine depleted and it is the positive class (1), while the right one is regular and it is the negative class (0) - (Right - image) Instances distribution to the two rat brain classes.

The annotated masks of the rat brain dataset are seen in Figure 6-2 (Left-image). Additionally, we observe that the classes are nicely balanced, we have 9083 instances in the "Left" class and 8855 instances in the "Right" class, which makes the analysis of the results based on accuracy as a performance metric trustworthy. Moreover, we use a train and test split of 80% and 20% respectively and 5-fold cross validation for the final model selection.

Preprocessing

Applying preprocessing to the data can have a large impact on the classification performance. For this dataset, we apply centering and outlier removal.

In general, statistical normalization of the data is recommended in ML, so that the features are ensured to belong to a similar scale, which can improve the performance. Min-max scaling, which scales the feature values to a fixed range, for a number of observations of a single feature X, is defined as:

$$X_{\text{scaled}} = \frac{X - \min(X)}{\max(X) - \min(X)} \tag{6-1}$$

In our case, the large discrepancies between feature intensities contain valuable biological information and normalization can reduce or remove this information by scaling all features to a similar range. Hence, the scenario of normalizing the data is dropped. However, data centering has been applied to our dataset. The mean intensity value for each feature column is calculated, and this value is subtracted from all of the intensity values in that column. This process results in a new dataset where the mean intensity value for each column is zero, while preserving the relative differences in intensity values which are valuable.

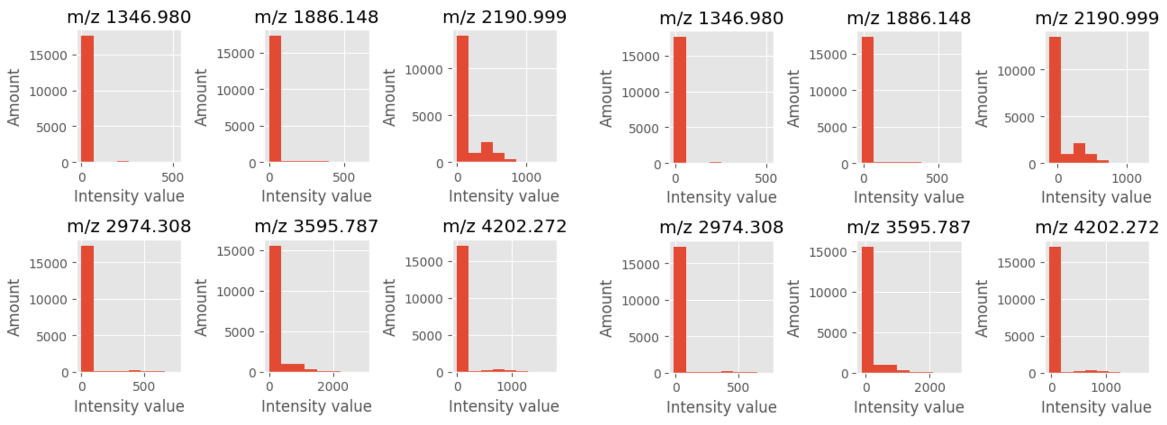


Figure 6-3: (Left - image) Examples of spectral data distribution - (Right - image) Results after centering the data

In Figure 6-3, we observe the distribution of the data for several m/z features. We can see that their distribution is right skewed. As we have already mentioned, we don't apply any normalization, we only center the data around zero. The effect of data centering is visible on the right image of Figure 6-3.

Additionally, as a preprocessing step of the raw dataset we remove its outliers by implementing an upper boundary using standard deviation. Outliers are data points which are significantly deviating from the majority of the rest data points and can have a negative impact on the classification performance if left untreated. In our case, we assume that any raw intensity value of each column and each row of the dataset that exceeds a threshold of 10 standard deviations is an outlier and it is substituted by the threshold value. Consequently, the data is now prepared to implement different classifiers.

6-1-2 Classification with Linear Models

In this section, we present the outcome of the application of two different linear models, namely Linear Discriminant Analysis (LDA) and Logistic regression. These two linear models are powerful tools for predicting categorical outcomes given input feature and target sets. In this context, we explore their suitability and evaluate their performance on the current IMS data classification task.

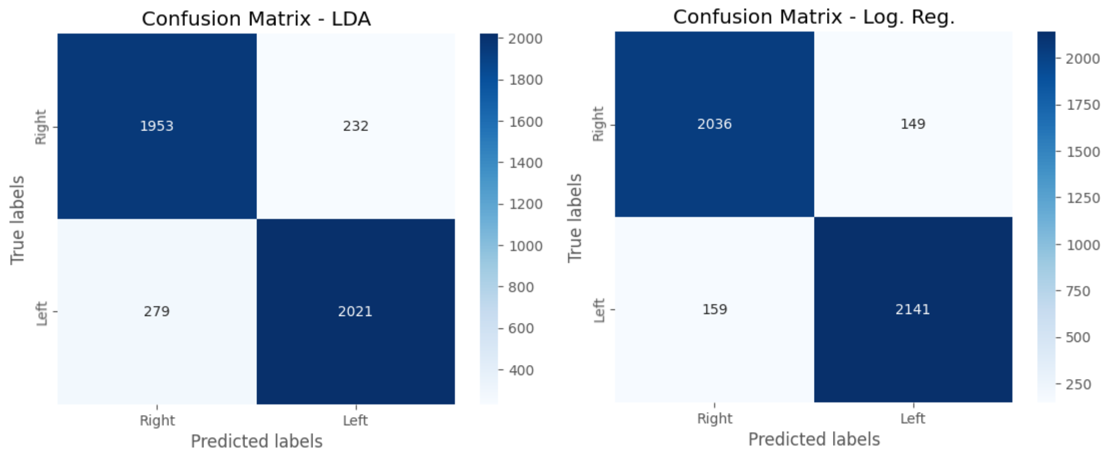


Figure 6-4: (Left - image) Confusion matrix for LDA model - (Right - image) Confusion matrix for Logistic Regression model for the Parkinson's disease cassification

From the confusion matrices in Figure 6-4, we can calculate the accuracy, which measures the overall correctness of the models, as:

$$\text{Accuracy} = \frac{(TP+TN)}{(TP+TN+FP+FN)} = \begin{cases} \frac{(2021+1953)}{(2021+1953+232+279)} = 0.8860 & \text{for LDA} \\ \frac{(2141+2036)}{(2036+2141+149+159)} = 0.9313 & \text{for Log. Reg.} \end{cases} \tag{6-2}$$

Hence, for the LDA model the accuracy is 88.6%, while for the logistic regression around 93%. This suggests that the logistic regression model is able to capture the underlying data patterns more efficiently in the given classification task, leading to higher prediction accuracy. Furthermore, we can calculate the the proportion of correctly predicted positive cases out of all predicted positive cases, which is the Precision metric and the proportion of correctly predicted positive cases out of all actual positive cases which is the Recall.

$$\text{Precision} = \frac{TP}{(TP+FP)} = \begin{cases} \frac{2021}{(2021+232)} = 0.8970 & \text{for LDA} \\ \frac{2141}{(2141+149)} = 0.9349 & \text{for Log. Reg.} \end{cases}$$

$$\tag{6-3}$$

$$\text{Recall} = \frac{TP}{(TP+FN)} = \begin{cases} \frac{2021}{(2021+279)} = 0.8786 & \text{for LDA} \\ \frac{2141}{(2141+159)} = 0.9308 & \text{for Log. Reg.} \end{cases}$$

In conclusion, logistic regression outperforms LDA overall in accuracy, precision, which suggests that it has a better ability to identify correctly positive (diseased) samples, and recall, which suggests that it captures a larger proportion of the true positive samples out of all the true positive samples.

6-1-3 Classification with a CNN

After following the methodology as described in chapter 5, the resulting overall architecture designed for this classification task is the one depicted in Table 6-1.

Table 6-1: Designed CNN architecture for Parkinson's disease classification (left vs. right rat brain hemisphere).

Hyperparameters	**TPE Architecture**
Kernel Size	9
Stride	1
Pooling Kernel	3
Pooling Stride	5
Total Conv. layers	3
Stacked Conv. layers, 1	1
Stacked Conv. layers, 2	1
Stacked Conv. layers, 3	1
Stacked Conv. layers, 4	0
no. of Filters for 1^{st} Hidden layer	60
no. of Fully Connected (FC) layers	1
no. of Nodes for FC layer	800
Learning Rate (LR) Policy	'Step'
Exp. decay rate	non-applicable
γ	0.4
step size	6
Initial LR	0.00226
Batch Size	32
L2 Regularization	0.021

As depicted in Table 6-1, the network is comprised of three convolutional layers with kernel size 9 and stride offset equal to 1, and one fully connected layer with 800 neurons, values derived from the Tree-structured Parzen Estimator (TPE) hyperparameter optimization. The initial number of feature maps/filters is 60. Additionally, we use dropout layers in order to reduce overfitting, without combining it with L2 normalization, since we observed that the performance was influenced negatively when both regularization methods were tried. Moreover, we use batch normalization layers at the end of each convolutional layer and average pooling, with kernel size 3 and stride equal to 5, to downsample the data.

Based on Figure 6-5, the accuracy of the CNN model after 100 training epochs on this task reaches 96% for the test split.

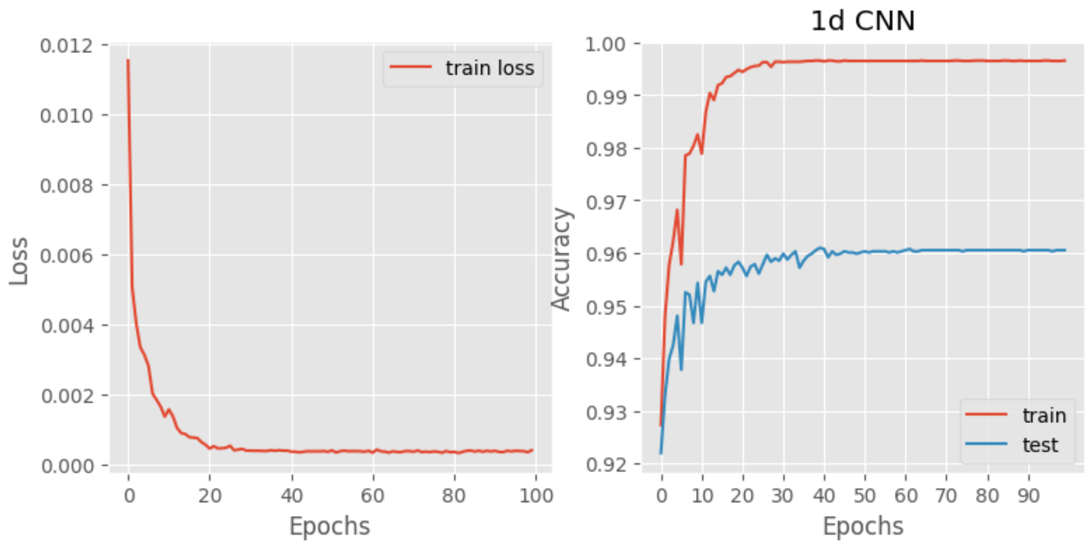


Figure 6-5: Performance curves for training and testing of the CNN model

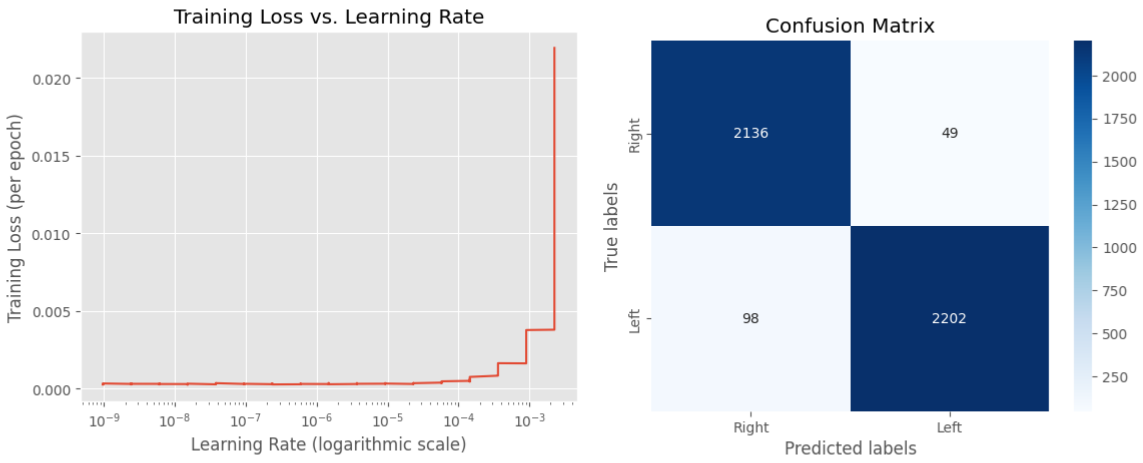


Figure 6-6: (Left - image) Relationship between learning rate and training error - (Right - image) Confusion matrix for CNN model

In Figure 6-6, we see that the learning rate which is being adjusted at specific steps (6, from Table 6-1), leads to a training loss close to zero when it reaches values less or equal to 10^{-5}, which suggests that below that optimal value for the LR the model converges more efficiently and helps its ability to find an optimal solution. On the contrary, for learning rate values that are larger than 10^{-3} the training loss increases sharply.

Additionally, on the right image of Figure 6-6, we see the corresponding confusion matrix, from which we can calculate the accuracy, the precision and recall as previously following Equation 6-2, Equation 6-3. Therefore, we calculate that the accuracy on the testing set is

0.9672 as already presented, while the precision and recall are equal to 0.9782 and 0.9573.

6-1-4 Classification with a SCS-NN

If we replace the convolutional layers with SCS layers (see Figure 6-7), we can achieve a similar performance with a simpler architecture since we don't use any batch normalization Batch Normalization (BN) layers nor Rectified Linear Unit (ReLU) activation layers and the trainable parameters are reduced in amount. In this thesis, we call the resulting model SCS-NN in contrast with the CNN. The SCS layer is implemented based on code from [2] and is not part of an imported Python library.

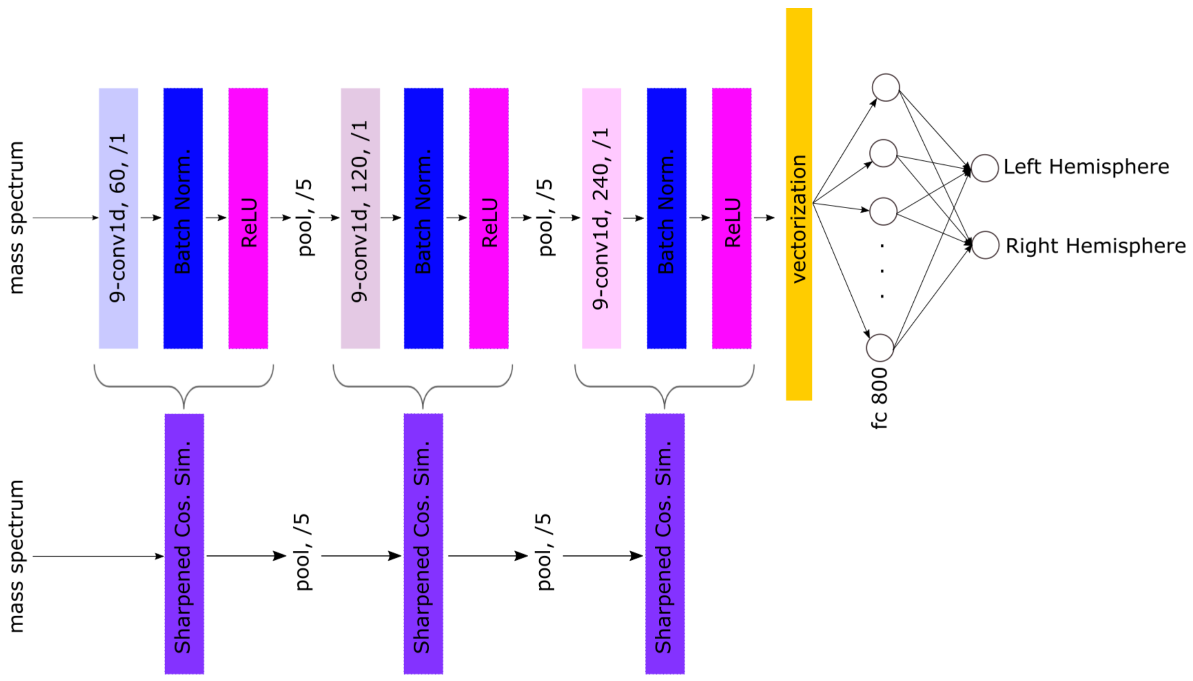


Figure 6-7: Replacement of CNN related layers convolutional, BN and ReLU activation layers by a single SCS layer. The FC neural network is not affected.

In Figure 6-8, we see that the test accuracy approaches the performance of the CNN model and is calculated from the confusion matrix in Figure 6-9 as 0.9536. The precision and recall are equal to 0.9567 and 0.9526, respectively. This suggests that the SCS-NN model achieves better performance than the linear models and a similar performance to the CNN model with the advantage of having less trainable parameters and less complex architecture (see Table 6-3).

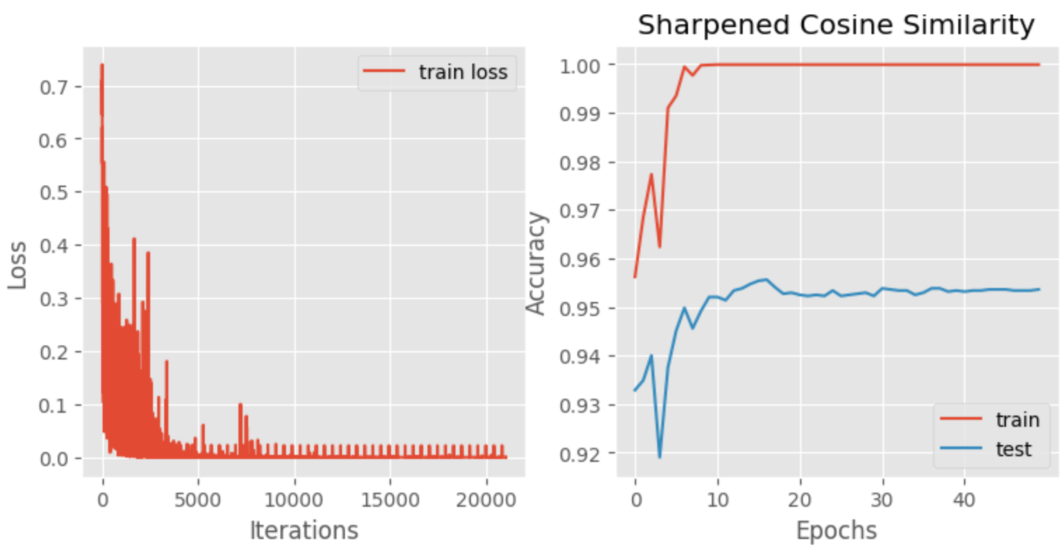


Figure 6-8: Performance curves for SCS-NN model

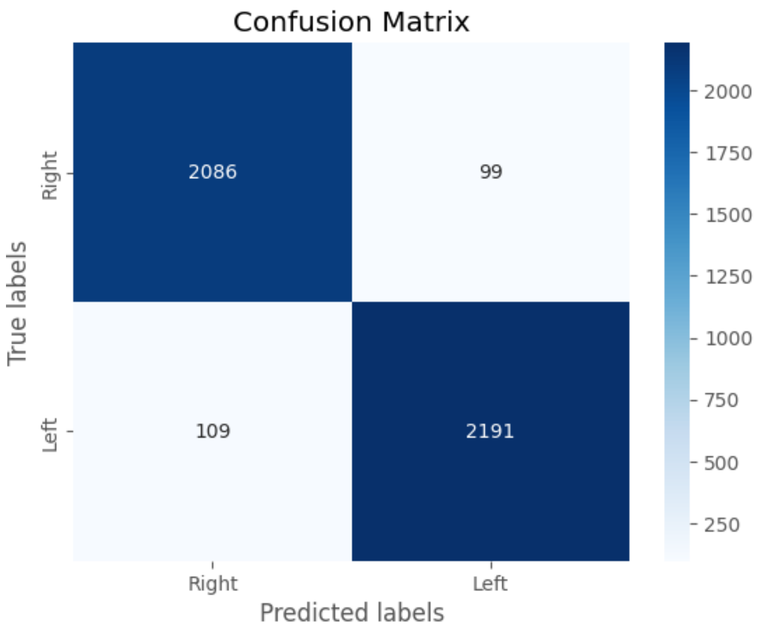


Figure 6-9: Confusion matrix corresponding to the SCS-NN model

6-1-5 CNN Model Interpretation

We apply the DeepSHAP explainer in order to interpret our 1-D CNN model. From Figure 6-10, we derive that the three features that play the most important role in the CNN model's predictions are m/z 1755.966, m/z 1754.964 and m/z 8564.515. We can see the ion images on the following figures.

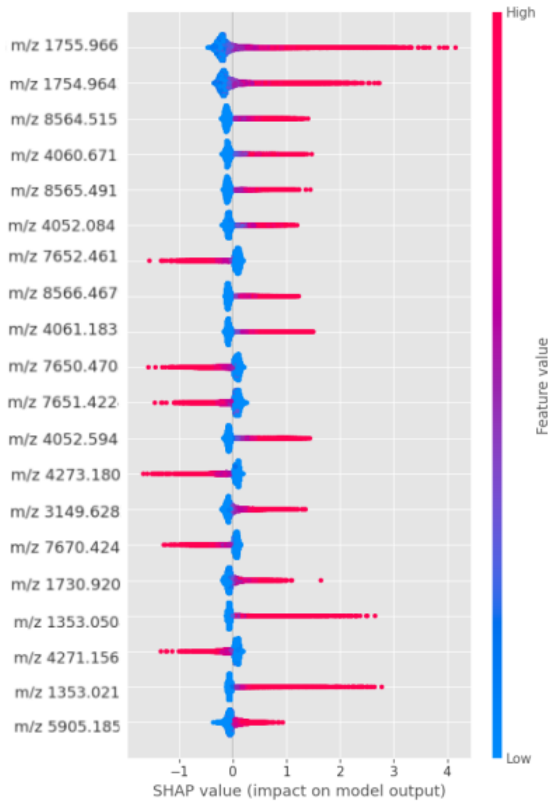


Figure 6-10: SHAP values for each feature sorted from most important to less important feature. The y-axis represents the feature names, while the x-axis represents the SHAP value for each feature. The colourbar shows the intensity of the feature on the sample, with red indicating high while blue indicating low intensity. This plot helps identify the most important features on the output of the CNN model as well as whether the impact of the feature is positive or negative based on its presence (high feature value) or absence (low feature value).

In Figure 6-10, we plot the contributions (SHAP values) of the most important features specifically for the positive class output, namely the left brain hemisphere. We see that for the three most important ions/features the higher the feature value (ion intensity), the larger the SHAP value in the positive axis. This means that when those ions have high intensity on the sample they contribute positively to the output of the model, while when they have low intensity they contribute negatively. Based on their ion images (see Figure 6-11, Figure 6-12, Figure 6-13) this is confirmed. Additionally, the corresponding SHAP maps are visualized in Figure 6-11, Figure 6-12, Figure 6-13 (Right - image), where we observe that indeed the regions where the feature value is high - on the left hemisphere - the corresponding SHAP value is positive, whereas the regions where the ion value is low the corresponding SHAP value is either negative or zero.

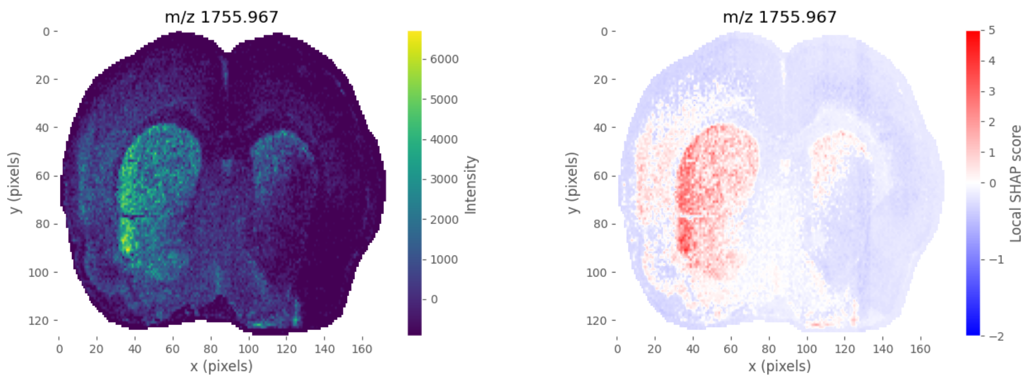


Figure 6-11: (Left - image) Ion image of the most important feature - (Right) SHAP score map of the most important feature for CNN

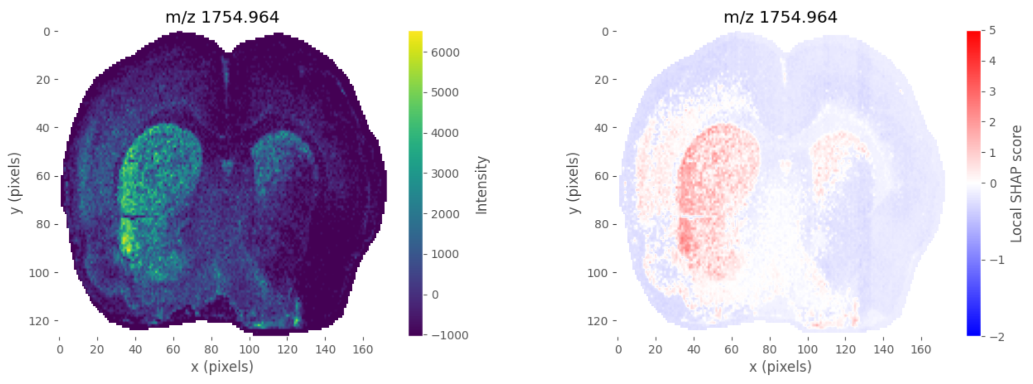


Figure 6-12: (Left - image) Ion image of the no.2 most important feature *m/z* 1756.964 - (Right) SHAP score map of the feature for CNN

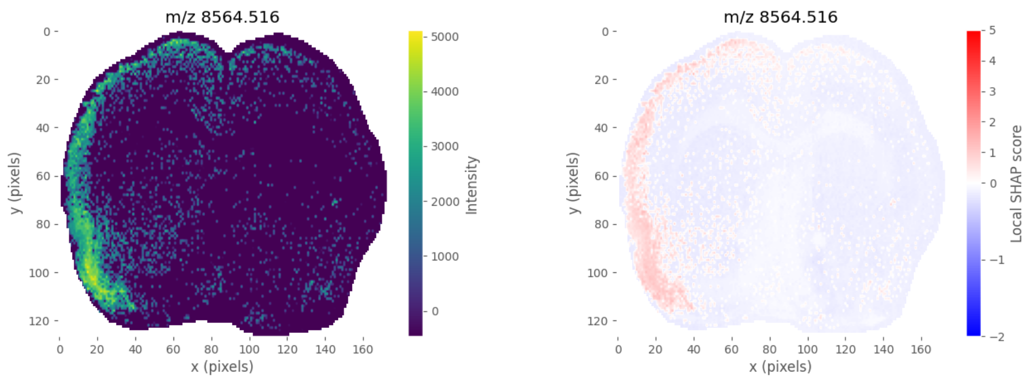


Figure 6-13: (Left - image) Ion image of the no.3 most important feature *m/z* 8564.516 - (Right) SHAP score map of the feature for CNN

Nevertheless, in Figure 6-10, by examining the features with the largest SHAP scores, we see that the fourth most important feature is m/z 4060.672. Interestingly, upon further investigation, it is discovered that this particular feature corresponds to a noisy input, as depicted in Figure 6-14 (Left). Despite being a noisy feature, it shows considerable influence over the model's decision-making process. This case highlights the vulnerability of the CNN model and the potential of relying on noise to make a prediction. Identifying such noisy samples, excluding them from the training dataset and retraining, would be crucial to enhance the robustness and trustworthiness of our model. This could be a form of model debugging.

Furthermore, there are also features such as m/z 7652.461, which is evaluated as contributing negatively to the model output when its intensity is high. In order to confirm this case we present the corresponding ion image for m/z 7652.461 in Figure 6-14 (Right). Based on the ion image, the ion's intensity is high on the right hemisphere and this validates the interpretation result, which assigns a negative impact on the output of the model (for the class "Left hemisphere") whenever this ion has high intensity.

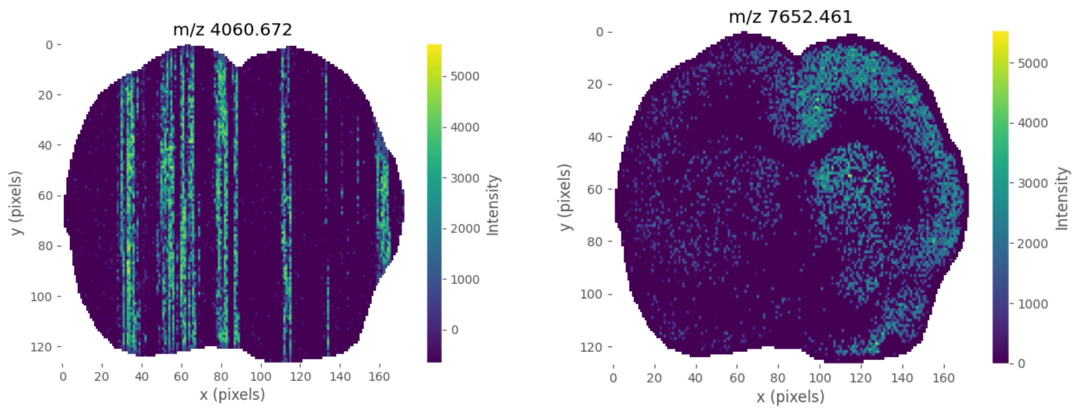


Figure 6-14: (Left) Fourth most important feature m/z 4060.672 representing a noisy sample (Right) Ion image of m/z 7652.461 corresponding to a negative impact on the model output when its intensity is high

6-1-6 Discussion

First of all, the results of the applied classifiers are summarized in Table 6-2. From the experimental results, it is obvious that the CNN model outperforms the rest of the other models. Especially compared to the linear model of LDA, the accuracy of the CNN model is 8% higher. On the other hand, the CNN model doesn't achieve significantly higher accuracy than logistic regression, 3% higher, neither than the SCS-NN, which is just 1% higher. Additionally, we can see that between the two different linear baseline methods, LDA performs worse than logistic regression. This could be attributed to the fact that LDA assumes that the data in the class follow a Gaussian distribution, however, from Figure 6-3, we see that the data are right skewed and this violates this assumption. On the other hand, logistic regression doesn't make any assumptions about the distribution of the features.

Table 6-2: Comparison between models for Parkinson's disease classification

Model	**Accuracy (%)**	**Precision (%)**	**Recall (%)**
LDA	88.60	89.7	87.8
Log.Regression	93.13	93.5	93.08
CNN	96.72	97.82	95.73
SCS-NN	95.36	95.67	95.26

Furthermore, if we focus on the comparison between the SCS layers and the convolutional layers, we can summarize the findings in Table 6-3. From this table we can conclude that:

Table 6-3: Comparison between CNN and SCS-NN models for Parkinson's disease classification

Architecture	**No. of Trainable Par.**	**Activation Layer**	**Normalization Layer**	**Train Duration (s)**
CNN	4,552,202	ReLU	BN	12
SCS-NN	4,551,362	None	None	16

- The SCS-NN achieves equally good accuracy but not better than the CNN model
- The SCS-NN requires less layers than the CNN architecture, since it doesn't involve batch normalization and ReLU activation layers. This leads to less trainable parameters and simpler architecture than the CNN model.
- Training epoch duration is higher for the SCS-NN model than the CNN, although less trainable parameters are involved.

6-2 Dataset no.2: Rat kidney

The second case study is based on a dataset that was obtained from a tissue sample of a rat kidney, which has been sectioned sagittally. The section is $12\mu m$ thick and the IMS data have been obtained using MALDI Quadrupole Time-of-flight (Q-TOF) with a 1,5-diaminonaphthalene matrix applied via sublimation [35].

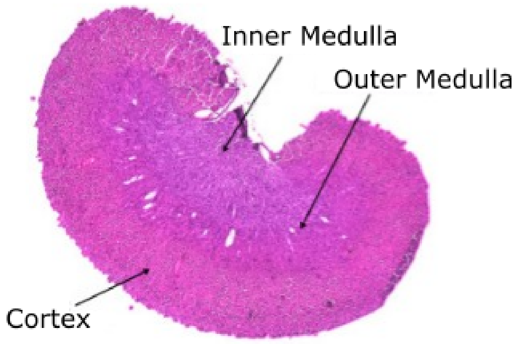


Figure 6-15: Microscopy image of the sagittal tissue section of the rat kidney [Image taken from Leonoor E.M. Tideman et al., Copyright ©2021, [35]]

In Figure 6-15, the hematoxylin and eosin stained microscopy image from the rat kidney tissue sample is illustrated. This sample is the one on which the IMS experiment is conducted and as a result the corresponding IMS datacube is obtained from it, using a prototype timsTOF fleX mass spectrometer (Bruker Daltonik, Bremen, Germany) [35]. In this case study, the objective of the classification task is to carefully differentiate among three discrete biological regions present in the rat kidney, namely:

1. Renal inner medulla (Class 1)
2. Renal outer medulla (Class 2)
3. Renal cortex (Class 3)

In a simpler manner, we can think of the rat kidney classification task as a problem with two classes - positive and negative - rather than three. The positive category includes one of the three specific renal regions each time, while the negative category includes all the other labelled pixels that don't belong to that region.

6-2-1 The Data

The size of the IMS data is comprised of totally 591,534 pixels with pixel size $15\mu m$. The detected m/z values per pixel lie within the range from 300 to 2000 and the total amount of the ion images is 1428.

Furthermore, for the purpose of this thesis, which applies supervised ML algorithms, annotated data was given. The annotated masks of the rat kidney dataset were defined manually, for the study conducted in [35], using exploratory analysis. Nevertheless, not all of the available pixels have been annotated because of difficulty, as illustrated in Figure 6-16 (Left).

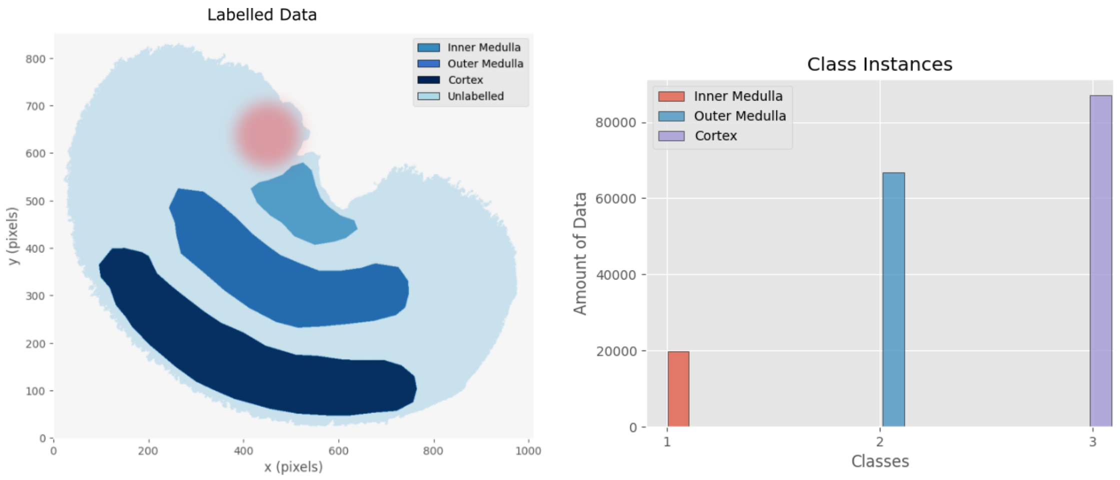


Figure 6-16: (Left) Pixel-wise annotation for the rat kidney (the red circle highlights the damaged region) - (Right) Instances distribution to the three rat kidney classes

Additionally, in this figure the reddish circle highlights a region that has been damaged due to sample preparation and thus, it's not trustworthy.

From Figure 6-16(Right), we derive that the total amount of the given labelled data sums up to 173,553 annotated pixels. From these, 19,727 belong to the 'Inner Medulla' class, 66,816 to the 'Outer Medulla' class and 87,010 to the 'Cortex' class.

Preprocessing

The data are centered prior to model training similarly to the previous dataset (see Figure 6-17). Additionally, 5-fold cross validation is used and train-test splits correspond to 80% and 20%, respectively.

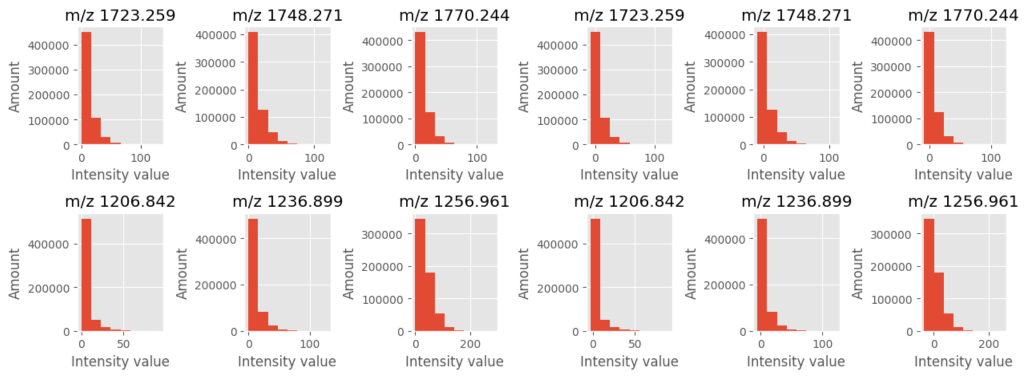


Figure 6-17: Examples of features distribution for kidney dataset (Left) before and (Right) after centering around the mean value of each feature column

Based on Figure 6-16, we have class imbalance, which can lead to poor performance of the trained classifiers due to the bias towards the class with the more observations, namely the majority class. When training a classifier based on an imbalanced dataset we expect to get a relatively low value for the error of the class with the more instances and an unreasonably high error for the class with the less instances. In [3], it is proposed that since in IMS datasets the amount of spectra is large, the imbalance issue can be tackled by downsampling the larger class, in order to achieve balance between the classes. This is what we apply in our case. More specifically, we use a reduction of the majority class, such that the minority class is equal to the 20% of the majority class.

6-2-2 Inner Medulla vs. Not Inner Medulla

For the classification problem 'Inner Medulla vs. Not Inner Medulla' the data are distributed in the negative (0) and positive (1) classes as $\{\textit{'0'}: 153,826, \textit{'1'}: 19,727\}$, meaning that the positive class is almost 12% of the total training dataset while the negative class instances constitute around the remaining 88%. We take care of this imbalance by randomly undersampling the majority class. After balancing the inner medulla class, we have $\{"0": 98,635, "1": 19,727\}$. We cannot further remove data-points because we lose important information and the classification is hindered rather than benefit.

Classification with a Linear Model - Logistic Regression

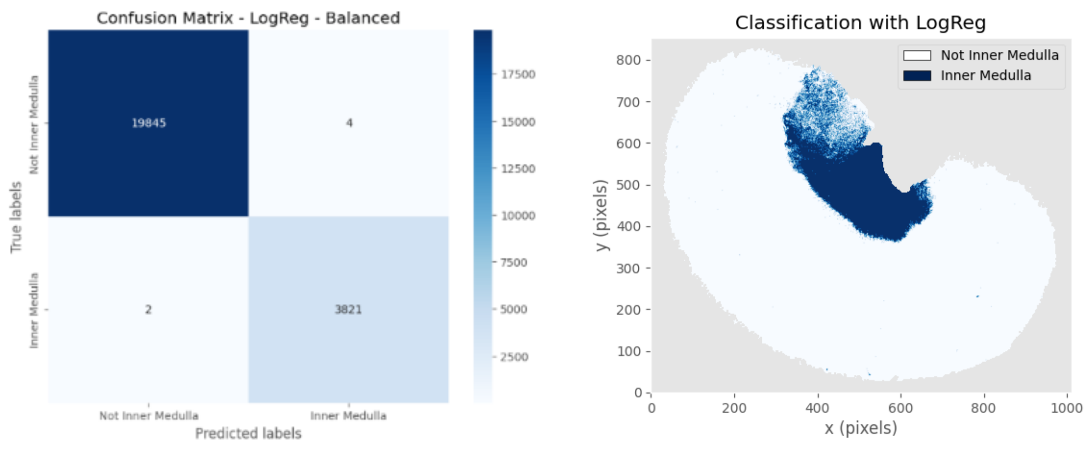


Figure 6-18: (Left) Confusion matrix of test split and (Right) classification output on "unseen" regions

From the confusion matrix in Figure 6-18 (Left), we calculate the precision = 0.9989, the recall = 0.9994, the F1 score=0.9991 and the balanced accuracy= 0.9995. Those results, indicate a very high performance of the logistic regression model on the test set. The F1 score and the balanced accuracy are chosen as measures of the model's performance because they take into account class imbalance in the data. Furthermore, in Figure 6-18 (Right), we observe the classification output of the trained logistic regression model on "unseen" data during training. The inner medulla region is classified correctly with good accuracy, however, we see that quite a few of the pixels that shouldn't be part of the inner medulla, since they belong to the noisy region based on Figure 6-16, are mistakenly classified as inner medulla pixels.

Classification with a CNN

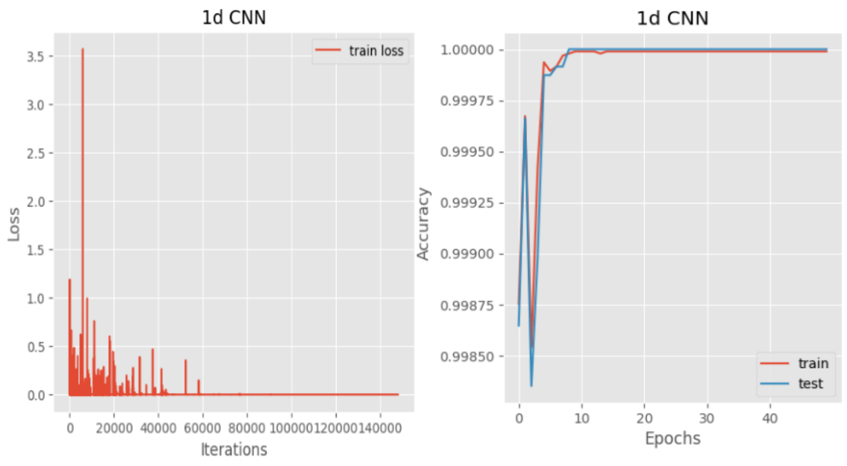
After following the methodology as described in chapter 5, the resulting architecture designed for this classification task is the one depicted in Table 6-4 and visualized in Figure A-2. We trained the model using Adaptive Moments (Adam) optimizer and adjust the initial learning rate, 0.00039, using the step policy every 4 epochs-step size- by a rate of decrease γ, 0.1249. As a regularization metric we use both L2 regularization and dropout layers with probability 40%, which was tuned manually via trial and error.

Table 6-4: Designed CNN architecture for inner medulla classification.

Hyperparameters	**TPE Architecture**
Kernel Size	7
Stride	4
Pooling Kernel	4
Pooling Stride	1
Total Conv. layers	10
Stacked Conv. layers, 1	1
Stacked Conv. layers, 2	4
Stacked Conv. layers, 3	4
Stacked Conv. layers, 4	1
no. of Filters for 1^{st} Hidden layer	30
no. of FC layers	1
no. of Nodes for FC layer	900
LR Policy	'Step'
Exp. decay rate	non-applicable
γ	0.1249
step size	4
Initial LR	0.00039
Batch Size	32
L2 Regularization	3e-06

In Figure 6-19 (Top), we see the performance curves of training the model for 50 epochs and testing it on a separate dataset. The model reaches an accuracy of 100% for the test split. In Figure 6-19 (Bottom), we observe the relationship between the learning rate and the resulting training loss. For LR values larger than approximately, 10^{-4} we get a sharp rise in the training loss whereas below that value the training loss stays almost steadily close to zero. This indicates that an optimal value for the learning rate for this optimization problem is less or equal to 10^{-4}.

Furthermore, from the confusion matrix in Figure 6-20, we can calculate the precision, recall, F1 score and balanced accuracy based on Equation 3-19 - Equation 3-22 to evaluate the CNN model's performance.



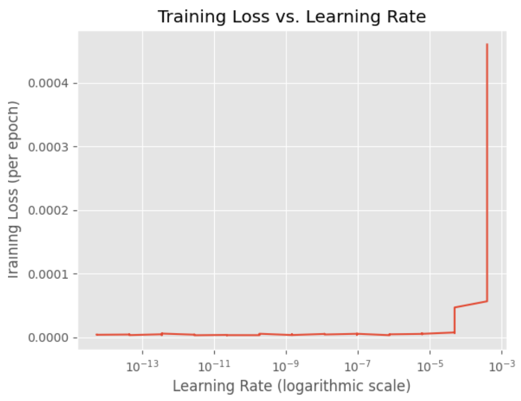
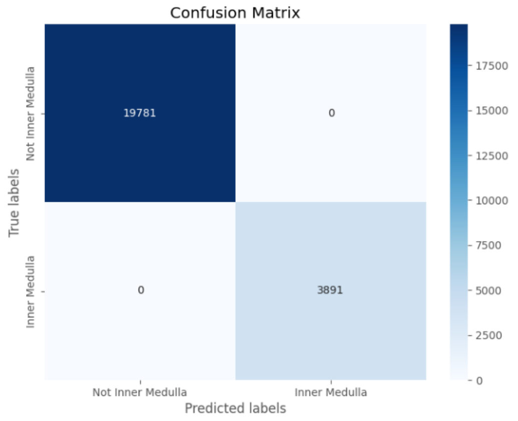


Figure 6-19: (Top) Training and testing performance curves of CNN model for renal inner medulla classification and (Bottom) training loss relationship with the learning rate decrease



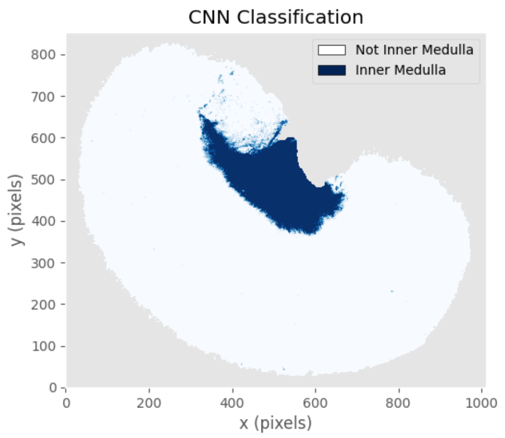


Figure 6-20: (Left) Confusion Matrix (Right) Classification output on unseen data for the task of inner medulla classification with a CNN model

The performance metrics for the CNN model based on the confusion matrix are: precision, 1.0, recall, 1.0, F1 score, 1.0 and bal. accuracy, 1.0. We evaluate how well the trained model generalizes, by testing it on "unseen" data, meaning data that hasn't been used for training or testing. The results are visible in Figure 6-20 (Right). We can see that most of the inner medulla pixels are correctly classified, with few misclassifications in the noisy region (see Figure 6-16). Nevertheless, the CNN model generalizes better than logistic regression.

Classification with a SCS-NN

If we replace the stacks of 1-D convolutional layers shown in Figure A-2 with SCS layers, we get an alternative less complex model, which achieves similarly high performance results as shown in the following figures.

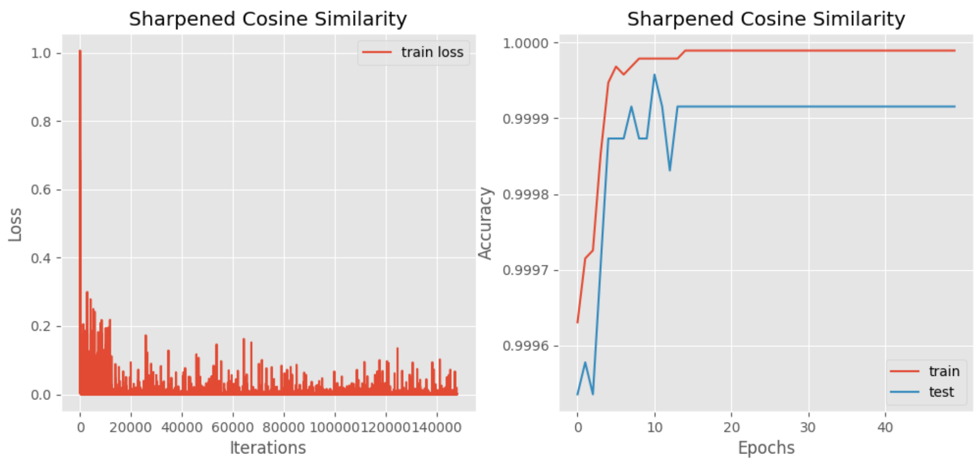


Figure 6-21: Training and Testing performance curves for SCS-NN model

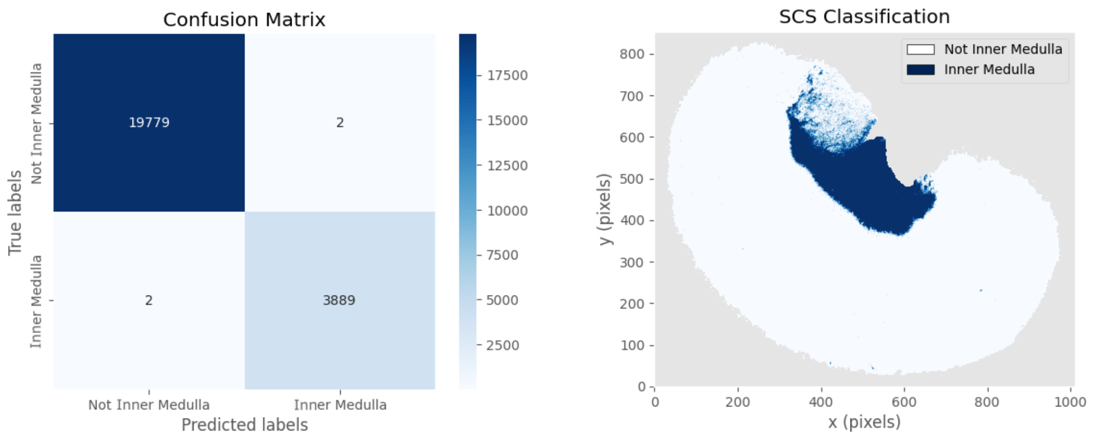


Figure 6-22: SCS-NN confusion matrix and classification of all available pixels for the task of inner medulla

In Figure 6-22, we see the confusion matrix (left figure) of the SCS-NN model on the test set and the corresponding classification output for the renal inner medulla on unseen data (right figure). We observe that the SCS-NN model manages to classify correctly the renal inner medulla pixels, with a few misclassifications on the noisy region. Furthermore, from the consufion matrix we calculate the precision as 0.9994, the recall as 0.9994, the F1 score as 0.9993 and the balanced accuracy as 0.9996.

CNN Model Interpretation

For the interpretation of the CNN model we are using the DeepSHAP algorithm as already mentioned in the methodology sections. The results of the SHAP score calculations are depicted in the following figures. In Figure 6-23, the most important features for the model output "Inner Medulla" are illustrated. Further, we present the SHAP score maps for the top-3 most important features next to the corresponding ion images.

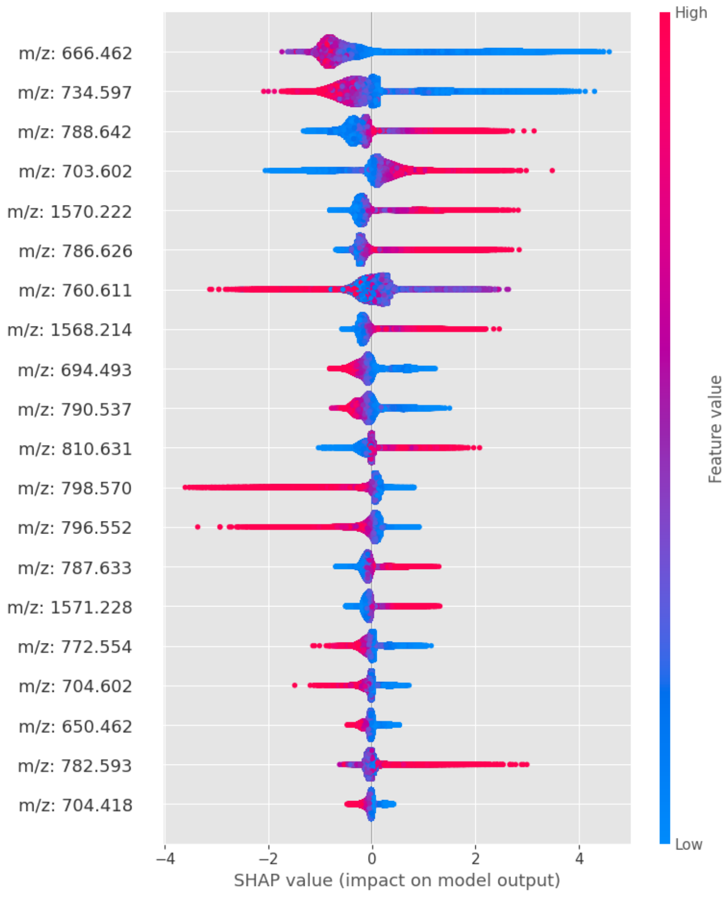


Figure 6-23: SHAP values, features in descending order based on their importance and impact on the model output "Inner Medulla"

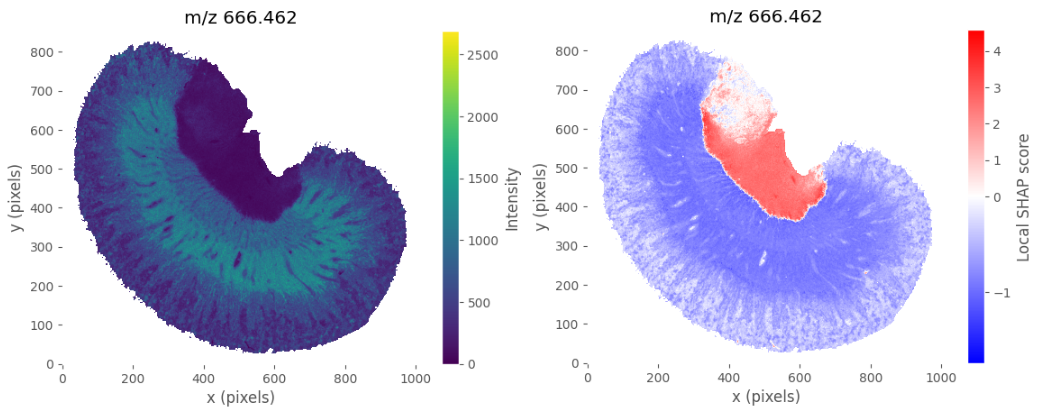


Figure 6-24: (Left) Ion image of the most important feature m/z 666.462 (Right) SHAP score map for the most important feature

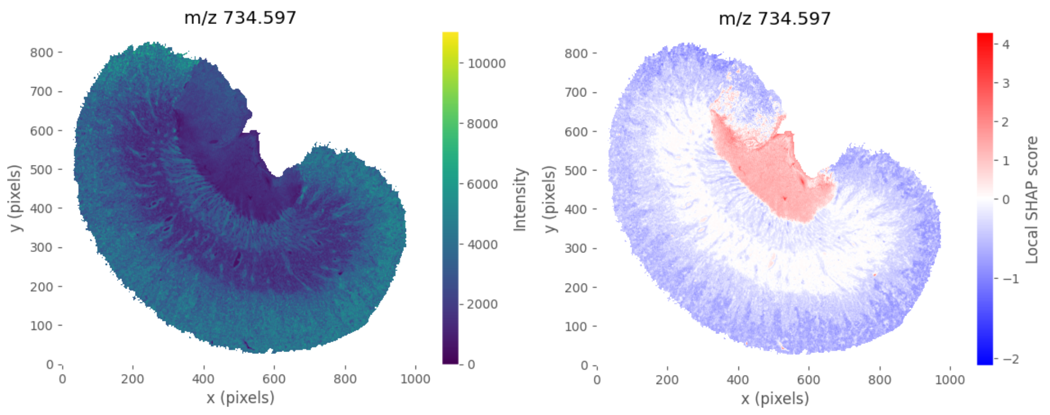


Figure 6-25: (Left) Ion image of the second most important feature m/z 734.597 (Right) SHAP score map for the second most important feature

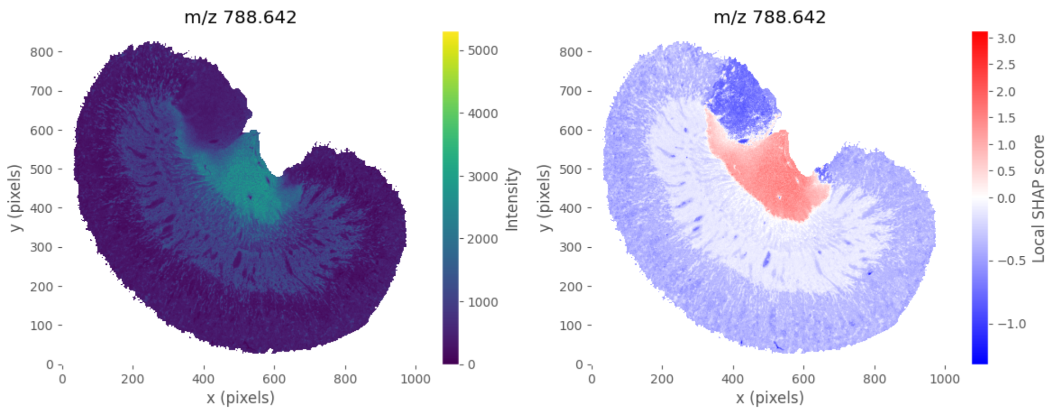


Figure 6-26: SHAP score map for the third most important feature

Discussion

In this study, we present a comparative analysis of three distinct models employed for the same classification task, namely the renal inner medulla classification. As a reference point, we use the linear model of logistic regression, which serves as a baseline for assessing the performance of more complex methodologies. Additionally, we introduce an alternative approach to the conventional convolutional layers, namely the Sharpened Cosine Similarity (SCS) layers, to evaluate their effectiveness in relation to the original CNN model. By comparing these models, we aim to investigate their respective capabilities in accurately categorizing the given data.

The logistic regression model, being a linear classifier, offers a simple method, frequently used as a baseline for comparison. On the other hand, we expect that the CNN model, making use of convolutional layers with non-linear activations, will have the potential for capturing more complex patterns and features within the data and achieveing higher performances. Furthermore, introducing the SCS layers provides an opportunity to explore an alternative method for IMS data classification, having the conventional CNN model as a basis.

Table 6-5: Comparison between models for renal inner medulla classification

Model	**Balanced Accuracy**	**Precision**	**Recall**	**F1 score**
Log.Regression	0.9995	0.9989	0.9994	0.9991
CNN	1.0	1.0	1.0	1.0
SCS-NN	0.9996	0.9994	0.9994	0.9993

From Table 6-8, we observe that all three models (Logistic Regression, CNN, and SCS-NN) demonstrate excellent performance in classifying renal inner medulla samples. The CNN model achieves perfect scores in all metrics, while the Logistic Regression and SCS-NN models achieve near-perfect accuracy, precision, recall, and F1-scores. Still, the non-linear methods, CNN and SCS-NN, give higher performances than the linear baseline method. Additionally, we see that also for this task the alternative of SCS-NN manages to perform equally well with the CNN with less complex architecture. In Table 6-3, we see an overview of both the CNN and SCS-NN models regarding their complexity and performance.

Table 6-6: Comparison between CNN and SCS-NN models for renal inner medulla classification

Architecture	**No. of Trainable Par.**	**Activation Layer**	**Normalization Layer**	**Train Duration (s)**
CNN	3,087,032	ReLU	BN	18
SCS-NN	3,022,502	None	None	48

Furthermore, it is also valuable to compare each model's performance on classifying data that is completely "unseen" during the training and testing process. In this way, we evaluate the generalization ability of each model. From Figure 6-27, we can conclude that the linear model achieves a significantly inferior performance in accurately distinguishing inner medulla pixels, since it is mistakenly including a lot of pixels that belong to the noisy region - as defined in Figure 6-16(Left) - in the inner medulla class. Additionally, the CNN model compared to the SCS-NN model generalizes better, since it avoids to classify the noisy region as inner medulla, whereas the SCS-NN model categorizes several pixels from the noisy region as belonging to the inner medulla, but still indicates better performance than the linear baseline model.

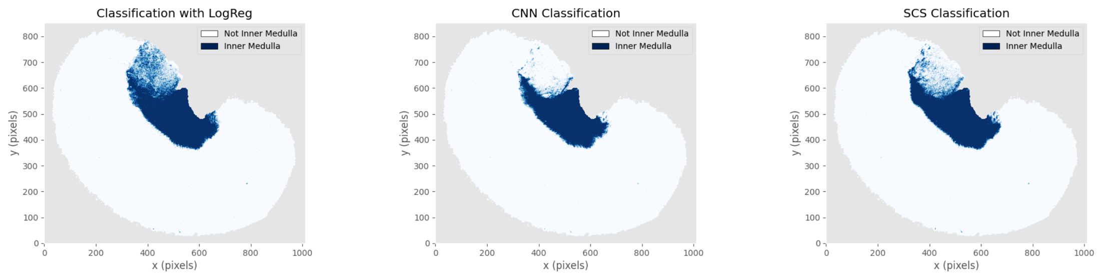


Figure 6-27: Classification output for renal inner medulla on "unseen" regions using (Left) logistic regression (Center) CNN and (Right) SCS-NN model

Furthermore, from the interpretation results on the CNN model, we can derive the following conclusions. Upon applying DeepSHAP, we observe the relationship between the three most important feature values and their corresponding SHAP scores. Specifically, we find that when the features' values are low, the SHAP score tends to be positive. This suggests that a lower ion intensity of those features positively influences the feature's contribution to the model's prediction concerning the inner medulla. To validate this finding, we have also conducted an analysis of the spatial distribution of these ions across the sample by referring to their corresponding ion images (Figure 6-24, Figure 6-25, Figure 6-26). Our observations confirm that these ions exhibit higher intensities primarily in regions of the kidney that do not belong to the inner medulla. This alignment between the CNN model's interpretation results and the actual features distribution offers credibility to the model's decision-making process. Consequently, we can infer that the model effectively focuses on relevant features rather than noise, providing trustworthy and reliable results.

6-2-3 Outer Medulla vs. Not Outer Medulla

For the classification problem 'Outer Medulla vs. Not Outer Medulla' the labelled data are distributed in the negative (0) and positive (1) classes as $\{'0' : 106,737, '1' : 66,816\}$, meaning that the positive class is almost 38% of the total training dataset while the negative class instances constitute around the remaining 62%. We take care of this small imbalance by randomly undersampling the majority class such that the minority class makes up 45% of the total amount of the labelled data.

Classification with a Linear Model - Logistic Regression

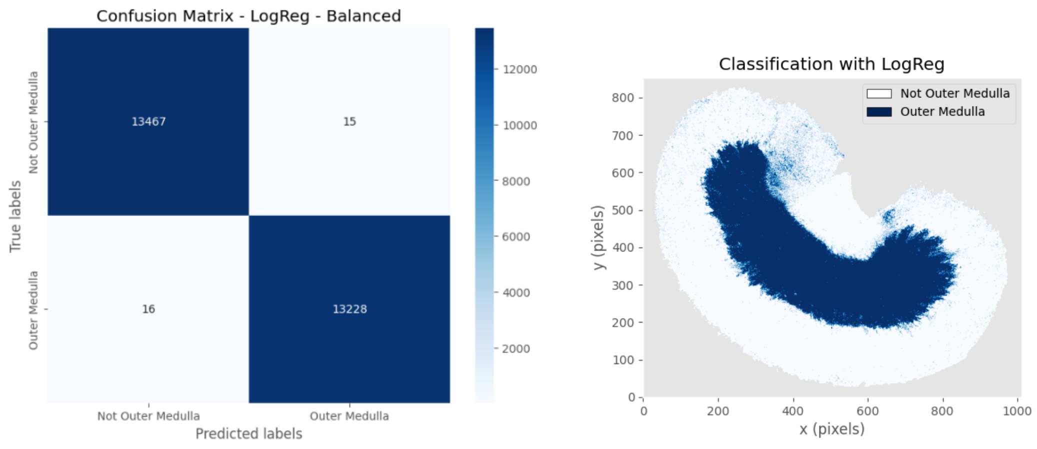


Figure 6-28: Confusion matrix on test split when classifier is trained with imbalanced classes (Left) and when it's trained with balanced data (Right) Classification output of logistic regression model for "unseen" data points

The precision and recall are calculated (Equation 3-22) as approximately 0.9988 and 0.9987, respectively. The F1 score of the resulting confusion matrix is almost 0.9987, and the balanced accuracy (Equation 3-19) is approximately 0.9988. Both of them are really high and they suggest that logistic regression achieves to capture the underlying data patterns and classify correctly the data. However, in Figure 6-28 (Right), we see the classification of data points that were not included in the training process. From that output we conclude that the logistic regression classifier is not able to generalize properly for "unseen" data, since we observe regions that clearly do not belong to the renal outer medulla to be classified as such.

Classification with a CNN

Herein, we would like to give an example of three alternative configurations of the best hyperparameters for the classification task involving the discrimination between Outer Medulla vs. Not Outer Medulla for three different random seeds. The hyperparameters are optimized using the Tree-structured Parzen Estimator (TPE) algorithm, (Table 6-7), that achieve equally good performances.

In Table 6-7, we observe that hyperparameters such as the kernel size, the stride offset, the pooling kernel and pooling stride step, which define the size and movement of the convolutional kernels and pooling operations differ slightly between the random seeds. Additionally, the number of convolutional layers varies across the different random seeds. Random Seed #1 and #2 have 10 layers, while Random Seed #3 has 7 layers. This suggests that the optimal depth of the convolutional architecture can vary. Overall, most of the hyperparameters vary across the different random seeds which demonstrates that different random seeds result in different optimal architectures. This variability suggests that the performance and behavior of the classification model can be influenced by the specific random seed used during the optimization process. Next, we need to distinguish between the alternative optimal configurations.

In order to select the most appropriate hyperparameter configuration we use 5-fold cross validation. The configuration with the highest mean performance across all validation splits is the one in the second column (Random Seed #2) and thus, this is the architecture used to derive the following results. Additionally, the architecture defined from the second column for the task of renal outer medulla classification is visualized in Figure A-3.

Table 6-7: Configurations of best hyperparameters for three different random seeds by TPE, for the classification task of Outer Medulla vs. Not Outer Medulla

Hyperparameters	**TPE Architecture**		
	Random Seed #1	**Random Seed #2**	**Random Seed #3**
Kernel Size	16	17	17
Stride	3	3	4
Pooling Kernel	2	2	2
Pooling Stride	4	4	3
Total Conv. layers	10	10	7
Stacked Conv. layers, 1	1	1	1
Stacked Conv. layers, 2	1	5	1
Stacked Conv. layers, 3	3	4	4
Stacked Conv. layers, 4	5	0	1
no. of Filters for 1st Hidden layer	55	55	15
no. of FC layers	1	1	1
no. of Nodes for FC layer	1000	800	600
LR Policy	'Exp'	'Step'	'Step'
Exp. decay rate	0.45	unapplicable	unapplicable
γ	unapplicable	0.3276	0.8899
step size	unapplicable	7	10
Initial LR	6.19e-05	0.0001	0.0002
Batch Size	64	32	32
L2 Regularization	3e-06	4e-06	4.05e-06

Next, we present the results of training the designed 1-D CNN architecture for 50 epochs. We analyze the decrease of the training loss (cross entropy) as the iterations increase. Addi-

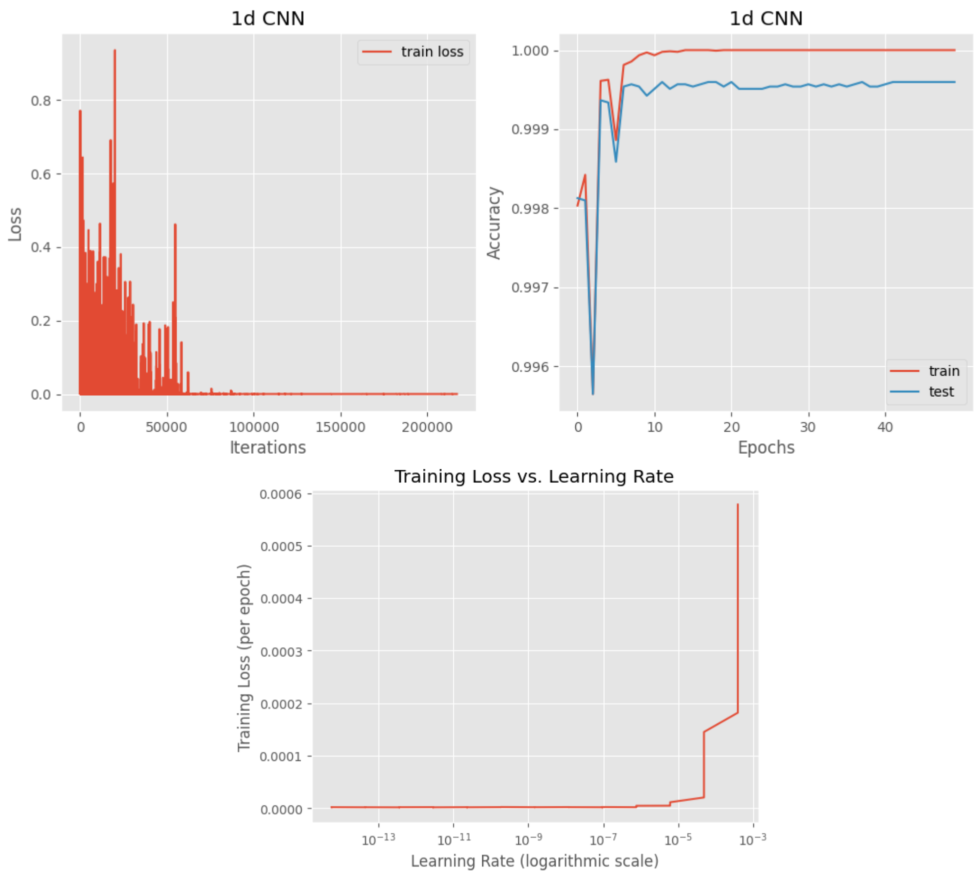
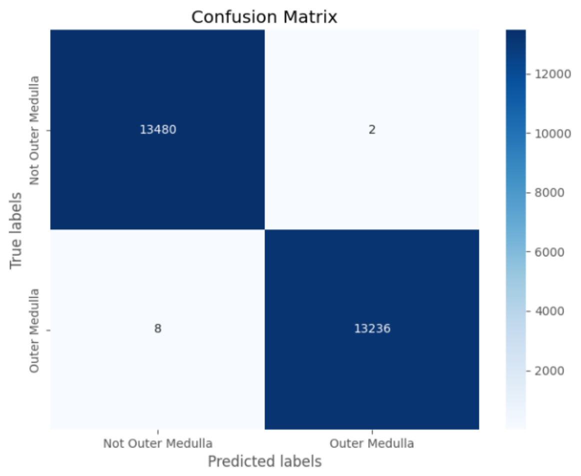


Figure 6-29: (Top) Training and testing curves for outer medulla classification with CNN model and (Bottom) training loss relationship with learning rate decrease



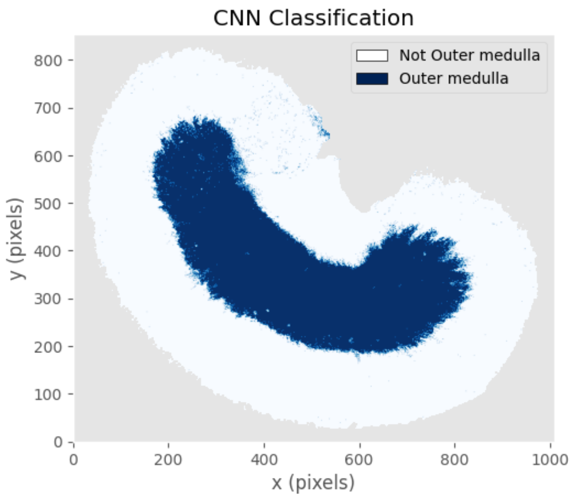


Figure 6-30: (Left) Confusion matrix of testing set for outer medulla classification (Right) CNN outer medulla classification result on "unseen" data

tionally, we visualize the overall accuracy on the training and testing set Figure 6-29 (Top) and we observe the realtionship between the learning rate and the training loss, Figure 6-29 (Bottom). Furthermore, we present the confusion matrix, Figure 6-30 (Left) of the model on the test set, which is a tabular representation that demonstrates the number of correctly and incorrectly classified samples per class - Outer Medulla, Not Outer Medulla - as well as the trained CNN model's output on "unseen" samples during the training process, Figure 6-30 (Right).

From the plot showing the training loss versus the learning rate, we observe that the learning rate policy that is used is efficient since it manages to reach optimal learning rate values. From the graph, it is obvious that learning rates lower than 10^{-5} achieve a training loss approximately equal to zero while above than 10^{-4} the training loss has a sharp increase, which by now we have seen that it is a typical relationship.

In Figure 6-29 (Top), we see that the selected model reaches an accuracy higher than 99.9% on the testing set. On the other hand, overall accuracy is not a representative metric since the classes are imbalanced, thus, next, we are also evaluating the performance based on more appropriate metrics. Additionally, from the confusion matrix in Figure 6-30 (Left) we calculate the precision, 0.9998, the recall, 0.9994, the F1 score, 0.9996 and the balanced accuracy, 0.9996. This shows that the CNN model does better than the linear model on all aspects. Furthermore, in Figure 6-30 (Right), it is observed that the CNN model classifies correctly most of the pixels belonging in the renal outer medulla, however, there are a few samples that are misclassified on the noisy circular region. Nonetheless, the amount of mistakenly classified pixels are much fewer than the ones presented in the logistic regression output.

Classification with a SCS-NN

The classification with the SCS layers is accomplished by replacing the convolutional layers of the selected architecture (Table 6-7) by sharpened cosine similarity layers, as well as removing the batch normalization and ReLU activation layers. The results are the following.

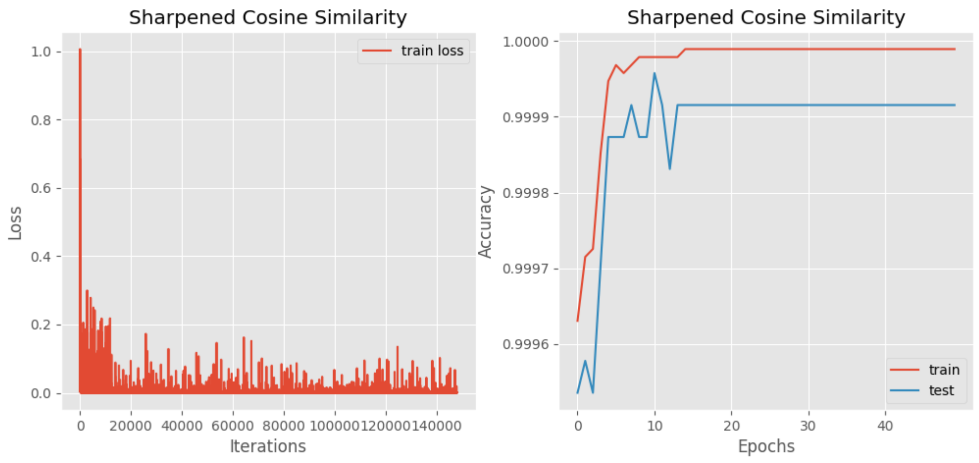
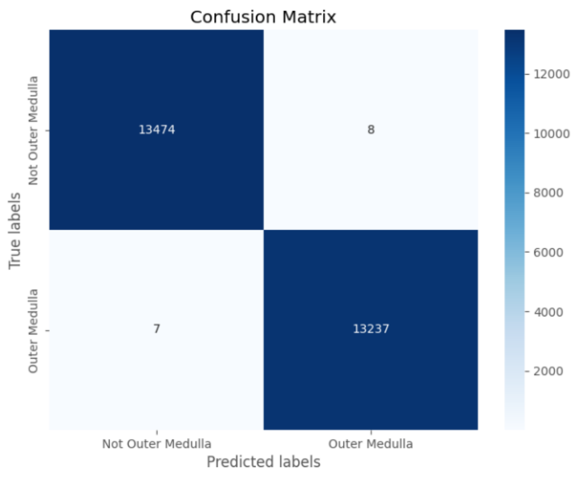


Figure 6-31: Training and Testing performance curves for SCS-NN model



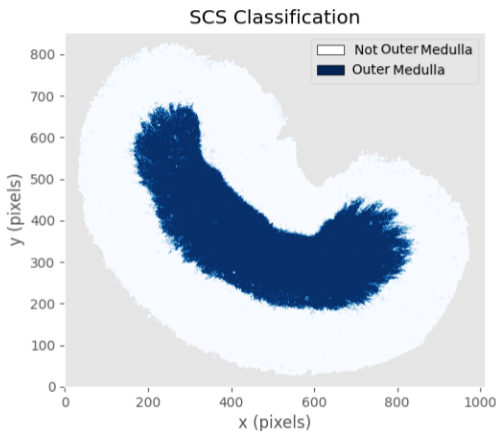


Figure 6-32: SCS-NN outer medulla confusion matrix and classification output for all available pixels for the task of outer medulla

From the confusion matrix in Figure 6-32, we can define the following properties: $TP = 13237$, $TN = 13474$, $FP = 8$ and $FN = 7$. Then, we can calculate the performance metrics precision, 0.9994, recall, 0.9995, F1 score, 0.9994 and balanced accuracy, 0.9995.

CNN Model Interpretation

For the interpretation of the CNN model we are using the DeepSHAP algorithm and the results of the impact of different features on the model output are depicted in Figure 6-33.

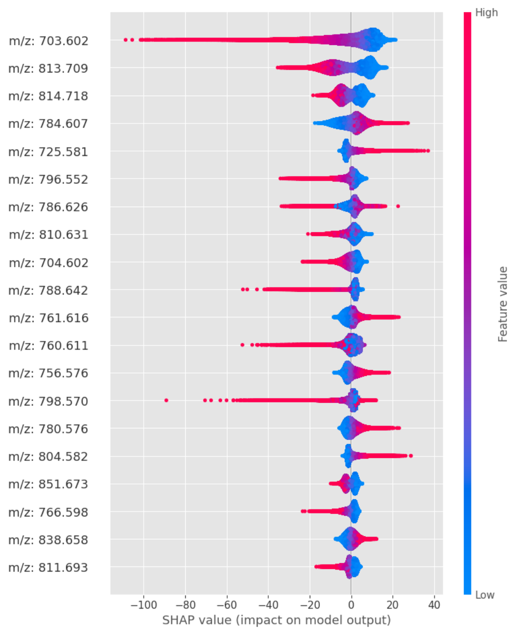


Figure 6-33: SHAP values and impact of features on the model output in the task of renal outer medulla classification

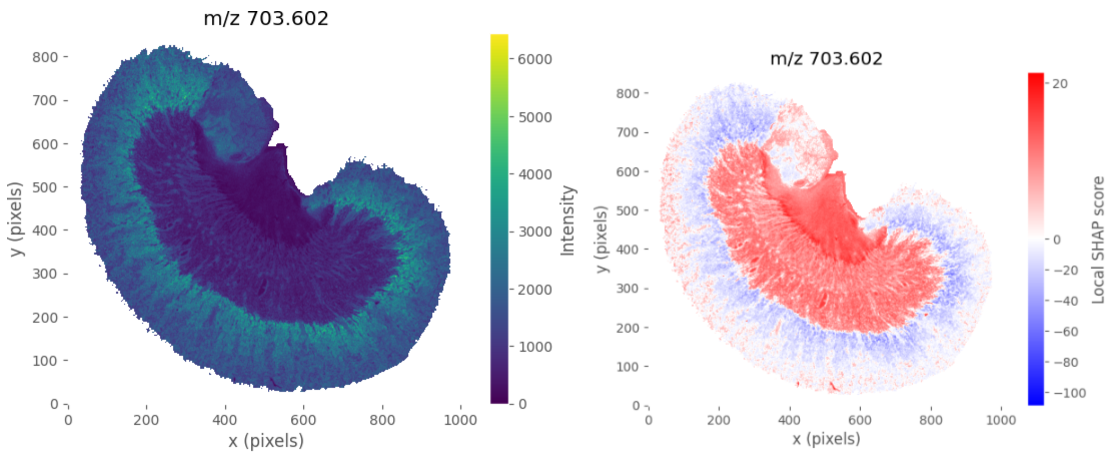


Figure 6-34: (Left) Ion image of m/z 703.602 (Right) SHAP score map for m/z 703.602 the most important feature for the CNN classifier in the renal outer medulla classification output

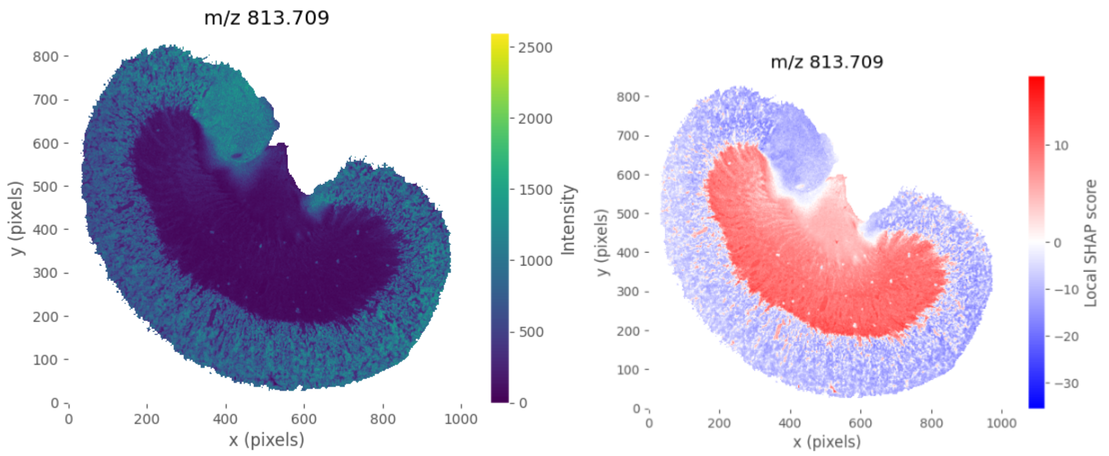


Figure 6-35: (Left) Ion image of m/z 813.709 (Right) SHAP score map for m/z 813.709 the second most important feature for the CNN classifier in the renal outer medulla classification output

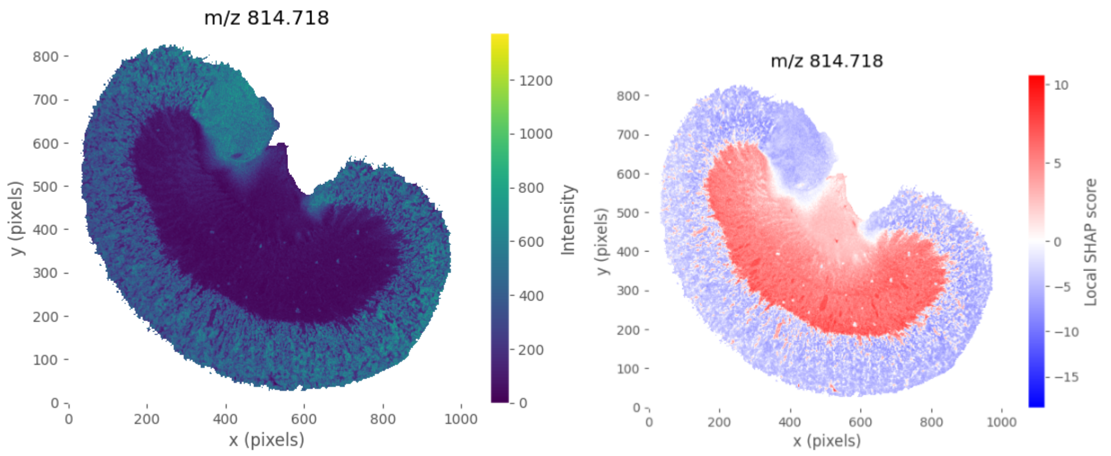


Figure 6-36: (Left) Ion image of m/z 814.718 (Right) SHAP score map for m/z 814.718 the third most important feature for the CNN classifier in the renal outer medulla classification output

In Figure 6-33, the relationship between the features, their corresponding SHAP values, and the model output is depicted. It is shown that the most important features for the renal outer medulla classification task are m/z 703.602, m/z 813.709 and m/z 814.718. The ion images, which show the spatial distribution of each of those features on the sample and SHAP score maps, which indicate the corresponding SHAP value of each feature across the surface of the sample, are depicted in Figure 6-34, Figure 6-35 and Figure 6-36. The SHAP values represent the contribution of each feature towards the model's output prediction. Negative SHAP values indicate that the corresponding feature has a negative effect on the probability of predicting 'Outer Medulla' in the output, whereas positive SHAP values indicate the opposite, while zero SHAP values indicate no impact of the corresponding feature on the model's output.

In the ion image of Figure 6-34, the regions where the m/z 703.602 ion has a high intensity are mostly across the renal cortex area as well as some part of the noisy region. On the other hand, the regions where it has a low intensity are mainly the renal outer and inner medulla parts. Based on Figure 6-33 and Figure 6-34 (Right), The SHAP values are negative for the regions with high intensity and positive for the regions with low intensity of this ion. This means that when this feature has high intensity it lowers the probability of predicting the outer medulla while when its intensity is low, the probability increases. Similarly, for the ion images of m/z 813.709 and m/z 814.718 the feature influences positively the probability of predicting outer medulla when it is absent and negatively when it is present.

Discussion

The results of opposing the three types of classifiers are summarized in Table 6-8. The evaluation metrics used for comparison include balanced accuracy, precision, recall, and F1 score.

Table 6-8: Comparison between models for renal outer medulla classification

Model	Balanced Accuracy	Precision	Recall	F1 score
Log.Regression	0.9988	0.9988	0.9987	0.9987
CNN	0.9996	0.9998	0.9994	0.9996
SCS-NN	0.9995	0.9994	0.9995	0.9994

When comparing the results, we observe once more that all of the models achieved high performances in all metrics. The CNN model achieved the highest scores in all evaluation metrics, indicating its superior performance. The sharpened cosine similarity model performs slightly lower than the CNN but still provided accurate classification results. The linear method achieved the lowest evaluation scores.

Table 6-9: Comparison between CNN and SCS-NN models for renal outer medulla classification

Architecture	Trainable Parameters	Activation Layer	Normalization Layer	Train Duration (s)
CNN	5,925,242	ReLU	BN	27
SCS-NN	5,922,272	None	None	73

The Table 6-9 compares the results and characteristics of two models, CNN and SCS-NN, for the classification of renal outer medulla. Some conclusions that are drawn from this table are that the CNN model employs ReLU activation and Batch Normalization for improved performance and generalization, while the SCS-NN model doesn't use such layers, which leads to less trainable parameters. Additionally, although SCS-NN has a simpler architecture, it requires almost three times longer to run a training epoch than the CNN model which takes much less time.

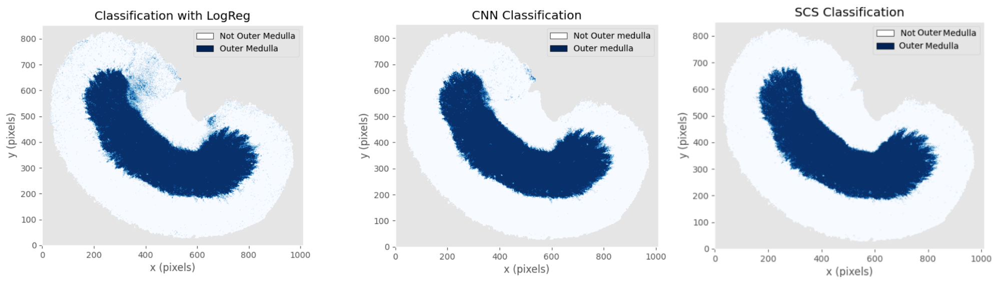


Figure 6-37: Renal outer medulla classification output on "unseen" data with (Left) logistic regression (Center) CNN (Right) SCS-NN model

In Figure 6-37, we oppose the three applied models outputs on data that hasn't been used during the training and testing procedure. The linear model is outperformed significantly by the CNN and the SCS-NN models, which have similar results.

6-2-4 Cortex vs. Not Cortex

For the classification problem 'Cortex vs. Not Cortex' the labelled data are distributed in the negative (0) and positive (1) classes as $\{\textit{'0'}: 86,543,\ \textit{'1'}: 87010\}$, meaning that the classes are almost perfectly balanced, thus, there is no need for application of any balancing technique.

Classification with a Linear Model - Logistic Regression

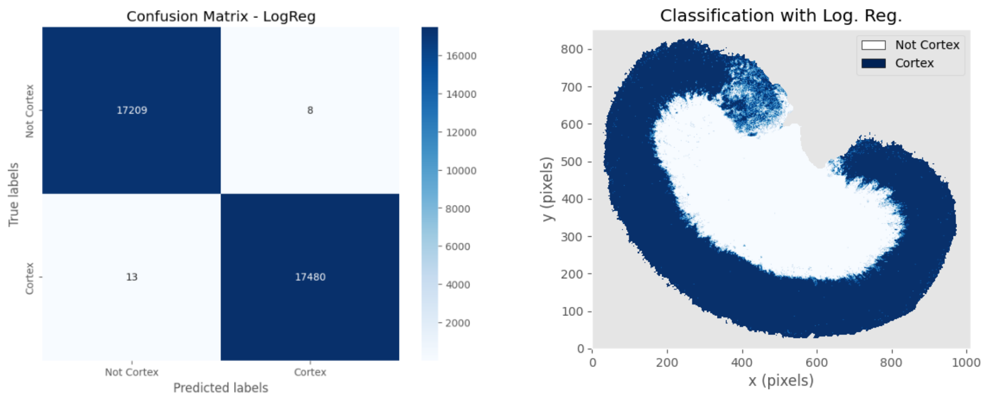


Figure 6-38: Renal cortex classification with logistic regression (Left) The confusion matrix on a test set (Right) The classifier's output for "unseen" data

Based on the confusion matrix on Figure 6-38 (Left), we can define TP = 17480, TN = 17209, FP = 8 and FN = 13 and use the Equation 3-19, Equation 3-20, Equation 3-21 and Equation 3-22 to calculate the corresponding balanced accuracy as 0.9993, the precision equal to 0.9995, the recall equal to 0.9992 and the F1 score equal to 0.9994. Precision represents the accuracy of positive predictions. A precision score of 0.9995 indicates a very high level of accuracy in correctly classifying cortex instances, while recall represents the ratio of correctly predicted cortex instances to the total actual cortex instances. A recall score of 0.9992 indicates a high level of sensitivity in detecting cortex instances. Finally, the F1-score combines precision and recall into a single metric and a score of 0.9994 reflects an excellent model effectiveness in both renal cortex prediction accuracy and capturing actual renal cortex instances. Additionally, in Figure 6-38 (Right) we observe the model's output for "unseen" data during the training and testing process. From the output we can conclude that the logistic regression classifier performs well on classifying the cortex region, however, it also classifies the noisy region (see Figure 6-16) as belonging to the renal cortex.

Classification with a CNN

The provided table, Table 6-16, outlines the designed CNN architecture for renal cortex classification, derived from the TPE hyperparameter optimization algorithm.

Table 6-10: Designed CNN architecture for renal cortex classification.

Hyperparameters	TPE Architecture
Kernel Size	16
Stride	2
Pooling Kernel	3
Pooling Stride	3
Total Conv. layers	8
Stacked Conv. layers, 1	1
Stacked Conv. layers, 2	2
Stacked Conv. layers, 3	2
Stacked Conv. layers, 4	3
no. of Filters for 1^{st} Hidden layer	40
no. of FC layers	1
no. of Nodes for FC layer	600
LR Policy	'Step'
Exp. decay rate	non-applicable
γ	0.4
step size	6
Initial LR	4.7e-05
Batch Size	32
L2 Regularization	0.00016

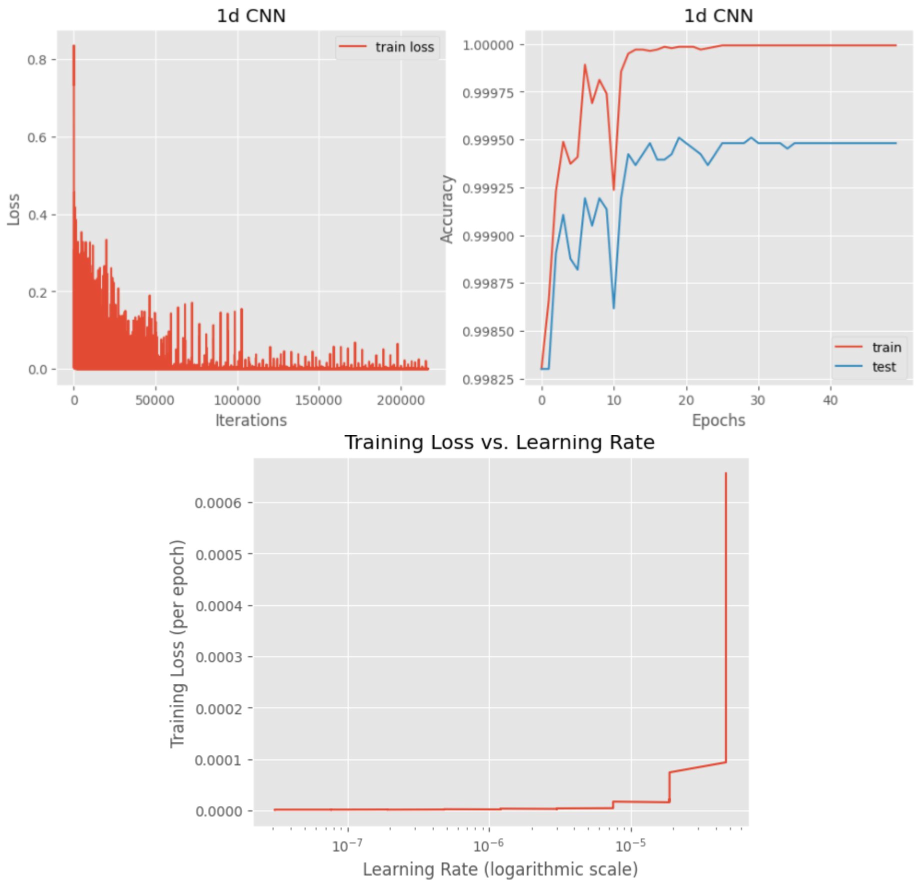


Figure 6-39: (Top) Training and testing performance curves for the task of renal cortex classification and (Bottom) the training loss relationship with the learning rate

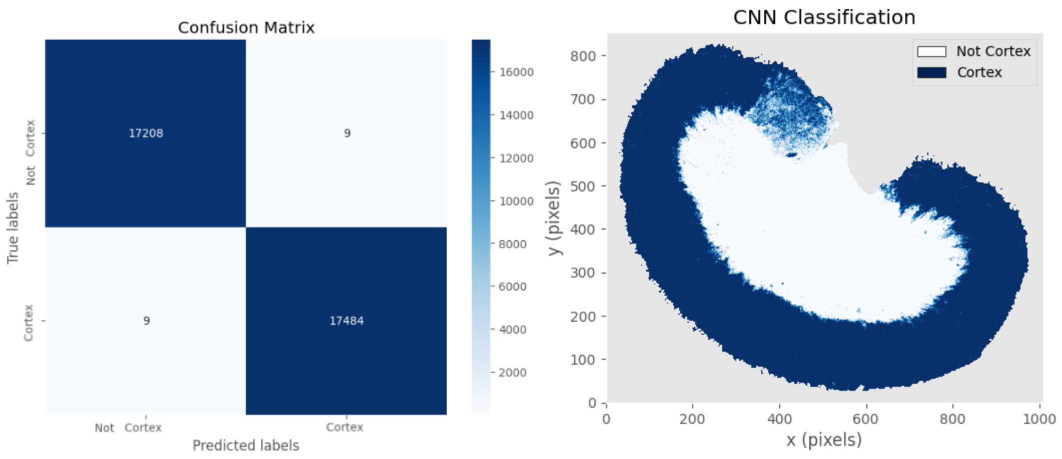


Figure 6-40: (Left) Confusion matrix and (Right) classification output of the CNN model on "unseen" data for the task of renal cortex classification

In Figure 6-39 (Top), we present the performance curves of the designed architecture during 50 training epochs. It is obvious that the model reaches a 100% accuracy for the training set and approximately 99.95 % accuracy for the testing set. Since our observations were evenly distributed into the two classes, we can trust the overall accuracy as a good metric. Additionally, as depicted in Figure 6-39 (Bottom), an analysis of the training loss per epoch in relation to the gradually decreasing learning rate per epoch reveals a substantial trend, which was also notable in all the other classification tasks. It becomes evident that when the learning rate exceeds a threshold of approximately 10^{-5}, there is a sharp increase in the training loss.

In Figure 6-40 (Left), from the confusion matrix, as we have already described, we can calculate the balanced accuracy, 0.9995, the precision, 0.9995, the recall 0.9995 and the F1 score 0.9995. Those metrics illustrate an excellent performance on classifying renal cortex. Furthermore, by examining the predictions of the CNN model on "unseen" data as depicted in Figure 6-40 (Right), we see that the CNN model manages to classify the renal cortex region quite effectively also for "unseen" data. This means that it generalizes well and it doesn't overfit just for the training data.

Classification with a SCS-NN

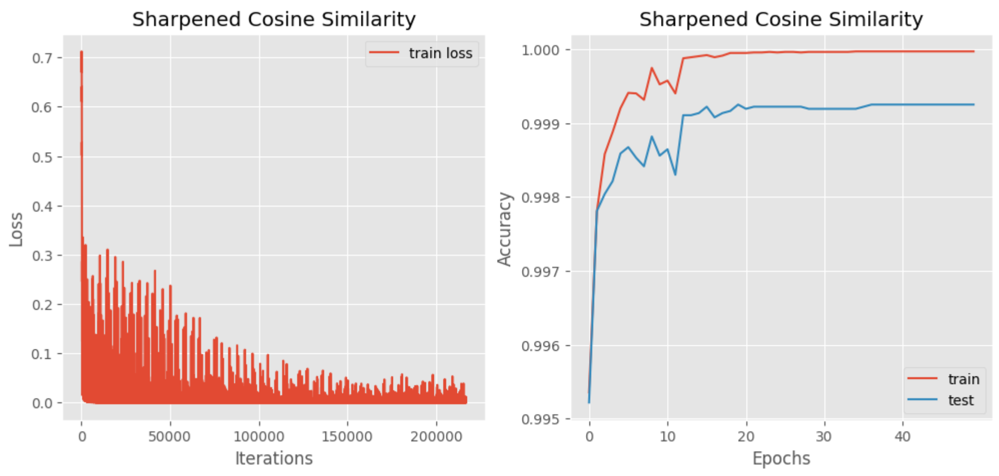


Figure 6-41: Training and testing curves for SCS-NN model for the task of renal cortex classification

In Figure 6-41, we present the performance curves for 50 training epochs using SCS layers instead of convolutional. The testing set balanced accuracy, as calculated based on the confusion matrix, Figure 6-42 (Left), is equal to 0.9993, the precision is equal to 0.9992, the recall is 0.9993 and finally, the F1-score is equal to 0.9993. Additionally, in Figure 6-42 (Right) we see the model's performance for "unseen" data, which reveals that the SCS-NN model effectively classifies the renal cortex region, however, the noisy region is also here included to the cortex region.

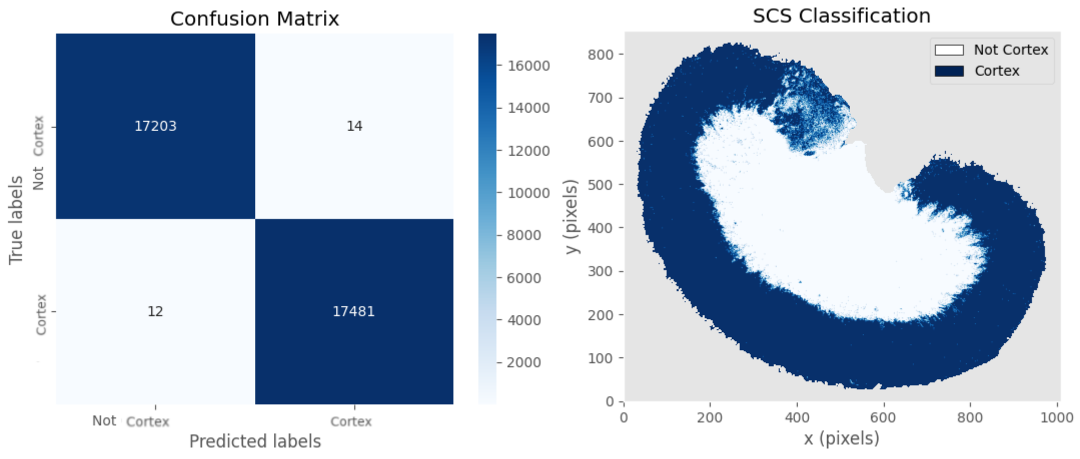


Figure 6-42: (Left) Confusion matrix and (Right) classification output of the SCS-NN model for all available pixels

CNN Model Interpretation

From applying, DeepSHAP as interpretation methodology for the CNN model, we derive the following results, regarding the features' importance, their corresponding SHAP values and their influence to the output in the task of renal cortex classification.

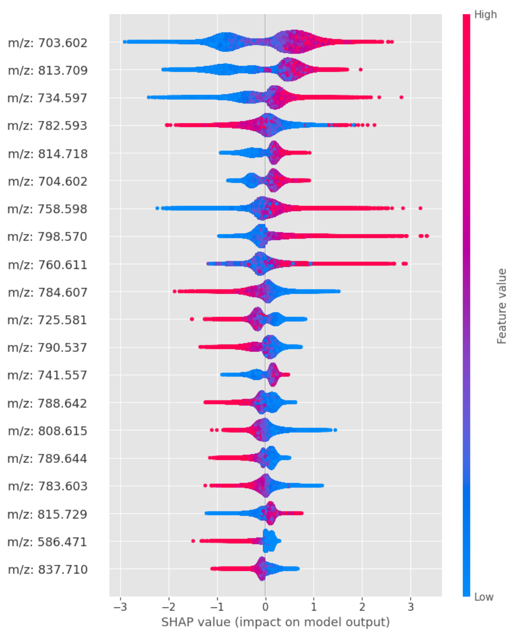


Figure 6-43: SHAP values and impact of features on the CNN model output in the task of renal cortex classification. The features are presented in descending order of importance.

As depicted in Figure 6-43, the three most important features sorted in descending order are m/z 703.602, m/z 813.709 and m/z 734.597. According to this figure, whenever these feature values are low (illustrated by the blue colour in Figure 6-43) their SHAP values are negative. This means that the locations of the tissue sample that the intensity of those ions is low contribute to a negative probability of classifying the renal cortex class.

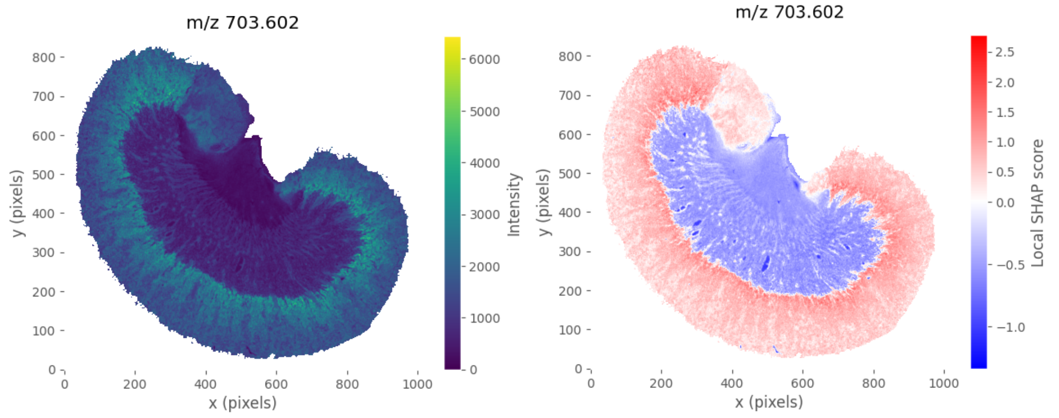


Figure 6-44: (Left) Ion image of m/z 703.602 (Right) SHAP score map for the most important feature, m/z 703.602

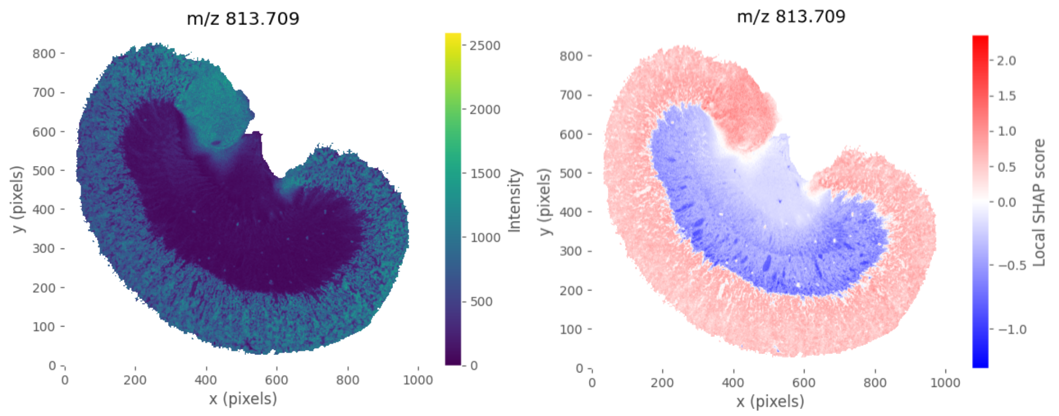


Figure 6-45: (Left) Ion image of m/z 813.708 (Right) SHAP score map for the second most important feature, m/z 813.708

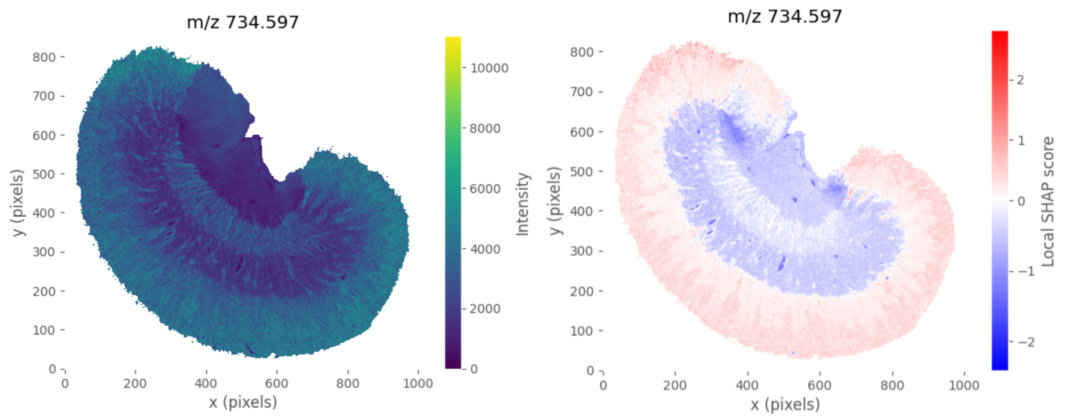


Figure 6-46: (Left) Ion image of m/z 734.597 (Right) SHAP score map for the third most important feature, m/z 734.597

From the respective ion images of the top three most important features in Figure 6-44 (Left), Figure 6-45 (Left) and Figure 6-46 (Left), it is evident that all three ions have low intensities in the regions of the kidney that do not correspond to the renal cortex. Therefore, the model correctly relies on inputs whose low intensities result in low probability of renal cortex classification. This is further illustrated in the SHAP maps Figure 6-44 (Right), Figure 6-45 (Right) and Figure 6-46 (Right), a visualization of the computed SHAP values per input pixel. From these maps we also validate that the pixels that have high intensities around the renal cortex region correspond to positive SHAP values, which indicates that they influence positively the probability of predicting the renal cortex class.

Discussion

Comparing the models, we can observe that all three models have very high performance across the evaluation metrics, indicating their effectiveness in classifying the renal cortex. However, there are slight differences in their results.

Table 6-11: Comparison between models for renal cortex classification

Model	Balanced Accuracy	Precision	Recall	F1 score
Log.Regression	0.9993	0.9995	0.9992	0.9994
CNN	0.9995	0.9995	0.9995	0.9995
SCS-NN	0.9993	0.9992	0.9993	0.9993

Based on Table 6-11, the CNN model consistently achieves the highest values for all metrics, including balanced accuracy, precision, recall, and F1-score. This suggests that the CNN model has the best overall performance and is capable of accurately classifying instances from the renal cortex data. Both logistic regression and SCS-NN models also demonstrate similar performance, however, they show slightly lower scores compared to the CNN model, indicating a slightly lower ability to classify instances accurately. Overall, based on the provided results, the CNN model stands out as the most promising option for renal cortex classification due to its consistently high performance across all evaluation metrics. However, we should also compare the generalization ability of each model as depicted in Figure 6-47.

Table 6-12: Comparison between CNN and SCS-NN models for renal cortex classification

Architecture	Trainable Parameters	Activation Layer	Normalization Layer	Train Duration (s)
CNN	9,286,882	ReLU	BN	20
SCS-NN	9,283,922	None	None	62

In Table 6-12, we present a more detailed comparison between the CNN and SCS-NN models. For the CNN model, almost 3,000 more trainable parameters are involved than the SCS model in addition to two extra activation and normalization layers. We observe that by replacing the convolutional layers by the simpler SCS layers the performance results in very high F1-score, indicating its equal effectiveness to the CNN model in classifying the renal cortex, with the advantage of not using additional layers and with less trainable parameters. Furthermore, the CNN model finishes one training epoch in 20 seconds, while the SCS-NN model takes 62 seconds. The CNN model's shorter training duration could be an advantage, as it allows for faster experimentation. However, it's important to consider that training duration alone does not provide a complete picture of the model's performance. Additionally, the longer training time needed for the SCS-NN model is contradictory to our expectation that it should take less time since it concerns updating less trainable parameters.

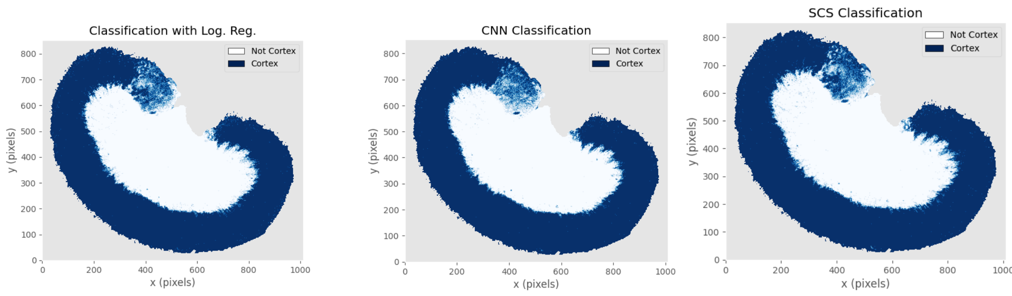


Figure 6-47: Renal cortex classification output on "unseen" data with (Left) logistic regression (Center) CNN and (Right) SCS-NN model

From Figure 6-47, we observe that all three models achieve similar results in predicting the renal cortex class for pixels that weren't included in the training and testing process. In conclusion, when it comes to the task of renal cortex classification, no model shows a substantial advantage over the others in terms of generalization and accuracy. However, the CNN model exhibits slightly better performance across all evaluated aspects.

6-3 Dataset no.3: Mouse brain hippocampal region

In this case study, the objective of the classification task is to differentiate among two discrete biological regions present in the mouse brain hippocampal region, namely:

- Ammon's horn (Class 1)
- Dentate gyrus (Class 2)

We view the classification problem as two different binary classification tasks, so that we categorize the desired positive class against the rest of the observations which are considered as the negative class. More specifically, we create two binary classification problems with "Amon's horn" vs. "Not Ammon's horn" and "Dentate gyrus" vs. "Not Dentate gyrus" the corresopnding positive and negative classes per problem.

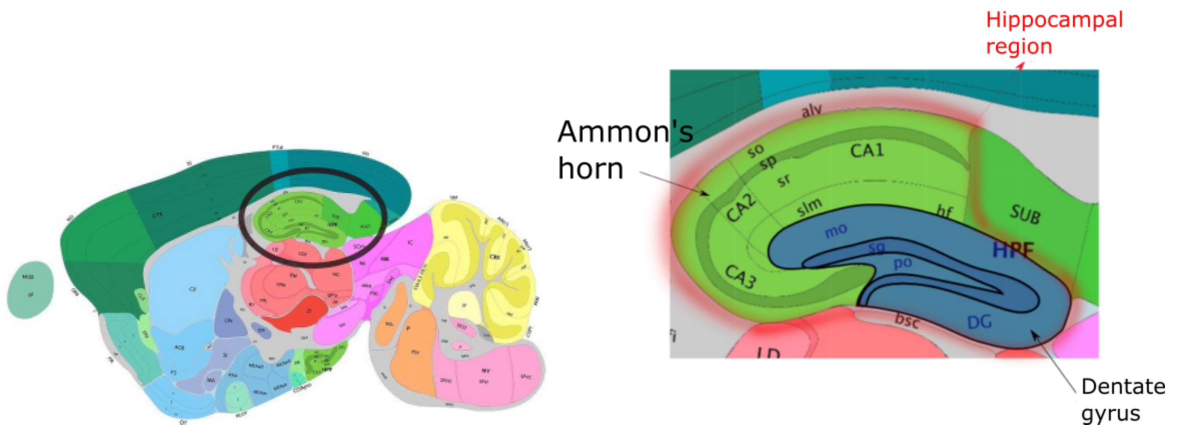


Figure 6-48: (Left) Sagittal section of a mouse brain taken from an interactive anatomical atlas [1] , where the black circle indicates the hippocampal region (Right) Zoom in the hippocampal region with the two distinct regions, namely Ammon's horn in light green and Dentate gyrus in blue color

In Figure 6-48, the hippocampus of a mouse brain is illustrated based on an anatomical atlas. On the right image of the figure, we see a zoomed in version of the brain, where we can distinguish the himpocampal region, encircled in red, divided in the Ammon's horn region depicted in light green colour and the Dentate gyrus region in blue.

6-3-1 The Data

The total amount of the data for the mouse brain hippocampal region is (71940, 438), with 71940 pixel positions and 438 detected features-m/z peaks. The data belonging to the class of Ammon's horn region comprise the 5% of the labelled data used for training, the rest 95% includes spectra labelled as Not Ammon's horn. This makes the classes extremely unbalanced, this is depicted in Figure 6-49.

Furthermore, the pixel-wise mask and the class instances distribution for the Dentate gyrus region are depicted in Figure 6-50. In this case, the amount of data contained in the positive class, 'Dentate gyrus', make up just 1% of the labelled data.

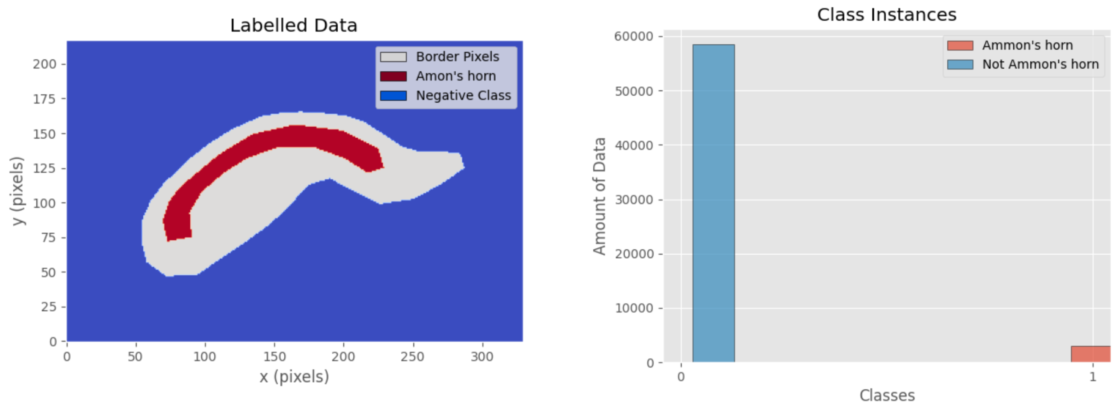


Figure 6-49: (Left) Mask of pixel-wise annotation for Ammon's horn (Right) The labelled data distribution into the positive - 'Ammon's horn' - and negative - 'Not Ammon's horn' - classes

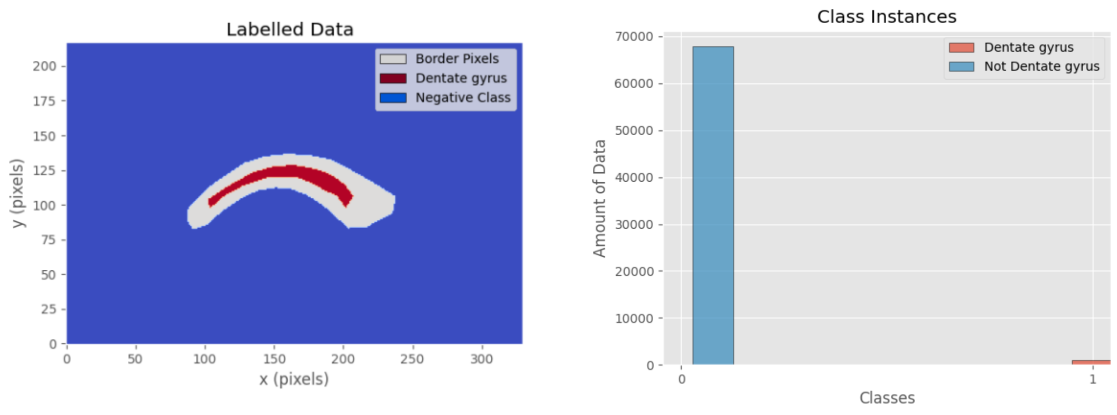


Figure 6-50: (Left) Mask of pixel-wise annotation for dentate gyrus (Right) The labelled data distribution into the positive-'Dentate gyrus'-and negative-'Not Dentate gyrus'-classes

Preprocessing

First of all, as we have already done for all the previous cases we apply data centering by subtracting the mean value of each feature across all observations Figure 6-51.

Next, we exclude the border pixels, shown in Figure 6-49 (Left) and Figure 6-50 (Left), from the training and testing sets due to the fact that it is not possible to label them with confidence. Moreover, we need to deal with the severe class imbalance for both classification problems. In order to compensate for the class imbalances, we randomly downsample the majority classes ("Not Ammon's Horn","Not Dentate Gyrus") by 20,000 observations - we don't remove further samples because then we lose important information.

Finally, we split the dataset into 80% training and 20% testing sets. Then we are ready to train our classifiers and evaluate their performance.

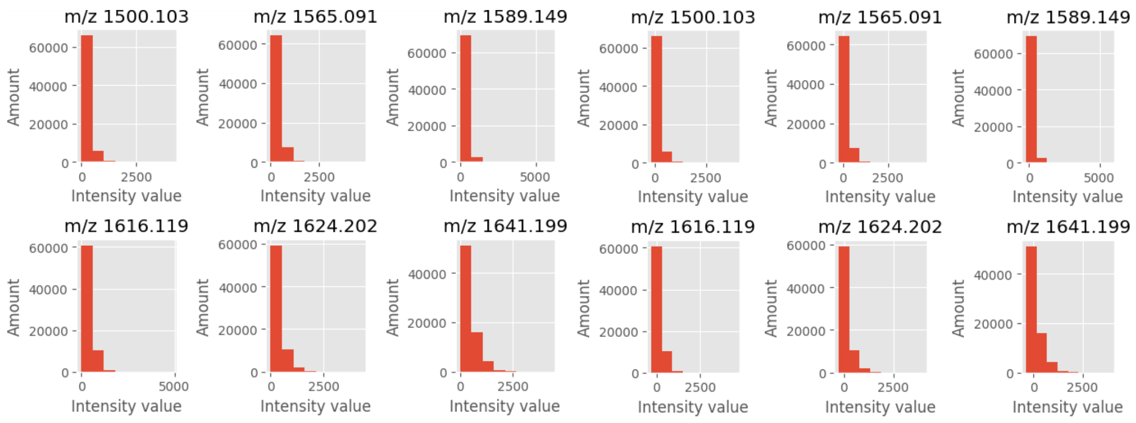
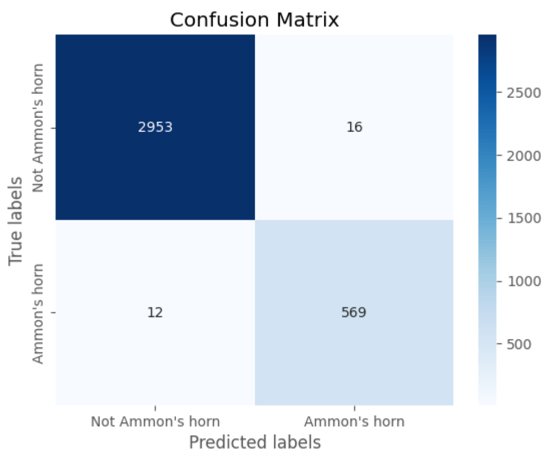


Figure 6-51: Data distribution before (left image) and after (right image) centering

6-3-2 Ammon's Horn vs. Not Ammon's Horn

Classification with a Linear Model - Logistic Regression



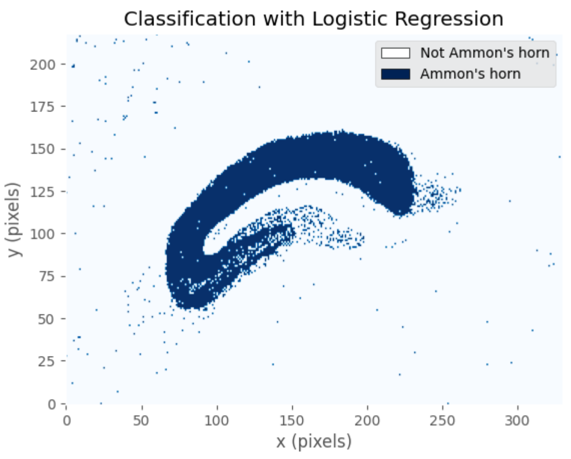


Figure 6-52: (Left) Confusion matrix for logistic regression (Right) Ammon's horn classification with logistic regression on unseen data

From the confusion matrix in Figure 6-52 (Left), we can compute the balanced accuracy 0.9912, the precision 0.9946, the recall 0.9959 and the F1 score 0.9947. In Figure 6-52(Right) we see the classifier's output also on "unseen" data. We notice that there are several pixels mistakenly classified as belonging to the "Ammon's horn" class although their location is irrelevant to the region of interest.

Classification with a CNN

The provided table, Table 6-13, describes the designed CNN architecture for Ammon's horn classification as generated from the TPE algorithm implementation.

Next, we present the corresponding performance curves while training and testing this model architecture, Figure 6-53. After 50 training epochs, the model reaches approximately 0.9955

Table 6-13: Designed CNN architecture for Ammon's horn classification.

Hyperparameters	**TPE Architecture**
Kernel Size	4
Stride	1
Pooling Kernel	4
Pooling Stride	1
Total Conv. layers	3
Stacked Conv. layers, 1	1
Stacked Conv. layers, 2	1
Stacked Conv. layers, 3	1
Stacked Conv. layers, 4	0
no. of Filters for 1^{st} Hidden layer	20
no. of FC layers	1
no. of Nodes for FC layer	700
LR Policy	'Step'
Exp. decay rate	non-applicable
γ	0.4
step size	3
Initial LR	7.5e-05
Batch Size	64
L2 Regularization	1e-04

accuracy in the testing dataset split. From Figure 6-53 (Bottom), we derive once more that after a specific threshold of the learning rate, the training loss abruptly increases. This threshold for the current optimization problem is a value close to 10^{-4}.

Furthermore, from the confusion matrix in Figure 6-54, we can calculate the balanced accuracy equal to 0.9931, the precision equal to 0.9966, the recall equal to 0.9980 and the F1 score equal to 0.9973. Additionally, in Figure 6-54 (Right), we see that the CNN model classifies the Ammon's horn region correctly without as many noisy pixels as the logistic regression classifier.

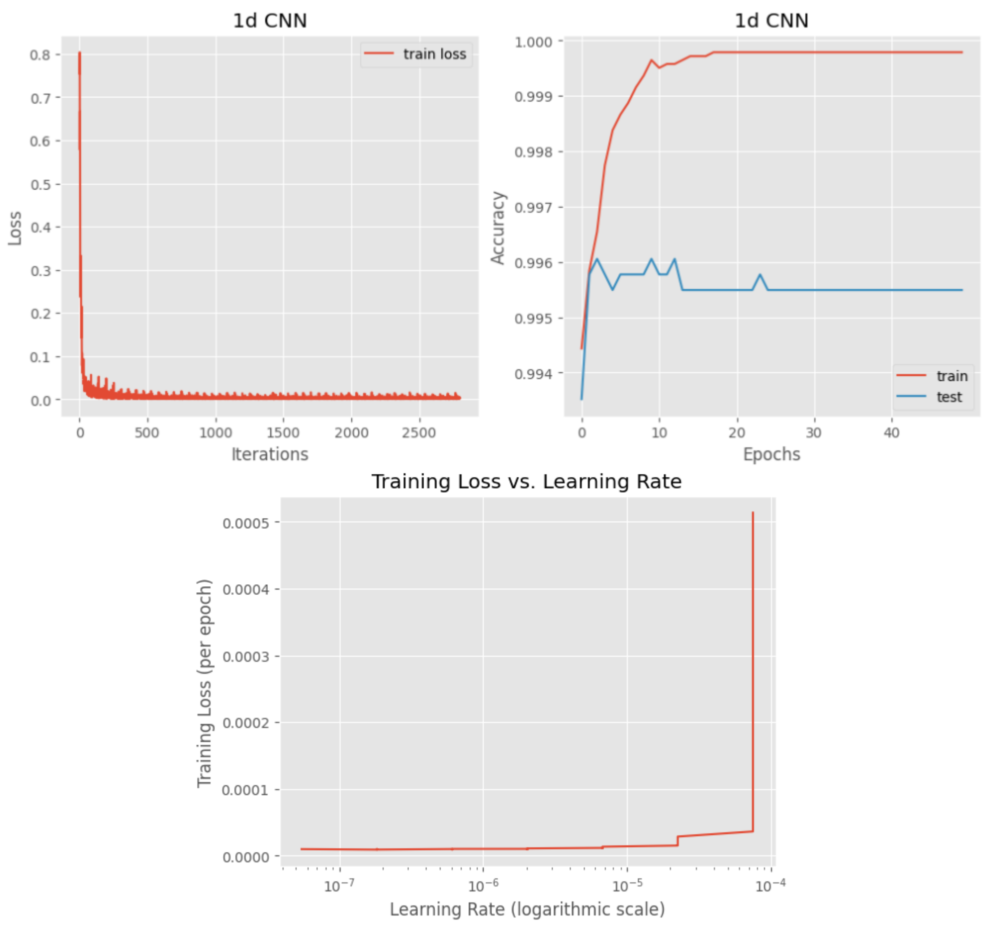


Figure 6-53: (Top) Training and testing performance curves for Ammon's horn classification with CNN and (Bottom) relationship between learning rate and training loss

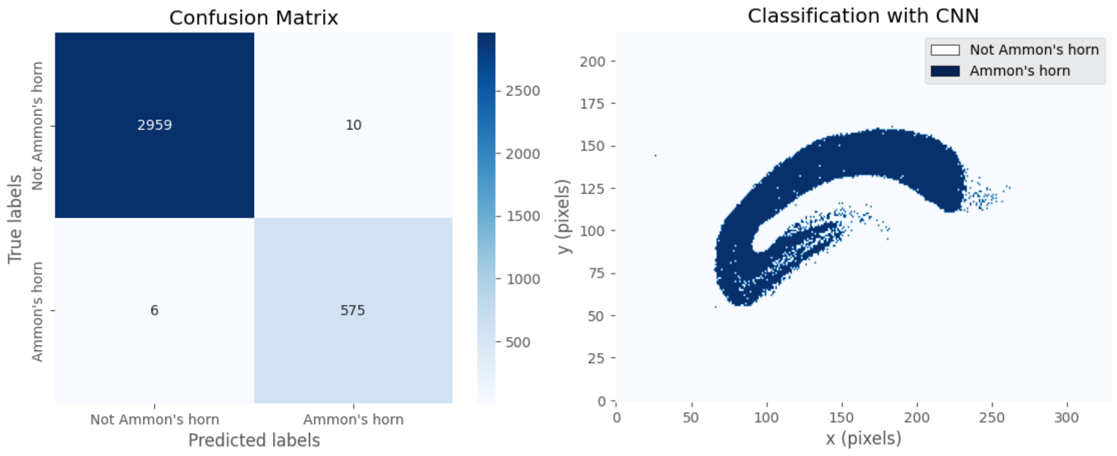


Figure 6-54: (Left) Confusion matrix for CNN model (Right) Ammon's horn classification using the CNN model

Classification with a SCS-NN

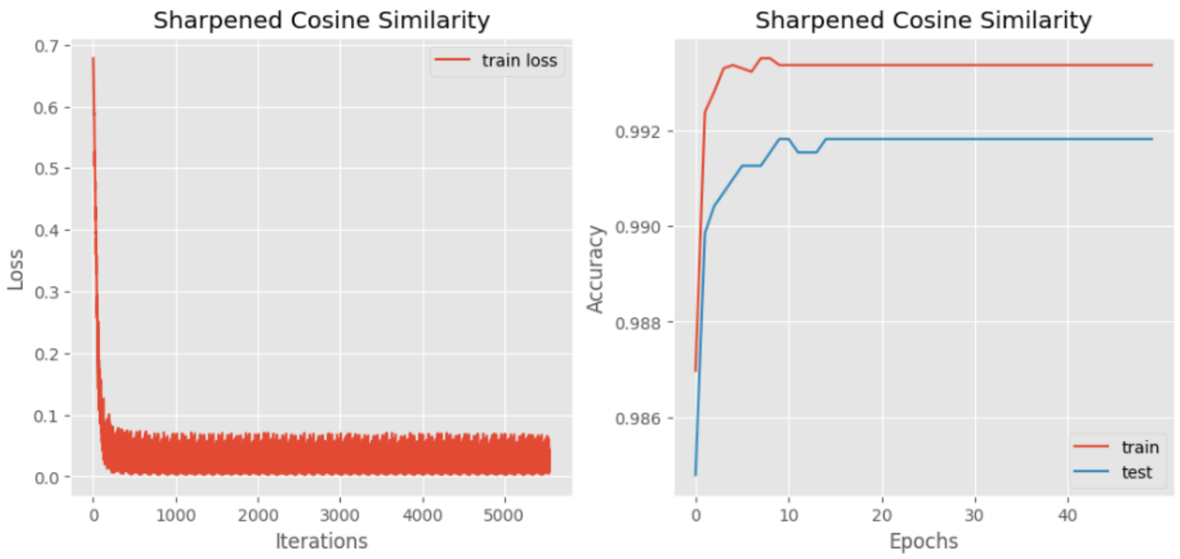


Figure 6-55: Performance curves for ammon's horn classification with SCS-NN model

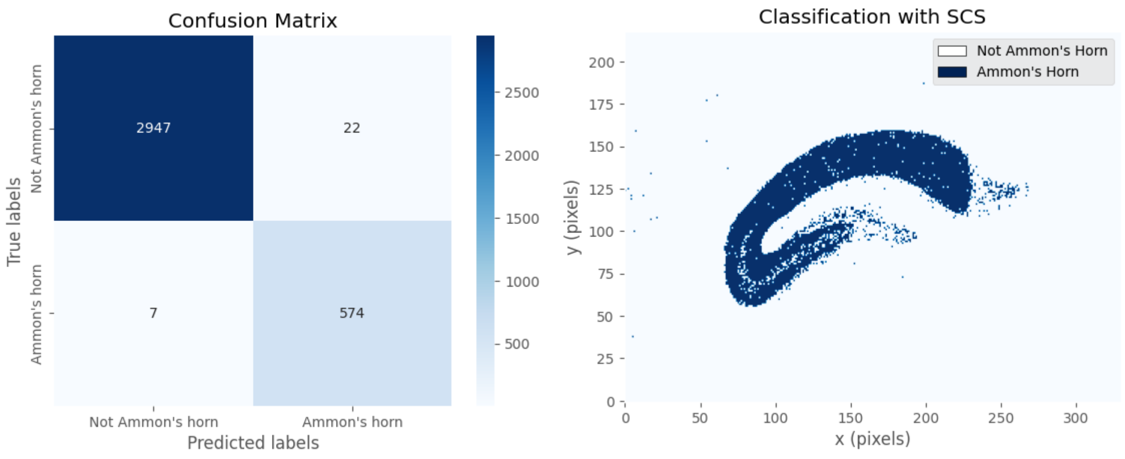


Figure 6-56: (Left) Confusion matrix for SCS-NN model (Right) Ammon's horn classification using SCS layers

From analysing the results in Figure 6-56, we can calculate the following performance metrics for the SCS-NN model in the task of Ammon's horn classification. The balanced accuracy is 0.9857, the precision is 0.9926, the recall is 0.9976 and the F1 score is 0.9951. Additionally, in Figure 6-56 (Right), we see that the SCS-NN model also correctly classifies the region of interest but introduces more mistaken pixels than the CNN.

CNN Model Interpretation

The results of interpreting the trained CNN model are presented in Figure 6-57, where we distinguish the most influential features to the model's output based on their SHAP scores

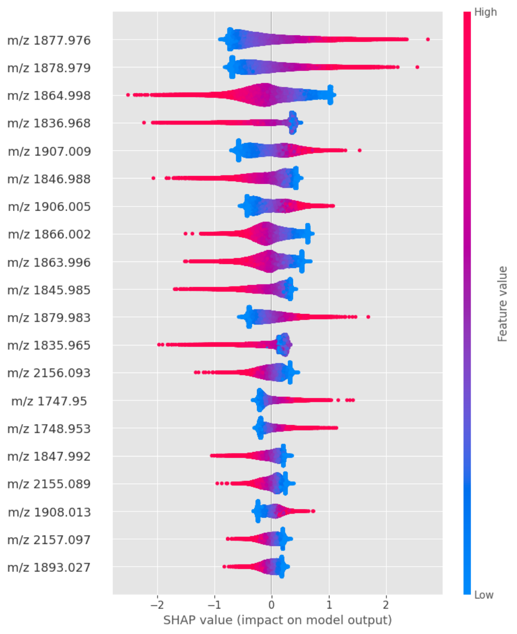


Figure 6-57: Impact on CNN model's output (SHAP values) in the Ammon's horn classification task of the most important features according to their value intensity

as well as their impact, positive or negative, depending on the feature's value, high or low. The three most important features for our 1D CNN model are m/z 1877.976, m/z 1878.979 and m/z 1864.998. According to the analysis of their impact based on Figure 6-57, for the first two features we get that when their intensity is high then their SHAP values are positive and thus, they have a positive impact on predicting the "Ammon's horn" class. On the other hand, for the feature m/z 1864.998 we get that when its value is high then its corresponding SHAP score is negative and thus, it reduces the probability of predicting the "Ammon's horn" class.

The ion images of the top-3 most important features for the CNN model decision making process are provided in Figure 6-58, Figure 6-59 and Figure 6-60. By inspecting those images we confirm the above results. For features m/z 1877.976 and m/z 1878.979, the ion intensity is high mainly in the region of Ammon's horn, which validates that their high value is positively correlated to the prediction of the region of interest. On the contrary, for the feature m/z 1864.998, the ion's intensity is mainly high in regions except for the Ammon's horn anatomical region. This validates the negative correlation of the high values of this features with the prediction of Ammon's horn region.

Additionally, we provide the SHAP maps of the features, meaning the corresponding SHAP values distributed per pixel, in Figure 6-58 (Right), Figure 6-59 (Right) and Figure 6-60 (Right). Again from those SHAP score illustrations we conclude that the features that are present on the Ammon's horn region have positive SHAP scores and contribute positively to the model's output, while the third most important feature contributes negatively to the output.

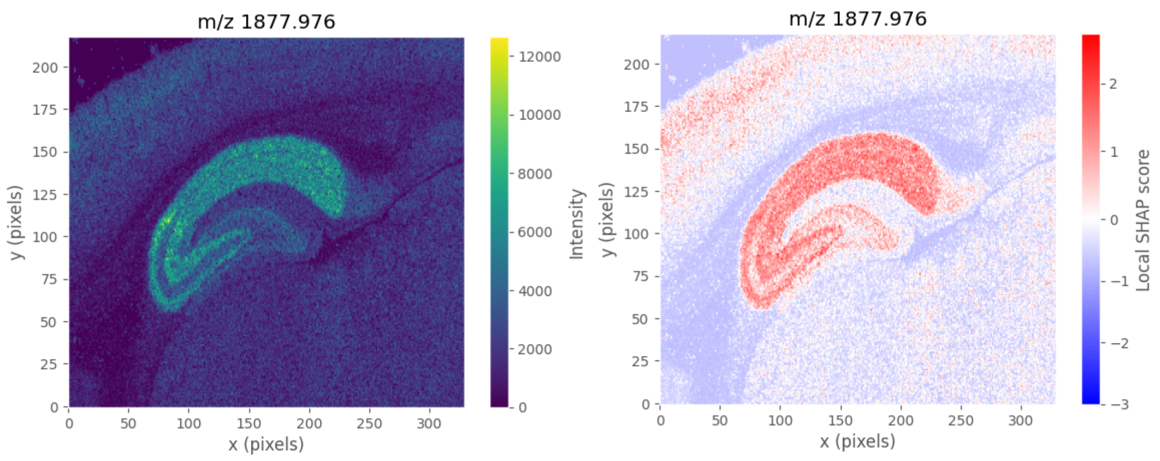


Figure 6-58: (Left) Ion image of m/z 1877.976, the most important feature (Right) SHAP score distribution across the pixels for the most important feature

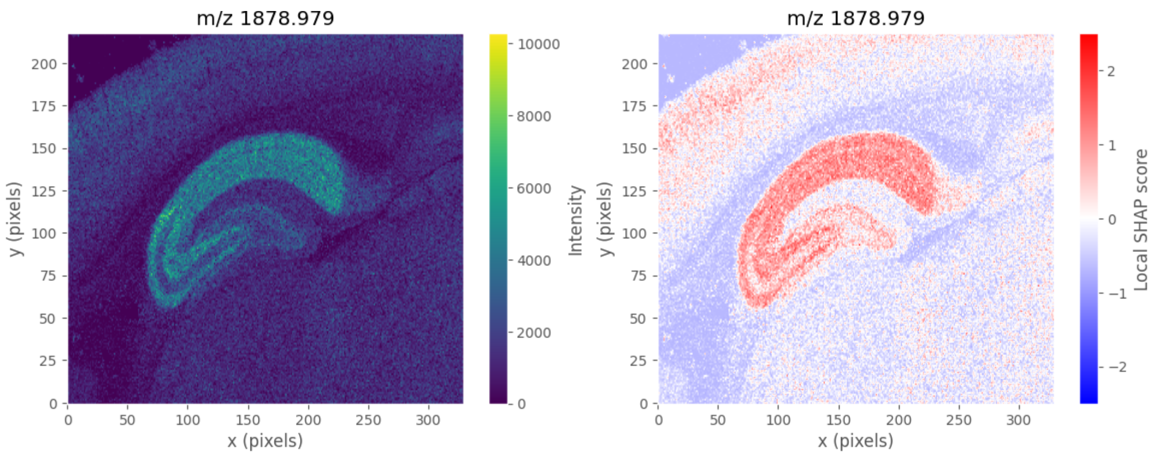


Figure 6-59: (Left) Ion image of m/z 1878.979, the second most important feature (Right) SHAP score distribution across the pixels for the second most important feature

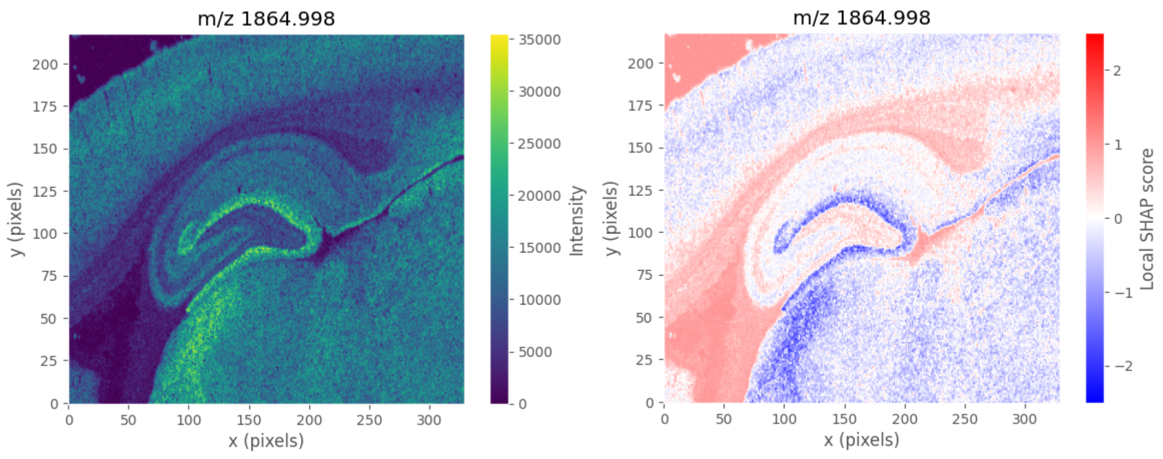


Figure 6-60: (Left) Ion image of m/z 1864.998, the third most important feature (Right) SHAP score distribution across the pixels for the third most important feature

Discussion

In this section, we oppose all the applied methodologies to each other and make several observations.

Table 6-14: Comparison between models for hippocampus Ammon's horn classification

Model	Balanced Accuracy	Precision	Recall	F1 score
Log.Regression	0.9912	0.9946	0.9959	0.9947
CNN	0.9931	0.9966	0.9980	0.9973
SCS-NN	0.9902	0.9926	0.9976	0.9951

Based on Table 6-17, the CNN model performs the best among the three models for the task of Ammon's horn classification. The CNN model achieves the highest balanced accuracy of 0.9931 and the highest average recall across the classes. The CNN model demonstrates a strong ability to correctly identify positive instances (high recall) while maintaining a low false-positive rate (high precision). On the other hand, SCS-NN model has the lowest precision, since it has the highest false-positive rate, meaning it mistakenly classifies actual "Not Ammon's horn" inputs as positive more than any other model. However, it has the second highest recall, meaning it does better in correctly classifying positive instances than the linear model.

A further comparison between the CNN and the SCS-NN model is presented in Table 6-15. From that table we once more conclude that a CNN and SCS-NN model can perform equally well while its architecture provides a simpler alternative.

Table 6-15: Comparison between CNN and SCS-NN models for renal cortex classification

Architecture	Trainable Parameters	Activation Layer	Normalization Layer	Train Duration (s)
CNN	30,706,322	ReLU	BN	1
SCS-NN	30,426,322	None	None	2.5

In Figure 6-61, the classification output of all the used classifiers on all pixel inputs is presented. The only pixels that haven't been used during the training and testing process and are considered "unseen" are the border pixels defined in the mask in Figure 6-49. Thus, we can conclude that the CNN model generalizes better than all the other models.

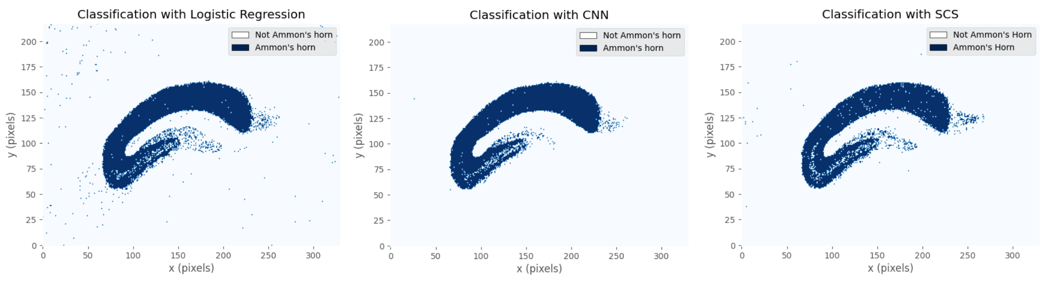


Figure 6-61: Mouse brain hippocampus Ammon's horn classification output on all available pixels with (Left) logistic regression (Center) CNN and (Right) SCS-NN model

6-3-3 Dentate Gyrus vs. Not Dentate Gyrus

Classification with a Linear Model - Logistic Regression

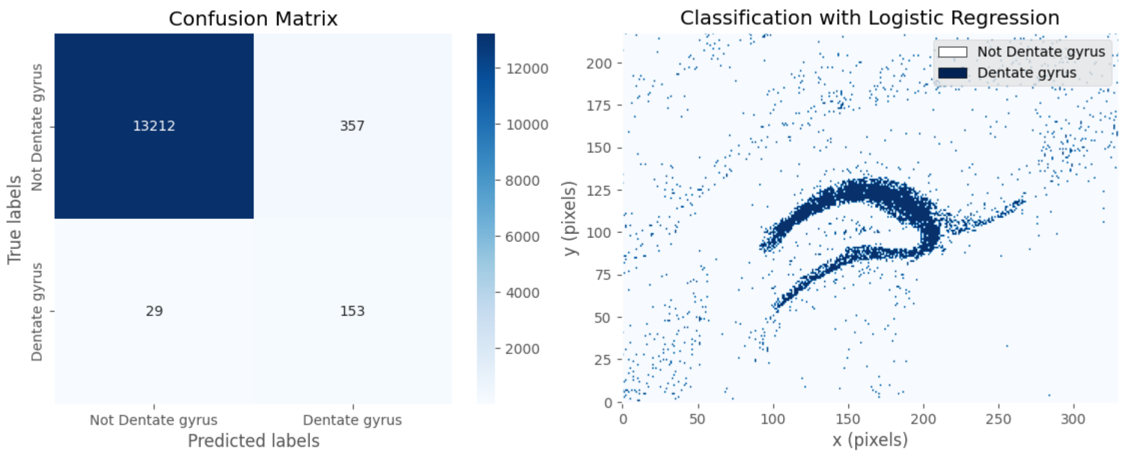


Figure 6-62: Dentate gyrus classification with logistic regression (Left) the corresponding confusion matrix for the test set (Right) the corresponding classification output for "unseen" and "seen" data

From the confusion matrix in Figure 6-62, we calculate the balanced accuracy as 0.9071, the precision as 0.3009, the recall as 0.8407 and the F1 score as 0.4435. The balanced accuracy is 0.9071 and emphasizes on the overall correct predictions. It takes into account both true positive rate and true negative rate. The precision is 0.3009, which suggests that out of all the instances predicted as positive, only 30% are actually true positives. Precision measures the accuracy of positive predictions and apparently, the low value suggests that there is a significant inefficiency in the positives correct prediction. The recall is relatively high, indicating that the model is effective at identifying positive instances but may miss some. The F1 score combines precision and recall, and in this case, it is low, indicating room for improvement in both precision and recall.

Furthermore, in Figure 6-62 (Right), we see the logistic regression model's output for the task of Dentate gyrus classification. We observe that there are a lot of noisy pixels that clearly don't belong to the Dentate gyrus region mistakenly classified as such. This is expected since from the confusion matrix we computed relatively low evaluation metrics for false positive rate and F1 score. Hence, it is obvious that the linear model performs quite bad, mainly due to the severely small amount of the positive labelled data, even though we reduce the majority class observations, for that reason the model is biased towards predicting the majority class and this bias can result in a higher number of false positive predictions, leading to a low precision.

Classification with a CNN

Table 6-16 presents the designed CNN architecture specifically tailored for Dentate gyrus classification. The table shows the various hyperparameters and their corresponding values used in the architecture. These hyperparameters are optimized using the TPE algorithm.

Table 6-16: Designed CNN architecture for Dentate gyrus classification.

Hyperparameters	**TPE Architecture**
Kernel Size	13
Stride	3
Pooling Kernel	2
Pooling Stride	5
Total Conv. layers	3
Stacked Conv. layers, 1	1
Stacked Conv. layers, 2	1
Stacked Conv. layers, 3	1
Stacked Conv. layers, 4	0
no. of Filters for 1^{st} Hidden layer	45
no. of FC layers	1
no. of Nodes for FC layer	800
LR Policy	'Step'
Exp. decay rate	non-applicable
γ	0.3
step size	5
Initial LR	0.00047
Batch Size	32
L2 Regularization	3e-06

The corresponding performance curves, namely the training loss and accuracy curves after 50 epochs are presented in Figure 6-63 as well as the training loss response to the learning rate decrease. It is evident that for learning rate values above 10^{-5} the training loss increases abruptly.

Moreover, from the confusion matrix in Figure 6-64 we calculate the metrics accuracy balance, 0.8668, precision, 0.7172, recall, 0.7376, F1 Score, 0.7273. In Figure 6-64, we see the classification output of the CNN model in the task of Dentate gyrus.

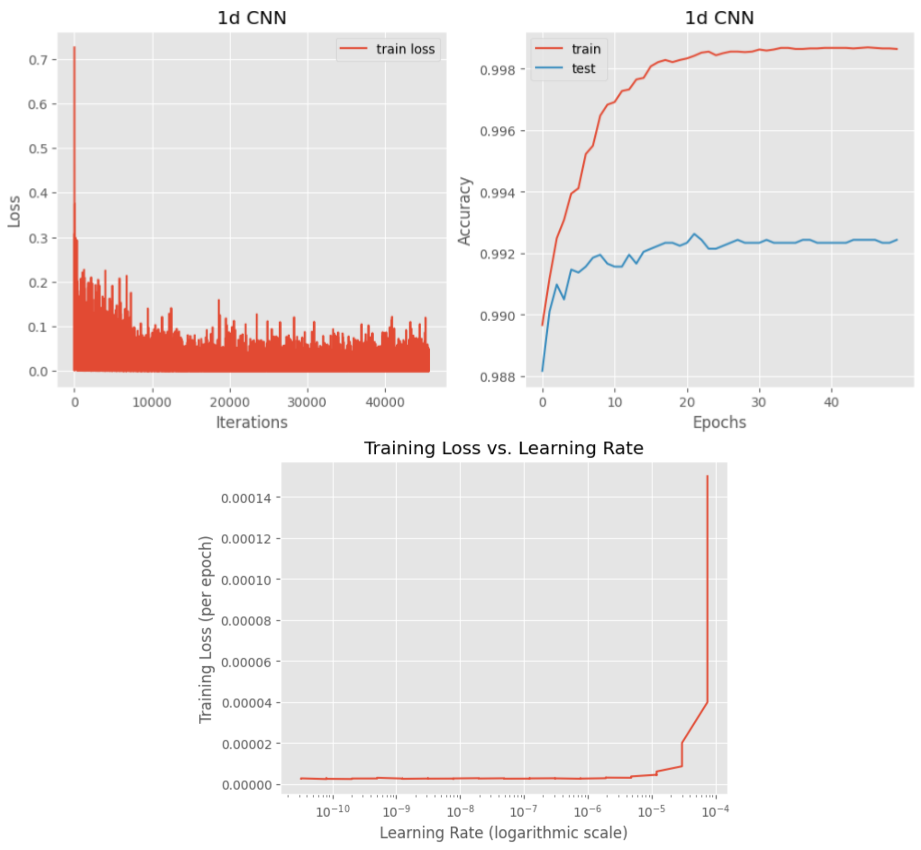


Figure 6-63: (Top) Performance curves for CNN model (Bottom) Learning rate relationship with training loss

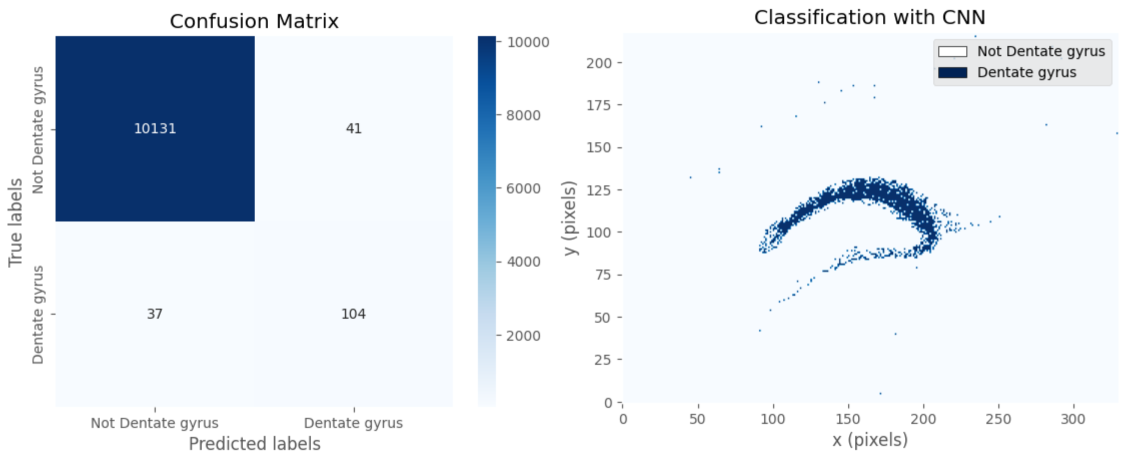


Figure 6-64: Dentate gyrus classification with CNN model (Left) Corresponding confusion matrix (Right) Classification output

Classification with a SCS-NN

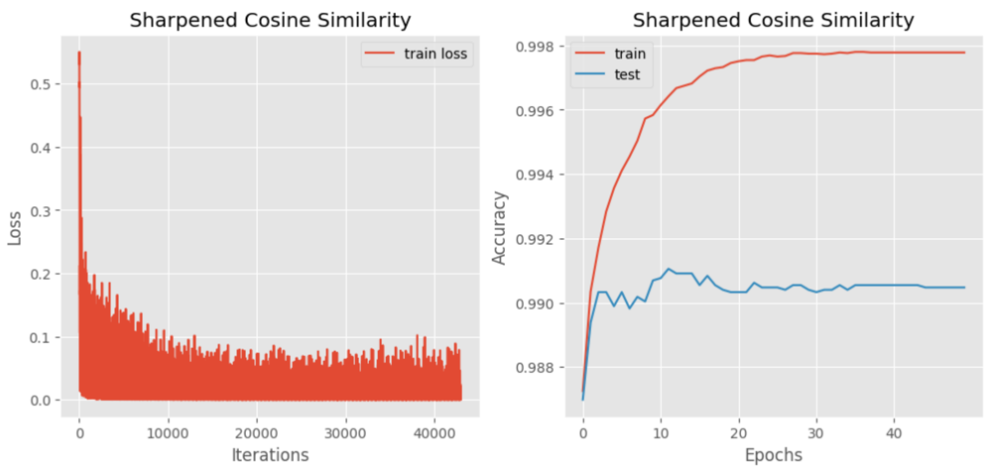


Figure 6-65: Training and Testing curves of the SCS-NN for the task of dentate gyrus classification

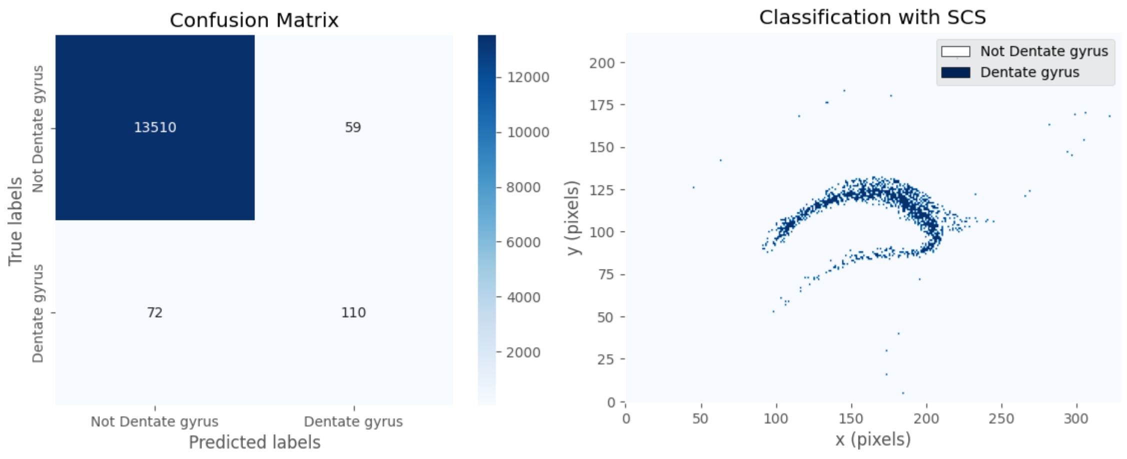


Figure 6-66: Dentate gyrus classification with SCS-NN model (Left) Corresponding confusion matrix (Right) Corresponding classification output

From the confusion matrix we calculate the balanced accuracy 0.80, the precision, 0.6508, the recall, 0.6043 and the F1 score 0.6265. Additionally, in Figure 6-66 (Right) we see the classification output result. The dentate gyrus region is correctly identified with a few misclassified pixels distributed across the rest of the area.

CNN Model Interpretation

In Figure 6-67, we see the features importances and their impact on the CNN model output for the task of Dentate gyrus prediction.

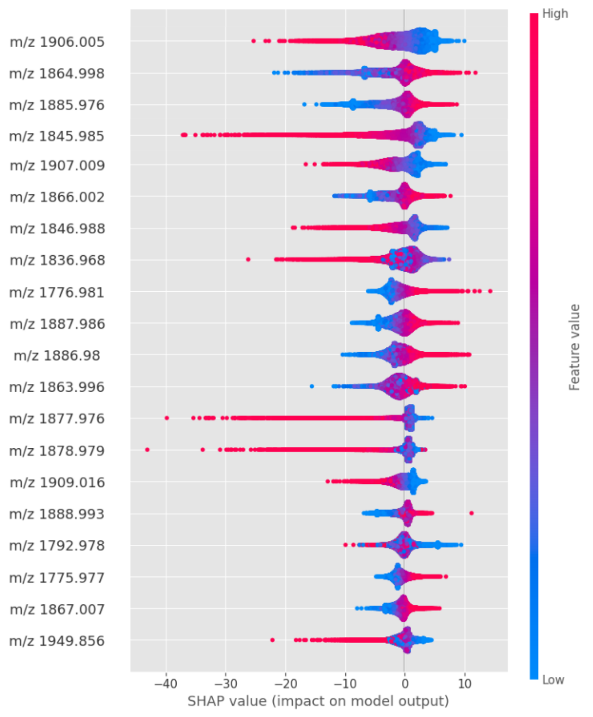


Figure 6-67: Impact of the most important features based on their SHAP values and their feature values

The three most important features sorted in descending order are m/z 1906.005, m/z 1864.998 and m/z 1886.976. According to the DeepSHAP results, the most important feature is contributing negatively to the prediction of Dentate gyrus when its value is high, whereas the second and the third feature contribute positively to the prediction of Dentate gyrus when their value is high. Their corresponding ion images are presented in Figure 6-68, where we observe that indeed, the m/z 1906.005 feature has a low intensity around the Dentate gyrus area thus, it is confirmed that it is negatively correlated to the model's prediction when its intensity is high. In Figure 6-69 and Figure 6-70, the ions have high intensities in the region of Dentate gyrus, which validates their positive correlation to the model's prediction. Moreover, within the provided SHAP maps, it can be observed that the Dentate gyrus region exhibits positive SHAP values. This indicates that the model relies on these particular features to increase the probability of predicting Dentate gyrus.

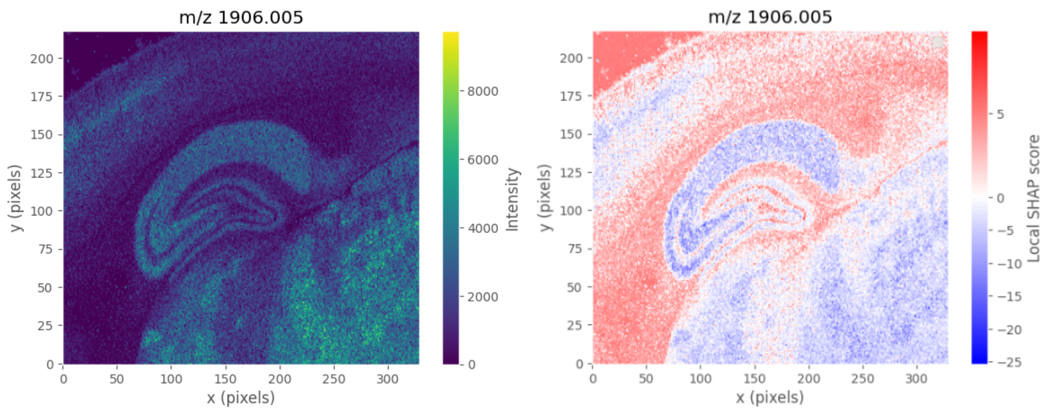


Figure 6-68: (Left) Ion image of m/z 1906.005 (Right) SHAP map across all pixels of the most important feature m/z 1906.005

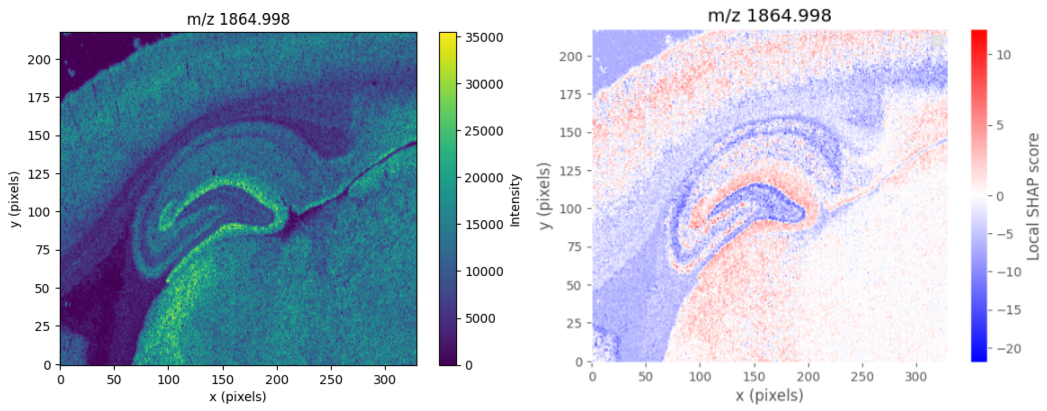


Figure 6-69: (Left) Ion image of m/z 1864.998 (Right) SHAP map across all pixels of the second most important feature m/z 1864.998

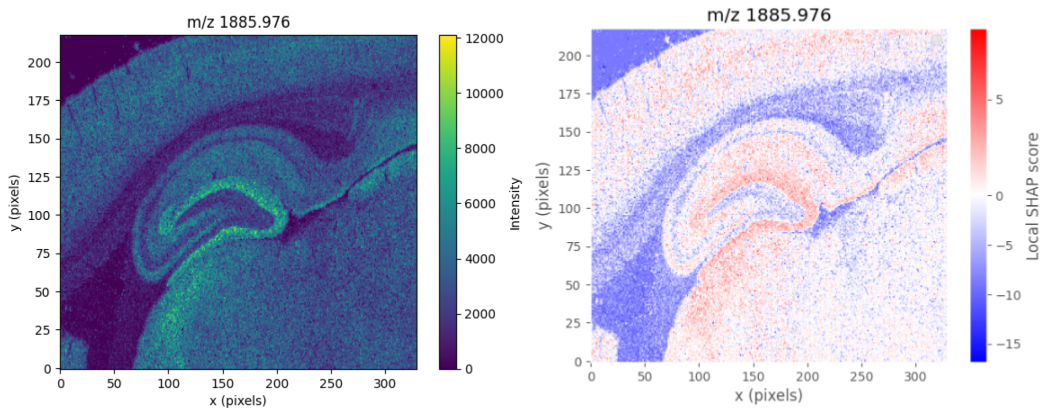


Figure 6-70: (Left) Ion image of m/z 1885.976 (Right) SHAP map across all pixels of the third most important feature m/z 1885.976

Discussion

First of all, we would like to mention that the above dataset involved having severe class imbalance. Therefore, in the beginning we attempted to use a combination of class balancing by using both majority class downsampling and minority class oversampling. For the minority class oversampling a popular resampling technique was used named Synthetic Minority Over-sampling Technique (SMOTE).

SMOTE is a technique that basically over-samples the minority class, used in machine learning, particularly in imbalanced classification tasks [10]. SMOTE generates synthetic examples rather than replicating existing examples. More specifically, for each minority class sample, the algorithm identifies its k nearest neighbors, then, from the k nearest neighbors, a specified number of neighbors, depending on the desired amount of over-sampling, are randomly selected. Once the neighbors are selected, the synthetic generation is accomplished by computing the difference of the feature vector of the current minority class sample and the corresponding nearest neighbor. Next, this difference is multiplied by a randomly produced value in the range of (0,1), and then, this scaled difference is added to the feature vector of the current minority class sample. With this procedure, synthetic/random datapoints are selected across the line segments connecting the minority class sample and its chosen neighbors and minority class over-sampling is achieved.

The results occuring from balancing the classes using both SMOTE and downsampling are the following.

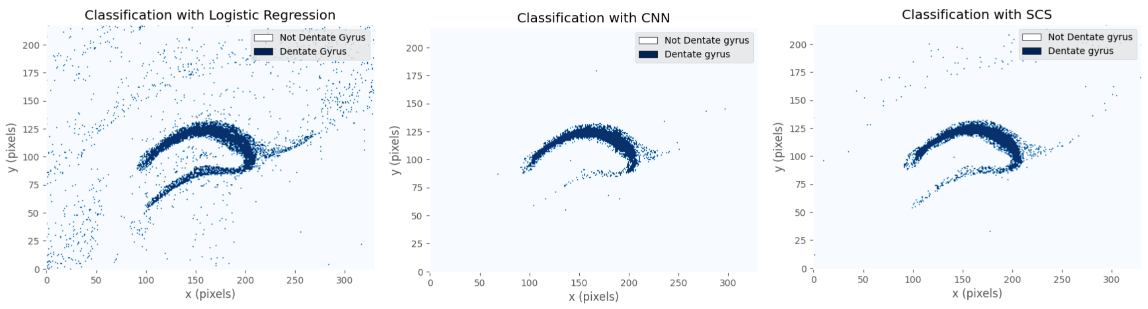


Figure 6-71: Classification results using data that has been resampled both by downsampling the majority class and by oversampling with SMOTE the minority class (Left) logistic regression model (Center) CNN model (Right) SCS-NN model

The linear model doesn't show any improvement after the applied data balancing technique. Additionally, by opposing the results in Figure 6-71 with the defined labelled mask for the classification problem in Figure 6-50, we see that the classification output seems to be overfitting the labelled mask especially for the CNN model case, whereas the SCS-NN model seems to generalize better and manage to classify the dentate gyrus region however, it introduces noisy pixels. In conclusion, we suggest that the severe class imbalance in this case study is not sufficiently dealt with SMOTE data augmentation, especially for the CNN model. On the other hand, the SCS-NN model output demonstrates a better ability in generalization and less overfitting. Nevertheless, next we compare the results we acquired solely from applying random majority class undersampling.

Table 6-17: Comparison between models for hippocampus Dentate gyrus classification

Model	Balanced Accuracy	Precision	Recall	F1 score
Log.Regression	0.9071	0.3009	0.8407	0.4435
CNN	0.8668	0.7172	0.7376	0.7273
SCS-NN	0.8001	0.6508	0.6043	0.6265

Moreover, we present the comparison between the applied classifiers in Table 6-17. The CNN model achieved a slightly lower balanced accuracy (0.8668) compared to logistic regression. However, its precision (0.7172), which represents the proportion of correctly predicted positive instances, indicates a much better performance in Dentate gyrus classification compared to the much lower logistic regression precision (0.3009). CNN's F1 score, which demonstrates a measure of balanced performance, is the highest among the other models. The SCS-NN model achieves similar but lower performance.

Table 6-18: Comparison between CNN and SCS-NN models for Dentate gyrus classification

Architecture	Trainable Parameters	Activation Layer	Normalization Layer	Train Duration (s)
CNN	6,673,182	ReLU	BN	3.6
SCS-NN	6,672,902	None	None	7.3

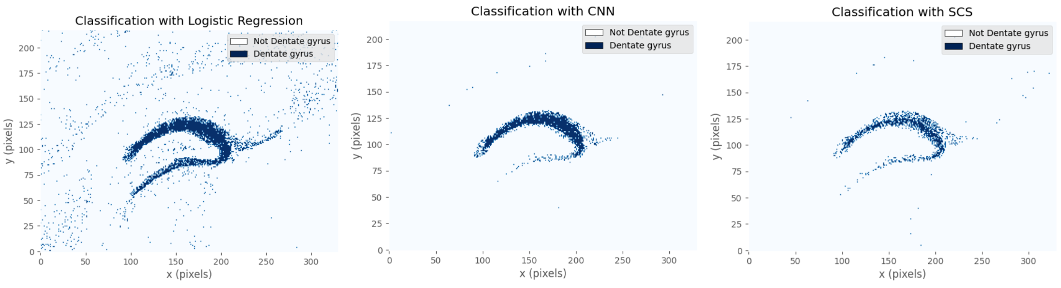


Figure 6-72: Mouse brain hippocampus Dentate gyrus classification output on all available pixels with (Left) logistic regression (Center) CNN and (Right) SCS-NN model

In Figure 6-72, we see the trained classifiers' output on both used and unused data during training and testing. It is evident that the linear model, logistic regression, is very noisy and doesn't achieve good results in classifying clearly the Dentate gyrus region. This is expected since the precision of the model was just 30%. Next, the CNN model almost eliminates the noisy pixels and detects the Dentate gyrus region, however, still the classification output is not clear and robust. It seems that the class imbalance hasn't been treated effectively and it affects the model's performance. The SCS-NN models provides a slightly more noisy output than the CNN but in general, similar to the CNN model, given that it involves less complex architecture (see Table 6-18).

Chapter 7

Conclusions and Future Work

In this section, we reflect upon the significant findings and outcomes of this research. We also discuss the limitations encountered during the implementation process and propose future directions for further extension of this work.

7-1 Overview of Research

In the context of constructing a Convolutional Neural Network (CNN) model for Imaging Mass Spectrometry (IMS) data classification, the optimal hyperparameter selection plays an inherently significant role in determining the model's accuracy and poses a vital challenge. This thesis presents the development and application of a comprehensive and effective pipeline for the design and interpretation of 1-D CNN models customized specifically for classifying Imaging Mass Spectrometry (IMS) data. The architecture of the CNN models was generated using an automated approach, namely the Tree-structured Parzen Estimator (TPE) algorithm. We implemented the TPE algorithm within the Optuna library coupled with PyTorch. After generating optimal architectures based on TPE, we used k-fold cross validation to select the best model. Then we trained and tested the designed model and evaluated the corresponding results based on performance metrics but also conducted comparative analysis against baseline linear methods, such as Linear Discriminant Analysis (LDA) and logistic regression. Furthermore, we explored a novel alternative to traditional convolution for feature extraction known as Sharpened Cosine Similarity (SCS), which was implemented and evaluated in all our case studies. Additionally, we adapted the implementation example of SCS in [2] to 1-D input structures, making the first application of SCS layers on IMS data, to the best of our knowledge.

Determining the CNN's reliability is another challenge in developing CNN models. Due to their complexity, CNNs are considered black-boxes. Consequently, in this thesis we have also implemented a comprehensive interpretation pipeline for the classification results of the CNN models in every case study, supported by helpful visualizations of the features' importance in the spatial dimension. With the combination of the interpretability framework and the corresponding CNN model accuracy findings we attempted to extract a reliable conclusion regarding the suitability and effectiveness of the designed CNN models. Through this attempt we aimed to enhance the trustworthiness of the CNNs predictions while achieving satisfactory performance accuracy.

7-2 Conclusions

From this research, there are several conclusions that can be drawn regarding various topics we explored within this work. Conclusions can be drawn regarding data preprocessing effects, architecture design, and the effectiveness of different approaches. Next we present the conclusions derived from this work:

Data Although data scaling is a recommended preprocessing step in Machine Learning (ML) algorithms in order to address significant variations in feature ranges and enhance performance, when applied to IMS data, where the ion intensities range from zero to thousands, it led to deteriorated results and thus, we don't suggest using it. On the other hand, we support that indeed data centering prior to training enhances the classification accuracy. Additionally, using Synthetic Minority Over-sampling Technique (SMOTE) as a data augmentation method led to overfitting the data and therefore, reduced the ability of the model to generalize well for "unseen" instances. For this reason, we suggest majority class downsampling as a basic method to compensate for minor class imbalances but we suggest that more data augmentation techniques should be explored.

Hyperparameter Optimization From our results, we conclude that using the automated hyperparameter optimization approach - TPE - is very effective and manages to explore and exploit well the hyperparameter search space to generate optimal CNN architectures requiring less time and effort. In this way, we can avoid the need for manually tailoring the model design based on biochemical and biological domain expertise, which would be cumbersome and expensive. On the other hand, we might lose some insight of the model decision making process that is connected to biochemical characteristics of the data. Moreover, multiple architectures with similar performances are generated by the algorithm for different random seeds due to its probabilistic nature. We suggest k-fold cross validation for the selection of the best model. Additionally, we conclude that TPE as a Bayesian method has the limitation that it can potentially have difficulties in escaping from local optima to reach the global optima. In order to face this limitation, we restart the algorithm several times so that it starts from different initial observations.

CNNs for IMS data Based on our experimental results, we conclude that CNNs outperform all other linear and SCS-NN models used for IMS data classification. Furthermore, the most significant contribution of a CNN model in accuracy compared to a linear model was an improvement of 8%. However, in some case studies there was no significant performance improvement by CNNs. This is mainly due to the limitation that the linear models already achieved a high accuracy for the given datasets and there was not much room for improvement.

In general, the linear models assume a linear relationship between the input features and the target, hence, when having inputs with more noise the linear methods struggle to model the complex data patterns. The CNN models use convolutions and non-linear activations, which allow them to model complex relationships between the input features and this leads to their superior results. CNNs capture non-linear dependencies in the m/z axis of IMS data, which are critical for accurately distinguishing between different classes. An additional conclusion is that CNNs learn IMS data features automatically in a hierarchical manner without the requirement of manual feature engineering prior to training. On the other hand, a negative aspect of CNNs is that they require GPU access for model training and introduce the need for model interpretation since they are not transparent.

Model Interpretation Through this study, the interpretation results show that most of our CNN models based their outputs on relevant inputs. However, there were cases where noisy inputs also had an impact on the model's output. This shows that it is important to debug the models in order for them to focus on reliable inputs. However, in this thesis we haven't applied any form of model debugging. Nevertheless, we conclude that interpretation of black box models is important and can highlight model weaknesses and help to resolve bugs to improve reliability.

SCS layers vs. Convolutional layers In this thesis, we explored the replacement of convolutional layers by Sharpened Cosine Similarity (SCS) layers for feature extraction. The SCS-NN model achieves less but comparable accuracy to the CNN model with the following advantages. It has a simpler architecture - no use of ReLU activation layers nor Batch Normalization layers. It involves less trainable parameters and requires less memory for training. On the contrary, one training epoch of such a model takes longer than the CNN, most likely because the mathematical expression of SCS requires the raise into the p^{th} power, which causes an increase in the computational time. We could conclude that SCS layers offer a good alternative over convolutional layers in terms of a less complex architecture or less memory demand preference, however, they don't yield any accuracy improvement, as far as our results demonstrated. Finally, in the case of data augmentation with SMOTE, we observed that the SCS-NN model demonstrated less data overfitting compared to the CNN model.

7-3 Future Directions

Based on the limitations and the outcomes of our research in this thesis we can define the following future work directions.

- *Investigation of alternative data augmentation techniques.* These techniques should aim to improve the generalizability of the models without introducing excessive noise or artificial patterns. An example could be the implementation of Generative Adversarial Networks (GANs), mainly applied for image data augmentation.
- *Model debugging for improved input reliability.* A method for detecting and removing the non-reliable, noisy inputs should be developed. For example, the noisy inputs could

be identified by using the Spearman correlation coefficient and then removed from the feature space.

- *Investigate using 2-D convolutional layers.* If we treat the m/z axis of the IMS datacube as the channels dimension in the CNNs framework and the (x, y) pixel positions as the height and width dimension of a 2-D convolutional layer, we can use TPE to design 2-D CNN architectures and evaluate their performance in comparison to the 1-D ones. Moreover, the 2-D CNN model could again be opposed and compared to a corresponding 2-D SCS-NN model.
- *Compare alternative Deep Learning (DL) techniques to CNNs for IMS data.* Since we prove that CNNs are undoubtedly an advantageous DL classification method for IMS data it would be interesting to see how well other DL approaches perform, such as transformers, which were initially developed for sequential data. Therefore, we propose a comparative analysis between an automatically designed CNN architecture and an alternative approach, such as transformers.

Appendix A

Automated CNN Architecture Design

Here, we present additional material for the automated architecture selection process.

A-1 Hyperparameter Importances

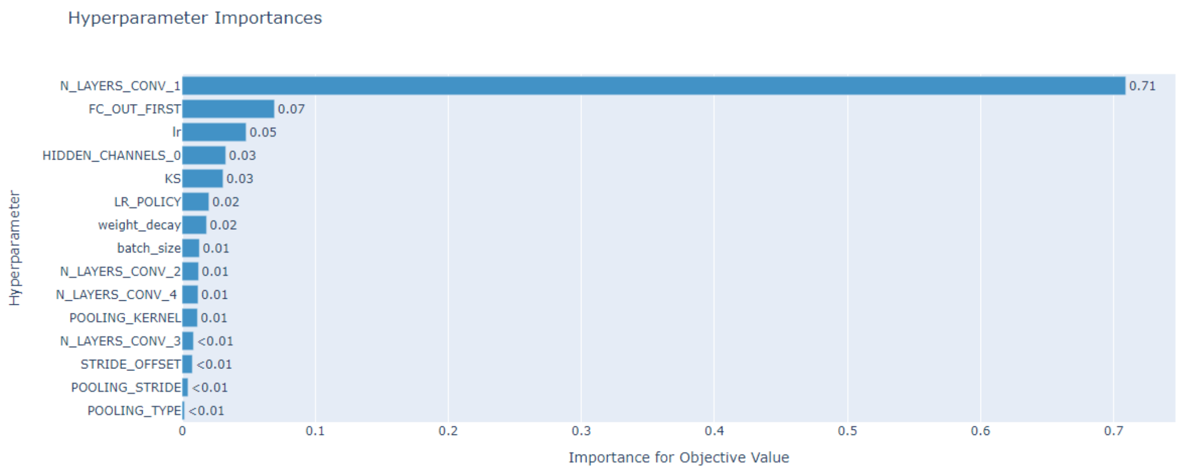


Figure A-1: Hyperparameter importances on a different run illustrating minimum importance for the pooling type.

A-2 Hyperparameter Optimization with Optuna and PyTorch

Here we present a toy example of how to use Optuna in Python to generate the architecture of a Convolutional Neural Network (CNN) model defined using PyTorch. The training and testing code snippet is missing because we want to focus on the definition of the search space with Optuna.

```python
import optuna

def objective(trial):

    # Number of convolutional layers per stack
    N_LAYERS_CONV_1 = trial.suggest_int("N_LAYERS_CONV_1 ", 1, 5, 1)
    N_LAYERS_CONV_2 =trial.suggest_int("N_LAYERS_CONV_2 ", 0, 5, 1)

    # Number of filters for the convolutional layers
    HIDDEN_CHANNELS_0 = trial.suggest_int("HIDDEN_CHANNELS_0", 5, 60, 5)

    # Number of nodes of the output of the first fully connected layer
    FC_OUT_FIRST = trial.suggest_int("FC_OUT_FIRST", 100, 1000, 100)

    # Number of fully connected layers
    N_LAYERS_FC = trial.suggest_int("N_LAYERS_FC", 1, 3, 1)

    # Batch size
    BATCH_SIZE = trial.suggest_categorical("batch_size", [32, 64, 128, 256])

    # Learning rate and policies
    LR_POLICY = trial.suggest_categorical('LR_POLICY', ['Fixed','Step', 'Exp', 'Cyclic'])
    lr = trial.suggest_float("lr", 1e-6, 1e-1, log=True)

    # L2 regularization
    weight_decay = trial.suggest_float("weight_decay", 1e-7, 1e-2, log=True)

    # Set the model
    model = TPENet(
        trial, IN_CHANNELS, WIDTH, N_LAYERS_CONV_1, N_LAYERS_CONV_2, N_LAYERS_FC, HIDDEN_CHANNELS_0, FC_OUT_FIRST, DEVICE, OUT_FEATURES).to(DEVICE)

    optimizer = torch.optim.Adam(model.parameters(), lr=lr, weight_decay=weight_decay )

    if LR_POLICY == 'Cyclic':
      scheduler = OneCycleLR(
        optimizer,
        max_lr=lr,
        steps_per_epoch=3846,
        epochs=1)
    elif LR_POLICY == 'Exp':
      decay_rate = trial.suggest_float('decay_rate', 0.0, 1.0)
      scheduler = ExponentialLR(optimizer, gamma=1-decay_rate)
    elif LR_POLICY == 'Step':
      gamma = trial.suggest_float("gamma", 0.1, 0.9, log=True)
      step_size = trial.suggest_int("step_size", 1, 10)
      scheduler = StepLR(optimizer, step_size=step_size, gamma=gamma)

    #Training and testing of the model
    for epoch in range(N_EPOCHS):
        ###
        # The code for training and testing
        ###

study = optuna.create_study( study_name='TPE', direction='maximize')
study.optimize(objective, n_trials=NUMBER_OF_TRIALS)

trial = study.best_trial

print('F1 score: {}'.format(trial.value))
print("Best hyperparameters: {}".format(trial.params))
```

During the trial started in Optuna, we need to define seperately the model, namely *TPENet*, shown below. The model requires the computation of the layers output size in a dynamical way. For the dynamical calculation of the output sizes per convolutional layer we use the formula from [12]. Thus, the output size, O, of a convolutional layer with input, I, $\forall$ kernel size (KS), stride (S) and padding (P) is computed as:

$$O = \lfloor \frac{I + 2P - KS}{S} \rfloor + 1 \tag{A-1}$$

in addition, the output after any pooling layer, since it doesn't involve any padding is calculated as [12]:

$$O = \lfloor \frac{I - KS}{S} \rfloor + 1 \tag{A-2}$$

when the requirement is to have the same padding, we can calculate the padding as, such that $O = I$:

$$Equation\ A-1 \Rightarrow P = \frac{I(S-1) + KS - S}{2} \tag{A-3}$$

hence, when we have unit strides the full padding is simplified to $P = \frac{KS-1}{2}$.

```python
"""
Convolutional  Neural  Network (CNN)  with  hyperparameter  values
obtained from the Tree-structured Parzen Estimator (TPE) algorithm.
"""

import numpy as np
import torch
import torch.nn as nn


class TPENet(nn.Module):
    """
    CNN optimized using TPE for IMS data

    Args
    -----
    trial: Optuna trial object used for hyperparameter optimization.
    IN_CHANNELS:   Number      of      channels    of     the  input.
    WIDTH:      Number    of    peaks    in    the    input    spectrum.
    N_LAYERS_CONV_1: Number of convolutional layers for the  first block.
    N_LAYERS_CONV_2: Number of convolutional layers for the second block.
    N_LAYERS_FC: Number of fully connected layers.
    HIDDEN_CHANNELS_0: Number of filters for the first convolutional
       layer.
    FC_OUT_FIRST: Output size of the first fully connected layer.
    DEVICE: Device used for training the model.
    OUT_FEATURES: Number of output features/classes.

    Methods
    -------
    forward(x): forward pass
    """

    def __init__(
        self,
        trial,
        IN_CHANNELS,
        WIDTH,
        N_LAYERS_CONV_1,
        N_LAYERS_CONV_2,
        N_LAYERS_FC,
        HIDDEN_CHANNELS_0,
        FC_OUT_FIRST,
        DEVICE,
        OUT_FEATURES,
    ):
        super(TPENet, self).__init__()

        # Definition of hyperparameter search space
        KS = trial.suggest_int("KS", 3, 20, 1)
        STRIDE_OFFSET = trial.suggest_int("STRIDE_OFFSET", 1, 4, 1)
        POOLING_KERNEL = trial.suggest_int("POOLING_KERNEL", 2, 5, 1)
        POOLING_STRIDE = trial.suggest_int("POOLING_STRIDE", 1, 4, 1)

        # Initialize
        self.N_LAYERS_CONV_1 = N_LAYERS_CONV_1
        self.N_LAYERS_CONV_2 = N_LAYERS_CONV_2

        self.convs1 = nn.ModuleList([])
        self.convs2 = nn.ModuleList([])

        self.fc_layers = nn.ModuleList([])

        filters = []
        output_size = []
        out_features = []
        padding = []

        ###
        # 1st stack of convolutional layers
        ###
        output_size.append(WIDTH)
        padding.append(
            int((output_size[-1] * (STRIDE_OFFSET - 1) + KS -
                STRIDE_OFFSET) / 2)
        )
        filters.append(IN_CHANNELS)
        for layer in range(N_LAYERS_CONV_1):
            self.convs1.append(
                nn.Conv1d(
                    in_channels=filters[-1],
                    out_channels=HIDDEN_CHANNELS_0,
                    kernel_size=KS,
                    stride=STRIDE_OFFSET,
                    padding=padding[-1],
                )
            )

            filters.append(HIDDEN_CHANNELS_0)

            self.convs1_bn = nn.BatchNorm1d(HIDDEN_CHANNELS_0)

            output_size.append(
                int(
                    np.floor((output_size[-1] + 2 * padding[-1] - KS) /
                        STRIDE_OFFSET)
                    + 1
                )
            )

            padding.append(
                int((output_size[-1] * (STRIDE_OFFSET - 1) + KS -
                    STRIDE_OFFSET) / 2)
            )

        if N_LAYERS_CONV_2 != 0:
            # 1st pooling layer output size
            output_size.append(
                int(np.floor(((output_size[-1] - POOLING_KERNEL) /
                    POOLING_STRIDE) + 1))
            )

            padding.append(
                int((output_size[-1] * (STRIDE_OFFSET - 1) + KS -
                    STRIDE_OFFSET) / 2)
            )

            ###
            # 2nd stack of convolutional layers output size
            ###
            for layer in range(N_LAYERS_CONV_2):
                self.convs2.append(
                    nn.Conv1d(
                        in_channels=filters[-1],
                        out_channels=2 * HIDDEN_CHANNELS_0,
                        kernel_size=KS,
                        stride=STRIDE_OFFSET,
                        padding=padding[-1],
                    )
                )

                filters.append(2 * HIDDEN_CHANNELS_0)

                self.convs2_bn = nn.BatchNorm1d(2 * HIDDEN_CHANNELS_0)

                output_size.append(
                    int(
                        np.floor(
                            (output_size[-1] + 2 * padding[-1] - KS) / STRIDE_OFFSET
                        )
                        + 1
                    )
                )

                padding.append(
                    int(
                        (output_size[-1] * (STRIDE_OFFSET - 1) + KS - STRIDE_OFFSET) / 2
                    )
                )

        # ReLU activation layer
        self.relu = nn.ReLU()

        # Dropout layer
        self.do = nn.Dropout()

        # Pooling layer
        self.pool = nn.AvgPool1d(kernel_size=POOLING_KERNEL, stride=POOLING_STRIDE)

        # Getting size of vectorized data
        output_vectorized = filters[-1] * output_size[-1]

        # Fully connected layers
        out_features.append(FC_OUT_FIRST)
        for i in range(N_LAYERS_FC):
            self.fc_layers.append(
                torch.nn.Linear(output_vectorized, out_features[-1]).to(DEVICE)
            )
            output_vectorized = out_features[-1]
            out_features.append(out_features[-1])

        self.fcX = nn.Linear(output_vectorized, OUT_FEATURES).to(DEVICE)

    def forward(self, x):
        """
        Performs the forward pass of the network by successively
        connecting  the  inputs  and  the  outputs  based  on
        the  layer architecture.
        """
        for i, conv_layer_i in enumerate(self.convs1):
            x = conv_layer_i(x)
            x = self.convs1_bn(x)
            x = self.relu(x)
        if self.N_LAYERS_CONV_2 != 0:
            x = self.pool(x)
            for i, conv_layer_i in enumerate(self.convs2):
                x = conv_layer_i(x)
                x = self.convs2_bn(x)
                x = self.relu(x)

        # Flatten
        x = x.view(x.size(0), -1)

        for i, fc_layer_i in enumerate(self.fc_layers):
            x = fc_layer_i(x)
            x = self.relu(x)

        x = self.fcX(x)

        return x
```

A-3 Designed CNN Architectures

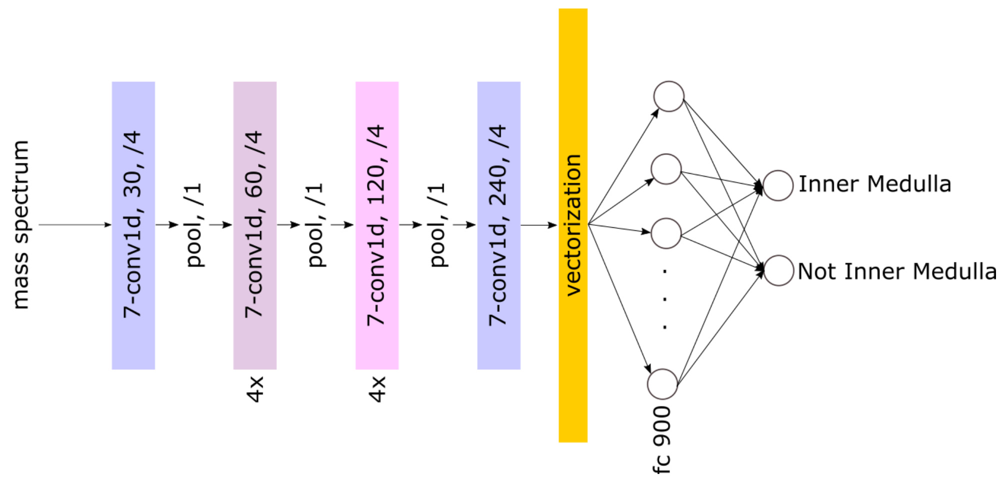


Figure A-2: Illustration of CNN architecture for renal inner medulla classification

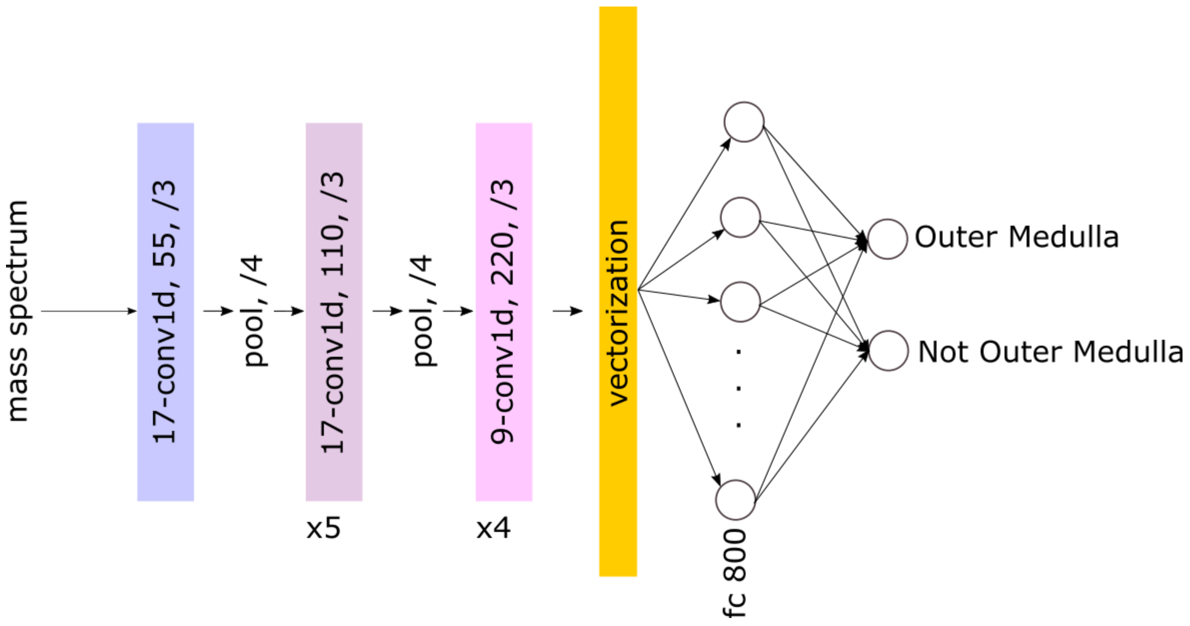


Figure A-3: Illustration of CNN architecture for renal outer medulla classification

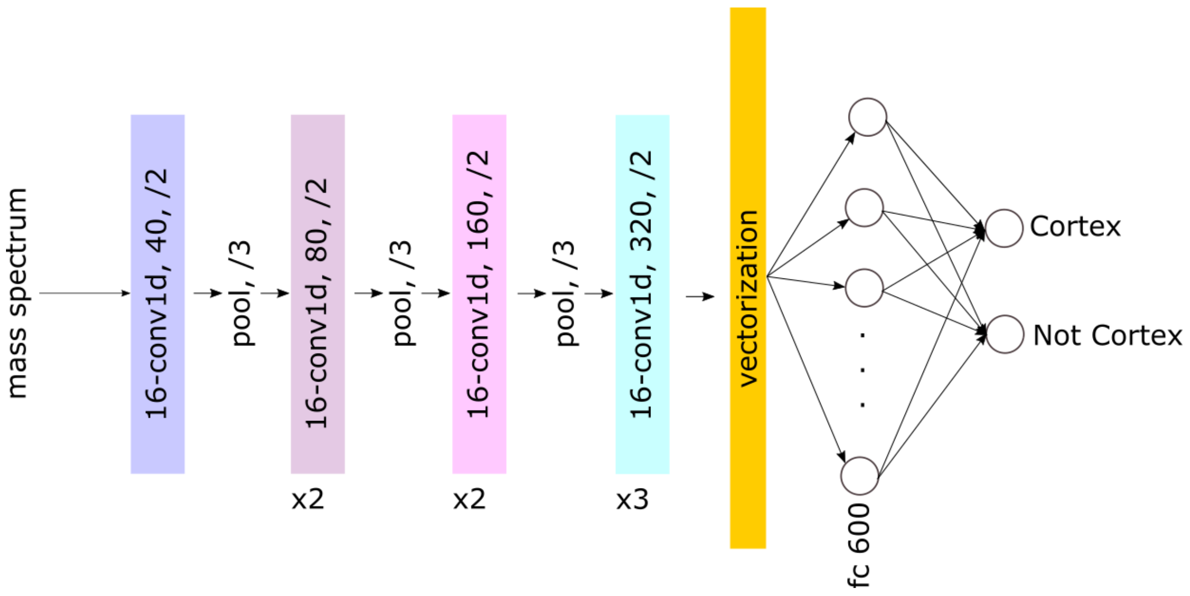


Figure A-4: Illustration of CNN architecture for renal cortex classification

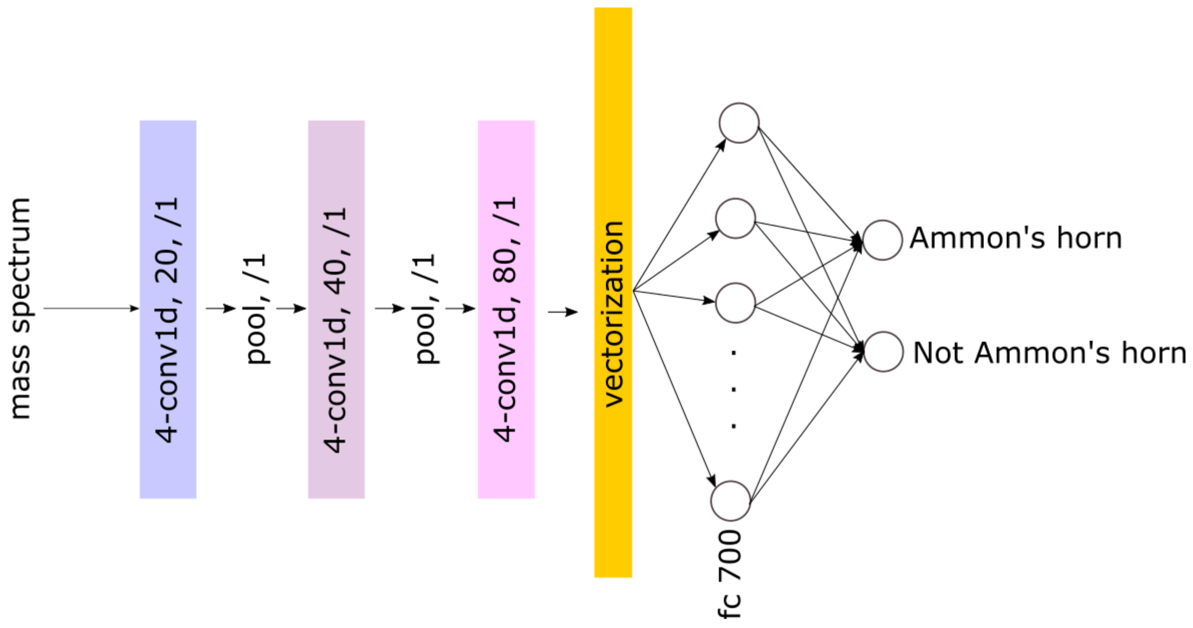


Figure A-5: Illustration of CNN architecture for Ammon's horn of mouse brain hippocampus classification

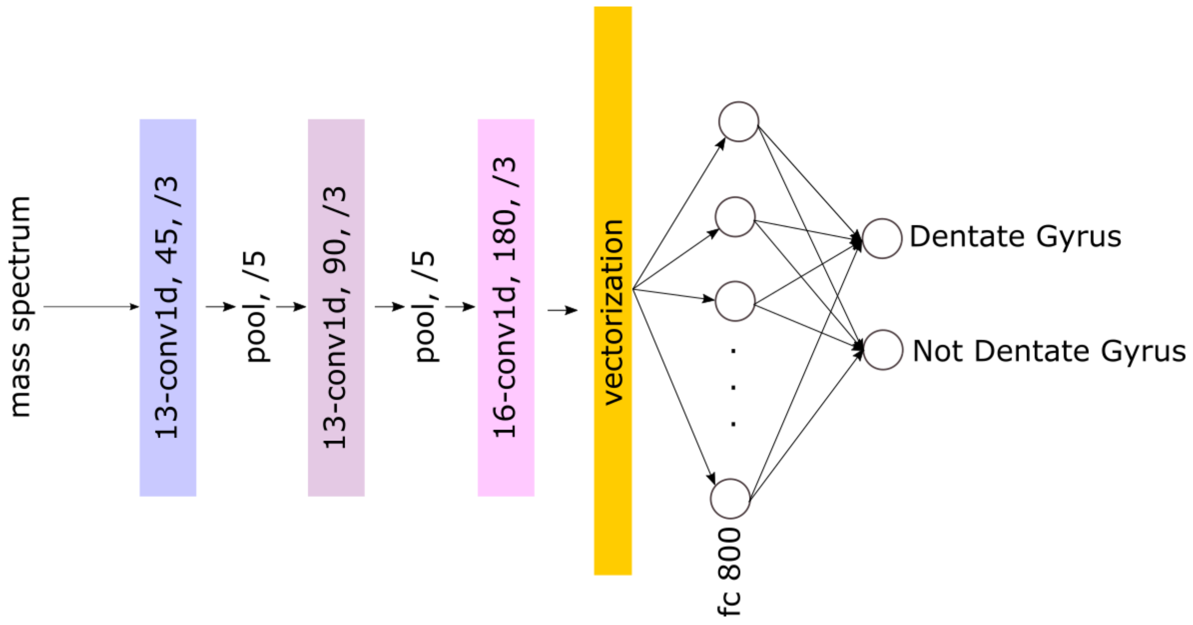


Figure A-6: Illustration of CNN architecture for Dentate gyrus of mouse brain hippocampus classification

Glossary

List of Acronyms

IMS	Imaging Mass Spectrometry
ML	Machine Learning
LDA	Linear Discriminant Analysis
RF	Random Forest
SVM	Support Vector Machine
DL	Deep Learning
SGD	Stochastic Gradient Descent
AI	Artificial Intelligence
MS	Mass Spectrometry
SIMS	Secondary Ion Mass Spectrometry
MALDI	Matrix-Assisted Laser Desorption Ionization
DESI	Desorption Electrospray Ionization
TOF	Time-of-flight
Q-TOF	Quadrupole Time-of-flight
FTICR	Fourier Transform Ion Cyclotron Resonance
CNN	Convolutional Neural Network
E2E	End-to-end learning
DNN	Deep Neural Network
MLP	Multilayer Perceptron
BN	Batch Normalization
LR	Learning Rate
SCS	Sharpened Cosine Similarity
SHAP	SHapley Additive exPlanations
SMOTE	Synthetic Minority Over-sampling Technique

FC	Fully Connected
SCS	Sharpened Cosine Similarity
TPE	Tree-structured Parzen Estimator
KDEs	Kernel Density Estimators
EI	Expected Improvement
ReLU	Rectified Linear Unit
DL	Deep Learning
Adam	Adaptive Moments
RMSProp	Root Mean Squared Propagation
CPU	Central Processing Unit
GPU	Graphics Processing Unit

List of Symbols

α	Momentum parameter that gives how fast gradients decay
ε	Learning rate
η	Hyperparameter observation
λ	Regularization parameter
ν	Number of classes
ω	Ion's angular velocity
Π	Set of permutations of the set of players
ψ	Loss for TPE
ρ	RMSProp hyperparameter
σ	Bandwidth in the kernel function of KDE
τ	Time step in Adam
ϑ	Model's weights and biases- trainable parameters
B	Magnetic field intensity
b	Bias
C	Total number of cooperating players
d	Gradient accumulation variable for Root Mean Squared Propagation (RMSProp)
e	Elementary charge of an electron
f	Ion's movement fequency
H	Configuration space
h	Hidden layer size
J	Regularized loss function
k	Number of subsets/folds the original set is divided into
L	Loss function
l	Separating distance of the ion source from the detector

M	Number of samples in the batch
m	Atomic mass of the ion
N	Total number of pixel positions/observations in IMS data
p	Sharpening factor
q	Small positive constant used to avoid numerical instability
R	Regularization term
r	Radius of circular trajectory of an ion in a magnetic field
s	Total number of m/z features
t	Truth value vector
U	Set of players in cooperative game theory
u	Velocity gained by the ion
V	Accelerating potential
v	Momentum parameter
w	Layer weight
X	Input in a deep learning model
x	Horizontal coordinate of the IMS data spatial dimension
y	Vertical coordinate of the IMS data spatial dimension
z	Number of ion charges
z'	Simplified inputs

Bibliography

[1] Interactive Atlas Viewer. http://atlas.brain-map.org/atlas?atlas=2&plate=100884125.

[2] Sharpened Cosine Similarity. https://github.com/brohrer/sharpened-cosine-similarity/blob/main/pytorch/sharpened_cosine_similarity.py.

[3] Theodore Alexandrov. MALDI imaging mass spectrometry: statistical data analysis and current computational challenges. *BMC Bioinformatics*, 13(S16):S11, November 2012.

[4] Jens Behrmann, Christian Etmann, Tobias Boskamp, Rita Casadonte, Jörg Kriegsmann, and Peter Maaβ. Deep learning for tumor classification in imaging mass spectrometry. *Bioinformatics*, 34(7):1215–1223, April 2018.

[5] Yoshua Bengio. Practical recommendations for gradient-based training of deep architectures, September 2012. arXiv:1206.5533 [cs].

[6] James Bergstra and Yoshua Bengio. Random Search for Hyper-Parameter Optimization. 2012.

[7] James Bergstra, Daniel Yamins, and David Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In Sanjoy Dasgupta and David McAllester, editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 115–123, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR.

[8] James S Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for Hyper-Parameter Optimization. 2011.

[9] Pierre Chaurand, Schwarz Sarah, Michelle L Reyzer, and Richard M. Caprioli. Imaging Mass Spectrometry: Principles and Potentials. 2005.

[10] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. SMOTE: Synthetic Minority Over-sampling Technique. *Journal of Artificial Intelligence Research*, 16:321–357, June 2002.

[11] Edmond de Hoffman and Vincent Stroobant. *Mass Spectrometry. Principles and Applications.* 3rd edition, 2007.

[12] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning, January 2018. arXiv:1603.07285 [cs, stat].

[13] Manuel Galli, Italo Zoppis, Andrew Smith, Fulvio Magni, and Giancarlo Mauri. Machine learning approaches in MALDI-MSI: clinical applications. *Expert Review of Proteomics*, 13:685–696, July 2016.

[14] Roman Garnett. *Bayesian Optimization.* Cambridge University Press, 2023.

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

[16] Jürgen H. Gross. *Mass Spectrometry.* Springer, 3rd edition, 2017.

[17] Trevor Hastie, Jerome Friedman, and Robert Tibshirani. *The Elements of Statistical Learning.* Springer Series in Statistics. Springer New York, New York, NY, 2001.

[18] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, Las Vegas, NV, USA, June 2016. IEEE.

[19] Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren, editors. *Automated Machine Learning: Methods, Systems, Challenges.* The Springer Series on Challenges in Machine Learning. Springer International Publishing, Cham, 2019.

[20] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 448–456, Lille, France, 07–09 Jul 2015. PMLR.

[21] Emrys A. Jones, Sören-Oliver Deininger, Pancras C.W. Hogendoorn, André M. Deelder, and Liam A. McDonnell. Imaging mass spectrometry statistical analysis. *Journal of Proteomics*, 75(16):4962–4989, August 2012.

[22] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization, January 2017. arXiv:1412.6980 [cs].

[23] Oliver Klein, Frederic Kanter, Hagen Kulbe, Paul Jank, Ddenkert Cartsen, Grit Nebrich, Wolfgang D. Schmitt, Wu Zhiyang, Catarina A. Kunze, Jalid Sehouli, Silvia Darb-Esfahani, Ioana Braicu, Jan Lellman, Herbert Thiele, and Eliane T. Taube. MALDI-Imaging for Classification of Epithelial Ovarian Cancer Histotypes from a Tissue Microarray Using Machine Learning Methods. *WILEY-VCH Verlag GmbH & Co. KGaA, Weinheim*, 2018.

[24] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, May 2017.

[25] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998.

[26] Scott M Lundberg and Su-In Lee. A unified approach to interpreting model predictions. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[27] Alan G. Marshall, Christopher L. Hendrickson, and George S. Jackson. Fourier transform ion cyclotron resonance mass spectrometry: A primer. *Mass Spectrometry Reviews*, 17(1):1–35, 1998.

[28] Liam A. McDonnell and Ron M.A. Heeren. IMAGING MASS SPECTROMETRY. 2007.

[29] Stephan Meding, Ulrich Nitsche, Benjamin Balluff, Mareike Elsner, Sandra Rauser, Cédrik Schöne, Martin Nipp, Matthias Maak, Marcus Feith, Matthias P. Ebert, Helmut Friess, Rupert Langer, Heinz Höfler, Horst Zitzelsberger, Robert Rosenberg, and Axel Walch. Tumor Classification of Six Common Cancer Types Based on Proteomic Profiling by MALDI Imaging. *Journal of Proteome Research*, 11(3):1996–2003, March 2012.

[30] Paul Mittal, Mark R. Condina, Manuela Klingler-Hoffmann, Gurjeet Kaur, Martin K. Oehler, Oliver M. Sieber, Michelle Palmieri, Stefan Kommoss, Sara Brucker, Mark D. McDonnell, and Peter Hoffmann. Cancer Tissue Classification Using Supervised Machine Learning Applied to MALDI Mass Spectrometry Imaging. *Cancers*, 13(21), October 2021.

[31] Grégoire Montavon, Alexander Binder, Sebastian Lapuschkin, Wojciech Samek, and Klaus-Robert Müller. Layer-Wise Relevance Propagation: An Overview. In Wojciech Samek, Grégoire Montavon, Andrea Vedaldi, Lars Kai Hansen, and Klaus-Robert Müller, editors, *Explainable AI: Interpreting, Explaining and Visualizing Deep Learning*, volume 11700, pages 193–209. Springer International Publishing, Cham, 2019. Series Title: Lecture Notes in Computer Science.

[32] Sebastian Ruder. An overview of gradient descent optimization algorithms, June 2017. arXiv:1609.04747 [cs].

[33] Lloyd S Shapley. A value for n-person games. In *Contributions to the Theory of Games 2.28*, page 307–317, 1953.

[34] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition, April 2015. arXiv:1409.1556 [cs].

[35] Leonoor E.M. Tideman, Lukasz G. Migas, Katerina V. Djambazova, Nathan Heath Patterson, Richard M. Caprioli, Jeffrey M. Spraggins, and Raf Van de Plas. Automated biomarker candidate discovery in imaging mass spectrometry data through spatially localized Shapley additive explanations. *Analytica Chimica Acta*, 1177:338522, September 2021.

[36] Jannis van Kersbergen, Farhad Ghazvinian Zanjani, Svitlana Zinger, Fons van der Sommen, Benjamin Balluff, Naomi Vos, Shane Ellis, Ron M.A. Heeran, Marit Lucas, Henk A. Marquering, Ilaria Jansen, C. Dilara Savci-Heijink, Daniel M. de Bruin, and Peter H. N. de With. Cancer detection in mass spectrometry imaging data by dilated convolutional neural networks. In John E. Tomaszewski and Aaron D. Ward, editors, *Medical Imaging 2019: Digital Pathology*, page 16, San Diego, United States, March 2019. SPIE.

[37] Nico Verbeeck, Jeffrey M. Spraggins, Monika J.M. Murphy, Hui-dong Wang, Ariel Y. Deutch, Richard M. Caprioli, and Raf Van De Plas. Connecting imaging mass spectrometry and magnetic resonance imaging-based anatomical atlases for automated anatomical interpretation and differential analysis. *Biochimica et Biophysica Acta (BBA) - Proteins and Proteomics*, 1865(7):967–977, July 2017.

[38] Shuhei Watanabe. Tree-structured Parzen estimator: Understanding its algorithm components and their roles for better empirical performance, April 2023. arXiv:2304.11127 [cs].

[39] Shuhei Watanabe, Noor Awad, Masaki Onishi, and Frank Hutter. Speeding Up Multi-Objective Hyperparameter Optimization by Task Similarity-Based Meta-Learning for the Tree-Structured Parzen Estimator, May 2023. arXiv:2212.06751 [cs].

[40] J.THROCK WATSON and O.DAVID SPARKMAN. *Introduction to Mass Spectrometry Instrumentation, Applications and Strategies for Data Interpretation.* John Wiley Sons, Ltd., 2007.

[41] Skyler Wu, Fred Lu, Edward Raff, and James Holt. Exploring the Sharpened Cosine Similarity. 2022.

[42] Tong Yu and Hong Zhu. Hyper-Parameter Optimization: A Review of Algorithms and Applications, March 2020. arXiv:2003.05689 [cs, stat].

[43] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. Dive into deep learning. *arXiv preprint arXiv:2106.11342*, 2021.