Project Report

Kimia Honari

***Abstract*—Time-series forecasting is a challenging problem in machine learning techniques. Several methods have been proposed for solving these kinds of problems, such as Recurrent Neural Network and LSTM. In this project, we compared two common models, such as LSTM and CNN, to predict time-series data. We investigated our model with two dynamic gas sensor datasets and measured their performance with three common metrics in forecasting. We found that both models could achieve a good performance on time-series datasets, but the CNN model is much faster than LSTM in the learning process.**

I. Introduction

Time series forecasting is a very important practical application with a diverse range of applications, including economic and business planning, inventory and production control, weather forecasting, signal processing, etc. Time series analysis comprises methods for analyzing time-series data to extract meaningful statistics and other characteristics of the data and using a model to predict future values based on previously observed values. Recurrent neural networks (RNNs) or LSTMs ("long short-term memory" units) are the most powerful and well-known subset type of artificial neural network designed to recognize patterns in sequences of data, such as numerical times series data emanating from sensors, stock markets, and government agencies (but also including text, genomes, handwriting, and the spoken word). RNNs are like short-term memory can remember things that happen in a previous couple of observations and apply that knowledge in the current inputs. Another type of network that can be used in time-series forecasting is a one-dimensional convolutional neural network (1D-CNN), which achieved a good performance on a sequence, and real-time datasets.

In this project, we wanted to compare 1D-CNN and LSTM networks performance on time-series dataset. To investigate our models, we used two "Gas sensor array under dynamic gas mixtures" datasets that are available in the UCI machine learning repository. The datasets have the information of 16 gas sensors in the presence of ethylene and methane, and ethylene and CO in air. We compared the accuracy of two models in a one-step-ahead forecast of all 16 streams based on previous observation.

For parametrizing the model, we need to generate the state space of previous observation and give it as an input to the model. For this purpose, we used delay embedding techniques to find the best delay and dimension for each sensor of the datasets. After preparing the data for performing the learning task, we parametrized the model three times in each dataset and measured the performance of the models with three common metrics in time-series forecasting such as Mean absolute error (MAE), Mean squared error (MSE) and Mean absolute scaled error (MASE). The final results show that although the validation and test loss is lower for the LSTM network, both 1D-CNN and LSTM can achieve excellent performance on time-series forecasting, but 1D-CNN is much faster than LSTM in parametrizing.

The remain sections organized as follows. In the next section, we will explain RNN, LSTM, and 1D-CNN architectures and the learning process for LSTM. In Section 3, we will describe the two datasets that we used in our experiment, along with the data preparation technique. In Section 4, we will elaborate on our models, metrics, and the methodology we used. .Finally, we provide our results and related work in the last two sections. The code of our implementation and the results are provided in the Appendix.

II. Network's Architecture

A. Recurrent Neural Network (RNN)

Recurrent neural networks (RNNs) are the most powerful and advanced artificial neural network designed to recognize patterns in sequences of data. All other neural networks do not have any memory, and each input processed independently, but RNNs are a generalization of a feed-forward neural network that has internal memory. The idea behind RNNs is to make use of sequential information. For example, If you want to predict the next word in a sentence, you would better know which words came before it.

Figure 1 shows the architecture of RNNs. The internal loop helps to process sequences by iterating through the sequence elements and maintaining a state containing information relative to what it has seen so far. Like traditional networks, we pass one sequence of data as a single data point, but the difference is that the data point is not processed just in a single step, and the network internally loops over sequence elements.

Each rectangle in Figure 1 represents a whole layer of neurons that connected to themselves through time. The neurons have some short-term memory, providing them with the possibility to remember what was in this neuron just previously. Therefore, the neurons can pass the information on to themselves in the future and analyze things. A generic recurrent neural network equation is as follows:

$$output\_t = activation(W.input\_t + U.state\_t + bo) \quad (1)$$

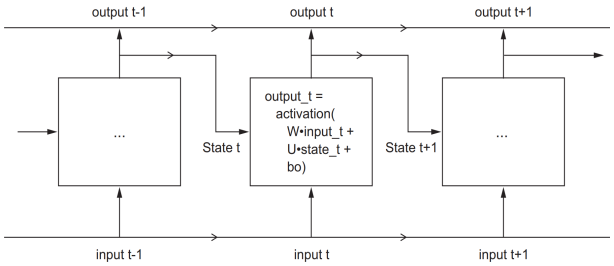

Fig. 1: Simple RNN architecture. [1]

U is the weight for the previous hidden state, W is the weight for the current input state, and bo is the bias. RNN can not process long sequence if we use tanh or relu as an activation function.

When we train the RNN model by gradient-based optimization techniques [2], the vanishing gradient problem may occur. This problem first time introduced by Sepp (Joseph) Hochreiter back in 1991 [3]. In each iteration of the gradient method, weight update by the partial derivative of the error function respect to the current weight. In some cases, the gradient will be vanishingly small, which prevents the weight from changing or in a worse case, stop the training.

1) Long Short-Term Memory (LSTM): Long Short-Term Memory (LSTM) networks are a modified version of recurrent neural networks introduced by Sepp Hochreiter in 1997 as a solution to the vanishing problem. Figure 2 shows the architecture of LSTM. LSTM had a three Module called Forget gate, Input gate, Output gate. LSTM has a Forget layer additional to RNN, which decides what information was going to throw away from the cell state and save for later.

As you can see in Figure 2, LSTM has an additional data flow that carries information across timestamps. This information will be combined with the input connection and the recurrent connection via a dot product with a weight matrix followed by a bias add and the application of an activation function, which affects the state being sent to the next timestamp.

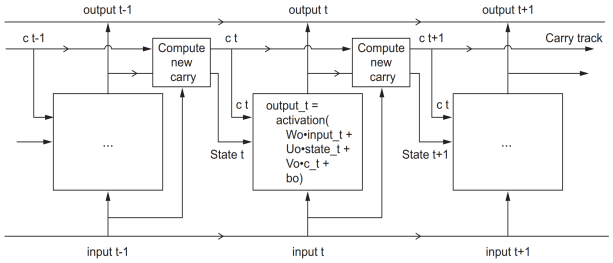

Fig. 2: LSTM architecture. [1]

B. Forward pass Backward pass in LSTM

We briefly defined LSTM architecture in the previous section. In this section, we will elaborate on the mathematics behind the LSTM learning procedure.

1) Forward Pass: Initially, at time t, the memory cells of the LSTM contain values from the previous iteration at time $(t-1)$. At time t, The LSTM receives a new input vector x^t (including the bias term), as well as a vector of its output at the previous time step, h^{t-1}.The gates are defined as:

$$Input\ activation = a^t = tanh(W_a x^t + U_a h^{t-1} + b_a) \quad (2)$$

$$Input\ gate = i^t = \phi(W_i x^t + U_i h^{t-1} + b_i) \quad (3)$$

$$Forget\ gate = f^t = \phi(W_f x^t + U_f h^{t-1} + b_f) \quad (4)$$

$$Output\ gate = o^t = \phi(W_o x^t + U_o h^{t-1} + b_o) \quad (5)$$

If the input x^t is of size $n*1$, and we have d memory cells, then the size of each of W_* and U_* is $d*n$, and $d*d$ resp. Note that each one of the d memory cells has its own weights W_* and U_* , and that the only time memory cell values are shared with other LSTM units is during the product with U_*.

$$\begin{bmatrix} W_a & U_a \\ W_i & U_i \\ W_f & U_f \\ W_o & U_o \end{bmatrix} * \begin{bmatrix} x^t \\ h^{t-1} \end{bmatrix} = W * I^t = gate^t = z^t = [a^t, i^t, f^t, o^t]^T \quad (6)$$

The values of the memory cells are updated with a combination of a^t, and the previous cell contents c^{t1}. The combination is based on the magnitudes of the input gate i^t and the forget gate f^t. $\otimes$ denotes element wise product (Hadamard product).

$$c^t = i^t \otimes a^t + f^t \otimes c^{t-1} \quad (7)$$

Finally, the LSTM cell computes an output value by passing the updated (and current) cell value through a non-linearity. The output gate determines how much of this computed output is actually passed out of the cell as the final output h^t.

$$h^t = o^t \otimes tanh(c^t) \quad (8)$$

2) Backward pass: The cell state at time t, c^t receives gradients from h^t as well as the next cell state c^{t+1}. The next equations focus on computing these two gradients. At any time step t, these two gradients are accumulated before being back-propagated to the layers below the cell and the previous time steps.consider we have:

Δ_t is the output difference as computed by any subsequent layers (i.e. the rest of your network), and;

Δ_{out} the output difference as computed by the next time-step LSTM (the equation for t-1 is below).

Therefore we have:

$$\delta h^t = \Delta_t + \Delta_{out} \quad (8)$$

$$\delta c^t = \delta h^t \otimes o^t \otimes (1 - tanh^2(c^t)) + c^{t+1} \otimes f^{t+1} \quad (9)$$

$$\delta a^t = \delta c^t \otimes i^t \otimes (1 - (a^t)^2) \tag{10}$$

$$\delta i^t = \delta c^t \otimes a^t \otimes i^t \otimes (1 - i^t) \tag{11}$$

$$\delta f^t = \delta c^t \otimes \delta c^{t-1} \otimes f^t \otimes (1 - f^t) \tag{12}$$

$$\delta o^t = \delta h^t \otimes tanh(c^t) \otimes o^t \otimes (1 - o^t) \tag{13}$$

$$\delta o^t = \delta h^t \otimes tanh(c^t) \otimes o^t \otimes (1 - o^t) \tag{14}$$

$$\delta I^t = W^T * \delta z^t (\delta h^{t-1} \; can \; be \; retrieved \; from \; \delta I^t) \tag{15}$$

If input x has T time-steps,W is then updated using an appropriate Stochastic Gradient Descent solver.

$$\delta W^t = \delta z^t * (I^t)^T \tag{16}$$

$$\delta W = \sum_{t=1}^{T} \delta W^t \tag{17}$$

C. One-dimensional Convolutional Neural Network

When we say Convolution Neural Network (CNN), generally, we refer to a 2-dimensional CNN used for image classification. But there are two other types of Convolution Neural Networks used in the real world, which are 1-dimensional and 3-dimensional CNNs. In this project, we used 1D CNN to predict the value of time series sensors.

1D CNN can perform activity recognition tasks from sensory data or accelerometer data, such as if the person is standing, walking, jumping, etc. The data in this situation has two dimensions. The first dimension is time-steps, and the other is the value of variable the time series. Figure 3 shows the 1D CNN and how the kernel will move on it.

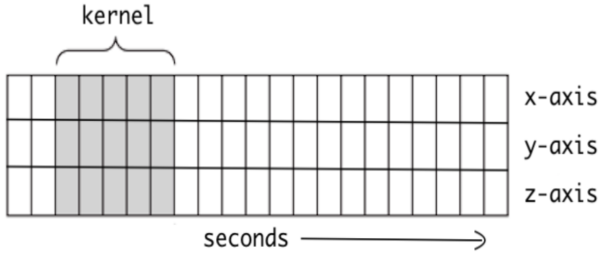

Fig. 3: kernel sliding on 1D-CNN with 3 variables. [1]

III. Dataset Description and Preparation

A. Dataset

We compared our models with the dataset called Gas sensor array under dynamic gas mixtures that is available in the UCI machine learning repository [14]. This dataset contains the information of 16 chemical sensors in the presence of time-varying mixtures of ethylene and methane or ethylene and CO in air. The values of sensors are gathered in 12 hours without interruption. The dataset has been collected at the ChemoSignals Laboratory in the BioCircuits Institute, University of California, and they used four different types of chemical sensors called TGS-2600, TGS-2602, TGS-2610, TGS-2620 for gathering the values. This dataset has two files one related to ethylene and methane and another related to ethylene and CO. each file has about 4 million rows and 19 columns. The first column shows the time in second that usually changed every 0.01 seconds. The next two columns show the value of gas concentration in air, and finally, the other 16 column contains the value of sensors.

B. Data Preparation

For preparing the data, we separated the 16 values of sensors from the file and normalized the data with the max-min technique. As the objective of this paper is time-series forecasting, we need to give the network some set of previous observations as an input to predict one or more future observations. Thus, there is no fixed distinction between input and output variables; all or most of the data will probably be an input and an output at different points in the course of a forecasting experiment. Machine Learning algorithm can accept a fixed-length feature vector as an input and determines the output variable based on that; Therefore, For prediction with our model, we need to convert the time series variables into a suitable form.

There is a common approach called delay embedding introduced in 1981 by Takens [4] that helps us to prepare the time-series data for prediction. In the first state of this theory, we need to reconstruct state space, which includes variables of a small set of the most recent previous observation. To find the best set, Takens considered a noise-free situation, focusing on delay coordinate maps or predictive models that are constructed from a time series, representing an observable from a dynamic system. A value of delay is so important because we always have to deal with an amount of noisy data. Hence, For reconstructing the state space, we need to find two parameters: The delay parameter d, which is the lag at which the time series has to be plotted against itself, and the embedding dimension parameter D, where D-1 is the number of times that the time series has to be plotted against itself using the delay d. In 1986, Fraser and Swinney [5] developed a method called mutual to find optimal time delayed coordinates that are as independent of each other as possible. The mutual information between the original time series x(t) and the time series x(t + d) shifted by d, is computed as follows:

$$I(x(t), x(t+d)) = \sum_{i,j} p_{ij}(d) log(\frac{p_{ij}(d)}{p_i p_j}) \quad (18)$$

p_i is the probability that x(t) is in bin i of the histogram constructed from the data points in x, and $p_{ij}(d)$ is the probability that $x(t)$ is in bin i and $x(t+d)$is in bin j. To obtain coordinates for time-delay embedding that are as independent as possible, Fraser and Swinney proposed using the position of the first minimum of the mutual diagram as the optimal value of d. The first minimum value shows the first coordinate x(t) of state space is maximally independent of the second coordinate x(t+d). In some cases, the mutual function does not have a local minimum, and it may be a monotonically decreasing function. Therefore, in this situation, we consider the d, the value of mutual function drops below the value $\frac{1}{e}$ [6].

This method has been designed for one-dimensional data, and for our project, we have 16 sensors. Several methods have been proposed to generalize this procedure for estimating the time delay to the case of multidimensional time series like [7], [8], and [9], but we compute the delay for each sensor separately. Figure 4 shows the plot of mutual value based on delay for the ethylene_methane dataset; most of the best value for each sensor in this dataset is around 7000. Figure 5 shows the mutual value for the ethylene_CO dataset. The value of delay for each sensor is available in Tables 1 and 2.

According to Takens' theorem, we can construct a time series y(t) of m-dimensional points from the original one-dimensional time series x(t) as follows:

$$y(t) = (x(t), x(t+d), \ldots, x(t+(m1)d) \quad (19)$$

Here, both t and d are integers used to index the sampled data, but they can be expressed in units of real-time when multiplied by the sampling interval. For calculating the dimension m, There is an approach that examines the change in distance between neighboring points in state space, as we progressively embed the original time series into higher dimensions. The basic idea underlying the estimation of the embedding dimension using False Nearest Neighbor (FNN) was proposed by Kennel [10]. If we embed the time series once using some time delay d, then we can calculate the distance between them to find if it has changed. If embedding changes the distance between the neighbors significantly, then it called false neighbors, and this indicates that the data need to be embedded further. If their distance does not change appreciably, then it called true neighbors, meaning that the current embedding dimension is sufficient. The following equation shows m-dimensional state space and denotes the rth nearest neighbor of a coordinate vector $y(t)$ by $y^{(r)}(t)$, then the square of the Euclidean distance between $y(t)$ and the rth nearest neighbor is:

$$R_m^2(t,r) = \sum_{k=0}^{m-1} [x(t+md) - x^{(r)}(t+md)]^2 \quad (20)$$

We calculated the false nearest neighbor for each sensor for dimensions between 1 to 20. Figures 5 and 6 show the false nearest neighbor for two datasets. Almost all of them have a decreasing function, and 20 is the best dimension for them. For computing embedding parameters, we first under sampled the data (10% of data). Then we used Tisean, which is a software project for the analysis of time series with methods based on the theory of nonlinear deterministic dynamical systems [11]. For computing the delay parameters, we used mutual function and estimated the best dimension with FNN function.

After finding the parameters, we split the data into training and test dataset. We considered 2/3rd data for training and reserved the rest of the data for testing. We also split the validation data, which contains 10 percent of the last recent data in the dataset. Then, based on the sample generator code, exist in Chollet book [1], we generated the state space based on the parameters m and d for each sensor. Details of implementation are available in Appendix A.

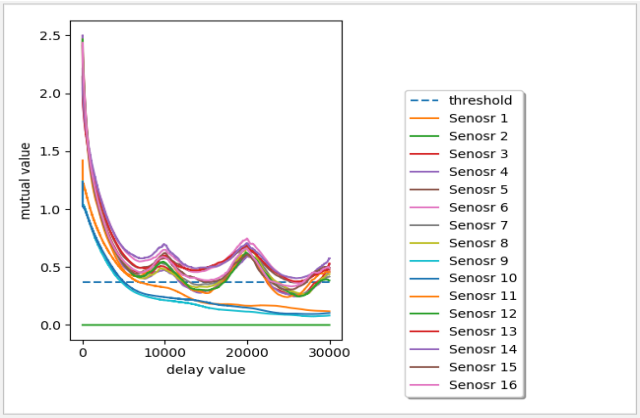

Fig. 4: Mutual value VS delay for ethylene_methane dataset.threshold shows the $1/e$ value.

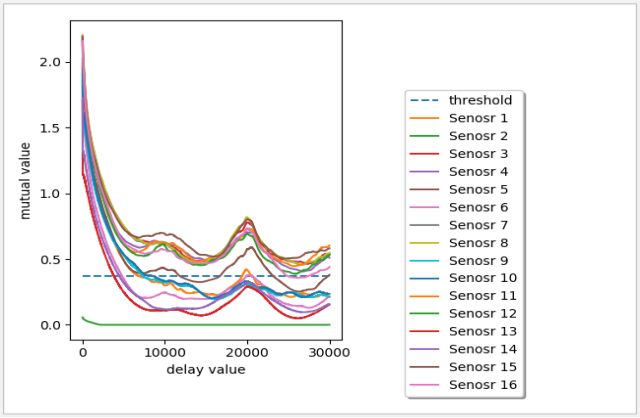

Fig. 5: Mutual value VS delay for ethylene_CO dataset.threshold shows the $1/e$ value.

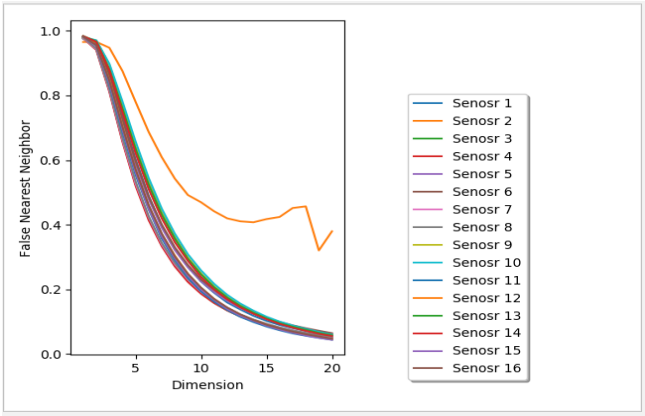

Fig. 6: False Nearest Neighbor VS Dimension for ethylene_methane dataset.

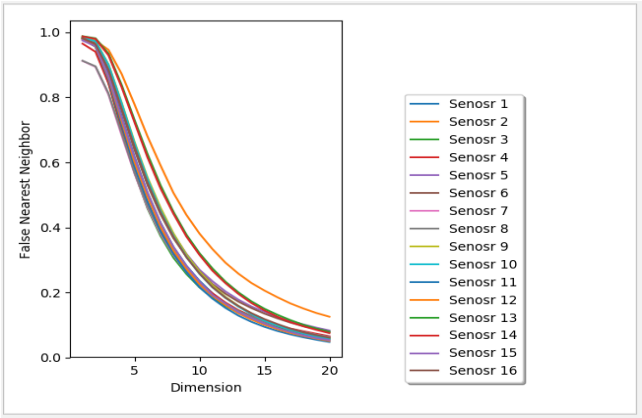

Fig. 7: False Nearest Neighbor VS Dimension for ethylene_CO dataset.

#sensor	delay	dimension (m)
1	6960	20
2	1	19
3	8000	20
4	8000	20
5	7000	20
6	7000	20
7	7000	20
8	4760	20
9	5080	20
10	7000	20
11	7000	20
12	7000	20
13	7000	20
14	7000	20
15	7000	20
16	7000	20

TABLE I: Embedding parameters for ethylene_methane dataset.

#sensor	delay	dimension (m)
1	6950	20
2	2500	20
3	15000	20
4	15000	20
5	11880	20
6	4830	20
7	14000	20
8	14000	20
9	7420	20
10	8650	20
11	6000	20
12	7000	20
13	3830	20
14	4480	20
15	16000	20
16	15000	20

TABLE II: Embedding parameters for ethylene_CO dataset.

IV. Methodology

In this section, we will explain the architecture of our two model, the metrics that we used to compare the performance of the models, and the learning procedure.

A. *Forecast Error Metrics for Time Series*

Three forecasting situations may happen in the forecasting method. First, We may generate forecasts for a single future period using multiple data series, such as a collection of products or items. Second, we can generate a series of one-period-ahead forecasts $F_{1+h}, ..., F_{m+h}$ where each F_{j+h} is based on data from times $t = 1, ..., j$. At last, We can compute forecasts from a common origin t for a sequence of forecast horizons $F_{n+1}, ..., F_{n+m}$ based on data from times $t = 1, ..., n$. This is the standard procedure implemented by forecasters in real-time.

It is useful to have a forecast accuracy metric that can be used for all three cases. Four types of forecast-error metrics are available: Scale-dependent metrics such as the mean absolute error (MAE), Relative-error metric, percentage-error metrics, and scale-free metrics. Both percentage-error and relative-error metrics are scale-independent, so they are frequently used to compare forecast performance between different data series, but they have a disadvantage of being infinite or undefined if there are zero values in a series. The scale-free error metric like the mean absolute scaled error (MASE) can be well suited to time series data, and it never gives infinite or undefined value. Based on the situation, each type of metric can be preferred. For this project, we picked three performance measurement techniques, which we will elaborate on and explain our reason in the following.

The first metric we used is the mean absolute error or MAE, which is calculated as the average of the forecast error values, where all of the forecast values are forced to be positive. We used this method because it is so simple and easy to understand and compute. As both the dataset and sensors are on the same scale and positive, and we normalized them, this metric is a good choice to measure the performance of our networks. Besides, it is a common measure of forecast error in time series

analysis. The following equation shows the MAE formula if Y_t denotes the observation at time t and F_t denotes the forecast of Y_t:

$$e_t = |Y_t - F_t| \tag{21}$$

$$MAE = mean(e_t) \tag{22}$$

The second metric is mean squared error (MSE), which is calculated as the average of the squared forecast error values. Squaring the forecast error values forces them to be positive; it also has the effect of putting more weight on large errors. Very large or outlier forecast errors are squared, which in turn has the effect of dragging the mean of the squared forecast errors out, resulting in a larger mean squared error score. In effect, the score gives worse performance to those models that make large wrong forecasts. The MSE is another common scale-dependent forecasting metrics in time series and can show us if the networks have a large and outlier forecast error.

$$MSE = mean(e_t^2) \tag{23}$$

At last, the third metrics we used is mean absolute scaled error, MASE, which was proposed by Hyndman and Koehler (2006) [12] as a generally applicable measurement of forecast accuracy without the problems seen in the other measurements. They proposed scaling the errors based on the in-sample MAE from the naïve forecast method. Using the naïve method, we generate one-period-ahead forecasts from each data point in the sample. Accordingly, a scaled error is defined as:

$$MASE = mean(\frac{e_t}{\frac{1}{n-1}\sum_{i=2}^{n}(Y_i - Y_{i-1})}) \tag{8}$$

The result is independent of the scale of the data. A scaled error is less than one if it arises from a better forecast than the average one-step, naïve forecast computed in the sample. Conversely, it is greater than one if the forecast is worse than the average one-step, naïve forecast computed in-sample. We chose MASE because it works very well in situations where there are very different scales, including data that are close to zero or negative. Besides, it is scale-free, and the only available accuracy measurement that can be used in all three forecasting situations described in the first paragraph of this section.

B. Networks' Architecture

In this project, we used two networks for forecasting the time series data. In the following, we elaborate more on the detail of each network.

The first network has three-layer, one input, one hidden layer, and one output layer. The hidden layer is a one-dimensional CNN, which has 120 feature maps, and a kernel size of 1. The hidden layer's activation function is ReLU and connected to the output layer via a flatten layer. The second network is based on the LSTM, and it has three layers like the first one, but instead of 1D-CNN in the hidden layer, we have the LSTM with the size of 120. In both networks, the size of input for the ethylene_methane dataset is a state-space vector with the size of (1,319), and for ethylene_CO dataset is (1,320). The size of the output layer is (1,16) and uses the sigmoid activation function. The details of architecture, codes, and the number of parameters for each network is available in the Appendix B. The reason for using the same size for both models is to have a fair comparison, and the size of 120 for hidden layer picked after testing a range of sizes, and we found that 120 is the best and optimal size for our models.

C. Parameter Exploration

In this project for the optimizer, we used RMSprop, which is an unpublished optimization algorithm designed for neural networks, first proposed by Geoff Hinton in lecture 6 of the online course "Neural Networks for Machine Learning" [13]. The RMSprop optimizer is similar to the gradient descent algorithm with momentum. The RMSprop optimizer restricts the oscillations in the vertical direction. Therefore, we can increase our learning rate, and our algorithm could take larger steps in the horizontal direction converging faster. In the time series problem that we explore in this project, RMSprop could achieve a good performance, so we used it to parametrize both networks.

We also used a callback function that available in the Keras called ReduceLROnPlateau to decay the learning rate by monitoring the validation loss. This callback monitors a value, and if no improvement is seen for a 5 number of epochs, the learning rate is reduced by multiplying to 0.3. Appendix C shows the implementation of callback functions.

One of the other hyper-parameter values that we need to choose is the number of batch size and epochs; after some experiment on different batch size value, we found out that the 128 is the best value for our model and dataset. We set the high number 10, but use to Keras callbacks to control our models. The first function is ModelCheckpoint, we used this function to monitor validation loss, and it will save the best model with the epoch number and the minimum validation loss in the filename. The other one is EarlyStopping, which monitors the validation loss, and if the quantity has stopped improving, it stops the training. At last, we measure the performance of our models with three metrics of MSE, MAE, and MASE. We used the MSE, and MAE loss function that are available in the Keras library, and implement a custom loss function of MASE. Appendix C show the implementation of MASE loss function.

V. Related Work

RNNs have recently shown promising results in a variety of applications, especially when there exist sequential dependencies in data [15] [16]. Long short-term memory (LSTM) [17] [15], a class of recurrent neural networks with sophisticated recurrent hidden and gated units, are particularly successful and popular due to its ability to learn hidden long-term sequential dependencies. [18] uses LSTMs to recognize patterns in multivariate time series, especially for multi-label classification of diagnoses.

CNN's are often used to learn an effective representation of local salience from raw data [19]. [20] and [21] make use of CNNs to extract features from raw time-series data for activity/action recognition. [22] focuses on the prediction of periodical time series values by using CNN and embedding time series with neighbors in the temporal domain. In this experiment, we compared these two architectures in forecasting two time-series datasets. our experiments show that both of these networks can achieve good performance on time-series forecasting.

VI. Result and Conclusion

For each dataset, we parametrize our model three times, and each time measured their performance with different metrics that we mentioned in the methodology section. At last, we ran the model another time and computed the validation loss of training, validation, and testing dataset with the metrics. As you can see in Appendix D, the LSTM network has a lower validation loss rather than 1D-CNN, but 1D-CNN performance is as good as the LSTM network. The advantage of 1D-CNN is the time that it takes to parametrize the model, which is nearly half of the time that we need to parametrize the LSTM network.

References

[1] Chollet, F. (2018). Deep Learning mit Python und Keras: Das Praxis-Handbuch vom Entwickler der Keras-Bibliothek. MITP-Verlags GmbH Co. KG.

[2] Pascanu, R., Mikolov, T., Bengio, Y. (2013, February). On the difficulty of training recurrent neural networks. In International conference on machine learning (pp. 1310-1318).

[3] Hochreiter, S. (1998). The vanishing gradient problem during learning recurrent neural nets and problem solutions. International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems, 6(02), 107-116.

[4] Takens, F. (1981). Detecting strange attractors in turbulence. In Dynamical systems and turbulence, Warwick 1980 (pp. 366-381). Springer, Berlin, Heidelberg.

[5] Fraser, A. (1986). M. and Swinney, HL "Independent coordinates for strange attractors from mutual information". Phys. Rev. A, 33, 1134-1140.

[6] Kantz, H., Schreiber, T. (2004). Nonlinear time series analysis (Vol. 7). Cambridge university press.

[7] Garcia, S. P., Almeida, J. S. (2005). Multivariate phase space reconstruction by nearest neighbor embedding with different time delays. Physical Review E, 72(2), 027205.

[8] Hirata, Y., Suzuki, H., Aihara, K. (2006). Reconstructing state spaces from multivariate data using variable delays. Physical Review E, 74(2), 026202.

[9] Vlachos, I., Kugiumtzis, D. (2009). State space reconstruction from multiple time series. In Topics on Chaotic Systems: Selected Papers from Chaos 2008 International Conference (pp. 378-387).

[10] Kennel, M. B., Brown, R., Abarbanel, H. D. (1992). Determining embedding dimension for phase-space reconstruction using a geometrical construction. Physical review A, 45(6), 3403.

[11] https://www.pks.mpg.de/ tisean/Tisean_3.0.1/index.html

[12] Hyndman, R. J., Koehler, A. B. (2006). Another look at measures of forecast accuracy. International journal of forecasting, 22(4), 679-688.

[13] Geoffrey Hinton Neural Networks for machine learning nline course. https://www.coursera.org/learn/neural-networks/home/welcome

[14] https://archive.ics.uci.edu/ml/datasets

[15] Chung, J., Gulcehre, C., Cho, K., Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. arXiv preprint arXiv:1412.3555.

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

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

[18] Lipton, Z. C., Berkowitz, J., Elkan, C. (2015). A critical review of recurrent neural networks for sequence learning. arXiv preprint arXiv:1506.00019.

[19] Bowman, S. R., Vilnis, L., Vinyals, O., Dai, A. M., Jozefowicz, R., Bengio, S. (2015). Generating sentences from a continuous space. arXiv preprint arXiv:1511.06349.

[20] Hammerla, N. Y., Halloran, S., Plötz, T. (2016). Deep, convolutional, and recurrent models for human activity recognition using wearables. arXiv preprint arXiv:1604.08880.

[21] Yang, C. M., Liu, H. W. (2015). U.S. Patent No. 9,214,987. Washington, DC: U.S. Patent and Trademark Office.

[22] Liu, L., Oza, S., Hogan, D., Perin, J., Rudan, I., Lawn, J. E., ... Black, R. E. (2015). Global, regional, and national causes of child mortality in 2000–13, with projections to inform post-2015 priorities: an updated systematic analysis. The Lancet, 385(9966), 430-440.

APPENDIX A
DATA PREPARATION

In this section Figure 8 shows the normalization and reading the data from file, and figure 9 shows the datagenerator function that we implemented to generate the state space.

```
#reading the data
fname = 'ethylene_CO.txt'
f = open(fname)
data = f.read()
f.close()
lines = data.split('\n')
header = lines[0].split(',')
lines = lines[1:]
import numpy as np
float_data = np.zeros((len(lines), 16))
d=0
for d, line in enumerate(lines):
    p=[]
    for x in line.split(' '):
        if x=='':
            continue
        p.append(x)
    if len(p)==0:
        continue
    values = [float(x) for x in p[3:]]
    float_data[d, :] = values

#split the data to training,testing, and validation set
print(float_data[0])
size=len(float_data)
training_size=int(2*size/3)
validation_size=int(1/10*size)
testing_size=(size-training_size)-validation_size

#normalization2
mi=float_data.min(axis=0)
ma=float_data.max(axis=0)
float_data-=mi
float_data/=(ma-mi)
```

Fig. 8: Reading data from file and normalize it with max-min method.

```
def dgenerator(data, dimensionarray, delay, min_index, max_index, batch_size, darray, shuffle=False):
    if max_index is None:
        max_index = len(data) - delay - 1
    samples = np.empty((0, batch_size, sum(dimensionarray)), float)
    targets = np.empty((0, batch_size, 16), float)
    max = 0
    for sensor in range(0, 16):
        dimension = dimensionarray[sensor]

        d = darray[sensor]
        if max < (dimension * d):
            max = dimension * d
    i = min_index + max
    while 1:
        if i + batch_size >= max_index:
            break
        rows = np.arange(i, min(i + batch_size, max_index))
        target = []
        sample = []
        for b in range(0, len(rows)):
            i = i + 1
            sa = []
            for sensor in range(0, 16):
                dimension = dimensionarray[sensor]
                d = darray[sensor]
                c = i
                for step in range(0, dimension):
                    c -= d
                    sa.append(data[c][sensor])
            sample.append(sa)
            target.append(data[i])
        samples = np.append(samples, np.array([sample]), axis=0)
        targets = np.append(targets, np.array([target]), axis=0)

    np.save("test_x_CO.npy", samples)
    np.save("test_y_CO.npy", targets)
    return samples, targets
```

Fig. 9: Generating the state space based on delay embedding parameters of each sensor.

APPENDIX B
NETWORKS' ARCHITECTURE

1D-CNN model:

```
model = Sequential()

model.add(layers.Conv1D(120,1, activation='relu',input_shape=(1,320)))
model.add(Flatten())
model.add(layers.Dense(16, activation="sigmoid"))

model.summary()
```

Fig. 10: Creating 1D-CNN model, keras code.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv1d_1 (Conv1D)            (None, 1, 120)            38400
_________________________________________________________________
flatten_1 (Flatten)          (None, 120)               0
_________________________________________________________________
dense_1 (Dense)              (None, 16)                1936
=================================================================
Total params: 40,336
Trainable params: 40,336
Non-trainable params: 0
_________________________________________________________________
```

Fig. 11: 1D-CNN parameter for learning.

LSTM model:

```
model = Sequential()

model.add(layers.LSTM(120,input_shape=(1,320)))

model.add(layers.Dense(16,activation="sigmoid"))

model.summary()
```

Fig. 12: Creating LSTM model, keras code.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_1 (LSTM)                (None, 120)               211200
_________________________________________________________________
dense_1 (Dense)              (None, 16)                1936
=================================================================
Total params: 213,136
Trainable params: 213,136
Non-trainable params: 0
_________________________________________________________________
```

Fig. 13: LSTM parameter for learning.

APPENDIX C
COMPILE MODEL AND CALLBACK FUNCTION

```
x_train=np.load("training_x_CO.npy")
y_train=np.load("training_y_CO.npy")

x_val=np.load("val_x_CO.npy")
y_val=np.load("val_y_CO.npy")
print(x_val.shape)

x=x_train.reshape(19418*128,320)
x=np.expand_dims(x, axis=1)
y=y_train.reshape(19418*128,16)

xv=x_val.reshape(787*128,320)
xv=np.expand_dims(xv, axis=1)
yv=y_val.reshape(787*128,16)

callbacks = get_callbacks()
model.compile(optimizer=keras.optimizers.RMSprop(), loss="mae")
history =    model.fit(x, y,
              batch_size=128,
              epochs=10,
              verbose=2,
              validation_data=(xv, yv),
              callbacks=callbacks)

import matplotlib.pyplot as plt
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(loss) + 1)
plt.figure()
plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss (MAE) Ethylene_CO')
plt.legend()
plt.show()
```

Fig. 14: compile and fit the model.

```
def get_callbacks():
    model_type = "LSTM"
    save_dir = os.path.join(os.getcwd(), 'LSTM_co/mae')
    model_name = "Methane_%s_model.{epoch:03d}.h5" % model_type
    if not os.path.isdir(save_dir):
        os.makedirs(save_dir)
    filepath = os.path.join(save_dir, model_name)

    checkpoint = ModelCheckpoint(
        filepath=filepath,
        monitor='val_loss',
        verbose=1,
        save_best_only=True)


    es = EarlyStopping(monitor='val_loss', mode='min', verbose=1, patience=50)
    lr_reducer = keras.callbacks.ReduceLROnPlateau(monitor='val_loss',
        factor=np.sqrt(0.1),
        cooldown=0,
        patience=5,
        min_lr=0.5e-6)

    return [checkpoint,es,lr_reducer]
```

Fig. 15: Callback function.

```
def MASE(training_series,prediction_series):

    d=K.mean(K.abs((training_series[-1]-training_series[1:])))

    errors = K.mean(K.abs(training_series - prediction_series),axis=-1)
    return errors / d
```

Fig. 16: MASE function.

APPENDIX D
RESULTS

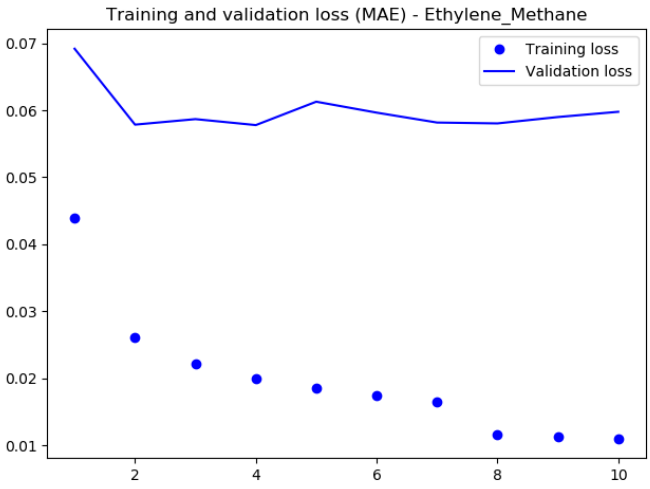

Fig. 17: 1D-CNN model

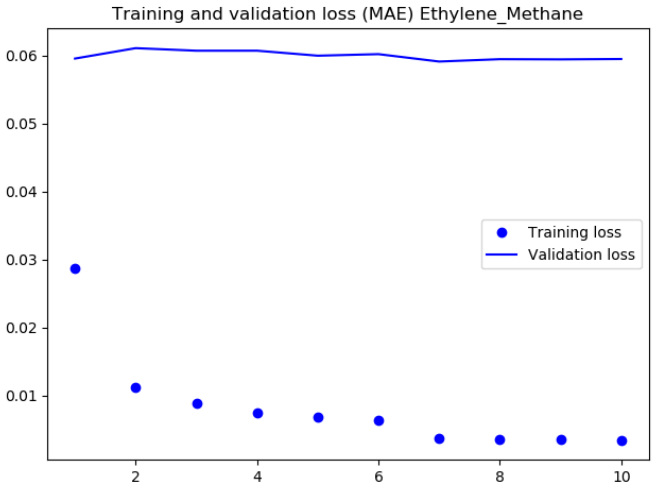

Fig. 18: LSTM Model.

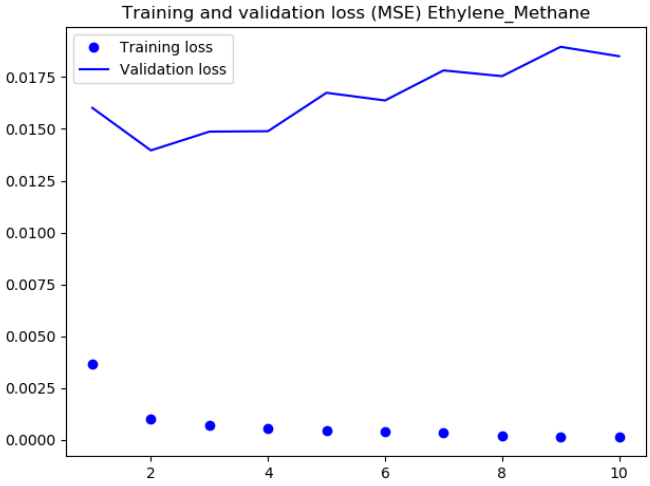

Fig. 19: 1D-CNN model

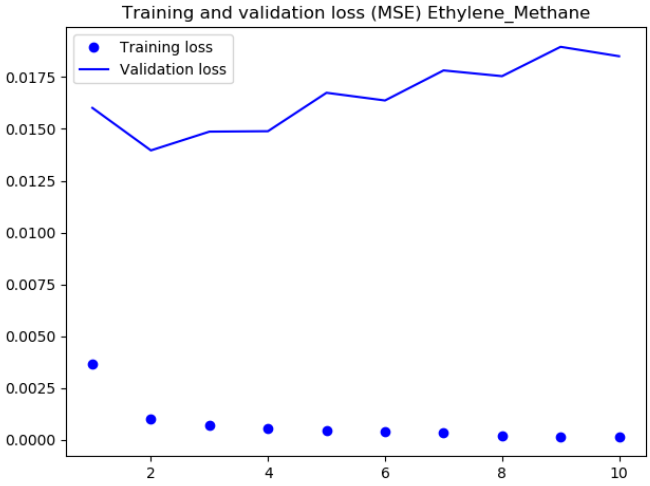

Fig. 20: LSTM Model.

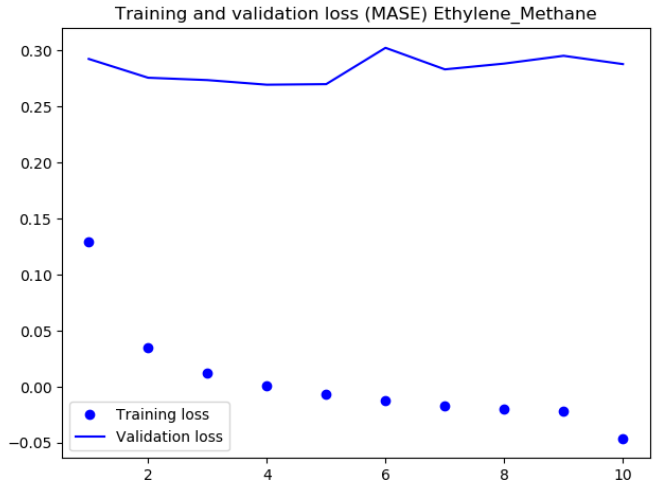

Fig. 21: 1D-CNN model

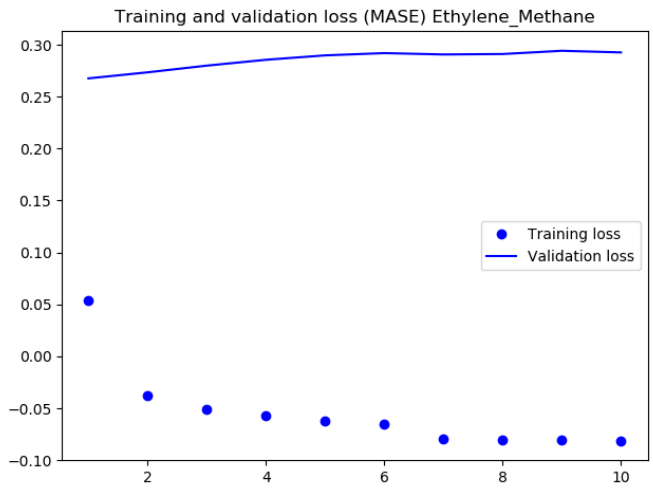

Fig. 22: LSTM Model.

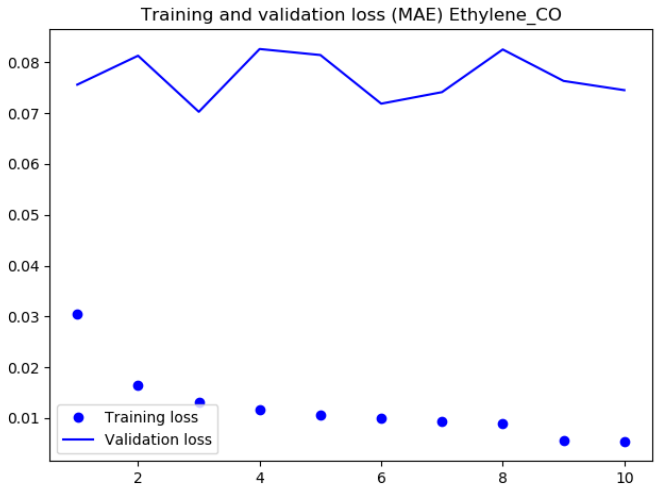

Fig. 23: 1D-CNN model

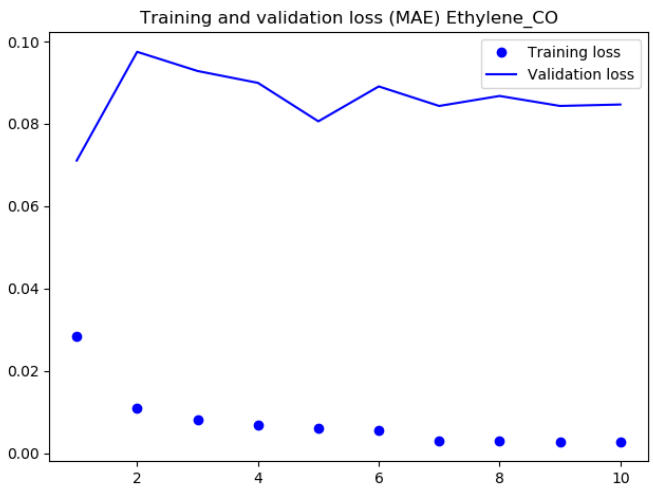

Fig. 24: LSTM Model.

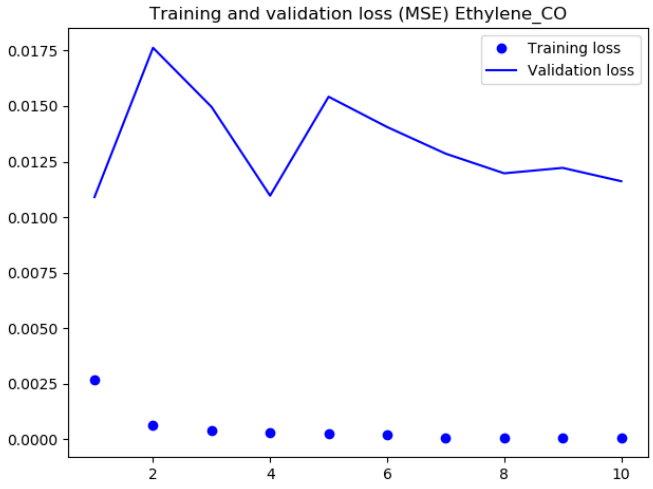

Fig. 25: 1D-CNN model

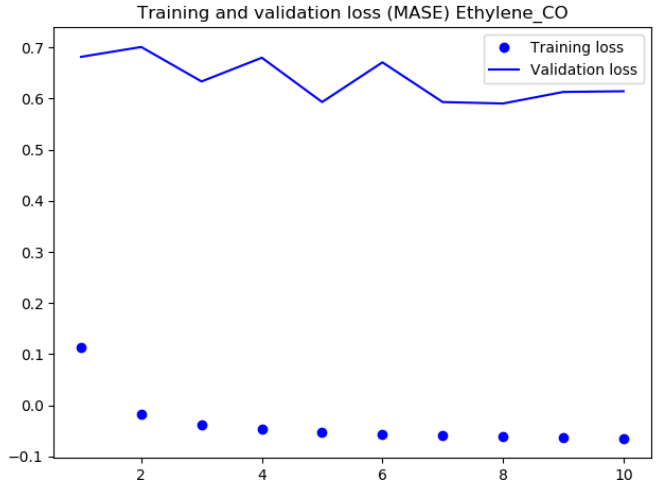

Fig. 26: LSTM Model.

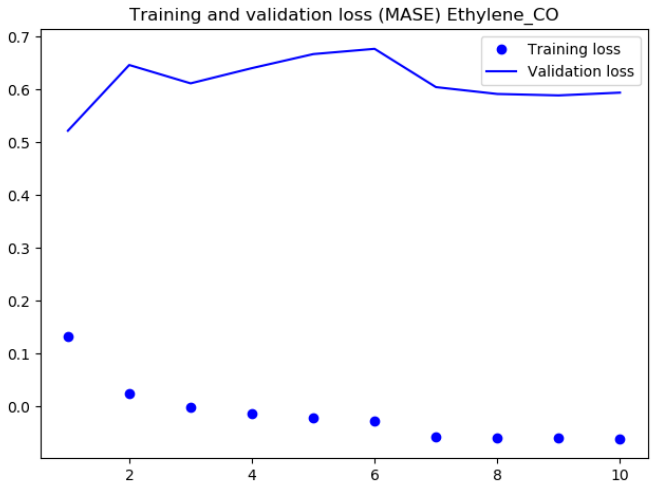

Fig. 27: 1D-CNN model

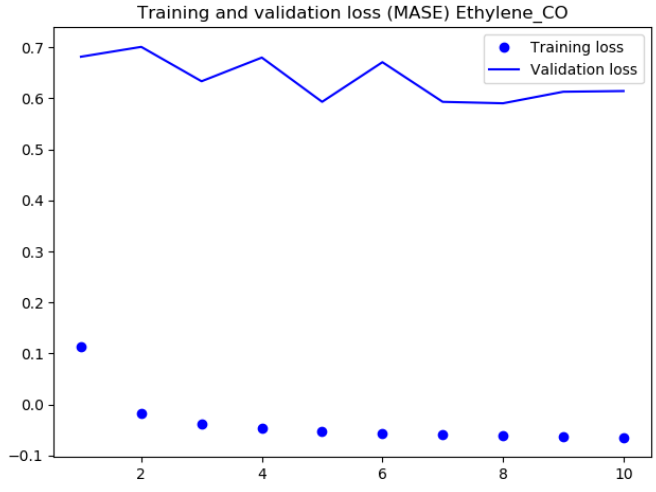

Fig. 28: LSTM Model.

Etheylene Methane Dataset									
	MAE			MSE			MASE		
	training	validation	testing	training	validation	testing	training	validation	testing
1D-CNN	0.0273	0.0723	0.1363	0.0018	0.0177	0.0438	0.1373	0.4258	0.7120
LSTM	0.0155	0.0679	0.1306	0.0006	0.0144	0.0368	0.0779	0.3995	0.6822
Ethylene CO Dataset									
1D-CNN	0.0176	0.0992	0.1006	0.0009	0.0182	0.0230	0.1147	0.8314	0.7112
LSTM	0.0135	0.0915	0.0991	0.0006	0.0172	0.0233	0.0882	0.7664	0.7008

Fig. 29: Result on one compiled model.