Application of Deep Learning to Coherent Fourier Scatterometry data

Davy Davidse

Thesis in partial fulfilment of the requirements for the degree of Master of Science in Applied Physics

Examination committee:	*Supervisors:*
Prof. Dr. S. Stallinga	Dr. Ir. S.F. Pereira
Dr. C. Smith	D. Kolenov MSc

April 18, 2020

TU Delft

ImPhys

Abstract

This thesis discusses the application of deep learning to Coherent Fourier Scatterometry data in order to quickly and reliably detect nanoparticles on surfaces. An introduction to deep learning is followed by a review of the experimental setup and used software. After that, results are presented of classification accuracy tests on various datasets containing images obtained from scatterometry scans. We show that a relatively simple convolutional neural network can achieve an accuracy as high as 98% on a 200 image test set. We compare this to the accuracy of a non-deep learning, clustering based classification algorithm and conclude that deep learning is a more suitable method for particle classification. Then, three methods of open set recognition are applied. We show that it is possible to reject 80% of a fooling dataset at the cost of rejecting 10% of the normal data. Finally, the results are discussed and placed in the context of future work on this subject.

Acknowledgement

I would like to thank my supervisor, Silvania Pereira, for having faith in my abilities and letting me pursue the directions I wanted. Thanks also to Dmytro Kolenov, for always being available for questions and for doing all the data acquisition, a job I don't envy.

Further thanks go to Roland Horsten for getting my PC at TU Delft into a workable state and for providing and installing a graphics card. Thanks also to Ronald Ligteringen who helped get the Python installation done properly and sat down with me for an additional troubleshooting session after what turned out to be a bugged package update.

And finally, thanks to all the great people at the Optics group who made my stay enjoyable!

Contents

Chapter 1

Introduction

This chapter will provide a short introduction of the topics discussed in this thesis.

1.1 Particle detection

Detection of small particles is becoming increasingly significant in the semiconductor industry. As feature size is shrinking, clean substrates are becoming more important. Furthermore, a diverse range of materials is being used today, including silicon and polymer substrates. It is desirable to have a non-destructive particle detection system that can operate on various substrates and resolve particles in the range of 100 nm or less [1, 2].

One such system is described in Ref. [4]. This system uses polarized light in a bright field particle detection setup to detect particles with a size of 100 nm. For this thesis, a similar system was used. For further discussion of the setup, see section 3.1.

1.2 Classification and machine learning

Machine learning is the use of artificial intelligence to provide a system with the ability to learn from experience and improve itself without being explicitly programmed. To illustrate how machine learning is used in this thesis, we will start with an example.

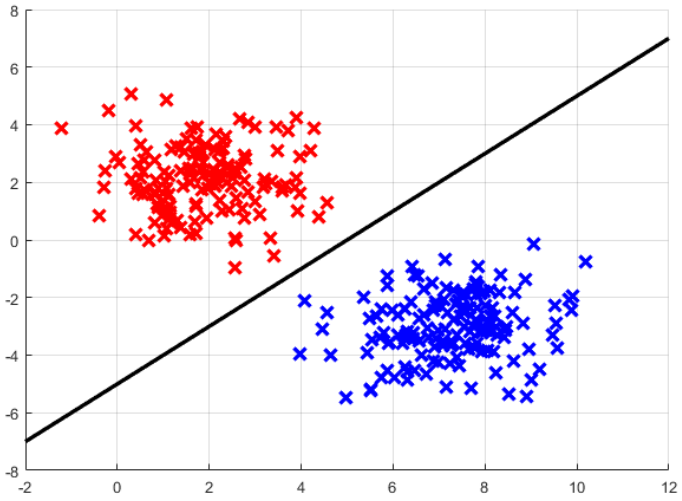


Figure 1.1: Two clusters of synthetic example data. They are labeled by color.

Consider the clusters of data in Fig. 1.1. They are labeled: all data points in the first cluster are red while all points in the second cluster are blue. What if we get a new point of data, without a label, and want to know which cluster it belongs to? This is referred to as a *classification task*: we want to assign a class to an unlabeled data point based on previously seen labeled data. In our example, the simplest way to do this is to draw a line between the clusters. We can then assign the label 'red' or 'blue' to new data points based on whether they are lying above or below the line.

In the above case, it is quite obvious how the line should be drawn. This is not always the case. For example, the clusters could lie closer to each other and partially overlap. Finding the line that best seperates the two clusters can then be a difficult task, which is where machine learning comes in. A line in a two dimensional space has the form:

$$y = ax + b \tag{1.1}$$

Though in the context of classification it is preferable to write:

$$\alpha_1 x + \alpha_2 y = \beta \tag{1.2}$$

For each data point a β can be calculated using the point's x and y values and the model's α_1 and α_2 parameters. If β is above a certain value, the point is classified as 'red', otherwise it is classified as 'blue'. The goal of a machine

learning algorithm is to find α_1 and α_2 such that for a given set of labeled training data, the model correctly classifies as many points as possible.

This is a simple example in which data points consist of only two numbers (x and y values) and have only two possible outcome classes. In reality, dimensionality of data is often far greater. For example, one might try to classify images. A 100x100 image contains 10,000 pixel values, and that is for a greyscale image (an RGB image would have thrice that number). The number of output classes greatly differs per application, but is nearly always higher than two, further increasing the complexity of the classification problem. Efficient machine learning algorithms are needed to handle these advanced problems.

1.3 Deep Learning

The data coming from our particle detection setup is two dimensional. In essence, we are creating images. It is important to realize then, that in the past decades, there has been major progress in the field of image processing and computer vision by using artificial neural neworks [5]. This is a branch of machine learning commonly called *deep learning*. More specifically, a type of artificial neural network known as convolutional neural network (CNN) is responsible for the success in this field in recent years [6, 7, 8]. There has also been an increasing use of CNNs to book successes in the field of optics [9, 10, 11].

1.4 Project goal

This raises the following questions: can we use deep learning to quickly and reliably detect and classify small particles? The goal of my project was to find the answer to that question.

1.5 Thesis outline

In chapter 2, we will discuss some of the theory behind deep learning. Chapter 3 contains both a description of the optical setup we used to generate data and the artificial neural network we used to process that data. In chapter 4, we present the results of various experiments that we performed in the pursuit of a well functioning, high accuracy network. Chapter 5 contains a discussion of the obtained results and possible future improvements. Finally, chapter 6 concludes this thesis.

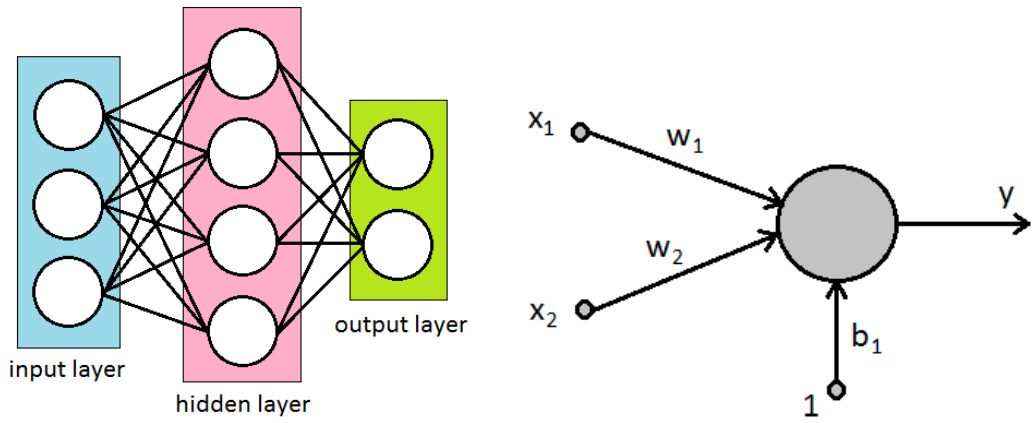
Chapter 2

Theoretical background

This chapter contains some of the theoretical background behind deep learning and some related concepts that are relevant to this thesis.

2.1 Artificial neural networks

The theory presented in this section is based on the online book *Deep Learning* by Goodfellow et al. (see Ref. [13]).



(a) (b)

Figure 2.1: a) Schematic of a simple artificial neural network. Each layer consists of a number of nodes, which run basic mathematical operations on the data that flows through the network. b) Schematic of a perceptron.

Deep learning is a form of machine learning that allows computers to learn from experience and understand the world in a way that is closer to the way

our human brain works than traditional machine learning algorithms are. To do so, it makes use of artificial neural networks (ANNs). Such networks consist of an input layer, an output layer and one or more layers in between (see Fig. 2.1a). ANNs are used in many fields today, such as regression analysis, data processing, and classification. The latter is the core of this thesis.

The network in Fig. 2.1a is *fully connected*. This means that all possible connections between the nodes of 2 adjacent layers are active. The amount of layers and amount of nodes per layer were chosen small because it is convenient for the figure. In reality, the required complexity of the network (which includes the amount of nodes and layers) depends on the task that the network is being designed for. For example, distinguishing between cats and dogs in photos will require a far more complex network than distinguishing between squares and circles.

2.1.1 Feedforward networks

To understand how data propagates through the network, we have to know what each node, or *neuron*, does. For that, we will look at a single node, also called a *perceptron* (see Fig. 2.1b). A perceptron can be used as a simple binary classifier. Our example has 2 inputs, called x_1 and x_2, two weights called w_1 and w_2, a bias denoted by b_1 and an output y. The output is calculated as follows:

$$y = w_1 x_1 + w_2 x_2 + b_1 \tag{2.1}$$

A fully connected neural network works the same way. Each layer takes the inputs coming from the previous layer, multiplies them with their respective weights and adds a bias value. The process where data passes through all the layers of the network this way is known as a *forward pass*. Networks based on this are called *feedforward neural networks*. This term only applies when information flows through the network in one direction. The alternative is to have feedback connections in the network, which is beyond the scope of this thesis. For more information on feedback networks, see Ref. [16].

We will use $\boldsymbol{\theta}$ to denote the parameters of the network, which are the weights and biases of all the nodes. A feedforward network defines a mapping $y = f(\boldsymbol{x}; \boldsymbol{\theta})$ and learns $\boldsymbol{\theta}$ such that this best approximates a desired mapping $y = f^*(\boldsymbol{x})$. For a classifier, during the learning process, called *training*, each data point $\boldsymbol{x}$ that is put into the network is accompanied by a label $y = f^*(\boldsymbol{x})$. The network must decide how to adjust its parameters $\boldsymbol{\theta}$ to get

its outputs closer to the desired label.

Note that the label only specifies what the network outputs must be; it does not specify what the values in each of the layers before the output layer need to be. This is why the layers between the input and output layer are typically called *hidden layers*: they are the internal workings of the network and as users we are not concerned with the values in those layers.

2.1.2 The softmax function

A neural network designed for classification has an output of 1 score per class. To convert these scores into probabilities, the softmax function can be used. It is defined as:

$$S_i(\boldsymbol{x}) = \frac{\exp v_i(\boldsymbol{x})}{\sum_j \exp v_j(\boldsymbol{x})} \tag{2.2}$$

Here $\boldsymbol{v}(\boldsymbol{x})$ is the vector containing the outputs for an input image $\boldsymbol{x}$, and $S_i(\boldsymbol{x})$ is the probability that $\boldsymbol{x}$ belongs to class i.

2.1.3 Introducing nonlinearity: activation functions

A perceptron is a linear model. It follows that a linear combination of perceptrons is also linear. This greatly limits our classification power, because the network has a very poor ability to approximate nonlinear functions $f^*(\boldsymbol{x})$. This is why we want to introduce nonlinearity to the network. We do this by implementing an *activation function* between layers, so that instead of working on the output of the previous layer $\boldsymbol{x_n}$, each layer now works on a transformed output $\phi(\boldsymbol{x_n})$.

Various activation functions have been succesful in deep learning, such as the sigmoid, tangent hyperbolic and softmax functions. However, since we are interested in image classification with CNNs (more on this later), the activation function that is of most interest to us is known as rectifier. Using a rectifying nonlinearity has been said to be the single most important way to improve the performance of a recognition system [17]. In 2011, it was proven to be very effective in deep neural networks [18], and it has been the most used activation function since then [13].

In the context of neural networks, the rectifier function has the following definition:

$$f(x) = \max(0, x) \tag{2.3}$$

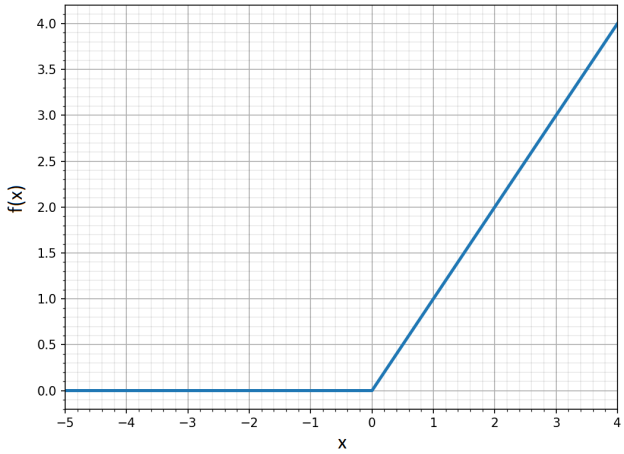


Figure 2.2: Graph of the rectifier function.

The rectifier is usually referred to as *ReLU*, which stands for Rectified Linear Unit. To clarify the terminology: ReLU is a node (or unit) in a neural network that applies the rectifier function.

ReLU is computationally faster than using other functions like tanh, because it does not involve computationally expensive elements like exponentials. Also, it has been observed to lead to significantly faster training of large neural networks. The downsides of ReLU are that it is not zero centered and that it is unbounded, which can hurt the performance of the neural network. However, techniques such as batch normalization (more on this later) can fix these issues [7].

2.1.4 Gradient descent

The training of a neural network is an optimization problem. We want to minimize some *error function*, also called *loss function* or *cost function*. This function represents how far away our network's outputs are from the desired outputs. For each input $\boldsymbol{x}$, the label c and the network outputs $\boldsymbol{v}(\boldsymbol{x})$ are fed into the loss function to produce a scalar loss value L. In order to adjust the network's parameters so that L gets smaller, we can compute the gradient of the loss function with respect to the weights. The gradient represents both the rate and direction of greatest increase, so if we take small steps in the opposite direction, we reduce the loss. This method is known as *gradient descent*.

Gradient descent updates the network parameters $\boldsymbol{\theta}$ as follows:

$$\boldsymbol{\theta}' = \boldsymbol{\theta} - \epsilon \nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}) \tag{2.4}$$

Here ϵ is the *learning rate*, which determines the size of the steps taken at each update, and f is the loss function. Note that $\nabla_{\boldsymbol{\theta}}$ denotes the gradient with respect to $\boldsymbol{\theta}$, not to be confused with a directional derivative (for which unfortunately the same notation is often used).

Gradient descent converges when all elements of the gradient are close to zero. It is important to realize that for non-convex loss functions with local minima, convergence need not be at the global minimum.

The algorithm that is commonly used to compute gradients is called *back-propagation*. This is because information propagates back through the network in order to calculate the partial derivatives of the error with respect to the weights and biases of earlier layers. A full treatment of the inner workings of back-propagation is beyond the scope of this thesis. For this, we refer to a well written chapter in the free online book *Neural Networks and Deep Learning* by M. Nielsen, see Ref. [14]. For further reading, see the original paper by D. Rumelhart et al. [15].

2.1.5 Training the network

Training a neural network is an iterative process. A dataset is split up in batches, which are passed through the network one by one. The term *epoch* is used to describe the cycle where each sample in the training data is used for training once. [19] The amount of epochs to train is an example of a *hyperparameter*, this is a term used in machine learning for the parameters that are set before training begins.

For each batch, a forward pass is performed and an error is calculated. After that, backpropagation calculates the partial derivatives of this error with respect to the network's parameters via gradient descent. At that point we have all the information we need to update the parameters. However, there are many functions that perform this update in different ways. Such a function is called an *optimizer*. The choice of optimizer plays a big role in the speed and efficiency of the training process.

After training concludes, efficacy of the process is assessed by a test accuracy, which is defined as the percentage of images in a test dataset that the network classifies correctly. The goal of the test set is to test the network's *generalization power*: its ability to use the knowledge it gained during training and apply that to images it has never seen before.

Training does not always go well. There are two main scenarios where the network does not converge to a high test accuracy: underfitting and overfitting.

Underfitting is when the network does not learn the test data well enough. Typically, this one is easiest to prevent. It can be reduced by increasing the

complexity of the network, feeding it more data or letting it train for a longer time.

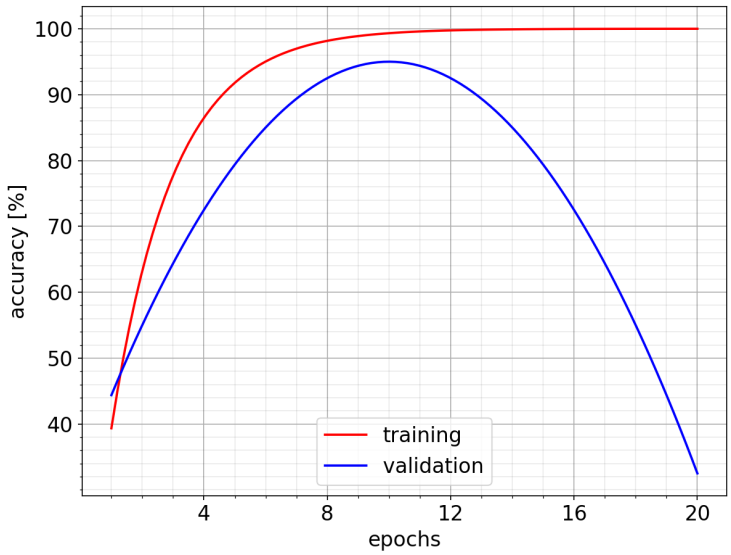


Figure 2.3: Example of overfitting. After 10 epochs of training, training accuracy continues to increase while validation accuracy drops. Thus, in order to create the network with the highest generalization power, it is best to stop training after 10 epochs.

Overfitting is when the network learns the test data too well, and thus can only accurately predict the examples from the test data. It then performs poorly on new data that was not seen during training. This is more tricky to solve. To visualize overfitting, a common practice is to split the dataset into 3 parts: a train set, a validation set, and a test set. The train set is used to train the network as described before. The validation set is used to test the accuracy of the network after each epoch. The optimal amount of epochs is then the amount after which the validation accuracy is highest. When the validation accuracy starts to drop, the network is overfitting (see Fig. 2.3). At this point, we want to stop training and use the test set to generate a test accuracy score, which is the main indicator of how succesful a training run was.

2.1.6 Reducing overfitting

There are three popular techniques to reduce overfitting. First, it is in our best interest to keep the weights in the network small. This is known as *regularization*. One way of regularization is to include it in the loss function.

For example, L2 regularization adds the square of each weight to the loss function, thus punishing large weights.

The second technique is called *dropout*. This randomly turns off a number of nodes in the network during each epoch of training. The consequence is that nodes will develop less co-dependency, so that their individual contribution to the network is better trained.

The third technique is called *batch normalization*. This technique updates the values of the data that goes through the network after each layer as follows:

$$\boldsymbol{x}_{i+1} = \frac{\boldsymbol{x}_i - E(\boldsymbol{x}_i)}{\sigma(\boldsymbol{x}_i)} \tag{2.5}$$

Here $E(\boldsymbol{x}_i)$ is the mean of the batch and $\sigma(\boldsymbol{x}_i)$ its standard deviation. The purpose of batch normalization is twofold: to speed up training and to reduce overfitting. The former is because inputting smaller values to the next layer is computationally advantageous while the latter is because batch normalization has a regularization effect. This results from the fact that batches are randomized, so that the $E(\boldsymbol{x}_i)$ and $\sigma(\boldsymbol{x}_i)$ are different every time.

Lastly, it should not come as a surprise that a larger dataset generally also reduces overfitting. The larger the training set, the more representative it will be for all possible inputs.

2.2 Convolutional neural networks

Our data comes in the form of images, and we wish to classify them based on whether or not a particle is detected, and if so, of what size. This is why we are most interested in a special type of neural network known as *convolutional neural network* (CNN). CNNs include one or more convolutional layers, and they are widely used in image classification and computer vision.

A convolutional layer is a network layer that applies a two dimensional convolution operator to the data. This means that a two dimensional kernel moves over the image and generates a new image via convolution. The size of the kernel and the size of its steps are the most important parameters to set. Additionally, a padding parameter can be used to zero pad the two dimensional data. This prevents the convolution operator from reducing the size of the image, because the layers that are lost are now the zero padded layers instead of the outer layers of our data.

The values of the kernel are adjusted by the network during training. It is also possible to have multiple kernels with different values in one convolu-

tional layer. Each of these kernels then produces a different stream of data. We call these streams *channels*.

2.2.1 Pooling layers

A pooling layer is similar to a convolution layer in a way: a kernal moves over the image and performs a mathematical operation to generate a new image. However, in a pooling layer, this operation is max, min or mean. Max pooling layers are most common in CNNs because they generally best preserve the important features of an image due to keeping the extremes. However, which type of pooling performs best depends on what type of images are being used as input data, and in case of classification, how they are seperated into classes.

In a max pooling layer, a kernel moves over the image and selects the maximum value in its window. For example, if a 2x2 kernel is selected, the maximum value in each 2x2 block of data will be taken to form the new output image. Once again, the step size of the kernel can be selected. Typically, step size 2 or greater is used. That way, the size of the image is greatly reduced. This speeds up computation and also reduces overfitting. The reason for the latter is that max pooling tends to preserve the most important features of the image, and thus the information that's thrown away is mainly irrelevant information that would otherwise be included in training.

2.2.2 Network architecture

Now that we have some knowledge of all the important components of a CNN, we can talk about its architecture. Typically, the convolution layers are followed by other layers in this order:

1. Convolutional layer
2. Max pooling layer
3. Activation layer (ReLU)
4. Batch normalization layer

This ensures that we gain all the benefits from these layers as described in the previous sections, while keeping computational cost as low as possible.

The convolutional layers mainly deal with feature extraction. This means that, as we get deeper into the network, the images will start to represent specific features of the input images. That is why the images that have been

processed by one or more convolutional layers are called *feature maps*. To give an example: if our inputs were images containing human faces, deep feature maps might look like ears or eyes.

The task of the network as whole, however, is classification. This is why after the convolutional layers we need one or more fully connected layers. These are layers similar to the hidden layers in Fig. 2.1a. Note that they are one dimensional layers, so before sending our data to them, we have to reshape it. In other words, instead of the two dimensional arrays coming out of the convolution and pooling layers, we have to send in a one dimensional list of numbers.

2.3 Network analysis tools

The most basic way to assess performance of a neural network is accuracy, which we define as the percentage of images from a dataset that was classified correctly. Thus, a higher accuracy means the network performed better. We can plot how accuracy changes during training by using the validation set, to get a visualization of the speed of convergence. However, this is still rather limited, because it makes no distinction between classes (what if one class has a classification accuracy of 100% while another has 50%?) and it's hard to get an idea about how changes to the architecture or optimizer affect the training process.

2.3.1 Confusion matrix

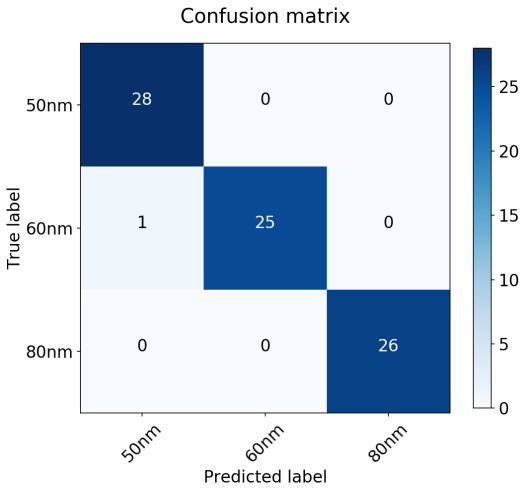


Figure 2.4: Example of a confusion matrix where one 60 nm particle is misclassified as a 50 nm particle.

The first of these issues can be fixed with a *confusion matrix*. This is a matrix that shows for each input class how much of it is classified into each output class. An ideal confusion matrix is diagonal, because the diagonal is where the predicted label equals the true label (see Fig. 2.4). Off-diagonal terms represent misclassification.

After generating a confusion matrix, we know which classes the network struggles to distinguish and to what degree. In case the network performs badly, this gives us an opportunity to inspect the input data and perhaps find out what the network struggles to get right .

2.3.2 Loss Landscape

It would be interesting to visualize how the performance of the network changes as a function of $\boldsymbol{\theta}$. To do this, we use the total loss L_T, which is the sum of the loss values generated by the images in the dataset. Varying $\boldsymbol{\theta}$ in two independent directions creates a surface for L_T known as a *loss landscape*. However, even a relatively small convolutional neural network can easily have an amount of parameters in the order of 10^6. How do we then choose the two directions that generate our landscape?

The easiest way is to choose them randomly. We can initialize 2 sets of random numbers of the same shape as $\boldsymbol{\theta}$ and repetitively add them to $\boldsymbol{\theta}$ to generate loss values at various "distances" away from our origin. To improve

this, we can use what is known as *filter normalization* [20]: for each layer of our network, we calculate the norm of the weight vector specific to that layer, and we adjust the corresponding weights in the randomly generated vectors such that they have the same norm. This way we make sure that the distance we move from the origin makes sense: when we add a randomly generated vector once, each network weight vector gets another weight vector with the same norm added to it.

There is, however, one problem with randomly generated landscapes: they have a low chance of capturing much of the variation of the weights, because most of the variation lies in a low dimensional space [20]. This means that we need some way to select specifically those dimensions where most of the variation resides.

Principal component analysis

If we have a dataset with p variables, we can define a data matrix $\mathbf{X}$ with $n \cdot p$ values, where each column corresponds to a variable. The objective of *principal component analysis*, or PCA, is to find the linear combination of these columns that has the highest variance. The linear combination takes the form:

$$\sum_{j=1}^{p} a_j \mathbf{x}_j = \mathbf{X}\mathbf{a} \tag{2.6}$$

The variance is given by:

$$var(\mathbf{X}\mathbf{a}) = \mathbf{a}'\mathbf{S}\mathbf{a} \tag{2.7}$$

Here $\mathbf{S}$ is the sample covariance matrix corresponding to the dataset, and $\mathbf{a}'$ denotes the transpose of $\mathbf{a}$. For our problem to be well defined, we impose the condition that $\mathbf{a}'\mathbf{a} = 1$. This means what we are trying to find is:

$$\max\left[\, \mathbf{a}'\mathbf{S}\mathbf{a} - \lambda(\mathbf{a}'\mathbf{a} - 1) \,\right] \tag{2.8}$$

Differentiating this and equating it to zero gives us:

$$\mathbf{S}\mathbf{a} - \lambda\mathbf{a} = 0 \tag{2.9}$$

This means that the vectors $\mathbf{a}_k$ are eigenvectors of $\mathbf{S}$, and λ_k are the eigenvalues. The *principal components* of the data matrix $\mathbf{X}$ are the products $\mathbf{X}\mathbf{a}_k$. Looking at the variance, we find:

$$\text{var}(\mathbf{X}\mathbf{a}) = \mathbf{a}'\mathbf{S}\mathbf{a} = \mathbf{a}'\lambda\mathbf{a} = \lambda \tag{2.10}$$

So the principal component corresponding to the highest variance is the one with the highest eigenvalue. The next highest variance is found in the component with the second highest eigenvalue, and so on. This allows us to select the 2 directions of highest variance, which we can use to generate our loss landscape.

For more information and a more detailed explanation of PCA, see Ref. [27].

2.4 Applying the theory

What we will do in the next chapters based on this theory is the following:

1. Create a CNN using *Python* code
2. Implement the inspection methods (confusion matrix and loss landscape)
3. Set and adjust the CNN's options (optimizer, loss function, hyperparameters) in such a way that it reaches the highest classification accuracy

Chapter 3

Setup

3.1 Scatterometry setup

The technique used in our setup for detecting nanoparticles on surfaces is called Coherent Fourier Scatterometry (CFS). In scatterometry, a beam of light is incident on an object of interest, which scatters it back into other directions than the direction of the incoming beam. This scattered light is then detected in the far field. It contains information about the object, even if the latter is of dimensions smaller than the wavelength of the incident light. In CFS, the incident light is a coherent laser beam. The use of coherence makes CFS more sensitive than incoherent scatterometry [21, 22].

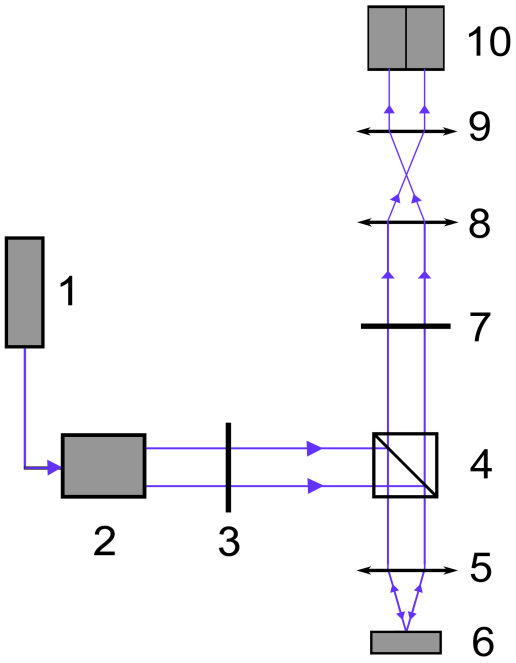
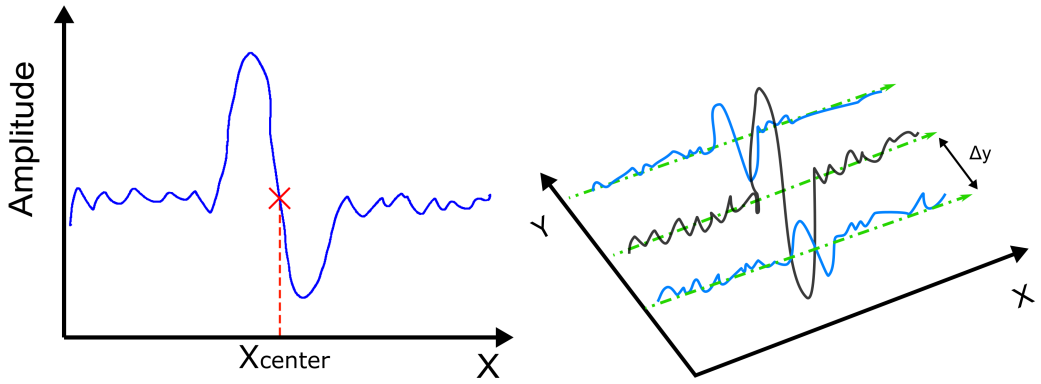


Figure 3.1: Optical setup. Light from a laser (1) is coupled to an optical fiber and processed by a collimator (2). It then passes through a linear polarizer (3). It then is directed to the sample by a 50% beam splitter (4) and focused on the sample surface by an objective (5). The sample is mounted on an X-Y-Z piezoelectric stage (6). The spurious reflection from the surface and the scattered light are coupled back through the objective (5) and go back towards the beam splitter (4). After being transmitted by the beamsplitter, they pass through another polarizer (7) and a telescopic arrangement of two lenses (8) and (9), before encountering the split detector (10).

For this work, we used a setup that had been built in an earlier project, see Ref. [4]. The setup is shown in Fig. 3.1. Light from a laser with a wavelength of 405 nm travels through a linear polarizer before being redirected to the sample and focused by an objective. The scattered light passes through another linear polarizer, which aligns the direction of polarization with the orientation of the detector. Then, it passes through two lenses in a telescopic arrangement, which image the back focal plane of the objective to the detector plane, before entering a bi-cell photodiode. This is a photodiode with two active detection areas, which allows for differential detection. This improves SNR, mainly through elimination of spurious reflections from the sample [23].



(a) (b)

Figure 3.2: a) single scanning line of a particle, b) 3D representation of multiple scanning lines over a particle

The sample is mounted on a 3D piezoelectric stage that can be precisely moved around. First, the best focus position is found by looking at the collimation of the reflected light. After the focus position is optimized, a raster scan is performed. We denote the direction of scanning with X and the spacing between the lines with ΔY. For each position X,Y the value of the differential voltage of the split detector is stored. The detector is aligned in such a way that the line that seperates its two halves is perpendicular to the direction of scanning. This way, when a particle is detected, initially only the first half of the detector sees it. This generates a positive differential signal. Then, when the particle is aligned in the middle of the beam, both halves see it, resulting in zero differential signal. Lastly, only the second half sees it, which results in a negative differential signal. Fig. 3.2a shows what the total signal for one particle looks like. The width of the pulse indicates the size of the particle. The amplitude of the pulse depends not only on the size of the particle, but also on the material the particle consists of and on how well the light is focused and centered on the particle. Finally, the symmetery of the pulse depends on the focus position. When it is in focus, the pulse is symmetric.

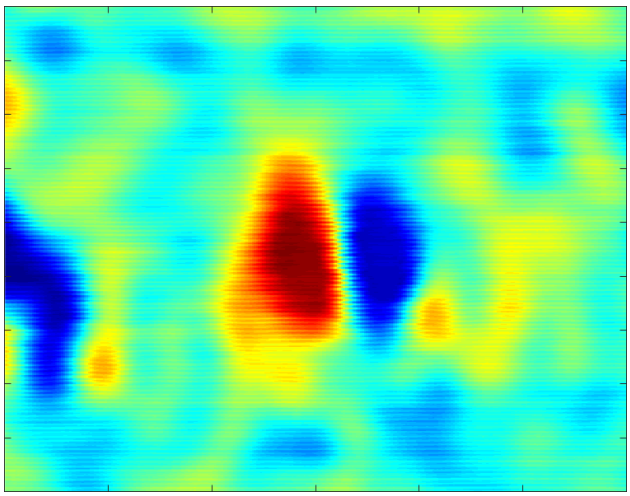


Figure 3.3: Image representation of part of a 2D scan obtained from a sample with 80 nm PSL spheres. The image shows an x-range of 1.35 µm and a y-range of 0.9 µm.

An example of how a particle shows up in a 2D scan is shown in Fig. 3.3.

In this thesis, all scans are made on polished silicon wafer surfaces that were artificially contaminated with spherical polystyrene latex (PSL) nanoparticles of known sizes. We also had access to samples containing gold particles, but at the same input laser power, those yielded a much stronger signal coming from the scattered light. PSL spheres are a greater challenge to detect. On top of that, they are more realistic, because in a production environment, not all contaminants scatter as strongly as gold.

3.2 Software

We wrote the neural network code in *Python*. For that, we used a *Miniconda* installation of Python 3.7 with the following additions:

- spyder
- matplotlib
- scipy
- scikit-learn
- torchvision, pytorch
- loss-landscapes

For more details about the installation, see Ref. [24]. Besides *Python*, *MATLAB* version R2016b was used extensively for data processing.

3.2.1 Using a GPU for computation

Initially, we ran our code on the CPU, but after some weeks we got access to an Nvidia graphics card, of the type "GeForce GTX 1050 Ti", which enabled the use of *Compute Unified Device Architecture* (CUDA). CUDA is software developed by Nvidia that allows computations to be performed with the help of the GPU, which offers a significant speed-up compared to only using the CPU. This is because a GPU is capable of a much higher degree of parallel computing. Most of the neural network experiments in this thesis were performed with the help of CUDA Toolkit 10.1.

3.3 Neural network

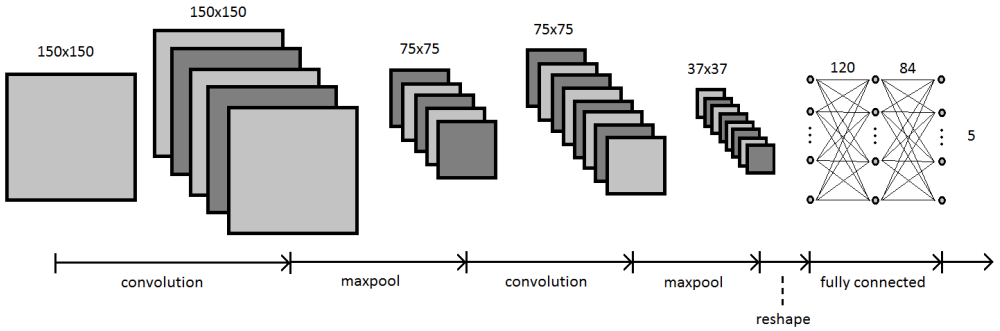


Figure 3.4: Schematic representation of our network

Our network is a modified version of LeNet-5 [25]. We use 2 convolutional layers, each followed by a max pool and a ReLU layer, and 3 fully connected layers, the first two of which are followed by a ReLU layer. The last layer outputs a vector containing a score for each output class. An overview of our network is shown in Fig. 3.4. Input image size was initially set to be 150x150 pixels, but this can be easily adjusted. Zero padding in the convolutional layers was initially set to 1.

3.3.1 Loss function

The loss function we used is called *cross entropy loss*. It is given by:

$$L\left(\boldsymbol{v}(\boldsymbol{x}), c\right) = -\log\left(\frac{\exp v_c(\boldsymbol{x})}{\sum_j \exp v_j(\boldsymbol{x})}\right) \tag{3.1}$$

Here $\boldsymbol{x}$ is an input image, c is the real class of the image (often referred to as *ground truth*) and $\boldsymbol{v}$ is the vector containing the network's output scores. Note that the argument of the logarithm is the softmax probability for the ground truth (see Eq. 2.2).

3.3.2 Optimizer

We used an optimizer called *Adam* [26]. It is given by the following scheme:

$$\begin{aligned} g_i &= \nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}_{i-1}) \\ \\ m_i &= \beta_1 m_{i-1} + (1-\beta_1) g_i \\ v_i &= \beta_2 v_{i-1} + (1-\beta_2) g_i^2 \\ \\ \hat{m}_i &= \frac{m_i}{1-\beta_1^i} \\ \hat{v}_i &= \frac{v_i}{1-\beta_2^i} \\ \\ \boldsymbol{\theta}_i &= \boldsymbol{\theta}_{i-1} - \alpha \frac{\hat{m}_i}{\sqrt{\hat{v}_i} + \epsilon} \end{aligned} \tag{3.2}$$

Here f is the loss function, α is the learning rate, m and v are exponential moving averages of the gradient and squared gradient, and β_1 and β_2 are hyper-parameters controlling the exponential decay rates of these moving averages. Note that in the case of β^i, i is a power, not an index. Furthermore, $\hat{m}$ and $\hat{v}$ are intialization corrected versions of m and v. This is necessary because this iterative scheme needs to start somewhere, meaning m and v need to be assigned some start value, the default of which is zero. Correction is needed for this start value to not have too great an impact on the early iterations. Lastly, ϵ is present to prevent a potential division by zero.

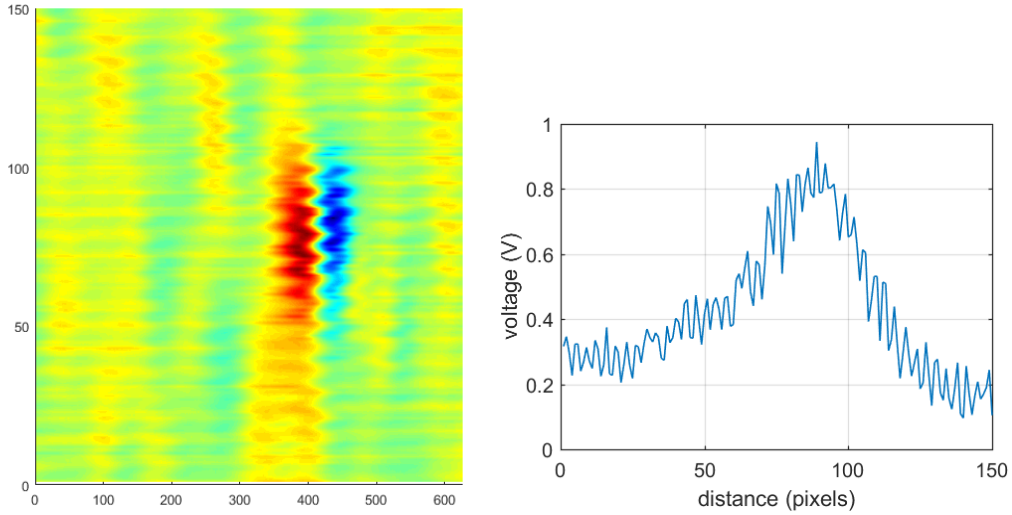
A useful property of Adam is that it is invariant to the scale of the loss function. If the f is multiplied by k, then both $\hat{m}$ and $\sqrt{\hat{v}}$ are also multiplied by k, therefore the update to $\boldsymbol{\theta}$ stays the same. Another useful property is that it has an adaptive learning rate. This means that it computes a seperate learning rate for each parameter, which also changes over time. This leads to significantly better convergence than a non-adaptive learning rate.

Chapter 4

Experiments and results

In this chapter, we will discuss the results of our experiments.

4.1 Analysis of scans obtained with the CFS setup



(a) (b)

Figure 4.1: a) Image of a scan around a particle affected by wobble. The detected particle is a 50 nm gold particle. The scan data was available to us from previous research with the scatterometry setup. Units are pixels. b) Vertical cut through the data at the positive peak of the particle

The data from the scatterometry setup contained a periodic variation, which we called *wobble*. The effect can be seen in Fig. 4.1.

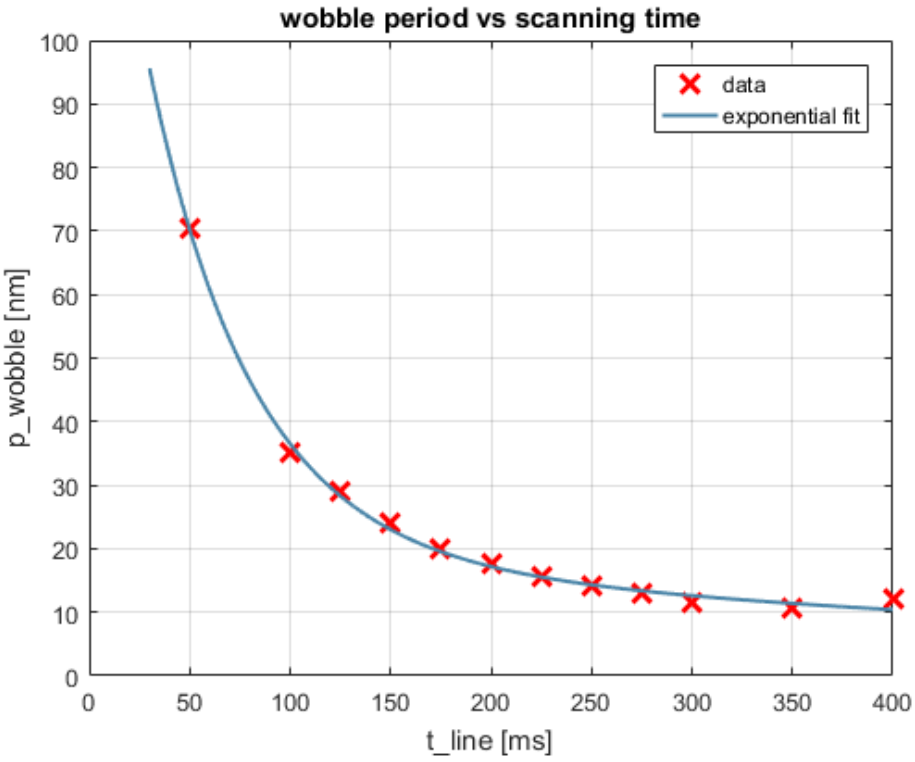
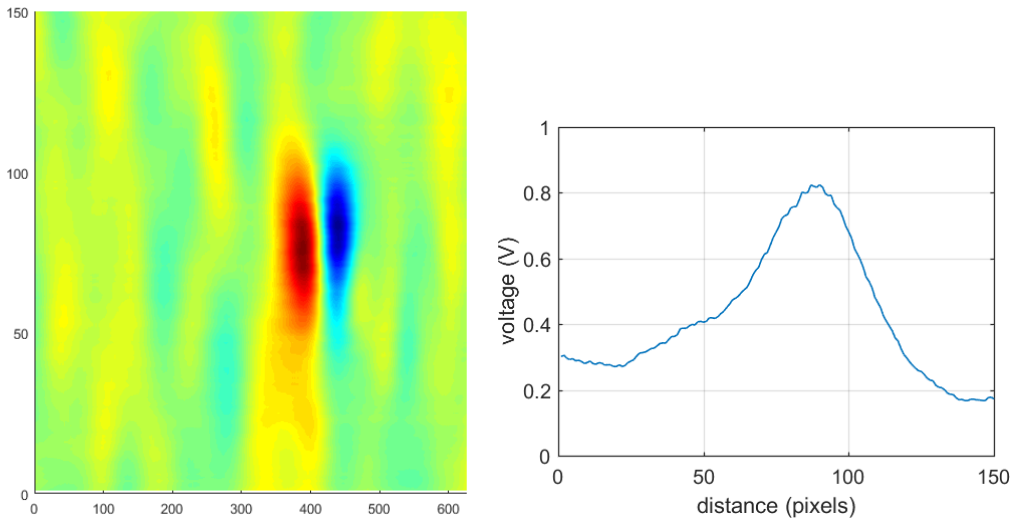


Figure 4.2: Wobble spatial period as a function of the scanning time per line.



(a) (b)

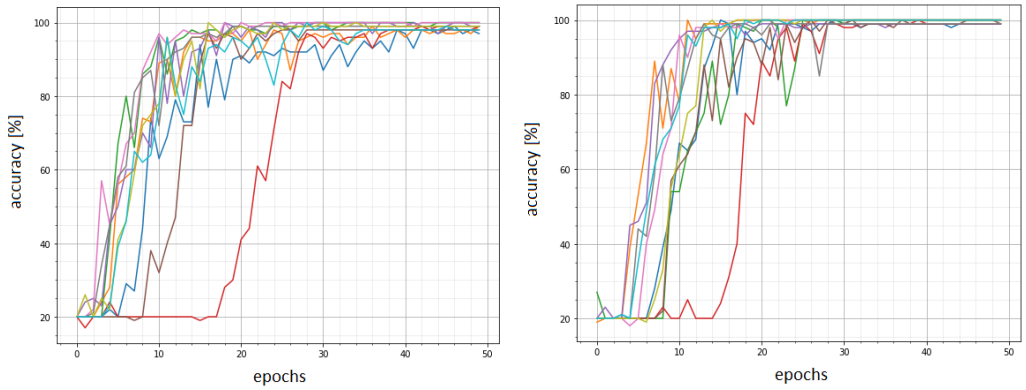
Figure 4.3: a) Scan data from Fig. 4.1a, smoothed. Units are pixels. b) Vertical cut through the positive peak of the particle, smoothed.

We observed that the spatial frequency of the wobble varied with the speed at which the stage moved during scanning, so we decided to measure the relation between the two. The result can be seen in Fig. 4.2. Since there is a clear dependency on the scanning time per line, it is likely that the cause of the wobble lies in the piezoelectric stage via mechanical vibration. However, since the wobble problem can be easily corrected with software while processing the data, we did not further investigate its cause. For the software correction, we used MATLAB to apply a simple local averaging filter, which updates the voltage $z(i, j)$ at each data point as follows:

$$z'(i,j) = \frac{1}{2n+1} \sum_{j=-n}^{n} z(i,j) \tag{4.1}$$

The number n was chosen to be 5, because this is the smallest value that still eliminates most of the wobble. The result is shown in Fig. 4.3. Given that the distance between our scanning lines is 2 nm, we are averaging over 22 nm.

4.2 Network tests on synthesized data



(a) (b)

Figure 4.4: a) Accuracy curves of the network with single layer padding, b) accuracy curves of the network with double layer padding.

In order to test the neural network early on, we used synthesized data. Each synthetic data image was created by the addition of triangular waveforms with opposite sign amplitudes, resulting in an X- signal that approximately resembles a particle detection's pulse. In the Y-direction, a number of these

signals were placed in parallel, with the amplitudes being taken from a Gaussian function.

We looked at both accuracy and stability, the latter being assessed by how flat the accuracy line was near the end of training. The most important result can be seen in Fig. 4.4. The network with double padding is clearly more stable after 30 epocs. Based on this, we made the network with double layer padding our default network.

For all results obtained with synthesized data testing, see Appendix A.

4.3 Data processing and its effect on particle detection data

In order to improve data quality, some processing was done on the raw data. First, a local averaging filter is applied as explaind in section 4.1. Then, in order to eliminate low frequency variations in the data, detrending is done in both dimensions by the *MATLAB* function `detrend()` (see Appendix C for a comparison between data with and without detrending).

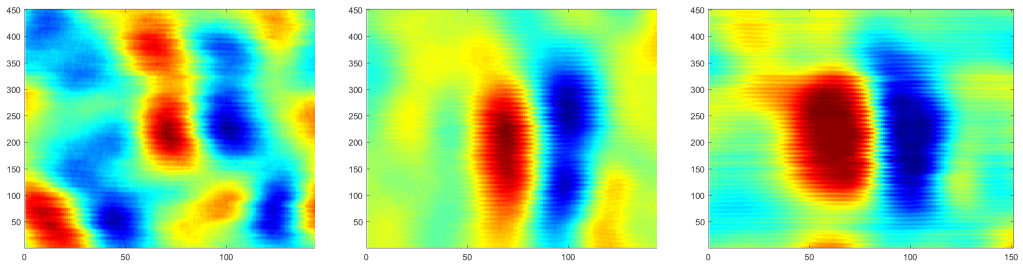
	80 nm	**60 nm**	**40 nm**
V_{pp} raw [V]	11.8	3.5	2.86
V_{pp} detrend [V]	11.8 (+0.0%)	3.5 (+0.0%)	2.86 (+0.0%)
V_{pp} smooth [V]	11.6 (-2.0%)	3.3 (-5.6%)	2.45 (-14.3%)
V_{pp} smooth + detrend [V]	11.6 (-2.0%)	3.3 (-5.6%)	2.45 (-14.3%)

Table 4.1: Results of smoothing and detrending tests

However, this data processing also has an effect on the size of particle detection peaks. The result is that for an 80 nm PSL sphere the V_{pp} of the detection signal decreases by 2%, while for a 40 nm PSL sphere it decreases by 14% (see Table 4.1). That number is caused by the local averaging filter, detrending has virtually no effect on it. This is something to keep in mind when trying to detect even smaller particles in the future, because the effect will most likely be even greater, which could limit our ability to spot detections in scans.

4.4 Network tests on real data

4.4.1 Network tests on 3 class scatterometry data



(a) 50 nm (b) 60 nm (c) 80 nm

Figure 4.5: Cuts of scan maps around particle detections of various sizes. The same scanning parameters were used to obtain data from the different samples that contained particles of different sizes. Units are pixels. These images were part of the dataset used to train the network. The 60 and 80 nm particles clearly stand out from the background, while the 50 nm particle does so much less.

In this section we show the results of the first network tests on real data.

The data consisted of 400 images containing cuts of scan maps around PSL sphere detections, divided roughly equally over 3 classes based on particle size: 80 nm, 60 nm and 50 nm. The data was obtained by making scans with the same laser power. This has one downside: using a high enough laser power to see smaller particles meant that the 80 nm particles scattered so much light that they saturated the detector. Because of that, the amplitudes of the 80 nm detections were lower than they should have been. However, they were still significantly higher than the amplitudes of the other categories, so it was our estimation that this should not be a limiting factor for classification.

Examples of images in the dataset can be seen in Fig. 4.5. The images were 150x150 pixels in size. We chose to use no more than 3 classes to make the network's task easier. After all, this was our first test on real data, and we did not yet know whether it would be any good.

It turned out that the network had no trouble with the dataset: accuracy in this experiment was consistently high and a test accuracy of 100% was achieved.

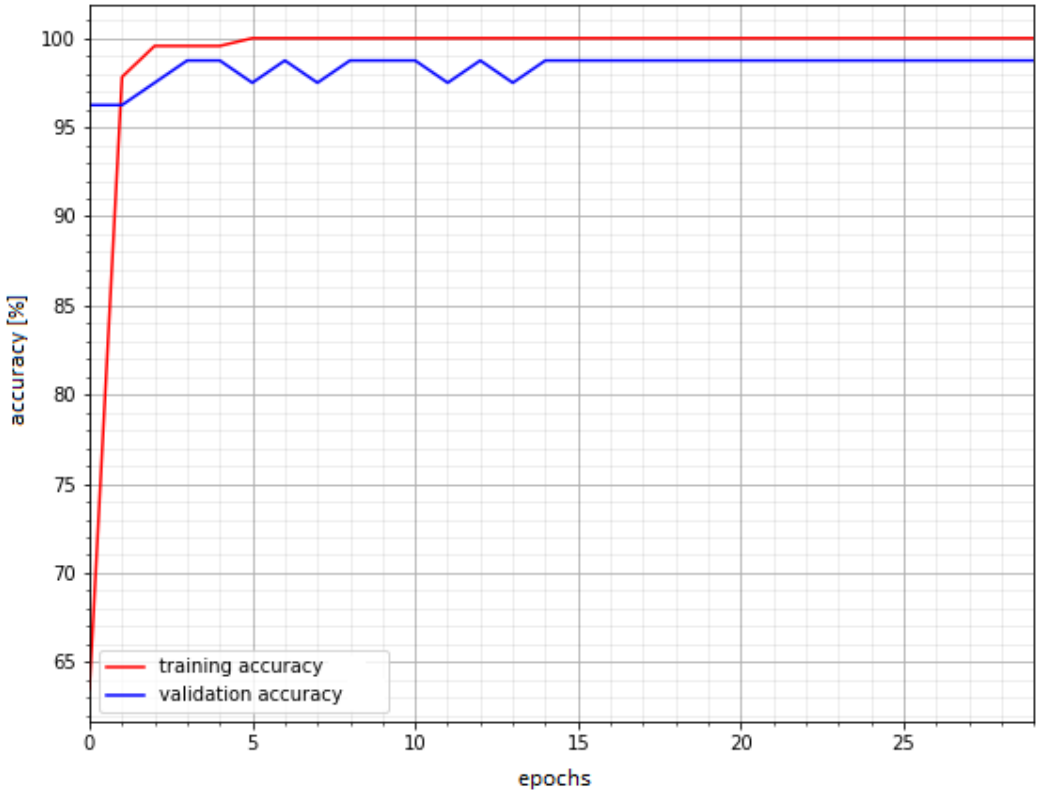
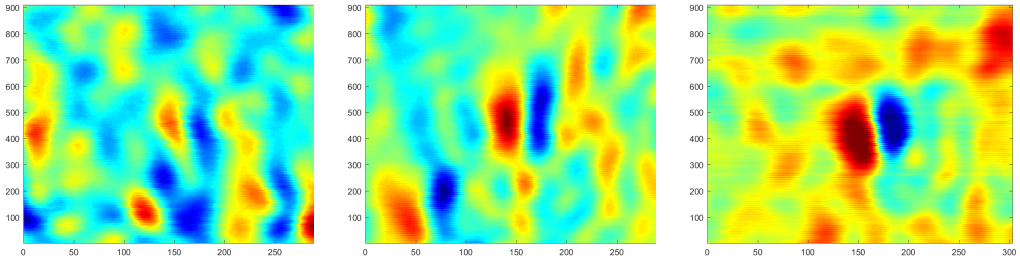


Figure 4.6: Accuracy of a 3 class network run on real data.

An example accuracy plot can be seen in Fig. 4.6. It is interesting to see that training accuracy shoots up to 98% after only 1 epoch, while validation accuracy is already high from the start (the latter is because the validation set is first passed through the network after the training set has already passed through once). Apparently, the relative simplicity of our inputs makes it so that very little training is needed.



(a) 50 nm (b) 60 nm (c) 80 nm

Figure 4.7: Cuts around particle detections. The physical area per cut is now 4 times as big as it was in Fig. 4.5. Units are pixels.

Then, we did the same experiment but with a larger area per cut of the scan map. This time, the cuts covered twice the distance both in the x and in the y direction, but they were downsampled to be 150x150 pixels in size, like the previous images. Examples of the new images can be seen in Fig. 4.7.

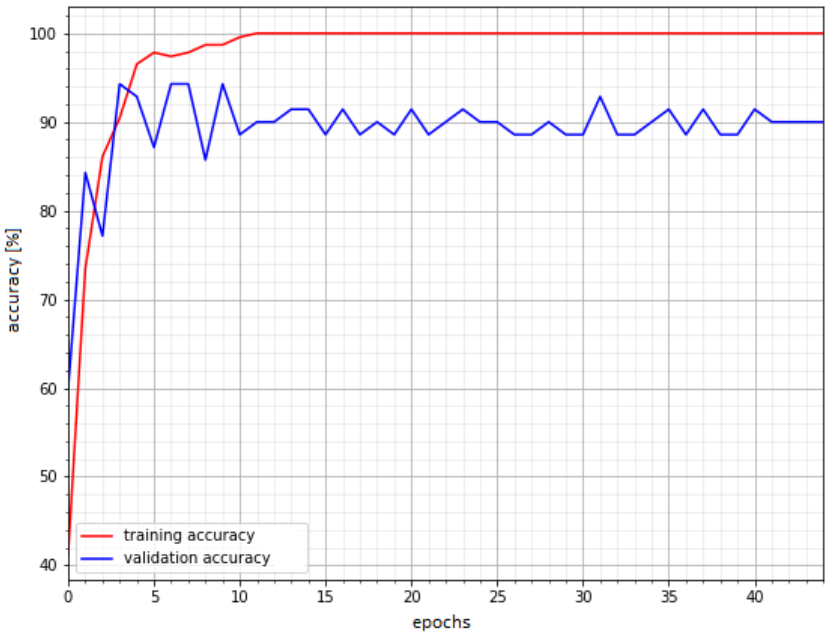


Figure 4.8: Accuracy of a 3 class network run on images that show a larger physical area.

After training the network with the larger cuts, the resulting accuracy was consistently lower than for the smaller cuts (see Fig. 4.8). The highest

test accuracy achieved for these images was 92%. However, we cannot be sure that this lower accuracy is caused by the larger cuts. It may also be caused by the information that's thrown away when downsampling to size 150x150. To put this to the test, an additional experiment was performed, where the wider cuts were instead put into the network as 300x300 pixel images. The result was a maximum test accuracy of 100%.

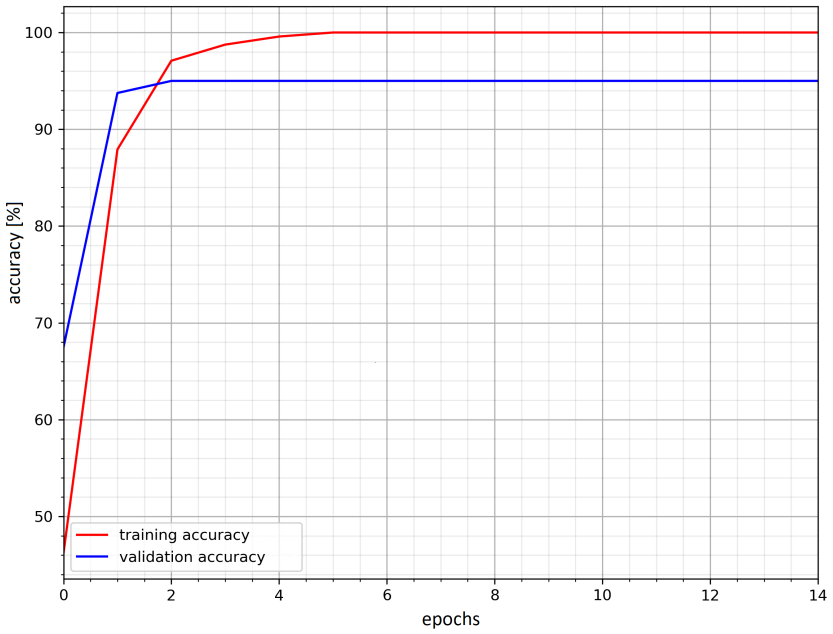


Figure 4.9: Accuracy of a 3 class network run on 300x300 pixel images

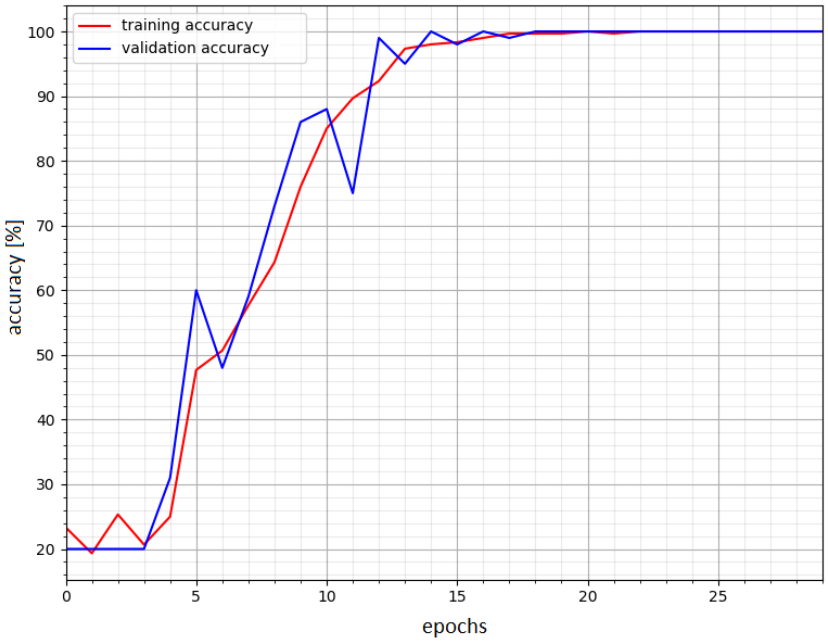


Figure 4.10: Accuracy curves of a network run on 5 class synthetic data. 100% accuracy was achieved on all sets.

It can be seen clearly in Fig. 4.6, 4.8 and 4.9 that the validation accuracy is consistently lower than the training accuracy, and does not converge towards it. That is unfortunate, because it means that the effect of training is limited. After a few epochs, it is no longer improving the network. This is a case of overfitting: the network learns the training set perfectly (100% training accuracy is achieved every time) but lacks the generalization power to perform as well on the other sets. For comparison, an example of an accuracy graph without this problem, generated with synthetic data, can be seen in Fig. 4.10. As we can see, there the validation accuracy nicely converges to 100%, just like the test accuracy.

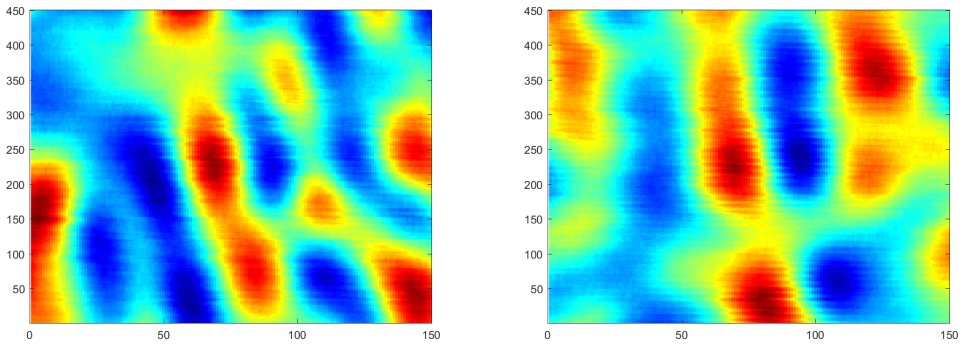
# classes	dataset	cut size	input size	test acc.
3	small	150x450	150x150	100%
3	small	300x900	150x150	92%
3	small	300x900	300x300	100%

Table 4.2: Results of 3 class network runs

For an overview of the results obtained with 3 classes, see Table 4.2.

4.4.2 Network tests on 4 class scatterometry data

Initial tests



(a) 40 nm (b) 50 nm

Figure 4.11: Examples of data from the 4 class dataset. Units are pixels.

After performing the 3 class tests, we gathered 40 nm particle data to have a 4th class. The main challenge here was that 40 and 50 nm particles are visually not very different on scans, so that labeling the dataset was difficult (see Fig. 4.11). However, the network had little trouble distinguishing between the two. An accuracy of 99% was achieved on the test set.

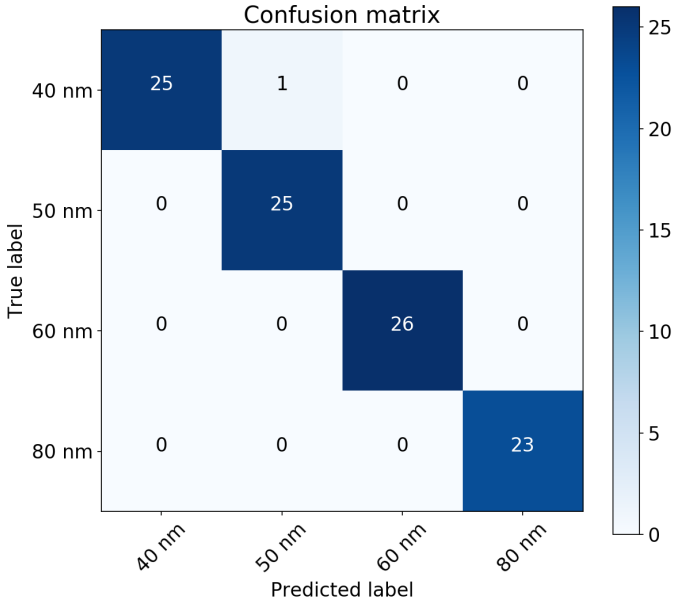


Figure 4.12: Confusion matrix for a network run on the initial 4 class dataset.

It is interesting to look at the confusion matrix (see Fig. 4.12). All misclassification is, as expected, between the 40 and 50 nm classes.

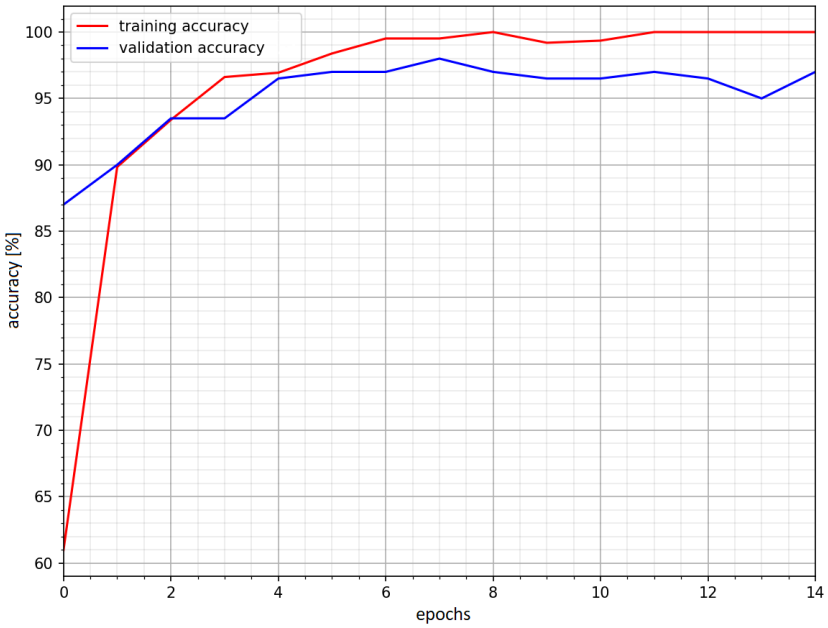


Figure 4.13: Accuracy plot for a network run on the large 4 class dataset.

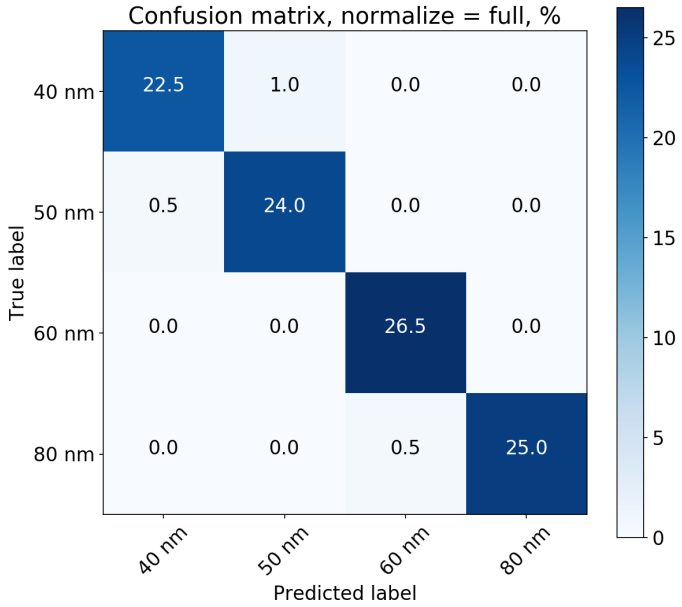
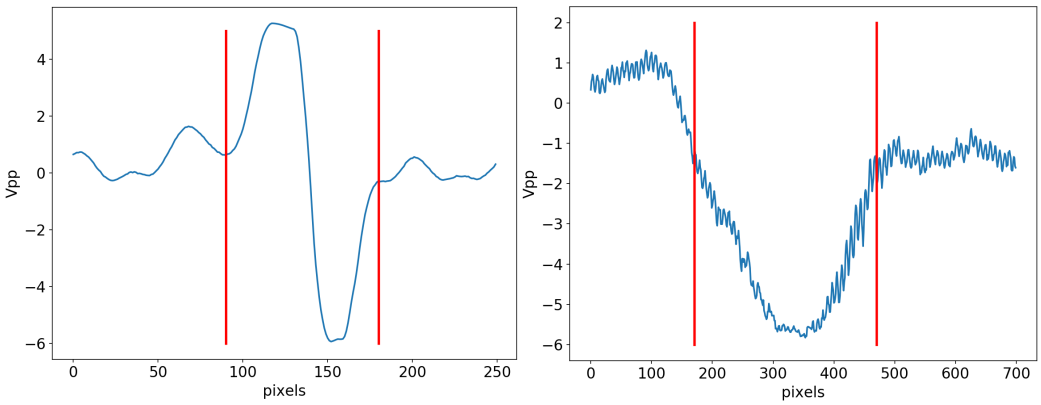


Figure 4.14: Confusion matrix for a network run on the large 4 class dataset.

Because the accuracy curves showed signs of overfitting, we thought increasing the size of the dataset might have a positive effect on our generalization power. More cuts were made until the dataset had 1055 images, spread roughly equally over 4 classes. The result can be seen in Figs 4.13 and 4.14. The validation accuracy curve still remains consistently lower than the training curve, and though overall accuracy is high, the highest test accuracy achieved was 98%.

Statistics and further tests



(a) (b)

Figure 4.15: a) X-slice through an 80 nm particle detection, the red lines indicate the width of the pulse. The somewhat flattened top and bottom are due to saturation of the detector for the relatively high amount of light scattered by 80 nm particles. b) Y-slice through an 80 nm particle detection, the red lines indicate the length of the pulse

In order to distinguish between classes, we saved three numbers for each particle during data labeling: peak-to-peak voltage, width and length. The width and length are defined as shown in Fig. 4.15.

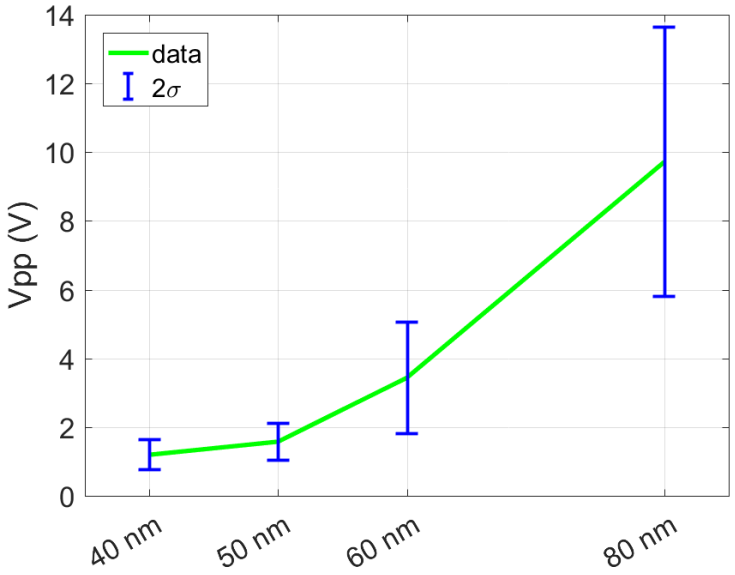


Figure 4.16: Peak-to-peak voltage of particle detections in the dataset. Error bars represent 2σ.

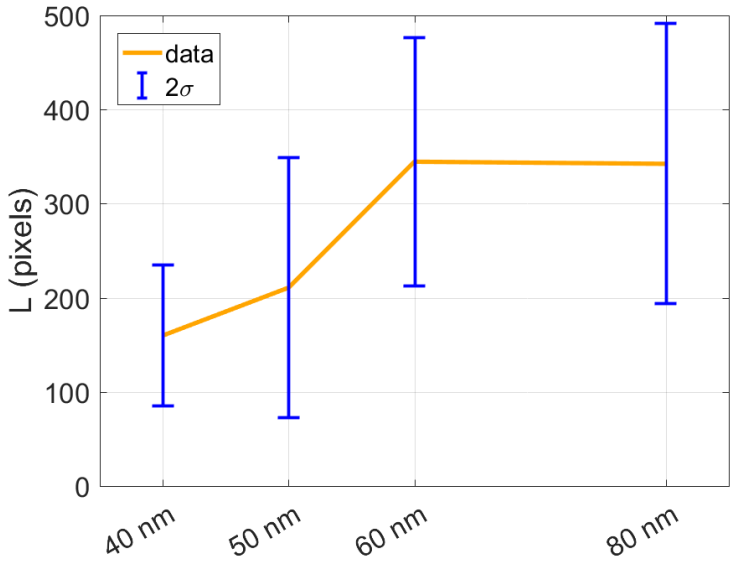


Figure 4.17: Length of particle detections in the dataset. Error bars represent 2σ.

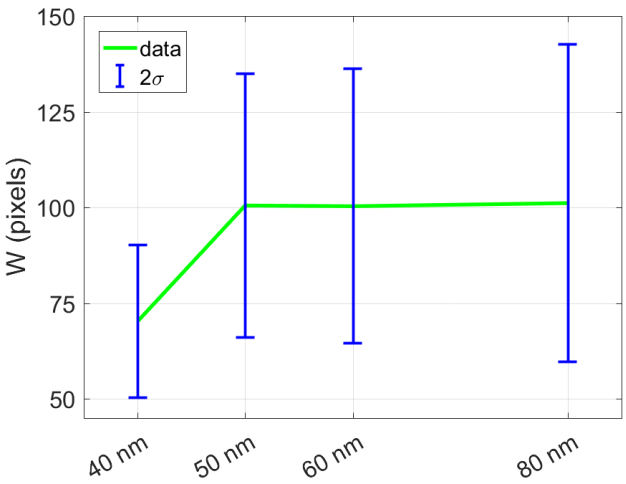


Figure 4.18: Width of particle detections in the dataset. Error bars represent 2σ.

We then took a look at the average and standard deviation of each of these statistics for each class. The results can be seen in Figs 4.16, 4.17 and 4.18. Note that error bars represent 2 times the standard deviation. Keep in mind that these statistics are for the network inputs, which consist of processed data. The most important part of the data processing is the local averaging filter described in Section 4.1.

There are two interesting points here. The first is that the average length decreases between 60 and 80 nm. This is unexpected. Maybe it has something to do with the way the samples were fabricated, or perhaps the particle sizes are not as accurate as we'd like.

	Small dataset	**Large dataset**	**Reduced dataset**
40 nm	127	254	241
50 nm	127	253	175
60 nm	127	276	259
80 nm	127	272	272
Total	508	1055	947

Table 4.3: Comparison of amount of images per class in each dataset. Small is the original set, large refers to the addition of data in an attempt to reduce overfitting and reduced is the dataset where images were removed to reduce overlap in V_{pp} between classes.

The second important thing is that the overlap between classes in length

and width is huge. Appearently, of these three properties, V_{pp} is the only one that can be used for class prediction. This lead to a question: can we improve network accuracy by entirely removing the already small amount of overlap in V_{pp}? To test this, we removed the images with a particle V_{pp} that was too close to another class from the dataset. This reduced in size mainly the 50 nm category, as it had overlap with both 40 and 60 nm. See Table 4.3 for a comparison between datasets. The result of reducing the dataset was a test accuracy of 100%.

# classes	dataset	cut size	input size	test acc.
4	small	150x450	150x150	99%
4	large	150x450	150x150	98%
4	reduced	150x450	150x150	100%

Table 4.4: Results of 4 class network runs

For an overview of the results obtained with 4 classes, see Table 4.4.

4.4.3 Network tests on 5 class scatterometry data

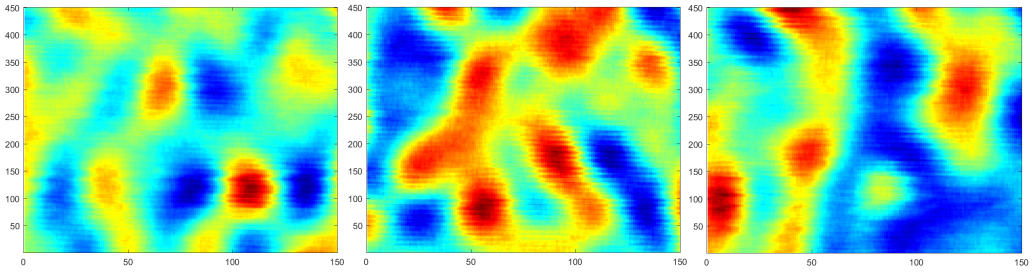


Figure 4.19: Examples of background images.

Up to this point, we only had classes for images that contained a particle detection. What was missing was a class for images that do not contain one, and only show the background coming from the substrate. Such images still show a lot of shapes. The question was whether the network could distinguish these shapes from particles. Examples of background images can be seen in Fig. 4.19.

We added a class with 247 background images. The reason for this size is that we wanted the background class to be comparable in size to the particle classes. The resulting accuracy was consistently lower than before, but not by

much. On the full dataset with added background category, a test accuracy of 93% was achieved. On the reduced dataset with added background category, a test accuracy of 95% was achieved.

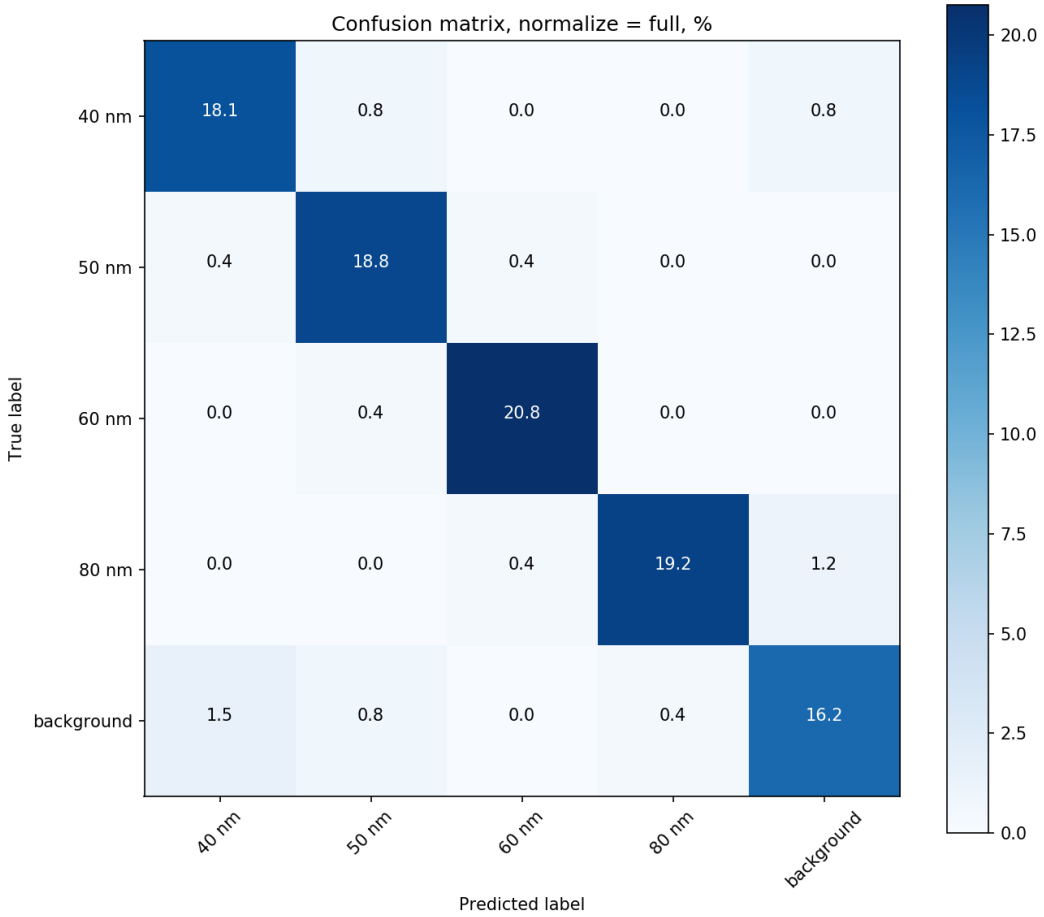


Figure 4.20: Confusion matrix for a network run on the full 5 class dataset.

In Fig. 4.20, the confusion matrix for the full dataset is displayed. It is clearly visible that most misclassification involves the background class, as expected.

Batch normalization

Initially, batch normalization was not used because relatively small datasets limited us to small batch sizes, for which batch normalization does not improve results [29]. However, we now had a dataset of large enough size to use larger batches and try out various batch sizes. Previous 5 class results were obtained with a batch size of 20. We decided to now test batch sizes between

5 and 70, and use two networks for this: the one we had been using, and one with BatchNorm layers added between the other layers. For the exact difference, see Appendix F.

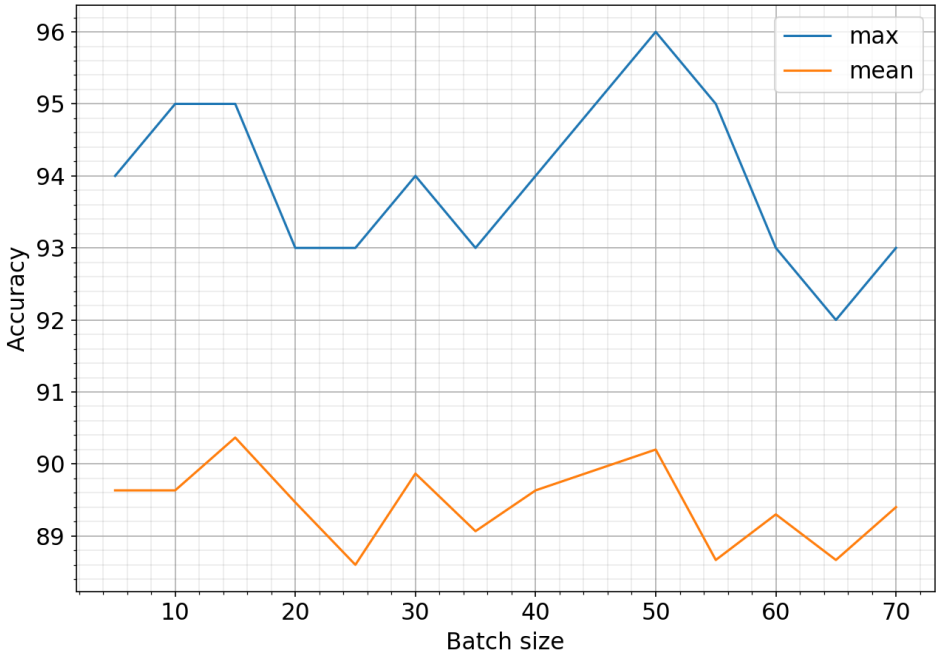


Figure 4.21: Accuracy vs batch size of a network without batch normalization.

The result of the basic network can be seen in Fig. 4.21. Interestingly, using batch size 50 with our basic network produced a test accuracy of 96%, something we had not seen before. However, if we compare the graph of the max to the graph of the mean, we see that this is most likely just random luck, an outlier. Another interesting point is that the batch size 20, which we had been using, was not a good choice. Maxima clearly occur at 15, 30, and 50.

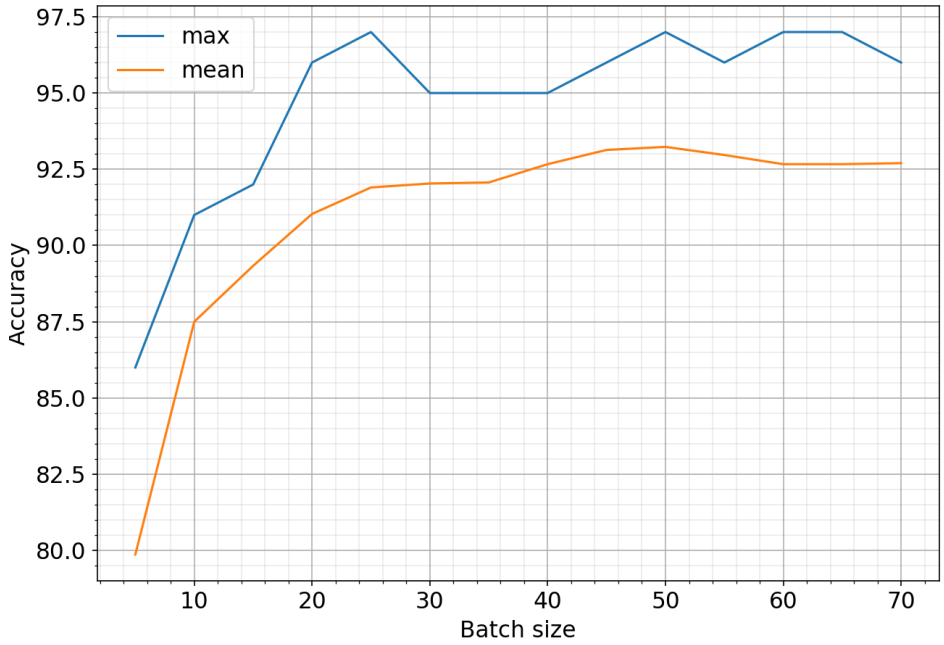


Figure 4.22: Accuracy vs batch size of a network with batch normalization.

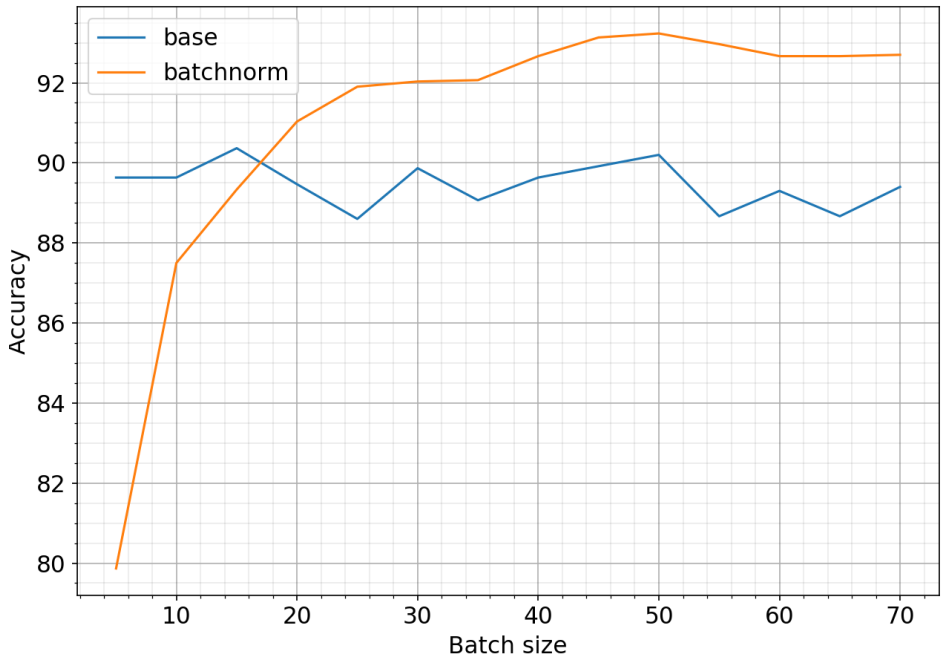


Figure 4.23: Comparison between the two networks of mean accuracy vs batch size.

In Fig. 4.22 the result of the network with batch normalization can be seen. As expected, it performs poorly for small batches, and climbs up after that. The interesting part is that at 4 different batch sizes, an accuracy of 97% was achieved. This is beyond what we expected to be possible for our dataset. In Fig. 4.23 a comparison can be seen to the base network (without batch normalization). There is a clear crossing point at roughly 17 images per batch, beyond which the batch normalization network consistently performs better.

Based on these figures, we made the batch normalization network the default, with a batch size of 50. There is one thing to be aware of though: because batch normalization needs a significant batch size, it cannot be used for single image classification. This means that training our network and then feeding it images one by one is not possible. This could be a problem if a network like this is ever implemented with the goal of classifying particles real time as the data comes in.

4.5 A note on reproducibility

Ideally, results in a scientific experiment have to be reproducible. However, when using neural networks, this is unfortunately not always possible. We performed tests where all random number generators in Python were seeded at the start of the code via the functions `random.seed()`, `numpy.random.seed()` and `torch.manual_seed()`. While this greatly reduced the variation in outcomes compared to unseeded runs, there was still significant variation present. According to the PyTorch website, this is due to the use of certain CUDA functions [28]. To verify this, we performed a test where all CUDA code was disabled and the default device was set to be the CPU. That indeed gave us a fully reproducible network run. So when it comes to experiments, loss of reproducibility is a fundamental downside of using a GPU via CUDA.

It is important to realize that this is not limited to training. Even when passing the same image to a fully trained network twice, it might give two different predictions.

4.6 Input size

It occurred to us that our maximum square input size of 150x150 need not be the best input size. We created new datasets by downsampling the 5 class dataset with MATLAB to various sizes. The result can be seen in Fig. 4.24. There are several interesting observations to be made.

Firstly, our maximum accuracy of 97% is achieved at an input size of 100x100. Appearently, the loss of information compared to 150x150 is so small that it does not influence our highest achievable result.

Secondly, the curve drops only very little as we lower input size until we get to such small sizes that there are barely any pixels left. Even 10x10 pixel images result in a surprisingly high maximum accuracy of 92%. If we require 95%, we can go down to an input size of 40x40.

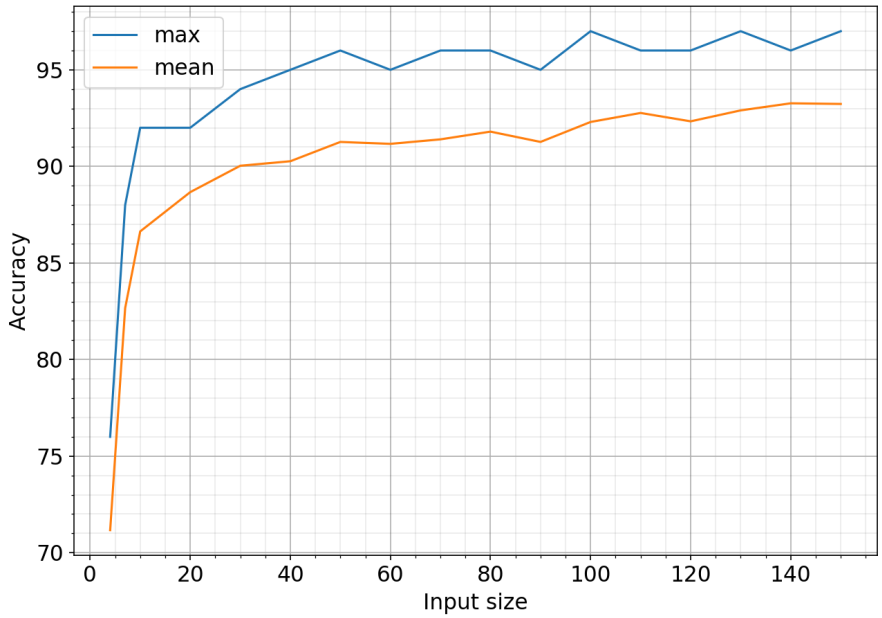


Figure 4.24: Accuracy vs input size in pixels. All inputs were square.

4.7 Loss landscapes

In this section, we will discuss loss landscapes as introduced in section 2.3.2.

4.7.1 Random planes

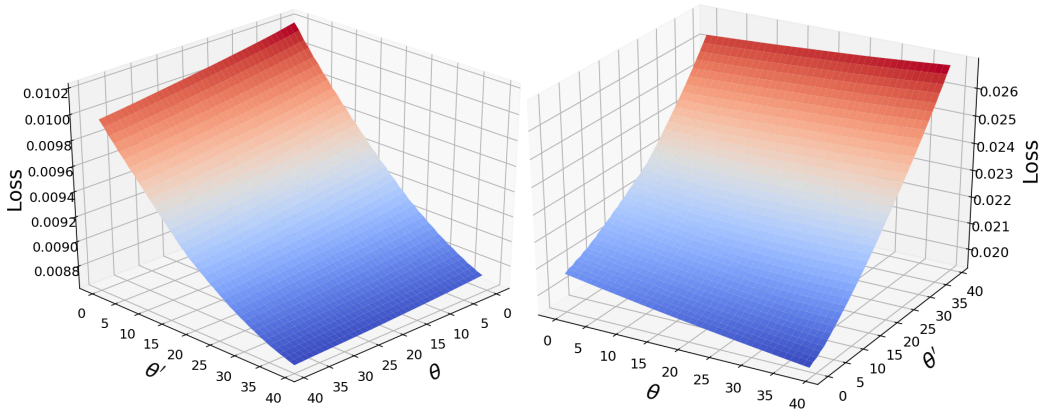


Figure 4.25: Two randomly generated loss landscapes. Θ and Θ' denote the amount of steps taken in the randomly chosen directions. Loss is a value calculated by the loss landscapes package based on cross entropy loss.

For PyTorch, a package is available online that plots loss landscapes. [31]. This package makes use of random planes, meaning it generates two random directions in which it calculates the loss at certain intervals. However, due to the random nature, most of the times they showed nothing of interest. Examples can be seen in Fig. 4.25. The units on the Z-axis show that the variation is small over the landscapes. They are also smooth and almost planar in nature, without features of interest. This is typical for randomly generated loss landscapes.

4.7.2 Principal component analysis

Then, we decided to use principal component analysis in an attempt to find dimensions along which the loss varies more, and hopefully gain more information from the resulting loss landscapes. For this, we used the `PCA` function from the module *sklearn.decomposition*.

The result can be seen in Fig. 4.26. This looks promising, because the variation in loss is large here (confirming that the PCA method does what we expect) and there is a minimum visible. We also see a fully convex landscape, which is expected for a neural network with relatively few hidden layers. [20]

However, after generating many landscapes, it became clear that a minimum is always visible. This is strange, because given our relatively small effect of training and large effect of random weight initialization, we cannot always end up near a local or global minimum. It therefore seems like the

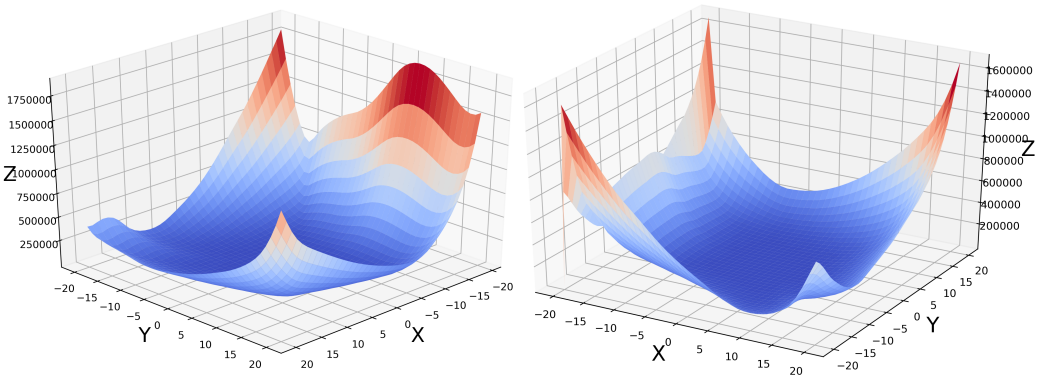


Figure 4.26: Two loss landscapes generated using PCA. X and Y denote the amount of steps taken in the PCA directions. Loss is the total cross entropy loss value resulting from passing the entire dataset through the network.

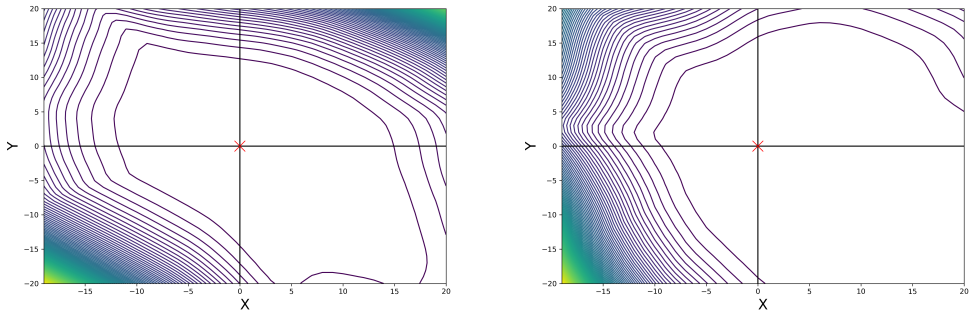


Figure 4.27: Two contour plots of loss landscapes with the minimum marked as 'x'.

minimum shown by the loss landscape is, in the full dimensional space of the weights, not a minimum at all.

To investigate this further, we plotted the loss landscape as contour plot and overlayed the lines x=0 and y=0. Then, we plotted the minimum of the landscape as a red 'x'. The result can be seen in Fig. 4.27. What these plots show us is that the minimum is always in the center. This is interesting, because the center represents the final state of the network during training, which means that PCA loss landscapes always show us a landscape that has a minimum at the end of our training run. That limits their use, because we then cannot use them to discover a place where the loss is lower than at the place our network is already in.

4.8 Open set recognition

There is one problem with the network we have used so far: it can only put data into one of the classes that it was trained for. If we were to feed the network an image of a 20 nm particle, it would not be classified as 20 nm, because we have no such class. Therefore, it would most likely be put into the background class. Just like if we were to detect some strange large structure, it would most likely be put into the 80 nm class. And if we were to think entirely out of the box and feed a picture of an animal to the network, it would return one of our particle classes too. This is undesirable behavior. Ideally, we want the network to not only classify an image, but also return an estimate of whether it belongs to the classes that the network was trained for at all, or whether instead it should be classified as unknown input. This is known as *open set recognition*. Among other things, open set recognition can be used for *anomaly detection* (detecting outliers in the data) and *novelty detection* (detecting inputs that are nothing like the trained classes).

Open set recognition is based on the presence of some algorithm that does an additional calculation on the network outputs. In this section, we will look at three such algorithms.

4.8.1 Probability thresholding

The simplest method of open set recognition is to threshold the output probabilities. This approach was also shown by M. Landgren and L. Tranheden in Ref. [32] (referred to as "baseline" in their thesis). We implemented it as follows:

1. Apply the softmax function to the network outputs to convert them to probabilities
2. Take the maximum probability p_{max} and compute $u = 1 - p_{max}$ as a measure of uncertainty
3. If u is above a certain threshold, reject the classification

There is one potential problem with this method: we cannot be sure that there is a clear relationship between high uncertainty and incorrect classification. To put this to the test, we used the method on the 5 class reduced dataset downsampled to 100x100 pixel images.

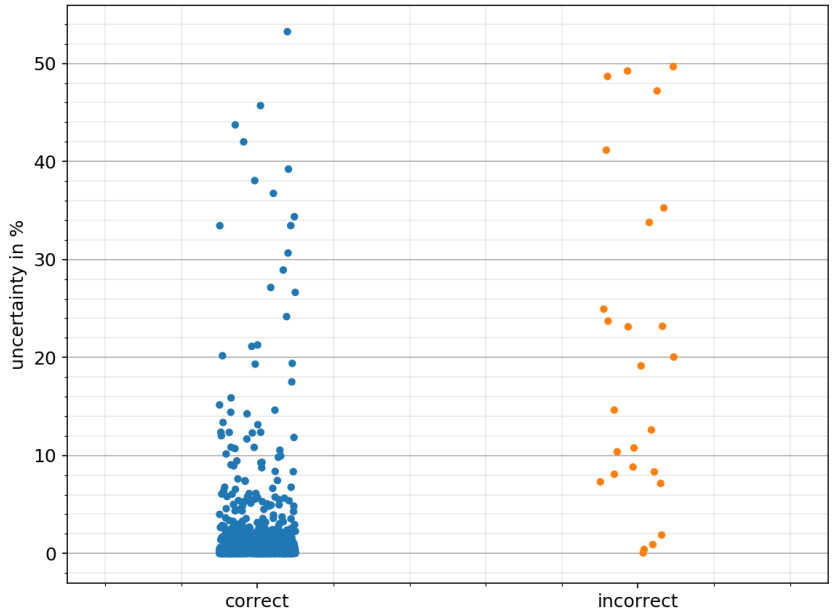


Figure 4.28: Strip plot to visualize the distribution of uncertainty values for correct and incorrect classifications.

In Fig. 4.28 the result can be seen of passing the entire dataset through a trained network. The correct data points look good: the vast majority of them lie in the very low uncertainty region. However, the incorrect ones span nearly the same range as the correct ones, meaning we cannot get rid of incorrect classifications by thresholding uncertainty. No matter what threshold we would set, we would always have some incorrect classification remaining while also throwing out a portion of the correct ones.

Fooling datasets

Next, we tested some datasets that contain images that don't belong to the trained classes. We will refer to these as *fooling datasets*. The first one is a mirrored dataset: this is simply the same dataset but with every image mirrored in the x-direction (the direction along which the stage moved during scanning), so that each positive-negative pulse of a particle detection becomes negative-positive.

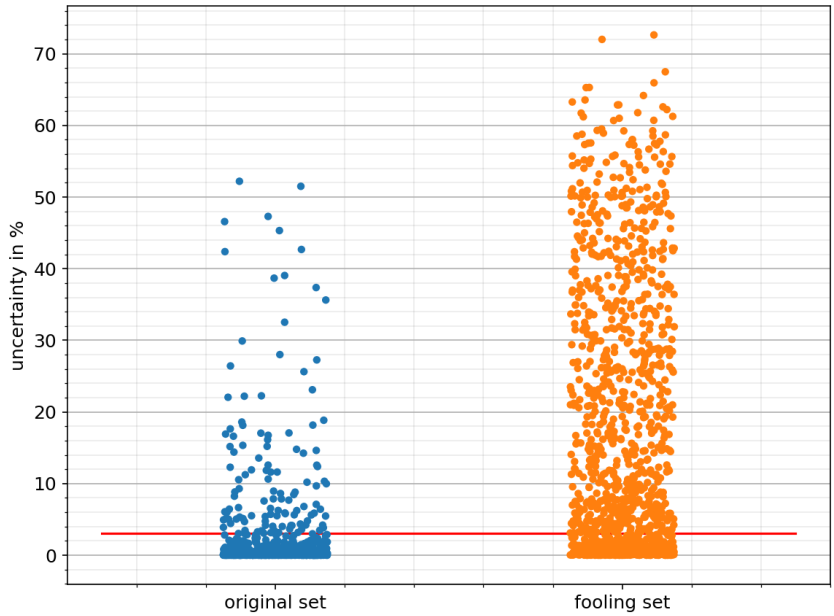


Figure 4.29: Strip plot to visualize the distribution of uncertainty values for the dataset containing images that were mirrored in the x-direction. 'original set' refers to normal data and 'fooling set' refers to the mirrored images. The red line indicates the threshold above which data is rejected from being classified.

As we can see in Fig. 4.29, the result is not ideal. The values range all the way from 0 to 72%, but the density is very high at low uncertainty, meaning that we can not get rid of all the mirrored images by thresholding. However, we can choose a threshold in such a way that we sacrifice a small portion of normal data to gain the ability to reject a large portion of the fooling data. We chose this threshold such that we reject approximately 10% of normal data. This is an arbitrary value, the acceptable maximum rejection of normal data would depend on application, and on how frequently images appear in the data that should be rejected.

The result of our experiment was that for a threshold of 3.1% (indicated by the red line in Fig. 4.29), we get rid of 68.3% of the mirrored images. At the same time, we throw away 10.3% of the real particle data.

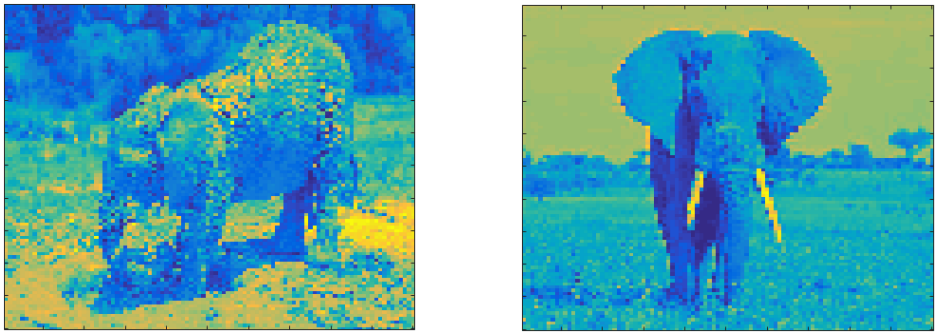


Figure 4.30: Two examples of images in the elephant dataset, visualized with MATLAB's `image()` function and default colormap.

Next, we tested a dataset that's entirely different from anything the network normally sees. This is to see how certain the network is on images that have no similarity to real data. For this, we used a dataset consisting of 1165 images of elephants, taken from a larger dataset called Animals-10 [37]. Examples can be seen in Fig. 4.30.

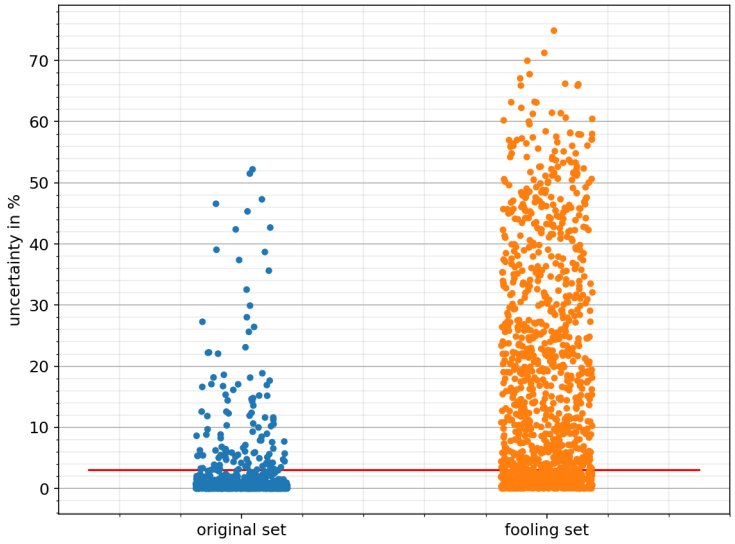


Figure 4.31: Strip plot to visualize the distribution of uncertainty values for the dataset containing images of elephants. 'original set' refers to normal data and 'fooling set' refers to the elephant images. The red line indicates the threshold above which data is rejected from being classified.

The result can be seen in Fig. 4.31. The distribution is quite similar to the one of the mirrored set. However, our result is slightly better this time: for 10.3% rejection of normal data, we rejected 71.9% of the elephant images.

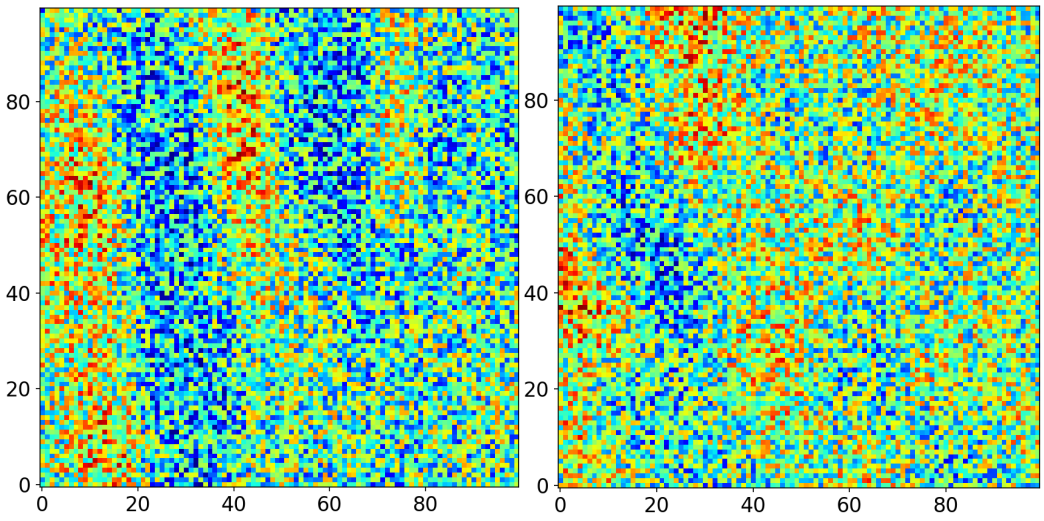


Figure 4.32: Examples of data from the noise set, a) 80 nm, b) 50 nm.

Both the fooling sets we used so far fall under novelty detection: they are something that will not be present in normal data. It would be interesting to also test a form of anomaly detection. To do that, we took the dataset and added noise. This was done by taking the voltage range of an image (by substracting the minimum from the maximum) and adding uniformly distributed noise with 1.5 times that range as amplitude. Examples of noise images can be seen in Fig. 4.32.

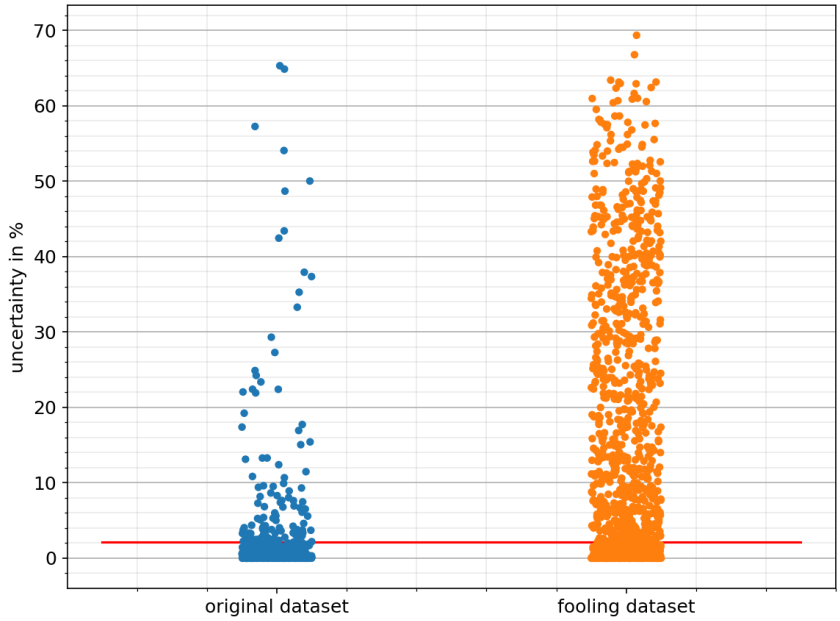


Figure 4.33: Strip plot to visualize the distribution of uncertainty values for the dataset with added noise. 'original set' refers to normal data and 'fooling set' refers to the noisy images. The red line indicates the threshold above which data is rejected from being classified.

See Fig. 4.33 for the resulting distribution. At a normal data rejection of 10.0%, the network rejected 65.3% of the noise set.

4.8.2 Activation vectors

Maximum class probability does not give us all the information that is contained in the network outputs. For that, we must use the vector containing all the network outputs, which we will call the activation vector. In Ref. [33], Bendale and Boult save the activation vectors of correct classifications and calculate the mean activation vector (MAV) per class. They then calculate the distance of each new activation vector to the MAV of its respective predicted class. In the paper, this information was used for the OpenMax algorithm. Here, we decided to first apply it directly.

We implemented an algorithm as follows:

1. Calculate the MAV for the correct classifications of each class

2. For each image x in the training and validation sets, obtain the activation vector $v(x)$ and predicted class $c(x)$. Then, calculate the distance

to the MAV with $d = ||v(x) - \text{MAV}_{c(x)}||$. Save values of d seperately for correct and incorrect classifications.

3. For each image in the test set, calculate d the same way, and if it is above some threshold, reject the classification (thus classifying it as unknown)

Excluding incorrect classification

The first thing we did was attempt to set the threshold such that all misclassification is excluded.

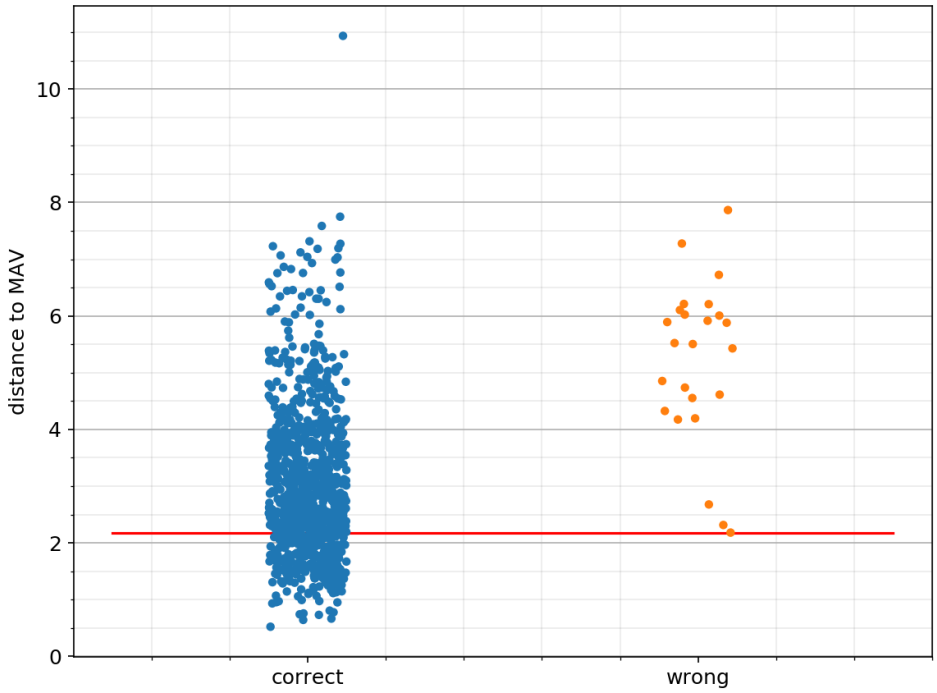


Figure 4.34: Distributions of the distance to the corresponding mean activation vector of each prediction. The red line indicates the minimum distance of incorrectly classified points, below which all classifications are correct.

To illustrate this, we plotted the distribution (see Fig. 4.34). There are many more correct than incorrect classifications due to the high accuracy of the network. It can be seen clearly that a significant part of the correct classifications lies below the lowest of the incorrect ones. That means we can separate them. We will refer to the set of classifications that lie below the red line in Fig. 4.34 as the *high confidence set*. If the set of images in our training and validation sets was representative for the whole dataset, this

means that all test classifications in the high confidence set will be correct. In the test where we captured the above distributions, this was indeed the case. The accuracy of the network on the entire test set was 95.0%, while accuracy on the high confidence set was 100%. However, the high confidence set consisted of only 14.5% of the test images.

To summarize: it is possible to add code that will, on top of an output class, return whether or not the network is highly confident, and if so, the chance for the classification to be correct is 100%, provided that the threshold is correctly chosen. In other words, the network is very sure of some particles, but we have no control over which particles. This means that, while interesting, this technique is not very useful.

Fooling datasets

Next, we tested the fooling datasets again, to see if the MAV approach yields a better result than probability thresholding. First, we tested the mirrored set.

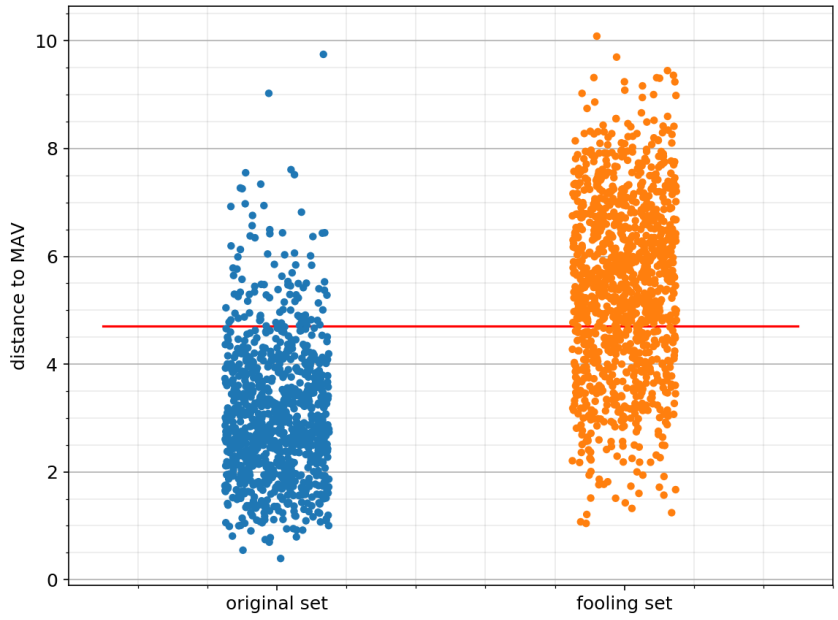


Figure 4.35: Strip plot of the distributions of distance to the MAV of each datapoint. The fooling set is the mirrored dataset. The red line indicates the threshold above which data is rejected from being classified.

Fig. 4.35 shows the result. At the threshold which rejected 10.1% of the original dataset, we rejected 68.3% of the mirrored images.

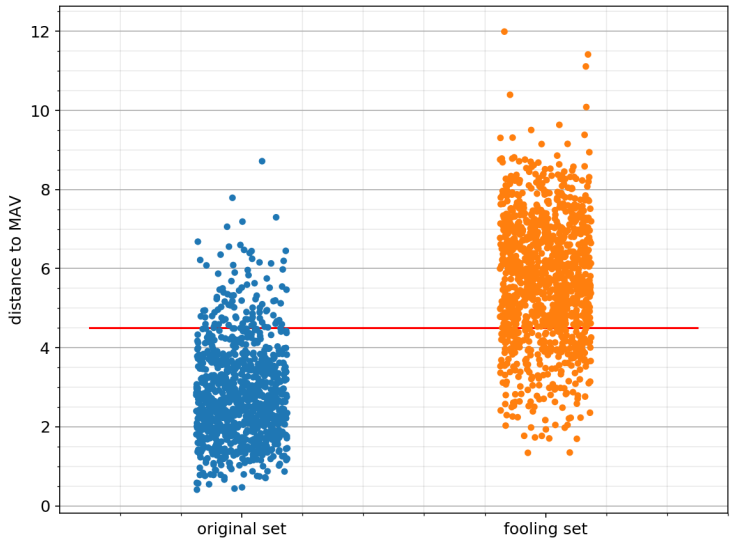


Figure 4.36: Strip plot of the distributions of distance to the MAV of each datapoint. The fooling set is the elephant dataset. The red line indicates the threshold above which data is rejected from being classified.

Next up is the elephant set. As we can see in Fig. 4.36, the distribution looks slightly better than the previous one. At a threshold which rejected 10.3% of the original dataset, we rejected 79.7% of the elephant images.

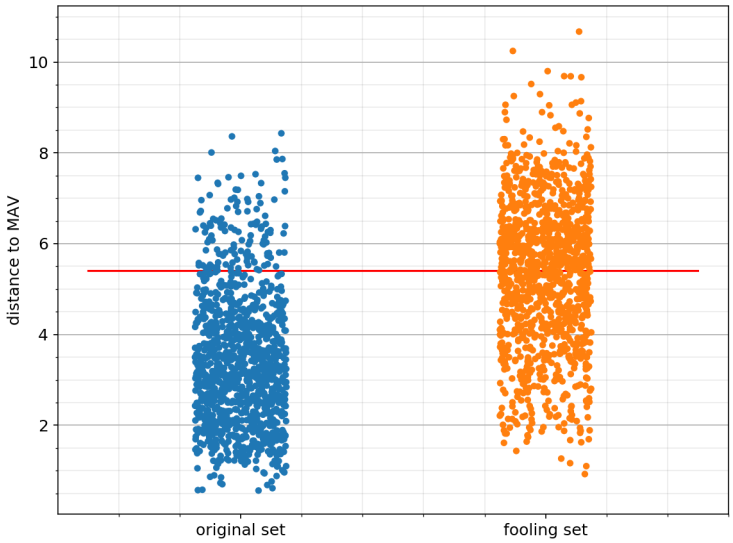


Figure 4.37: Strip plot of the distributions of distance to the MAV of each datapoint. The fooling set is the noise dataset. The red line indicates the threshold above which data is rejected from being classified.

Lastly, we tested the noise dataset. The result can be seen in Fig. 4.37. While rejecting 10.8% of the original data, we rejected 51.9% of the noise data.

4.8.3 OpenMax

OpenMax is an extension of SoftMax that adds a probability for the input to be of an unknown class. This is based on Extreme Value Theory (EVT). Extreme value distributions are the distributions that result from the maxima of a large collection of random samples from an arbitrary distribution [34]. In our case, the "arbitrary distribution" is the distance to the MAV of a large number of input images. It has been proven that if a system has multiple failure modes, the extreme values are best modeled by the Weibull distribution [35]. This is given by [36]:

$$W(z) = \frac{\tau}{\lambda}\left(\frac{z-\tau}{\lambda}\right)^{\kappa-1} e^{\left(\frac{z-\tau}{\lambda}\right)^{\kappa}} \tag{4.2}$$

Here $\tau > z$, $\lambda > 0$ and $\kappa > 0$.

In OpenMax, the function FitHigh() from the code library *libMR* is used to fit the extreme values of our distribution to a Weibull distribution. This function uses a maximum likelihood estimate to estimate τ, κ and λ. This is done per class, and only data from correctly classified images is included. These Weibull models can then be used on new inputs to generate a per image probability that the image does not belong to the trained classes and should be rejected. For that, the method selects the m highest activations per activation vector.

This last point is why we expected this approach to not yield us good results. Our data has only 5 classes, meaning our activation vectors have only 5 entries. There is not much selection possible in this case. Figure 1 of Ref. [33] visualizes activation vectors for a 450 class system and provides some intuition about where the information resides that is used in OpenMax. It is clear that for a 5 class system, these visualizations would be very discrete, and a lot less information could be gained for them.

Nevertheless, we still wanted to implement OpenMax in our network code to see what it can do. Our implementation is based on the code that was used in Ref. [32], which in turn is based on the code that was used in Ref. [33].

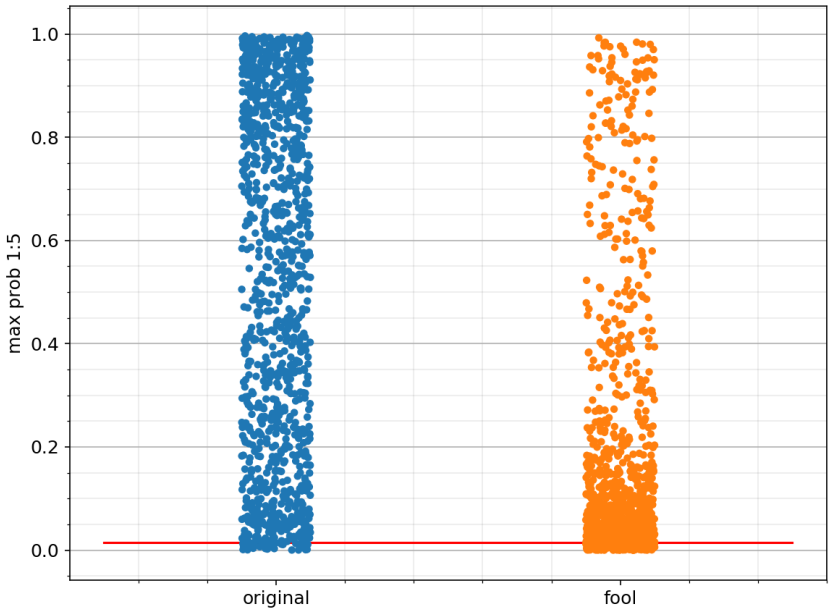


Figure 4.38: Plot of the maximum probability that the input image belongs to the first 5 classes. "Fool" refers to the mirrored dataset. The red line indicates the threshold for which roughly 10% of the original dataset is rejected.

Results

First, we tested OpenMax on the mirrored dataset. The resulting distributions can be seen in Fig. 4.38. The original dataset is very spread out, while the mirrored set clearly sits more towards the low end. However, the results were not very promising: while rejecting 11.1% of the original dataset, our OpenMax implemenation only rejected 19.1% of the mirrored set.

If we look at the distributions, it seems more intuitive to threshold at 0.2. There, we are rejecting a much larger part of the mirrored set. Testing this gave us a rejection of 74.7% on the mirrored set, but also a rejection of 50.3% on the original dataset. While the difference between the two sets is now larger, this does not seem very useful, we're rejecting too much normal data.

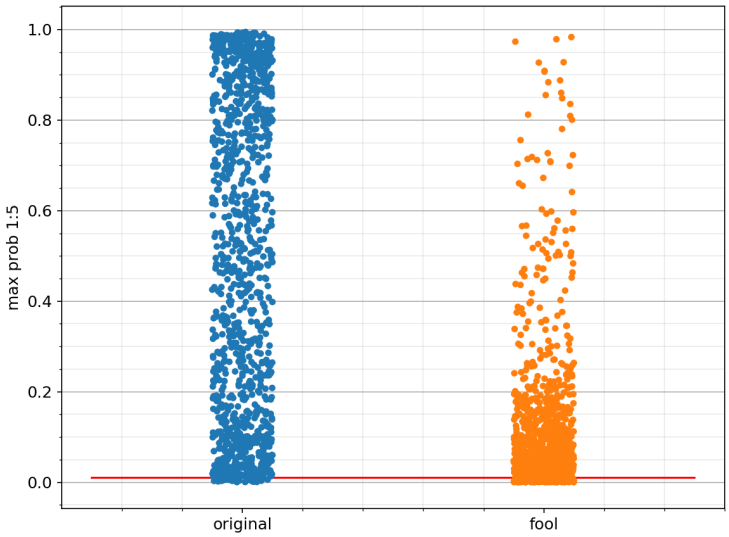


Figure 4.39: Plot of the maximum probability that the input image belongs to the first 5 classes. "Fool" refers to the elephant dataset. The red line indicates the threshold for which roughly 10% of the original dataset is rejected.

Then, we tested the elephant dataset. The resulting distributions are shown in Fig. 4.39. For a cost of 10.5% of the original data, the algorithm rejected 18.3% of the elephant images.

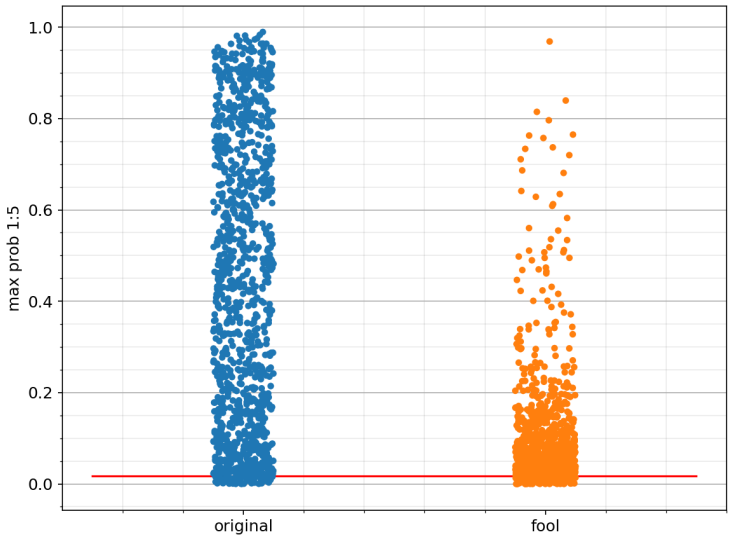


Figure 4.40: Plot of the maximum probability that the input image belongs to the first 5 classes. "Fool" refers to the noise dataset. The red line indicates the threshold for which roughly 10% of the original dataset is rejected.

Finally, we tested the noise set. The result can be seen in Fig. 4.40. While rejecting 10.8% of the original dataset, the algorithm was able to reject 16.1% of the noise dataset.

4.8.4 Summary

dataset	method	orig. reject	fool. reject
mirrored	probability thr.	10.3	68.3
mirrored	MAV distance	10.1	68.3
mirrored	OpenMax	11.1	19.1
elephant	probability thr.	10.3	71.9
elephant	MAV distance	10.3	79.7
elephant	OpenMax	10.5	18.3
noise	probability thr.	10.0	65.3
noise	MAV distance	10.8	51.9
noise	OpenMax	10.8	16.1

Table 4.5: Summary of results for different methods of rejecting unknown inputs. All numbers are percentages.

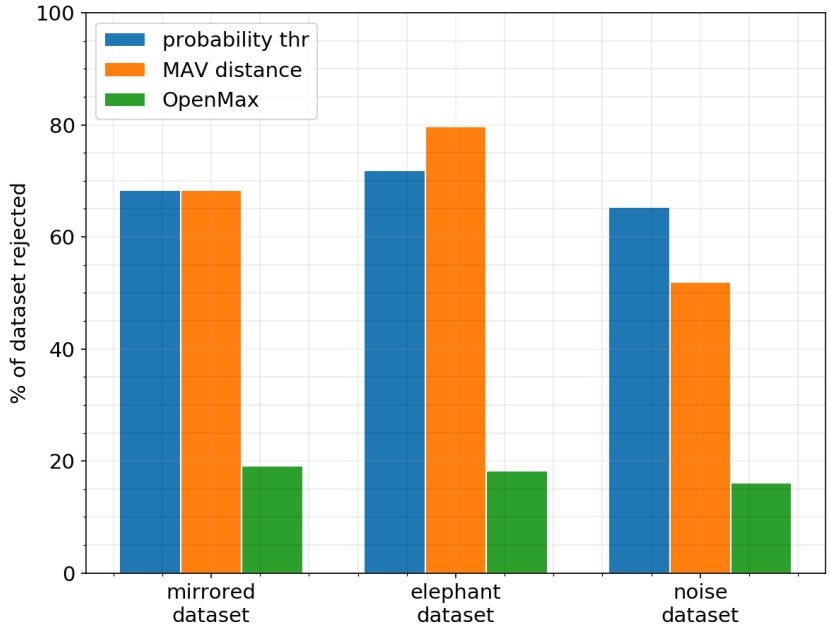


Figure 4.41: Graphical representation of the results from Table 4.5.

The results of this section can be seen in Table 4.5 and Fig. 4.41. On the elephant dataset, the best performing algorithm is the one based on distance to MAV. Clearly, the additional information gained from using the entire vector of outputs rather than just the maximum helps with rejecting unknown inputs there. However on the mirrored set it performs no better than probability thresholding, and on the noise set it performs significantly worse.

OpenMax gave us poor results on all sets. This is in line with the expectations we had based on the fact that we have very small activation vectors.

4.9 Time and computational speed

Because we used a relatively small neural network on a PC with a GPU, elapsed times were short. Training the network for 12 epochs with batch normalization on an 800 image training set took 5 seconds. Passing a 1200 image dataset through an already trained network took 0.4 second. Since the PC we used was not high end, this time could be reduced considerably by using a powerful modern system. This means that computational time should not be a limiting factor for using the system in a production environment.

4.10 Network architecture changes

In this section, we look at the effect of changing the amount of layers in the network.

4.10.1 Removing layers

Our network performs exceptionally well given that it has only 5 layers. Most likely, this is due to our relatively simple data. A scatterometry scan with a positive peak followed by a negative is orders of magnitude simpler than, say, an image of a bicycle. This leads to the question: can we make our network even smaller? To put this to the test, we changed the network architecture by removing one or more convolutional and/or linear layers. In order to obtain statistics, 50 runs were done for each architecture. All runs were done with batch normalization and 12 epochs of training. For the network code, see Appendix F.

network layers	mean	max	std
2 conv 3 lin (base)	93.2	97.0	2.3
2 conv 2 lin	92.2	95.5	2.4
2 conv 1 lin	90.5	95.0	2.5
1 conv 3 lin	91.6	96.5	2.5
1 conv 2 lin	90.7	95.5	2.2
0 conv 3 lin	86.3	90.5	2.3

Table 4.6: Accuracies in % for network architectures with fewer layers.

The result can be seen in Table 4.6. We can clearly see that removing layers reduces accuracy. Also, removing a convolutional layer has a bigger impact than removing a linear layer. This is in line with expectations, given the two dimensional nature of our data.

4.10.2 Adding convolutional layers

# conv layers	channels	mean	max	std
2 (base)	8	93.2	97.0	2.3
3	8	93.2	97.5	2.0
3	10	93.1	96.5	2.2
4	8	93.6	98.0	2.1
4	10	93.8	97.5	1.8
5	10	93.7	97.0	2.4
5	12	93.2	97.5	2.3
6	10	92.2	96.5	2.2

Table 4.7: Accuracies in % for network architectures with extra convolutional layers.

To test whether 2 convolutional layers is the optimal amount, we also added convolutional layers. Once again, we used batch normalization, 12 epochs of training and 50 runs per architecture. This time, we did not vary the amount of linear layers. The result can be seen in Table 4.7. Clearly, there is a benefit to adding layers: both the highest mean and the highest maximum accuracy were achieved at 4 layers. Adding a 5th layer does not grant a further increase. This is probably due to the fact that we have a pooling layer with a step size of 2 after each convolutional layer, meaning the data

gets reduced in size by a factor of 2 in each dimension. Because of this, after 5 layers, our feature maps are of size 3x3.

After 6 layers, each feature map is a single number. Surprisingly, this does not have a big negative impact on accuracy. Since we have 10 channels, the data comes out of the convolutional layers as only 10 numbers per image. Apparently, that is still enough for accurate classification.

4.10.3 Summary

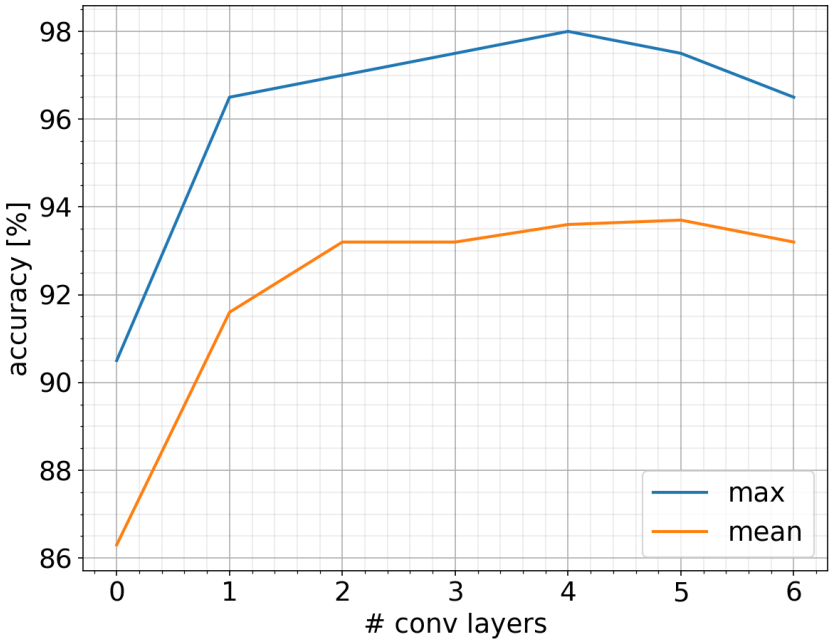


Figure 4.42: Test accuracy as a function of the amount of convolutional layers in the network architecture.

The results of our convolutional layer tests can be seen in Fig. 4.42. We conclude that the optimal amount of convolutional layers is 4. Given the mean and standard deviation, it is likely that 5 layers would achieve the same maximum accuracy as 4 layers if we performed enough runs. However, it is unlikely that it would achieve a higher accuracy.

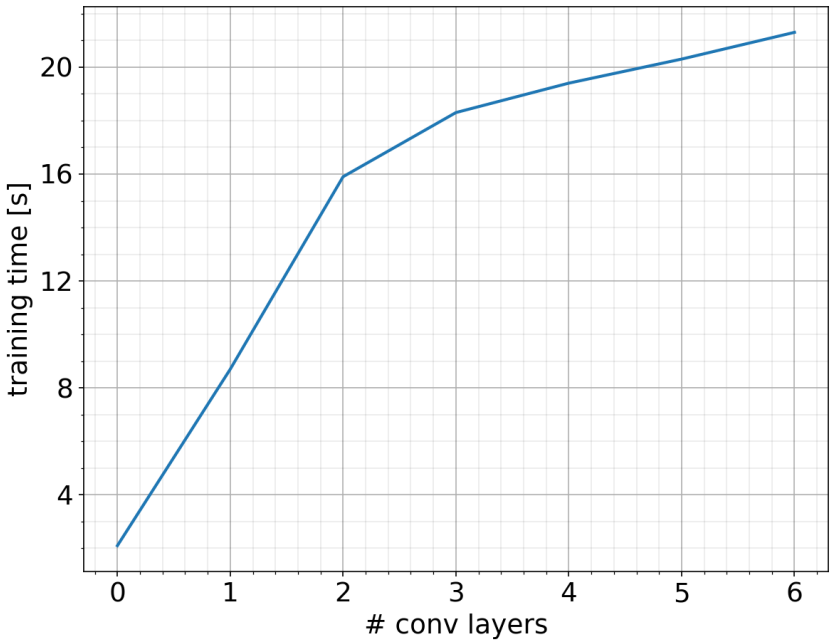


Figure 4.43: Training time as a function of the amount of convolutional layers in the network architecture. Training time includes computational time from all layers layers, including the linear layers.

Fig. 4.43 shows us how the total training time varies with the amount of convolutional layers. At the time that we tested this, we did not have access to a CUDA compatible GPU, so the tests were performed using only a CPU, of the type "AMD Ryzen 7 3800X". This is why the times are higher than the ones described in Section 4.9.

It is interesting to see that after 2 convolutional layers, training time increases less fast than before. Because of this, going from 2 to 4 layers results in only a 22% increase in training time. However, we cannot be sure that these numbers are the same when using a GPU.

4.11 Comparison between deep learning and clustering based classification

In this last section, we look at a comparison of the deep learning approach to a method based on unsupervised clustering that has been recently developed my my supervisor Dmytro Kolenov. The method works as follows:

1. Search for the type of signals that represent particles (positive-negative pulses)

2. Customize unsupervised clustering algorithms to define the group of signals that are attributed to a single scatterer
3. Use a calibration curve (acquired during the calibration with many samples) to return a class label for the particle

Class	**Size**
Background	247
40 nm	205
50 nm	217
60 nm	276
80 nm	272
Total	1217

Table 4.8: Comparison dataset. Images were cut from raw data as 150x450 and downsampled to 150x150.

To test this, we used a dataset that was different from the ones used previously, though it was created from the same raw data. It is a 5 class dataset with 150x150 pixel images. The distribution of images over the classes can be seen in Table 4.8.

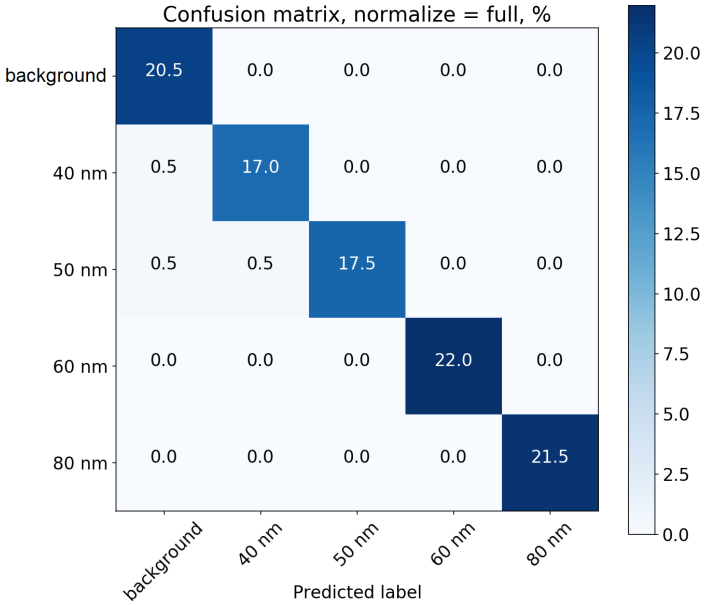


Figure 4.44: Normalized confusion matrix for a high accuracy network run with the dataset from Table 4.8.

First, we used it to train a neural network with 2 convolutional layers and batch normalization. After 12 epochs, it yielded a test accuracy of 98.5%. The confusion matrix can be seen in Fig. 4.44.

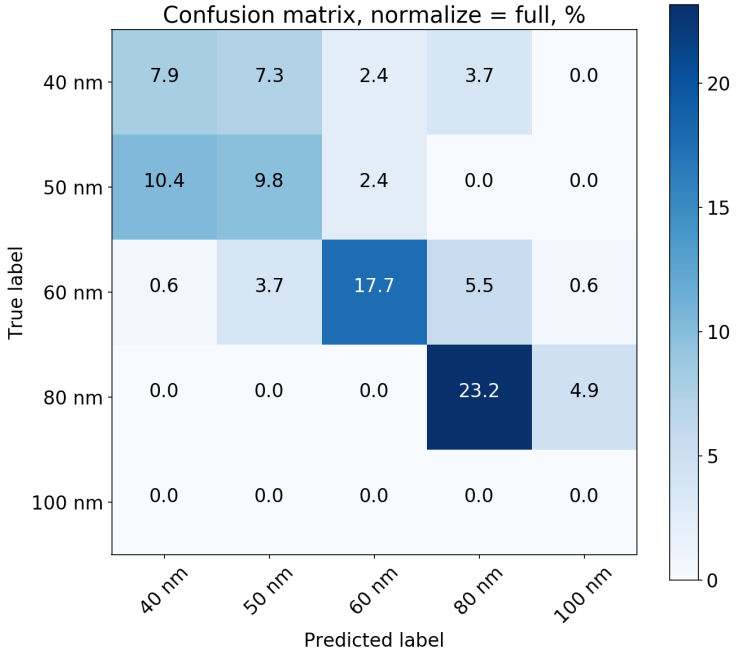


Figure 4.45: Normalized confusion matrix for clustering based classification with the dataset from Table 4.8.

Then, we used the clustering method to classify the same test set as the network. We limited the comparison to the classification of particles, so we excluded backgrounds from the clustering test. However, the clustering method supported the classification of 100 nm particles. That's why we still have a confusion matrix of size 5x5 (see Fig. 4.45).

Clearly, the clustering method performed a lot worse. Overall accuracy was 58.5%, with relatively accurate classification on the 60 and 80 nm classes and a lot of confusion on the 40 and 50 nm classes. Based on this, we conclude that a deep learning based approach is more suitable for accurate particle classification.

4.12 Conclusion

This concludes the results of our experiments. We will further discuss them in the next chapter.

Chapter 5

Discussion

5.1 Results

Overall, the network performance was better than we expected. Accuracies were very high and the network required little training. Most likely, this is due to our data being relatively simple to interpret for a CNN. A positive-negative peak from a particle detection has far fewer features than most objects found in real world images. This allowed us to keep our network small, which means it is fast, and a fast network is both convenient for experimentation and useful for applications that require the detection of particles on small timescales.

5.2 Future improvements

5.2.1 Dataset

One possible way to improve the results is to improve the quality of the dataset. Particles of sizes 60 nm and 80 nm were very clearly visible on the scans, and there was no doubt about how the cuts should be made that are used as network inputs. For sizes 40 nm and 50 nm, it was a lot less clear. Especially for 40 nm, the process of selecting particles could often be best described as "educated guess". And for the background class, we cannot guarantee that small particles are not present in it, though we did our best to avoid it. We do not see an immediate solution for these problems, but if one were to be found, it would surely benefit accuracy. Given that 98% was already achieved, it is not unthinkable that the 5 class accuracy would then go to 100%.

Another point of improvement is multiple particles. In the datasets used

in this thesis, we did not distinguish between one or more particles being present in an image. Therefore, every classification means "there is at least one particle of this size present in this image". In the future, a new dataset could be made that separates images with single particles from those with multiple. The latter could then be handled in one of two ways: as extra classes (this would require a large amount of multiple particle images for training) or as anomalous data, to be seperated by open-set recognition.

And lastly, detector saturation could be avoided by measuring samples that contain larger particles at $1/x$ times the normal laser power, and then multiplying the resulting data values by x.

5.2.2 Open-set recognition

When batch normalization is used, images within batches are not independent. They get normalized in each layer based on batch statistics, which depend on all the images in the batch. Therefore, passing the real dataset and the fooling datasets seperately, like we did for our open-set recognition tests, is not ideal. In a real world scenario, the unknown images would be present in the regular dataset, and thus the two types of images would influence each other. However, to run a test that accounts for this would require us to make a decision about what portion of the data should be fooling data, and this is a decision that cannot be made without context. It would require some knowledge or intuition about how often anomalous images would appear in real data. That is why for the experiments in this thesis, we decided not to mix the datasets.

To further test open-set recognition, additional datasets could be created. It would be interesting to do additional testing of anomaly detection, to find out whether probability thresholding is consistently better at that than the other methods. One possible way of creating an anomaly dataset would be to make scans of particles with a diameter of 100 nm, to test the network on particles that are larger than the trained classes. Another way would be to artificially increase the size of the background.

5.2.3 Smaller Particles

Based on what we have seen, we do not think the neural network is a limiting factor in detecting particles of 30 nm. The limiting factor is our ability to spot and label such particles in the raw data in order to form a dataset that can be used for training. For this, it would be essential to eliminate the wobble effect on the experimental side, so that the local averaging filter in data processing is no longer needed. However, even then, some additional

progress would need to be made on the particle selection side to make reliable labeling of 30 nm particles possible.

5.3 Expanding the system

In the future, an additional neural network could be used to perform object detection [38]. Then, data would no longer need to be processed manually to cut large scans into small images. The system, consisting of two neural networks, could both detect particles in raw data and classify them. It could then be used in a production environment where wafers need to be checked for contamination.

Chapter 6

Conclusion

In this thesis, we looked at data obtained from scans made with a Coherent Fourier Scatterometry setup for detection of nanoparticles on surfaces. We made cuts around particle detections on these scans and used these cuts as input images for a convolutional neural network. We showed that a network with two convolutional layers and three linear layers is very effective at particle classification, with a test set accuracy of 100% for 4 classes which each represent one particle size, and a test set accuracy of 97% if additionally a background class is present. Also, we showed that changing the network architecture by adding 2 convolutional layers increases the maximum accuracy to 98%.

We compared these numbers to the accuracy of a clustering based classification algorithm, which achieved 58.5% accuracy on our test set, and concluded that deep learning is more suitable for the task of particle classification.

Next to that, we showed that it is, to a certain degree, possible to reject inputs that do not belong to the trained classes. For a cost of rejecting 10% of the real data, our algorithms could reject 80% of an elephant image dataset, 72% of a mirrored particle image dataset or 65% of a dataset with artificially added noise.

We noted that it is hard to label a dataset for 40 nm particles, because it is hard for a human to spot the detections and to distinguish them from background. This will be a limiting factor when attempting to add a 30 nm class to the dataset.

Lastly, we also explored loss landscapes, but concluded that in their current state they do not give us useful information.

This research can lead to future research in the application of object detection to scatterometry data. This is the last step required to make an automated system that detects particles directly from scatterometry data.

Appendices

Appendix A

Testing network parameters on synthesized data

In each of the following figures, a plot with 10 accuracy curves is displayed to show variance, and an average accuracy is displayed to show average speed of convergence. The y-axis shows validation accuracy in % while the x-axis shows the number of epochs. Input size is 28x28 pixels unless specified otherwise. The reason we chose to do 10 runs per setting is that at the moment we performed this experiment, we did not have access to a GPU yet, which made the runs considerably slower.

Code commands for the convolutional layers are included to show the difference between networks. The syntax is conv2d(input, output, kernel, stride, padding). Input is the number of input layers, output is the number of output layers, kernel is the one dimensional size of the square kernel used for performing the convolution, stride is the step size that is used to move the kernel over the image, and padding is the amount of zero layers added to the image on the outside, to prevent the convolutional layer from reducing image size.

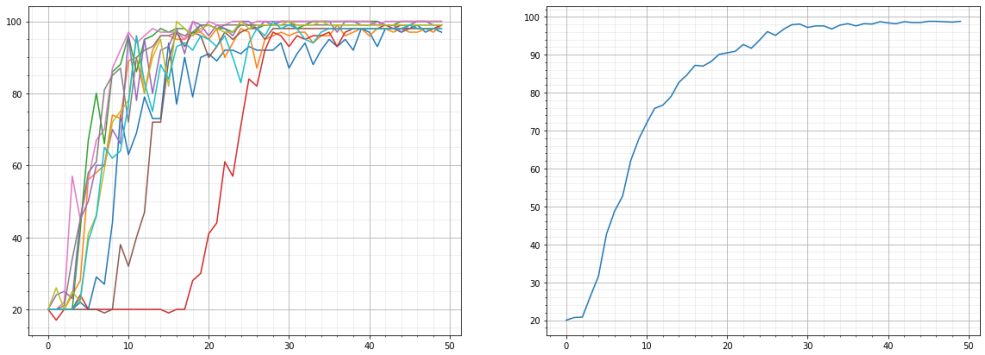


Figure A.1: Base network with conv2d(1,5,5,1,1) and conv2d(5,8,5,1,1)

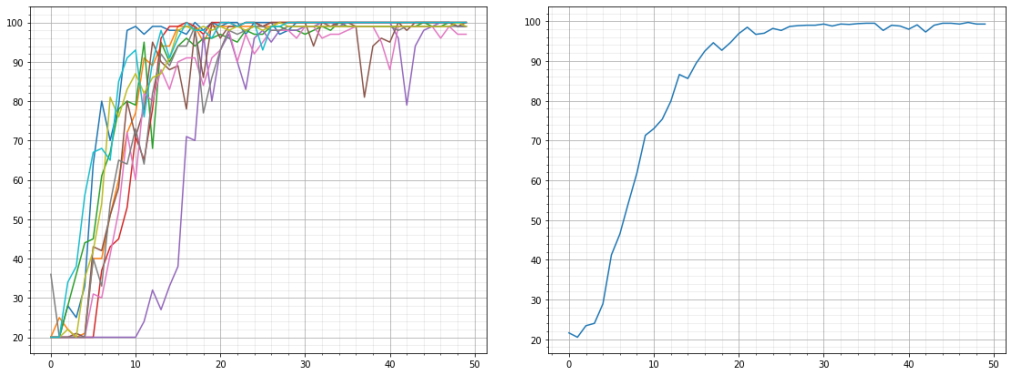


Figure A.2: Network with more channels: conv2d(1,6,5,1,1) and conv2d(6,12,5,1,1)

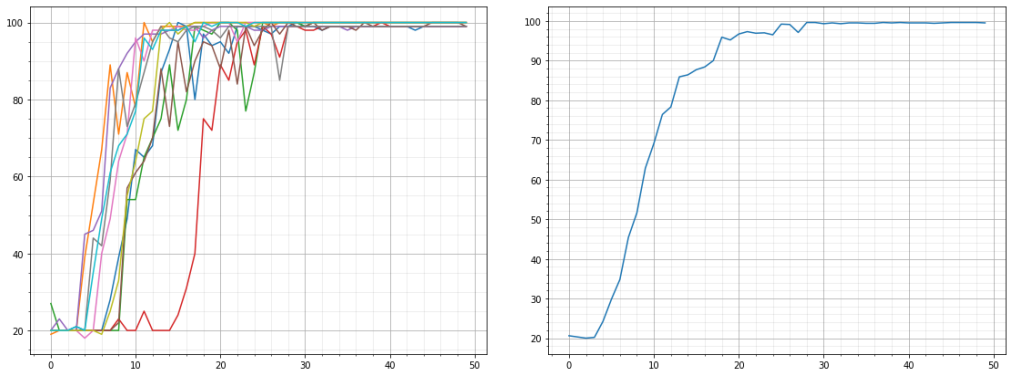


Figure A.3: Network with padding of size 2: conv2d(1,5,5,1,2) and conv2d(5,8,5,1,2)

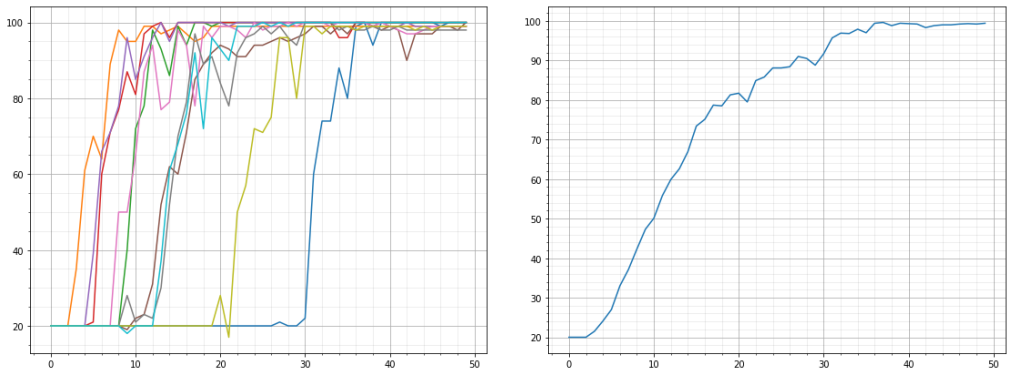


Figure A.4: Network with more channels and padding of size 2: conv2d(1,6,5,1,2) and conv2d(6,12,5,1,2)

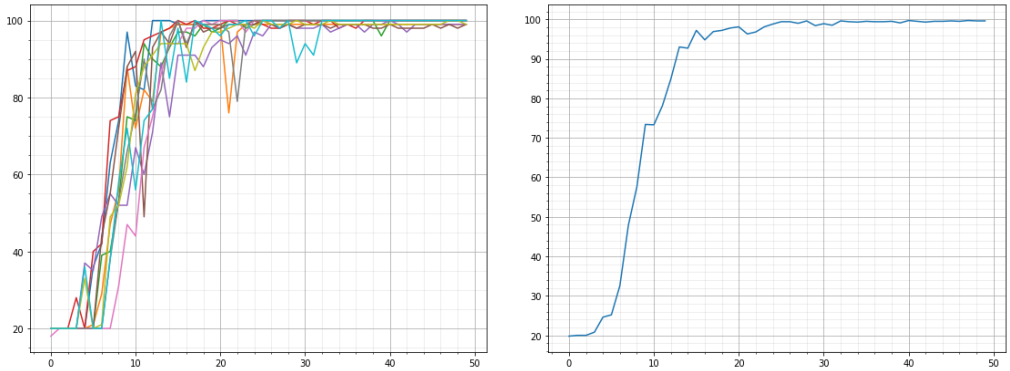


Figure A.5: Network with extra conv layer: conv2d(1,5,5,1,1), conv2d(5,8,5,1,1), conv(8,10,3,1,1)

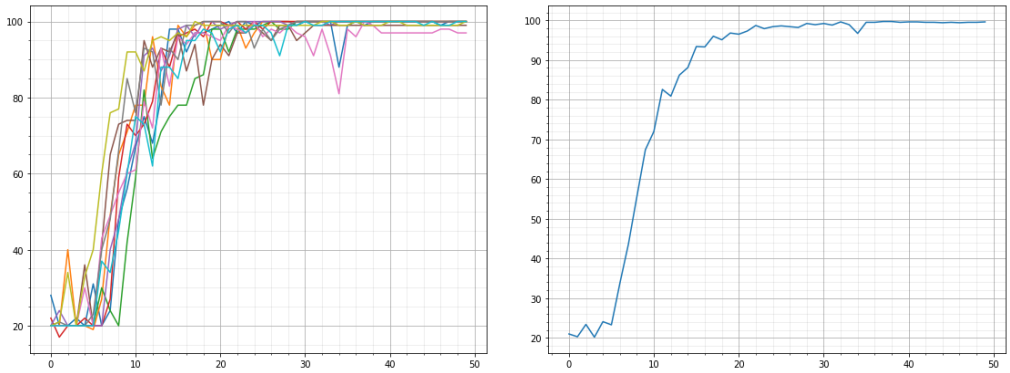


Figure A.6: Network with extra conv layer and more channels: conv2d(1,5,5,1,1), conv2d(5,10,5,1,1), conv(10,15,3,1,1)

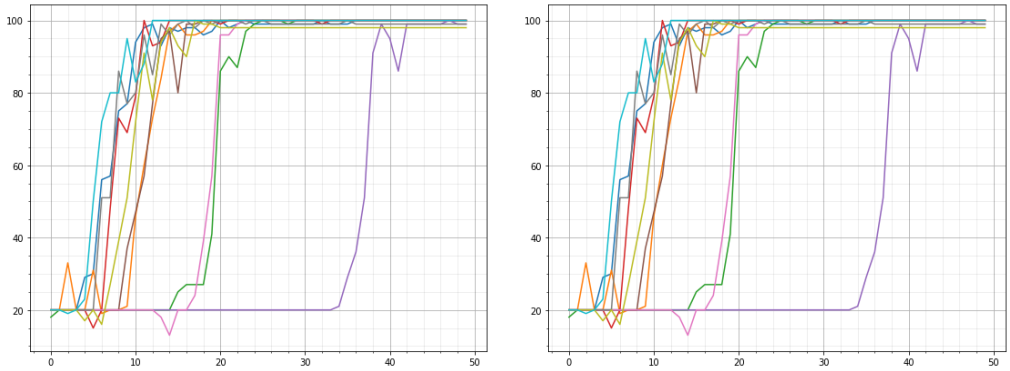


Figure A.7: Network with double padding and input size 40x40: conv2d(1,5,5,1,2), conv2d(5,8,5,1,2)

Appendix B

MATLAB tool

Individually classifying a thousand particles, while at the same time extracting parameters such as width and peak-to-peak voltage, is a lot of work. Because of that, we made a tool that makes the process as easy and fast as it can reasonably be. The tool is a MATLAB app written in the app designer, which is part of MATLAB since 2016. It is available on github, see Ref. [24].

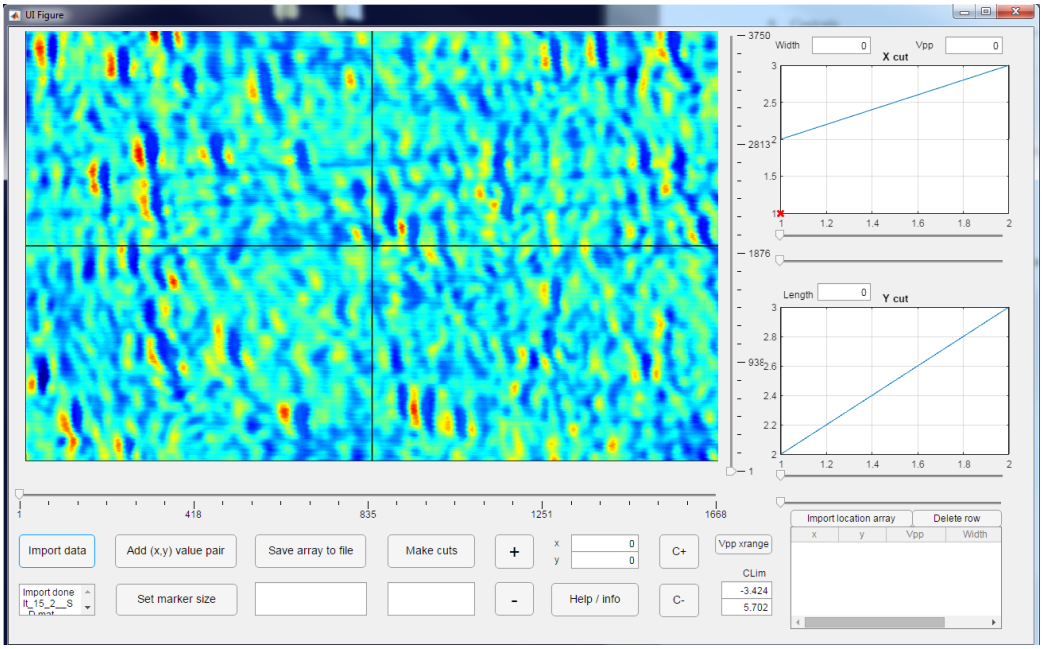


Figure B.1: The tool with a scan of an 80 nm sample loaded. Colors are the default colors assigned by MATLAB, with colormap 'jet'.

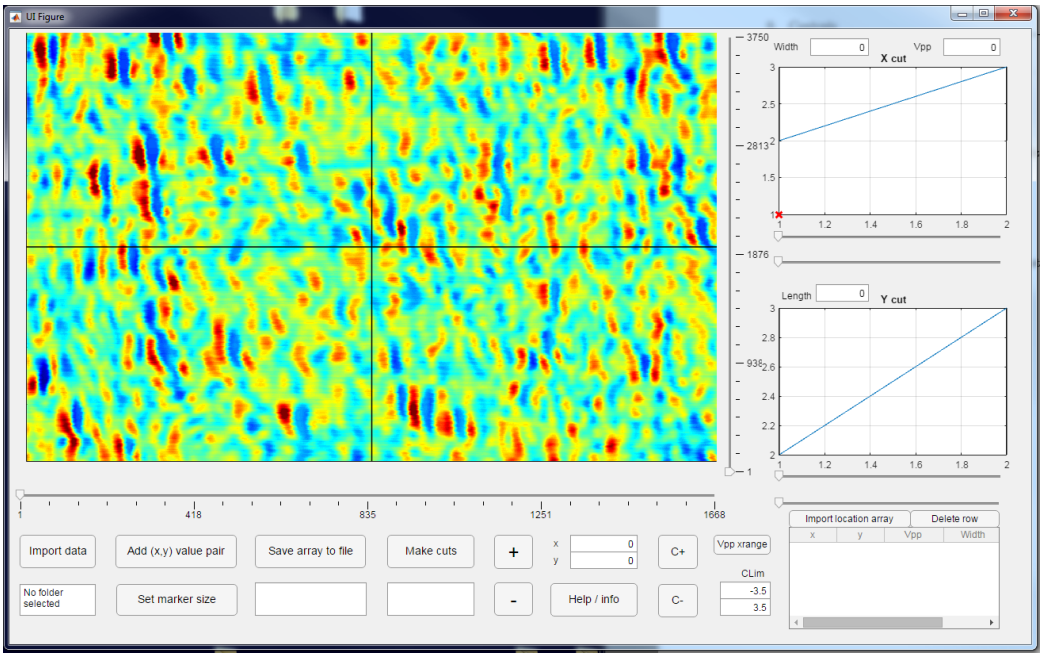


Figure B.2: The same scan, now with manually adjusted colors via the CLim edit fields. This allows us to see differences in intensity much more clearly.

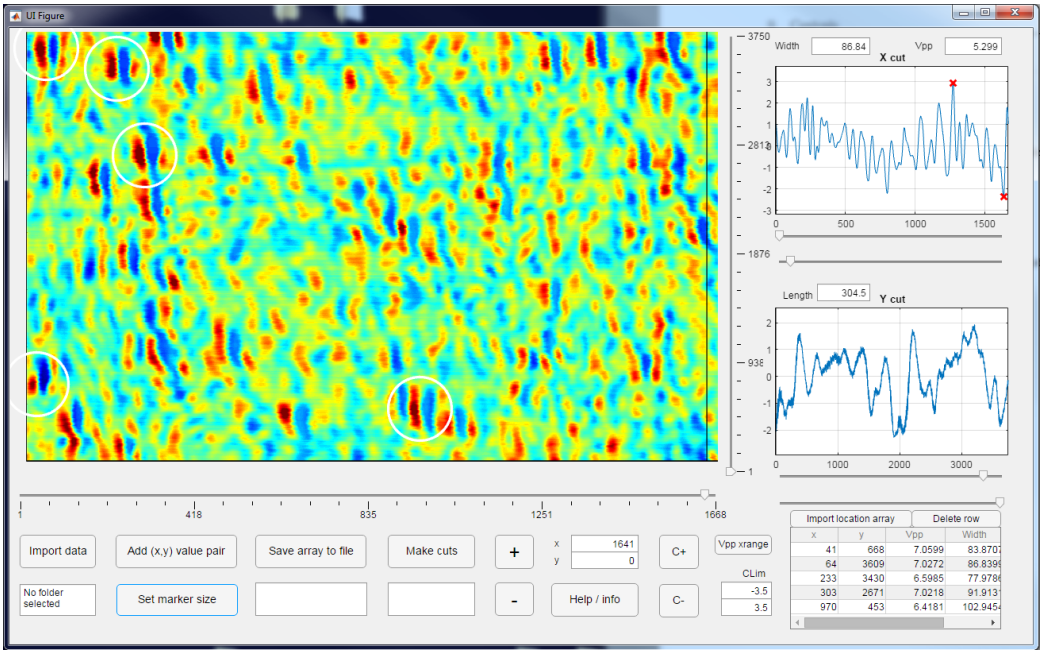


Figure B.3: The same scan with marked particles (white circles). More particles can be seen, but they were excluded intentionally because of their lower V_{pp} (we were selecting for 80 nm particles only)

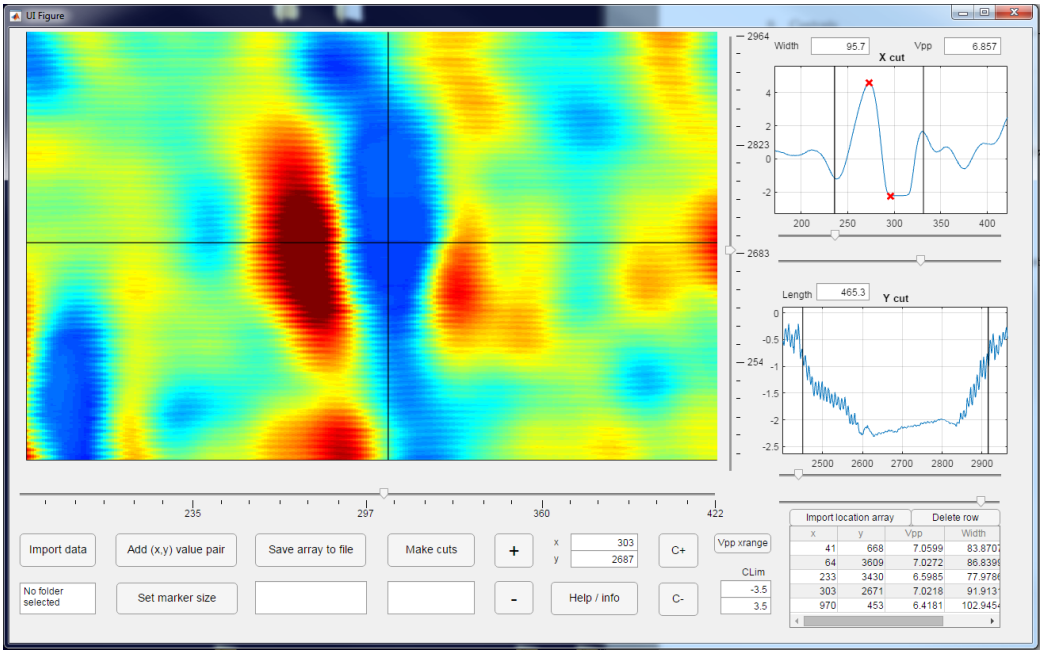


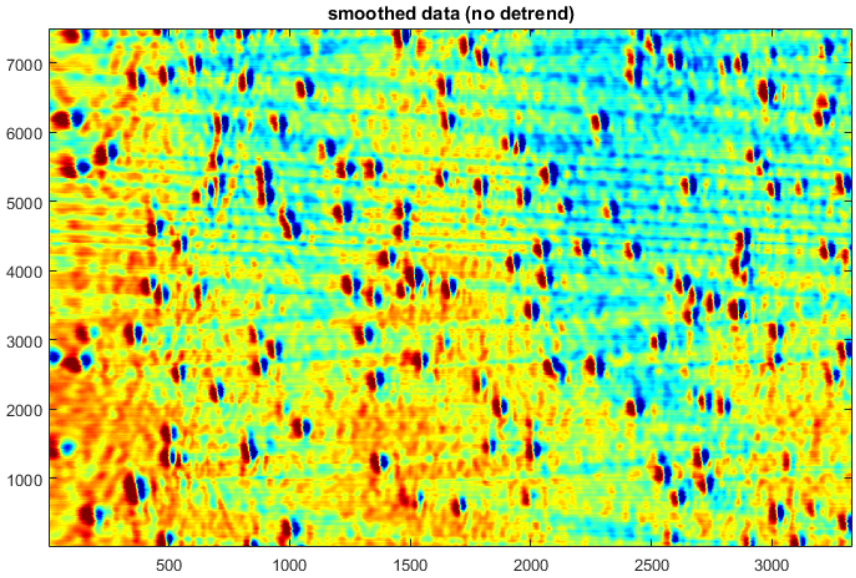
Figure B.4: Zoomed in view of a single detection. The graphs are intensity profiles along the plotted black lines. The "X cut" graph represents the typical signal coming from a particle detection with a split photodetector. In this case, it is cut off at the bottom because the laser power was sufficiently high for 80 nm particles to saturate the detector. The vertical black lines in the graph are controlled by the sliders underneath the graph and used for calculating width (in the "Y cut" graph, they are used for calculating length). V_{pp} is calculated automatically and denoted by the red "x" markers.

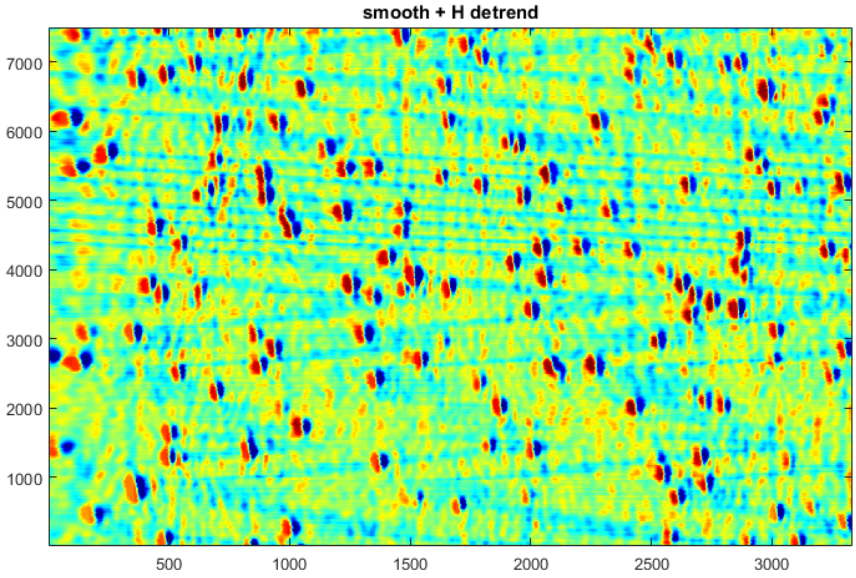
A particle is saved with the "Add (x,y) value pair" button, which adds its statistics to the array in the bottom right, and marks it with a white circle in the zoomed out version of the plot. When all particles are marked this way, the array can be exported with the "Save array to file" button. This allows other scripts to loop over these arrays later and quickly produce images of a desired size that contain particle detections.

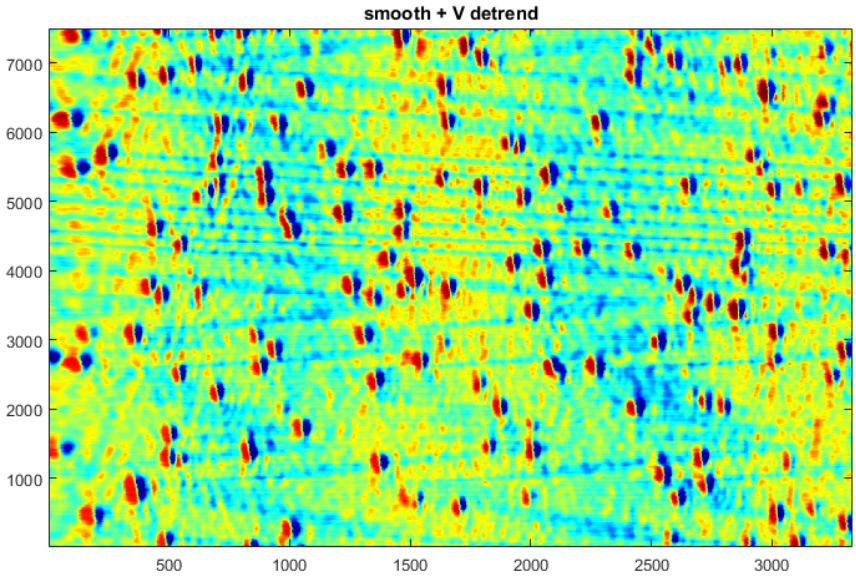
Appendix C

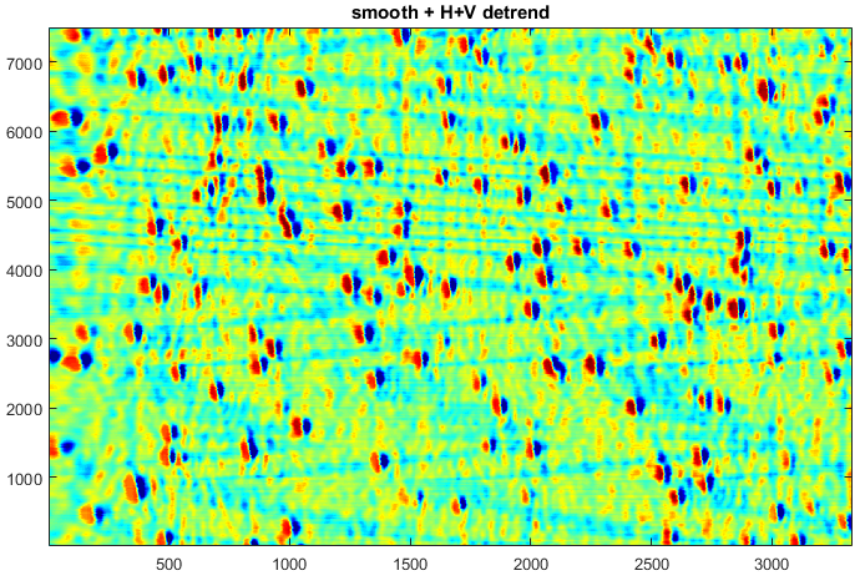
The effect of detrending on scans

To visualize the effect of detrending, we chose a scan with clearly visible trends, and detrended in three ways: horizontally, vertically, and both. The scan was taken from an 80 nm PSL sample. The results can be seen on the next pages. In order to make comparison easy, the color scheme was kept the same for all images.


smoothed data (no detrend)
7000
6000
5000
4000
3000
2000
1000
500
1000
1500
2000
2500
3000


smooth + H detrend
7000
6000
5000
4000
3000
2000
1000
500
1000
1500
2000
2500
3000


smooth + V detrend
7000
6000
5000
4000
3000
2000
1000
500
1000
1500
2000
2500
3000


smooth + H+V detrend
7000
6000
5000
4000
3000
2000
1000
500
1000
1500
2000
2500
3000

Appendix D

Max pooling vs average pooling

	maxpool	**avgpool**
max	97.0	95.0
mean	92.9	90.6
std	2.5	1.9

Table D.1: Comparison between maxpool and avgpool. Numbers are accuracy in %.

To verify that max pooling is the correct choice for our problem, we tested the network with average pooling. We compared 50 runs with average pooling, batch size 50, and 12 epochs to an equal amount of runs with max pooling.

The result is shown in Table D.1. As expected, max pooling performs better on all fronts.

Appendix E

Training trajectories on loss contour plots

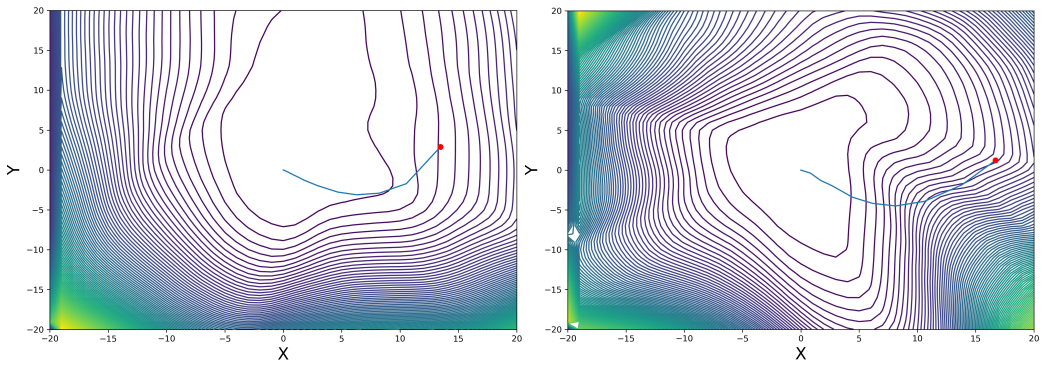


Figure E.1: Loss contour plots with training trajectories. A red dot marks the start of training.

In Ref. [20], training trajectories are projected to the two dimensional space of the loss landscape so they can be overlaid on contour plots. We did the same, resulting in the plots of Fig. E.1. However, we have not been able to find a use for this in our case.

Appendix F

Network code

See the next pages for the PyTorch code of our neural networks. Placement of the BatchNorm layers was chosen to be after ReLU activation layers, as recommended by Chen et al in [30]. The parameter `Size_2` depends on the dimensions of the network inputs.

Code of our basic network after switching to double padding:

```
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()

        self.features = nn.Sequential(
                nn.Conv2d(1,5,5,1,2),
                nn.MaxPool2d(2,2),
                nn.ReLU(inplace=True),
                nn.Conv2d(5,8,5,1,2),
                nn.MaxPool2d(2,2),
                nn.ReLU(inplace=True),
                )

        self.classifier = nn.Sequential(
                nn.Linear(8*Size_2*Size_2, 120),
                nn.ReLU(inplace=True),
                nn.Linear(120,84),
                nn.ReLU(inplace=True),
                nn.Linear(84,5)
                )

    def forward(self, x):
        x = self.features(x)
        x = x.view(-1, 8*Size_2*Size_2)
        x = self.classifier(x)

        return x
```

Code of our network with batch normalization:

```
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()

        self.features = nn.Sequential(
                nn.Conv2d(1,5,5,1,2),
                nn.MaxPool2d(2,2),
                nn.ReLU(inplace=True),
                nn.BatchNorm2d(5),
                nn.Conv2d(5,8,5,1,2),
                nn.MaxPool2d(2,2),
                nn.ReLU(inplace=True),
                nn.BatchNorm2d(8)
                )

        self.classifier = nn.Sequential(
                nn.Linear(8*Size_2*Size_2, 120),
                nn.ReLU(inplace=True),
                nn.BatchNorm1d(120),
                nn.Linear(120,84),
                nn.ReLU(inplace=True),
                nn.BatchNorm1d(84),
                nn.Linear(84,5)
                )

    def forward(self, x):
        x = self.features(x)
        x = x.view(-1, 8*Size_2*Size_2)
        x = self.classifier(x)

        return x
```

Code used for average pooling tests:

```
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()

        self.features = nn.Sequential(
                nn.Conv2d(1,5,5,1,2),
                nn.AvgPool2d(2,2),
                nn.ReLU(inplace=True),
                nn.BatchNorm2d(5),
                nn.Conv2d(5,8,5,1,2),
                nn.AvgPool2d(2,2),
                nn.ReLU(inplace=True),
                nn.BatchNorm2d(8)
                )

        self.classifier = nn.Sequential(
                nn.Linear(8*Size_2*Size_2, 120),
                nn.ReLU(inplace=True),
                nn.BatchNorm1d(120),
                nn.Linear(120,84),
                nn.ReLU(inplace=True),
                nn.BatchNorm1d(84),
                nn.Linear(84,5)
                )

    def forward(self, x):
        x = self.features(x)
        x = x.view(-1, 8*Size_2*Size_2)
        x = self.classifier(x)

        return x
```

Code used for testing a network with 2 convolutional and 2 linear layers:

```
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()

        self.features = nn.Sequential(
                  nn.Conv2d(1,5,5,1,2),
                  nn.MaxPool2d(2,2),
                  nn.ReLU(inplace=True),
                  nn.BatchNorm2d(5),
                  nn.Conv2d(5,8,5,1,2),
                  nn.MaxPool2d(2,2),
                  nn.ReLU(inplace=True),
                  nn.BatchNorm2d(8)
                      )

        self.classifier = nn.Sequential(
            nn.Linear(8*Size_2*Size_2, 120),
            nn.ReLU(inplace=True),
            nn.BatchNorm1d(120),
            nn.Linear(120,5)
            )

    def forward(self, x):
        x = self.features(x)
        x = x.view(-1, 8*Size_2*Size_2)
        x = self.classifier(x)

        return x
```

Code used for testing a network with 2 convolutional and 1 linear layer:

```
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()

        self.features = nn.Sequential(
              nn.Conv2d(1,5,5,1,2),
              nn.MaxPool2d(2,2),
              nn.ReLU(inplace=True),
              nn.BatchNorm2d(5),
              nn.Conv2d(5,8,5,1,2),
              nn.MaxPool2d(2,2),
              nn.ReLU(inplace=True),
              nn.BatchNorm2d(8)
                  )

        self.classifier = nn.Sequential(
            nn.Linear(8*Size_2*Size_2, 5)
            )

    def forward(self, x):
        x = self.features(x)
        x = x.view(-1, 8*Size_2*Size_2)
        x = self.classifier(x)

        return x
```

Code used for testing a network with 1 convolutional and 3 linear layers:

```
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()

        self.features = nn.Sequential(
                  nn.Conv2d(1,5,5,1,2),
                  nn.MaxPool2d(2,2),
                  nn.ReLU(inplace=True),
                  nn.BatchNorm2d(5),
                      )

        self.classifier = nn.Sequential(
            nn.Linear(8*Size_2*Size_2, 120),
            nn.ReLU(inplace=True),
            nn.BatchNorm1d(120),
            nn.Linear(120,84),
            nn.ReLU(inplace=True),
            nn.BatchNorm1d(84),
            nn.Linear(84,5)
            )

    def forward(self, x):
        x = self.features(x)
        x = x.view(-1, 8*Size_2*Size_2)
        x = self.classifier(x)

        return x
```

Code used for testing a network with 1 convolutional and 2 linear layers:

```
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()

        self.features = nn.Sequential(
                  nn.Conv2d(1,5,5,1,2),
                  nn.MaxPool2d(2,2),
                  nn.ReLU(inplace=True),
                  nn.BatchNorm2d(5),
                      )

        self.classifier = nn.Sequential(
            nn.Linear(8*Size_2*Size_2, 100),
            nn.ReLU(inplace=True),
            nn.BatchNorm1d(100),
            nn.Linear(100,5)
            )

    def forward(self, x):
        x = self.features(x)
        x = x.view(-1, 8*Size_2*Size_2)
        x = self.classifier(x)

        return x
```

Bibliography

[1] "Embracing the organics world" *Nature Materials* **12**, 2013

[2] H. Hoppe et al., "Organic solar cells: An overview" *Journal of Materials Research*, Vol **19**, Issue **7**, Jul 2004

[3] M. Lapedus, "Inspecting Unpatterned Wafers", *Semiconductor Engineering*, August 2018

[4] S. Roy et al., "High speed low power optical detection of sub- wavelength scatterer", *Review of Scientific Instruments* **86**, 123111, 2015

[5] Y. LeCun et al., "Deep learning", *Nature* **521**, 2015

[6] R. Parloff, "From 2016: Why Deep Learning Is Suddenly Changing Your Life", *Fortune*, October 2016 issue

[7] A. Krizhevsky et al., "ImageNet Classification with Deep Convolutional Neural Networks", *NIPS 2012*

[8] D. Cireşan et al., "Multi-column Deep Neural Networks for Image Classification", *2012 IEEE Conference on Computer Vision and Pattern Recognition*

[9] M.D. Hannel et al., "Machine-learning techniques for fast and accurate feature localization in holograms of colloidal particles", *Optics Express* 15221, Vol. **26**, No. **12**, Jun 2018

[10] C.L. Chen et al., "Deep Learning in Label-free Cell Classification, *Scientific Reports* **6**, 21471, 2016

[11] D. Cunefare et al., "Deep learning based detection of cone photoreceptors with multimodal adaptive optics scanning light ophthalmoscope images of achromatopsia", *Biomedical Optics Express* 3740, Vol. **9**, No. **8**, Aug 2018

[12] A.J. Cox et al., "An experiment to measure Mie and Rayleigh total scattering cross sections", Am. J. Phys.**70** (6), Jun 2002

[13] I. Goodfellow et al., Deep Learning, An MIT Press book, 2016

[14] M. Nielsen, "Neural Networks and Deep Learning", free online book, Chapter 2, released Dec 2019, retrieved 10-3-2020

[15] D. Rumelhart et al., "Learning representations by back-propagating errors", *Nature*, vol. **323 9**, Oct 1986

[16] A.R. Zamir et al., "Feedback Networks", *CVPR 2017*

[17] K. Jarrett et al., "What is the best multi-stage architecture for object recognition?", *2009 IEEE 12th International Conference on Computer Vision, Kyoto*, pages 2146-2153, 2009

[18] X. Glorot et al., "Deep Sparse Rectifier Neural Networks", *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, PMLR 15:315-323, 2011

[19] "What is the Difference Between a Batch and an Epoch in a Neural Network?", `https://machinelearningmastery.com/difference-between-a-batch-and-an-epoch/`, retrieved 21-2-2020

[20] H. Li et al., "Visualizing the Loss Landscape of Neural Nets, 32nd Conference on Neural Information Processing Systems, Montreal, Canada, 2018

[21] N. Kumar, "Coherent Fourier Scatterometry", PhD thesis, TU Delft, 2014

[22] S. Roy et al., "Coherent Fourier Scatterometry for detection of nanometer-sized particles on a planar substrate surface" *OSA*, Vol. **22**, No. **11**, 2014

[23] D. Kolenov et al., "Heterodyne Detection System for Nanoparticle Detection using Coherent Fourier Scatterometry", *SPIE proceedings*, Vol. **11056**, 2019

[24] GitHub repository for this project, `https://github.com/ddavidse/ConvNetMEP`, [NOT PUBLIC YET]

[25] Y. Lecun et al., "Gradient-Based Learning Applied to Document Recognition", Proceedings of the IEEE, Vol. **86**, Issue **11**, Nov 1998

[26] D.P. Kingma, J.L. Ba, "Adam: A Method for Stochastic Optimization", 3rd International Conferencer for Learning Representations, San Diego, 2015

[27] I.J. Jolliffe, Jorge Cadima, "Principle component analysis: a review and recent developments", *Philosophical Transactions of the Royal Society A*, Vol. **374**, Issue **2065**, Apr 2016

[28] "Reproducibility", pytorch.org, retrieved 23-1-2020

[29] S. Ioffe, C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducting Internal Covariate Shift", Proceedings of the 32^{nd} International Conference on Machine Learning, Lille, France, 2015

[30] G. Chen et al., "Rethinking the Usage of Batch Normalization and Dropout in the Training of Deep Neural Networks", arXiv:1905.05928v1 [cs.LG], May 2019

[31] M. De Bernardi, `https://pypi.org/project/loss-landscapes/`

[32] M. Landgren, L. Tranheden, Input Verification for Deep Neural Networks, Master's thesis in Electrical Engineering, Chalmers University of Technology, Sweden, 2018

[33] A. Bendale, T. Boult, "Towards Open Set Deep Networks", arXiv:1511.06233 [cs.CV], Nov 2015

[34] W.J. Scheirer et al., "Meta-Recognition: the Theory and Practice of Recognition Score Analysis, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. **33**, Issue **8**, Aug 2011

[35] E.J. Gumbel, "Statistical Theory of Extreme Values and Some Practical Applications. A Series of Lectures.", U.S. Government Printing Office, 1954

[36] A. Bendale, "Open World Recognition", PhD thesis, Department of Computer Science, University of Colorado, 2015

[37] C. Alessio, Animals-10 dataset, retrieved 26-2-2020

[38] J. Rieke, "Object detection with neural networks — a simple tutorial using keras", online article, retrieved 12-3-2020