Title
Network Sensor Error Quantification and Flow Reconstruction Using Deep Learning

Author
Maheshwari, Saurabh

Publication Date
2020

Network Sensor Error Quantification and Flow Reconstruction Using Deep Learning

By

SAURABH MAHESHWARI

THESIS

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Civil and Environmental Engineering

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

_____

Yueyue Fan, Chair

_____

Michael Zhang

_____

James Sharpnack

Committee in Charge

2020

Abstract

In this study, we approach the problem of quantifying the network sensor errors as a supervised learning problem and leverage deep neural networks to map observed traffic flow counts to the systematic errors in the sensors. We aim at building a model that could reconstruct the erroneous flow irrespective of the level of random noise in the sensors, which is unknown in the real-world. By reconstructing the erroneous flow with high accuracy, the transportation planners could gauge the true traffic flow demand in the network and can make informed infrastructure related decisions.

We begin by simulating the traffic network under dynamic flow assignment settings to generate the base flow that we treat as the ground truth. We then introduce measurement errors to the base flow to generate the observed flow which is transformed into a multi-dimensional time-series tensor data, where each time step has dimension equal to the number of sensors in the network. Next, we introduce deep neural network comprising of 1-Dimensional Convolutional Neural Networks (1-D CNNs) to extract high-level spatial-temporal features from the observed flow time series data. To understand the generalization capability of the deep learning model, we deploy it against numerous test cases with varied levels of random errors and proportion of malfunctioning sensors in the network. Results indicate that the flow reconstructed using the deep learning model is very close to the ground truth flow and that the model predicts the systematic errors in the test cases with high accuracy.

The major advantages of this study are that, firstly, our model is robust to the flow imbalance in the network unlike most of the network sensor health studies in the past. Secondly, our approach escapes dealing with complicated flow-density relations one might encounter while modeling dynamic flow using traditional analytical statistical approaches.

Acknowledgements

First and foremost, I would sincerely like to thank my advisor Prof. Yueyue Fan for her constant support during my graduate studies at UC Davis. Her guidance and ever motivating attitude helped me cruise through my studies at UC Davis very smoothly. I can not thank her enough for giving me this opportunity and providing me with a pressure-free environment that helped me grow holistically. Secondly, I would like to thank Prof. Michael Zhang and Prof. James Sharpnack for agreeing to serve on my thesis committee. I would also like to thank my lab mates - Yudi Yang and Han Yang for helping me settle into my research and being on my side whenever I was facing academic difficulties. Lastly, I would like to thank my parents because of whom I am here. None of this would have been possible without their support and understanding.

Table of Contents

List of Tables

List of Figures

1. Introduction

In 21st century, where data is revolutionizing every other sector, field of transportation engineering is no way behind. Traffic sensors have been widely installed on urban roads to capture real-time traffic data which is extensively used in academic research and applications. For example, In California, Caltrans (California Department of Transportation) PeMS stores traffic data captured by loop detectors which consists of 30 seconds and 5 minutes vehicle count or volume, vehicle occupancy and average speeds. This data can be used in monitoring traffic congestion, effective traffic management such as signal timing optimization, pollutant emission control, and support decision making for transportation agencies.

Although, there are numerous applications of traffic data, in reality, the sensors are prone to errors which causes reliability issues. These errors could be attributed to a number of factors such as double counting of lane-changing vehicles, weather conditions, pavement failures or just random fluctuations while transmitting data (Coifman, 2006). It is reported that out of total freeway sensors in PeMS, only two-thirds of the loop detectors installed on California freeways are properly working (Rajagopal and Varaiya, 2007). This quantifies the problem at hand and the need to put a system in place that not only detects malfunctioning sensors (partially or fully), but also corrects the erroneous data.

In order to quantify the sensor health, it is important to classify measurement error in a sensor into systematic and random errors. The magnitude of data corruption in a sensor could be attributed to the degree of systematic error it possesses inherently. Random errors exists in any system and pertains to the random fluctuations in the data such as white noise, hence should not be considered when quantifying the health of the sensor. Once the magnitude of systematic errors are estimated, one can prioritize efforts to scrutinize the most unfit sensors first. In addition to this,

systematic errors could help recover the erroneous flow from the partially-malfunctioning sensors that will practically help researchers and practitioners to use the data, which most of the current studies discard.

In order to estimate the systematic errors, it is not sufficient to study an individual sensor at a time, rather one must study the traffic flow at the network and temporal level to incorporate for spatial and temporal correlations among sensors. As the network becomes larger, the number of sensors to be considered grows significantly. Given the amount of data that is readily available to us, we need algorithms that can exploit the spatial and temporal characteristics of the network and learn trends in the historical data. Traditional formal statistical methods could be complex, time consuming and could suffer with complicated assumptions. To overcome this and automate the process of knowledge acquisition, there is a need to consider machine learning algorithms. Deep Learning, a subset of machine learning has provided state-of-the-art results in various domains, such as image recognition, natural language processing, and time-series forecasting where data is usually multi-dimensional. Since, the traffic data can be treated as multi-dimensional time-series data, deep learning can prove to be very effective in estimating sensor errors.

Researchers have used convolutional neural networks (CNNs), which is a special kind of deep learning architecture to extract important local non-linear feature information from multi-dimensional time-series data. Depending on the task at hand such as forecasting, regression or classification, these features are then fed into other deep learning architectures like fully connected neural networks for further processing. Given the capability of CNNs, the idea is to extract high-level features from traffic data and learn a non-parametric mapping to predict systematic errors in network sensors. In this work, we focus on:

1) Developing a methodology to estimate network sensor errors using deep learning in dynamic traffic assignment settings and reconstruct erroneous traffic flow
2) Understanding how well a deep learning model can learn the spatial and temporal correlations in the traffic data without explicitly using any information about the network structure

The rest of the thesis is organized as follows. The second section dives deep into literature review where we discuss previous work in the domain of sensor health problem and their limitations. In additions to that, we also briefly look at the applications of deep learning in the field of transportation engineering to gauge its potential in this domain. In third section, we discuss the sensor error model formulation, our approach to represent traffic data as multi-dimensional time-series tensor and the theory of deep learning. Fourth section discusses the methodology in a detailed step-by-step fashion. In the fifth section, we highlight the results of our work and analyze the generalization capability of our model. In the sixth section, we finally discuss the conclusions of this study and the potential future work.

2. Literature Review

2.1 Traffic sensor quality

Research endeavors related to assessing traffic sensor quality can be broadly classified into two categories:

1) Identifying the malfunctioning sensors
2) Reconstructing the missing or erroneous traffic data

Identifying the malfunctioning sensors in a network is usually referred to as sensor health problem. Most of the work under this category is focused on filtering completely malfunctioning data by designing various validation criteria, for example threshold on the maxima of occupancy and volume, the numbers of samples with non-zero volume but zero speed, and the average effective vehicle lengths (Turochy and Smith, 2000). Other studies such as Hu et al. (2001); Chen et al. (2003), and Turner et al. (2004) are based upon the same idea with more complex validation criteria. These studies are local in nature, i.e., they consider one sensor at a time. These methods are easy to use in practice but suffer with two major limitations. Firstly, the threshold could depend on the location and factors like weather or traffic incidents, hence determining the range could be challenging. Secondly, these methods do not exploit spatial correlations on the network level. Other works in this area utilize the flow from adjacent sensors for identifying bad sensors in the network. Vanajakshi and Rilett (2004) adopted a generalized reduced gradient method following the flow conservation principle. Kwon et al. (2004) proposed semiautomatic and automatic methods for detecting sensor errors on the basis of strong correlations between measurements made by spatially close sensors.

Later, Sun et al. (2016), proposed a method to study sensor health problem at the network level where the basic idea was to check for consistency among traffic flow sensors based on flow

conservation. They define the health index for a sensor on a scale of 0 to 1, where sensors with low health index are more likely to be erroneous. The major drawback of this study including the previously mentioned studies is that they do not quantify the magnitude of data corruption, for example, the degree by which a sensor over-estimates or under-estimates the traffic counts.

There has been a lot of focus on imputing the missing or corrupted data by utilizing potential temporal and spatial relationships in the data. The research in this area could be classified into 3 categories: prediction, interpolation and statistical learning (Li et al., 2014). Traditional prediction methods such as time-series forecasting using ARIMA have been used to map historical data to missing future data (Nihan, 1997). The drawback of prediction methods would be inability to use data succeeding to missing data occurrences, thus leading to discarding the useful data. For interpolation, Allison (2001) used a history model that used historical data from a site to estimate missing or corrupted data at the same site at the same time interval. Other approaches for interpolation have used methods such as weighted average of neighboring sensors (Chen et al., 2003) and K-Nearest Neighbors (Liu et al., 2008). Major drawback with interpolation is that forcing the data to be close to neighboring sensor data could lead to unaccountability in local traffic variations. Lastly, researchers have proposed statistical learning methods such as Probabilistic Principal Component Analysis (Qu et al., 2009), Markov Chain Monte Carlo multiple imputation method (Farhan and Fwa, 2013), Neural Networks (Lv et al., 2015) and Deep Learning (Duan et al., 2016). Basically, statistical learning methods use the observed data to learn the trend and then infer the missing or corrupted traffic data.

For the above mentioned sensor health problem and imputation studies, there are three major challenges:

1) All the aforementioned studies based on diagnosing sensor health do not quantify the magnitude of sensor data corruption nor do they differentiate systematic errors from the random errors. Sensors could be systematically corrupted due to any reason (double counting long vehicles, counting adjacent lane vehicles and so on), but the data they provide still carries useful information
2) In most of the cases, imputation studies use the uncorrupted sensors to reconstruct the missing or corrupt data, hence discarding useful information from partially malfunctioning sensors

Yang et al. (2019) attempted to address these problems by proposing a novel method based on Generalized Method of Moments (GMM) to quantify the magnitude of systematic error in data and identify partially malfunctioning sensors. They differentiate random errors from systematic errors, and only use systematic errors to mark the health of a sensor. This is of great practical value because this method enables utilization of information from partially malfunctioning sensors for reconstructing corrupt data from other sensors in the network. However, this study suffers from following drawbacks:

1) They have applied GMM method in static flow setting, whereas, in real-world, flow is usually dynamic. Solving additional microscopic flow-density constraints, for instance, in the form of partial differential equations, could be tedious and sometimes even impossible in case of analytical models
2) GMM method suffers if there is congestion in the network, i.e., if the network flow is imbalanced. To deal with this, the authors aggregate the flow over a suitable time interval, such as 24 hours. This results in smoothing the flow, hence not exploiting the inherent variations in the temporal dimension

3) GMM method behaves well only if there are a few calibrated sensors in the network

In this study, we adopt a similar framework as Yang et al. (2019). Though, rather than using traditional statistical methods, we adopt a deep learning approach to deal with the above limitations. In the following section, we dive into the various applications of deep learning in transportation engineering so as to explain why deep learning approach for learning the static and temporal correlations in the traffic data is really promising.

2.2 Deep Learning in transportation engineering

Similar to many domains, transportation has also entered the world of big data due to significant amount of information collected by CCTV cameras, GPS, loop inductors, probe, and so on. Nguyen et al. (2018) in their review highlighted the application of various deep learning algorithms in processing traffic data for popular topics such as transportation network representation, traffic flow forecasting, traffic signal control, automatic vehicle detection, traffic incident processing, travel demand prediction, autonomous driving and driver behavior.

Deep learning networks can extract important features from multi-dimensional data and hence are capable to modeling spatial and temporal relationships in the traffic networks. Ma et al. (2015) used a combination of recurrent neural networks (RNN) and deep restricted boltzmann machines (RBM) to predict congestion evolution in a high-dimensional transportation network based on GPS data from taxi. Further, Fouladgar et al. (2015) used convolutional neural networks (CNN) and long short term memory (LSTM) to predict congestion. They claimed that short term future traffic conditions on a road segment can be predicted reliably on the basis of traffic patterns of the neighboring road segments, without even using historical data for the particular road segment.

Traffic flow prediction is a very basic problem for transportation planners and several attempts have been made to leverage statistical methods for predicting number of vehicles on a road segment for future time intervals. Jia et al. (2017) introduced deep belief network (DBN) and LSTM for urban traffic flow prediction considering the impact of rainfall. They concluded that with the consideration of additional rainfall factor, the deep learning predictors outperformed the existing predictors and were an improvement over the original deep learning models without rainfall input. Zhao et al. (2017) proposed a LSTM network for traffic flow prediction and claimed that the LSTM model outperformed not only traditional machine learning models such as ARIMA and support vector machines (SVM) but also RNNs, especially in long term prediction.

Deep learning has also been combined with reinforcement learning to create deep reinforcement learning (DRL) method which has been used in traffic signal control to help planners prevent congestion. Genders and Razavi (2016) proposed DRL based on CNNs to reduce cumulative delay by 82%, queue length by 66% and travel time by 20%. Van Der Pol, and Li et al. (2016) also applied DRL method and concluded that the results obtained using deep learning were remarkably better than traditional methods for deciding appropriate signal timings.

CNNs have produced state-of-the-art results for travel mode detection based on their adaptability to image recognition or signal processing tasks. Deep learning is making technologies like unmanned aerial vehicle detection or camera-based vehicle recognition viable. Liang and Wang (2017) used 1-dimensional CNNs to classify 7 transportation modes using accelerometer readings from smartphone platform with an accuracy of 94.48%, which they claim to be highest among existing studies. Ma et al. (2017) used CNNs to convert traffic speeds data to images to predict large-scale, network-wide traffic speeds. They claimed that CNNs outperformed traditional statistical algorithms namely, ordinary least squares, k-nearest neighbors, artificial neural network,

and random forest by an average accuracy improvement of 27.96% within acceptable execution time. Adu-Gyamfi et al. (2017) also proposed CNN to detect and classify vehicles into seven classes, and achieved average recall rates between 89% and 99% for seven classes of vehicles.

Travel demand forecasting is another fundamental problem in transportation engineering that basically deals with predicting the number of users of transportation infrastructure in future. As mentioned above, data from GPS, CCTV cameras, probe, sensors is enormous and complicated. Exploiting these data sources using deep learning has made short term demand prediction accurate as it has claimed to learn the spatial and temporal correlations in the travel demand. Ke et al. (2017) proposed a fusion convolutional long short term memory network to predict short term travel demand to simultaneously consider the spatial, temporal and exogenous dependencies for an on-demand transport service. Yao et al. (2018) used deep multi-view spatial temporal network to predict taxi demand. Xu et al. (2017) also predicted traffic demand using RNN using recent demand and other variables such as weather, drop-offs, etc. Liu and Chen (2017), and Baek and Sohn (2016) predicted public transit demands using deep neural networks.

Predicting driver behavior is a very essential aspect in autonomous driving and accident prevention. Dwivedi and Biswaranjan (2014) used CNNs to detect drowsy driver behavior. CNNs have set benchmark accuracies on computer vision tasks, and in this study CNNs were utilized to extract complex facial features and non-linear feature interactions, resulting in an accuracy of 92% in detecting drowsy behavior. Similarly, due to computer vision applications, CNNs have also been used extensively in autonomous vehicle research (Grigorescu et al., 2019).

Given all this research, it is clear that deep neural networks have been successful in learning spatial and temporal correlations in traffic data. Also, given the complexity and volume of the data

readily available today, deep neural networks are very promising in producing state-of-the-art results for various transportation tasks.

3. Theory

3.1. Framework for modeling errors

Here we employ a similar framework for modeling errors as Yang et al. (2019), the nuances of which are described in this section. Suppose that a road network has sensors installed that continuously report flow variables such as vehicle count and occupancy at constant time intervals. Let the count recorded when the s^{th} vehicle passes over sensor a be $1 + \varepsilon_a^s$, in which case, the total flow count over sensor a for a particular time period is given as

$$V_a = Z_a + \sum_{s=1}^{Z_a} (\varepsilon_a^s)$$

where, Z_a is the actual flow count and V_a is the observed vehicle count. Note that ε_a^s is discrete in nature as a vehicle passing over the sensor should be reported as a discrete number. The total error in the measurement time interval for sensor a can then be broken into systematic and random error components.

$$\sum_{s=1}^{Z_a} (\varepsilon_a^s) = V_a - Z_a = (E(V_a|Z_a) - Z_a) + (V_a - E(V_a|Z_a))$$

where, the prior component on right hand side is the systematic error, and later one is the random error. The assumption here is that the error generating mechanism is invariant across different time intervals and the parameters that control the measurement error are constant over a suitable time interval, for example, 24 hours. Thus, let μ and σ be the time independent vectors related to systematic and random error, respectively.

Thus, one could write the first two moments of the observed traffic counts conditioned on Z as a deterministic function of Z, such as,

$$E(V_a|Z_a) = f(Z_a; \mu_a) \ \ and \ Var(V_a|Z_a) = \Phi(Z_a; \sigma_a^2)$$

Exact forms of f and Φ would depend upon the nature of distribution of measurement errors, but we assume additional assumptions to narrow down our focus to a simplified model, where,

$$E(V_a|Z_a) = Z_a + \mu_a Z_a \ \ and \ Var(V_a|Z_a) = \sigma_a^2 Z_a$$

Here, $\mu_a Z_a$ is the systematic error of the traffic counts, μ_a is the systematic error ratio, and σ_a is the random error ratio. Further, we can define

$$\beta_a = 1/(1 + \mu_a)$$

$$E(V_a|Z_a) = Z_a/\beta_a$$

Now, one can see that μ_a is the magnitude with which the sensor could over-estimate or under-estimate the traffic flow counts. When the flow aggregation window is small, say 10 minutes, the systematic error could end up being a non-integer value. Since, the observed traffic counts could only be integers, we modify the observed traffic count function as

$$E(V_a|Z_a) = [Z_a + \mu_a Z_a] = [Z_a/\rho_a]$$

where, $[.]$ is the nearest integer function that rounds the non-integer values to the nearest integer.

In this model framework, we consider that a road segment has one sensor to capture traffic in one direction. Hence, the transportation network should be abstracted to meet the above mentioned topology by using the algorithm provided in section 4.1.

3.2. Multi-dimensional time-series tensor representation

Consider a traffic network abstracted in the form of a directed graph, $G = \{N, A\}$, where N and A are the set of nodes and links in the graph, respectively. Let $n = |I|$ and $s = |A|$, where I is the set of intermediate nodes and |.| represents cardinality of the set. Let $A^{+}_{(i)}$ and $A^{-}_{(i)}$ be the set of links that enter or exit an intermediate node i. Let P be the node-link adjacency matrix of size $n\ by\ s$, where $P_{ia} = 1$ if $a \in A^{+}_{(i)}$, -1 if $a \in A^{-}_{(i)}$, 0 otherwise. Now, since every link contains a sensor recording the flow variables, s is also the number of total sensor in the graph G.

As mentioned above, it is safe to assume that measurement error parameters are constant over a suitable time range t. Typically, sensors report the traffic flow variables such as counts, occupancy and average speed at constant time intervals c. For example, in case of PeMS, c is 30 seconds and 5 minutes. Thus, total number of time steps for which μ_a and σ_a can be assumed constant is $m = t/c$. For example, if we consider t = 24 hours and c = 10 mins, then the length of vector of observations for a particular sensor where μ_a and σ_a can be assumed constant would be $m = t/c$ = 144. Since we have n sensors in the network G, consider a matrix of dimension $s\ by\ m$. ***We define this matrix as a time-series of length m with s features or an s-dimensional time series with m steps***. Now, depending on our estimation horizon, we could have varying number of time-series samples, denoted by q. Thus, we consider the flow observations as a 3-D tensor of dimension $q\ by\ s\ by\ m$. In figure 1, f_{123} represents the flow variable for 2nd sensor at 3rd time step of the 1st time-series sample.

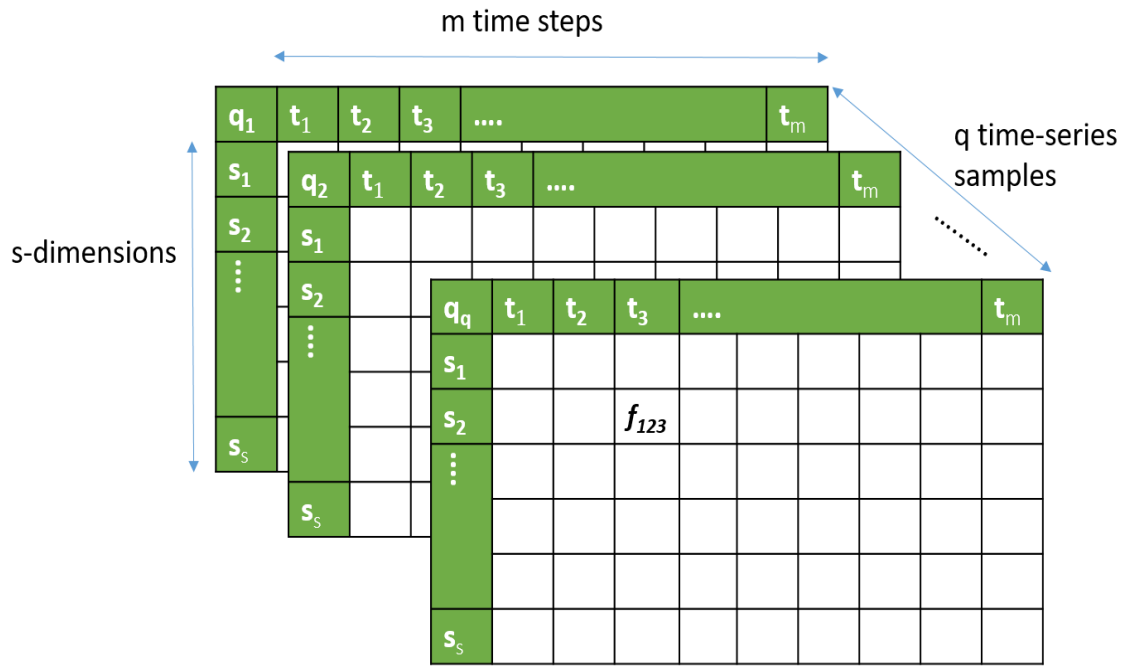


Figure 1. Time-series representation of observed data

For a sample q_i, we define a β_i, a vector of length s representing sensor systematic ratios. In practice, β_i is a vector containing $1/(1+\mu_a)$ values corresponding to the network sensors.

Thus, we define the problem as mapping the flow observations to the corresponding vector β.

$$q_i \xrightarrow{D} \beta_i, \forall\, i$$

Here, D is a mapping that takes time-series samples as input and returns vectors of the corresponding β values. Now, goal is to learn this mapping D from the observed flow data. In this study, we will be using a deep neural network as this mapping D.

3.3. 1-D Convolutional Neural Networks

Convolutional Neural Networks (CNN) are special kind of deep learning models that are usually used for computer vision tasks. Some of the advantages of using CNNs are as follows:

1) They are capable of extracting features straight from the raw data, whereas in traditional statistical models, features are usually engineered manually
2) CNNs enable weight sharing over different parts of input that reduces the number of parameters effectively, making it computationally feasible to train on large set of data as compared to traditional deep fully connected networks
3) CNNs are immune to small transformations in the data such as scaling, translating, skewing and distorting

Usually for 3-D data such as images, conventional 2-D CNNs are used. For this study, since we treat our data as a multi-dimensional time-series data, we essentially need convolutions in temporal dimension only. Hence, we adopt 1-D CNNs for our research that are a modified version of 2-D CNNs. Recent studies have shown that 1-D CNNs with shallow depths have demonstrated a superior performance on applications which have a limited data and high signal variations (i.e., patient ECG, civil, mechanical or aerospace structures, high-power circuitry, power engines or motors, etc.) (Kiranyaz et al., 2019). Another advantage of using 1-D CNNs over 2-D CNNs is that 1-D CNNs require simple array operations rather than matrix operations (in case of 2-D CNNs), due to which 1-D CNNs are computationally less expensive and do not require special hardware setup like GPU farms.

As mentioned above, we are going to use 1-D CNNs to extract features from the traffic data. These features are then passed to another algorithm or deep learning architecture for further processing. Figure 2 illustrates the flow we use in this study.

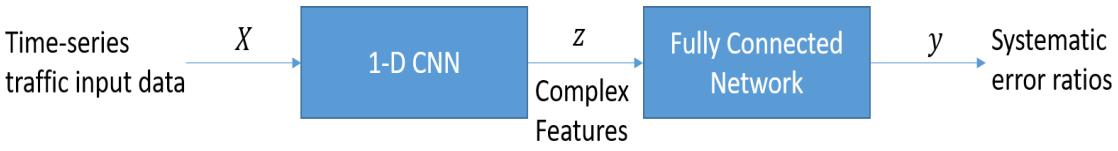


Figure 2. Model Flow

Consider a set of time-series samples $X = [x_1, x_2, \dots, x_q]'$, where x_i is a time-series with m steps in s-dimensional space. Let each time-series be associated with $Y = [y_1, y_2, \dots, y_q]'$, where y_i is an s-dimensional vector containing the sensor systematic error ratios. A 1-D CNN constitutes of L convolutional layers, where each layer l is composed of c^l sub-layers or channels. Each layer l performs convolutions using appropriate filters, the cumulated output of which is then passed through a batch-normalization layer, followed by an activation function and finally subsampling (pooling). The output of this layer is then provided to the next layer as input. In this section, we will be discussing all of the above mentioned layers in detail.

Each layer l has a set of filters, where each filter has the same number of channels as the input to layer l, i.e. c^{l-1}. After convolution with one filter, one channel of output is formed. Thus, c^l filters produce an output with c^l channels. This process of convolutions and accumulating the output of each filter to produce cumulative output is called forward-propagation.

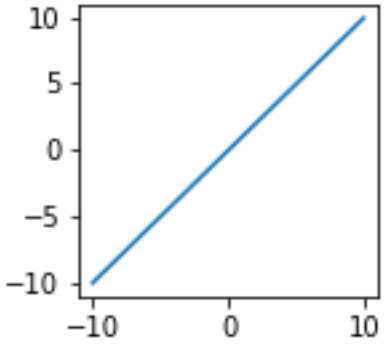
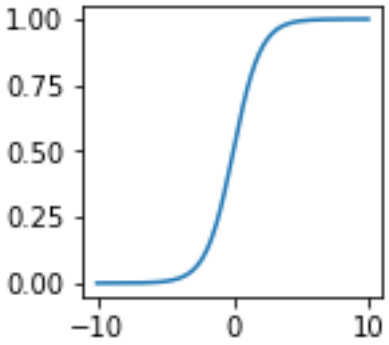
3.3.1 Forward Propagation in CNNs

During the forward propagation, the channels of each layer l are the result of accumulation of the final output (after back-propagation and pooling) of the previous layer $l-1$.

$$o_i^l = bias_i^l + \sum_{j=1}^{c^{l-1}} conv1D\left(w_{ij}^l,\ s_j^{l-1}\right), \qquad i = (1,2,\dots,c^l)$$

$$s_i^l = f\_act\left(o_i^l\right)$$

where, o_i^l is the channel i of layer l after convolution, $bias_i^l$ is the bias that is added to the convolved output for channel i of layer l, w_{ij}^l is the weight of the filter that maps the channel j of the layer $l-1$ to channel i of the layer l, s_j^{l-1} are the activations of channel j of layer $l-1$, and $f\_act(.)$ is the non-linear activation function. Following are the common choices for activation function $f\_act(.)$:

	Activation function		$f\_act =$	$f'_{act} =$
i)	Linear		x	1
ii)	Sigmoid		$\frac{1}{1+e^{-x}}$	$f\_act(x)(1 - f\_act(x))$

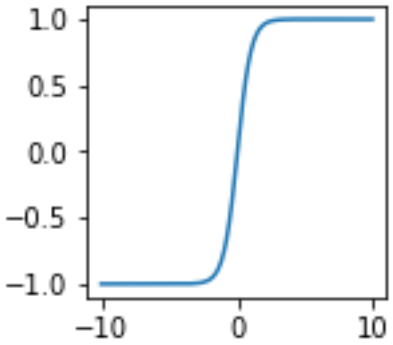
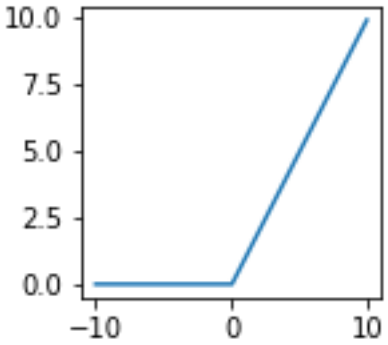
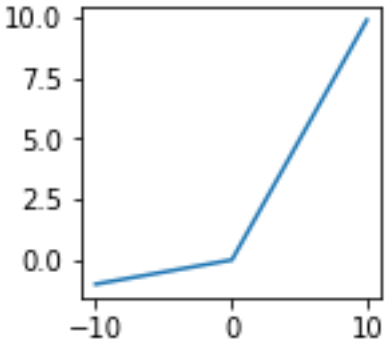
iii)	tanh		$\frac{2}{1+e^{-2x}} - 1$	$\frac{1}{1+x^2}$
iv)	ReLU		$\begin{cases} 0, & x < 0 \\ x, & x \geq 0 \end{cases}$	$\begin{cases} 0, & x < 0 \\ 1, & x \geq 0 \end{cases}$
v)	Leaky ReLU		$\begin{cases} ax, & x < 0 \\ x, & x \geq 0 \\ & a > 0 \end{cases}$	$\begin{cases} a, & x < 0 \\ 1, & x \geq 0 \\ & a > 0 \end{cases}$

Table 1. Commonly used activation functions for deep learning

Usually, the dimension of a channel in layer l is lesser than the dimension of a channel in the preceding layer $l-1$ because of convolutions. In order to keep the dimensions of the channels in the current layer same as the preceding layer, we can pad the channels with zero, also known as *zero-padding*.

3.3.2 Batch-normalization

After performing the convolution operation and before applying the activation function, the output o_i of channel i is normalized with the mean μ_i and standard deviation σ_i that are computed for the channel i across the batch of input samples (b) and dimension of the channel(d). After normalization, a channel-wise affine transformation parameterized through γ_c and β_c is performed, which are learnable parameters of the model.

$$o_i^{norm} = \gamma_c * \frac{o_i - \mu_i}{\sqrt{\sigma_i^2 + \varepsilon}} + \beta_c,$$

$$\mu_i = \frac{1}{|B|} * \sum_{b,d} o_i \text{ and } \sigma_i^2 = \frac{1}{|B|} \sum_{b,d} (o_i - \mu_i)^2$$

where, B contains all the activations in channel i across all the samples b. It is observed that batch-normalization accelerates training, enables higher learning rates, and improves generalization accuracy.

3.3.3 Subsampling (Pooling)

Pooling layer is usually inserted after the convolutions are performed with an objective to reduce the computation in the network. Its function is to progressively reduce the temporal dimension of the channels to reduce the number of parameters and avoid overfitting. The pooling layer acts independently on each channel of the input layer and resizes it using a specific operation. The most common types of operations for pooling are *Max Pooling* and Average *Pooling*. Let us assume that the dimension of the pooling filters is $d_{pool}\ by\ 1$. The pooling layer moves the filter over the input and does the appropriate operation on d_{pool} numbers at a time. Max operation retains the maximum number out of the d_{pool} numbers, whereas Average operation returns the average of the d_{pool} numbers. The number of channels before and after the operation remain the same. The

pooling filter can also be shifted over the input with a stride of length s_{pool}. For example, if s_{pool} is 2, then the filter skips a number while shifting to a new set of numbers. An illustration of max pooling is provided in the figure 3 where the input has 2 channels with 10 dimensions each. Figure 4 depicts l^{th} convolutional layer in a network of 1-D CNNs.

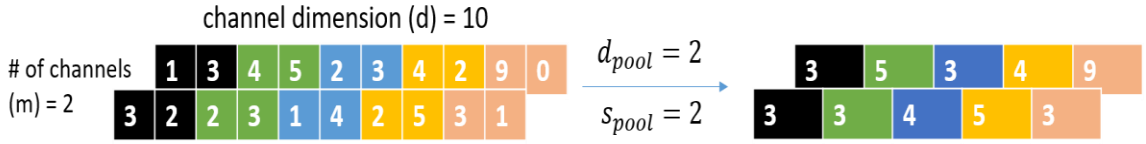


Figure 3. Max Pooling

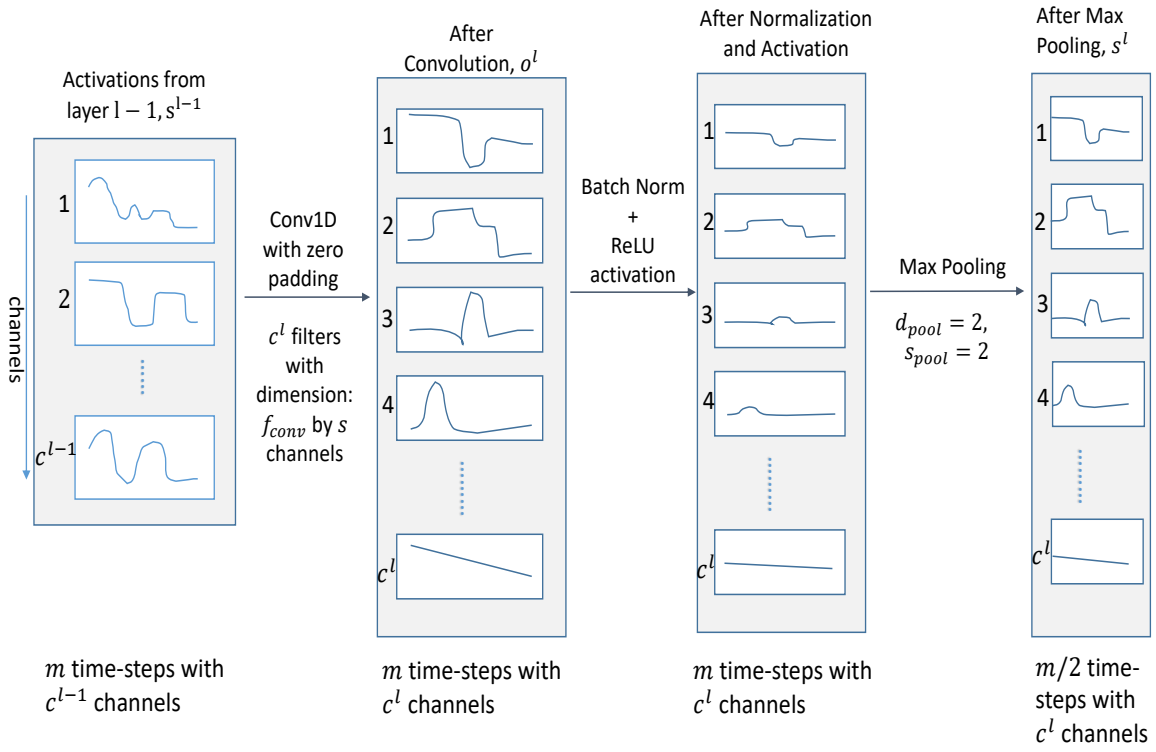


Figure 4. l^{th} Layer of a 1-D CNN

3.3.4 Fully Connected Network

We pass the output of the 1-D CNN to the fully connected network that predicts the systematic error ratios. A fully connected network is a regular artificial neural network where a neuron is connected to all activations in the previous layer. Let z be the features outputted by the 1-D CNN. Since z is the output of the last layer L, it consists of c^L channels, where each channel is of d^L dimension. We concatenate all the channels together to create a long output of dimension $d^L * m^L \; by \; 1$. Then, these activations are passed on to a fully connected neural network. The output of the fully connected layer is given by

$$y = s^{L+1} = f_{act}\,(bias^{L+1} \; + \; w^{L+1} \times z)$$

where, $\times$ is the matrix multiplication, w^{L+1} is the weight matrix of dimension $s \; by \; d^L * m^L$, and y is the s dimensional vector representing systematic error ratios. In case of regression function, f_act for the ultimate layer is usually the linear activation function.

3.3.5 Loss function for model optimization

Let $\hat{y}$ be the output of the last layer of deep learning model and y be the target systematic error ratio vector. We define a loss function $L(y, \hat{y})$ that computes the error between predicted and target values. While training the neural network model, we aim at minimizing the loss function value by optimizing the model parameters such as weights and biases of the layers. Choice of the loss function depends on the nature of problem we are dealing with, for example, classification or regression. In this case, as we are mapping traffic flow observations to systematic error ratio, it is a regression problem by nature. Table 2 summarizes some of the common loss functions adapted to train a deep neural network for a regression problem. For example, Wu et al. (2018) used MSE as the loss function for traffic flow prediction task. Yu et al. (2017) used both MAPE and MSE as

the loss function for traffic forecasting task with different deep learning models. Li and Wang (2019) used both MSE and MAE as the loss functions for short term traffic flow prediction.

	Loss function	$L(y, \hat{y}) =$
i)	Mean Square Error (MSE)	$\frac{1}{ns}\sum_{i=1}^{n}\sum_{j=1}^{s}\left(y_{ij} - \hat{y}_{ij}\right)^2$
iii)	Mean Absolute Error (MAE)	$\frac{1}{ns}\sum_{i=1}^{n}\sum_{j=1}^{s}\lvert y_{ij} - \hat{y}_{ij}\rvert$
iv)	Mean Absolute Percentage Error (MAPE)	$\frac{1}{ns}\sum_{i=1}^{n}\sum_{j=1}^{s}\frac{\lvert y_{ij} - \hat{y}_{ij}\rvert}{y_{ij}} * 100\%$

Table 2. Loss functions for a regression problem

In this table, n is the total number of time-series samples and s is the total number of sensors in the network. Next, we update the model parameters by optimizing the loss function using gradient descent, the process of which is called back propagation.

3.3.6 Back propagation in CNNs

As any other non-constrained optimization, we need to compute $L(y, \hat{y})$ and its gradient $dL/d\hat{y}$ for training the 1-D CNN. Ultimately we would need to calculate dL/dw_{ij}^{l} and $dL/dbias_{i}^{l}$ to iteratively update weights and biases, and minimize the loss $L(y, \hat{y})$. Here a high level idea of back propagation is provided. Using the chain rule,

$$\frac{dL}{dw_{ij}^{l}} = \left(\frac{dL}{do_{i}^{l}}\right)\left(\frac{do_{i}^{l}}{dw_{ij}^{l}}\right)$$

From forward propagation, we can see that

$$\frac{do_{i}^{l}}{dw_{ij}^{l}} = s_{j}^{l-1}$$

Hence, the above equation becomes

$$\frac{dL}{dw_{ij}^{l}} = \left(\frac{dL}{do_i^l}\right) s_j^{l-1} = \left(\frac{dL}{do_i^l}\right) f(o_i^{l-1})$$

Due to forward propagation we already know the values of s_j^{l-1}. Hence, we now need to compute $\left(\frac{dL}{do_i^l}\right)$. Again, we can use the chain rule as

$$\frac{dL}{do_i^l} = \left(\frac{dL}{ds_i^l}\right)\left(\frac{ds_i^l}{do_i^l}\right) = \left(\frac{dL}{ds_i^l}\right)\left(\frac{df(o_i^l)}{do_i^l}\right) = \left(\frac{dL}{ds_i^l}\right) f'(o_i^l)$$

Now, depending on the activation function, the derivative f' can be evaluated at o_i^l, which we know from forward propagation. Calculation of $\left(\frac{dL}{ds_i^l}\right)$ is performed again by the chain rule as we propagate the errors to the previous layers. For example

$$\frac{dL}{ds_i^{l-1}} = \left(\frac{dL}{do_i^l}\right)\left(\frac{do_i^l}{ds_i^{l-1}}\right)$$

Thus, starting from the last layer, calculation of $\frac{dL}{ds_i^L}$ is used for calculating the derivative of L w.r.t previous layer activations. Now, completing the cycle, we can write

$$\frac{do_i^l}{ds_i^{l-1}} = w_{ij}^l$$

Hence, now we have everything to compute $\frac{dL}{dw_{ij}^l}$, thus w_{ij}^l can be updated using the gradient descent algorithm as follows

$$w_{ij}^{l*} = w_{ij}^l - \alpha \frac{dL}{dw_{ij}^l}$$

where, α is the learning rate and w_{ij}^{l*} is the updated weight. Same way, biases of the deep learning model are optimized.

3.3.7 Gradient descent and its variants

As explained in the previous section, back propagation iteratively updates the parameters of the deep learning model till the loss function is minimized, i.e.,

Repeat till convergence {

$$w_{ij}^{l*} = w_{ij}^{l} - \alpha \frac{dL}{dw_{ij}^{l}},$$

$$bias_{i}^{l*} = bias_{i}^{l} - \alpha \frac{dL}{dbias_{i}^{l}}$$

}

This update is simultaneously done for all the parameters. But there are a few challenges to the above mentioned gradient descent algorithm:

1) Local Minima or Saddle Points: Deep neural networks are complicated functions with numerous non-linear transformations. Thus, the loss function that depends on large number of parameters (usually > 1e6) is highly complex. An example of a typical loss landscape for a deep learning model is given in the figure 5. Thus, it is very easy to get stuck in a sub-optimal local minima or saddle point.

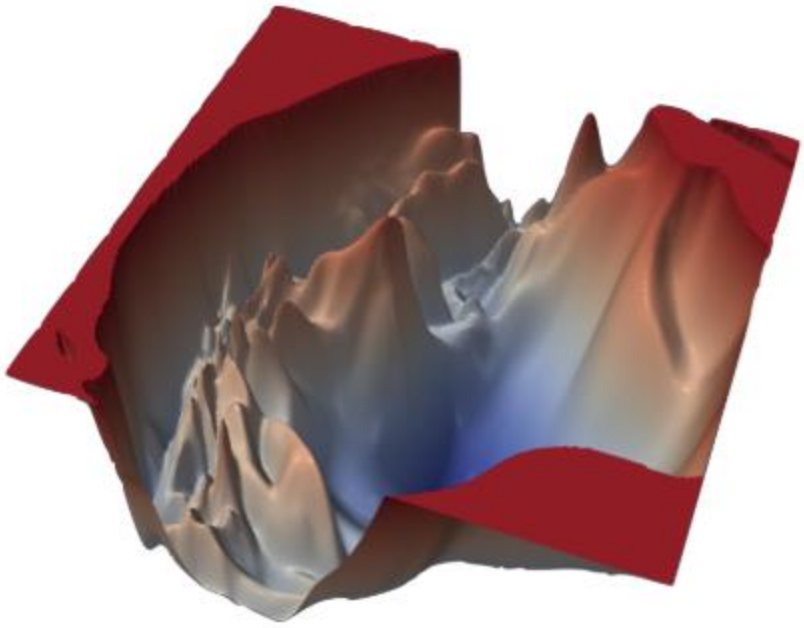


Figure 5. Loss landscape of ResNet-100 (Li et al. 2018)

2) Sub-optimal learning rate: In the gradient descent algorithm mentioned above, all the parameters are updated simultaneously with the same learning rate. This could be a problem if our data is sparse, as we might want to update certain parameters with larger learning rate than others. Also, choosing a smaller learning rate could lead to very slow convergence, whereas a larger learning rate could make loss function diverge.

In order to escape the sub-optimal local minima and to explore the parameter space exhaustively, the following updates to the conventional gradient descent algorithm are adopted:

1) Stochastic and Mini-Batch gradient descent: So far, we have been using the traditional batch gradient descent algorithm where we sum across the loss over the entire training samples.

In stochastic gradient descent (SGD), gradient updates are performed using one training sample at a time, where the sample is chosen at random without replacement. The update rule is modified as:

Repeat till convergence {

for i in 1:q {

$$w_{ij}^{l*} = w_{ij}^{l} - \alpha \frac{dL}{dw_{ij}^{l}},$$

$$bias_{i}^{l*} = bias_{i}^{l} - \alpha \frac{dL}{dbias_{i}^{l}}$$

}

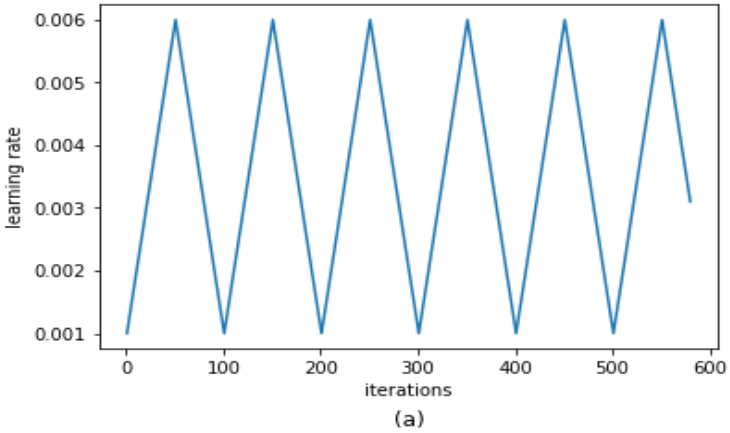
}

where, q are the total training samples. Due to frequent updates, there is variance in parameter updates which makes search more exhaustive. Though, due to variance in parameter updates, the convergence in the parameters could be adversely affected.

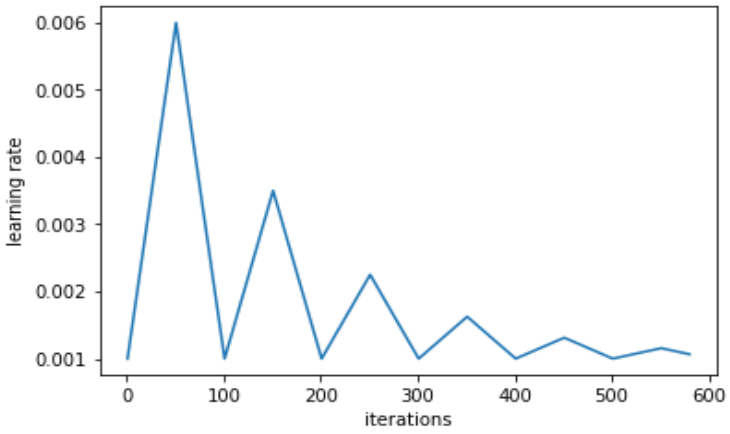
This drawback of SGD is rectified using mini-batch gradient descent, where a random sample or mini-batch of training examples is used to update the parameters at a time. This version of gradient descent is more suitable as it reduces the variance in parameter updates and is computationally less expensive as compared to SGD. The size of the mini-batch, known as batch size is usually chosen as an exponent of 2, for example, 32, 64, 128, etc.

2) Cyclical learning rate: Smith (2017) proposed scheduling the learning rate (α), such that it oscillates between a designated upper and lower bound in a cyclical manner. The objective of cyclical learning rates is to increase the α for certain batches and then decrease the learning rate while training the remaining batches in an epoch. An epoch is defined when all the training

samples are used for gradient update once. Increasing the α helps escape sub-optimal local minima and saddle points, whereas decreasing the α helps in convergence. Figure 6 shows a few commonly used cyclical learning rate patterns.



(a)



(b)

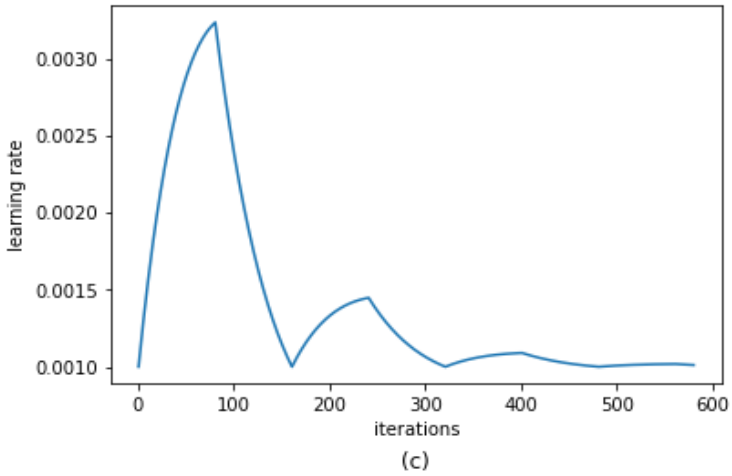


Figure 6. Various policies for cyclical learning rates

3.3.8 Optimizing gradient descent

SGD and mini-batch gradient descent introduced above, along with proper learning rates help escaping sub-optimal local minima and saddle points, but because of the highly complex loss landscape, parameter updates tend to be very noisy, which impedes convergence to optimal minima. Also, so far the learning rate for updating each parameter is same, which could be an issue, for example in the case of sparse data. Thus, in practice a further optimized version of gradient descent update rule is used, some of which are explained here. These update rules tend to accelerate parameter updates in a streamlined fashion.

1) Momentum: In this technique, we replace the current gradient with exponential average of gradients. Hence, in addition to accounting for the current gradient, Momentum accumulates the gradients from past steps to determine the next gradient direction. The algorithm is as follows:

   Repeat till convergence {

$$v_{ij}^{t} = \eta v_{ij}^{t-1} + (1-\eta)\frac{dL}{dw_{ij}}$$

$$w_{ij}^{t} = w_{ij}^{t-1} - \alpha v_{ij}^{t}$$

}

where, η is the coefficient of momentum, usually kept at 0.9 and t denotes the current iteration. v_{ij}^{0} is initialized at 0 and v_{ij}^{t} accounts for the previous gradients while determining current parameter update. Momentum dampens the oscillations in parameter search while speeding the training process.

2) RMSProp: In addition to accumulating gradients, RMSProp adjusts the learning rates for individual parameters. The update steps in this technique are as follows:

Repeat till convergence {

$$v_{ij}^{t} = \rho v_{ij}^{t-1} + (1-\rho)\left(\frac{dL}{dw_{ij}}\right)^{2}$$

$$\Delta w_{ij} = -\frac{\alpha}{\sqrt{v_{ij}^{t} + \varepsilon}}\left(\frac{dL}{dw_{ij}}\right)$$

$$w_{ij}^{t+1} = w_{ij}^{t} + \Delta w_{ij}$$

}

where, the hyper-parameter ρ is usually kept at 0.9, v_{ij}^{0} is initialized at 0 and t denotes the current iteration. ε is a small number to the order of 1e-10 to prevent division by zero. RMSProp automatically reduces the learning rate if the gradient step is large, hence tuning each parameter with different learning rates.

3) Adam: Adam or Adaptive Moment Optimization combines the heuristics of Momentum and RMSProp. While Momentum accelerates the search in direction of minima and RMSProp

impedes the search in direction of oscillations, ADAM combines the benefit of both and usually performs very well in practice. The update equations are as follows:

Repeat until convergence {

$$v_{ij}^{t} = \beta_1 v_{ij}^{t-1} + (1-\beta_1)\frac{dL}{dw_{ij}}$$

$$s_{ij}^{t} = \beta_2 s_{ij}^{t-1} + (1-\beta_2)\left(\frac{dL}{dw_{ij}}\right)^2$$

$$\Delta w_{ij} = -\alpha \frac{v_{ij}^{t}}{\sqrt{s_{ij}^{t} + \varepsilon}}\left(\frac{dL}{dw_{ij}}\right)$$

$$w_{ij}^{t+1} = w_{ij}^{t} + \Delta w_{ij}$$

}

where, β_1 and β_2 are hyper-parameters, whose values are usually 0.9 and 0.99.

3.3.9 Parameter initialization

As seen in the forward propagation, the activations in the subsequent layers depend on the product of weights from the previous layers. Thus, initializing the weights with very small or very large values could result in vanishing or exploding gradients, respectively. Also, initializing weights with a constant number or zeros does not result in any learning as all the activations and gradients during back propagation undergo same updates and there is no source of asymmetry between the neurons.

To avoid exploding or vanishing gradients, and help neural network converge faster, the following initialization methods are commonly used:

1) Xavier Initialization: This initialization method is usually used in case of symmetric-around-zero activation functions such as tanh. This initialization sets weights from a uniform

distribution with bounds $\pm\sqrt{\frac{6}{n_i+n_{i+1}}}$. n_i and n_{i+1} are the number of input and output units in the weight tensor, also known as fan in and fan out, respectively. Biases are initialized at zero. This makes sure that the variance of activations and back propagated gradients throughout the neural network in maintained.

2) Kaiming Initialization: As mentioned above, Xavier's initialization is valid when a symmetrical about zero activation function is used. In case of other activation functions such as ReLU or Leaky ReLU, weights are usually initialized using Kaiming initialization. The weight are initialized using a normal distribution centered at 0 with variance $2/n_i$, where n_i is the number of input units in the weight tensor. Biases are initialized at zero.

The aim of using these initializations is to make sure that activations and gradients in deep neural networks are maintained to avoid vanishing and exploding gradient problem and help network converge faster.

3.3.10 Regularization

A deep learning network contains millions of parameters that could result in overfitting the training data resulting in lack of generalization. In order to control the capacity of neural networks to prevent overfitting, the following methods are used in general:

1) L2 regularization: In this technique, we add the Frobenius norm of the weight parameters in the loss function, pushing weights close to zero. The updated loss function is:

$$L(y, \hat{y}, w) = L(y, \hat{y}) + \lambda_F \left|\left|w\right|\right|_F^2$$

This regularization is also called weight decay as the weights are decayed linearly with a factor of $\lambda_F w$ during backpropagation.

2) L1 regularization: In this regularization, for each weight w_{ij}, we add the $\lambda_m|w\_ij|$ term to the loss function, called 1-norm. Thus, the updated loss function becomes:

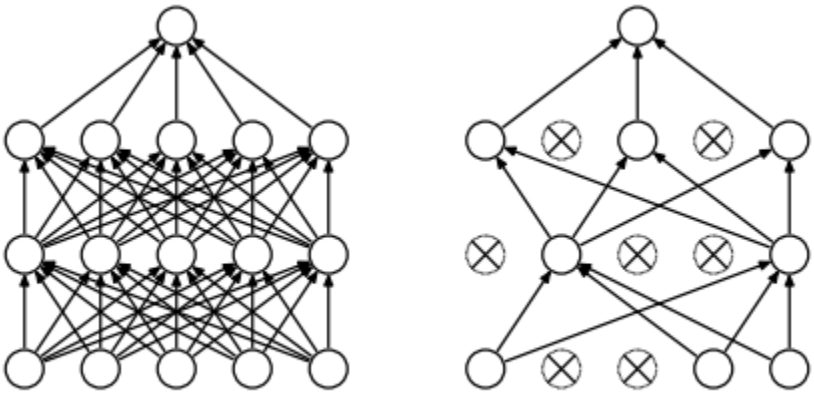
$$L(y, \hat{y}, w) = L(y, \hat{y}) + \lambda_m \left|\left|w\right|\right|_1$$

This technique makes the weight matrix sparse during optimization. Thus, it leads to a subset of most important weight parameters.

3) Elastic net regularization: This regularization lets us combine L1 and L2 regularizations and add the following penalty to the loss function:

$$L(y, \hat{y}, w) = L(y, \hat{y}) + \lambda_m \left|\left|w\right|\right|_1 + \lambda_F \left|\left|w\right|\right|_F^2$$

4) Dropout: This is an extremely efficient, effective and simple way that complements the above mentioned regularization methods. In Dropout, we only keep a subset of neurons (chosen with a probability p_{drop}) active during a cycle of forward propagation. This is like sampling a neural network from a full neural network and only updating the parameters of the sampled neural network based on the input data. Figure 7 which is taken from Srivastava et al. (2014), describes Dropout pictorially.



(a) Standard Neural Net (b) After applying dropout.

Figure 7. Dropout regularization

3.3.11 Bias-Variance tradeoff, Overfitting and Underfitting, and model generalization

A model is said to be having high bias if it is too simple to capture the underlying distribution in the data, whereas, model is said to have high variance when it fits too well to the training data. In the case of high variance, the model is not generalizable and performs poorly on the unseen data. Usually linear algorithms such as linear regression suffer from high bias, whereas algorithms such as neural networks suffer from high variance due to a large number of parameters in the model. When a model is too sensitive and fits very flexibly to the training data, it is said to overfit the training data. On the other hand, when a model is not able to capture important signals from the training data and is not flexible enough to generalize, then it is said to underfit the training data. A model with high bias or high variance is not generalizable enough and is liable to perform poorly on the unseen dataset. Figure 8 depicts the case of underfitting and overfitting.

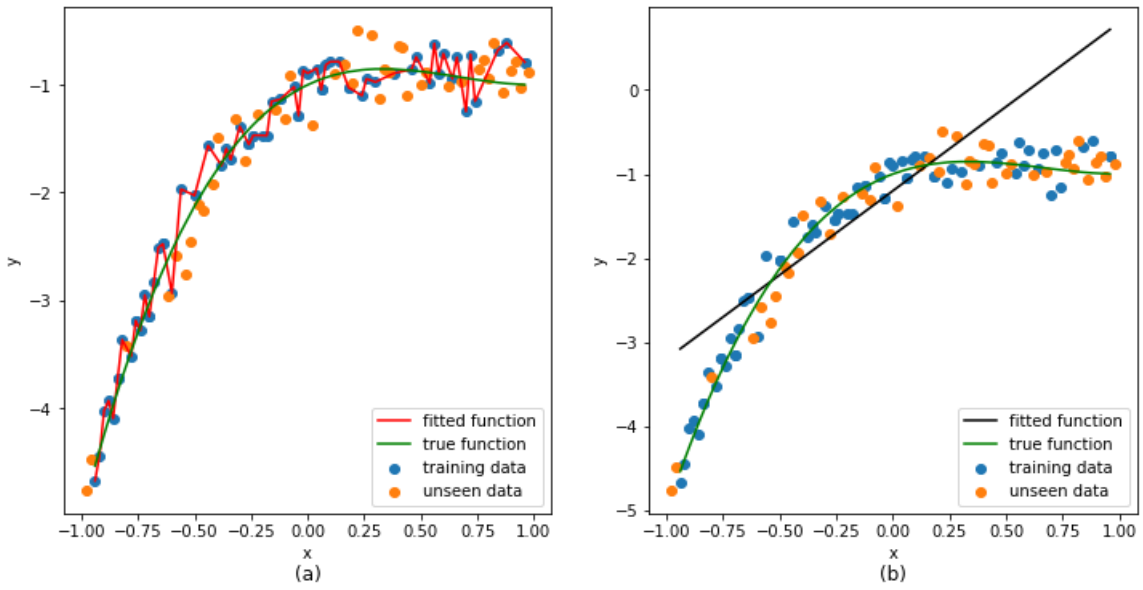


Figure 8. (a) Overfitting or high variance, (b) Underfitting or high bias

For a statistical algorithm, as model complexity increases, bias decreases and variance increases. In other words, for a predictive model, lower bias in parameter estimation tends to have

a higher variance, and vice versa. This is called Bias-Variance tradeoff. One can show that loss function such as mean-squared error can be described as a function of bias and variance, i.e.,

$$y = f + \epsilon,$$

$$E[(f - \hat{y})^2] = \left(f - E(\hat{y})\right)^2 + E\left[\left(\hat{y} - E(\hat{y})\right)^2\right] + \sigma^2,$$

$$E[(f - \hat{y})^2] = [Bias(\hat{y})]^2 + Var(\hat{y}) + \sigma^2$$

where, f is the target function and ϵ is noise (unavoidable error) such that $E(\epsilon) = 0$ and $Var(\epsilon) = \sigma^2$. Thus, the loss function is minimized when Bias-Variance tradeoff is balanced.

3.3.12 Tuning important hyper-parameters

As any other statistical learning algorithm, there are important hyper-parameters that we should tune in order to maintain bias and variance tradeoff. Thus, to find the sweet spot for Bias-Variance tradeoff and to help model converge better, we need to tune important hyper-parameters as mentioned below:

1) Learning rate (α): This is the most important hyper-parameter to be tuned for successful neural network convergence. In practice, different parameters need different learning rates. Also, smaller learning rates help in convergence but larger learning rates help avoiding sub-optimal minima or saddle points. Adaptive learning rate methods such as Adam provide parameter specific learning rates. In addition to that, cyclical learning rates makes the learning rate oscillate between bounds at regular intervals. Hence, once we set appropriate bounds for the learning rate, rest is taken care of.
2) Number of epochs (n_{epochs}): One epoch is when all the training samples are used for gradient update once. Using a large number of epochs can lead to overfitting, but a lower number may result in insufficient training. In practice, we use a technique called *Early Stopping*, in which the training stops when the desired loss does not improve after a certain number of consecutive

training epochs. Thus, we need to set the number of epochs for early stopping, after which training stops if the loss does not improve. Once the training stops, the parameter corresponding to the best loss value are returned.

3) Batch size (n_{batch}): This refers to the number of samples fed into the model at once for training while using mini-batch gradient descent. It is recommended to use a larger batch size as much could be supported by system's memory. Batch size also affects the computational time, lower batch size results in higher training time per epoch.
4) Dropout probability (p_{drop}): Dropout is used to control overfitting, though using a higher than required probability can cause lead to insufficient learning.
5) L1 (λ_m) and L2 (λ_f) regularization penalties: Higher L1 penalty leads to sparser parameter space and higher L2 penalty pushes weights closer to zero, hence controlling overfitting
6) Number of layers (l) and filter size (f_{conv}): Increasing the number of layers leads to a more complex network that results in increase in number of parameters. This in turn results in increased computational time and also potential overfitting. On the other hand, using lower number of layers could lead to a very simple network. Large filter size has a larger receptive field, hence captures more generic feature from the data and results in lower dimensional output, whereas a smaller filter size extracts more local information.

3.3.13 Training-Validation-Test sets and learning curves

Typically in machine learning, the data is divided into 3 parts: Training, Validation and Testing. The idea is to use training data for learning the parameters of the neural network, use the validation data to tune the hyper-parameters and finally use the test data to gauge the unbiased performance of the final model to achieve a benchmark for the performance the model will have on the real world data. In order to tune the hyper-parameters, for example learning rate, we use

validation loss as a criteria for early-stopping. Theoretically, it is possible to achieve a zero training loss by increasing the network complexity and training for longer period of time, but the validation loss serves as an indicator for overfitting. A few important points to keep in mind while creating the Training-Validation-Test sets are as follows:

1) Most of the data should be used for training as deep learning models thrive on the quantity of data available. Typically, over 70% of data serves as training that could go above 90% depending upon the problem at hand.
2) Testing data should be a sample from the real world data that our model is likely to encounter in practice. In case the test data does not represent the true data distribution, the accuracy on the test set would not be a good indicator for the model performance.
3) Validation and test sets should have a similar distribution. This will make sure that the hyper-parameters are tuned with respect to the true data distribution.

A learning curve is the plot of model leaning performance over time. As a model trains, one can study the learning curves to understand model behavior such as overfitting/underfitting or unrepresentative training/validation datasets by observing the learning curve. A learning curve basically demonstrates training loss and validation loss in the same plot against training epochs. Figure 9 shows an example of the learning curve.

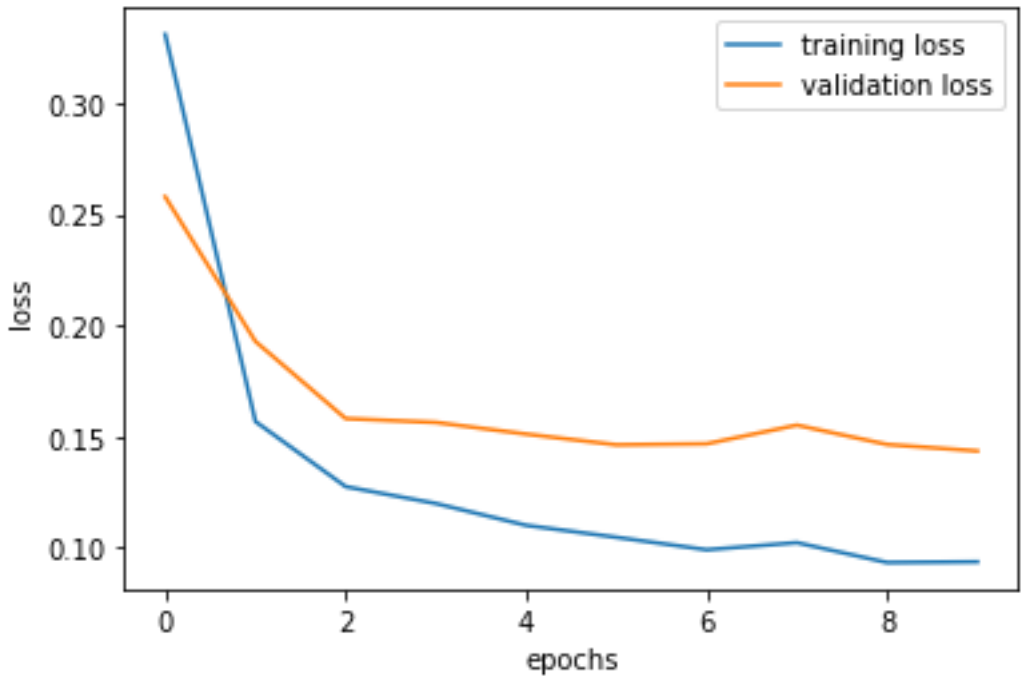


Figure 9. Learning curve

3.4. Sensor error estimation algorithm

In this section, we introduce the algorithm that has been implemented to quantify sensor errors and reconstruct erroneous flow. Firstly, we begin by abstracting the concerned road network to a graph G such that road segments containing multi-link level sensor sets are divided into multiple links where a link contains only one sensor, links with no sensors are eliminated and origin-destination nodes that connect more than one node are divided into multiple nodes. Next, we simulate traffic flow in the network using the O-D demand. Flow could be simulated assuming static or dynamic assignment, but the dynamic flow distribution would be closer to the true traffic distribution. We consider the generated flow as the ground truth and introduce measurement errors to generate observed flow samples as explained in section 3.1. Next, we divide observed flow samples to training, validation and test sets, where neural network model is trained on the

training set and hyper-parameters are tuned using the validation set. Once trained, the performance on the test set serves as the benchmark for the accuracy of the model. Lastly, the predicted systematic errors are used to reconstruct the flow. Figure 10 depicts the algorithm as a flow chart.

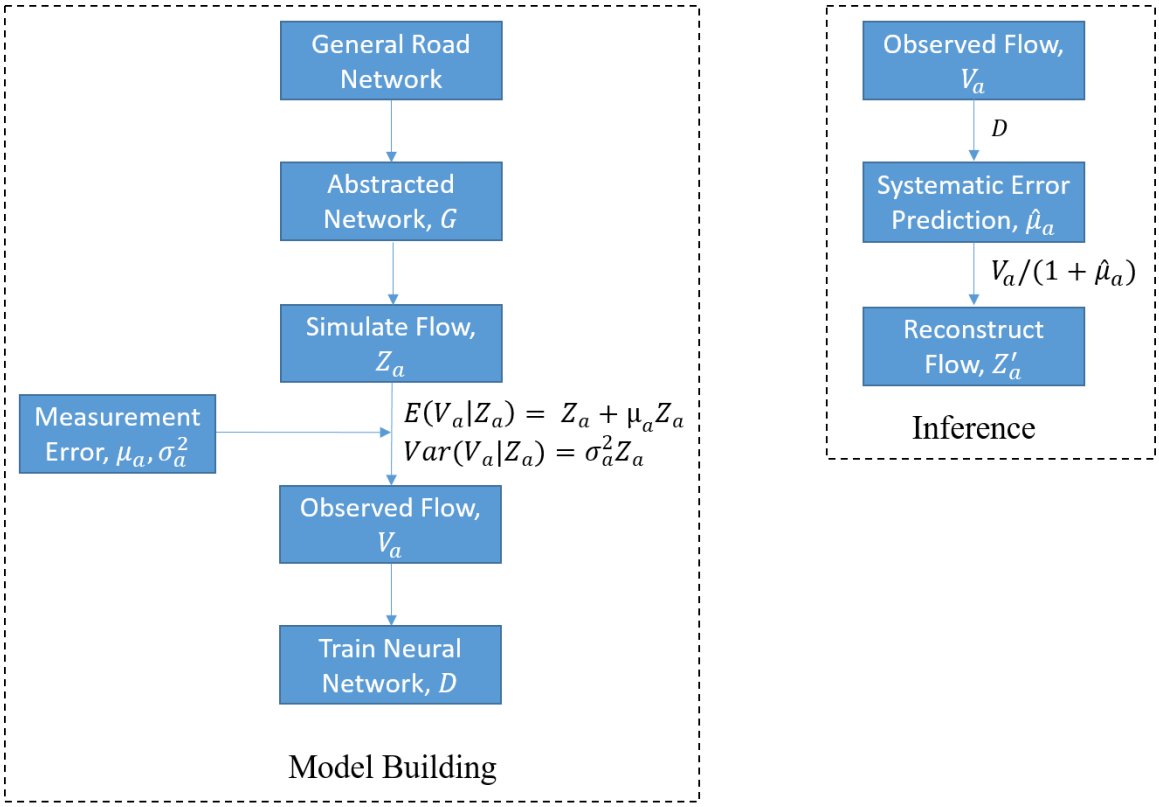


Figure 10. Sensor error estimation algorithm

4. Methodology

4.1. Network in consideration and graph abstraction

In this study, we consider a network in which there are no multi-link level sensor sets and every link has one sensor. In the real-world road networks, this is hardly the case. Usually, the freeways consist of multiple lanes, hence a segment has multiple sensors. Thus, before we apply our algorithm, we need to perform graph abstraction as introduced by Yang et al. (2017). The nuances of the algorithm are explained here.

As described in section 3.2, $G = \{N, A\}$, where N and A are the set of nodes and links in the graph, respectively. Let I be the set of intermediate nodes, $n = |I|$ and $s = |A|$, where $|.|$ represents cardinality of the set. Let $A^{+}_{(i)}$ and $A^{-}_{(i)}$ be the set of links that enter or exit a node i. Let P be the node-link adjacency matrix of size $n\ by\ s$, where $P_{ia} = 1$ if $a \in A^{+}_{(i)}$, -1 if $a \in A^{-}_{(i)}$, 0 otherwise. Let $\tilde{G}$ be the graph obtained after contraction, and $\tilde{P}$ be the node-link adjacency matrix for graph $\tilde{G}$. Let $\tilde{K}$ be a vector that stores the number of link level sensor sets on an edge and $\tilde{N}$ be the vector that designates whether a node is intermediate or not by $n_i = 1$, if a node is intermediate. The graph contraction algorithm is provided below.

Algorithm for Graph Contraction

$G = \tilde{G}, P = \tilde{P}, \mathrm{N} = \tilde{N}, \mathrm{A} = \tilde{A}$
for $a = \{i^-, i^+\} \in A$:
    while $K_a > 1$:
        Add a new row i' to $\tilde{P}$ and a new column a'
        $p_{i^-,a'} = -1, p_{i^+,a} = 1, p_{i',a'} = 1, p_{i',a} = -1$
        $K_a = K_a - 1, K_{a'} = 1, N_{i'} = 1$
    end
    if $K_a = 0$:
        Remove row i^- and column a
        $p_{i^+,:} = p_{i^+,:} + p_{i^-,:}, N_{i^+} = N_{i^+} N_{i^-}$
    end
end
for $a = \{i^-, i^+\} \in A$:
    if $a = \phi$:
        Add two new row j and j'
        $p_{j,a} = 1, p_{j',a} = -1$
    end
end
for $i \in N$:
    if $N_i = 0$:
        for $a \in A_{(i)}$:
            if $|A_{(i)}| > 1$:
                Add a new row i'
                $p_{i',a} = p_{i,a}, p_{i,a} = 0, N_{i'} = 0$
            end
        end
    end
end

Next, we consider the modified Nguyen-Dupuis network that is shown in figure 11. This network consists of 6 origin-destination (OD) pairs, 19 intermediate nodes, and 50 directed links and is already abstracted. All the links in this network are bi-directional with asymmetric flows. In the next section, we will be simulating flow in this network to generate flow count data.

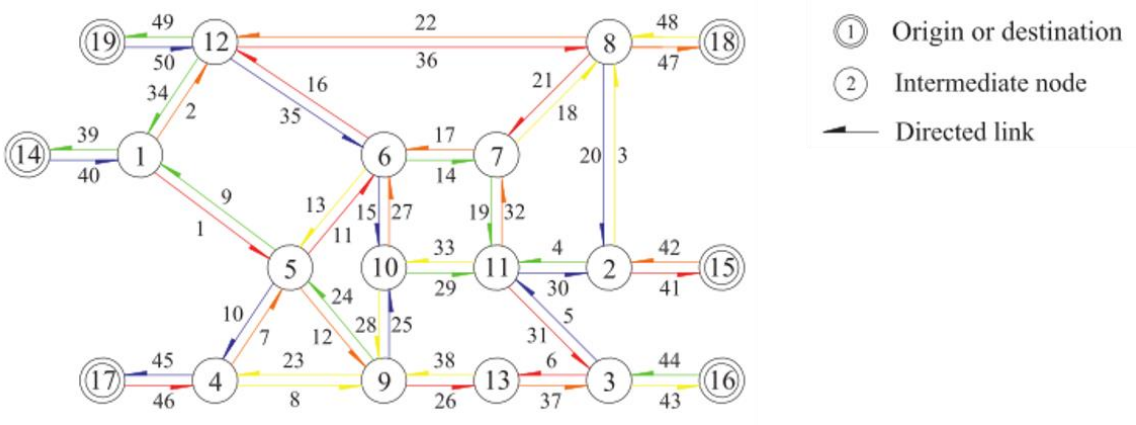


Figure 11. Modified Nguyen-Dupuis network

4.2. Generating data by flow simulation

We generated the simulated traffic flow data via the dynamic traffic assignment using Sumo. Sumo (Simulation of Urban MObility) is an open source microscopic traffic simulation package, with the capability of performing dynamic traffic assignment. A bidirectional modified Nguyen-Dupuis network is constructed within Sumo environment. The network consists of 6 origin/destination, 13 intermediate nodes and 50 directed links. Each of the intermediate node is considered to be a signal-controlled intersection. One inductive loop sensor is embedded underneath each link adjacent to its downstream intersection.

In total we carried out 365 independent simulations. Each simulation simulated the traffic flow for a 24-hour time period, among which the total 24-hour OD was split into each hour, based on some randomized daily traffic timelines, the average hourly ratio of which is shown in Figure 12. The 24-hour OD pairs for each of the simulations was generated by adding normal randomness onto an OD matrix, which is deterministic throughout the 365 simulations. Since the OD and the traffic signal controls themselves were not of interests, they were tuned only to enable non-fixed routing choices, i.e., the traffic flow of some OD pairs will have more than one routing option so

that the OD-to-traffic flow does not follow solely linear mapping. Figure 13 shows the average daily traffic profiles for 3 specific links, where the different traffic flow patterns indicate the existence of the non-linear mapping. In addition to this, the settings allowed network to have congestion which results in an imbalanced traffic flow conservation as shown in Figure 14. This also shows that in reality, the effect of shockwaves caused by queuing and dequeuing between links may still cause a significant violation of the flow conservation at the intersection.

Traffic data (traffic counts and speeds) were collected from each of the inductive loop sensor at a 10-minute aggregation level. Therefore, for each sensor, we obtained 365*24*6=52560 samples for both traffic counts and speeds.

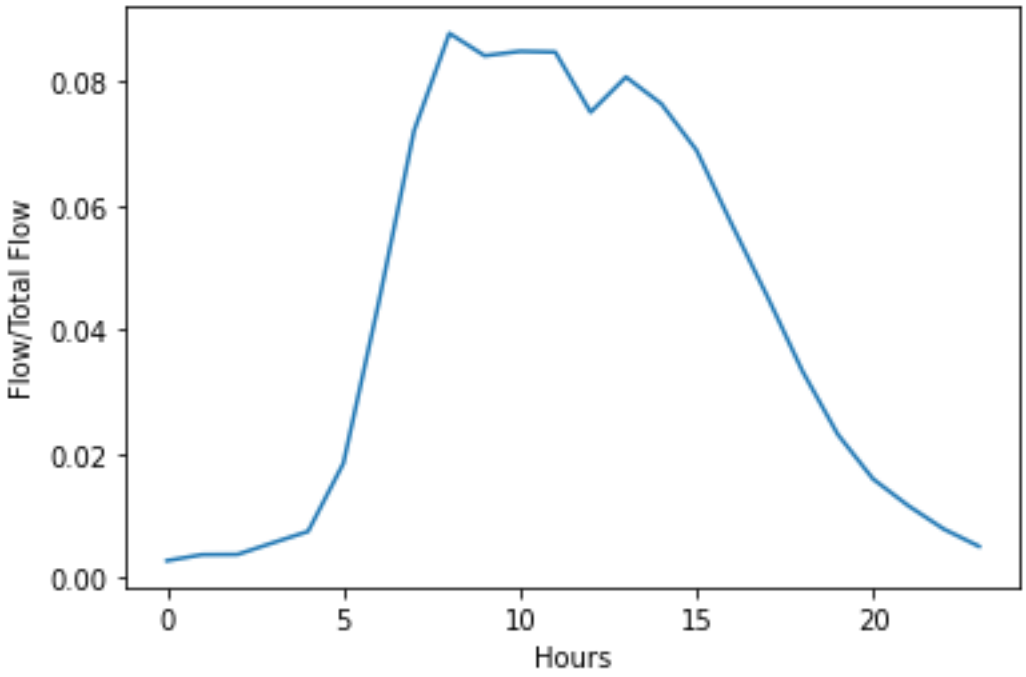


Figure 12. Average hourly flow ratio in network

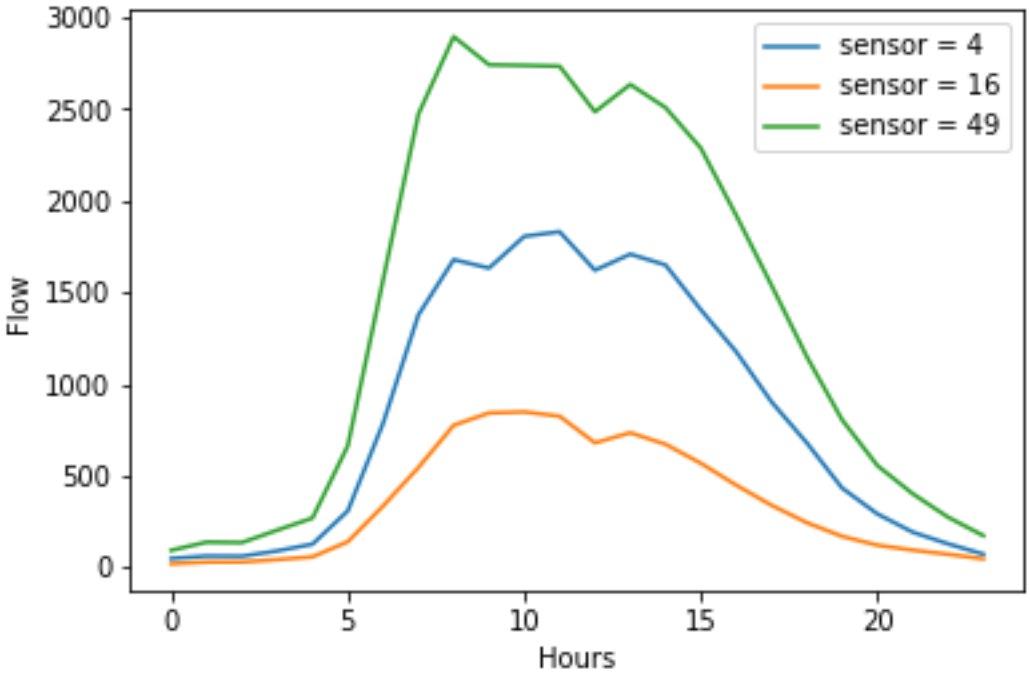


Figure 13. Average daily traffic profiles for specific sensors

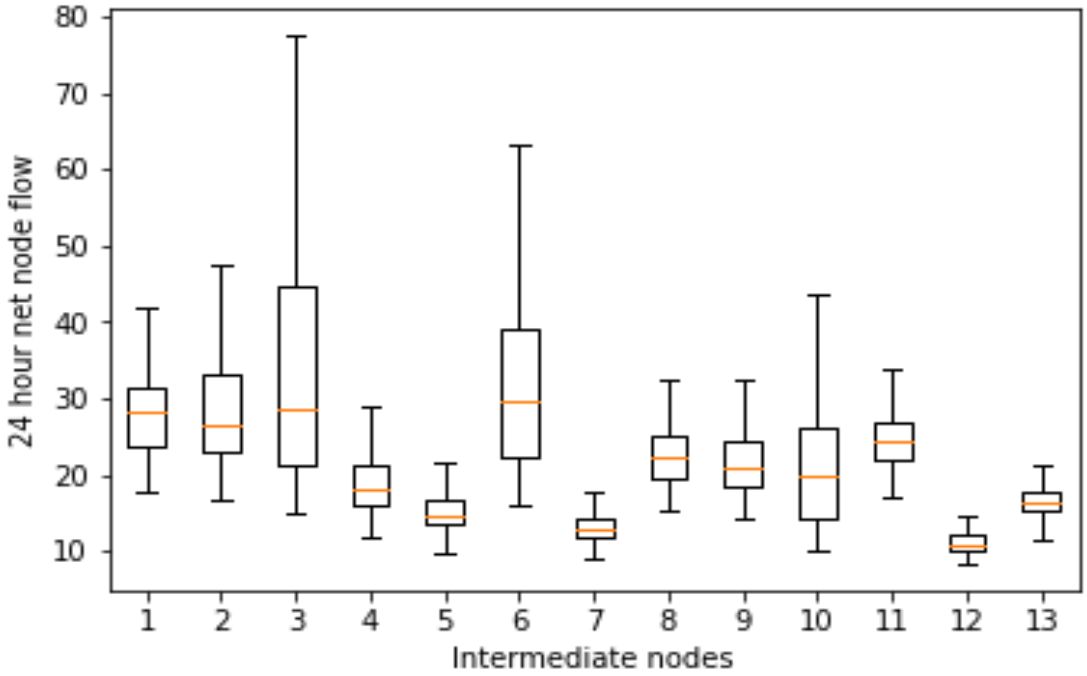


Figure 14. Average 24 hour node flow imbalance

4.3. Generating observed flow

After simulating the network to generate the base flow, we introduce measurement errors to obtain the observed flow. Here we explain how to generate measurement errors for a sample, then we can repeat these steps to generate measurement errors for multiple samples:

1. Firstly, we generate two s-dimensional vector representing mean systematic error ratios, $\mu_i \sim U^s(-0.5, 0.5)$ and random error coefficient, $\sigma_i' \sim U^s(0, \sigma_a)$ for a group i where U^s is a s-dimensional uniform distribution. σ_a is the random error ratio as defined in section 3.1 that dictates the magnitude of unexplained measurement error and places an upper bound on the accuracy of reconstructed flow with respect to the base flow. Each value in the vector μ_i and σ_i' correspond to a sensor in the network. We limit the systematic error ratios between -0.5 and 0.5, where the extreme values are a representative of the worst sensor condition.
2. Next, we demarcate sensors as calibrated with a probability p_{calib}, i.e., a sensor is marked fit with a probability p_{calib}. This helps in creating real-world like scenario where a few sensors are likely to be fit. In addition to this, it also adds slight randomness in the training data that helps in robust training.
3. Now, we generate systematic error ratio for a sample d of group $i, \mu_{i,d} \sim N(\mu_i, \sigma_\mu \mu_i)$. Here a reasonable assumption is made that the mean systematic error ratio, μ_i remains constant for over a group of samples (for example, a group could be a year with each day as a sample, so d would be from 1 to 365) and systematic error ratios for a sample are random fluctuations from the μ_i. For our study we assume σ_μ to be 0.3. The idea is to generate enough randomness in the error ratios such that our model learns robust features and can perform well during the testing phase.

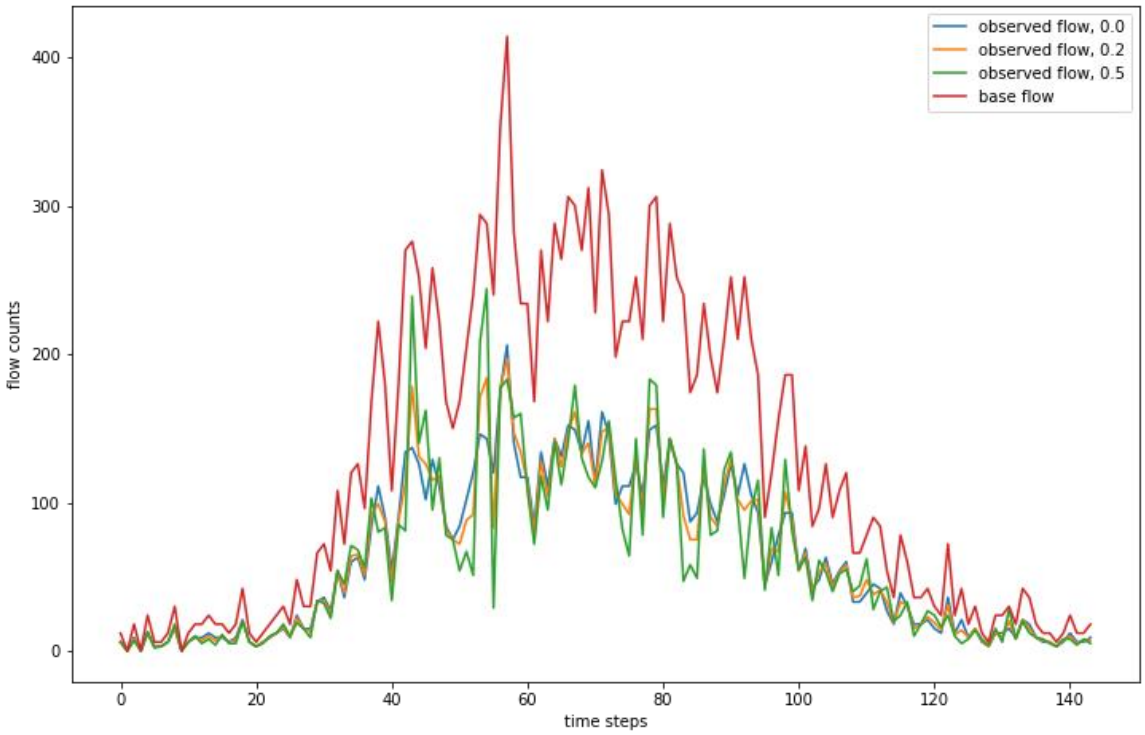
4. Next, we introduce random errors for sample d, $r_d \sim N(0, \sigma_i' Z_d^{0.5})$ for all the sensors for group i, where Z_d is the base flow for the sample d. Finally, we update the base flow for a sample d using $\mu_{i,d}$ and r_d to generate the observed flow $V_{i,d}$.

We repeat these steps to generate systematic error ratio for D samples in the group i and then to generate observed flow for multiple groups. The algorithm for generating observed flow for y groups is summarized below.

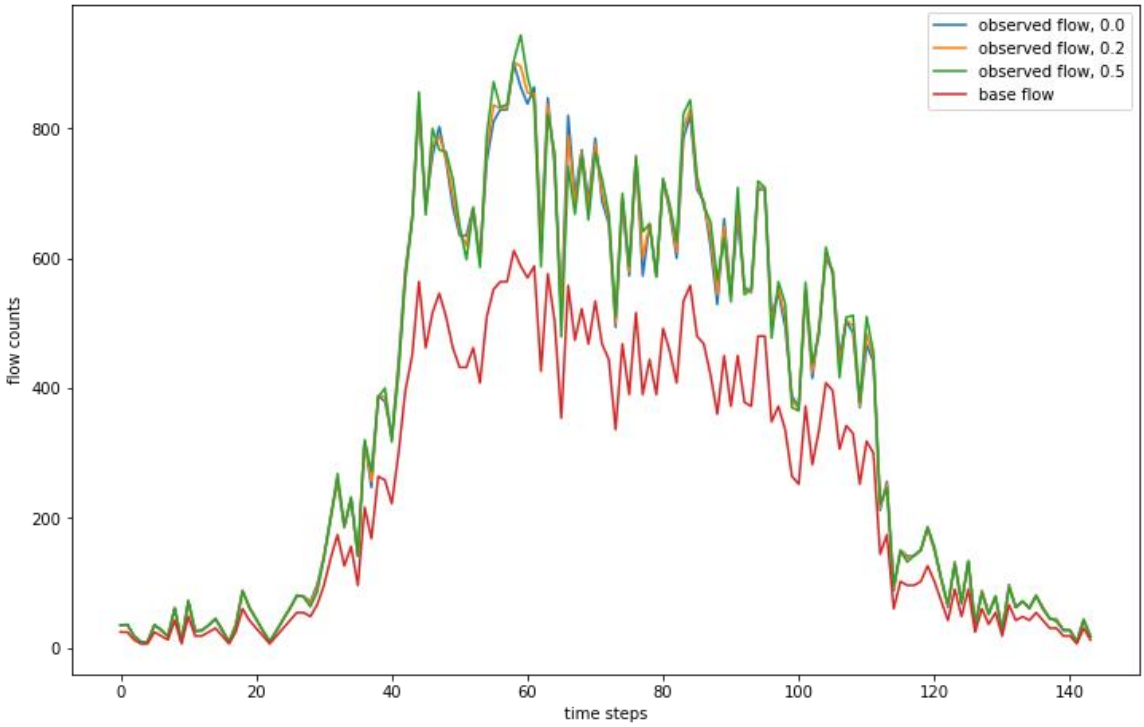
Algorithm for generating observed flow

```
# s is the number of sensors
# Y is the number of total groups
# Z_d is the base flow for sample d, where total samples for group i are D
for i in 1:Y:
    μ_i = U(-0.5, 0.5, size = s)
    c = binom(s, p)
    μ = μ *(1-c)
    σ_i' = U(0, σ_a, size = s)
    for d in 1:D:
        μ_d = N(μ_i , 0.3* |μ_i|)
        r_d = N(0, σ_i' * Z_d^0.5)
        V_d = Z_d*(μ_d+1) + r_d
        V_d = round(V_d)
        save (V_d, μ_d)
    end
end
```

Thus, we generate $D * Y$ number of time-series samples with the corresponding systematic error ratios. Figure 15 compares the base flow with the generated observed flow for different levels of random error ratio, σ_a.



(a) Sensor = 4, $\mu_{i,d} = -0.5$



(b) Sensor = 49, $\mu_{i,d} = 0.5$

Figure 15. Base vs Observed Flow with varying σ levels

4.4. Deep neural network training

In the previous section, we generated the observed flow data (X) with corresponding systematic error ratio vectors (y). Thus, we have a supervised learning problem at hand, where we use the X to predict y by training a deep neural network. The idea is to start with an initial model, and then update the hyper-parameters such that our model learns the important high-level spatial-temporal features without losing the generalization ability. In this section, after necessary data manipulations we introduce the initial neural network along with the initial hyper-parameter setting.

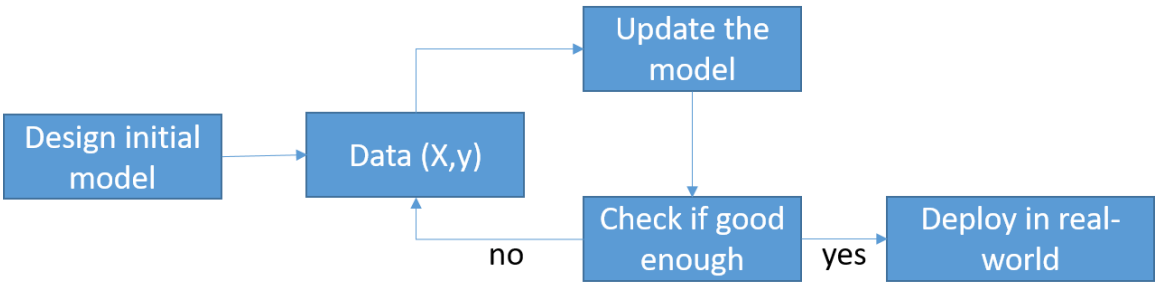


Figure 16. Typical training flow for supervised learning

4.4.1. Data generation and manipulation

We begin by generating 140 groups ($Y = 140$), where each group has 365 samples ($D = 365$). The total data is then split in training and validation sets. The random error ratio σ_a and probability of calibrated sensors p_{calib} both are set to 0.2. The dimensions of the total flow data is therefore (51100, 144, 50) that is mapped to systematic error ratio of dimension (51100, 50), where there are 140*365 (=51100) time-series samples, where each sample has 144 time-steps (temporal dimension) and each time-step has 50 sensor readings (spatial dimension). In this study, we use 10% data as validation data, hence we have 120 groups of data for training and 20 groups of data

for validation. Thus, the resulting dimensions of the training flow data and validation flow data are (43800, 144, 50) and (7300, 144, 50), respectively with corresponding systematic error ratios of dimension (43800, 50) and (7300, 50).

Next, we center and scale our training and validation datasets. This process, called standardization has proved to be vital for successful neural network training. We compute mean and variance using the training data and standardize the total dataset using these parameters. The mean and variance are computed for each spatial-temporal location.

$$\mu_{train} = \frac{1}{q_{train}} \sum_{i=1}^{q_{train}} X_{train}$$

$$\sigma_{train}^2 = \frac{1}{q_{train}} \sum_{i=1}^{q_{train}} (X_{train} - \mu_{train})^2$$

$$\frac{X - \mu_{train}}{\sigma_{train}} \rightarrow X$$

In our case, q_{train} = 43800. The dimensions of μ_{train} and σ_{train} is (144, 50).

4.4.2. Why CNNs?

As per our knowledge, this is the first time when deep learning is being used for estimating sensor errors, hence, a natural question arises: why CNNs? Although first time for studying sensor health, CNNs have been used for a variety of transportation research that involves traffic data. The reason being the capability of CNNs to leverage the convolutions to exploit the multi-dimensional traffic data to obtain high-level spatial-temporal features. The beauty of CNNs is that there is no need for manual feature generation as in the case of other statistical algorithms, rather passing raw data (in the structured format, for example, time-series tensor in this case) as input is sufficient.

With the tensor representation of traffic count data in this study, where time and space are represented with separate dimensions, CNNs work with both the dimensions together. Due to these advantages, CNNs have been used on traffic data previously in studies related to predicting congestion (Fouladgar et al. 2015), passenger demand forecasting (Ke et al. 2017) and traffic flow prediction (Wu et al. 2018). But we should not limit ourselves to transportation literatures and should also draw motivation from other sectors where data is in the form of multi-dimensional time series. CNNs were used to study patient Electrocardiogram (ECG) signals to identify patient-specific cardiovascular problem and solution (Kiranyaz et al. 2014). There are many more examples like these, but the idea is clear that CNNs are a natural choice for extracting high-level features from multi-dimensional time-series data.

4.4.3. Initial neural network setup and training

We begin by defining our baseline architecture that consists of 1 convolutional layer (CNN) followed by 2 fully connected layers. A CNN layer as defined in figure 4 contains a convolutional layer, followed by a batch-normalization layer and a max pooling layer, the output of which is passed to a ReLU activation function. The output of the CNN layer is passed onto the 2 fully connected layers (with batch norm and ReLU), which are the last 2 layers in our architecture. Figure 17 depicts our baseline architecture.

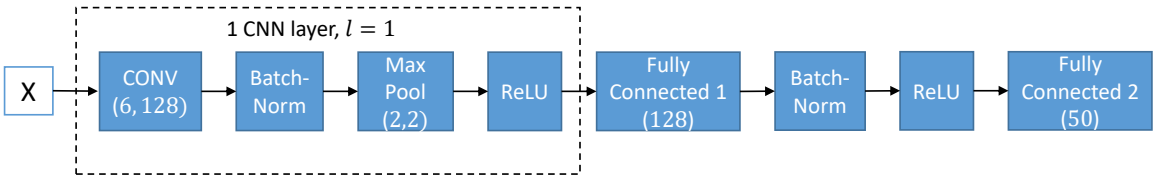


Figure 17. Baseline neural network architecture

The hyper-parameters for our baseline architecture are set as follows:

1. Learning rate and number of epochs: As explained in section 3.1.12, we use learning rate scheduler such that the learning rate jumps between the bounds cyclically. In the section 4.4.3 we explain how to set these bounds. We use triangular learning rate scheduler for this study as it results in a stable learning curve. Number of epochs (n_{epochs}) are regulated using early-stopping with patience parameter 20 as explained in section 3.1.12. Although, to limit the training time, we set the maximum n_{epochs} to 500.
2. Batch size: Batch sizes are typically set as an exponent of 2 (16, 32, 64,128, and so on). For the initial model, we keep the batch size at 128. Using a higher batch size can risk memory overflow and hence, might not be suitable for training.
3. Regularization: In our initial model, we have no regularization. We observe the learning curve obtained after training and then regularize the model to avoid overfitting.
4. Number of channels (c^l) and filter size (f_{conv}): As our initial baseline model only has 1 CNN layer, we need to set the number of channels and filter size for the weight kernel of only this layer. We set the number of channels to 128. Hence, after convolution, our 50-channel data in transformed to a 128 channel output. This is a reasonable number of channels to begin with as it maintains tradeoff between computational complexity and amount of features extracted from the data. We set the filter width (also known as kernel width) to 6. Choosing a larger filter results in extracting more general temporal information, whereas a smaller filter extracts more local information.
5. Pooling: We use max pooling with both filter width d_{pool} and stride s_{pool} set at 2. Thus, the shape of the output will be reduced by half in the temporal dimension, keeping the computational complexity in check.

6. Fully connected layer neurons: In our base network, we pass the output of CNN layer to 2 fully connected layers. The number of neurons for the first fully connected layer is set to 128, while for the second layer is set to be 50 (equal to the length of systematic error ratio vector, y).
7. Performance metric and loss function: In this study, we use Mean absolute flow reconstruction error (MAFRE) as the performance metric. MAFRE is calculated by reconstructing hourly flow by using the predicted systematic errors and then comparing it with the ground truth flow. Basically, MAFRE quantifies the accuracy of reconstructed flow, which resonates with our ultimate goal of correcting the erroneous flow as best as possible. The formula for evaluating MAFRE is given below:

$$MAFRE\% = \frac{1}{n}\sum_{i=1}^{n}\left|\frac{flow\_re_i - flow\_true_i}{flow\_true_i}\right| * 100\%$$

where, $flow\_re_i$ is the reconstructed hourly flow, $flow\_true_i$ is the actual hourly flow and n are the total number of samples that are used to evaluate model performance. Mean absolute percentage error (MAPE) criteria defined in table 2 is the natural candidate for the loss function because minimizing the MAPE in predicted systematic ratio reflects directly towards minimizing MAFRE. Hence, during back propagation, the parameter updates will be performed using MAPE loss. The difference between loss metric and performance metric is that, the loss metric is used to back propagate errors to train neural network, whereas performance metric is an easily interpretable measure that is used to quantify the model performance.

One important point to be noted here is that the loss function does not contain any physical aspects related to the network topology as is the case in usual statistical analytical models. The data itself abides by all the constraints and model learns those patterns during training. Thus,

the loss function is only dependent on predicted (function of model parameters) and true systematic error ratios.

8. Optimizer and parameter initialization: We use Adam optimizer that is an adaptive learning rate optimization method as discussed in section 3.1.8. As we are using ReLU as the activation function, we use Kaiming Initialization to initialize the kernel weights. Biases are initialized with 0.

We use the deep learning library Keras for our neural network computations and leverage the NVIDIA CUDA 10.1 GPU provided by the cloud platform Google Colaboratory to fasten the training. After data manipulation and defining the model architecture with the above mentioned hyper-parameters, the following steps explain the training process:

1. Learning rate bounds: We train our model for a few epochs by varying the learning rate. Figure 18 depicts the loss vs learning rate (lr) plot which helps us to choose the learning rate bounds for cyclical learning rate scheduler. From the plot we choose the learning rates corresponding to the steepest slope or maximum decrease in the loss. In this case, we choose the learning rate bounds to be (1e-3, 2e-3).

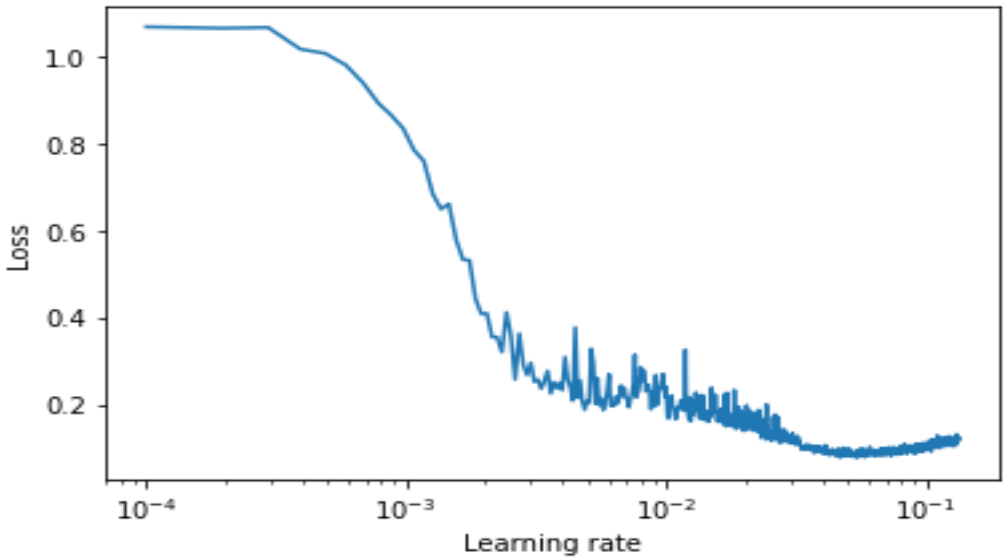


Figure 18. Loss vs learning rate curve

2. Next, we fit the deep learning model with the above learning rate bounds. Since we are using early-stopping with patience criteria set as 20, the learning will stop automatically once the validation loss does not improve for 20 successive epochs. Figure 19 depicts the architecture of our baseline deep learning model with the total number of parameters.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 144, 50)           0
_________________________________________________________________
conv1d_1 (Conv1D)            (None, 144, 128)          38528
_________________________________________________________________
batch_normalization_1 (Batch (None, 144, 128)          512
_________________________________________________________________
activation_1 (Activation)    (None, 144, 128)          0
_________________________________________________________________
max_pooling1d_1 (MaxPooling1 (None, 72, 128)           0
_________________________________________________________________
flatten_1 (Flatten)          (None, 9216)              0
_________________________________________________________________
dense_1 (Dense)              (None, 128)               1179776
_________________________________________________________________
batch_normalization_2 (Batch (None, 128)               512
_________________________________________________________________
activation_2 (Activation)    (None, 128)               0
_________________________________________________________________
dense_2 (Dense)              (None, 50)                6450
=================================================================
Total params: 1,225,778
Trainable params: 1,225,266
Non-trainable params: 512
_________________________________________________________________
```

Figure 19. Baseline model architecture

3. After the training stops, we compute the MAFRE on the validation set and plot the learning curve as shown in Figure 20. From the learning curve one can observe that the training terminated around 56th epoch, and so the validation loss cease to improve after the 36th epoch hinting insufficient learning and potential overfitting. The MAFRE on validation set is 5.24% that means our baseline model reconstructs the validation flow counts with an accuracy of 94.76%.

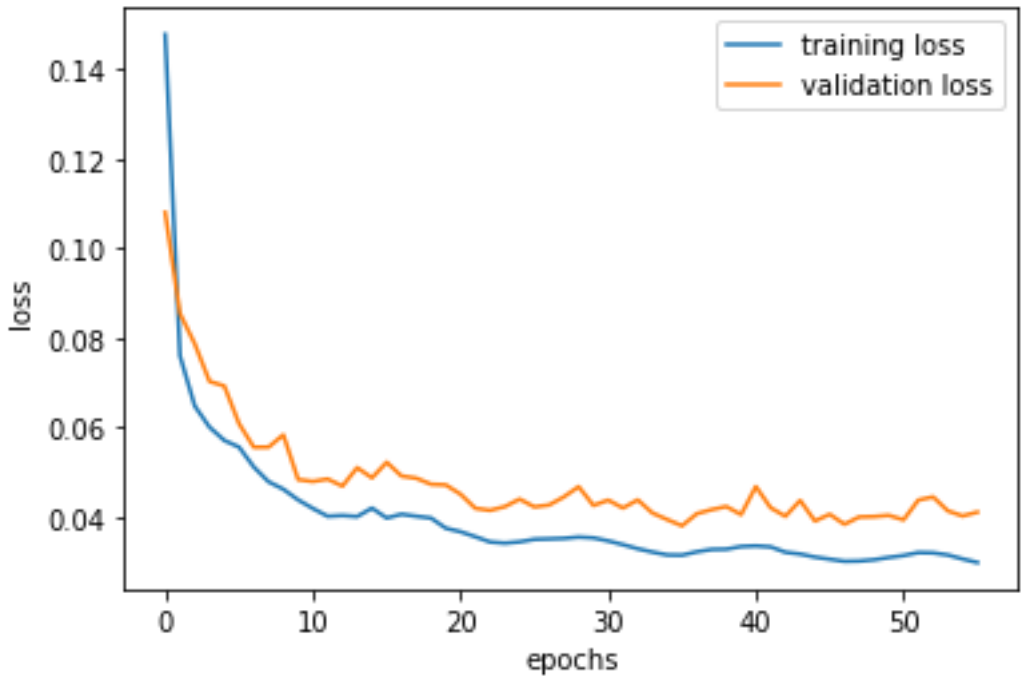


Figure 20. Learning curve for baseline model

The MAFRE value on the validation set serves as our baseline and in the next section we tune our hyper-parameters to improve the performance metric. The combination of hyper-parameters that results in the best performance is chosen as our final architecture.

4.4.4. Hyper-parameter tuning

In this section, we tune the baseline model to achieve better performance metric. The most important hyper-parameters that should be tuned are dropout, number of layers and filter width. Lastly, a few other architectures are also trained and compared to the tuned model. Following is the description of the process used to tune hyper-parameters:

1. Dropout (p_{drop}): Dropout is one of the most effective measures for controlling the complexity of the model and avoiding overfitting. We try 3 different values of p_{drop} and train the baseline

model with a dropout layer after the batch-norm layer of 1st fully connected layer as shown in figure 21.

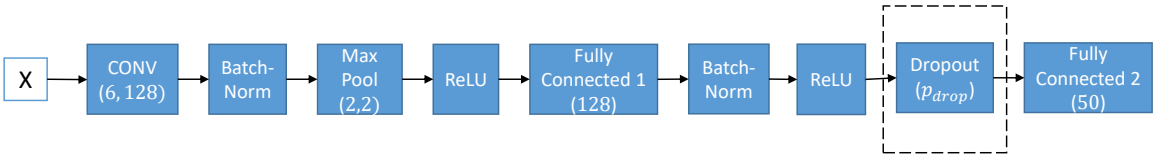


Figure 21. Baseline model with Dropout

Figure 22 shows the MAFRE with different values of dropout, from which we can infer that the network with p_{drop} = 0.05 performs best and the MAFRE metric improves from 5.24% (Baseline) to 4.21%. For further hyper-parameter tuning we fix the p_{drop} to 0.05 and our baseline performance metric to 4.21%.

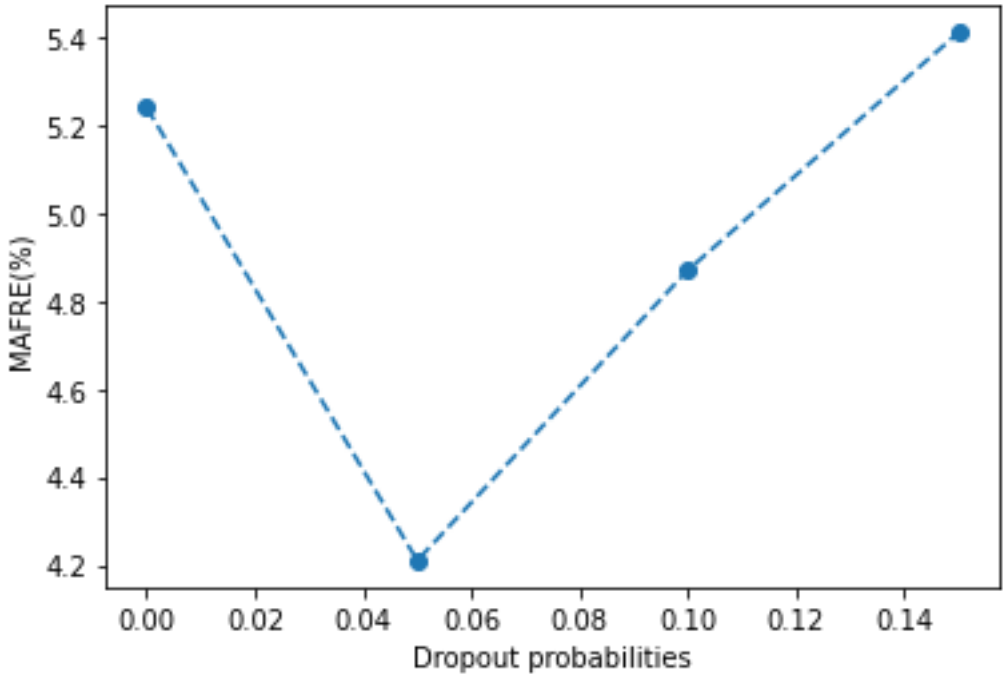


Figure 22. Effect of different Dropout probabilities on MAFRE(%)

2. Filter size (f_{conv}): We choose different filter widths and train the baseline model for each, keeping dropout probability at 0.05. Figure 23 shows the effect of filter width on the

performance metric. Choosing filter width as 1 implies that highly local features are extracted, not taking into account the temporal relationship in the traffic data, whereas choosing a higher filter width extracts more generic features. As can be seen from figure 23, filter width of 24 (corresponding to 4 hours) results in the best performance metric of 4.18%. Hence, we treat this as our new baseline.

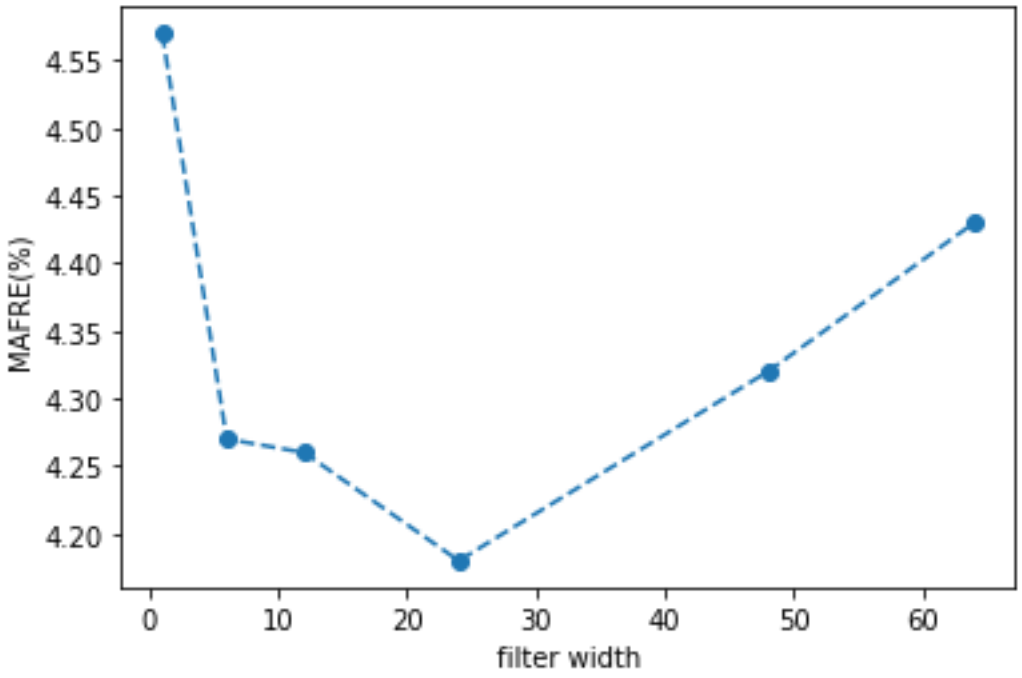


Figure 23. Effect of filter width on MAFRE (%)

3. Number of channels (c^l): For the baseline model with (f_{conv}, p_{drop}) = (24, 0.05), we evaluate performance for different number of channels for the CNN layer. As it can be seen from figure 24, 128 channels results in the best MAFRE value of 4.18% (current baseline).

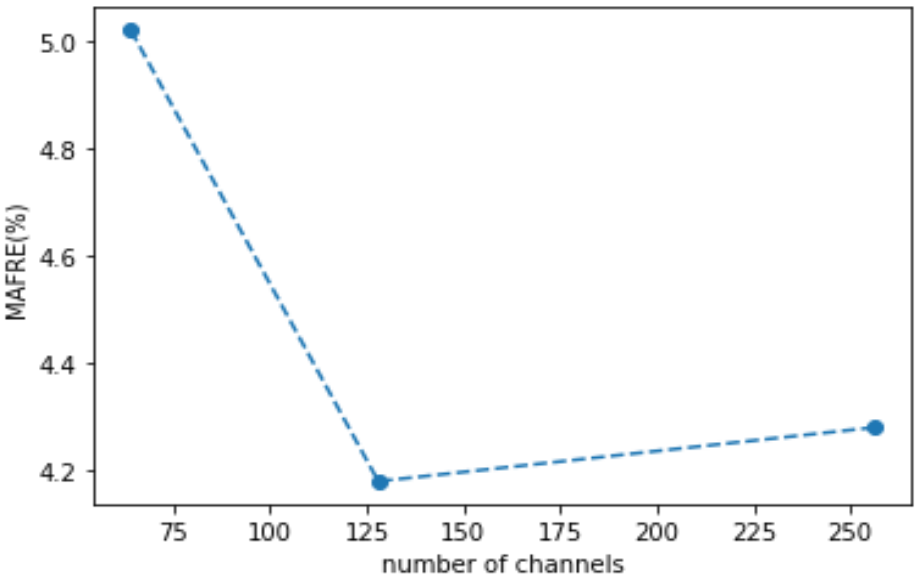
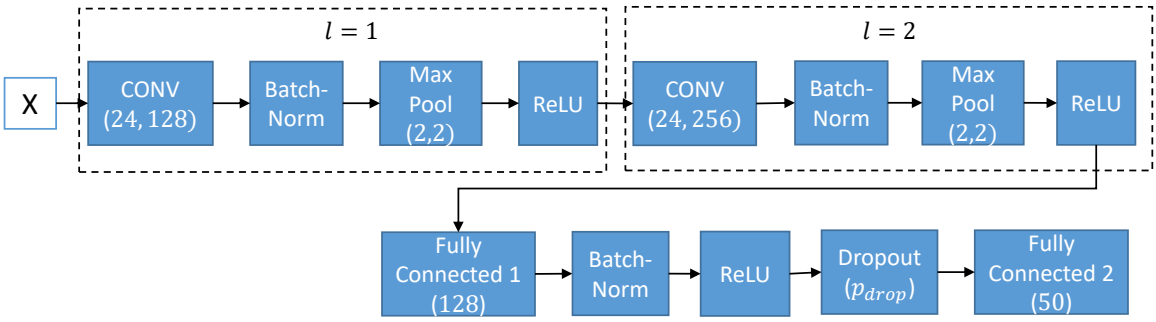
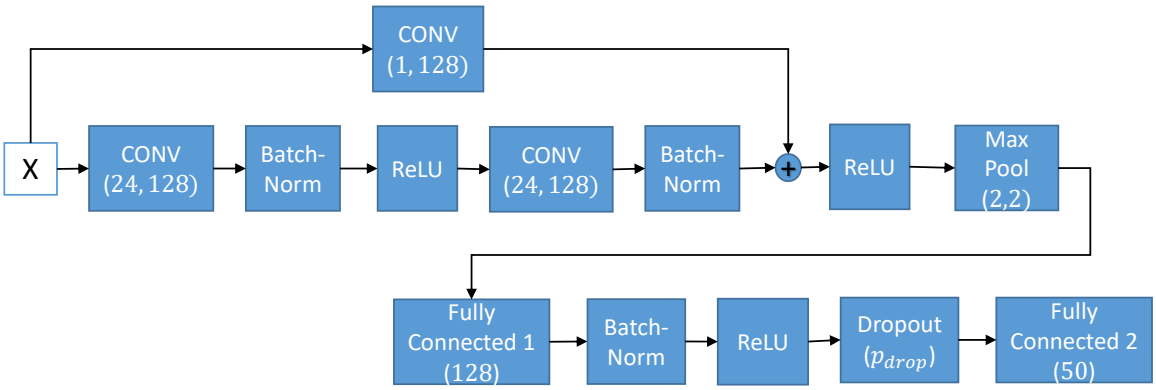


Figure 24. Effect of number of channels on MAFRE (%)

4. Different architectures: After tuning the dropout, filter width and number of channels for our baseline model, we have achieved the performance metric MAFRE at 4.18%. Next, we try other promising architectures to improve the performance metric further. Figure 25 shows other candidate architectures. In figure 25 (a), the model has an added CNN layer, thus increasing the complexity of the model and extracting even higher-level features from the traffic data. In figure 25 (b), we introduce a skip connection with a CNN layer with $f_{conv} = 1$. The skip connections have proved to improve learning in deep CNN architectures, and are worth trying in this study. Table 3 shows the performance metric for various architectures corresponding to different hyper-parameter values.



(a)



(b)

Figure 25. Architecture with (a) 2 CNN layers, (b) skip connection

Model	MAFRE (%)
Baseline	5.24
Baseline (tuned)	4.18
2 layer CNN (with $p_{drop} = 0.05$)	4.62
2 layer CNN-2 (with $p_{drop} = 0.1$)	4.82
Skip CNN-1 (with $p_{drop} = 0.05$)	4.48
Skip CNN-2 (with $p_{drop} = 0.1$)	5.02

Table 3. MAFRE(%) values corresponding to different architectures

Thus, after hyper-parameter tuning the best MAFRE value achieved is 4.18%. Figure 26 shows the learning curve for the final model.

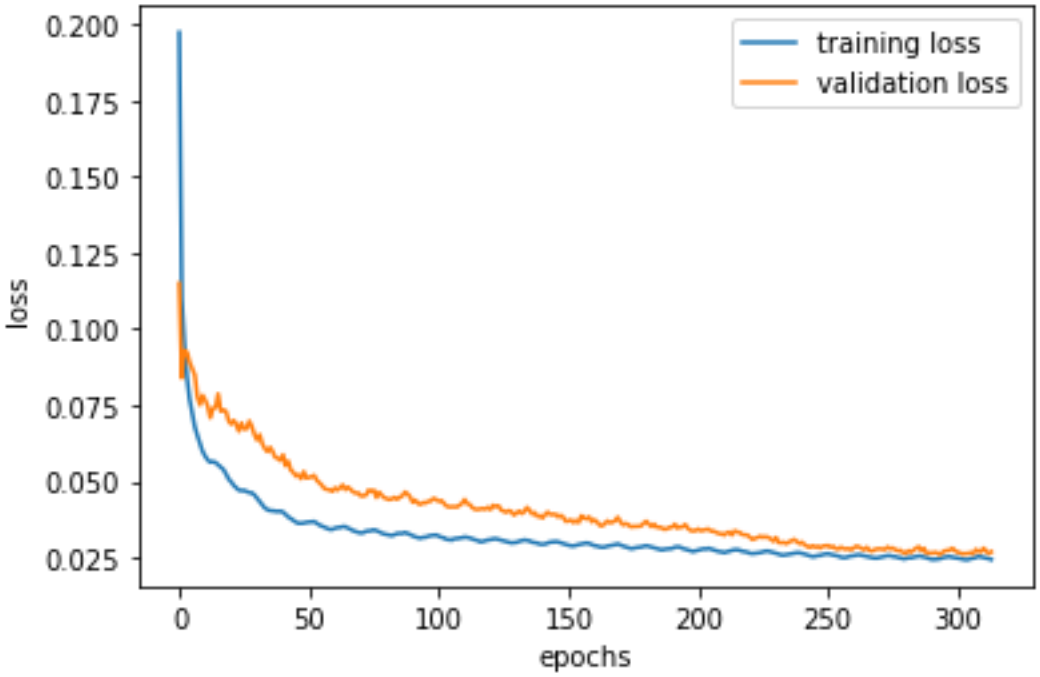


Figure 26. Learning curve for final model

From the learning curve, we can see that the validation loss approaches the training loss, which implies that the model had sufficient learning and is not overfitting in nature. Finally, we will train the model with appropriate hyper-parameters on the full data set (training + validation) and use it for further inferences. In the next section, we use the trained model for analyzing its performance and inferring errors on various test case scenarios.

5. Results

So far, we trained a deep learning model on traffic count data to predict systematic error ratios. To generate the data, we used 0.2 random error ratio (σ_a) and 0.2 as the probability of calibrated sensors (p_{calib}). The logic behind using these values was to introduce randomness in the training data to make the learning more robust. In reality, our network can have sensors with any level of σ_a and p_{calib}. For example, in California, according to PeMS around 2/3rd of the sensors are functioning properly. Our final model should be robust enough to identify the calibrated sensors as well as quantify the actual degree of systematic error irrespective of the level of random errors. In this section, we use our model to predict systematic errors for various test scenarios and validate the generalization power of the final model.

5.1. Effect of σ_a and p_{calib} on quality of flow reconstruction

We generate test cases corresponding to various levels of σ_a and p_{calib}. In total, 96 test cases are generated using combinations of 16 levels of σ_a (varying from 0 to 0.3) and 6 levels of p_{calib} (0 to 1). Note that these combinations also represent extreme cases, for example, where all the sensors are calibrated and the model is expected to predict that. We also incorporate random error ratio levels that are greater than the one used while training the model ($\sigma_a = 0.2$). Figure 27 shows the MAFRE metric values for these test cases before correction, after correction using predicted systematic error ratios and the unavoidable bias (error only due to random errors).

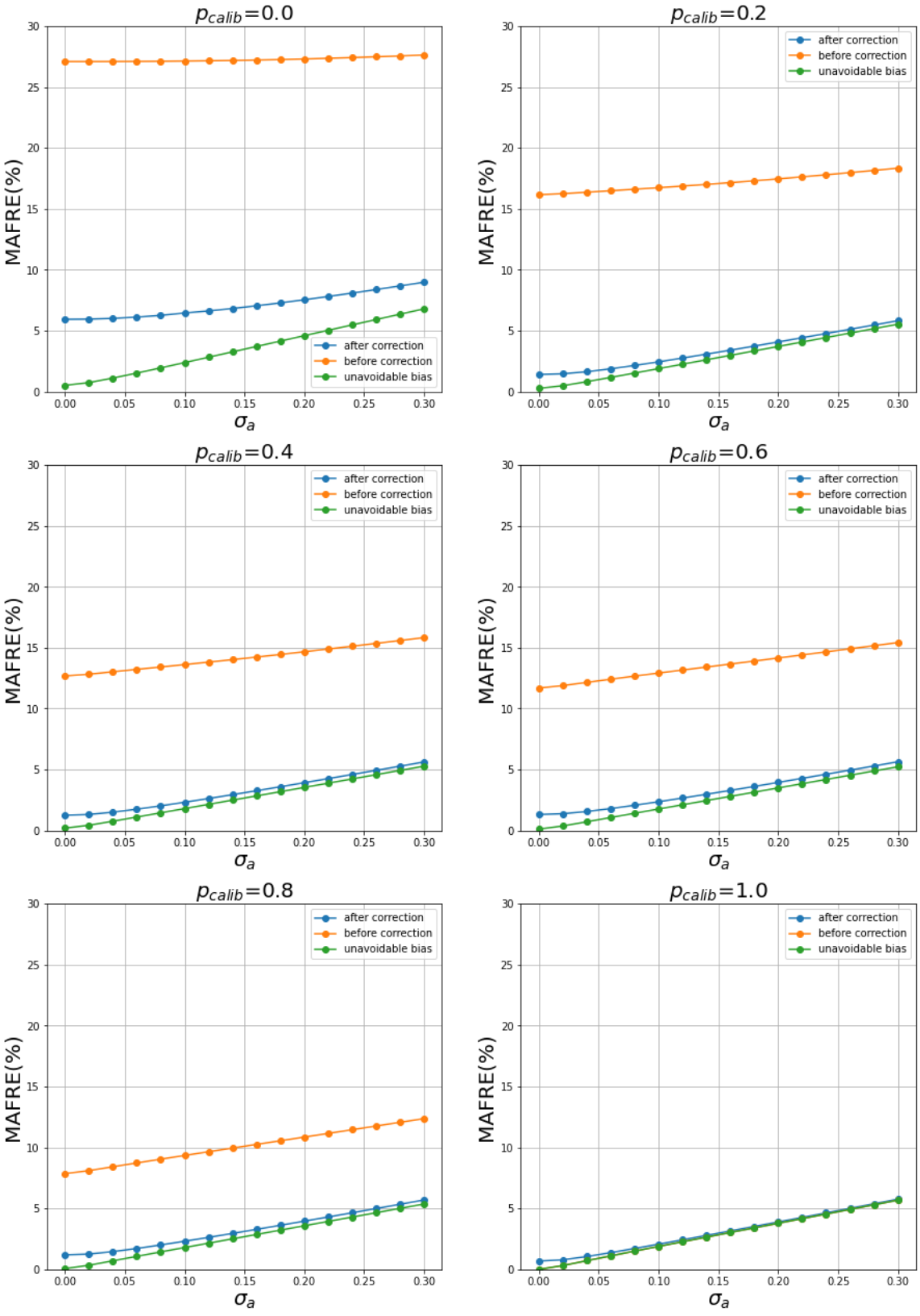


Figure 27. MAFRE (%) corresponding to various levels of σ_a and p_{calib}

From this figure, we can observe that except for the case where $p_{calib} = 0$, the MAFRE value after correction is very close to the unavoidable bias. In all the cases, the MARFE value has significantly improved relative to the MAFRE value before reconstruction. Also, in the extreme case, our model actually predicts that all the sensors are calibrated and the reconstructed flow has error only due to the random error. Figure 28 shows the percentage improvement after the flow correction along with maximum possible improvement.

$$improvement\ (\%) = \frac{MAFRE_{after} - MAFRE_{before}}{MAFRE_{before}} * 100\%$$

where, $MAFRE_{after}$ and $MAFRE_{before}$ are the MAFRE after and before flow reconstruction. From figure 28, one can observe that improvement (%) in all the cases is very close to the best improvement possible except when $p_{calib} = 0$. For the test case when $p_{calib} = 1$, as all the sensors are error free and there is nothing to improve and hence, the corresponding plot is not included. Overall, the average improvement in the flow reconstruction is 75.81%, where the average best possible improvement in flow reconstruction is 81.89%.

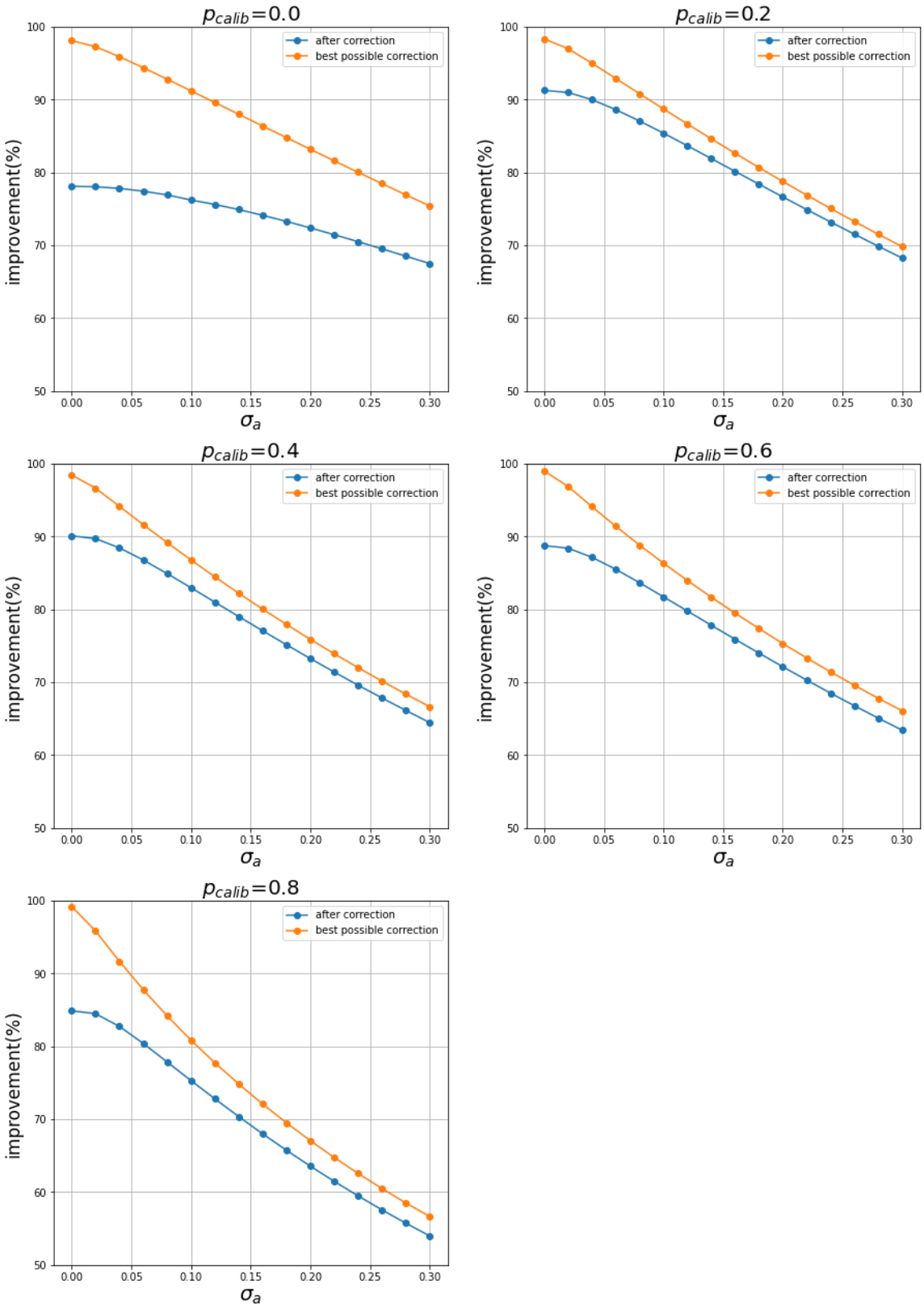


Figure 28. Improvement (%) after and before flow reconstruction

5.2. Error in predicted systematic error ratio

Next, we analyze the deviation in systematic error ratio prediction ($\hat{\mu} - \mu$) for different σ_a levels. Figure 29 depicts histograms of predicted and true systematic error ratios at different p_{calib} levels. Note that for the case $p_{calib} = 1$, the x axis has a very small domain. As we can observe, for all the p_{calib} values except 0 (edge case), the predicted and true systematic error ratio frequencies are comparable. Figure 30 represents boxplots for $\Delta = \hat{\mu} - \mu$ for different levels of σ_a and p_{calib}. As p_{calib} increases, the variance in Δ seems to decrease. Variance in Δ increases with increase in σ_a. Nonetheless, for all the values of σ_a and p_{calib} except 0, the median is approximately around 0, with interquartile range is around 0.1, which implies that the model is able to predict the systematic error ratios with a reasonable degree of accuracy.

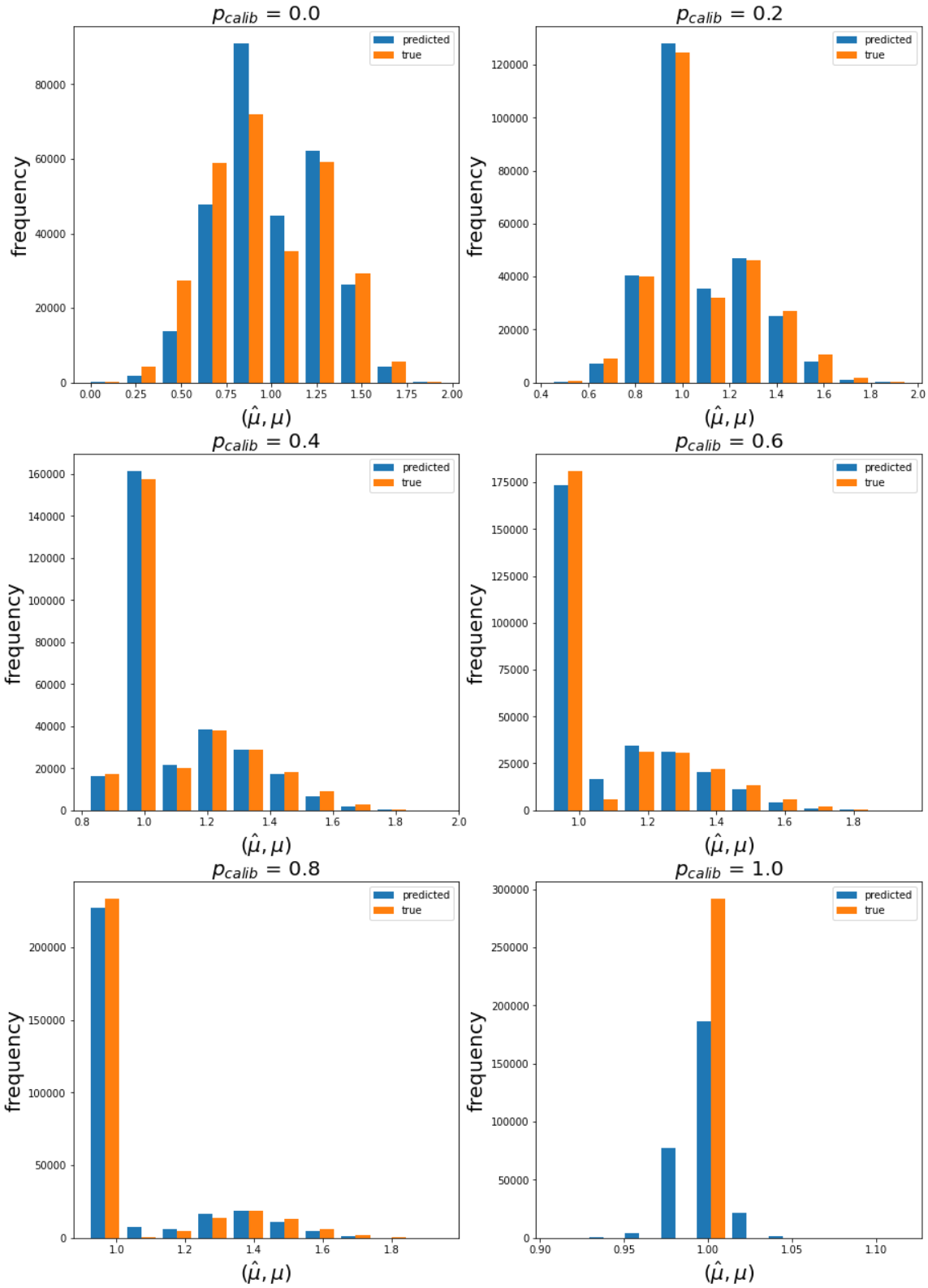


Figure 29. Histograms of predicted and true systematic error ratios

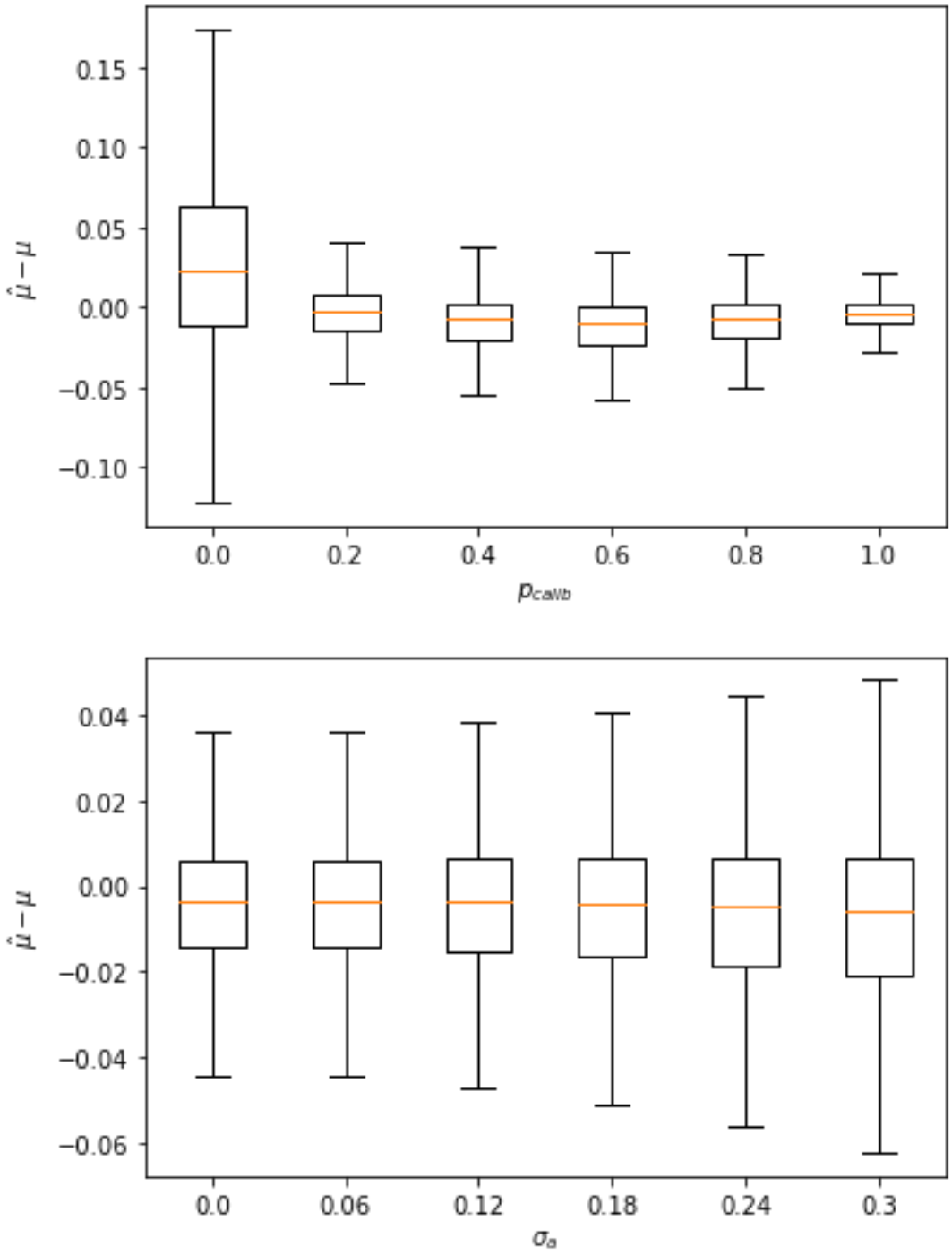
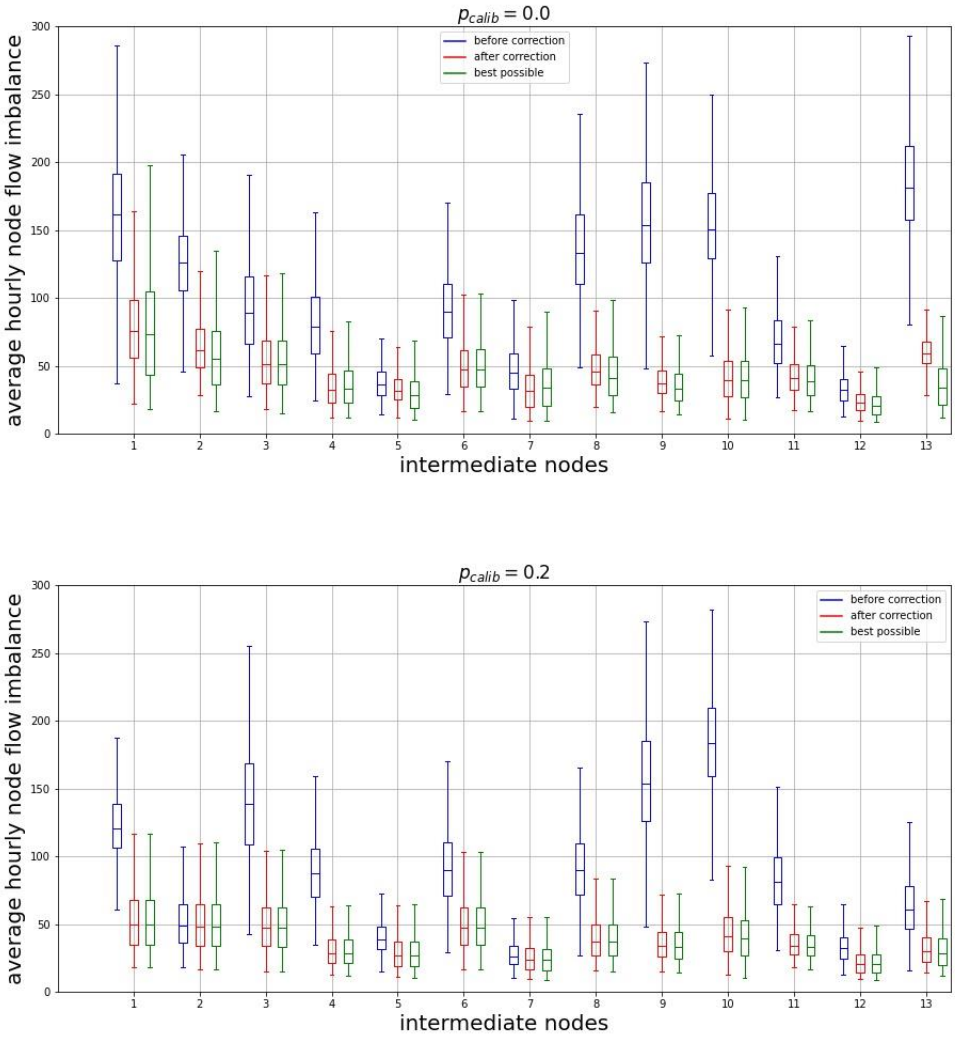
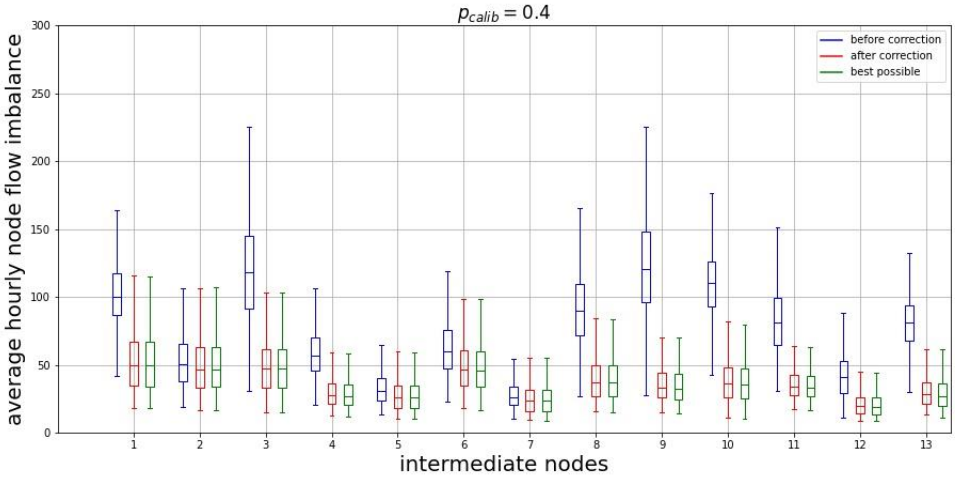


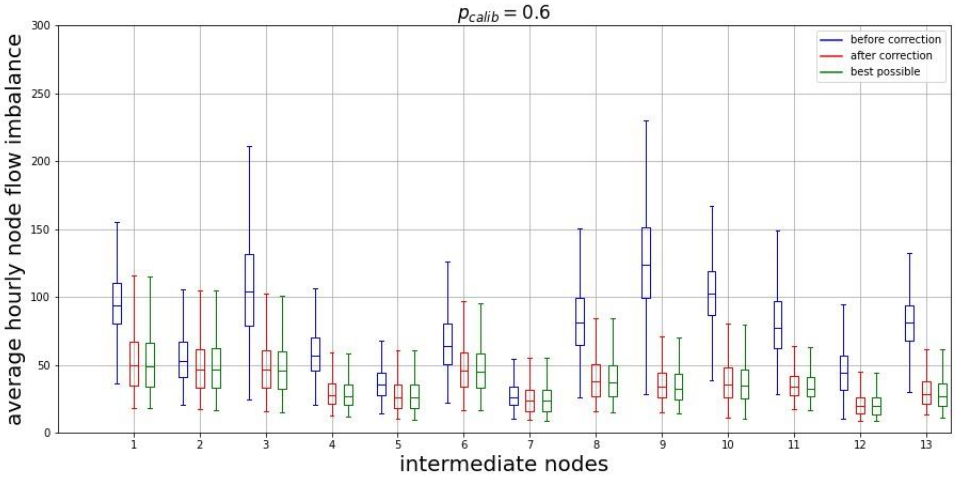
Figure 30. $(\hat{\mu}-\mu)$ for different levels of σ_a and p_{calib}

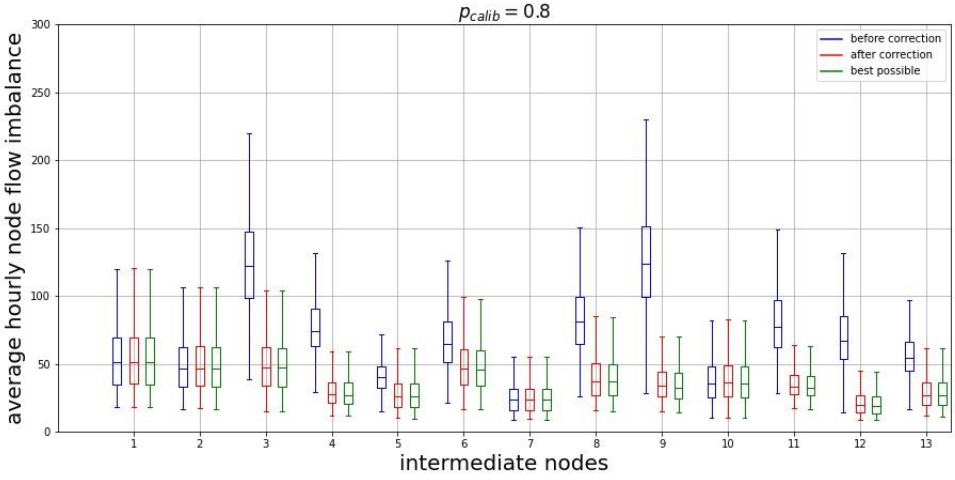
5.3. Flow imbalance in reconstructed flow

As mentioned in section 4.2, in real-world situation, dynamic flow can have flow imbalance due to the effect of shockwaves caused due to queuing and dequeuing between nodes. In this section, we analyze whether the reconstructed flow can minimize flow imbalance in the flow data. Figure 31 depicts hourly imbalance flow counts in the network at each of the 13 nodes before reconstructing and after reconstructing the erroneous flow, and the flow imbalance in the ideal case (perfect reconstruction). As one can observe, in all the cases, the reconstructed flow has the flow imbalance very close to the perfect reconstruction case.




$p_{calib} = 0.4$
before correction
after correction
best possible
average hourly node flow imbalance
intermediate nodes


$p_{calib} = 0.6$
before correction
after correction
best possible
average hourly node flow imbalance
intermediate nodes


$p_{calib} = 0.8$
before correction
after correction
best possible
average hourly node flow imbalance
intermediate nodes

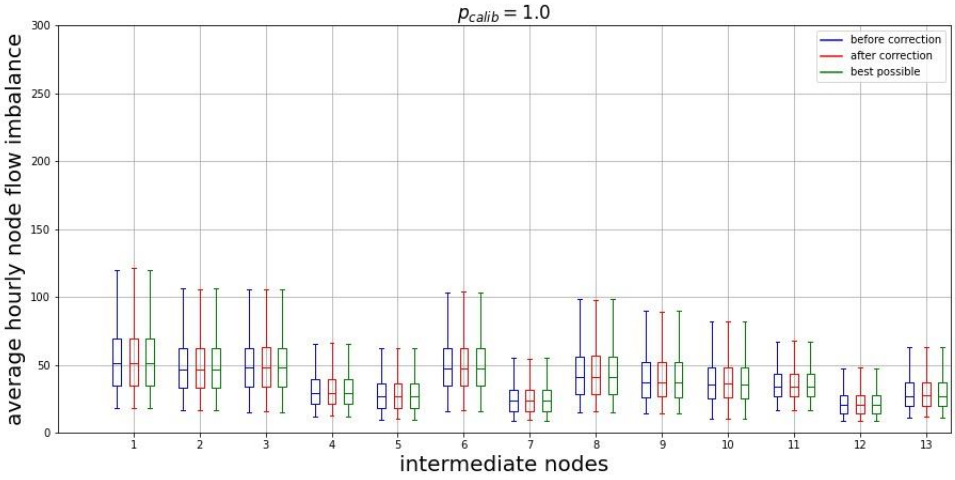
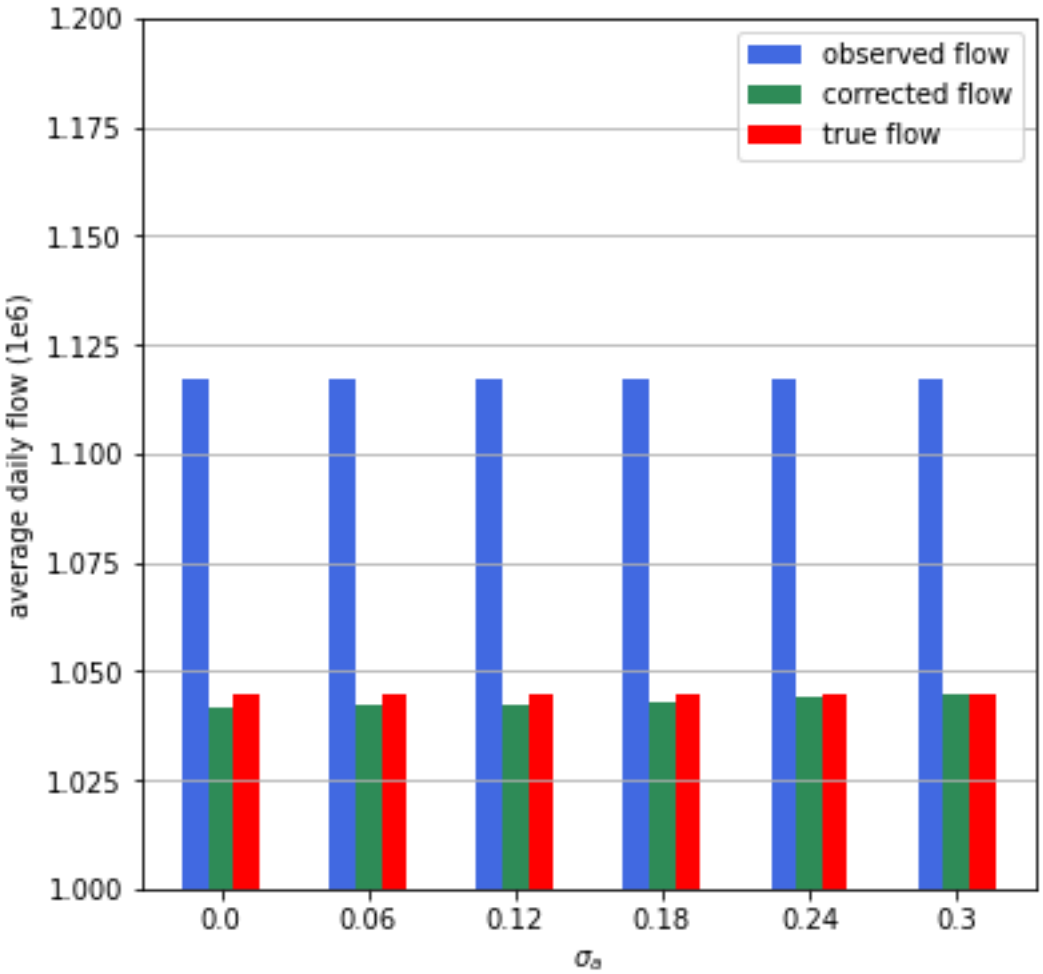


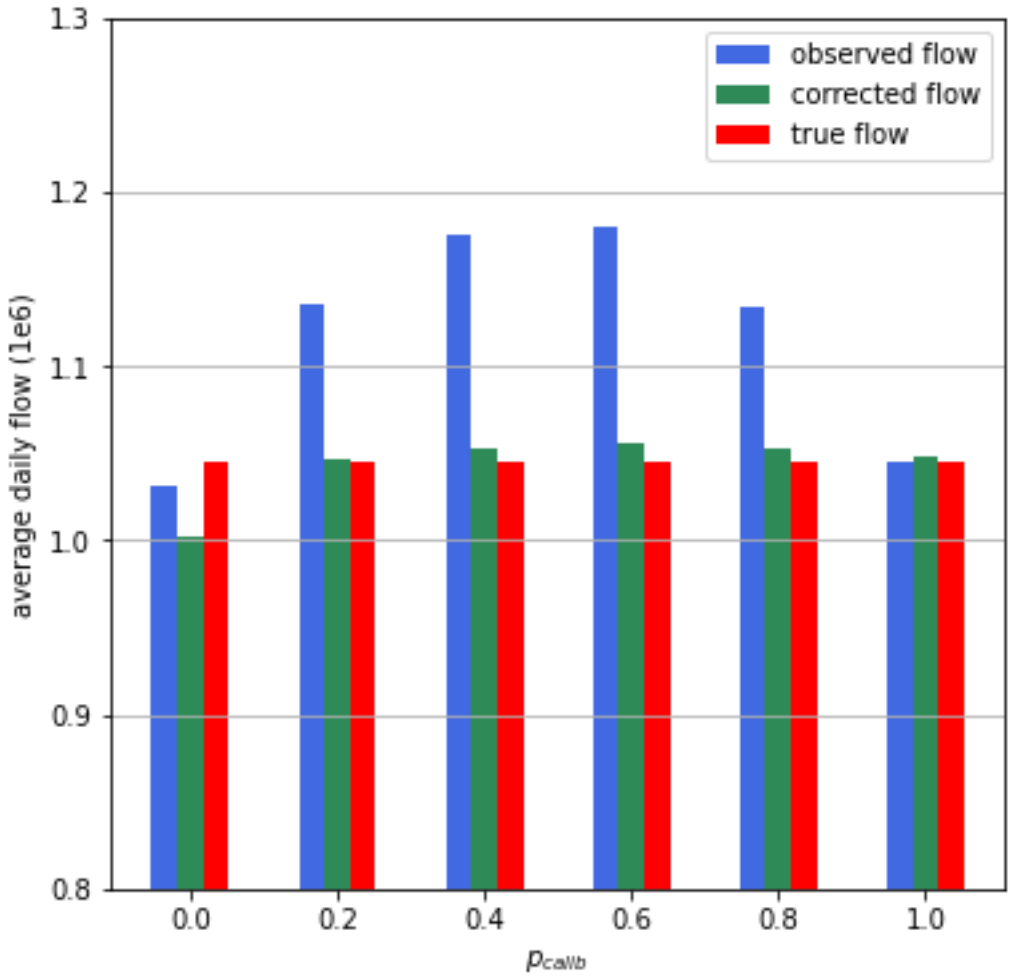
Figure 31. Flow imbalance in the network for various p_{calib} levels

5.4. Average daily flow in the network

One of the major advantages of this study is to reconstruct the erroneous flow that provides us the correct flow demand the network might experience. Once the actual flow demand is known, engineers can design infrastructure with appropriate capacity to enable a properly functioning network. Figure 32 depicts bar plots showing the average daily flow in the network and it could be observed that the corrected flow count in the network are close to the true flow count values.


observed flow
corrected flow
true flow
average daily flow (1e6)
1.200
1.175
1.150
1.125
1.100
1.075
1.050
1.025
1.000
0.0
0.06
0.12
0.18
0.24
0.3
σ_a

(a)



(b)

Figure 32. Average Daily Flow in the Network for various σ_a and p_{calib} levels

6. Discussion of Results, Conclusions and Future Work

We developed a methodology for estimating network sensor errors and reconstructing corrupted flow using deep learning. Deep learning model was able to capture the patterns in the dynamic flow without explicitly given any physical topological aspects. For training the model, the base flow was generated by fluctuating the OD demand over independent simulations, to which measurement errors were added to generate training data where a level of maximum random error coefficient (σ_a) and p_{calib} were maintained. To gauge the generalization ability, we tested the model on a variety of test cases generated by using combinations of different levels of σ_a and p_{calib} where the results showed that the tuned deep learning model can successfully quantify network sensor errors.

The model is able to achieve close to the best possible performance even for the flow data having σ_a greater than the maximum σ_a used for generating training data and is able to quantify sensor errors for all levels of sensor calibration in the network, despite keeping p_{calib} at 0.2 for all the training cases. For all the test cases, the improvement in MAFRE was comparable to the best possible improvement. This shows that the model is able to capture the temporal and spatial correlations in the data and predict the systematic error ratios irrespective of the level of randomness in the test data. Also, after flow reconstruction, the model could predict the total flow in the network with high precision, hence playing a vital role in prescribing the network capacity to the transportation planners for making infrastructure decisions.

Unlike most of the work in this area, the deep learning model does not depend on flow conservation in the network. Model is never given physical topological aspects of the network rather we let the model learn all the patterns from the data during the training phase. As we showed, the base flow itself has flow imbalance, and adding measurement errors exaggerates the imbalance.

Irrespective of this deviation from flow conservation at intermediate nodes, the model is able to reconstruct the flow, where the reconstructed flow imbalance is very close to the true flow imbalance.

Although the deep learning model successfully quantifies the systematic errors, compared to other test cases it struggles at the edge case when $p_{calib} = 0$. This brings to attention a very important concept of machine learning that the model is only as good as the data it is trained on and we need to manufacture the training data more thoughtfully. Incorporating more training cases with no calibrated sensors in this case for example will help boost the performance of the model for test cases when all the sensors suffer from systematic errors. By the same principle, the model can handle fluctuations in demand from time to time as the simulated flow was generated by varying OD demand over different independent simulations. Hence, the model is capable of estimating systematic error ratios with reasonable accuracy as long as the network topology is maintained.

One of the drawbacks of this study is that for a larger network, flow simulation would require heavy computational resources. Also, with a larger network, in order for the model to exploit the underlying spatial and temporal relationship, one would have to generate a larger quantity of data samples and training the deep learning model can further get computationally expensive. Although, cloud computing services like Amazon Web Services (AWS) provide dependable computing resources (GPUs) required for deep learning, the cost of these services could compile up quickly if used extravagantly. Another drawback of using deep learning is that minor changes in topology can render the model useless. The reason being the training data was created using a certain network topology and the trained model expects the same while testing. This drawback

could be dealt by designing the training cases intelligently by incorporating the edge cases and probable future network failure scenarios.

This brings us to the proposed future work:

1) Designing the training cases to account for edge cases and potential network failures: As discussed above, the training data should incorporate all the possible edge cases to perform well during the testing phase. More importantly, we encounter lane closures on a day-to-day basis for construction purposes. These closures would render the model useless if the model does not learn to deal with the modified network topology. Thus, the training cases should also account for potential network failure scenarios. This could be achieved by incorporating lane capacities in the training data. These capacities could simply be changed to 0 or a certain threshold to help model identify the perturbed network scenario.
2) Estimate random error ratios: As explained earlier, measurement errors have 2 components, systematic error and random error. Systematic error is what we focused in this study as it is the error that the system should not have intrinsically and can help identify sensors that need priority maintenance. But in order to reconstruct the flow, random error ratios are also important, as they add to the systematic errors to corrupt the traffic flow readings. Thus, identifying random error rations could help determine confidence bounds for the reconstructed traffic flow with higher accuracy.
3) Understanding how deep learning works: In machine learning there is usually a trade-off between the complexity of the model and its explain-ability. Given that neural networks are highly complex models, explain-ability of deep learning models is still an open area for research. Various researchers have proposed visualizations of hidden layer activations for computer vision applications, but this is yet to be done for traffic data effectively.

7. References

1) Coifman, Benjamin. "Vehicle level evaluation of loop detectors and the remote traffic microwave sensor." *Journal of transportation engineering* 132, no. 3 (2006): 213-226.
2) Rajagopal, Ram, and Pravin Pratap Varaiya. Health of California's loop detector system. California PATH Program, Institute of Transportation Studies, University of California at Berkeley, 2007.
3) Turochy, Rod E., and Brian L. Smith. "New procedure for detector data screening in traffic management systems." Transportation Research Record 1727, no. 1 (2000): 127-131.
4) Vanajakshi, Lelitha, and L. R. Rilett. "Loop detector data diagnostics based on conservation-of-vehicles principle." Transportation research record 1870, no. 1 (2004): 162-169.
5) Sun, Zhe, Wen-Long Jin, and ManWo Ng. "Network sensor health problem." Transportation Research Part C: Emerging Technologies 68 (2016): 300-310.
6) Chen, Chao, Jaimyoung Kwon, John Rice, Alexander Skabardonis, and Pravin Varaiya. "Detecting errors and imputing missing data for single-loop surveillance systems." Transportation Research Record 1855, no. 1 (2003): 160-167
7) Duan, Yanjie, Yisheng Lv, Yu-Liang Liu, and Fei-Yue Wang. "An efficient realization of deep learning for traffic data imputation." Transportation research part C: emerging technologies 72 (2016): 168-181.
8) Li, Yuebiao, Zhiheng Li, and Li Li. "Missing traffic data: comparison of imputation methods." IET Intelligent Transport Systems 8, no. 1 (2014): 51-57.
9) Nihan, Nancy L. "Aid to determining freeway metering rates and detecting loop errors." Journal of Transportation Engineering 123, no. 6 (1997): 454-458.

10) Hu, Ping, R. Goeltz, and R. Schmoyer. Proof of Concept of ITS as An Alternative Data Resource: A Demonstration Project of Florida and New York Data. No. ORNL/TM-2001/247, 2001.

11) Turner, Shawn, Rich Margiotta, Tim Lomax, and Cambridge Systematics. Monitoring urban freeways in 2003: current conditions and trends from archived operations data. No. FWHA-HOP-05-018. Texas Transportation Institute, 2004.

12) Kwon, Jaimyoung, Chao Chen, and Pravin Varaiya. "Statistical methods for detecting spatial configuration errors in traffic surveillance sensors." Transportation research record 1870, no. 1 (2004): 124-132.

13) Allison, Paul D. Missing data. Vol. 136. Sage publications, 2001.

14) Liu, Zhaobin, Satish Sharma, and Sandeep Datla. "Imputation of missing traffic data during holiday periods." Transportation Planning and Technology 31, no. 5 (2008): 525-544.

15) Qu, Li, Li Li, Yi Zhang, and Jianming Hu. "PPCA-based missing data imputation for traffic flow volume: A systematical approach." IEEE Transactions on intelligent transportation systems 10, no. 3 (2009): 512-522.

16) J. Farhan, T. Fwa Airport pavement missing data management and imputation with stochastic multiple imputation model Transp. Res. Rec.: J. Transp. Res. Board (2013), pp. 43-54

17) Lv, Yisheng, Yanjie Duan, Wenwen Kang, Zhengxi Li, and Fei-Yue Wang. "Traffic flow prediction with big data: a deep learning approach." IEEE Transactions on Intelligent Transportation Systems 16, no. 2 (2014): 865-873.

18) Jia, Yuhan, Jianping Wu, and Ming Xu. "Traffic flow prediction with rainfall impact using a deep learning method." Journal of advanced transportation 2017 (2017).

19) Ma, Xiaolei, Zhuang Dai, Zhengbing He, Jihui Ma, Yong Wang, and Yunpeng Wang. "Learning traffic as images: a deep convolutional neural network for large-scale transportation network speed prediction." Sensors 17, no. 4 (2017): 818.

20) Grigorescu, Sorin, Bogdan Trasnea, Tiberiu Cocias, and Gigel Macesanu. "A survey of deep learning techniques for autonomous driving." Journal of Field Robotics 37, no. 3 (2020): 362-386.

21) Nguyen, Hoang, Le-Minh Kieu, Tao Wen, and Chen Cai. "Deep learning methods in transportation domain: a review." IET Intelligent Transport Systems 12, no. 9 (2018): 998-1004.

22) Ma, Xiaolei, Haiyang Yu, Yunpeng Wang, and Yinhai Wang. "Large-scale transportation network congestion evolution prediction using deep learning theory." PloS one 10, no. 3 (2015).

23) Fouladgar, Mohammadhani, Mostafa Parchami, Ramez Elmasri, and Amir Ghaderi. "Scalable deep traffic flow neural networks for urban traffic congestion prediction." In 2017 International Joint Conference on Neural Networks (IJCNN), pp. 2251-2258. IEEE, 2017.

24) Zhao, Zheng, Weihai Chen, Xingming Wu, Peter CY Chen, and Jingmeng Liu. "LSTM network: a deep learning approach for short-term traffic forecast." IET Intelligent Transport Systems 11, no. 2 (2017): 68-75.

25) Genders, Wade, and Saiedeh Razavi. "Using a deep reinforcement learning agent for traffic signal control." arXiv preprint arXiv:1611.01142 (2016).

26) van der Pol, Elise. "Deep reinforcement learning for coordination in traffic light control." Master's thesis, University of Amsterdam (2016).

27) Li, Li, Yisheng Lv, and Fei-Yue Wang. "Traffic signal timing via deep reinforcement learning." IEEE/CAA Journal of Automatica Sinica 3, no. 3 (2016): 247-254.

28) Liang, Xiaoyuan, and Guiling Wang. "A convolutional neural network for transportation mode detection based on smartphone platform." In 2017 IEEE 14th international conference on mobile Ad Hoc and sensor systems (MASS), pp. 338-342. IEEE, 2017.

29) Adu-Gyamfi, Yaw Okyere, Sampson Kwasi Asare, Anuj Sharma, and Tienaah Titus. "Automated vehicle recognition with deep convolutional neural networks." Transportation Research Record 2645, no. 1 (2017): 113-122.

30) Ke, Jintao, Hongyu Zheng, Hai Yang, and Xiqun Michael Chen. "Short-term forecasting of passenger demand under on-demand ride services: A spatio-temporal deep learning approach." Transportation Research Part C: Emerging Technologies 85 (2017): 591-608.

31) Yao, Huaxiu, Fei Wu, Jintao Ke, Xianfeng Tang, Yitian Jia, Siyu Lu, Pinghua Gong, Jieping Ye, and Zhenhui Li. "Deep multi-view spatial-temporal network for taxi demand prediction." In Thirty-Second AAAI Conference on Artificial Intelligence. 2018.

32) Xu, Jun, Rouhollah Rahmatizadeh, Ladislau Bölöni, and Damla Turgut. "Real-time prediction of taxi demand using recurrent neural networks." IEEE Transactions on Intelligent Transportation Systems 19, no. 8 (2017): 2572-2581.

33) Liu, Lijuan, and Rung-Ching Chen. "A MRT daily passenger flow prediction model with different combinations of influential factors." In 2017 31st International Conference on Advanced Information Networking and Applications Workshops (WAINA), pp. 601-605. IEEE, 2017.

34) Baek and, Junghan, and Keemin Sohn. "Deep-learning architectures to forecast bus ridership at the stop and stop-to-stop levels for dense and crowded bus networks." Applied Artificial Intelligence 30, no. 9 (2016): 861-885.

35) Dwivedi, Kartik, Kumar Biswaranjan, and Amit Sethi. "Drowsy driver detection using representation learning." In 2014 IEEE international advance computing conference (IACC), pp. 995-999. IEEE, 2014.

36) Kiranyaz, Serkan, Onur Avci, Osama Abdeljaber, Turker Ince, Moncef Gabbouj, and Daniel J. Inman. "1D convolutional neural networks and applications: A survey." arXiv preprint arXiv:1905.03554 (2019).

37) Srivastava, Nitish, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. "Dropout: a simple way to prevent neural networks from overfitting." The journal of machine learning research 15, no. 1 (2014): 1929-1958.

38) Li, Hao, Zheng Xu, Gavin Taylor, Christoph Studer, and Tom Goldstein. "Visualizing the loss landscape of neural nets." In Advances in Neural Information Processing Systems, pp. 6389-6399. 2018.

39) Kiranyaz, Serkan, Turker Ince, and Moncef Gabbouj. Multidimensional particle swarm optimization for machine learning and pattern recognition. NewYork: Springer, 2014.

40) Wu, Yuankai, Huachun Tan, Lingqiao Qin, Bin Ran, and Zhuxi Jiang. "A hybrid deep learning based traffic flow prediction method and its understanding." Transportation Research Part C: Emerging Technologies 90 (2018): 166-180.

41) Yu, Rose, Yaguang Li, Cyrus Shahabi, Ugur Demiryurek, and Yan Liu. "Deep learning: A generic approach for extreme condition traffic forecasting." In Proceedings of the 2017 SIAM

international Conference on Data Mining, pp. 777-785. Society for Industrial and Applied Mathematics, 2017.

42) Li, JiaWen, and JingSheng Wang. "Short term traffic flow prediction based on deep learning." In CICTP 2019, pp. 2457-2469. 2017.