Forecasting Air Pollutants using Deep Learning

Masters Thesis

submitted to
Indian Institute of Science Education and Research Pune
in partial fulfillment of the requirements for the
BS-MS Dual Degree Programme

Submitted by

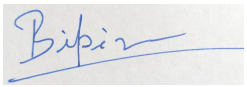
Sushrut Ishwar Gaikwad
20171197

Indian Institute of Science Education and Research, Pune

Supervisor: Dr. Bipin Kumar (IITM Pune)
TAC Expert: Dr. Amit Apte (IISER Pune)
Co-Supervisior: Dr. Manmeet Singh (IITM Pune)

Certificate

This is to certify that this dissertation entitled **Forecasting Air Pollutants using Deep Learning** towards the partial fulfillment of the BS-MS dual degree programme at the Indian Institute of Science Education and Research, Pune, represents study/work carried out by Sushrut Ishwar Gaikwad at Indian Institute of Tropical Meteorology, Pune under the supervision of Dr. Bipin Kumar during the academic year 2021-2022.



Dr. Bipin Kumar
Scientist E, HPCS
IITM Pune

Committee:

Supervisor: Dr. Bipin Kumar
TAC Expert: Dr. Amit Apte
Co-Supervisor: Dr. Manmeet Singh

I dedicate this thesis to my family

Declaration

I hereby declare that the matter embodied in the report entitled **Forecasting Air Pollutants using Deep Learning** are the results of the work carried out by me at the Indian Institute of Tropical Meteorology, Pune, under the supervision of Dr. Bipin Kumar, and the same has not been submitted elsewhere for any other degree.

Sushrut Ishwar Gaikwad

Acknowledgments

I am grateful to my supervisor Dr. Bipin Kumar for allowing me to work under his supervision at IITM Pune. I thank him for his motivation and guidance.

I am grateful to Dr. Sachin Ghude, Dr. Gaurav Govardhan, and their team for their invaluable inputs to this study. I thank my TAC expert Dr. Amit Apte for his valuable suggestions and comments during my mid-year evaluation. I thank my friends Shyam Krishna P for his input at the beginning of this study, Rishabh Dev for his guidance at the beginning of my machine learning journey, and Kaustubh Atey for his suggestions on the model implementation.

I express my heartfelt gratitude towards my parents for their sacrifices, hard work, and dedication toward me. I owe every bit of my achievement to them.

I wish to thank my friends Shashank for being a great roommate and Samyuktha for her constant support during various phases of my life at IISER.

Last but not least, I express my gratitude to Dr. Andrew Ng for his open-source lectures on machine learning and deep learning and to all the machine learning researchers for their generous and invaluable open-source contributions.

Abstract

Fire incidences have recently increased due to climate change and other human-induced factors. Due to incidents such as stubble burning in Punjab-Haryana, forest fires in various parts of India, like the north-east and central India, lead to dangerously high levels of particulate matter of aerodynamic diameter smaller than 2.5 microns (PM2.5). Timely forecasts of PM2.5 can help prevent air quality-related public health issues, plan ahead of time, and implement temporary control measures. However, most operational air quality forecasting systems worldwide have to employ a persistent assumption for representing fire emissions due to a lack of information about the future evolution of fires. Under the persistence assumption, we assume near-real-time fire emissions are constant for the entire forecast cycle, which can lead to significant errors in air quality forecasts if the fire emissions change significantly daily. We aim to fill this gap by forecasting fire emissions, like PM2.5, for the next 2-3 days using spatiotemporal deep learning models such as the convolutional long short-term memory (ConvLSTM). Using this approach, we can get a reliable correlation coefficient. We attempt to improve further by adding variables like normalized difference vegetation index (NDVI), relative humidity, temperature, surface pressure, and total cloud cover during model training.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Since the industrial revolution, particularly in the last several decades, air pollution has been on a steep rise. Asian countries like Bangladesh, India, and China frequently lead the list of most polluted countries on the planet. Unfortunately, India has one of the world's worst air quality indexes (AQI). Indians have frequent exposure to unhealthy levels of ambient PM2.5 (particles with a size smaller than 2.5 microns, which is about one-thirtieth the width of a human hair), the most harmful pollutant - emanating from multiple sources. Around 93% of Indians live in areas where PM2.5 levels are worse than the World Health Organization's (WHO) least stringent norms, reducing the average life expectancy in the country by 1.5 years (more than cancer), according to the reports of the US-based Health Effects Institute. It claims that poor air quality accounts for at least one in nine global deaths. PM2.5 particles, owing to their small size, can readily travel through the respiratory tract and cause deadly diseases such as lung cancer, stroke, and heart disease. An estimate is that unhealthy levels of PM2.5 caused 1.7 million premature deaths in India in 2019, which is a huge cost to bear for the Indian economy.

There are numerous sources of PM2.5. The most common is gasoline, oil, diesel fuel, or wood combustion. In the northwest, particularly in the Delhi-Punjab-Haryana region, stubble burning is one of the significant factors producing PM2.5 emissions and, subsequently, poor air quality. Stubble burning is intentionally setting fire to the straw stubble after the harvest. They do this to clear the land for a new sowing season. New Delhi frequently slips into the critical air quality zone in stubble burning season every year. From mid-September to November each year, farmers, mainly in the Punjab-Haryana

region, burn an estimated 35 million tonnes of crop waste. The smoke created from this process is so huge that it is visible from space, resulting in declaring air pollution emergencies in New Delhi and nearby areas every year. Forest fires also led to significant emissions of PM2.5, particularly in summer. Due to changing climate and frequent heat waves, forest fire incidents have also risen significantly. They mainly occur in drier Indian states like Madhya Pradesh, Odisha, Chhattisgarh, central-eastern Maharashtra, and in the northeastern Indian subcontinent that includes Assam, Meghalaya, Tripura, Nagaland, Manipur, Mizoram, and countries like Myanmar.

So, analyzing and forecasting PM2.5 values can be crucial in preventing air quality-related emergencies and preplanning and implementing temporary but crucial measures to avoid further damage. PM2.5 particles are light, making them get carried away by the wind. When it comes to forest fire, wind, available vegetation, temperature, precipitation, humidity, and various other variables hugely affect how it would propagate. Human behavior plays a role in fires caused due to stubble burning, along with the previously mentioned factors. So, PM2.5 values have a very complex interaction with these factors on many different temporal and spatial scales. The traditional approach to forecasting using fluid dynamical equations and solving higher-order non-linear differential equations would require an impractically large amount of computational resources. Incorporating the effect of various variables would further increase the complexity. A machine learning approach to forecasting could yield excellent results without using enormous computational resources, as seen in various other fields. The availability of massive meteorological data is another reason promising us the same. Deep learning, a machine learning subfield, is hugely data-hungry and can learn complex non-linear mappings between input and output. Deep learning also does not require hand-designing features (done in machine learning algorithms) to incorporate spatiotemporal variation since it learns them independently. We use deep learning-based models to learn spatiotemporal variation in PM2.5 values and forecast for three days into the future.

1.1 Literature Review

Various methods are developed and applied for time series analysis and forecasting, broadly classified into three categories, whose discussion is as follows.

1.1.1 Statistical Methods

Some of the most common statistical methods for time series analysis and forecasting are listed below:

- Exponential Smoothing (ETS),
- Holt-Winters,
- ARMA (AutoRegressive Moving Average),
- ARIMA (AutoRegressive Integrated Moving Average),
- SARIMA (Seasonal ARIMA),
- SARIMAX (SARIMA with eXogenous variable),
- VARMA (Vector ARMA).

These methods are designed specifically for time series analysis and forecasting. One of their most significant advantages is high interpretability. They give us coefficients that indicate the trend and seasonality of the data, which helps us gain a deeper insight into it and is fruitful in guiding us towards approaching the forecasting problem.

1.1.2 Machine Learning Methods

General machine learning algorithms for regression can be applied for time series forecasting. The only extra step needed is to split the data accordingly, also known as forming the supervised splits of the data. Some standard machine learning regression algorithms are listed below:

- Linear Regression,
- Support Vector Machine,
- Random Forest,
- Adaptive Boosting (AdaBoost),
- Extreme Gradient Boosting (XGBoost).

Although these algorithms are not explicit time series forecasting algorithms, they can give a decent forecast after conducting various statistical tests and transformations on the data and forming supervised splits before using them.

1.1.3 Deep Learning Methods

Recent advancement in deep learning has generated much hype. Details regarding deep learning are discussed in chapter 2. Some standard deep learning algorithms are listed below:

- Feedforward Neural Network,
- Convolutional Neural Network (CNN),
- Recurrent Neural Network (RNN),
- Gated Recurrent Unit (GRU) Network,
- Long Short-Term Memory (LSTM) Network.

A feedforward neural network can be used for general regression tasks, whereas CNN is designed to work with images. Again, though these are not explicit time series forecasting algorithms, they can give a decent forecast after carrying out the exact steps discussed earlier in the machine learning section.

RNN is designed to work with sequence data. Due to its structure, it allows differing amounts of input into it. Hence, it is one of the ideal choices for working with time series data. However, RNN is prone to the problem of vanishing and exploding gradients which makes learning long-term dependencies very difficult. LSTM (Hochreiter & Schmidhuber, 1997) and GRU (Cho et al., 2014) networks were developed to address this issue. They have unique functions known as gates that help them "remember" the long-term dependencies in the data, which can be helpful in forecasting, and "forget" information that is not useful. Hence, GRU and LSTM networks are widely used today for analyzing and forecasting sequence data.

In literature, these methods have been applied for PM2.5 forecasting. (Huang & Kuo, 2018) proposed a deep CNN-LSTM model for PM2.5 forecasting, (Harishkumar, Yogesh, Gad, et al., 2020) used various machine learning regression models for forecasting PM2.5, whereas (Karimian et al., 2019) evaluated various machine learning approaches for the same.

The collective drawback of all these methods for our task is that they, by default, do not work with spatiotemporal sequences. As mentioned earlier, using these models would require us to hand-design features and feed them to the model to incorporate spatiotemporal variation. We use deep learning-based spatiotemporal forecasting models to address this issue.

Spatiotemporal Forecasting Methods

Deep learning-based spatiotemporal forecasting methods are mainly used for precipitation nowcasting, i.e., short-term precipitation forecasting. Convolutional LSTM or ConvLSTM (Shi et al., 2015) is one of the most fundamental yet powerful methods for the same. Details regarding ConvLSTM are discussed in section 2.2.1. We propose using ConvLSTM to forecast PM2.5 emissions.

1.2 Problem Statement

Suppose we have a spatial region represented by an $M \times N$ grid (or a matrix). Inside each grid cell, P variables can be measured at a particular time. So, the observation at any time can be represented by a tensor $\mathcal{X} \in \mathbf{R}^{P \times M \times N}$, where $\mathbf{R}$ is the domain of the observed variables. Let $\widehat{\mathcal{X}}_1$, $\widehat{\mathcal{X}}_2$, $\ldots$, $\widehat{\mathcal{X}}_t$, represent a sequence of observations. The spatiotemporal sequence forecasting problem is to predict the most likely length-K sequence in the future given the previous J observations, including the current one:

$$\widetilde{\mathcal{X}}_{t+1}, \ldots, \widetilde{\mathcal{X}}_{t+K} = \underset{\mathcal{X}_{t+1}, \ldots, \mathcal{X}_{t+K}}{\arg\max}\ p\left(\mathcal{X}_{t+1}, \ldots, \mathcal{X}_{t+K} \,\middle|\, \widehat{\mathcal{X}}_{t-J+1}, \widehat{\mathcal{X}}_{t-J+2}, \ldots, \widehat{\mathcal{X}}_t\right) \tag{1.2.1}$$

The formulation is same as that given in (Shi et al., 2015). The following bullet points summarize this formulation:

- We have a 3-D array[1] corresponding to each observation, i.e., a 3-D array for each time step.
- We want to predict the most likely sequence of K such arrays given the past sequence of J arrays.

[1] We can think of this as a matrix with P values in each position since there are P variables.

Chapter 2

Theoretical Background

2.1 Neural Networks and Deep Learning

Deep learning, a subfield of machine learning, has recently gained much traction due to its wide range of applications ranging from image analysis, object detection, and self-driving cars to machine translation, sentiment analysis, forecasting, and finance. It tries to mimic the human brain. The most fundamental element of any deep learning model is known as an *artificial neuron* (figure 2.1.2). It is supposed to be a highly simplified version of the biological neuron (figure 2.1.1).

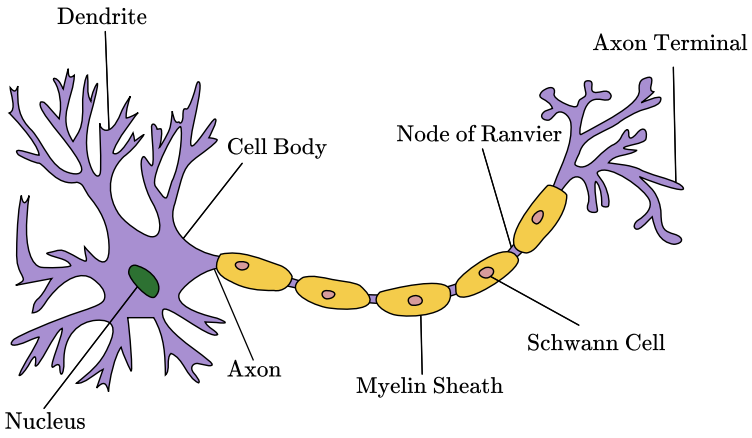


Figure 2.1.1: A biological neuron.

Biological neurons take input electrical signals through dendrites and out-

put them through the axon. The output signals depend upon the input.

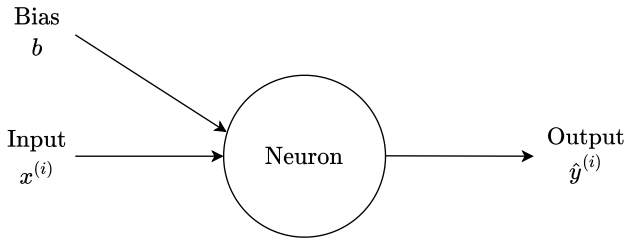


Figure 2.1.2: An artificial neuron.

2.1.1 Working of an Artificial Neuron

Say, we have m examples for training our artificial neuron. We can represent them as $\left\{\left(x^{(1)}, y^{(1)}\right), \left(x^{(2)}, y^{(2)}\right), \ldots, \left(x^{(m)}, y^{(m)}\right)\right\}$, where $\left(x^{(i)}, y^{(i)}\right)$ stands for $(\text{feature}, \text{value})$ of the ith example. We represent the predicted value from the neuron for the feature of the ith example as $\hat{y}^{(i)}$ (also known as *activation*, denoted as $a^{(i)}$). An artificial neuron combines the input $x^{(i)}$ with some *weight* w and a *bias* b, applies some function g, and outputs $\hat{y}^{(i)}$. Mathematically, this is nothing but the following equations.

$$z^{(i)} = wx^{(i)} + b \tag{2.1.1a}$$

$$a^{(i)} = \hat{y}^{(i)} = g\left(z^{(i)}\right) \tag{2.1.1b}$$

This is only for one particular example i consisting of one feature $x^{(i)}$, and one value $y^{(i)}$. Here, g is some (mostly a non-linear) function known as *activation function*[1]. Weights and biases are parameters that a neural network learns[2] during training.

If examples consist of multiple features, say n_x, then equations 2.1.1 take the following form (see figure 2.1.3).

$$z^{(i)} = \mathbf{w}^\intercal \mathbf{x}^{(i)} + b \tag{2.1.2a}$$

$$a^{(i)} = \hat{y}^{(i)} = g\left(z^{(i)}\right) \tag{2.1.2b}$$

Here, $\mathbf{w} \in \mathbb{R}^{n_x}$ is a column vector consisting of weights for all the features, $\mathbf{x}^{(i)} \in \mathbb{R}^{n_x}$ is a column vector consisting of features for the ith example.

[1] Activation functions are discussed in section 2.1.3.

[2] Details on how a neural network learns are discussed in section 2.1.5.

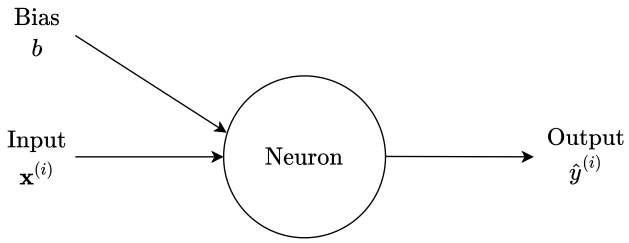


Figure 2.1.3: An artificial neuron for examples with multiple features.

We can write the column vectors $\mathbf{w}$ and $\mathbf{x}^{(i)}$ as

$$\mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_{n_x} \end{bmatrix} \quad \text{and} \quad \mathbf{x}^{(i)} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n_x} \end{bmatrix}^{(i)} \tag{2.1.3}$$

In equation 2.1.2, "$\mathbf{w}^\intercal\mathbf{x}^{(i)}$" represents matrix multiplication between the row vector $\mathbf{w}^\intercal$ (transpose of the column vector $\mathbf{w}$) and the column vector $\mathbf{x}^{(i)}$, and $\hat{y}^{(i)}$ is the value[3] predicted by the neuron. Figure 2.1.4 schematically represents equations 2.1.2.

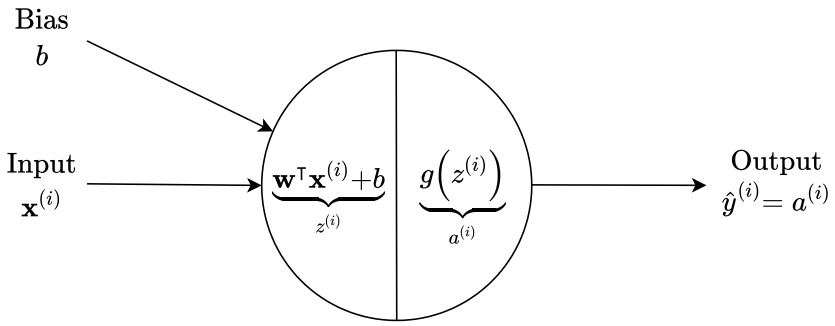


Figure 2.1.4: Schematic representation of equations 2.1.2.

The input vector of features and the predicted values can be represented in a matrix simultaneously for all the m examples in the following way.

[3]This is just a single value and not a vector since one neuron outputs only one value.

$$\underset{n_x \times m}{\mathbf{X}} = \begin{bmatrix} | & | & \cdots & | \\ \mathbf{x}^{(1)} & \mathbf{x}^{(2)} & \cdots & \mathbf{x}^{(m)} \\ | & | & \cdots & | \end{bmatrix} \tag{2.1.4a}$$

$$\underset{1 \times m}{\widehat{\mathbf{Y}}} = \begin{bmatrix} \hat{y}^{(1)} & \hat{y}^{(2)} & \cdots & \hat{y}^{(m)} \end{bmatrix} \tag{2.1.4b}$$

Using this matrix notation, we can write a single vectorized form of equation 2.1.2a for all the m examples, which is shown below.

$$\begin{bmatrix} z^{(1)} & z^{(2)} & \cdots & z^{(m)} \end{bmatrix} = \begin{bmatrix} w_1 & w_2 & \cdots & w_{n_x} \end{bmatrix} \begin{bmatrix} | & | & \cdots & | \\ \mathbf{x}^{(1)} & \mathbf{x}^{(2)} & \cdots & \mathbf{x}^{(m)} \\ | & | & \cdots & | \end{bmatrix} + \underbrace{\begin{bmatrix} b & \cdots & b \end{bmatrix}}_{m \text{ times}}$$

The above equation in matrix notation is the following.

$$\underset{1 \times m}{\mathbf{Z}} = \underset{1 \times n_x}{\mathbf{w}^\intercal} \underset{n_x \times m}{\mathbf{X}} + \underset{1 \times m}{\mathbf{b}}$$

$$\therefore \begin{bmatrix} z^{(1)} & z^{(2)} & \cdots & z^{(m)} \end{bmatrix} = \begin{bmatrix} \mathbf{w}^\intercal \mathbf{x}^{(1)} + b & \mathbf{w}^\intercal \mathbf{x}^{(2)} + b & \cdots & \mathbf{w}^\intercal \mathbf{x}^{(m)} + b \end{bmatrix}$$

Similarly, we can write equation 2.1.2b as

$$\underset{1 \times m}{\mathbf{A}} = \begin{bmatrix} a^{(1)} & a^{(2)} & \cdots & a^{(m)} \end{bmatrix} = \underset{1 \times m}{\widehat{\mathbf{Y}}} = \begin{bmatrix} \hat{y}^{(1)} & \hat{y}^{(2)} & \cdots & \hat{y}^{(m)} \end{bmatrix} = \underset{1 \times m}{g\left(\mathbf{Z}\right)}$$

$$\therefore \underset{1 \times m}{\mathbf{A}} = \underset{1 \times m}{\widehat{\mathbf{Y}}} = \begin{bmatrix} g\left(z^{(1)}\right) & g\left(z^{(2)}\right) & \cdots & g\left(z^{(m)}\right) \end{bmatrix}$$

Here, the activation function g is applied to the matrix $\mathbf{Z}$ element wise, as shown. So, the vectorized version of equations 2.1.2, which includes all the m examples simultaneously is the following.

$$\underset{1 \times m}{\mathbf{Z}} = \underset{1 \times n_x}{\mathbf{w}^\intercal} \underset{n_x \times m}{\mathbf{X}} + \underset{1 \times m}{\mathbf{b}} \tag{2.1.5a}$$

$$\underset{1 \times m}{\mathbf{A}} = \underset{1 \times m}{\widehat{\mathbf{Y}}} = \underset{1 \times m}{g\left(\mathbf{Z}\right)} \tag{2.1.5b}$$

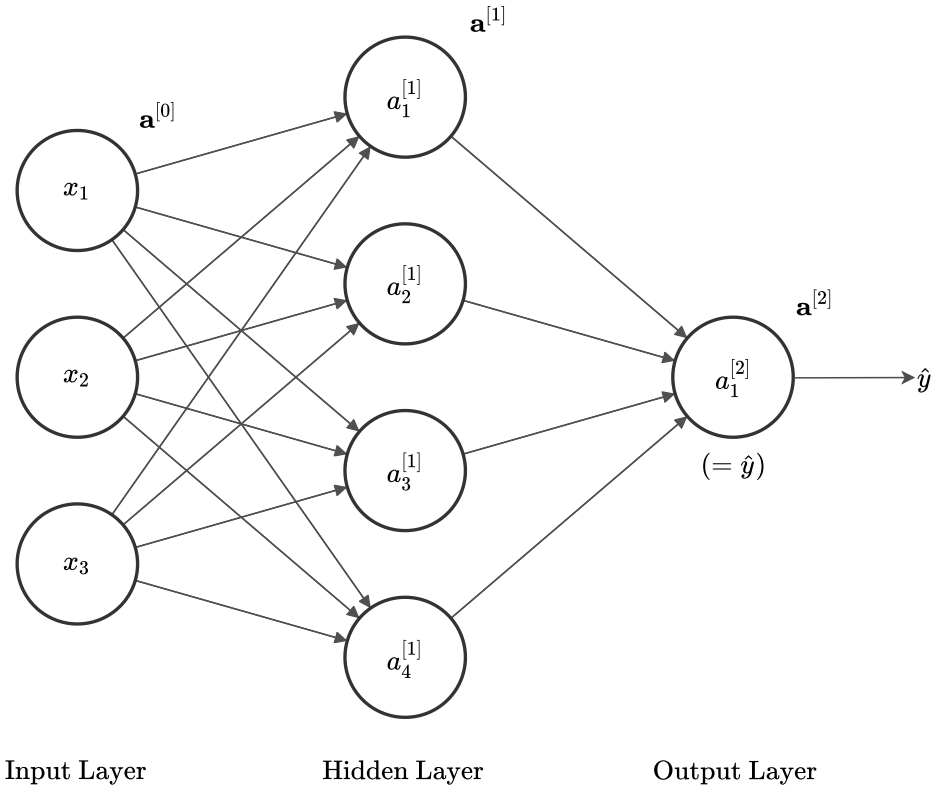


Figure 2.1.5: A neural network with one hidden layer[4].

2.1.2 Feedforward Neural Networks

It is common to consider neural networks consisting of multiple *layers* of neurons. These are known as *feedforward neural networks* or *fully connected neural networks*. In such networks, the output (or the predicted value) of each neuron in some lth layer is fed as input to each neuron in the $(l+1)$th layer. Figure 2.1.5 illustrates a neural network with one hidden layer.

[4]Biases are not represented for simplicity.

Notation

- $a_i^{[l]}$ stands for the activation of the ith neuron in the lth layer,
- $\mathbf{w}_i^{[l]}$ stands for the weights vector of the ith neuron in the lth layer,
- $b_i^{[l]}$ stands for the bias of the ith neuron in the lth layer,
- Activations in the lth layer can be collectively viewed as a column vector denoted by $\mathbf{a}^{[l]}$,
- Note that superscript (j) stands for the jth example, whereas superscript $[j]$ stands for the jth layer.

In figure 2.1.5, we have represented the input features as individual neurons in a layer known as the *input layer*. Each example, in this case, consists of three features, denoted by x_1, x_2, and x_3. Equations 2.1.2 hold true for all the neurons in the hidden and the output layer. We calculate the activation for each neuron in a particular layer denoted as $a_i^{[l]}$ (activation of the ith neuron in the lth layer) and then feed this activation as input to all the neurons in the next layer. The input layer is also known as the zeroth layer, and the inputs x_i are also denoted as $a_i^{[0]}$ (since $l = 0$). So, the activation vectors for the neural network in figure 2.1.5 can be written as

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \mathbf{a}^{[0]} = \begin{bmatrix} a_1^{[0]} \\ a_2^{[0]} \\ a_3^{[0]} \end{bmatrix}, \quad \mathbf{a}^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix}, \quad \mathbf{a}^{[2]} = \begin{bmatrix} a_1^{[2]} \end{bmatrix} = \hat{y} \tag{2.1.6}$$

Writing equations 2.1.2 for all the neurons in the hidden layer, we get the following.

$$\begin{aligned}
z_1^{[1]} &= \mathbf{w}_1^{[1]\intercal}\mathbf{x} + b_1^{[1]}, \quad a_1^{[1]} = g\left(z_1^{[1]}\right) \\
z_2^{[1]} &= \mathbf{w}_2^{[1]\intercal}\mathbf{x} + b_2^{[1]}, \quad a_2^{[1]} = g\left(z_2^{[1]}\right) \\
z_3^{[1]} &= \mathbf{w}_3^{[1]\intercal}\mathbf{x} + b_3^{[1]}, \quad a_3^{[1]} = g\left(z_3^{[1]}\right) \\
z_4^{[1]} &= \mathbf{w}_4^{[1]\intercal}\mathbf{x} + b_4^{[1]}, \quad a_4^{[1]} = g\left(z_4^{[1]}\right)
\end{aligned}$$

Matrix version of the above equations is the following.

$$\begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix} = \begin{bmatrix} — & \mathbf{w}_1^{[1]\intercal} & — \\ — & \mathbf{w}_2^{[1]\intercal} & — \\ — & \mathbf{w}_3^{[1]\intercal} & — \\ — & \mathbf{w}_4^{[1]\intercal} & — \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix} = \begin{bmatrix} \mathbf{w}_1^{[1]\intercal}\mathbf{x} + b_1^{[1]} \\ \mathbf{w}_2^{[1]\intercal}\mathbf{x} + b_2^{[1]} \\ \mathbf{w}_3^{[1]\intercal}\mathbf{x} + b_3^{[1]} \\ \mathbf{w}_4^{[1]\intercal}\mathbf{x} + b_4^{[1]} \end{bmatrix}$$

Using the notation introduced in the notation box, we can write the above equation as

$$\underset{4\times1}{\mathbf{z}^{[1]}} = \underset{4\times3}{\mathbf{w}^{[1]}}\ \underset{3\times1}{\mathbf{x}} + \underset{4\times1}{\mathbf{b}^{[1]}} = \underset{4\times3}{\mathbf{w}^{[1]}}\underset{3\times1}{\mathbf{a}^{[0]}} + \underset{4\times1}{\mathbf{b}^{[1]}}$$

And, the activation vector would be given by

$$\underset{4\times1}{\mathbf{a}^{[1]}} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix} = \begin{bmatrix} g\left(z_1^{[1]}\right) \\ g\left(z_2^{[1]}\right) \\ g\left(z_3^{[1]}\right) \\ g\left(z_4^{[1]}\right) \end{bmatrix} = g(\underset{4\times1}{\mathbf{z}^{[1]}})$$

Similarly, for the second (or the output) layer for the neural network in figure 2.1.5, the equations are the following.

$$\underset{1\times1}{\mathbf{z}^{[2]}} = \underset{1\times4}{\mathbf{w}^{[2]}}\underset{4\times1}{\mathbf{a}^{[1]}} + \underset{1\times1}{\mathbf{b}^{[2]}}$$

$$\underset{1\times1}{\mathbf{a}^{[2]}} = g(\underset{1\times1}{\mathbf{z}^{[1]}}) = \hat{y}$$

So, for any layer l, the equations are given by

$$\mathbf{z}^{[l]} = \mathbf{w}^{[l]}\mathbf{a}^{[l-1]} + \mathbf{b}^{[l]} \tag{2.1.7a}$$

$$\mathbf{a}^{[l]} = g\left(\mathbf{z}^{[l]}\right) \tag{2.1.7b}$$

Vectorizing across multiple examples

To compute the output of our neural network shown in figure 2.1.5 across all the m examples, we need to implement equations 2.1.7 for $l = 1$ and $l = 2$ m times, i.e., for $i = 1, 2, \ldots, m$, implement the following

$$\mathbf{z}^{[1](i)} = \mathbf{w}^{[1]}\mathbf{a}^{[0](i)} + \mathbf{b}^{[1]}$$

$$\mathbf{a}^{[1]\,(i)} = g\left(\mathbf{z}^{[1]\,(i)}\right)$$
$$\mathbf{z}^{[2]\,(i)} = \mathbf{w}^{[2]}\mathbf{a}^{[1]\,(i)} + \mathbf{b}^{[2]}$$
$$\mathbf{a}^{[2]\,(i)} = g\left(\mathbf{z}^{[2]\,(i)}\right) = \hat{y}^{(i)}$$

A vectorized implementation would simultaneously include all the m examples, and hence would be computationally efficient. This is done in the following way.

$$\mathbf{Z}^{[1]} = \mathbf{W}^{[1]}\mathbf{X} + \mathbf{b}^{[1]} = \mathbf{W}^{[1]}\mathbf{A}^{[0]} + \mathbf{b}^{[1]}$$
$$\mathbf{A}^{[1]} = g\left(\mathbf{Z}^{[1]}\right)$$
$$\mathbf{Z}^{[2]} = \mathbf{W}^{[2]}\mathbf{A}^{[1]} + \mathbf{b}^{[2]}$$
$$\mathbf{A}^{[2]} = g\left(\mathbf{Z}^{[2]}\right) = \widehat{\mathbf{Y}}$$

Here, the uppercase matrices are constructed by stacking the corresponding lowercase vectors for each example (i) as columns, just like the definition of $\mathbf{X}$ and $\widehat{\mathbf{Y}}$ in equations 2.1.4. So, for any general lth layer in a feedforward neural network, the vectorized equations are

$$\mathbf{Z}^{[l]} = \mathbf{W}^{[l]}\mathbf{A}^{[l-1]} + \mathbf{b}^{[l]} \tag{2.1.8a}$$

$$\mathbf{A}^{[l]} = g\left(\mathbf{Z}^{[l]}\right) \tag{2.1.8b}$$

2.1.3 Activation Functions

An *activation function* is (mostly a non-linear) function that gets applied to the output of neurons before feeding it into the neurons of the proceeding layer. The use of activation functions is one of the primary reasons behind the competency of neural networks in learning non-linearity in the data. Following is a description of some of the most commonly used activation functions.

Sigmoid (σ)

Following is the mathematical expression for the sigmoid activation function.

$$g(x) = \sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.1.9}$$

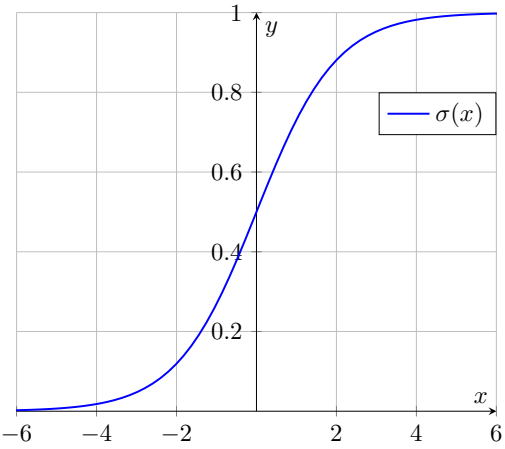


Figure 2.1.6: The sigmoid activation function (see equation 2.1.9).

Figure 2.1.6 shows that the sigmoid function lies in the range $(0, 1)$. Hence, it is an ideal choice for tasks predicting probability. However, it is flat for large and small values of x, which makes it prone to vanishing and exploding gradients. Hence, the sigmoid activation function is mainly used only in the output layer in tasks predicting probability.

Hyperbolic Tangent (tanh)

Following is the mathematical expression for the hyperbolic tangent (or the tanh) activation function.

$$g(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.1.10}$$

Figure 2.1.7 shows that the hyperbolic tangent or the tanh function lies in the range $(-1, 1)$, and is centered around 0. Like the sigmoid function, tanh is also prone to vanishing and exploding gradients for similar reasons. However, due to centering around 0, the movement of gradients is accessible in both directions. The hidden layers mostly use tanh as the activation function.

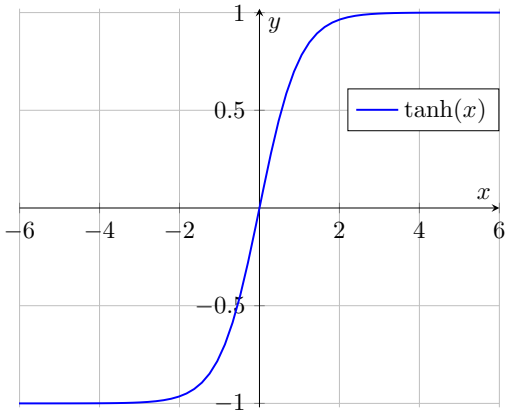


Figure 2.1.7: The hyperbolic tangent activation function (see equation 2.1.10).

Rectified Linear Unit (ReLU)

Following is the mathematical expression for the rectified linear unit (or the ReLU) activation function.

$$g(x) = \max(0, x) \tag{2.1.11}$$

Figure 2.1.8 shows that the ReLU function is 0 for all $x < 0$, and is equivalent to the line $y = x$ for all $x > 0$. As the gradients are not flat for $x > 0$, this partially solves the problem of vanishing and exploding gradients. However, the zero gradients for the negative inputs can be a slight hindrance while training since the weights corresponding to such inputs will not get updated. This is also known as the dying ReLU problem. Again, the hidden layers mostly use the ReLU activation function.

Leaky ReLU

Following is the mathematical expression for the leaky ReLU activation function.

$$g(x) = \max(\alpha x, x) \tag{2.1.12}$$

where, $\alpha < 1$.

Figure 2.1.9 shows the leaky ReLU activation function. The leaky ReLU activation solves the dying ReLU problem by giving a slight negative slope

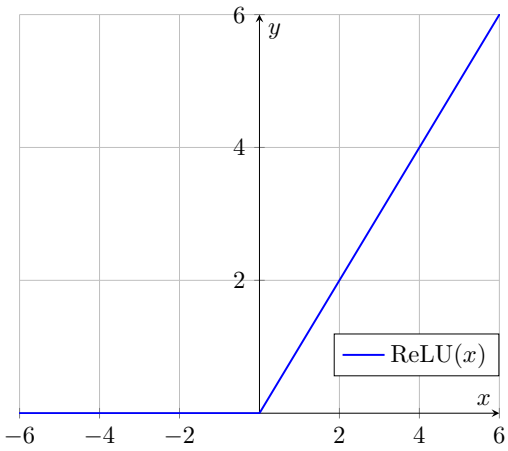


Figure 2.1.8: The rectified linear unit (ReLU) activation function (see equation 2.1.11).

to the negative inputs. Again, the hidden layers mostly use the leaky ReLU activation function.

2.1.4 Loss and Cost Functions

Learning refers to updating the parameters w and b of a neural network such that the predicted output $\hat{y}$ closely matches the actual value y. The *loss function* ($\mathcal{L}$) and the *cost function* ($\mathcal{J}$) quantify this closeness. The loss quantifies it for one particular example (i), whereas the cost simultaneously quantifies it for all the m examples. So, the cost is nothing but the loss summed over all the m examples.

Binary Cross-Entropy (BCE)

The BCE loss and cost is used in binary classification problems.

$$\mathcal{L} = -\left(y^{(i)} \log \hat{y}^{(i)} + \left(1 - y^{(i)}\right) \log\left(1 - \hat{y}^{(i)}\right)\right) \tag{2.1.13}$$

$$\mathcal{J} = -\sum_{i=1}^{m} \left(y^{(i)} \log \hat{y}^{(i)} + \left(1 - y^{(i)}\right) \log\left(1 - \hat{y}^{(i)}\right)\right) \tag{2.1.14}$$

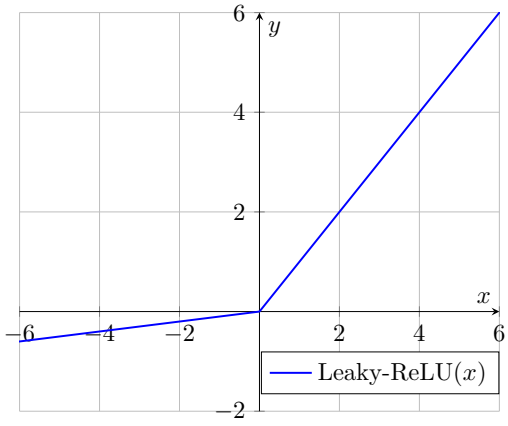


Figure 2.1.9: The Leaky ReLU activation function with $\alpha = 0.1$ (see equation 2.1.12).

Categorical Cross-Entropy (CCE)

The CCE loss and cost extends the idea of BCE to multi-class classification. For n output classes, the CCE loss and cost are given by the following.

$$\mathcal{L} = \sum_{c=1}^{n} y_c \log \hat{y}_c \tag{2.1.15}$$

$$\mathcal{J} = \sum_{i=1}^{m} \left(\sum_{c=1}^{n} y_c \log \hat{y}_c \right) \tag{2.1.16}$$

Mean Squared Error (MSE)

Regression tasks most commonly use MSE. It is common to report it by taking the square root to match the dimensions, also known as root-MSE (RMSE).

$$\mathcal{L} = \left(y^{(i)} - \hat{y}^{(i)} \right)^2 \tag{2.1.17}$$

$$\mathcal{J} = \frac{1}{m} \sum_{i=1}^{m} \left(y^{(i)} - \hat{y}^{(i)} \right)^2 \tag{2.1.18}$$

Due to the square, outliers influence MSE a lot.

Mean Absolute Error (MAE)

Again, regression tasks most commonly use MAE.

$$\mathcal{L} = \left| y^{(i)} - \hat{y}^{(i)} \right| \tag{2.1.19}$$

$$\mathcal{J} = \frac{1}{m} \sum_{i=1}^{m} \left| y^{(i)} - \hat{y}^{(i)} \right| \tag{2.1.20}$$

Outliers do not influence MAE as much as MSE. It also, by default, gives the error in the proper dimensions.

2.1.5 Gradient Descent (How a Neural Network Learns)

A neural network learns by updating the parameters w's and b's by trying to minimize the cost function $\mathcal{J}(w, b)$. An optimization algorithm known as *gradient descent* carries this out. We want to find all the parameters w's, b's of the neural network that minimizes $\mathcal{J}(w, b)$. Gradient descent first randomly initializes these parameters and then carries out the following steps.

$$\begin{aligned} &\text{repeat until convergence: } \{ \\ &\quad w \leftarrow w - \alpha \frac{\partial \mathcal{J}(w, b)}{\partial w} \\ &\quad b \leftarrow b - \alpha \frac{\partial \mathcal{J}(w, b)}{\partial b} \\ &\} \end{aligned} \tag{2.1.21}$$

Here, α is known as the *learning rate*. Usually, $\alpha < 1$. The larger the learning rate, the larger the strides taken by gradient descent, and vice versa.

The cost function $\mathcal{J}(w, b)$ can be thought of as a surface with respect to the weights w and biases b. Each step of gradient descent can be thought of as traversing this surface in the steepest instantaneous direction, finally reaching a minima, as shown in figure 2.1.10. Once these weights and biases are found corresponding to the minima, we say that the model is trained. Figure 2.1.11 shows a general workflow of a supervised deep learning task.

2.1.6 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are used widely on spatial data like images. Various cutting-edge deep learning applications, like self-driving

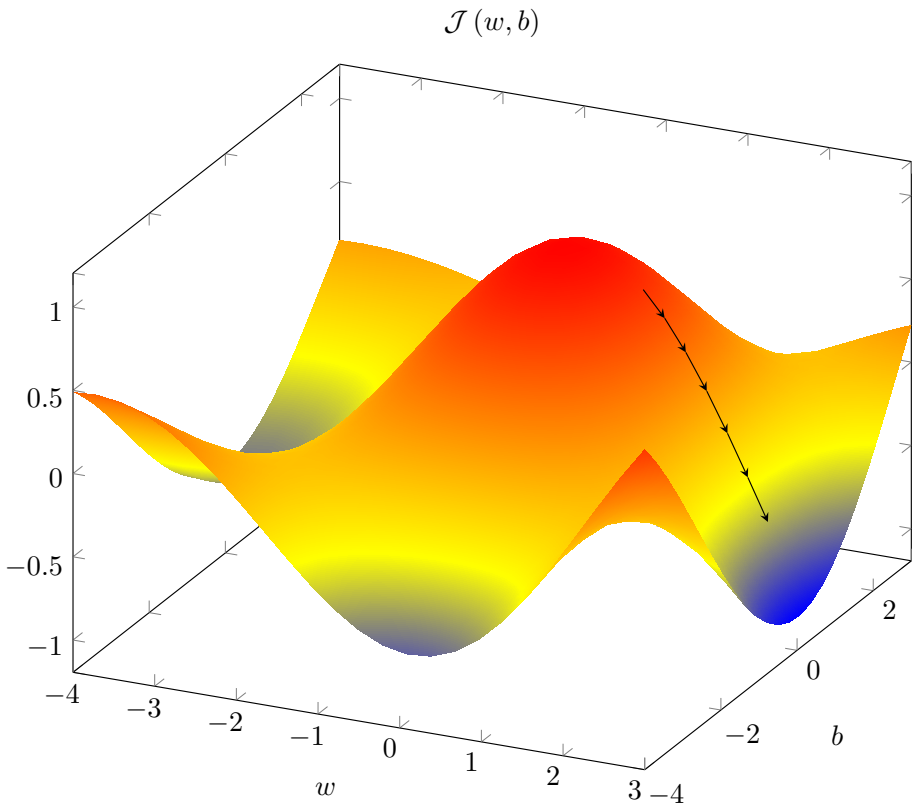


Figure 2.1.10: Pictorial representation of gradient descent algorithm.

cars, object detection, image segmentation, image classification, recommender systems, medical image analysis, brain-computer interfaces, and many more, involve CNNs.

A grayscale image is nothing but a 2D array (or a matrix) consisting of a pixel value for each pixel. A colored image is a 3D array with three values for each pixel (corresponding to RGB channels). If we want to feed in a $1000 \times 1000 \times 3$ image into a fully connected neural network, the input would be $1000 \times 1000 \times 3 = 3$ million dimensional. So, the input layer will consist of 3 million neurons. Also, assuming the first hidden layer has 1000 neurons, the weights between the input and the first hidden layer would be 3 billion. With these many parameters, it is almost impossible to get enough data to prevent the neural network from overfitting. Also, the computational and memory requirements to train such a neural network is practically infeasible. This is

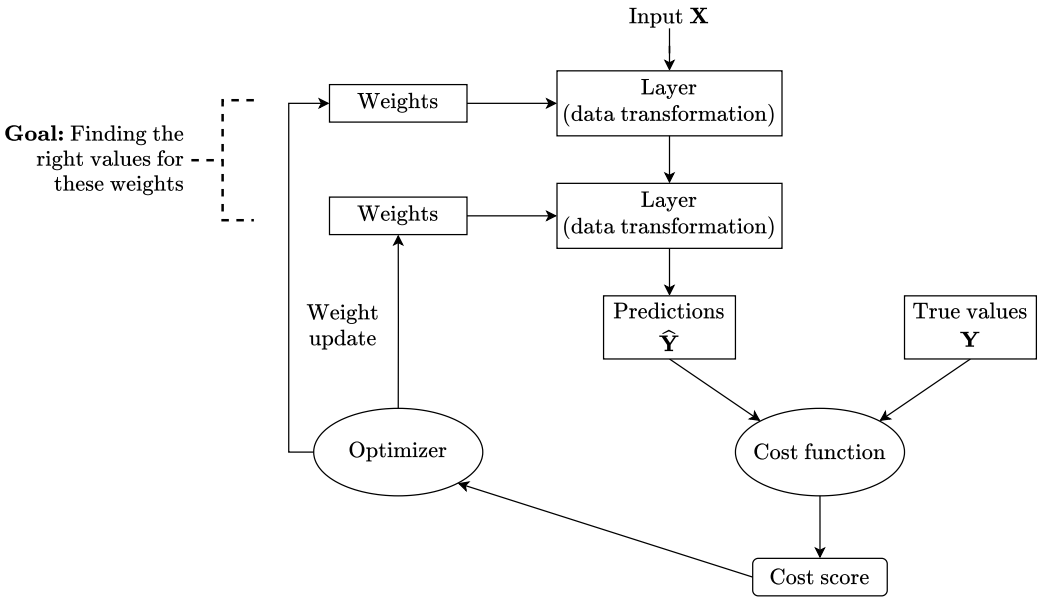


Figure 2.1.11: General workflow of a supervised deep learning task (Chollet, 2021).

where the convolution operation comes to our rescue, which is a fundamental building block of CNNs.

2D Convolution

Figure 2.1.12 shows the convolution of image $\mathbf{I}$ with a *kernel* (also known as a *filter*) $\mathbf{K}$. Their convolution is denoted as $\mathbf{I} * \mathbf{K}$. Elements in their convolution are obtained by placing the kernel over the image and adding the element-wise product of the overlapping elements. We can also see that if $\mathbf{I}$ is an $n \times n$ image and $\mathbf{K}$ is a $k \times k$ kernel, then their convolution has dimensions $(n-k+1) \times (n-k+1)$. The kernel is what learns various patterns in the image. A neural network can learn the kernel elements to perform a particular task like image classification, edge detection, and much more.

Padding the given image with a border of zero-valued pixels around it is common to ensure that the convolution does not shrink with each applied kernel. In this case, the convolution has dimensions $(n + 2p - k + 1) \times (n + 2p - k + 1)$, where p is the number of padded borders.

Stride is also a standard adjustable parameter during convolution. It is

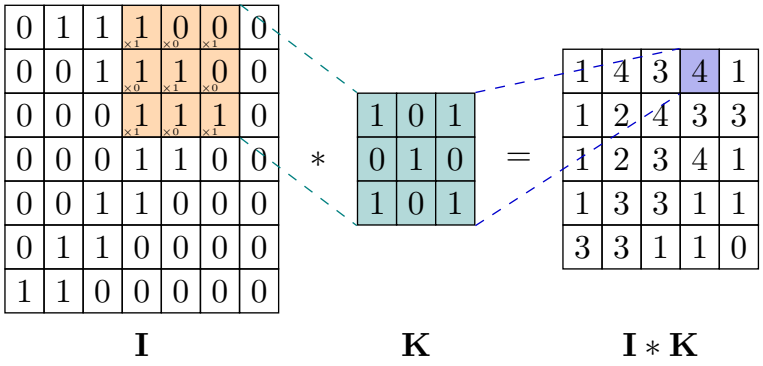


Figure 2.1.12: 2D Convolution of an image **I** with a kernel (also known as a filter) **K**.

nothing but the number of rows or columns a kernel can skip while traversing the image. In this case, the convolution has dimensions

$$\left\lfloor \frac{n+2p-k}{s} + 1 \right\rfloor \times \left\lfloor \frac{n+2p-k}{s} + 1 \right\rfloor \tag{2.1.22}$$

where s is the stride length[5].

3D Convolution

RGB images, i.e., images with three channels, use 3D convolution. It works analogously to 2D convolution, except the image and the kernel are now 3D arrays. For a general case with an $n \times n$ image and a $k \times k$ kernel, both having c channels, their convolution will again have dimensions given by equation 2.1.22. If there are n_k kernels, then the resulting array formed by taking convolutions will have the following dimensions.

$$\left\lfloor \frac{n+2p-k}{s} + 1 \right\rfloor \times \left\lfloor \frac{n+2p-k}{s} + 1 \right\rfloor \times n_k \tag{2.1.23}$$

Using multiple kernels allows the network to learn multiple patterns in the images.

[5] $\lfloor\ \rfloor$ denotes the floor function.

One Layer of a Convolutional Neural Network

Figure 2.1.13 illustrates the working of a single layer of a convolutional neural network with two kernels.

First, convolution between the image and the two kernels gives two 2D matrices, as indicated by figure 2.1.13a. Then, we add[6] biases b_1 and b_2 to each element of these convolutions, and an activation function g is applied. Then, we concatenate the resulting 2D matrices to form a $4 \times 4 \times 2$ array, as shown in figure 2.1.13b. For n_k kernels, this operation would have resulted in a $4 \times 4 \times n_k$ array. This array would act as activation for this layer.

Convolutional vs. Fully Connected Neural Networks

Two significant advantages of using convolutional neural networks over fully connected neural networks for image data are *parameter sharing* and *sparsity of connections*.

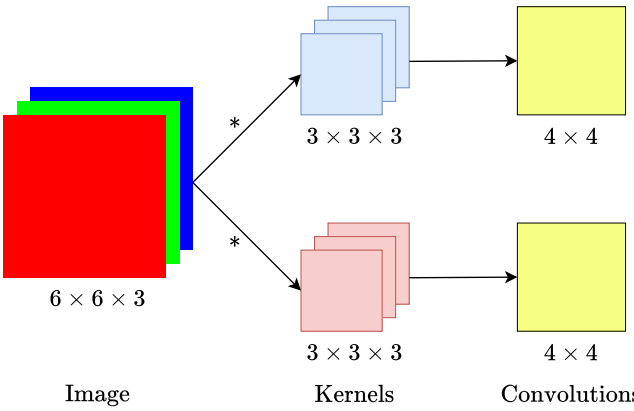
Parameter Sharing A kernel learned to detect a particular feature (like edges) is helpful in all the parts of an image. So, the same parameters are useful to detect a particular feature in an image. So, using kernels on an image results in far fewer parameters than flattening an image and using fully connected layers. This makes a CNN computationally efficient as compared to a fully connected neural network.

Sparsity of Connections As we can see in figure 2.1.12, each value in $\mathbf{I} * \mathbf{K}$ depends only on a small number of values in image $\mathbf{I}$. Due to this, the number of computations in a convolutional layer is far fewer than in a fully connected layer, which makes a CNN computationally efficient.

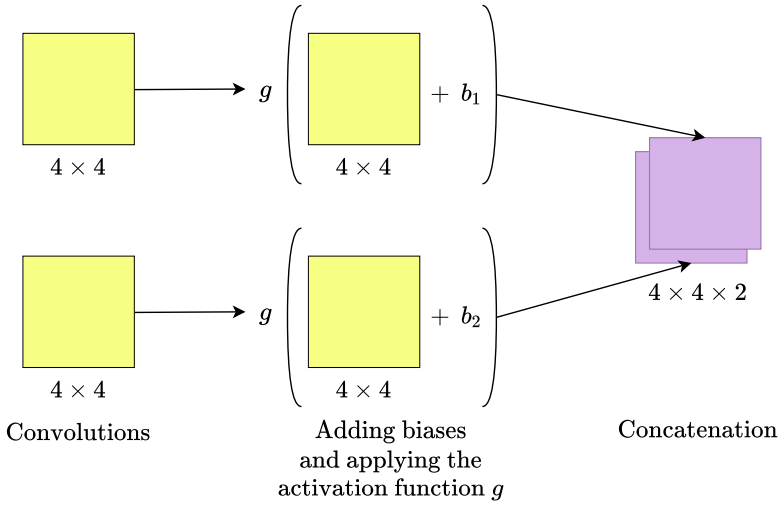
2.1.7 Recurrent Neural Networks

By design, recurrent neural networks (RNNs) work with sequence data. Sequence data is only data ordered into sequences, such as audio, video, and time series data. Some examples of tasks carried out by RNNs are speech recognition, DNA sequence analysis, sentiment classification, machine translation, and time series analysis/forecasting. In each of these examples, the

[6]This is also known as *broadcasting* and is done using the NumPy library in Python.



(a) Convolution of the image with two kernels ("$*$" denotes the convolution operation).



(b) Adding biases, applying the activation function, and concatenation.

Figure 2.1.13: One layer of a convolutional neural network.

input to the model is a sequence. However, this sequence does not necessarily have to be the same length. For example, in machine translation, the input sequence can be lengthy, as well as a short sentence. The same is the case with sentiment analysis. This is one of the significant reasons why fully connected neural networks are useless for such tasks.

Notation

- We represent an input sequence of length T_x as $x^{\langle 1 \rangle}$, $x^{\langle 2 \rangle}$, $\ldots$, $x^{\langle t \rangle}$, $\ldots$, $x^{\langle T_x \rangle}$. Likewise, we represent an output sequence of length T_y as $y^{\langle 1 \rangle}, \ldots, y^{\langle T_y \rangle}$.
- $\mathbf{W}_{pq}$ denotes the weights matrix which has to be multiplied by a quantity corresponding to "q" to calculate a quantity corresponding to "p" (see equations 2.1.24).

Figure 2.1.14 shows the construction of a simple recurrent neural network with $T_x = T_y$. We can see how an input sequence $x^{\langle 1 \rangle}, \ldots, x^{\langle T_x \rangle}$ is fed into

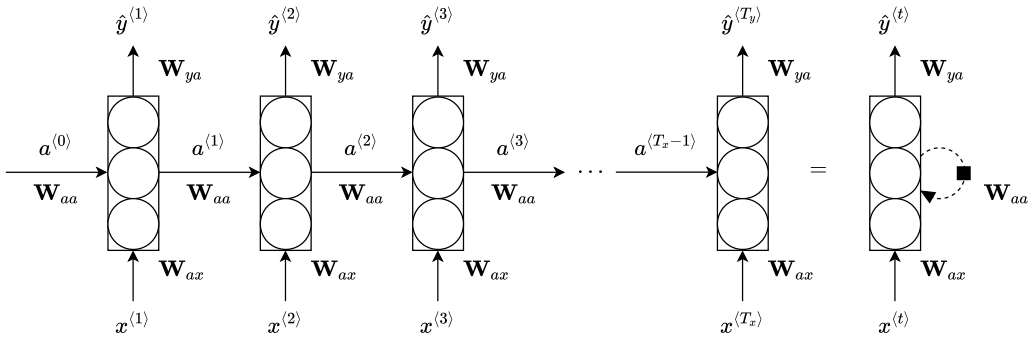


Figure 2.1.14: A recurrent neural network.

a neural network layer and the predicted output sequence $\hat{y}^{\langle 1 \rangle}, \ldots, \hat{y}^{\langle T_y \rangle}$ is obtained. Note that this is just a single neural network layer represented at multiple time steps from left to the right. We can see that the layer predicts $\hat{y}^{\langle t \rangle}$ using not just $x^{\langle t \rangle}$ but also the activation from the previous time step, i.e., $a^{\langle t-1 \rangle}$. The activation at the zeroth time step, i.e., $a^{\langle 0 \rangle}$, is usually just a vector of zeros. The right side of the figure shows the general representation of the same layer at a time step t. The parameters (i.e., the weights and biases) this layer uses at each time step are shared. The matrices $\mathbf{W}_{aa}$, $\mathbf{W}_{ax}$,

and $\mathbf{W}_{ya}$, contain the weights. The following equations calculate the outputs in an RNN.

$$a^{\langle t \rangle} = g_1\left(\mathbf{W}_{aa}a^{\langle t-1 \rangle} + \mathbf{W}_{ax}x^{\langle t \rangle} + b_a\right) \tag{2.1.24a}$$

$$\hat{y}^{\langle t \rangle} = g_2\left(\mathbf{W}_{ya}a^{\langle t \rangle} + b_y\right) \tag{2.1.24b}$$

where, g_1 and g_2 are activation functions, whereas b_a and b_y are biases. Usually, $g_1 = \tanh$.

A simplified version of the above equations can be written as follows.

$$a^{\langle t \rangle} = g_1\left(\mathbf{W}_a\left[a^{\langle t-1 \rangle}, x^{\langle t \rangle}\right] + b_a\right) \tag{2.1.25a}$$

$$\hat{y}^{\langle t \rangle} = g_2\left(\mathbf{W}_y a^{\langle t \rangle} + b_y\right) \tag{2.1.25b}$$

where, $\mathbf{W}_y = \mathbf{W}_{ya}$, the matrix $\mathbf{W}_a$ is a matrix formed by horizontally stacking the matrices $\mathbf{W}_{aa}$ and $\mathbf{W}_{ax}$, i.e.,

$$\mathbf{W}_a = \begin{bmatrix}\mathbf{W}_{aa} & \mathbf{W}_{ax}\end{bmatrix}$$

and, $\left[a^{\langle t-1 \rangle}, x^{\langle t \rangle}\right]$ is a matrix formed by vertically stacking the vectors $a^{\langle t-1 \rangle}$ and $x^{\langle t \rangle}$, i.e.,

$$\left[a^{\langle t-1 \rangle}, x^{\langle t \rangle}\right] = \begin{bmatrix}a^{\langle t-1 \rangle} \\ x^{\langle t \rangle}\end{bmatrix}$$

We compute the cost score and apply gradient descent to minimize it. This is how an RNN learns. Figure 2.1.15 shows a pictorial representation of an RNN unit.

Vanishing Gradients Problem

Sometimes, sequences such as words, time series, and more can have long-term dependencies. For example, whether a word at the end of a sentence should be singular or plural may depend on the subject which comes right at its beginning. It turns out that a basic RNN is not good at capturing such long-term dependencies. For longer sequences, computing gradients to carry out gradient descent is difficult. So, the gradients at the latter part of the RNN may have a hard time affecting the gradients at the initial part (see figure 2.1.14). This makes it difficult for the newly learned information to affect the weights at the initial part of the network, making it harder to capture long-term dependencies. So, the basic RNN model only has many local influences, i.e., the output $\hat{y}^{\langle t \rangle}$ is mainly influenced by values close to it.

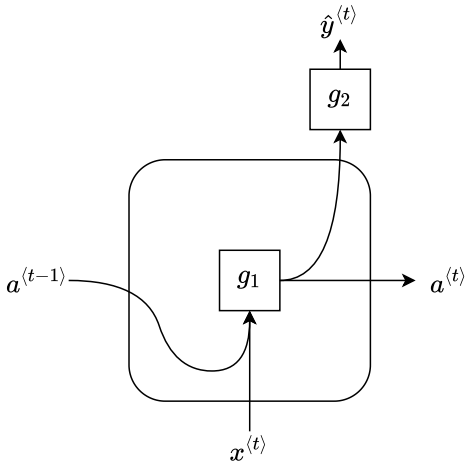


Figure 2.1.15: Pictorial representation of an RNN unit (see equations 2.1.25).

Gated Recurrent Unit (GRU)

The gated recurrent unit, or GRU (Cho et al., 2014), modifies the RNN layer discussed in the previous section. It makes it much better at capturing long-term dependencies in the data and helps with the vanishing gradients problem. The GRU unit has a new variable, c, the *memory cell*, which provides some "memory" to learn long-term dependencies. So, at time t, the memory cell will have some value, $c^{\langle t \rangle}$, which will also be equal[7] to the output of the GRU unit, i.e., $a^{\langle t \rangle}$. Equations 2.1.26[8] that govern the computations of a GRU unit.

$$\tilde{c}^{\langle t \rangle} = \tanh\left(\mathbf{W}_c \left[\Gamma_r \circ c^{\langle t-1 \rangle}, x^{\langle t \rangle}\right] + b_c\right) \tag{2.1.26a}$$

$$\Gamma_u = \sigma\left(\mathbf{W}_u \left[c^{\langle t-1 \rangle}, x^{\langle t \rangle}\right] + b_u\right) \tag{2.1.26b}$$

$$\Gamma_r = \sigma\left(\mathbf{W}_r \left[c^{\langle t-1 \rangle}, x^{\langle t \rangle}\right] + b_r\right) \tag{2.1.26c}$$

$$c^{\langle t \rangle} = \Gamma_u \circ \tilde{c}^{\langle t \rangle} + (1 - \Gamma_u) \circ c^{\langle t-1 \rangle} \tag{2.1.26d}$$

$$a^{\langle t \rangle} = c^{\langle t \rangle} \tag{2.1.26e}$$

$$\hat{y}^{\langle t \rangle} = g\left(a^{\langle t \rangle}\right) \tag{2.1.26f}$$

[7] For LSTMs (next sub-subsection), $c^{\langle t \rangle} \neq a^{\langle t \rangle}$.

[8] In these equations, "$\circ$" denotes the Hadamard product (also known as element-wise product).

At every time step t, we will consider overwriting the memory cell $c^{\langle t \rangle}$ with a value $\tilde{c}^{\langle t \rangle}$ given by equation 2.1.26a, which will be a candidate that can replace $c^{\langle t \rangle}$. Γ_u is known as the *update gate*, given by equation 2.1.26b, whose value lies between 0 and 1. The job of the update gate is to decide when to update the memory cell $c^{\langle t \rangle}$ with its candidate value $\tilde{c}^{\langle t \rangle}$. Equation 2.1.26d reflects this decision. Γ_r is known as the *relevance gate*, given by equation 2.1.26c, which tells us how relevant the previous memory cell value, i.e., $c^{\langle t-1 \rangle}$, is to calculate the next candidate value, i.e., $\tilde{c}^{\langle t \rangle}$. Figure 2.1.16 shows a pictorial representation (Olah, 2015) of a GRU unit.

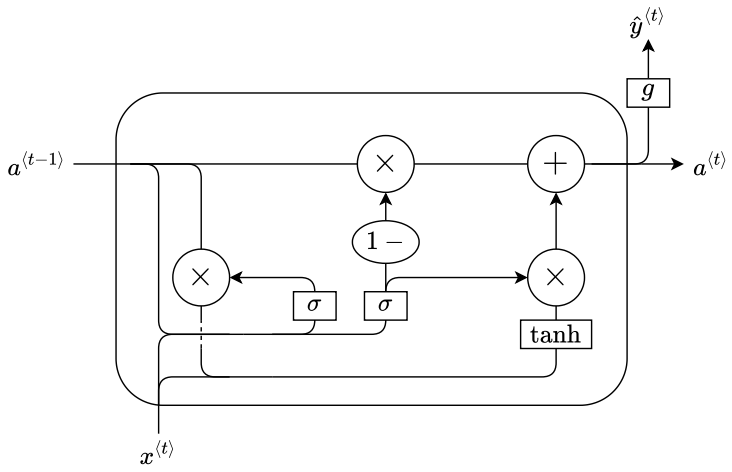


Figure 2.1.16: Pictorial representation of a GRU unit (see equations 2.1.26).

Long Short-Term Memory (LSTM)

Long short-term memory, or LSTM (Hochreiter & Schmidhuber, 1997), is a more robust and general version of the GRU. Equations 2.1.27 govern the computations of an LSTM unit.

$$\tilde{c}^{\langle t \rangle} = \tanh\left(\mathbf{W}_c \left[a^{\langle t-1 \rangle}, x^{\langle t \rangle}\right] + b_c\right) \tag{2.1.27a}$$

$$\Gamma_u = \sigma\left(\mathbf{W}_u \left[a^{\langle t-1 \rangle}, x^{\langle t \rangle}\right] + b_u\right) \tag{2.1.27b}$$

$$\Gamma_f = \sigma\left(\mathbf{W}_f \left[a^{\langle t-1 \rangle}, x^{\langle t \rangle}\right] + b_f\right) \tag{2.1.27c}$$

$$\Gamma_o = \sigma\left(\mathbf{W}_o \left[a^{\langle t-1 \rangle}, x^{\langle t \rangle}\right] + b_o\right) \tag{2.1.27d}$$

$$c^{\langle t \rangle} = \Gamma_u \circ \tilde{c}^{\langle t \rangle} + \Gamma_f \circ c^{\langle t-1 \rangle} \tag{2.1.27e}$$

$$a^{\langle t \rangle} = \Gamma_o \circ \tanh\left(c^{\langle t \rangle}\right) \tag{2.1.27f}$$

$$\hat{y}^{\langle t \rangle} = g\left(a^{\langle t \rangle}\right) \tag{2.1.27g}$$

Here, two new gates, the *forget gate* (Γ_f) and the *output gate* (Γ_o), are introduced (see equations 2.1.27c and 2.1.27d, respectively). Figure 2.1.17 shows a pictorial representation (Olah, 2015) of an LSTM unit.

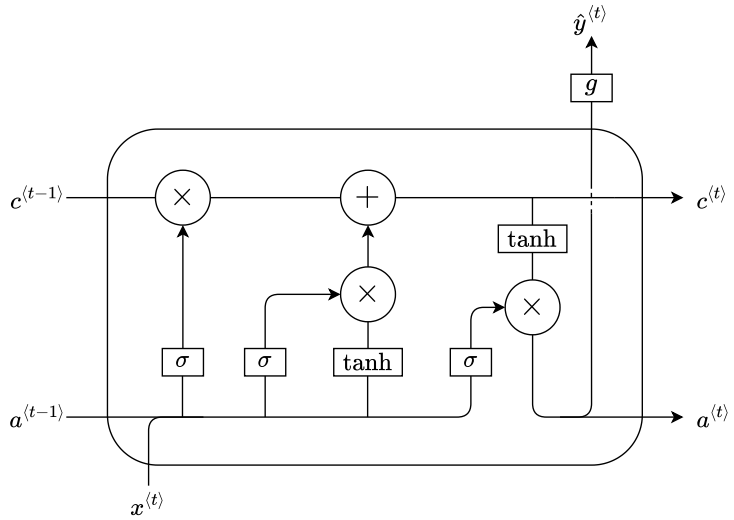


Figure 2.1.17: Pictorial representation of an LSTM unit (see equations 2.1.27).

Deep RNNs

We can construct a deep RNN/GRU/LSTM network by stacking layers of the corresponding units shown in figures 2.1.15, 2.1.16, and 2.1.17. This is mainly useful for learning complex functions.

2.2 Spatiotemporal Forecasting using Deep Learning

The use of deep learning-based spatiotemporal forecasting methods is mainly for precipitation nowcasting, i.e., short-term precipitation forecasting. Here, instead of forecasting a sequence of numbers (i.e., a temporal sequence), the aim is to forecast a series of matrices (i.e., a spatiotemporal sequence), as mentioned in section 1.2.

2.2.1 Convolutional LSTM (ConvLSTM)

Deep learning researchers have proposed various models for spatiotemporal forecasting. Convolutional LSTM or ConvLSTM (Shi et al., 2015) is one of the most fundamental yet powerful models for spatiotemporal forecasting. A ConvLSTM unit is very similar to the LSTM unit, except it employs the convolution operation instead of matrix multiplication that we see in the LSTM equations (equations 2.1.27). Convolution helps extract spatial patterns from the spatiotemporal sequence. Equations 2.2.1 govern the computations of a ConvLSTM unit ("$*$" denotes convolution). Compare these with LSTM equations, i.e., equations 2.1.27.

$$\widetilde{\mathbf{C}}^{\langle t \rangle} = \tanh\left(\mathbf{W}_{\widetilde{\mathbf{C}}\mathbf{X}} * \mathbf{X}^{\langle t \rangle} + \mathbf{W}_{\widetilde{\mathbf{C}}\mathbf{A}} * \mathbf{A}^{\langle t-1 \rangle} + b_c\right) \tag{2.2.1a}$$

$$\Gamma_u = \sigma\left(\mathbf{W}_{\Gamma_u\mathbf{X}} * \mathbf{X}^{\langle t \rangle} + \mathbf{W}_{\Gamma_u\mathbf{A}} * \mathbf{A}^{\langle t-1 \rangle} + \mathbf{W}_{\Gamma_u\mathbf{C}} \circ \mathbf{C}^{\langle t-1 \rangle} + b_u\right) \tag{2.2.1b}$$

$$\Gamma_f = \sigma\left(\mathbf{W}_{\Gamma_f\mathbf{X}} * \mathbf{X}^{\langle t \rangle} + \mathbf{W}_{\Gamma_f\mathbf{A}} * \mathbf{A}^{\langle t-1 \rangle} + \mathbf{W}_{\Gamma_f\mathbf{C}} \circ \mathbf{C}^{\langle t-1 \rangle} + b_f\right) \tag{2.2.1c}$$

$$\Gamma_o = \sigma\left(\mathbf{W}_{\Gamma_o\mathbf{X}} * \mathbf{X}^{\langle t \rangle} + \mathbf{W}_{\Gamma_o\mathbf{A}} * \mathbf{A}^{\langle t-1 \rangle} + \mathbf{W}_{\Gamma_o\mathbf{C}} \circ \mathbf{C}^{\langle t-1 \rangle} + b_o\right) \tag{2.2.1d}$$

$$\mathbf{C}^{\langle t \rangle} = \Gamma_u \circ \widetilde{\mathbf{C}}^{\langle t \rangle} + \Gamma_f \circ \mathbf{C}^{\langle t-1 \rangle} \tag{2.2.1e}$$

$$\mathbf{A}^{\langle t \rangle} = \Gamma_o \circ \tanh\left(\mathbf{C}^{\langle t \rangle}\right) \tag{2.2.1f}$$

$$\widehat{\mathbf{Y}}^{\langle t \rangle} = g\left(\mathbf{A}^{\langle t \rangle}\right) \tag{2.2.1g}$$

Like previously discussed RNN architectures, we can construct a deep ConvLSTM network consisting of multiple ConvLSTM layers. Input to each ConvLSTM layer is a 4D array of dimensions "sequence length $\times$ height $\times$ width $\times$ channels". In our implementation, we provide multiple input samples (training examples) simultaneously, which makes it a 5D array of dimensions "samples $\times$ sequence length $\times$ height $\times$ width $\times$ channels". In terms of our data

(discussed in section 3.1), these dimensions can also be written as follows.

$$\text{samples} \times \text{sequence length} \times \text{latitudes} \times \text{longitudes} \times \text{variables} \quad (2.2.2)$$

Chapter 3

Data and Methodology

3.1 Data

For PM2.5 emissions, we have used the FINN[1] (Fire INventory from NCAR) (Wiedinmyer et al., 2011) data. In addition, we have used NDVI[2] (Normalized Difference Vegetation Index) from National Oceanic and Atmospheric Administration (Vermote et al., 2019), temperature, surface pressure, wind (u, v), and total cloud cover from ECMWF reanalysis[3] version 5 (Hersbach et al., 2020) as additional variables. We have data from 2002 to 2018, i.e., 17 years. Table 3.1 summarizes the variables.

Variable (Unit)	Source	Spatial Resolution	Temporal Resolution
PM2.5 ($\mu g\, m^{-2}\, s^{-1}$)	FINN	$0.1° \times 0.1°$	Hourly
NDVI (None)	NOAA	$0.05° \times 0.05°$	Daily
Temperature (K)	ERA5	$0.25° \times 0.25°$	Hourly
Surface Pressure (Pa)	ERA5	$0.25° \times 0.25°$	Hourly
Wind (u, v) ($m\, s^{-1}$)	ERA5	$0.25° \times 0.25°$	Hourly
Total Cloud Cover (None)	ERA5	$0.25° \times 0.25°$	Hourly

Table 3.1: Summary of the data.

[1] https://www.acom.ucar.edu/Data/fire/

[2] https://www.ncei.noaa.gov/products/climate-data-records/normalized-difference-vegetation-index

[3] https://www.ecmwf.int/en/forecasts/datasets/reanalysis-datasets/era5

3.2 Data Preprocessing

3.2.1 Matching the Spatial Resolution

To construct the model, we need the spatial resolution of all the variables to be consistent with each other. Depending upon the variable, we upscale or downscale its spatial resolution to match the spatial resolution of the target variable, i.e., PM2.5, which is $0.1° \times 0.1°$. Table 3.2 summarizes this step.

Variable	Before	Scaling Technique	After
PM2.5	$0.1° \times 0.1°$	None	$0.1° \times 0.1°$
NDVI	$0.05° \times 0.05°$	Nearest Coordinates	$0.1° \times 0.1°$
Temperature	$0.25° \times 0.25°$	Linear Interpolation	$0.1° \times 0.1°$
Surface Pressure	$0.25° \times 0.25°$	Linear Interpolation	$0.1° \times 0.1°$
Wind (u, v)	$0.25° \times 0.25°$	Linear Interpolation	$0.1° \times 0.1°$
Total Cloud Cover	$0.25° \times 0.25°$	Linear Interpolation	$0.1° \times 0.1°$

Table 3.2: Matching the spatial resolution of the variables.

3.2.2 Temporal Resolution

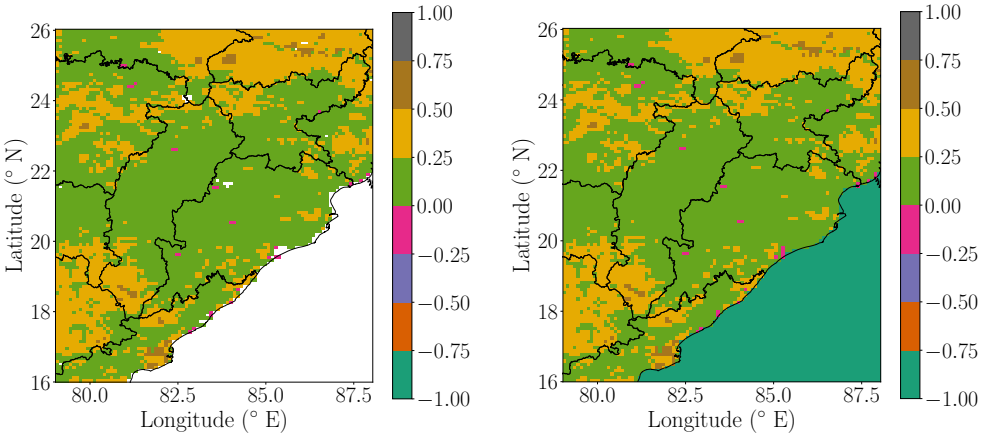
We want a three-day forecast for PM2.5 emissions. Hence, we convert the temporal resolution of all the variables from hourly to daily. In other words, we consider a single value for each day.

3.2.3 Imputation of Missing Values

A missing value occurs when the instrument does not record data at a particular point. Computers represent it as "`NaN`" (not a number). Missing values are highly detrimental to deep learning models as they hinder calculating cost scores and gradients. Hence, imputing these missing values with some educated guesses is crucial.

NDVI

Fortunately, only NDVI had missing values in our data. There were two types of missing values: a small amount of randomly missing values for some



(a) NDVI before `NaN` imputation. (b) NDVI after `NaN` imputation.

Figure 3.2.1: NDVI before and after `NaN` imputation.

(latitude, longitude) pairs and missing values corresponding to the Bay of Bengal region in the central India and northeast Indian subcontinent data.

Random Missing Values We impute these randomly occurring missing values using their surrounding values via interpolation.

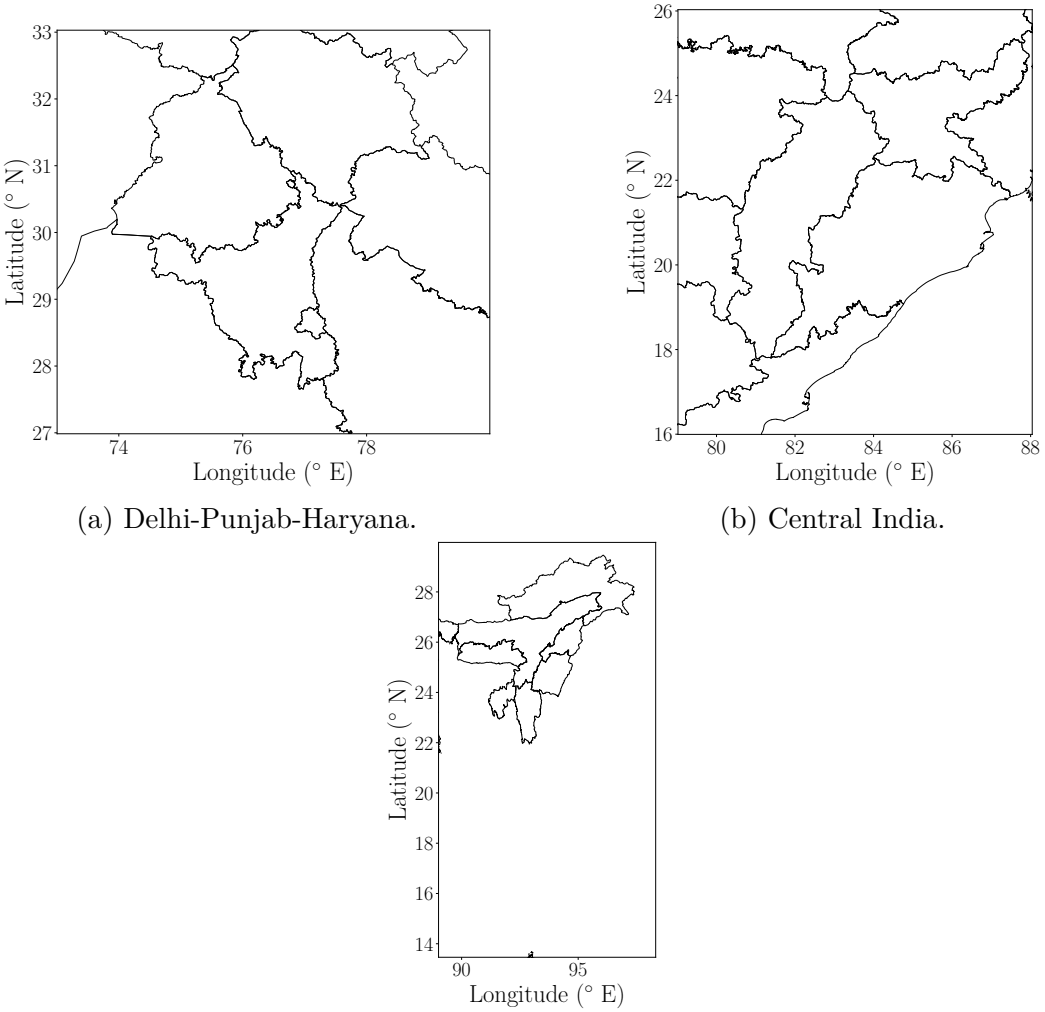
Bay of Bengal Missing Values To impute these missing values, we mask the bay of Bengal area with -1. The reason we select -1 is that the NDVI value corresponding to water is -1.

Figure 3.2.1 shows the before imputation and after imputation plots of NDVI for the central India region.

3.2.4 Data Slicing

Spatial Slicing

As we are only interested in the stubble burning region (Delhi-Punjab-Haryana), the northeast Indian subcontinent, and central India, we spatially slice the data into three subsets corresponding to these three regions. Table 3.3 shows the latitudinal and longitudinal extent of these three regions of interest. Figure 3.2.2 shows the spatial extent of the three spatially sliced data corresponding to the three regions of interest.



(a) Delhi-Punjab-Haryana.

(b) Central India.

(c) Northeast Indian Subcontinent.

Figure 3.2.2: Spatial extent of the three subsets formed after spatially slicing the data corresponding to the three regions of interest.

Area	Latitudinal Extent	Longitudinal Extent
Delhi-Punjab-Haryana	27° N to 33° N	73° E to 80° E
Northeast India	13.5° N to 30° N	89° E to 98.3° E
Central India	16° N to 26° N	79° E to 88° E

Table 3.3: Spatially slicing the data corresponding to the three regions of interest.

Temporal Slicing

Stubble burning and forest fires occur only in a particular period during a year. Table 3.4 shows the corresponding periods of the three regions of interest.

Area	Period
Delhi-Punjab-Haryana	September 15 to November 30
Northeast India	January 1 to May 31
Central India	February 1 to May 31

Table 3.4: Temporally slicing the data corresponding to the periods of the three regions of interest.

As these events occur only in their corresponding period, including the data for the entire year would add much noise to the model and hinder the training process. Hence, we only consider the corresponding temporal slice of the data for the three regions of interest to build the model.

3.2.5 Supervised Splits

Recall that in section 1.2, we said the following:

- We have a 3-D[4] array corresponding to each observation, i.e., a 3-D array for each time step.
- We want to predict the most likely sequence of K such arrays given the past sequence of J arrays.

[4]We can think of this as a matrix with seven values in each position as there are seven variables, namely, PM2.5, NDVI, temperature, surface pressure, wind (u, v), and total cloud cover.

Since our data is daily, we have a 3-D array corresponding to each day. Now, we pick $J = K = 5$, i.e., given data for five consecutive days, we want to forecast for the next five consecutive days. Figure 3.2.3 summarizes what we want our model to do.

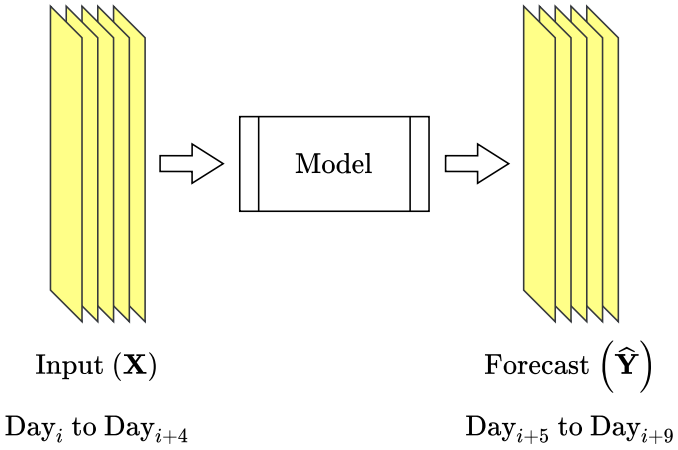


Figure 3.2.3: We want our model to take in five consecutive days as input and forecast the next five consecutive days..

We must split the data according to this input-output format to train the model. In other words, we must form the supervised splits of the data. Table 3.5 shows the supervised splits for the three regions of interest. We form these splits for all 17 years. We will simultaneously show our model the input and output arrays during training.

3.2.6 Data Transformation

Data transformation helps convert the raw data into a format or structure more suitable for model training. We need data transformation for the following principal reasons:

- It helps reduce the skewness of the data distribution.
- It helps in managing outliers present in the data.

There are three common types of transformations used in time series analysis: *log transformation*, *exponential transformation*, and *power transformation*.

Input ($\mathbf{X}$)	Output $\left(\widehat{\mathbf{Y}}\right)$
Sept. 15 to Sept. 19	Sept. 20 to Sept. 24
Sept. 16 to Sept. 20	Sept. 21 to Sept. 25
⋮	⋮
Nov. 21 to Nov. 25	Nov. 26 to Nov. 30

(a) Delhi-Punjab-Haryana.

Input ($\mathbf{X}$)	Output $\left(\widehat{\mathbf{Y}}\right)$
Feb. 1 to Feb. 5	Feb. 6 to Feb. 10
Feb. 2 to Feb. 6	Feb. 7 to Feb. 11
⋮	⋮
May 22 to May 26	May 27 to May 31

(b) Central India.

Input ($\mathbf{X}$)	Output $\left(\widehat{\mathbf{Y}}\right)$
Jan. 1 to Jan. 5	Jan. 6 to Jan. 10
Jan. 2 to Jan. 6	Jan. 7 to Jan. 11
⋮	⋮
May 22 to May 26	May 27 to May 31

(c) Northeast Indian Subcontinent.

Table 3.5: Supervised splits of the data.

Log Transformation

As the name suggests, log transformation is a data transformation technique that replaces each data point x with $\log(x)$.

$$x_{\text{final}} = \log\left(x_{\text{initial}}\right)$$

If the data contains zeros, it is common to add one before taking the logarithm, i.e.,

$$x_{\text{final}} = \log\left(x_{\text{initial}} + 1\right)$$

Log transformation generally helps in converting a right-skewed distribution to a normal distribution.

Exponential Transformation

As the name suggests, exponential transformation is a data transformation technique that replaces each data point x with $\exp(x)$. It is essentially the inverse of log transformation.

$$x_{\text{final}} = \exp\left(x_{\text{initial}}\right)$$

Exponential transformation generally helps in converting a left-skewed distribution to a normal distribution.

Power Transformation

As the name suggests, power transformation is a data transformation technique that replaces each data point x with x^{γ}.

$$x_{\text{final}} = \left(x_{\text{initial}}\right)^{\gamma}$$

Commonly, $\gamma = 1/2$ (square root transform), $1/3$ (cube root transform), ...

Power transformation with $\gamma < 1$ helps convert a right-skewed distribution to a normal one, and power transformation with $\gamma > 1$ helps convert a left-skewed distribution into a normal one.

3.2.7 Data Scaling

Scaling helps to scale the data down to a particular range, which significantly benefits gradient descent to converge faster. Two common data scaling techniques are *standardization* and *normalization*.

Standardization

Standardization changes the data distribution such that the values center around the mean of the data with a unit standard deviation.

$$x_{\text{final}} = \frac{x_{\text{initial}} - \mu}{\sigma}$$

where, μ is the mean, and σ is the standard deviation of the raw data.

Normalization

Normalization rescales the data such that the values end up ranging between 0 and 1. Gradient descent converges faster if all the variables/features in the data are normalized.

$$x_{\text{final}} = \frac{x_{\text{initial}} - x_{\text{min}}}{x_{\text{max}} - x_{\text{min}}}$$

where, x_{max} and x_{min} are the maximum and the minimum, respectively, of the raw data.

Per-day Normalization Sometimes, instead of scaling with the maximum and minimum of the entire data, it is beneficial to scale each day with its corresponding maximum and minimum.

$$x_{i\text{final}} = \frac{x_{i\text{initial}} - (x_{i\text{initial}})_{\text{min}}}{(x_{i\text{initial}})_{\text{max}} - (x_{i\text{initial}})_{\text{min}}}$$

We do this for all the days, i.e., for all values of i.

3.2.8 Training-Validation-Test Split

Training-Validation-Test split is a common splitting technique used in data science. The model is trained on the training set and evaluated on the validation set. We make necessary changes in the model based on the model's performance on the validation set (also known as hyperparameter tuning). Finally, using these parameters, we train the final model on the training and the validation set and report the performance on the test set. Table 3.6 shows the training-validation-test split.

Training Set	Validation Set	Test Set
2002 to 2014	2015 to 2016	2017 to 2018

Table 3.6: The training-validation-test split done on the data.

3.3 Model

As mentioned earlier, we propose using ConvLSTM to forecast PM2.5 emissions. It is a fundamental yet powerful deep learning-based model for spatiotemporal forecasting. We will construct a multi-layered ConvLSTM model architecture for the same using Keras API with TensorFlow as the backend.

3.3.1 Model Architecture

Tables 3.7, 3.8, and 3.9 show the ConvLSTM architectures that gave us the best result for the Delhi-Punjab-Haryana, northeast Indian subcontinent, and central India region. (Note: Names of the layers in the column "Layer (type)", written in `typewriter` font, follow the Keras/TensorFlow nomenclature.)

Layer no.	Layer (type)	Kernel Size	Kernels	Activation (g)
1	`ConvLSTM2D`	(9, 9)	64	tanh
2	`ConvLSTM2D`	(5, 5)	32	tanh
3	`Conv3D`	(7, 7, 7)	32	tanh
4	`Conv3D`	(5, 5, 5)	16	tanh
5	`Conv3D`	(3, 3, 3)	16	tanh
6	`Conv3D`	(1, 1, 1)	1	ReLU

Table 3.7: ConvLSTM architecture that gave us the best result for the Delhi-Punjab-Haryana region.

`ConvLSTM2D`[5] is an inbuilt layer in Keras consisting of convolutional LSTM units, whereas `Conv3D`[6] is another inbuilt layer in Keras that performs 3D convolution.

3.3.2 Hyperparameter Tuning

We carried out hyperparameter tuning based on the model's performance on the validation set. We tuned the following parameters.

[5] https://keras.io/api/layers/recurrent_layers/conv_lstm2d/
[6] https://keras.io/api/layers/convolution_layers/convolution3d/

Layer no.	Layer (type)	Kernel Size	Kernels	Activation (g)
1	`ConvLSTM2D`	(9, 9)	64	tanh
2	`ConvLSTM2D`	(7, 7)	64	tanh
3	`ConvLSTM2D`	(5, 5)	64	tanh
4	`Conv3D`	(7, 7, 7)	64	tanh
5	`Conv3D`	(7, 7, 7)	64	tanh
6	`Conv3D`	(5, 5, 5)	32	tanh
7	`Conv3D`	(5, 5, 5)	32	tanh
8	`Conv3D`	(3, 3, 3)	16	tanh
9	`Conv3D`	(3, 3, 3)	16	tanh
10	`Conv3D`	(1, 1, 1)	1	ReLU

Table 3.8: ConvLSTM architecture that gave us the best result for the north-east Indian subcontinent region.

Layer no.	Layer (type)	Kernel Size	Kernels	Activation (g)
1	`ConvLSTM2D`	(9, 9)	64	tanh
2	`ConvLSTM2D`	(5, 5)	32	tanh
3	`Conv3D`	(7, 7, 7)	32	ReLU
4	`Conv3D`	(5, 5, 5)	16	ReLU
5	`Conv3D`	(3, 3, 3)	16	ReLU
6	`Conv3D`	(1, 1, 1)	1	ReLU

Table 3.9: ConvLSTM architecture that gave us the best result for the central India region.

Number of `ConvLSTM2D` and `Conv3D` layers

We observed that having the number of `ConvLSTM2D` layers to around 2 to 4 gave a good model performance. In contrast, increasing it further decreased the model performance. We observed a similar situation for the `Conv3D` layers, where 4 to 7 `Conv3D` layers seemed to be the sweet spot.

Number of Kernels

We observed that having more kernels in `ConvLSTM2D` layers improved the model performance. Having more kernels increases computational complexity. After a point, having more kernels does not substantially improve the

model performance at the cost of computational complexity. Hence, we decided to put a decreasing order of the number of kernels with depth.

Kernel Size

Kernel size refers to the size of the kernel used over an image (see figure 2.1.12). Generally, a larger kernel captures more spatial information in an image but, on the other hand, also leads to a more considerable training time. We observed that using a smaller kernel like 3×3 for `ConvLSTM2D` or $3 \times 3 \times 3$ for `Conv3D` led to poorer model performance. At the same time, having a larger kernel like 9×9 or $9 \times 9 \times 9$ did not significantly improve the model at the cost of computational complexity. Hence, we decided to start with a larger kernel and decrease the size for the deeper layers.

Activation Functions

The activation function is one of the most crucial hyperparameters for any neural network. The activation function for the output layer depends upon the task at hand. For example, in a task predicting probability, using sigmoid (σ) as the activation function makes more sense since the output of sigmoid lies between 0 and 1, which is the same as probability. The choice of activation function for the input and the hidden layers can be arbitrary. However, some activation functions work better with a particular layer (e.g., tanh works well with the `ConvLSTM2D` layer).

We tried various activation functions for the input and the hidden layers like tanh, ReLU, sigmoid, PReLU (Parametric ReLU), ELU (Exponential Linear Unit), etc. We found that for the `ConvLSTM2D` layer, tanh gave us the best results. On the other hand, ReLU and tanh both worked well with the `Conv3D` layers.

Since the output for our model lies between 0 and 1, we had to choose the activation function for the output layer accordingly. We had two choices: sigmoid or ReLU. On trying both, we observed that ReLU worked better for our output layer.

Batch Normalization

Batch normalization is a technique in which we normalize the output of a particular layer before feeding it into the next layer. For many applications,

this reduces the learning time. However, we observed that not using batch normalization worked well for our problem.

Dropout

Dropout is a regularization technique in which we shut off outputs from some random neurons from a particular layer before feeding them into the next layer. Using dropout makes sure that the model does not overfit the training set. We observed that a dropout of 0.3[7] was the sweet spot for the `ConvLSTM2D` layers. It is better to avoid dropout for convolutional layers as it hinders the final matrix formation, which we also observed during training.

3.3.3 Model Compilation and Callbacks

Loss/Cost Function

Since we have a regression problem, we monitored mean squared error (MSE) and mean absolute error (MAE) during model training. In other words, our model calculated MAE and MSE between the ground truth ($\mathbf{Y}$) and the forecast $\left(\widehat{\mathbf{Y}}\right)$. We also used the `ModelCheckpoint`[8] callback from Keras to save our models.

Optimization Algorithm

We chose the AdaM (Adaptive Moment Estimation)[9] algorithm with an initial learning rate (α) of 10^{-4} as our optimization algorithm to optimize our loss/cost score. We also used the `ReduceLROnPlateau`[10] callback from Keras to decrease the learning rate whenever the loss/cost score plateaus.

Visualizing Model Training

We used the `TensorBoard`[11] callback from Keras to help us visualize the training process by plotting the loss/cost score with each iteration/epoch.

[7] A dropout of 0.3 means that the probability that a neuron in a given layer will shut off is 0.3.

[8] https://keras.io/api/callbacks/model_checkpoint/

[9] https://keras.io/api/optimizers/adam/

[10] https://keras.io/api/callbacks/reduce_lr_on_plateau/

[11] https://keras.io/api/callbacks/tensorboard/

Epochs

A single *epoch* is when the following steps happen consecutively and exactly once:

1. Calculation of the loss/cost score using the entire data,
2. Using the optimization algorithm to optimize the loss/cost score.

We repeat these two steps multiple times to improve the loss/cost score. In other words, we train our model for multiple epochs. We chose the number of epochs to be 1000. However, in most cases, the model got trained (i.e., the loss/cost score plateaued) within 500 epochs.

3.4 Data Preprocessing Pipeline

We tried many combinations of data transformation and data scaling techniques discussed in subsections 3.2.6 and 3.2.7. Figure 3.4.1 shows a flowchart of the entire data preprocessing pipeline.

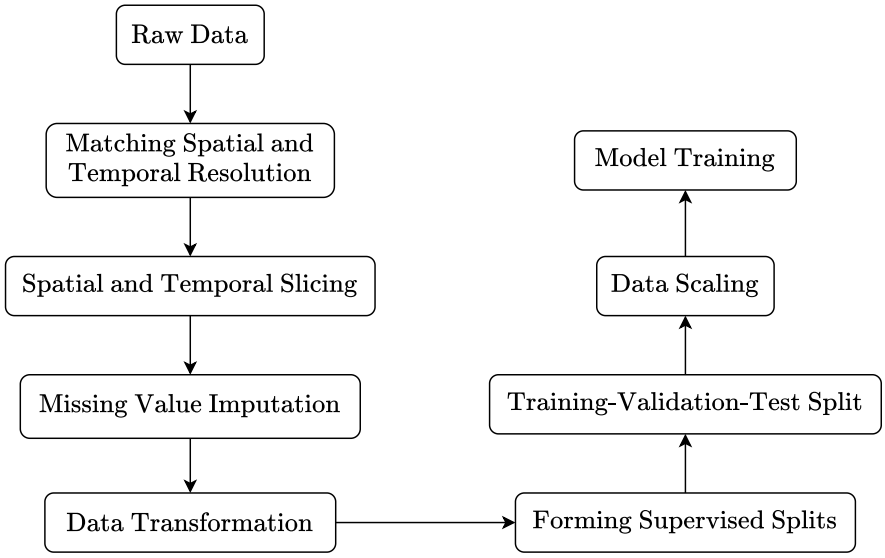


Figure 3.4.1: A flowchart of the data preprocessing pipeline.

Since our data had a right-skewed distribution, we had to go ahead with only either log transformation or with power transformation with $\gamma < 1$.

We chose a different pair of data transformation and data scaling techniques for each variable. Table 3.10 summarizes these choices. Subsection 3.4.1 mentions the reasons for choosing these pairs of data transformation and data scaling techniques.

Variable	Data Transformation	Data Scaling
PM2.5	$\log(x+1)$	Per-day Normalization
NDVI	None	Normalization
Temperature	None	Normalization
Surface Pressure	None	Normalization
Wind (u, v)	None	Normalization
Total Cloud Cover	None	None

Table 3.10: Data Transformation and Data Scaling for each variable.

3.4.1 Choosing the Data Transformation and Data Scaling Pair

PM2.5

The primary reason behind selecting the $\log(x+1)$ transform over the $\log(x)$ transform is the presence of zeros in PM2.5 data. On the other hand, the primary reason behind selecting the pair "$\log(x+1)$" and "Per-day Normalization" was to improve the distribution of the data. The raw PM2.5 data distribution has an extreme right skew. No other pair improved the right skew compared to this. Per-day normalization helped scale this distribution between 0 and 1, which is optimum for deep learning.

NDVI

The NDVI data already had a pretty normal-looking distribution. Hence, we did not use any data transformation. On the other hand, the range of NDVI is between -1 and 1, which is why we used per-day normalization to scale it between 0 and 1.

Temperature

Temperature also had a pretty normal-looking distribution, so we did not use any data transformation. It ranged between 220 K to 320 K. Hence, we used per-day normalization to scale it between 0 and 1.

Surface Pressure

Surface pressure also had a pretty normal-looking distribution, so we did not use any data transformation. It ranged between 50 000 Pa to 110 000 Pa. Hence, we used per-day normalization to scale it between 0 and 1.

Wind (u, v)

Wind (u, v) also had a pretty normal-looking distribution, so we did not use any data transformation. It ranged between $-30\,\mathrm{m\,s^{-1}}$ to $30\,\mathrm{m\,s^{-1}}$. Hence, we used per-day normalization to scale it between 0 and 1.

Total Cloud Cover

Total cloud cover did not have any significant skew in the data. Hence, using a data transformation did not help significantly. It already ranged between 0 and 1, so we did not use any data scaling technique.

3.5 Total Models to Train

As mentioned in section 3.1, we have five variables in addition to PM2.5, with PM2.5 being the variable we want to forecast. This means that we have 32 combinations of variables with PM2.5 since

$$\binom{5}{0} + \binom{5}{1} + \binom{5}{2} + \binom{5}{3} + \binom{5}{4} + \binom{5}{5} = 2^5 = 32$$

In other words, we have 32 models to train. We train these 32 models for the three regions of interest, i.e., Delhi-Punjab-Haryana, the northeast Indian subcontinent, and central India. So, we have a total of $32 \times 3 = 96$ models to train.

Chapter 4

Results and Discussion

4.1 Metrics

We evaluate our model based on mean squared error (MSE) (refer subsubsection 2.1.4) and Pearson correlation coefficient.

4.1.1 Pearson Correlation Coefficient

Pearson correlation coefficient (r) is a metric that shows the relationship between two arrays. The following equation gives the Pearson correlation coefficient between two 1-dimensional arrays, x and y.

$$r = \frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_i (x_i - \bar{x})^2 \sum_i (y_i - \bar{y})^2}} \tag{4.1.1}$$

here, $\bar{x}$ and $\bar{y}$ denote the mean of the arrays x and y, respectively.

Correlation (r) ranges between -1 and 1.

- If $r < 0$, we say that the arrays are negatively correlated.
- If $r > 0$, we say that the arrays are positively correlated.
- If $r \approx 0$, we say that the arrays are not correlated.

Correlation Coefficient between two n-dimensional Arrays

To find the correlation between two n-dimensional arrays, we flatten[1] them into 1-dimensional arrays and then use equation 4.1.1.

As mentioned earlier, we forecast up to three days into the future. The obtained forecast and its corresponding ground truth are 5-dimensional arrays of dimensions (Samples $\times$ Forecasted Days $\times$ Latitude $\times$ Longitude $\times$ 1[2]), which can further split into three 4-dimensional arrays of dimensions (Samples $\times$ Latitude $\times$ Longitude $\times$ 1), corresponding to each of the three days. So, we have six 4-dimensional arrays per sample, three for the ground truth and the remaining three for the forecast. We flatten them and find the correlation between the corresponding arrays for the first, second, and third days. We get a single value of correlation per sample. So, we get a distribution of correlation values for all the available samples. We plot this distribution for better visualization.

Spatial Correlation between two Arrays

Let $\mathbf{G}$ and $\mathbf{P}$ denote the ground truth and the predicted arrays for a particular day having dimensions (Samples $\times$ Latitude $\times$ Longitude $\times$ 1). We have two time series at each (Latitude, Longitude) pair, one from $\mathbf{G}$ and the other from $\mathbf{P}$. We find the correlation coefficient between these two time series. We repeat this for all values of latitudes and longitudes and form a matrix with dimensions (Latitude $\times$ Longitude) with correlation values corresponding to each (Latitude, Longitude) pair. This matrix, known as *spatial correlation*, shows how the two arrays, $\mathbf{G}$ and $\mathbf{P}$, are related in time. In this way, we compare the predictions of our model with the ground truth. We will plot these two plots, i.e., correlation distribution and the spatial correlation for all the day 1's, day 2's and day 3's, for the three regions of interest.

[1] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.flatten.html

[2] We have a 1 here since we are only forecasting one variable, i.e., PM2.5.

4.2 Delhi-Punjab-Haryana

4.2.1 Comparison with the Ground Truth

For the Delhi-Punjab-Haryana region, the best-performing variable combination was with the two variables PM2.5 and wind (u, v). Figure 4.2.1 compares the model forecast with the ground truth for a particular three consecutive days.

Observations

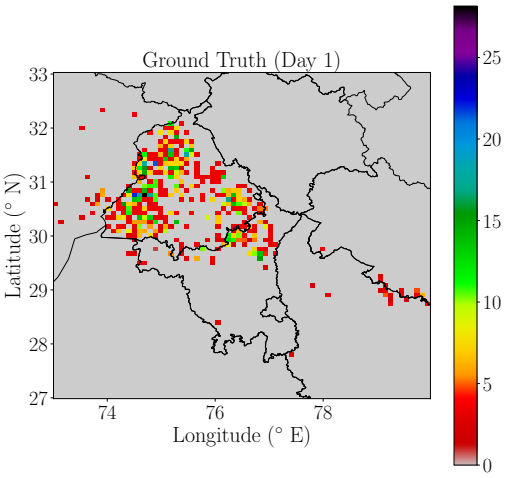
- The model has captured the spatial pattern of PM2.5 emissions over the Punjab region.
- PM2.5 emissions are denser on the first and the last days, whereas they are sparser on the second. The model captures this pattern well and gives the forecast accordingly. In other words, the model has also captured the temporal pattern in the data.
- The forecast looks like a smoother version of the ground truth. In other words, the model does not capture the sudden high-intensity peaks. This smoothening is quite a common issue with all the spatiotemporal forecasting models, including those more complicated than convolutional LSTM.

4.2.2 Correlation Distribution

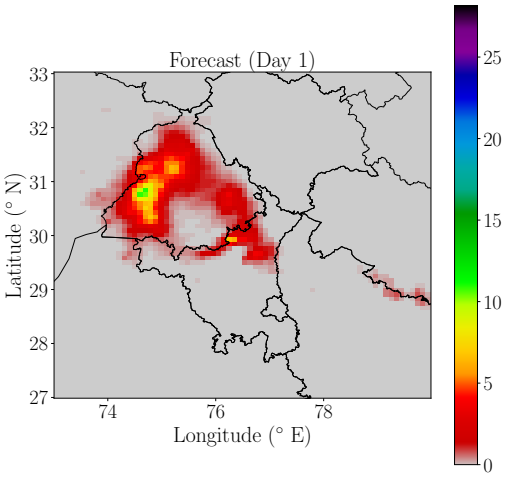
Figure 4.2.2 shows the correlation distribution for all sets of three consecutive days.

Observations

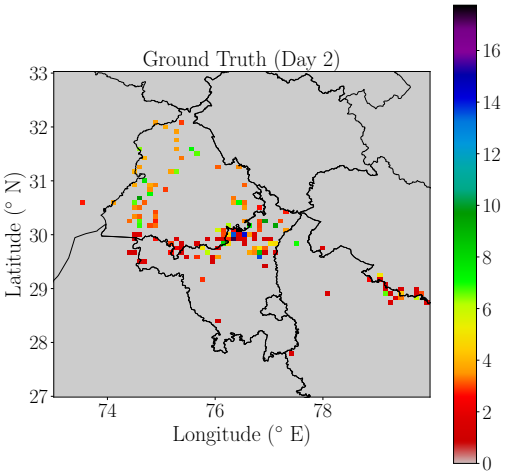
- The correlation distribution plot has the most right shift for the first day, an intermediate right shift for the second, and the least relative right shift for the third. This decreasing forecast accuracy for future time steps is typical in time series forecasting.



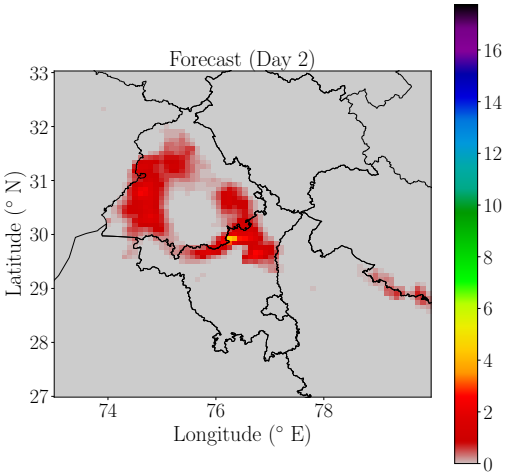
(a) Ground Truth (Day 1).



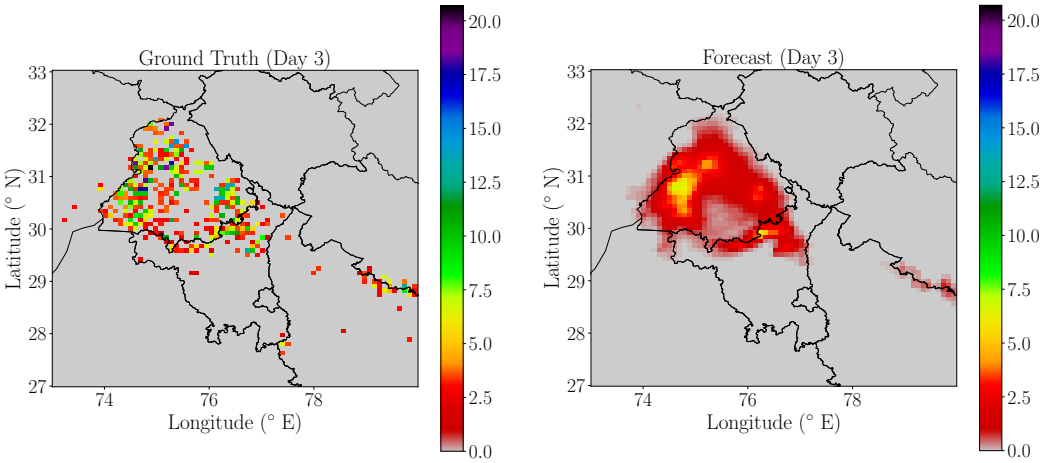
(b) Forecast (Day 1).



(c) Ground Truth (Day 2).



(d) Forecast (Day 2).



(e) Ground Truth (Day 3).

(f) Forecast (Day 3).

Figure 4.2.1: Ground Truth vs. Forecast for the Delhi-Punjab-Haryana region.

4.2.3 Spatial Correlation

Figure 4.2.3 shows the spatial correlation for all sets of three consecutive days.

Observations

- Most of the points on day 1 in Punjab have a correlation value of 0.5 to 0.7.
- Correlation values decrease with the coming days.

4.2.4 Correlation Distribution for Different Combinations of Variables on Day 1

Figure 4.2.4 shows the correlation distribution plot corresponding to the best combination of 0, 1, 2, 3, 4, and 5 variables in addition to PM2.5 on the first day.

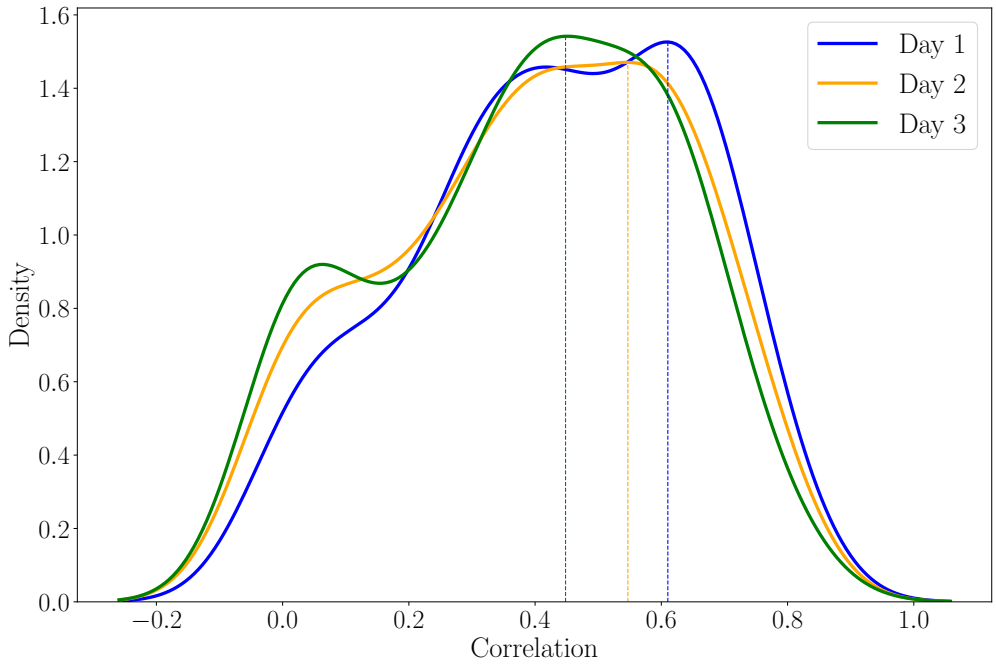


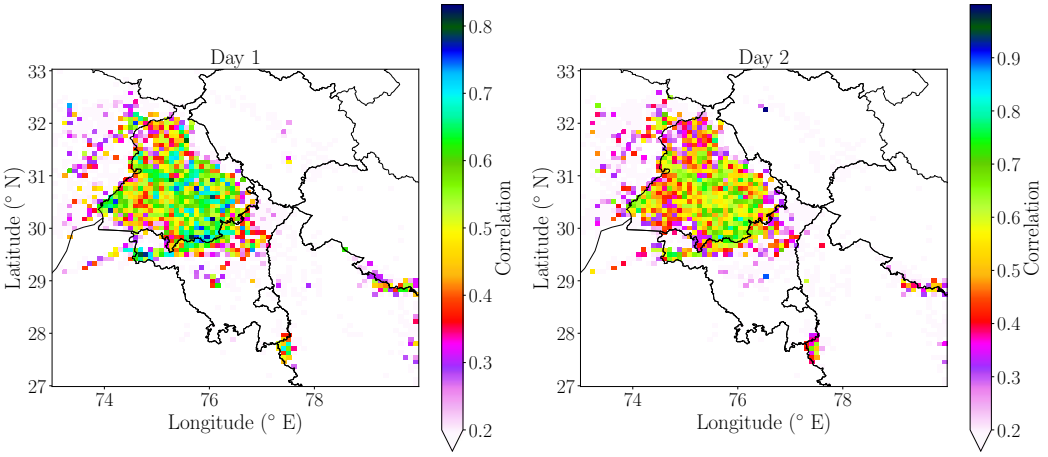
Figure 4.2.2: Correlation distribution plot for the Delhi-Punjab-Haryana region.

Observations

- The two-variable model consisting of PM2.5 and wind gave us the best (most right-shifted) correlation distribution.
- We can infer that wind is the most crucial variable in forecasting PM2.5 emitted due to stubble burning in the Punjab-Haryana region, followed by variables like Surface Pressure, NDVI, Temperature, and Total Cloud Cover.

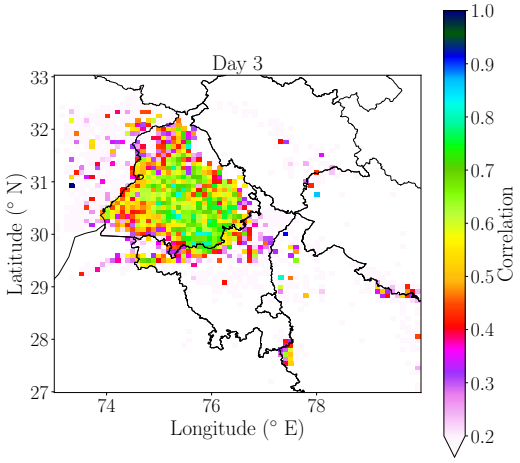
4.2.5 Normalized Mean Squared Error vs. Number of Epochs

Figure 4.2.5 shows the decrease in normalized mean squared error with the number of epochs.



(a) Spatial Correlation (Day 1).

(b) Spatial Correlation (Day 2).



(c) Spatial Correlation (Day 3).

Figure 4.2.3: Spatial Correlation for all the day 1's, day 2's, and day 3's in the Delhi-Punjab-Haryana region.

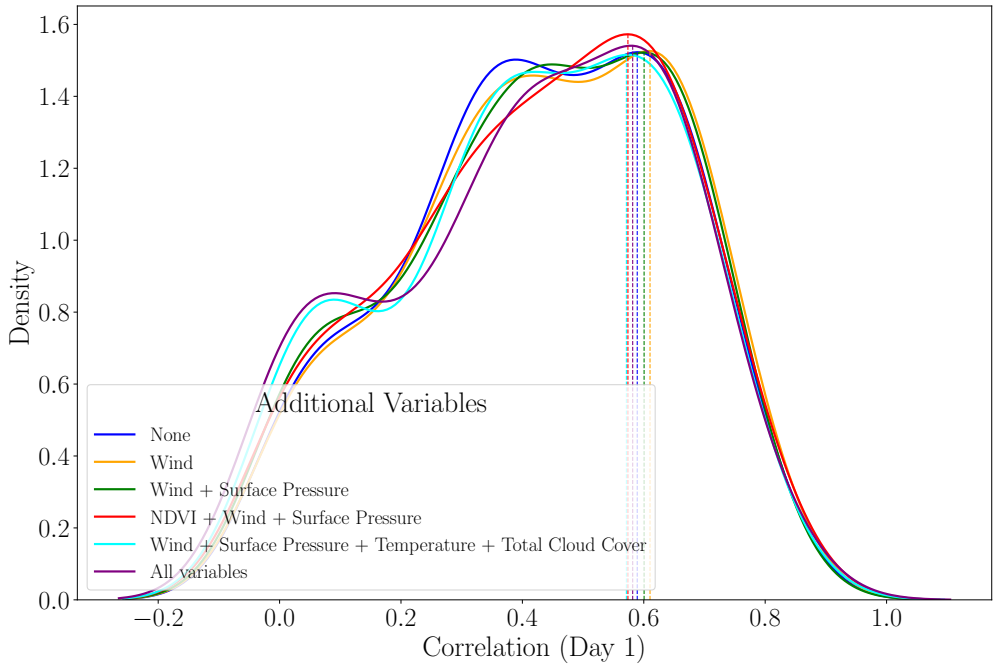


Figure 4.2.4: Correlation distribution plot corresponding to the best combination of 0, 1, 2, 3, 4, and 5 variables in addition to PM2.5 for the Delhi-Punjab-Haryana region on day 1.

Observations

- The normalized mean squared error decreases with every epoch till it converges to a particular value at around epoch 140.

4.3 Northeast Indian Subcontinent

4.3.1 Comparison with the Ground Truth

For the northeast Indian subcontinent, adding additional variables did improve the model performance, whereas performance between models corresponding to different combinations of variables was comparable. Figure 4.3.1 compares the model forecast with the ground truth for a particular three consecutive days.

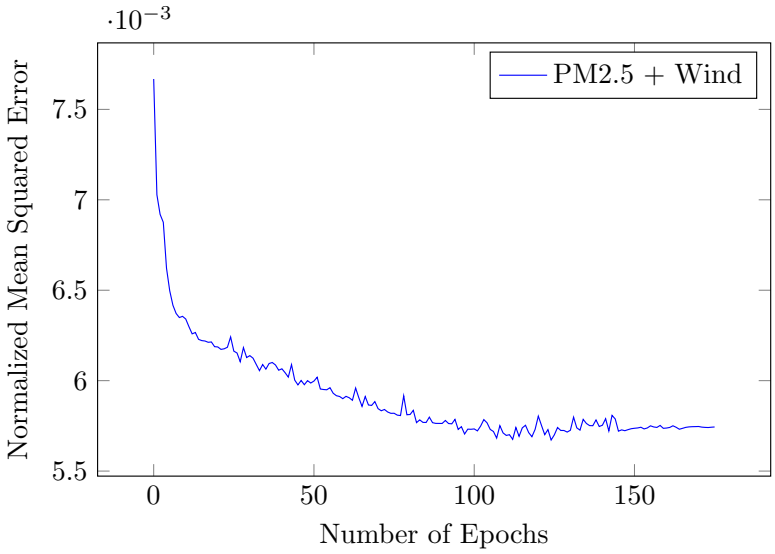


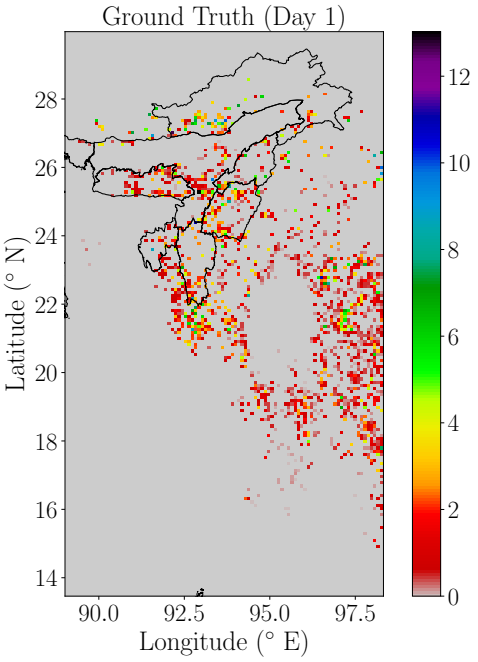
Figure 4.2.5: Normalized mean squared error vs. epochs plot for the "PM2.5 + Wind" model (Delhi-Punjab-Haryana region).

Observations

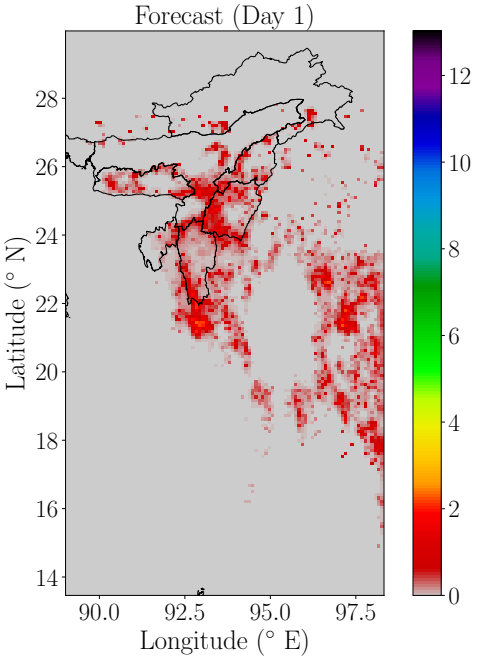
- The model has captured the spatial pattern of PM2.5 emissions over the northeast Indian subcontinent region. The model has captured perfectly a region of no PM2.5 emissions in the center-right part of the map.
- The forecast is sharper on the first day with relatively higher peaks and gets smoother/blurrier for the following days. The intensity of the forecast also decreases for the following days. Again, this smoothening is quite a common issue with all spatiotemporal forecasting models.

4.3.2 Correlation Distribution

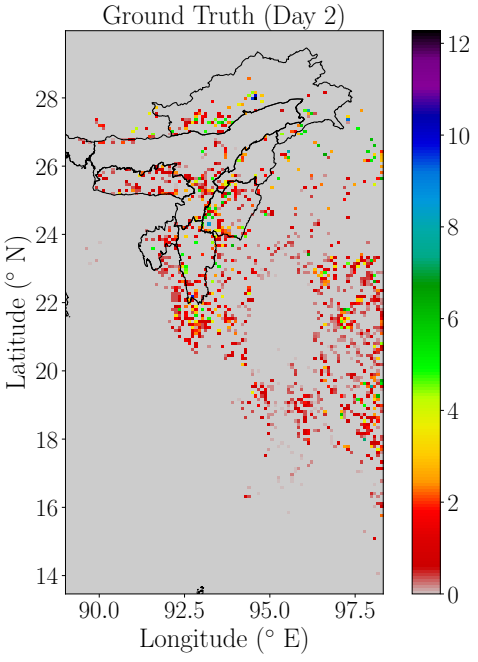
Figure 4.3.2 shows the correlation distribution for all sets of three consecutive days.



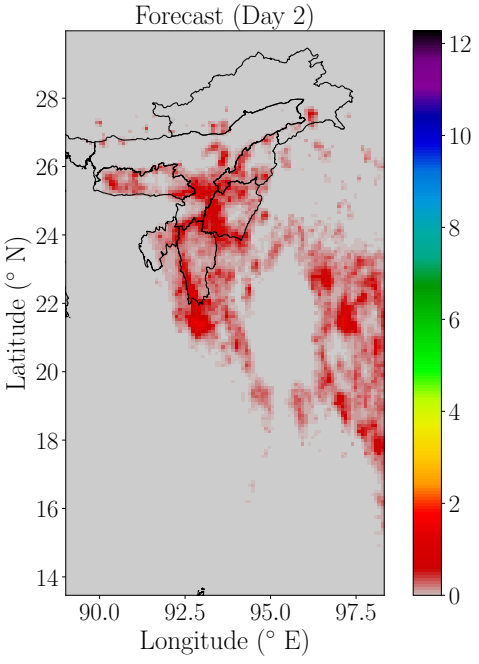
(a) Ground Truth (Day 1).



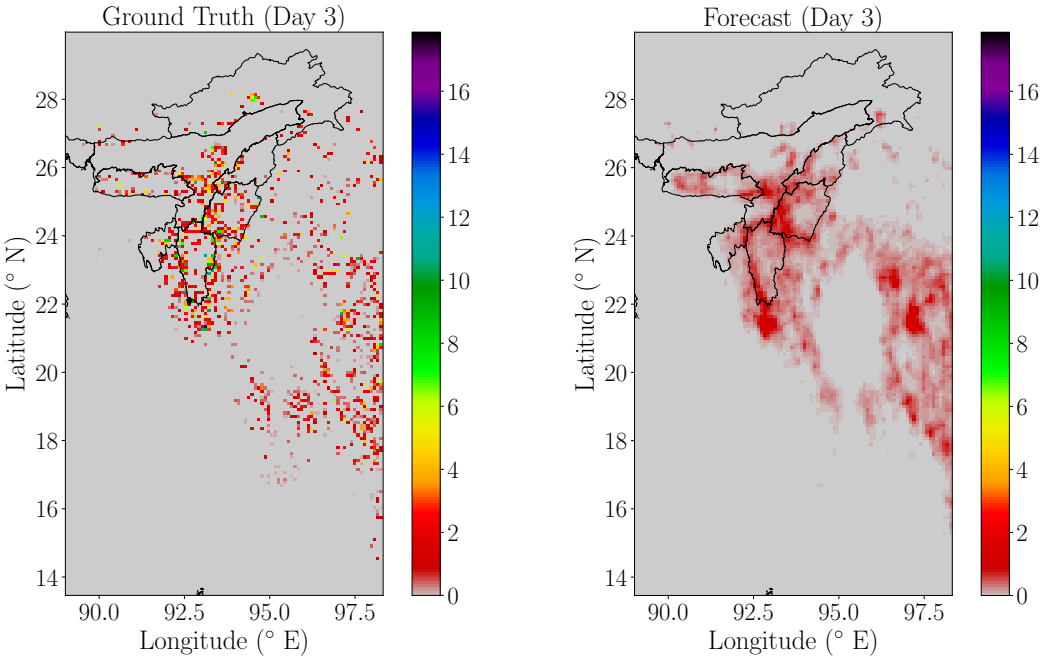
(b) Forecast (Day 1).



(c) Ground Truth (Day 2).



(d) Forecast (Day 2).



(e) Ground Truth (Day 3).

(f) Forecast (Day 3).

Figure 4.3.1: Ground Truth vs. Forecast for the northeast Indian subcontinent region.

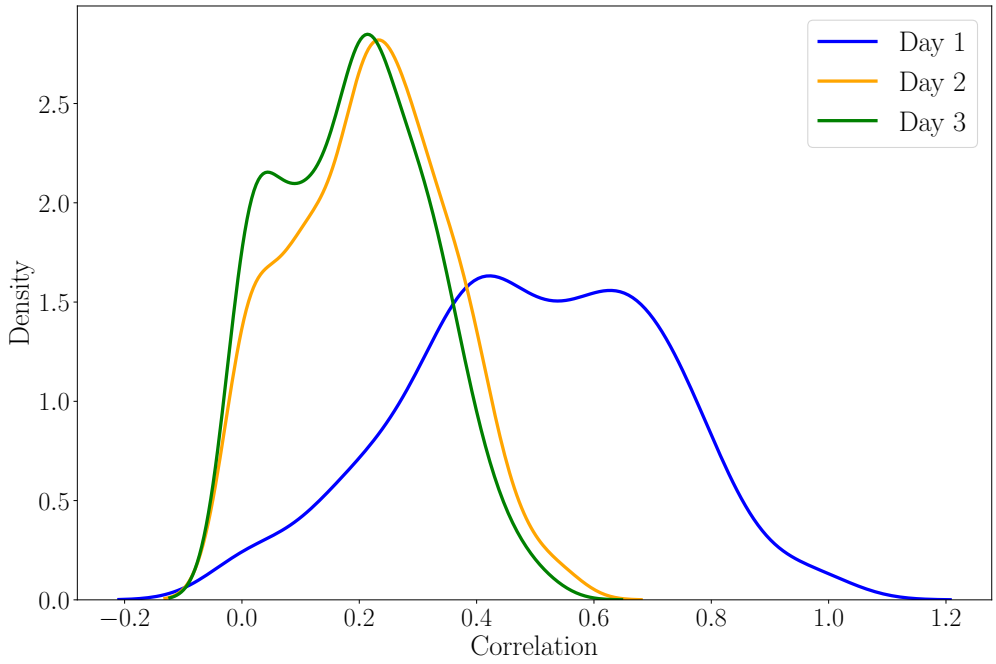


Figure 4.3.2: Correlation distribution plot for the northeast Indian subcontinent region.

Observations

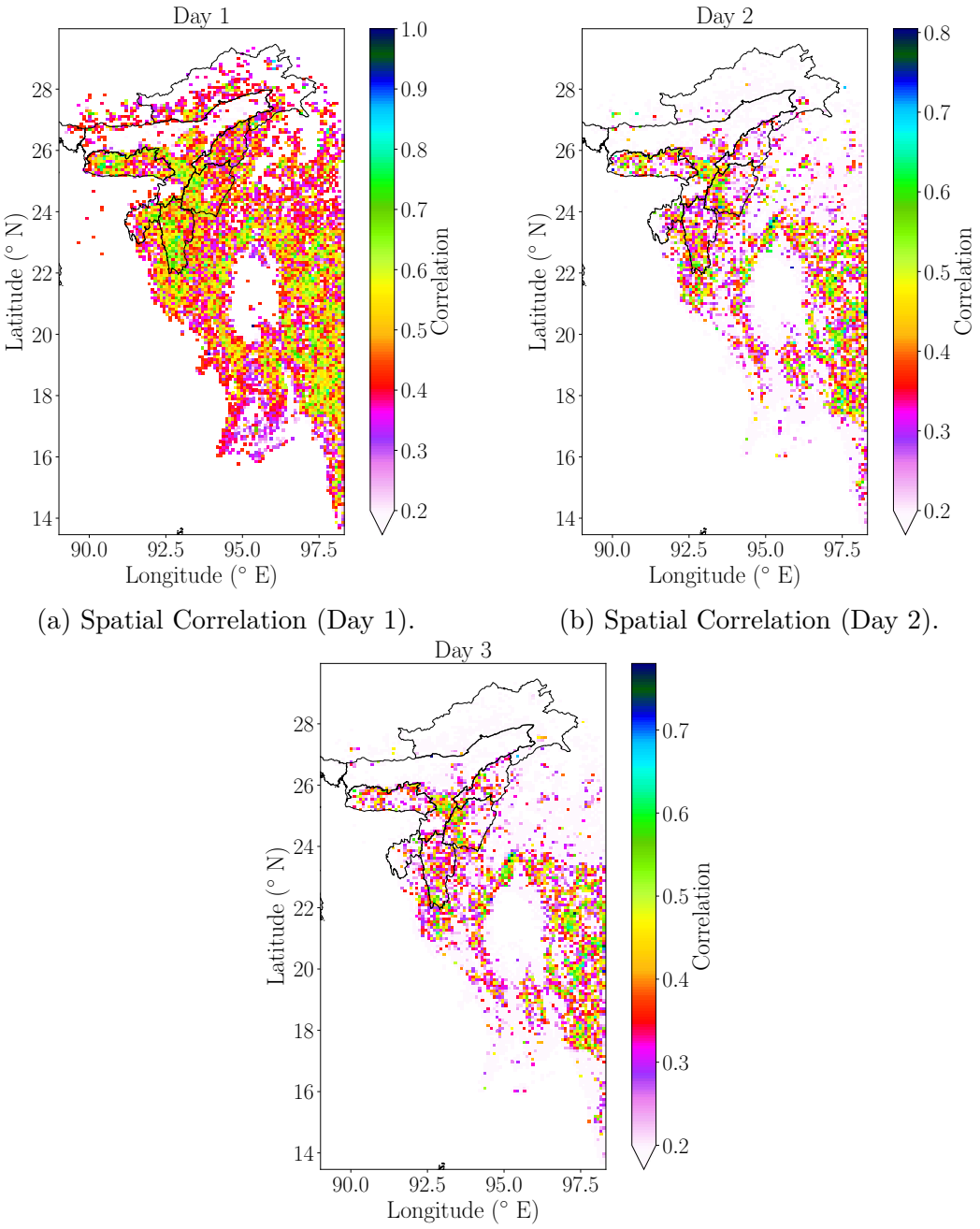
- The model gave a good correlation distribution for the first day. However, the correlation for the second and third days is relatively low. This is also evident from the ground truth vs. forecast plot (see figure 4.3.1), as the intensity of the forecast decreases for the second and third days.

4.3.3 Spatial Correlation

Figure 4.3.3 shows the spatial correlation for all sets of three consecutive days.

Observations

- On the first day, many points have a correlation lying in the range of 0.5 to 0.8.



(a) Spatial Correlation (Day 1).

(b) Spatial Correlation (Day 2).

(c) Spatial Correlation (Day 3).

Figure 4.3.3: Spatial Correlation for all the day 1's, day 2's, and day 3's in the northeast Indian subcontinent region.

- The density of such high correlation points decreases for the following days.

4.3.4 Correlation Distribution for Different Combinations of Variables on Day 1

Figure 4.3.4 shows the correlation distribution plot corresponding to the best combination of 0, 1, 2, 3, 4, and 5 variables in addition to PM2.5 on the first day.

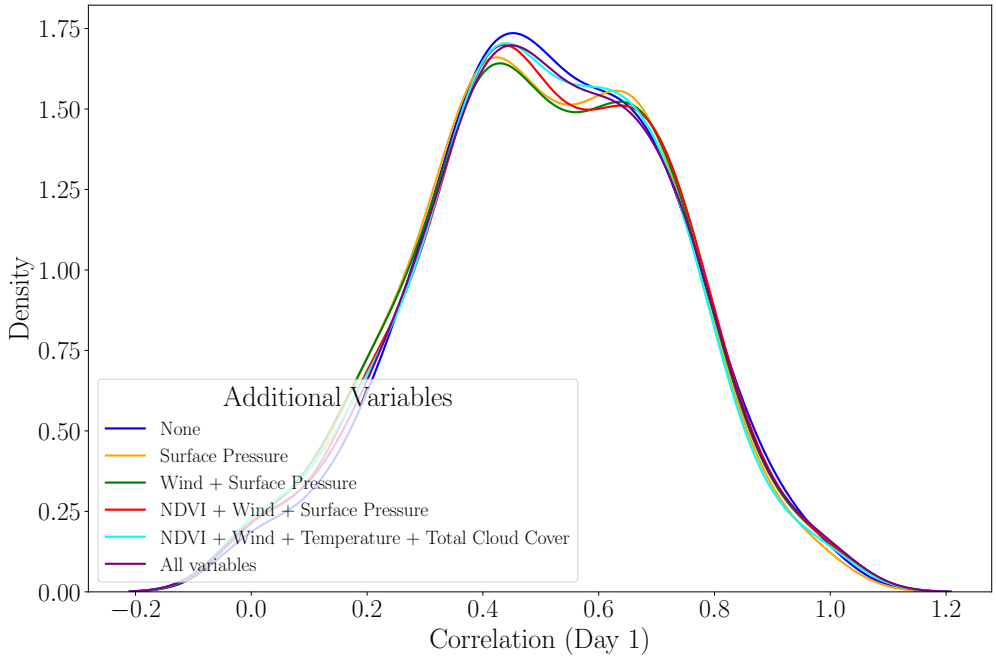


Figure 4.3.4: Correlation distribution plot corresponding to the best combination of 0, 1, 2, 3, 4, and 5 variables in addition to PM2.5 for the northeast Indian subcontinent region on day 1.

Observations

- All the models have a very similar performance. Most of the points have a correlation ranging between 0.4 and 0.8.

- The two-variable model consisting of PM2.5 and surface pressure, and the four-variable model consisting of PM2.5, NDVI, wind, and surface pressure seem to give the most right-shifted correlation distribution.

4.3.5 Normalized Mean Squared Error vs. Number of Epochs

Figure 4.3.5 shows the decrease in normalized mean squared error with the number of epochs.

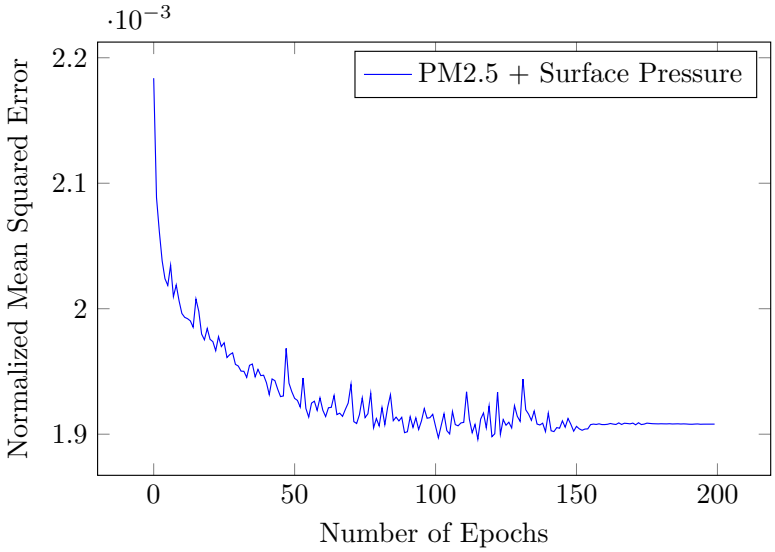


Figure 4.3.5: Normalized mean squared error vs. epochs plot for the "PM2.5 + Surface Pressure" model (Northeast Indian Subcontinent region).

Observations

- The normalized mean squared error decreases with every epoch till it converges to a particular value at around epoch 160.

4.4 Central India

4.4.1 Comparison with the Ground Truth

For the central Indian region, the best performing variable combination was with the four variables PM2.5, temperature, surface pressure, and total cloud cover. Figure 4.4.1 compares the model forecast with the ground truth for a particular three consecutive days.

Observations

- The model has captured the spatial pattern of PM2.5 emissions over the central Indian region, particularly the circular arc of PM2.5 emissions seen in the map. The forecast is very close to the ground truth for the first day.
- Just like in the case of the northeast Indian subcontinent, the forecast is sharper on the first day and gets smoother/blurrier for the following days, which is a common issue with all spatiotemporal forecasting models.

4.4.2 Correlation Distribution

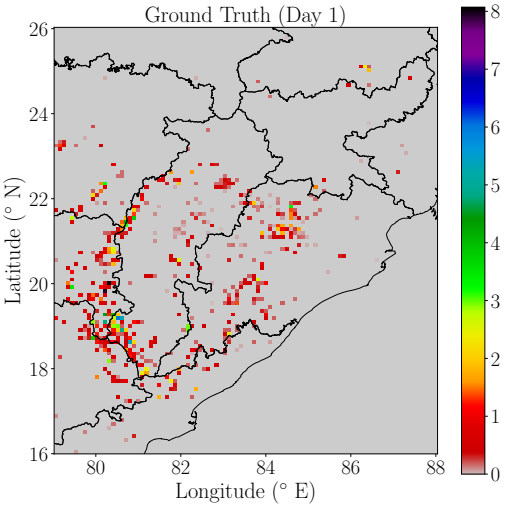
Figure 4.4.2 shows the correlation distribution for all sets of three consecutive days.

Observations

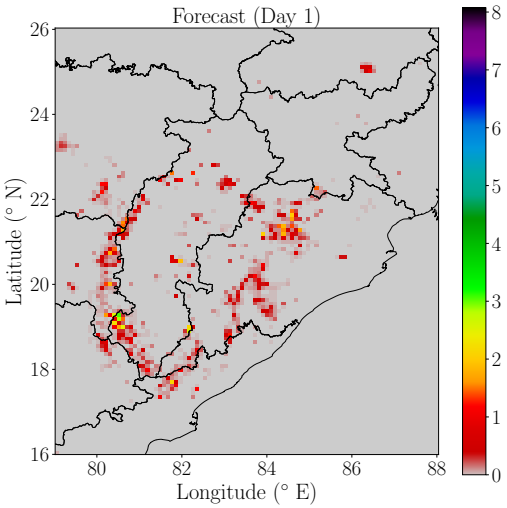
- Just like in the case of the northeast Indian subcontinent region, the model gave a good correlation distribution for the first day. However, the correlation for the second and third days is relatively low. This is also evident from the ground truth vs. forecast plot (see figure 4.4.1), as the intensity of the forecast decreases for the second and third days.

4.4.3 Spatial Correlation

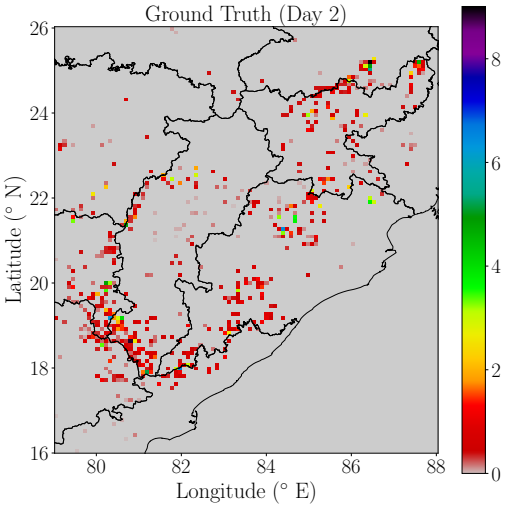
Figure 4.4.3 shows the spatial correlation for all sets of three consecutive days.



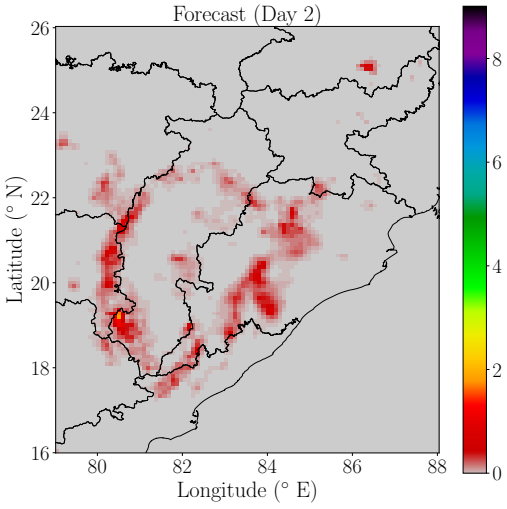
(a) Ground Truth (Day 1).



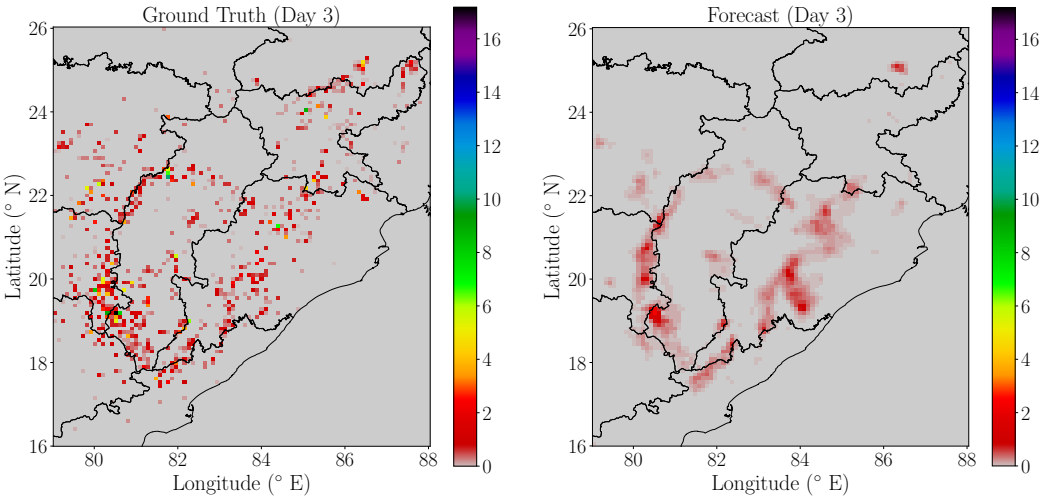
(b) Forecast (Day 1).



(c) Ground Truth (Day 2).



(d) Forecast (Day 2).



(e) Ground Truth (Day 3). (f) Forecast (Day 3).

Figure 4.4.1: Ground Truth vs. Forecast for the central Indian region.

Observations

- On the first day, many points have a correlation lying in the range of 0.4 to 0.6. For the following days, the density of such high correlation points decreases.

4.4.4 Correlation Distribution for Different Combinations of Variables on Day 1

Figure 4.4.4 shows the correlation distribution plot corresponding to the best combination of 0, 1, 2, 3, 4, and 5 variables in addition to PM2.5 on the first day.

Observations

- All models, excluding the univariate PM2.5 model, have a similar performance. Most of the points have a correlation ranging between 0.4 and 0.8.
- The four variable model corresponding to the variables PM2.5, temperature, surface pressure, and total cloud cover has the most right shift and a higher second peak.

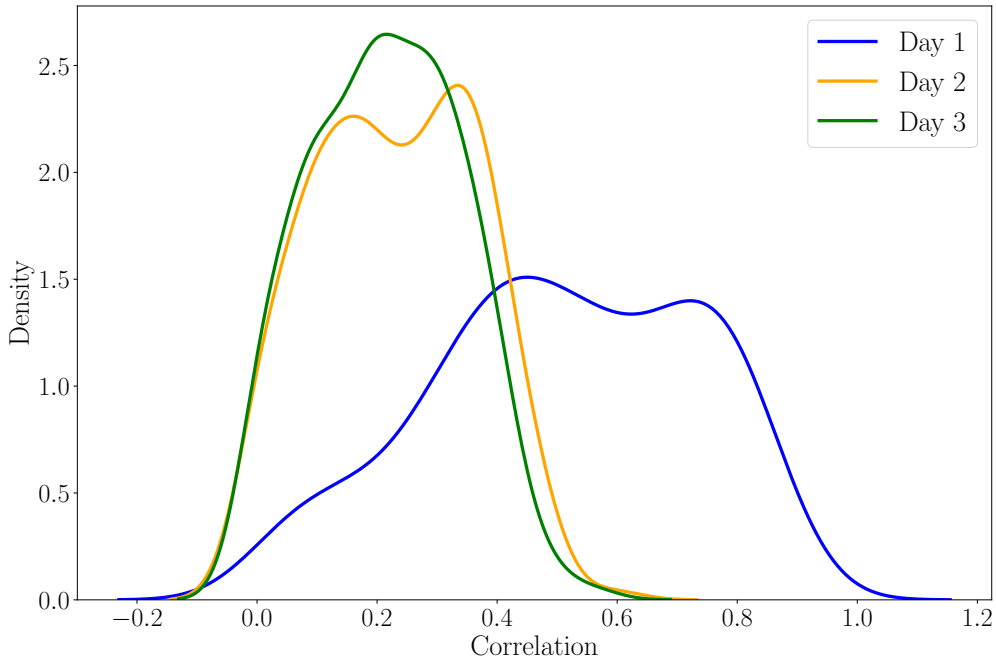


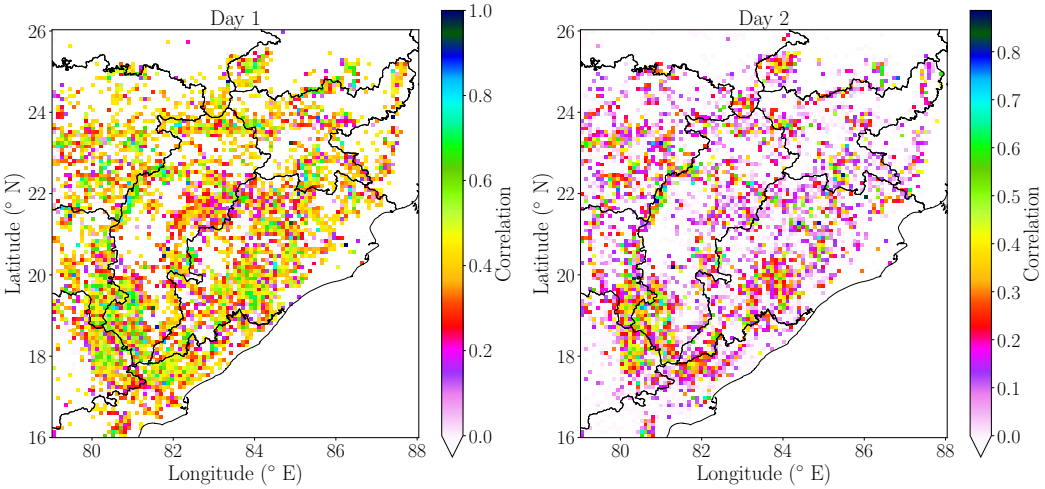
Figure 4.4.2: Correlation distribution plot for the central Indian region.

4.4.5 Normalized Mean Squared Error vs. Number of Epochs

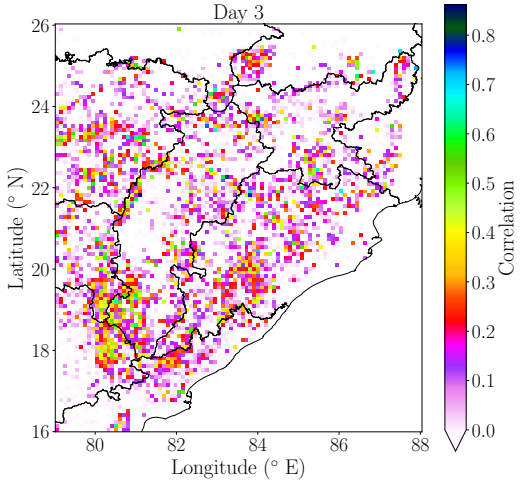
Figure 4.4.5 shows the decrease in normalized mean squared error with the number of epochs.

Observations

- The normalized mean squared error decreases with every epoch till it converges to a particular value at around epoch 100.



(a) Spatial Correlation (Day 1). (b) Spatial Correlation (Day 2).



(c) Spatial Correlation (Day 3).

Figure 4.4.3: Spatial Correlation for all the day 1's, day 2's, and day 3's in the central Indian region.

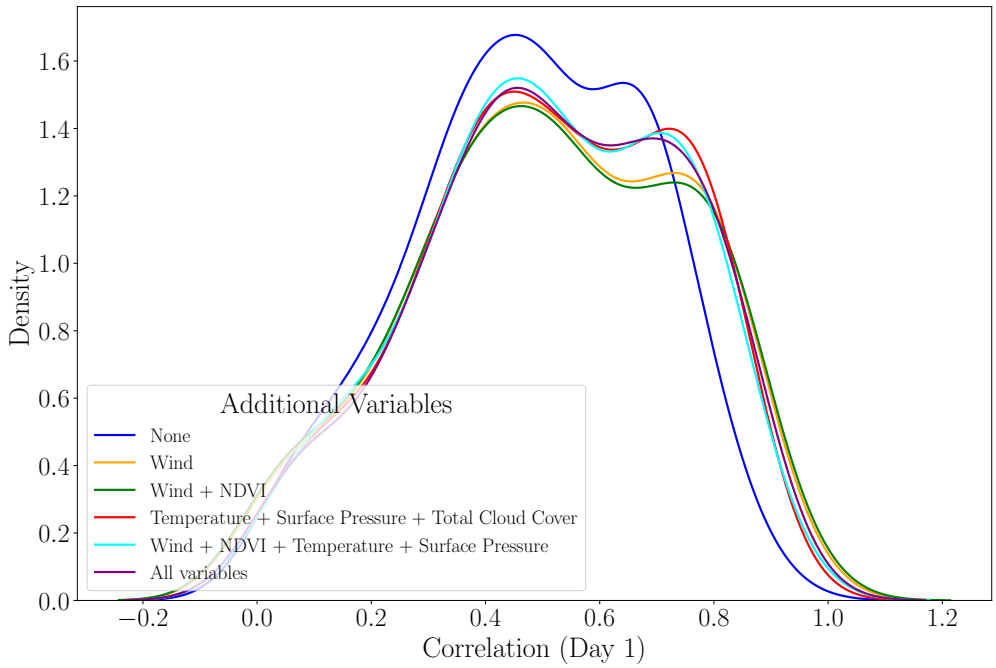


Figure 4.4.4: Correlation distribution plot corresponding to the best combination of 0, 1, 2, 3, 4, and 5 variables in addition to PM2.5 for the central Indian region on day 1.

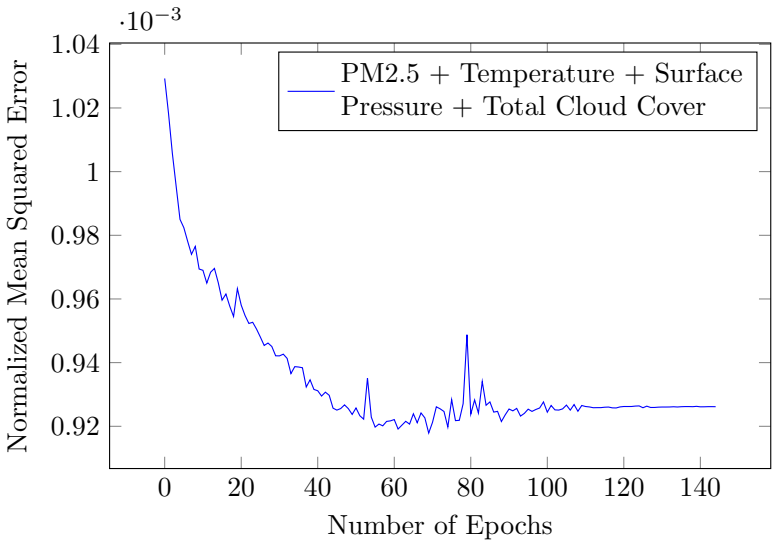


Figure 4.4.5: Normalized mean squared error vs. epochs plot for the "PM2.5 + Temperature + Surface Pressure + Total Cloud Cover" model (Central Indian region).

Chapter 5

Conclusion

We applied ConvLSTM for forecasting PM2.5 emissions for up to 3 days. Our goal was to forecast PM2.5 emissions due to stubble burning in the Punjab-Haryana-Delhi area and forest fires in the central and northeast Indian subcontinent. We obtained a good result for the Punjab-Haryana-Delhi area. On the other hand, for the central and the northeast Indian subcontinent, we obtained a good result on the first day. Our model captured the spatial pattern of PM2.5 emissions for all three regions of interest. In other words, our model can reasonably accurately predict the location (i.e., the coordinates) where the burning activity occurs, as evident from the ground truth vs. forecast plots. However, our model could not predict the intensity accurately, i.e., the forecast was blurry, particularly for the second and third days. We can take many steps to improve this result. Section 5.1 elaborates on this. So, ConvLSTM can forecast spatiotemporal sequences in climate science, particularly for short-term forecasting. Figure 5.1.1 shows a flowchart of the conclusion.

5.1 Future Work

- Our study uses an older version of FINN data with a temporal extent from 2002 to 2018. A newer version of FINN data is now available with a temporal extent till 2021. Including it in the model can help improve performance.
- We can use better methods to match the spatial resolution of the data (see subsection 3.2.1) than linear interpolation. One of the best meth-

ods to do the same is to apply SRGAN (Super Resolution Generative Adversarial Networks) (Ledig et al., 2017), a deep learning-based method for resolution enhancement.

- We can use more complicated deep learning-based spatiotemporal forecasting models like convolutional GRU (Ballas, Yao, Pal, & Courville, 2015) or trajectory GRU (Shi et al., 2017).
- We can also try custom/hybrid models to improve the forecast accuracy.
- We can try hyperparameter tuning of the model, like trying different architectures, number of layers, custom loss functions, learning rate, regularization methods, and skip connections. Custom loss functions or loss functions designed to work with spatiotemporal data may improve the blurry forecast.

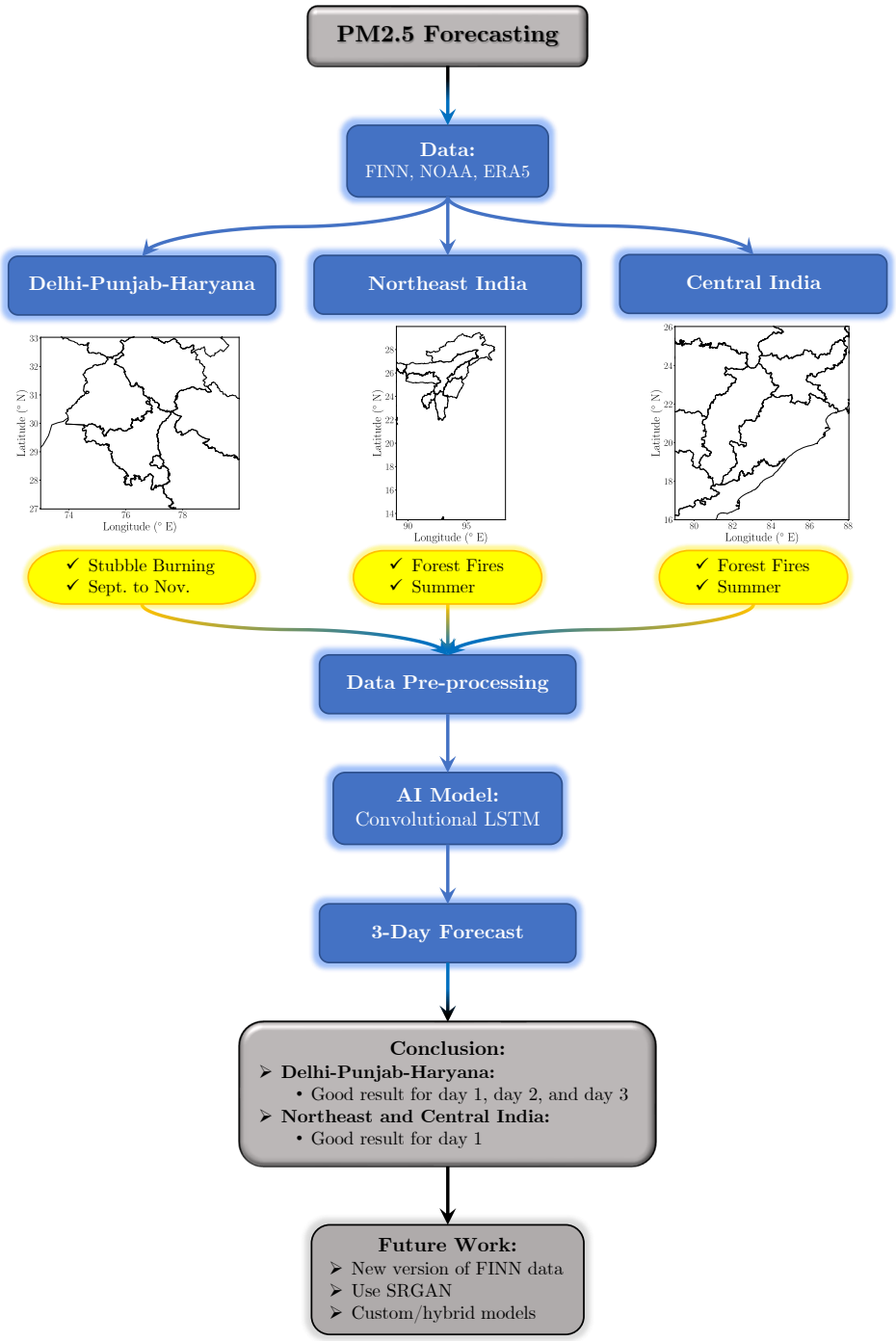


Figure 5.1.1: A flowchart of the conclusion.

References

Ballas, N., Yao, L., Pal, C., & Courville, A. (2015). Delving Deeper into Convolutional Networks for Learning Video Representations. *arXiv preprint arXiv:1511.06432*. https://arxiv.org/abs/1511.06432.

Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014). Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. *arXiv preprint arXiv:1406.1078*. https://arxiv.org/abs/1406.1078.

Chollet, F. (2021). *Deep learning with Python*. Simon and Schuster. https://www.manning.com/books/deep-learning-with-python.

Harishkumar, K., Yogesh, K., Gad, I., et al. (2020). Forecasting Air Pollution Particulate Matter ($PM_{2.5}$) Using Machine Learning Regression Models. *Procedia Computer Science*, *171*, 2057–2066. https://www.sciencedirect.com/science/article/pii/S1877050920312060.

Hersbach, H., Bell, B., Berrisford, P., Hirahara, S., Horányi, A., Muñoz-Sabater, J., ... others (2020). The ERA5 global reanalysis. *Quarterly Journal of the Royal Meteorological Society*, *146*(730), 1999–2049. https://rmets.onlinelibrary.wiley.com/doi/10.1002/qj.3803.

Hochreiter, S., & Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation*, *9*(8), 1735–1780. https://ieeexplore.ieee.org/abstract/document/6795963.

Huang, C.-J., & Kuo, P.-H. (2018). A Deep CNN-LSTM Model for Particulate Matter ($PM_{2.5}$) Forecasting in Smart Cities. *Sensors*, *18*(7), 2220. https://www.mdpi.com/1424-8220/18/7/2220.

Karimian, H., Li, Q., Wu, C., Qi, Y., Mo, Y., Chen, G., ... others (2019). Evaluation of Different Machine Learning Approaches to Forecasting PM2.5 Mass Concentrations. *Aerosol and Air Quality Research*, *19*(6), 1400–1410. https://aaqr.org/articles/aaqr-18-12-oa-0450.

Ledig, C., Theis, L., Huszár, F., Caballero, J., Cunningham, A., Acosta,

A., ... others (2017). Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 4681–4690). https://arxiv.org/abs/1609.04802.

Olah, C. (2015). Understanding LSTM Networks. https://colah.github.io/posts/2015-08-Understanding-LSTMs/.

Shi, X., Chen, Z., Wang, H., Yeung, D.-Y., Wong, W.-K., & Woo, W.-c. (2015). Convolutional LSTM Network: A Machine Learning Approach for Precipitation Nowcasting. *Advances in Neural Information Processing Systems*, *28*. https://arxiv.org/abs/1506.04214.

Shi, X., Gao, Z., Lausen, L., Wang, H., Yeung, D.-Y., Wong, W.-k., & Woo, W.-c. (2017). Deep Learning for Precipitation Nowcasting: A Benchmark and A New Model. *Advances in Neural Information Processing Systems*, *30*. https://arxiv.org/abs/1706.03458.

Vermote, E., et al. (2019). NOAA Climate Data Record (CDR) of AVHRR Normalized Difference Vegetation Index (NDVI), Version 5. *NOAA CDR Program, NOAA National Centers for Environmental Information*. https://www.ncei.noaa.gov/products/climate-data-records/normalized-difference-vegetation-index.

Wiedinmyer, C., Akagi, S., Yokelson, R. J., Emmons, L., Al-Saadi, J., Orlando, J., & Soja, A. (2011). The Fire INventory from NCAR (FINN): A high resolution global model to estimate the emissions from open burning. *Geoscientific Model Development*, *4*(3), 625–641. https://gmd.copernicus.org/articles/4/625/2011/gmd-4-625-2011.html.