WS '19/'20 / 03-ME-712.07 / Prof. Dr. Frank Kirchner — Handed in on: 29.01.2020
Machine Learning for Autonomous Robots — Estimated accumulated time: **65 hours**
Solution of **group 11**: Andras Szabo, Fabrizio D'Ascenzo, Livio Guidotti, Carlo Attardi — Exercise sheet 6

All source code is hosted here.

Problem 6.1 (Variational Autoencoder (VAE))

a) We chose the TensorFlow framework. The main reasons which led us to choose that are:

- It is one of the most popular framework, so it is easy to find the documentation, moreover it has a strong community around it.
- It has high performance application
- It provides a lot of functionality

b) The code for the variational autoencoder can be found here.
The model consists of 2 neural network: an encoder and a decoder. As the definition of variational autoencoder, the encoder is learning the latent representation of the input by returning the mean and the variance. (log of the variance)
To teach the decoder, we are sampling the output of the encoder by following this formula:

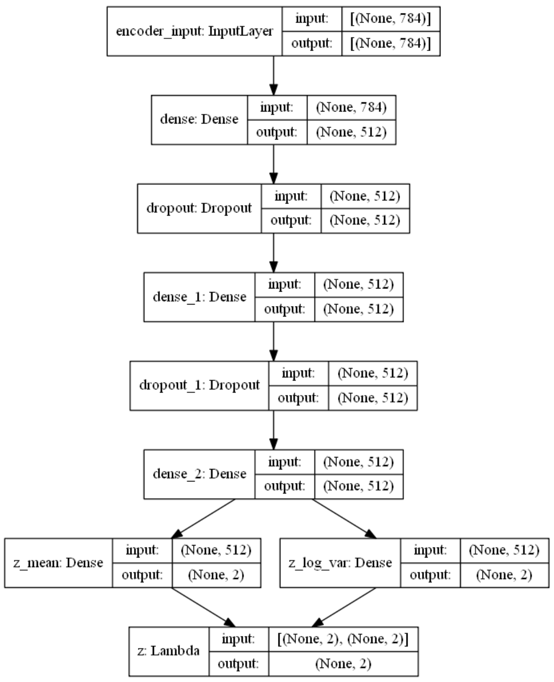
$$z = \mu + \sigma + \epsilon, \quad \epsilon \in N(0, 1)$$

This technique, called reparametrization trick, makes it possible to use the backpropagation algorithm. To implement this we used the lambda layer of Keras.

c) The loss function is: reconstruction loss + KL divergence to N(0, 1) where:
Reconstrucion loss is basically a mean square error
KL divergence:

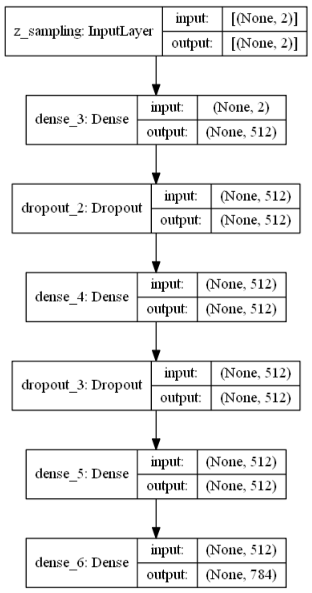
$$-0.5 * \sum 1 + \log(\sigma^2) - \mu^2 - \sigma^2$$

d) We tested 2 types of neural networks for this task. One where each neural network is a normal multilayer perceptron and one where the neural networks have convolutional layers also. The best architecture that we found for the network is:



encoder_input: InputLayer | input: [(None, 784)] | output: [(None, 784)]
dense: Dense | input: (None, 784) | output: (None, 512)
dropout: Dropout | input: (None, 512) | output: (None, 512)
dense_1: Dense | input: (None, 512) | output: (None, 512)
dropout_1: Dropout | input: (None, 512) | output: (None, 512)
dense_2: Dense | input: (None, 512) | output: (None, 512)
z_mean: Dense | input: (None, 512) | output: (None, 2)
z_log_var: Dense | input: (None, 512) | output: (None, 2)
z: Lambda | input: [(None, 2), (None, 2)] | output: (None, 2)

(a) Encoder



z_sampling: InputLayer | input: [(None, 2)] | output: [(None, 2)]
dense_3: Dense | input: (None, 2) | output: (None, 512)
dropout_2: Dropout | input: (None, 512) | output: (None, 512)
dense_4: Dense | input: (None, 512) | output: (None, 512)
dropout_3: Dropout | input: (None, 512) | output: (None, 512)
dense_5: Dense | input: (None, 512) | output: (None, 512)
dense_6: Dense | input: (None, 512) | output: (None, 784)

(b) Decoder

With this architecture we achieved the mean square error validation loss around 26. The following diagram shows the learning process:

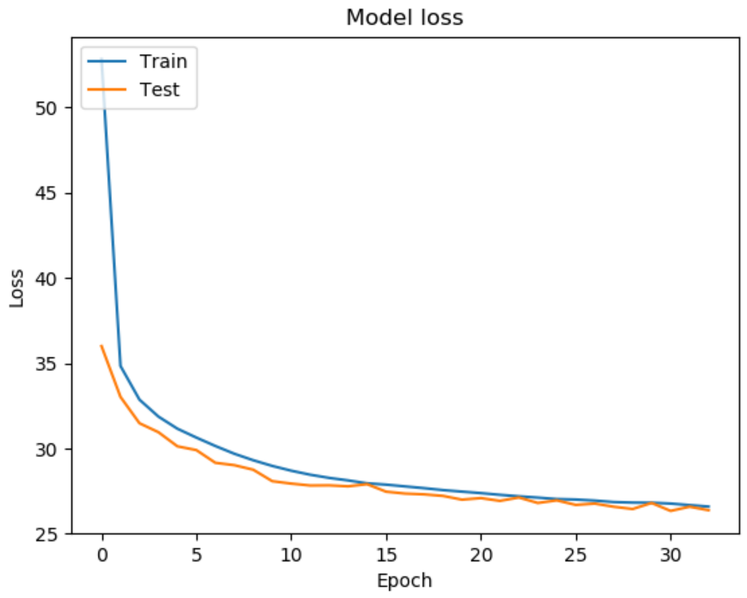


Abbildung 2: Training process of the variational autoencoder

To prevent overfitting we used different techniques such as dropout between the fully connected layers and early stopping.
To select to optimal architecture for the model we tried a lot of variant of fully connected layers with different number of nodes, we tried different optimizers (rms-prop, adam, sgd etc.) with different learning rates. Our class is optimized to be easy to change these values.
We also tried an another approach, where we placed convolutional layers on the top of the fully connected layers. This approach requires a little bit different preprocessing on the data. If we add convolutional layers to the encoder, we need to do some modification on the decoder also: at the end of the decoder we need to make deconvolution on the data to get back the same image as the input. We do also a lot of experiment with the size of the filter in the convolutional layers.
At last we stayed at the normal multilayer architecture without any convolutional layer, because they did not increased the performance as much as we expected, however they made the network much more complex, and the training much slower. The reason behind this is the simplicity of the dataset.

e) Training data in the 2 dimensional latent space. The diagram shows exactly what we expected. The classes are separated, the simliar classes are closer to eachother (like the ankleboot- sneaker) and the different classes are further (like the t-shirt - sneaker) The classes have some overlaps, but its normal because in the latent space we have much less dimension to describe the data.

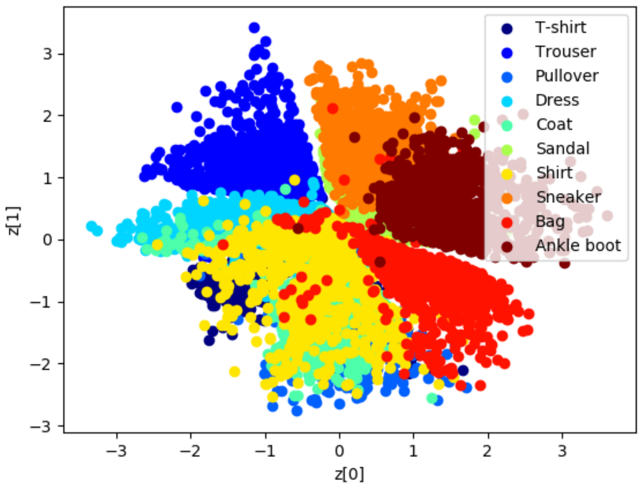


Abbildung 3: Latent space representation

f) Data from latent space projected into back to data-space using the decoder.
The results are also as expected. The simliar classes are closer to eachother, and the transition between the classes can be easily observed.

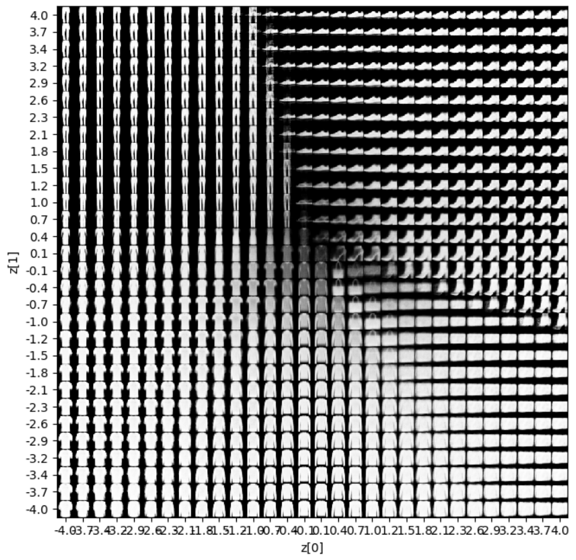


Abbildung 4: Data projection from latent space

Problem 6.2 (Two-Headed Model Architecture)

a) The code for the two-headed model architecture is located here.

The degnied model consists of a body, which is used to calculate a regression over train samples and two heads, one for mean enstimation and one for variance estimation.

A synthetic dataset is generated by a random sampling in the give interval $(-\pi, \pi)$ for train set and test set. The sampling set is split in the following way: $\frac{3}{4}$ of samples are given to the train set meanwhile, the remaining $\frac{1}{4}$, is given to the test set. A *RMSPropOptimizer* is used with learning rate 0.006 for each point of this exercise, as various experiments shown these parameters where best suited for the task.

The body is made by two hidden layers *h1, h2*. The first layer *h1* has 50 neurons, meanwhile the second layer *h2* has 20. The output of *h2* is used as input for the mean head and the variance head. Each of these heads are made by one hidden layer and an output layer.

The mean head has 60 neurons on the hidden layer *mu1*, the output layer *mu_out* has 1 neuron.

The variance head has 40 neurons on the hidden layer *sigma1*, the output layer *sigma_out* has 1 neuron.

All layers use a *relu* activation function except the *mu_out* layer which use a linear combination and layers in the variance head that use a *sigmoid* activation function.

The variance represents a tube within the true function is supposed to be, it is larger in regions where the uncertainty is more. As shown in the figures, the variance confidence is larger where the x values are positive, as the synthetic dataset is constructed in order to have more uncertainty on the positive x axis.

The resulting plot for this point is the following.

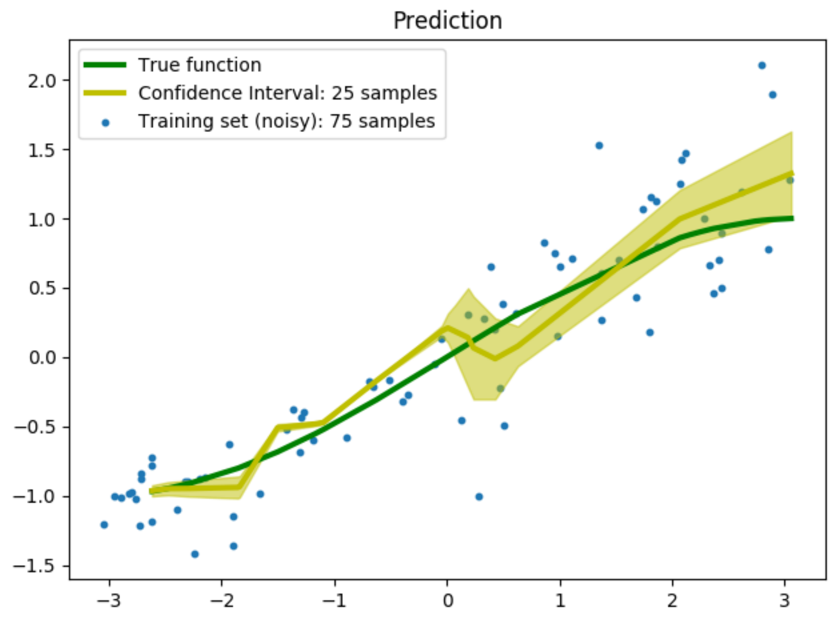
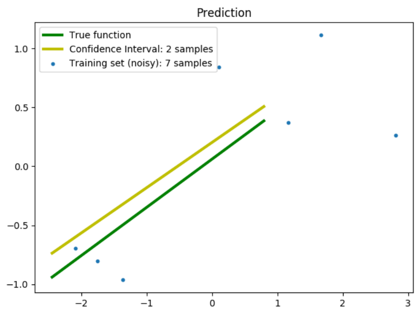
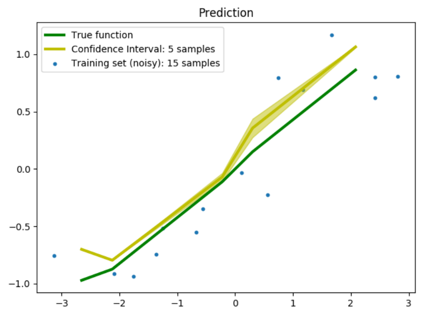


Abbildung 5: Confidence intervals for 100 samples

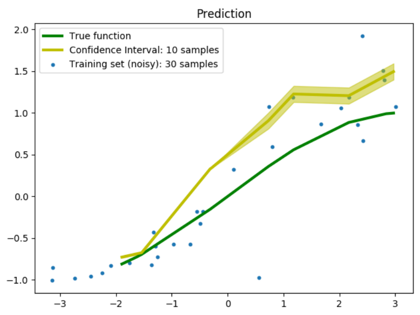
b) For this point 8 datasets have been created, each dataset’s total cardinality is the double of the previous one, the split between train set and test set is the same of previous point. The number of total points vary from 10 to 1280. As it can be seen in the graphics, the precision of the model increases as the dataset gets bigger. This can be seen by the mean of the estimation that gets closer to the true function as the dataset grows. Also, the variance increaes where the points in the dataset are far from the true function. This result can be seen nearly in all the tests done for this point and it happens on the positive side of the x axis by the construction of the dataset.



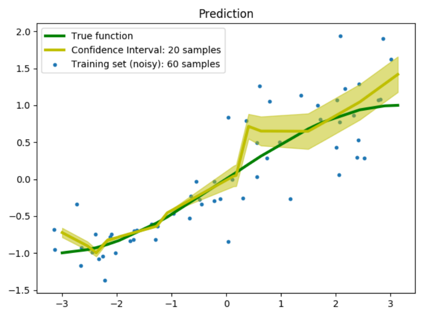
(a)



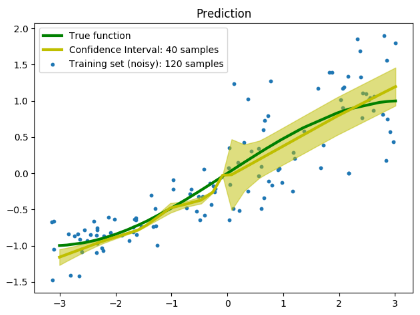
(b)



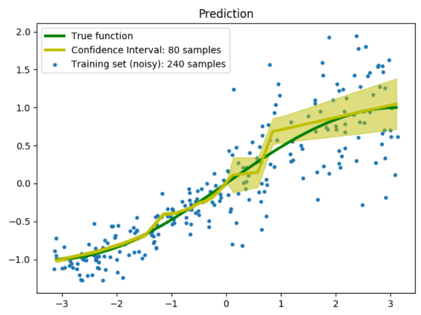
(c)



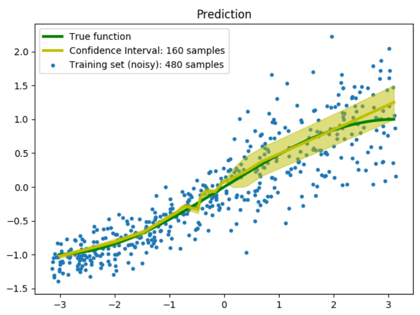
(d)



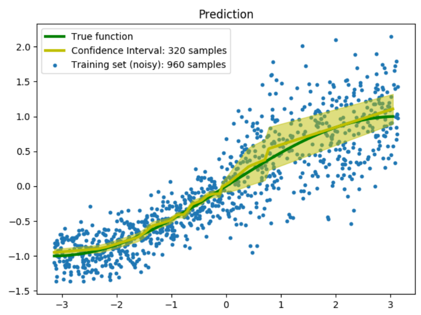
(e)



(f)



(g)



(h)

c) By sampling the test set outside the train set, one can expect that the neural network learns how to sample the whole sine function, but since it is trained only on an interval, it can only know what happens inside that interval. Also, the considered interval is not the full period of the sine thus, the model thinks about it as it was a kind of "straight line" as shown in the figure.

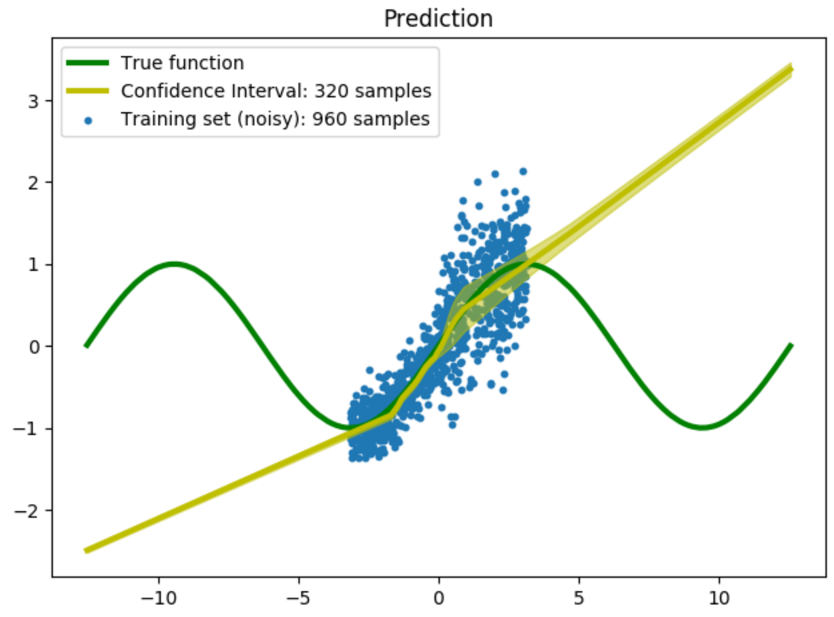


Abbildung 7: Test set points outside the training set

d) The model has been chosen with just one hidden layer on each head because, in a previous version with two layers each, it presented overfitting as the training set grew up expecially with a big number of epochs. An example of this behaviour is shown in the next figure.

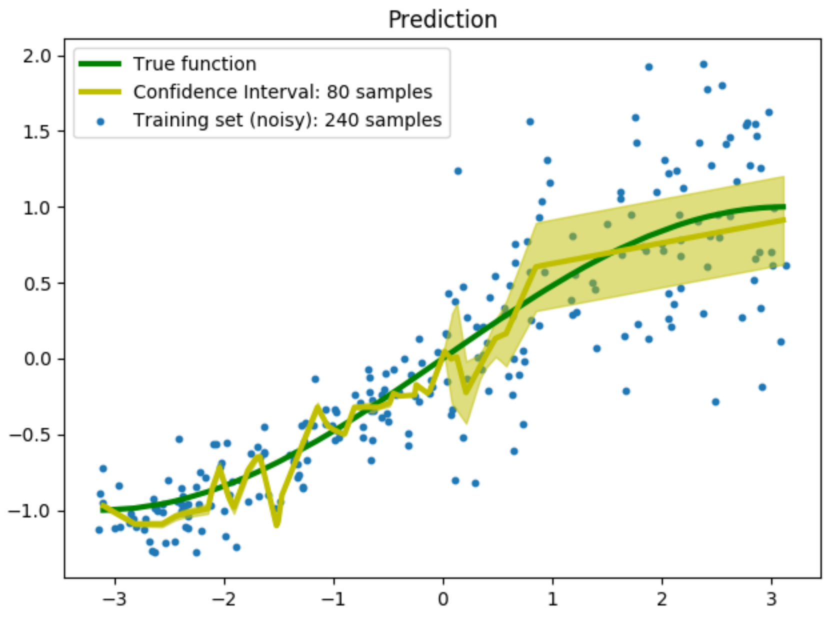


Abbildung 8: overfitting using two layers heads trained for 8000 epochs

The model chosen does not present overfitting as shown in the graphs of point a) and b).

Problem 6.3 (Time Series Regression with RNNs)

a) The code for the recurrent neural network is located here.
The task consist of 3 part: preprocessing, training and evaluating.

i **Preprocessing**
First we load the datasets using the *pandas* library. Then we filter out the columns which contains the data about the PM. There are more than 1 column, because in one city there is more than one measurement station. For this we used the pandas library filter function on the dataframes. Then we took the mean between the measurement stations with the mean function of pandas. (For each dataset)
Then our goal was to create 1 prediction for all the cities. To reach this, we took the mean between the cities and we predicted this value. This perdiction can be interpreted for example as an average PM level of the country. (For this we are supposing that there is correlation between the measured values in the different cities. In the end, we will show that if this is true, there are some correlations)
To take the mean between the dataframes we need to load the columns from the datasets which contains the time, and merge the PM dataframes based on these time columns. In this way we can prevent the inconsistency in our data in case if one row is missing from one datasource. Then we have one big dataframe which contains the datas from the cities in the columns. Then we can merge the columns together taking the means.
There are hours when there was no data measured. We have to eliminate these rows (with nan values) because we can not fit these rows to the neural network.
Next step is the normalization. To scale the data between 0-1, we used the scikit learn libary *Minmaxscaler* function.
Next we separating the data into 2 set: train and test with train factor 0.8. It means we select 80% of the data for the training and 20% for testing.
Next we are converting the dataset into time series with the specified sliding window size. One row will be a set of data from n to n+t where t is the size of the window, and the label (what we predicting) is n+t+1.

ii **Training**
Our neural network has an LSTM (long-short term memory) layer on the top. This layer makes it possible for the network to remember. (Its needed because our data is time series) We used 2 hidden fully connected layer and 1 output layer with 1 neuron, because we are predicting only the next step.
To get this architecture we tried a lot of combinations before (more hidden layer, 2 stacked lstm, bidirectionality etc.) but this combination seemed to be a acceptable tradeof between training time and accuracy.
We used mean square error for the loss, because this is a most common for the task regression. We tried also mean absolute error and some other, but mse seemed to be the best one. We optimized the loss with the Adam optimizer. It was also selected with experiment.Note that with proper parameters Stochastic gradient descent and rmsprop resulted similar results as adam.

iii **Evaluate**
To evaluate our network first we have to use it and make prediction for the train and the test set. After we have these predictions we are reverting the normalizations with an inverse transformation using a function from scikit-learn libary. (It is the same scaler that we used in the preprocessing phase, now we only inverse that transformation) After we have the result we are calculating the root of the mean square error both for the train and the test set.
In the evaluation phase its also useful if we visualize the results. (Its better than 2 number) For this we are plotting 2 plot: the training process and the result.
The result is not perfect of course, because we are predicting the mean between cities and there can be some differences which influence the dataset. But the loss is good,and after a few epoch its already good enough. Also there are no overfitting and the MSE is small.

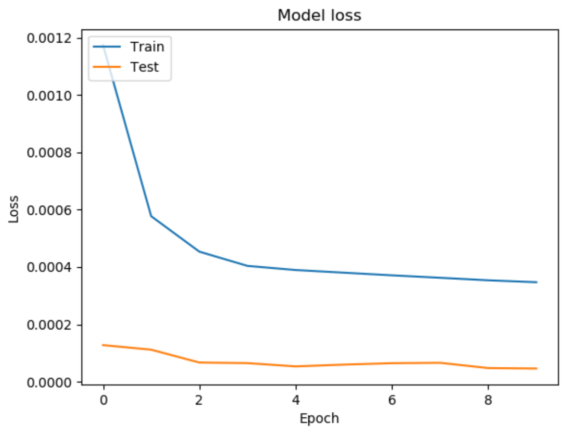


Abbildung 9: Train and test loss during training

In the figure we can see how our predictions and the original data relate to eachother. We can see the predictions and the original data are not far from eachother. Our regression model is good. The yellow curve is the prediction for the training data, the green one is for the test data and the blue one is the original data.

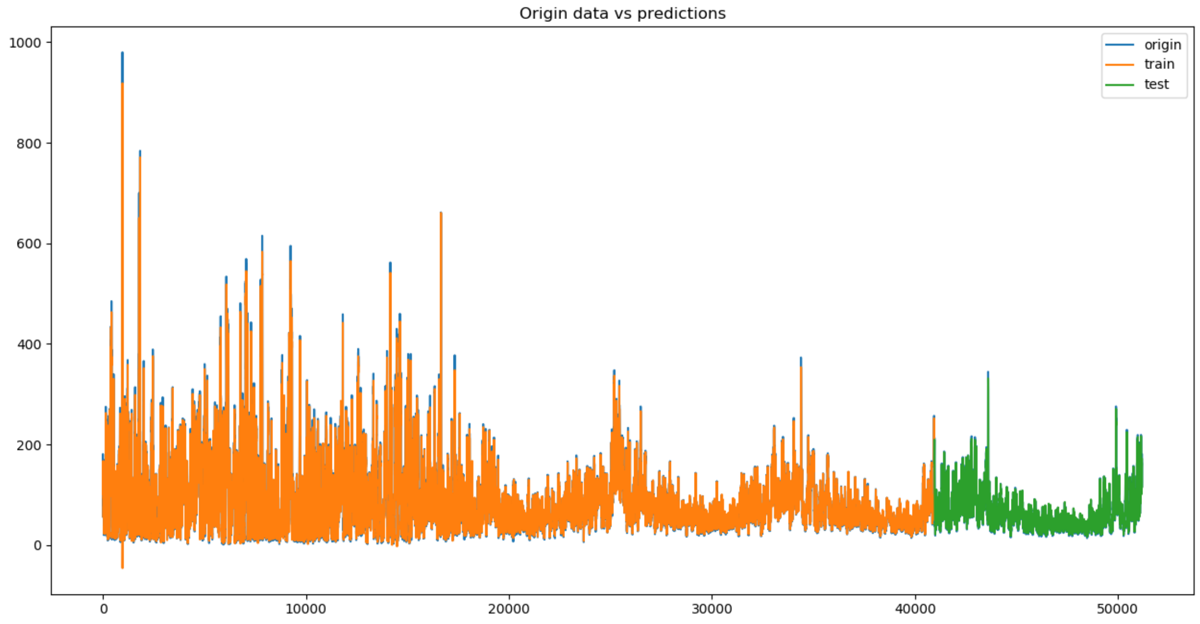


Abbildung 10: Original data vs. prediction

b) To find the best value of the sliding window size we did an experiment. The source code of the experiment can be found: here.
To find the best size we used the model described in the task a. We started the experiment from window size 2 until 28. We tried each window size 10 times, and then we took the average of the mean square errors. The following diagram show the results:

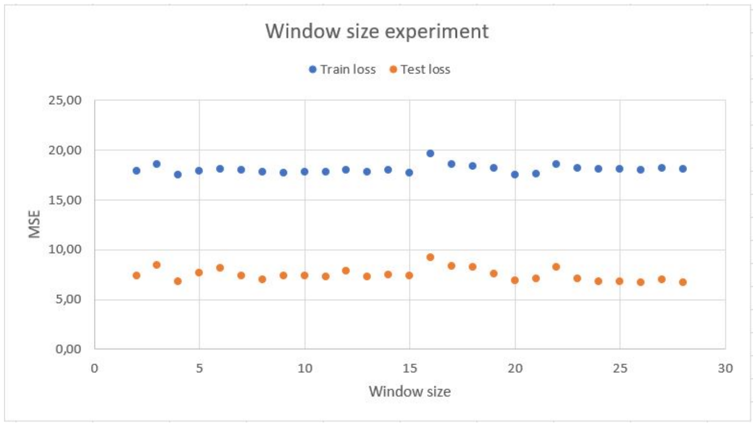
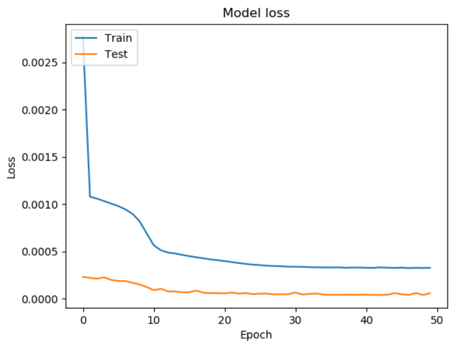


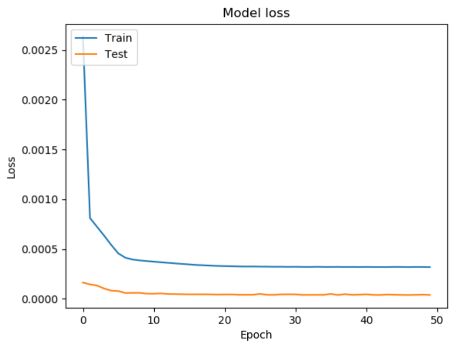
Abbildung 11: Window size experiment

The results are interesting. The diagram says clearly that the window size is not influencing the accuracy of the model. One reason behind this can be that there is not too much coherence between the hours. It is possible, because with the preprocessing we taked the mean between the cities, which is a serious generalization of the PM value.
Due to the lack of the necessary resources, we were unable to continue the experiment, because these calculation requires a lot of computational power.

c) Our attention mechanism:
As our self attention we used a simple, small fully connected neural network on the top of the encoded state of the recurrent layer. The neural network has 1 neuron in the output layer which correspond to the attention weight we will assign. We used 2 + 1 output layer in this network and softmax activation. The code for the attention located here.
You can see the results in the following diagram:



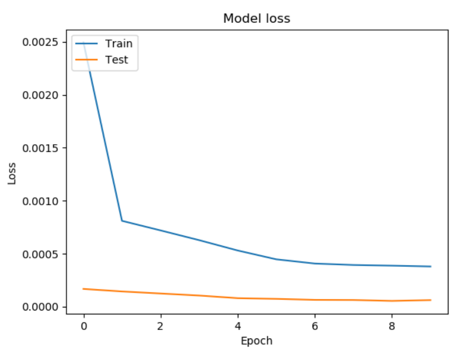
(a) With attention



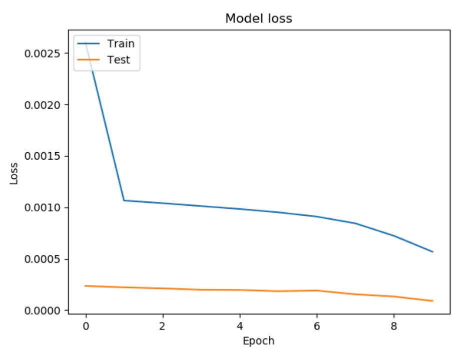
(b) Without attention

These results were made with the same neural network (lstm reccurent layer, and all the parameters specified in the task a) the only difference is the attention.
You can see on the diagrams that the one with the attention resulted a more smoother learning curve. In this concrete example it did not bring any big improvement to the accuracy, the results are almost the same with and without the attention mechanism. The reason behind this is that even without attention our network was able model the data properly. As mentioned in the previous tasks we did some generalizations in the preprocessing, and for this reason making a better regression is not possible.

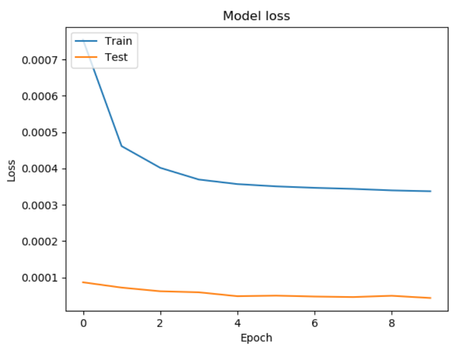
In the figure you can see our experiment comparing different reccurent layers with and without our attention mechanism.



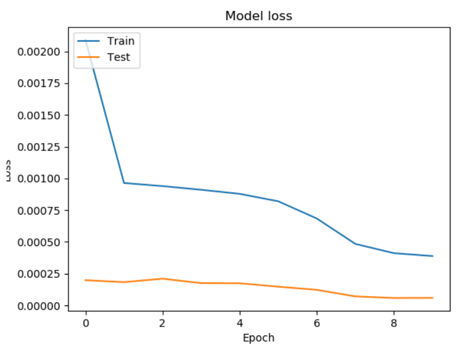
(a) LSTM



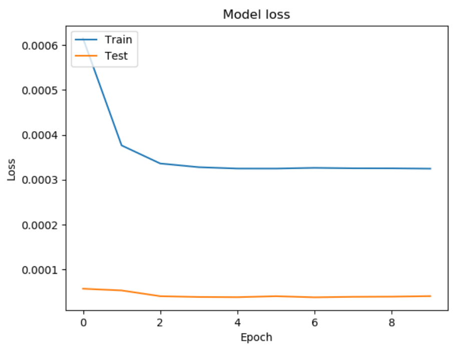
(b) LSTM with attention



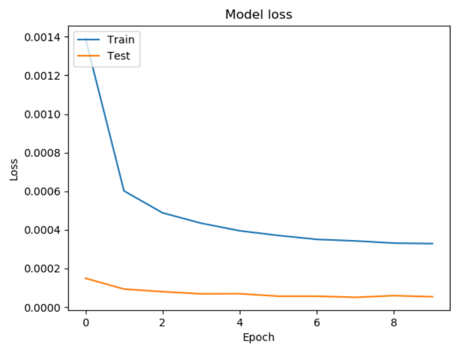
(c) GRU



(d) GRU with attention



(e) RNN



(f) RNN with attention

Abbildung 13: Comparing different reccurent units

The code for this experiment is located here.
We used LSTM - Long short term memory, GRU - Gated reccurent unit and RNN - Simple reccurent layer. We tried every reccurent layer with and without our attention.
The final loss with each reccurent layer is more or less the same. (0.0005 - for the reasons explained above) The shape of the learning curves are different, the different layers resulted different curves. Also with attention the curves are more convex, while the curves without attention are more concave. The convergence with attention is slower a bit, because it has a small reguralization effect also.