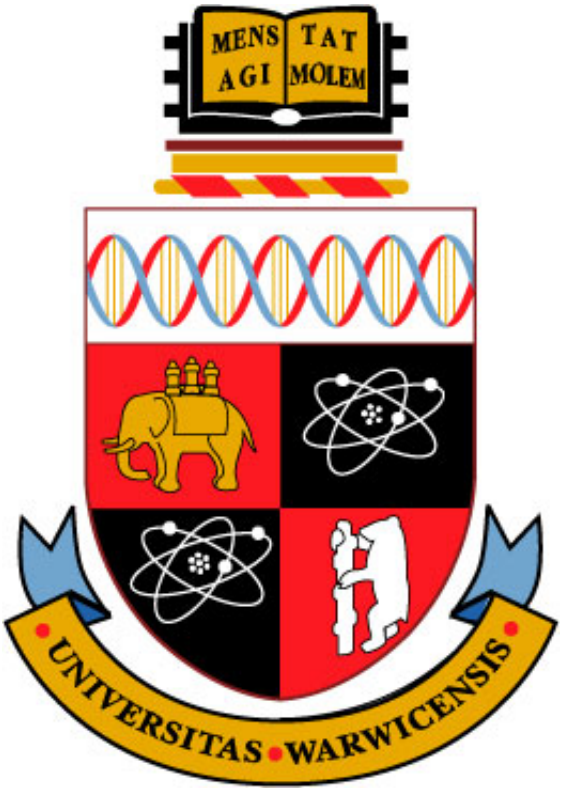
Semi-Convolutional LSTMs: A New Approach to Traffic Forecasting

William Duckett
Supervised By: Prof. Theo Damoulas and Sueda Ciftci

March 2024



Abstract

This study introduces a novel machine learning architecture in the field of computer vision exploring its application to traffic forecasting. The primary objective of this research is to develop a robust machine learning model capable of generating accurate traffic forecasts, even when trained on small and noisy datasets. Our model, the Semi-Convolutional LSTM shows a significant improvement on the Convolutional LSTM, outperforming it by 8.3% in accuracy. Ultimately the traffic forecast is intended to be used as a feature in the London Air Quality Projects' air-quality forecasting model to help produce accurate air-quality forecasts.

Keywords: Machine Learning, Traffic Forecasting, Semi-Convolutional LSTM, Computer Vision, SCOOT dataset

Contents

1 Introduction

1.1 Motivation: Improving the modeling of air-pollution

This thesis introduces a novel machine learning method for traffic forecasting using data from the London Air Quality Project. Traffic forecasting is an active area of research [1]. Its significance stems from the important role traffic forecasts play in both urban planning and daily commuting. Accurate short-term traffic forecasts enable applications, such as Google Maps, to accurately route commuters thereby significantly reducing their travel times. Further, accurate traffic forecasts allow for dynamic traffic management in urban areas which can dramatically reduce congestion [2]. This thesis is primarily concerned with traffic forecasting due to the role accurate traffic forecasts can play in producing accurate quality air-quality forecasts.

Air pollution continues to be a critical public health issue, particularly in densely populated urban areas where the combination of vehicle emission, industrial activity and high population densities detrimentally affect the air-quality. The implications of prolonged exposure to contaminated air are severe, impacting not only the environment but also the health of residents. It is well-documented that air quality closely correlates with public health. Indeed, studies have shown that pollutants such as nitrogen dioxide (NO_2) and particulate matter (PM2.5 and PM10), which are prevalent in vehicle emissions, are linked to a range of serious health issues. These include respiratory conditions such as asthma and bronchitis, cardiovascular diseases, and even neurological impairments [3]. Indeed, it is estimated that over 9,000 residents of Greater London die prematurely each year due to air-quality related issues [4]. The gravity of this issue has brought air-quality to the forefront of public discourse. Consequently, legislation is beginning to be passed in an attempt to improve the air-quality around Greater London. One of the more significant pieces of recent legislation that has been passed is the expansion of the Ultra Low Emission Zone (ULEZ). The ULEZ is designed to improve the air-quality around central London by imposing stricter regulations on the vehicles that can move freely within the ULEZ zone thereby, reducing emission from vehicles. This legislation is part of a broader environmental strategy aimed at transforming transportation habits, promoting cleaner alternative modes of transport, and enhancing overall urban air quality.

Historically, measuring the impact on air pollution of legislation such as the ULEZ expansion has been difficult due to a lack of localised air-quality data. Over the past decade, air-quality sensors have become increasingly affordable, allowing many more to be placed over Greater London. This provides

far more localised air-quality data than had previously been available. The London Air Quality Project, jointly undertaken by the University of Warwick and the Alan Turing Institute, aims both to aggregate this data into a centralised dataset and to use it to produce accurate localised air quality forecasts for Greater London [4].

Each of these goals is of great importance, the consolidation of air-quality data from multiple sources into a single dataset enhances the ability of regulatory officials to implement changes in legislation and facilitates research by providing a single access point for air-quality data. Centralised access to air-quality data simplifies the evaluation of legislative impacts and therefore helps to facilitate the implementation air-quality related legislation. In addition, the improved air quality forecasts produced by the London Air Quality project have a multitude of important purposes. Accurate localised air-quality forecasts allow for dynamic management of air-quality over London for by example managing traffic flow to redirect vehicles away from areas with poor air quality, thereby improving overall air quality conditions. Alongside this, accurate local air-quality forecasts allow for applications such as the Clean Air Route Finder [5] to be produced. The Clean Air Route Finder uses air-quality forecasts to guide users around London via the route with best air-quality. This allows users to avoid consistent exposure to air-pollution thus mitigating the associated negative health impacts.

The London Air Quality Project is currently using satellite and air quality sensor data to produce accurate air-quality forecasts using variations of Gaussian Process models. While these data sources are able to capture a lot of information on the dynamics of air pollution, vehicles are significant contributors to urban air pollution [6]. Therefore, if we are able to produce accurate traffic forecasts, which could then be fed into the Gaussian Process model as a feature, the London Air Quality Project may be able to produce more accurate air-quality forecasts.

1.2 Data-Driven Traffic Forecasting

Traffic forecasting is a complex task, these complexities stem from many different factors. Traffic flows exhibit spatio-temporal relationships which are often challenging to model [7]. To add to the complexity, traffic flow often suffers from exogenous shock events. Such factors can abruptly alter traffic flow in unpredictable ways, for example a road may suddenly become congestion due to an accident or a sudden change in weather may cause an increase in traffic [1]. These exogenous shocks cause a substantial uncertainty in the underlying data generating process. In addition to exhibiting these tempo-

rary exogenous shocks, traffic flows are inherently non-stationary. There are a multitude of causes for the non-stationary nature of traffic flows, many of which we are not able to model. Cultural shifts, such as the recent trend of working from home, legislative changes, for example the extension of the ULEZ zone, and infrastructure developments, like the High Speed Two, all cause a fundamental change in traffic flow. When observing data which spans only short time periods, these changes to traffic flow are indistinguishable from long-term temporal relationships. The combination of these factors make understanding and forecasting traffic conditions a complex task.

Historically, statistical models have been used to forecast traffic conditions. However, these models have often struggled to make accurate forecasts [8]. Statistical models, such as the ARIMA model [9], are often designed for stationary time-series which exhibit linear relationships. However, as discussed above, traffic flows are inherently non-stationary. Further, traffic flows are often observed to exhibit non-linear relationships [1]. Statistical models can also struggle to respond to the short-term exogenous shocks that occur in traffic data. For these reasons, statistical models are often not deemed appropriate for the task of traffic forecasting [10].

The inability of statistical models to accurately forecast traffic, alongside the substantial volume of traffic data that has been collected in recent years, has lead to an increase in the popularity of data-driven models [1]. Indeed, there has been significant research into potential machine learning models [11], with such models become the standard for traffic forecasting [12].

Having laid out the motivation and rationale for this thesis, the following section will discuss the specific objectives and methodology of the study.

1.3 Objectives

Setting objectives for a thesis is critical as objectives act as a guide throughout the research process and allow for a critical evaluation of whether the intended goals have been achieved. Three key objectives were outlined in the project specification and are outlined below.

The first objective of this project is to develop the necessary background and investigate how machine learning architectures can be applied to the task of traffic forecasting. The success of this objective is demonstrated in sections 2 and 4. The next objective of the project is to develop and evaluate a machine learning model to forecast traffic. The success of this objective is demonstrated in sections 4.4 6 and 8. Finally the overarching goal of this

thesis is to implement the model into the LAQN code-base, allowing the model to be used in a practical context.

Ensuring compatibility with the LAQN code-base has shaped many decisions thought the project. For instance, as discussed in section 6, it has been important to develop the model to have a minimal number of external dependencies to allow the model to align with the dependencies already included in the LAQN code-base. The most significant ramification of ensuring comparability with the LAQN code-base has been the requirement to use the SCOOT (Split Cycle Offset Optimization Technique) dataset, a traffic dataset already integrated into the LAQN code-base. With the dataset choice influenced by compatibility with the LAQN code-base, let us now shift our focus to the SCOOT dataset.

1.4 The SCOOT Dataset

The SCOOT dataset consists of data gathered from sensors installed throughout Greater London. The SCOOT sensors are built into the roads and are designed to count the number of vehicles that pass over them each hour. There are 12,421 of these sensors distributed across Greater London. Figure 1 illustrates this distribution with each red dot representing a sensor's location on the map of Greater London.

After accounting for missing readings, the dataset comprises approximately 9.45 million data points for the 12,421 sensors. While seemingly substantial, this equates to a relatively small number of data points per sensor. For example, consider that the LargeST benchmark traffic dataset contains approximately 4.52 billion data points across 8600 nodes [13]. Consequently, the SCOOT dataset's data density per node falls just short of being four orders of magnitude smaller than that of the LargeST benchmark dataset.

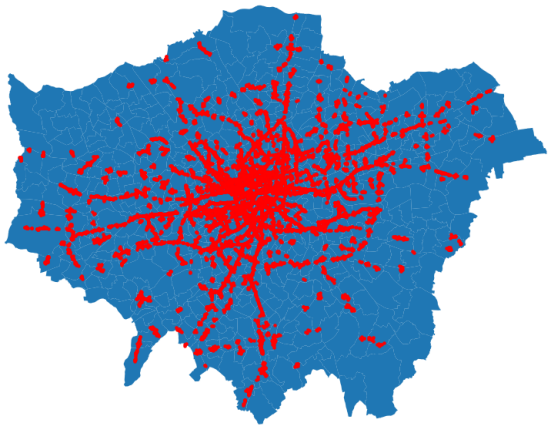


Figure 1: A map of the SCOOT sensors across Greater London

The dataset covers data received from the sensors from April 1st, 2021, to May 29th, 2021, providing two months worth of data. A more detailed exploration of the dataset and an explanation of the data pre-processing techniques is provided in Section 3 below.

1.5 Contributions of the Thesis

In applying a data driven model to the SCOOT dataset, this thesis provides two significant contributions. Firstly, it addresses the gap in applying machine learning methodologies to the SCOOT dataset. Thereby, providing accurate traffic forecasts to the London Air Quality Project which in turn can be used to enhance the accuracy of their air-quality forecasts.

Secondly, this thesis presents the development of the Semi-Convolutional LSTM, a novel computer vision architecture specifically designed for the task of traffic forecasting. Throughout this thesis, we detail the rationale behind its design process and demonstrate its increased performance over the Convolutional LSTM, showcasing a notable 8% decrease in the Mean Absolute Error of the forecasts produced.

Throughout this thesis, we will provide the necessary background in machine learning needed to understand the Semi-Convolutional LSTM architecture, alongside this we will explain the process of transforming the SCOOT dataset into air quality forecasts such as the one seen in Figure 2.

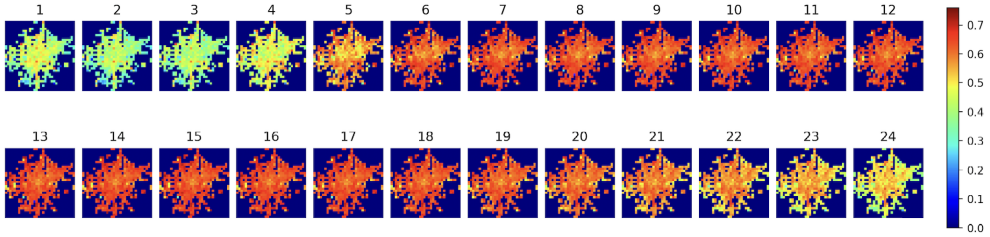


Figure 2: An example of a 24 hour traffic forecast where color represents the normalised traffic

2 Background Theory

This section of the thesis aims to methodically describe the foundational machine learning knowledge required to understand the architecture of the Semi-Convolutional LSTM. The section begins with a discussion of the basic machine learning concepts such as Perceptrons and Neural Networks, focusing on describing the mathematics behind these models. A core understanding of such concepts is required to understand more complex machine learning architectures.

Once we have gained an understanding of Neural Networks we move on to exploring Recurrent Neural Networks. This are of particular importance to this thesis as Recurrent Neural Networks (RNNs) are specifically designed for time-series forecasting and are therefore highly applicable to the task of traffic forecasting. Motivated by RNNs, specifically their weakness to the exploding and vanishing gradient problems, we introduce the Long-Short Term Memory (LSTM) Network. The LSTM is a form of RNN designed specifically to mitigate the affects of the exploding and vanishing gradient problems. LSTMs themselves have been successfully used for traffic forecasting [14] and therefore are a key architecture to understand. Furthermore, as the name suggests Semi-Convolutional LSTMs are based of the LSTM architecture and therefore a strong grounding in LSTMs is crucial for an understanding of the Semi-Convolutional LSTM.

Finally, the section finishes off by discussing additional topics related to Machine Learning such as Cost Functions and the convolution operator. Understanding of this section allows for a better understanding of hyper-parameter selection and the Semi-Convolutional LSTM.

2.1 Perceptrons

Perceptrons are an excellent starting point for gaining intuition into Neural Networks as they are one of the simplest forms of Neural Networks. This relative lack of complexity allows us to better grasp the theory behind how they learn providing valuable insight into the principles underlying more complex neural network architectures. Perceptrons were introduced in the 1950s by Frank Rosenblatt [15], as a mathematical formulation of how a single neuron in the brain functions. They are a simple form of neural network designed for binary classification tasks. Perceptrons classify an input vector into a binary class by calculating the dot product between the input vector and a weight vector and comparing this to a threshold value. First let us

define the Heaviside step function $H : \mathbb{R} \to \mathbb{R}$,

$$H(x) = \begin{cases} 1, x \geq 0 \\ 0, x < 0 \end{cases} \tag{2.1.1}$$

Now that we have the Heaviside Step Function, we can write the Perceptron equation. The Perceptron take as input a weight vector $\mathbf{w} \in \mathbb{R}^d$ an input vector $\mathbf{x} \in \mathbb{R}^d$ and a bias value $\theta \in \mathbb{R}$ and outputs a value, $\hat{y} \in \{0, 1\}$. The Perceptron can be defined as:

$$\hat{y} = H(\mathbf{w}^T\mathbf{x} - \theta) \tag{2.1.2}$$

Now we have this equation, it is clear to see that it has a geometric interpretation. The weights $\mathbf{w}$ can interpreted as a vector from the origin in $\mathbb{R}^d$, this vector is perpendicular a hyperplane, $P \in R^{d-1}$ defined by $P = \{\mathbf{v} \in \mathbb{R}^d : \mathbf{w}^T\mathbf{v} = 0\}$. The threshold value can be interpreted as the shift of P away from the origin in the direction of $\mathbf{w}$, this new affine hyper-plane, $P_a \in R^{d-1}$, is defined by $P_a = \{\mathbf{v} \in \mathbb{R}^d : \mathbf{w}^T\mathbf{v} - \theta = 0\}$. The affine hyper-plane P_a can be interpreted as the decision boundary of a perceptron, any $\mathbf{x}$ above the affine hyper-plane P_a will be assigned a value of 1 and any $\mathbf{x}$ below P_a will be assigned the value 0. This is a bijection between affine hyper-planes and weight matrix threshold pairs, therefore, a perceptron is able to emulate any function which is linearly separable. For the rest of the section, we will be considering perceptrons with a threshold value of $\theta = 0$, let us show why this does not affect any results. First let us notice that:

$$\mathbf{w}^T\mathbf{x} - \theta = [\mathbf{w}^T, -\theta] \cdot \begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix} \tag{2.1.3}$$

Therefore, we can define a new weight matrix $\tilde{\mathbf{w}} \in \mathbb{R}^{d+1}$, $\tilde{\mathbf{w}} = \begin{bmatrix} \mathbf{w} \\ -\theta \end{bmatrix}$. We can further define a new input vector $\tilde{\mathbf{w}} \in \mathbb{R}^{d+1}$ in a similar manner, $\tilde{\mathbf{x}} = \begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix}$. By noticing that $H(\tilde{\mathbf{w}}^T\tilde{\mathbf{x}}) = H(\mathbf{w}^T\mathbf{x} - \theta)$ and that $H(\tilde{\mathbf{w}}^T\tilde{\mathbf{x}})$ can be interpreted as a perceptron with $d + 1$ dimensional weights and inputs, it is clear to see that all results for perceptrons with a threshold value of 0 hold for general perceptrons. Therefore, for the remainder of this section we will be working with the simplified perceptron equation:

$$\hat{y} = H(\tilde{\mathbf{w}}^T \tilde{\mathbf{x}}) \tag{2.1.4}$$

We have shown how given a weight vector, a perceptron can be used to classify points into binary categories. However, the selection of the weight vector has not been discussed. Given an ordered set X of n input vectors $X = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n\}$ and an ordered set Y of n class labels $Y = \{y_1, y_2, ..., y_n\}, y_i \in \{0, 1\}$ if the classes are linearly separable, the Perceptron Learning Algorithm (PLA) can find a weight vector $\tilde{\mathbf{w}}$ such that $y_i = H(\tilde{\mathbf{w}}^T \tilde{\mathbf{x}}_i), \forall i \in [1, n]$. PLA works by taking an arbitrary weight vector $\mathbf{w_0}$ and iteratively updating the vector by a pre-defined learning rate, η whenever the a point is miss-classified. PLA can be defined as follows:

Algorithm 1 Perceptron Learning Algorithm

$\tilde{\mathbf{w}} \leftarrow \tilde{\mathbf{w_0}}$
while $\exists \mathbf{x}_i | H(\tilde{\mathbf{w}}^T \tilde{\mathbf{x}}_i) \neq y_i$ **do**
    $\tilde{\mathbf{w}} \leftarrow \tilde{\mathbf{w}} + \eta y_i \tilde{\mathbf{x}}_i$
end while

PLA will terminate with $\tilde{\mathbf{w}}$ that correctly classifies all points in the set X if the classes are linearly separable [16].

2.2 Neural Networks

We can generalise the concept of a perceptron by replacing the Heaviside Step Function with a general non-linear activation function f, we will refer to these as neurons for the remainder of this section. The conventional notation changes slightly between perceptrons and neurons, the weight vector $\mathbf{w}$ and the input vector $\mathbf{x}$ remain the same but the concept of a threshold value, θ is replaced with a bias value $b = -\theta$ and the output is denoted as a rather than $\hat{y}$. Therefore, the equation for the output a of a neuron is given by:

$$a = f(\mathbf{w}^T \mathbf{x} + b) \tag{2.2.1}$$

The output range of a neuron is dependant on the activation function. This allows for tasks other than binary classification such as regression. Further, we can stack multiple neurons on top of each other to create Neural Networks. Neural networks consist of the following, an input layer, one or more hidden layers and an output layer. Let us note that the output layer does not necessarily have to consist of a single neuron, for example if one is performing

a classification task with p classes, the output layer may have p neurons. Let us visualise a neural network.

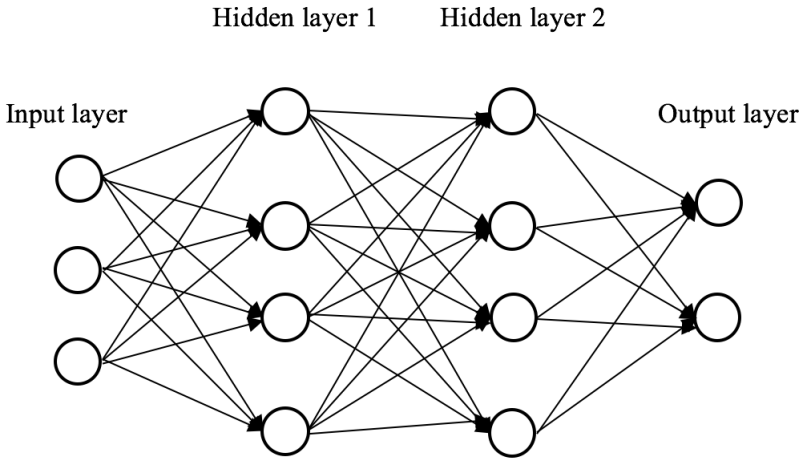


Figure 3: A Neural Network with two hidden layers. Image from [17]

Figure 3 illustrates a neural network with two hidden layers. As illustrated, neurons in consecutive layers are connected with weighted edges. The weights of these edges form a weight vector for each neuron. We can concatenate all the weight vectors for a particular layer to form the weight matrix for that layer. Let layer l consist of n neurons and layer $l-1$ consist of m neurons. Let us denote the weight vector for the $i'th$ neuron in layer l as $\mathbf{w}_i^{[l]} \in \mathbb{R}^m$. Further let $a_i^{[l]} \in \mathbb{R}$ denote the output of the $i'th$ neuron in the $l'th$ layer and $\mathbf{a}^{[l-1]} \in \mathbb{R}^m$ be the vector of the outputs of the neurons from layer $l-1$. Let $b_i^l \in \mathbb{R}$ be the bias for the $i'th$ neuron in layer l. Thus we have that $a_i^{[l]} = f(\mathbf{w}_i^{[l]T}\mathbf{a}^{[l-1]} + b_i^l)$.

One can see that by concatenating the weight vectors for a layer l into a weight matrix, $\mathbf{W}^{[l]} \in \mathbb{R}^{m\times n}$ and the bias values into a bias vector, $\mathbf{b}^{[l]} \in \mathbb{R}^n$, one can compute $\mathbf{a}^{[l]}$ with a single computation: $\mathbf{a}^{[l]} = f(\mathbf{W}^{[l]}\mathbf{a}^{[l-1]} + \mathbf{b}^{[l]})$. Therefore, the equation for a K layer neural network with input vector $\mathbf{x}$, output vector $\hat{\mathbf{y}}$ and activation function f^l for layer l is as follows:

$$\begin{aligned}
\mathbf{a}^{[0]} &= \mathbf{x} \\
\mathbf{a}^{[1]} &= f^{[1]}(\mathbf{W}^{[1]}\mathbf{a}^{[0]} + \mathbf{b}^{[1]}) \\
\mathbf{a}^{[2]} &= f^{[2]}(\mathbf{W}^{[2]}\mathbf{a}^{[1]} + \mathbf{b}^{[2]}) \\
&\vdots \\
\hat{\mathbf{y}} &= f^{[K]}(\mathbf{W}^{[K]}\mathbf{a}^{[K-1]} + \mathbf{b}^{[K]})
\end{aligned} \tag{2.2.2}$$

Now that we have the equations for a neural network, there is some further notation to define. Firstly, let us denote the application of a neural network to an input $\mathbf{x}$ as $net(\mathbf{x})$. Further, let the set of all weight matrices and bias vectors be denoted as θ, θ is referred to as the set of parameters for a network. In a similar manner to perceptrons, we require an algorithm that for a training set $X = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_N\}$ with labels $Y = \{\mathbf{y}_1, \mathbf{y}_2, ..., \mathbf{y}_N\}$ selects a "good" parameter set. Since, unlike perceptrons, neural networks can perform different tasks such as regression and multi-class classification, it is not clear how to measure the performance of a parameter set. The performance of a network on a single input $\mathbf{x}_i$ with label $\mathbf{y}_i$ can be measured using a metric, this is referred to as the loss function and is denoted as $L(net(\mathbf{x}_i), \mathbf{y}_i)$. The loss of each input over the training set can be summed to calculate the cost function. For a fixed training set, the cost function is a function of the parameter set θ and is denoted as $J(\theta)$. The cost function can be expressed as:

$$J(\theta) = \frac{1}{N}\sum_{i=1}^{N} L(net(\mathbf{x}_i), \mathbf{y}_i) \tag{2.2.3}$$

It is important to note that in practice, the loss and the cost function are often referred to interchangeably.

Now that we have a method of quantifying the performance of a network, namely the value of the cost function, we would like to find an algorithm to select an optimal parameter set. There are multiple methods for selecting parameter sets, the most common and the one that we will focus on is known as gradient descent. For the remainder of this background section for the sake of simplicity, we are not going to consider neural networks to have bias vectors. Similarly to how we could remove the threshold values when thinking about perceptrons, the bias vectors can be thought of as a neuron in the previous layer that always has value 1.

2.2.1 Gradient Descent

Gradient Descent is an optimisation algorithm which uses the gradients of a weight matrix with respect to the cost function, $\frac{\partial J}{\partial \mathbf{W}^{[i]}}$, to update iterative update the weight matrix towards one that minimises the cost function[18]. Gradient Descent minimises the cost function by iteratively subtracting a value proportional to $\frac{\partial J}{\partial \mathbf{W}^{[i]}}$ from $\mathbf{W}^{[i]}$. This constant of proportionality is known as the learning rate as is denoted as η. The weight matrix is updated until the cost function converges for a particular training set. In practice this means that the algorithm is run for a fixed number of iterations or until $\frac{\partial J}{\partial \mathbf{W}^{[i]}}$ is below a certain threshold value. Gradient descent trains all parameters concurrently, therefore, in this section $\mathbf{W}$ is a vector of all weight matrices and in a similar manner:

$$\frac{\partial J}{\partial \mathbf{W}} = \begin{bmatrix} \frac{\partial J}{\partial \mathbf{W}^{[1]}} \\ \frac{\partial J}{\partial \mathbf{W}^{[2]}} \\ \vdots \\ \frac{\partial J}{\partial \mathbf{W}^{[K]}} \end{bmatrix} \tag{2.2.4}$$

Using this notation, the Gradient Descent Algorithm can be expressed as follows:

Algorithm 2 Gradient Descent

```
initialise W
while W has not converged do
    W ← W − η ∂J/∂W
end while
```

Gradient Descent will always converge to an optimal value of $\mathbf{W}$ if the cost function., $J(\theta)$ is continuous, convex and η is sufficiently small. While a formal proof of the convergence of the Gradient Descent Algorithm is beyond the scope of this project, it is nevertheless useful to provide an intuition as to why Gradient Descent converges under these conditions.

At each step of the iteration, if η is sufficiently small, the value of $J(\theta)$ must decrease. This decrease is due to the nature of the update rule which adjusts $\mathbf{W}$ in the direction of steepest descent of J at θ. Therefore, as we have monotonic decrease of J where J is continuous and bounded below, we have that J must converge to a stationary point. Provided J is convex, as all stationary points of convex functions are global minima, J will converge to a

global minimum. Therefore, Gradient Descent will return an optimal value for $\mathbf{W}$.

Stochastic Gradient Descent is a less computationally intensive variation of the Gradient Descent Algorithm [19]. It works by splitting the training set into batches and updating the weight matrix depending on the gradient of each batch. As the gradient on the full training set is not being calculated at each step, the convergence of the weight matrix is less stable. This is often an advantage as it can allow $J(\theta)$ to escape shallow local minima or saddle points, allowing $J(\theta)$ to converge to smaller values. The Stochastic Gradient Descent can be found at Algorithm 3.

Algorithm 3 Stochastic Gradient Descent

initialise $\mathbf{W}$
partition X, Y into subsets $X_1, ..., X_R$, $Y_1, ..., Y_R$
while $\mathbf{W}$ has not converged **do**
  $i \leftarrow Random([1, R])$
  $J_i(\theta) \leftarrow J$ restricted to the subset X_i
  $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J_i}{\partial \mathbf{W}}$
end while

Adaptive Moment Estimation (ADAM) is a stochastic optimisation algorithm that builds upon Stochastic Gradient Descent [20]. The ADAM algorithm uses the first and second moments of the gradient to dynamically adjust the learning rate of each parameter. The ADAM optimiser has become one of the default optimisation algorithms for updating the weights and biases.

2.2.2 Back Propagation

In Gradient Descent, $\frac{\partial J}{\partial \mathbf{W}^{[l]}}$ is used to update the weight matrix $\mathbf{W}^{[l]}$. However, it is not clear how $\frac{\partial J}{\partial \mathbf{W}^{[l]}}$ is calculated. In order to calculate $\frac{\partial J}{\partial \mathbf{W}^{[l]}}$, we can use a method known as back propagation [21].

Back propagation uses the chain rule to calculate the necessary partial derivatives. Back propagation relies on a mathematics formula for computing the derivative of the composition of functions, called the chain rule. To express the chain rule first let g be a function that is differentiable at the point c, let f be a function that is differentiable at the point $g(c)$ then:

$$(f \circ g)'(c) = f'(g(c)) \cdot g'(c) \tag{2.2.5}$$

Let us now consider a neural network with L layers, where each layer l contain $n^{[l]}$ neurons. In this case, a neuron in layer l has $n^{[l-1]}$ weighted inputs, let us denote the weighted input to the $i'th$ neuron in layer l as $z_i^{[l]} \in \mathbb{R}$ with $z_i^{[l]} = \mathbf{w}_i^{[l]T}\mathbf{a}^{[l-1]} + b_i^{[l]}$. We can compute all the pre-activation values for a layer in a single calculation by using the layers weight matrix, the vector of a layers pre-activation values is denoted as $\mathbf{z}^{[l]} \in \mathbb{R}^{n^{[l]}}$ with, $\mathbf{z}^{[l]} = \mathbf{W}^{[l]}\mathbf{a}^{[l-1]} + \mathbf{b}^{[l]}$.

Now that we have that notation let us notice that as $\frac{\partial J}{\partial \mathbf{W}^{[l]}} = \frac{1}{N}\sum_{j=1}^{N} \frac{\partial L}{\partial \mathbf{W}^{[l]}}(net(x_j), y_j)$, rather than focusing on calculating $\frac{\partial J}{\partial \mathbf{W}^{[l]}}$ for the remainder of the section we will be focused on calculating $\frac{\partial L}{\partial \mathbf{W}^{[l]}}$ for a single training point $\mathbf{x}, \mathbf{y}$. Using the notation introduced in this section we see that for the final layer, $\hat{\mathbf{y}} = f^{[L]}(\mathbf{W}^{[L]}\mathbf{a}^{[L-1]} + \mathbf{b}^{[L]}) = f^{[L]}(\mathbf{z}_L)$. This means we can apply the chain rule as follows:

$$\frac{\partial L}{\partial \mathbf{W}^{[L]}} = \frac{\partial L}{\partial \hat{\mathbf{y}}} \cdot \frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{W}^{[L]}} = \frac{\partial L}{\partial \hat{\mathbf{y}}} \cdot \frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{z}^{[L]}} \cdot \frac{\partial \mathbf{z}^{[L]}}{\partial \mathbf{W}^{[L]}} \tag{2.2.6}$$

As we know the derivative of the loss function, $\frac{\partial L}{\partial \hat{\mathbf{y}}}$ is known, in a similar manner we have $\frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{z}^{[L]}}$. Further as $\mathbf{z}^{[L]} = \mathbf{W}^{[L]}\mathbf{a}^{[L-1]} + \mathbf{b}^{[L]}$, $\frac{\partial \mathbf{z}^{[L]}}{\partial \mathbf{W}^{[L]}} = \mathbf{a}^{[L-1]}$. Therefore we can calculate $\frac{\partial L}{\partial \mathbf{W}^{[L]}}$, using the intuition gained from the calculation of the gradients for the output layer, we can find a general formula for computing the gradients in the hidden layers. Let us start by using the chain rule to expand $\frac{\partial L}{\partial \mathbf{W}^{[l]}}$ for a given l.

$$\frac{\partial L}{\partial \mathbf{W}^{[l]}} = \frac{\partial L}{\partial \mathbf{a}^{[l]}} \cdot \frac{\partial \mathbf{a}^{[l]}}{\partial \mathbf{z}^{[l]}} \cdot \frac{\partial \mathbf{z}^{[l]}}{\partial \mathbf{W}^{[l]}} \tag{2.2.7}$$

Since the derivative of the activation function is known, we can calculate $\frac{\partial \mathbf{a}^{[l]}}{\partial \mathbf{z}^{[l]}}$. Further, $\frac{\partial \mathbf{z}^{[l]}}{\partial \mathbf{W}^{[l]}} = \mathbf{a}^{[l-1]}$. Therefore, the only partial derivative that is left to calculate is $\frac{\partial L}{\partial \mathbf{a}^{[l]}}$. Let us focus on this calculation, first we should notice that $\mathbf{a}^{[l]}$ is used to compute the vector $\mathbf{z}^{l+1}$, therefore we have that:

$$\frac{\partial L}{\partial \mathbf{a}^{[l]}} = \frac{\partial L}{\partial \mathbf{z}^{l+1}} \cdot \frac{\partial \mathbf{z}^{l+1}}{\partial \mathbf{a}^{[l]}} \tag{2.2.8}$$

Since $\mathbf{z}^{[l+1]} = \mathbf{W}^{[l+1]}\mathbf{a}^{[l]} + \mathbf{b}^{[l+1]}$, $\frac{\partial \mathbf{z}^{l+1}}{\partial \mathbf{a}^{[l]}} = \mathbf{W}^{[l+1]}$. Therefore, the only derivative left to calculate is $\frac{\partial L}{\partial \mathbf{z}^{l+1}}$. This is where back propagation gets its name from, we do know $\frac{\partial L}{\partial \mathbf{z}^{L}}$. Thus, if we first compute $\frac{\partial L}{\partial \mathbf{z}^{L}}$, we can use the result to

compute $\frac{\partial L}{\partial \mathbf{W}^{[L-1]}}$. We can iteratively repeat this process working our way backwards through the network and therefore we can calculate $\frac{\partial L}{\partial \mathbf{W}^{[l]}} \forall l$.

In this section we have ignored bias values as conceptually they can be thought of as weights and a similar process can be used to compute $\frac{\partial L}{\partial \mathbf{b}^{[L]}}$. By storing the gradients for $\frac{\partial L}{\partial \mathbf{z}^{l}}$ as we compute them, we can efficiently compute all the necessary partial derivatives to perform back propagation. This allows us to efficiently train networks using variations of the Gradient Descent algorithm.

2.2.3 Activation Functions

Up to this point, we have discussed the use of non-linear activation functions. This section delves deeper into non-linear activation functions, specifically it addresses why they are used and introduces some commonly used non-linear activation functions.

A natural question that arises is why do we use non-linear activation functions as opposed to linear activation functions? While linear functions are heavily studied and simple to compute, using linear functions leads to the following limitation. No matter how many layers are in a neural network, if only linear activation functions are used, the neural network is equivalent to a single-layer network. This equivalence is because the composition and addition of linear functions results in a linear function. Since single-layer networks are capable of representing linear functions, multi-layer networks with linear activation functions are no more powerful than single layer networks.The use of non-linear activation functions overcomes this limitation. By introducing non-linearity, neural networks are able to learn non-linear mappings from the inputs to the outputs. This significantly increases the complexity of functions neural networks can emulate leading to much improved performance on real world tasks.

An activation function should also be differentiable, since the derivative of the activation function is needed during back propagation. Some commonly used activation functions are not differentiable at 0. While this appears to be a problem, there are many heuristics to fix this issue, for example manually setting the value of the derivative at 0. At each layer during back propagation, the derivative of the activation function at a given point is introduced as a factor. This can cause the exploding gradient problem or the vanishing gradient problem if the derivative of the activation function is consistently large or small. A further factor to consider when choosing an activation function is the computational efficiency of the function. If a function is computationally

intense to compute, the time taken for propagation or back propagation may increase, causing the model to take longer to train.

Activation functions tend to be monotonic. This is because during gradient descent we increase or decrease the value of the weights depending on the gradient of the cost function with respect to that weight. Thus, when increasing the magnitude of a weight, the influence a neuron has on neuron in the following layer should increase. If a non-monotonic activation function were to be used, this may not be the case. This can cause instability in training. However, there is much current research on the use of non-monotonic activation functions [22].

Below one can see a list of commonly used activation functions along with their advantages and disadvantages.

1. Sigmoid: $\sigma(x) = \frac{1}{1+e^{-x}}$
   The Sigmoid function is differentiable everywhere. Further, it has an output range between 0 and 1 making it a good choice for classification tasks. However, the Sigmoid function saturates at both tails which can cause the vanishing gradient problem to occur in deep networks with Sigmoid activation functions. Moreover, the output is not 0 centered and so is not suited for all tasks. Further, due to the presence of the exponential function, the Sigmoid function is computationally expensive to compute.

2. tanh: $\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$
   Unlike the Sigmoid function, tanh is 0 centered and has a a larger maximum gradient. The tanh function is also differentiable everywhere. However, similarly to the Sigmoid function, the tanh function also saturates at both tails potentially causing the vanishing gradient problem in deep networks. The tanh function is also computationally expensive to compute.

3. Rectified Linear Unit (ReLU): $\mathrm{ReLU}(x) = \begin{cases} x, x > 0 \\ 0, x \leq 0 \end{cases}$
   ReLU has become increasingly popular recently and is considered the standard hidden layer activation function as the ReLU function is simple, efficient to compute and it does not suffer from the vanishing gradient problem. Although it may appear that ReLU not being differentiable at 0 might cause issues, it is possible simply set the derivative at 0. The ReLU function still suffers from the following issues, firstly the output from the ReLU function is not 0 centered. Secondly, if a neuron outputs a value of below 0, since the gradient will be 0, it is unlikely it

will ever output a non-zero value for that input. This is known as the dying ReLU problem [23], and can make a model with ReLU activation functions much more sensitive to its initialisation and learning rate.

2.2.4 Cost functions

In sections 2.2 and 2.3 the use of cost functions is briefly discussed. This section explores cost functions in greater detail, specifically the desired properties of a cost function and some examples of commonly used. Different cost function are compatible with different tasks, as traffic forecasting is a regression task, for the remainder of this section we will be focusing solely on cost functions for regression tasks.

As the gradient of the cost function is needed during gradient descent, the cost function should be differentiable. Further, a cost function and its derivative should not be computationally expensive to compute as this would increase the time needed to train a model. Further, a cost function should have a stable gradient so that the gradient never gets too large or too small as otherwise this could cause the exploding or vanishing gradient problems. For a given training set $X = \{\mathbf{x}_1, ..., \mathbf{x}_N\}$ with labels $Y = \{\mathbf{y}_1, ..., \mathbf{y}_N\}$ the following are some commonly used cost functions for regression tasks:

1. Mean Squared Error (MSE): $J_{\text{MSE}}(X, Y) = \frac{1}{N} \sum_{i=1}^{N} |net(\mathbf{x}_i) - \mathbf{y}_i|^2$
2. Mean Absolute Error (MAE): $J_{\text{MAE}}(X, Y) = \frac{1}{N} \sum_{i=1}^{N} |net(\mathbf{x}_i) - \mathbf{y}_i|$
3. Root Mean Squared Error (RMSE): $J_{\text{RMSE}}(X, Y) = \sqrt{\frac{1}{N} \sum_{i=1}^{N} |net(\mathbf{x}_i) - \mathbf{y}_i|^2}$

The primary deciding factor between which of these cost functions to use is the desired sensitivity to large errors. If one wants a large error to be more penalised than several small errors summing to the same value, MSE is the appropriate choice. Conversely, if the dataset contains anomalies, leading to inevitable large errors, MAE is a more suitable choice. This is as MSE may cause the model to over-fit in an attempt to minimise the large errors caused by these anomalies. RMSE offers a middle ground between MSE and MAE so is an appropriate choice if the dataset is relatively clean but still contains some anomalous points.

2.3 Recurrent Neural Network (RNN)

So far we have introduced neural networks and how their parameter set can be optimised via back propagation. The understanding of neural networks

is foundation for the understanding of more complex machine learning architectures. Traffic forecasting is fundamentally a time-series prediction task and therefore, it is important to understand machine learning architectures designed to capture temporal relationships in data. One such architecture is known as a Recurrent Neural Network (RNN). Recurrent Neural Networks are a class of neural network designed to recognise patterns in sequential data [24]. This capability is achieved through the introduction of memory into the architecture of each RNN unit.

This memory is introduced through the use of a "hidden state", which retains information about the inputs the RNN cell has previously seen. Thus, the output of an RNN cell is dependant on the value of the hidden state for the previous time step and its current input. This sequential processing mechanism allows RNNs to capture temporal dependencies, making them suited for time series prediction tasks. RNNs form the foundations for the Semi-Convolutional LSTM and therefore, an deep understanding of RNNs is important for this project.

RNNs process sequential data, therefore a single input, x_i, is a sequence of input vectors: $x_i = \{\mathbf{x}_{i_1}, \mathbf{x}_{i_2}, ..., \mathbf{x}_{i_T}\}$. The time-step of an RNN refers to the current input being fed into the RNN for example, if an RNN is at time-step t then the input $\mathbf{x}_{i_t}$ would currently be being fed into the RNN cell. Similarly to the input, the output of an RNN $\hat{y}_i$ is a sequence of output vectors: $\hat{y}_i = \{\hat{\mathbf{y}}_{i_1}, \hat{\mathbf{y}}_{i_2}, ..., \hat{\mathbf{y}}_{i_G}\}$.

RNNs can be split into three categories depending on length of the sequences of their inputs and outputs. These three categories are one-to-sequence, sequence-to-sequence and sequence-to-one. The RNN equation we will give in this section is for a sequence-to-sequence RNN meaning it takes as input a sequence of vectors and produces as output a sequence of output vectors.

Since an RNN cell at time-step t not only depends on the input at time-step t but also the hidden-state at time-step $t-1$, an RNN cell has multiple weight matrices and bias vectors. In order to keep the notation concise we will not be focused on how an RNN acts on a training set, but rather how an RNN acts over a single input. Therefore, $\mathbf{x}_t \in \mathbb{R}^x$ is the input vector, $\mathbf{h}_t \in \mathbb{R}^h$ is the hidden state vector and $\mathbf{o}_t \in \mathbb{R}^o$ is the output of the RNN at time-step t. Further, we have two bias vectors, the hidden state bias vector $\mathbf{b}_h \in \mathbb{R}^h$ and the output bias vector, $\mathbf{b}_o \in \mathbb{R}^o$. We further have to introduce three weight matrices, the input weight matrix $\mathbf{W}_{xh} \in \mathbb{R}^{h \times x}$, the hidden state transition matrix $\mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$ and the output weight matrix $\mathbf{W}_{ho} \in \mathbb{R}^{o \times h}$. Further, let us note that an RNN cell uses two non-linear activation functions, the hidden state non linear activation function f and the output non-linear activation

function g. Using this notation the equation of an RNN cell at time-step t is as follows:

$$\begin{aligned} \mathbf{h}_t &= f(\mathbf{W}_{xh}\mathbf{x}_t + \mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{b}_h) \\ \mathbf{o}_t &= g(\mathbf{W}_{ho}\mathbf{h}_t + \mathbf{b}_o) \end{aligned} \tag{2.3.1}$$

The hidden state at time-step 0 has to be initialised, it is common to initialise it to the $\mathbf{0}$ vector. In a similar manner to neural networks, the weight matrices and biases can be iteratively updated using Gradient Descent. However, due to the dependency on the previous value of the hidden state, the mathematics to compute the necessary partial derivatives is more complex. Instead of using Back Propagation, to compute the partial derivative of the loss with respect to the weights and biases, we must use a process known as Back Propagation Through Time.

2.3.1 Back Propagation Through Time

As mentioned the gradients needed to perform Gradient Descent with an RNN architecture can be computed through a process known as Back Propagation Through Time [25]. We should recall that RNNs have the following weight matrices and bias vectors to train: $\mathbf{W}_{xh}, \mathbf{W}_{hh}, \mathbf{W}_{ho}, \mathbf{b}_h$ and $\mathbf{b}_o$. The method of calculating $\frac{\partial L}{\partial \mathbf{W}_{ho}}$ and $\frac{\partial L}{\partial \mathbf{b}_o}$ is very similar to regular back propagation due to the lack of temporal dependencies. Therefore, we will be focusing on the calculation of $\frac{\partial L}{\partial \mathbf{W}_{xh}}$, $\frac{\partial L}{\partial \mathbf{W}_{hh}}$ and $\frac{\partial L}{\partial \mathbf{b}_h}$. The deviations behind each of these partial derivatives are very similar. Thus in this section we will be focused on the calculation of $\frac{\partial L}{\partial \mathbf{W}_{hh}}$. Let us first notice that for a sequence-to-sequence RNN, each time-step contributes to the total loss and therefore we have that:

$$\frac{\partial L}{\partial \mathbf{W}_{hh}} = \sum_{i=1}^{T} \frac{\partial L_t}{\partial \mathbf{W}_{hh}} \tag{2.3.2}$$

Therefore, let us shift our focus to the calculation of $\frac{\partial L_t}{\partial \mathbf{W}_{hh}}$ for a given time-step t. Using the chain rule we see that $\frac{\partial L_t}{\partial \mathbf{W}_{hh}}$ can be expressed as follows:

$$\frac{\partial L_t}{\partial \mathbf{W}_{hh}} = \frac{\partial L_t}{\partial \mathbf{o}_t} \cdot \frac{\partial \mathbf{o}_t}{\partial \mathbf{h}_t} \cdot \frac{\partial \mathbf{h}_t}{\partial \mathbf{W}_{hh}} \tag{2.3.3}$$

By using a loss function with a known derivative, $\frac{\partial L_t}{\partial \mathbf{o}_t}$ is known. In a similar manner we are able to calculate, $\frac{\partial \mathbf{o}_t}{\partial \mathbf{h}_t}$. Therefore, the only partial derivative

required to be calculated is $\frac{\partial \mathbf{h}_t}{\partial \mathbf{W}_{hh}}$. In order to gain some insight into how we can calculate $\frac{\partial h_t}{\partial \mathbf{W}_{hh}}$ let us think about how a change in the values of $\mathbf{W}_{hh}$ affects the value of $\mathbf{h}_t$. From the RNN equations we have that $\mathbf{h}_t = f(\mathbf{W}_{xh}\mathbf{x}_t + \mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{b}_h)$. This not only gives us a formula for $\mathbf{h}_t$ but also a formula for $\mathbf{h}_{t-1}$. Therefore, by expanding the term $\mathbf{h}_{t-1}$ we have that:

$$\mathbf{h}_t = f(\mathbf{W}_{xh}\mathbf{x}_t + \mathbf{W}_{hh}f(\mathbf{W}_{xh}\mathbf{x}_{t-1} + \mathbf{W}_{hh}\mathbf{h}_{t-2} + \mathbf{b}_h) + \mathbf{b}_h) \tag{2.3.4}$$

The expanded formula illustrates how a small change in $\mathbf{W}_{hh}$ would not only change the value of $\mathbf{h}_t$ due to the matrix multiplication of $\mathbf{W}_{hh}$ and $\mathbf{h}_{t-1}$, but would also change the value of $\mathbf{h}_{t-1}$ and all other preceding hidden states. To manage this dependency, we must use a concept known as the total derivative. The total derivative formula assumes that all variables are possible co-dependant. The notation behind the total derivative is rather confusing to the extent that there are articles attempting to resolve the inaccuracy captured by the notation [26]. The notation we will be using is the notation given in [27] for what they refer to as the "immediate derivative". If $\phi : \mathbb{R}^D \to \mathbb{R}$ then the immediate derivative of ϕ with respect to a variable y_i is the partial derivative of phi with respect to y_i assuming all other variables are independent. The notation for the immediate derivative of ϕ with respect to x_i is: $\frac{\partial^+ \phi}{\partial y_i}$. Using this notation, the total derivative of ϕ with respect to a variable y_1 can be expressed as follows:

$$\frac{\partial \phi}{\partial y_1} = \frac{\partial^+ \phi}{\partial y_1} + \sum_{i=2}^{D} \frac{\partial^+ \phi}{\partial y_i}\frac{\partial y_i}{\partial y_1} \tag{2.3.5}$$

As $\mathbf{h}_t$ is dependent on $\mathbf{h}_{t-1}$ and $\mathbf{W}_{hh}$, where $\mathbf{h}_{t-1}$ is co-dependant on $\mathbf{W}_{hh}$, we can use the total derivatives formula to get $\frac{\partial \mathbf{h}_t}{\partial \mathbf{W}_{hh}}$. By applying the formula we get that:

$$\frac{\partial \mathbf{h}_t}{\partial \mathbf{W}_{hh}} = \frac{\partial^+ \mathbf{h}_t}{\partial \mathbf{W}_{hh}} + \frac{\partial^+ \mathbf{h}_t}{\partial \mathbf{h}_{t-1}} \cdot \frac{\partial \mathbf{h}_{t-1}}{\partial \mathbf{W}_{hh}} \tag{2.3.6}$$

We can reuse this formula to expand the final term, $\frac{\partial \mathbf{h}_{t-1}}{\partial \mathbf{W}_{hh}}$. This can be done until we get to the term $\frac{\partial \mathbf{h}_0}{\partial \mathbf{W}_{hh}}$. Since $\mathbf{h}_0$ is an initialised constant we have that $\frac{\partial \mathbf{h}_0}{\partial \mathbf{W}_{hh}} = 0$. This gives us the crucial equation for back propagation through time of a single time-step:

$$\frac{\partial \mathbf{h}_t}{\partial \mathbf{W}_{hh}} = \frac{\partial^+ \mathbf{h}_t}{\partial \mathbf{W}_{hh}} + \sum_{i=1}^{t-1} (\frac{\partial^+ \mathbf{h}_{t-i+1}}{\partial \mathbf{W}_{hh}} \cdot \prod_{j=i+1}^{t} \frac{\partial^+ \mathbf{h}_j}{\partial \mathbf{h}_{j-1}}) \tag{2.3.7}$$

We can compute all the partial derivatives in this equation efficiently, assuming that our activation function has a derivative, we can calculate $\frac{\partial \mathbf{h}_t}{\partial \mathbf{W}_{hh}}$, allowing us to calculate $\frac{\partial L_t}{\partial \mathbf{W}_{hh}}$ and ultimately $\frac{\partial L}{\partial \mathbf{W}_{hh}}$. Therefore, as we can calculate $\frac{\partial L}{\partial \mathbf{W}_{hh}}$ we are able to perform gradient descent on RNNs. However, while the formula for calculating $\frac{\partial \mathbf{h}_t}{\partial \mathbf{W}_{hh}}$ allows us to perform gradient descent, it also highlights major problems with the RNN architecture. These problems are known as the Exploding and Vanishing Gradient Problems.

2.3.2 The Exploding and Vanishing Gradient Problems

Now that we have explored RNNs we must ask why they are not suitable for the task, this is due to a problem known as the exploding and vanishing gradient problem. By examining this problem we are able to better understand how RNNs can be modified to mitigate its effects.

The concept of the exploding and vanishing gradient problems, in the context of RNNs, was first introduced in the 1994 paper "Learning long-term dependencies with gradient descent is difficult" [28]. The exploding gradient problem refers to when the gradient of the cost with respect to a weight or bias grows exponentially large in magnitude during back propagation. As a result, during gradient descent, the updates made become exponentially large. This leads to both numerical instability and the inability of the model's parameters to converge. Similarly, the vanishing gradient problem refers to when the gradient of the cost with respect to a weight or bias decreases exponentially in magnitude during back propagation. As a result, during gradient descent, the updates become exponentially small. This may cause the model to take significantly longer to train or cause the model to not converge to a local minimum of the loss function. To understand why these problems occur let us investigate $\frac{\partial \mathbf{h}_t}{\partial \mathbf{W}_{hh}}$. Let us focus on the vanishing gradient problem. It can be shown that [27]:

$$\frac{\partial^+ \mathbf{h}_j}{\partial \mathbf{h}_{j-1}} = \mathbf{W}_{hh}^T diag(f'(\mathbf{W}_{xh}\mathbf{x}_j + \mathbf{W}_{hh}\mathbf{h}_{j-1} + \mathbf{b}_h)) \tag{2.3.8}$$

Therefore,

$$\prod_{j=i+1}^{t} \frac{\partial^{+}\mathbf{h}_j}{\partial \mathbf{h}_{j-1}} = \prod_{j=i+1}^{t} \mathbf{W}_{hh}^{T} diag(f'(\mathbf{W}_{xh}\mathbf{x}_t + \mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{b}_h)) \tag{2.3.9}$$

Thus by the triangle inequality,

$$\begin{aligned}
\left\| \prod_{j=i+1}^{t} \frac{\partial \mathbf{h}_j}{\partial \mathbf{h}_{j-1}} \right\| &= \left\| \prod_{j=i+1}^{t} \mathbf{W}_{hh}^{T} \operatorname{diag}\left(f'(\mathbf{W}_{xh}\mathbf{x}_t + \mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{b}_h)\right) \right\| \\
&\leq \prod_{j=i+1}^{t} \left\| \mathbf{W}_{hh}^{T} \right\| \left\| \operatorname{diag}\left(f'(\mathbf{W}_{xh}\mathbf{x}_t + \mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{b}_h)\right) \right\| \\
&= \left\| \mathbf{W}_{hh}^{T} \right\|^{t-i} \prod_{j=i+1}^{t} \left\| \operatorname{diag}\left(f'(\mathbf{W}_{xh}\mathbf{x}_t + \mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{b}_h)\right) \right\|
\end{aligned} \tag{2.3.10}$$

This shows how if, f' is stable and not consistently above 1 and $\left\|\mathbf{W}_{hh}^{T}\right\| < 1$ then, $\left\|\prod_{j=i+1}^{t} \frac{\partial \mathbf{h}_j}{\partial \mathbf{h}_{j-1}}\right\|$ decreases as an exponential of $t - j$. This causes the gradient due to errors made in the early time-steps of the RNN to tend to 0 meaning the weight matrices cannot be appropriately updated during back-propagation to reduce these errors. If f' is bounded below in a similar manner we can show why the exploding gradient problem occurs in RNNs.

A simple heuristic known as gradient clipping can be used to mitigate the effect of the exploding gradient problem [29]. Gradient clipping is where, if the gradient is above a certain threshold, it is set to the threshold value. However, heuristics to mitigate the vanishing gradient problem are more complex, one approach is the Long-Short Term Memory (LSTM) architecture.

2.3.3 Long-Short Term Memory Networks (LSTMs)

Long Short-Term Memory networks (LSTMs) are a variant of the RNN designed to address the vanishing and exploding gradient problems. This advancement allows LSTMs to better capture long-term dependencies in sequence data, making them particularly useful for tasks like traffic forecasting where weekly and monthly variations are expected. LSTMs were first proposed by Hochreiter and Schmidhuber in their 1997 paper [30].

The core idea behind LSTM networks is the introduction of a gating mechanism that regulates the flow of information. Alongside the hidden state,

$\mathbf{h}_t$, a LSTM also has the "forget gate", $\mathbf{f}_t$, the "input gate", $\mathbf{i}_t$, the "output gate",$\mathbf{o}_t$, and the "cell state" $\mathbf{C}_t$ each of which is a vector in $\mathbb{R}^h$. The cell state can be interpreted as the memory of the LSTM cell. There are also several new weight matrices and bias vectors, $\mathbf{W}_{xi}, \mathbf{W}_{xf}, \mathbf{W}_{xo}, \mathbf{W}_{xc} \in \mathbb{R}^{h\times x}$, $\mathbf{W}_{hi}, \mathbf{W}_{hf}, \mathbf{W}_{hc}, \mathbf{W}_{ho} \in \mathbb{R}^{h\times h}$ and $\mathbf{b}_i, \mathbf{b}_f, \mathbf{b}_c, \mathbf{b}_o \in \mathbb{R}^h$. Thus LSTM equations can be expressed as follows:

$$\begin{aligned}
\mathbf{i}_t &= \sigma(\mathbf{W}_{xi}\mathbf{X}_t + \mathbf{W}_{hi}\mathbf{h}_{t-1} + \mathbf{b}_i) \\
\mathbf{f}_t &= \sigma(\mathbf{W}_{xf}\mathbf{X}_t + \mathbf{W}_{hf}\mathbf{h}_{t-1} + \mathbf{b}_f) \\
\mathbf{C}_t &= \mathbf{f}_t \circ \mathbf{C}_{t-1} + \mathbf{i}_t \circ \tanh(\mathbf{W}_{xc}\mathbf{X}_t + \mathbf{W}_{hc}\mathbf{h}_{t-1} + \mathbf{b}_c) \\
\mathbf{o}_t &= \sigma(\mathbf{W}_{xo}\mathbf{X}_t + \mathbf{W}_{ho}\mathbf{h}_{t-1} + \mathbf{b}_o) \\
\mathbf{h}_t &= \mathbf{o}_t \circ \tanh(\mathbf{C}_t)
\end{aligned} \tag{2.3.11}$$

To better understand the functionality of the gates in an LSTM cell, it is useful to consider the behaviour of the LSTM cell when the gates take their extremal values, the $\mathbf{0}$ or $\mathbf{1}$ vector. This approach allows us to examine simple cases and extrapolate the gates functionality from them.

Let us focus on the input gate, $\mathbf{i}_t$. When we set $\mathbf{i}_t = \mathbf{0}$, the gate stops the input data from influencing the cell state. Conversely, when $\mathbf{i}_t = \mathbf{1}$, the cell state can be updated by input data. In essence, the input gate controls the influence of the input data on updating the cell state. This allows the LSTM cell to distinguish between input data that is predictive and input data which is essentially noise.

Let us move onto the the forget gate, $\mathbf{f}_t$. When we set $\mathbf{f}_t = \mathbf{0}$, the gate stops the previous value of the cell state from influencing the current cell state. Conversely, when $\mathbf{f}_t = \mathbf{1}$, the new cell state retains all previous information from the past cell state which is then updated depending on if the input is deemed important by the input gate. The forget gate allows the LSTM cell to selectively remove previously stored information and therefore focus on the input data.

Finally let us investigate the output gate, $\mathbf{o}_t$. We can see that the output gate is used to determine the hidden state through the equation $\mathbf{h}_t = \mathbf{o}_t \circ \tanh(\mathbf{C}_t)$. This illustrates how the output gate controls the extent to which the cell state is revealed to the outside world.

Now that we understand the functionality of the gating mechanisms in the LSTM cell, we can investigate why the LSTM architecture mitigates the effects of the exploding and vanishing gradient problems. The formal derivation of back-propagation through time for a LSTM cell is outside the scope of

this project, however a diagram of how information flows through an LSTM cell can offer significant insight into how these problems are mitigated.

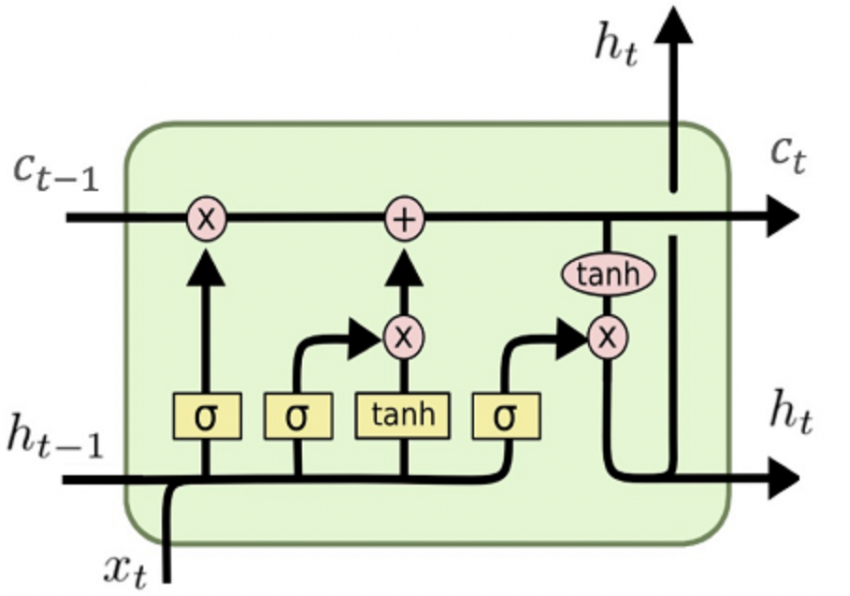


Figure 4: Diagram of an LSTM cell reproduced from [31]

Figure 4 illustrates how information from the previous cell state flows relatively unimpeded to the following cell state. The only updates to the cell state are element-wise multiplication by the forget gate and addition. This means that the gradient can flow backwards from the cell state to the previous cell state relatively unimpeded. This allows the gradient to flow backwards through the time-steps via the cell state without having to be back-propagated through a matrix multiplication or an activation function. Thereby, mitigating the effects of the exploding and vanishing gradient problems.

The diagram illustrates the flow of information from the previous cell state to the subsequent one in a LSTM architecture. This flow is largely uninterrupted, with updates to the cell state occurring due to element-wise multiplication by the forget gate and an addition. As a result, the gradient can be back-propagate from the current cell state to the preceding one relatively unimpeded. This backward flow through the cell state bypasses the need for back-propagation through matrix multiplications or activation

functions across each time step. Consequently, this mitigates the effects of the exploding and vanishing gradient problems allowing LSTMs to better capture long-term relationships than RNNs and thus be better suited to the task of traffic forecasting.

2.4 The Convolution Operator

A mathematical operator that is important to introduce in the background section is the convolution operator. Discrete convolutions are frequently used in computer vision tasks to extract spatially contextualised information from images. As suggested by the name, semi-convolutional LSTMs use convolutions to extract contextualised information about input therefore an understanding of convolutions is required for an understanding of semi-convolutional LSTMs. The convolution operation, denoted by *, combines two functions, f and g to produce a third function, $f * g$. This operation is defined for continuous functions as:

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau \tag{2.4.1}$$

The convolution of functions has applications in various fields of mathematics and engineering such as signal processing. The above is the definition of a convolution in a general context, however, due to their relevance to computer vision and machine learning tasks, in this section our focus will be on discrete convolutions.

$$(f * g)[n] = \sum_{i=-\infty}^{\infty} f[i]g[n-i] \tag{2.4.2}$$

When the functions have finite domain [-M,M] this simplifies to:

$$(f * g)[n] = \sum_{i=-M}^{M} f[i]g[n-i] \tag{2.4.3}$$

One may notice that the above definition defines a convolution over functions of a single variable. This can be generalized to functions of multiple variables. Due to their importance to machine learning, for the remainder of this section, we will focus on discrete convolutions for functions with finite domains. A 2D discrete convolution of functions $f, g : [-M, M] \times [-\mu, \mu] \to \mathbb{R}$ can be defined as follows:

$$(f * g)[n, p] = \sum_{j=-\mu}^{\mu} \sum_{i=-M}^{M} f[i, j]g[n - i, p - j] \tag{2.4.4}$$

In computer vision tasks, 2D and 3D discrete convolutions are used to extract "features" such as edges in images. This is done by treating the image as as a function and then convolving it with a kernel. For the remainder of this section we will be focusing on 2D convolutions, as the mathematics and intuition are similar for both 2D and 3D. While kernel functions can vary in shape and size, they typically take the form $\mathbf{K} : [-(2n+1), 2n+1]^2 \to \mathbb{R}$. The kernel is used to extract localised information from the image by sliding the kernel across the image and computing the convolution at each position. In the early stages of machine learning computer vision tasks, kernel functions were designed by hand. However, it is now much more common for the kernel function to be a trained parameter, optimised using back propagation.

The convolution of a subsection of an image by a kernel function captures highly localised data about the surrounding pixels for a given pixel in the image. Further, convolutions exploit the high degree of spatial regularity that often occur in image data.

3 Data Exploration

Before deciding on an appropriate model architecture, it is important to investigate the dataset the model will be trained on. An in depth understanding of the dataset allows for improved data pre-processing and model architecture selection. It also allows for the detection of unforeseen issues in the dataset and their mitigation via data augmentation and cleaning. Therefore in this section, we first explore the characteristics of the SCOOT dataset. Once we have gained an understanding of the dataset, we focus our attention on how we can best clean and normalise the dataset. As with many real world datasets, the dataset has missing entries. As a result, the first stage of data exploration involves an investigation into the missing data.

3.1 Missing Data

When collecting real world data from approximately 12500 sensors, it is likely that there will be missing data points. Therefore, it is important to conduct a thorough analysis into the missing data. We first quantify how many data points each sensor is missing, as displayed in Figure 5 which illustrates the number of missing data points for different sensors. We can see in Figure 5 a clear spike at 1392 data-points missing per sensor. This value reflects sensors that recorded no measurement over the entire dataset. On investigation we find that approximately 2500 of the 12500 sensors are permanently broken. Further, it is clear from observation of Figure 5 that all sensors are missing a significant number of data points, with the majority of sensors missing over half of their data points. In sections 3.2 and 3.4 we will show how we can use a combination of data aggregation and masking of null results during the calculation of the loss function to mitigate the impact of missing data.

Given the importance of temporal dynamics in our dataset, it is also important to understand how the number of missing data points changes with time, as shown in Figure 6, which displays the number of missing data points across all sensors vs time. One can see from Figure 6 that the number of missing data points increases as a function of time. Furthermore, the plot illustrates just how noisy the dataset is, with, for example, an average of over 10000 of the 12500 sensors missing their reading each hour in May. The high proportion of missing data points causes two major concerns. Firstly, traffic modeling requires concurrent measurements, however since such a large proportion of the hourly readings are missing, there are few concurrent readings for each sensor. Secondly, assuming the trend of increasing volumes of missing data continues, we may not be able to extend our dataset. Secondly,

towards the end of May, the proportion of missing data-points gets so large that, assuming the trend continues, using the following month's data may not be a feasible method of extending the dataset as the signal to noise ratio may become too small for the model to learn from.

Figure 5: Histogram depicting the number of missing data points for each sensor.

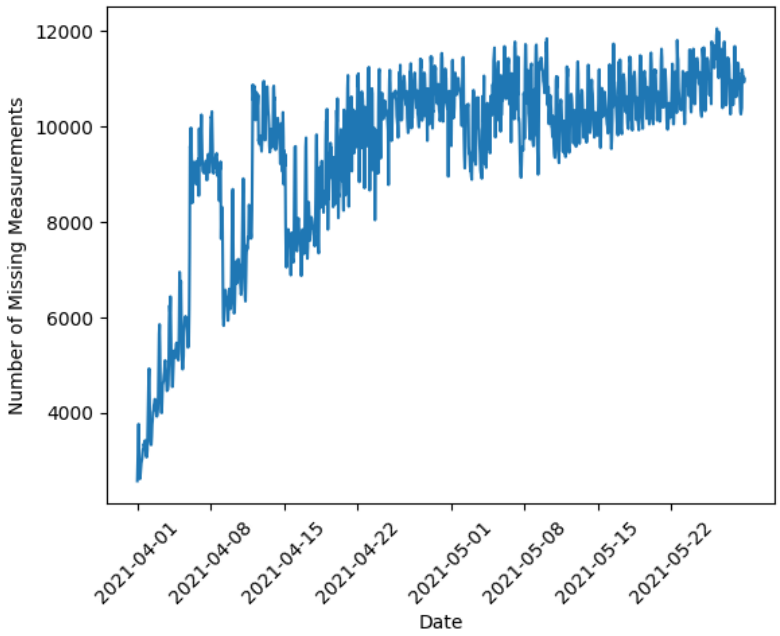


Figure 6: A plot showing the number of missing measurements vs. time

The dataset analysis discussed in this section highlights the frequency of missing data and thus the need for data aggregation in order to increase the number of concurrent measurements and decrease the noise. Further, this analysis highlights the need for a noise resistant model architecture that can be trained on a relatively small dataset.

3.2 Removal of Noise Via Data Aggregation

To mitigate the impact of missing data from individual sensors, we opted for sensor aggregation. Specifically, we selected a spatial aggregation performed at each time-step as this form of aggregation preserves the spatio-temporal characteristics of the data. By aggregating only the sensors with readings at a given time-step, null readings would not skew the aggregated value decreasing the detrimental effect of the missing data. Aggregating sensors also helps to mitigate the impact of noise due to individual sensors having anomalously low or high readings.

Two forms of spatial aggregation were considered. The first is K-mean clustering, which would cluster our sensors into K similarly sized groups. Aggregating sensors into similarly sized groups avoids having clusters with dispro-

portionately low numbers of sensors. Such clusters with few sensors could be problematic due to their increased sensitivity to noise and missing data. However, due to the geographic distribution of the sensors, clusters in the center of London would cover a significantly smaller area than clusters in the outskirts of London.

This causes various ethical and modeling issues as our dataset would be disproportionate dominated by central clusters. This would cause a class imbalance in our aggregated dataset making it difficult for the model to learn the spatio-temporal relationships of the under-represented clusters located on the outskirts of London [32]. Therefore, by choosing to use K-means clustering, we are making the decision to prioritise the accuracy of the forecasts in central London at the expense of outer London. This in turn may lead to worse air-quality forecasting in outer London simply as there is less funding and therefore fewer SCOOT sensors. Further, the differing spatial scales adds a layer of complexity to the data, as the model would have to learn how the size of the cluster affects various properties such as the time taken for a vehicle to travel across the cluster.

The second form of spatial aggregation considered is grid cell aggregation. This method divides London into equally sized grid cells and combines all sensors within each cell. As opposed to K-mean clustering, this method of aggregation does not cause a class imbalance in our aggregated dataset. This mitigates both the ethical concerns and the challenges associated with modeling under-represented classes. Further, grid cell aggregation transforms a dataset with minimal structure into a highly organized one with a consistent structure that can be leveraged by specific machine learning models. For example, if padding is applied, each grid cell has precisely eight neighbours, allowing for operations such as convolutions to be applied in a structure preserving manner.

Before applying grid cell aggregation, we must determine an appropriate size for the grid. The more grid cells, the higher the resolution of the forecasts produced. However, as we increase the number of gird cells, as on average each grid cell contains fewer sensors. This increases the sensitivity of the grid cells to missing data and erroneous readings. Further, as the number of grid cells increases so does the computational cost of training our model. Initially, a grid of size 64 by 64 was trialed, however, the gird cells were highly sensitive to missing data and the computational costs of training the model were too high. Therefore, a grid of size 32 by 32 was ultimately used. The resulting grid used to represent Greater London is displayed in Figure 7.

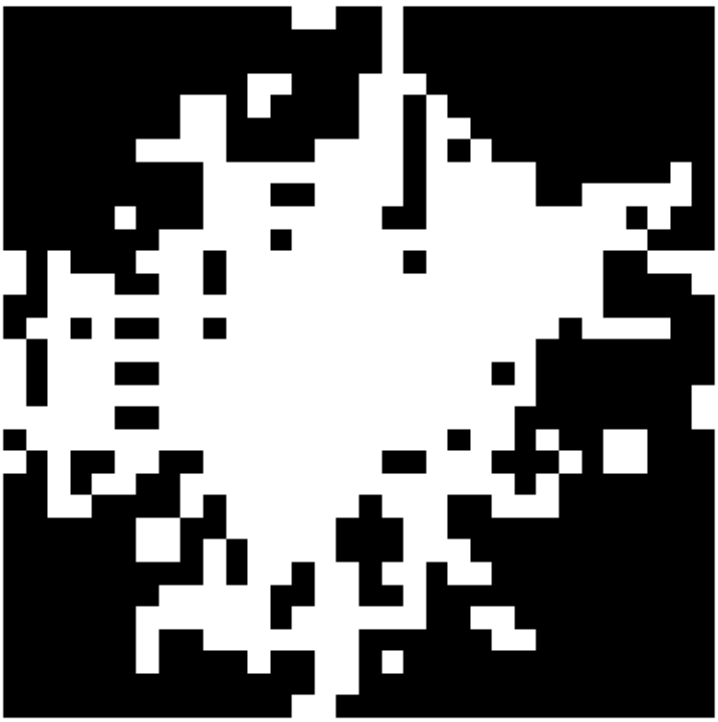


Figure 7: Image depiction of the sensor map after grid cell aggregation has been completed. The white cells represent grid cells in which there is at least one sensor while the black cells represent grid cells which contain no sensors.

After grid cell aggregation has been completed, by encoding the traffic conditions as the brightness of each grid cell, each time-step can be interpreted as an image and so the dataset can be interpreted as a film. Therefore, computer vision methods are applicable to the grid cell aggregated dataset.

3.3 Exploration of Further Data Cleaning Techniques

Although data aggregation has mitigated some noise in the dataset, it is important to acknowledge that the aggregated dataset still contains a large number of missing entries. Therefore, further consideration of data cleaning methods is needed. An initial consideration was to only use time intervals with a low volume of missing entries to train the model. This method would have dramatically reduced the size of the dataset. Further, as the model would not have encountered missing data during training, the model's ability to generalise onto test datasets, which would contain missing data, may

suffer. Therefore, this approach was not deemed suitable for this study.

Another potential method involved a coarser aggregation of the data. While this initially seemed appeared promising, the approach was deemed unsuitable for the following reasons. Firstly, a significant proportion of the missing data appears to stem from systematic outages affecting entire regions. Consequently, an increase in the average number of sensors per grid cell may not effectively reduce the proportion of missing data points. Further, a coarser aggregation would also decrease the resolution of the forecasts. The issues surrounding systematic outages are illustrated in Figure 8, which depicts a frame containing a systematic outage over North West London. The figure demonstrates why a coarser aggregation would not significantly reduce the proportion of cells with missing data-points. This is since, in the figure, even if the grid cells were significantly larger, those located in North West London would still not produce a measurement.

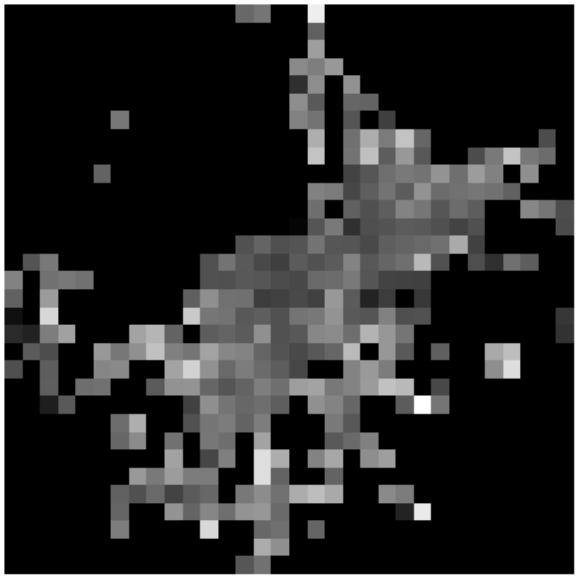


Figure 8: Example of a frame showing a systematic outage across North West London, where brighter cells denote higher traffic, and black cells indicate missing measurements.

While researching potential data cleaning methods for traffic datasets, we encountered a paper that used linear interpolation to fill in missing values [33]. However, traditional linear interpolation, along with many other time-

based interpolations, rely on data from subsequent time steps, making them unsuitable for real-time application. Given that our model is intended for integration into the LAQN system, where access to subsequent time-steps of data is unavailable, any imputation methods relying on such data should be avoided. Thus, only data regarding the current and past traffic conditions should be used for imputation.

Post aggregation the main issue surrounding missing data is that of systematic regional outages. During a systematic regional outage, the current data will not be particularly helpful for imputation and thus, only the past traffic conditions can be used to perform data imputation. This is the same as using past data to predict a time-step, which is the task of our model. So performing imputation and then training a model on a clean dataset is no less difficult than training a model on a clean dataset. Furthermore, the performance of our model is bounded by the performance of the model that performs data imputation adding another layer of complexity to the project. Therefore, data imputation was not deemed an appropriate method of filling missing data.

Considering the limitations of these possible methods, it became clear that cleaning the data further to reduce the proportion of missing entries would not be the most effective strategy. Instead, we deemed it more beneficial to select a model architecture that was resilient to missing data.

3.4 Reducing the Impact of Missing Data: Masking

Given that even after data aggregation our dataset still contains a significant number of missing entries, and our pre-processing method assigns the value 0 to a cell with no measurement, an issue arises when computing the loss function. Including these missing readings introduces a bias towards predicting 0, negatively impacting the models accuracy. To address this, we implemented masking during the calculation of the loss function, excluding any cells for which the ground truth value was 0 from the prediction. Additionally, we tracked the number of masked cells which allowed us to appropriately scale the error. This approach assumes that a 0 entry in the aggregated dataset indicated missing data rather than an actual 0 value. This assumption is deemed safe since the minimum value in the SCOOT dataset was 32 vehicles.

3.5 Data Normalisation

Following data augmentation, normalisation of the data is required before the data can be used to train our model. As the data is strictly positive, it was natural to normalise the data into the range $[0, 1]$. A common normalisation technique involves dividing each entry by the largest value. While this method is straightforward and often effective, it may not be suitable for the SCOOT dataset. As illustrated in Figure 9, the SCOOT data has a long-tailed distribution. Inspecting the dataset further, there are approximately 13,250 measurements greater than 2000. The long-tail distribution of the dataset means that if we simply divided all entries by the largest value, the majority of data-points would fall into the range $[0, \frac{1}{6}]$. Consequently, the low dispersion of this distribution would make it challenging for the model to distinguish between small and medium measurements.

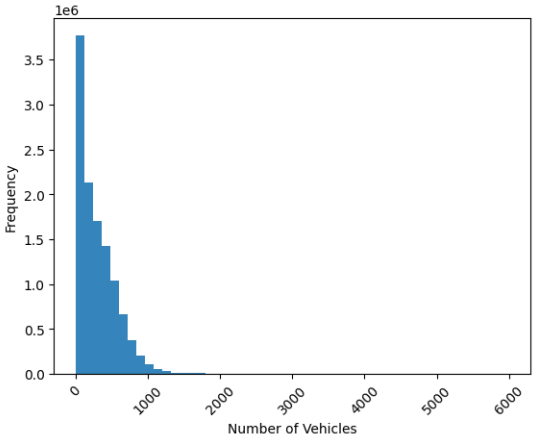


Figure 9: A histogram showing the distribution of sensor measurements

To avoid these issues logarithmic normalization was used. Logarithmic normalisation helps to transform the distribution of our long tailed dataset into a distribution with greater dispersion before compressing it into the range [0,1]. Therefore making it easier for our model to distinguish between non-extreme values. When performing logarithmic normalization, as the minimum value of our dataset is 0, we first add one to all values before applying the logarithm function to prevent values being mapped to negative infinity. Hence, all values mapped to the [0,1] range as intended. Logarithmic normalization of the dataset X as follows:

$$\hat{X} = \frac{\log(X+1)}{\max_{x \in X} \log(x+1)} \tag{3.5.1}$$

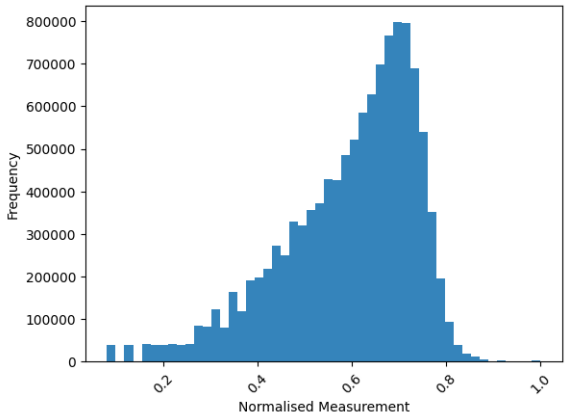


Figure 10: A histogram showing the distribution of the measurements post logarithmic normalisation

After applying logarithmic normalization, we examine the distribution of data points to ensure the intended effect has been achieved. Figure 10 reveals a distribution with a much larger dispersion indicating that logarithmic normalisation is more suitable to the SCOOT dataset than simply dividing by the largest entry.

4 Exploration of Model Architectures

This section provides an exploration of various machine learning architectures that were considered for this project in view of the analysis of the SCOOT dataset presented in the previous section. This section is heavily motivated by the extensive literature review performed in the early stages of the project. The aim of this section is to critically evaluate existing models that could be applied to the traffic forecasting. Through the understanding the strengths and limitations of these models in the context of the SCOOT dataset, we were able to develop the Semi-Convolutional LSTM. This approach helps illustrate the logical flow of ideas and the specific challenges the Semi-Convolutional LSTM addresses, enhancing the reader's understanding of the innovative solutions it offers.

4.1 Spatio-Temporal Graph Convolutional Networks (STGCN)

From reviewing the relevant literature, Spatio-Temporal Graph Convolutional Network (STGCN) stood out as an interesting architecture for this specific problem [33]. At the core of the STGCN model are the "Spatio-Temporal convolution (ST) blocks". These blocks consist of a temporal convolution followed by a spacial convolution followed by another temporal convolution. Two ST blocks are followed by a fully connected output layer to form the STGCN model. One of the most interesting components of the STGCN model is the use of spectral graph convolutions to implement convolutions that one would see in Convolutional Neural Networks over graphs. Convolutions are a highly power full tool in machine learning and signal processing. The success of Convolutional Neural Networks at dealing with image processing tasks has raised the question of if the notion of convolutions can be generalised to graphs.

While further ideas on generalising convolutions to graphs can be found in [34] and [35], to understand the STGCN architecture we must first examine how convolutions can be extended to graphs. Let us start with some definitions.

Let the $G = (V, E)$ be an undirected graph, furthermore let $|V| = N$. Firstly let us address that there are multiple forms of a graphs Laplacian, for the remainder of this section when we refer to the Laplacian of G we are referring to the matrix L such that:

$$L_{i,j} := \begin{cases} deg(v_i), i = j \\ -1, \{v_i, v_j\} \in E \\ 0, else \end{cases}$$

This matrix is called the Laplacian due to its relation to the Laplacian operator. From the definition it is clear to see that L is symmetric with real valued matrix. Thus L is diagonizable hence, $L = U\Sigma U^T$ with Σ diagonal and U orthogonal. Note the column vectors of U form a basis of eigenvectors L and the diagonal entries of Σ consist of the corresponding eigenvalues. Furthermore all of these eigenvalues are real. It can be further shown that L is positive semi-definite thus, the eigenvalues of L are non-negative. Let us denote the i'th eigenvector as v_i and the i'th eigenvalue as λ_i

Now let us consider functions on the graph, more specifically functions of the form $f : V \mapsto R$, were V is the vertex set of G. One can think of these functions as a column vector. One can now define the Graph Fourier transform of a function: $\hat{f}(\lambda_j) =< f, v_j >$, where $< ., . >$ is the inner product, note this function is only defined for eigenvalues of L. Let us now note that the graph Fourier transform has an inverse, $f(l) = \sum_{i=0}^{N-1} \hat{f}(\lambda_i) v_{l_i}$. It can be shown that Parseval's identity holds, that is let $f, g : V \mapsto R$ then $< f, g >=< \hat{f}, \hat{g} >$. One can use this identity to define a graph convolution as follows

$$(f * g)(l) = \sum_{i=0}^{N-1} \hat{f}(\lambda_i)\hat{g}(\lambda_i) v_{i_l}$$

Graph Neural Networks have gained significant attention in relation to traffic forecasting and have demonstrated promising results [33, 36]. However, in order to evaluate whether Spatio-Temporal Graph Neural Networks are an appropriate architecture for our problem we must examine them in relation to our dataset. While representing we can represent our data as a graph, in doing so we may lose information about the dataset. While we can encode the distance between two sensors in the weight of their connection, it is difficult to encode their relative locations and therefore their directional dependencies. These directional dependencies influence traffic flow for instance, traffic typically flows into London in the morning and out in the evening. Therefore, knowing a sensors' relative location to surrounding sensors can inform the model on its spatio-temporal relationships. While, as evidenced by the strong results of Graph based approaches to traffic forecasting, the lack of directional dependencies does not prevent a model from producing accurate traffic forecasts, given the small size and high noise of the SCOOT dataset,

learning individual relationships between sensors could be significantly more challenging than learning general traffic flow patterns. Therefore, we may find we get better results from models that are able to exploit this relative location based inductive bias. Consequently, it was decided that Spatio-Temporal Graph Convolutional Network would not be an appropriate choice of architecture for our task.

4.2 Vision Transformer

The weakness of the STGCN architecture was that it did not fully exploit the known spatial dependencies of the sensors. As our dataset is relatively small with only two months worth of data, we would like a model that exploits the full structure of the dataset, thereby minimising the volume of information the model needs to learn. As discussed in the Data Exploration section, each time-slice of our data can be interpreted as an image. Computer vision methods are designed to exploit the spatial relationships that exist in image data. Thus, the application of computer vision techniques addresses the shortcomings of the STGCN when applied to the SCOOT dataset. The current state of the art in computer vision methods is the Vision Transformer (ViT) and therefore these were the next architecture considered.

Transformers are a relatively recent architecture initially developed for natural language processing tasks [37]. Transformers exhibit the capability to manage sequential data without relying on recurrent structures. This attribute enables parallel processing during training, resulting in significantly expedited training duration. A detaied description of the transformer architecture can be found in [38].

While the transformer architecture is not directly applicable to computer vision tasks, the Vision Transformer (ViT), introduced in [39] is a transformer designed for computer vision tasks. As Vision Transformers are the state of the art in computer vision tasks, consideration was given to using them in the current study. However as discussed in [39], transformers lack certain inductive biases found in Convolutional Neural Networks (CNNs), such as translation equivariance and locality. Consequently, this absence of biases may impede the generalization performance of transformers when trained on limited datasets. Further, as discussed in [40], Transformers can be highly vulnerable to noise in datasets. Therefore, as the SCOOT dataset is both small and noisy, Vision Transformers were not considered appropriate for modeling the SCOOT dataset.

4.3 Convolutional Long Short-Term Memory

Although Vision Transformers were considered unsuitable for the SCOOT dataset, the concept of approaching the problem as a computer vision task remained appealing. Thus, the search focused on identifying a model with higher noise resistance and a stronger inductive bias. Consequently, the exploration shifted towards the Convolutional LSTM [41]. Convolutional LSTMs were designed to capture long-term spatio-temporal relationships within sequential image data. This is done by integrating the convolution operator within the LSTM mechanism. As opposed to traditional LSTM which have weight matrices, the Convolutional LSTM has convolutional kernels denoted by $\mathbf{K}$. Furthermore, both the hidden state and the input are now two dimensional. In order to aid distinguishing between the the hidden state vector, the hidden state matrix will be represented as $H_t \in \mathbb{R}^{h_1 \times h_2}$. Mathematically, a Convolutional LSTM unit can be expressed with the equations:

$$\begin{aligned}
\mathbf{i}_t &= \sigma(\mathbf{K}_{xi} * \mathbf{X}_t + \mathbf{K}_{hi} * \mathbf{H}_{t-1} + \mathbf{b}_i) \\
\mathbf{f}_t &= \sigma(\mathbf{K}_{xf} * \mathbf{X}_t + \mathbf{K}_{hf} * \mathbf{H}_{t-1} + \mathbf{b}_f) \\
\mathbf{C}_t &= \mathbf{f}_t \circ \mathbf{C}_{t-1} + \mathbf{i}_t \circ \tanh(\mathbf{K}_{xc} * \mathbf{X}_t + \mathbf{K}_{hc} * \mathbf{H}_{t-1} + \mathbf{b}_c) \\
\mathbf{o}_t &= \sigma(\mathbf{K}_{xo} * \mathbf{X}_t + \mathbf{K}_{ho} * \mathbf{H}_{t-1} + \mathbf{b}_o) \\
\mathbf{H}_t &= \mathbf{o}_t \circ \tanh(\mathbf{C}_t)
\end{aligned} \tag{4.3.1}$$

Convolutional LSTMs are highly resistant to missing data since both the spatial and temporal dimension of the dataset can be used to gain information about missing data points. The convolution operator allows the Convolutional LSTM to extract information about the surroundings of a cell even if that cell is missing data. Further, the LSTM mechanism allows the Convolutional LSTM to retain relevant information from previous time steps which can be used to infer information about missing data points. Alongside the Convolutional LSTMs resilience to missing data, due to the use of convolutional kernels, the Convolutional LSTM also possesses a strong inductive bias allowing it to better train on small datasets. Therefore, a Convolutional LSTM overcomes the weaknesses affecting the Vision Transformer when applied to the SCOOT dataset.

As discussed the Convolutional LSTM architecture is highly applicable to the task however, there are still disadvantages with the Convolutional LSTM that make it less suitable for our task. These issues stem from the use of the convolution operator being applied to both the input data and to the hidden state. As mentioned, the convolution operator exploits the strong spatial regularity exhibited in most image data. This is often a desired

property for tasks involving an image since, for example, a dog is a dog independent of whether it is in the top right or bottom left hand side of an image. However, the SCOOT data may not exhibit as much spatial regularity as most images do. To illustrate this consider traffic flowing out of London at the end of the day. In the East of London, traffic will generally flow out of London to the East, whereas in the West of London, traffic will generally flow out of London to the West. Therefore, in the same image we expect there to be sections of the image that exhibit different spatial relationships. By only using convolutions, as convolutions are spatially invariant, it may be more difficult to capture the different relationships in different sections of the image.

Further using only convolutions also causes restraints to be placed on the locality of each pixel. In a single convolutional layer, a pixel can only affect another pixel for which it is in the kernel of. However, while in general local grid cells affect the traffic locally, the relevant neighbourhoods of our grid cells may still vary. For example, if one grid cell was on a motorway we would expect it to have an impact on grid cells much further away as opposed to a grid cell in a residential neighbourhood. Further, when a convolution is repeatedly applied to the hidden state over successive time steps, the signal from a previous hidden state can become diffused across the spatial dimension [42]. Therefore, potentially the unique temporal features of each cell may be difficult to capture due to the repeated convolution of the hidden state. For these reasons, the Convolutional LSTM architecture may not be the most suitable architecture for the SCOOT dataset.

4.4 Semi-Convolutional LSTM

As discussed in the previous section, Convolutional LSTMs are highly applicable to the task of modeling the SCOOT dataset due to their robustness against noise and inherent inductive bias. However, due to the use of convolutions on both the input data and hidden state too strong a spatial regularity is inherent in the model. To address the limitations associated with the Convolutional LSTMs application to the SCOOT dataset, we designed the Semi-Convolutional LSTM architecture. This architecture strikes a balance between leveraging the convolutional operator to exploit the expected spatial regularity in the input while using matrix multiplication in the hidden state transition matrix allowing for spatial irregularity to be modeled. This allows for strong relationships to exist between distant grid cells to exist in the Semi-Convolutional LSTM. Moreover, as the Semi-Convolutional LSTM uses matrix multiplication for the hidden state transitions, unique tempo-

ral relationships of individual grid cells may be easier to capture, enhancing the models performance. The Semi-Convolutional LSTM can be thought of as an intermediary between the LSTM and the Convolutional LSTM, using convolutional kernels for the input data and a weight matrix for the hidden state transitions.

The use of convolutional kernels for the input and matrix multiplication for the hidden state posed a challenge. Applying a convolutional kernel to a two dimensional input gives a two dimensional output, while in order to allow for relationships to exist between all grid cells via the hidden state transition, the hidden state must be a one dimensional vector. This problem can be solved due to the isomorphism that exists between $\mathbb{R}^{d_1 \times d_2}$ and $\mathbb{R}^{d_1 \cdot d_2}$. This means that during matrix multiplication, we treat the hidden state as a vector. Afterwards, we reshape the resulting vector back into a matrix, allowing us to add the result to the result of the convolution on the input data. Further technical detail about the reshaping of vectors and the incorporation of multiple convolutional filters can be found in Section 6. In order to increase the interpretability of the equations, the complications that arise from the need to reshape the hidden state is ignored, therefore the equations for the Semi-Convolutional LSTM can be expressed as follows:

$$\begin{aligned}
\mathbf{i}_t &= \sigma(\mathbf{K}_{xi} * \mathbf{X}_t + \mathbf{W}_{hi}\mathbf{h}_{t-1} + \mathbf{b}_i) \\
\mathbf{f}_t &= \sigma(\mathbf{K}_{xf} * \mathbf{X}_t + \mathbf{W}_{hf}\mathbf{h}_{t-1} + \mathbf{b}_f) \\
\mathbf{C}_t &= \mathbf{f}_t \circ \mathbf{C}_{t-1} + \mathbf{i}_t \circ \tanh(\mathbf{K}_{xc} * \mathbf{X}_t + \mathbf{W}_{hc}\mathbf{h}_{t-1} + \mathbf{b}_c) \\
\mathbf{o}_t &= \sigma(\mathbf{K}_{xo} * \mathbf{X}_t + \mathbf{W}_{ho}\mathbf{h}_{t-1} + \mathbf{b}_o) \\
\mathbf{h}_t &= \mathbf{o}_t \circ \tanh(\mathbf{C}_t)
\end{aligned} \tag{4.4.1}$$

Semi-Convolutional LSTMs are designed to be highly applicable to the task of forecasting with the SCOOT dataset. This is due to the fact that they are able to capture long-term temporal dependencies through the use of the LSTM mechanism while being able to exploit the spatial irregularity while also having the ability to model spatial irregularity.

5 Benchmarking

Determining a robust benchmark is an integral part of this project, serving as an important tool for the evaluation of the model. The dataset contains a high level of noise stemming from factors such as missing sensor readings, erroneous sensor readings and exogenous factors such as road closures or traffic accidents, thereby making non-comparative evaluation very difficult. Developing a benchmark helps to set measurable project goals, transforming the qualitative goal of creating a high performance model into the quantitative goal of outperforming the benchmark. There is no single universally suitable benchmark, and determining an appropriate benchmark is a key part of the project [43].

5.1 Naive Forecasting

The main factor that affects traffic conditions is the time of day and as a result traffic conditions exhibit a strong 24 hour periodicity. We can exploit this periodicity to produce a forecast for each grid cell by simply using the reading from the same grid cell 24 hours previously. This baseline performs better as further discussed in Section 8. One might expect reduced performance to due to variation in traffic across different weekdays. However, due to a phenomenon known as induced demand [44], the traffic conditions remain similar across weekdays and weekends and therefore this naive forecasting provides a solid benchmark.

5.2 Convolutional LSTM

When evaluating the success of the semi-convolutional LSTM architecture, the convolutional LSTM architecture can be used as a benchmark. As discussed in Section 4.3, convolution LSTMs are able to exploit the strong spatial regularity of an image. However, they may be susceptible to temporal diffusion and difficulty capturing non-local relationships. Therefore, a comparison of the results from semi-convolutional LSTMs and convolutional LSTMs may help us to gain some insight into whether a semi-convolutional LSTM can solve these limitations. Furthermore, convolutional LSTMs have been heavily studied and are have been shown to produce accurate forecasts similar problems such as precipitation forecasting [41].

6 Implementation

The Semi-Convolutional LSTM was written to be directly integrated into the Keras library. This approach was achieved by extending the existing "ConvLSTM" and "convLSTMCell" classes from Keras and overwriting the necessary methods such as call and build. This provided several advantages. Firstly, due to its inheritance from Keras classes, Semi-Convolutional LSTM seamlessly integrates into the Keras library facilitating model development, training, and deployment. Secondly, a high standard of code was maintained since the "ConvLSTM" class has been optimised and widely tested. Therefore, the base code upon which the Semi-Convolutional LSTM was built upon was heavily optimised and tested and so there were afar fewer potential points of inefficiency or failure compared to writing the code from scratch. Finally, it allowed me to maintain a minimal number of external dependencies. This was very important for the project, as ideally the code should not extend the list of dependencies needed for the London Air Quality Project's code base. As Keras and TensorFlow more generally are already dependencies of the London Air Quality Project, using the Keras library allowed for the implementation of the model without extending the list of dependencies. Now that we have discussed the coding practices used, let us discus some of the intricacies in the design.

6.1 Multiple Filters

Having multiple filters is crucial for capturing diverse spatial features and patterns data. Each filter is able to capture specific patterns allowing the model to lean complex representations of the data. However, there were challenges when implementing multiple filters in the Semi-Convolutional LSTM.

Initially each convolutional filter was designed to have it's own set of hidden state transition matrices. However, in a Semi-Convolutional LSTM the size of the hidden state tends to be quite large since $\dim(\mathbf{h}_t) \cong \dim(\mathbf{K} * \mathbf{X}_t)$. This causes the weight matricides to be large as matrix multiplication of the hidden state by the hidden state transition matrix must retain the dimensionality of the hidden state. To give an example of this, for the Semi-Convolutional LSTMs application to the SCOOT dataset, as $\mathbf{K} * \mathbf{X}_t \in \mathbb{R}^{32\times32}$, each of the weight matrices, $\mathbf{W} \in \mathbb{R}^{1024\times1024}$. Due to the large size of these matrices, it was clear that this would be computationally infeasible. Therefore, as a compromise between performance and computational cost, while increasing the number of filters of the model would increase the number of convolutional kernels used, only one set of hidden state transition weight matrices would

be used. This approach enabled us to leverage multiple convolutional kernels without the computational cost associated with model training and execution to drastically increase.

6.2 Reshaping of the Hidden State

As discussed in Section 4.4, the hidden state in the Semi-Convolutional LSTM equations must be interpreted as both a vector and a matrix at different stages. When implementing the Semi-Convolutional LSTM in TensorFlow, this is addressed by reshaping the hidden state tensor as needed. In TensorFlow, reshaping a tensor does not change the ordering of its elements, therefore, repeated reshaping of a tensor within the Semi-Convolutional LSTM cell does not cause any issues.

Algorithm 4 Semi-Convolutional LSTM Cell

$\mathbf{h}_{t-1} \leftarrow$ Tensor of shape [height, width]
$\mathbf{h}'_{t-1} \leftarrow \mathbf{h}_{t-1}$ reshaped to a tensor of shape [height $\cdot$ width]
$\mathbf{R}'_i \leftarrow \mathbf{W}_{xi}\mathbf{h}'_{t-1}$
$\mathbf{R}'_f \leftarrow \mathbf{W}_{xf}\mathbf{h}'_{t-1}$
$\mathbf{R}'_c \leftarrow \mathbf{W}_{xc}\mathbf{h}'_{t-1}$
$\mathbf{R}'_o \leftarrow \mathbf{W}_{xo}\mathbf{h}'_{t-1}$
$\mathbf{R}_i \leftarrow \mathbf{R}'_i$ reshaped into a tensor of shape [height, width]
$\mathbf{R}_f \leftarrow \mathbf{R}'_f$ reshaped into a tensor of shape [height, width]
$\mathbf{R}_c \leftarrow \mathbf{R}'_c$ reshaped into a tensor of shape [height, width]
$\mathbf{R}_o \leftarrow \mathbf{R}'_o$ reshaped into a tensor of shape [height, width]
$\mathbf{i}_t \leftarrow \sigma(\mathbf{K}_{xi} * \mathbf{X}_t + \mathbf{R}_i + \mathbf{b}_i)$
$\mathbf{f}_t \leftarrow \sigma(\mathbf{K}_{xf} * \mathbf{X}_t + \mathbf{R}_i + \mathbf{b}_f)$
$\mathbf{C}_t \leftarrow \mathbf{f}_t \circ \mathbf{C}_{t-1} + \mathbf{i}_t \circ \tanh(\mathbf{K}_{xc} * \mathbf{X}_t + \mathbf{R}_c + \mathbf{b}_c)$
$\mathbf{o}_t \leftarrow \sigma(\mathbf{K}_{xo} * \mathbf{X}_t + \mathbf{R}_o + \mathbf{b}_o)$
$\mathbf{h}_t \leftarrow \mathbf{o}_t \circ \tanh(\mathbf{C}_t)$

Algorithm 4 shows how by reshaping the hidden state as required, we can ensure that the dimensions match during the addition of the convoluted input and matrix multiplied hidden state. Now that we have discussed the technical details behind the implementation of the Semi-Convolutional LSTM, we are able to move on to discussing the application of the Semi-Convolutional LSTM with respect to the SCOOT dataset.

7 Hyper-Parameter Selection

In this section, we explore the process of selecting hyper-parameters for the Semi-Convolutional LSTM Traffic forecasting model. Semi-Convolutional LSTMs are computationally expensive to train, this is not a problem unique to the Semi-Convolutional LSTM architecture but rather an issue inherent to the Convolutional LSTM [45] on which Semi-Convolutional LSTMs are based. Consequently, traditional hyper-parameter selection methods such as grid-search were not available due to resource constraints. Therefore, we had to use knowledge of the problem alongside small scale trials to determine appropriate hyper-parameters.

7.1 Choice of Loss function

For this project a suitable loss function should be robust to outliers. This is because there is not only noise in the data set from erroneous readings and missing data-points but also in the data generation itself. Traffic data can be highly influenced by unpredictable events such as accidents and external factors such as scheduled road closures, for which we do not information in the dataset. This means there will be data points for which our dataset is not predictive. If a loss function like mean squared error is chosen, which is not robust to outliers, it may cause our model to over-fit to the training set and not generalise or converge. Therefore, as outliers are expected in both the training and any test dataset, a loss function that is robust to outliers should be used to train our model. The Mean Absolute Error (MAE) was selected for our model as it is significantly less sensitive to outliers than other commonly used loss functions, such as the Root Mean Squared Error.

7.2 Data Split

In this section, we discuss the decision process of how the dataset was split into training, validation and test sets. This decision is very influential in both the performance of the model and the subsequent hyper-parameter selection process. Especially considering the relatively small size of the SCOOT dataset, it is beneficial for the training set to be as large as possible. However, if the training set takes up a high proportion of the dataset, the results from both the validation and test sets may be inaccurate which could lead to inaccurate hyper-parameter selection and model evaluation. Therefore, we decided upon a 80:10:10 split of the dataset.

Now that we have determined the proportions of the training, validation and

test set, the next step is to address how we partition the data points into these sets. A common approach is to randomly allocate data points to each set, however, this method is not suitable to our task. This is due to the sequential nature of our data, the model uses the previous 7 days worth of data to produce the forecast for the next hour. Let us give an example to illustrate the issues this causes, say we have one data point in the test set and the subsequent data point being in the training set, then during training as the subsequent point is in the training set, the model is optimised on information about the point in the test set. Thereby, invalidating the test and validation set. For this reason, we have to allocate concurrent chunks of data to each of the sets.

This introduces additional complexities since, as discussed in Section 3.1, the number of missing data points is an increasing function of time. Consequently, when allocating concurrent chunks of data to the training, validation and test sets, each will exhibit different levels of noise. Given that the number of missing data points increases over time, and the model is intended to be used for forecasting current traffic conditions, the end portion of the dataset was allocated to the test set. Using the end portion of the dataset as the test set allows us to get an estimation of the models performance under conditions as similar to the ones expected during model deployment. Regarding the training and validation split, the validation set was positioned in the middle of the training set, as this would allow the training set to have on average the most similar noise levels to the training set.

7.3 Number of Filters

In this section we experiment with varying the number of filters in the first layer of our two layer models. Due to the increase computational cost associated with training three layer models, we were not able to experiment with varying the filter count on them.

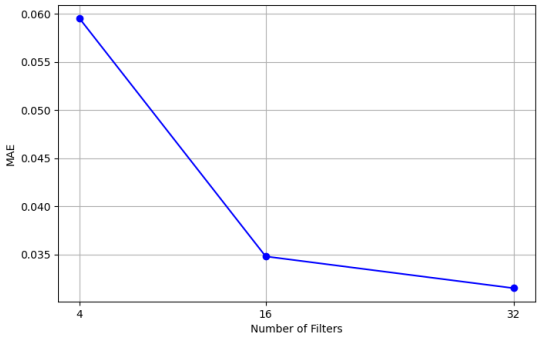


Figure 11: Plot showing the MAE of the validation datasets vs. number of filters used in each layer of the convolution

As depicted in Figure 11, we observed that when the filter count increases, the models accuracy on the validation test set increases. Consequently, when evaluating models on the test set, we opted to use models with 32 filters. Although it would have been beneficial to explore how the number of filters impacted three layer models and higher filter counts, due to the computational expense of training these large models, we were not able to do so.

7.4 Number of Layers

Experimentation of models with a higher number of layers was limited due to the high computational costs associated with their training. As a result, we were limited to exploring two and three layer models. Furthermore hyper-parameter testing would be extremely costly for three layer models, with the training of a three layer model exceeding a days worth of computational time. Consequently, we only examined one three-layer model, this was chosen to have 32 filters in the first and second layer and then one filter in the final layer. This was chosen due to the increased performance of the 32 filter two-layer Semi-Convolutional LSTM over smaller models.

7.5 Additional hyper-parameters

In this section we discus the selection of several remaining hyper-parameters. One such hyper-parameter is the number of epochs used to train the model. If this hyper-parameter is set too low, the model may not have time to converge during training, if this hyper-parameter is too high, the model may

overfit damaging its ability to generalise. To ensure the model had time to converge, we opted to use 100 epochs. While during training, convergence often happened around the 60^{th}, since early stopping was implemented, there were did not appear to be issue with overfitting. Furthermore, we incorporated a reduction of learning rate on plateaus to enable the model to better converge to its local minimum.

The decision to use the ADAM optimiser, introduced in introduced in Section 2.2.1, was driven by the significant presence of noise in our dataset. The ADAM optimiser is particularly suitable for tasks involving noisy gradients [20]. As the ADAM optimiser is based on the Stochastic Gradient Descent algorithm, we must also determine what batch size to use. We opted to use a batch size of 64 to achieve a balance between reducing training times and ensuring accurate gradient estimation. Now that we have discussed the hyper-parameters, let us inspect the training of a Semi-Convolutional LSTM model.

7.6 Model Training

This section examines the training of the three layer Semi-Convolutional LSTM. The first two layers have 32 filters and the final output layer of the model has 1 filter. This specific architecture was chosen for evaluation of training as it was both the largest model trained and had the lowest validation loss of all models tested. Figure 12 suggests that the batch size hyper-parameter may have been too low. This is indicated by spikes in the training loss such as the one we see at epoch seven. However, this does not seem to have had a material impact on the model as both the training loss and validation loss converged.

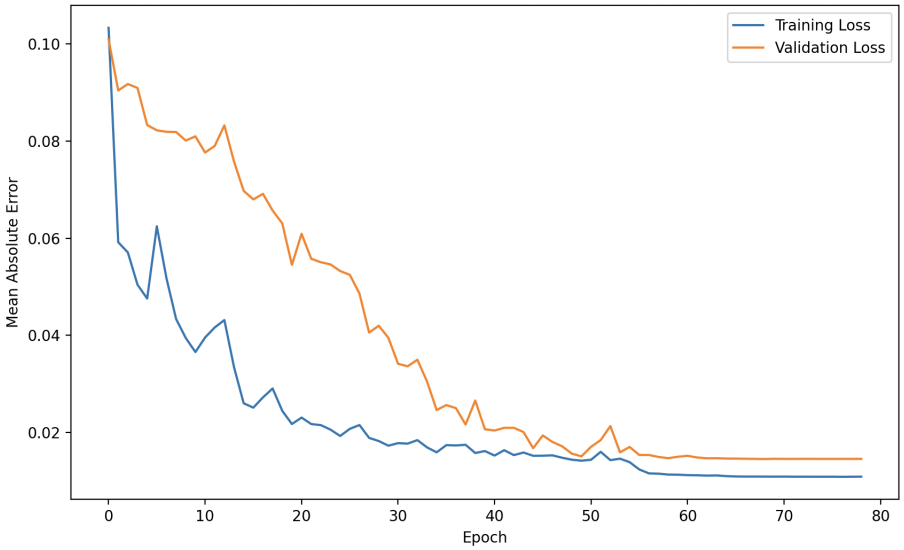


Figure 12: Loss Curve

Upon examination of the loss curve, there are some encouraging indicators. Notably, after roughly 55 epochs, both the training loss and the validation loss plateau. The leveling off of the validation loss curve suggests that the model has not overfit to the data and therefore suggests that the model may generalise well onto a test set.

8 Results

This section presents an evaluation of the application of the Semi-Convolutional LSTM to the SCOOT dataset. In particular, we demonstrate its improvement against the Convolutional LSTM, with the Semi-Convolutional LSTM outperforming the Convolutional LSTM by over 8% across each of the MAE, MSE and RMSE.

In this section, the interpretability of results is crucial. To ensure that our findings are easily accessible and interpretable, we have chosen to use the error metric calculated on the de-normalised values. This means that the results are provided in real world highly interpretable terms for example the MAE of the de-normalised values is simply the mean number of cars our prediction was off from the ground truth.

8.1 Model Performance

In this section we evaluate the performance of the Semi-Convolutional LSTM against our benchmark forecasting methods considering the MEA, RMSE and MSE across the test data set.

Model Architecture	MAE	RMSE	MSE
Semi-Convolutional LSTM 32f $\rightarrow$ 32f $\rightarrow$ 1f	**26.20**	**44.50**	**1980**
Convolutional LSTM 32f $\rightarrow$ 32f $\rightarrow$ 1f	**28.57**	**48.67**	**2368**
Semi-Convolutional LSTM 32f $\rightarrow$ 1f	**36.94**	**57.12**	**3260**
Convolutional LSTM 32f $\rightarrow$ 1f	**54.98**	**89.21**	**7959**
Naive Forecast	**79.9**	**158.6**	**25156**

Table 1: Comparative Results of Benchmark Models and the Semi-Convolutional LSTM on the Test Dataset

The results depicted in Table 1 show an increase in performance provided by the Semi-Convolutional LSTM over the naive forecasting methods across all tested machine learning architectures. Indeed, the three-layer Semi-Convolutional LSTM offers over a 67% improvement across all error metrics against the naive forecast.

The findings indicate that by using three layers as opposed to two in both the Convolutional LSTM and the Semi-Convolutional LSTM, we improve performance on the test dataset. Specifically, the three-layer Convolutional

LSTM offers over a 45% improvement in all error metrics compared to the two-layer Convolutional LSTM. Similarly, the three-layer Semi-Convolutional LSTM offers over a 22% improvement in all error metrics against its two-layered counterpart.

Of particular importance to this project is the comparison between the Semi-Convolutional LSTM and Convolutional LSTM architectures. Notably, the three-layer Semi-Convolutional LSTM offers an over 8.30% improvement across all error metrics when compared to the three-layer Convolutional LSTM. Remarkably, the two-layer Semi-Convolutional LSTM offers an over 32% improvement compared to the two-layer Convolutional LSTM across all metrics. The strong performance of the Semi-Convolutional LSTM is further illustrated in Table 2 which shows the percentage improvements against the baselines. These results underscore the considerable potential of the Semi-Convolutional LSTM in traffic forecasting tasks using the SCOOT dataset.

Model Architecture	Percentage Improvement		
	MAE	**RMSE**	**MSE**
ConvLSTM 32f $\rightarrow$ 32f $\rightarrow$ 1f	**8.30**	**8.57**	**16.39**
Semi-ConvLSTM 32f $\rightarrow$ 1f	**29.07**	**22.09**	**39.26**
ConvLSTM 32f $\rightarrow$ 1f	**52.35**	**50.12**	**75.12**
Naive Forecast	**67.21**	**71.94**	**92.13**

Table 2: Percentage improvements of the MAE, RMSE, and MSE for the three-layer Semi-Convolutional LSTM compared to the other architectures

8.2 Ablations

In this section, we explore the relationship that exists between the dataset size and model performance. The goal of this is to illustrate how the relatively small size of the SCOOT dataset may have constrained the models performance.

This experimentation was performed using a two-layer architecture, with 32 features in the first layer. Notably, we chose not to use the three-layer architecture which performed better on the full dataset. This was due to the fact that larger models generally need more data to train effectively. With less data, such models can often overfit to the training dataset. Therefore, for smaller datasets, simpler models tend to generalise better. Alongside this the computational cost of the two layer model is significantly smaller than the computational cost of the three layer model allowing for faster training.

Given our goal is to accurately assess how different dataset sizes impact the model's ability to forecast the SCOOT dataset, it was deemed more appropriate to use a model that aligns better with the constraints of smaller datasets.

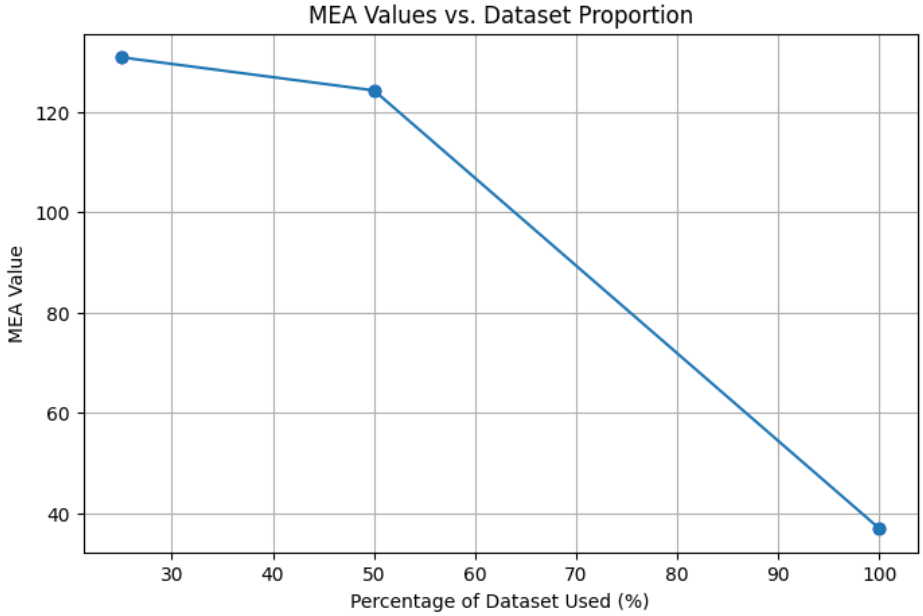


Figure 13: MAE vs. Dataset Size

Figure 13 clearly shows that the models performance improves with increasing dataset size. This is consistent with machine learning principles as larger datasets tend to provide a more comprehensive representation of the underlying data generating process. Consequently, the model is able to learn more robust and general patterns when trained on larger datasets which leads to lower error rates on unseen data. Extrapolating the observed trend, we can hypothesise that if we were able to extend the SCOOT dataset, we may be able to better forecast traffic conditions.

9 Future Work

9.1 Moving MNIST

As previously discussed the dataset for this project is both small and noisy. Therefore, it is difficult to draw any strong conclusions about the general performance and merits of the Semi-Convolutional LSTM as an architecture. Therefore, in order to further explore the merits of the Semi-Convolutional LSTM architecture, it would be useful to study the Semi-Convolutional LSTM's performance on a standardised dataset. In [41], the Moving MNIST dataset is used as a standardised dataset to benchmark the Convolutional LSTM. The Moving MNIST dataset is an extension of the traditional MNIST dataset [46], designed to asses the performance of computer vision models in sequence prediction task. The Moving MNIST dataset comprises of 20 frame videos depicting two MNIST digits in motion.

The Moving MNIST dataset serves as an extension of the conventional MNIST dataset, specifically tailored for assessing the performance of computer vision models in sequence prediction tasks. It comprises brief video sequences depicting two MNIST digits in motion within an image frame, undergoing movement trajectories that involve bouncing off the frame edges.

By evaluating the performance of the Semi-Convolutional LSTM on the Moving MNIST dataset, we would get an interesting comparison of the Semi-Convolutional LSTM not only to the convLSTM but to many other models as well. However, evaluation of the performance of the Semi-Convolutional LSTM was outside the scope of this project due to time constraints.

9.2 Further hyper-parameter optimisation

Computational limitations constrained our ability to explore four-layer models, as well as prohibiting an in-depth investigation into three layer models and a grid search of hyper-parameters for two layer models. This means that we have likely not found the overall optimum configuration of hyper-parameters and therefore a natural continuation of the thesis would involve in a grid search for optimal hyper-parameters alongside more experimentation with larger models.

9.3 Ensemble Methods

Ensemble methods in machine learning involve aggregating multiple models in order to enhance performance. By harnessing multiple models as opposed

to a single one, ensemble methods are often more accurate and robust than individual models [47]. Additionally ensemble methods also can enable accurate uncertainty estimation [48]. There are many types of ensemble methods, one such method is called bagging. In bagging, multiple models are trained on different subsets of the data [49]. This approach may be of interest for SCOOT forecasting, as it would allow models to be trained on data segments with varying levels of missing data-points. Subsequently, these models could be combined using a weighted average dependent on the number of missing data points in the segment of data being used. This would allow us to produce a forecast using models that had the appropriate trade off between accuracy and resilience to noise. However, the current size of the SCOOT dataset prohibits this technique. As illustrated in Section 8.2, reducing the size of the dataset significantly decreases model performance.

10 Conclusion

Through this project, I have gained a strong insight into machine learning, gaining a comprehensive understanding of the mathematical background underpinning various key concepts such as back propagation. The practical experience I gained from creating and implementing machine learning models has set me in good stead to undertake machine learning projects in the future.

Further, this project has taught me the importance of a detailed dataset exploration. The volume of missing data posed a substantial challenge guiding a large amount of the decision making process and if a detailed dataset exploration had not occurred at the beginning stages of the project this could have lead to major setbacks. The challenges faced due to missing entries added significant complexity to the project but were ultimately successfully identified and overcome.

The challenges faced during hyper-parameter selection due to the high computational cost of the Semi-Convolutional LSTM and the Convolutional LSTM have taught me the importance of managing computational resources and the potential value of lightweight architectures.

In reflection, I am very pleased with the outcomes of this project. The development of a novel architecture that provides over a 8% improvement on the Convolutional LSTM was far outside my expectations at the beginning of this project. The successful completion of the project despite the added complexity highlighted strong project management which will be further discussed.

10.1 Project Management

At the start of the project, the goals of the project were decided and discussed in the research specification. The first stage of the project was a literature review. This began with a focus on building a strong background in technical aspects of machine learning such as back propagation. One the background knowledge had been acquired, the focus shifted to specific architectures that may be applicable to the project. This greatly aided the architecture selection portion. Without both strong fundamentals and knowledge of specific architectures that could be used for traffic forecasting, it would not have been possible to develop the idea of the Semi-Convolutional LSTM. This section of the project spanned the first term including the Christmas holidays. This portion of the project was successful not only aiding the current project but

also laying the foundations for similar projects in the future.

The second stage of the project involved a data exploration allowing me to gain insight into possible data cleaning methods and aiding architecture selection. This stage offered the first unexpected challenge, with the SCOOT data set being much more noisy than expected. Since the data exploration was undertaken early in the project, there was plenty of time to research and ultimately overcome this unexpected challenge. For this reason this stage of the project has been deemed successful.

The third stage of the project involved deciding upon and implementing an architecture. An agile design methodology was used allowing me to rapidly develop, test and deploy architectures. The complexity of this section was far greater than expected due to the creation of a new architecture as opposed to the implementation of an existing one. This provided valuable insights into the architecture development process which will be invaluable moving forward. In this stage the Semi-Convolutional LSTM was both developed and implemented. While this was outside the scope of the original project, with increased resources devoted to the project, I was able to meet the original timetable created for the project specification. Therefore, this stage of the project was deemed successful. Further, while implementing the Semi-Convolutional LSTM an emphasis was put on minimising dependencies

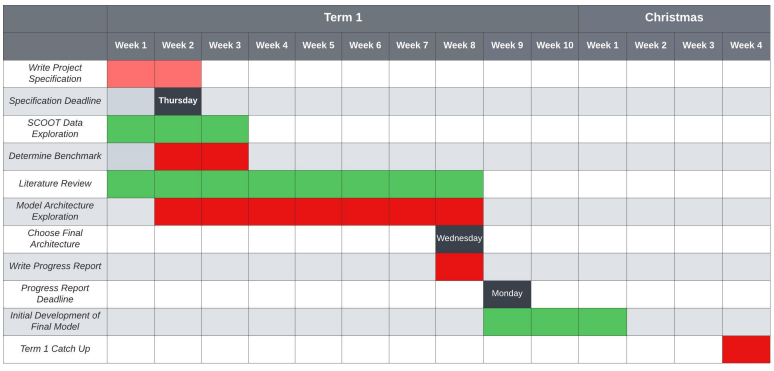


Figure 14: Project timeline for first term

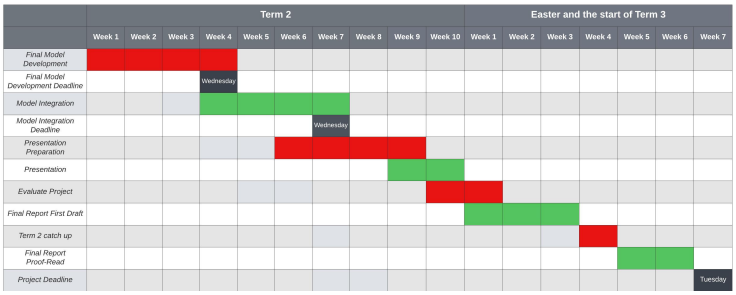


Figure 15: Project timeline for second term

During the project, I was able to stick to the timeline provided for both first term 15 and second terms 14. This was aided by meetings with both Professor Theo Damoulas and Sueda Ciftci which were incredibly influential for the projects success. These meetings offered guidance when unexpected obstacles emerged.

10.2 Ethical concerns

Forecasting data intended for public use comes with many ethical and legal concerns, the majority of these concerns come from one of the three following areas: biases, imperfect forecasts, and the interpretation of the models results. To give an understanding of the sorts of ethical problems that may emerge imagine the following scenario: there is a borough on the outskirts of London, for which, due to a lack of funding, there are very few SCOOT sensors. As our model has less data from this borough, one would expect the predictions to be worse. Let us imagine that our model consistently predicts a much lower level of traffic than is the true level for the borough. This will then get passed onto the London Air Quality model, causing the air pollution forecast to predict a lower level of air pollution than the borough actually has. This would have a chain effect, that may cause the air-quality related issues in the borough to be overlooked. Throughout the project, such ethical issues were considered for example, in Section **??** we discussed the ethical ramifications surrounding using K-means clustering and how it could lead the model to prioritise accurate traffic forecasting in the center of London at the expense of the outskirts. While we managed to address some concerns, many of the ethical issues arise due to the data being supplied and so are sadly outside the scope of the project to address.

References

1. Vlahogianni, E. I., Karlaftis, M. G. & Golias, J. C. Short-term traffic forecasting: Where we are and where we're going. *Transportation Research Part C: Emerging Technologies* **43.** Special Issue on Short-term Traffic Flow Forecasting, 3–19. ISSN: 0968-090X. https://www.sciencedirect.com/science/article/pii/S0968090X14000096 (2014).
2. Papageorgiou, M. & Kotsialos, A. Freeway ramp metering: an overview. *IEEE Transactions on Intelligent Transportation Systems* **3,** 271–281 (2002).
3. *Understanding the Health Effects of Comonents of the Particulate Matter Mix : Progress and Next Steps* in (2002). https://api.semanticscholar.org/CorpusID:212668276.
4. Institute, T. A. T. *London Air Quality* https://www.turing.ac.uk/research/research-projects/london-air-quality. Accessed: 07/10/2023.
5. City of London. *Clean Air Route Finder* https://www.london.gov.uk/programmes-strategies/environment-and-climate-change/pollution-and-air-quality/clean-air-route-finder. Accessed: 2024-04-01. 2024.
6. I, M., E, S., A, S. & E., B. Environmental and Health Impacts of Air Pollution: A Review. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC7044178/ (Feb. 2020).
7. Shekhar, S., Evans, M. R., Kang, J. M. & Mohan, P. Identifying patterns in spatial information: A survey of methods. *WIREs Data Mining and Knowledge Discovery* **1,** 193–214. eprint: https://wires.onlinelibrary.wiley.com/doi/pdf/10.1002/widm.25. https://wires.onlinelibrary.wiley.com/doi/abs/10.1002/widm.25 (2011).
8. Yun, S.-Y., Namkoong, S., Rho, J.-H., Shin, S.-W. & Choi, J.-U. A Performance evaluation of neural network models in traffic volume forecasting. *Mathematical and Computer Modelling* **27,** 293–310. ISSN: 0895-7177. https://www.sciencedirect.com/science/article/pii/S089571779800065X (1998).
9. Newbold, P. ARIMA model building and the time series analysis approach to forecasting. *Journal of Forecasting* **2,** 23–35. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/for.3980020104. https://onlinelibrary.wiley.com/doi/abs/10.1002/for.3980020104 (1983).

10. Polson, N. G. & Sokolov, V. O. Deep learning for short-term traffic flow prediction. *Transportation Research Part C: Emerging Technologies* **79,** 1–17. ISSN: 0968-090X. http://dx.doi.org/10.1016/j.trc.2017.02.024 (June 2017).
11. Yuan, H. & Li, G. A Survey of Traffic Prediction: from Spatio-Temporal Data to Intelligent Transportation. *Data Science and Engineering* **6** (Jan. 2021).
12. Yin, X. *et al.* Deep Learning on Traffic Prediction: Methods, Analysis, and Future Directions. *IEEE Transactions on Intelligent Transportation Systems* **23,** 4927–4943. ISSN: 1558-0016. http://dx.doi.org/10.1109/TITS.2021.3054840 (June 2022).
13. Liu, X. *et al. LargeST: A Benchmark Dataset for Large-Scale Traffic Forecasting* 2023. arXiv: 2306.08259 [cs.LG].
14. Chen, W., Zhao, Z., Liu, J., Chen, P. C. Y. & Wu, X. LSTM Network: A Deep Learning Approach For Short-Term Traffic Forecast. *IET Intelligent Transport Systems* **11** (Jan. 2017).
15. Rosenblatt, F. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review* **65 6,** 386–408. https://api.semanticscholar.org/CorpusID:12781225 (1958).
16. Perantonis, S. & Virvilis, V. Efficient perceptron learning using constrained steepest descent. *Neural networks : the official journal of the International Neural Network Society* **13 3,** 351–64 (2000).
17. Hodnett, M., Wiley, J. F., Liu, Y. & Maldonado, P. *Deep Learning with R for Beginners* (Packt Publishing, 2023).
18. Ruder, S. *An overview of gradient descent optimization algorithms* 2017. arXiv: 1609.04747 [cs.LG].
19. Bottou, L. in *Neural Networks: Tricks of the Trade: Second Edition* (eds Montavon, G., Orr, G. B. & Müller, K.-R.) 421–436 (Springer Berlin Heidelberg, Berlin, Heidelberg, 2012). ISBN: 978-3-642-35289-8. https://doi.org/10.1007/978-3-642-35289-8_25.
20. Kingma, D. & Ba, J. Adam: A Method for Stochastic Optimization. *International Conference on Learning Representations* (Dec. 2014).
21. Rumelhart, D. E., Hinton, G. E. & Williams, R. J. Learning representations by back-propagating errors. *nature* **323,** 533–536 (1986).
22. Misra, D. *Mish: A Self Regularized Non-Monotonic Activation Function* 2020. arXiv: 1908.08681 [cs.LG].
23. Arnekvist, I., Carvalho, J. F., Kragic, D. & Stork, J. A. *The effect of Target Normalization and Momentum on Dying ReLU* 2020. arXiv: 2005.06195 [cs.LG].

24. Hopfield, J. Neural Networks and Physical Systems with Emergent Collective Computational Abilities. *Proceedings of the National Academy of Sciences of the United States of America* **79,** 2554–8 (May 1982).
25. Bird, G. & Polivoda, M. E. *Backpropagation Through Time For Networks With Long-Term Dependencies* 2021. arXiv: 2103.15589 [cs.LG].
26. Chubykalo, A. & Alvarado, R. A Critical Approach to Total and Partial Derivatives. *Hadronic Journal* **25** (Nov. 2002).
27. Pascanu, R., Mikolov, T. & Bengio, Y. *On the difficulty of training Recurrent Neural Networks* 2013. arXiv: 1211.5063 [cs.LG].
28. Bengio, Y., Simard, P. & Frasconi, P. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks* **5,** 157–166 (1994).
29. Zhang, J., He, T., Sra, S. & Jadbabaie, A. *Why gradient clipping accelerates training: A theoretical justification for adaptivity* 2020. arXiv: 1905.11881 [math.OC].
30. Hochreiter, S. & Schmidhuber, J. Long Short-Term Memory. *Neural Computation* **9,** 1735–1780 (1997).
31. Rahuljha. *LSTM Gradients* https://towardsdatascience.com/lstm-gradients-b3996e6a0296. Accessed: April 3, 2024. 2023.
32. He, H. & Garcia, E. A. Learning from imbalanced data. *IEEE Transactions on knowledge and data engineering* **21,** 1263–1284 (2009).
33. Yu, B., Yin, H. & Zhu, Z. *Spatio-Temporal Graph Convolutional Networks: A Deep Learning Framework for Traffic Forecasting* in *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence* (International Joint Conferences on Artificial Intelligence Organization, July 2018). http://dx.doi.org/10.24963/ijcai.2018/505.
34. Shuman, D. I., Narang, S. K., Frossard, P., Ortega, A. & Vandergheynst, P. The emerging field of signal processing on graphs: Extending high-dimensional data analysis to networks and other irregular domains. *IEEE Signal Processing Magazine* **30,** 83–98. ISSN: 1053-5888. http://dx.doi.org/10.1109/MSP.2012.2235192 (May 2013).
35. Hammond, D. K., Vandergheynst, P. & Gribonval, R. *Wavelets on Graphs via Spectral Graph Theory* 2009. arXiv: 0912.3848 [math.FA].
36. Lange, O. & Perez, L. *Traffic Prediction with Advanced Graph Neural Networks* Accessed: April 13, 2024. 2023. https://deepmind.google/discover/blog/traffic-prediction-with-advanced-graph-neural-networks/.
37. Devlin, J., Chang, M.-W., Lee, K. & Toutanova, K. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding* 2019. arXiv: 1810.04805 [cs.CL].

38. Vaswani, A. *et al. Attention Is All You Need* 2023. arXiv: 1706.03762 [cs.CL].
39. Dosovitskiy, A. *et al. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale* 2021. arXiv: 2010.11929 [cs.CV].
40. Passban, P., Saladi, P. S. M. & Liu, Q. *Revisiting Robust Neural Machine Translation: A Transformer Case Study* 2021. arXiv: 2012.15710 [cs.CL].
41. Shi, X. *et al. Convolutional LSTM Network: A Machine Learning Approach for Precipitation Nowcasting* 2015. arXiv: 1506.04214 [cs.CV].
42. Wiatowski, T., Grohs, P. & Bölcskei, H. *Energy Propagation in Deep Convolutional Neural Networks* 2018. arXiv: 1704.03636 [cs.IT].
43. Lino Ferreira da Silva Barros, M. H. *et al.* Benchmarking Machine Learning Models to Assist in the Prognosis of Tuberculosis. *Informatics* **8.** ISSN: 2227-9709. https://www.mdpi.com/2227-9709/8/2/27 (2021).
44. Lee Jr, D. B., Klein, L. A. & Camus, G. Induced traffic and induced demand. *Transportation Research Record* **1659,** 68–75 (1999).
45. Su, J. *et al. Convolutional Tensor-Train LSTM for Spatio-temporal Learning* 2020. arXiv: 2002.09131 [cs.LG].
46. Deng, L. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine* **29,** 141–142 (2012).
47. Opitz, D. & Maclin, R. Popular Ensemble Methods: An Empirical Study. *Journal of Artificial Intelligence Research* **11,** 169–198. ISSN: 1076-9757. http://dx.doi.org/10.1613/jair.614 (Aug. 1999).
48. Andersson, T. R., Hosking, J. S., Pérez-Ortiz, M., *et al.* Seasonal Arctic sea ice forecasting with probabilistic deep learning. *Nat Commun* **12,** 5124. https://doi.org/10.1038/s41467-021-25257-4 (2021).
49. Breiman, L. Bagging Predictors. *Machine Learning* **24,** 123–140 (1996).