# Project report (overview)

## Definition

This project will aim at making X hours predictions for air quality. Since this is a “data for social good” and a volunteer project with no funding I will also throughout the development look into how to minimize cost used to train and deploy the model (i.e. reducing AWS usage cost). Some ideas here is to use Sagemaker’s capability to do local training (if an open source framework is used) and to look into if the municipalities need real-time predictions or if predictions can be made just a few times a day (removing the need to have an endpoint “live” at all times, thus reducing AWS endpoint EC2 cost).

The definition section of the project is mostly (and naturally) taken from the initial project description.

### Project overview

Dust is known to impose a severe health risk. An extreme example is “black lung disease” that plagued coal miners before necessary protection means were put in place. Unfortunately, we are becoming more and more aware that particle dust in large city environments is imposing a similar health risk for citizens. Municipalities have therefore found it necessary to monitor the level of dust and also to try to mitigate potential future high concentrations in some way or another to protect the public. This capstone project aims at making forecasts about particle dust levels (PM10) so that municipalities can take protective actions.

Many municipalities in Norway have already deployed sensing equipment for monitoring different pollution levels. The public road administration has been monitoring road traffic with inductive loops since 2004 in Norway and the national MET office have deployed a lot of weather monitoring stations throughout Norway. All of this data is publicly available. This means that today there exists a lot of data with potential predictive power (for particle dust levels) out there and this data can be used to create data driven models for air quality pollution.

Some previous work with data driven forecasting models for air quality have already shown good results [1, 2, 3] and might give some ideas on how to approach this area.

In Norway the **Norwegian Environment Agency** (NEA) has deployed air quality monitoring stations throughout the country (mainly in cities). These stations provide hourly data about the levels of particle dust (PM1, PM2.5 and PM10) and other polutants such as CO, CO2, COx, NO, NO2, NOx gases. At the time of writing there exists 44 such industry grade stations. Data from these sensors are available through public, open API's.

There also exists other publicly available data sources (such as weather data and traffic data) that might add predictive power when it comes to air quality. I will collect traffic data from the **Public Road Administration** (PRA) and weather data from the **Norwegian MET office** (MET).

The API's I will use to collect data from are:

**NEA**: <https://api.nilu.no/>

**PRA**: <https://www.vegvesen.no/trafikkdata/api/>

**MET**: <https://frost.met.no/api.html>

All of these API’s delivers data in JSON format and this data will have to be flattened and transformed into an appropriate format to serve as input to a machine learning algorithm.

### Problem Statement

I organized meetings with the municipality’s environmental department in Trondheim in order to understand their needs. During the discussion they highlighted this:

* For public health, PM10 is probably the most important factor
* They need support for decisions on when to and where to clean the streets
* They really want to avoid that PM10 levels increase above the regulated limit (30ug/m3)

After some thoughts about the input from the municipality I realized that a good prediction of PM10 48 hours into the future would provide input that would match their needs. Since such a forecast focuses on PM10 levels it matches the first highlighted need and since it gives them a 48 hour heads up it will make it possible for them to plan needed street cleaning. Also, if the forecast manages to capture sudden changes (e.g., fast changing levels of PM10) it would also be able to warn of near future breaches of regulations for PM10 allowing the municipality to take actions. Some relevant actions could be temporarily banning cars in the city center, intensive cleaning of streets or rerouting of traffic.

I decided to use publicly available air quality, road traffic and weather observation data as input to a machine learning algorithm in order to train the model to predict future values of PM10 in major cities in Norway. PM10 (particles with up to 10 micrometer size) is known to be a major health risk.

A prediction algorithm that provides a 48-hour forecast could probably be created using an unsupervised learning strategy where the algorithm tries to find out how the past influences the future (i.e., finding past patterns matching future outcomes). Regression algorithms could be an option but excluding logistic regression since it is only appropriate for classification problems. Logistic regression could have been relevant for the last highlighted problem described by the municipality since it could be able to predict if the next 48 hours would breach the regulated maximum for PM10 or not. Linear regression might be too “limited” to be able to solve the 48-hour forecasting problem with multivariate time series input data. Random Forest is another alternative for time series forecasting, but it requires that the time series dataset is transformed into a supervised learning problem. In my case I would like to avoid having to label data since it is very time consuming but using future values already in the data could probably transform this into a supervised learning problem.

It might also be possible to create a good forecasting algorithm using “good old statistical” Autoregressive Methods with Moving Averages (ARMA or ARIMA) or Exponential Smoothing (ES) algorithms. Some deep learning algorithms also seem to be promising with respect to time-series forecasting, and the more prominent candidates in this area seems to be Recurrent Neural Networks (RNN), Long Short Term Memory (LSTM) networks or Gated Recurrent Unit Networks (GRU). As a start, and because I want to learn more about deep learning approaches, I decided to look into usage of RNN and LSTM algorithms and the combination of the two (RNN with LSTM)[[1]](#footnote-1).

### Metrics and baseline

The forecast from the model I develop will be measured against a baseline where one assumes that the next X hours will be equal to the previous X hours.

So, what type of metric would be appropriate in order to evaluate the quality of a time series prediction? Since we are trying to predict the future values of a single variable (PM10) based on a series of other variables this is a regression problem. According to an article posted by Alvira Swalin[[2]](#footnote-2), when dealing with regression problem, some of the most useful metrics are: Mean Squared Error (MSE) or Root Mean Squared Error (RMSE), Mean Absolute Error (MAE), R2 or adjusted R2. MAE is very easy to understand, it just computes the absolute differences (i.e. treating negative and positive deviation equally) and calculates the mean deviation for all values in the prediction. RMSE, on the other hand, will penalize large differences more (by squaring it) but it is often the preferred metric since it is smoothly differentiable (hence making the algorithm converge faster since the needed computation to calculate the gradient is fast). R2 and adjusted R2 both needs a baseline to compare with and since I have that (next 48 hours will be equal to the previous 48 hours) it could have been used. Since I do not know how good this baseline is, I decided to not use R2 or adjusted R2.

Since the comparison with the baseline will be comparison between two timeseries I will use Mean Absolute Error (MAE) as the metric to compare the model with the baseline and also use MAE (on test and validation data) as the metric for selecting the best trained model. The choice is motivated by selecting the metric most easily understood (by humans) when comparing two time series. If this is a good choice or not is a bit unclear to me and I will probably try both MAE and MSE or RMSE to see if it makes a difference on the accuracy of the trained model.

## Analyze the problem

My hypothesis is that road traffic is one of the main factors contributing to particle dust and that the weather can influence the dispersion and depletion of particle dust in city environments. I also assume that particle dust in PM10 size is a relatively local phenomenon and that previous concentrations will have impact on future concentration. The hypothesis and the assumptions influence how data should be collected. I decided that I would use the Norwegian Environment stations as “anchors” since they can be used to establish previous concentrations and since my predictions can be validated towards measurements from these stations.

### Data Exploration

After some thought (and plotting in maps) I came up with the following scheme to collect data using the API´s:

1. Find all NEA station that are still active and contains at least PM10, PM2.5 and NO2 measurements
2. Find all PRA and MET stations within a 50km range from the NEA stations
3. Collect data from all identified NEA, PRA and MET stations in Norway

Fortunately, I established that all three API’s have methods to retrieve metadata about the stations and metadata from all providers contains data about their position (in lat, long format). The 50km radius is probably to high but I wanted to collect data from at least the number of stations that I later believed that I would need. Figure 1 shows a couple of example plots in maps that I made. The original maps were interactive, and it was possible to move around in them and zoom.

Map

Description automatically generated Map

Description automatically generated

Figure 1: Plotting of NEA stations in blue and PRA stations in red with a 50km radius (left side) and plotting of NEA stations and PRA stations with a 3km radius for the city of Trondheim (right side).

With this at hand, station lists for all NEA, PRA and MET stations I wanted to collect data from was created and data was collected using Airflow. Due to internal workings of the API’s, data had to be retrieved by collecting one month of data at a time. The Airflow schedules collected data from 40 NEA stations, 1307 PRA stations and 1255 MET stations. The NEA and MET station API´s were REST API´s and the PRA API was a GraphQL API. Figure 2 shows the Airflow GUI with schedules for retrieval of MET and NEA data.

Graphical user interface, application

Description automatically generated

Figure 2: Airflow data collector schedules

All of the Airflow schedules were designed to store the data in AWS S3. The data retrieved were in JSON format, and Figure 3 shows some typical examples. In total, around 150.000 monthly files were retrieved.

Text

Description automatically generatedText

Description automatically generatedText

Description automatically generated

Figure 3: JSON formatted data retrieved from the API´s.

All of the data was transformed from JSON to parquet format using python scripts with Pandas library (and other python libraries). The parquet files were also aggregated to yearly files for each station and in addition aggregated further to combined yearly parquet files including data from NEA, PRA and MET stations in each yearly file. In this final combination I used data from all PRA stations within 3km radius and MET data for all sensors values I wanted from the nearest MET station found in the data available (after collection). Note that for the MET data this implies that data for temperature could for example be values captured closer to the NEA station than for example snow depth data (or the other way around depending on availability within proximity to the NEA station).

### Exploratory Visualization

Using simple plotting of the data in the combined parquet files showed that we have a lot of missing data. Figure 4 shows an example plotting some of the columns (sensor values) for one of the 40 NEA stations we have collected data for. As can be seen, the plot shows that we have a lot of missing data for this NEA station and the same is true for the other stations.

Timeline

Description automatically generated

Figure 4: Timeseries plots shows that we have lots of missing data

I also did a correlation visualization of the data and there seems to be a correlation between PM10 and road traffic count. The correlation between PM10 and observed weather parameters seems to be low. This is a bit discouraging but hopefully the machine learning algorithm will be able to detect some hidden or more subtle correlations as well. Figure 5 shows a plot of the correlation matrix for the Elgeseter station.

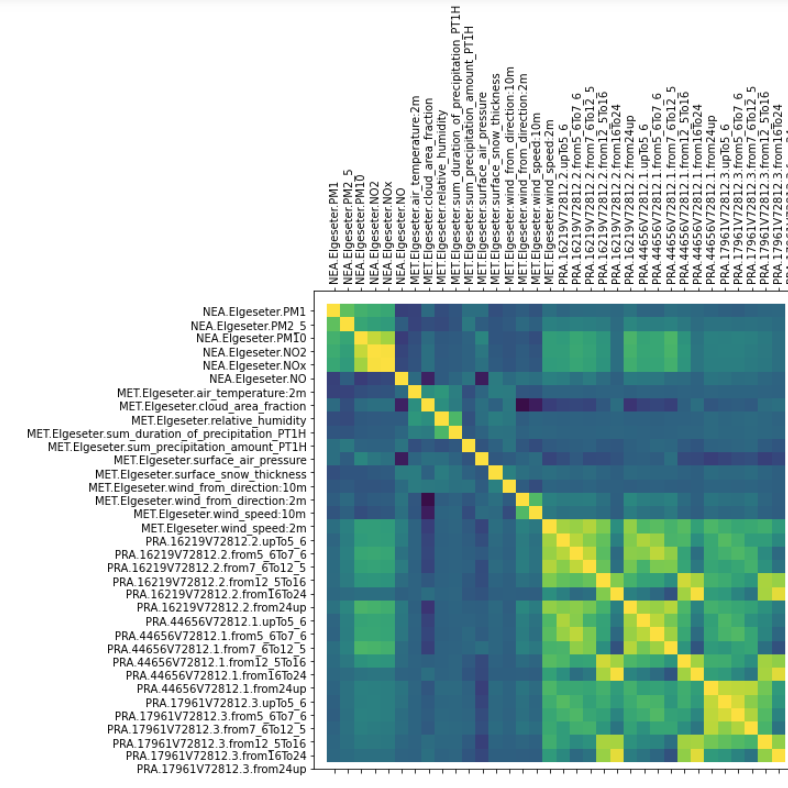


Figure 5: The correlation matrix between columns in the resulting combined dataframe for one NEA station

### Algorithms and Techniques

For the collection of data, I soon realized that a scheduler was needed, and I decided to use Airflow [[https://airflow.apache.org](https://airflow.apache.org/)]. Appropriate DAG´s for retrieval of data from the NEA, PRA and MET sources were developed. In the first version the actual retrieval was performed by an AWS Fargate task (running on AWS resources). A Fargate task is in our case a Docker container with a relatively simple Python script that does the actual retrieval using the provider API. I realized later that the retrieval of data could be done in a bit more cost (and time) effective manner using an AWS Lambda function instead of a Fargate task and the MET retrieval was therefore redesigned to use a Lambda function instead (since I had to redo retrieval of MET data due to an error that resulted in that rain data was not retrieved).

As part of the exploration of the data and searching for relevant machine learning algorithms it was fairly early established that the “state of the art” for timeseries prediction is usage of Recurrent Neural Networks with Long Short Term Memory (RNN LSTM). I therefore decided to use an RNN LSTM algorithm and ended up with Tensorflow/Keras as the underlying framework since Google has provided a good time series prediction tutorial for their Tensorflow framework.

Recurrent Neural Networks (RNN) have shown to be powerful in cases where you need contextual insight to solve the problem at hand. In time-series, the context is defined to be previous values of a variable, in languages the context is previous words and sentences that brings meaning into the next words and sentences. A RNN algorithm allows each neuron in the network to have knowledge of its own output. This brings in a bit of context but only a very short one (since only the last output is known). To overcome this, Long Short Term Memory (LSTM) was introduced in RNNs and this allows each neuron in an RNN to have an understanding of longer term dependencies. Figure 6 illustrates the difference in how a RNN network and a RNN LSTM network is composed. Note that the three neurons illustrated is actually an attempt to visualize the recurrent versions of the same neuron (hence the name **Recurrent** NN).

Graphical user interface, application

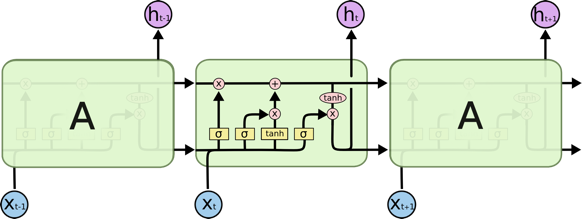
Description automatically generated 

Figure 6: Illustration of a regular RNN neurons (left) and a RNN LSTM neurons (right)[[3]](#footnote-3)

The illustration above brings some notion of that an LSTM network has some sort of memory, but how does it actually work? For a more detailed description I will refer to the original post where the figures above are borrowed from3 and to the original paper describing the RNN LSTM architecture[[4]](#footnote-4).

It was also established that the data includes a lot of missing data. This is partly due to that the different station types were deployed at different time and partly due to outage of stations during their operational period. Data imputation is therefore necessary, and I looked into methods like interpolation, backward and forward filling, filling with mean (or some other statistical measure). I decided that using backward filling could in a sense propagate knowledge about the future into the present, so interpolation was in the end used instead except for filling in empty holes after interpolation.

### Benchmark

In order to evaluate how good the model I create performs, it is necessary to have something to measure it against. This is called a benchmark test for the model. Some good benchmarks would be to compare it to someone else´s model trying to predict the same, compare towards a model created with a different algorithm (e.g., ARIMA), or towards some sort of assumption. A valid assumption in our case would be to compare against that the next 48 hours will be as the previous 48 hours. If the model is put into production it is also natural to benchmark it towards how good predictions are compared to the later observed measurements.

In my case, the MET office in companionship with the Norwegian Environmental Office do provide a model for air prediction. Unfortunately, their model only predicts if the values for the following prediction period will be within 5 (or actually 4 categories). The categories they operate with is: “Very high”, “High”, “Moderate”, “Low” and “No data”.

Since the time frame available for doing this project is relatively limited, I will not try to make an ARIMA model in addition to the RNN LSTM model (although this would have been interesting so I will perhaps try that later on). I therefore ended up deciding to use the relatively intuitive and simple assumption that the next 48 hours will be as the previous 48 hours and do the benchmark testing towards this assumption.

The benchmark value will be computed with just a little bit of code, as the following example in Figure 7, for the “Elgeseter” station, shows:

Text

Description automatically generated

Figure 7: Calculation of benchmark MAE

In order to do better than the benchmark the model therefore needs to have a MAE below the value computed (on the training data) by this code on a validation data set (i.e., data not seen during training).

## Methodology and Implementation

### Data Preprocessing

As mentioned previously, the collected data is in JSON format. All providers have hourly data for each and every sensor value they provide. JSON format is not suited to be fed into a machine learning algorithm so preprocessing of the data is needed. The steps followed during preprocessing is briefly described like this:

1. Transform the JSON data by flattening and aggregating it where suited (relevant for road traffic data since we have separate measurements for lanes and directions) and store the result of the flattening in monthly and yearly parquet files for each station for each provider
2. Combine data from different providers (into yearly combined dataframes) where the columns represent a provider’s sensor value. The columns are named to reflect where the data came from including both provider name, station id and sensor name). Examples: *NEA.Elgeseter.NO2* (where NEA is Norwegian Environment Agency, Elgeseter is name of the station and NO2 is a particular sensor value), *MET.Elgeseter.air\_temperature:2m*. For the MET data I added a separate column to identify the MET station id the data came from (the closest station with the sensor value in question). For PRA data, all stations within 3km is present in the combined data and the naming of columns is: *PRA.17961V72812.x.from5\_6To7\_6*, where PRA I provider (Public Road Administrator), *17961V72812* is station identifier, x will be a number indication proximity to the NEA station, 1 is closest, 2 is second closest etc. The last part (*from5\_6To7\_6)* is the sensor name (in this case indicating vehicles with size between 5.6 meters to 7.6 meters long.
3. Impute missing data using different strategies
4. Normalize the data using normalization techniques

Steps 1) and 2) were done using python scripts while 3) and 4) were first done in the Notebook *“local/aws-aq-tensorflow-local-ipynb” ,* but later done in scripts as well (for efficiency and to prepare for automation across the 40 NEA stations) or when looping through different hyperparameter settings as part of the experimentation to get as good a trained model as possible.

Steps 3) is done using the code in Figure 8 for imputation:

Text

Description automatically generated

Figure 8: Code to impute missing data

I have, and will further, experiment more with different imputation techniques. As an example, I have observed that for many of the 40 NEA station locations the data set has relatively large “holes” in it. Imputing these holes with the above-mentioned techniques creates “flat-lining” of data. For many datasets this will probably work fine but for the air quality data set this is probably not the best imputation strategy since it does not preserve the normal hourly, daily and weekly variations in the data. I will therefore further experiment with creating “synthetic” data to fill the holes where the “synthetic” data captures to a certain extend the natural variations in the data set and the “surrounding” values.

Step 4) normalization is done using the code in Figure 9:

Text

Description automatically generated

Figure 9: Code used for normalization of the data

In the current version of the code, the NEA columns are normalized using z-score, the PRA columns are normalized using min-max and the MET columns are also normalized using z-score.

### Implementation

Reading about machine learning on timeseries data it became clear that “state of the art” or at least the “goto” method is usage of the RNN LSTM algorithm deep supervised learning. I also discovered that Google has provided a nice tutorial on using a Tensorflow/Keras implementation of RNN LSTM. I decided to use the tutorial as a baseline for my own endeavor on the air quality data that I had collected. On top of the combined dataframe (resulting from the data preprocessing) the code uses a sliding window approach to constantly feed the RNN LSTM with an X hour input data window and a Y hour label window. As a metric for deciding the best trained algorithm I decided to use Mean Absolute Error (MAE) comparing the model performance on a validation data set and stopping the training if the algorithm did not improve within a persistency value number of epochs.

The final best model that I have trained so far has a simple RNN LSTM network defined as shown in Figure 10. The hyperparameter settings for the best model will be discussed in more details in the section “Model Evaluation and Validation”.

Text

Description automatically generated Shape

Description automatically generated

Figure 10: The RNN LSTM model used with one LSTM layer

I first tried to run the Google tutorial in a local Tensorflow/Keras environment and after a bit of hazzle I managed to run it. I then transformed the tutorial to use my air quality data and changed the code so that it could run in a Sagemaker environment. I soon discovered two annoying issues: 1) Training using Sagemaker is expensive and 2) The development cycle when using Sagemaker is irritatingly slow (since new EC2 instances needs to be spun up when you want to use a Notebook or when you want to start a training session. I discovered that when using an open-source framework (like Tensorflow/Keras, Scikit-learn, PyTorch etc.) Amazon allows you to run the training locally. I therefore installed a local Anaconda environment (including Notebook server) and transformed my training setup to use a local training server instance that I could run in our own local data center. At first, I used a 32 CPU core virtual server but later on also managed to get the local training container to work on a 40 CPU core and 8 GPU server. It took a bit of effort to manage to run it using GPU´s since I had to rebuild the Sagemaker Docker image from scratch in order to match the Cuda 10.1 environment on the server that I had at hand. It was however worth it in the end since the training speed was significantly improved. I also discovered that the Sagemaker CPU version of the Tensorflow/Keras Docker container has a severe memory leak problem resulting in that my 90GB of available RAM was consumed after around 10 Epocs of training and therefore resulted in the training session to abort. This problem was not present in the GPU enabled version of the Tensorflow/Keras training container that I created myself from scratch.

All in all, at the end, I was very satisfied with the setup since it allows me to save cost when doing training but still gives me the option to scale up using AWS resources if needed. Switching to usage of AWS resources is simply done by changing one line (the instance\_type setting) in the training script. It is also very easy to deploy an endpoint on AWS resources once I have a trained model.

When this type of prediction model is deployed as an inference endpoint it requires input data of same length, structure (including synthetic features) and normalization that the model was trained on. For my model in particular, retrieving this input data is actually a complex task since the data needs to be collected from the different sources (i.e. using the API´s from the data providers). In addition, the data needs to be retrieved from the same stations as in the training data (resulting from the combination of data given proximity constrains). To solve this problem I decided to save meta data when preparing the training data for each NEA station. The meta data is created using the column names of the training data, since the column names includes provider, station id and sensor name (and for MET data, we also have to use the additional station ID column since the naming of MET columns is slightly different). The meta data file created is subsequently used in the AWS lambda *lastx* function so that it can retrieve input data from the exact same station (and retrieving data from the same sensors). The code for *lastx* is in the **lastx** folder. As can be seen in the prediction (inference) setup below the lastx function is deployed as an AWS lambda function and the *column*\_list input to this function is the meta data file created when preparing the training data. The lambda layer used for the *lastx* function is found in the **lambda** folder.

Having a trained model deployed as an AWS endpoint also incurs running costs since it implies having an EC2 instance constantly running. During our talks with the municipality, we became aware that their needs are not a real-time, always on inference. For their use it was only needed to have “fresh” forecasts available every 2 or 4 hours. I therefore designed a setup that could run the inference at regular intervals and store the predictions (for each NEA station) in S3. To serve the latest prediction I used a Lambda functions that simply retrieves the latest forecast from S3 (as shown in the right hand side of figure 6). The complete setup for this is shown in Figure 10.

Diagram

Description automatically generated

Figure 10: The prediction pipeline setup to run every 4 hours

The AWS Step functions also gives me a graphical view so that I can monitor if the prediction pipeline run for a station succeeded or not. Figure 11 shows an example of this.

Diagram, text, chat or text message

Description automatically generated

Figure 11: Monitoring a prediction pipeline

### Refinement

During the training I did set up several different training sessions where I varied different hyperparameter tuning settings (like learning rate, batch size and number of epochs) and model specific variables as input window size and label size (although I was mostly interested in a label width of 48 hours since that is what the municipality wanted). I did see some improvements in the models but unfortunately, they all seem to overfit very fast. I also tried to use dropout (since the model was overfitting very fast) but it did not improve the MAE. In order to keep track of experiments I also installed Tensorboard. Figure 12 shows comparison of a few different training sessions.

Chart

Description automatically generated

Figure 12: Training sessions visualized in Tensorboard

## Results

The best model trained so far is performing better than a baseline where you assume that the next 48 hours will be equal to the previous 48 hours. Another observation made is that the prediction is “relatively flat”, just varying a little around the mean for the 48 hours predicted indicating that it will not be good at prediction spikes in PM10 values. The best MAE on a validation set that I have observed is around 7.16 (ug/m3) while the regulated maximum value for PM10 is 30 ug/m3.

### Justification

When running the code in the benchmark section on the example “Elgeseter” data set (which is one of the air quality 40 datasets I have created) the MAE for the benchmark is computed to be 9.99 ug/m3. For the best model that I have trained so far, the MAE on the same dataset is 7.16 ug/m3. The selection of the best model is shown in Figure 13 (using Tensorboard to find the lowest value by sorting ascending on validation set MAE for the different training runs).

Graphical user interface, application

Description automatically generated

Figure 13: Selecting (and displaying the MAE) for the best trained model

Although the best trained model is beating the benchmark it is still not good enough to be put into production. Further improvements are needed and some suggestions on how to do that is put forward in the Conclusion section.

### Model Evaluation and Validation

The data set itself is a multi-variate time-series data set with daily, weekly and season variations in it. This indicates that the model used should be able to capture long trends. It was therefore decided to use a Recurrent Neural Network with LSTM cells (RNN LSTM). A RNN network has in its own right a limited “understanding” of history due to the recurrent architecture. Adding LSTM cells increases the “understanding” of history. In order to test with different hyperparameter settings I created a test script that loops through permutations of the settings for: training rate, dropout rate and input width. The script used to automatically start training sessions with variations of hyperparameter settings is shown in Figure 14. It would of course be possible to do variations on other parameters as well and this is just a first attempt and more variations will be tested later.

Text

Description automatically generated

Figure 14: Script used to automatically run training sessions with different hyperparameter settings

The best trained model so far was achieved using the following settings for the hyperparameters:

* Training rate = 0.1
* Dropout rate = 0.2 (meaning drop 20 % of the 48-hour windows in the training set)
* Input width = 48 (show the algorithm 48 hours of data at a time)

Since the best training rate is the highest one it could imply that we are not stuck in a local minimum during gradient decent even when using a relatively high training rate. It is a bit interesting that the best model actually uses a dropout rate. I was expecting that droput would not help at all since the data is continuous and that it should not vary (very much) in form. After inspecting the imputed data set a bit more, I realized that the imputation strategy I have used might not be the best. It results in “flat lining” of data in the missing data parts that are imputed. The form of imputation I have does therefore not preserve the natural periodic variations in the data and probably “confuses” the machine learning algorithm. Using dropout would probably result in that some of these “confusing” areas of data are avoided. I will try to change the imputation strategy and instead generate some sort of synthetic data for the missing data. The synthetic data I will try to create should try to preserve the periodic variations. This can probably be done letting the imputed data be based on values from “surrounding” data or possibly similar data from the days and weeks prior and/or posterior to the data that needs to be imputed. This is considered future work and there is no time to include the results in this report.

I have also noted that the training overfits relatively fast and the reason for this can be that the data set is so small that the algorithm starts to learn very specific features of the training data or that the collected data does not have good enough predictive power. This could also be why dropout improves the trained model.

### Conclusion

It is relatively clear that the current best trained model that I have is not able to warn of upcoming spikes that will end up over the regulated limit, so improvements are needed. Since models are over fitting fast it looks like that the predictive power of the data is limited. This indicates that there are more influencing factors that significantly contributes to the values of PM10. Some probably important influencing factors, currently missing in our data, but that I will add as “features” going forward with the model training are: street cleaning schedules (e.g. the municipality do sweep the streets regularly and this impacts behavior of PM10), weather forecasts (if it is going to rain in the prediction period or if it is very cold have large impact on how particle dust behaves), fireplaces and simulation of usage of such, industrial pollution, etc. This also illustrates that domain knowledge is very important and that the (business) problem owners must collaborate tightly with the machine learning experts.

The “hybrid” cloud/local approach that AWS Sagemaker support for open-source frameworks is very interesting since it:

* Allows you to create an efficient work cycle during initial experimentations with the setup (possibly on a subset of your data). Instantiating EC2 instances for Notebooks and training sessions is time consuming and if only a cloud approach was supported you would be stuck in such a cloud supported only environment and wasting a lot of time waiting for EC2 instances to spin up
* Makes it possible to save cost if you have sufficient local resources for training sessions. This will of course depend on your needs but switching to a cloud supported setup, if needed, is easy
* Always allows you to use the Sagemaker SDK, e.g. save the training session output artifacts (model.tar.gz) in S3 regardless of if the training is done local or in AWS cloud
* Makes it possible to easily deploy your trained models on AWS resources

All in all, I now have a very good infrastructure set up that allows me to do local training on CPU only and CPU+GPU infrastructure that we have in-house. After adding more data for other PM10 influencing factors I will therefore be in a very good position to experiment more. I also have a very convenient inference model that will make it possible to provide air quality predictions for all over Norway in a cost-efficient manner using AWS resources. This is important for volunteer projects where cost will always be an issue, and in many cases it will also be important in a business perspective since cost reductions will always be important for your bottom line.

Note that this is still work in progress and the report is a snapshot at current time (November 2020) of the project.

1. Hochreiter, Sepp & Schmidhuber, Jürgen. (1997). Long Short-term Memory. Neural computation. 9. 1735-80. 10.1162/neco.1997.9.8.1735. [↑](#footnote-ref-1)
2. https://www.kdnuggets.com/2018/04/right-metric-evaluating-machine-learning-models-1.html [↑](#footnote-ref-2)
3. Images and details from: https://colah.github.io/posts/2015-08-Understanding-LSTMs/ [↑](#footnote-ref-3)
4. https://www.researchgate.net/publication/13853244\_Long\_Short-term\_Memory [↑](#footnote-ref-4)