UNIVERSITY OF
GOTHENBURG

Convolutional neural networks for semantic segmentation of FIB-SEM volumetric image data

Master's thesis in Mathematical Statistics

FREDRIK SKÄRBERG

Department of Mathematical Sciences
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2020

Master's thesis 2020

Convolutional neural networks for semantic segmentation of FIB-SEM volumetric image data

FREDRIK SKÄRBERG

UNIVERSITY OF GOTHENBURG

Department of Mathematical Sciences
Division of Applied mathematics and statistics
Statistical learning and AI
University of Gothenburg
Gothenburg, Sweden 2020

Convolutional neural networks for semantic segmentation of FIB-SEM volumetric image data

FREDRIK SKÄRBERG

Supervisor: Magnus Röding, RISE Research Institutes of Sweden
Examiner: Aila Särkkä, Department of Mathematical Sciences

Master's Thesis 2020
Department of Mathematical Sciences
Division of Applied mathematics and statistics
Statistical learning and AI
University of Gothenburg
SE-412 96 Gothenburg

Convolutional neural networks for semantic segmentation of FIB-SEM volumetric image data
FREDRIK SKÄRBERG
Department of Mathematical Sciences
University of Gothenburg

Abstract

Focused ion beam scanning electron microscopy (FIB-SEM) is a well-established microscopy technique for 3D imaging of porous materials. We investigate three porous samples of ethyl cellulose microporous films made from ethyl cellulose and hydroxypropyl cellulose (EC/HPC) polymer blends. These types of polymer blends are used as coating materials on various pharmaceutical tablets or pellets and form a continuous network of pores in the film. Understanding the microstructures of these porous networks allow for controlling drug release. We perform semantic segmentation of the image data, separating the solid parts of the material from the pores to accurately quantify the microstructures in terms of porosity. Segmentation of FIB-SEM data is complicated because in each 2D slice there is 2.5D information, due to parts of deeper underlying cross-sections shining through in porous areas. The supposed shine-through effect greatly complicates the segmentation in regards to two factors; uncertainty in the positioning of the microstructural features and overlapping grayscale intensities between pore and solid regions.
In this work, we explore different convolutional neural networks (CNNs) for pixel-wise classification of FIB-SEM data, where the class of each pixel is predicted using a three-dimensional neighborhood of size (n_x, n_y, n_z). In total, we investigate six types of CNN architectures with different hyperparameters, dimensionalities, and inputs. For assessing the classification performance we consider the mean intersection over union (mIoU), also called Jaccard index. All the investigated CNNs are well suited to the problem and perform good segmentations of the FIB-SEM data. The so-called standard 2DCNN performs the best overall followed by different varieties of 2D and 3D CNN architectures. The best performing models utilize larger neighborhoods, and there is a clear trend that larger neighborhoods boost performance. Our proposed method improves results on all metrics by 1.35 - 3.14 % compared to a previously developed method for the same data using Gaussian scale-space features and a random forest classifier. The porosities for the three HPC samples are estimated to 20.34, 33.51, and 45.75 %, which is in close agreement with the expected porosities of 22, 30, and 45 %. Interesting future work would be to let multiple experts segment the same image to obtain more accurate ground truths, to investigate loss functions that better correlate with the porosity, and to consider other neighborhood sizes. Ensemble learning methods could potentially boost results even further, by utilizing multiple CNNs and/or other machine learning models together.

Keywords: Deep learning, convolutional neural networks, image analysis, semantic segmentation, focused ion beam scanning electron microscopy, porous materials, controlled drug release

Acknowledgements

I would like to express my very great appreciation to Magnus Röding for his valuable and constructive suggestions during all parts of the work. I also extend my thanks to RISE for providing the computational resources needed for the project. Many thanks to my examiner Aila Särkkä for feedback and additional help on the thesis. Finally, I wish to thank my family and friends for their support and encouragement.

Fredrik Skärberg, Gothenburg, November 2020

Contents

List of Figures

List of Tables

1

Introduction

1.1 Background

Focused ion beam scanning electron microscopy (FIB-SEM) is a well-established microscopy technique for 3D imaging of porous materials [1]. In FIB-SEM, a conventional SEM is used to image a single 2D cross-section of the data. A 3D data set is constructed in a serial fashion by applying the conventional SEM to a stack of cross-sections. Cross-sections are revealed by a focused ion-beam (FIB), bombarding the target material with heavy ions, milling away atoms of the materials surface [2]. By continuing to apply the FIB to cross-sections followed by imaging with the SEM, a 3D representation of the material can be obtained. The process is repeated until the desired volume of the imaged region is obtained in terms of the number of slices and volume size.

FIB-SEM is different from e.g. X-ray tomography in the sense that the data in each 2D slice contains 2.5D information, due to parts of deeper underlying cross-sections shining through in porous areas. The supposed shine-through effect is more prevalent the more porous the material is and makes image segmentation noticeably more difficult. The shine-through artefacts in the 2D slices greatly complicate the segmentation in regards to two factors; uncertainty in the positioning of the microstructural features and overlapping grayscale intensities between pore and solid regions. Several image processing methods for segmentation of FIB-SEM image data have been developed, e.g local threshold backpropagation [3, 4], morphological operations [5], watershed segmentation [6], and combinations of watershed, variance filtering and morphological operations [7]. Segmentation remains a challenging problem, and new segmentation methods that more accurately can quantify the FIB-SEM image data are sought after.

The FIB-SEM image data investigated in this project is acquired from three samples of ethyl cellulose microporous films made from ethyl cellulose and hydroxypropyl cellulose (EC/HPC) polymer blends. These types of polymer blends are used as coating materials on various pharmaceutical tablets or pellets to control drug release. HPC is water-soluble and leaches out when exposed to water. Once the HPC has leached out, a continuous network of pores is left in the film. Understanding the microstructures of these networks allows for controlled drug release, since it directly influences the transport of the drug through the coating. It is therefore important that methods to accurately quantify the microstructures in terms of e.g porosity, pore size distribution, and connectivity are developed. The first step towards quantification is accurate image segmentation of the pores, separating the solid (EC) from pores (leached out

HPC)[8].

In a collaboration involving RISE (Research Institutes of Sweden), the departments of Mathematical Sciences and the Department of Physics at Chalmers, and several major companies working in pharmaceutics, coatings, and packaging, FIB-SEM is currently utilized for 3D characterization of different types of porous materials. As it stands now, large amounts of manually labelled training data are available for analysis. A method to segment the same FIB-SEM data that is studied herein using Gaussian scale-space features and a random forest classifier have demonstrated good agreement with manual segmentation [8]. Deep learning may yield improved segmentation results. In a previous masters thesis [9], Convolutional Neural Networks(CNN) of U-Net type were investigated [10]. Their results showed that deep learning can be applied to the segmentation of FIB-SEM data. One advantage and disadvantage with U-Net is that it performs segmentation of a large patch of an image at once. This creates very abstract models with millions of parameters, leading to long training times and making hyperparameter optimization significantly harder. This approach also limits the amount of available data, as a single manually labelled mask is considered a sample. Hence, heavy data augmentation is needed to increase the number of unique samples.

To mitigate some of these problems, CNNs can be used for pixel-wise classification instead. This approach makes it more comparable to the original methodology [8]. It also increases the number of available samples considerably, because one sample equals one pixel and its neighborhood. There will be a high correlation between the samples due to adjacent pixels having almost the same neighborhood. However, deep learning methods generally benefit from larger sample sizes [11], and it is hypothesized that this approach might be beneficial.

In this thesis, we investigate different CNN architectures for pixel-wise classification. The focus is on standard 2DCNN architectures, but we also investigate 3DCNNs and more niche variants. Another area that is investigated is the input to the CNNs. We consider a three-dimensional neighborhood around the pixel of interest, with size (n_x, n_y, n_z). The (n_x, n_y) neighborhood contains the pixels in the cross-sectional plane and n_z is the number of 2D adjacent cross-sections. How well the CNNs perform with different inputs is investigated in terms of classification performance and estimated porosity. These choices also have computational implications that need to be taken into account.

1.2 Project aims

The aim of the project is to contribute with new insights to the segmentation of the FIB-SEM data, building upon previous work, and investigating new neural network architectures that have not yet been tested. We focus on investigating the segmentation performance of different CNNs for pixel-wise classification. The aim is to establish which neural network architectures, hyperparameters, and training schemes that work best. Another goal is to determine which input sizes yield the best performance for the different CNNs. Computational considerations of these input sizes are also taken into account and analyzed.

The project will involve implementing and benchmarking the different CNNs along with different neighborhood sizes (n_x, n_y, n_z). The end goal is to implement a method, that from start to finish segments the entire FIB-SEM data in a way that is applicable in practice. A researcher should be able to input a 3D FIB-SEM dataset and receive a fully segmented volume. The method should be able to segment FIB-SEM data of polymer coatings used for controlled drug release of different porosities.

1.3 Limitations

In this work, several different CNN architectures and inputs are considered. In total, we will investigate 50 unique CNN architectures. This is a considerable amount, and we limit ourselves to these architectures due to time restrictions. Hyperparameter optimization is not performed for all CNN architectures. We decide to tune the most promising architectures more in-depth in comparison to the low-performing ones.

The project involves investigating this particular FIB-SEM dataset, and all optimization is performed with respect to this dataset. The obtained CNNs could potentially produce satisfactory results on other similar datasets. However, it was not within the scope of this project to evaluate their performance in regards to other datasets.

A few methods of preprocessing the data were investigated, yet there exist many possible choices and we can only make statements about the ones investigated. Data augmentation was limited to three operations: rotations, mirroring, and intensity transforms of the inputs.

The coding was performed in Python 3.7, utilizing the TensorFlow [12] framework with Keras [13] as Application Programming Interface (API). Some minor details of the code were implemented in MATLAB [14], due to the original methodology being implemented there. However, the final code is standalone Python 3.7.

1.4 Thesis outline

In Chapter 2, the theory is presented by introducing FIB-SEM, semantic segmentation, metrics, convolutional neural networks, and how the training of a network is performed. In Chapter 3, the method is presented by describing the FIB-SEM dataset, how to extract neighborhoods, CNN architectures, hyperparameter optimization, model evaluation, postprocessing, and implementation details. In Chapter 4, the results are presented by showing the results of the hyperparameter optimization, comparisons between different CNNs, model selection, and segmentation of the full FIB-SEM dataset. Finally, in Chapters 5 and 6 we reflect upon the results and methodology. We also discuss related work and possible future work related to the thesis.

2
Theory

2.1 Focused ion beam scanning electron microscopy

Focused ion beam scanning electron microscopy (FIB-SEM) is a powerful tool that can be utilized for characterizing internal microstructures of materials. The focused ion beam (FIB) can mill away atoms by bombarding the target material with heavy ions [2]. The procedure is performed with high spatial resolution and reveals planar cross-sections of the material. In FIB tomography, both imaging and milling are performed by the ion beam [15]. This however proved to produce unsatisfactory results due to the ion beam damaging the target surface. A combination of FIB and SEM was therefore proposed, where the milling is performed by the FIB and the imaging with the SEM [1]. This retained high spatial resolution without subjecting the target material to damage. By repeatedly making high-resolution cross-sectional cuts with the FIB followed by imaging with the SEM, a 3D dataset can be acquired. The resulting stack of 2D SEM images is then a representation of the 3D volume of the material. FIB-SEM is different from e.g. X-ray tomography in the sense that the data in each 2D cross-section contains 2.5D information, due to parts of deeper underlying cross-sections shining through in porous areas [8]. These shine-through artifacts make segmentation noticeably more difficult in comparison to e.g. segmentation of non-porous materials.

In Figure 2.1 a schematic view of the FIB-SEM procedure is shown. An ion beam is milling away parts of the material along the z axis, revealing a new planar cross-section to be imaged by SEM. The process is repeated until the desired number of slices and volume size is obtained.

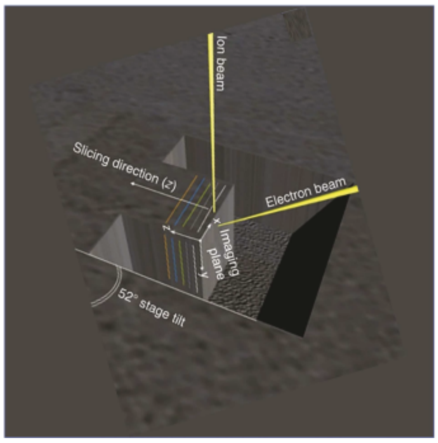


Figure 2.1: Schematic view of the FIB-SEM procedure. Illustration taken from [16].

2.2 Semantic segmentation

Semantic segmentation or image segmentation is the task of assigning each pixel in an image to an object class. It may be viewed as classification at the pixel level, with the end goal to cluster and draw boundaries between objects. When applied to a stack of images, the resulting segmentation of each image can be used to create 3D reconstructions, as is the case in FIB-SEM. Semantic segmentation is utilized in many different applications. It is commonly used in software for medical professions, e.g. for finding tumours or X-ray analysis. In Figure 2.2, the task of segmenting a X-ray image is shown [17]. Given the input image, the segmentation method assigns each pixel to one of three classes.

In this work, the segmentation is a binary task with the two classes, pore(0) and solid(1).

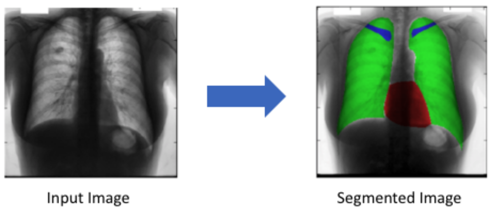


Figure 2.2: Segmentation procedure for an X-ray image of a chest, containing three classes: heart (red), lungs (green), and clavicles (blue). Image source [18].

2.2.1 Metrics

In order to evaluate the segmentation performance, various metrics can be considered. Throughout this report, three different metrics are used; accuracy, Jaccard index and porosity. The labels and predictions represents pixels belonging to one of two classes.

2.2.1.1 Accuracy

The most common measure for assessing classification performance is accuracy. Using the standard notation, with true positive (TP), true negative (TN), false positive (FP) and false negative (FN), one defines accuracy as the proportion of correct classifications

$$\text{Accuracy} = \frac{\#\text{correct classifications}}{\#\text{total classifications}} = \frac{\text{TP+TN}}{\text{TP +TN+ FP+FN}} \tag{2.1}$$

In this scenario, the pores can be viewed as negative and the solids as positive.

2.2.1.2 Jaccard index (IoU)

Jaccard index or intersection over union (IoU) is commonly used for assessing classification performance in image segmentation. One of its advantages over other metrics, e.g. accuracy, is that it takes class imbalance into account and generates a much fairer metric. The IoU is generally defined as

$$\text{IoU} = \frac{I}{U} = \frac{\mid L \cap P \mid}{\mid L \cup P \mid} \tag{2.2}$$

where IoU$\in [0, 1]$, L constituting the labels and P the predictions [19]. If the labels and predictions do not overlap then the IoU becomes 0, whereas if L and P are equal it becomes 1. In this thesis the prediction (P) will come from a model where the output has been thresholded to obtain binary predictions. To obtain the IoU as a class-symmetric measure one can take the average intersection over union, denoted as mIoU. Consider a binary classification problem with classes 0 and 1. It is then defined as

$$\text{mIoU} = \frac{1}{2}\left(\frac{\mid L_0 \cap P_0 \mid}{\mid L_0 \cup P_0 \mid} + \frac{\mid L_1 \cap P_1 \mid}{\mid A_1 \cup B_1 \mid}\right) \tag{2.3}$$

where L_0 and P_0 constitute the labels and predictions of pores(0), and where L_1 and P_1 the labels and predictions of solids (1).

2.2.1.3 Porosity

Porosity is the most fundamental geometrical quantity of a porous material and is very relevant to our problem. It is therefore important that there exists a way to quantify the porosity. The definition of porosity is simply how much empty space there is in the material, meaning, how large proportion of all pixels are pores.

$$\text{Porosity} = \frac{\text{number of pores}}{\text{total number of pixels}} = \frac{\sum_{i=1}^{N} 1_{pore}(p_i)}{N} \tag{2.4}$$

where N is the total number of pixels and $1_{pore}(p_i)$ the indicator function.

2.3 Convolutional neural networks

Convolutional Neural Networks (CNNs) are feedforward artificial neural networks (ANNs) that are inspired by animal visual cortexes [20]. In recent years they have been applied in numerous different fields such as pattern recognition, image analysis, natural language processing, and video analysis [21]. The most prominent characteristic of CNNs is its use of a convolution kernel. The kernel introduces weight sharing properties, making it possible to extract more information but with fewer trainable parameters. Furthermore, CNNs are highly hierarchical and learn hierarchical connections between the layers. These properties make the CNN highly adaptive, and able to find a mix of both low and high-level features in the data.

A typical CNN consists of three types of layers: convolution, pooling, and fully connected. The first two are unique to the CNN and perform feature extraction, whereas the fully connected layer maps the features to an output, such as a classification or a continuous value. A CNN can therefore be viewed as an extension of the general ANN, performing both feature extraction and classification. ANNs are made up of multiple fully connected layers, consisting of neurons and a bias. The lack of feature extraction possibilities makes the ANN unsuitable for handling image data. In an ANN, every pixel of the input image would be connected to a neuron. Therefore, the number of parameters would be extremely large, even for fairly small images [21]. Fully connected layers are still very powerful for structured data and classification tasks and they play an intricate part in the CNN architecture. In Figure 2.3, a holistic view of a typical CNN architecture is shown. The different concepts of the typical CNN architecture will be explained in-depth in the following subsections 2.3.1-2.3.4.

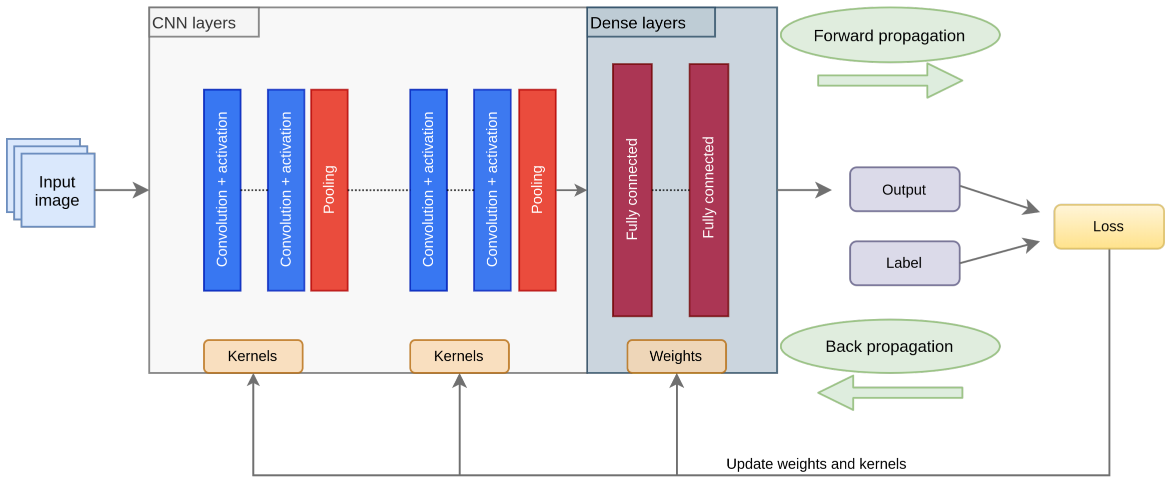


Figure 2.3: Illustration of a typical CNN architecture. An input image is fed to the network, producing feature maps in the first convolution layer followed by a pooling layer performing dimension reduction. After the last pooling operation, the feature maps are flattened to a 1D vector and linked to a fully connected layer. In the last fully connected layer an output is produced. The process of feeding an image through the network to obtain an output is called forward propagation. Based on the output and a known label, the performance of the model is evaluated and a loss is computed. The weights of the kernels and the fully connected layers are updated via back propagation, utilizing the gradient descent optimization algorithm.

2.3.1 Building blocks of a typical CNN architecture

A typical CNN architecture is made up of three different types of layers: convolution, pooling, and fully connected. Generally, a stack of several convolution and pooling layers followed by one or more fully connected layers make up the network. The convolution and pooling layers are unique to the CNN and perform different operations on features maps which are fed forward through the network. The so-called feature extraction occurs in the convolution and pooling layers, with the fully connected layers acting as a classifier on the extracted features.

The input to a CNN is any form of spatial array. In Figure 2.3, the input is a two-dimensional array with three channels. Channels can be used to represent color (an RGB image) or an additional spatial dimension. Consequently, CNNs can handle a great variety of different input shapes. The properties of the convolution operation extend to all dimensions. Thus, a CNN can handle data of all dimensions, but most commonly the data is either a 2D or 3D spatial array. In the following sections, we explain the concepts in 2 dimensions for simplicity, but it can be extended in a straightforward manner.

2.3.1.1 Convolution layer

In a convolution layer, feature maps are constructed by convolving the input with a convolution kernel consisting of weights that are to be optimized. As the kernel traverses the spatial array, elements are convolved by the kernel to the feature map. Consider a kernel of size $N \times M$, the feature map F_{ij} is then defined as the discrete convolution

$$F_{i,j} = f\Big(\sum_{n=1}^{N}\sum_{m=1}^{M} w_{nm}x_{n+s(i-1),m+s(j-1)} - \theta\Big) \tag{2.5}$$

where f is the activation function introducing non-linearity, w the kernel, s the stride, $x_{n+s(i-1),m+s(j-1)}$ the value of the spatial array in the given location and b the bias. It is important to note that the same weights of the kernel are used in all different locations of the spatial array, defining the behaviour of a kernel that is translation-invariant [22]. By feedforwarding the feature maps to the succeeding convolution layers, more abstract features can be extracted. Optimizing the weights of the kernels, allow them to learn how to identify certain local features, which together can represent global features [21].

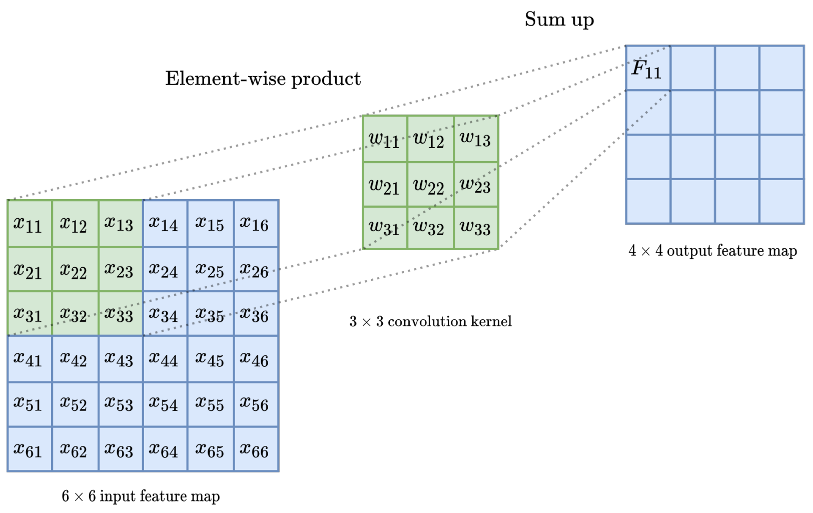

Sum up
Element-wise product
F_{11}
w_{11} w_{12} w_{13}
w_{21} w_{22} w_{23}
w_{31} w_{32} w_{33}
x_{11} x_{12} x_{13} x_{14} x_{15} x_{16}
x_{21} x_{22} x_{23} x_{24} x_{25} x_{26}
x_{31} x_{32} x_{33} x_{34} x_{35} x_{36}
x_{41} x_{42} x_{43} x_{44} x_{45} x_{46}
x_{51} x_{52} x_{53} x_{54} x_{55} x_{56}
x_{61} x_{62} x_{63} x_{64} x_{65} x_{66}
4×4 output feature map
3×3 convolution kernel
6×6 input feature map

Figure 2.4: Visualization showing the convolution kernel (green) mapping a value from the input feature map to the output feature map. The kernel will continue to traverse the entire input feature map, convolving one block at a time.

It is common practice to choose unevenly sized kernels as they symmetrically divide the previous feature maps pixels around the output pixel. Without such symmetry, there will be subtle distortions between the layers. The most common kernel sizes are 3×3 and 5×5, any larger ones drastically increase the number of weights and computational time needed for training the network. The benefit of using larger kernels is unclear, as it is possible to obtain a similar receptive field by stacking multiple smaller kernels. This approach has become the go-to method in convolution layers as it restrains the number of weights whilst allowing higher receptive fields.

Equation 2.5 can be extended to allow for an arbitrary number of channels and kernels [23]. This is common practice in CNNs, since more feature maps and kernels allow for more complex feature extraction. Let K be the number of kernels and R the number channels. The equation is then extended to

$$F_{i,j,k} = f\Big(\sum_{n=1}^{N}\sum_{m=1}^{M}\sum_{r=1}^{R} w_{nmkr}x_{n+s(i-1),m+s(j-1),r} - \theta_k\Big) \tag{2.6}$$

The stride s effectively determines the size of the feature maps. The larger the stride, the smaller the feature map, due to the kernel working on fewer values. Furthermore, the center of the kernel will never reach the edges. Thus there is always a small decrease in dimension of the feature maps regardless of the stride. If larger kernels are used, the effect is even greater. In order to keep the size of the feature maps unchanged one can perform padding, adding zeros to all edges. This is common practice and beneficial for feature extraction, as pixels on the edges would otherwise be processed by fewer filters than the pixels in the centers.

2.3.1.2 Pooling layer

Another layer type is the pooling layer. Pooling layers are usually placed after one or more convolution layers as a means of preventing overfitting by decreasing the dimensionality of the data. By downsampling to a lower dimension, new features in a different length scale can be extracted. Aside from dimension reduction, pooling also promotes invariance to translations, rotations and scales. The reason is that information of the exact position of a certain value in the input feature map is not transferred to the output feature map. The pooling layer does not have any tuneable weights, but the stride and the size of the pooling window needs to be set. The most common operations are max-pooling and average-pooling. Max-pooling enhances edges and is important for edge detection, whereas average-pooling constructs more smooth feature maps. In Figure 2.5 an example of max-pooling with a 2×2 window taking strides of size 2 is shown.

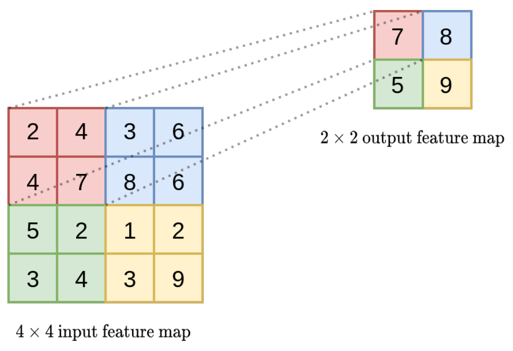


Figure 2.5: Max-pooling with a 2×2 window taking strides of size 2. On the left the input array is shown, and to the right the produced feature map.

2.3.1.3 Fully connected layer

At the end of a stack of several convolution and pooling layers one or more fully connected layers follow. The feature maps from the final convolution or pooling layer are typically flattened, i.e. converted to a one-dimensional vector, and linked to a fully connected layer. In a fully connected layer, every input neuron is connected to every output neuron. All such connections are defined by a weight which is trainable. If there are more than two fully connected layers one usually refers to the middle ones as hidden [11]. As in the previous layers, an element-wise-product is performed followed by a activation to obtain an output.

The final fully connected layer is typically different than the others as it expresses the final output of the network. Usually this layer is made of a single neuron for regression or binary classification tasks, or as many neurons as there are classes in a multi-class classification problem. This implies differences in its activation, making other activation functions more suitable. In the following section some of these activation functions are described.

2.3.2 Activation functions

After a weighted sum of the inputs and bias has been computed, an activation function f is applied to the result to obtain the output. This operation occurs in the convolution layer, in the fully connected layers as well as in the final output. The activation is important because it disrupts the linear combination of the inputs, and allows the neural network to approximate a non-linear function. There exist many different activation functions. In this thesis, the exponential linear unit (ELU) [24] and the sigmoid function are utilized. In the convolution and the fully connected layers ELU activation is applied, defined as

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1) & \text{if } x \leq 0 \end{cases} \tag{2.7}$$

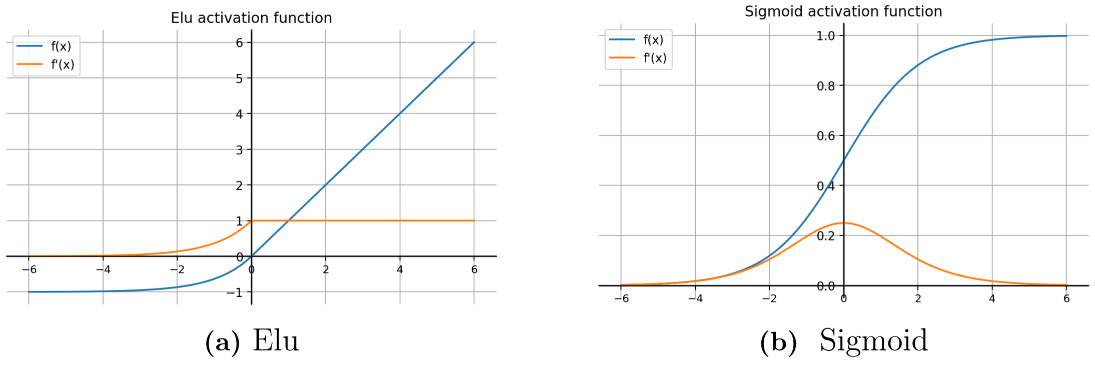
where α controls the activation for negative inputs, most often $\alpha = 1$. The derivative of the ELU activation is

$$f'(x) = \begin{cases} 1 & \text{if } x > 0 \\ \alpha e^x & \text{if } x < 0 \end{cases} \tag{2.8}$$

The sigmoid activation function is a common choice for activation of the output in the last fully connected layer as it maps to [0, 1], whereas its use between layers is uncommon due to the vanishing gradient problem in deep networks. This occurs since the slope of the sigmoid function is close to zero for a large range of values, leading to a gradient tending towards zero [11]. The sigmoid function is defined as

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.9}$$

with derivative $f'(x) = f(x)\big(1 - f(x)\big)$. In Figure 2.6, the two activation functions and their derivatives are shown.



(a) Elu (b) Sigmoid

Figure 2.6: Elu and sigmoid activation and their derivatives for $x \in [-6, 6]$.

2.3.3 Training a network

The so-called learning is performed by minimizing a loss function using some form of optimization algorithm. The task of an optimization algorithm is to find the optimal weights for the model. Generally, the task is supervised, and known labels can be compared with predictions from a model. The shape of the loss function is poorly understood and hard to visualize in practice. The consensus is that almost all loss functions are non-convex, having multiple local minimums and saddle-points [25, 26]. A non-convex optimization problem makes theoretical guarantees about convergence to local minimums weak. However, there exist proofs showing that gradient descent converges (the gradient becomes arbitrarily small) for non-convex function, although this does not rule out saddle-points, local maximums or regions of zero-gradients [27]. This is not meant to discourage the use of gradient descent, but it is good to highlight the intricacies of optimization in deep learning. In turn, the practitioner can never guarantee that the best model has been found neither nor the convergence to the same point for different initial conditions.

The weights are updated via a procedure called Back propagation. We will assume that the reader is familiar with the concept, and only briefly mention it and its notation. In short, Back propagation is the method of calculating the derivatives of the loss function [28]. The derivatives are computed utilizing the chain rule, going from the last layer (which is connected to the loss) to the weight in question. When writing $\nabla_{w_i}\mathcal{L}(\mathbf{x}; \mathbf{y})$, Back propagation is used for calculating the derivative of the loss function w.r.t weight w_i.

2.3.3.1 Loss functions

As previously stated, all optimization algorithms need a loss function to estimate the current state of the model in a meaningful way. Generally, the loss function should be differentiable and able to quantify differences between the known labels and predictions. There exist a plethora of different loss functions e.g. mean-squared loss (MSE), hinge loss and Kullback Leibler divergence loss [10]. Depending on the problem, different loss functions are suitable. In table 2.1, some commonly used combinations of loss functions and final activations for various tasks are shown [29].

Table 2.1: Common combinations of final activation and loss function for different tasks.

Task	**Final activation**	**Loss function**
Binary classification	Sigmoid	Binary cross-entropy
	Tanh	Hinge-loss
Multi-Class Classification	Softmax	Cross-entropy
		Kullback Leibler divergence loss
Regression	Linear	MSE

In this work, we have a binary classification problem, hence binary cross-entropy (or log-loss) together with a sigmoid activation function is the most suitable choice. The combination work well since binary cross-entropy assumes the outputs to be predicted probabilities, and the output from a sigmoid function may be interpreted as such since it maps to [0, 1].

Binary Cross-Entropy

The binary cross-entropy loss function defines the loss for N training examples according to

$$\mathcal{L}(y_i, f(x_i)) = -\frac{1}{N}\sum_{i=1}^{N} y_i \cdot \log(f(x_i)) + (1-y_i)\cdot\log(1-f(x_i)) \tag{2.10}$$

where $y_i \in \{0,1\}$ is the label and $f(x_i) \in [0,1]$ the prediction from the model. The prediction $f(x_i)$ is interpreted as a probability of the label belonging to class 1. The loss for one sample is computed in regards to the known label y_i and entails two cases

$$\begin{aligned}
&\text{case 0: } y_i = 0\\
&\qquad \mathcal{L}(y_i = 0, f(x_i)) = -\log(1-f(x_i))\\
&\text{case 1: } y_i = 1\\
&\qquad \mathcal{L}(y_i = 1, f(x_i)) = -\log(f(x_i))
\end{aligned}$$

Avoiding the obvious problem with log(0), most deep learning software handles the issue by adding a small value $\epsilon > 0$ to the output. The prediction $f(x_i)$ is then mapped to $[\epsilon, 1-\epsilon]$, making the loss computations more robust. As the correct predictions of the model get more confident (closer to either 0 or 1) the loss decreases. The derivative of the binary cross-entropy w.r.t each label is defined as

$$\frac{\partial\mathcal{L}(y_i, f(x_i))}{\partial f(x_i)} = \frac{1}{N}\Big(\frac{1-y_i}{1-f(x_i)} - \frac{y_i}{f(x_i)}\Big) = \frac{f(x_i)-y_i}{Nf(x_i)\big(1-f(x_i)\big)} = \begin{cases} -\frac{1}{Nf(x_i)}, & y_i = 1\\ -\frac{1}{N(1-f(x_i))}, & y_i = 0\end{cases}$$

2.3.3.2 Optimization algorithms

In this section, we describe the properties of the optimization algorithms in deep learning. Generally, all optimization algorithms stem from the gradient descent framework, having their unique twist on updating the weights. The differences between algorithms are mainly described by two factors: how much data is used in the optimization and how the learning rate is tweaked.

Gradient descent

Gradient descent is an iterative algorithm starting at an random point of a function and travels down its slope (in the direction of the negative gradient) until reaching a local (or global) optimum. As long as the loss function $\mathcal{L}$ is differentiable, the weights w can be updated according to

$$w_{i+1} = w_i - \eta \nabla_{w_i} \mathcal{L}(\mathbf{x}; \mathbf{y}) \tag{2.11}$$

where η is the learning rate, w_i the previous weight and $(\mathbf{x},\ \mathbf{y})$ the data [30, 31]. Gradient descent can be computationally heavy for large sample sizes. All samples in the dataset are needed to be evaluated to compute the loss and in turn update the weights. Therefore, variants of gradient descent that differ on how much data is used for computing the gradient have been developed, such as Batch gradient descent (BGD), Stochastic gradient descent (SGD), and Mini-batch gradient descent (MBGD). BGD is equivalent to gradient descent and performs one update of the weights in regards to the entire dataset.

In SGD, the weights are updated for each training sample $(x^{(i)}, y^{(i)})$. The training samples are randomly chosen and the weights are updated in accordance with

$$w_{i+1} = w_i - \eta \nabla_{w_i} \mathcal{L}(x^{(i)}; y^{(i)}) \tag{2.12}$$

The approach is typically advantageous compared to BGD, since BGD suffers from redundant computations when using large sample sizes. The redundancy occurs when training samples are very similar, something that is common in large datasets.

Due to the frequency of updates in SGD, the loss function often suffers from high variance. One way to mitigate this is to update the weights based on small batches of data. This decreases the variance of the loss function whilst still exploiting the randomness aspects in SGD. In MBGD, the weights are updated for each batch of size b according to

$$w_{i+1} = w_i - \eta \nabla_{w_i} \mathcal{L}(x^{(i:i+b)}; y^{(i:i+b)}) \tag{2.13}$$

MBGD is typically the standard setting in most software packages for deep learning. The method takes advantage of the positive aspects of both SGD and BGD.

Stochastic gradient descent with momentum (SGDm)

It is well known that the gradient descent algorithm can be very slow, in particular when the surface of the loss function has long and narrow valleys. In this situation, there might be erratic behavior; oscillating back and forth, making slow progress. The reason is that the direction of the gradient will be almost perpendicular to the long axis of the valley, while the update will oscillate back and forth along the short axis. By including a momentum term γ, the rate of convergence has been shown to increase considerably [32]. In Figure 2.7, the method is visualized on a contour plot of a surface. The idea is to help SGD accelerate in a meaningful direction towards the local minimum by taking larger and more intuitive steps.

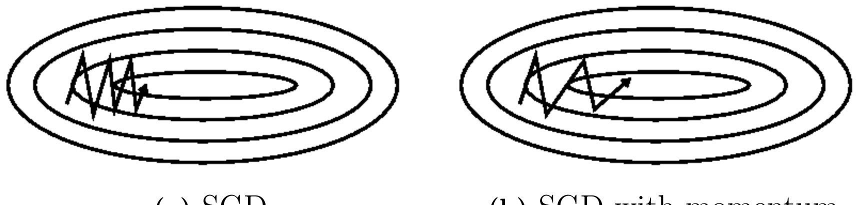

(a) SGD **(b)** SGD with momentum

Figure 2.7: Conceptually how SGDm performs an update [33].

The momentum term γ scales how much the previous step will impact the update of the current weights. By recursively storing the information of the previous step in v_{t-1} the equation for updating the weights is defined as

$$v_t = \gamma v_{t-1} + \eta \nabla_{w_i} \mathcal{L}(x^{(i:i+b)}; y^{(i:i+b)}) \tag{2.14}$$

$$w_{i+1} = w_i - v_t \tag{2.15}$$

with $\gamma \in [0, 1]$. If the momentum method is still inadequate, there is Nesterov accelerated gradient (NAG). NAG further builds upon the momentum equation, introducing a way to look ahead of the update of the gradient by evaluating the loss function at $(\theta - \gamma v_{t-1})$ instead. The idea is to make the weight update less volatile, by e.g. slowing down close to valleys. The NAG method updates the weight according to the equation

$$v_t = \gamma v_{t-1} + \eta \nabla_{w_i} \mathcal{L}((x^{(i:i+b)}; y^{(i:i+b)}) - \gamma v_{t-1}) \tag{2.16}$$

$$w_{i+1} = w_i - v_t \tag{2.17}$$

Optimizers using adaptive learning rate strategies

There exist many different optimizers that adapt the learning rate to the parameters in different ways. These optimizers are often referred to as adaptive learning rate optimizers. They are frequently used in various deep learning tasks and allow for easy implementation as they work well with the practitioner only having to set the initial learning rate. For our task however, these types of optimizers did not produce as good results as gradient descent with momentum. Therefore, we only briefly introduce ADAGRAD, RMSProp and ADAM.

In ADAGRAD, the learning rate is tweaked so that uncommon features are prioritized over common features. This is achieved by using larger learning rates for uncommon features and lower for common features [34].

RMSProp further builds upon ADAGRAD by including a time-perspective, scaling the learning rate by an exponentially decaying average of squared gradients. The assumption is that after training for many epochs we are closer to the actual local minimum, hence only small steps are required [35].

Another optimizer is the ADAM optimizer. ADAM is a combination between Adagrad and RMSProp which introduces a new parameter for the exponentially decaying average of squared gradients. The parameter describes the second moment of the gradient, in perspective to RMSProp that only evaluates the exponential decaying term for the first moment [36].

Weight initialization

How the weights of the network are initialized can have a significant impact on how the optimization algorithm performs. Recall that in gradient descent, the function is initially evaluated at a random point, and then steps are taken in the direction of the negative gradient. This random point constitutes the initialization of the weights. There are mainly two types of weight initializers, one for initializing the biases and one for initializing the weights of the kernels and neurons. It is common practice to initialize all biases as either 0 or 1. For the weights of the kernels and neurons, Glorot uniform (also called Xavier uniform) initializer is commonly utilized [37]. In Glorot uniform, the starting weights are drawn from the uniform distribution on [-a, a], where $a = \sqrt{\frac{6}{n_i+n_o}}$, n_i =the number of input units and n_o =the number of output units of the weight tensor.

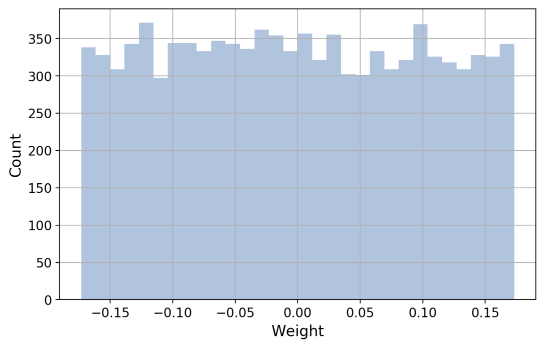


Figure 2.8: Distribution of 10 000 samples drawn from Glorot uniform with a (100,100) weight tensor. In this case $a \approx 0.17$

2.3.4 Overfitting and regularization

The goal of any training is to identify a model that is able to generalize well to new inputs. Therefore, the learned features should be general and not unique to the training set. Overfitting can be mitigated by different techniques, e.g. model-specific methods such as dropout, weight decay, and batch normalization, training specific such as learning rate and early stopping, or data specific such as data augmentation or the use of larger sample sizes. Furthermore, striving towards a simpler yet sufficient model architecture is preferable [29].

Overfitting is routinely checked by monitoring the loss and other metrics on the training and validation data. If the training loss continues to decrease whilst the validation loss increases the model has likely been overfitted to the training data. It is important to recognize this phase and interrupt training once this occurs. For handling such cases it is common to use a method called early stopping, breaking the training process once the validation loss no longer benefits from more training. In Figure 2.9, the concepts of overfitting are shown, note that a model can both underfit and overfit to the data.

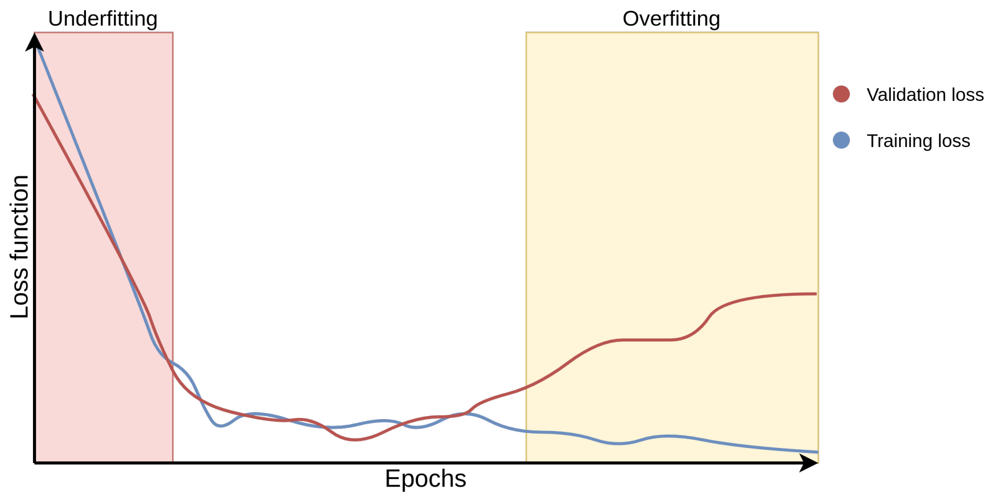


Figure 2.9: Monitoring of the training and validation loss as a function of epochs. The longer the model is trained, the better it will perform on the training data. There will be a point were the generalizability of the model to the validation data is maximized. The goal of any training is to identify this point.

Dropout is a commonly used operation between the fully connected layers. It randomly drops outputs by a certain probability, usually between 20-60 %. This prevents the neurons from co-adapting and has been shown to reduce overfitting [38]. By contrast to the conventional dropout, spatial dropout drops entire feature maps, something that is preferable to dropout in the convolution layers as adjacent pixels often are highly correlated within the feature maps.

In l_2 regularization (weight decay) a penalty term $c||w||^2$ is added to the loss function to be minimized [39]. The process effectively penalizes small weights which could be considered noise. By penalizing the small weights, they no longer interfere with the tuning of the more important weights, leading to better weight updates.

Batch normalization is an operation, acting as a supplemental layer, that normalizes the values between layers. This has been shown to mitigate the risk of overfitting, leading to less volatile gradients and less dependence on the initialization of the network [40]. However, batch normalization is not always suitable and works best with outputs that have been obtained by certain activation functions. It was shown that the ELU activation did not benefit from batch normalization, partially explained by the mean activation being pushed closer to zero, similarly as in batch normalization [24].

Data augmentation increases the diversity of the training data, leading to better generalizations, and has been shown to be a highly effective strategy in deep learning [41]. With data augmentation, the data set can be made substantially larger without new data having to be collected. Unique samples can be constructed by performing different operations such as cropping, padding, flipping, and rotating of the existing samples. By having a wider variety of samples, the model will be less inclined to tune the weights to sample-specific features. This enhances learning, as the learned features are more general and not specific to certain samples in the training data. The variety also increases the potential for the network to learn more features in total. In practice, it is rare to have large, well-annotated data sets. In a perfect world, the practitioner would likely tune the model to as much data as reasonably possible. Handling large data sets is a field on its own and contains many computational aspects to be taken into consideration, such as disk space usage and memory requirements. In such instances, data augmentation can be helpful, as it reduces the need to store large datasets by repeatedly constructing new data from the base dataset. However, memory consumption might be increased by such procedures.

In this work, three different augmentation techniques are implemented: random rotations in {0°, 90°, 180°, 270°}, mirroring and random perturbations of the pixel intensities in the inputs. In Figure 2.10, rotation and mirroring are shown. These techniques are in essence harmless, as the information content will be symmetric around the center of the image. Interestingly, a combination of these two techniques yields an 8-fold increase of unique samples in the training data, showing the usefulness of data augmentation.

(a) Rotation, 180°. **(b)** Mirroring.

Figure 2.10: Rotation and mirroring of an image.

3
Methods

First, in section 3.1, we investigate the FIB-SEM dataset and describe how the labelled data is obtained. Second, in sections 3.2-3.3, we describe how to preprocess the data and how the data is split into training, validation and test. Third, in sections 3.4-3.5, we give details on how to extract data for input to the CNNs and the rational behind the neighborhood sizes (n_x, n_y, n_z) chosen.

In Figure 3.1 the procedure of finding a model for segmentation of the entire FIB-SEM dataset is shown. A hyperparameter search is performed to find suitable parameters for the models, these models are then evaluated and a few candidate models are identified, see sections 3.6-3.7. With the top model, a final training is performed with larger sample size to further optimize its performance. Using that top model, all labelled square regions are then predicted accompanied by a postprocessing step to further improve the result, see section 3.8. Finally, with the best model and the best postprocessing parameters, the entire FIB-SEM dataset is segmented, see section 3.9.

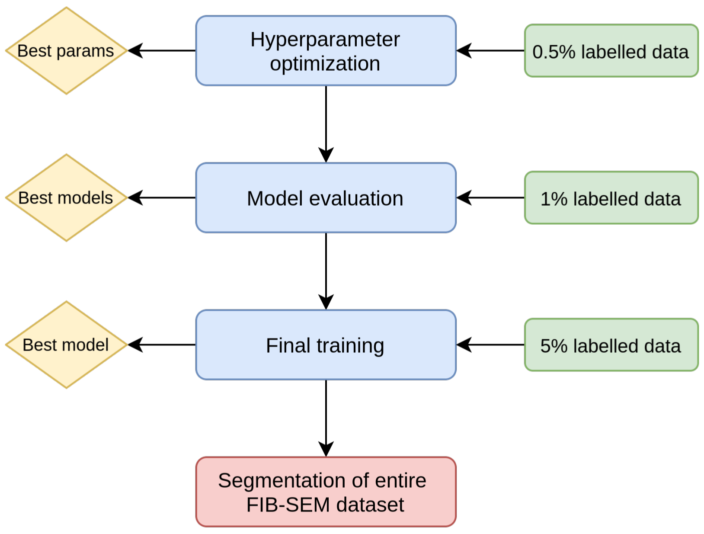


Figure 3.1: Flowchart showing the procedure of finding a model for segmentation of the entire FIB-SEM dataset.

3.1 FIB-SEM dataset description

The FIB-SEM dataset consists of three samples from ethyl cellulose microporous films which are made from ethyl cellulose and hydroxypropyl cellulose (EC/HPC) polymer blends. All three acquired image data volumes have the same size and resolution but differ in terms of porosity. The different porosities are 22%, 30%, and 45%, and the datasets will be referred to as HPC22, HPC30, and HPC45 in the following sections. The idea behind having three samples of different porosity is that we want to develop a method that works well for a range of porosities.

Each sample is a representation of a 3D volume, with 200 cross-section images (slices) of size 2247×3372. This leads to each sample containing in total 1 515 376 800 pixels, and the entire dataset 4 546 130 400 pixels. In Figure 3.2, one cross-section of the HPC45 sample is shown. The gray homogeneous areas contain the solid part of the material, and the less homogeneous areas contain the pores. In the less homogeneous areas, it is possible to look into the material and see underlying cross-sections. This is the result of the shine-trough effect as explained in earlier sections. These regions are generally identified as being darker. The objective is to develop a method to distinguish these pores from the solid to construct a realistic representation of the volume using automatic image segmentation.

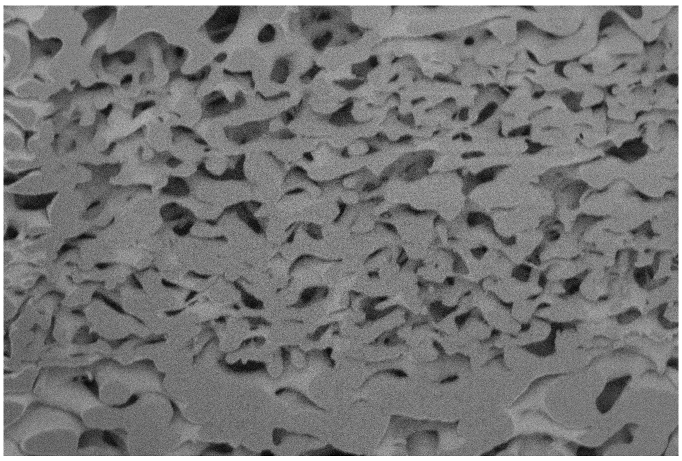
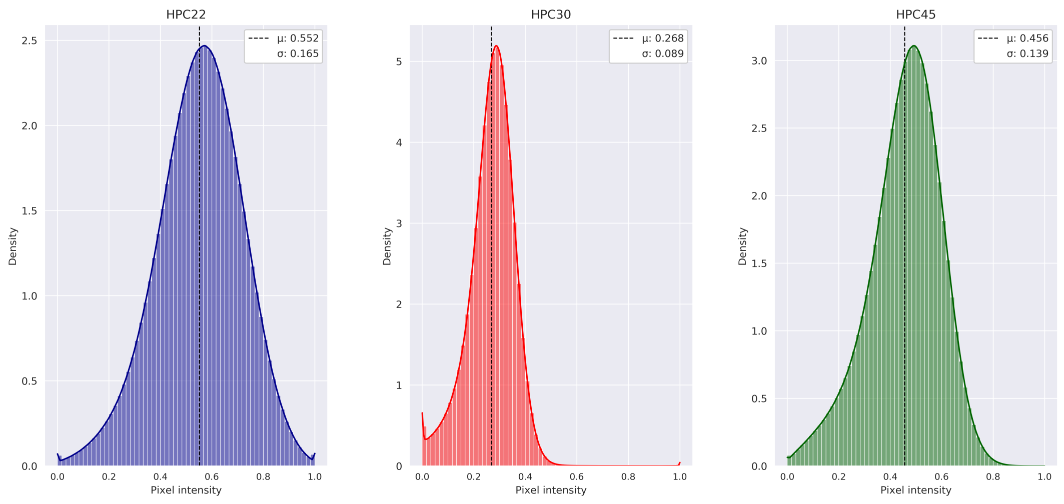


Figure 3.2: Figure showing one cross-section (20) of the HPC45 sample.

Aside from the samples having different porosities, they also have somewhat different pixel intensities. In Table 3.1, characteristics of the pixel intensities are shown. The HPC30 sample is somewhat darker than the other two. Furthermore, there exist noticeable differences on all metrics, which could hint towards some preprocessing being needed. It is important to mention that the entire FIB-SEM data is taken into account, and not each sample separately when training the models. In Figure 3.3, a histogram with a kernel density estimate plotted on top of it is illustrated. Here the differences between the samples are also evident.

Table 3.1: Distribution characteristics of the pixel intensities for the HPC samples.

	HPC22	**HPC30**	**HPC45**
μ	0.552	0.268	0.456
σ	0.165	0.089	0.139
$\tilde{x}$(median)	0.554	0.276	0.459



(a) HPC22 **(b)** HPC30 **(c)** HPC45

Figure 3.3: Pixel intensity distribution for the three HPC samples, shown as histograms and kernel density estimates. In the top right corner of each figure the sample mean and standard deviation are shown. Note that the HPC30 sample is darker.

3.1.1 Manual segmentation

To construct labelled data, manual segmentation is performed by an expert. Due to the amount of available data, only a subset of the total data is manually segmented. For each HPC sample, 100 square regions of size 256×256 are randomly chosen for manual segmentation. To help the expert in the segmentation process, regions of size $384 \times 384 \times 7$ (i.e. information of three adjacent slices in both directions) around the chosen mask are taken out. This information is important to the expert since image information in a neighborhood around the pixel of interest simplifies the segmentation. In Figure 3.4 all the information available to the expert is shown with the corresponding mask.

From the 100 square regions of each HPC sample, we calculate the estimated porosity and record its mean and standard deviation. By assuming independence between the square regions, the mean porosity(%) and 95 % confidence intervals can be found in Table 3.2.

Table 3.2: Estimated porosity with 95 % confidence intervals for the manual segmented square regions from each HPC sample.

Sample	**95% Confidence intervals**
HPC22	21.72 ± 1.44
HPC30	29.54 ± 1.84
HPC45	44.86 ± 2.63

Assuming the HPC phase is fully leached out, the expected porosities of the three samples are 22%, 30%, and 45%. The estimated porositites are in fairly close agreement with these expected values, being somewhat lower, but not significantly so.

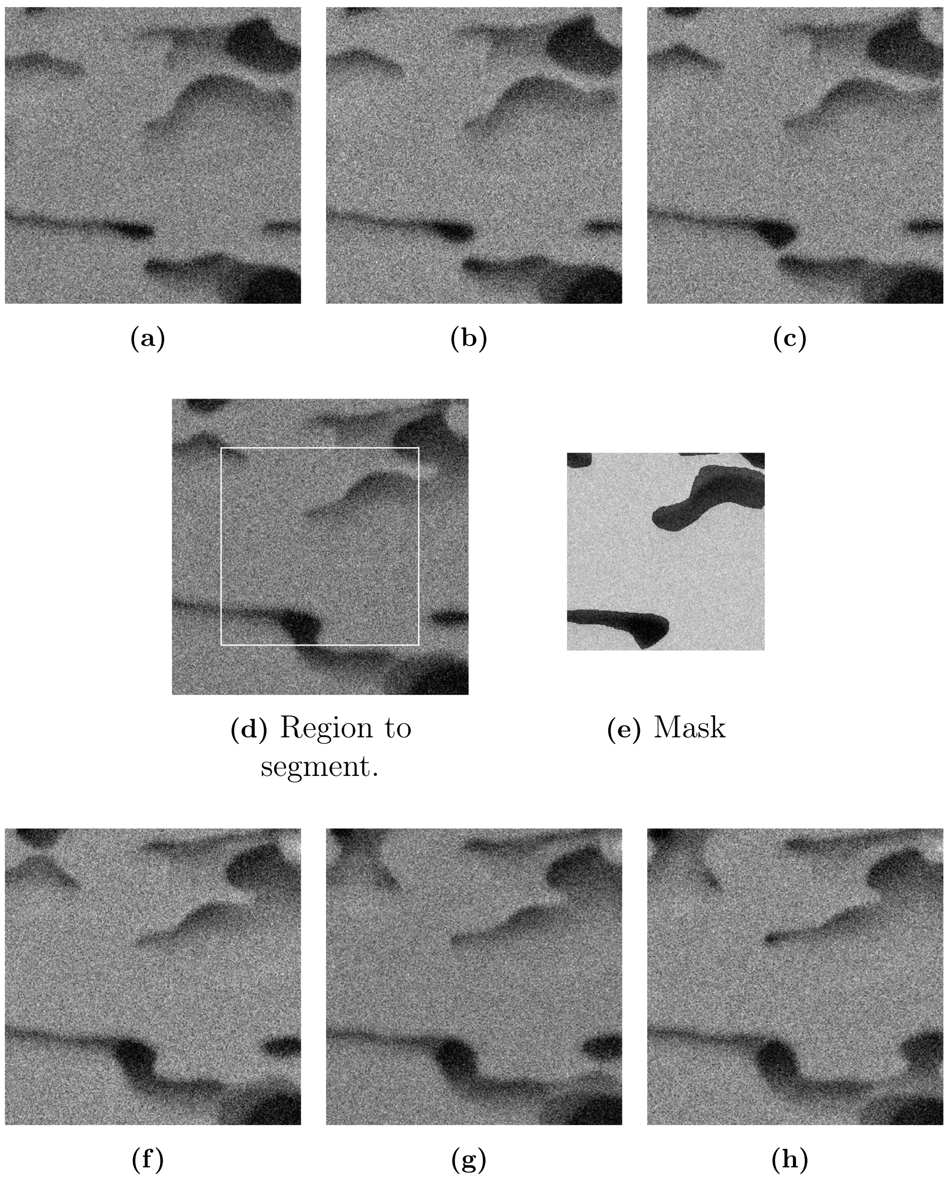


Figure 3.4: Images **(a)**-**(d)** and **(f)**-**(h)** show 7 adjacent slices in the HPC30 sample of the FIB-SEM data. Image **(d)** highlights a bounding box of size 256 × 256 of the region to be manually segmented. Image **(e)** shows the manual segmentation (mask) of the region in image **(d)**.

3.2 Preprocessing

Some different ways of preprocessing the data have been investigated. In the article, of which the data was first analyzed, a method to remove the presence of an intensity gradient was proposed [8]. More details about this method are presented below.

Once this method has been applied to the data, further ways of preprocessing has been looked at, e.g. standardization, normalization, and a method to normalize the mode of the distribution to 1. Somewhat surprisingly, neither of these methods improved classification performance on any metric. However, we should be careful drawing any general conclusions, as there exist many possible strategies and we can only exclude the ones investigated. One explanation might be that the data is already in a fairly good format after the first preprocessing step. The pixel intensity of each HPC sample is almost in the [0, 1] interval after the gradient has been removed and can easily be matched together.

Intensity gradient removal

The data is stored as `.tif` files encoded as 16-bit, each file representing a cross-section. These files are imported and transformed to $[0, 1]$ range. In the data, an intensity gradient in the x direction is present. This can be seen in Figure 3.5 by the intensity profiles decreasing along the x-axis. It is noticed that the intensity gradient is almost linear, and therefore can be fitted by a linear function using least squares. Once the linear function is determined, it is subtracted from the intensity gradient and the mean of the linear function is added. Thus, retaining a good approximation of the original intensity range. In Figure 3.5 the results of the correction are shown.

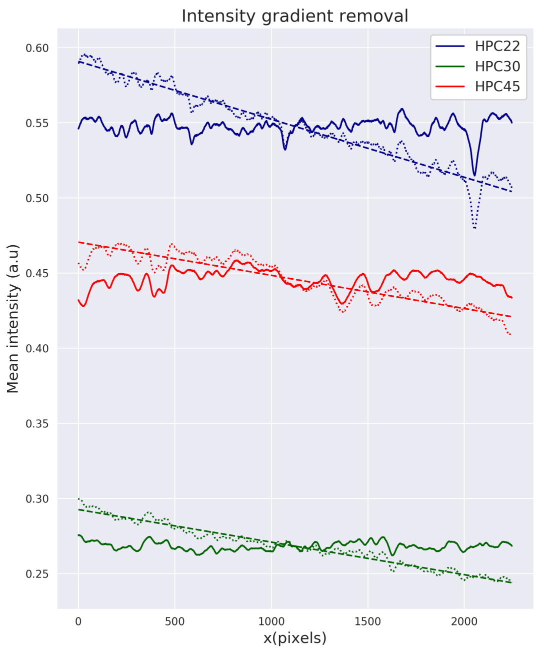


Figure 3.5: Method of removing the intensity gradient in the x direction. Figure showing the intensity profiles (*dotted lines*), fitted linear function with least squares (*dashed lines*) and the corrected intensity profiles (*solid lines*).

3.3 Data split

From each HPC sample, 100 labelled square regions are extracted. Out of these, 60 are randomly chosen for training, 20 for testing, and 20 for validation. This sum up to 180 labelled masks for training, 60 for testing, and 60 for validation. As our goal is to classify each pixel belonging to either the solid(1) or the pore(0) class, we further extract pixels from these masks. For training, the total amount of available labelled pixels are 11 796 480, and for test- and validation 3 932 160. As this is a large amount of data, we choose only a limited amount of the available pixels for the actual training process due to computational workload and memory limitations.

Furthermore, it is ensured via stratified random sampling that we have class balance in each of the datasets. It is beneficial to have class balance when training neural networks, as this leads to both better training performance and generalizations [42]. In reality, there is not 50/50 class balance in the full dataset, as seen by the estimated porosities in Table 3.2, an issue we will address later on. In Figure 3.6, the whole procedure of extracting the training data is shown as a flowchart.

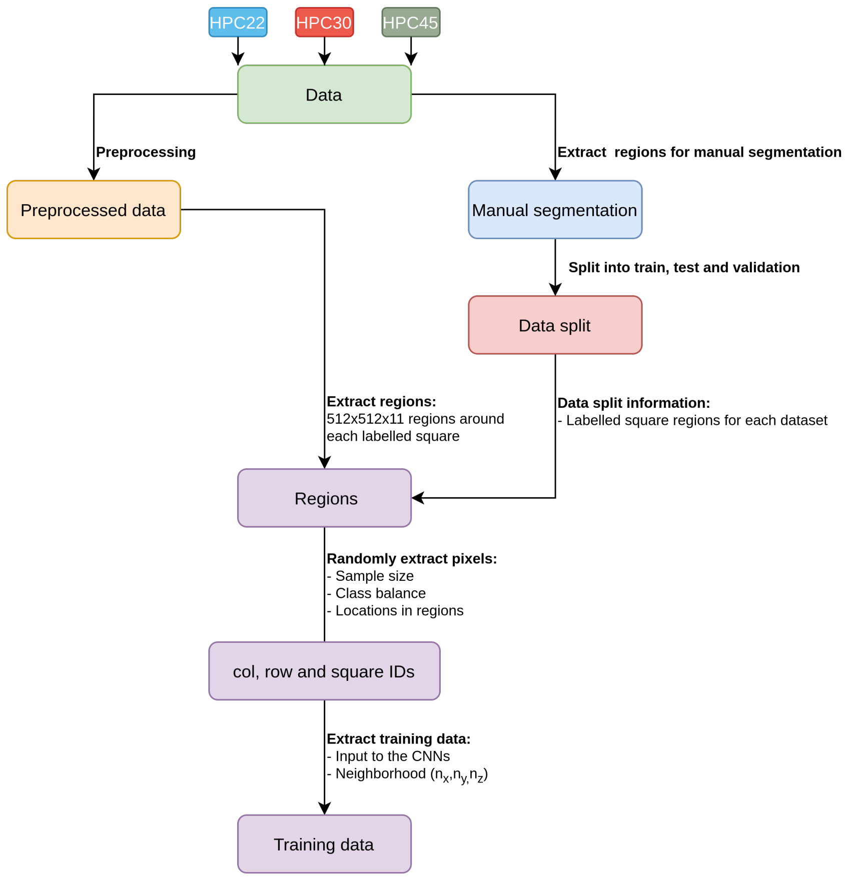


Figure 3.6: Once the data has been preprocessed, volumes of size $512 \times 512 \times 11$ around the manually labelled squares are extracted for each of the datasets. From the labelled squares, a random sample of pixels are chosen, assuring class balance via stratified random sampling. By storing the locations of each of these pixels, training data can be extracted. In section 3.4, extraction of the training data is described.

In Figure 3.7, the pixel distribution of pores and solids for all labelled pixels in each of the sets is shown. There is a good agreement between the three sets, with the pixel intensity distribution for validation (pores) being somewhat different. One could expect minor variation due to the inherent difference between the square regions chosen for training.

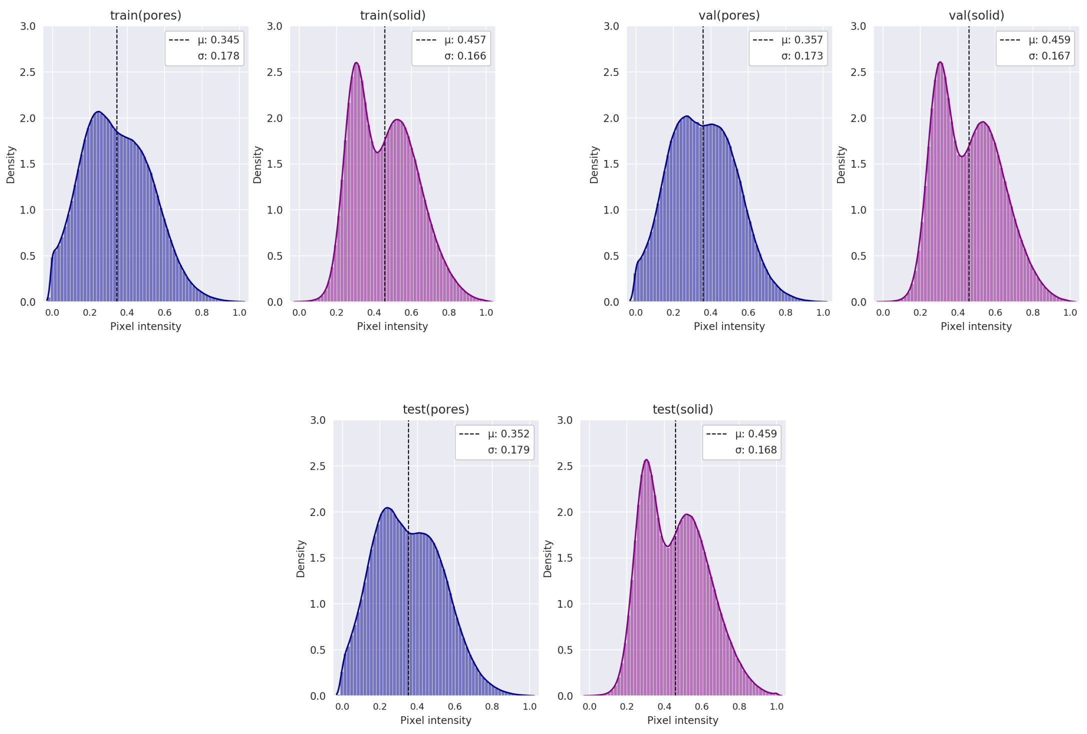


Figure 3.7: Figure showing the pixel intensity distribution for the training, validation and test sets for the two classes: solid(1) and pores(0). Note that this is after removing the gradient in x. There is good agreement in regards to the distribution of pores and solids between the three sets.

3.4 Training data extraction

Once the pixels for classification have been randomly chosen, we extract data for input to the CNNs. The data will consist of 3-dimensional neighborhoods (n_x, n_y, n_z) of varying sizes around each labelled pixel. n_z is limited to 11 cross-sections, having the middle slice and five adjacent slices in both directions. The resolution in n_z is far less than in n_x and n_y, hence the correlation with the middle slice is not as high. The choice also avoids heavy usage of padding, since many regions would otherwise extend the boundaries of the z-axis. The (n_x, n_y) neighborhood is however not restricted in the same way, and we choose a max size of (81, 81) to investigate.

Five different sizes in the 2-dimensional (n_x, n_y) neighborhood are investigated, all of them are squares because the information content should be symmetric with respect to the x and y axes due to isotropy of the material. For the n_z dimension, we investigate six an uneven number of cross-sections between 1 and 11.

The sizes of the (n_x, n_y) neighborhoods were chosen by considering both computational and implementation based aspects and are except for the smallest, defined by $16k + 1$ for $k \in \{2, 3, 4, 5\}$. These sizes facilitate easier model comparisons and implementation, partially explained by the convolution and max-pooling operations in the networks. All in all, the following sizes are possible as input to the CNNs

$$(n_x, n_y) \in \big\{(11, 11), (33, 33), (49, 49), (65, 65), (81, 81)\big\} \tag{3.1}$$
$$n_z \in \big\{1, 3, 5, 7, 9, 11\big\} \tag{3.2}$$

This leads to 30 different combinations of neighborhoods (n_x, n_y, n_z) to investigate. In Figure 3.8, we show conceptually how a neighborhood is described in terms of its dimension (n_x, n_y, n_z).

In Figure 3.9, the approach of extracting the data is visualized. Images **(a-e)** and **(h-e)** constitute the adjacent cross-sectional slices to the middle slice **(f)** before and after, respectively. Image **(g)** shows the entire mask, with the labelled pixel highlighted in red, in this case belonging to class solid (1). The surrounding squares of varying sizes represent the (n_x, n_y) neighborhoods that are to be extracted for classification, in increasing order; 11, 33, 49, 65 and 81. Consider an example of extracting training data for the neighborhood $(n_x, n_y, n_z) = (33, 33, 7)$; that would constitute a volume of 7 slices **(c-e, f, h-l)** of the second white square.

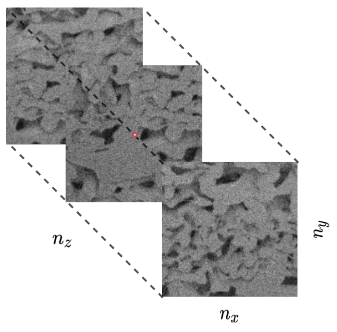
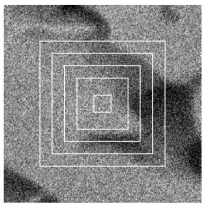
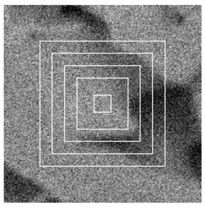
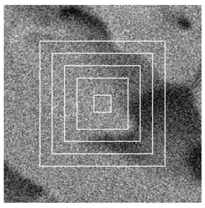
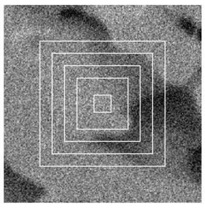


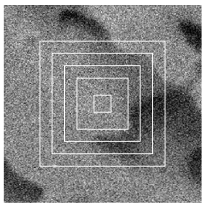
Figure 3.8: A neighborhood is described by a volume with dimensions (n_x, n_y, n_z). The red dot in the middle slice represents the labelled pixel of interest.

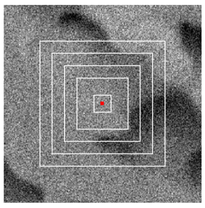

(a)

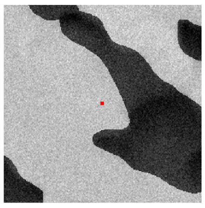

(b)

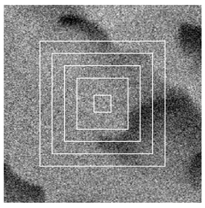

(c)

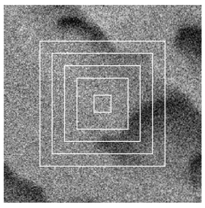

(d)

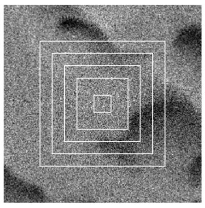

(e)

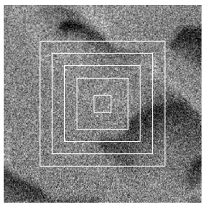

(f) Middle slice.


(g) Labelled pixel.


(h)


(i)


(j)


(k)

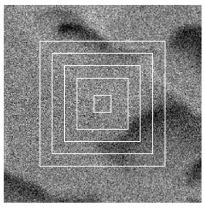

(l)

Figure 3.9: Information available for input to the CNNs. Images **(a-e)** shows the cross-sectional slices before the middle slice in descending order, and **(h-l)** the cross-sectional slices after the middle slice in increasing order. Image **(f)** shows the middle slice, i.e. the slice that is segmented by the expert. Image **(g)** shows the entire mask, with the labelled pixel of interest highlighted in red, in this case belonging to class solid (1). The surrounding squares of varying sizes represent the (n_x, n_y) neighborhoods that are to be extracted for classification, in increasing order; 11, 33, 49, 65 and 81. All images are 128×128 pixels.

3.5 CNN architectures

In this project, a wide range of different CNN architectures are investigated. The main emphasis has been put on developing the standard 2DCNN. This approach is hypothesized to be the most suitable for the problem and allows for the most variation in regards to the (n_x, n_y, n_z) neighborhood to be extracted. The CNNs differ in their hyperparameters as well as in their input and operations; e.g. convolutions of different dimensionality. Some of the CNN architectures are known beforehand as being somewhat lacking, but might be insightful towards constructing a baseline for future improvements and considerations. This mostly regards the 1DCNN and MV2DCNN, as they do not take advantage of all spatial information.

In Table 3.3 the different input dimensions for the CNNs are summarized. Consider a spatial array defined by height(h), width(w), depth(d) and number of channels(c). An input can then be expressed in terms of these dimensions for a 1D, 2D, and 3D spatial array for different batch sizes(N).

Table 3.3: Summary of the input dimensions for the different CNN architectures. The input array is defined by height(h), width(w), depth(d), number of channels(c) and batch size(N).

Model architecture	Input dimension
2DCNN Architectures $(n \times h \times w \times c)$	
Standard 2DCNN	$N \times n_x \times n_y \times n_z$
MV2DCNN	$N \times n_x \times n_y \times 1$
TriplanarCNN	$(N \times n_x \times n_y \times 1)$, $(N \times n_x \times n_z \times 1)$, $(N \times n_y \times n_z \times 1)$
MultichannelCNN	$(N \times n_x \times n_y \times n_z)$, $(N \times n_x \times n_z \times n_y)$, $(N \times n_y \times n_z \times n_x)$
3DCNN Architectures $(n \times h \times w \times d \times c)$	
3DCNN	$N \times n_x \times n_y \times n_z \times 1$
1DCNN Architectures $(n \times h \times c)$	
1DCNN	$N \times (h = n_x \times n_{z_{5 \text{ middle cross-sections}}}) \times n_y$

3.5.1 Standard 2DCNN

The input to a standard 2DCNN is an image with one or more channels. In our case, the different cross-sections will be considered as channels. The input size will vary depending on the neighborhood (n_x, n_y, n_z) that is extracted, leading to N images of size $n_x \times n_y \times n_z$, where n_z are the channels. While there is some information in n_z, the correlation between cross-sections is far less than within cross-sections. Hence, treating (n_x, n_y) differently than n_z might be warranted. In Figure 3.10, we show conceptually how the standard 2DCNN is structured.

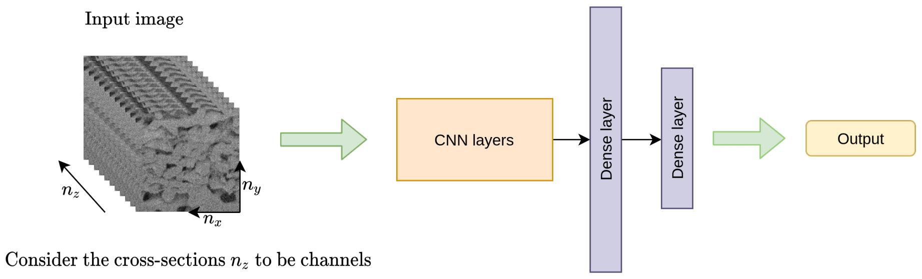


Figure 3.10: Holistic view of the standard 2DCNN architecture

3.5.2 MV2DCNN (Mean-Valued 2DCNN)

The input to a MV2DCNN is constructed by taking the mean along the z-axis of the 3D volume to obtain a single image as input. The resulting image will be of dimension $n_x \times n_y \times 1$. This strategy greatly simplifies the information available to the CNN. However, the idea is that some spatial information from the n_z neighborhood will still exist in the mean valued image. The simplicity leads to faster training times but at the cost of lower classification performance. In this setup, $n_z = 11$ always.

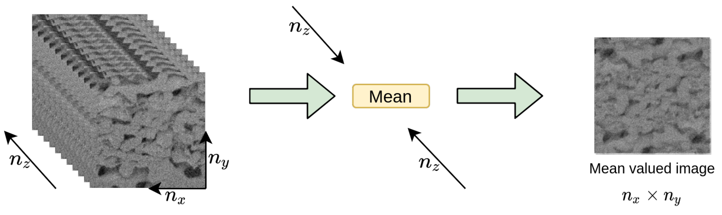


Figure 3.11: The process of constructing a mean-valued image for input to the MV2DCNN. The mean is taken along the z axis to obtain a single mean valued image.

3.5.3 TriplanarCNN

In a TriplanarCNN, three orthogonal 2D image slices passing through the pixel of interest are extracted. Due to the n_z neighborhood being significantly smaller, images of three different sizes will be created. These images will have dimensions: $(n_x \times n_y \times 1)$, $(n_x \times n_y \times 1)$, and $(n_y \times n_z \times 1)$. This effectively makes n_z no longer considered a channel, as were the case in the standard 2DCNN. Due to having different dimensionality of the input, three separate CNN paths are constructed which are merged together into a dense layer. In this setup, $n_z = 11$ always. Figure 3.12 highlights how three image planes can be viewed in a volume. In Figure 3.13, the process of extracting data and the construction of three separate CNN paths is shown.

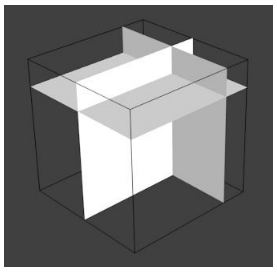


Figure 3.12: View of three image planes in a 3D volume.

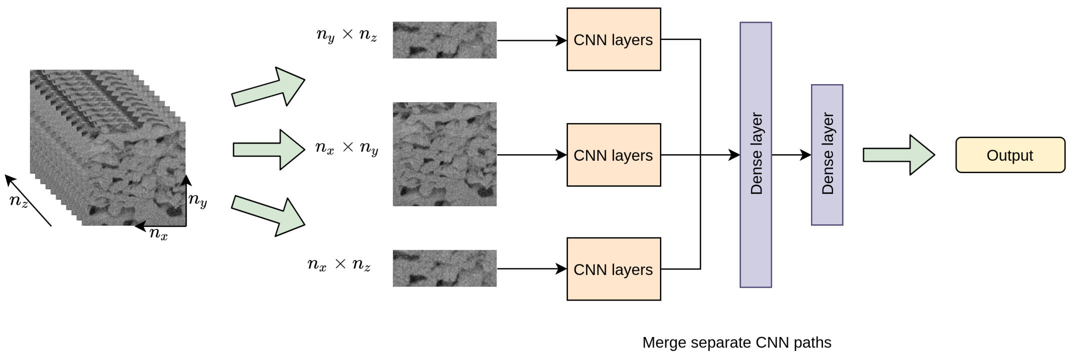


Figure 3.13: Holistic view of the TriplanarCNN architecture.

3.5.4 MultichannelCNN

The MultichannelCNN further builds upon the framework of creating three separate CNN paths that merge into a dense layer. Instead of constructing three images with the last channel being 1 as in the TriplanarCNN, we instead consider n_x, n_y, and n_z to be a channel in each of the paths. Effectively constructing three images with dimensions: $(n_x \times n_y \times n_z)$, $(n_x \times n_z \times n_y)$, and $(n_y \times n_z \times n_x)$. The first input image is identical to the standard 2DCNN for the same neighborhood.

This makes the input to the CNN very large, in fact, three times larger than the standard 2DCNN with the same neighborhood size. A lot of information could potentially be extracted, but more data is not beneficial on its own and it needs to contain important information to yield any improvements. We will always consider all information in n_z, i.e. $n_z = 11$. Figure 3.14 shows a holistic view of the MultichannelCNN architecture.

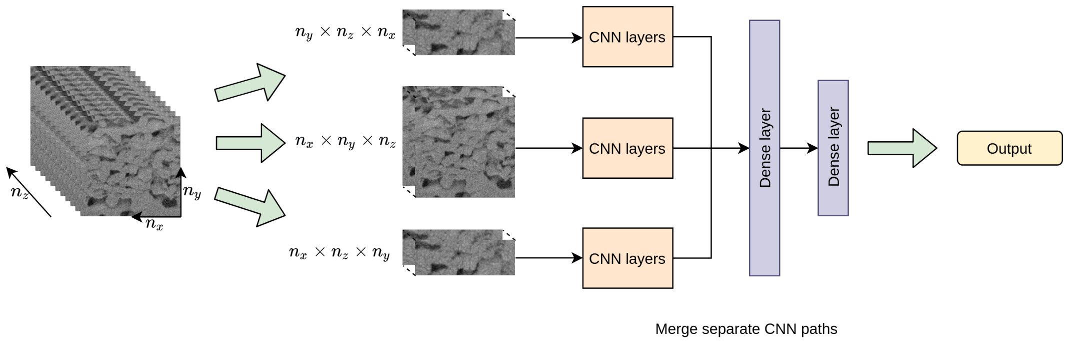


Figure 3.14: Holistic view of the MultichannelCNN architecture.

3.5.5 3DCNN

In a 3DCNN, the convolution and max-pooling operations are three-dimensional, and the input is considered to be a 3D volume with one or more channels. In medical imaging, 3DCNNs have been shown to be highly effective for segmentation [43]. For a 3DCNN to serve its purpose, the data has to be viewed as three-dimensional. Certainly, our data can be considered three-dimensional, as we have a stack of 2D images. However, as previously mentioned, the correlation between cross-sections is far less than within cross-sections. By considering a 3DCNN, the different axes n_x, n_y, and n_z will be treated uniformly. The input will be N 3D volumes of size $n_x \times n_y \times n_z \times 1$. We will always consider all information from the n_z neighborhood, i.e. $n_z = 11$, due to the limitation of data in this axis.

3.5.6 1DCNN

The input to a 1DCNN is a vector sequence, and therefore the image data has to be flattened, i.e. converted to a 1D vector. This removes spatial correlation in 2D considerably. The vector sequence is still somewhat sorted, as pixels that are close in one axis will still be close in the vector sequence. Hence, some 2D correlation still exists. Avoiding the sequence becoming too immense we limit the n_z neighborhood to

5, i.e. 2 adjacent slices in each direction as input. This was shown to be highly more effective than considering the entire n_z neighborhood. In Figure 3.15, the process of constructing the input to the 1DCNN is shown.

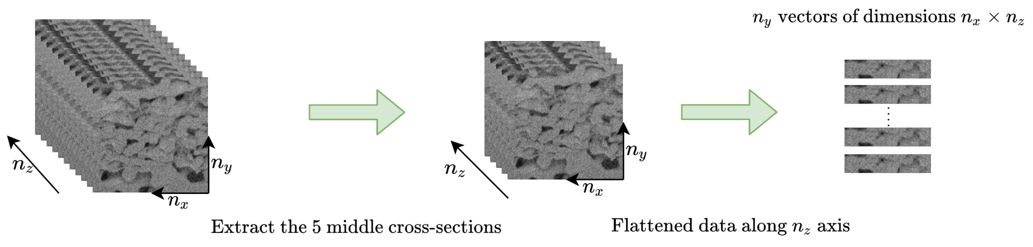

n_y vectors of dimensions $n_x \times n_z$
n_z
n_y
n_x
Extract the 5 middle cross-sections
Flattened data along n_z axis

Figure 3.15: Figure showing the process of constructing n_y vectors with dimension $n_x \times n_z$ for input to the 1DCNN.

3.6 Hyperparameter search

Hyperparameters are optimized by a combination of using intuition and random optimization. As there are many models to consider not all have had their hyperparameters tuned specifically. There are in total 50 individual CNN models that would have needed tuning, a hyperparameter search of all is outside the scope of this project due to time restrictions.

The hyperparameters are optimized w.r.t the validation loss. The sample size is slightly decreased in order to be able to search for more combinations, having 58 982 training samples and 19 660 validation samples.

A few optimizers were at first investigated, them being SGD, Adam, and RMSProp. It was found that the SGD optimizer outperformed the others significantly. The hyperparameter search was therefore performed with the SGD optimizer, having momentum and initial learning rate as two hyperparameters.

For all models, ELU activation is utilized between the convolution and fully connected layers. In the last fully connected layer, the output is obtained by utilizing the sigmoid activation function. Weights of the networks are initialized with the Glorot uniform (also called Xavier uniform) initializer [37].

It is common to add a small weight decay term to neural networks as it can improve generalization [44]. The weight decay (l_2-norm) was at first investigated in the hyperparameter search but was later set to a constant value of 1e-6 for the standard 2DCNNs and to 1e-5 for the 3DCNN. This was shown to improve the training slightly, in this phase at least.

We also make use of early stopping, terminating the training process once the validation loss has not improved within 10 epochs. A threshold for the number of epochs is also set, terminating the training process once the model has been trained for more than 125 epochs. Data augmentation is also utilized, by performing rotations of 90° and mirroring of the cross-sections.

3.6.1 Standard 2DCNN

The difference between a 2DCNN with similar input regions, e.g. (33,33,7) or (33,33,5) are minor, and are set to have the same architecture for simplicity in the implementation. In Table 3.4, the possible hyperparameters in the random search are shown. A combination of all hyperparameters would lead to 24 300 unique trainings just for one architecture. We propose a search where the number of convolution layers and initial learning rate are searched in full, with the rest of the parameters being randomly sampled. This is a trade-off, but necessary in order to avoid running too many trainings. The hyperparameter search is ran for 100 individual trainings for each considered model. This leads to the combination of convolution layers and initial learning rate being searched 4 times along with a random subset of other hyperparameters.

Table 3.4: Hyperparameters in the random search for the standard 2DCNN.

Hyperparameter	**Range**
Convolution layers	{4, 6, 8, 10, 12}
Kernel size	{(3,3), (5,5)}
Filter size	{8, 16}
Fully connected layers	{1, 2, 4}
Neurons	{32, 64, 128}
Dropout	{0.2, 0.4, 0.5}
Initial learning rate	{0.01, 0.0075, 0.005, 0.001, 0.0001}
Momentum	{0, 0.9, 0.95}
Batch size	{64, 128, 256}

The number of convolution and fully connected layers determine the overall structure of the CNN. It is common practice to arrange convolution layers in pairs followed by a max-pooling layer. We call such a sequence a block, effectively leading us to search for block sizes between 2 and 6. The number of fully connected layers are 1, 2, or 4, any larger would lead to more weights without a significant increase in performance. In Figure 3.16, the strategy of adding blocks and fully connected layers is shown.

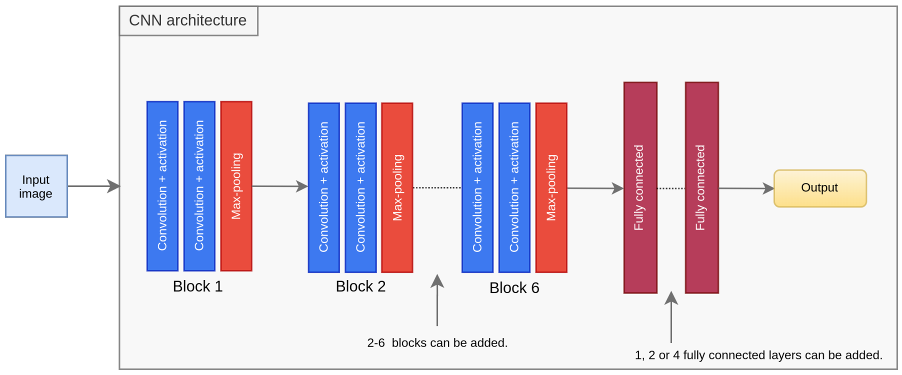


Figure 3.16: The overall CNN architecture is constructed by adding blocks (2-6) followed by fully connected layers (1, 2 or 4).

The number of possible convolutions for models with small input regions is somewhat limited due to max-pooling effectively decreasing the size of the feature maps. This is an important step of doing dimension reduction for the larger models, which would otherwise suffer from having too many trainable parameters. In extent, models with larger input regions have greater potential for deeper networks. Larger regions do contain more information, hence such a structure is warranted. The number of feature maps is increased as the networks grow deeper. We multiple the filter size with a constant, for each block we increase the number of filters by a factor of 1, 2, 4 and 5.

Other strategies of adding layers have been investigated: by adding 1 convolution layer followed by a max-pooling layer as well as avoiding max-pooling entirely. This however did not improve the results on any metric. Further ways of avoiding overfitting were also investigated, by the use of a spatial-dropout layer. This approach improved performance in some instances. However, it was later removed and standard dropout was deemed sufficient to regularize the networks.

3.6.2 3DCNN

Training a 3DCNN requires longer training times due to increased complexity. This is partially explained by the increased amount of operations performed by the convolution kernel. Another explanation might be that the implementation in Tensorflow is not as efficiently parallelized.

Nevertheless, due to the increased training time, a few more hyperparameters are set beforehand to avoid unnecessary runs. The kernel size is fixed to $3 \times 3 \times 3$, the momentum term is set to 0.90, and the batch size to 256. Furthermore, we search in smaller ranges for the parameters. The narrowing of the ranges was done based on knowledge obtained by tuning the standard 2DCNNs. In Table 3.5, the possible hyper-parameters in the random search are shown. The same strategy of adding convolution layers as for the standard 2DCNN is used. Because of the n_z neighborhood being significantly smaller than the (n_x, n_y) neighborhood, performing standard max-pooling would make the z-axis shrink too fast. This can be problematic since information in n_z might be lost. To mitigate some of these issues, max-pooling with a $2 \times 2 \times 1$ kernel is used instead, shrinking only the x and y axes.

Table 3.5: Hyperparameters in the random search for the 3DCNN.

Hyperparameter	**Range**
Convolution layers	{2, 4, 6, 8, 10}
Filter size	{8, 16}
Fully connected layers	{1, 2}
Neurons	{32, 64}
Dropout	{0.3, 0.5}
Initial learning rate	{0.01, 0.005, 0.001}

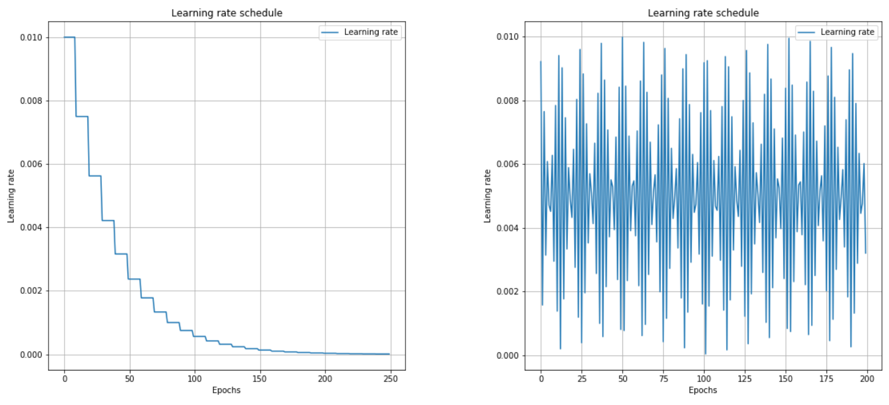
3.6.3 Other CNN models

The other CNN architectures: TriplanarCNN, MultichannelCNN, MV2DCNN, and 1DCNN have had their hyperparameters set to reasonable values based on intuition and known working architectures. It is simply not possible to find ideal parameters for all models within a reasonable time. In the case of the MV2DCNN, the same architecture as the standard 2DCNN with $n_z = 1$ is used, as the input is also a single image. The MultichannelCNN architecture is inspired by a previous implementation [45].

For all models, the emphasis has been put on finding a suitable learning rate and a reasonable amount of regularization. Therefore, a couple of runs with different learning rates and regularization parameters has been carried out. The knowledge received from tuning the standard 2DCNNs and 3DCNN was also taken into consideration.

3.6.4 Learning rate schedule

The SGD optimizer does not use adaptive learning rate strategies. Therefore, a suitable learning rate and a scheme must be defined for the training. It has been well established that increased performance and faster training times can be achieved if the learning rate is tuned to a specific problem [31]. In Figure 3.17, two common learning rate schedules are shown. Both of these approaches have been investigated. They worked well in practice, but step decay was sufficient and easier to control compared to the cyclic learning rate procedure.



(a) Step-decay **(b)** Cyclic

Figure 3.17: Strategies for changing the learning rate during training. In **a)**, the initial learning rate is 0.01, and is set to decrease every 8th epoch by a factor of 0.75 until reaching the minimum learning rate 1e-5. In **b)**, the learning rate oscillates between the initial learning rate and the minimum learning rate.

3.7 Model evaluation

Once the hyperparameters of the models have been determined we want to compare and evaluate them. It is computationally infeasible to run all models a large number of times, yet we want to make sure that the results are robust and not simply random. Hence, we perform five individual training runs of each model, recording the mean and standard deviation for each. The random seed is different for each run, which introduces randomness for the initial weights and shuffling of the data in the batches.

For the runs, 1% of all labelled data is used, leading to 117 964 training samples and 39 320 test and validation samples. We make use of early stopping, breaking the training process once the validation loss has not improved within 10 epochs. Data augmentation of the training samples is performed, utilizing rotations of 90° and mirroring, leading to an 8-fold increase in unique samples. Some additional training information is also recorded e.g. number of epochs, min-loss, and training time for more model insights.

For the standard 2DCNN, we look into all (n_x, n_y) neighborhoods, whereas for the other CNN architectures we disregard the largest (n_x, n_y) neighborhood, i.e. (81,81), due to computational limitations. All in all, there are 30 standard 2DCNN models and 20 other CNN models. This means that we in total have 50 models to evaluate, which summarizes to 250 individual training runs.

After one or two candidate model(s) have been identified in accordance with the above scheme, training is performed without the restriction of early stopping. The sample size is increased to 5% of all labelled data to maximize performance. Also, more data augmentation is performed by introducing intensity perturbations, where each sample has a probability of its pixel intensities being slightly shifted. The model showing the best validation mIoU scores in this step will be our final model. This model will be used for the segmentation of the entire FIB-SEM data.

3.8 Postprocessing

During training, the predicted labels have been obtained from the scores that are output from the CNNs by thresholding at $T = 0.5$. This impacts mIoU and accuracy since they are computed with binary class labels. The threshold was reasonable during training as we had 50/50 class-balance in the data. However, once predicting on all labelled data, and later on the entire FIB-SEM dataset, this needs to be tuned. One could assume that $T < 0.5$, since the percentage of pores and solid are approximately 32 % and 68 % in the entire dataset.

Whilst we know there exists spatial correlation in the neighborhood (n_x, n_y), it is not possible to take this into account during training as the model predicts the label of each pixel independently. However, once an entire mask has been obtained, Gaussian smoothing can be performed on the score array. Gaussian smoothing in 2 dimensions is performed by a convolution with a kernel defined as

$$G(x, y) = \frac{1}{2\pi\sigma_{xy}} e^{-\frac{x^2+y^2}{\sigma_{xy}^2}} \tag{3.3}$$

where σ_{xy} is the standard deviation. By convolution with the Gaussian kernel on the raw score, the predicted mask becomes less noisy, having more smooth edges. These shapes are more in agreement with the actual shapes of the porous structures.

The optimal settings for the two parameters T and σ_{xy} are found via a grid search, optimized on the validation mIoU using all labelled data. How many rounds to do the smoothing and thresholding is also investigated. In Table 3.6, the combinations in the grid search can be found. The grid for T and σ_{xy} constitute 90×25 different combinations. We search for small objects that are not connected to the rest of the

Table 3.6: Hyperparameters in the gridsearch of the postprocessing parameters.

Hyperparameter	**Range**
Threshold(T)	$\{0.30, 0.3017, \ldots, 0.45\}$
σ_{xy}	$\{0.5, 0.6875, \ldots, 5\}$
Rounds	$\{1, 2\}$

structure, both isolated pores and isolated pieces of solid, as both of these types are likely the result of classification noise. A threshold for the number of connected pixels are investigated in $\{0, 100, 200\}$ connected pixels.

3.9 Segmentation of the entire FIB-SEM data

Once the very best model has been found, it is time to perform segmentation of the entire FIB-SEM data. Some computational aspects of this process need to be considered. Recall that the total number of pixels to classify per HPC volume is $2247 \times 3372 \times 200 = 1\,515\,376\,800$. For each of these pixels, we will need to extract neighborhoods (n_x, n_y, n_z) according to the best model input. The neighborhoods will be extracted via the sliding window technique, traversing the entire FIB-SEM data with one pixel as the step size. In Figure 3.18, the method is visualized.

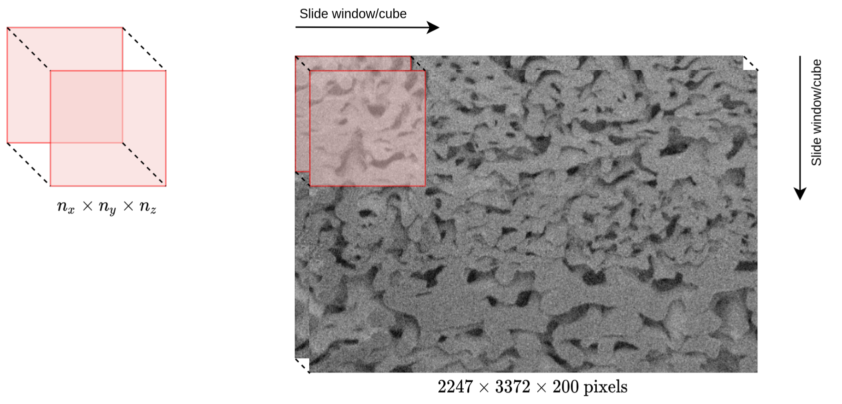

$n_x \times n_y \times n_z$
Slide window/cube
Slide window/cube
$2247 \times 3372 \times 200$ pixels

Figure 3.18: Sliding window technique is utilized when extracting neighborhoods (n_x, n_y, n_z) for pixel-wise classification.

The amount of data needed to be extracted directly impacts the speed of the segmentation. For the method to be applicable in practice we want a method that is sufficiently fast. A slight decrease in classification performance might be warranted in case of a very long computational time. We will investigate the computational workload for different neighborhoods and present our findings.

For the segmentation, we use the final model along with postprocessing parameters that are optimized w.r.t validation mIoU. The metric of interest once segmenting the entire FIB-SEM data is the porosity. Our estimated porosities will be compared to the values obtained from manual segmentation for the manually labelled regions as well as with the expected values for the entire datasets.

3.10 Implementation details

The code is implemented in Python 3.7, utilizing the TensorFlow framework with Keras as Application Programming Interface (API). Tensorflow is an end-to-end open-source machine learning platform developed by Google that allows for fast tensor computations [12]. Keras is used for structuring the neural networks, allowing the user to add layers, optimizers and other hyperparameters [13]. Some standard Python libraries, e.g. NumPy, pandas, scikit-learn, and matplotlib were also utilized. Minor details of the code were implemented in MATLAB, due to the original work being implemented there. Training of the different CNN architectures was carried out on the graphics processing units (GPUs) NVIDIA Titan V and NVIDIA Titan XP. In Table 3.7, additional computer specifications are shown.

Table 3.7: Computer specifications.

GPU 0	NVIDIA Titan V
GPU 1	NVIDIA Titan XP
CPU	Intel Xeon CPU E5-2699 v4 @ 2.20GHz with 88 cores.
System memory	256 GB
OS	Ubuntu 18.04

4
Results

In Section 4.1, the result of the hyperparameter optimization is shown. In Section 4.2, the different models are compared and a few candidate models are identified. In Section 4.3, a final training is performed with the best model. In Section 4.4, the result of predicting all labelled data with and without postprocessing for the best model is shown. In Section 4.5, we show the results (in terms of porosity) for segmentation of the entire FIB-SEM dataset and discuss some computational considerations.

4.1 Hyperparameter optimization

In this section, the result from the hyperparameter optimization is summarized. We present the network architectures for the standard 2DCNN and the 3DCNN. We do not disclose all hyperparameters for the rest of the CNNs herein, but some can be found in the appendix, see Appendix A.

In Table 4.1, the general architecture for the standard 2DCNN is shown. Generally, models with larger neighborhoods as input have more complex architectures, entailing deeper networks with more convolution and max-pooling operations. As the input get larger more dropout is needed in order to regularize the models. In Table 4.2, the general architecture for the 3DCNN is shown. A structure similar to the standard 2DCNN can be seen. Models with larger neighborhoods as input have more complex architectures. The filter size is decreased from 16 to 8 and the dropout is set to 0.5 for all architectures. A 3DCNN is more prone to overfitting in every step, hence it is reasonable that it performs better with less tuneable weights.

Table 4.1: General architecture for the standard 2DCNNs.

	Layers					
Model (n_x, n_y, n_z)	**Conv.**	**Dense.**	**Max-pool.**	**#Filters***	**Neurons**	**Dropout**
2DCNN (11,11, all)	4	2	2	{16,32}	{128,64}	0.1
2DCNN (33,33, all)	6	2	3	{8,16,32}	{128,64}	0.25
2DCNN (49,49, all)	8	2	4	{16,32,64,80}	{128,64}	0.3
2DCNN (65,65, all)	8	2	4	{16,32,64,80}	{128,64}	0.3
2DCNN (81,81, all)	8	2	4	{16,32,64,80}	{128,64}	0.5

*Convolution layers come in pairs with the same filter size.

In Table 4.3, we show the optimization parameters for all CNNs. The initial learning rate varies between 0.0025 and 0.0075, the momentum is either 0.90 or 0.95 and the batch size varies between 64 and 256. All 2DCNN architectures utilize a 3×3 kernel,

Table 4.2: General architecture for the 3DCNNs.

Model (n_x, n_y, n_z)	Layers **Conv.**	**Dense.**	**Max-pool.**	**#Filters***	**Neurons**	**Dropout**
3DCNN (11,11,11)	4	2	2	{8,16}	{64,32}	0.5
3DCNN (33,33,11)	8	2	3	{8,16,24,32}	{64,32}	0.5
3DCNN (49,49,11)	8	2	3	{8,16,24,32}	{128,64}	0.5
3DCNN (65,65,11)	10	2	4	{8,16,24,32,48}	{128,64}	0.5

*Convolution layers come in pairs with the same filter size.

all 3DCNNs a 3 × 3 × 3 kernel, and all 1DCNNs a 5 × 1 kernel. In Table 4.4, we disclose the number of trainable parameters for the different CNN architectures. Larger neighborhoods as input results in more trainable parameters. The architectures are quite simple (in the deep-learning context) and do not require tuning of millions of parameters. The MultichannelCNN has the most parameters in total, whereas the 1DCNN has the fewest. This is not surprising, as the MultichannelCNN has three times the input of a standard 2DCNN for example.

In Figure 4.1, we show one standard 2DCNN architecture with neighborhoods $(n_x, n_y, n_z) = (81, 81, 3)$ as input. By considering this image as a template, it is easy to visualize what the rest of the standard 2DCNNs architectures look like. In many instances, the only difference is in the dropout probability. In Figure 4.2, we show one 3DCNN architecture with neighborhoods $(n_x, n_y, n_z) = (65, 65, 11)$ as input.

Table 4.3: Optimization parameters for all CNNs.

Model	**Optimizer**	**Loss**	**Initial lr**	**Momentum**	**Batch size**
Standard 2DCNN	*'SGDm'*	log-loss	0.005	0.95	128
3DCNN	*'SGDm'*	log-loss	0.005	0.90	256
TriplanarCNN	*'SGDm'*	log-loss	0.005	0.95	128
MultichannelCNN	*'SGDm'*	log-loss	0.0075	0.95	64
MV2DCNN	*'SGDm'*	log-loss	0.0075	0.95	128
1DCNN	*'SGDm'*	log-loss	0.0025	0.90	256

Table 4.4: Number of trainable parameters for all CNNs.

Model name	**Trainable parameters**				
(n_x, n_y)	(11,11)	(33,33)	(49, 49)	(65,65)	(81,81)
Standard 2DCNN*	42,641	92,745	277,681	349,361	441,521
3DCNN	22,729	129,913	279,545	275,465	
TriplanarCNN	52,337	245,585	433,361	468,177	
MultichannelCNN	56,657	256,241	448,625	488,049	
MV2DCNN	41,201	92,025	276,241	347,921	
1DCNN	19,281	70,321	101,201	117,841	

*We consider $n_z = 11$ for simplicity.

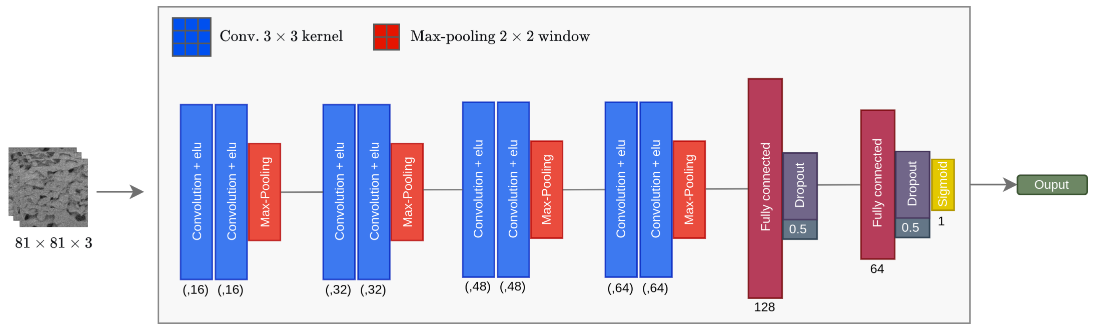


Figure 4.1: CNN architecture for the standard 2DCNN with neighborhoods $(n_x, n_y, n_z) = (81, 81, 3)$ as input.

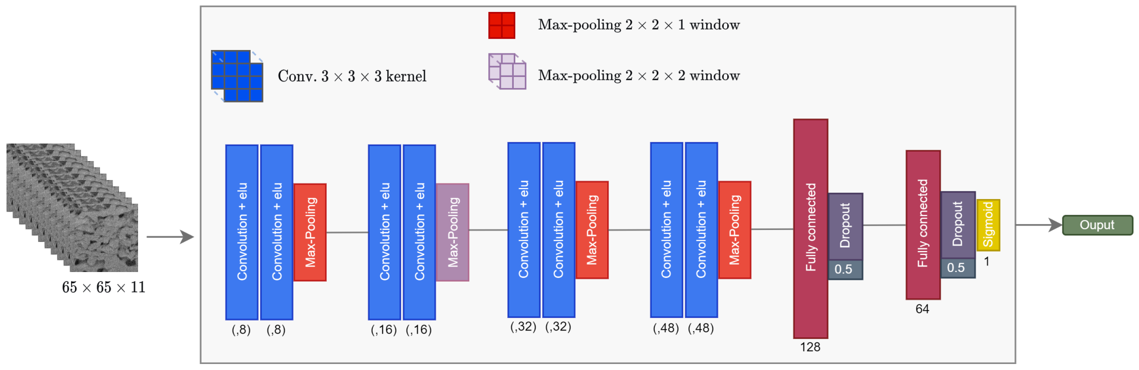


Figure 4.2: CNN architecture for the 3DCNN with neighborhoods $(n_x, n_y, n_z) = (65, 65, 11)$ as input.

4.2 Model comparisons

In this section, the different models are compared. The standard 2DCNNs results are shown In Section 4.2.1, and all other CNNs are shown together in the section 4.2.2. Generally, the performance of the standard 2DCNNs exceeds that of the other CNNs. In Section 4.2.3, we investigate the top three standard 2DCNN models and top three of the other CNN models in more detail, by showing training plots, classification performance, and predictions of a single mask.

The models are compared based on their validation mIoU. As a reference point for the mIoU, a classification completely at random represents a mIoU score of ≈33 %, given that we have 50/50 class-balance.

4.2.1 Standard 2DCNN

In Table 4.5, the result of the five runs for all standard 2DCNN models is summarized. It appears beneficial to have larger (n_x, n_y) neighborhoods as input, whereas a larger n_z neighborhood seems to have less impact on the classification performance. Increasing the (n_x, n_y) neighborhood from (11,11) to (33,33) gives the largest boost in performance. For the top models, the validation loss settles somewhere between 0.251-0.255. The standard deviation of all metrics is low in all models, indicating that the models are robust with respect to the initial parameter values, the random shuffling of the batches, and the random data augmentations.

The results from Table 4.5 are also shown in Figure 4.3. The classification performance levels out somewhere around $(n_x, n_y) = (49, 49)$ for most of the models. However, there is a slight increase in classification performance for all models after this point as well. A reasonable assumption is that the performance will continue to increase for larger (n_x, n_y) neighborhoods, however, how much is hard to tell. The standard 2DCNNs with $n_z \in \{1, 3\}$ appear to benefit the most when increasing the (n_x, n_y) neighborhoods. It is likely that these models need more information, and loose predictability power when considering too small neighborhoods as input.

As previously stated, there is no strong trend showing whereas more information in the n_z dimension boosts performance. Presumably, when the input gets more complex, the model has greater difficulty extracting the necessary information. The data is simply more noisy, and a good representation of the weights is harder to achieve. It is hypothesized that more complex models might be needed for such inputs.

Table 4.5: Training results of the five runs for all **standard 2DCNNs** showing validation mIoU % (mean ± std), validation loss (mean ± std) and training time (mean ± std). Results sorted according to the validation mIoU.

Model name	**val mIoU**	**val Loss**	**Training time(s)**
2DCNN(81,81,3)	80.08 ± 0.29	0.251 ± 0.002	1712 ± 287
2DCNN(81,81,11)	80.04 ± 0.19	0.253 ± 0.001	3187 ± 288
2DCNN(81,81,7)	79.95 ± 0.07	0.253 ± 0.002	2558 ± 366
2DCNN(81,81,1)	79.81 ± 0.13	0.252 ± 0.001	1048 ± 165
2DCNN(65,65,3)	79.79 ± 0.27	0.253 ± 0.002	1100 ± 165
2DCNN(65,65,5)	79.78 ± 0.31	0.254 ± 0.002	1422 ± 192
2DCNN(81,81,5)	79.76 ± 0.30	0.253 ± 0.002	1797 ± 182
2DCNN(65,65,11)	79.72 ± 0.33	0.255 ± 0.003	2461 ± 625
2DCNN(65,65,1)	79.69 ± 0.22	0.255 ± 0.001	725 ± 89
2DCNN(81,81,9)	79.65 ± 0.32	0.252 ± 0.003	2990 ± 789
2DCNN(65,65,7)	79.62 ± 0.24	0.255 ± 0.002	1929 ± 375
2DCNN(49,49,11)	79.47 ± 0.18	0.252 ± 0.001	1688 ± 213
2DCNN(49,49,3)	79.35 ± 0.09	0.258 ± 0.001	1066 ± 157
2DCNN(65,65,9)	79.28 ± 0.28	0.256 ± 0.001	1593 ± 267
2DCNN(49,49,7)	79.22 ± 0.26	0.259 ± 0.002	1347 ± 247
2DCNN(49,49,5)	79.10 ± 0.15	0.260 ± 0.002	920 ± 139
2DCNN(49,49,9)	79.04 ± 0.09	0.257 ± 0.002	1412 ± 284
2DCNN(49,49,1)	78.94 ± 0.08	0.265 ± 0.001	711 ± 90
2DCNN(33,33,11)	78.76 ± 0.20	0.262 ± 0.001	1386 ± 187
2DCNN(33,33,3)	78.54 ± 0.08	0.272 ± 0.001	678 ± 34
2DCNN(33,33,5)	78.47 ± 0.09	0.270 ± 0.001	889 ± 141
2DCNN(33,33,7)	78.42 ± 0.07	0.269 ± 0.001	890 ± 206
2DCNN(33,33,9)	78.31 ± 0.20	0.267 ± 0.001	968 ± 245
2DCNN(33,33,1)	77.51 ± 0.17	0.287 ± 0.001	575 ± 163
2DCNN(11,11,9)	75.13 ± 0.18	0.323 ± 0.003	596 ± 181
2DCNN(11,11,11)	75.05 ± 0.30	0.324 ± 0.003	518 ± 177
2DCNN(11,11,7)	74.87 ± 0.38	0.329 ± 0.004	514 ± 176
2DCNN(11,11,5)	74.65 ± 0.19	0.330 ± 0.001	577 ± 122
2DCNN(11,11,3)	74.33 ± 0.08	0.337 ± 0.002	478 ± 139
2DCNN(11,11,1)	72.28 ± 0.18	0.360 ± 0.002	391 ± 72

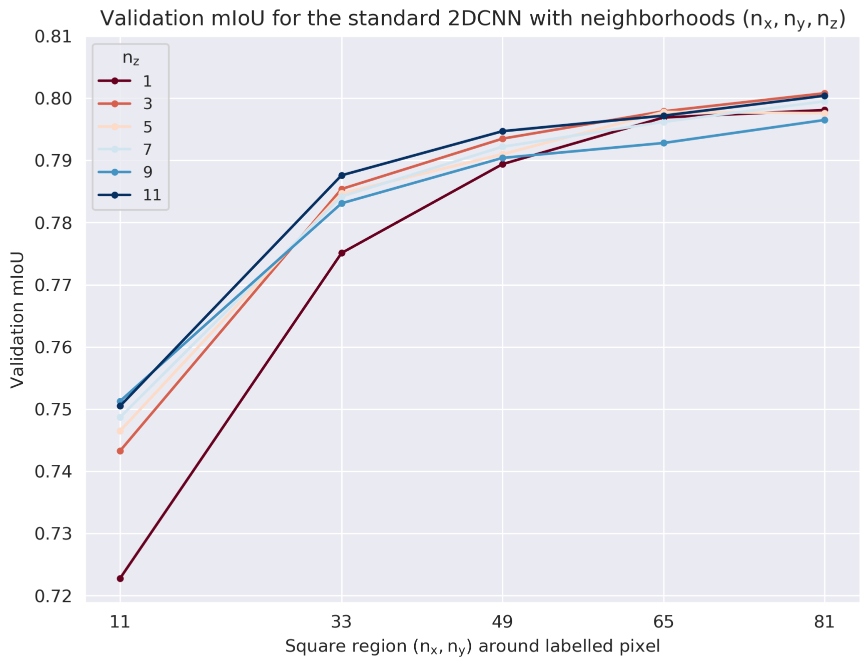


Figure 4.3: Mean validation mIoU for every **standard 2DCNN** model.

4.2.2 Other CNN models

In Table 4.6, the result of the five runs for all other CNN models is summarized. The results are more varying in comparison to the standard 2DCNN. The best performing models are the 3DCNN, TriplanarCNN, and MultichannelCNN. Yet again, larger (n_x, n_y) neighborhoods appear to be beneficial. The validation loss for the top models is between 0.26-0.265, somewhat higher than for the top standard 2DCNNs.

In Figure 4.4, the contrast between the different models is also shown in a plot. All models gain performance by increasing the (n_x, n_y) neighborhoods. The 1DCNN performs surprisingly well considering its limitations, surpassing the MV2DCNN in classification performance. The MV2DCNN performance increases as larger regions are used but from a very low level. The difference between the MultichannelCNN, 3DCNN, and TriplanarCNN are very subtle; they all perform well in regards to classification performance. However, none of the models were able to achieve mIoU values of 0.80 as for the best standard 2DCNNs. Worth noting is that even when considering the same neighborhoods, i.e. (65,65,11), the standard 2DCNN outperforms the best other CNNs.

Table 4.6: Training results of the five runs for all **other CNNs** showing validation mIoU % (mean ± std), validation loss (mean ± std) and training time (mean ± std). Results sorted according to the validation mIoU.

Model name	val mIoU	val Loss	Training time(s)
3DCNN(65,65,11)	79.60 ± 0.29	0.263 ± 0.002	9147 ± 1166
TriplanarCNN(65,65,11)	79.39 ± 0.34	0.265 ± 0.004	1934 ± 170
MultichannelCNN(65,65,11)	79.26 ± 0.43	0.262 ± 0.005	3561 ± 465
TriplanarCNN(49,49,11)	79.16 ± 0.10	0.269 ± 0.002	1321 ± 198
MultichannelCNN(49,49,11)	79.13 ± 0.28	0.260 ± 0.005	3142 ± 623
3DCNN(49,49,11)	79.06 ± 0.34	0.262 ± 0.002	4857 ± 620
MultichannelCNN(33,33,11)	78.52 ± 0.19	0.267 ± 0.002	2136 ± 87
3DCNN(33,33,11)	78.40 ± 0.14	0.268 ± 0.002	2298 ± 567
TriplanarCNN(33,33,11)	78.37 ± 0.07	0.279 ± 0.001	835 ± 117
1DCNN(49,49,5)	77.09 ± 0.17	0.292 ± 0.001	1981 ± 180
1DCNN(65,65,5)	77.06 ± 0.11	0.291 ± 0.001	3832 ± 357
1DCNN(33,33,5)	76.59 ± 0.17	0.301 ± 0.001	1161 ± 130
3DCNN(11,11,11)	75.25 ± 0.14	0.320 ± 0.001	292 ± 51
MV2DCNN(65,65,11)	74.92 ± 0.07	0.314 ± 0.001	2000 ± 496
MV2DCNN(49,49,11)	74.31 ± 0.25	0.322 ± 0.001	1366 ± 241
MV2DCNN(33,33,11)	73.86 ± 0.20	0.332 ± 0.001	915 ± 54
MultichannelCNN(11,11,11)	73.85 ± 0.57	0.342 ± 0.005	647 ± 181
1DCNN(11,11,5)	73.61 ± 0.17	0.348 ± 0.001	263 ± 69
TriplanarCNN(11,11,11)	73.40 ± 0.36	0.350 ± 0.004	470 ± 60
MV2DCNN(11,11,11)	71.20 ± 0.14	0.373 ± 0.001	403 ± 69

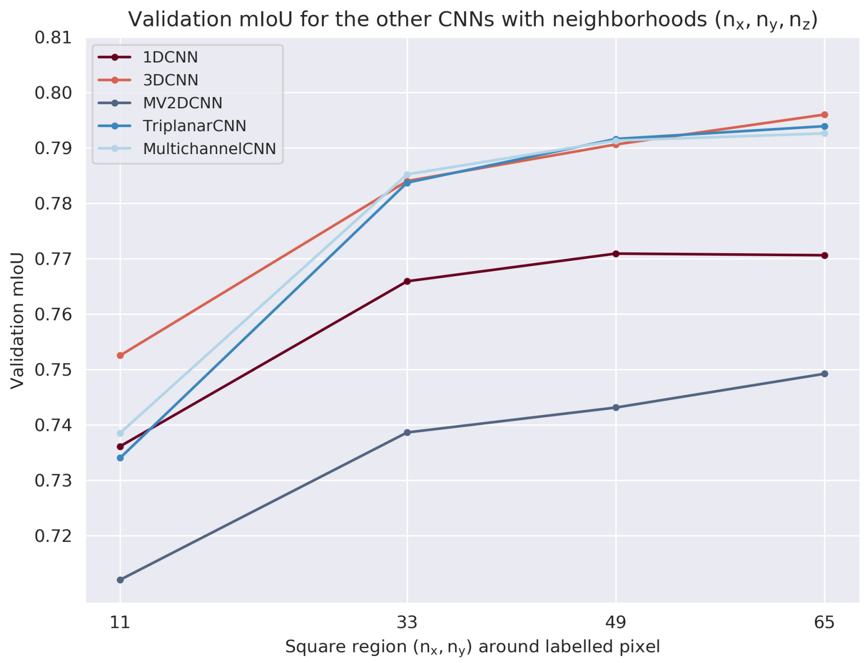


Figure 4.4: Mean validation mIoU for all **other CNN** models.

4.2.3 Comparison between the best CNN models

Based on the model comparison, we show additional information for the top three standard 2DCNN models and the top three of the other CNN models. In Table 4.7, we show their performance in terms of mIoU and accuracy on the train, validation, and test sets. The standard 2DCNN having neighborhoods $(n_x, n_y, n_z) = (81, 81, 3)$ as input performs the best overall, having both the highest validation and test scores. All models generalize well to both validation and test sets, with slightly higher training scores as to be expected.

In Figure 4.5, a boxplot for the six CNN models is shown. Although 5 observations is a very small sample size for a boxplot, some interesting trends can be seen. All standard 2DCNNs have observations that are considered outliers, and the interquartile range(IQR) of the other CNNs is much greater than IQR of the standard 2DCNNs.

In Figure 4.6, training plots for each of the models are illustrated by showing mean log-loss and 95 % confidence intervals based on the five runs. The learning curves appear to be good for most of the models. The 3DCNN(65,65,11) and 2DCNN(81,81,11) are slightly overfitted, but early stopping has been applied, and the best model is saved once the validation loss is minimized. The MultichannelCNN(65,65,11) has the highest variation between the runs, indicating that the initial conditions have a greater impact.

In Figure 4.7, we show an automatic segmentation of one mask (256×256 pixels) for each of the models. These images are for illustration purposes and we cannot draw any conclusion about the segmentation performance overall. However, we can see some interesting differences between the models. Three of the models; 2DCNN(81,81,3), TriplanarCNN(65,65,11), and 2DCNN(81,81,7) detect a small porous area in the right bottom corner, whereas the others do not. Other noticeable differences are the estimated pore sizes. The 3DCNN(65,65,11) performs the best overall, having the highest mIoU scores and closest predicted porosity with respect to the mask. It is important to mention that this is without postprocessing, and different models might benefit more or less from postprocessing.

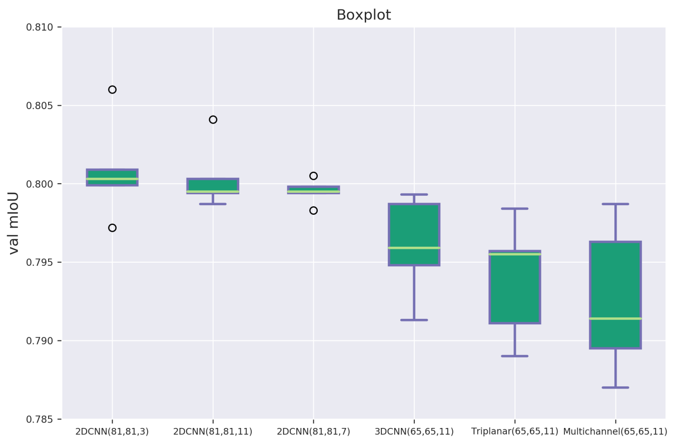
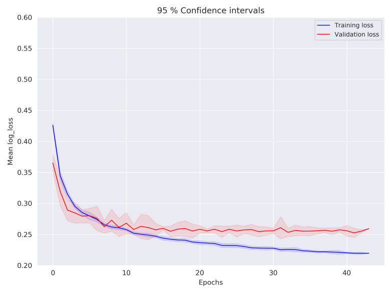


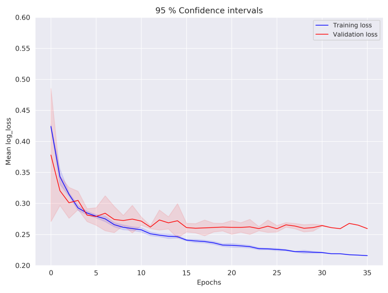
Figure 4.5: Boxplot for the best performing CNN models. Notice that outliers are present in all standard 2DCNN models.

Table 4.7: Top CNN models (with respect to validation mIoU) and their mean scores and standard deviation % (mean ± std) for each individual data set.

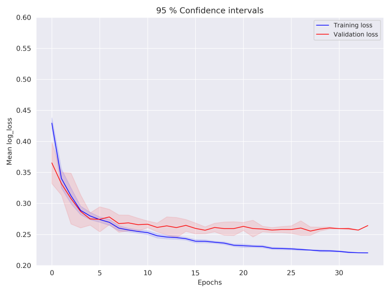
Model name & scores	**Train**	**Val**	**Test**
2DCNN(81,81,3)			
mIoU (%)	82.17 ± .22	80.08 ± .29	80.53 ± .13
Accuracy (%)	90.21 ± .26	88.94 ± .18	89.22 ± .13
2DCNN(81,81,11)			
mIoU (%)	82.67 ± .24	80.04 ± .19	79.82 ± .22
Accuracy (%)	90.51 ± .14	88.92 ± .12	88.78 ± .14
2DCNN(81,81,7)			
mIoU (%)	82.02 ± .27	79.95 ± .07	79.92 ± .20
Accuracy (%)	90.12 ± .17	88.86 ± .04	88.84 ± .12
3DCNN(65,65,11)			
mIoU (%)	82.21 ± .28	79.60 ± .29	79.47 ± .13
Accuracy (%)	90.24 ± .17	88.64 ± .18	88.56 ± .08
TriplanarCNN(65,65,11)			
mIoU (%)	81.25 ± .29	79.39 ± .34	79.48 ± .29
Accuracy (%)	89.66 ± .13	88.51 ± .21	88.56 ± .18
MultichannelCNN(65,65,11)			
mIoU (%)	81.92 ± .60	79.26 ± .43	78.85 ± .37
Accuracy (%)	90.07 ± .36	88.43 ± .27	88.18 ± .23



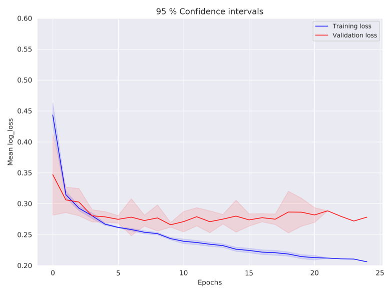
(a) *2DCNN (81,81,3)*



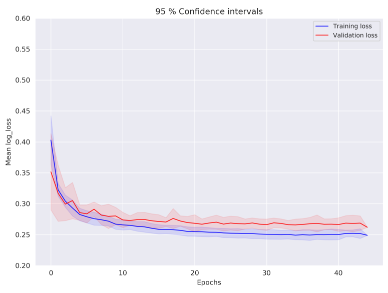
(b) *2DCNN (81,81,11)*



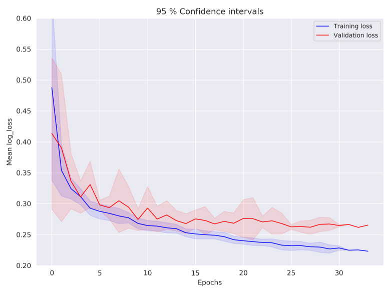
(c) *2DCNN (81,81,7)*



(d) *3DCNN (65,65,11)*

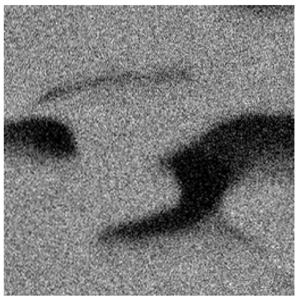


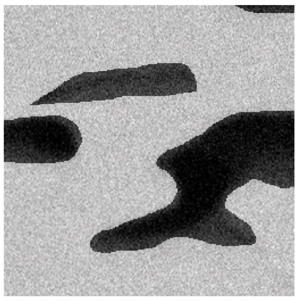
(e) *TriplanarCNN (65,65,11)*



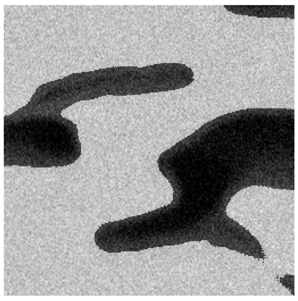
(f) *MultichannelCNN (65,65,11)*

Figure 4.6: Mean log-loss with 95 % confidence intervals based on the five runs for the top CNN models. Showing training loss (blue) and validation (loss) red.

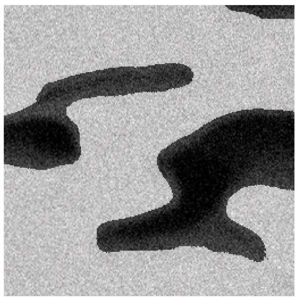




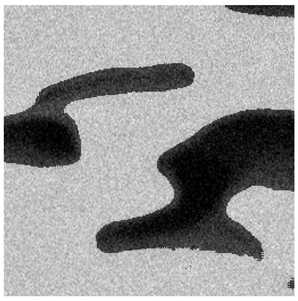
(b) Porosity = 0.2592



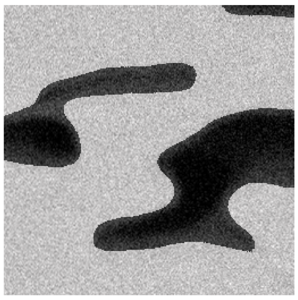
(c) Porosity= 0.2955
mIoU = 0.8728



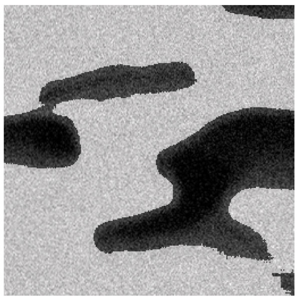
(d) Porosity = 0.2966
mIoU = 0.8670



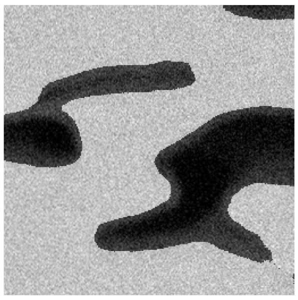
(e) Porosity = 0.2882
mIoU = 0.8810



(f) Porosity= 0.2815
mIoU = 0.8971



(g) Porosity = 0.3043
mIoU = 0.8661



(h) Porosity = 0.3000
mIoU = 0.8580

Figure 4.7: Predictions of one mask for different CNN models. The binary predictions are obtained by thresholding the score with $T = 0.5$, i.e. no postprocessing. As a cautionary remark, these images show only the performance on a single mask and cannot be generalized to the entire FIB-SEM dataset.

4.3 Model selection

Based on the model evaluation we conclude that the standard 2DCNN with neighborhoods $(n_x, n_y, n_z) = (81, 81, 3)$ performs the best. The fact that n_z is rather small for the best-performing model simplifies final training, as less data is needed to be extracted and stored. Training is performed with 5% of all labelled data, resulting in 589 820 training samples and 196 606 validation samples.

In Figure 4.8, the learning curve for the final training is visualized. In Table 4.8, the training scores for the best model is shown. The loss is slightly lower than during the model evaluation (at least on average). This could indicate that more data are beneficial for the learning, albeit to a very small degree.

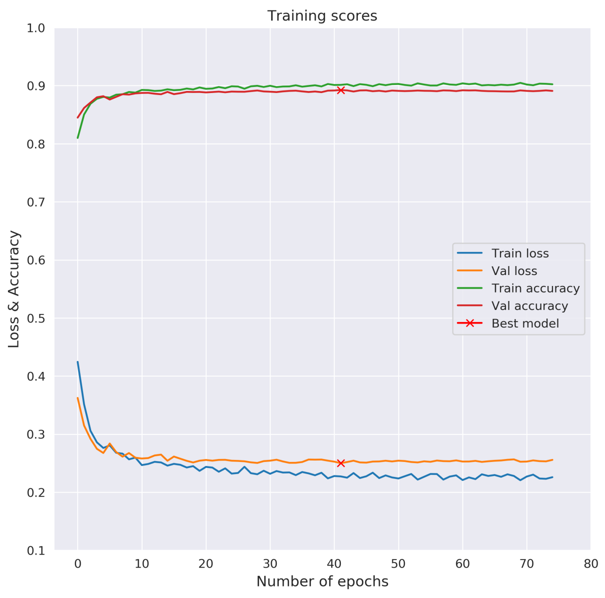


Figure 4.8: Learning curve for the final training, showing binary cross-entropy loss and accuracy.

Table 4.8: Training scores for the final training of the model.

	Train	**Val**	**Test**
mIoU	0.8279	0.8059	0.8034
Accuracy	0.9058	0.8925	0.8910
Loss	0.2272	0.2500	0.2504

In Table 4.9, we show confusion matrices for the train and validation predictions. The confusion matrices are shown in percentage, where all values in the table sum up to 100%. Recall that we have 50/50 class balance in the data, hence a value of 50 % in the diagonal is considered a perfect classification for the given class. The model missclassifies pores and solids to almost the same degree, which is a nice property and reasonable as we had class-balance during the training. One could argue that the validation predictions have a more even spread in its missclassifications, however, the percentage of correct classifications are lower.

Table 4.9: Confusion matrices for the train and validation scores.

Train

		Predicted	
		Pores(0)	Solids(1)
Actual	Pores(0)	45.688 %	4.3120 %
	Solids(1)	5.1030 %	44.897 %

Validation

		Predicted	
		Pores(0)	Solids(1)
Actual	Pores(0)	44.527 %	5.4730 %
	Solids(1)	5.2730 %	44.727 %

4.4 Prediction of all labelled data with best model

In Table 4.10, the result of predicting all labelled data using the best model is shown. The automatic segmentation shows good agreement with the manual segmentation. The HPC45 sample appears to be the hardest to classify, whereas HPC30 appears to be the easiest. For the HPC45 sample, the variability between the manual porosities is the greatest. Hence, this sample is more heterogeneous, and perhaps an explanation to why it is the hardest to classify. The model overestimates the porosity for all HPC samples, which indicates that postprocessing is needed. The threshold $T = 0.5$ is not well suited for unbalanced data, which is especially evident for the HPC22 and HPC30 samples.

Table 4.10: Classification performance on all labelled data using the best CNN model **before** postprocessing. Table showing the result of classifying each HPC sample and the combined result. Threshold $T = 0.5$ is used to obtain the binary predictions.

	Train	**Val**	**Test**
HPC22			
mIoU	0.7716	0.7617	0.7645
Accuracy	0.9041	0.8986	0.9057
Porosity % (manual)	22.07	22.00	20.42
Porosity % (automatic)	24.95	25.49	22.92
HPC30			
mIoU	0.8468	0.8308	0.8101
Accuracy	0.9277	0.9201	0.9084
Porosity % (manual)	29.76	29.14	29.27
Porosity % (automatic)	33.12	33.44	32.62
HPC45			
mIoU	0.7777	0.7521	0.7581
Accuracy	0.8753	0.8586	0.8639
Porosity % (manual)	44.17	49.62	42.20
Porosity % (automatic)	50.63	52.90	46.67
Combined result			
mIoU	0.8059	0.7910	0.7846
Accuracy	0.9024	0.8924	0.8927

4.4.1 Postprocessing

In Table 4.11, the results of the grid search for the best postprocessing parameters are shown. Most notable is that the obtained threshold T is lower than 0.5. This was expected due to not having class-balance in the entire FIB-SEM dataset.

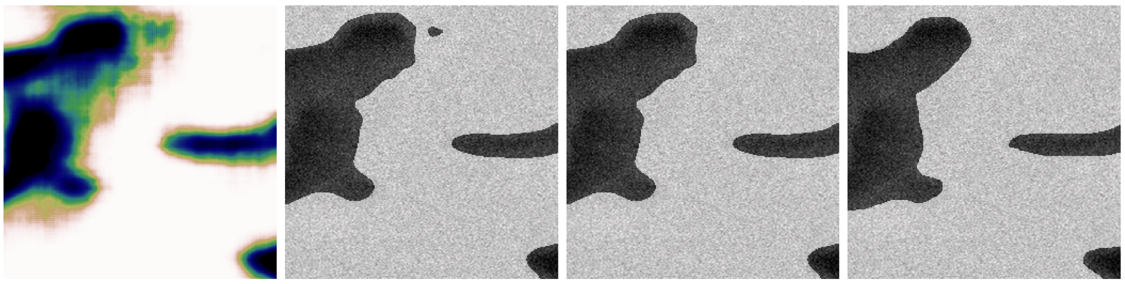
In Figure 4.9, one example of the postprocessing step is shown, illustrating how the Gaussian smoothing and thresholding together with the removal of small connected objects are performed for one mask.

In Table 4.12, the result after postprocessing is shown. It is evident that the mIoU, accuracy and estimated porosities improves w.r.t the manual segmentations. Recall that the porosities determined by manual segmentation for the three sets are (mean and 95 % confidence intervals) 21.72 ([20.28, 23.17])%, 29.54 ([27.70, 31.38])% and 44.86 ([42.23, 47.49])%. Based on the automatic segmentation, the estimated porosities are (mean and 95 % confidence intervals) 20.34 ([19.04, 21.94])%, 30.33 ([28.34, 32.33])% and 45.62 ([43.24, 48.00])%. From these observations, we conclude that the manual and automatic segmentations are not significantly different in terms of porosity. It is worth noting that the manual segmentations constitute less than 0.5% of all data, hence these results might not be completely representative in terms of porosity for the entire FIB-SEM dataset.

In Figure 4.10, we show a comparison between manual and automatic segmentation for one square region from each HPC sample. There is good agreement overall, with some misclassifications of both types. For the HPC45 mask, the model appears to misclassify pores as solid to a greater extent, whereas for the HPC30 mask the opposite is true. The pore regions in the automated segmented masks are generally much smoother than those in the manually segmented masks. This is a consequence of how the model produces an output, as can be seen in the raw scores in Figure 4.10 (b), (f), and (j). The raw scores are in a sense almost symmetrical, and the predictions get less confident the closer they are to a pore edge. Consequently, the predicted masks cannot have very angular or rugged shapes.

Table 4.11: Best postprocessing parameters, optimized w.r.t the validation mIoU using all labelled data.

σ_{xy}	**Threshold(T)**	**Smooth rounds**	**Min size**	**Connectivity**
3.875	0.3994	1	100	2



(a) Score-array (b) Gaussian smoothing and thresholding. (c) Removing small connected objects. (d) Mask

Figure 4.9: On the score-array in **a)** gaussian smoothing with $\sigma_{xy} = 3.875$ followed by thresholding with $T = 0.3994$ is performed, showing the result in **b)**. In **c)**, small connected objects are removed, in this case one small connected object is removed. In **d)**, the mask is shown.

Table 4.12: Classification performance on all labelled data using the best CNN model **after** postprocessing, using parameters in Table 4.11. Table showing the result of classifying each HPC sample and the combined result.

	Train	**Val**	**Test**
HPC22			
mIoU	0.7790	0.7697	0.7703
Accuracy	0.9132	0.9083	0.9139
Porosity % (manual)	22.07	22.00	20.42
Porosity % (automatic)	20.70	21.23	19.10
HPC30			
mIoU	0.8563	0.8386	0.8176
Accuracy	0.9341	0.9259	0.9146
Porosity % (manual)	29.76	29.14	29.27
Porosity % (automatic)	30.40	30.60	29.98
HPC45			
mIoU	0.7995	0.7634	0.7692
Accuracy	0.8895	0.8659	0.8724
Porosity % (manual)	44.17	49.62	42.20
Porosity % (automatic)	46.12	48.56	42.19
Combined result			
mIoU	0.8194	0.8001	0.7930
Accuracy	0.9123	0.9000	0.9003

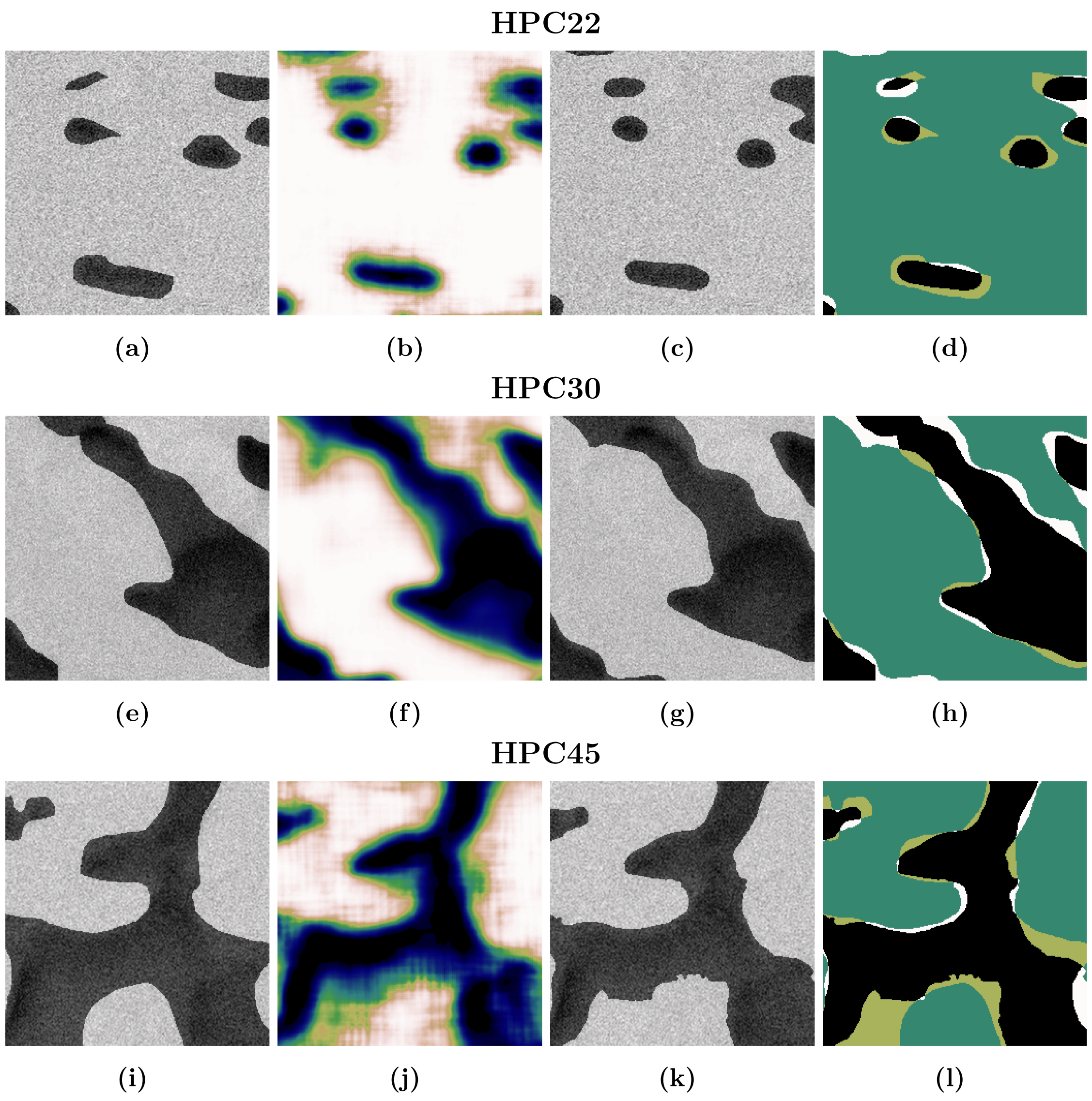


Figure 4.10: Comparison between manual and automatic segmentation for one mask from each HPC sample. First column: **(a,e,i)**, the manual segmentation is superimposed on top of the image data, showing pores (black) and solid (grey). Second column: **(b,f,j)**, the raw score is shown. Third column: **(c,g,k)**, the automatic segmentation is superimposed on top of the image data, showing pores (black) and solid (grey). Fourth column: **(d,h,l)**, an overlay of the manual and automatic segmentation is shown, with correctly classified solid (dark-green), correctly classified pores (black), solid incorrectly classified as pores (white), and pores incorrectly classified as solid (light-green)

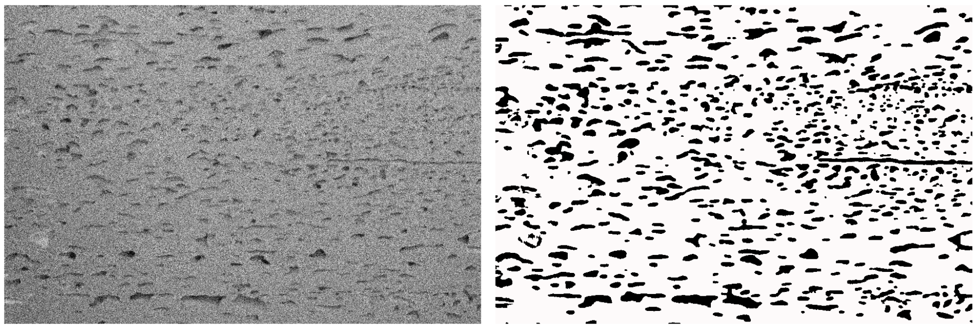
4.5 Segmentation of the entire FIB-SEM dataset

The estimated porosities were determined to be 20.34, 33.51, and 45.75 % by segmentation with the 2DCNN(81,81,3) model. Assuming that the true porosities are 22, 30, and 45 %, these porosities are somewhat wrong. The reason could be difficulties in the segmentation due to the shine-through effect and/or incomplete leaching of the

HPC samples. Assuming independence between cross-sections (which is somewhat false), confidence intervals for the estimated porosities are (mean and 95 % confidence intervals) 20.34 [19.59, 21.08] %, 33.51 [32.98, 34.04] %, and 45.75 [45.06, 46.46].

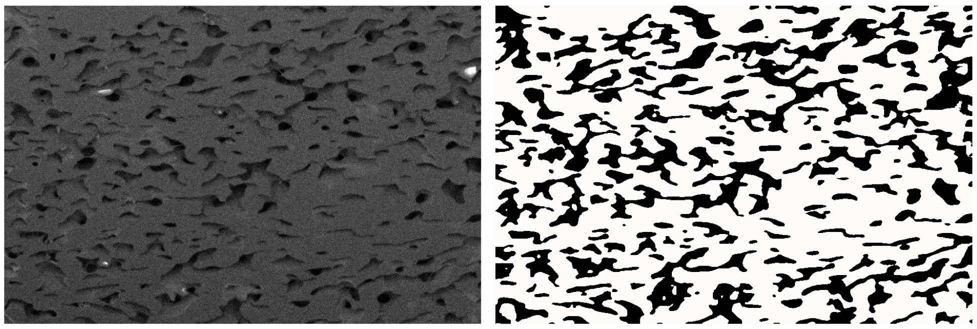
The time required to segment one HPC sample was approximately 38h, utilizing two GPUs simultaneously. In Figure 4.11, three automatic segmentations of a cross-section from each HPC sample are shown. The overall appearance looks to be good, although, by zooming in some minor deviations from the original image can be identified. Note that these images cannot be quantified to a measure for comparison, yet they show that the procedure works in practice.

HPC22



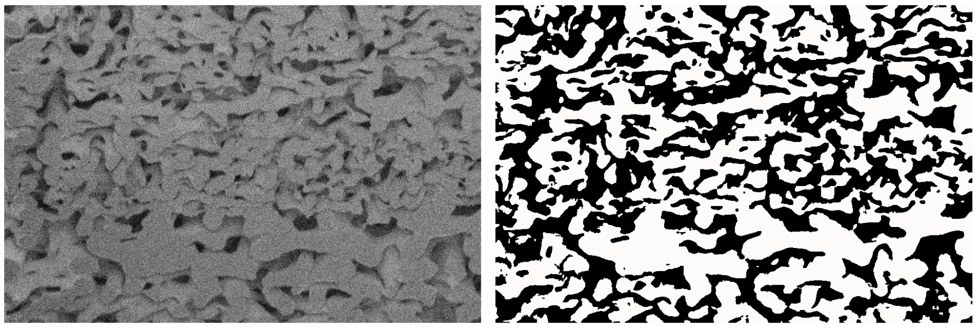
(a) HPC22 cross-section. **(b)** Automatic segmentation.

HPC30



(c) HPC30 cross-section. **(d)** Automatic segmentation.

HPC45



(e) HPC45 cross-section. **(f)** Automatic segmentation.

Figure 4.11: Showing automatic segmentation **(b,d,f)** with pores (black) and solid (white) accompanied by the original images in **(a,c,e)** for comparison.

4.5.1 Computational considerations

In Table 4.13, the data needed to be extracted for the segmentation of one cross-section is calculated. For the segmentation of a complete HPC sample, 200 of these cross-sections need to be segmented. We also show the estimated time for the segmentation of a complete HPC sample, where the estimation is obtained by multiplying the time needed for the segmentation of one cross-section by 200. The predictions have been computed with the standard 2DCNN architecture for the given neighborhood. The code has been executed on an NVIDIA Titan V GPU with 12GB memory. In Figure 4.12, computational considerations are shown as a plot. A somewhat linear pattern can be identified. As more data is needed to be extracted, the segmentation of one cross-section increases linearly. Increasing the sizes of the neighborhoods to e.g. (81,81,9) or (65,65,11) quickly makes the segmentation task troublesome, as memory requirements and segmentation time increase. However, we shall mention that computational limitations might not be an issue for everyone. The code can easily be parallelized to run on multiple GPUs simultaneously, thus drastically decreasing the computational time required for the segmentation.

Table 4.13: The amount of data needed to be extracted in order to segment **one** cross-section for different neighborhoods is calculated. The computational time includes extraction of neighborhoods and prediction. The data is assumed to be stored as 32 bits floating points and the prediction to be executed on NVIDIA Titan V GPU with 12GB memory.

(n_x, n_y, n_z)	Data to be extracted (Gb)*	time/slice (s)	time/HPC sample (h)
(11,11,11)	≈ 38 Gb	≈ 101s	≈ 5.6h
(81,81,1)	≈ 185 Gb	≈ 544s	≈ 30.2h
(49,49,3)	≈ 203 Gb	≈ 500s	≈ 27.8h
(33,33,11)	≈ 338 Gb	≈ 689s	≈ 38.3h
(65,65,3)	≈ 358 Gb	≈ 820s	≈ 45.5h
(81,81,3)	≈ 556 Gb	≈ 1517s	≈ 84.3h
(65,65,7)	≈ 835 Gb	≈ 1718s	≈ 95.4h
(65,65,11)	≈ 1313 Gb	≈ 2369s	≈ 131.6h
(81,81,9)	≈ 1668 Gb	≈ 3190s	≈ 177.2h

*For one cross-section.

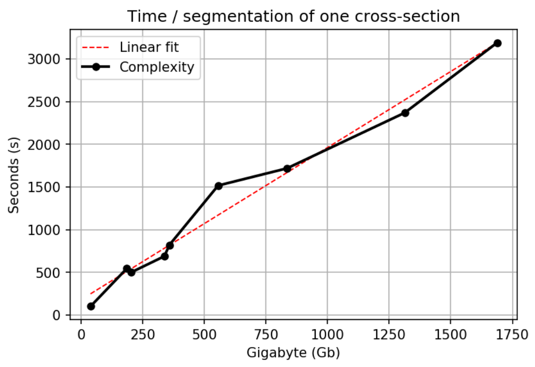


Figure 4.12: Visualization of the data in Table 4.13, showing the time needed for segmentation of **one** cross-section as a function of the amount of data needed to be extracted.

5 Discussion

In this Chapter, we discuss the obtained results further and compare it with related works. We also give ideas to future work and improvements, and discuss the limitations of our proposed method.

5.1 Discussion of results

In Tables 4.5 and 4.6 the results of five runs for all CNN models can be seen. The standard 2DCNN achieved the highest performance overall, having both the lowest validation losses and highest mIoU scores for its top models. The best standard 2DCNN model is the one having neighborhoods $(n_x, n_y, n_z) = (81, 81, 3)$ as input, showing mean validation mIoU of 0.8008. For the other CNN architectures; the 3DCNN, TriplanarCNN, and MultichannelCNN also showed promising results, achieving mean validation mIoU scores of 0.7960, 0.7939, and 0.7926 for its top models. The 1DCNN and MV2DCNN performed worse overall, not surprisingly. However, they were able to achieve mean validation mIoU scores of 0.7709 and 0.7492 for their top models.

More information in (n_x, n_y) appears to benefit all models, whereas more information in n_z seems to have less impact. Certainly, all information that exists in smaller neighborhoods also exists in larger neighborhoods. Networks utilizing more channels as input have the potential of making more advanced connections. However, more kernels are likely needed to appropriately handle all the channels. More kernels would in turn lead to more tuneable parameters, and a network requiring longer training times and a higher risk of overfitting. In this work, we limited the number of kernels to 8 or 16 in the first convolution layer. One could argue that this is too few to take advantage of the information in all channels. Also, n_z is not considered a channel in all networks. The MultichannelCNN and TriplanarCNN both have two paths where n_z is not considered to be a channel. For these networks, the paths where n_x or n_y are considered to be channels, the dimension would be even more "skewed". Then, 16 kernels in the first convolution layer would almost certainly be too few, as we would consider a spatial array with e.g. 49 or 65 channels for the larger neighborhoods.

Treating (n_x, n_y) differently than n_z was established prior to our investigation. The correlation within cross-sections is far greater than between cross-sections due to the way FIB-SEM images are acquired. This is likely the main reason for the superior performance of the standard 2DCNN. The standard 2DCNN distinguishes the (n_x, n_y) neighborhood from n_z appropriately, not allowing n_z to have as great influence. If the data were truly 3D such as in e.g. X-ray CT, it is hypothesized that a 3DCNN would perform the best. Furthermore, the MultichannelCNN or TriplanarCNN might also gain performance in that case [45, 46].

Figure 4.6 shows the training plots for the six investigated models. In AppendixA.2, we also show additional training plots for other models that were not investigated as closely. The learning curves are in general very good, and all models appear to more or less converge sufficiently. Overfitting is only present for the 3DCNN and the largest standard 2DCNN models. A contributing factor is the number of weights in the networks, seen in Table 4.4. Most models have about 200,000 - 450,000 weights, which in the deep-learning context is quite small. This is a great advantage with the proposed method, as it allows for easier hyperparameter optimization and training of the networks. The models also converge quite fast, mostly within 20-40 epochs, which is partially explained by the small number of weights.

The estimated porosities were determined to be 20.34, 33.51, and 45.75 % for each HPC sample by automatic segmentation with the standard 2DCNN(81,81,3). Assuming that the true porosities are exactly 22, 30, and 45 %, our estimations are somewhat wrong. The reason could be the difficulties in the segmentation due to the shine-through effect and/or incomplete leaching of the HPC samples. HPC22 is underestimated, whereas HPC30 and HPC45 are overestimated. An explanation is that the HPC22 sample is close to the percolation threshold, leading to HPC-rich domains not being connected to the rest of the HPC phase. This could result in a lower porosity than expected for the HPC22 sample. The samples are also very heterogeneous, and we cannot ascertain that the true porosities are exactly 22, 30, and 45%. Nevertheless, the estimated porosities are in close agreement with the expected porosities.

5.2 Related work

The result presented in this work can directly be compared to the result in the original article [8]. In the original article, the feature extraction was performed by extracting linear scale-space features, i.e. Gaussian smoothed images at different scales. This step also considers a neighborhood around the pixel of interest, determined by the size of σ and the number of cross-sections. The extracted features were classified with a random forest classifier, that also predicts independently and outputs a raw score between 0 and 1. In our work, the feature extraction and classification step are combined, since a CNN performs both feature extraction and classification jointly. In theory, one could use an ANN as a classifier of the Gaussian scale-space features for even more resemblance. However, the main difference between the methods is the extraction of the features. In this area, CNNs have shown to be highly effective, lending themselves to the extraction of many abstract features.

It is also ensured that the data split, in terms of the square regions chosen for training, validation, and testing are identical. Therefore, a direct comparison can be made on all labelled data. We show such a comparison in Table 5.1. Our proposed method improves results on all metrics by 1.35 - 3.14 %.

Table 5.1: Comparison between the best performing CNN model vs. Gaussian scale-space features and a random forest classifier [8]. The table shows the combined result on all labelled data for the validation and test set.

	val mIoU	**val Accuracy**	**test mIoU**	**test Accuracy**
Gaussian scale-space features + random forest	0.7752	0.8865	0.7616	0.8825
2DCNN with neighborhoods $(81, 81, 3)$	0.8001	0.9000	0.7930	0.9003
Difference (%)	+2.49%	+1.35%	+3.14%	+1.78%

5.3 Limitations and future work

The amount of available data is somewhat limited, as there are only 300 labelled masks in total. However, as prediction is performed pixel-wise, we are not limited by the number of available pixels. The extracted neighborhoods are highly correlated because adjacent (or almost adjacent) pixels will represent neighborhoods that are only slightly shifted in different directions. Hence, many of the same porous structures will exist in many samples. We showed that increasing the number of pixels from 1% to 5% of all labelled data only slightly improved the performance. This indicates that more samples are not really needed. However, it is expected that more labelled masks could yield improved performance, as greater variety in the data would be achieved.

Manually labelling of masks requires an expert with good knowledge of the material and the intricacies of the shine-through effect in the images. It is a very time-consuming endeavor, and it is a challenge to label all masks consistently. Labeling is also somewhat subjective and minor deviations over many images add up. One could argue that the segmentation performed by the CNNs are better than the expert in some instances. However, we need to view the manual segmentation as the truth. For future work, it would be interesting to let multiple experts segment the same image, thus obtaining a quantification of the uncertainty of the manual segmentation.

The binary cross-entropy loss is utilized in all networks. It is not clear how well this loss correlates with the mIoU and porosity. However, we achieve very good mIoU scores and reasonable porosities for all datasets, which is an indication that it does. The previous masters thesis investigated a mixed loss, by combining weighted binary cross-entropy with an IoU loss [9]. They found that this approach yielded faster convergence and better results. However, their training was not performed with 50/50 class-balance and other CNN architectures were considered, hence it might not apply to our problem. For future work, one could look into the IoU loss and/or implement a custom loss function that better correlates with the porosity. This might be problematic, as a quantification of the porosity in terms of a differentiable function is needed.

Hyperparameter optimization in neural networks is always challenging. The practitioner needs to find a balance between explorative search and the investigation of the most promising configurations. In this work, we have searched very reasonable ranges that were found by exploration in the initial phase of the project. However, we cannot guarantee that there would exist even better configurations. Nevertheless, we can say with confidence that the hypothetical improvement for such configurations will be minor.

It is worth discussing the problem of having imbalanced data. This does not regard the training data, since we assured class-balance, however, the entire FIB-SEM dataset is imbalanced. Recall that approximately 68 % of pixels are solids and 32 % are pores in the entire data. This is a direct consequence of having three discrete HPC samples with expected porosities of 22, 30, and 45. One could train a CNN model for each of these HPC samples and hypothetically achieve higher performance on that particular HPC sample. However, the generalizability of the obtained model would not extend to other porosities. The postprocessing step is affected by the imbalance as well. The threshold (T) will not be perfect for either of the HPC samples and will almost certainly affect the HPC22 and HPC45 samples more negative than the HPC30 sample. It is worth noting that we strive for a model that will generalize to a wider range of porosities. Hence, a model with great generalizability and postprocessing parameters which work well enough for different porosities is considered to be optimal for our purpose.

What features the CNNs are able to learn are hard to tell, but one can imagine different forms of edge, corner, and blob detection. In Appendix B, we show the feature maps for one input sample for a standard 2DCNN model. It is clear that some form of edge and corner detection is present. Generally, CNNs extract low-level features in the earlier layers and more high-level features as the network grow deeper. One can view edge and corner features as low-level, and complete structures e.g. blobs as high-level features. This is an attempt of removing the so-called "black-box" nature of our proposed networks, and can be investigated further to give additional context to our proposed method.

It is possible that ensemble learning methods could boost performance even further, i.e. let multiple CNNs and/or other machine learning models independently classify the same pixel, followed by a majority vote to assign the pixel to a class. This approach is more computationally heavy, and a trade-off between the predicting power and computational time need to be considered. Perhaps transfer learning can apply to the problem as well. There might exist pretrained weights that have been trained on similar data, or at least similar enough to boost performance in terms of smoother training and convergence.

6 Conclusion

All the investigated CNNs were able to perform good segmentations of the FIB-SEM data. The standard 2DCNN perform the best, achieving a mean validation mIoU score of 0.8008 for its top model. This model utilizes neighborhoods $(n_x, n_y, n_z) = (81, 81, 3)$ around the pixel of interest as input. Other promising networks are the 3DCNN, TriplanarCNN, and the MultichannelCNN. These models achieve mean validation mIoU scores of 0.7960, 0.7939, and 0.7926 for their top models.

Having larger (n_x, n_y) neighborhoods as input benefits all models, whereas more information in n_z appears to be more or less negligible. It is hypothesized that utilizing larger (n_x, n_y) neighborhoods could increase the performance even further. However, there is a trade-off here to consider, as computational limitations would quickly become an issue. The proposed networks are quite simple and allow for straightforward hyperparameter optimization and training. The networks do not overfit and appear to be well suited to the problem. The need for a vast amount of data is combated by classifying pixel-wise, as one sample equals one pixel and its entire neighborhood.

The standard 2DCNN(81,81,3) is determined to be our final model and utilized for the segmentation of the full FIB-SEM data. The porosities are estimated to be 20.34, 33.51, and 45.75 %. These estimations are in good agreement with the expected porosities, albeit somewhat wrong. However, we cannot ascertain that the true porosities are 22, 30, and 45 % due to factors such as incomplete leaching and the samples being very heterogeneous.

The proposed method enables automated segmentation of microporous polymer films. The segmentation of one HPC sample takes approximately 38h for our final model, executed on two GPUs simultaneously. The segmentation process can easily be parallelized to run on multiple GPUs or CPUs. Consequently, there is a great potential of decreasing the computational time required for the segmentation.

We have shown that CNNs are highly effective for semantic segmentation of porous FIB-SEM data. Our proposed method improves results on all metrics by 1.35 - 3.14 % in comparison to Gaussian scale-space features and a random forest classifier [8].

Interesting future work would be to let multiple experts segment the same image to obtain more accurate ground truths. It would also be interesting to investigate the IoU loss and/or implement a custom loss that better correlates with the porosity. Other neighborhood sizes could be investigated, perhaps there exists an optimum size. Ensemble learning methods could potentially boost results even further, by utilizing multiple CNNs and/or other machine learning models.

References

1. Inkson, B., Mulvihill, M. & Möbus, G. 3D determination of grain shape in a FeAl-based nanocomposite by 3D FIB tomography. *Scripta Materialia* **45,** 753–758. ISSN: 1359-6462. `http://www.sciencedirect.com/science/article/pii/S1359646201010909` (2001).
2. *Chalmers Materials Analysis Laboratory* (Chalmers tekniska högskola (Chalmers University of Technology)). `https://www.chalmers.se/en/researchinfrastructure/CMAL/instruments/FIB/FIB/Pages/default.aspx`.
3. Salzer, M., Thiele, S., Zengerle, R. & Schmidt, V. On the importance of FIB-SEM specific segmentation algorithms for porous media. *Materials Characterization* **95.** ISSN: 1044-5803 (Sept. 2014).
4. Salzer, M., Spettl, A., Stenzel, O., Smått, J.-H., Lindén, M., Manke, I. & Schmidt, V. A two-stage approach to the segmentation of FIB-SEM images of highly porous materials. *Materials Characterization* **69,** 115–126. ISSN: 1044-5803. `http://www.sciencedirect.com/science/article/pii/S1044580312001003` (2012).
5. Prill, T. & Schladitz, K. Simulation of FIB-SEM Images for Analysis of Porous Microstructures. *Scanning* **35,** 189–195. `https://onlinelibrary.wiley.com/doi/abs/10.1002/sca.21047` (2013).
6. Taillon, J. A., Pellegrinelli, C., Huang, Y.-L., Wachsman, E. D. & Salamanca-Riba, L. G. Improving microstructural quantification in FIB/SEM nanotomography. *Ultramicroscopy* **184,** 24–38. ISSN: 0304-3991. `http://www.sciencedirect.com/science/article/pii/S0304399116302261` (2018).
7. Reimers, I. A., Safonov, I. V. & Yakimchuk, I. V. Segmentation of 3D FIB-SEM data with pore-back effect. *Journal of Physics: Conference Series* **1368,** 032015. `https://doi.org/10.1088%2F1742-6596%2F1368%2F3%2F032015` (Nov. 2019).
8. Fager, C., Röding, M., Olsson, A., Lorén, N., von Corswant, C., Särkkä, A. & Olsson, E. Optimization of FIB–SEM Tomography and Reconstruction for Soft, Porous, and Poorly Conducting Materials. *Microscopy and Microanalysis* **26,** 837–845 (2020).
9. Lennefors, M. & Visuri, W. J. *Deep learning for semantic segmentation of FIB-SEM volumetric image data* (Chalmers tekniska högskola (Chalmers University of Technology), June 2020).
10. Lecun, Y., Bottou, L., Bengio, Y. & Haffner, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE* **86,** 2278–2324 (1998).
11. Rawat, W. & Wang, Z. Deep convolutional neural networks for image classification: A comprehensive review. *Neural computation* **29,** 2352–2449 (2017).
12. Martın Abadi *et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems* Software available from tensorflow.org. 2015. `http://tensorflow.org/`.

13. Chollet, F. *et al. Keras* `https://github.com/fchollet/keras`. 2015.
14. MATLAB. *9.9 (R2020b)* (The MathWorks Inc., Natick, Massachusetts, 2020).
15. Kirk, E. *In situ microsectioning and imaging of semiconductor devices using a scanning ion microscope* in *Microscopy of Semiconducting Materials Conference, Oxford, UK, 6-8 April 1987* (1987).
16. Monteiro, S. N. & Paciornik, S. From historical backgrounds to recent advances in 3D characterization of materials: an overview. *JOM* **69,** 84–92 (2017).
17. Long, J., Shelhamer, E. & Darrell, T. *Fully convolutional networks for semantic segmentation* in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2015), 3431–3440.
18. Novikov, A. A., Lenis, D., Major, D., Hladvka, J., Wimmer, M. & Bühler, K. Fully convolutional architectures for multiclass segmentation in chest radiographs. *IEEE transactions on medical imaging* **37,** 1865–1876 (2018).
19. Rahman, M. A. & Wang, Y. *Optimizing Intersection-Over-Union in Deep Neural Networks for Image Segmentation* in *Advances in Visual Computing* (eds Bebis, G. *et al.*) (Springer International Publishing, Cham, 2016), 234–244. ISBN: 978-3-319-50835-1.
20. Jain, A. K. & Li, S. Z. *Handbook of face recognition* (Springer, 2011).
21. Koushik, J. Understanding convolutional neural networks. *arXiv preprint arXiv:1605.09081* (2016).
22. Liu, Y. H. *Feature extraction and image recognition with convolutional neural networks* in *Journal of Physics: Conference Series* **1087** (2018), 062032.
23. Mehlig, B. Artificial neural networks. *arXiv preprint arXiv:1901.05639* (2019).
24. Clevert, D.-A., Unterthiner, T. & Hochreiter, S. *Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)* 2015. arXiv: `1511.07289 [cs.LG]`.
25. Choromanska, A., Henaff, M., Mathieu, M., Arous, G. B. & LeCun, Y. *The loss surfaces of multilayer networks* in *Artificial intelligence and statistics* (2015), 192–204.
26. LeCun, Y. *Who is afraid of non-convex loss functions* in *2007 NIPS workshop on Efficient Learning, Vancouver, December* **7** (2007).
27. Ruszczynski, A. *Nonlinear optimization* (Princeton university press, 2011).
28. Hecht-Nielsen, R. in *Neural networks for perception* 65–93 (Elsevier, 1992).
29. Yamashita, R., Nishio, M., Do, R. K. G. & Togashi, K. Convolutional neural networks: an overview and application in radiology. *Insights into imaging* **9,** 611–629 (2018).
30. Ruder, S. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747* (2016).
31. Bottou, L. in *Proceedings of COMPSTAT'2010* 177–186 (Springer, 2010).
32. Rumelhart, D. E., Hinton, G. E. & Williams, R. J. *Learning internal representations by error propagation* tech. rep. (California Univ San Diego La Jolla Inst for Cognitive Science, 1985).
33. Orr, G. B. *Momentum and Learning Rate Adaptation* 2020. `https://www.willamette.edu/~gorr/classes/cs449/momrate.html`.
34. Duchi, J., Hazan, E. & Singer, Y. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research* **12** (2011).

35. Hinton, G. Lecture notes, 6e Coursera Class12. `http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`.
36. Kingma, D. P. & Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).
37. Glorot, X. & Bengio, Y. *Understanding the difficulty of training deep feedforward neural networks* in *Proceedings of the thirteenth international conference on artificial intelligence and statistics* (2010), 249–256.
38. Kanellopoulos, I. & Wilkinson, G. G. Strategies and best practice for neural network image classification. *International Journal of Remote Sensing* **18,** 711–725 (1997).
39. Van Laarhoven, T. L2 Regularization versus Batch and Weight Normalization. *CoRR* **abs/1706.05350.** arXiv: `1706.05350`. `http://arxiv.org/abs/1706.05350` (2017).
40. Ioffe, S. & Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167* (2015).
41. Perez, L. & Wang, J. The effectiveness of data augmentation in image classification using deep learning. *arXiv preprint arXiv:1712.04621* (2017).
42. He, H. & Garcia, E. A. Learning from Imbalanced Data. *IEEE Transactions on Knowledge and Data Engineering* **21,** 1263–1284 (2009).
43. Singh, S. P., Wang, L., Gupta, S., Goli, H., Padmanabhan, P. & Gulyás, B. 3D Deep Learning on Medical Images: A Review. *arXiv preprint arXiv:2004.00218* (2020).
44. Krogh, A. & Hertz, J. A. *A Simple Weight Decay Can Improve Generalization* in (Morgan Kaufmann Publishers Inc., Denver, Colorado, 1991), 950–957. ISBN: 1558602224.
45. Hu, J., Kuang, Y., Liao, B., Cao, L., Dong, S. & Li, P. A multichannel 2D convolutional neural network model for task-evoked fMRI data classification. *Computational intelligence and neuroscience* **2019** (2019).
46. Prasoon, A., Petersen, K., Igel, C., Lauze, F., Dam, E. & Nielsen, M. *Deep feature learning for knee cartilage segmentation using a triplanar convolutional neural network* in *International conference on medical image computing and computer-assisted intervention* (2013), 246–253.

A

Appendix 1

A.1 CNN architectures

A.1.1 MultichannelCNN(65,65,11)

In figure A.1, the architecture for the MultichannelCNN with neighborhoods $(n_x, n_y, n_z) = (65, 65, 11)$ as input is shown. Notice that there are three CNN paths where n_x, n_y and n_z is considered a channel in each.

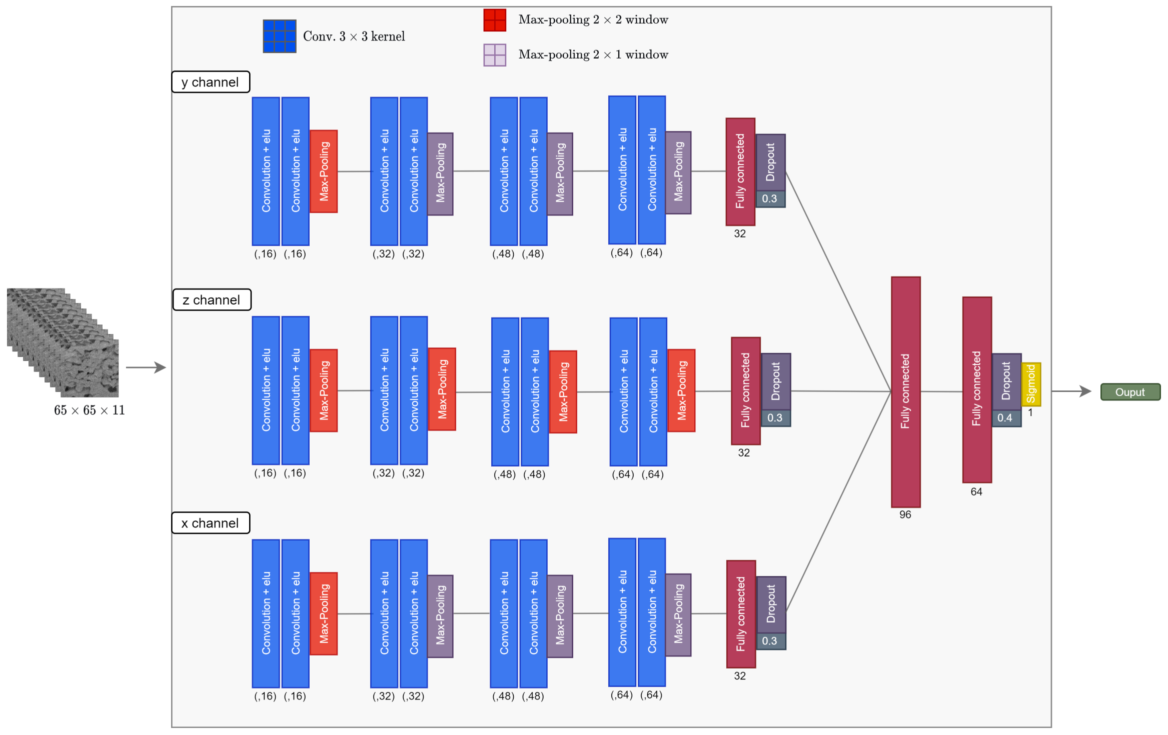
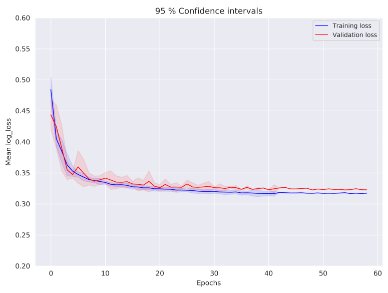


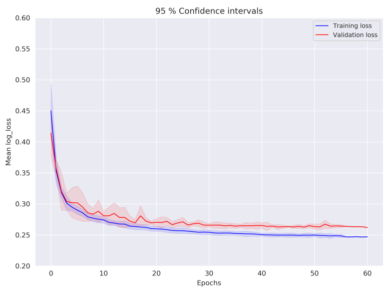
Figure A.1: CNN architecture for the MultichannelCNN(65,65,11).

A.2 Training plots for various CNN architectures

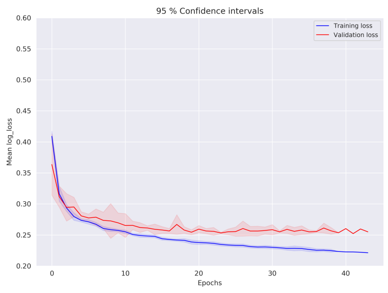
Down below, training plots for various CNN architectures and inputs are shown. The figures show mean log-loss with 95 % confidence intervals based on five runs, with training scores (blue) and validation scores (red).



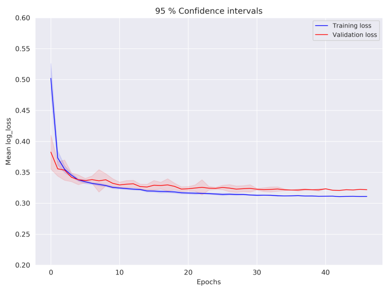
(a) *2DCNN(11,11,11)*



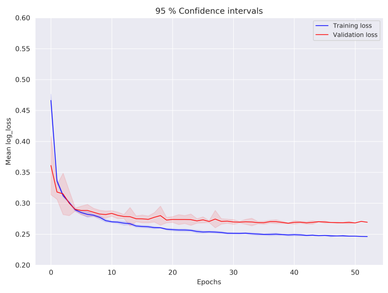
(b) *2DCNN(33,33,11)*



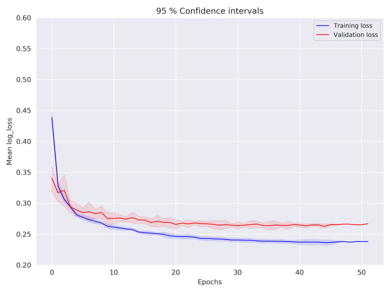
(c) *2DCNN(49,49,11)*



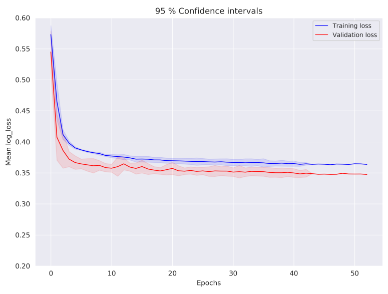
(d) *3DCNN(11,11,11)*



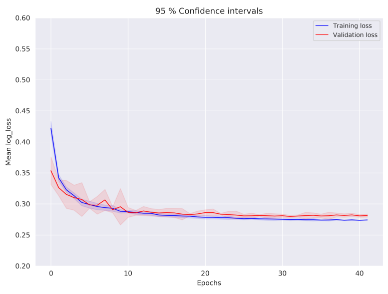
(e) *3DCNN(33,33,11)*



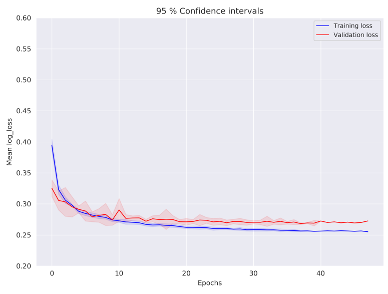
(f) *3DCNN(49,49,11)*



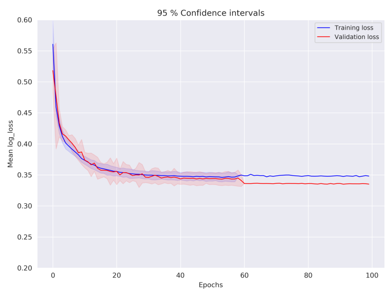
(g) *TriplanarCNN(11,11,11)*



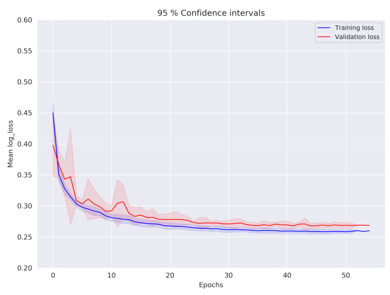
(h) *TriplanarCNN(33,33,11)*



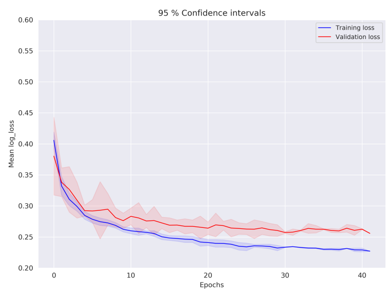
(i) *TriplanarCNN(49,49,11)*



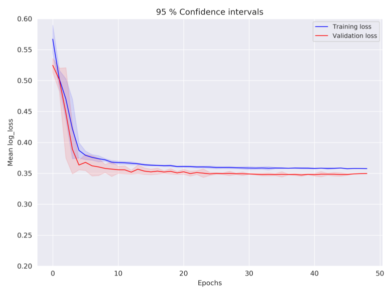
(j) *MultichannelCNN(11,11,11)*



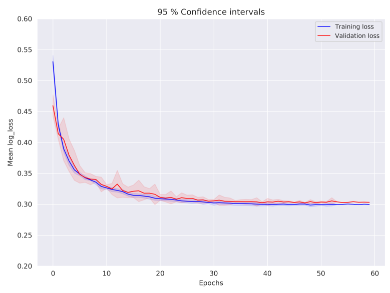
(k) *MultichannelCNN(33,33,11)*



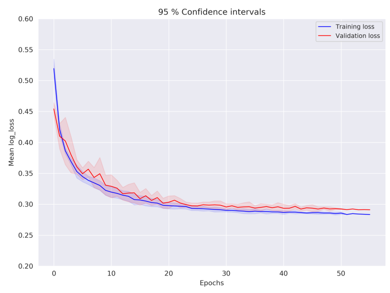
(l) *MultichannelCNN(49,49,11)*



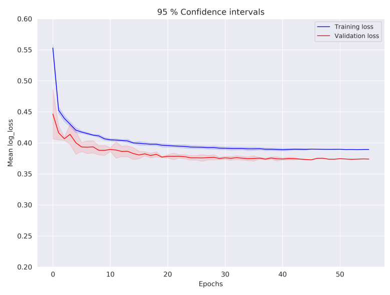
(m) *1DCNN(11,11,5)*



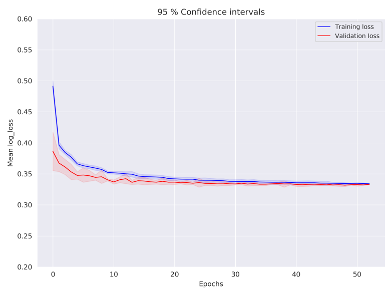
(n) *1DCNN(33,33,5)*



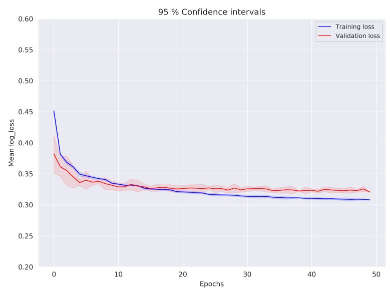
(o) *1DCNN(49,49,5)*



(p) *MV2DCNN(11,11,11)*



(q) *MV2DCNN(33,33,11)*



(r) *MV2DCNN(49,49,11)*

Figure A.2: Mean log-loss with 95 % confidence intervals based on five runs for various CNN architectures and input shapes.

B

Appendix 2

B.1 Feature map visualization

Visualization of feature maps for the standard 2DCNN with neighborhoods $(n_x, n_y, n_z) = (65, 65, 3)$ as input. Feature maps are produced by changing the output of the network while keeping the trained weights of the kernels. The example is shown for one input sample, see following figure.

Input image: $(n_x, n_y, n_z) = (65, 65, 3)$

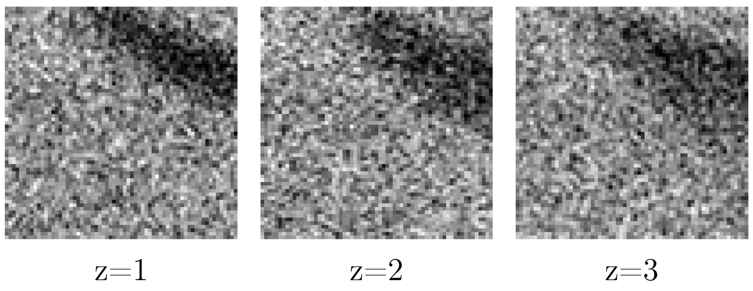
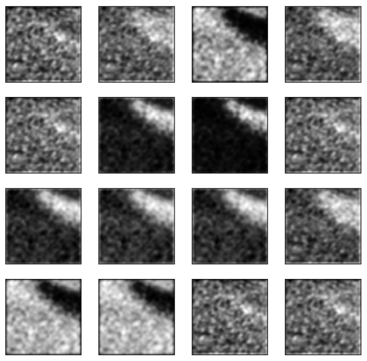
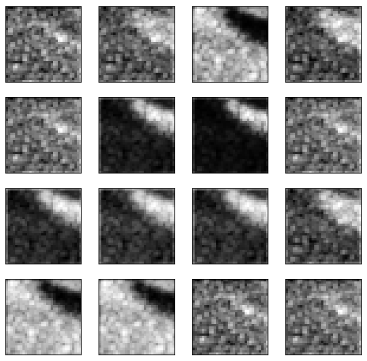


Figure B.1: Visualization of feature maps for the standard 2DCNN with neighborhoods $(n_x, n_y, n_z) = (65, 65, 3)$ as input.

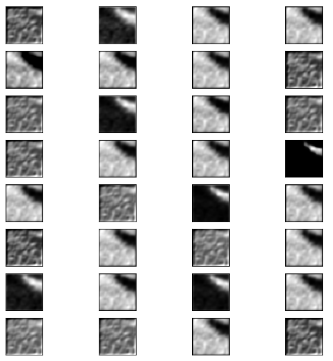
Feature maps

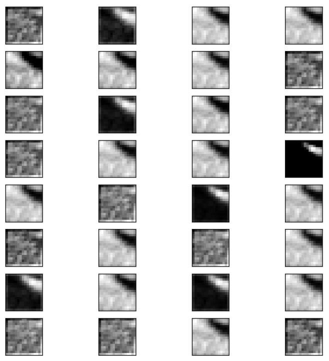


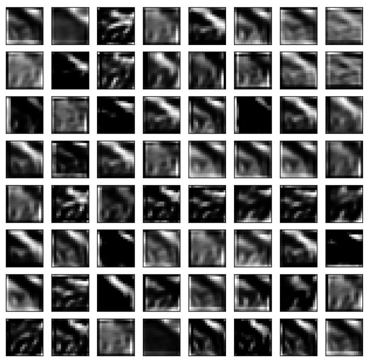
(a) Conv1

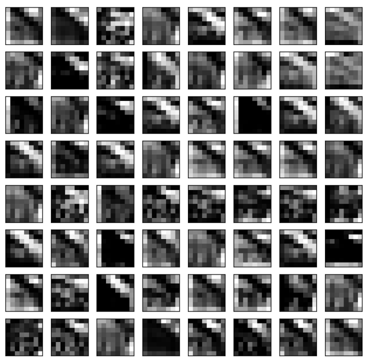


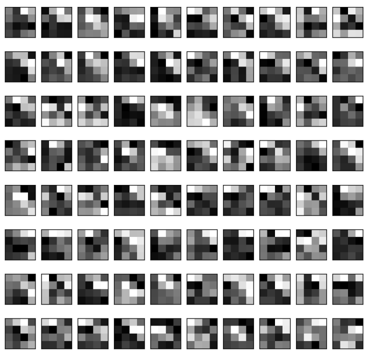
(b) Conv2

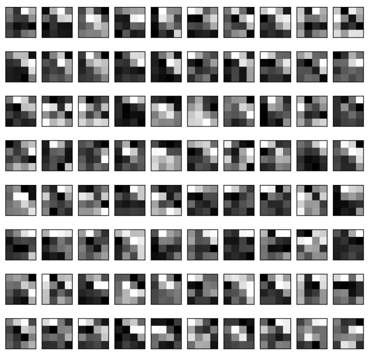

(c) Conv3


(d) Conv4


(e) Conv7


(f) Conv8


(g) Conv9


(h) Conv10