School of Information and Communication Technology

Hanoi University of Science and Technology

Hanoi, Vietnam

PROJECT 1: IT3190E

Topic: Air Pollution Forecasting

Nguyễn Tuấn Dũng - 20194427

dung.nt194427@sis.hust.edu.vn

Phùng Quốc Việt - 20194463

viet.pq194463@sis.hust.edu.vn

Professor: Huỳnh Quyết Thắng

thang.huynhquyet@hust.edu.vn

Table of content

1.Introduction

Recently, air pollution has become a rising problem in many countries. Because of rapid industrialization along with the use of non-renewable materials, the air quality is greatly reduced, which poses a danger to the general public. The PM 2.5, which stands for particulate matter that is up to 2.5 microns in diameter, is one of the common airborne hazards contributing to air pollution. These particles are small and light, which allows them to stay in the atmosphere for longer periods of time. This pollutant is capable of causing asthma, decreased lung function, and even premature death. As a growing public health concern, environmental researchers have been exploring different methods to predict future PM 2.5 indices. Having future air pollution data in hand, researchers can unveil insights into air pollution patterns and behaviours. In addition, government officials can carry out appropriate measures in a timely manner.

Many data driven approaches, specifically machine learning models have been experimented on this air pollution forecasting task. In this project, we investigate the use of multiple machine learning techniques, either linear and nonlinear, namely SVR, LSTM and encoder-decoder with attention mechanism, to forecast the PM 2.5 index based on various airborne features.

This report will have the following structure: Section 2 describes the theoretical backgrounds of the models we used for this project, Section 3 describes our dataset, and the preparation and preprocessing to be used for experimentation, Section 4 describes the experimental methods, Section 5 discusses the results of our experiments, and finally Section 6 gives some conclusions, comments and future works for our project.

The source code of our project can be found here: Project 1

2.Theoretical backgrounds

In this project, we first propose two persistent baseline Machine Learning models that use various features of the previous hourly time steps to predict the PM 2.5 index: SVR

and LSTM. Next, developing on the baseline LSTM, we propose an Encoder-Decoder architecture that can output the air pollution levels multiple time steps into the future.

2.1. SVR

Support Vector Machine [1] (SVM for short) is a popular Machine Learning algorithm commonly used for classification tasks. SVR [2] is a slightly modified version of SVM, but using the same principle, used for regression tasks.

2.1.1. SVM

SVM tries to find the best decision boundary which separates two classes with the highest generalisation capability. Generalisation is achieved by selecting the line with the greatest gap between classes.

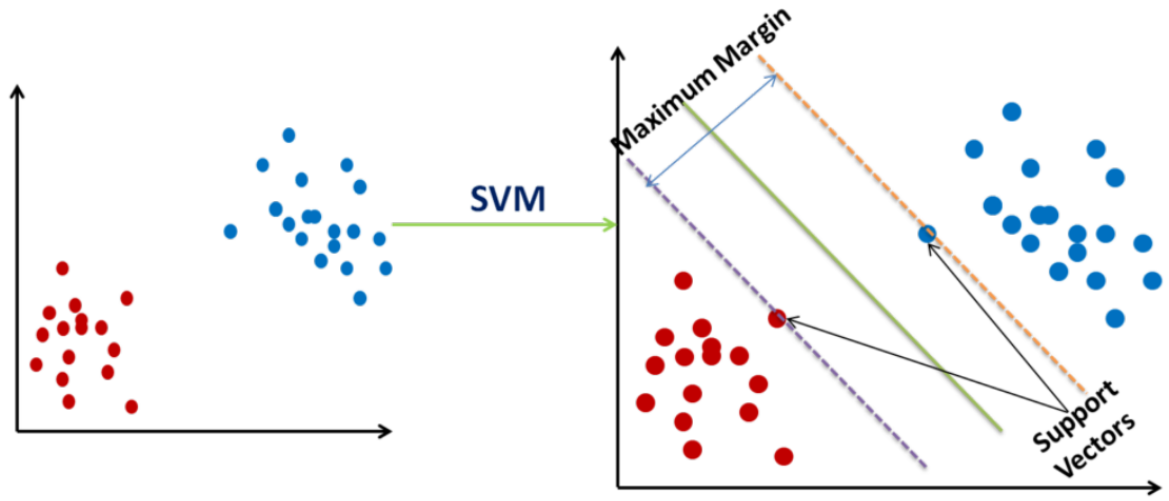


FIgure 1: Two classes separated after applying SVM

The decision function is specified by a subset of training samples which are called support vectors. The formulation of SVM with soft margin is as follows:

$$minimize: \quad \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n}|\xi_i|$$

$$constraints: \quad \begin{cases} y_i - w_i x_i \leq 1 - \xi_i & \forall i = 1..r \\ \xi_i \geq 0 & \forall i = 1..r \end{cases}$$

where y_i is the target, w_i is the coefficient, x_i is the predictor and ξ_i is the slack variable that measures the distance from the point to its marginal hyperplane. In case the data points are not linearly separable, they are mapped to a higher dimensional space, in which they are linearly separable. This is done by the kernel function. Kernel functions are generalised functions that take two vectors (of any dimension) as input and output a score that denotes how similar the input vectors are:

$K(x, y) =< \phi(x), \phi(y) >$

where x, y are input vectors, ϕ is a transformation function and $<, >$ denotes dot product operation.

2.1.2. SVR

SVR works in a very similar fashion to SVM, with some differences. Our objective function and constraints are as follows:

$$minimize: \quad \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n}|\xi_i|$$

$$constraints: \quad |y_i - w_i x_i| \leq \epsilon + |\xi_i|$$

where y_i is the target, w_i is the coefficient, x_i is the predictor. Compared to SVM, SVR has an additional tunable parameter ϵ. This parameter determines the width of the tube around the hyperplane. Points which lie inside this tube are considered correct predictions and are not penalised. The support vectors in SVR are the points that fall outside of the margin rather than just the ones at the margin

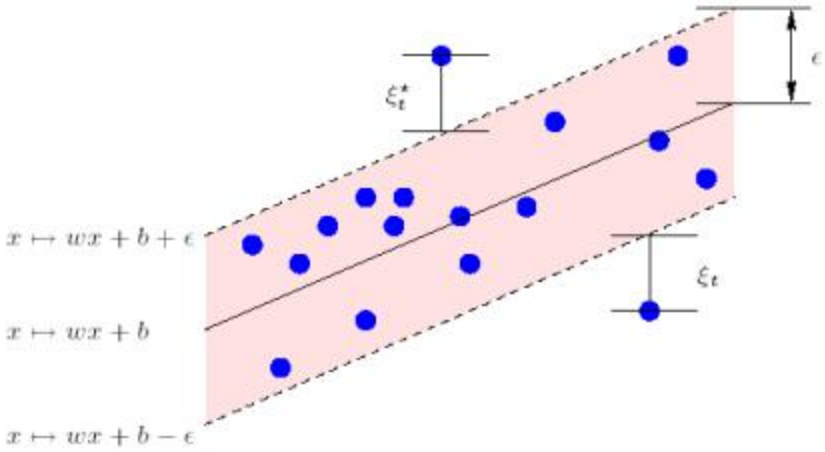


Figure 2: The tube along with slack variables and selected data points

ξ_i measures the distance between the tube and point i which lies outside of the tube. This value is placed in the objective function because we also want the hyperplane to be close to the mispredicted points. The degree of proximity is controlled by the hyperparameter C, which is similar to SVM.

2.2. LSTM

Long Short-Term Memory [3] (LSTM) is a special type of Recurrent Neural Network (RNN). RNN addresses one issue that traditional neural networks have: lack of sequential information. These images below are taken from the following blog [4].

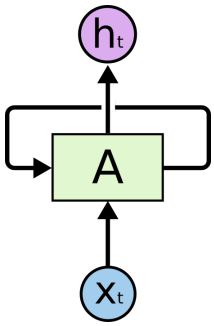


Figure 3: Recurrent Neural Network's structure

In practice, traditional RNNs suffer from long-term dependencies. As the gap between information grows, RNNs become unable to retain the link between them. In order to tackle this problem, LSTM was born. LSTM was designed to be able to remember information for long periods of time. This is done by introducing multiple gates, which

give LSTM the ability to remove or add information to the cell state. These images below summarise the difference between traditional RNN and LSTM:

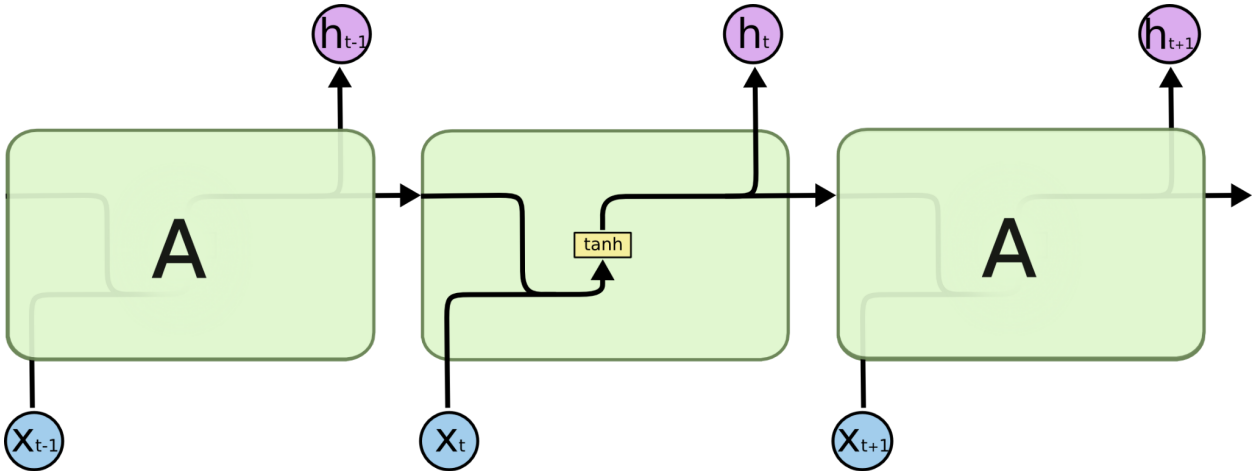


Figure 4: A single module of traditional RNN

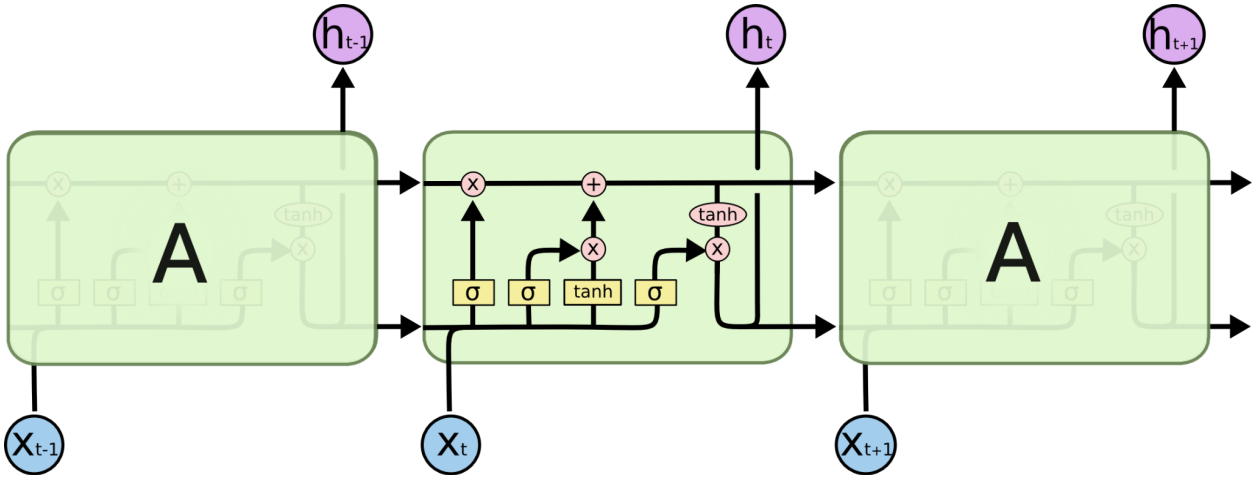


Figure 5: A single module of LSTM

The cell state C_t is the key to LSTM. It serves as a pipeline for information to flow along easily. The information added to or removed from the cell state is regulated by gates. They are composed of a sigmoid layer, which outputs a number between 0 and 1, effectively describing how important this piece of information is.

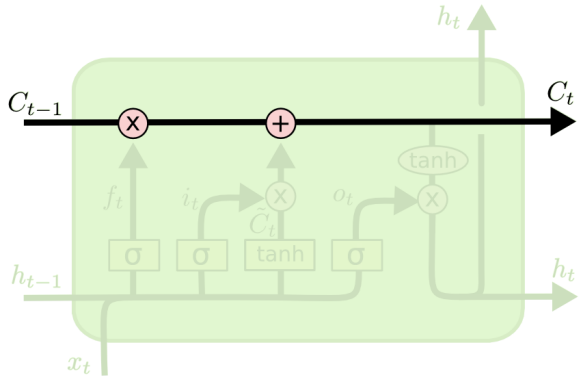
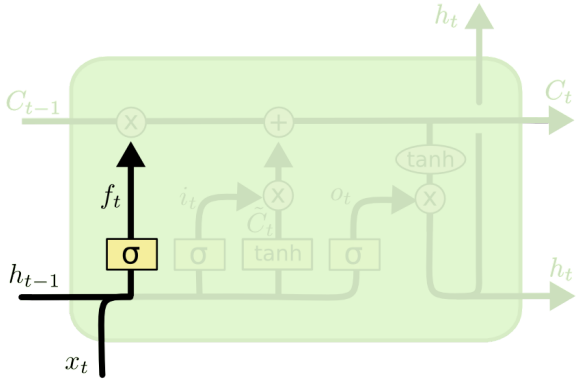


Figure 6: Cell state's pathway

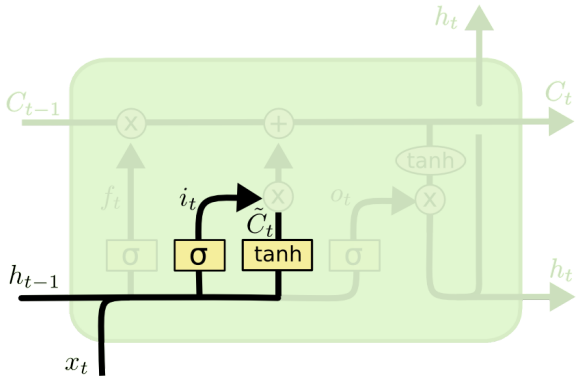
Firstly, LSTM will decide which information is irrelevant and should be removed from the cell state. This is made possible by a sigmoid layer called "forget gate". It takes the previous hidden state h_{t-1} and the current input x_t, and outputs a number between 0 (completely useless) and 1 (very important).



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

Figure 7: The forget gate

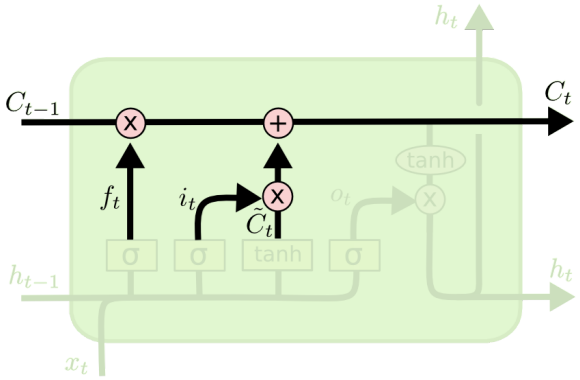
Secondly, LSTM decides which new information it is going to store in $\tilde{C}_t$ the cell state. Using another sigmoid layer called "input gate", LSTM selects which values are significant to keep. Next, a tanh layer creates the candidate values which could be added to the cell state.



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

Figure 8: The input gate and candidate values

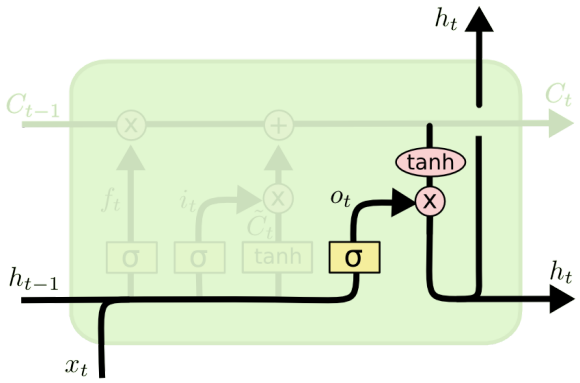
Now, a new cell state C_t is created using a combination of the old $\tilde{C}_t$ cell state C_{t-1} and the candidate $\tilde{C}_t$.



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Figure 9: The final cell state

Lastly, we compute the output and the hidden state in a similar fashion to traditional RNN



$$o_t = \sigma\left(W_o \; [\, h_{t-1}, x_t] \; + \; b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

Figure 10: The output and the hidden state

2.3. LSTM Encoder-Decoder

2.3.1. Encoder-Decoder architecture

The Encoder-Decoder is a neural network architecture based on the Recurrent Neural Network (RNN), that learns to encode a variable length sequence into a fixed length vector representation, and decodes an output sequence with variable length using that encoded vector. The Encoder-Decoder architecture consists of two main components: the Encoder and the Decoder, both being an RNN (or in our particular case, LSTM).

The Encoder is an LSTM which takes in the input sequence $x_1, x_2, x_3 \ldots x_T$. At time step t, the current hidden state h_t is calculated using the previous hidden state h_{t-1} and the input token x_t :

$$h_t = f(x_t, h_{t-1}) \quad [5]$$

Next, our encoder will transform all the hidden states at all time steps t into a context vector c, which will serve as the vectorized summary of the input sequence's information. In our implementation, our context vector is calculated using the Attention Mechanism, which we will elaborate on later.

Given the context vector c, the Decoder (which is another LSTM) is trained to generate the output sequence, predicting $y_{t'}$ at each time step t'. Here, the output $y_{t'}$ and decoder hidden state $h_{t'}$ at time step t' are dependent on the previous decoder output y_{t-1}, the decoder hidden state $s_{t'-1}$ and the context vector c summarised by the encoder.

$$s_{t'} = f'(s_{t'-1}, y_{t'-1}, c) \quad [6]$$

In our implementation, at each time step t' the previous decoder output $y_{t'-1}$ and the context vector c will be concatenated, this concatenated vector will be the input to the decoder LSTM cell, along with the previous hidden state $s_{t'-1}$. To predict the output $y_{t'}$ we will simply feed the current hidden state $s_{t'}$ through a fully connected linear layer.

In our particular problem, at time step t, the input sequence will be the sequence of feature vectors at time step $t - M$, $t - (M + 1)$, $t - (M + 2)$,..., $t - 2$, $t - 1$ (M is the input time window, or the number of previous hours we want to use as input, eg. 24 hours).

The basic Encoder-Decoder scheme is denoted using the image below:

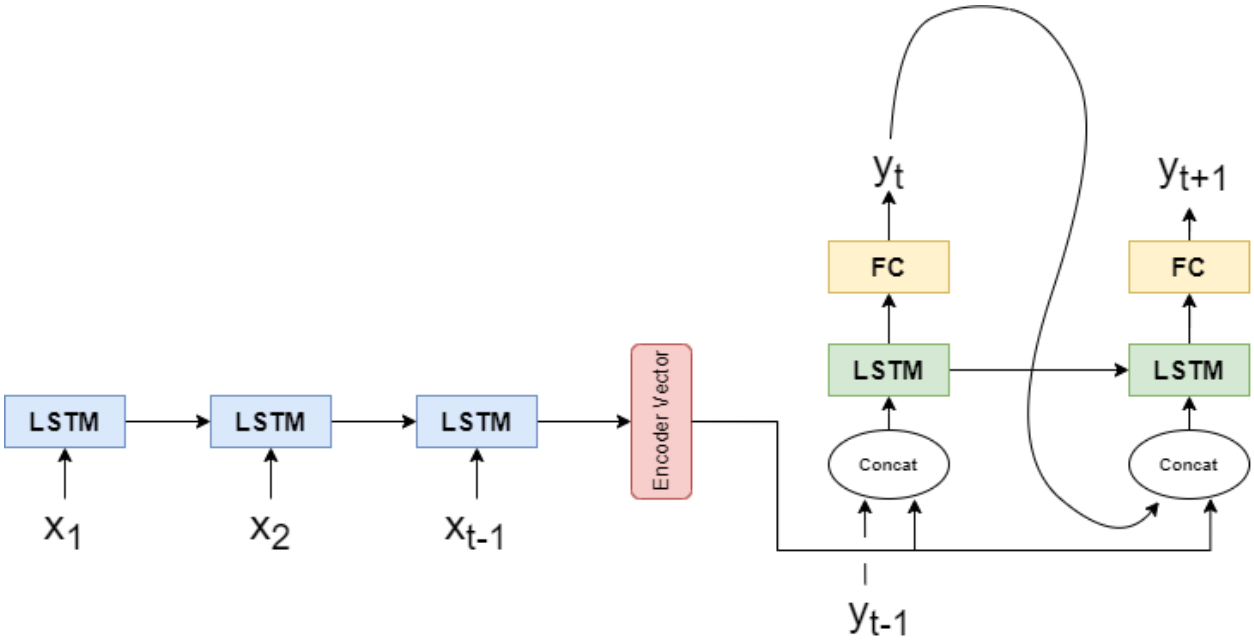


Figure 11: The basic Encoder-Decoder architecture

2.3.2. Self - Attention mechanism

One significant disadvantage of the Encoder-Decoder architecture is the problem of information bottlenecks, where the Encoder has to encode all the information of the source sequence into a single fixed length vector. The longer and more complex the source sequence, the more information might be lost after passing through the encoder.

One solution to this problem is the self attention mechanism. "*Self-attention, sometimes called intra-attention is an attention mechanism relating different positions of a single sequence in order to compute a representation of the sequence.*" [Attention is all you need]. The Self-attention mechanism consists of three main components: queries q of size R^q, the keys k of size R^k with th and the values v of size R^v. The self attention mechanism can be considered a mapping between a query and different sets of key-value pairs to map the current query to the most similar keys.

In this project, we implemented additive attention, based on Bahdanau's [6] and the implementation on *"Dive into deep learning*" [7]. Firstly, we compute the affinities between the query q and every key k with the additive scoring function. Given a query $q \in R^q$, a key $k \in R^k$, we have:

$$e(q,k) = w_v^T \tanh(W_q q + W_k k) \in R$$

whereas $W_q \in R^{h \times q}$, $W_k \in R^{h \times k}$, $w_v \in R^h$ are learnable parameters. We fed the query q and the key k through linear fully connected layers with the parameters W_q, W_k respectively, with the same hidden size h. Next, we added these two and fed them to another fully connected layer with the parameters w_v , with the tanh activation function to obtain the scores between the query and key. Next, to compute the attention weights between q and k, we fed the attention scores through a softmax layer:

$$a(q,k) = softmax(e(q,k))$$

Suppose we have a query $q \in R^q$, and key-value pairs (k_1, v_1), (k_2, v_2),… (k_m, v_m) with $k_i \in R^k$, $v_i \in R^v$, the attention can be computed as a weighted sum of the values dependent on the attention weights between the query and the values' corresponding keys:

$$c_q = \sum_{i=1}^{m} a(q, k_i) v_i$$

The attention of the query q can be summarised by figure 12 below. Figure 12 is taken from the book *"Dive into Deep Learning"* [7].

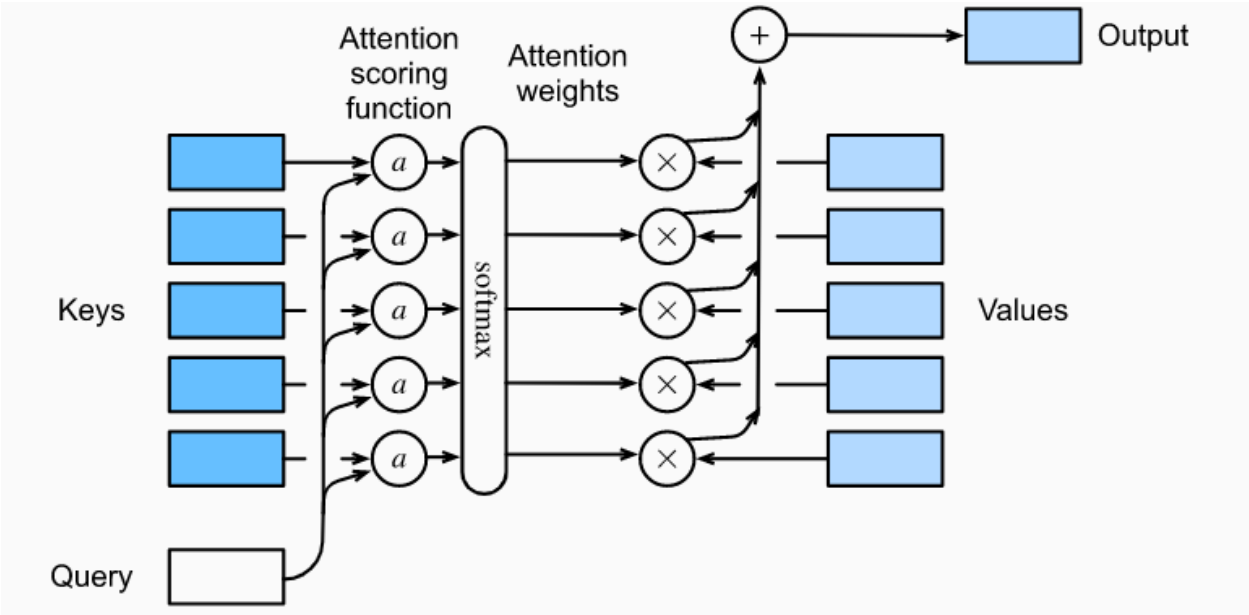


Figure 12: The attention's computation scheme

In our implementation, the queries are the current decoder hidden state, and the keys and values are taken from all the outputs of the encoder. Suppose there are T time steps in the input sequence, the context vector at the decoding time step t' will be computed as follows:

$$c_{t'} = \sum_{t=1}^{T} a(s_{t'-1}, h_t)\, h_t$$ [8]

The attention mechanism enables our network to spread the information through the tokens in the input sequence, which can be selectively retrieved by the decoder accordingly through each decoding step. As we mentioned above, the context vector $c_{t'}$ will be concatenated with the previous decoder output $y_{t'-1}$ and fed through the next decoder LSTM layer. Our final Encoder-Decoder architecture with self-attention can be summarised using the figure below:

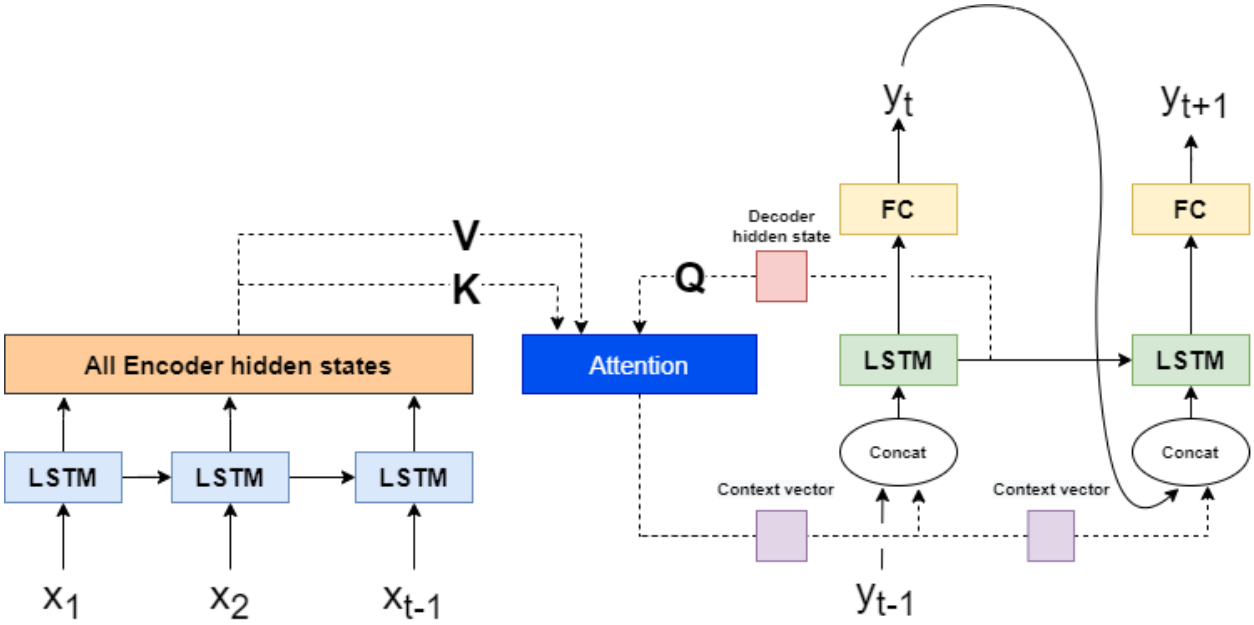


Figure 13: The Encoder-Decoder architecture with self-attention

3. Data

3.1. Data description

In this project we used the meteorological data in Chiba - the capital city of Chiba prefecture in Japan. The data is measured every hour from 1/4/2016 to 31/3/2020 at 65 stations, which are located 35-36°N, 139.8-140.6°E. We only used one station to train and evaluate our models. The dataset consists of 16 indices: PM 2.5, temperature, humidity, wind direction, wind speed, NO, NO_2, solar radiation, CH4, CO, CO_2, SO_2, non methane hydrocarbon, nitrogen oxide, oxidant, and total hydrocarbon. All of these indices are numerical features, with the exception of the wind direction. Excluding wind direction which is categorical, every other variable is numeric. Among these 16 features, PM 2.5 is the target variable.

3.2. Data preparation

Our data contains 16 csv files corresponding to 16 features: PM 2.5, CH4, CO… of all the stations measured every hour. We collected the information of each station from these feature files, and created separate files for each individual station. Each of these individual files contain all 16 features of that station measured every hour from 1/4/2016 to 31/3/2020.

After some inspection, we picked the first station to develop and test our models, since it has no missing features, and relatively few missing values and noises compared to other stations.

3.3. Data preprocessing

Firstly, we noticed that our dataset contains no information from time step 35064 onwards, so we dropped these columns from our dataset.

Next we divided our data into 3 datasets: training, validation and test set with a ratio of 75%-15%-15%, respectively. The next preprocessing procedures will be performed separately for each dataset.

There are noises in the PM 2.5 feature in our dataset, such as missing values denoted with the * symbol, rows with the PM 2.5 indices being less than or equal to 0. We filled these missing or noisy values with 0. We then replaced the rows with 0 PM 2.5 values with the average of the 6 nearest past values and 6 nearest future values.

Next, we handled missing values in other features as well. For the numerical features, we filled the missing values with the mean value of 6 nearest past time steps and 6 nearest and future steps of that feature. For the missing rows with all of its nearest time steps also missing, we filled them with the global mean of that feature. We also performed the same process on the single categorical feature of our data: the wind direction, but using the mode value instead. After filling all missing values, we performed one-hot encoding for this categorical feature.

After some basic data analysis, we noticed that our raw dataset contains many outliers. To reduce the effects of outliers and smoothen our data, we performed singular

spectrum analysis [9] (SSA) for the training set. The theory behind SSA is written in the Appendix section at the end of our report.

Due to computational resources' constraints, we broke the training set into 1000 small segments, and transformed each segment using SSA with the window length L = 250 and reconstructing the series using the first 50 matrices.

Here's the result of our PM 2.5 feature on the training set after applying SSA:

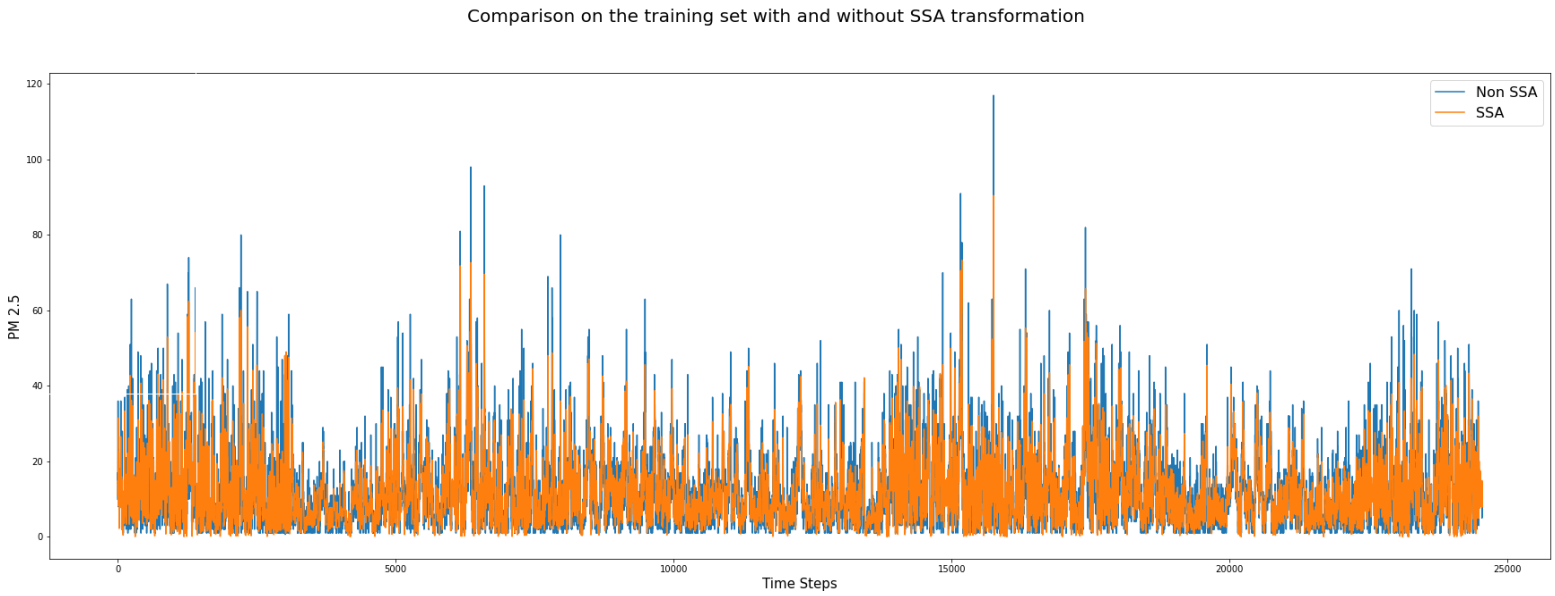


Figure 14: The training set with or without SSA transformation

Applying SSA on our training dataset helps smooth out the outliers, however, this can affect the models' ability to predict future outliers. Thus, we decided to train and test our models on the training set with and without SSA transformation to make sure we get the best result.

Finally, we scaled the features in the training, validation, test sets to a range of (0,1) using the MinMaxScaler by Scikit-learn.

4.Methods

We trained the SVR and the LSTM as the two baseline models for our main LSTM Encoder -Decoder model. All 3 models take the features from M previous time steps as input; the baseline models predict the PM 2.5 pollution level at the next single time step while our main Encoder - Decoder model predicts the pollution at N future time steps. We experimented with different input and output time windows, as well as other hyperparameters such as the C value (SVR), learning rate, number of LSTM layers (LSTM and Encoder-Decoder), and chose the best configurations by measuring their MSE losses. Finally, we compared the performance between the two baseline architectures and the main Encoder-Decoder architecture by computing some criterions: MSE loss, MAE loss and R^2 score, as well as plotting the prediction of our models on the test set.

Our experimental methods take inspiration from the "*Deep Air*" project, credited at **[10].**

4.1 SVR

In order to train a SVR on this dataset, we need to convert the dataset into a supervised one. Features from previous n timesteps that are used to predict the PM 2.5 index of the current time step are placed together on the same row along with the current PM 2.5.

pm(t-n)	CH_4(t-1)	CO(t-1)	…	pm(t-n+1)	CH_4(t-n+1)	CO(t-n+1)	…	pm(t)

Table 1: Data in each row of the dataset for SVR

We experimented with 3 numbers of previous time steps: 1, 3 and 5 time steps. For each case we performed a grid search with various settings as below. We also tested the models' performance with or without the SSA transformation. The best configurations will be chosen by comparing the MSE loss on the test set.

Hyperparameter	Value
C	0.1, 1, 10
kernel	rbf
ε	0.01, 0.1
γ	0.001, 0.01, 1

Table 2: The tunable hyperparameters for SVR

4.2. LSTM

For each data point t, our baseline LSTM model takes the environmental features at the past M time steps as input, and the PM 2.5 index at time step t as the target. After some experimenting, we set the hidden size of our LSTM model as 20. We picked 3 variables as tunable hyperparameters during the training process: the input time window, the learning rate and the number of LSTM layers. Like mentioned above, we also tested our models trained on the training dataset with or without SSA transformation with these configurations.

Hyperparameter	Value
Input time window	24, 48, 72 (hours)
Learning rate	0.01, 0.001
Number of LSTM layers	1, 2

Table 3: The tunable hyperparameters for baseline LSTM

The models are trained for 10 epochs with a batch size of 64 on Google Colab using GPU runtime. The optimizer used is the standard Adam optimizer with β1 = 0.9, β2 = 0.999 and ε = 1e-08 . The learning rate scheduler is the Cosine scheduler with Warm-up, with the number of warm-up steps being 15% of the total steps required for

the whole training process. The model evaluates the training loss and the MSE loss on the Dev set every 50 steps, and the model's current state is saved if the evaluation loss achieves the best performance. After training all the models, we evaluate the loss of these models on the test set to pick the model with the highest performance as our baseline model.

4.3. Encoder-Decoder

For each data point t, the Encoder-Decoder takes the environmental features at the past M time steps as input. Since we modelled the Encoder-Decoder for multi-step time series prediction, the target will be the N length sequence with the PM 2.5 index from time step t to time step $t + N$ as the target. Similar to the LSTM baseline, we picked the hidden size as 20. We tuned 4 adjustable hyperparameters: the input and output time window, the learning rate, and the number of LSTM layers. Out of all these, the output time window is the most significant to us, since we want to test our model's predicting performance on varying future time steps. Like for the two baseline models, we also tested these configurations on the training set with and without SSA transformation.

Hyperparameter	Value
Output time window	6, 9, 12 (hours)
Input time window	24, 48, 72 (hours)
Learning rate	0.01, 0.001
Number of LSTM layers	1, 2

Table 4: The tunable hyperparameters for Encoder-Decoder

We train the Encoder-Decoder with the similar settings as the LSTM baseline model. The models are trained for 10 epochs with a batch size of 64 on Google Colab using GPU runtime. The optimizer used is the standard Adam optimizer with β1 = 0.9, β2 =

0.999 and ε = 1e-08 . The learning rate scheduler is the Cosine scheduler with Warm-up, with the number of warm-up steps being 15% of the total steps required for the whole training process. We set the teacher force ratio to be 0.5. The model evaluates the training loss and the MSE loss on the Dev set every 50 steps, and the model's current state is saved if the evaluation loss achieves the best performance. We also implemented early stopping for our models. After a number of steps which is equivalent to 2.5 epochs, if there are no improvements over the dev set then the model will stop the training process. After training with all of the configurations above, we evaluate the loss of these models on the test set. For each output time window (6,9,12 hours), we picked the best performing model for further analysis.

5.Experimental Results

5.1. SVR

After training the SVR on different hyperparameter configurations using grid search, we ended up with the best models for each of these input time steps as below:

Time steps	SSA				No SSA			
	MSE	MAE	R^2	Time (s)	MSE	MAE	R^2	Time (s)
1	38.19	3.95	0.53	27	34.51	3.76	0.58	34
3	73.5	5.64	0.10	120	**31.89**	3.67	0.61	138
5	52.94	4.81	0.35	3120	36.38	3.79	0.55	144

Table 5: The best SVR models for each input time steps

From this table, we can see that the model's performance without SSA surpasses that with SSA, especially the 3-time-step and the 5-time-step model. The best performance overall is achieved by the 3-time-step model without SSA. It appears that using 3-time-step, the model can take advantage of more information from the past, thereby generating a closer prediction to the actual value. But when more steps are used, the number of features becomes increasingly large. This negates the information gained from using more time steps as the model becomes more difficult to train. The 5-time-step model with SSA takes significantly longer than the others due to the overwhelming number of features.

5.2. LSTM

Time steps	SSA				No SSA			
	MSE	MAE	R^2	Time (s)	MSE	MAE	R^2	Time (s)
24	34.93	3.86	0.57	58.48	**30.71**	3.64	0.62	31.86
48	34.46	3.83	0.58	62.73	30.72	3.63	0.62	31.70
72	34.08	3.76	0.58	129.70	30.98	3.66	0.62	53.01

Table 6: Best LSTM models with different input windows

The table above denotes the performance of the best models with 24,48,72 hours size window as input. We can see that generally, models trained without SSA all achieved slightly better performance compared to those trained with SSA. The best models for each input time step don't seem to have much impact in terms of performance, and the best performing model is trained with only 24 time steps as input. Meanwhile, increasing the input window size seems to increase the training time as well, so a 24 hour window length might be the best choice for training.

We will pick the best performing model, which is the model with 24 input window length and no SSA to further analyse in the comparison section. This model is trained with a learning rate of 0.01, and 2 LSTM layers.

5.3. Encoder-Decoder

Output steps	SSA				No SSA			
	MSE	MAE	R^2	Time (s)	MSE	MAE	R^2	Time (s)
6	63.96	5.32	0.21	28.58	**49.59**	4.65	0.39	93.07
9	69.14	5.55	0.15	137.51	**57.33**	4.94	0.29	90.07
12	71.12	5.63	0.12	52.75	**61.41**	5.22	0.24	96.46

Table 7: Best Encoder-Decoder models with different output windows

Once again, it seems like training without SSA transformation results in better performance for our models for all the output time windows. Like we stated earlier, since our problem has to deal with a lot of extreme outliers, using SSA to smooth out our data might weaken the model's ability to predict these extreme outliers.

We noticed that the number of output time steps greatly affects the performance, and as the number of output time steps grow with the MSE and MAE loss all increases and R^2 score decreases significantly. This is expected, and we can hypothesise that when decoding the output sequence, as the sequence gets longer, the model loses more and more information, and thus the model's accuracy starts to decrease as well, making later predictions less accurate compared to earlier ones.

Also, it's worth noting that since we implemented early stopping for our models, some models with configurations that converge quickly might exit the training process sooner than others, making the time really uneven.

Input time steps	MSE	MAE	R^2
24 hours - 6 hours	51.71	4.69	0.36
48 hours - 6 hours	50.59	4.67	0.38
72 hours - 6 hours	49.59	4.65	0.39

Table 8: The best Encoder-Decoder models with each input time window corresponding to 6 output time steps

Input time steps	MSE	MAE	R^2
24 hours - 9 hours	57.32	5.00	0.29
48 hours - 9 hours	57.33	4.94	0.29
72 hours - 9 hours	60.60	5.13	0.25

Table 9: The best Encoder-Decoder models with each input time window corresponding to 9 output time steps

Input time steps	MSE	MAE	R^2
24 hours - 12 hours	61.41	5.22	0.24
48 hours - 12 hours	64.58	5.28	0.21
72 hours - 12 hours	63.90	5.21	0.21

Table 10: The best Encoder-Decoder models with each input time window corresponding to 12 output time steps

As we can see from the 3 tables corresponding to each input-output window length above, the input time window size doesn't seem to have a significant impact on the models' performance.

We will pick the 3 non ssa models for the 3 output time steps 6,9,12 as the best models for further analysis. Here are the detailed configurations of these 3 models.

Output time steps	Input time steps	Learning rate	Number of LSTM layers	SSA
6	72	0.01	2	No
9	24	0.01	2	No
12	24	0.01	2	No

Table 11: The best Encoder-Decoder models with each output time window

5.4. Comparison

Best model	MSE	MAE	R^2
SVR	31.89	3.67	0.61
LSTM	30.71	3.64	0.62
Encoder-Decoder (6h)	49.59	4.65	0.39
Encoder-Decoder(9h)	57.33	4.94	0.29
Encoder-Decoder(12h)	61.41	5.22	0.24

Table 12: Comparison between our baseline models and the Encoder-Decoder models

As we can see, the two baseline models SVR and LSTM are quite similar in terms of performance, with LSTM being slightly better in all 3 criterions.

We can see that the measures for the Encoder-Decoders are worse compared to the baseline models, and the Encoder-Decoders themselves worsen as the output window size increases. As we have hypothesised above in the Encoder-Decoder section, the longer the output sequence grows, the more information the decoder loses and that worsens the model's performance on the later time steps. Since the two baseline models only have to predict one immediate future time step compared to multiple time steps of the Encoder-Decoder, the two baseline models' general accuracy will be better.

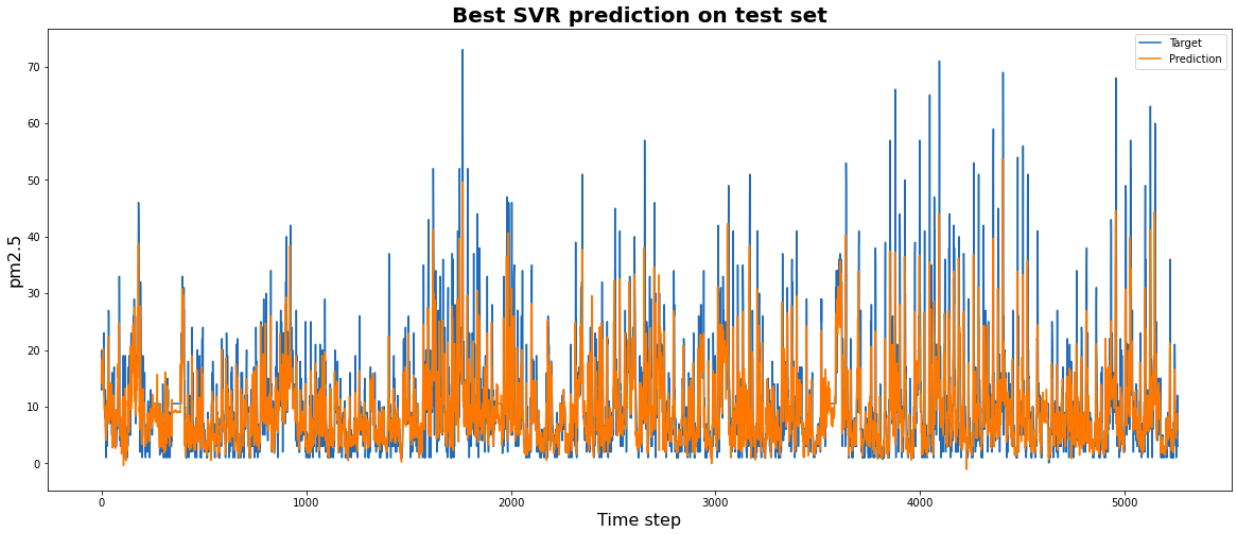


Figure 15: The SVR predictions on our test set

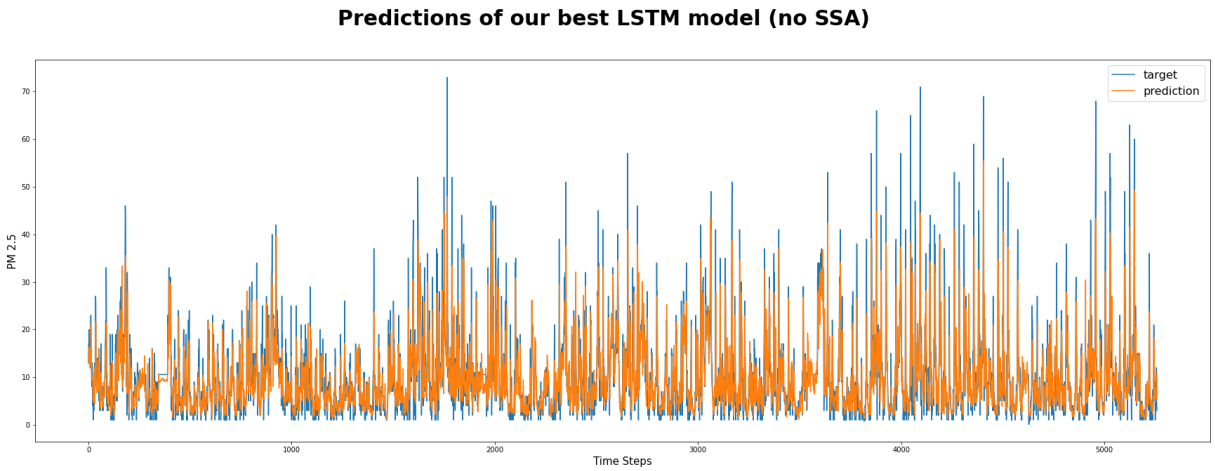


Figure 16: The LSTM Baseline predictions on our test set

The two figures above display the prediction of the SVR and LSTM baseline models on the test set. As we can see, the two models fit the time series relatively well. While there are a lot of extreme outliers that the two models failed to get close to, most outliers are detected and whenever there are spikes in our target time series, the models' predictions also spike up as well.

Next, we want to analyse the Encoder-Decoder performance at each output prediction step. We will evaluate the first, the 3^{rd}, the 6^{th}, the 9^{th} and the 12^{th} future hour prediction at every output sequence of our model. We will use the Encoder-Decoder model with an output window length of 6 to evaluate at the first, the 3^{rd}, the 6^{th} future hour predictions, the model with an output window length of 9 to evaluate at the 9^{th} future hour predictions, and the model with an output window length of 12 at the 12^{th} future hours prediction.

Prediction step	MSE	MAE	R^2
1	30.92	3.69	0.35
3	50.79	4.71	-0.46
6	60.225	5.15	-1.53

9	69.4	5.59	-2.69
12	72.68	5.66	-6.14

Table 13: The best Encoder-Decoder models' performance at future time steps

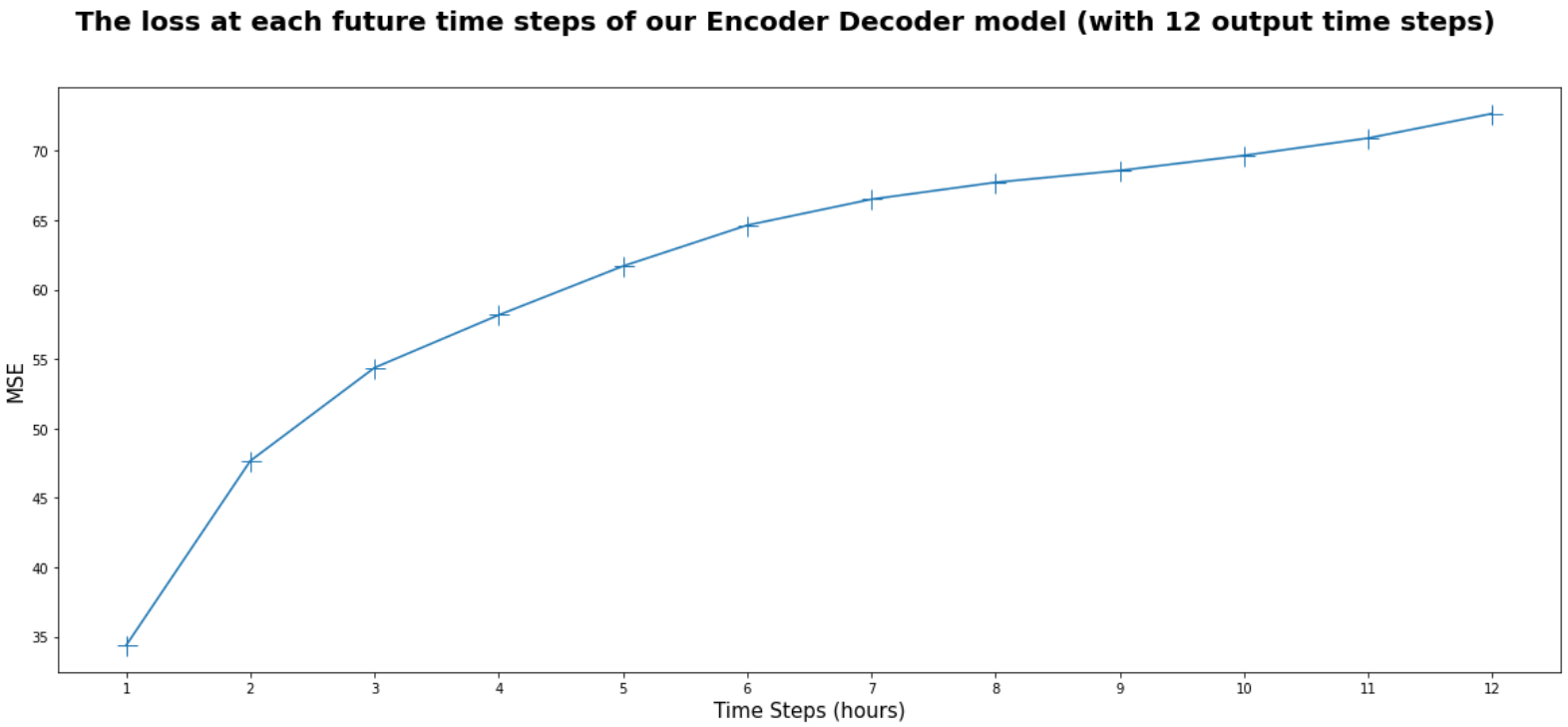


Figure 17: The Encoder-Decoder's MSE loss at each future time step on our test set

Figure 17 denotes the MSE *loss at each future time step of our Encoder-Decoder model with 12 output time steps.* As we can see from table 13 and figure 17, the accuracy degenerates heavily as we get further into future predictions.

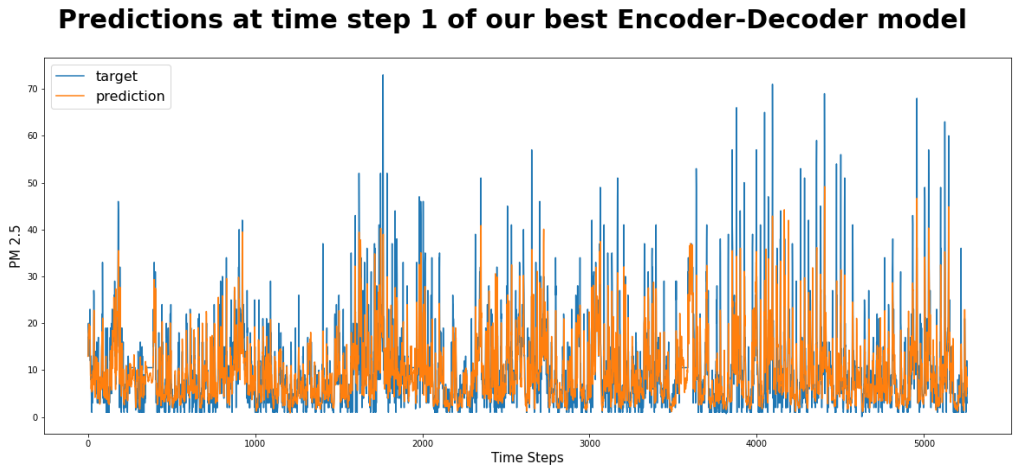


Figure 18: The Encoder-Decoder predictions at the first future time step on our test set

The prediction of the first hour into the future is quite good. As we can see, the model's prediction generally fits quite well with the time series, and while not getting too close it can still detect and approach extreme outliers. The criterions comparable to our two baseline models. (30.92 MSE compared to 31.89 and 30.71 MSE, 3.69 MAE compared to 3.67 and 3.64 MAE, 0.35 R^2 compared to 0.61 and 0.62 R^2)

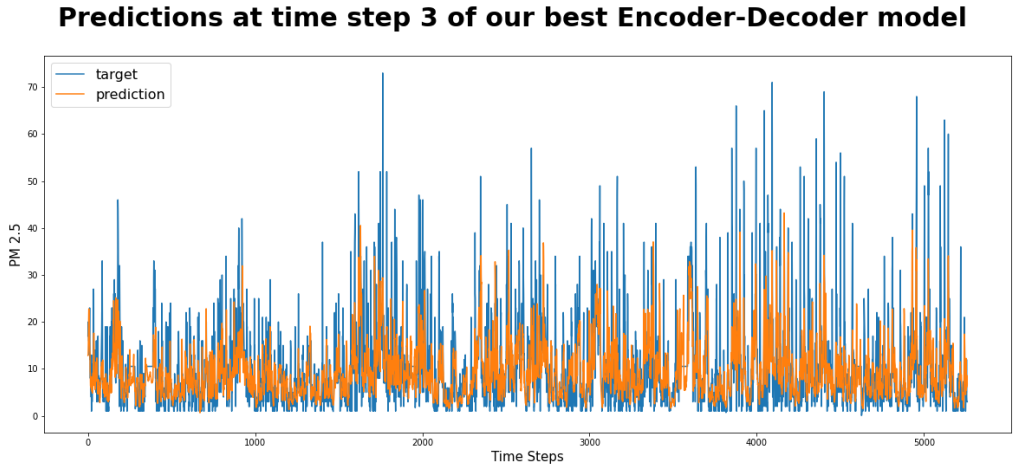


Figure 19: The Encoder-Decoder predictions at the 3^{rd} future time step on our test set

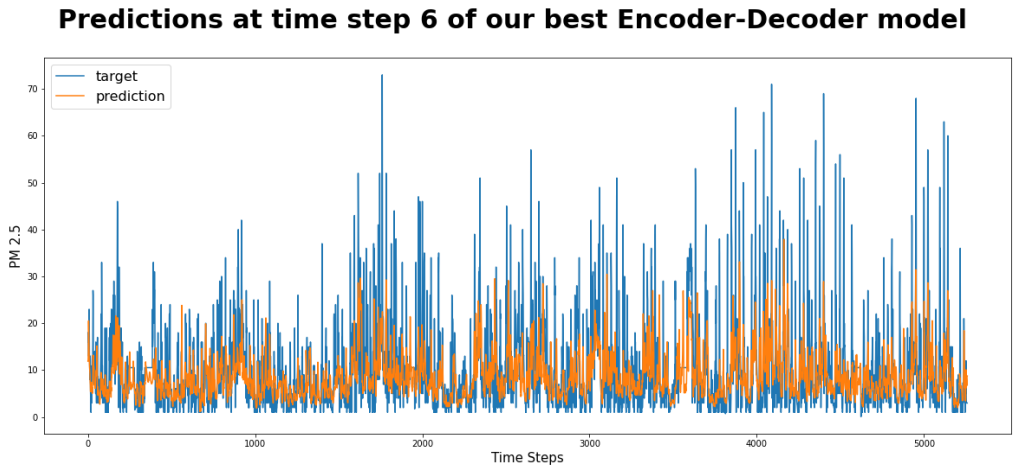


Figure 20: The Encoder-Decoder predictions at the 6^{th} future time step on our test set

As we predict further into the future hours, particularly 3 and 6 hours, we can see that the model's accuracy decreases significantly. All the criterions have worsened considerably (30.92 MSE to 50.79 and 60.225 MSE…). We can clearly notice that the predictions don't fit quite well compared to the first hour anymore. However, the model still manages to predict the general pattern of the test dataset, and still manages to predict the sudden increases as well.

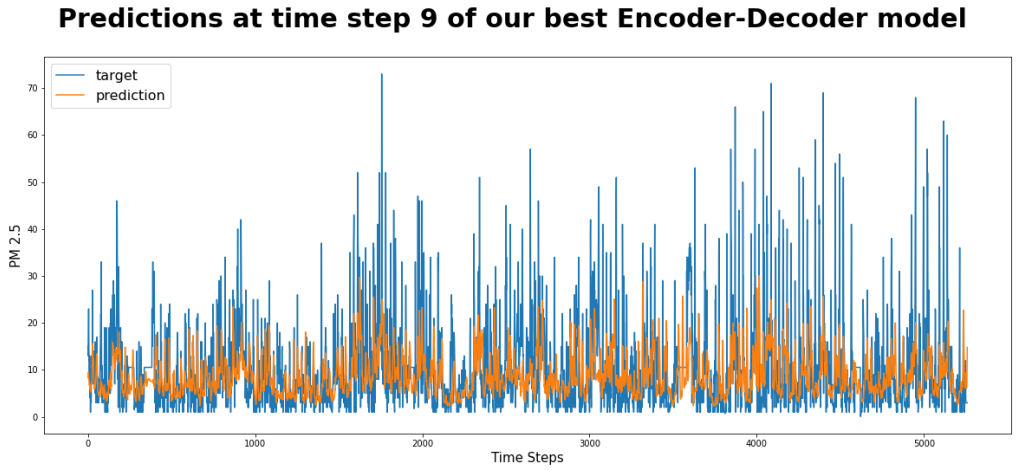


Figure 21: The Encoder-Decoder predictions at the 9^{th} future time step on our test set

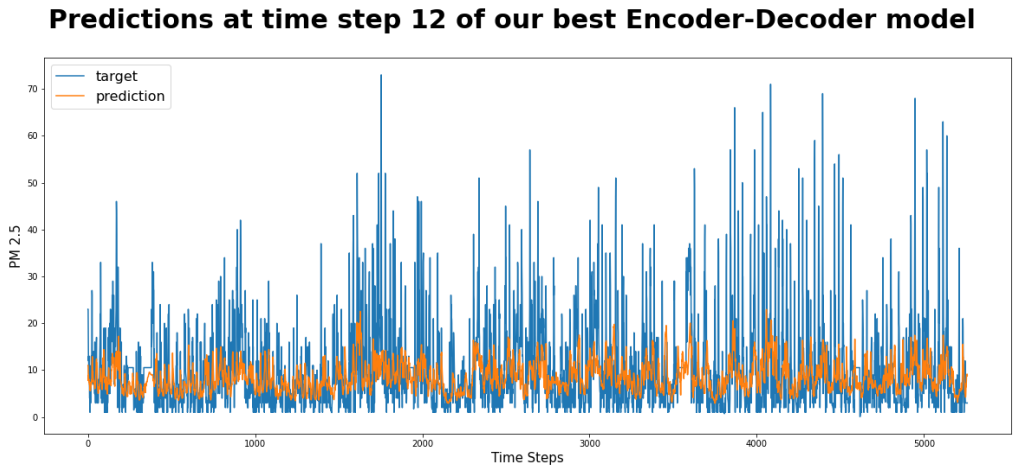


Figure 22: The Encoder-Decoder predictions at the 12^{th} future time step on our test set

As we get to 9 and 12 hours ahead, the model's performance has worsened even more considerably. All the criterions are significantly low, (69.4 MSE for the 9^{th} hour, 72.68 MSE for the 12^{th} hour). Especially for the 12^{th} hour predictions, where the model's output barely fits with the target time series anymore. However, we can see the models still can recognize the basic up and down patterns of our data, and rises up in some cases of outliers, thus our model at the 12^{th} hour might not be completely useless.

6. Conclusion

6.1. Comments

In this project, we built three different Machine Learning and Deep Learning models to forecast the PM 2.5 pollution index for the next future hour with all models achieving relatively good performance. Our main Encoder-Decoder can also forecast the pollution for multiple future time steps, however with a decline in accuracy as the future time step increases. The Encoder models' performance are still relatively acceptable for short future time steps (about below 6 hours), but after that the models' predictions differ

considerably from the ground truth, while the main patterns of the pollution level are still recognizable.

Forecasting air pollution for multiple future hours can be a quite valuable feature, which can help local authorities to prepare to deal with air pollution in advance with necessary measures. Our current Encoder-Decoder model's accuracy degenerates the longer the time distances grow, making it only suitable for short-term forecasting (about under 6 hours), however in many cases the public might require predicting the pollution for days in advance. This leaves our project a lot to be desired, as we want to develop more sophisticated and effective models that have better long term forecasting capability.

6.2. Future works

As we stated, our works in this project still have plenty of room for improvement. One particular possible approach is implementing more advanced Encoder-Decoder architectures, such as the Transformer, which utilises the Multi-head Attention mechanism based on the Self-Attention that can handle long-range dependencies effectively without using RNNs or convolutional layers.

When it comes to air pollution prediction, geographical location might be an important factor to take into account. To forecast one station, we can find a way to include the information on the other stations as well, such as the environmental indices in neighbouring stations, or distances between these stations. Models based on the Graph Neural Network (GNN) might be suitable for this task, with each station and their attribute serving as a node.

7. Appendix

7.1. Singular Spectrum Analysis

7.1.1. Singular value decomposition

Singular value decomposition (SVD) is a very well-known matrix factorization technique. It decomposes a matrix into three different components:

$$X = U\Sigma V^T$$

Where X is an arbitrary matrix, U and V are orthogonal matrices ($UU^{-1} = I$ and $VV^{-1} = I$) and $\boldsymbol{\Sigma}$ is a diagonal matrix. The significance of this method is that it allows us to represent the matrix X as the sum of d elementary matrices:

$$X = \sum_{j=0}^{d} X_i$$

Where X is the i elementary matrix of X. The elementary matrices X_i are ordered by how much they contribute to the original matrix X. The lower i is, the more significant the elementary matrix is.

7.1.2. Singular spectrum analysis

Singular spectrum analysis (SSA) is a method for decomposing a time series into a set of components. The sum of these components is exactly the original time series. Using this technique, we can remove the components that are considered as noises to obtain a smoother time series with less extreme outliers. Given a time series:

$$F = \{f_0, f_1, ..., f_{N-1}\}$$

with length N, .SSA consists of the following:

- Map the time series F to a sequence of multi-dimensional series X_0 , X_1,..., X_{N-L} with vectors:

  $$X_i = (f_i, ..., f_{i+L-1})$$

  where L is the window length $(2 \leq L \leq N/2)$ .The result of this step is the trajectory matrix X:

  $$X = \begin{bmatrix} f_0 & f_1 & f_2 & \cdots & f_{N-L} \\ f_1 & f_2 & f_3 & \cdots & f_{N-L+1} \\ f_2 & f_3 & f_4 & \cdots & f_{N-L+2} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ f_{L-1} & f_L & f_{L+1} & \cdots & f_{N-1} \end{bmatrix}$$

- Then, X is decomposed using SVD to obtain the d elementary matrices. The first l matrices are used to approximate the original time series.

References

[1] V. Vapnik. Statistical Learning Theory. John Wiley and Sons, New York, 1998.

[2] Drucker, Harris, et al. "Support vector regression machines." *Advances in neural information processing systems* 9 (1997): 155-161.

[3] Hochreiter, S., Schmidhuber, J., “Long Short-Term Memory”, Neural Computation 9 (8), 1997, pp. 1735–1780

[4] http://colah.github.io/posts/2015-08-Understanding-LSTMs

[5] 9.7. Sequence to Sequence Learning — Dive into Deep Learning 0.17.2 documentation (d2l.ai)

[6] [1409.0473] Neural Machine Translation by Jointly Learning to Align and Translate (arxiv.org)

[7] 10.3. Attention Scoring Functions — Dive into Deep Learning 0.17.2 documentation (d2l.ai)

[8] 10.4. Bahdanau Attention — Dive into Deep Learning 0.17.2 documentation (d2l.ai)

[9] Hassani, H. (2007). Singular spectrum analysis: methodology and comparison

[10] Deep Air | UC Berkeley School of Information