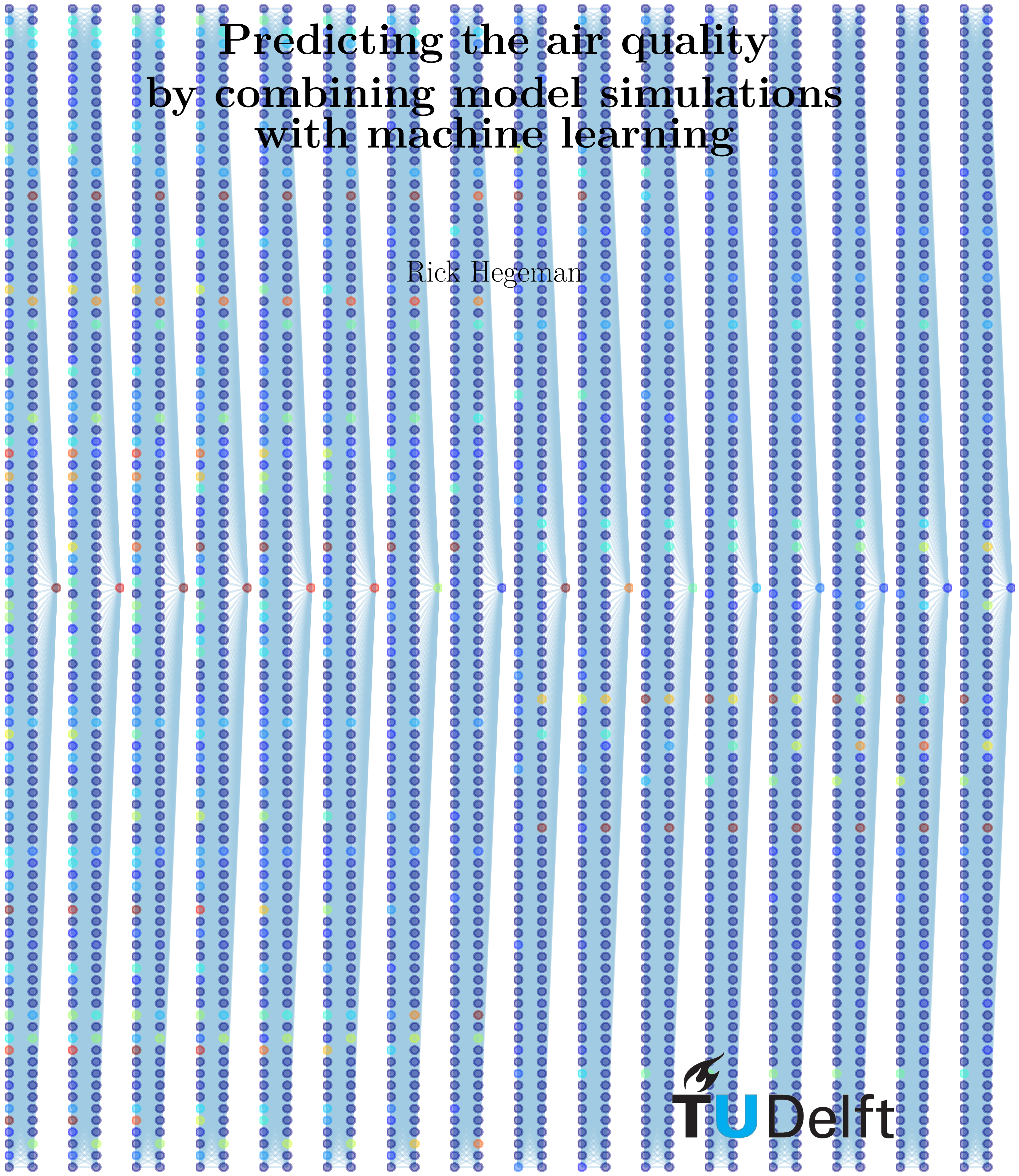

Predicting the air quality by combining model simulations with machine learning

Rick Hegeman

Predicting the air quality by combining model simulations with machine learning

by

Rick Hegeman

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Wednesday January 22, 2020 at 11:00

Student number:	4393058	
Project duration:	February 1, 2019 – January 22, 2020	
Thesis committee:	Prof. dr. ir. H. X. Lin,	TU Delft, supervisor
	Prof. dr. ir. A. W. Heemink,	TU Delft, supervisor
	Dr. ir. M. B. van Gijzen,	TU Delft

Abstract

Combating air pollution has proven to be a difficult task for countries with rapidly developing economies. Poor air quality can be hazardous to people doing any outdoor activities. So being able to make accurate, short term air quality predictions can be very useful. However, making these predictions has proven to be quite difficult, since there are a lot of different physical and chemical processes involved in the emission and transport of the various aerosols that contribute to air pollution. So instead of the more traditional Chemical Transport Models (CTMs) we will be using neural networks in order to make predictions of one of these aerosols, $PM_{2.5}$. In particular, we will be using a Long Short Term Memory (LSTM) network. In addition, we will include the simulations results from a CTM, LOTOS-EUROS, as input data to the LSTM network to improve the performance of the neural network. One of the main drawbacks of the LSTM approach is that whenever the $PM_{2.5}$ concentration changes a lot, the predictions made by the LSTM network take some time to change as well, causing a visible time delay when looking at the measurements and predictions in the same time series plot. We will also try a simpler type of neural network, a Feedforward Neural Network (FNN) and compare its performance to that of LSTM. We found that using the simulation data does indeed improve the LSTM network. Not only in terms of the loss function used by the neural network and, but in particular in the amount gross overestimations by the network, which we use to quantify the LSTM time delay problem. We also found that FNN outperforms the LSTM approach, in particular on samples of high $PM_{2.5}$ concentrations, which we argue is primarily caused by a low amount of samples in our dataset.

Acknowledgements

Before we begin, I would like to thank some people who have been a tremendous help to me during my work.

First of all, I would like to thank my supervisors, Hai Xiang and Arnold, who have provided a lot of useful advice and suggestions, both during the actual research and while I was writing. Their insights helped me find my footing in the new and sometimes confusing world of machine learning which eventually lead to this thesis.

Furthermore, I would like to thank Jianbing, who during my research was also my supervisor and provided me with all the data which I use for the experiments in this thesis. He also helped me understand what I was really trying to accomplish and he discussed the results with me along with Hai Xiang and Arnold. My research would not have worked out without your guidance.

Also, I would like to thank Martin van Gijzen, for taking the time to be a member of my evaluation committee.

I also want to thank Kees Lemmens, who is the admin of the computer cluster which I used for my experiments, for his help and patience throughout.

Next, I would like to thank Carlos, Marco, Massimo, Mike, Pieter and Roel, for their friendship throughout our time at TU Delft. While working on my research project it was helpful that I could complain to you about my troubles with Tensorflow/machine learning/computers in general and beating you at playing cards during lunch break was very relaxing as well. Also I would like to thank Marco and Massimo in particular for proofreading my thesis.

Finally, my family who, among many, many other things, taught me how to count. Truly the first and most important step for any mathematician. Thank you.

Contents

1 Introduction **1**

2 Deep learning **5**
2.1 The Feedforward Neural Network 6
2.2 The Recurrent Neural Network 7
2.3 Long short-term memory 8
2.4 The Training Process 9
2.4.1 Batch gradient descent 10
2.4.2 Adam 10
2.5 Dropout 11
2.5.1 Dropout for FNN 12
2.5.2 Dropout for LSTM 13
2.6 Activation function 14
2.6.1 Tanh 14
2.6.2 ReLU 15
2.7 Cost functions 16

3 The input **17**
3.1 Measuring stations 17
3.2 Correlation 18
3.3 The input matrix 20
3.4 The inclusion of simulation data 20

4 Preprocessing **23**
4.1 Availability of the data 23
4.2 Interpolation techniques 24
4.3 Normalization 25

5 Results **27**
5.1 Baseline example 28
5.2 Different loss functions 32
5.3 Different activation functions 35
5.4 Different learning rates 37
5.5 Including the simulation data 39
5.6 LSTM vs FNN 42

5.7 Large scale experiments 44
5.7.1 LSTM 45
5.7.2 FNN 47

6 Conclusions **49**
6.1 Recommendations and future work 49

Bibliography **53**

A Additional results **55**
A.1 LSTM scatter plots 55
A.2 FNN scatter plots 62
A.3 RMS losses 69

Chapter 1

Introduction

All over the world, poor air quality has been a big problem in recent years. Especially countries with rapidly developing economies, such as China, have had to go to great lengths to lower the concentrations of various air pollutants to a healthy average [Wang and Hao, 2012]. One of those types air pollutants is particulate matter, or PM. These are very fine particles suspended in the air. Particulate matter concentrations are often separated into two categories, distinguished by particle size. PM_{10} consists of all particles with a diameter of 10 micrometers or smaller, while all particles that are smaller than 2.5 micrometers across are considered for the $PM_{2.5}$ concentration. $PM_{2.5}$ is particularly threatening, as the particles are small enough to get past the hairs in the nasal cavity and reach the lungs, where they can remain stuck and cause various problems to the human body. These problems include asthma, lung inflammation, cancer and even death [Yu-Fei Xing and Lian, 2016]. As such, it is important to limit exposure to high concentrations of $PM_{2.5}$, especially for people that are vulnerable to respiratory problems. To this end accurate predictions of pollutants such as the $PM_{2.5}$ concentrations are quite useful. After all precautions such as limiting the amount of strenuous activity when the $PM_{2.5}$ concentration is high may require some planning beforehand. Unfortunately air quality predictions are difficult as there are many sources of the various important pollutants. Emission sources range from sea salt, wildfires, emissions from vegetation and emissions for the soil just to name a few [Manders et al., 2017]. There are also many other geological, meteorological and chemical factors that influence $PM_{2.5}$. As a result a model that aims to predict $PM_{2.5}$ concentrations tends to be quite complicated.

One such model is LOTOS-EUROS. This model is a Chemical Transport Model (CTM) which uses the convection–diffusion equation in three dimensions. In full, the main equation, as given by [Manders-Groot et al., 2016], looks as follows:

$$\begin{aligned}\frac{\partial C}{\partial t} + U\frac{\partial C}{\partial x} + V\frac{\partial C}{\partial y} + W\frac{\partial C}{\partial z} =& \frac{\partial}{\partial x}\left(K_h\frac{\partial C}{\partial x}\right) + \frac{\partial}{\partial y}\left(K_h\frac{\partial C}{\partial y}\right) + \frac{\partial}{\partial z}\left(K_z\frac{\partial C}{\partial z}\right) \\ &+ E + R + Q - D - W\end{aligned} \tag{1.1}$$

Where C is the concentration of various different aerosols. Furthermore, we can recognize wind velocities U, V and W as well as the diffusion coefficients K_h and K_z. The terms E, R, Q, D and W together make up the source term in the convection–diffusion equation. The model uses operator splitting to solve this equation numerically, meaning that within the same time step the model does performs separate calculations for each right hand side term of Equation 1.1. With the equation, the model is able to make predictions of, among other things, $PM_{2.5}$ concentrations. In particular, we will be using LOTOS-EUROS to make predictions in China, in an area around Beijing where there are a lot of available measurements.

In the horizontal direction, the model uses a rectangular longitude-latitude grid, with in our case a grid size of $0.25° \times 0.25°$, which in the area around Beijing corresponds to approximately $28km \times 21km$. Vertically, the model uses four layers. First a surface layer of $25m$ which the model can use for accurately describing surface processes and then three layers on top of the surface layer which together reach up to $3.5km$. These three layers can change height, depending on the meteorological data the model uses [Manders-Groot et al., 2016].

Unfortunately, research [Timmermans et al., 2017] has shown that the calculated results do not always match up with the corresponding measurements. An example of this can be seen in the plot below:

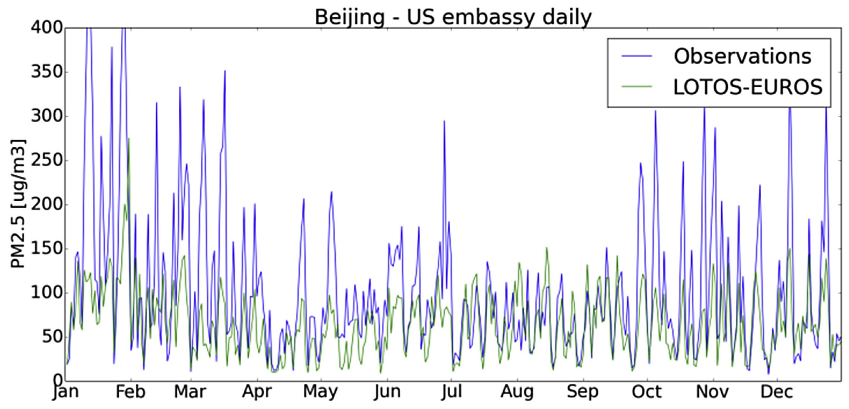

Figure 1.1: Plot comparing $PM_2.5$ measurements to LOTOS-EUROS results for 2013. Source [Timmermans et al., 2017].

This appears to have mainly been caused by the difficulty of accurately determining the parameters involved with the emission sources. In order to bridge this gap between model and measurements, numerical techniques have been developed. These are referred to as data assimilation techniques. One such technique is the ensemble Kalman filter. This technique considers the parameters of the model to have some uncertainty. From that it follows that the model results are uncertain as well. Measurement equipment also always has some amount of uncertainty. By combining these two uncertainties, the filter can find a middle road between model output and measurements depending on the size of these uncertainties. The ensemble Kalman filter can use this to, for example, get a better estimate for initial conditions used by the model, or it can be used to better estimate some of the parameters in the model. The ensemble Kalman filter has been applied before to LOTOS-EUROS, where it was found that data assimilation does indeed improve the model, although the implementation of the algorithm seems to be quite complex [Barbu et al., 2009].

The physical models that predict future air quality do not seem to perform as well as one would hope, but we might find some more improvements by investigating alternative methods. In recent years, deep learning has been used with surprising success on a variety of tasks, such as image recognition and machine translation [Schmidhuber, 2015]. Deep learning involves a neural network, which is a system inspired by the structure of a brain. In a neural network a layered series of nodes, or neurons, are connected to all the nodes of a previous layer via synapses, as shown in the example below:

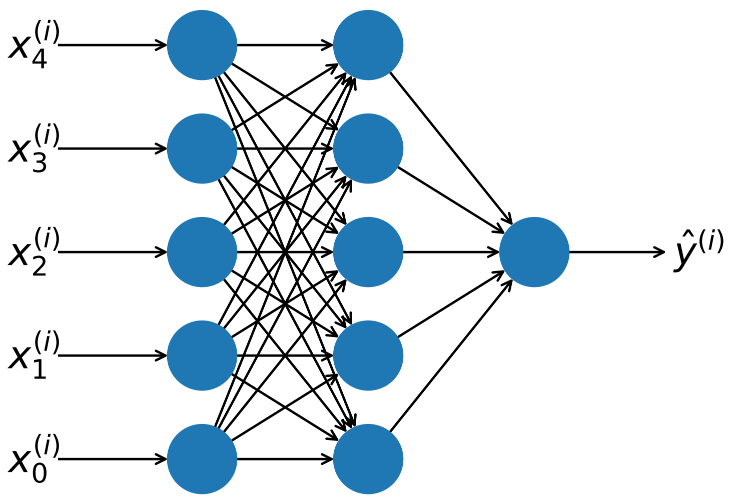

Figure 1.2: Basic example of a neural network that takes the inputs $x_j^{(i)}$ and yields output $\hat{y}^{(i)}$.

Each neuron determines the strength of its output by calculating a weighted sum of all the outputs of the previous layer, which eventually results in one final output. With a training set containing a lot of examples of inputs with the correct corresponding output, a machine learning algorithm can change the weights in the network such that the network can take new input data and output an accurate estimate.

This technique has proven to be useful in the field of air quality prediction in various ways. For example, machine learning has proven to be a promising way of filling in missing or very inaccurate measurements needed for a model simulation [Jin, 2019]. It can also be used to make short term predictions of concentrations of specific pollutants (such as $PM_{2.5}$) by using recent air quality and meteorological measurements [Xie, 2018]. In this thesis, we will attempt to improve these short term predictions by using the long term air quality predictions of LOTOS-EUROS. The main question of our research is:

In what sense does the inclusion of simulation data improve the $PM_{2.5}$ predictions of a neural network, if at all?

The structure of this thesis is as follows. First, we will look at the theoretical background of machine learning as well as the configurations that we will consider in our experiments in Chapter 2. Then, we will discuss what data we will be using as input in Chapter 3. Next, we will briefly look at how we will preprocess the data in Chapter 4. Then, in Chapter 5, we will take an extensive look at the results of the various experiments laid out in the previous chapters. Finally, we will draw some conclusions from the results, make some recommendations and suggest future work that might further improve the capabilities of the neural network in Chapter 6.

Chapter 2

Deep learning

In this thesis we will make use of a neural network to make a prediction of a future $PM_{2.5}$ concentration. At the heart of any problem involving machine learning we find the function f:

$$y = f(X, W, b) \tag{2.1}$$

Here, y is simply the value of the quantity that we aim to predict, in this case the $PM_{2.5}$ concentration some time in the future, but for other tasks such as handwriting recognition it might also represent a choice between several possibilities. Next, f is determined by the configuration of the network. Then, X is the input matrix that contains all the information that we wish to use for the prediction. Finally, W and b are the weight matrix and the biases, respectively. These are the coefficients that govern the connections between the different neurons in the network. The machine learning algorithm is able to fine-tune these values in order to improve the accuracy of y. These two variables together are often represented by the symbol θ if it is not necessary to distinguish between the two.

In this chapter, we will take a close look at f, W and b. That is, we will first consider some different types of networks and the different ways we can configure them in order to determine what might give us the most accurate prediction. Then, we will start to introduce the various aspects that are involved in the learning process that determines W and b. These two topics have a tendency to overlap, so we will look at them in the same chapter. Then, in the next two chapters, we will consider the configuration and preprocessing of the input X. The results y will be discussed in Chapter 5.

2.1 The Feedforward Neural Network

There are a few different types of neural networks, depending on how the neurons function and how they are arranged. The most basic type that we will look at is the Deep Feedforward Network, or Feedforward Neural Network (FNN). These neurons only have one output value, which it passes on to other neurons. In an FNN, the neurons are arranged as in Figure 2.1.

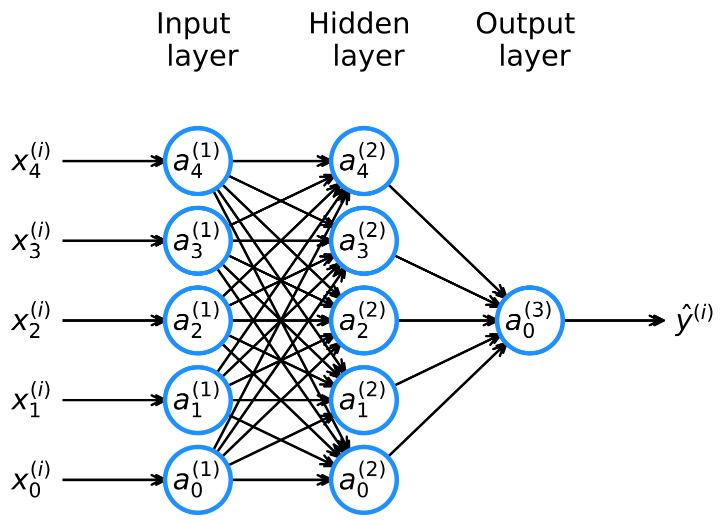

Figure 2.1: General arrangement of an FNN. The number of hidden layers and the number of neurons can be changed.

In Figure 2.1 we can identify three different layers: The input layer, the hidden layer(s) and the output layer. The input layer is fairly straightforward, as it primarily hands the values of X to the rest of the network without altering them much. The output layer is quite simple as well. It combines all the information given by the previous layer to give an answer. This layer will look different when using the network for a classification task, where the network has to sort the inputs between different categories i.e. handwritten number recognition. Our problem however is a regression task, where the expected output is just a single number. For the classification task the output layer contains a neuron for each option which indicates the likelihood of each option, but for regression tasks the output layer contains only one neuron.

The hidden layers are more complicated. The hidden layers are responsible for transforming the raw input into a usable result for the output layer. Unfortunately it is difficult to be more specific than that. Intuitively, the different hidden layers are there to separate the overall task into smaller steps. If for instance a neural network is used for image recognition, one hidden layer might be responsible for recognizing edges, with a different layer piecing those edges together to recognize shapes [Goodfellow et al., 2016]. Or, for our example of using past measurements to predict a future measurement, a neuron in a hidden layer might end up approximating the derivatives of certain input quantities much like a finite difference method which can then be used to calculate the final answer. In practice however, the specific purpose of the neurons in a hidden layer are nearly impossible to grasp. So for the most part it is best to treat these layers as black boxes.

Mathematically, the neurons output some value determined by the values given by the neurons in the previous layer. Specifically, the output of a neuron is given by the activation formula:

$$a = g(\Sigma_{i=1}^{n} w_i x_i + b) \tag{2.2}$$

Here a is the output, or activation of the neuron, x_i is output of the ith neuron in the previous layer with corresponding weight w_i, b is the bias, n is the total number of neurons in the previous layer and g is the activation function. The weights determine which of the previous neurons are important for this neuron and the bias is there to force the input to be sufficiently high before the neuron can become active. The activation function can force the output of the neuron into a certain range, as well as govern how gradually the input should increase for the neuron to become more active. More on the activation function can be found in Section 2.6. As we can see in Figure 2.1, the output gets sent to every neuron in the next layer. This formula is calculated for each neuron in the layer, with each neuron having its own bias and set of weights. These are the parameters that the machine learning algorithm will tweak during the training process which we will describe in Section 2.4. The activation function g, on the other hand, is the same for every neuron in the layer.

2.2 The Recurrent Neural Network

While the FNN can perform quite well on various tasks, it might not be the best choice for the time series prediction considered in this thesis. An FNN might not be able to properly realize that the different inputs form a time series for different variables. After all, the input matrix gets flattened into a vector and it is up to the network to learn that these separate input variables are supposed to form a time sequence. As such, it might be better to use a network that has some kind of concept of time built-in.

One such type of network used for processing sequential data is the Recurrent Neural Network, or RNN, which can create a sense of memory by using feedback loops [Goodfellow et al., 2016]. As shown in Figure 2.1, the signal from the nodes in an FNN only travels forward to the next layer in the network. In contrast, the output of neurons in an RNN can go to neurons in the same layer or sometimes to neurons in the previous layers the next time the network looks at a sample. This way, the neural network can remember the input from a sample from before the sample that it is currently considering, as it can save the state of previous samples in these feedback loops.

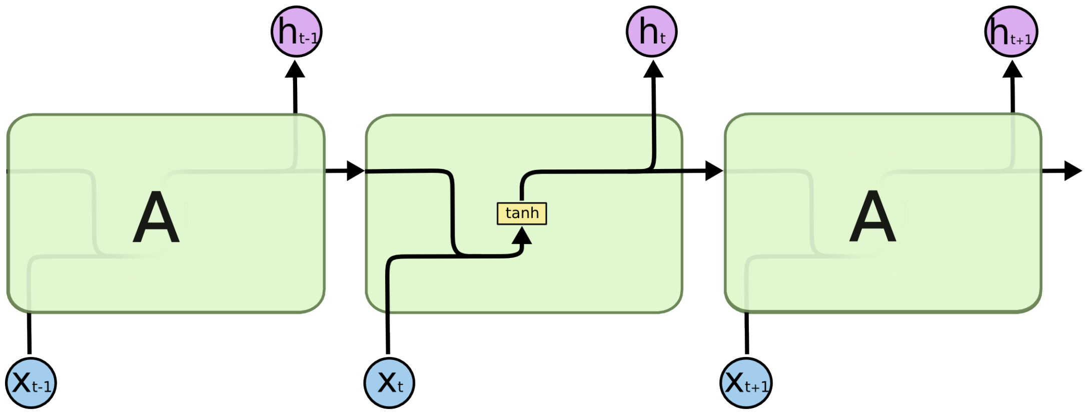

Figure 2.2: A simple example of a diagram depicting the internal structure of an RNN neuron. X_t is the input of the neuron and h_t is the output. Source [Olah, 2015].

These feedback loops do of course come with an additional cost in computational time and memory, as the feedback connections come with their own weights and biases which need to be adjusted, but the added complexity is often beneficial for complex tasks such as air quality prediction.

2.3 Long short-term memory

There are many ways in which the feedback connections can be made. The method used in this thesis is the so called Long short-term memory (LSTM) network [Hochreiter and Schmidhuber, 1997]. The layout of this network is similar to the basic FNN described in Section 2.1, except that all the neurons are replaced by LSTM cells. The LSTM cell keeps an internal cell state which it uses as a way to remember information from previous samples. Compared to the more basic feedback loop described in Section 2.2, the LSTM cell connects the cell state and the current sample with three gates: the input gate, the forget gate and the output gate. The input and forget gates are responsible for updating the memory using the information from the current sample. The first determines the new information that the cell should remember while the latter determines what piece of information stored in the cell state is no longer useful and should therefore be forgotten. The output gate on the other hand determines what information stored in the memory is relevant to the current sample. These gates exist to make sure that the memory is not overwritten by irrelevant information and to make sure that irrelevant memories do not get in the way of the prediction task.

Internally, the LSTM cell looks as follows:

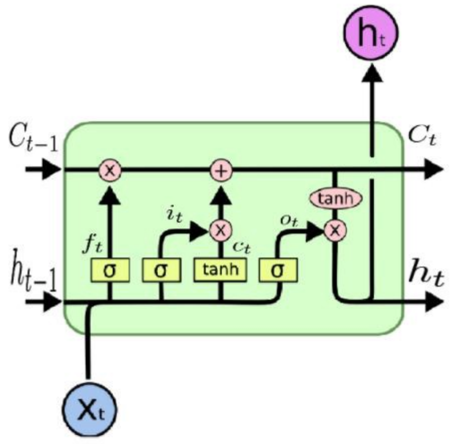

Figure 2.3: A diagram depicting the layout of an LSTM cell. Source [Olah, 2015].

In this figure, x_t and h_t represent the input and the output of the cell. Furthermore, C_t is the cell state that the cell uses as memory. In this case, the subscript denotes the sample; t is the current sample and $t-1$ the previous sample. So h_{t-1} and C_{t-1} are the output and cell state from the previous sample. f_t, i_t, c_t and o_t are the activations of the various gates: f_t for the forget gate, i_t and c_t for the input gate and o_t for the output gate. The yellow blocks represent the activation equations for the gates much like Equation 2.2. For f_t, i_t and c_t the corresponding activation function will be fixed to the one shown in the Figure 2.5 (either the sigmoid or the hyperbolic tangent) but we are free to chose the activation function for the output o_t, which will be discussed in Section 2.6.

2.4 The Training Process

In essence, the training process for a neural network is very similar to fitting the parameters of some simple model to training samples. An optimization algorithm tweaks the parameters in an attempt to minimize some cost function. Then, the dataset is split into two parts: the training set and the validation set. The training set is used for tweaking the parameters and the validation set serves as a control group which consists of unseen data that is used to evaluate the performance of the model once it is finished training. Since the number of samples and changeable parameters is so large, using a good optimization method is very important. The most popular optimization method for neural networks is called gradient descent. In this method, the gradient of all tweakable parameters is calculated with respect to the error. All the parameters then take a step in the direction, according to the formula:

$$\theta_t = \theta_{t-1} - \eta \nabla f(\theta_{t-1}) \tag{2.3}$$

Here, θ_t denotes the parameters of the current iteration of the algorithm and θ_{t-1} denotes those of the previous iteration. f is the cost function and η is the learning rate, a parameter that determines by how much the weights will change in the direction of the gradient. This parameter typically is decreased over the course of the training period in order to pinpoint a good set of weights by the end of the training process. Of course, the gradient can only point at a local optimum, which might not turn out to be the global optimum. As such, it is important to not decrease the learning rate too quickly. After all, choosing a fairly large learning rate will speed up the learning process as it changes the weights more rapidly. Choosing the learning rate to be too high can be problematic, however, as the big change in weights might cause them to overshoot the optimum that the gradient is pointing towards, causing the weights as well as the cost function to oscillate. On the other hand a learning rate that is too low not only makes the learning process very slow, it might also cause learning to stop at a bad point due to the weights being unable to escape a very local optimum.

2.4.1 Batch gradient descent

When using gradient descent it is important to consider how to use the input data. Because the goal of the algorithm is to minimize the cost function over the entire dataset, it seems reasonable to calculate the gradient over the entire dataset as well. Alternatively it is possible to only calculate the gradient and update the weights for just one sample and go through the entire dataset separately in a random order. In practice taking the middle road by selecting a set number of samples to calculate the gradient seems to work best. As such, we will be using a batch size of 128 samples. Choosing just one sample is considerably slower, as using multiple samples at a time means that the weights do not have to change as often. Also, the gradients for each sample in a batch can be calculated simultaneously as they are only added together at the end of the batch when changing the weights. This allows for some parallelization. The learning process also tends to be more noisy when going by one sample at a time, as the outliers in the training data do not immediately get averaged out. On the other hand, using the entire dataset at once seems to hinder the performance of the network when predicting unseen data. According to the findings of [Keskar et al., 2016], using a large number of samples per batch leads to the algorithm settling on a so called sharp minimizer more often. A sharp minimizer is a local extreme value where the value of the loss function varies quickly in the neighbourhood around the extreme value. Since the data used to train the algorithm does not perfectly match the unseen data, the learned parameters do not match the minimum of the loss function for the unseen data either. As a result, the performance of the algorithm will be worse because the minimizer is sharp. Conversely, smaller batches tend to converge to flat minimizers where the loss function varies a lot less. This appears to be caused by the noisy nature of the small batches, as the neighbourhoods where the gradient points towards a the local minimum tends to be smaller for sharp minimizers.

2.4.2 Adam

As mentioned previously, it is often useful to adjust the learning rate during the training period in order for the network to properly converge to a final set of weights. While it is possible to change the learning rate with a simple method, such as a linear decay from a high learning rate to a lower one, some more advanced techniques have been developed that are computationally inexpensive. The technique that we will be using for the experiments in this thesis is called adaptive moment estimation, or Adam [Kingma and Ba, 2014]. Adam uses a separate learning rate for each different parameter which is calculated using the moving average of the first and second moment of the corresponding parameter. This way the algorithm can determine how close the network is to being done learning and adjust the learning rates accordingly.

For every step of Adam, the gradient is first calculated much like the regular gradient descent algorithm. Next, the moving averages of the two moments are updated with the latest moments. Then, the moving averages are corrected for the bias that is caused by the fact that they first have to be initialized before they can be calculated for the first step. Then the weights are updated using the two moments instead of the gradient. The exact formulas of this process are shown below:

$$\begin{aligned}
g_t &= \nabla_\theta f_t(\theta_{t-1}) \\
m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\
v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \\
\hat{m}_t &= m_t/(1-\beta_1^t) \\
\hat{v}_t &= v_t/(1-\beta_2^t) \\
\theta_t &= \theta_{t-1} + \alpha \hat{m}_t/(\sqrt{\hat{v}_t} + \varepsilon)
\end{aligned} \tag{2.4}$$

Adam comes with four different parameters that can be chosen. α is the base learning rate before any adaptations are made due to the first and second moment. β_1 and β_2 represent the decay rates of the two moving averages. Finally ε is simply a small parameter that ensures that the algorithm does not divide by zero. In this thesis, we chose the values for β_1, β_2 and ε to be equal to the default values suggested by [Kingma and Ba, 2014], that is $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\varepsilon = 10^{-8}$. The value for α will be considered later in Section 5.4.

2.5 Dropout

As mentioned previously, the most optimal values for the weights and biases are determined by fitting them to training data. While this training data is chosen to be representative of the situations that the model will try to predict, in practice the two will not quite match up. So often the neural network will end up learning behaviours that might improve its performance on the training data, but which makes the performance on unseen data worse. This phenomenon is referred to as overfitting and while it is to some extent inevitable, there are several techniques that help reduce the negative effect on the performance.

One of the contributing factors to overfitting is the fact that some nodes in the same layer might end up becoming dependent on one another. So, for the training data at least, an error made by one node might be compensated by an error made in another node. After all, the weights are all updated at the same time, so the nodes are to some extent aware of what other nodes are doing. This behaviour does not generalize well to unseen data. The technique for combating overfitting that we use for the experiments in this thesis is dropout [Srivastava et al., 2014]. The term dropout refers to the idea that some of the nodes should randomly "drop out" making the remaining nodes more autonomous as the node that it might depend on could disappear next sample. The application of dropout is slightly more complex for LSTM networks than for FNNs, so we will first explain dropout for FNN and then point out the difference in application for LSTM.

2.5.1 Dropout for FNN

Again, we will first look at how the technique works for FNNs before considering how it works for the LSTM networks used in the experiments later. During training, every node in the hidden layers of the network has a chance p to not be present during the next batch of samples. In particular, the output of a removed node will be set to zero. This is illustrated in the following diagram:

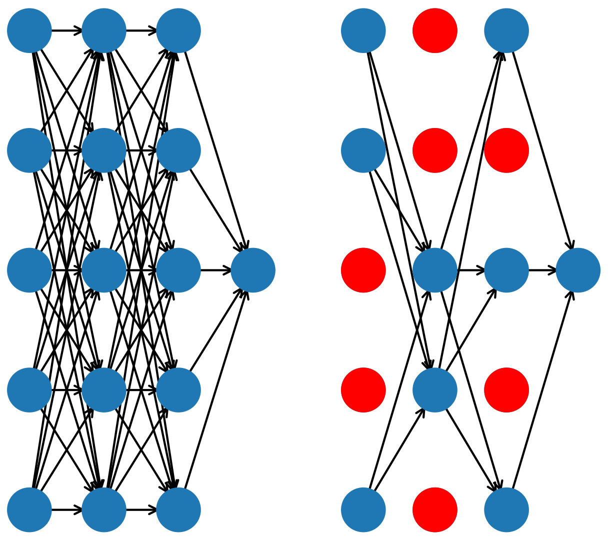

Figure 2.4: The diagram on the left depicts a regular neural network, the one on the right is an example of what it could look like after dropout has been applied, with the dropped out nodes in red.

This way, when there is a node that is dependent on a different one there will be batches when that different node is dropped out. So when the optimizer calculates the gradient along which the weights should change to minimize the error, the gradient of the dependent node will not be compensated and as a result the node will end up changing to be less dependent. As the learning process is repeated, the nodes will all end up creating some sort of feature of their own, as they have to learn to work with an entirely random set of other nodes. This method also forces the gradients for a dropped out node to be equal to 0. This ensures that the associated weights will not change and as such being dropped out will not negatively affect the performance of a particular node. After the training period, the nodes will no longer have a chance to drop out. One thing to note is that because of the nodes dropping out, the network trains with fewer nodes available than during testing, which means that the weights have to become higher in order to get the same total input. During testing, however, this means that generally speaking the total input of the nodes ends up being too strong. To compensate for this all weights corresponding to dropout nodes are multiplied by the chance of dropping out p during testing. Otherwise, the nodes in the next layer would learn to work with an incorrect total input and would not work well during testing.

2.5.2 Dropout for LSTM

When first applying this technique on an LSTM network, however, it was found that dropout did not work. By completely setting the output of an LSTM cell to zero, the cell state is lost as well. As such, it becomes very difficult for an LSTM network to store any sort of long term memory during training. Fortunately, [Zaremba et al., 2014] found a way to make dropout work for LSTM as well. By only dropping out the output of a neuron that goes to the next layer in the network and not the output that goes to the same neuron for the next sample, dropout can still effectively do its job while still keeping the long term memory intact.

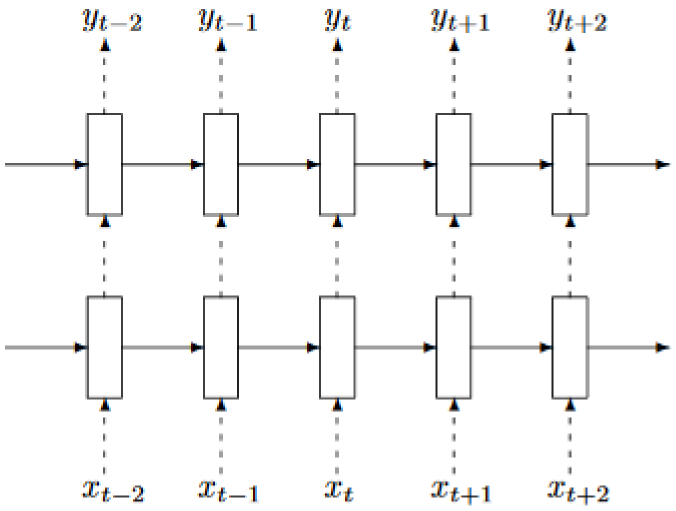

Figure 2.5: Schematic representation of an LSTM network. The rectangles represent layers of neurons. The dotted arrows go from input to output and the solid arrows represent feedback loops within the same layer. Source [Zaremba et al., 2014].

In Figure 2.5, the dotted arrows can be affected by dropout while the solid arrows cannot. This means that information from previous inputs can still be used to find the next output as it can freely travel horizontally. This way a neuron still has to learn to work with a random set of other neurons without losing its memory due to dropping out.

2.6 Activation function

As mentioned earlier, the activation function determines the way neurons in the neural network connect to each other. On the small scale, the neuron takes the weighted sum of the previous neurons, then it shifts the sum by its bias and then reshapes the different results according to its activation function. This idea of a neuron being activated or not stems from biology, where the neurons on a brain that are currently not important, or rather, neurons that get very few input signals, will not fire signals to other neurons. On the large scale, networks with different activation functions will end up learning and converging slower or faster for different problems. They can also have more specific drawbacks that makes them potentially unsuitable for solving certain problems. Efficiency can also be a factor. In general, some desirable traits of activation functions include non-linearity, as using multiple linear layers does not add anything meaningful. It is also useful for an activation function to be continuously differentiable, as the gradient descent algorithm needs a gradient after all. In this section we will consider the advantages, as well as any possible drawbacks, of two different activation functions: the hyperbolic tangent and the Rectified Linear Unit, or ReLU. The performance of these two activation functions on air quality prediction will later be compared in Section 5.3.

2.6.1 Tanh

The hyperbolic tangent is given by:

$$f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.5}$$

It is presented graphically in Figure 2.6.

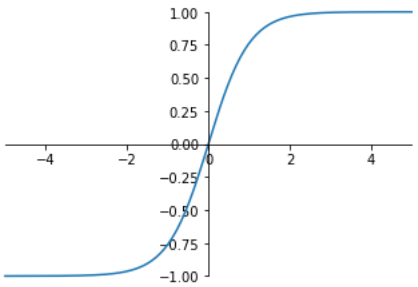

Figure 2.6: Visual representation of the tanh function. The x-axis represents the weighted sum of all the incoming signals, the y-axis represents the activation of the neuron.

The function is clearly both non-linear and continuously differentiable, which as stated earlier are desirable traits for an activation function. The function also provides a bounded output, which is not a requirement for an activation function. Nonetheless, it is a useful property as it prevents the output from blowing up. When looking at the plot of the tanh we can see that the shape is somewhat similar to that of a step function, which is useful because intuitively, the way a biological neuron activates is similar to a step function. The tanh changes quite rapidly when the input is close to zero, which tends to make distinguishing between low and high inputs easier for the neuron. The downside is that tanh neurons can experience saturation, which means that neurons that typically output either a very low or very high value will get stuck. This is because at either end the derivative of the tanh is close to 0, so the gradient descent algorithm will not try to change the weights for that neuron by much [Gulcehre et al., 2016].

2.6.2 ReLU

A Rectifier Linear Unit is a neuron first introduced by [Glorot et al., 2011]. It uses the rectifier as its activation function:

$$f(x) = \text{rectifier}(x) = \max(0, x) = \begin{cases} x, & x \geq 0 \\ 0, & x < 0 \end{cases} \tag{2.6}$$

which results in the following plot:

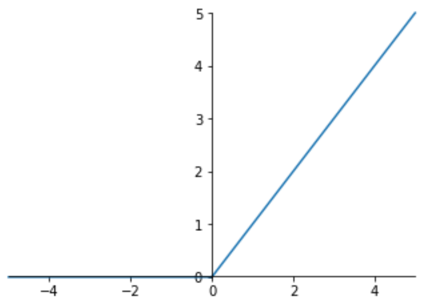

Figure 2.7: Visual representation of the ReLU activation function. The x-axis represents the weighted sum of all the incoming signals, the y-axis represents the activation of the neuron.

Looking at the function it quickly becomes apparent that the derivative of this function is not differentiable at $x = 0$. Fortunately, [Glorot et al., 2011] found that the gradient descent algorithm can still manage to work fine in spite of this. The output of ReLU is unbounded, which on the one hand might be problematic as the output might blow up, but on the other hand it prevents the saturation that is present when using the tanh activation function. A big advantage of ReLU is that it is a very efficient activation function. The function and its derivative is very easy for a computer to calculate compared to exponents of the tanh. ReLU also tends to result in very sparse networks, meaning that when the neural network considers a sample, a portion of the neurons will remain completely inactive. This is useful for a multitude of reasons, in particular as having neurons that output exactly 0 will get ignored by gradient descent, making the learning algorithm much faster. This can also be detrimental, because the weights and bias of a neuron might get so low that the neuron will never activate again. And if a neuron never activates, it is also unable to change the weights and bias to become usable again. This is called the dying ReLU problem, where a large portion in the network contribute nothing to the network [Karpathy et al., 2016].

2.7 Cost functions

The choice between different cost functions depends on the problem that the neural network aims to solve. For example, if it is more important for the network to accurately predict high concentrations then the loss function should be higher for those high concentrations such that the learning algorithm will focus more on learning from those high concentration samples. The functions considered are the Root Mean Squared Error (RMSE), the Mean Absolute Error (MAE) and the Relative Absolute Error (RAE). These correspond to the following equations:

$$E_{RMS} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y_i})^2} \tag{2.7a}$$

$$E_{MA} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y_i}| \tag{2.7b}$$

$$E_{RA} = \frac{1}{n}\sum_{i=1}^{n}\frac{|y_i - \hat{y_i}|}{y_i} \tag{2.7c}$$

Here E denotes the value of the error, n is the number of samples, y_i is the actual value of sample i and $\hat{y_i}$ is the corresponding estimated value calculated by the neural net. Looking at the formulas it becomes apparent that using RMSE penalizes large errors more harshly, which tend to occur when the actual $PM_{2.5}$ value is either very high or very low. The MAE on the other hand is a more moderate cost function. Compared to the RMSE a network trained by minimizing the MAE should have more very bad estimations, but the estimations that are already close to correct should in general be closer still. Finally the relative error scales with the actual value of the sample, so the small samples should be very accurate, while the bigger samples will not be accurate as they will contribute very little to the total error either way. For the purposes of predicting $PM_{2.5}$ the RMSE seems to be the best fit, as very high concentrations are more dangerous and as such should be predicted more accurately, but we will look at the effect of these different errors on a simple training set later in Section 5.2.

Chapter 3

The input

Unlike regular models, it can often be unclear what inputs are useful for a neural network. While theoretically adding new, unrelated parameters should not diminish the performance of a network (assuming that the parameters do not contradict each other), in practice it might take a lot of learning for a neural network to realize what information is important and what is not. As such, it is important to put some consideration into the input for our neural network.

In this chapter, we will first briefly look at which measurements are available. Then, we will look at correlation which gives us an idea of how useful previous measurements and measurements from surrounding stations are for a prediction in the future. From there, we can decide from how far back we should use the data and how long in advance we could make our predictions, which allows us to construct the input matrix that the neural network will learn from. Finally, we can look at what advantages the inclusion of the LOTOS-EUROS model simulations could bring and how we should implement the extra data into the input matrix.

3.1 Measuring stations

The measurement stations provide hourly measurements for the concentration of six different aerosols; namely CO, NO_2, O_3, SO_2, PM_{10} and $PM_{2.5}$. The station provides meteorological observations as well; in particular the surface temperature, the amount of rain and two wind speed components. We have data available from September 1, 2014 until December 31, 2016. While there are thousands of these stations spread all over China, performing experiments for all these stations would take too much time. The main station that we will focus on is located in Beijing and is labeled as the first in the group. This measurement station is called station 1001A. The experiments comparing various configuration settings for the neural network will be done for that site, as it is in one of the most heavily polluted places so accurate predictions are particularly valuable there.

For the experiments with and without including the simulation data, we will look a bit further, because we want to study the effect for different areas. After all, the amount of pollution can differ a lot between urban and country areas and also between cities. To this end, we have selected 25 stations at random out of the available measurements, which hopefully gives us a good variety in pollution. The location of these stations can be found in Figure 3.1.

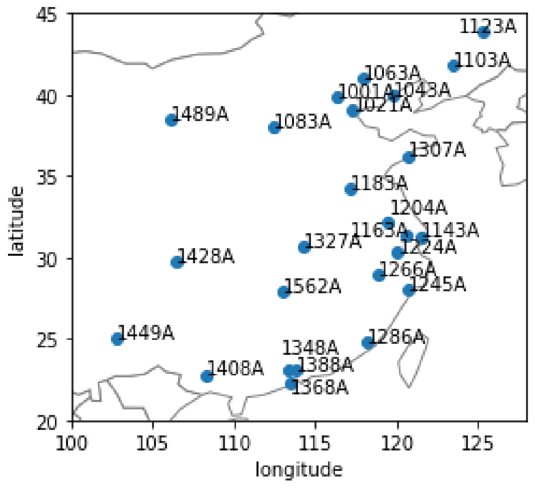

Figure 3.1: Map of the locations of the measurement sites used for our experiments.

Note that we also have data from stations that are not shown on this map. While we will not focus on these stations in any of our experiments, their measurements could still help when predicting air quality at the main stations. More on that in the next section.

3.2 Correlation

One useful metric when considering what input parameters should be included is the Pearson correlation coefficient ρ, which is a measure of the linear correlation of two different variables. In our context of air quality predictions, we will look at two different correlations. First, we will compare the $PM_{2.5}$ concentration at various measuring stations with the values at the stations closest to it for the spatial correlation. Then, we will also compare the linear correlation of $PM_{2.5}$ concentrations at a certain time stamp to the concentration at various different moments in the past, yielding what is known as the autocorrelation. If these coefficients suggest a high amount of correlation, then it is a worthwhile inclusion to the network input. That is to say, if the spatial correlation is high then that would theoretically justify the inclusion of $PM_{2.5}$ measurements from neighbouring stations, while the autocorrelation should give us some idea of how far ahead of time we can reasonably predict the future, while also indicating until which time data from the past is still sufficiently useful for the algorithm.

The correlation coefficient ρ is always in the interval $[-1, 1]$. A correlation of 1 implies a positive, completely linear correlation, while a value of -1 implies a negative linear correlation. If the correlation is equal to 0 there is no correlation at all between the two parameters. In the case of considering the usefulness of parameters as neural network input, the sign of ρ is irrelevant, we only need it to imply some correlation, be it positive or negative. While the value of ρ is different for each station, we will only take a close look at the coefficients for a few representative stations as most stations yield very similar results. First, the spatial correlations for station 1002A with respect to its closest neighbours are as follows:

Table 3.1: Pearson correlation coefficient of the $PM_{2.5}$ concentrations for station 1002A with respect to the given stations.

station	ρ
1003A	0.9606
1004A	0.9713
1005A	0.9518
1006A	0.9595
1007A	0.9293
1008A	0.8872
1010A	0.8285
1011A	0.9434
1012A	0.9081

From this table it becomes clear that there is quite a strong connection between the $PM_{2.5}$ values at a measurement station and at neighbouring stations. For the autocorrelation, we have used the same stations as the ones in Table 3.1 and plotted the autocorrelation coefficient against the time difference to get a loose idea of how much information is usable. The time difference is increased with steps of one hour, as the measurement stations only conduct measurements every hour.

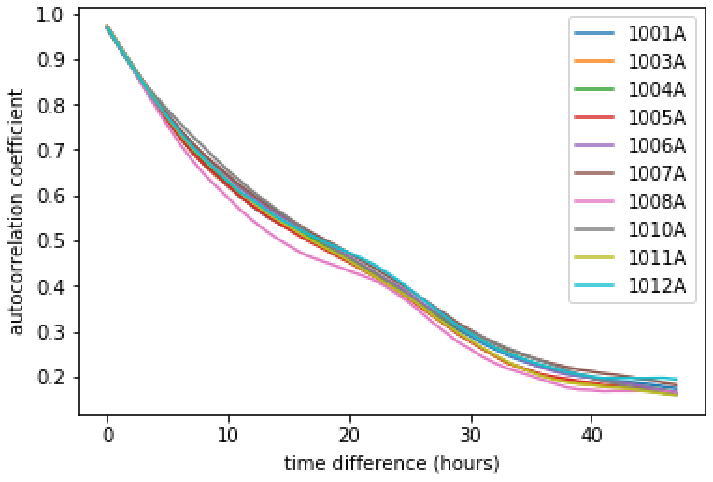

Figure 3.2: The autocorrelation of 10 different stations plotted against the time gap.

As we can see in Figure 3.2, the autocorrelation is very similar across the different stations, where the value of ρ goes below 0.75 after approximately 6 hours and below 0.5 after approximately 18 hours. As such we seem to be able to make a decision on what data to include for any station that we want to analyze.

3.3 The input matrix

For the input of the network we will consider all the measurements provided by the measuring station, both the meteorological measurements and the air quality measurements. The $PM_{2.5}$ measurements of the three closest neighbouring stations will also be included, as those values are all very strongly correlated to one another, as seen in Table 3.1. All these parameters will be taken for a number of hours, which allows the neural network to also consider the change in these parameters over recent time. The input matrix now looks as follows:

$$\begin{bmatrix} A_{t_0-t_1} & \dots & A_{t_0} \\ W_{t_0-t_1} & \dots & W_{t_0} \\ N_{t_0-t_1} & \dots & N_{t_0} \end{bmatrix} \xrightarrow{LSTM} \hat{y}_{t_0+t_2}$$

Where t_0 is the most resent time for which we have a measurement, t_1 represents how many hours the data will go back and t_2 represents how far in advance we want to predict the future $PM_{2.5}$ concentration. A, W and N represent the air quality, the meteorological data and the neighbours, respectively. Looking at the autocorrelation data in Figure 3.2, it seems reasonable to choose t_2 to be equal to six hours, with an autocorrelation of about 0.75. Furthermore, we choose t_1 to be equal to 16 hours, as that is right before a steep drop off in correlation. When using the simulation results as input as well, the input matrix will look slightly different, which we will see in the next section.

3.4 The inclusion of simulation data

The simulation data used as input comes from the LOTOS-EUROS model. This model is a Chemical Transport Model (CTM) which was primarily developed to be used in Europe, but [Timmermans et al., 2017] have shown that it is also applicable to air quality predictions in China. They showed that while most emission sources for $PM_{2.5}$ were fairly accurate, some were underestimated and other factors which were ignored entirely by the model turned out to be more influential than anticipated. Overall, when compared to measurements from both the ground and from satellites LOTOS-EUROS captures a lot of aerosols reasonably well, but systemically underestimates the particulate matter concentrations.

Nonetheless, the data produced by LOTOS-EUROS should be a worthwhile inclusion to the input of the neural network, as suggested by [Jin et al., 2019], because it captures the profile of the $PM_{2.5}$ measurements well. That is, the model is good at predicting when the $PM_{2.5}$ concentration suddenly peaks and when it suddenly drops. This is something that LSTM networks can have some trouble with in general, where the LSTM predictions will closely resemble the behaviour of the measurements, but with a time delay equal to t_2 [Deng et al., 2019]. So using the CTM results might help the neural network by combating this specific problem. Another advantage of the model simulation compared to the measurement data is that it is a prediction of the future instead. This is useful for the neural network because it means that instead of forcing a 6 hour gap between the most recent measurement and the data point that we wish to predict, the input from the CTM at the time we wish to predict can be used. This becomes especially valuable for those sudden peaks or drops, as those often happens in a shorter time span than 6 hours. So if the neural network can learn to recognize those peaks or drops by looking at the simulation data, it might be able to predict them better.

When relating the output of LOTOS-EUROS and the measurements from the measuring stations, it is important to note that their locations are different. After all, LOTOS-EUROS produces results on a rectangular grid of about $28km \times 21km$, while the measurement stations are placed at available locations in the real world and not on any sort of grid structure. In order to bridge this gap we will simply use the output from the grid point closest to the station as input for the neural network. After all, in Section 3.2 we found that the spatial correlation for $PM_{2.5}$ is quite high for the surrounding area. So the simulation results at the nearest grid point should still be useful to the neural network.

Including extra data from the future does however mean that the input matrix will be bigger as well, which may require a bigger neural network. In order to make a fair comparison between the networks with and without the CTM results as input it is important to keep the rest of the network configuration the same. So to compensate we cut off the furthest six hours of the input matrix, which insures that we do not input more information than before. After all the hours 16 to 22 do not seem very important when looking at Figure 3.2. So, to summarize, the input matrix when including the CTM looks as follows:

$$\begin{bmatrix} A_{t_0-t_1+t_2} & \dots & A_{t_0} & \dots & 0 \\ W_{t_0-t_1+t_2} & \dots & W_{t_0} & \dots & 0 \\ N_{t_0-t_1+t_2} & \dots & N_{t_0} & \dots & 0 \\ C_{t_0-t_1+t_2} & \dots & C_{t_0} & \dots & C_{t_0+t_2} \end{bmatrix} \xrightarrow{LSTM} \hat{y}_{t_0+t_2}$$

Where C represents the CTM data. Note that LOTOS-EUROS produces results for both $PM_{2.5}$ and PM_{10} concentrations so we will include both.

Chapter 4

Preprocessing

An important aspect of any machine learning problem is the preprocessing of the data. After all, the insides of the finished algorithm are very complicated, so it is important that the input data is easy for the neural network to learn from. In this chapter, we will discuss various aspects of preprocessing. In particular, we will first analyze the availability of the measurement data, then we will discuss the interpolation techniques used to patch up some of the missing data. Finally, we will briefly mention the normalization of the input data.

4.1 Availability of the data

As with any measurement, the air quality measurements used for learning are not perfect. Specifically, there are measurements that are missing. While a neural network should in theory be able to function with an incomplete dataset to learn from, it will still make the performance of the algorithm worse and as such the problem of missing values should be addressed. First, the rates at which the measurements are available are given in the table below:

Table 4.1: The rate of availability for the six different aerosols at station 1006A from 2013 until 2018.

particle	%
CO	88.742355
NO_2	88.728976
O_3	85.537080
SO_2	89.319572
PM_{10}	69.319572
$PM_{2.5}$	89.252676

Next, in order to get a better understanding of the spread of the availability of the data, a matrix representation of the sample availability across the entire time period has been made:

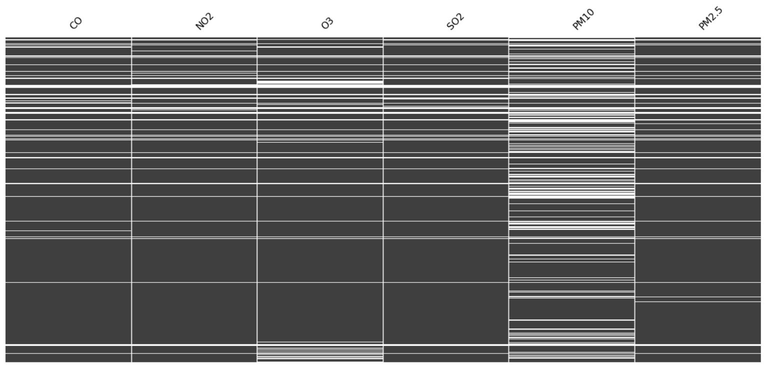

Figure 4.1: A matrix representation of the availability of each aerosol across all of the taken samples. Here a white line indicates that a sample is missing. The width of a white line indicates how many samples are missing in a row.

So we can see that, in particular for PM_{10}, the amount of data that is missing can be surprisingly high. Not only that, but in Figure 4.1 we can see that the periods where data is missing can sometimes be quite long. While we can patch up the smaller gaps with the interpolation techniques described in the next section, it is important to realize that using too much interpolation in one sample might make the sample too inaccurate to be useful and as such should be discarded instead.

4.2 Interpolation techniques

For smaller time gaps, the missing data can simply be replaced by using linear interpolation. However, if there is a big time gap between two available data points, then it might not be wise to rely on just linear interpolation. After all, the correlation coefficient drops off quite quickly as the difference in time grows. As such, we use the k-nearest neighbour algorithm for when the gap of missing data is longer than 6 hours. This interpolation technique estimates the missing value by taking the averages over the values of k of its closest neighbours, weighted by the inverse of the distance. The formula is given by:

$$x(t) \approx \sum_{i=1}^{k} w_i x_i(t) \tag{4.1}$$

with

$$w_i = \frac{(1/r_i)}{\sum_{i=1}^{k} (1/r_i)} \tag{4.2}$$

Here, $x(t)$ is the missing value at some station at time t, x_i is the corresponding value at station i and r_i is the distance between the two stations. Throughout the experiments the value of k used was chosen to be equal to 3. If the measurement for one of the three nearest stations is also missing, we will use the next closest station instead. However, as stated in the previous section, we want the interpolations to be reasonably accurate. So we will only use neighbouring stations within a certain distance of the main station. If there are less than three available measurements, then we will give up on interpolation and discard any sample that would use the missing measurement.

4.3 Normalization

The magnitude of the different types of input data can vary a lot. After all, the temperature only varies about 50 degrees while the $PM_{2.5}$ concentration can vary from $1\mu g/m^3$ to over $500\mu g/m^3$. For a human brain it is not so difficult to identify a meaningful difference in temperature or $PM_{2.5}$, especially when we know the units of the two quantities. However, a neural network that just started learning cannot tell that a difference of 25 is much more significant for the temperature than the $PM_{2.5}$ concentration. While the network is capable of learning the difference by changing the input weights, it does get in the way of learning to tackle the actual problem of learning to predict the future $PM_{2.5}$ concentration. So while not strictly necessary, it is a good idea to sort this problem out ourselves before the training starts. We do this by normalizing each quantity in the input matrix. We do this for each measurement using the following formula:

$$x = \frac{x - \mu}{\sigma} \tag{4.3}$$

Here, x is a measurement, μ is the mean of said measurement across the entire dataset and σ is the corresponding standard deviation. Using this formula, we can make sure that every quantity in the input matrix has a mean of 0 and a standard deviation of 1. This ensures that initially the neural network does not incorrectly get dominated by inputs that happen to be of a higher magnitude than others and it also ensures that the network can detect outliers more easily.

Chapter 5

Results

In this thesis, we try to predict $PM_{2.5}$ concentrations six hours ahead of time using a neural network. As we could see in the previous chapters, there are a lot of different factors when it comes to training the neural network. There are also many features that define the configuration of a network. For some of these network features we will conduct some experiments to see how they affect the performance of the network. Unless specified otherwise, the so-called hyperparameters we use are given in Table 5.1.

Table 5.1: Hyperparamters used for the neural networks unless stated otherwise.

Parameter	Value
Number of hidden layers	2
Neurons in hidden layers	100/100
Epochs	100
Network type	LSTM
Batch size	128
Optimization algorithm	Adam
Base learning rate	0.0005
Dropout probability	0.5
Loss function	Root Mean Squared Error
Activation function	ReLU
Input parameters	13
Input interval	16 hours

As mentioned in Chapter 1, the main goal of this thesis is to see if including results from a Chemical Transport Model (CTM) improves the performance of the neural network when predicting $PM_{2.5}$ concentrations. The performance of a neural network can be quite a broad term, however. The most obvious metric would be the loss function across the validation set, but that does not tell a lot about how the model handles all the different data points. After all, the error gives some kind of average across all samples, while we are also interested in certain specific samples. Instead, we will primarily be looking at scatter plots in this chapter which put the concentration as predicted by the neural network on one axis and the corresponding measurement on the other for each sample in the validation set. This way, we can get a good indication of how well the network performs when predicting high values while also getting an impression of how the model performs on the lower values, which are typically much more abundant in the dataset.

Furthermore, for samples that are particularly interesting, we will look at a time series plot containing the predictions made by both the neural networks with and without model data, as well as the measured data and the model data itself, which will hopefully illustrate why including the model data does or does not work for those specific instances. Some experiments might also feature training history plots. These plots visualize the learning process by plotting the validation error as a function of epochs. The value of the validation error is an indicator of the overall performance of the model and the plot might give more insight into the learning process when changing certain hyperparameters pertaining to the network configuration.

In this chapter, we will first look at a basic example at station 1001A using all the hyperparameters specified in Table 5.1. That way we can get acquainted with the various plots described in the previous paragraphs and we can also set a baseline to which we can compare our other results. Then, in Sections 5.2 - 5.4, we will look at the effect of the loss function, activation function and learning rate on the performance of the neural network. Next, in Section 5.5, we will also use the simulation data as input, but still only for station 1001A. After that, we will look at the performance of FNN compared to LSTM. Finally, in Section 5.7, we will repeat our experiments for all 25 stations plotted in Figure 3.1 both with and without the inclusion of the CTM data.

5.1 Baseline example

First, we will thoroughly examine the results for station 1001A, without including the simulation data. The neural network uses the configuration described in Table 5.1. We will first consider the scatter plot to see which samples the network can predict well.

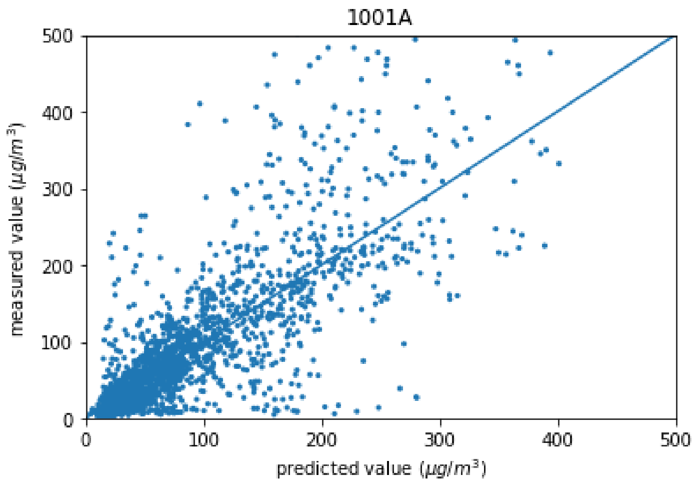

Figure 5.1: Scatter plot of the $PM_{2.5}$ concentration predictions made by the neural network against the corresponding measurements for the test dataset. The line $x = y$ is included for reference.

Intuitively, the closer a point is to the blue line, the more accurate the prediction becomes. The points that are far to the right of the line are points that were overestimated and points far to the left are underestimated. In this plot we can see that there seems to be a lot more low $PM_{2.5}$ values which are all clumped up at the bottom left of the graph. As a lot of samples seem to overlap in the corner, we will henceforth indicate the density of the samples with a color gradient going from yellow for a high density to blue for a low density:

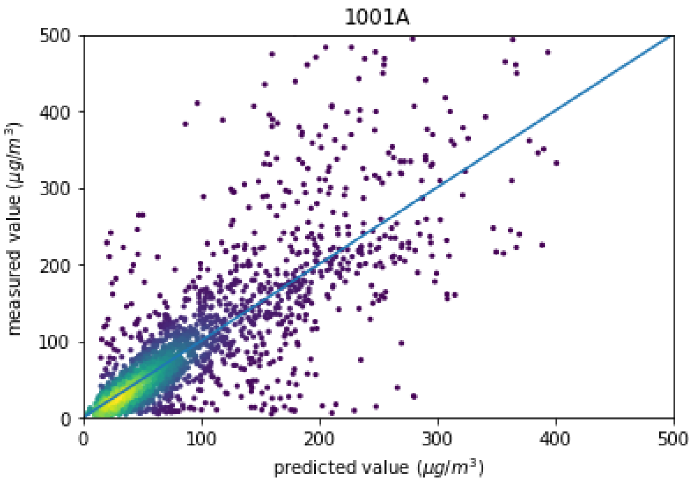

Figure 5.2: The same scatter plot as Figure 5.1, but now with a color gradient to indicate the point density.

In this plot we can now see that most low measurement predictions are fairly accurate, while the higher measurements, which are less abundant, are seemingly harder to predict. From only this scatter plot the network appears to not perform well in general. In order to get a better impression of the overall performance we can take a look at the validation error. That is, the RMS loss across the entire validation set. A useful way of judging the validation error is by comparing the model to a very simple prediction. In our case, we will compare it with a prediction algorithm that claims that the most recent $PM_{2.5}$ value, the one at t_0, will be equal to the value at $t_0 + t_2$. If the neural network can at least predict better than that, then it should be useful in some capacity. We can compare the results of the neural network to this benchmark in the history plot:

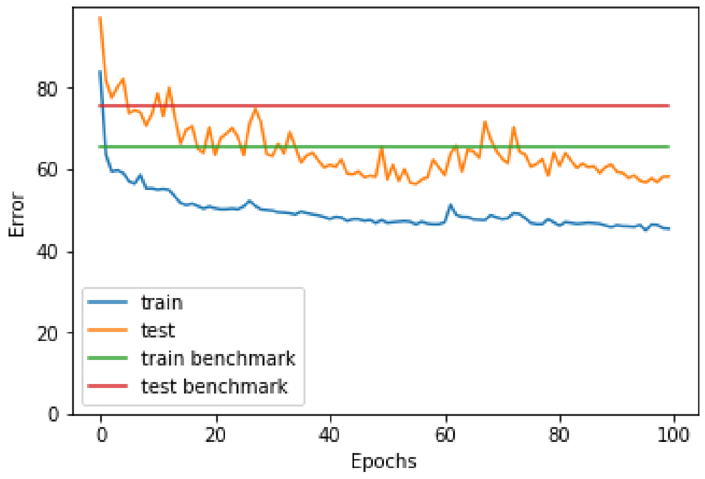

Figure 5.3: Value of the training and test error across the entire training period when compared to the benchmark set by taking $c(t_0 + t_2) = c(t_0)$.

In Figure 5.3, we can clearly see that the final algorithm does indeed perform much better than the benchmark algorithm, so in this regard the model training has gone well. However the issue still remains that in Figure 5.2 the network seems to struggle with the higher values. This might have something to do with the fact that there are so few high valued samples. This has been visualized in the following histogram:

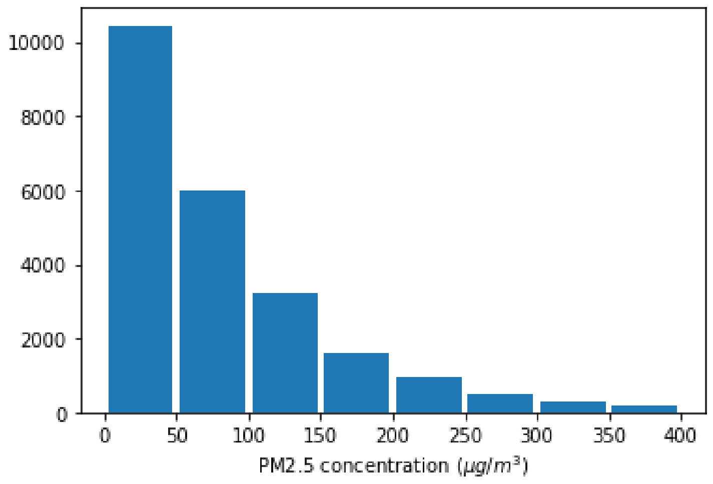

Figure 5.4: Histogram of the available $PM_{2.5}$ measurements for the period 2014 - 2016.

This figure depicts that as the concentration increases, the amount of available samples for training decreases exponentially. The problem with this is twofold. Not only does it mean that there might not enough examples for the LSTM network to pull from when these kind of samples come up during the validation phase, it also means that during training, the high samples will be vastly outnumbered. As mentioned in Section 2.4.1, the training happens in batches of 128 samples, meaning that if a batch only contains one or two higher valued samples the change in weights will be dominated by the smaller samples and as such the network will not learn how to predict these higher samples. However, the network is trained to minimize the RMSE and not to specifically perform well on these high valued samples, so in that sense the network still functions like it should. We will discuss some potential remedies for this problem later in Section 6.1.

While not as striking as the higher $PM_{2.5}$ concentration measurements, Figure 5.2 shows that there are also some lower values which the neural network has trouble with. In an effort to find out why we will look at the time series for some of these problematic points:

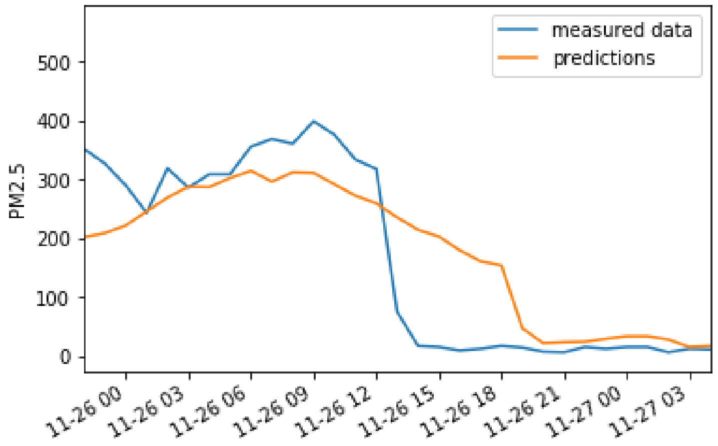

Figure 5.5: Time series plot showing the $PM_{2.5}$ concentration as measured by the measurement station and as predicted by the machine learning algorithm across the given time period. The time period has been chosen to display a few high overestimations of the algorithm.

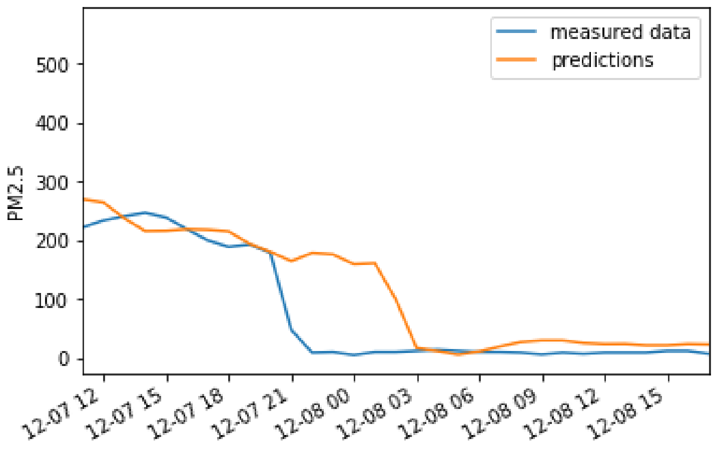

Figure 5.6: Time series plot showing the $PM_{2.5}$ concentration as measured by the measurement station and as predicted by the machine learning algorithm across a different time period. The time period is again chosen to display a few high overestimations of the algorithm.

From these plots it becomes clear that the measurement data, which is given by the blue line, experiences a very sharp drop-off. The orange line representing the neural network on the other hand, does not follows this trend until six hours later, which is the value of the time delay t_2 which we set earlier in Section 3.3. So it seems that the input parameters we give to the LSTM network are not enough to predict these kind of sudden drops, as we alluded to earlier in Section 3.4. There we hypothesized that including the simulation data might improve the ability of the neural network to predict the shape of the $PM_{2.5}$ concentration. We will look at those results in Section 5.5.

5.2 Different loss functions

As stated previously in Section 2.7, choosing different loss functions will result in different behaviours in a neural network. In particular, the expected behaviours for the Root Mean Squared Error (RMSE), the Mean Absolute Error (MAE) and the relative error were discussed. Here, in the interest of furthering our understanding of neural networks, we will compare the scatter plots corresponding to networks with different errors to see if these expectations hold water. Let us first consider the MAE:

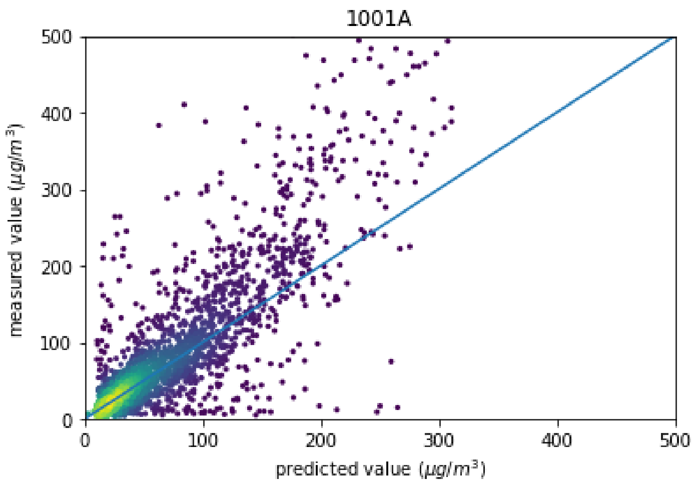

Figure 5.7: Scatter plot for a neural network that uses the Mean Absolute Error as its loss function.

In Section 2.7, we predicted that, compared to RMSE, the MAE would be more accurate when predicting smaller values, while performing worse on the high values. We concluded this from the fact that the RMSE penalizes very poor predictions more harshly. When comparing this scatter plot to Figure 5.2, we can see that for lower measured values, until around $100\mu g/m^3$, overall the points are a bit closer to the line $x = y$ for the MAE network. This does align with our expectations. When considering the higher values, we can see that while the points for the RMSE are more spread out, the MAE network does not seem to make prediction for above $300\mu g/m^3$. This is not surprising, as the machine learning algorithm has more of an incentive to not make grave errors when using the RMSE, so it has to try a bit harder for these values. These predictions do seem a little bit more accurate when eyeballing it, although the somewhat disappointing performance of the RMSE does make it hard to tell.

Next, we will consider the relative error:

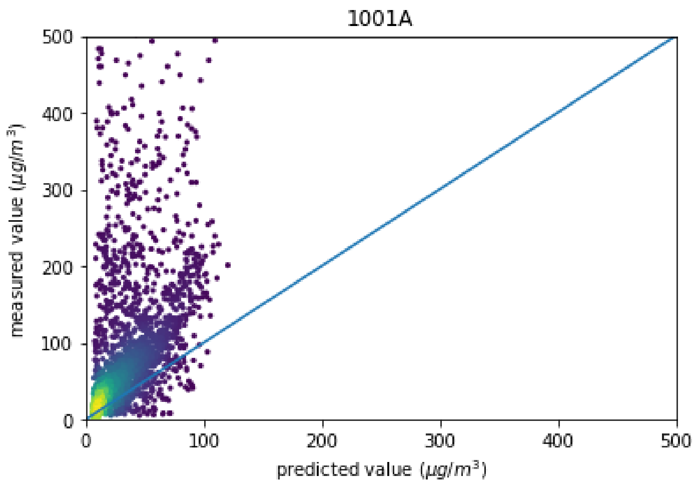

Figure 5.8: Scatter plot for a neural network that uses the relative error as its loss function.

At first glance, it looks as though something has gone wrong. However upon calculating the relative error for this network and the network trained using RMSE, we find that this network does perform better (the losses are 0.551 and 0.789, respectively). So in that regard the machine learning algorithm did its job. Nevertheless, from this scatter plot it seems that the relative error is not a usable loss function for our problem. While it does admittedly seem to perform quite well on the extremely low measurements, as the $PM_{2.5}$ concentration increases the network very quickly develops a tendency to underestimate. To a certain extend we expected similar behaviour, although it is surprising how quickly it seems to go wrong. This plot highlights that it is important to realize that the machine learning algorithm is only interested in its cost function. As such, it can sometimes forgo trying to learn specific, less impactful behaviours in favour of being more accurate for the more common and important samples.

A different way of getting a network with a more balanced performance is by turning the problem into a classification problem rather than a regression problem. This was done to great effect by [Xie, 2018], who among other things used LSTM for visibility predictions at Beijing Capital International Airport. In those experiments, the visibility levels are divided into three categories based on safety regulations. With this the weighted cross-entropy loss function was used, which adds a scaling parameter that increases the loss depending on which category the measurement is in. By using this technique, the neural network is forced to focus on learning specific measurements. This does mean that the network will not give a precise estimation, instead predicting an interval, but in the practical application that was not a problem. A classification problem results in a so-called confusion matrix, which is very similar to the scatter plots that we use except that it turns the group of points into a few separate bins instead.

In the opposite direction, we will consider what happens when we use a loss function that puts an even bigger emphasis on the large errors. For this we will take a loss function that is very similar to RMSE, but uses a fourth power instead of the square:

$$E_4 = \sqrt[4]{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y_i})^4} \tag{5.1}$$

Using this loss function to train a network, we can again find a scatter plot:

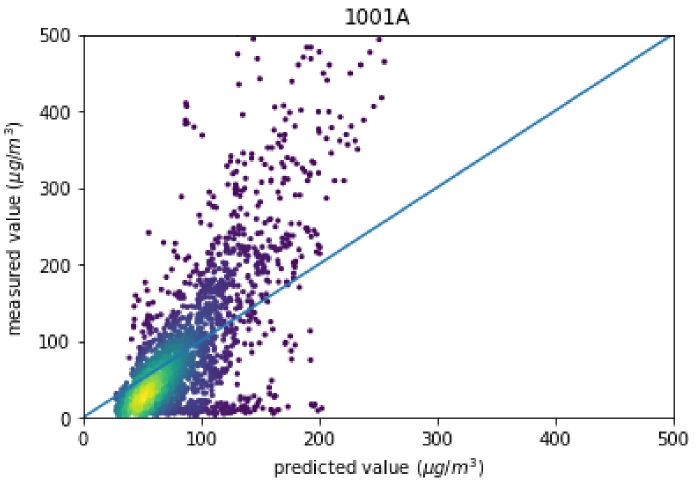

Figure 5.9: Scatter plot of a neural network that uses the fourth power loss as defined in Equation 5.1 as its loss function.

Compared to Figure 5.2, we can see that the lower blob of points is a lot further away from the y-axis. This nicely showcases the loss function's indifference to smaller errors. Furthermore, to the naked eye the measurements around the $150 - 250\mu g/m^3$ the points do on average seem a bit closer to $x = y$, as expected. As the measurements get even higher, however, the network once again seems to have quite a difficult time making accurate predictions. This further supports the claim that for those points the problem is mainly caused by the dataset rather than the neural network itself.

Overall, it seems that the RMSE is the most suitable for our application so that will be the error function that we use going forward.

5.3 Different activation functions

As mentioned previously in Section 2.6, we have considered two different activation functions for our problem. The results for the ReLU function were shown earlier in this chapter; the results for tanh will be presented here:

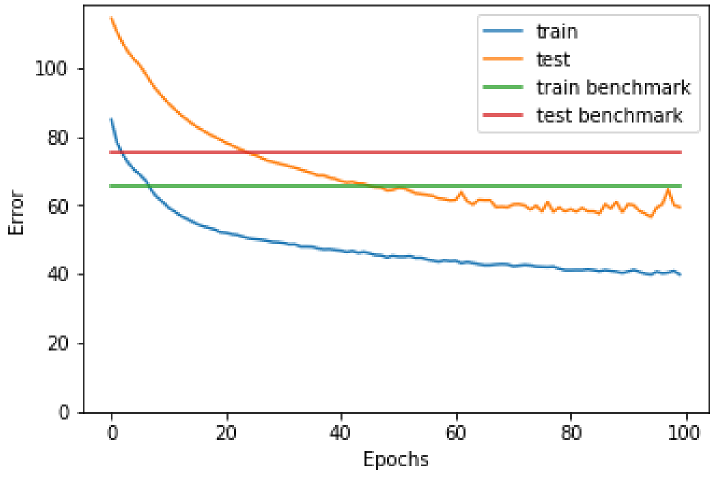

Figure 5.10: Value of the training and test error across the entire training period when compared to the benchmark set by taking $c(t_0 + t_2) = c(t_0)$. The network uses the tanh activation function.

When compared to the history graph for ReLU in Figure 5.3, we can see see that the tanh network learns a lot slower, but a lot more gradual. When looking at just the history, tanh seems like a good choice. Next, we will look at the scatter plot:

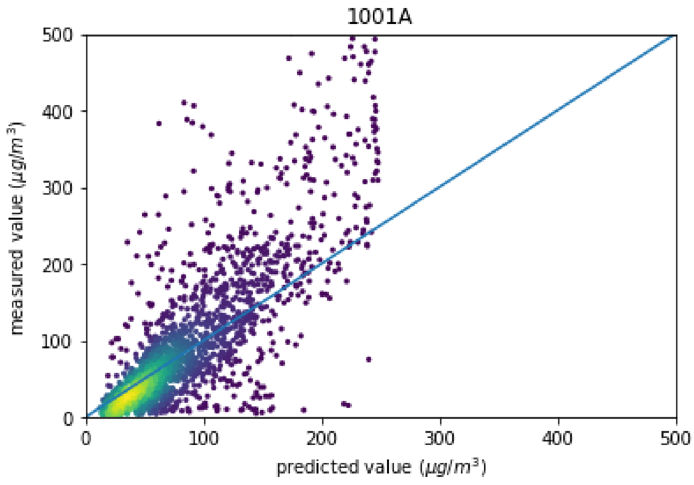

Figure 5.11: Scatter plot for a neural network that uses the tanh activation function for the hidden layers.

Here we can clearly see that the tanh network has a lot of trouble with the high values. In particular, it looks as though the network cannot make a prediction of $250\mu g/m^3$ or higher. This might be because, as mentioned in Section 2.6.1, a tanh neuron can get a bit stuck once it gets to a very high value. So what might be happening here is that a lot of the neurons responsible for handling the high measurements have reached this saturation point and are not able to differentiate between them anymore. This argument is further supported by the fact that ReLU does not have this problem.

However, the network that uses the ReLU activation function seems to have some issues as well, which might be caused by the activation function. In particular, the shape of the two curves in Figure 5.3 is not what you would typically expect from a neural network. The curves are typically very smooth, not bumpy as in Figure 5.3. There are two possible explanations for this. Firstly, as mentioned in Section 2.6.2, the activation function is unbounded which in practice means that during learning a particular batch the weights and bias of a neuron can change such that for most samples, the output of the neuron becomes too high which hinders the predictive capabilities of the network as a whole. A different explanation might be the dying ReLU problem, which we also mentioned in Section 2.6.2. In that case, the weights and bias of a neuron might change such that the activation of the neuron always becomes zero. If that happens to a neuron that is important for a lot of samples, it might take the network a few epochs to recover, possibly resulting in peaks in the test error like we can see Figure 5.3. Note that these are just hypotheses and that there might be a different cause. We will look at possible remedies later in Chapter 6.

Overall, it seems that ReLU is still the better option. While using that activation function does cause some problems, they are not completely detrimental to the capabilities of the neural network. The network that uses tanh, on the other hand, seems unable to handle certain samples, which might hinder our experiments.

5.4 Different learning rates

As mentioned previously in Section 2.4.2, one of the parameters that we can tweak in order to improve the performance of the neural network is the base learning rate α. In particular, changing the value of α will change how quickly the weights change during the learning process, particularly at the beginning. A learning rate that is too high might cause the algorithm to overshoot a minimum, causing the algorithm to potentially fail to find the solution altogether. A learning rate that is too low might get stuck in a less optimal local minimum. Here, we will study the effect of different values of α on the learning process by looking at training history plots so that we can hopefully find a fitting value for our experiments. We will look at three different values: the value that we use for the experiments, $\alpha = 0.0005$; the default value suggested by [Kingma and Ba, 2014], $\alpha = 0.001$ and an even lower value $\alpha = 0.00025$. The first value can be found in Figure 5.3 and the other two in Figures 5.12 and 5.13:

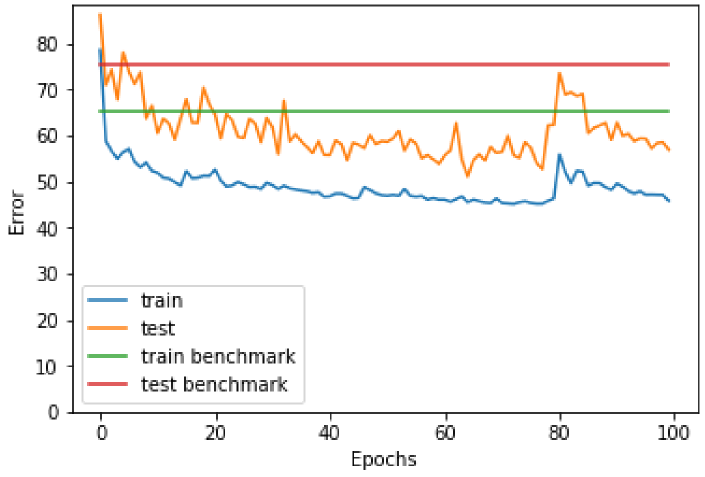

Figure 5.12: Value of the training and test error across the entire training period for a neural network with $\alpha = 0.001$.

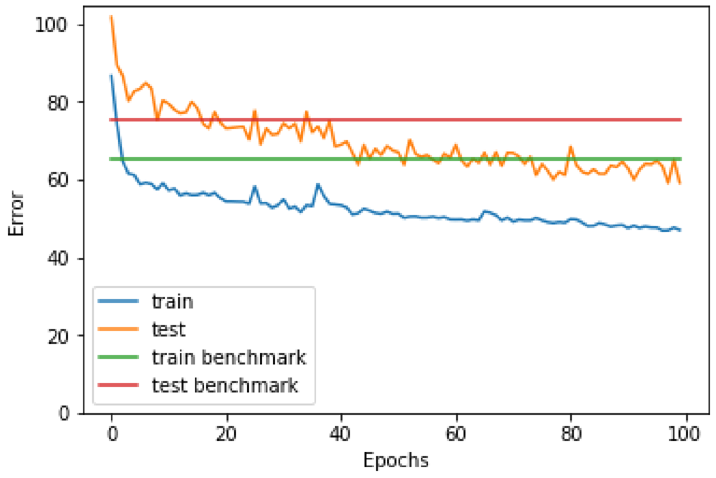

Figure 5.13: Value of the training and test error across the entire training period for a neural network with $\alpha = 0.00025$.

In Figure 5.12, we can see that both the training and test errors go down the fastest of the three, but we can see that especially the test error varies quite rapidly. This seems to imply that the algorithm has trouble converging to a local minimum. It can also cause a sudden high increase in the error. This trait is not desirable, as the algorithm can at times have a lot of trouble getting back to a minimum afterwards.

The error in Figure 5.13, on the other hand, does not vary nearly as much as the other two plots. However, we can also see that the errors decrease much more slowly. While this might eventually be resolved by learning for more epochs, this network might never end up performing as well as the other two because the Adam algorithm causes the learning rate to decay further, slowing things down even more.

For the other experiments throughout this thesis we will end up using $\alpha = 0.0005$ as it seems to be a good middle road between the different values. The learning rate is still high enough for it to find a satisfying minimum, while the magnitude of the variations appears to be tolerable.

5.5 Including the simulation data

In this thesis, we are trying to see if adding simulation data from a CTM as input of a neural network could improve its performance. At the end of Section 5.1, we found that without the simulation data our LSTM network had trouble predicting a sudden decrease in $PM_{2.5}$ concentration. We discussed that this might be a possibility earlier in Section 3.4, where we hypothesized that adding the simulation data to the input matrix could help mitigate this issue.

To check this hypothesis, we will train a new network, this time including the simulation data as input. First, we will consider the scatter plot again:

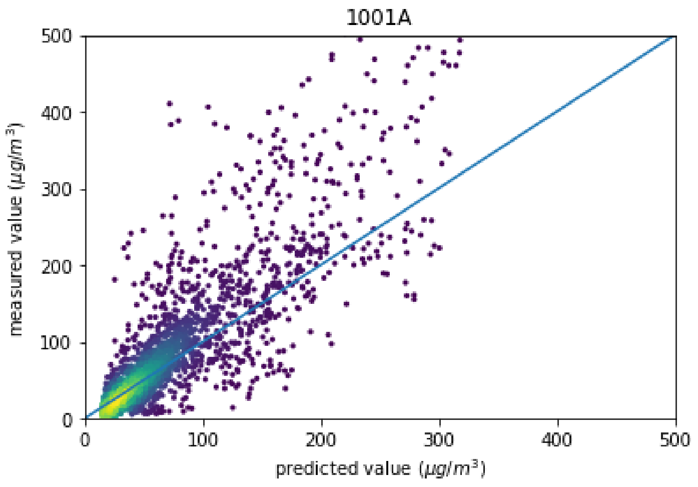

Figure 5.14: Scatter plot for a neural network whose input parameters include the simulation data.

When contrasting this plot with Figure 5.2 a few observations can be made. First of all, we can see that the big group of data points in the bottom left of the graph does not line up quite as well with the reference line as it did for the experiments that do not use simulation data. We can also see that this group is removed a short distance from the y-axis, indicating that the neural network refuses to predict a value below a certain threshold. (This behavior is also present when looking at Figure 5.2 except the limit is much lower). This strange quirk seems to be an unfortunate side-effect of training the network using the Root Mean Squared Error, as mentioned earlier in Section 5.2. After all predicting these very low values better would only marginally improve the total error, while it would increase the error by a lot for the points that it grossly underestimates. So the network has to balance these two problems out during training and as a result it seems to impose a limit on how low the concentration can be. Either way practically this behaviour is not problematic because in the real world the difference between the predicted and measured concentration would be negligible.

Overall, it seems from the scatter plots, that the inclusion of simulation data does improve things, if only slightly. This is easier to analyze by considering history plot below:

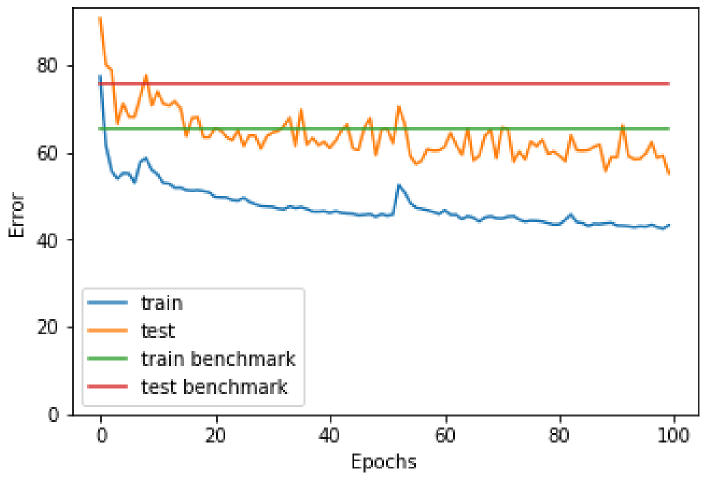

Figure 5.15: Value of the training and test error across the entire training period when compared to the benchmark set by taking $c(t_0 + t_2) = c(t_0)$. The input parameters of the network include the simulation data.

Comparing this history plot to the history plot from the network without simulations (Figure 5.3) we can see that the validation errors at the very end of the training periods are very close. Considering how much the error tends to vary between epochs, we cannot to draw any further conclusions for now. Instead, one would need to repeat the experiment many times and compare for instance the average final RMS loss to see which is better. We will not do that for now but when we consider all the different stations in Section 5.7 we have a lot more samples to compare and hopefully a clear trend will appear.

Most importantly, the experiment with simulation data does not have as many points in the bottom right of the scatter plot when compared to the experiment without the simulation data. As mentioned earlier in Section 3.4, we hypothesized that using the simulation data would help the neural network at predicting when the $PM_{2.5}$ concentration suddenly drops. As we saw in Figures 5.5 and 5.6 the network has a hard time with these predictions without the simulation data, which was the cause of these points in the bottom right area of the scatter plot. In order to look at these two periods in more detail, consider the following two plots:

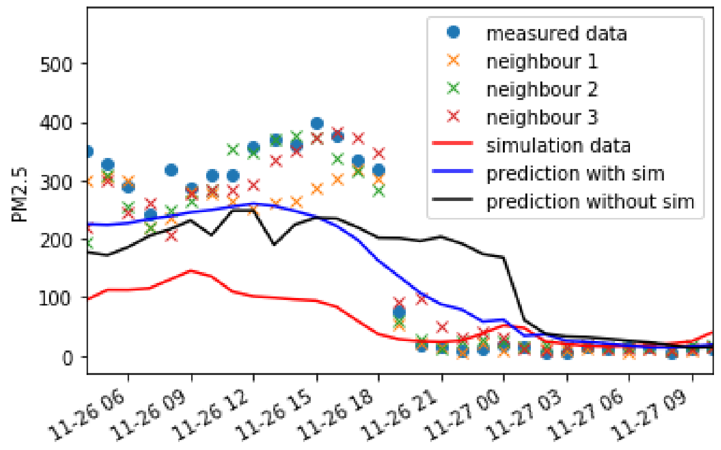

Figure 5.16: Time series plot showing the $PM_{2.5}$ concentration of measurements from the main station as well as three neighbouring stations, simulation results and predictions from networks with and without using the simulation data. The time period corresponds to the one in Figure 5.5.

In this first time series plot, we also include the neighbours, the simulation data from the LOTOS-EUROS model and both predictions. First, when comparing the simulation data to the measured data we can get a nice example of its behaviour described in 3.4. It has a clear problem with underestimating the $PM_{2.5}$ concentration when it is moderately high, but it does seem to be able to predict when the actual concentration might start dropping. If we then look at the prediction with simulations, we can see that while the simulation data is at its highest, it stays close to the prediction without. But as the simulation data starts to decrease the prediction starts to decrease as well. As a result prediction with simulation data does not overestimate these points nearly as much.

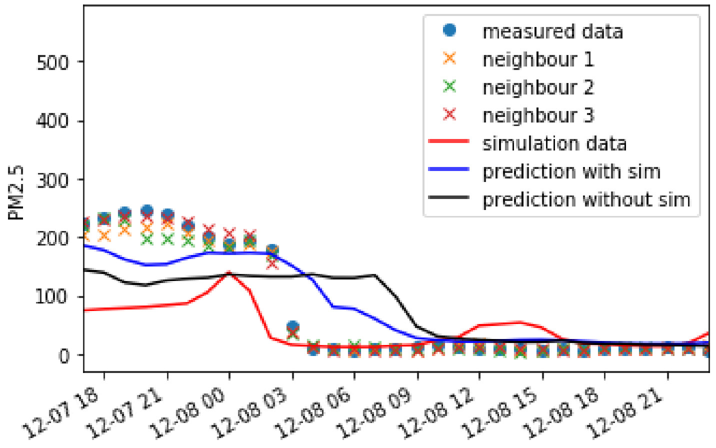

Figure 5.17: Another time series plot showing the $PM_{2.5}$ concentration of measurements from the main station, three neighbouring stations, simulation results and predictions from networks with and without using the simulation data. For this plot the time period corresponds to the one in Figure 5.6 instead.

Here, we can see that the simulation data first increases and then drops off a lot more abruptly. While the time of the drop off seems very accurate, the prediction with simulations seems to respond a little more slowly. Nevertheless, the neural network that uses simulation data does clearly outperform the other network for this sample as well.

5.6 LSTM vs FNN

In Section 2.2, we argued that for our time series problem a simple Feedforward Neural Network might not be sufficient. After all, a neural network cannot directly tell from the input parameters that we use measurements from the same parameters for a number time steps. So we considered that it might be useful to use an LSTM network which can create a sort of memory of past measurements via its feedback loop. However, it turns out that for some other time series experiments, the FNN performs better than LSTM. For example, [Lago et al., 2018] use neural networks to forecast spot electricity prices and they found that DNN, a variation of FNN that uses many hidden layers, performs better than LSTM. This might be possible because choosing LSTM over FNN comes with a trade-off. LSTM cells do provide extra potential capabilities, but that also means that the neural network has to learn a more complicated model, which might get in the way if the best solution turns out to be quite simple.

In order to find out if this trade-off is worth it for our $PM_{2.5}$ predictions we have repeated our experiments, both with and without simulation data, using basic neurons instead of LSTM cells in our neural network. First, for the experiment without simulation data as input, we have:

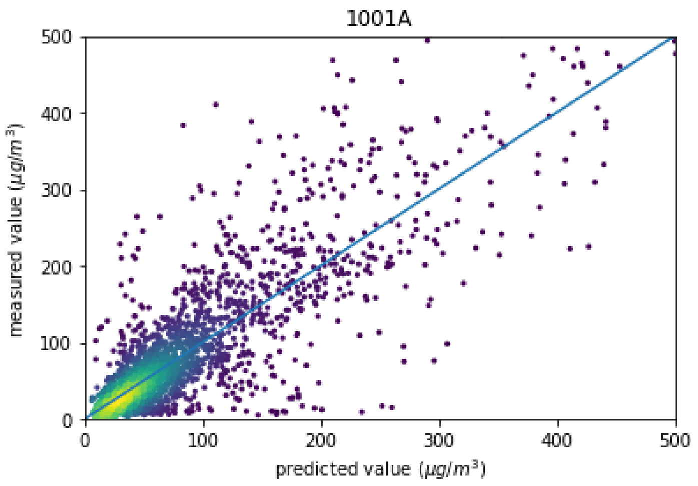

Figure 5.18: Scatter plot for a network that is an FNN instead of a LSTM neural network.

Comparing this result to the scatter plot found for LSTM (Figure 5.2), the FNN seems to be much better than the LSTM network at predicting the very high values. For the other values, the FNN and LSTM network perform similarly. Comparing the loss function of the two experiments, we find that LSTM has an RMS loss of approximately 55.0 and FNN has an RMS loss of 50.5. This is a significant difference. This implies that, for this particular station at least, using FNN is better than using LSTM. This might only be caused by the limited dataset, as the more complex LSTM network has more weights and therefore requires more data to learn. So before drawing any conclusions, we will look at the group of multiple stations, later in Section 5.7. After all, most other stations have a much smaller range of $PM_{2.5}$ concentrations so the amount of training samples should not be much of an issue for those stations. Next, for the experiment with simulation data as input, we find the result given by Figure 5.19.

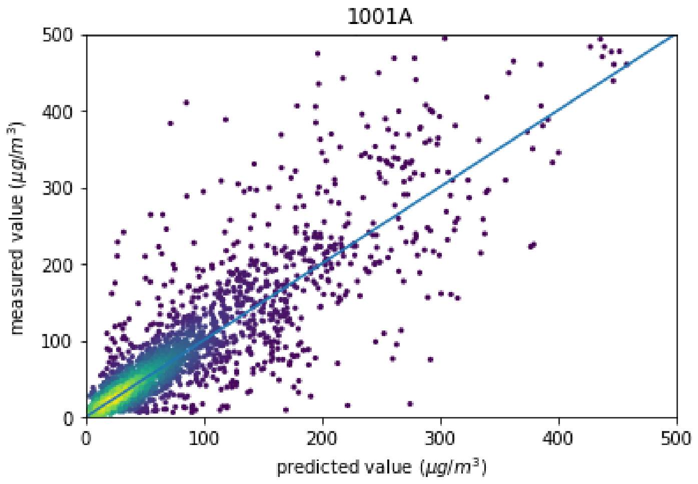

Figure 5.19: Scatter plot for a network that is an FNN instead of an LSTM neural network. This network also uses simulation data as input.

For LSTM, the most notable result when including the simulation data was that the network was much better at not making large overestimations. For FNN, this does not seem to be the case. This might be due to the fact that the FNN generally makes higher estimations for high values when compared to the LSTM network, which also means that there are more opportunities for the network to make a mistake. Either way, we will again hold off on drawing any conclusions until after we have looked at the other stations.

5.7 Large scale experiments

As mentioned earlier, we want to consider neural networks for all stations in Figure 3.1. Primarily in order to analyze the effect of the CTM data on the neural network under different circumstances. But since the performance of our ReLU network for LSTM seems to vary a lot it might be useful to consider how the two different inputs affect performance between many different networks.

In Section 3.4, we hypothesized that adding CTM data might alleviate a common problem for LSTM when applied to time series, that being having a time delay between the measurement data and the predictions of the LSTM. In Figures 5.5 and 5.6 we found two nice examples of this problem, which in the scatter plot is expressed by a group of points in the bottom right of the graph. However, it is important to note that this time delay problem is not the only possible cause for these kinds of points. After all, the network might at times have learned some incorrect behaviour from the training data. Or the sample might have been difficult to predict due to some other source unrelated to any of the input parameters.

Furthermore, these are not the only errors caused by the time delay problem. Gross underestimations could also be caused by the $PM_{2.5}$ concentrations suddenly spiking with the LSTM network needing time to realize that this has happened. The primary difference for us is that there are a lot more underestimations than overestimations. Also, when looking closely at some of these points, it appears that in these situations the LSTM network simply struggles in general rather than due to this time delay issue. So while these errors are still present for high valued samples, it is a lot harder to spot them from just the scatter plots. So going forward it is important to keep in mind that, while we will be using the amount of underestimations as way to quantify the time delay problem, this is not a perfect metric by any means and not the only symptom of this problem. However it is a very convenient metric and by looking at time series plots we can also see for ourselves the difference in performance.

First, we will take an extensive look at the results for the experiments that use LSTM and then we will look at the results using FNN to see if those networks still perform better.

5.7.1 LSTM

First, we will look at the scatter plots for all 25 different stations. These can be found in Appendix A.1 in Figure A.1. Looking at all these different locations, a few things become apparent. First we can see that the air quality at station 1001A is indeed very poor compared to other areas. So while the location does have a problem with a low amount of data, improving the predictions does seem more important for that particular station. Furthermore, a lot of stations seem to have very few predictions that are wrong by a lot, so there is not a very big need for small improvements to the machine learning process.

That being said, there are still a couple of stations that exhibit the distinct group of points that are overestimated by a lot in the no simulation scatter plots. In particular, 1021A, 1043A, 1083A and to a lesser extend 1582A all seem to have the same problem as 1001A. For these stations, including simulation data as input for the neural network causes an improvement in this area. Surprisingly, for station 1103A, the simulation data actually seems to have the opposite effect in that it adds several gross overestimations instead. Some of these points can be seen in the following time series plot:

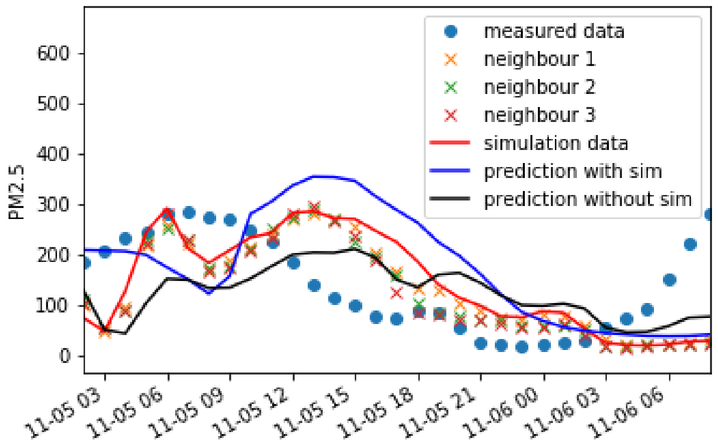

Figure 5.20: Time series plot for station 1103A showing the $PM_{2.5}$ concentration of measurements from the main station as well as three neighbouring stations, simulation results and predictions from networks with and without using the simulation data.

Here, we can see that the simulation data more closely follows the neighbouring stations instead of the main station. This is probably because, as mentioned in Section 3.4, the results produced by LOTOS-EUROS are on a grid that does not necessarily align with the locations of the measurement stations. So instead we use the data from the nearest grid point, which in this case appears to be very close to these neighbouring stations. In general this should not be an issue as the distance between the nearest stations are quite small (in this case they are approximately 4, 5 and 8 kilometers away) so the simulation data should still be indicative of the $PM_{2.5}$ profile at the main station. Here it is not the case unfortunately and as the neural network generally follows the profile of the simulation as we saw in Section 5.5, the network that uses the simulation performs worse than the one that does not.

By adding the results of all these stations we are hoping to be able to draw some general conclusions. After all this is difficult to do using just one station given how much the dataset can influence the behavior of the network and the random nature of mini batch gradient descent. For this we will consider two different metrics. First we will simply look if the loss function has decreased for each station. Then we will try to more precisely quantify how much and how often the networks make large overestimations.

Looking at the errors in Appendix A.3 (Table A.1) reveals a few things. Most notably, the final RMS loss is lower for the network with simulations as input for all stations except 1266A. So the inclusion of simulation data almost always seems to lead to improvement. It does seem as though the difference is often very small, especially for networks that can already make fairly accurate predictions without the extra data. Furthermore, for some stations the difference is so small that the stochastic nature of machine learning might have made the difference. Still, an improvement for 24 out of 25 stations strongly suggest that the inclusion of simulation data to the input of a neural network improves the network in terms of minimizing its loss function.

Earlier we stated that we expect that including the CTM results will help the network with predicting the profile of the $PM_{2.5}$ concentration, which is especially notable when the concentration suddenly decreases to a very low amount, If the network has trouble with such a sample, the prediction ends up being a big overestimation. We have seen in Section 5.5 that there is an improvement, but in order to draw a general conclusion we will consider all 25 stations. In particular, we want to both consider how often the measurements are overestimated and by how much. For this, we will use the following table:

Table 5.2: Number of overestimations across all 25 stations. The rows indicate whether the simulation data was included or not and the columns indicate by how much the prediction is off.

	$150\mu g/m^3$	$100\mu g/m^3$
sim	10	71
no sim	26	129

These numbers are across the test periods for every station. These periods are a little under half a year each assuming none of the data is missing, but in practice they will be shorter still. The number of overestimations seems very insignificant either way, even if you account for the fact that only four or five of the stations even have this kind of problem. But we do have to keep in mind that, as we discussed at the beginning of this section, we are not specifically interested in combating overestimations. These points are just an easy way to identify the main problem, a time delay in the predictions of an LSTM network, which can occur for a lot of different samples. So in that regard the difference between simulation and no simulation is much more important. In Table 5.2 we can see that for the first category the amount of points has gone down by about 62% and the for the second the decrease is approximately 45%. This result strongly supports the hypothesis that including simulation data as input data helps the LSTM network when predicting sudden changes.

5.7.2 FNN

As mentioned in Section 5.6, we will also repeat these experiments for all 25 stations using FNN. The results can be found in Appendix A.2 (Figure A.2). When we looked at only the station 1001A, we found that FNN performed better than LSTM, both in terms of RMS loss and particularly when predicting very high values. This trend seems to hold for most of the other stations as well, although the difference in RMS loss is very small for the stations in very clean areas. This strongly suggests that FNN is the better option for our specific problem.

Looking at the use of simulation data when using FNN, we noticed for station 1001A that the RMS loss does indeed improve slightly but, unlike our findings for LSTM, the simulation data did not seem to help for big overestimations. Looking at the overall results for the 25 stations, we notice that overall using simulation data does in fact help in this area. This is displayed in the following table:

Table 5.3: Number of overestimations across all 25 stations using FNN. The rows indicate whether the simulation data was included or not and the columns indicate by how much the prediction is off.

	$150\mu g/m^3$	$100\mu g/m^3$
sim	32	105
no sim	68	279

Here we can clearly see that aside from the RMSE, adding simulation data also helps combat overestimations. Compared to Table 5.2, we can see that overestimation is a lot more common for FNN than for LSTM. As mentioned in Section 5.6 this is not too surprising as the networks make higher estimates for high measurements, which means that when those predictions are overestimates, they overestimate by a lot more.

Chapter 6

Conclusions

In this thesis we have made predictions of $PM_{2.5}$ concentrations six hours in advance using neural networks. For this we mostly used an LSTM network, although we did some experiments using FNN as well. As stated in Chapter 1, the main research question is:

In what sense does the inclusion of simulation data improve the $PM_{2.5}$ predictions of a neural network, if at all?

We can conclude that the inclusion of simulation data does improve the predictive capabilities of the neural networks. As tested on a group of 25 air quality measuring stations, including the simulation data lead to a lower overall loss across the validation set for every station except one. We hypothesized that the simulation data would be especially helpful in a particular problem of LSTM, where the network displays a delay when the measurements that the network tries to predict exhibit a rapid change. This problem is particularly noticeable when looking at large overestimations of the neural network. Counting all samples overestimated by $150\mu g/m^3$ and $100\mu g/m^3$ across all 25 stations, we found that the amount of points in the first group goes down by roughly 62% and the amount of points in the second by 45%. It appears that the simulation is useful for this purpose because the simulation data can be produced far in advance, which means that the neural network can use data from the time that it wants to predict as well.

Furthermore, we found that contrary to our initial assumption, the LSTM neural network performs worse than the much more simple FNN, both in terms of its loss function and more specifically in the ability of the neural network to accurately predict higher $PM_{2.5}$ measurements. We think that the most likely reason for that is the fact that the training set used by the neural network does not have a lot of samples. As a result, the network with less weights, the FNN, is able to learn better.

6.1 Recommendations and future work

From the results presented in this these we can obviously conclude that it is beneficial to include simulation data to your input matrix and that it is better to use FNN over LSTM. Aside from that, we have spent little time in this thesis on fine tuning some of the neural network hyperparameters.

So if one were to do another experiment focused on the values of predictions rather than for studying neural networks, one might be able to find some improvements by tweaking the number of nodes, dropout probability, etc. Also, while we did briefly look at different learning rates and different activation functions, those experiments were done under the assumption that LSTM was the better choice. As this assumption proved to be wrong, the conclusions made in those two sections might not hold anymore either.

Another aspect of the machine learning algorithm that can still be tweaked is the preprocessing. In Chapter 4, we argued that whenever there is a long period of time where data from one of the aerosols was missing, we should discard the sample entirely. Now that we have done all of our experiments one cannot help but wonder if the criteria for discarding were too strict. Especially since a low amount of data in the training set has been a thorn in our side throughout our discussion of the results.

In Section 5.3, we found that using ReLU as the activation seemed to produce better results than the tanh activation function. However, there were still some problems in the history plot which we think might be caused by the activation function. In particular, we mentioned the activation function being unbounded and the dying ReLU problem. In the paper that first introduces the ReLU activation function [Glorot et al., 2011], the authors actually address the unbounded behaviour by using L1-regulation. This regulation technique limits how much the weights can change in one epoch, which should keep the activations from becoming too high if that is problematic for other batches. When combating the dying ReLU problem, there are a few slight variations on the ReLU activation functions which have a small gradient instead of a flat zero for $x < 0$ such as Leaky ReLU [Karpathy et al., 2016] or ELU [Clevert et al., 2015]. This small gradient allows a neuron that would be dead when using regular ReLU to recover, because the small gradient means that the neuron still has a small output which allows the gradient descent algorithm to still change the weights corresponding to the neuron.

Throughout the discussion of our results we have found that we may not have enough samples in our dataset. In particular, we found that the neural networks performed poorly on samples of a high concentration. There are several ways to at least partially remedy this problem. One of these possible solutions is by using weighted cross-entropy [Xie, 2018]. With this loss function, you separate the samples into categories based on the $PM_{2.5}$ concentration of the target measurement. These different categories each have a scaling parameter attached to them which you can set yourself which allows you to penalize the network more harshly depending on the category of the measurement. By setting the penalty for high values very high, we can force the network to pay more attention to those samples during training, which should lead to a better performance on the higher samples in the validation set.

Another technique for improving performance on rare scenarios is by resampling. When resampling, we duplicate the high concentration samples in the training set. While this technically does not add any new information for the network to learn, it does mean that the high samples are not outnumbered as much by the low samples, meaning that the machine learning algorithm will pay more attention to those during learning.

At the start of Section 5.7, we argued that counting the number of high overestimations was a convenient metric representative of the time delay problem exhibited by LSTM. After seeing the low amount of points in Table 5.2, however, one has to wonder if there is maybe a better metric that we could use. After all, making a decision based on convenience does imply the possibility of a better, less convenient alternative. Unfortunately, trying to find an alternative involves looking closely at a lot of time series plot in order to find other distinct points where the time delay problem is clearly present. Or alternatively you could select all points where the $PM_{2.5}$ concentration changes by a lot in a short time span and look at the performance of the network on only those points.

Finally, considering the improvements made by including the PM simulation data, it might be a good idea to look into including simulation data for other parameters as well. Some of the other aerosols such as O_3 have also been modeled using LOTOS-EUROS [Manders-Groot et al., 2016]. Simulations of meteorological data should also be available in the form of weather predictions. While these likely will not have as much of an impact as the $PM_{2.5}$ simulations, adding them to the input matrix of the neural network could still improve the performance of the network a little bit.

Bibliography

[Barbu et al., 2009] Barbu, A., Segers, A., Schaap, M., Heemink, A., and Builtjes, P. (2009). A multi-component data assimilation experiment directed to sulphur dioxide and sulphate over Europe. *Atmospheric Environment*, 43(9):1622 – 1631.

[Clevert et al., 2015] Clevert, D.-A., Unterthiner, T., and Hochreiter, S. (2015). Fast and accurate deep network learning by exponential linear units (ELUs). *arXiv preprint arXiv:1511.07289.*

[Deng et al., 2019] Deng, T., Cheng, A., Han, W., and Lin, H.-X. (2019). Visibility forecast for airport operations by LSTM neural networks. *Proc. ICAART.*

[Glorot et al., 2011] Glorot, X., Bordes, A., and Bengio, Y. (2011). Deep sparse rectifier neural networks. In Gordon, G., Dunson, D., and Dudík, M., editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *Proceedings of Machine Learning Research*, pages 315–323, Fort Lauderdale, FL, USA. PMLR.

[Goodfellow et al., 2016] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

[Gulcehre et al., 2016] Gulcehre, C., Moczulski, M., Denil, M., and Bengio, Y. (2016). Noisy activation functions. In *International conference on machine learning*, pages 3059–3068.

[Hochreiter and Schmidhuber, 1997] Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.

[Jin, 2019] Jin, J. (2019). Dust storm emission inversion using data assimilation. https://doi.org/10.4233/uuid:c90cefff-2a6e-42f0-9893-a1c2cc11f1ac.

[Jin et al., 2019] Jin, J., Lin, H. X., Segers, A., Xie, Y., and Heemink, A. (2019). Machine learning for observation bias correction with application to dust storm data assimilation. *Atmospheric Chemistry and Physics*, 19(15):10009–10026.

[Karpathy et al., 2016] Karpathy, A. et al. (2016). Cs231n convolutional neural networks for visual recognition. *Neural networks*, 1.

[Keskar et al., 2016] Keskar, N. S., Mudigere, D., Nocedal, J., Smelyanskiy, M., and Tang, P. T. P. (2016). On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836.*

[Kingma and Ba, 2014] Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980.*

[Lago et al., 2018] Lago, J., De Ridder, F., and De Schutter, B. (2018). Forecasting spot electricity prices: Deep learning approaches and empirical comparison of traditional algorithms. *Applied Energy*, 221:386–405.

[Manders et al., 2017] Manders, A. M. M., Builtjes, P. J. H., Curier, L., Denier van der Gon, H. A. C., Hendriks, C., Jonkers, S., Kranenburg, R., Kuenen, J. J. P., Segers, A. J., Timmermans, R. M. A., Visschedijk, A. J. H., Wichink Kruit, R. J., van Pul, W. A. J., Sauter, F. J., van der Swaluw, E., Swart, D., Douros, J., Eskes, H., van Meijgaard, E., van Ulft, B., van Velthoven, P., Banzhaf, S., Mues, A. C., Stern, R., Fu, G., Lu, S., Heemink, A., van Velzen, N., and Schaap, M. (2017). Curriculum vitae of the LOTOS–EUROS (v2.0) chemistry transport model.

[Manders-Groot et al., 2016] Manders-Groot, A., Segers, A., Jonkers, S., Schaap, M., Timmermans, R., Hendriks, C., Sauter, F., Kruit, R. W., van der Swaluw, E., Eskes, H., et al. (2016). LOTOS-EUROS v2.0 reference guide. *TNO report TNO2016*, 10898.

[Olah, 2015] Olah, C. (2015). Understanding LSTM networks. https://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[Schmidhuber, 2015] Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural networks*, 61:85–117.

[Srivastava et al., 2014] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958.

[Timmermans et al., 2017] Timmermans, R., Kranenburg, R., Manders, A., Hendriks, C., Segers, A., Dammers, E., Zhang, Q., Wang, L., Liu, Z., Zeng, L., van der Gon, H. D., and Schaap, M. (2017). Source apportionment of PM2.5 across China using LOTOS-EUROS. *Atmospheric Environment*, 164:370 – 386.

[Wang and Hao, 2012] Wang, S. and Hao, J. (2012). Air quality management in China: Issues, challenges, and options. *Journal of Environmental Sciences*, 24(1):2–13.

[Xie, 2018] Xie, Y. (2018). Deep learning architectures for PM2.5 and visibility predictions. https://repository.tudelft.nl/islandora/object/uuid:c7ea9e3e-d6c2-426c-b4d5-8c7143052257?collection=education.

[Yu-Fei Xing and Lian, 2016] Yu-Fei Xing, Yue-Hua Xu, M.-H. S. and Lian, Y.-X. (2016). The impact of PM2.5 on the human respiratory system. *Journal of thoracic disease*, 8(1):E69–74.

[Zaremba et al., 2014] Zaremba, W., Sutskever, I., and Vinyals, O. (2014). Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*.

Appendix A

Additional results

In this appendix, we will be displaying some of the results found in Section 5.7. First there are two sets of scatter plots, one for LSTM and one for FNN, showing the results for all 25 different stations with and without including the simulation data as input. After that is a table displaying the RMS loss of all the experiments in this appendix for easy comparison.

A.1 LSTM scatter plots

Here, we will plot the scatter plots for $PM_{2.5}$ predictions done at 25 different stations using LSTM. The input matrices for the plots on the right also include simulation data, while the input matrices for the plots on the left do not.

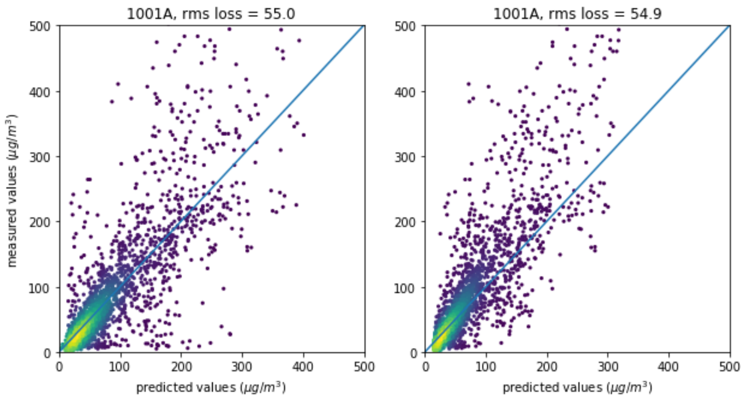

Figure A.1: Scatter plots for all 25 stations using LSTM. The plots on the right use simulation data for training, the plots on the left do not.

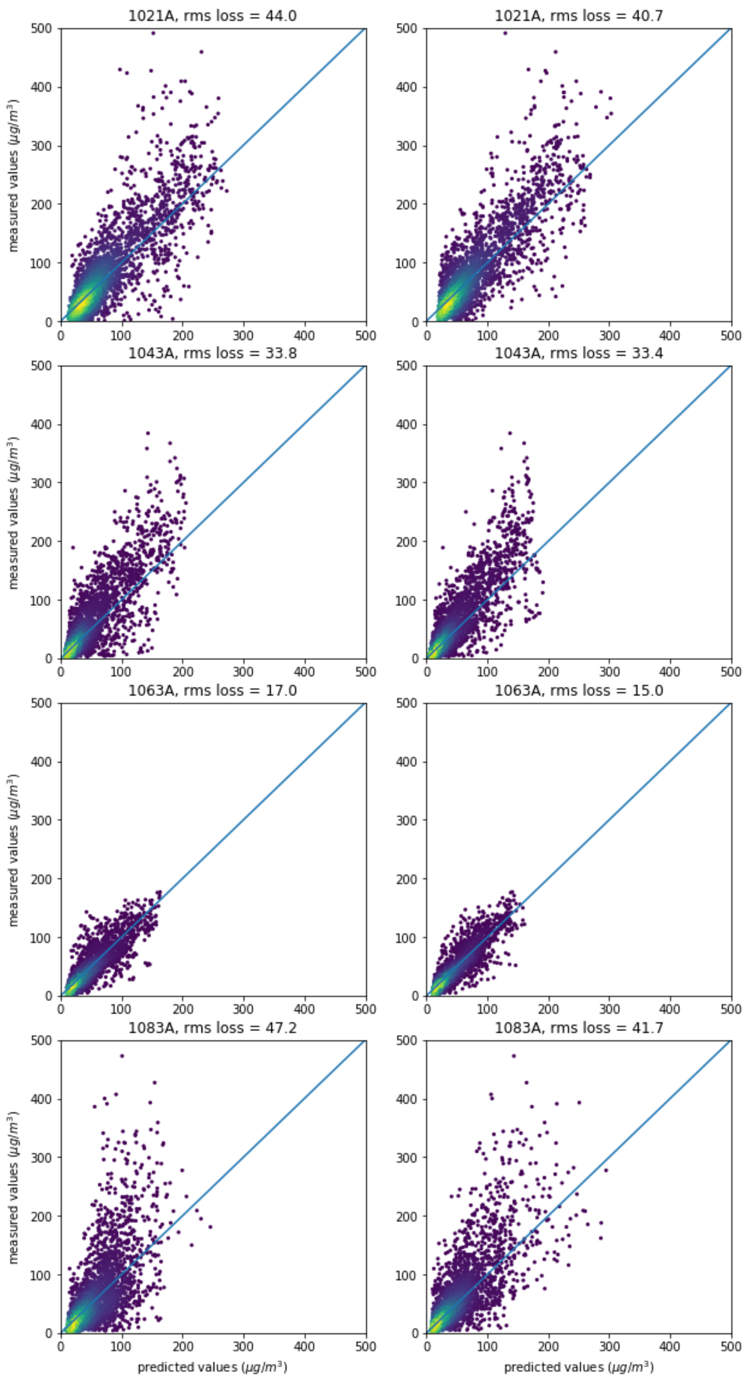

Figure A.1: Scatter plots for all 25 stations using LSTM. The plots on the right use simulation data for training, the plots on the left do not. (continued)

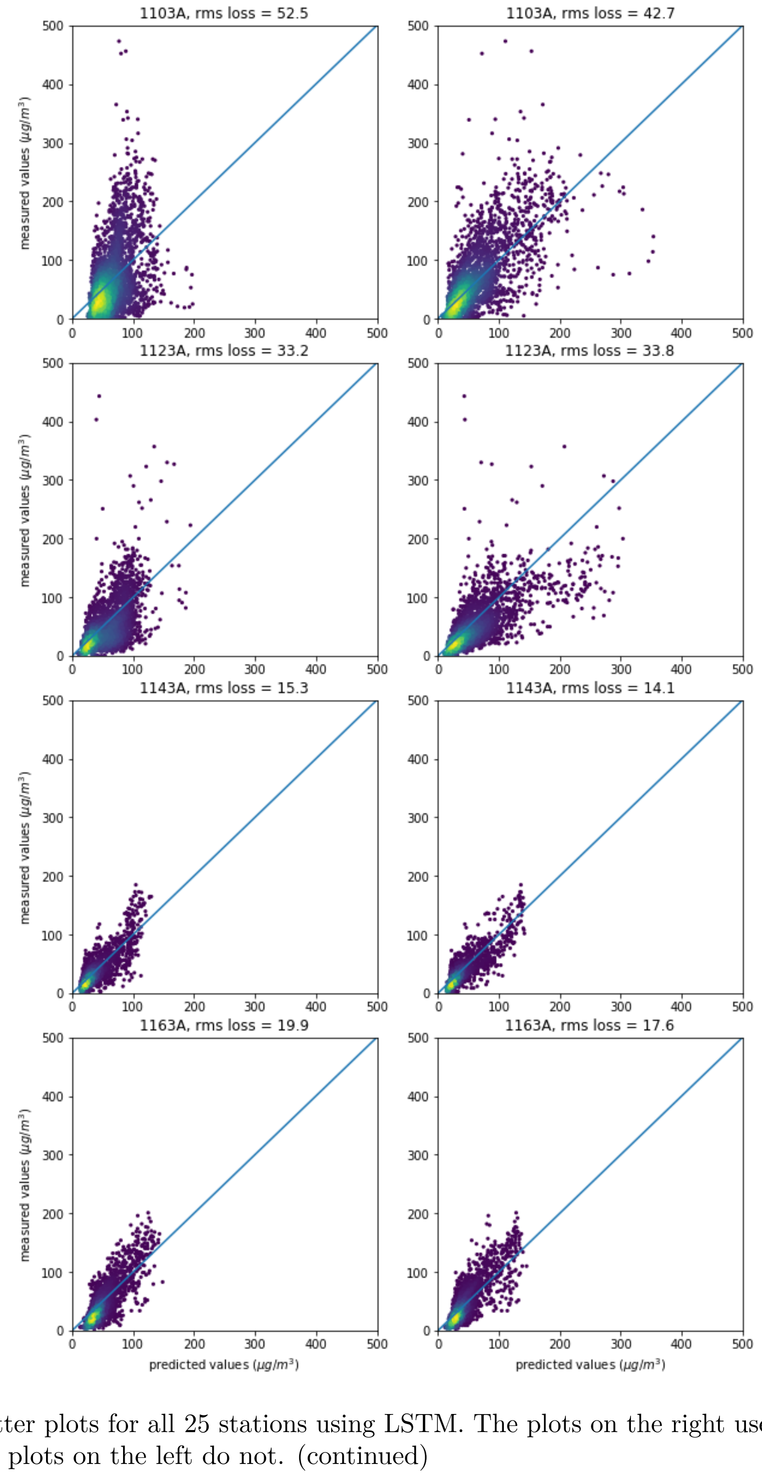

Figure A.1: Scatter plots for all 25 stations using LSTM. The plots on the right use simulation data for training, the plots on the left do not. (continued)

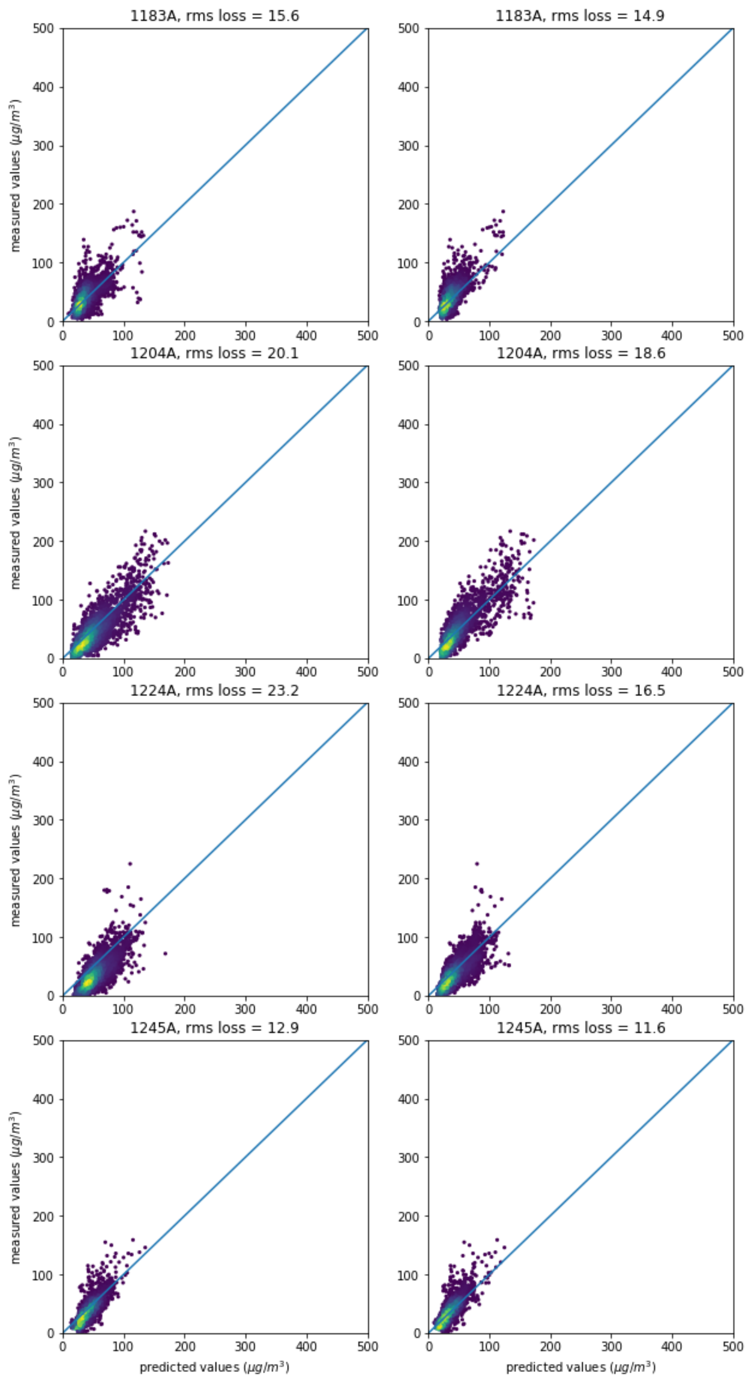

Figure A.1: Scatter plots for all 25 stations using LSTM. The plots on the right use simulation data for training, the plots on the left do not. (continued)

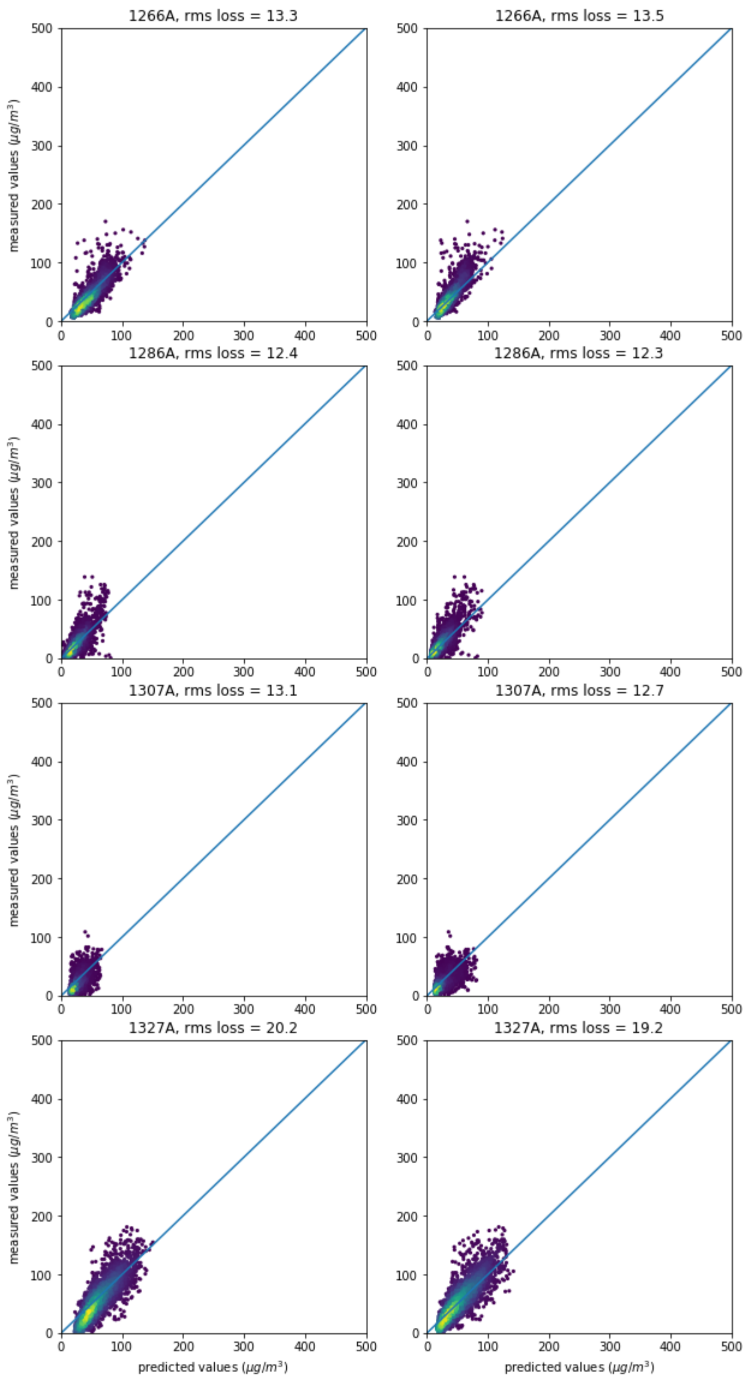

Figure A.1: Scatter plots for all 25 stations using LSTM. The plots on the right use simulation data for training, the plots on the left do not. (continued)

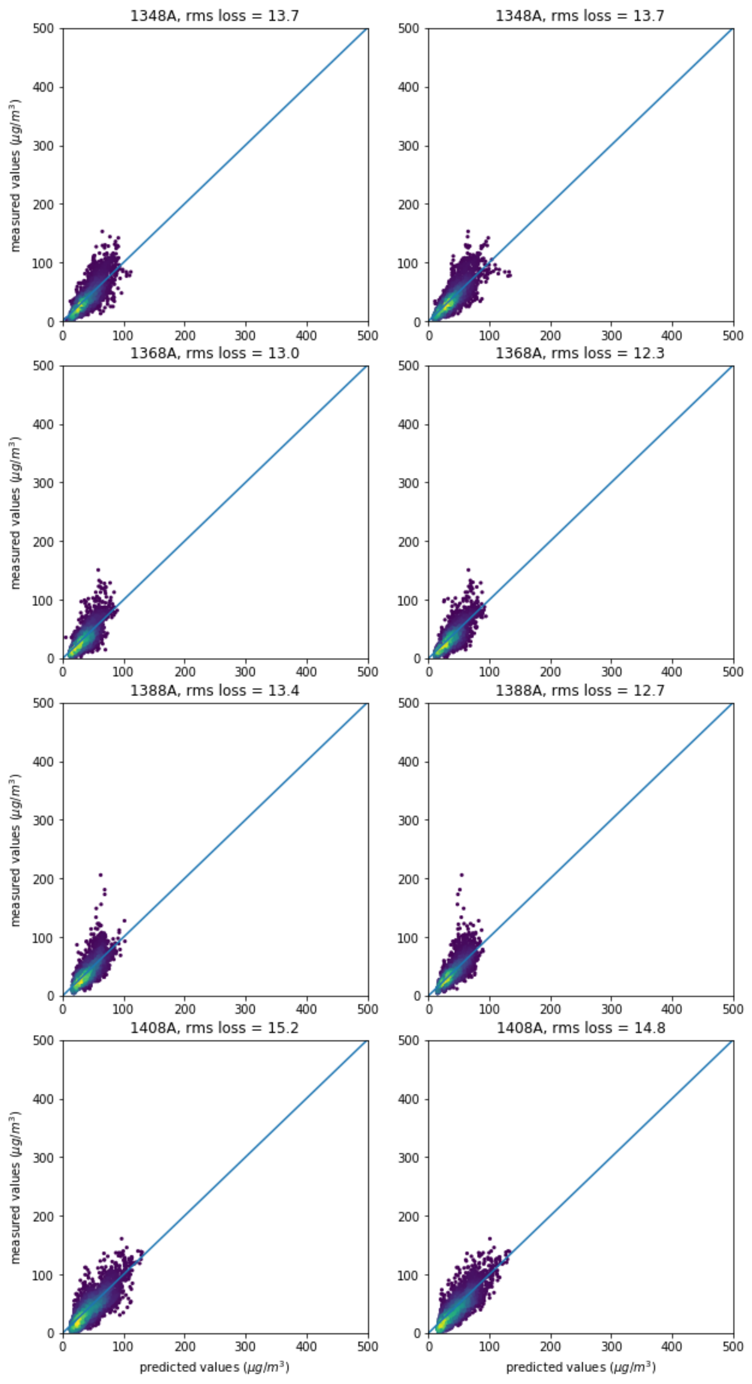

Figure A.1: Scatter plots for all 25 stations using LSTM. The plots on the right use simulation data for training, the plots on the left do not. (continued)

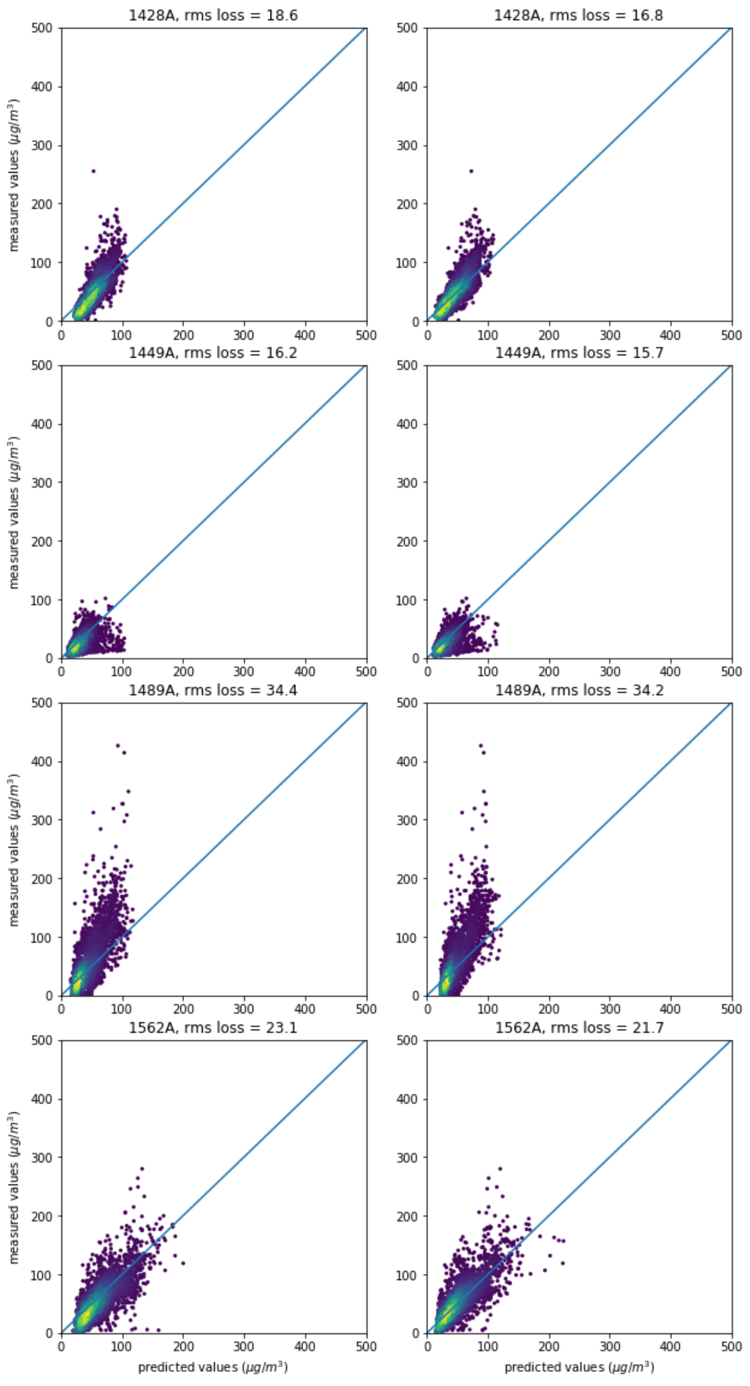

Figure A.1: Scatter plots for all 25 stations using LSTM. The plots on the right use simulation data for training, the plots on the left do not. (continued)

A.2 FNN scatter plots

Here, we will plot the scatter plots for $PM_{2.5}$ predictions done at the 25 different stations using FNN instead of LSTM. Again, the input matrices for the plots on the right also include simulation data, while the input matrices for the plots on the left do not.

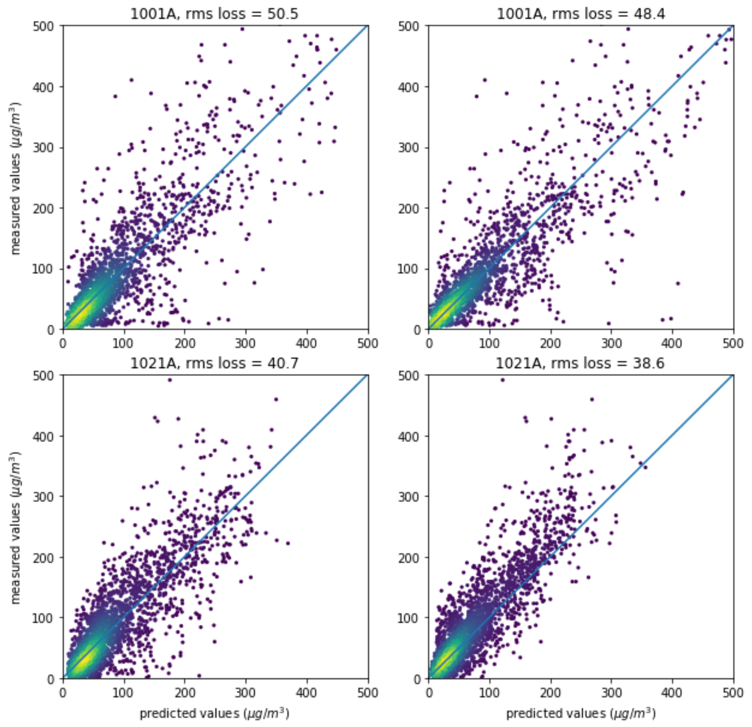

Figure A.2: Scatter plots for all 25 stations using an FNN. The plots on the right use simulation data for training, the plots on the left do not.

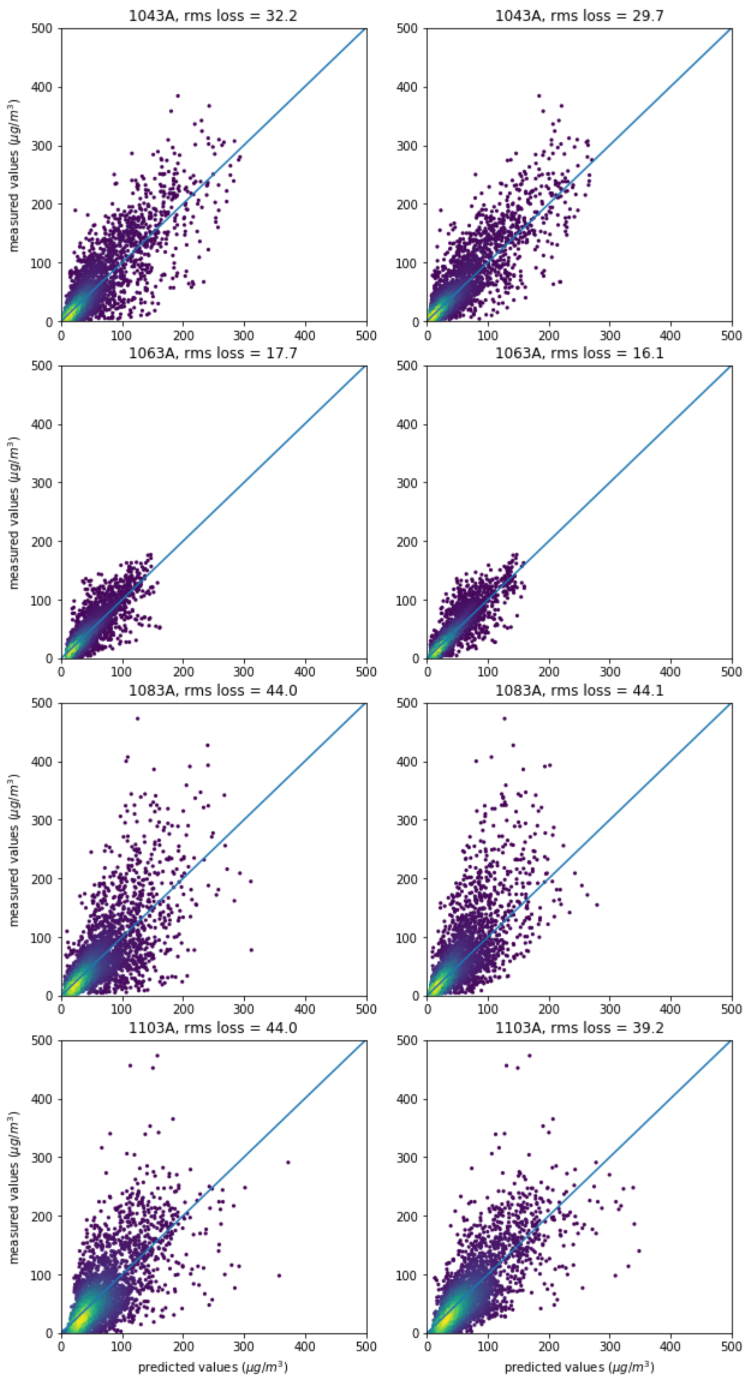

Figure A.2: Scatter plots for all 25 stations using an FNN. The plots on the right use simulation data for training, the plots on the left do not. (continued)

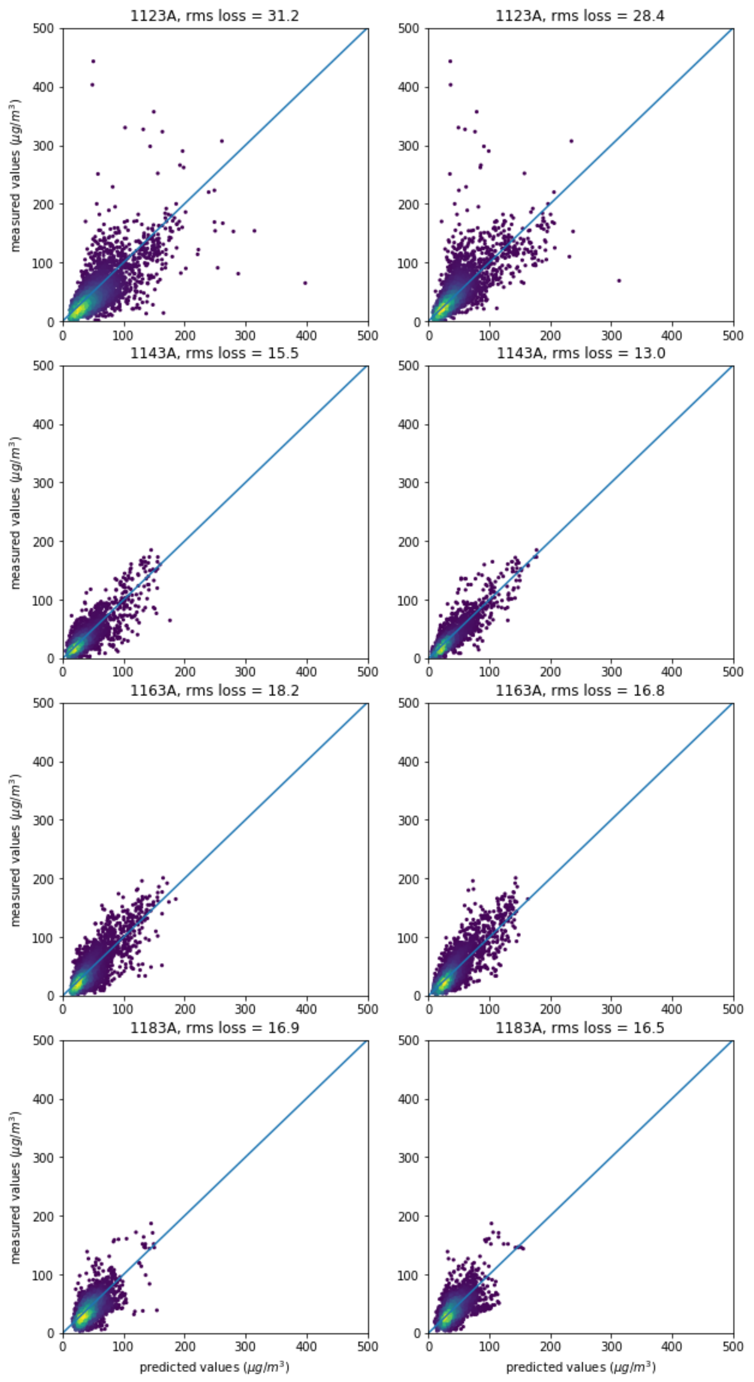

Figure A.2: Scatter plots for all 25 stations using an FNN. The plots on the right use simulation data for training, the plots on the left do not. (continued)

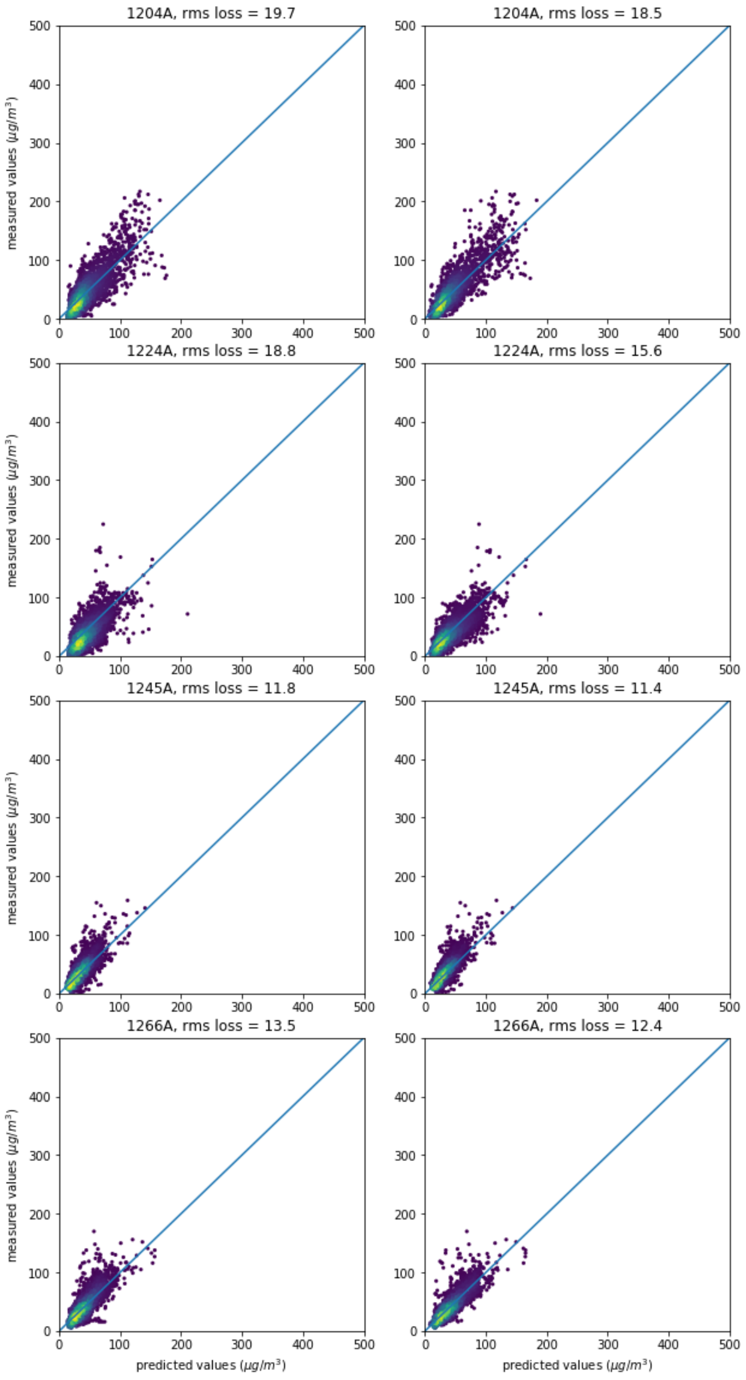

Figure A.2: Scatter plots for all 25 stations using an FNN. The plots on the right use simulation data for training, the plots on the left do not. (continued)

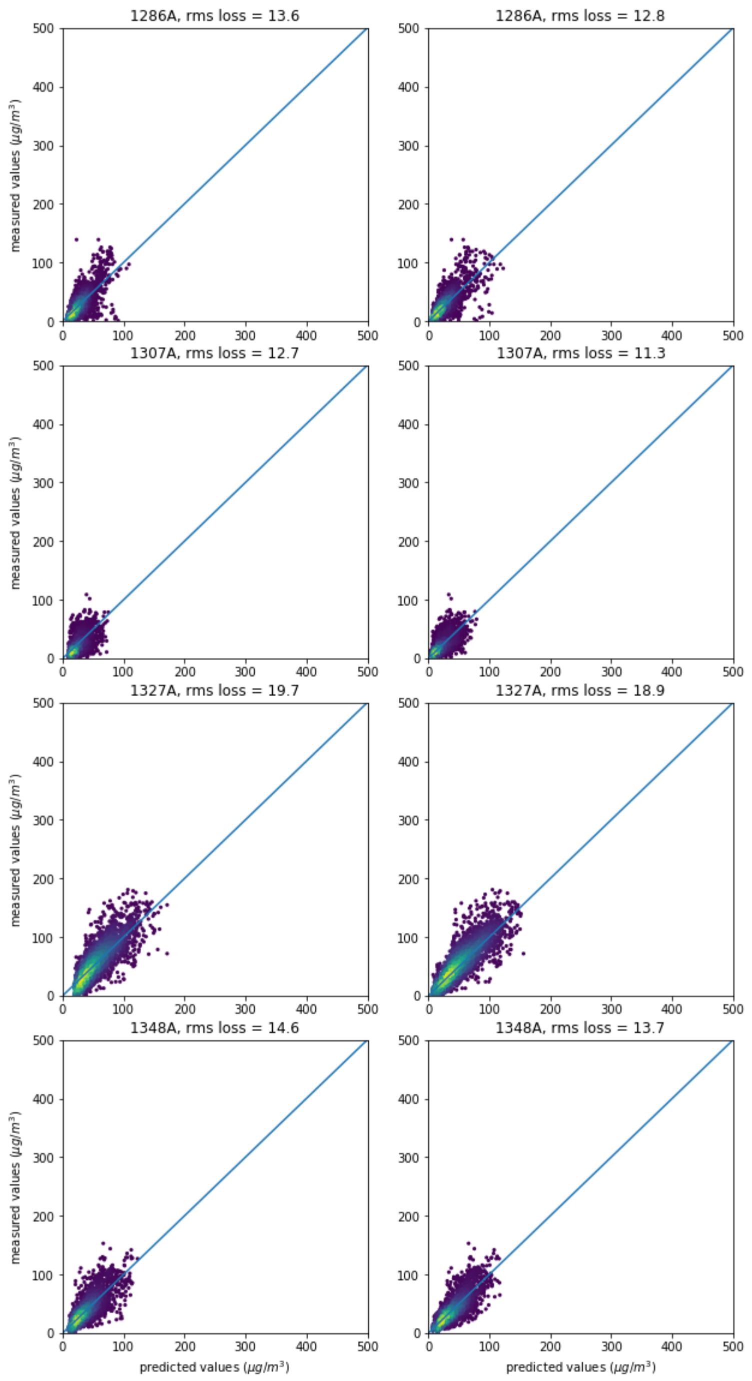

Figure A.2: Scatter plots for all 25 stations using an FNN. The plots on the right use simulation data for training, the plots on the left do not. (continued)

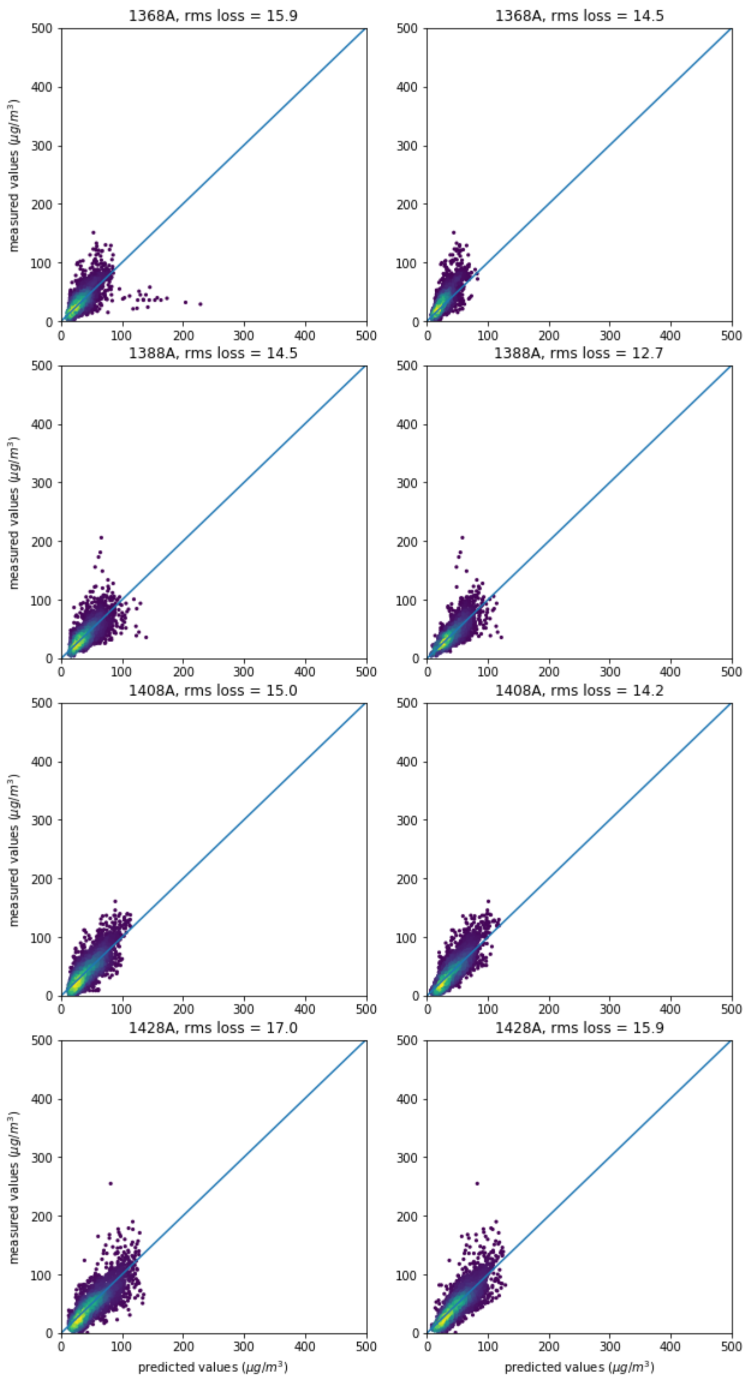

Figure A.2: Scatter plots for all 25 stations using an FNN. The plots on the right use simulation data for training, the plots on the left do not. (continued)

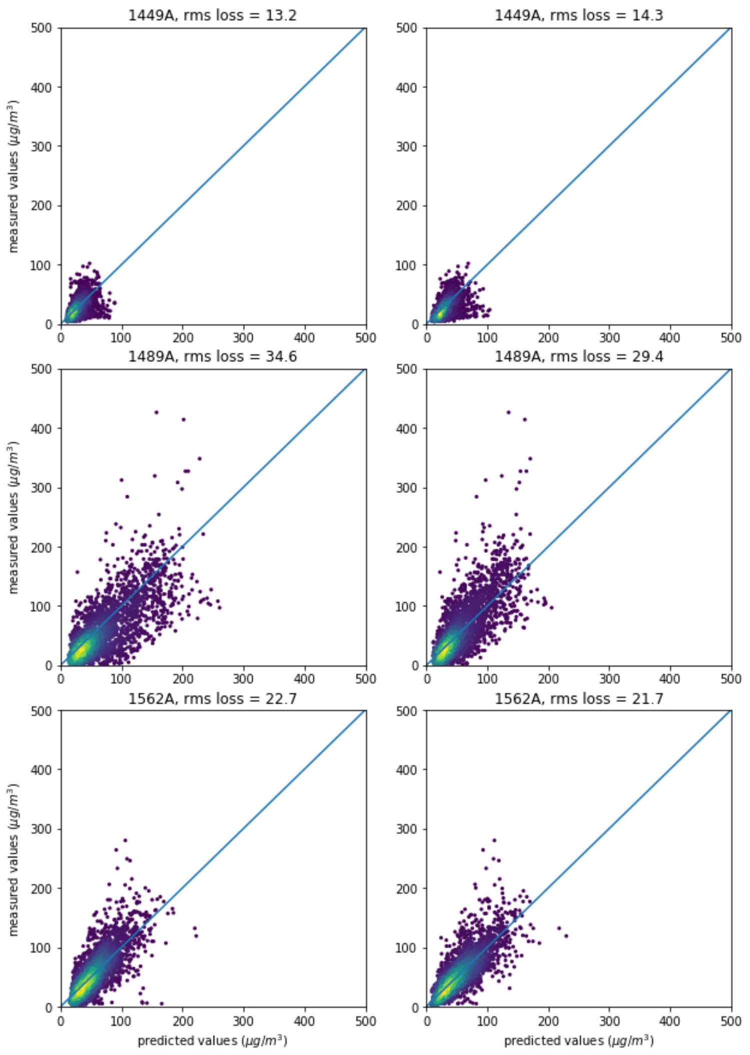

Figure A.2: Scatter plots for all 25 stations using an FNN. The plots on the right use simulation data for training, the plots on the left do not. (continued)

A.3 RMS losses

In the previous two sections, we plotted scatter plots for four different experiments across 25 different stations. We also showed the RMS loss for each of these experiments. In order to better compare these losses, consider the table below, which gives us an overview of all the different experiments.

Table A.1: Overview of all rms losses in Figures A.1 and A.2.

Station name	LSTM no sim	LSTM sim	FNN no sim	FNN sim
1001A	55.0	54.9	50.5	48.4
1021A	44.0	40.7	40.7	38.6
1043A	33.8	33.4	32.2	29.7
1063A	17.0	15.0	17.7	16.1
1083A	47.2	41.7	44.0	44.1
1103A	44.8	42.7	44.0	39.2
1123A	33.2	32.1	31.2	28.4
1143A	15.3	14.1	15.5	13.0
1163A	19.9	17.6	18.2	16.8
1183A	15.6	14.9	16.9	16.5
1204A	20.1	18.6	19.7	18.5
1224A	23.2	16.5	18.8	15.6
1245A	12.9	11.6	11.8	11.4
1266A	13.3	13.5	13.5	12.4
1286A	12.4	12.3	13.6	12.8
1307A	13.1	12.7	12.7	11.3
1327A	20.2	19.2	19.7	18.9
1348A	13.7	13.7	14.6	13.7
1368A	13.0	12.3	15.9	14.5
1388A	13.4	12.7	14.5	12.7
1408A	15.2	14.8	15.0	14.2
1428A	18.6	16.8	17.0	15.9
1449A	16.2	15.7	13.2	14.3
1489A	34.4	34.2	34.6	29.4
1562A	23.1	21.7	22.7	21.7