# A Deep Learning Based Approach for Time-series Modeling

by

**Dhaifallah Alghamdi**

M.S. of Industrial Engineering, University of Illinois at Chicago, 2017

Submitted to the Graduate Faculty of

the Swanson School of Engineering in partial fulfillment

of the requirements for the degree of

**Doctor of Philosophy**

University of Pittsburgh

2022

UNIVERSITY OF PITTSBURGH

SWANSON SCHOOL OF ENGINEERING

This dissertation was presented

by

Dhaifallah Alghamdi

It was defended on

May 24, 2022

and approved by

Jayant Rajgopal Ph.D., Professor, Department of Industrial Engineering

Bo Zeng Ph.D., Associate Professor, Department of Industrial Engineering

Hoda Bidkhori Ph.D., Assistant Professor, Department of Industrial Engineering

Masoud Barati, Ph.D., Assistant Professor, Department of Electrical Engineering

Dissertation Director: Jayant Rajgopal Ph.D., Professor, Department of Industrial Engineering

**A Deep Learning Based Approach for Time-series Modeling**

Dhaifallah Alghamdi, PhD

University of Pittsburgh, 2022

Big data has evolved as a new research domain in the digital era in which we live today. This domain deals with the study of huge datasets with numerous different features, whose volumes are rapidly snowballing with time. These types of datasets can be produced by different autonomous sources, including scientific experiments, engineering applications, government records, financial transactions, etc. The availability of big data is of a great value because of the opportunity this provides for making better-informed decisions, but it also requires advanced analytical tools to derive important insights for these decision. This is the main reason that artificial intelligence (AI) and machine learning (ML) have gained immense popularity in recent years.

Time-series forecasting is an important application area for machine learning. It is important because there are so many prediction problems from various application domains that involve a time component. However, the temporal dimension also makes time-series problems more challenging to handle as opposed to many other prediction tasks. For this purpose, the goal of this dissertation is to design end-to-end frameworks and build advanced models for time-series forecasting that are based on deep learning. The discussed frameworks in this dissertation share three important characteristics: 1) the ability to generate forecasts for multiple steps ahead in the future, 2) the ability to provide estimates of uncertainty associated with these forecasts, and 3) the flexibility to incorporate exogenous factors. Our approach is to harness the encoder-decoder architecture to learn from historical data and capture important relationships embedded in the time-series, and to then use this knowledge to generate forecasts for multiple steps in the future along with estimates on the uncertainty associated with these forecasts.

In our study, we validate the proposed models on real data from two important application domains: intelligent transportation systems (ITS) (Chapter 3 & Chapter 4), and finance (Chapter 5).

# Table of Contents

## List of Tables

**List of Figures**

## Preface

This work would not have been completed without the constant support from my advisor Dr. Jayant Rajgopal. He has profound knowledge and experience in conducting fundamental theoretical and practical research in the realm of Operations Research and Advanced Analytics. He also has a unique way of advising, where he really cares about the learning and development of his students and the quality of their research as well as how it is aligned with their future career goals. Dr. Rajgopal has encouraged me to work on complex problems that I am most interested in and kept challenging the initial research ideas to refine them and to contribute to the research field by addressing interesting research questions. He has also guided me through each step of the process to get my work published so other researchers and practitioners have access to it. Dr. Rajgopal has also showed me empathy and support when I started planning for the next chapter in my life so I can start off my future career properly. During my meetings with Dr. Rajgopal, I have always witnessed that he is not only a professional academic advisor with demonstrated history in the academic world, but also a great person who has deep understanding and unique perspective of life. He has inspired me unconsciously to be a better person and to find the right balance for me in life. Thank you Dr. Rajgopal!!

I also would like to acknowledge my dissertation committee, for their patience, support and ideas that have been crucial in the completion of this study. Dr. Bo Zeng, Dr. Hoda Bidkhori and Dr. Masoud Barati have shown me that I made a wise choice requesting their presence on my dissertation committee, by bringing each of their own expertise to my projects. Special thanks go to my colleagues and friends for all the time we spent together.

To my lifelong partner, my everlasting love, to the future Dr.Assma Alghamdi, thank you for being there for me all the time. I wouldn't have reached the finish line without your understanding and wonderful support. I owe you a great deal and I hope I can be as supportive during your times of need especially in your Ph.D.

To my lovely sweet heart daughter Malak, thank you for the joy you have brought with you to our lives. Thank you for your patience during the times where your mom and I had to work for extended hours. You have contributed to this achievement, and you will always be our hero.

Most importantly, I would like to express tremendous gratitude to my parents, my brothers, my sisters, my nephews & nieces, and the rest of my family. I am the person who I am today because you have shaped me in every way. Mom & dad, you are my idols and I am lost for words to express how grateful I am for the love and support you have given to me.

## 1.0 Motivation and Research Objectives

Over the years, technology has revolutionized our world and radically changed our daily lives. Technology has created powerful tools and resources, putting lots of useful information at our fingertips. These tools and resources are driven by complex systems interacting with dynamic environments. Modeling these dynamics is vital when we seek to create a smart system tailored towards a particular objective function that eventually serves humanity and improves the quality of life. Although new tools and applications are being introduced every day, many of the current systems are still far from optimal and their design and operation can be significantly improved. Artificial intelligence (AI) has shown great potential in improving these systems, by helping machines to think and act like humans. In particular, machine learning (ML) is a subset of AI that focuses on algorithms that help machines to automatically learn and improve from experience without explicitly programming them; ML has become the core engine for most of the smart systems behind many of the applications we use today. In the last decade, a sub-field of machine learning called *deep learning* (DL), has been leading the state of the art in modeling complex systems in areas such as speech recognition and natural language processing (NLP). In a nutshell, deep learning uses the composition of many nonlinear functions to model the complex dependency between input features and target labels. It is widely acknowledged that two vitally important factors have contributed to the success of deep learning:

- Huge datasets that often contain millions of samples
- Immense computing power resulting from clusters of graphics processing units ((GPU))

Our work is motivated by: 1) the necessity to process substantial amounts of time-series data in many smart systems, and 2) the promising performance of deep learning on a broad class of problems. With reference to the latter point, *sequence-to-sequence* (Seq2Seq) learning is the approach most closely related to our work. In simple words, Seq2Seq is about training models to map sequences from one domain to sequences in another domain. For example, in a translation machine, we might convert sentences (sequences of words) in English to sentences (equivalent sequence of words) in French. The contribution of our work is in combining advanced deep learning tools in order to design end-to-end predictive modeling frameworks with novel complex architectures for analyzing complex dynamic systems. The specific focus of this work is to explore how we can leverage the power of deep learning to address complex real-life systems that deal with large volumes of time-series data.

Broadly speaking, a time-series comprises data that collectively represents the progression of a system over time. A major distinguishing characteristic for this type of data is the temporal dimension (i.e., the *timestamp*). Time-series data has long been valuable, but is of growing significance with the rapid growth in the Internet of Things (IoT). The availability of time-series data of continuously increasing size is being driven by the emergence of modern sensor, communication, and computing technologies. However, time-series processing and forecasting is more challenging compared to simpler tasks such as regression and classification; the temporal dependencies between data points add to the complexity of the problem. At the same time, the temporal structure of the data adds important context and enhances the modeling with the underlying trend and seasonality. Traditionally, linear regression and its variants (e.g. ARIMA), or heuristics such as exponential smoothing have been the dominant methods to model time-series data due to their effectiveness, simplicity and interpretability [11, 56]. Nevertheless, these models exhibit major limitations:

- The assumption of a linear relationship might not be valid
- The temporal dependencies between observations must be diagnosed and the number of lags need to be specified in advance
- There is a lack of flexibility in incorporating exogenous factors

Classical machine learning models have also been developed to model time-series data [80, 19, 90, 50, 33, 16]. For example, a Support Vector Machine (SVM) based model has shown good experimental results. SVM is suitable for handling regression tasks; it constructs a linear decision function by mapping samples from the original space to a higher dimensional space [76]. An ensemble boosting based model has also been proposed [68]. However, this class of models cannot learn features independently and requires extensive feature engineering.

The power and capabilities of neural networks suggests that deep learning could be a good fit for modeling time-series. Those capabilities can be summarized as following:

- Neural networks use a feature learning mechanism to understand the complex underlying interactions in the data.
- Neural networks learn arbitrary mapping functions.
- Neural networks can handle non-stationary time-series.
- With a properly designed architecture, neural networks support multivariate inputs as well as multi-step outputs.
- Different architectures provide different capabilities; Convolutional Neural Networks (CNN) support feature learning while Recurrent Neural Networks (RNN) enable learning of temporal dependencies.
- Hybrid models combining different architectures improve the learning process.

In Chapters 3 & 4, our focus is on *intelligent transportation system* (ITS). Specifically, we focus on building a travel demand prediction model for both taxi and car sharing companies that can be used to allocate resources more efficiently. The world's population is growing significantly, and almost 50% of humanity today lives in cities. In many instances, current infrastructure is not readily scaled to the exponential growth of cities as people move in from rural areas seeking better education and jobs [22]. Thus there is an increasing emphasis on so-called smart cities, where digital enhancements enable a more comfortable life. An intelligent transportation system (ITS) is one of the essential components of a smart city. However, existing decision support systems for efficiently managing transportation within a successful shared economy (including ride-sharing services) encounter various challenges related to demand prediction and matching supply with demand [63]. Demand-supply imbalances can cause severe problems for the entire system, including traffic congestion, price surges, poor resource utilization, and an unpleasant overall user experience. In order to be able to pick up customers as soon as possible after they request service, drivers need to be allocated appropriately. This is a challenge because one does not have full prior knowledge on where demand might occur. Effective vehicle distribution and dispatching strategies will help both drivers and passengers minimize wait-times, and accurate demand prediction is vital in order to organize the fleet and plan operations effectively by distributing the available fleet based on the demand across the entire city [95, 73].

We start by building a region-based prediction model (Chapter 3) that focuses on forecasting demand at specific micro-geographic locations. Next, we extend this further to predict demand for specific origin-destination pairs (Chapter 4). The two different tasks can be used for different use cases that eventually contribute to the development of smarter dispatch systems. For example, solo ride products such as traditional taxi services and UberX benefit the most from knowing in advance the pick up locations that are likely to have high demand in order to be there at the right time when riders need the service. At the company level, car-sharing providers use the region-based demand forecasting models to allocate resources properly and feed their pricing models with this information for optimization purposes. Car sharing providers also use the region-based models to ensure the availability of supply when demand is expected in order to maintain reliability of service. Also, drivers' promotion and incentives are designed to encourage more supply for specific locations during specific times.

The challenges associated with building a region based forecasting model revolve around the spatial & temporal interactions within the dynamic environment. For instance, demand around a stadium will be generally stable and at low to moderate level if not surrounded by other attractions, but when there is a game or other event at the stadium, the demand in this region can spike. This in turn requires prior supply allocation. Understanding this trend and the causality between a sporting event and high demand around the stadium is very important to capture in the region-based model. The complexity of the task increases with finer resolutions in terms of time and space. Destination incorporation further adds to the complexity of the modeling since graphs representing the flow between different zones will be introduced; this will result in more computational challenges with the use of large sparse adjacency matrices representing the demand flow across the entire region being studied.

The second application domain is finance, where we design a deep-learning model for stock price prediction (Chapter 5). Our model accounts for the different factors of sequence modeling in the finance world such as market volatility and trends. In the following chapters, detailed problem definitions will be presented for each topic along with statements on specific contributions made by this work.

# 2.0 Deep Learning Methods

In this chapter, we introduce fundamental deep learning concepts and architectures that we will use to build our models in subsequent chapters.

## 2.1 Feed Forward Neural Networks

Feed forward neural networks lie at the heart of deep learning models, and were initially proposed to imitate how the human brain works. Their general structure is shown in Figure 1. The first neural network ever proposed was the perceptron [70]. There are two observations that motivated the development of the perceptron:

1. The human brain demonstrates intelligent behavior, and therefore intelligent systems can be built by reverse engineering.
2. Conversely, building mathematical models for intelligent systems can help answer scientific questions on how the human brain works.

The perceptron is a version of a feed forward neural network where loops are disallowed while connecting the nodes in the network. It receives one or multiple inputs which are multiplied by weights and then aggregated. The concept of weights is derived from the role of the synapse (the gap between biological neurons in the brain) that can give a different emphasis to different signals transmitted between neurons. This aggregated value is then passed to an activation function which simulates the status of the neuron (firing or not firing). The mathematical representation of a perceptron function is as follows:

$$f(x) = \begin{cases} 1 & \text{if} \quad x.w + b \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.1}$$

A *multi-layer perceptron* (MLP) is simply a set of stacked layers where each layer contains a set of perceptrons. The learning algorithm for the perceptron is straightforward; it is based on reducing the weights via a pre-determined learning rate if the difference between the ground truth and the predicted output is negative and increasing the weights if the converse is true. The MLP can also be thought of as a deep feed forward neural network. Unfortunately, there was a major issue with the multi-layer perceptron in the 1980's that caused a huge drop in interest in exploring neural networks. The issue was the lack of a suitable learning algorithm for deep, stacked layers. This lasted until the development of the *backpropagation* algorithm to resolve this issue and unblock the untapped potential of neural networks [71]. The algorithm is based on gradient descent, which is an iterative process to find a local minimum for a loss function by moving in the direction of the negative gradient. Minimization of the loss function is the main objective of the learning process in neural networks. The gradient descent iteratively computes the gradient of the loss function relative to each set of weights and biases and then updates these accordingly with the aim of improving the loss function. The bias is basically a vector of constant values corresponding to each layer, and the value for a particular layer is added to the inner product of the input and weight vectors corresponding to that layer. It has the role of moving the resultant value either negatively or positively in order to improve the learning.

In a supervised learning setting, we are given a set of input-output pairs. The main objective is to find a proper mapping from the input to the output. This can be achieved via the following general steps:

1. Split the data into training and validation sets
2. Initialize the weights of the network randomly
3. Feed forward the training input data through the network and calculate the error between the ground truth values and the predicted ones (i.e. loss function)
4. Perform backpropagation by calculating the gradients backward to updates the weights
5. Iterate till the weights are optimized
6. Use cross validation to measure the generalization capability of the network

In essence, neural networks enable learning the structure of the data or information and have been used widely to perform tasks such as clustering, classification, and regression.

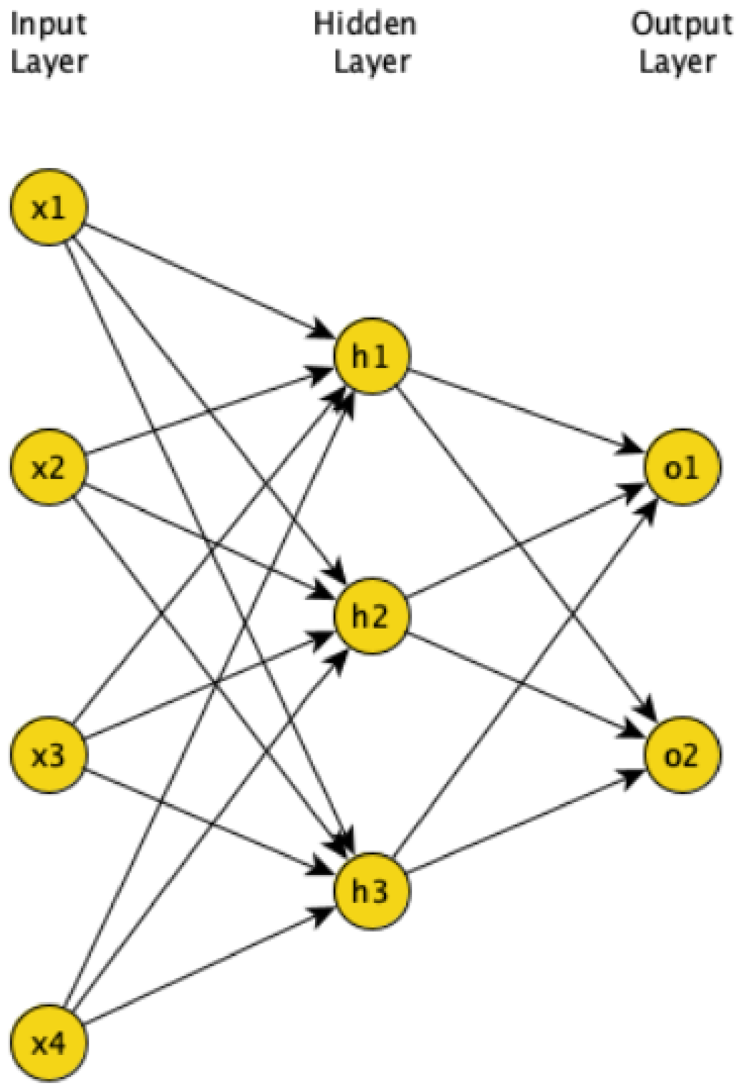


Figure 1: General structure for a classical Feed Forward Network: input layer, hidden layer/layers, and output layer [4]

## 2.2 Convolutional Neural Networks

The Convolutional Neural Network (a.k.a ConvNet or CNN) is an artificial neural network (ANN) that has been most commonly used for analyzing images. Although image analysis has been the most popular use case, CNNs can also be exploited for other analysis or classification problems. CNNs are powerful artificial neural networks capable of feature extraction and pattern detection. We could think of them as specialized structures for picking up patterns and interpreting them. The main differentiator between a CNN and a classical neural network in the form of multi-layer perceptron or MLP (i.e. stack of layers) is the presence of special layers called *convolutional* layers.

A typical convolution network is composed of an input layer, a series of convolutional and non-convolutional layers, and finally an output layer. The input layer could be two-dimensional, like with an image, and we do not need to flatten it or convert it to a 1-D vector as we do for a classical artificial neural network. The basic flow of a CNN comprises receiving an input and passing it through a sequence of layers, including the convolutional ones. For each convolutional layer we need to specify the number of filters (a.k.a kernels) which are responsible for detecting patterns. A pattern could be edges, shapes, or textures for an image; or trends or seasonality for time-series. Different filters detect different patterns. For example, some filters can detect edges and are called corner detectors, while other filters detect squares, circles or any regular geometries. The deeper one goes within the network, the more sophisticated these filters become. Hence, for layers in the middle of deep networks, the filter might detect objects like leaves, eyes, ears, etc. The filters attached with layers even deeper in the network and closer to the output layer, can detect even more sophisticated objects like flowers, cats, dogs, etc.

The convolution operators have five hyper-parameters to be specified:

1. Filter size
2. Stride
3. Padding
4. Dilation
5. Activation function

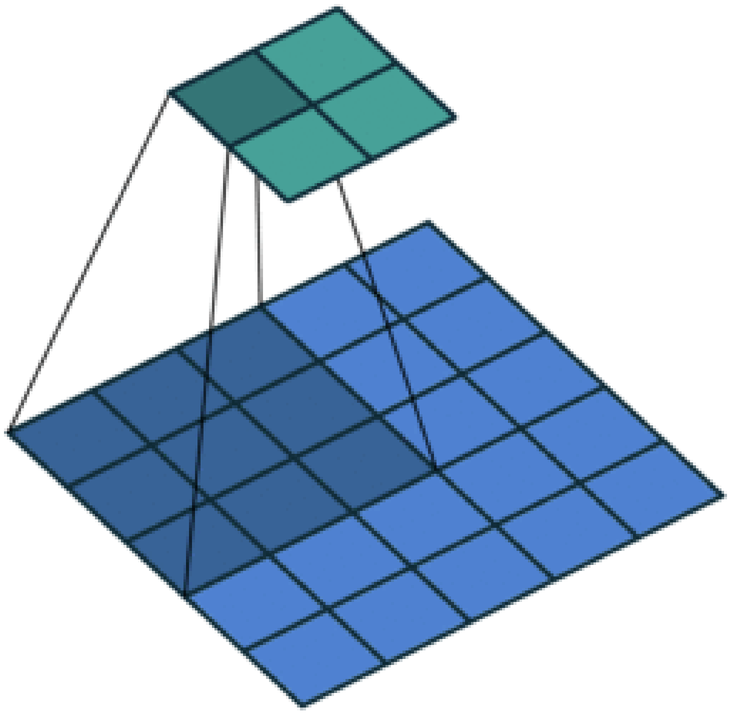


Figure 2: Convolutional neural network [20]

To illustrate this further, assume that we have a 5 pixel by 5 pixel image (Figure 2). Each pixel has a value representing the darkness of its color (0 for black, and 255 for white). The convolution process can be described as passing an $n \times n$ filter over the image starting from the top left corner, multiplying the filter values by the pixels values and aggregating the results. From a computational point of view, smaller filters are preferred over larger filters since the learning algorithm needs to learn the values within each filter, and as the size increases so does the training complexity. Then the filter moves one or more steps to the right depending on the specified value for a hyper-parameter called the stride. Once the filter hits the far right pixel of the image, it moves down again based on the stride value. The process is carried on till the filter scans all pixels in the image.

Clearly, by applying smaller filters, the resultant image will be smaller in size. If we want to keep the same size of the image to apply filters of subsequent layers on, then we can use padding. Padding is basically introducing hypothetical pixels in each direction of the image (left, right, top, down) as shown in Figure 3. The values of the imaginary new pixels could be zeros or any other specified values. Figure 3 shows a $3 \times 3$ filter applied to $5 \times 5$ image with zero-padding of size 1 in each direction. The dilation parameter is used for a larger receptive field (i.e. portion of the image that impact the calculation of the values of the filter). As shown in Figure 4, the dilation frees the filter from being applied only to contiguous pixels and allows it to cover wider receptive field. This way the learning process can proceed and converge faster.

After the filter passes through the entire image, an activation function is applied to introduce non-linearity. The most common function used in literature is ReLU which maps every negative number to 0 (or a small positive number in the case of a *Leaky* ReLu) and retains all positive values (Figure 5).

The main advantage of using convolutions is with parameter sharing. In simple terms, the filter being used on a portion of the image is used to scan the remaining pixels of the image without adding more filters. The intuition here is that a filter applied to a portion of the image to learn local features will most likely be able to also capture other local features in other parts of the image. Hence, a single filter is convolved over the whole input image and the parameters (i.e. values of the filter) are shared. Moreover, sparsity of connections

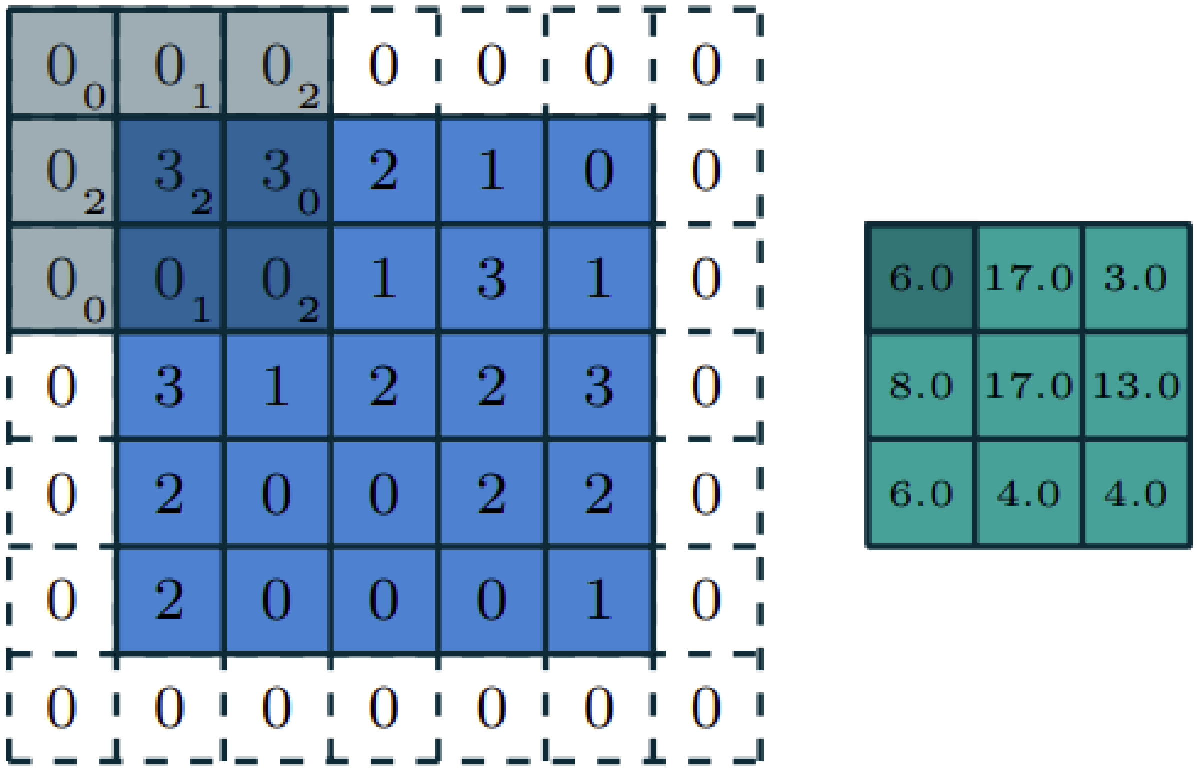


Figure 3: Illustration of padding in CNNs [20]

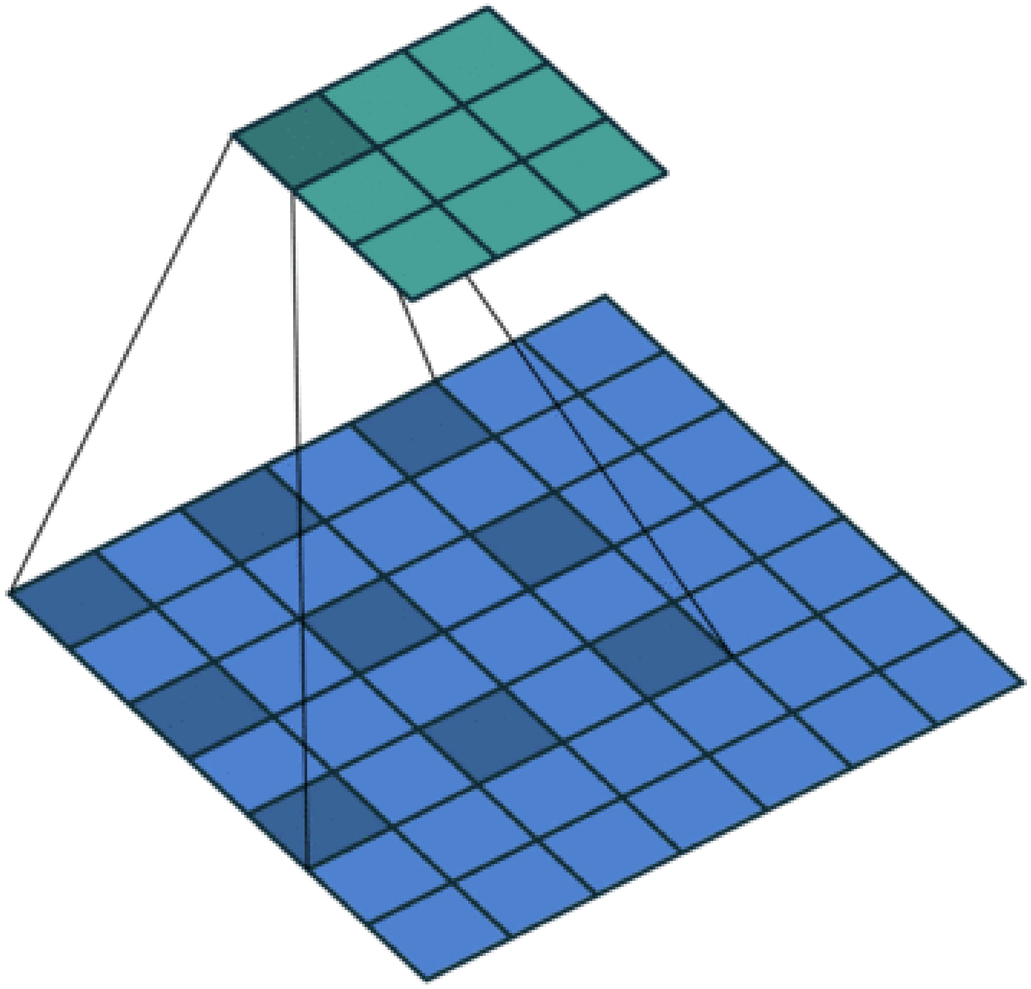
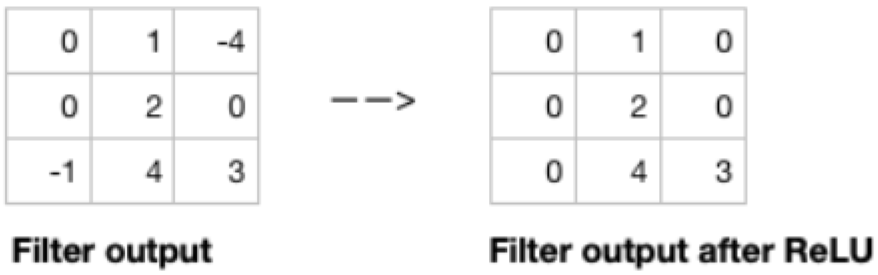


Figure 4: Illustration of Dilation [20]



|   |   |   |
|---|---|---|
| 0 | 1 | -4 |
| 0 | 2 | 0 |
| -1 | 4 | 3 |

**Filter output**

--->

|   |   |   |
|---|---|---|
| 0 | 1 | 0 |
| 0 | 2 | 0 |
| 0 | 4 | 3 |

**Filter output after ReLU**

Figure 5: Illustration of ReLu activation function

present another advantage since each filter value is dependent on a relatively small number of inputs instead of the entire set of pixel values of the image. Convolutional neural networks can also be applied to 1-D vectors, such as time-series and voice data, and 1-D CNNs are efficient and do not require any feature extraction or dimension reduction.

### 2.3 Recurrent Neural Networks

In a feed forward network (e.g., CNN) information flows only in a forward direction, starting from the input nodes, passing thorough stacked hidden layers and then to the output layer. They have no cycles or loops in the architecture. While CNNs and other types of feed forward neural networks have shown great success in regression and classification problems including image processing and object detection, there are some issues that limit where they can be used:

1. They cannot handle sequential data
2. They consider only the current input
3. They lack the ability to memorize previous input

A Recurrent Neural Network (RNN) is a special type of neural network that was designed to overcome these limitations and enable working with sequential data. Sequential data include any data form where order (or context) matters, such as audio and text. RNN is a generalization of feed forward networks with an internal state (i.e. memory) that can represent context information. The internal memory enables RNNs to remember important aspects about the received input sequence, which allows them to better predict the next output. This is the main reason why RNNs are preferred over other architectures for sequential data including time-series, financial data, speech recognition, text, weather, etc. The ability to capture dependencies within the input sequence is what makes RNNs popular for many complex tasks such as machine translation, time-series forecasting, financial analysis, weather forecasting and video processing. In other neural networks, it is assumed that the inputs are independent. With RNNs, in order to produce the next-step output, information cycles through feedback loops that enable us to share information while considering the memorized context along with the current input. In other words, RNNs add the immediate past to the current to predict the next output. Figure 6 shows the difference between feed forward neural networks and recurrent neural networks. Similar to any neural network, RNNs assign weights randomly and then try to optimize them using the backpropagation algorithm. However, RNNs account for both current and past information when optimizing weights through the gradient descent algorithm and back-propagation through time (BPTT).

While feed forward neural networks can only map one input to one output (e.g. image to class) there are other different mappings that RNNs can perform (Figure 7):

1. One to many: music generation
2. Many to one: voice classification
3. Many to many: machine translation

RNNs are the proper choice for modeling sequential data where there is a time dependency between input data; however, they cannot handle dependencies in data that are significantly far apart in time. This is due to the vanishing (exploding) gradient problem, where the gradient of the loss function decays (grows) exponentially with time. In response

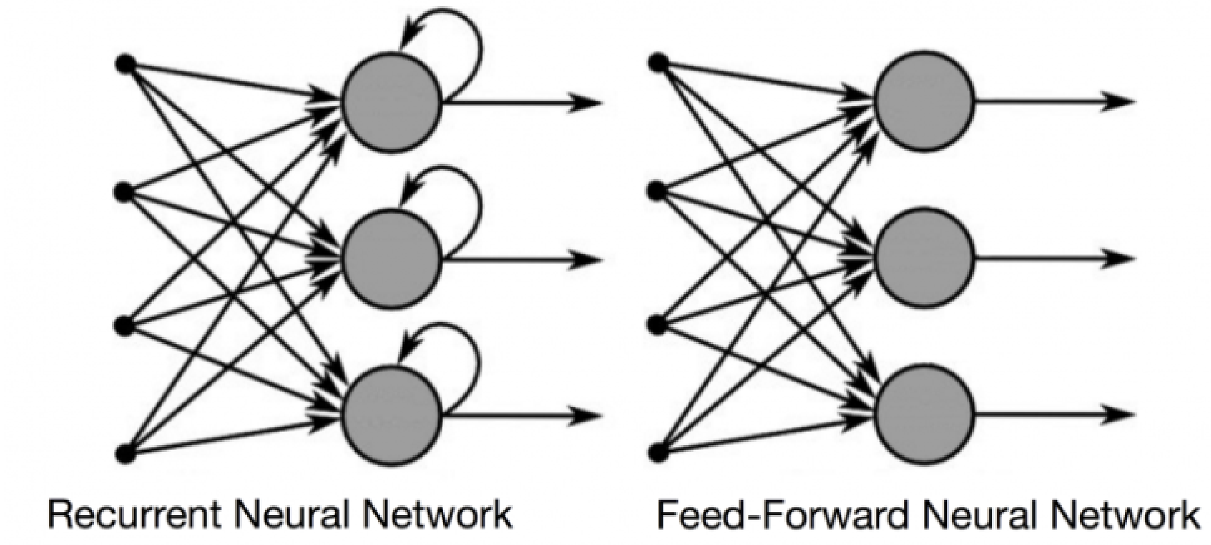


Figure 6: Recurrent Neural Networks vs. Feed Forward Networks [5]

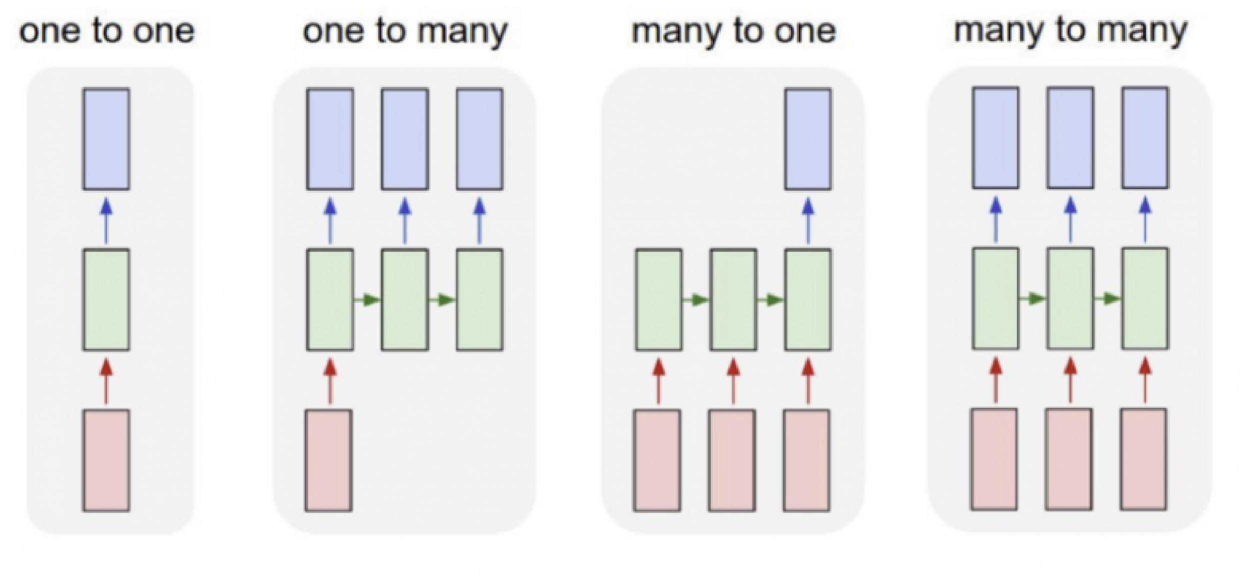
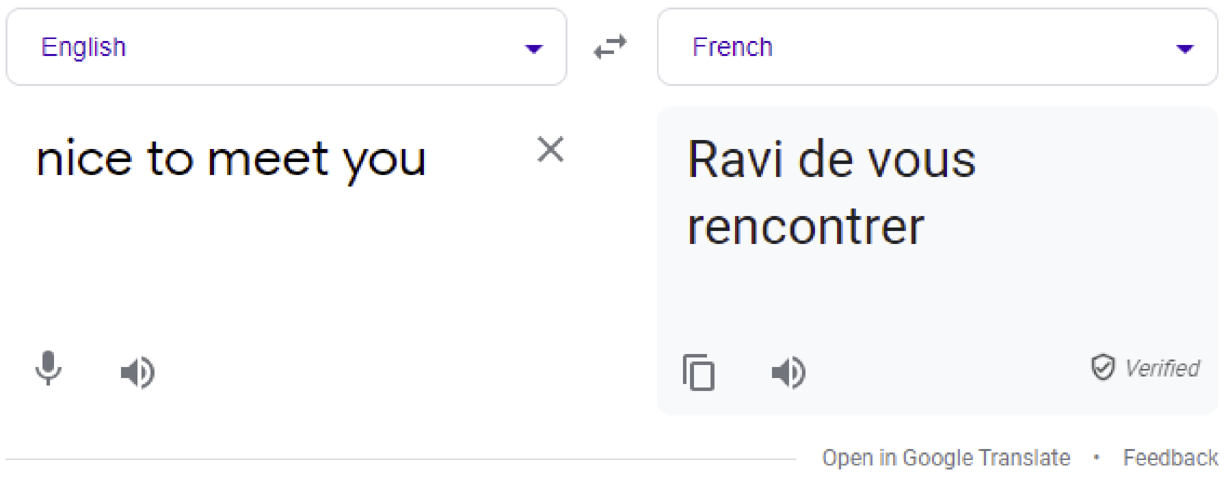


Figure 7: Recurrent Neural Networks types [5]



nice to meet you

Ravi de vous rencontrer

Figure 8: seq2seq example: machine translation [5]

to this shortcoming, Long Short Term Memory (LSTM), which is a special kind of RNN was introduced [35]. LSTM is capable of handling long-term dependency between input data while avoiding the the vanishing (exploding) gradient problem. It has a gating mechanism that controls the flow of information and decides what past information to keep or forget. This is discussed further in the context of each of our models where we introduce them.

### 2.4 Sequence Modeling and seq2seq

Sequence modeling deals with data where either the input or the output is a sequence of data (e.g. text, time-series). A special class of sequence modeling is *seq2seq* where both the input and the output are sequences. Consider an example where the input is a sentence in English: “nice to meet you” and the output is its translation in French: “ravi de vous rencontrer” (Figure 8). Here, we have an input sequence, a sentence in English, and an output sequence, a sentence in French. Google translation is based on seq2seq modeling.

For seq2seq models, we leverage an architecture called the *encoder-decoder* architecture (Figure 9). At a high level, the encoder-decoder architecture is composed of three parts:

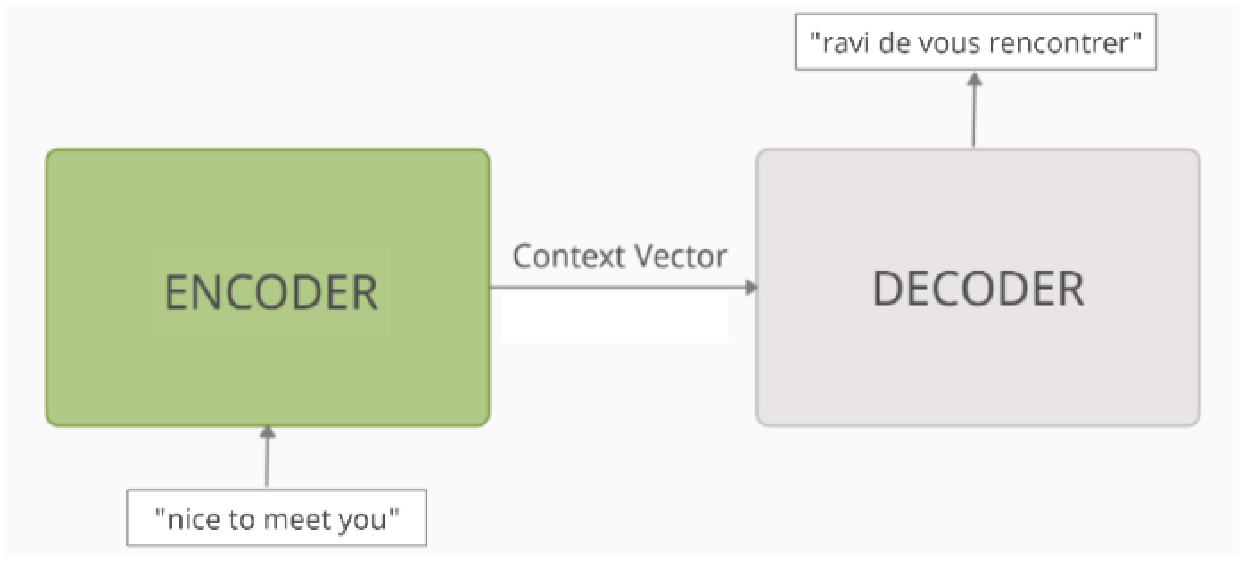


Figure 9: Encoder-decoder general structure

1. Encoder: It reads each token (i.e. word) in the input sequence to try and extract all its associated information and then compresses it into a vector (i.e. the context vector). The context vector is of a fixed size length. After processing all tokens in the input sequence, the encoder produces the context vector and passes it to the decoder.
2. Context vector: It is constructed with the expectation that it will contain all the important information associated with the input in order to help the decoder generate accurate predictions. We can think of it as the final internal hidden state of the encoder block.
3. Decoder: The decoder receives the resultant vector and uses it to predict the output sequence, token by token.

Although the encoder-decoder structure is very efficient and widely used for seq2seq modeling, its performance begins to degrade as the size of the sequence grows. This is mainly because it becomes difficult to encapsulate all the compressed information of a long sequence within a fixed size context vector. The overall prediction accuracy can thus be negatively impacted by long input sequences. This is where “attention” comes into play.

Attention is simply a way to help the decoder know what portions of the input sequence to focus on when predicting each token in the output sequence [60]. The attention mechanism applies different weights to each part of the input sequence for different tokens in the output sequence. For instance, in the context of the machine translation example given above, more weight will be assigned to the word “nice” in the input sequence when the decoder is trying to predict “ravi” and more weight to “meet” when predicting “rencontrere”. Attention can be viewed as a shortcut to align the most relevant words in the input sequence for each word in the output sequence.

The internal architecture of the encoder or decoder typically employed RNNs & LSTMs by default. However, more recently a new architecture called Temporal Convolution Networks (TCNs) was proposed as an alternative. TCN is a descriptive term that refers to a family of architectures and is a variation of the convolutional neural network (CNN). The notion of using CNNs for sequence modeling was proposed by Google DeepMind [81]. There are two major characteristics of TCNs:

1. They can handle a sequence of any length
2. The convolutions are causal; they cannot move bidirectionally and so they do not allow information leakage from the future to the present.

TCN may be considered as a set of stacked, dilated causal convolution layers. Dilated causal convolutions are preferred to simple causal convolutions because they allow the receptive field to grow exponentially with every additional layer. As discussed in the subsection on CNNs, a larger dilation enables a wider range of inputs to be represented by an output at the top level. Figure 10 shows a dilated causal convolution with dilation factors: 1, 2, 4 and filter size 3.

In summary, the purpose of this chapter is to introduce the most basic elements of important deep learning concepts for a reader who is not familiar with common architectures such as FNN, CNN, RNN, LSTM, and seq2seq. These architectures will be used as the building blocks in our novel models in subsequent chapters.

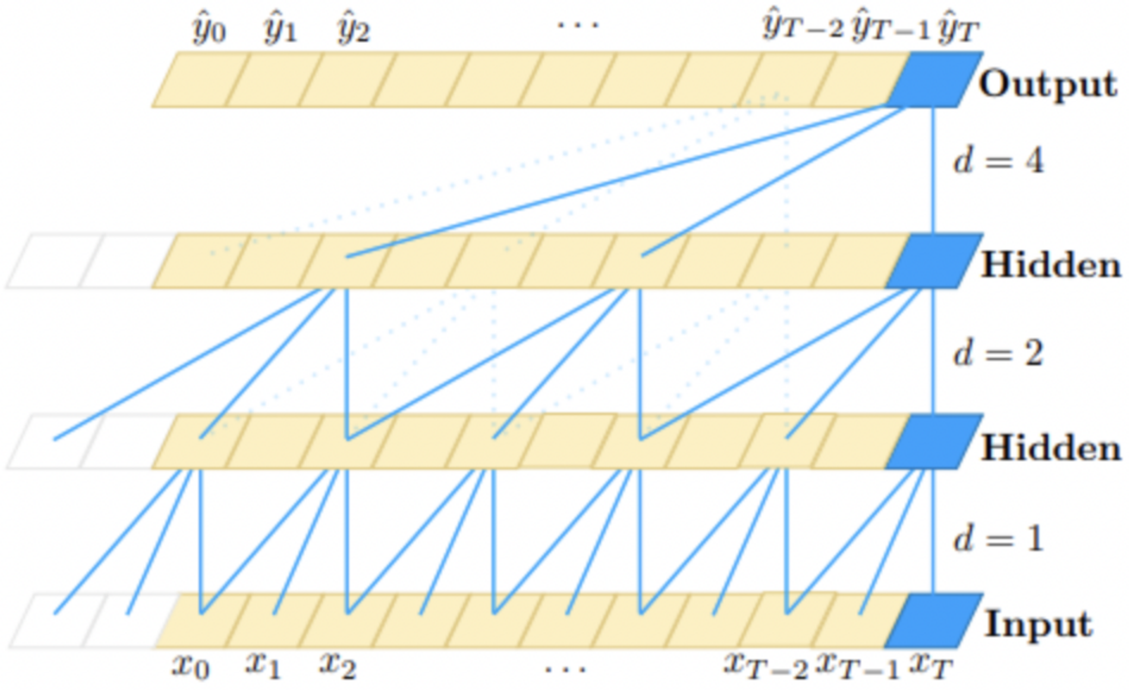

$\hat{y}_0$ $\hat{y}_1$ $\hat{y}_2$ ... $\hat{y}_{T-2}$ $\hat{y}_{T-1}$ $\hat{y}_T$
Output
$d = 4$
Hidden
$d = 2$
Hidden
$d = 1$
Input
$x_0$ $x_1$ $x_2$ ... $x_{T-2}$ $x_{T-1}$ $x_T$

Figure 10: TCN structure illustration [9]

# 3.0 Region Based Probabilistic Prediction for Travel Demand

## 3.1 Introduction

Ride-sharing has disrupted the way people commute, by leveraging the concept of a shared economy to provide more price-competitive products than traditional taxi services. In a shared economy resources are turned into services so that individuals and groups share them in a collaborative way. For ride-sharing, a driver shares his or her own car and time to provide a service for an individual or a group in exchange for money. Recent advancements in technology have accelerated the adoption of ride-sharing services where a request with a specific location can be placed through an app and then an offer with an estimated time of arrival and an up-front price is presented. This is different than the traditional taxi services where price is not determined beforehand. A major contributor to the success of ride-sharing services is the concept of surge pricing (a.k.a. dynamic pricing), where supply and demand are encapsulated within the determination of price. Therefore, demand prediction becomes even more important with the ride-sharing form of transportation in order to manage supply and ensure reliable service, which is critical for strategic growth. Prior knowledge of demand enables the service providers to take the right course of action such as notifying drivers where demand is or incentivizing non-active drivers to log in by offering a platform fees discount.

Arriving at accurate predictions at a micro-geographical level (as opposed to the level of a city, precinct or large neighborhood) is a challenging problem. Furthermore, obtaining these predictions for multiple future time steps along with reliable estimates of prediction uncertainty adds another layer of complexity. The problem becomes even more challenging when demand is volatile, and during holidays and special events (e.g., sports fixtures or concerts) when many external factors contribute to the sudden demand changes. In this setting, better multi-step-ahead forecasts of demand across relatively small geographical areas, along with accurate uncertainty estimation, can help us build robust systems that are responsive to demand fluctuations. It is worth noting that in recent years there has been a growing interest within the research community in addressing this topic and in developing reliable solutions for smart transportation systems [61, 83, 92, 84, 44, 93].

This chapter is motivated by the fact that prior work has largely focused on macro-geographic, deterministic, next-time step prediction, and without explicit uncertainty estimation [15, 30, 96, 37, 38, 44, 103, 99]. We believe that for a robust and responsive system, it is important to have real-time, multi-step ahead forecasts at a micro-geographical level across an entire city or metropolitan area, along with estimates of the uncertainty associated with these forecasts.

There are several practical reasons for each of our desired features. First, multi-step-ahead forecasting enables us to make operational and tactical decisions over longer time horizons, rather than making short-sighted decisions that might negatively affect resource distribution during later time periods. Second, a micro-geographical breakdown enables us to capture characteristics of specific locations over time. The nature of travel demand prediction requires us to account for the interaction between the spatial and the temporal aspects. Demand distribution at a particular location can be totally different from that at another location during the same time period, and similarly, the distribution at a given location could be very different at different points in time. Third, effective decision-making requires that we quantify our confidence about the results produced by the model through an accurate estimation of its uncertainty. In addition to these three objectives, it is also important to incorporate exogenous factors (e.g. weather, time-series trends and seasonality, and temporal clustering) into the forecasts, especially during time periods with high variability. Temporal clustering basically groups specific times (day of the week and hour of the day) based on their similarities in terms of average demand and variance. For example, there are specific hours of the week with high average demand and low variance such as afternoon peak-time and Friday early evening. These hours can belong to the same cluster (high average demand and low variance).

In this chapter, we address the issues listed above and develop a method that exploits the deep-learning techniques described in the previous chapter to obtain accurate and reliable multi-step-ahead forecasts of demand at a micro-geographical level. There are specific technical challenges that must be addressed, such as:

- Accounting for complex spatial/temporal and pick-up/drop-off interactions
- Incorporating important exogenous factors into sequence modeling
- Obtaining acceptable computational performance
- Estimating uncertainty associated with forecasts in a stochastic setting
- Obtaining robust multi-step ahead forecasts

To briefly elaborate on each of these challenges, first, demand and drop-off at a certain node (location) might affect the demand at another node in the network (city). Similar to how an earthquake often triggers the occurrence of more earthquakes in the same area within a specific period of time, pick-up and drop-off at a particular node might affect the demand at other nearby nodes [57]. Demand can also be time dependent not just with respect to the immediate prior point in time, but possibly over longer periods of time. Second, traditional ML approaches to sequence modeling often lack the flexibility to include external factors along with the time series data. Third, the methods suffer from slow convergence in the process of capturing long time dependence. Fourth, uncertainty estimates are hard to obtain with a classical neural network. Finally, for sequential multi-step forecasting, errors in the early steps can propagate and affect the forecasts at subsequent steps.

Motivated by recent advances in deep learning for sequential modeling [13, 78, 87, 66], we propose an end-to-end encoder-decoder framework with a novel architecture in order to address the foregoing challenges. We provide a high level overview here and the details of our approach are provided in Section 2.3. Our framework encodes demand history, which is then decoded to predict future demand. We use a multi-stage model; in the first stage we pass only the sequential data, while in the second we do transfer learning, and incorporate the exogenous factors. To overview our model, in the first stage, the input to our encoder is a sequence of spatiotemporal demand that is passed through a convolutional layer to extract the spatiotemporal features, along with pick-up and drop-off interactions. Subsequently, we exploit the power of convolution operations and pass the data stream through a temporal convolutional network to learn additional features that encapsulate the hidden characteristics of the sequence. For each time step in the forecasting horizon, the decoder receives the demand information from the previous time period and passes it through long short term memory cells (LSTM). We then deploy an attention mechanism on the resulting hidden representation of the LSTM layer in the decoder along with the output of the encoder. This allows the model to address the importance of each segment of the input at that specific time step in the prediction horizon. The attention is then concatenated with the output of the final LSTM layer and fed to a multi-layer perception (MLP). In addition, we use a sampling trick (teacher-forcing) to improve the robustness of our model to any deviation that takes place in the early steps in the forecast horizon. In the second stage, we use the encoder from the pre-trained model to concatenate its output with exogenous factors and pass them to a MLP. Finally, a Monte Carlo dropout (MC dropout) technique is used to measure uncertainty [25].

The contributions of the work in this chapter are threefold:

- From a *methodological* perspective, we introduce Multi-stage Probabilistic Temporal Convolution Network (MSP-TCN) as a novel deep learning architecture. The first stage combines the power of learning a latent representation with less expensive computation for time-series data. The second stage deploys transfer learning and incorporates exogenous factors (e.g., time-series trends & seasonality, and temporal clustering) to enhance the learning process. The transfer learning is essential to separate the stage where learning occurs from the time-series and the stage where exogenous factors are incorporated.
- From an *application* perspective, we address an important problem of predicting travel demand that has received a lot of attention during the last few years. Our framework enables multi-step ahead predictions with the ability to generate demand distributions for each step in the prediction horizon. The proposed framework neither require a Bayesian graphical network nor results in major changes to the architecture design.
- From a *computational* perspective, we conduct a comprehensive study using two different real-world datasets. We demonstrate that (with both datasets) our deep learning approach results in very good predictions, with performance that is superior along sev-

eral different evaluation metrics to other common DL approaches for multi-step ahead prediction. Moreover, the number of parameters to be learned by our approach is an order of magnitude smaller than that of the only comparable DL approach in our tests..

The remainder of this chapter is organized as follows. Section 3.2 introduces related work on prediction applications with taxi data, and sequence learning applications of LSTMs, Section 3.3 provides preliminaries including background and definitions, and Section 3.4 describes in detail, the proposed deep multi-stage sequence learning model. In Section 3.5, we discuss our experimental study and performance metrics, and present the results. Finally, in Section 3.6, we provide a brief summary along with conclusions.

## 3.2 Related Work

### 3.2.1 Travel Demand Prediction

We begin with a review of the main streams of work related to travel demand prediction and for each of these we summarize some of the shortcomings that our approach aims to address.

Traditionally, prediction of travel demand based on historical data has been done mainly via classical time series models. More specifically, these have largely been variants of auto-regressive integrated moving average (ARIMA) models [11, 56]. Ensemble methods have also been investigated along with streaming data to predict the spatial distribution of taxi passengers [58], where a combination of a Poisson model and ARIMA was used to generate predictions. Space–time auto-regressive integrated moving average (STARIMA) models that consider the spatial correlation among locations have also been studied [17]. However, these models have limited fidelity and limited tolerance for incorporating exogenous factors, which requires multiple training rounds, feature extraction, and extensive manual parameter tuning. Moreover, they are scale-sensitive since their performance degrades when working with multi-dimensional time series.

In a second stream of work, recent studies have investigated traditional machine learning models that also incorporate exogenous factors such as weather, demographics and special events data [80, 19, 90, 50, 33, 16]. For example, Deng et al. [19] used a latent space model for road networks to capture their sophisticated topological dependencies and the dynamic environment with changing road conditions. They introduced an online algorithm that exploits real-time sensor data to make real-time predictions. Other advanced machine learning algorithms such as support vector machines (SVM) have been explored to predict short term passenger flow [76], where a hybrid model combining Wavelet Transform and SVM was proposed. The approach first decomposes the sequence into low and high frequencies using Wavelet Transforms, then SVM is used to perform prediction. Another recent study [68] used boosting Gaussian conditional random field (boosting-GCRF) to build a short-term demand prediction model. Conditional random fields (CRF) constitute a well-known class of discriminative models best suited to prediction tasks where contextual information or dependence structure among outputs affect the prediction [65]. The proposed model is an ensemble model that uses GCRF as a base learner to model the interaction between every pair of elements in a historical demand sequence and output data. A boosting approach similar to the one used in the well-known Adaboost algorithm is implemented. This specific approach is sensitive to noise in the data, increases complexity and is hard to implement in real time. Also, we believe shadow base learners are not strong enough to capture the sophisticated pick-up and drop-off interactions or the nonlinear spatiotemporal relations that exist with the problem that we are addressing herein. One of the assumptions made in the study is the Gaussian distribution for any output at a specific time in the future. While this setup does enable uncertainty estimation, it limits the mapping function to just Gaussian distributions, and might vary significantly with any change in the baseline network structure. In addition, the model neither incorporates exogenous factors nor captures pick-up and drop-off interactions. In general, the literature indicates that while traditional machine learning models have shown better results than the classical time series models, they all have limitations in learning hidden features in order to capture the complex and dynamic spatiotemporal interactions.

A third and more recent stream of research in this field, has focused on leveraging the power of deep learning [61, 83, 92, 84, 89, 40, 29, 26, 100, 59]. This line of research is driven by the ability of neural networks to learn features and model nonlinear interactions. Xu et al. [89] introduced a deep learning based sequential model that remembers demand history and relevant information through LSTM units to anticipate taxi demand. A recent study by Tang et al. [79] proposed a multi-community passenger demand prediction model by leveraging a graph convolutional network. Temporal correlation is encoded by using a Gated Recurrent Unit (GRU), while the spatial correlation among regions is encoded in a graph, followed by a Louvain algorithm to generate predictions. Yao et al. [92] proposed a unified multi-view model that jointly captures the spatial, temporal and semantic relations. First, spatial dependency among nearby region is modeled by using a local CNN. Then, an LSTM model and graph embedding are used to capture the temporal and semantic aspects. These models are superior to traditional machine learning when it comes to learning complex spatiotemporal features. However, the models reviewed focus on next-step prediction, without any accompanying confidence measure. Also, the computational cost of these models is high due to the relatively large number of parameters that must be tuned with these.

### 3.2.2 Sequence to Sequence Learning with Neural Networks

In this subsection we examine prior work on other sequence learning applications of LSTMs. The sequence to sequence (*seq2seq*) architecture was first introduced to translate a sentence from French to English [78]. This work was a breakthrough in the field of natural language processing and also inspired researchers from other, different domains [36, 89]. The proposed architecture consists of two components: an encoder and a decoder. In the encoder, a stack of LSTM layers is used to map the input sentence to a vector. This mathematical representation is then fed to another multilayered LSTM (the decoder) to generate the sentence in English. It is worth mentioning that the input sentence and the generated translation can be of different lengths (no. of words) and orders (word sequence). The "listen attend and spell" (LAS) model extended this work to perform speech recognition, where an audio input generates an output that is a transcript [13]. In this study, at each input step the LSTM uses its hidden state to guide an attention mechanism to compute a "context" vector from the high level representation of the listener (encoder). The speller (decoder) uses the context along with the LSTM output to generate a probability distribution for the next word, conditioned on all previous outputs. The encoder-decoder framework has also been used in the field of image processing to perform complex tasks such as image restoration and image segmentation [8, 52].

### 3.2.3 Bayesian Neural Networks

Finally, we review the work related to estimation of uncertainty. A Bayesian Neural Network (BNN) is comprised of a probabilistic model along with a neural network. The design of a BNN aims to exploit the strengths of neural networks in a stochastic setting. The main difference between BNN and other designs is that weights have probability distributions attached to them. The probability distribution is used to capture uncertainty with respect to the best set of weights, and ultimately, can be used to measure prediction uncertainty. The network is given a prior distribution for the weights and the goal is to find the posterior distribution. However, an analytical solution for the posterior in neural networks doesn't always exist and can be hard to find even when it does exist. Lately, several methods have been proposed to approximate the posterior distribution [62, 32, 28, 49, 21, 47, 86]. The basic idea with most of these algorithms is variational inference. Parameters for the distribution of weights are learned rather than the weights themselves. The objective function of this approach is to minimize the Kullback-Leibler divergence between a prior assumed distribution and the true posterior distribution. However, this line of work requires working on different optimization problems guided by the adjusted loss functions. Also, the network architecture needs to be adjusted and the computational effort is impacted by the substantial growth in the number of learnable parameters.

Dropout is a well-known technique that serves as a regularization to avoid overfitting [75]. During the learning process, randomly selected neurons are dropped, but only during training, in order to reduce the generalization error. During testing, predictions are deterministic and no further random dropping is done. On the other hand, the Monte Carlo (MC) dropout approach employs dropout during both training and testing [25]. The prediction during testing is no longer deterministic, but depends on the randomly selected neurons. Therefore, there can be different predictions for a single data point. The predictions generated can be interpreted as samples from a probabilistic distribution. This framework does not require any change to the network architecture and provides model uncertainty estimates without any added computational complexity. A recent study discussed this framework and sources of randomness with respect to travel demand data [102]. The study showed that the prediction uncertainty can be divided into model uncertainty, inherent noise and model mis-specification. A BNN framework was proposed for time series prediction, along with uncertainty estimation by using MC dropout. However, this study was not for real-time prediction and considered only a macro-geographical level. Also, the spatiotemporal interaction was not part of the model.

Inspired by the BNN framework, in this study, the proposed model is designed to include MC dropout to enable uncertainty estimation for real-time multi-step predictions.

## 3.3 PRELIMINARIES

### 3.3.1 Background

We begin by briefly introducing the deep learning tools exploited in our implementation. A deep neural network is a composition of $L$ layers where layer $i$ is a functional representation of the input domain [64]. Each layer $i$ takes the output of the previous layer $i-1$ as input and passes it through a nonlinear activation function. The output of the layer is controlled by the choice of the activation function and a set of parameters $\boldsymbol{\theta}_i$ which are weights connecting the layers that form the entire network [23]. Thus, the final output $y$ from a neural network is obtained by performing the following sequence of computations, given an input $\mathbf{x} = (x_1, ..., x_T)$ to the first layer:

$$y = f_L(\boldsymbol{\theta}_L, f_{L-1}(\boldsymbol{\theta}_{L-1}, f_{L-2}(\boldsymbol{\theta}_{L-2}, \ldots, f_1(\boldsymbol{\theta}_1, \mathbf{x})))) \tag{3.1}$$

where $f_i$ is the activation function at layer $i$.

**3.3.1.1 Convolutional Network** Deep Convolutional Neural Networks (CNNs) have been successfully applied in many fields such as Natural Language Processing (NLP) and image recognition [42, 45]. Motivated by the success of CNNs, researchers have started exploiting them for time series analysis [91]. In image processing, CNNs apply a sliding filter over two dimensions (width and height). However, the sliding filter for time series is uni-dimensional and convoluted only across time. The general approach to applying CNN over a time stamp $t$ with a filter $F$ of length $l$ was described by Fawaz et al. [23] as follows:

$$C_t = \sigma(F \circledast X_{t-l/2:t+l/2} + b) \qquad \forall t \in T \tag{3.2}$$

where $C_t$ is the result of applying the filter $F$ on the uni-variate time series $X$ of length $T$ at time $t$, $b$ is a bias, and $\sigma$ is a nonlinear function. The weights of the filter are dependent on the dataset and have to be learned. A discriminative classifier resulting from a pooling operation follows the convolutional layer to ensure that we have a discriminative filter [23].

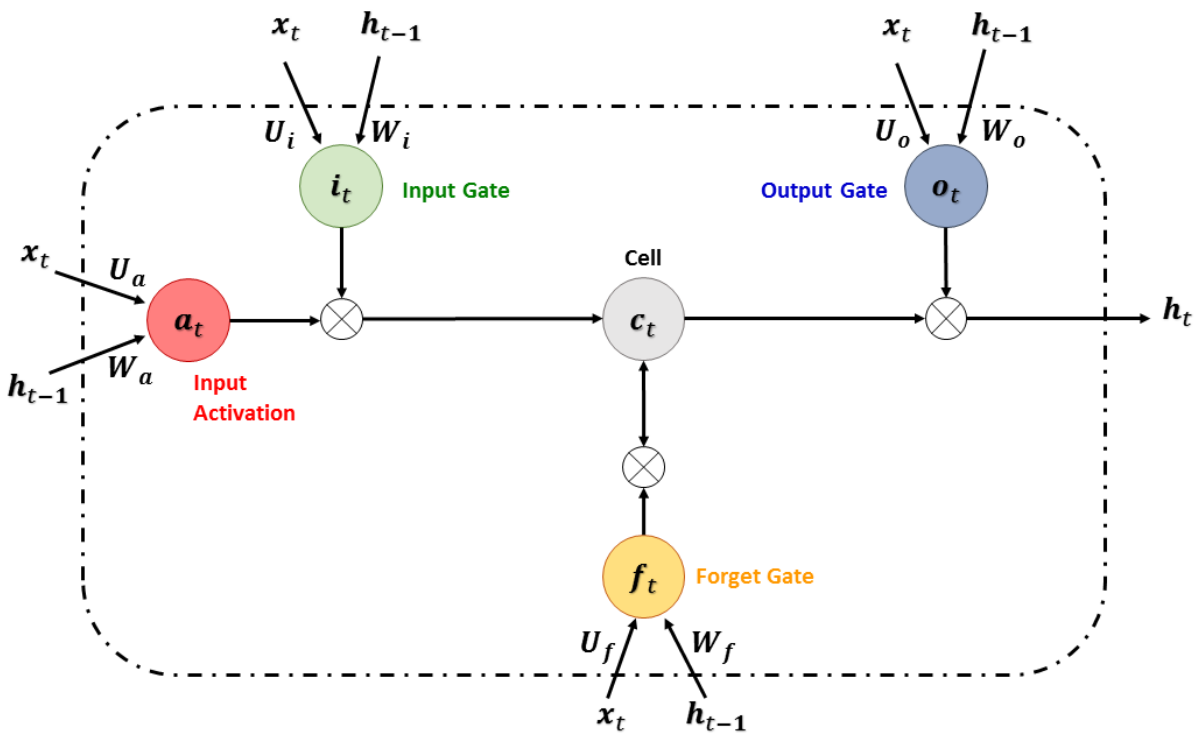


Figure 11: Detailed structure of Long Short-Term Memory (LSTM) cell [14]

A downsampling pooling operation takes the time series and reduces it to either a single value when using global pooling or to $\frac{T}{l}$ values with local pooling, such as max-pooling [23].

**3.3.1.2 Recurrent Neural Network** The idea behind recurrent neural networks (RNNs) is to make use of sequential information. RNNs are called "recurrent" due to the internal closed loop resulting from feedback connections. They provide recursive dynamics in the network, which capture nonlinear dependencies exhibited in time series data. In general, a recurrent model can be seen as a nonlinear dynamical system with a differentiable state-transition function. Suppose $\mathbf{x}_t \in \mathbb{R}^d$ is the input to the system at time $t$, and $\boldsymbol{\theta} \in \mathbb{R}^m$ is the parameter vector. Given two matrices of weights $\mathbf{W} \in \mathbb{R}^{n\times n}$, $\mathbf{U} \in \mathbb{R}^{n\times d}$, bias vector $\mathbf{b}_h \in \mathbb{R}^n$, and the nonlinear, differentiable state-transition function $\sigma$, the internal or hidden state of the system $\mathbf{h}_t \in \mathbb{R}^n$ at time $t$ for a recurrent neural network [14, 74, 54] is given by:

$$\mathbf{h}_t = \sigma(\mathbf{W}\,\mathbf{h}_{t-1} + \mathbf{U}\,\mathbf{x}_t + \mathbf{b}_h) \tag{3.3}$$

However, when the sequential data have long-term temporal dependency, standard RNNs might not be the proper choice. This is due to the vanishing (exploding) gradient problem, where the gradient of the loss function decays (soars) exponentially with time. This limitation leads to Long Short-Term Memory (LSTM), which is a special kind of RNN that is capable of learning long-term dependencies through a gating mechanism as illustrated in Figure 11 [35]. The core of an LSTM cell is the memory unit $c_t$. The main objective of this unit is to represent the information in the input sequence up to that point. The memory unit takes the previous hidden state $h_{t-1}$ and current input $x_t$, and produces the hidden state for the current step $h_t$. The main distinction between LSTM and RNN is the structure of the gating mechanism in LSTM cells, which includes three gates: input, forget, and output. The purpose of these gates is to regulate information flow to and from the memory cell. The forget gate decides which relevant information from the prior steps is important and needed. The input gate determines what essential information can be incorporated from the current step, and the output gate produces the next hidden state [14]. As an example in the context of our application, suppose that users are requesting transportation to a baseball game. Then the LSTM can remember that many requests had the same drop-off location and predict that demand will be high when the game is over. It can also forget this information for the same time during the following week if the data shows there is no scheduled game at that time. LSTMs perform the same task for every element of a sequence, with the output being reliant on the previous computations. More formally, let the internal state vector be a pair of vectors $\mathbf{s} = (\mathbf{c}, \mathbf{h})$. Then an LSTM layer has the following weights:

- Recurrent weights: $\mathbf{W}_a, \mathbf{W}_i, \mathbf{W}_f, \mathbf{W}_o\, \mathbf{W}_y \in \mathbb{R}^{n \times n}$
- Input weights: $\mathbf{U}_a, \mathbf{U}_i, \mathbf{U}_f, \mathbf{U}_o \in \mathbb{R}^{n \times d}$
- Bias weights: $\mathbf{b}_a, \mathbf{b}_i, \mathbf{b}_f, \mathbf{b}_o\, \mathbf{b}_y \in \mathbb{R}^{n}$

The state-transition function of the LSTM (forward pass at a layer) can be written as [74] [54]:

$$\mathbf{a}_t = \sigma(\mathbf{U}_a\ x_t + \mathbf{W}_a\ \mathbf{h}_{t-1} + \mathbf{b}_a) \tag{3.4}$$

$$\mathbf{i}_t = \sigma(\mathbf{U}_i\ \mathbf{x}_t + \mathbf{W}_i\ \mathbf{h}_{t-1} + \mathbf{b}_i) \tag{3.5}$$

r

$$\mathbf{f}_t = \sigma(\mathbf{U}_f\ \mathbf{x}_t + \mathbf{W}_f\ \mathbf{h}_{t-1} + \mathbf{b}_f) \tag{3.6}$$

$$\mathbf{o}_t = \sigma(\mathbf{U}_o\ \mathbf{x}_t + \mathbf{W}_o\ \mathbf{h}_{t-1} + \mathbf{b}_o) \tag{3.7}$$

$$\mathbf{c}_t = \mathbf{a}_t \odot \mathbf{i}_t + \mathbf{f}_t \odot \mathbf{c}_{t-1} \tag{3.8}$$

$$\mathbf{h}_t = \tanh(\mathbf{c}_t) \odot \mathbf{o}_t \tag{3.9}$$

$$\hat{\mathbf{y}}_t = \mathbf{W}_y\ \mathbf{h}_t + \mathbf{b}_y \tag{3.10}$$

where $\mathbf{a}_t$, $\mathbf{i}_t$, $\mathbf{f}_t$, $\mathbf{o}_t$ are the input activation, input gate, forget gate, and output gate, respectively, $\sigma$ is a point-wise nonlinear activation function (e.g. ReLU, sigmoid), and $\hat{\mathbf{y}}_t$ is the predicted output at time $t$. A bidirectional LSTM is a variation of LSTM that takes into consideration the relationship between the current output and the subsequent as well as the previous values [101]. For instance, in text translation the context is important to predict the next word, which in turn, can be strongly related to words that come later in the sentence. In the Bidirectional setting, the forward LSTM passes through the input sequence in order from $\mathbf{x}_1$ to $\mathbf{x}_T$ and produces a hidden forward sequence $\overrightarrow{\mathbf{h}}_t$, while the backward LSTM passes through the input in reverse order and computes a backward hidden sequence $\overleftarrow{\mathbf{h}}_t$. The two sequences are then combined to compute the output [101]:

$$\overrightarrow{\mathbf{h}}_t = \sigma(\mathbf{U}_{\overrightarrow{\mathbf{h}}}\mathbf{x}_t + \mathbf{W}_{\overrightarrow{\mathbf{h}}}\overrightarrow{\mathbf{h}}_{t-1} + \mathbf{b}_{\overrightarrow{\mathbf{h}}}) \tag{3.11}$$

$$\overleftarrow{\mathbf{h}}_t = \sigma(\mathbf{U}_{\overleftarrow{\mathbf{h}}}\mathbf{x}_t + \mathbf{W}_{\overleftarrow{\mathbf{h}}}\overleftarrow{\mathbf{h}}_{t+1} + \mathbf{b}_{\overleftarrow{\mathbf{h}}}) \tag{3.12}$$

$$\hat{\mathbf{y}}_t = \mathbf{W}_{\overrightarrow{\mathbf{y}}}\overrightarrow{\mathbf{h}}_t + \mathbf{W}_{\overleftarrow{\mathbf{y}}}\overleftarrow{\mathbf{h}}_t + \mathbf{b}_y \tag{3.13}$$

### 3.3.2 Definitions

The city to be studied is split into $L$ micro-geographical zones, and the aggregate demand per unit time (e.g., 1 hour) for each zone is obtained. This pre-processing step converts the data into a time-sequence of demand in each zone.

*Definition 1*: Let $\mathbf{D}_t = (d_t^1, \ldots, d_t^L)$ be a demand vector representing the demand at locations (1,2,...,$L$) for a specific time period $t$. Each element in this vector indicates the demand (i.e. number of trip requests) for location $l$ in time period $t$.

*Definition 2*: Let $\mathbf{V}_t = (v_t^1, \ldots, v_t^L)$ represents the number of drop-offs at locations (1,2,...,$L$) in time period $t$.

*Definition 3*: Let $\mathbf{J}_t^D = (\mathbf{D}_{t-k}, \ldots.., \mathbf{D}_t)$ be a demand (pick-up) trajectory for all locations, i.e., a time-ordered sequence where $k$ is a hyper-parameter for the look-back period that we are going to specify in our implementation.

*Definition 4*: Let $\mathbf{J}_t^V = (\mathbf{V}_{t-k}, \ldots.., \mathbf{V}_t)$ represent a drop-off trajectory over all locations over the same look-back period.

Note that we use $\mathbf{J}_t^V$ to explicitly consider the drop-off sequence and its effect on demand. The two trajectories for pick-up and drop-off are combined to form the input vector $\mathbf{J}_t =$

$\{\mathbf{J}_t^D \cup \mathbf{J}_t^V\}$. The goal of our model is to predict demand at all locations over the next $F$ time periods $(\mathbf{D}_{t+1}, ....., \mathbf{D}_{t+F})$, based upon the trajectory $(\mathbf{J}_t)$.

*Definition 5*: Let $\hat{\mathbf{D}}_t = (\hat{\mathbf{d}}_t^1, \ldots, \hat{\mathbf{d}}_t^L)$ be the predicted demand (i.e. number of trip requests) at all locations for a specific time period $t$. The element $l$ in this vector indicates the predicted demand for location $l$ at time period $t$.

*Definition 6*: The predicted demand trajectory for all locations $\hat{\mathbf{J}}_t = (\hat{\mathbf{D}}_t, \ldots.., \hat{\mathbf{D}}_{t+F})$, is a time-ordered sequence where $F$ is a hyper-parameter for how many steps in the future to forecast.

## 3.4 Multi-Stage Encoder Decoder Framework

We now describe our two-stage deep learning model in detail. The objective of the first stage is to learn the best latent representations for the spatiotemporal travel demand data, while the objective of the second stage is to incorporate exogenous factors in order to generate more accurate predictions. Our approach leverages an integrated architecture that receives historical time-series data on demand and drop-off along with exogenous factors, and generates multi-step ahead travel demand predictions. The purpose of the two-stage modeling approach is to enhance the learning process: in the first stage the focus is on the historical time-series data and on extracting a suitable embedding that captures only demand/drop-off history. In the second stage, this pure time-series embedding is further exploited by considering exogenous factors in order to obtain more accurate predictions for multiple steps ahead. This two-stage sequence helps with de-noising the time-series by mapping it to a latent representation before incorporating the exogenous factors, which in turn helps our predictive model generalize beyond the training data.

### 3.4.1 First Stage

In this stage we adopt an encoder-decoder structure, which is commonly used for sequence to sequence prediction, to be trained on the given data. In a nutshell, the encoder maps the

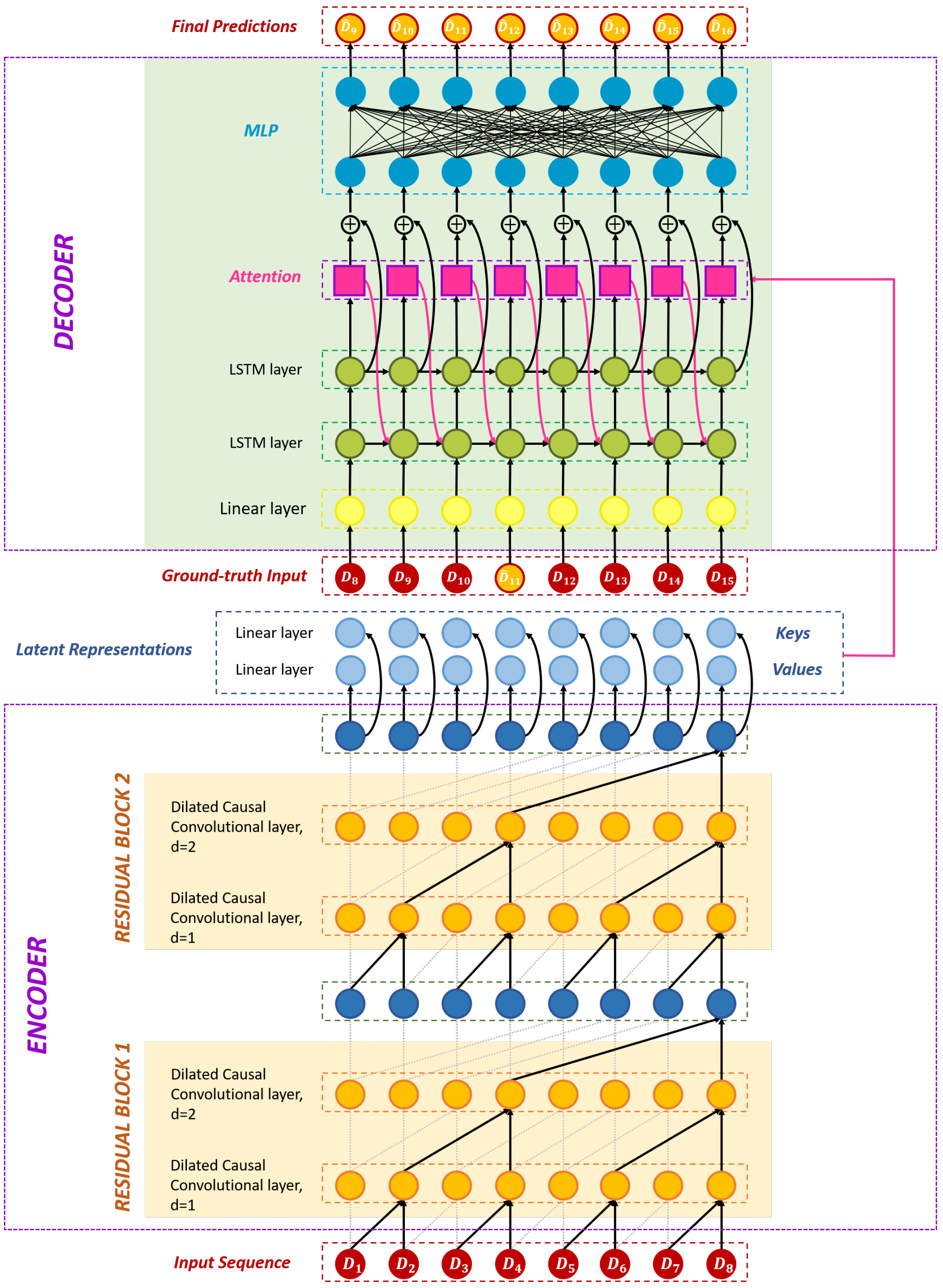

Final Predictions
$D_9$ $D_{10}$ $D_{11}$ $D_{12}$ $D_{13}$ $D_{14}$ $D_{15}$ $D_{16}$
DECODER
MLP
Attention
LSTM layer
LSTM layer
Linear layer
Ground-truth Input
$D_8$ $D_9$ $D_{10}$ $D_{11}$ $D_{12}$ $D_{13}$ $D_{14}$ $D_{15}$
Latent Representations
Linear layer
Linear layer
Keys
Values
ENCODER
RESIDUAL BLOCK 2
Dilated Causal Convolutional layer, d=2
Dilated Causal Convolutional layer, d=1
RESIDUAL BLOCK 1
Dilated Causal Convolutional layer, d=2
Dilated Causal Convolutional layer, d=1
Input Sequence
$D_1$ $D_2$ $D_3$ $D_4$ $D_5$ $D_6$ $D_7$ $D_8$

Figure 12: Illustration of region-based travel demand model

input sequence onto some latent space of fixed dimension (i.e., a latent representation), and this mapping is then fed to the decoder to generate a demand sequence in future time steps. Our model is illustrated in Figure 12.

The encoder of our model employs TCN, which leverages parameter-sharing and local connectivity of convolutional layers to reduce the total number of trainable parameters, thereby achieving more efficient computational performance. TCN is a generic term that represents a family of architectures with two distinguishing characteristics: 1) the architecture can only handle input-output sequences of the same length; and 2) the convolutions in the architecture are causal, i.e., an output $y_t$ at time $t$ is convolved only with elements $y_\tau$ in the previous layer, where $\tau \leq t$ [9].

TCN may be viewed as a set of stacked, dilated causal convolution layers. Dilated causal convolutions are preferred to simple causal convolutions because they allow the receptive field to grow exponentially with every additional layer. In other words, a larger dilation enables a wider range of inputs to be represented by an output at the top level. More formally, given an input sequence, $\mathbf{x} \in \mathbb{R}^k$ and a kernel function $\psi(.) : \{0, \ldots, N-1\} \longrightarrow \mathbb{R}$, the dilated convolution $\mathbf{C}(.)$ on element $s$ of the sequence is defined as

$$\mathbf{C}(s) = (\psi \circledast_d \mathbf{x})(s) = \sum_{i=0}^{N-1} \psi(i) \, . \, \mathbf{x}_{s-d*i}, \tag{3.14}$$

where $d = 2^\eta$ is the dilation factor, $\eta$ is the depth of the network, $N$ is the kernel size, and $s - d * i$ accounts for the direction of the past. Dilation can be seen as a sweep of fixed step-size between every two adjacent kernels. To ensure that there is some kernel convolving with each element within the history while allowing for an extremely large effective history using deep networks, $d$ is increased exponentially with the depth of the network $\eta$ [9]. For example, consider Figure 12 (i.e. input sequence $\mathbf{x} = \mathbf{D}$), and suppose the index $s = 8$. Then the dilated convolution $\mathbf{C}(.)$ of factor $d = 1$ and kernel size $N = 2$ will be:

$$\mathbf{C}(s) = \sum_{i=0}^{1} \psi(i) \, . \, \mathbf{D}_{8-d.i} = \psi(0) \, . \, \mathbf{D}_8 + \psi(1) \, . \, \mathbf{D}_7 \tag{3.15}$$

A residual block stacks two dilated causal convolution layers together, and the results from the final convolution are added back to the input to obtain the output of the block. If there is a dimension mismatch between the width (number of in-channels) of the inputs and the width (number of out-channels) of the second dilated causal convolution layer, a 1-D convolution is applied to the input before adding the block outputs to make the dimensions match. Weight normalization is applied to the kernels of both layers, followed by rectified linear units (ReLU) and a spatial dropout for regularization. Residual blocks effectively help avoid the problem of exploding/vanishing gradients.The output (latent representations) of the TCN is stored in two parallel linear layers, which produce the so-called `keys` and `values` as shown in Figure 12.

The ground-truth sequence containing the last demand element in the input sequence along with the subsequent ones, is used to initialize the decoder. The decoder predicts the sequence of future travel demand at each location. The decoder of our model consists of a linear layer, followed by two LSTM layers, an attention model, and a two-layer MLP.

For convergence efficiency and learning stability, we employ teacher-forcing. This is a technique for efficiently training recurrent neural networks that utilizes both the ground truth and the model prediction output from the previous time step. In a nutshell, teacher-forcing generally selects the ground truth from a prior time step and passes it on as an input. However, it occasionally (as determined by some specified probability $p$) passes the prediction generated by the model from a previous time step as an input instead of the ground truth. This is illustrated in Figure 12, where we pass the predicted demand $\mathbf{\hat{D}_{11}}$ to the decoder instead of the ground truth $\mathbf{D_{11}}$. Teacher-forcing guards against overfitting and ensures better generalization.

The teacher-forced input sequence is passed to a linear layer whose output $\nu_\tau$ is then fed to the first LSTM layer. The first cell in the first LSTM layer uses the output of the linear layer $\nu_\tau$ to initialize its internal state $s^l_\tau$, while its hidden state $h^l_\tau$ is initialized randomly. These are both used to then produce both internal and hidden states for the next time step or layer. Other cells take internal states from previous layers $s^{l-1}_\tau$, hidden states from the prior time step $h^l_{\tau-1}$ and the attention context from the prior time step $c_\tau$ to generate $\{s^l_\tau, h^l_\tau\}$. For further details see the training algorithm (Algorithm 1).

Attention is an interface that monitors the flow of contextual information from the encoder to the decoder. It allows the model to pay more attention to significant elements of the input sequence and learn the association between them [60]. The attention mechanism varies its focus for different parts of the sequence by assigning a score to each element in the sequence. More precisely, at each decoder time step $\tau$, the attention model computes a score, also known as energy, by applying batch matrix multiplication between the current output of the last LSTM layer ($q_\tau$) and its corresponding element of the `keys` sequence at encoder time step $t$ ($k_t$) as

$$e_{t,\tau} = k_t^T \, q_\tau, \tag{3.16}$$

where $e_{t,\tau}$ denote the energy computed at decoder time step $\tau$ and encoder time step $t$. Then, a normalized vector over the encoder times steps ($\alpha_{t,\tau}$) is derived using the softmax function as

$$\alpha_{t,\tau} = \frac{\exp\left(e_{t,\tau}\right)}{\sum\limits_t \exp\left(e_{t,\tau}\right)} \tag{3.17}$$

Finally, the normalized vector is used to compute the attention context vector ($c_\tau$) at each decoder time step $\tau$. This operation is carried out by applying batch matrix multiplication between the energy ($e_{t,\tau}$) at decoder time step $\tau$, and its corresponding element of the `values` sequence ($v_t$) as

$$c_\tau = \sum_t \alpha_{t,\tau} \, v_t \tag{3.18}$$

The attention context vector is then concatenated with the current output of the last LSTM layer ($q_t$) and fed to an MLP to generate the final predictions of the travel demand sequence ($\hat{\mathbf{D}}_t$).

**Algorithm 1** Training of MSP-TCN

```
Input: training data $\{\mathbf{J}_t\}_{t=1}^{M}$, window size $R$
Output: $\{\hat{\mathbf{J}}_t\}_{M+t=1}^{M+R-1}$; MSP-TCN model $\mathcal{M}(.)$
    *First-stage training:*
    Initialize the encoder $\mathcal{G}(.)$, and decoder $\mathcal{F}(.)$ in Fig. 12
    while stopping criteria not met do
      pick a batch of instances uniformly from the training data
      for $t = 1, \ldots,$ do
        $\mathbf{k}_t, \mathbf{v}_t = \mathcal{G}(\mathbf{J}_t)$
        pick $\mathbf{J}_t = \{\mathbf{D}_\tau, \mathbf{V}_\tau\}_{\tau=t-k}^{t}$ such that $\forall\tau$,
            $$\mathbf{D}_\tau = \begin{cases} \hat{\mathbf{D}}_\tau, \hat{\mathbf{V}}_\tau & \text{w.p. } p \\ \mathbf{D}_\tau, \mathbf{V}_\tau, & \text{otherwise} \end{cases}$$
        $\{\nu_{t-k}, \ldots, \nu_t\} = \phi(\mathbf{J}_t)$, where $\phi(.)$ is a linear layer
        For each LSTM cell and hidden state,
            $$\{\mathbf{s}_\tau, \mathbf{h}_\tau\}_{\tau=t-k}^{t} = \begin{cases} lstm(\nu_\tau, \mathbf{h}_{\tau-1}^{l}, \mathbf{c}_{\tau-1}), \text{ if } l = 1 \\ lstm(\mathbf{s}_\tau^{l-1}, \mathbf{h}_{\text{initialized}}^{l}, \mathbf{0}), \text{ if } \tau = 1 \\ lstm(\mathbf{s}_\tau^{l-1}, \mathbf{h}_{\tau-1}^{l}, \mathbf{c}_{\tau-1}), \text{ otherwise} \end{cases}$$
        use $\mathbf{k}_t, \mathbf{v}_t$ to compute $c_\tau$ as in (3.16) - (3.18).
        $i_\tau = concatenate[c_\tau, \mathbf{s}_\tau]$
        $\hat{\mathbf{J}}_t = MLP(\{i\}_{\tau=t-k}^{t})$
        perform backward passes to update parameters of $\mathcal{G}(.)$ and $\mathcal{F}(.)$ by minimizing the loss
        function $\mathcal{L}(\{\hat{\mathbf{J}}_t\}_{t=M+1}^{M+R+1}, \{\mathbf{J}_t\}_{t=M+1}^{M+R+1})$
      end for
    end while
    output the trained encoder model $\mathcal{G}(.)$
    *Second-stage training:*
    while stopping criteria not met do
      pick a batch of instances uniformly from the training data
      for $t = 1, \ldots,$ do
        $\mathbf{k}_t, \mathbf{v}_t = \mathcal{G}(\mathbf{J}_t)$
        $o_\tau = concatenate[g_t, \mathbf{k}_t, \mathbf{v}_t]$
        $\{\hat{\mathbf{J}}_t\}_{t=M+1}^{M+R+1} = MLP(\{o\}_{t=M+1}^{M+R+1})$
        perform backward passes to update parameters of $MLP(.)$ by minimizing the loss function
        $\mathcal{L}(\{\hat{\mathbf{J}}_t\}_{M+t=1}^{M+R+1}, \{\mathbf{J}_t\}_{t=M+1}^{M+R+1})$
      end for
    end while
    return $\{\hat{\mathbf{J}}_t\}_{t=M+1}^{M+R+1}$; trained MSP-TCN model $\mathcal{M}(.)$
```

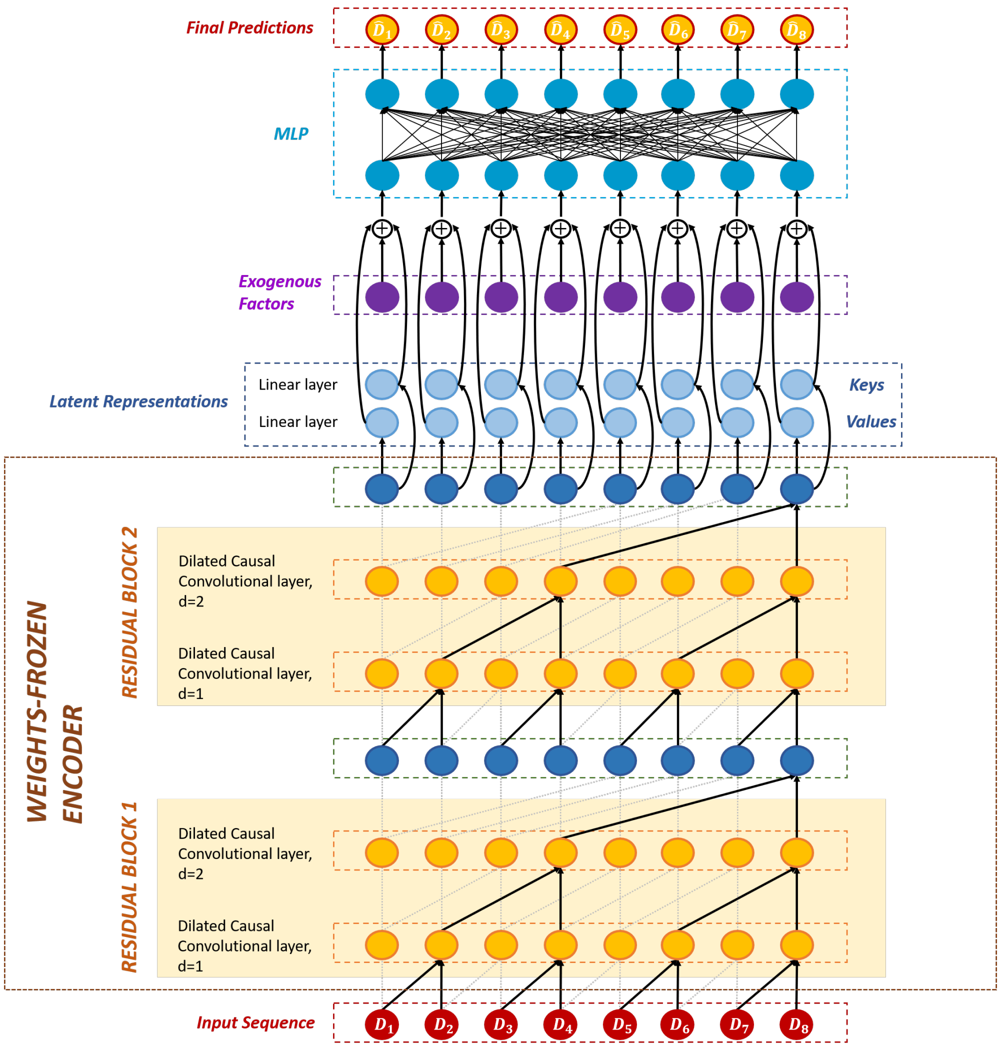

Final Predictions
$\hat{D}_1$ $\hat{D}_2$ $\hat{D}_3$ $\hat{D}_4$ $\hat{D}_5$ $\hat{D}_6$ $\hat{D}_7$ $\hat{D}_8$
MLP
Exogenous Factors
Latent Representations
Linear layer
Linear layer
Keys
Values
WEIGHTS-FROZEN ENCODER
RESIDUAL BLOCK 2
Dilated Causal Convolutional layer, d=2
Dilated Causal Convolutional layer, d=1
RESIDUAL BLOCK 1
Dilated Causal Convolutional layer, d=2
Dilated Causal Convolutional layer, d=1
Input Sequence
$D_1$ $D_2$ $D_3$ $D_4$ $D_5$ $D_6$ $D_7$ $D_8$

Figure 13: Illustration of the second stage

**Algorithm 2** Inference of MSP-TCN

```
Input: testing data $\{\mathbf{J}_t\}_{t=P}^{Q}$; trained MSP-TCN model $\mathcal{M}(.)$, exogenous factors vector $g_t$,
    dropout probability $p$, number of iterations $N$
Output: prediction mean $\mu_{\mathbf{J}_t}$ and uncertainty $\xi_{\mathbf{J}_t}$
1: for $i = 1, \dots, N$ do
2:     $\hat{\mathbf{J}}_t^i = Dropout(\mathcal{M}(\mathbf{J}_t, g_t), p)$
3: end for
4: $\mu_{\mathbf{J}_t} = \frac{1}{N} \sum_{i=1}^{N} \hat{\mathbf{J}}_t^i$
5: $\xi_{\mathbf{J}_t}^2 = \frac{1}{N} \sum_{i=1}^{N} (\hat{\mathbf{J}}_t^i - \mu_{\mathbf{J}_t})^2$
6: return $\mu_{\mathbf{J}_t}$, $\xi_{\mathbf{J}_t}$
```

### 3.4.2 Second-Stage

The objective of the second stage is to perform transfer learning from the first stage and incorporate the vector $\mathbf{g}_t$ representing exogenous factors, in order to generate more accurate travel demand predictions; details about $\mathbf{g}_t$ are provided where our experimental study is discussed. Our model is illustrated in Figure 13. The trained encoder of the first stage is used with the weights frozen in order to produce the best latent representations, $\mathbf{k}_t$ and $\mathbf{v}_t$, of the spatiotemporal travel demand data. The latent representations are concatenated with the exogenous factors and fed as input to the MLP, which generates the final travel demand sequence predictions at future time steps. The only trainable part in the second stage is the MLP, and thus the computational burden is much lower here.

## 3.5 Experimental Study

In this section, we evaluate and compare the performance of our multi-stage probabilistic TCN (MSP-TCN) model to other advanced models. We first describe the datasets used in the study as well as the experimental settings, and we then discuss the results in details. The city we consider is New York City, with two types of transport services: traditional

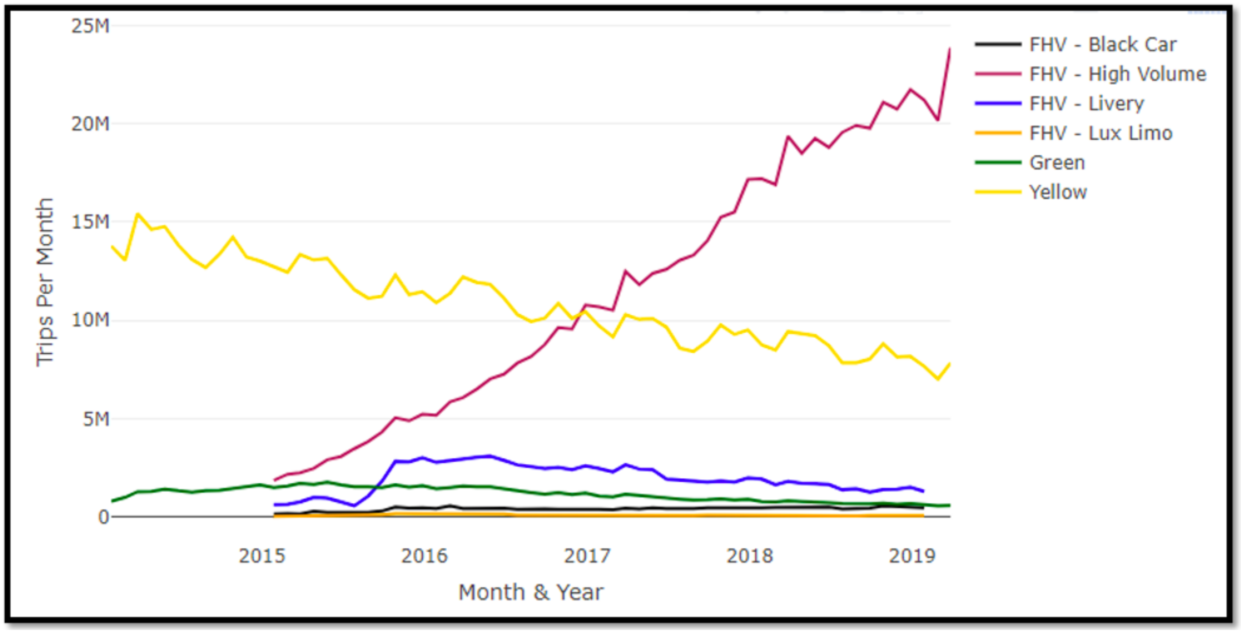


Figure 14: Travel Demand in NYC [6]

ride-hailing (Yellow and Green Taxis), and For-Hire Vehicles (FHV). The latter includes all ride-sharing companies, which are refereed to as High Volume For-Hire Vehicles (HVFHV).

### 3.5.1 Datasets

Our experiments are conducted on two real-world datasets. The first dataset addresses demand for ride-hailing service and in particular, for Yellow Taxis. The second dataset is for the largest ride-sharing service provider in the city (Uber). We test our model on data from both services. As Figure 14 shows, there has been a substantial shift from ride-hailing to car-sharing between 2015 and 2019. We use 2018 data in our experiments, where the average monthly demand for Yellow Taxis is around 8 million, compared to approximately 21 million trips per month for HVFHV (Figure 15). The data was obtained from the NYC Taxi and Limousine Commission (TLC).

Figure 16 shows a heat map of average hourly demand for both Uber and Yellow Taxis over the entire New York area, which is divided into 265 pick-up zones. The darker a zone,

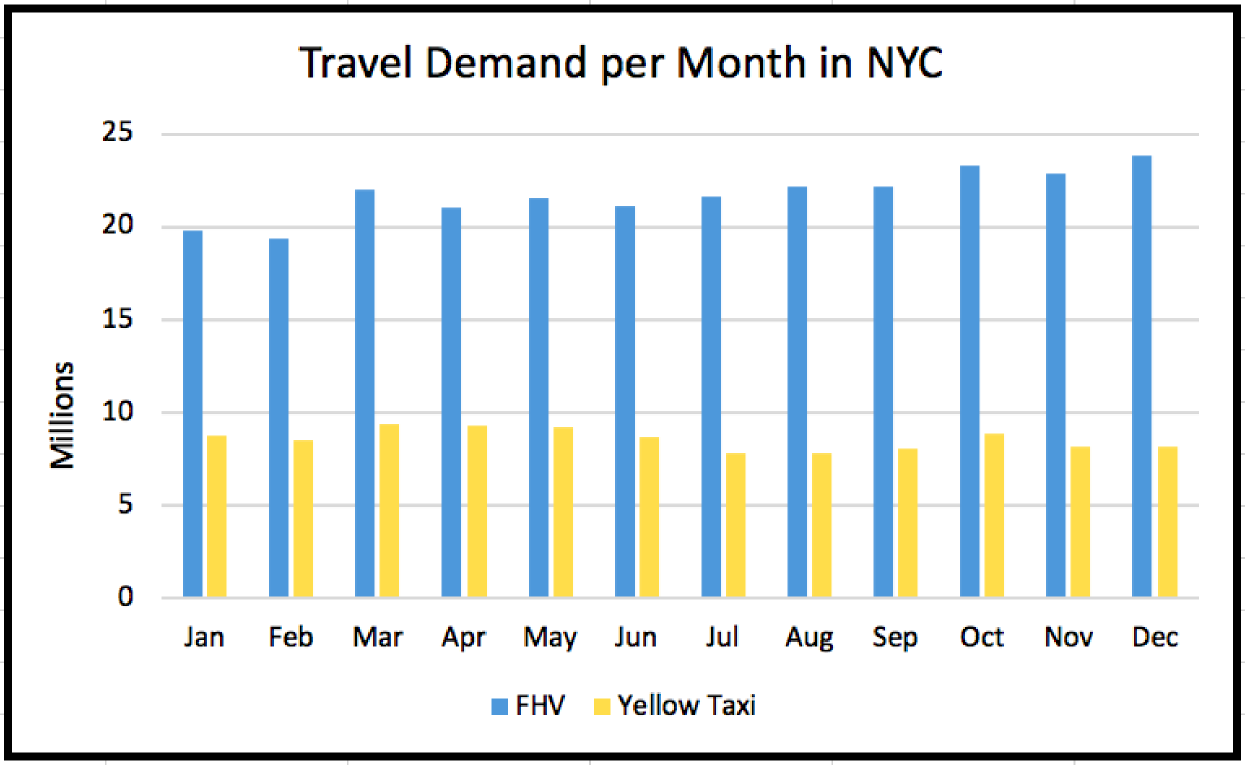


Figure 15: Trip requests per month in NYC for yellow taxi and HVFHV [6]

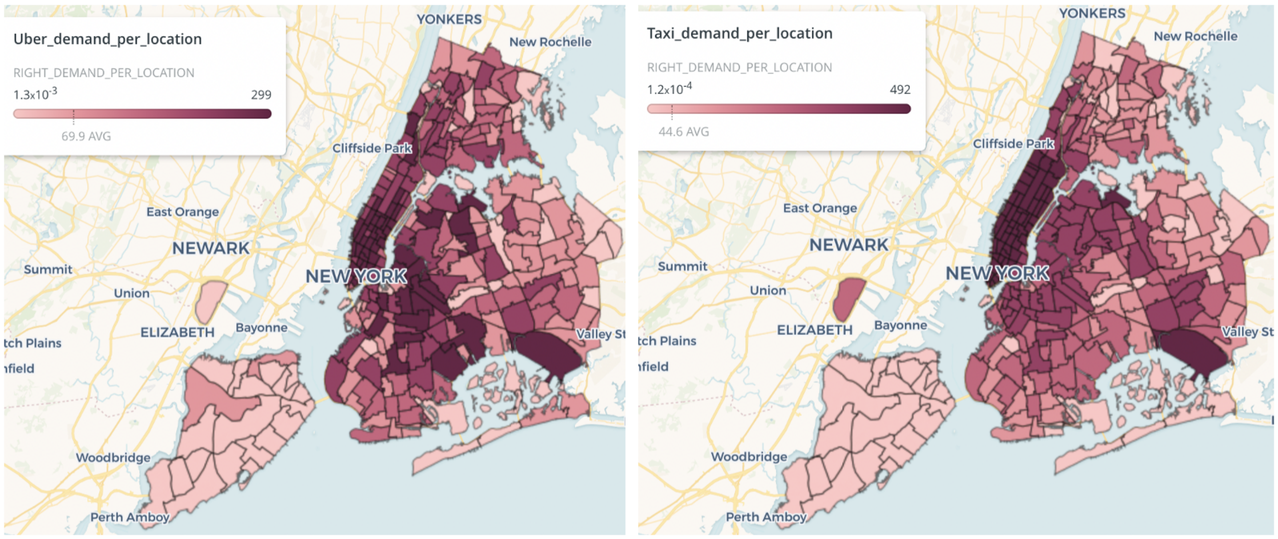


Figure 16: Hourly average of actual demand per location

the higher is the average demand that it has. For Yellow Taxis, it appears that the majority of demand is generated from zones within Manhattan and from JFK international airport. Uber's demands exhibit a similar pattern, but with additional high demand locations in some zones in Brooklyn as well as a few in Queens.

As one might expect in practice, an examination of the data showed numerous defective records (e.g., negative or extremely high values for distance/payment). These records were dropped, but given the size of our dataset, the number of deleted records represents less than 0.001% of the data. We used a well defined micro-geographical breakdown to represent the pick-up zones according to the boundaries predetermined by authorities. This breakdown enables real-time prediction at a micro-spatial level. We computed the aggregate number of trips per hour for each zone, and the trip records were further pre-processed to convert the data into a proper format that can be used for sequential modeling. The sequence to be fed to the Stage 1 model is a trajectory of pick-up and drop-off data. For the exogenous factors in the Stage 2 Model, we consider weather, and additional temporal information such as day of the week and holidays. Weather data was extracted from the official weather information for NYC from the National Oceanic and Atmospheric Administration (NOAA), and pre-processed to only include temperature, humidity, wind speed, weather description and a binary attribute to identify extreme weather conditions. We also added temporal clustering to our exogenous factors, which clusters specific times of the day/week based on their historical average demand and variation. This information helps the model understand the demand behavior specific to a particular time. The temporal clustering attributes encapsulate this additional information and feed it to the model in the second stage to improve its performance. A hierarchical clustering technique was used and resulted in 10 clusters. Furthermore, we also normalized all inputs, demand and pick-up trajectories and exogenous factors prior to the training process so as to ensure learning stability and efficiency [67].

### 3.5.2 Experimental Setup

We compare (MSP-TCN) with the following five models, ranging from simple to more sophisticated ones. In all cases, parameters are fine-tuned and best performance is reported:

1. Multi-Layer Perceptron (MLP): Four fully connected linear layers with batch normalization and ReLU activation.
2. Vanilla LSTM: An LSTM model that has a single hidden layer of LSTM units, and an output layer used to make a prediction.
3. Multi-LSTM: Three LSTM layers followed by an output layer.
4. seq2seq ([31]): An encoder-decoder with 2 BLSTM layers.
5. Modified LAS ([13]): LAS is a sequence model with attention mechanism that is based on the encoder-decoder framework. The encoder consists of 3 pyramidal bidirectional LSTM (pBLSTM) layers and the decoder is composed of 2 BLSTM layers. A convolution layer was added to the encoder described in this study which enhanced the learning of this model. The major characteristic of the convolutional layer is that it learns local features by convolving kernels. Hence, it works as an automatic feature extractor.

We also separately evaluate the impact of adding the drop-off information (as it could have an impact on the pick-up information), and exogenous factors on the model's performance.

We implement all the methods in Python and use PyTorch for the neural network-based approaches on Google Colab Pro. The MSP-TCN is trained using Adam optimizer with a learning rate of 0.001 without momentum for both datasets. In terms of additional parameters associated with the architecture of MSP-TCN, we performed extensive experimentation and fine-tuning to get to the best possible performance (local minimum). The hidden sizes of dilated convolution, linear, and LSTM layers are 528, 256, and 256, respectively. We have used 8 workers with a batch size of 32 for both datasets. Our independent test dataset covers a total of $T = 2891$ time steps for each location in the city. The look back window we consider in our model is 50 time steps and for each location and each time step we obtain forecasts for the next 50 steps. In stage 1, we have 2 residual blocks with TCN architecture with dilated convolution layers with a hidden size of 528. The two stacked linear layers representing the output of the encoder have 265 neurons each.

We study the performance of our model and each of the aforementioned models using three different metrics: Root Mean Square Error (RMSE), Symmetric Mean Absolute Percentage Error (sMAPE), and Smooth L1 (sL1). The last measure, also called Huber loss, is less sensitive to outliers and uses a squared term only if the absolute error is less than 1.

$$RMSE = \frac{1}{K * T} * \sqrt{\sum_{t=1}^{T} \sum_{k=1}^{K} (\mathbf{D}_t^k - \hat{\mathbf{D}}_t^k)^2} \tag{3.19}$$

$$sMAPE = \frac{1}{K * T} * \sum_{t=1}^{T} \sum_{k=1}^{K} \frac{|\mathbf{D}_t^k - \hat{\mathbf{D}}_t^k|}{(|\mathbf{D}_t^k| - |\hat{\mathbf{D}}_t^k|)/2 + 1} \tag{3.20}$$

$$smoothL1 = \frac{1}{K * T} * \sum_{t=1}^{T} \sum_{k=1}^{K} \mathbf{z}_t^k \tag{3.21}$$

where $\mathbf{z}_t^k$ is given by:

$$\mathbf{z}_t^k = \begin{cases} 0.5 * (\mathbf{D}_t^k - \hat{\mathbf{D}}_t^k)^2, & \text{if } |\mathbf{D}_t^k - \hat{\mathbf{D}}_t^k| < 1 \\ |\mathbf{D}_t^k - \hat{\mathbf{D}}_t^k| - 0.5, & \text{otherwise} \end{cases} \tag{3.22}$$

Table 1: Performance of the Region-based Travel Demand Models

| Dataset | Model | RMSE | sMAPE | sL1 |
|---|---|---|---|---|
| Uber | MLP | 0.1446 | 0.1147 | 0.0107 |
| | Vanilla LSTM | 0.1828 | 0.1165 | 0.0112 |
| | multi-LSTM | 0.1381 | 0.1074 | 0.0093 |
| | seq2seq | 0.1133 | 0.0965 | 0.0076 |
| | modified LAS | 0.0983 | 0.0853 | 0.0064 |
| | **MSP-TCN** | **0.0971** | **0.0793** | **0.0055** |
| Yellow Taxi | MLP | 0.0746 | 0.0499 | 0.0030 |
| | Vanilla LSTM | 0.0752 | 0.0530 | 0.0031 |
| | multi-LSTM | 0.0691 | 0.0492 | 0.0026 |
| | seq2seq | 0.0568 | 0.0346 | 0.0019 |
| | modified LAS | 0.0421 | 0.0289 | **0.0011** |
| | **MSP-TCN** | **0.0396** | **0.0276** | **0.0011** |

Note that we have a total of $K = L * 50$ forecasts for each time step. Here $t$ is the timestamp order in the sequence, $k$ is a forecast index, and $\mathbf{D}_t^k$ and $\hat{\mathbf{D}}_t^k$ are the actual and predicted demand vectors corresponding to forecast $k$.

In addition to the three performance metrics we are also interested in studying the uncertainty associated with our forecasts (Algorithm 2). For this, we keep the dropout layers activated during inference and run the model iteratively for 1000 epochs. We focus only on the one-step-ahead forecasts, and for each location and each time step in our test dataset we thus have a total of 1000 separate one-step-ahead forecasts. This allows us to build a suitable confidence interval for each of the forecasts (we constructed both 90% and 95% intervals). The metric we use to evaluate the model's uncertainty estimation is the coverage probability (CP), which is the proportion of the confidence intervals built for the one-step-ahead forecasts at each time step and each location that contain the true value of the demand being forecast.

### 3.5.3 Results

Table 1 shows the performance, based on our three metrics, of our model and the other five models, evaluated over the entire test dataset for both the Uber and the Yellow Taxi datasets. The first thing that stands out is that forecasts for the Yellow Taxi dataset are better than those for the Uber dataset for all models. This is due to the fact that there is less variability in demand (this is borne out by the sMAPE values, which are higher for the Uber dataset than for the Yellow Taxi dataset, indicating that the former has higher variation). The sL1 values show that even by reducing the effect of outliers, the error is still higher for the Uber dataset.

In looking at the relative performance of the different models, as one might expect, the simplest methods - MLP and Vanilla LSTM - have the poorest performance. Although Vanilla LSTM is designed to capture long-term relationships, MLP actually outperforms it by a small amount, and even though this might appear counter-intuitive, the depth of MLP seems to provide a more powerful representation. The multi-LSTM clearly outperforms the first two models, because it possesses both LSTM cells and network depth. Adopting the

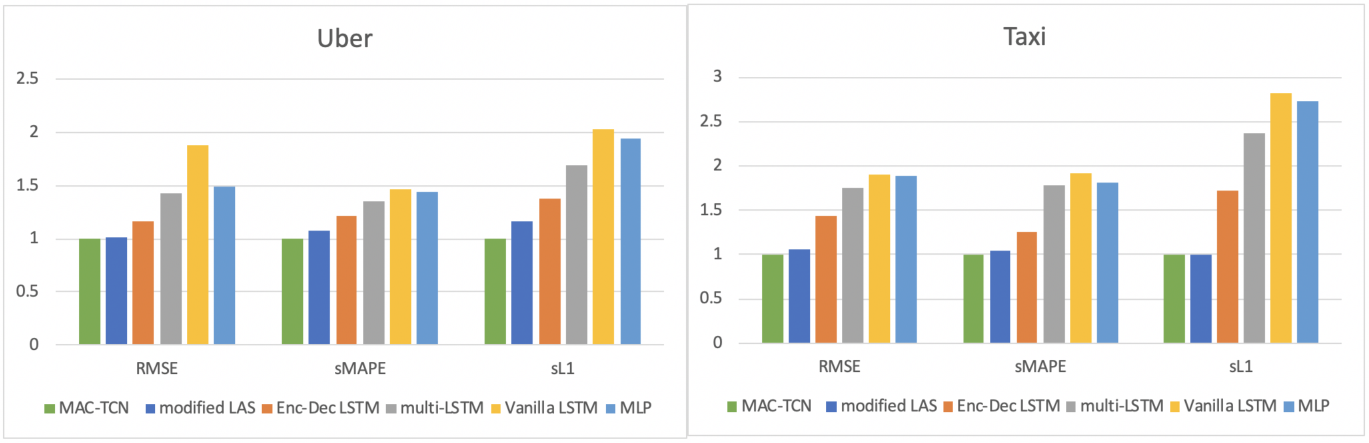


Figure 17: Relative Performance of the Different Models

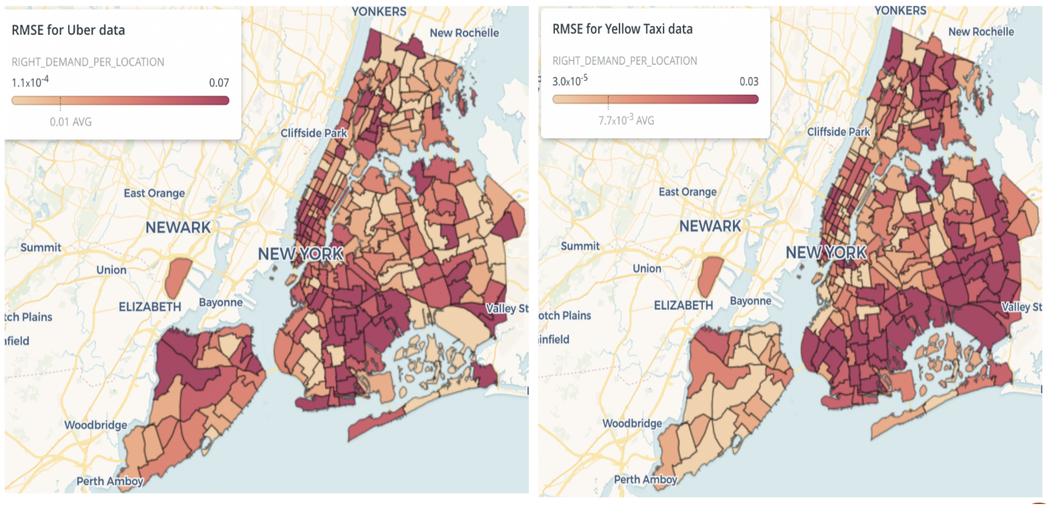


Figure 18: RMSE for each pickup zone

seq2seq framework with LSTM boosts the performance significantly. The reason is that learning the latent representations allows for more accurate data representation. Finally, the adoption of an attention mechanism in the modified LAS, yields significant gains in accuracy, because the implemented attention better exploits long-term associations between inputs and outputs of the dataset.

Our MSP-TCN model brings additional enhancement to the table along two significant dimensions: (1) the two-stage framework leverages transfer learning of the underlying latent representations and robustly incorporates exogenous factors, and (2) The TCN encoder leverages local connectivity and parameter-sharing to achieve more efficient and stable learning of the first-stage. This allows the MSP-TCN to outperform all the other models along all three performance metrics and on both datasets. Figure 17 graphically depicts the performance of each method relative to the best, where the best method is scaled to a value of 1.00 for each metric. This figure shows that our MSP-TCN outperforms all five other methods along all three metrics and for both test datasets. It is obvious that the first four models are clearly inferior, and that the modified LAS approach is the only one that is competitive with our model. However, as we discuss a little later in this section, the MSP-TCN model is far superior from a computational perspective.

We next examine the effect of incorporating pick-up/drop-off interactions and exogenous factors into our model. For the Uber dataset, Table 2 indicates that including pick-up/drop-off interactions results in an approximate 5% drop in the RMSE. The sMAPE and sL1 are also reduced by around 10% and 12% respectively. Incorporating both pick-up/drop-off and exogenous factors yields even better results, with reductions of $7\%, 17\%$ and 24% in the RMSE, sMAPE and sL1, respectively. The corresponding reductions with the Yellow Taxi dataset are even better. These results clearly display the value of including drop-off information and exogenous factors when making forecasts.

Figure 18 demonstrates the prediction performance of MSP-TCN in terms of the RMSE, across all areas of New York for both datasets. While our model outperforms the other models considered, its accuracy varies slightly from one area to another and Figure 18 shows how accurate our model is for any given area (the lighter the better). From a practical standpoint, such information can be leveraged for better resource allocation. For example,

Table 2: Incorporation of Additional Information to the Region-based Travel Demand Prediction Model

| Dataset | Model | RMSE | sMAPE | sL1 |
|---|---|---|---|---|
| Uber | Pickup Only | 0.1043 | 0.0886 | 0.0072 |
| | Pickup+Drop off | 0.0994 | 0.0793 | 0.0063 |
| | **Pickup+Drop off+Exogenous** | **0.0971** | **0.0793** | **0.0055** |
| Taxi | Pickup Only | 0.0498 | 0.0376 | 0.0016 |
| | Pickup+Drop off | 0.0412 | 0.0315 | 0.0013 |
| | **Pickup+Drop off+Exogenous** | **0.0396** | **0.0276** | **0.0011** |

from the figure, it appears that the model yields somewhat less accurate forecasts around the JFK airport area. Such an observation can be used by the taxi company to increase fleet capacity assigned to that area to hedge against this reduced accuracy.

Next, we examine the uncertainty associated with our model in order to provide some measure of confidence in our forecasts. We limit our comparisons to one-step-ahead forecasts and only with the modified LAS, since this is the only method that could be considered as being competitive with our MSP-TCN method in terms of performance. Table 3 compares the coverage probability (CP) of these two methods and captures the uncertainty surrounding forecasts and the errors associated with them, at multiple confidence levels, and in an efficient and actionable manner. From a decision-making standpoint, an estimate of the percentage of times that our forecasts fall within predetermined confidence levels is more valuable than knowing the characteristics of the error distribution, or the variance associated with the forecasts produced by the model. With both datasets, the CP of MSP-TCN is higher than that of the modified LAS by 1% to 2% at the 95% confidence level, and by 2% to 3% at the 90% confidence levels. Obviously, the CP of both models increases significantly as we decrease the confidence level (because the intervals are now wider). The advantage of MSP-TSN is somewhat more pronounced as well when this happens. In addition, both models

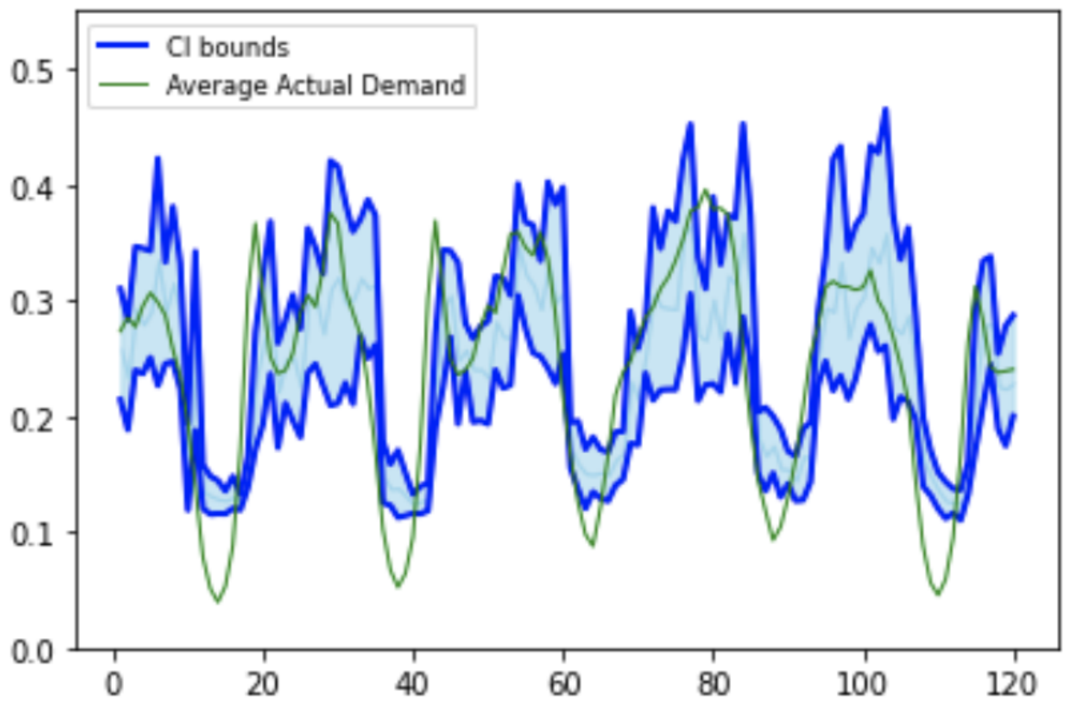


Figure 19: 95% confidence interval of the hourly average of the actual demand for a five days period

display higher CP for the Uber dataset. Our analysis suggests that the higher variation and smaller size of the Yellow Taxi dataset account for this.

Table 3: Coverage Probability of the Region-based Travel Demand Prediction Model

| Dataset | Model | CP at 95% | CP at 90% |
|---|---|---|---|
| Uber | Modified LAS | 0.80 | 0.88 |
| | **MSP-TCN** | **0.81** | **0.90** |
| Yellow Taxi | Modified LAS | 0.77 | 0.86 |
| | **MSP-TCN** | **0.79** | **0.89** |

Figure 19 corresponds to the Uber dataset and provides an illustration of our uncertainty estimates. It shows the hourly average of the actual demand across all pickup zones of NYC for each of the 120 one-hour time steps over a five-day period that we selected at random. The lower and upper bands around this plot correspond to the average (across all locations) of the lower and upper bounds, respectively, of the individual 95% confidence intervals of the forecasts for each time step. It is clear from the figure that on average, our model's confidence bounds miss only the bottoms of the actual demand curve. From a practical standpoint, this is significant because our model is conservative here and its forecasts ensure that the demand is always met. A more serious problem would be when demand exceeds our upper confidence limit on the forecast,but as the figure shows, this happens quite infrequently.

We end this section with some observations related to computational efficiency. This is an essential contribution of our work and the results related to this are shown in Figure 20. The comparison shows the number of learnable parameters with MSP-TCN as compared to the only other competitive model among the others, i.e., the modified LAS approach. For Stage 1, the modified LAS has more than 14 million parameters, while MSP-TCN has less than 3 million parameters, which reflects a very significant reduction.

The implementation of MSP-TCN without transfer learning requires training 1.3 million parameters, which is under 10% of the corresponding number with the modified LAS. Transfer learning allows us to reduce this figure further to only about 0.5 million parameters. In this case, for Stage 2, our model does not have a computational advantage over the modified

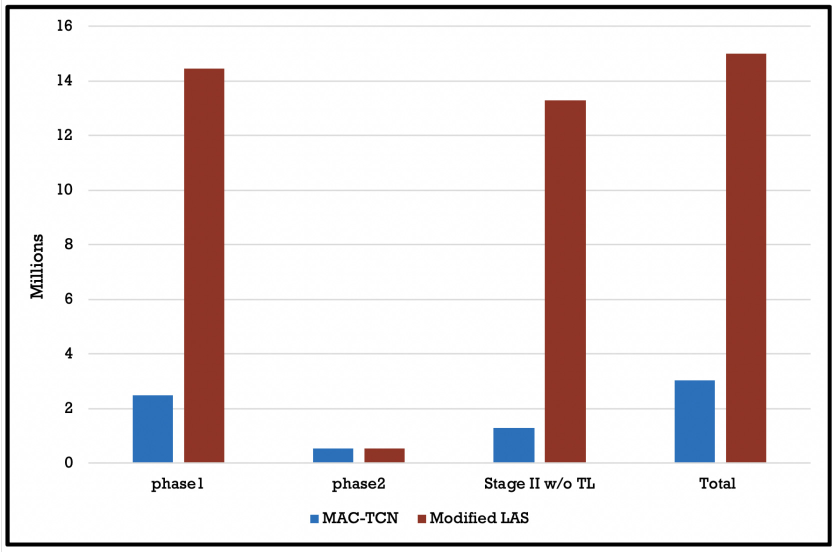


Figure 20: Number of learnable parameters for TCN and modified LAS

LAS since the network weights to fine tune are the same for both models. However, by looking at the total number of learnable parameters with transfer learning, MSP-TCN is more efficient, with a reduction of 79% in this number.

## 3.6 Summary & Conclusions

In this chapter we develop an approach that overcomes several of the limitations of existing methods for forecasting travel demand, and we demonstrate its effectiveness and superior performance compared to other advanced machine learning models through numerical comparisons using two real-world datasets (Uber and Yellow Taxi). In particular, our method is based on a novel multi-stage, deep, probabilistic model. It accounts for spatiotemporal interactions and incorporates exogenous factors to develop multi-step ahead forecasts of demand at a micro-geographical level. In addition to addressing the technical challenges associated with our goals it also provides uncertainty estimates for the predictions. We design a two-stage model that takes historical pick-up and drop-off sequences and predicts the demand for multiple steps ahead in the future. In the first stage, we leverage the encoder-decoder framework to map the input to a latent representation. The encoder structure consists of a temporal convolution network (TCN), while the decoder is a two-stacked LSTM component. An attention mechanism is used, and teacher-forcing helps the model recover when deviation occurs in the early steps of prediction. In the second stage, we perform transfer learning from the first stage and incorporate exogenous information; our tests show that this significantly improves the model's accuracy. Monte Carlo dropout is used to design a probabilistic setting that enables us to quantify uncertainty in the predictions. This can be achieved by running the model multiple times for each time step to obtain confidence intervals instead of single forecasts. Our experiments show that our model outperforms other advanced models.

Unlike other models, ours provides distributions of accurate demand forecasts for the entire city, and for multiple steps ahead. This specific form of output helps with planning dispatch operations more efficiently. A second advantage that our model has when compared to the baseline methods described, is its superior computational performance with a significant reduction in the number of learnable parameters. This is obtained by leveraging the TCN structure in the model's architecture. In terms of possible limitations, our model assumes access to external data beside the demand drop-off history. Also, our model does not capture the flow of demand which can be very valuable for shared-ride products. In other words, our model predicts where demand originates but doesn't specify the destination. Joint origin-destination (O-D) demand prediction can further empower better supply-demand matching with multiple passengers possibly sharing the same supply along the same route. This O-D prediction will be the focus of the next chapter.

This work can be further extended to exploit more information such as specific spatial characteristics. A second promising direction is to extend this work and study the interactions between supply and demand. The model can be deployed in a more comprehensive setting to develop a supply-demand matching algorithm and a more reliable dispatching decision support system. With autonomous vehicles that have the potential to revolutionize transportation mobility, such systems will become even more important. Finally, with necessary modifications to reflect the actual system dynamics, we could also explore the application of our model to other domains with time series characteristics, such as finance.

# 4.0 Travel Demand Flow Prediction

## 4.1 Introduction

This chapter continues the discussion on travel demand prediction with a shift of focus from region based demand to origin-destination demand. Despite the fact that demand prediction at the region level can empower fleet dispatch systems, incorporating trip destination forecasting in the modeling task can lead to better quality supply-demand matches. The origin-destination (O-D) level forecasting captures the flow between different geographical regions, which is especially important for ride-sharing products (e.g. UberPool, Lyft ride-share). Ride-sharing products are key to offering inexpensive ride options and gaining more market share while saving a lot of supply hours. The Uber X4Less product is an example that illustrates the importance that car-sharing companies place on expanding this segment of products. In this product, a customer requesting a solo trip receives an offer to join a shared ride at a discounted price with the guarantee that it will be a last-in first-out (LIFO) trip.

However, these products have more operational challenges than solo ride products because the pricing model and dispatch system associated with them have more factors to take into account. With limited supply, a company needs to decide on the competitive price to attract users to switch to the shared ride option while also keeping sufficient supply resources available for future requests. Moreover, supply-demand matching becomes more complex due to the fact that route optimization and utilization goals need to be achieved for multiple pick-ups and drop-offs. For example, for a ride-share supply that includes an active trip (i.e. one with a rider in the car) originating in region A with region B being the destination, it is more efficient to have additional matches (i.e. riders added to the trip) on the same general route. However, additional matches that significantly deviate from the route would cause damage to the supply network as a whole because it will require additional supply time while also leading to a more unpleasant user experience. It should also be noted that assigning available supply to awaiting requests is done sequentially based on short session windows, with the possibility of re-matching within a short period of time. Thus, knowing in advance the predicted demand flow between two regions can reduce the uncertainty associated with trip requests on the same route received in the next few minutes. In short, the matching algorithm needs to be empowered with demand *flow* forecasting, which can improve the quality of matches, reduce service time, and improve user experience.

This work is motivated by the fact that prior work has largely focused on deterministic next step demand prediction at the region level [30, 96, 44]. However, only a few studies have discussed passenger demand flow prediction [48, 103, 96, 39]. The latter approach, where the focus is shifted from pick-up zones to origin-destination pairs, is more challenging for a number of reasons: 1) spatiotemporal correlations between flow pairs are complex, dynamic and bi-directional, 2) the demand flow matrix associated with short time intervals is usually sparse, and 3) computations related to demand flow are more expensive [97]. To illustrate the dynamic bidirectional spatiotemporal interactions, consider demand flows from a residential region to two different commercial zones. Thee flows will tend to be positively correlated during morning rush-hour because flows from residential zones to commercial ones can be expected to be high during morning peak-time. It is reasonable to see similar demand flows but in the opposite direction during evening peak-hours as people return to their homes. These relationships between O-D pairs are important to capture and incorporate within the modeling. Furthermore, we believe that for a reliable and responsive system, it is imperative to have real-time, multi-step-ahead flow forecasts for the entire city network. Many aspects of the platform can benefit from long-term forecasting by making decisions to reduce expenses, allocate resources, take advantage of trends and avoid surprises. We also need to quantify our confidence about the model's output through an accurate estimation of its uncertainty in order to make better-informed decisions.

In this study, we investigate the problem of long-term demand flow prediction in a network, and address the challenges mentioned above. We propose a deep learning framework with a novel architecture capable of producing reliable demand flow networks for multiple steps ahead along with uncertainty estimation. First, we construct dynamic demand flow graphs to represent the ride-hailing demand data across the entire city. Then, we encode a time-series sequence of historical demand flow graphs into a latent representation. This representation is then decoded to produce demand flow networks for multiple steps ahead. More precisely, we pass the historical sequence of demand flow networks through special graph convolution layers to extract the hidden features of topological and spatial demand flow. Then, LSTM layers are used to capture the temporal interactions. The output produced by the encoder is exploited by the decoder to generate demand flow networks for future time intervals. For uncertainty estimation, we leverage Monte Carlo (MC) dropout layers and keep them activated during inference, similar to what was done in the previous chapter.

The main contributions of this study can be summarized as follows:

- From a *task* perspective: we forecast travel demand network distributions instead of deterministic region-level forecasting.
- From a *methodological* perspective: we propose a novel end-to-end stochastic encoder-decoder architecture for multi-step ahead forecasting. Our architecture understands the graphical representation of demand flow by using GNN, which helps in looking at the data as interrelated entities instead of isolated data points. The proposed architecture can produce demand network distributions without major changes to the encoder-decoder design by using MC dropout layers.
- From an *experimental* perspective: we evaluate our model on both taxi and ride-hailing services, both before and after the onset of COVID-19; we show that the proposed framework outperforms baselines.

The reminder of this chapter is organized as follows. Section 4.2 discusses related work. Section 4.3 provides the problem definition and details of our proposed framework. Section 4.4 covers the experimental study, including the settings and results. Finally, in Section 4.5, we provide a summary and conclusions.

## 4.2 Related Work

Recently, the deep learning research community has realized the importance of incorporating both origin and destination in the transportation demand prediction task. A number of deterministic regression models aiming at O-D travel demand prediction using graphical neural networks have been proposed [48, 103, 96, 39]. However, the research done in this area is still limited and can be further expanded to improve the ride-hailing user experience.

### 4.2.1 Graphical Neural Network

Deep learning has demonstrated its ability in capturing patterns of many types of data (e.g. images, text, and videos). This is achieved by applying linear algebra operations on fixed-size matrices. The classical deep learning toolbox (e.g. CNN, RNN, VAE) was actually designed for Euclidean type of data (linear sequences, fixed-size grid). However, this toolbox is not applicable to data of arbitrary size from non-Euclidean domains represented as graphs. The arbitrary size and complex topological structure of graphs make the task of capturing patterns more complex. Graphs lack the spatial locality (i.e. regular geometry/ fixed size of neighborhood set) that exists in fixed-size grid linear sequences. In addition, they are order invariant which means there is no reference point (left, right, up, down) or fixed ordering (Figure 21). The arbitrary size of graphs makes it very challenging to build models that can handle this inherited feature. This is mainly due to the fact that most of the existing architectures require fixed-size input on which to to apply mathematical operations. For example, CNNs require fixed size images (i.e. number of pixels) in order to be able to apply filters and aggregate the result at each layer and pass it through to the next layer. As discussed in Chapter 2, the hyper-parameters of CNN (e.g. stride and padding) and the sizes of the hidden layers are determined based on the input size. Thus, when the fixed-size input condition is violated, it is not feasible to perform the mathematical operations for both feed-forward and back-propagation algorithms and to iterate through stochastic gradient descent for obtaining a local minimum. Similarly, RNNs require knowledge of the previous word (left) and next word (right) to understand the temporal dependencies within a text sequence and to perform back-propagation through time.

Incorporating edge information for a graph is another challenge within the modeling. In some cases the topology of a graph changes dramatically over time. Consider a social network example, where nodes represent users and edges show the connection between users: if two

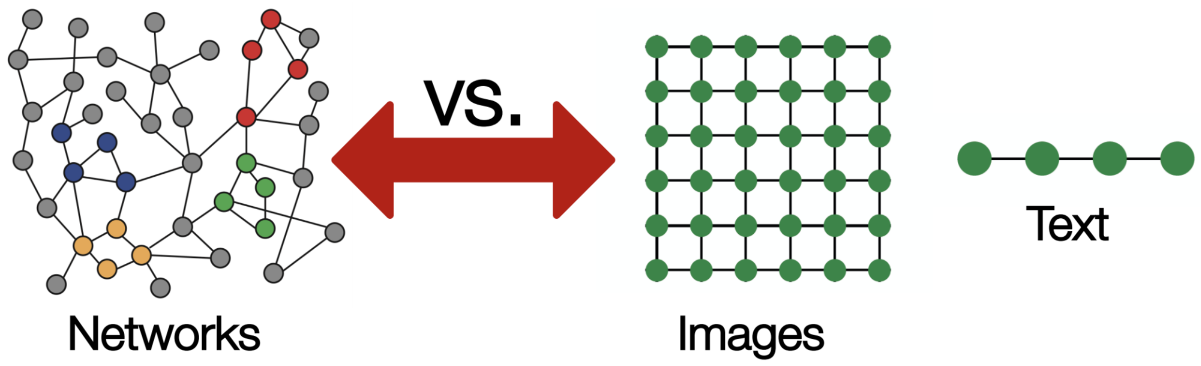


Figure 21: Networks have arbitrary structure while images & text have a reference point [1]

users (i.e. nodes in the graph) “unfriend” each other at a specific point in time, the edge will disappear, which in turn will affect the entire structure of the graph. In fact, even generating the graph is a major challenge to start with. Therefore, graphs, which we find naturally in many application domains surrounding us, require more specialized algorithms to handle their complex structure and detect patterns that exist in their dynamic environments.

The structure of a Convolution Neural Networks (CNN) can be used to empower machines and help them perform tasks like image classification, image recognition, or object detection. The way CNN works on images is by first transforming the image into a grid of pixels, then sliding a convolutional operator window across the two-dimensional grid. An image can be viewed as a ‘grid graph’ where each node corresponds to a pixel, and adjacent to its four neighbors (Figure 22) . The computation performed by the sliding convolutional operator is twofold: 1) collect information “messages” from the center node as well as nodes in the scanned part of the image (i.e. neighbors), and 2) aggregate the collected messages. The main idea in our GNN is to consider graphs as a strict generalization of images and apply the convolution concept.

Graph Neural Networks (GNNs) are neural networks that can be directly applied to graphs, and perform prediction at one of the following levels (Figure 23):

- Node-level: Node classification is performed by mapping each entity to an embedding

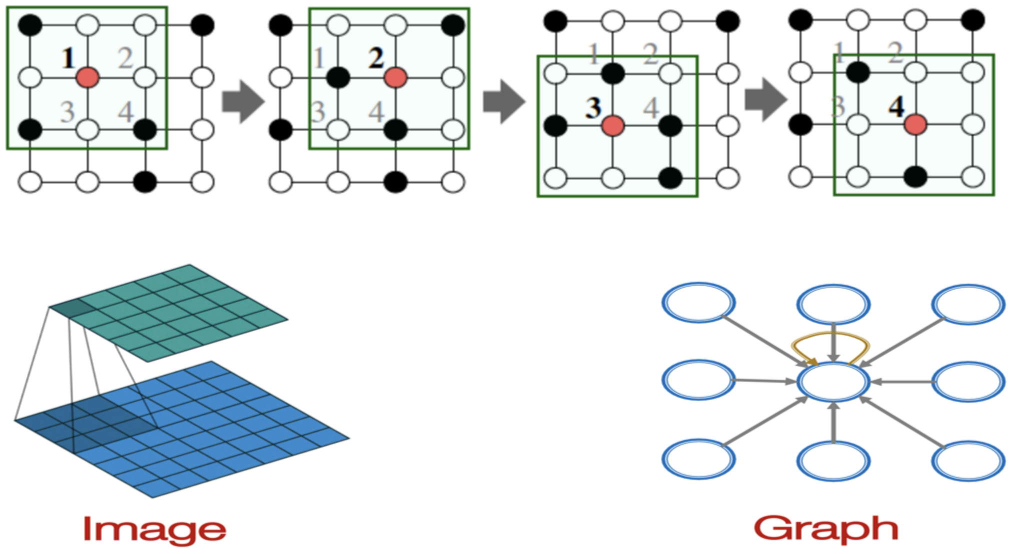


Figure 22: Illustration of viewing an image as a graph [2]

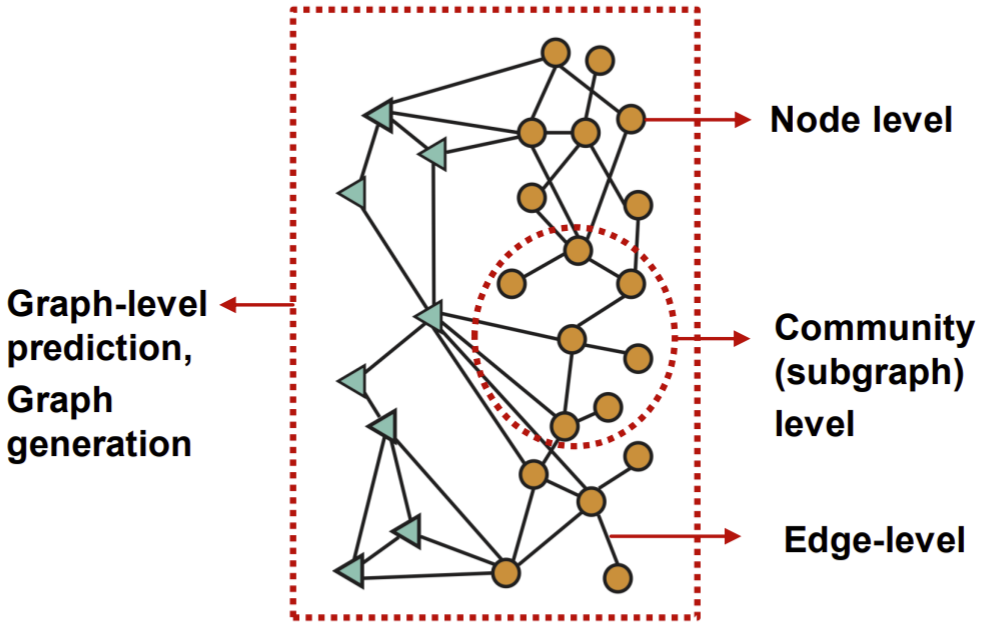

Node level
Graph-level prediction, Graph generation
Community (subgraph) level
Edge-level

Figure 23: GNN scope of task illustration [3]

and then classifying a node based on the similarity of its embedding to the embedding of other nodes. Problems of this type are mostly trained in a semi-supervised way, with only a portion of the graph being labeled. A common example of this is categorizing online users in e-commerce. In this scenario, user-item graphs are generated where a node could be an item or a user and an edge between a user and an item represents the quantity of purchased units of the item by the user. Understanding these graphs can help in customer segmentation, which is important for targeted marketing and personalized ads.

- Edge-level: In this case, the aim is to predict the existence or strength of a relationship between several entities in a graph. This can for example be used in recommendation systems for social networks to identify potential connections. For example, Facebook uses this type of modeling to understand connection within the network and provide friend suggestions to grow its user-base and improve engagement with the platform. The nodes in this network are users and the edges represent a binary relationship (i.e. if two users are friends, there is an edge connecting them). The friend suggestion could be 2-hop neighbor (a friend of a friend).
- Community-level:The goal of this type of task is to classify a sub-graph in a vast network, e.g., fraud detection in ride-hailing or financial services networks. Ride-sharing companies use network analysis to detect fraud. For example, consider collusion behavior, which basically means cooperative fraud among users. Here users (i.e. nodes) collude by taking fake trips with stolen credit cards resulting in charge back (a bank-initiated refund for a credit card purchase). Edges would represent connections between users including both drivers and riders. GNNs can be used to detect these communities.
- Graph-level: Graph-level tasks include graph classification and graph representation learning. The task of graph classification is to label the entire graph into different classes. It is similar to image classification, but the target changes into the graph domain. Graph classification is widely used and range from determining whether a protein is an enzyme or not in bioinformatics, to categorizing on-demand streaming content reviews and classifying documents in NLP. The learning of travel demand network embedding discussed in this chapter can be viewed as an example of graph representation.

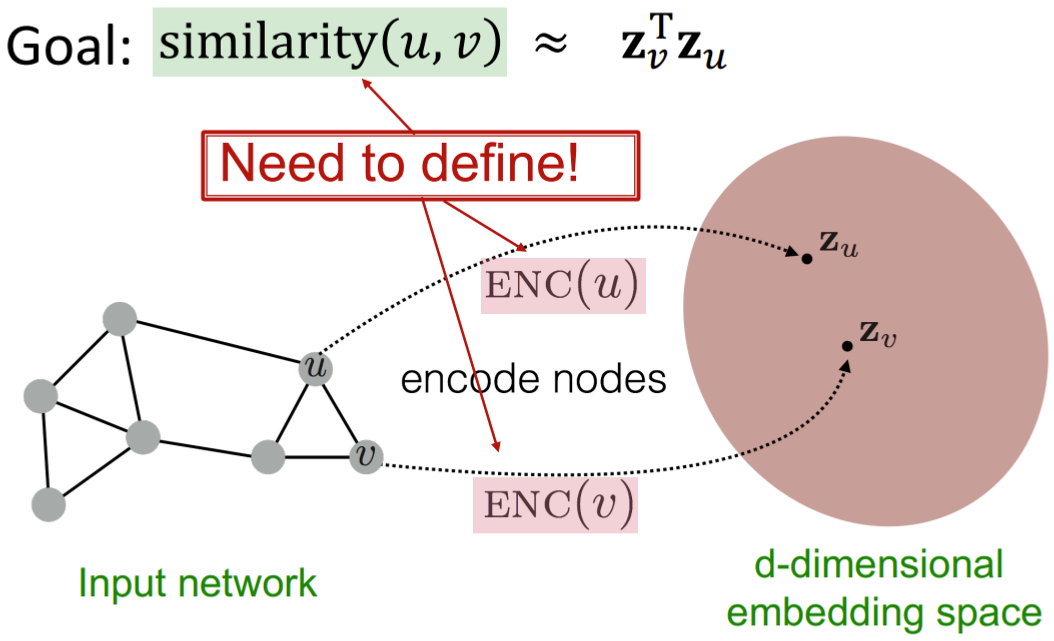

Goal: similarity$(u, v)$ $\approx$ $\mathbf{z}_v^{\mathrm{T}}\mathbf{z}_u$
Need to define!
ENC$(u)$
ENC$(v)$
encode nodes
$\mathbf{z}_u$
$\mathbf{z}_v$
$u$
$v$
Input network
d-dimensional embedding space

Figure 24: Node embedding [3]

Next, we will briefly discuss how neural networks can work on graphs. In graph theory, we use the concept of a Node embedding (Figure 24). In other words, we map a node's feature vector to a lower dimensional space rather than the actual dimension of the node's feature vector, so similar nodes in the graph are embedded close to each other in the latent space. The mapping is done by using the encoder-decoder framework and then applying the typical deep learning training algorithm. The training is performed to optimize a defined loss function measuring the distance between the feature vectors of the nodes after they are mapped to the latent space (e.g. cosine similarity function).

The encoder function needs to find the locality in the network and then aggregate information from the node itself and its neighbors. The locality information is obtained through the use of computational graphs, as shown in Figure 25. Increasing the depth of the computational graph to 2 layers will provide access to the information about the 2-hop neighbors. Clearly, as we increase the number of layers, we expand access to other connected nodes in the network (i.e. $n$ layers will access and aggregate information about all nodes which

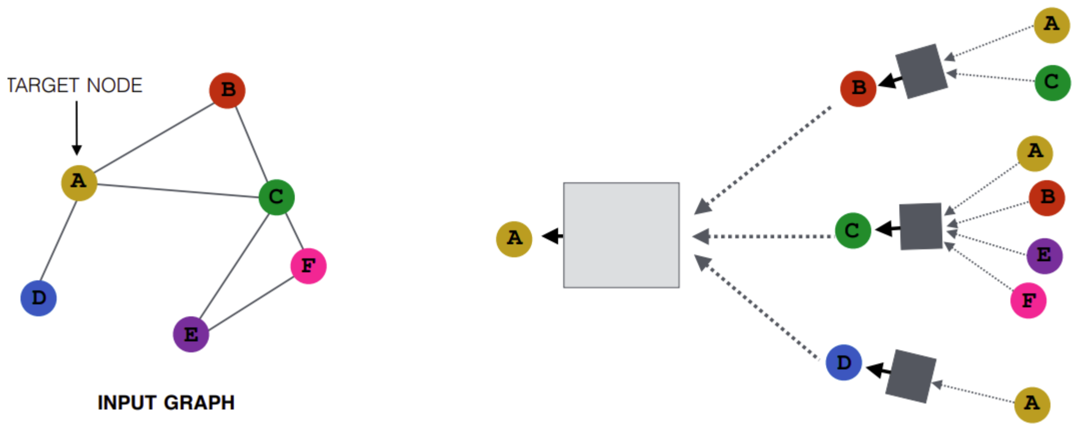


Figure 25: Computational Graph [3]

are $n$-steps away from the central node of interest). The aggregation function must be permutation-invariant (e.g. sum, average, and max) in order to use a neural network for the process of learning the embedding.

At the graph-level, the same process can be applied to obtain network embedding (Figure 26). Graph Auto Encoder, is an example of network embedding. The goal is to map the network to a latent representation, from which we can reconstruct the original network. First, the encoder exploits the feature vectors of the nodes in the network along with the corresponding adjacency matrix, which captures the edge level information, to understand the topological characteristics of a graph. Then the decoder uses the network embedding to generate the original adjacency matrix. The loss function to optimize for during training is the reconstruction error.

With respect to the theoretical work that has accelerated the implementation of GNN, Kipf and Welling [43] have introduced Graphical Convolution Network (GCN) which uses a layer-wise update rule that is based on first-order approximations of spectral convolutions on networks. The main contribution of this work is the symmetric normalization of the adjacency matrix by multiplying it by the inverse square root of the degree matrix on both sides. This approach is very scalable and has the ability to learn how to encode both

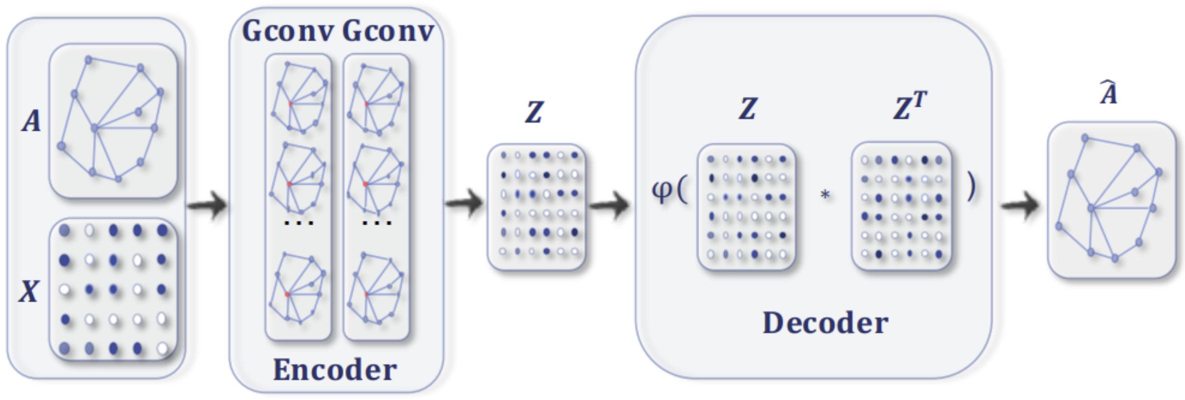


Figure 26: Graph Autoencoder [88]

graph structure and node attributes. An extension of this work has focused on edge-wise mechanisms where messages (i.e. arbitrary vectors) are computed along graph edges before using a permutation-invariant aggregation function [27, 82].

## 4.3 Proposed Framework

In this section we introduce the notation used in this study, and then describe the details of our framework.

### 4.3.1 Notation and Problem Definition

First, the city to be studied is split into $N$ micro-geographical zones, and the aggregate demand flow per time unit (e.g., 1 hour) is obtained for each O-D pair. This pre-processing step converts the data into a time-sequence of O-D graphs. Demand flow will be presented as a graph $G(V, E, A)$ where the set of vertices $V$ represents the disjoint geographical zones (nodes; $|V| = N$), $E$ denotes the set of edges, and $A \in \mathbb{R}^{N\times N}$ represents the adjacency matrix with entries corresponding to the demand flow between O-D pairs. The demand flow between the O-D pair is captured by the edge weight. In other words, the weight of an edge going from node $i$ to node $j$ in the network representation is equivalent to the adjacency matrix entry ($a_{ij}$). Looking at the past $T$ time steps, we define a tensor

$$\mathcal{G} = \{G_1, G_2, ....., G_T\} \in \mathbb{R}^{T \times N \times N} \tag{4.1}$$

which represents the historical demand flow networks over those $T$ steps. This tensor will be the input to our model with an objective of predicting demand flow for multiple steps ahead

$$\hat{\mathcal{G}} = \{\hat{G}_{T+1}, \hat{G}_{T+2}, ....., \hat{G}_{T+F}\} \tag{4.2}$$

where $F$ is a hyper-parameter denoting the length of the prediction horizon. Also, let the vector $\mathbf{Q}_t$ represent the exogenous factors at time $t$; this will be discussed in more detail in the experimental study section.

### 4.3.2 Model

We now introduce our deep learning based model: Probabilistic Encoder-Decoder Graph Neural Network (PED-GNN) (Algorithm 3). Our model exploits the encoder-decoder structure, which is commonly used for sequence-to-sequence prediction. In essence, the encoder reads the historical demand flow sequence and maps the corresponding sequence of adjacency matrices to a fixed size vector in some latent space. This is done by passing them through multiple GNN and LSTM layers. This embedding is then fed to the decoder to generate demand flow networks at future time steps. Our approach is based on the conjecture that an appropriate latent representation for the historical travel demand flow can result in more accurate forecasts. The architecture of our model is illustrated in Figure 27.

The input sequence to the encoder is a historical time-series of demand flow networks. Our encoder leverages graph convolution network (GCN) layers to understand the topological characteristics of the citywide demand flow. The encoder receives the demand flow time-series $\mathcal{G}$ and passes each graph through two stacked GCN layers. The deployed graph convolutional operator is

$$\mathbf{X}' = \hat{\mathbf{D}}^{-1/2}\hat{\mathbf{A}}\hat{\mathbf{D}}^{-1/2}\mathbf{X}\mathbf{W} \tag{4.3}$$

where $\hat{\mathbf{A}} = \mathbf{A} + \mathbf{I}$ denotes the adjacency matrix with inserted self-loops to account for the possibility of having demand flow within the same zone (i.e. node on our graph). $\hat{\mathbf{D}}$ denotes the degree matrix of the graph with self-loops (a diagonal matrix with entries equal to the number of edges incident to the corresponding node). $\mathbf{X}$ represents the node feature matrix (a matrix containing the feature vectors for all nodes), which we initialize with the identity matrix $\mathbf{I}$. $\mathbf{W}$ denotes the weight matrix (a matrix that represents the connections between two layers in the neural network) of the GCN. The node-wise formulation is given by:

$$\mathbf{x}_i^{'} = W \quad . \sum_{j \in \mathcal{N}(i) \cup \{i\}} \frac{a_{j,i}}{\sqrt{\hat{d}_j \hat{d}_i}} \ \mathbf{x}_i \tag{4.4}$$

wehere $\mathcal{N}(i)$ is the set of neighbors of node $i$.

The GCN layers are then followed by two stacked convolution layers to further understand the local characteristics of our input. Subsequently, an LSTM layer is applied to capture the temporal interactions in our sequence. This is done by passing the concatenated vector encapsulating the intermediate embedding produced by the convolution layers through a layer of LSTM cells. This yields the encoder output, which is a latent representation of the historical demand flow time-series.

For the decoder architecture, we use two stacked LSTM layers followed by a multi-layer perceptron (MLP). First, we initialize the decoder with the embedding of the ground-truth graph of the previous step with some predetermined probability $\alpha$, or the embedding of the most recent predicted graph with probability $(1 - \alpha)$. This teacher-forcing technique is the same as the one used in the previous chapter, and is used to stabilize the learning and yield more efficient convergence. It basically reduces the effect of inaccurate predictions on the subsequent forecasts over the prediction horizon. This is especially important because the focus of our model is to generate predictions for multiple steps into the future. For each time step in the prediction horizon, the teacher-forced graph needs to go through a graph embedding block identical to the one described in the encoder architecture. The graph embedding then passes through the first LSTM layer. For the most immediate time step in the prediction horizon, the latent representation produced by the encoder is used to initialize the hidden states. This is the point where the decoder receives and leverages

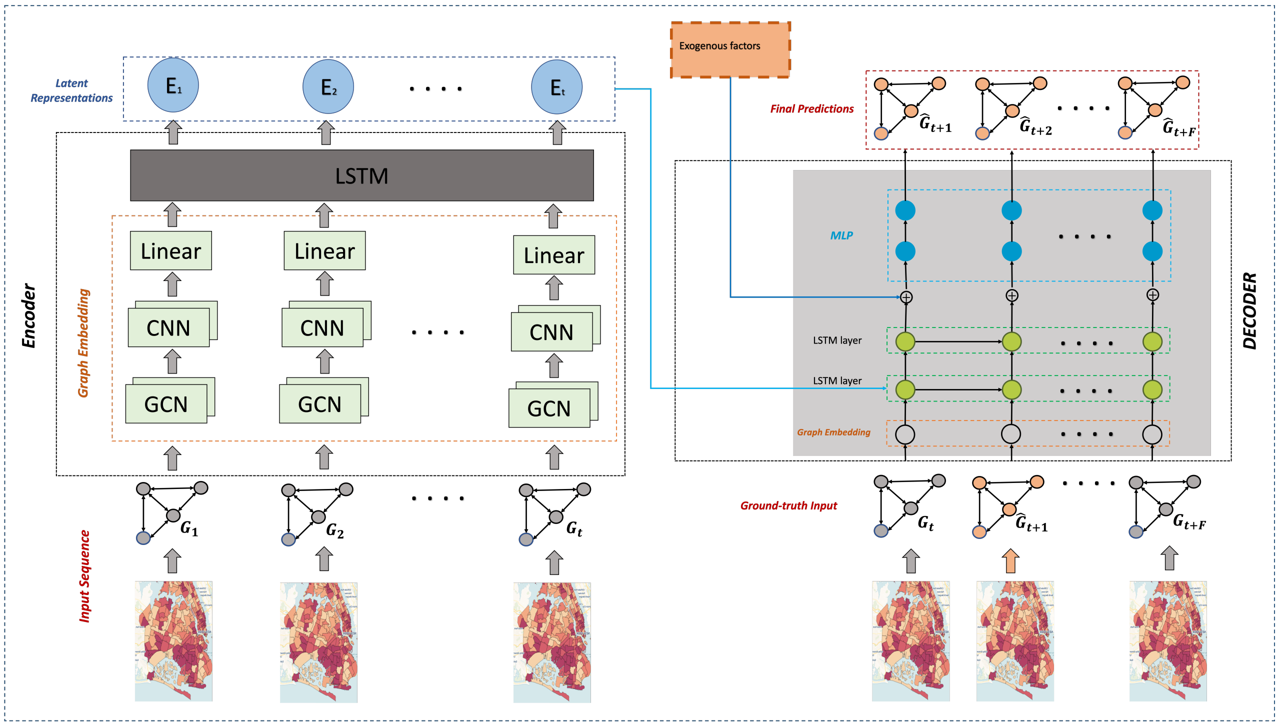


Figure 27: Illustration of our O-D encoder-decoder model

the learned mapping of historical demand flow to generate forecasts for future demand flow. Finally, to further enhance the learning process, we concatenate the output of the second LSTM layer with the exogenous factors vector and pass the concatenated vector through the final component of the decoder (MLP). The exogenous factors considered in this study are discussed in the following section.

## 4.4 Experimental Study

In this section we introduce the experiments conducted on the same two real-world datasets described in Chapter I to validate the efficiency of our model. We compare our model against four baselines and evaluate the performance of each in predicting travel demand flow while using three different evaluation metrics.

### 4.4.1 Datasets

We obtained Uber (largest ride-hailing service provider in NYC) and taxi trips data from the NYC Taxi and Limousine Commission (TLC). Both datasets have millions of records; each includes the trip pickup/drop-off locations as well as the time stamps of pick-up and drop-off events. The longitude and latitude can be mapped to administrative geographical zones pre-determined by TLC. We use the same geographical breakdown in our modeling for consistency purposes. A sample of the geographical breakdown for some neighborhoods in NYC is shown in Figure 28. The daily trips for ride-hailing apps and traditional Yellow Taxi is illustrated in Figure 29 . At first glance, there is a growing popularity of ride-hailing apps, while the demand for Yellow Taxis is declining over the years. In particular, 2017 appears to be when demand for ride-hailing services started to exceed the demand for Yellow Taxis, thus disrupting the transportation model that was dominant for decades. Another important observation is the disruption caused by COVID-19, which resulted in a steep sudden decline during early 2020. The same pattern along with the impact of COVID-19 are observed across major ride-hailing operators in NYC such as Uber & Lyft. The positive trend towards the

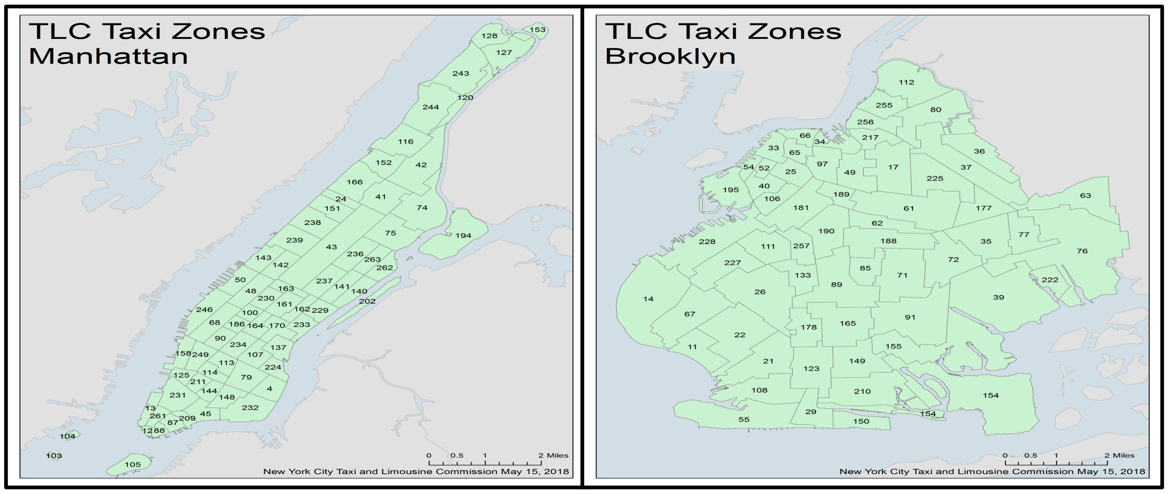


Figure 28: Geographical breakdown for Manhattan and Brooklyn areas[6]

end of 2020 through 2021 suggests an after-pandemic demand recovery. We test our model on travel flow both before the pandemic (Jul-Aug 2018), and after the pandemic had started to subside (April-May 2021) so as to evaluate the robustness of our model and avoid missing anything related to the impact of COVID-19.

With respect to exogenous factors, we incorporated a 6-dimensional vector containing (a) weather-related features (temperature, humidity, wind speed, and a categorical weather description feature such as rainy, sunny, etc.), and (b) time-related features (day of the week and hour of the day). With respect to the weather data, our data source was the official weather information for NYC from the National Oceanic and Atmospheric Administration (NOAA). The data was extracted and pre-processed to only include the items of interest listed above.

### 4.4.2 Experimental Setup

We performed data pre-processing on the raw trip records to convert the data into a proper format for graph-supervised learning. We used the *Networkx* package in *Python* to generate graphs, where the nodes represent the micro-geographical zones and the edge

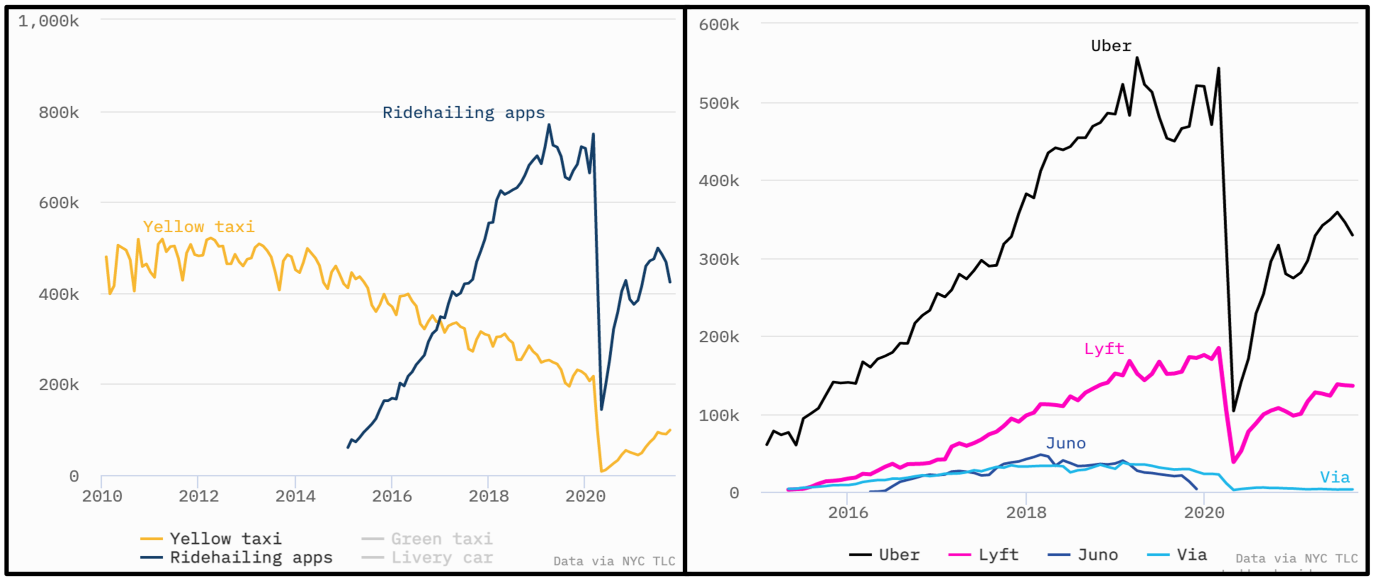


Figure 29: Trips per day in NYC [6]

weights represent the aggregated demand flow between the source (pick-up zone) and destination (drop-off zone).

We compare the performance of our model with that of each of the following five methods:

1. Multi-Layer Perceptron (MLP): Five fully connected linear layers with activation function (ReLU).
2. Vanilla LSTM (VLSTM): A single LSTM layer followed by an output layer.
3. seq2seq [31]: An encoder-decoder with 2 BLSTM layers.
4. LAS [13]: Listen, attend and spell (LAS) is a sequence model with attention mechanism. It uses an encoder-decoder framework where the encoder uses three pyramidal bidirectional LSTM (pBLSTM) layers, and the decoder uses two BLSTM layers.

All methods were implemented in *Python* using the *Adam* optimizer with a learning rate of 0.001, without momentum. We used *PyTorch* on *Google Colab Pro* for the neural network-based approaches.

We use 10 steps as the size of the sliding window for the historical demand flow networks, and we forecast future demand flow networks over three different forecast horizon lengths:

one-, three- and five-steps ahead. Additionally, we separately assess the effect of exogenous factors. In our analysis, we used the same set of evaluation metrics as the ones in the previous chapter: Root Mean Square Error ($RMSE$), Symmetric Mean Absolute Percentage Error ($sMAPE$), and Smooth L1 ($sL1$). The impact of outliers is better captured by the $RMSE$ and $sMAPE$, since $sL1$ (a.k.a. Huber loss) is less sensitive to anomalies. However, learning the degree to which outliers disrupt the demand flow network is not our focus here, and this is left as future work.

$$RMSE = \frac{1}{T} * \sqrt{\sum_{t=1}^{T} (\mathbf{G}_t - \hat{\mathbf{G}}_t)^2} \tag{4.5}$$

$$sMAPE = \frac{1}{T} * \sum_{t=1}^{T} \frac{|\mathbf{G}_t - \hat{\mathbf{G}}_t|}{(|\mathbf{G}_t| - |\hat{\mathbf{G}}_t|)/2 + 1} \tag{4.6}$$

$$smoothL1 = \frac{1}{T} * \sum_{t=1}^{T} \mathbf{z}_t \tag{4.7}$$

where $\mathbf{z}_t$ is given by:

$$\mathbf{z}_t = \begin{cases} 0.5 * (\mathbf{G}_t - \hat{\mathbf{G}}_t)^2, & \text{if} |\mathbf{G}_t - \hat{\mathbf{G}}_t| < 1 \\ |\mathbf{G}_t - \hat{\mathbf{G}}_t| - 0.5, & \text{otherwise} \end{cases} \tag{4.8}$$

We keep the MC dropout layers active during inference in order to enable uncertainty estimation (Algorithm 4). Since different neurons are being activated each time we run the model, different forecasts are generated for the same input. In our experiments, we run the model 100 times and obtain 100 sets of forecasts for each input in the test dataset. We then compute the mean and standard deviation of the forecasts for each step in the forecast horizon, and construct our prediction intervals. To assess the accuracy of our prediction intervals, we used Coverage Probability ($CP$), which is an empirical statistical measure that represents the proportion of times that our prediction intervals contain the ground truth being forecast. The significance levels considered in this study are 95% and 90%.

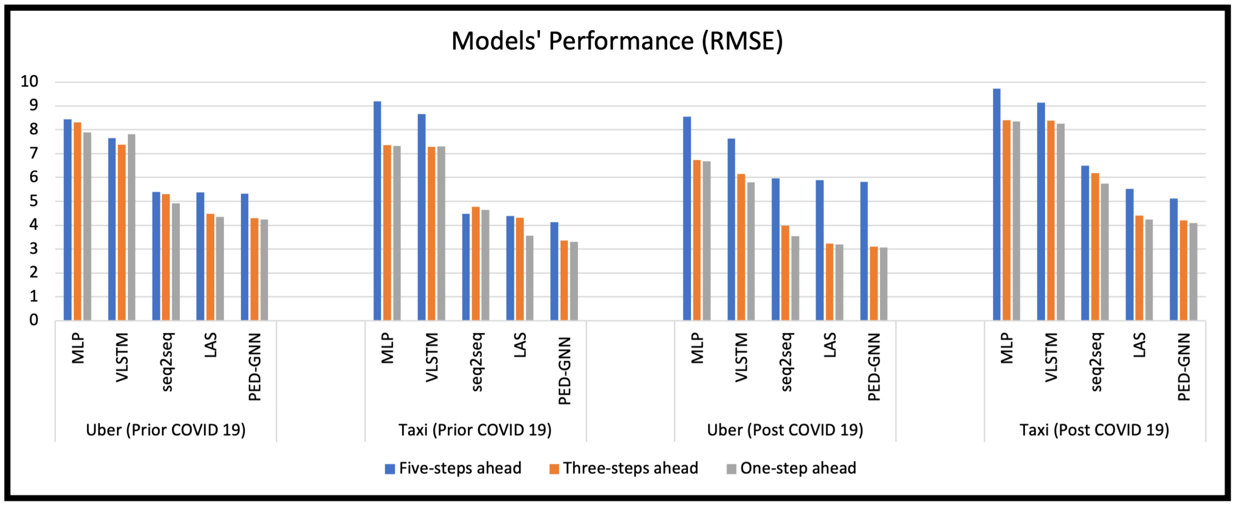


Figure 30: Comparison of the RMSE metric

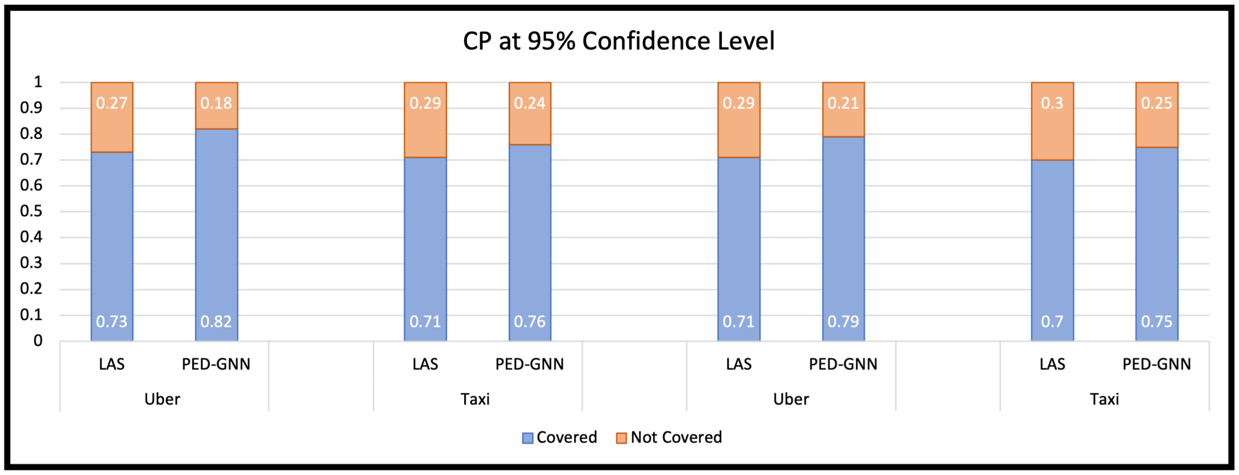


Figure 31: Coverage Probability at 95% Confidence Level

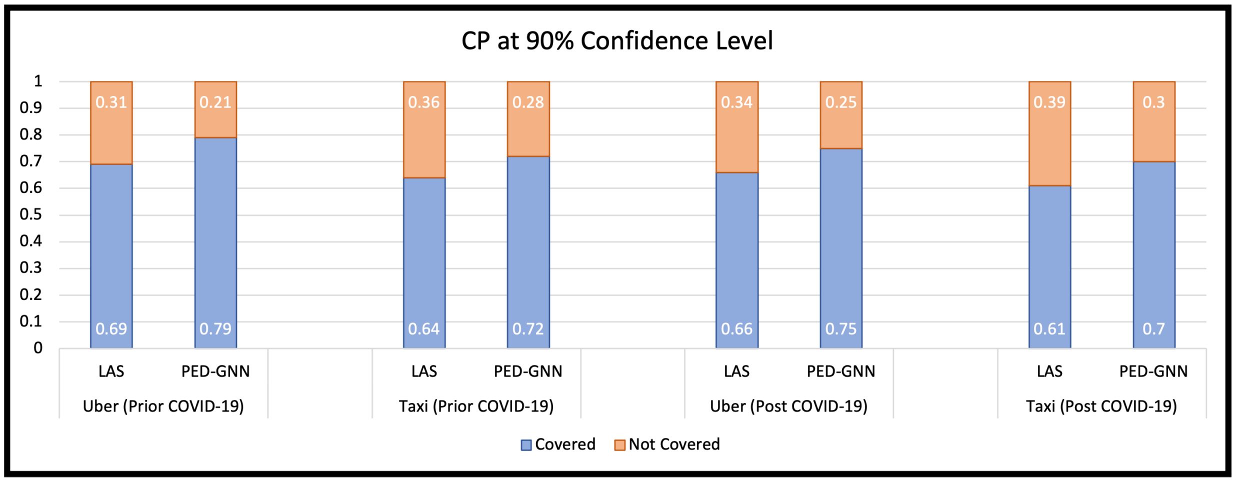


Figure 32: Coverage Probability at 90 % Confidence Level

Table 4: Performance of the Models

| Forecast window | | One-step | | | Three-steps | | | Five-steps | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Dataset | Model | $RMSE$ | $sMAPE$ | $sL1$ | $RMSE$ | $sMAPE$ | $sL1$ | $RMSE$ | $sMAPE$ | $sL1$ |
| Uber (Prior COVID-19) | MLP | 7.89 | 0.03 | 0.59 | 8.31 | 0.03 | 0.59 | 8.44 | 0.03 | 0.62 |
| | VLSTM | 7.82 | 0.03 | 0.59 | 7.37 | 0.02 | 0.58 | 7.65 | 0.03 | 0.61 |
| | seq2seq | 4.91 | 0.027 | 0.55 | 5.30 | 0.02 | 0.57 | 5.40 | 0.02 | 0.58 |
| | LAS | 4.35 | **0.01** | 0.53 | 4.47 | 0.013 | 0.57 | 5.37 | 0.016 | 0.58 |
| | **PED-GNN** | **4.24** | **0.01** | **0.52** | **4.29** | **0.01** | **0.55** | **5.32** | **0.015** | **0.57** |
| Taxi (Prior COVID-19) | MLP | 7.32 | 0.025 | 0.55 | 7.35 | 0.027 | 0.56 | 9.20 | 0.03 | 0.59 |
| | VLSTM | 7.31 | 0.025 | 0.54 | 7.28 | 0.026 | 0.55 | 8.66 | 0.03 | 0.59 |
| | seq2seq | 4.65 | 0.018 | 0.51 | 4.77 | 0.019 | 0.52 | 4.47 | 0.02 | 0.57 |
| | LAS | 3.56 | 0.009 | **0.48** | 4.32 | **0.01** | 0.52 | 4.38 | 0.02 | 0.56 |
| | **PED-GNN** | **3.31** | **0.007** | 0.49 | **3.36** | **0.01** | **0.51** | **4.13** | **0.018** | **0.55** |
| Uber (Post COVID-19) | MLP | 6.68 | 0.026 | 0.59 | 6.74 | 0.03 | 0.60 | 8.55 | 0.03 | 0.61 |
| | VLSTM | 5.79 | 0.022 | 0.59 | 6.14 | 0.027 | 0.60 | 7.63 | 0.029 | 0.59 |
| | seq2seq | 3.55 | 0.012 | 0.57 | 3.98 | 0.013 | 0.57 | 5.97 | 0.015 | 0.57 |
| | LAS | 3.20 | 0.008 | 0.54 | 3.23 | 0.009 | **0.54** | 5.89 | 0.01 | **0.54** |
| | **PED-GNN** | **3.07** | **0.006** | **0.53** | **3.11** | **0.006** | **0.54** | **5.81** | **0.009** | 0.55 |
| Taxi (Post COVID-19) | MLP | 8.34 | 0.025 | 0.56 | 8.41 | 0.03 | 0.57 | 9.73 | 0.03 | 0.61 |
| | VLSTM | 8.25 | 0.025 | 0.55 | 8.38 | 0.028 | 0.57 | 9.13 | 0.03 | 0.59 |
| | seq2seq | 5.75 | 0.013 | 0.54 | 6.18 | 0.015 | 0.55 | 6.49 | 0.019 | 0.57 |
| | LAS | 4.23 | 0.01 | **0.53** | 4.40 | **0.01** | **0.53** | 5.52 | 0.016 | 0.56 |
| | **PED-GNN** | **4.10** | **0.009** | **0.53** | **4.21** | **0.01** | **0.53** | **5.12** | **0.013** | **0.55** |

### 4.4.3 Results

Table 4 shows the performance of our model compared to the four baseline methods, evaluated over the entire dataset for both Uber and taxi services and for both prior to and after the COVID-19 pandemic. Looking at the simple MLP, the evaluation metrics all show weakness in learning the underlying spatiotemporal relationships in the data across all specified prediction horizons. Leveraging the LSTM unit, which is focused on understanding the long term temporal interactions of the time-series input, resulted in only a marginal improvement (the vanilla LSTM model). The relatively poor performance of these two models compared to the other baseline methods as well as our proposed model indicates that simple models struggle with understanding the sophisticated dynamics of travel demand flow. However, when we modify the LSTM model to have more depth by stacking multiple layers and deploy an encoder-decoder framework, the performance improves significantly. The observed improvement using the *seq2seq* model is mainly due to the learned representation of historical time-series within the encoder-decoder framework. Finally, the learning process was enhanced even further by adopting the attention mechanism in the LAS model, which decides what parts of the input sequence to focus on at each step in the prediction horizon. Also, the analysis unsurprisingly suggests an inverse relationship between the length of the prediction horizon and the performance of the model, i.e., as we look further into the future, the performance of the model degrades.

Coming to our model (PED-GNN), Table I shows that it is, in general, superior to to all baseline methods, with only the LAS model being competitive in a subset of the cases. This conclusion is consistent across all forecast windows and for all evaluation metrics, for both before and after COVID-19. The superior performance of PED-GNN can be attributed to two key factors: 1) the graph embedding component which captures the topological characteristics of the demand flow networks, and 2) the representation learning of historical demand flow, which is achieved by passing the O-D demand adjacency matrix through the encoder part of our model. Shifting the data representation from isolated origin and destination data points into networks encapsulating the topological relationships of demand flow, and incorporating that into the learning process leads to the superiority of PED-

Table 5: Incorporation of Additional Information

| Dataset | Model | RMSE | sMAPE | sL1 |
|---|---|---|---|---|
| Uber (Prior COVID-19) | DF Only | 4.59 | 0.012 | 0.53 |
| | **DF + Exogenous** | **4.42** | **0.01** | **0.52** |
| Taxi (Prior COVID-19) | DF Only | 3.41 | 0.008 | 0.51 |
| | **DF + Exogenous** | **3.31** | **0.007** | **0.49** |
| Uber (Post COVID-19) | DF Only | 3.25 | 0.008 | 0.53 |
| | **DF + Exogenous** | **3.07** | **0.006** | **0.53** |
| Taxi (Post COVID-19) | DF Only | 4.32 | 0.01 | 0.54 |
| | **DF + Exogenous** | **4.10** | **0.009** | **0.53** |

GNN. Furthermore, the construction of the decoder with stacked LSTM Cells applied to the graph embedding, along with teacher-forcing, also assist in generating more accurate forecasts for steps beyond the most immediate in the forecasting horizon. Figure 30 provides a visualization of the RMSE metric across all prediction horizon lengths for all models considered in this paper.

Next, we assess the value of incorporating data beyond the demand flow (DF) networks by performing an ablation test on the models with the one-step ahead forecasts. The focus here is on the importance of incorporating the exogenous factors discussed in Section 3. Table 5 illustrates the learning enhancement achieved by incorporating the external factors. More specifically, the RMSE of the model for the Uber dataset improves by 4% before and 6% after the pandemic. With respect to the taxi data, the RMSE score improves by 3% for the pre-COVID-19 dataset, and 5% for the post-COVID-19 dataset. The same general trend can be noticed for the other two evaluation metrics for all datasets included in this study. This analysis demonstrates the importance of incorporating exogenous factors when designing the model's architecture.

Table 6: Coverage Probability

| Dataset | Model | CP at 95% | CP at 90% |
|---|---|---|---|
| Uber (Prior COVID-19) | LAS | 0.73 | 0.69 |
| | **PED-GNN** | **0.82** | **0.79** |
| Yellow Taxi (Prior COVID-19 | LAS | 0.71 | 0.64 |
| | **PED-GNN** | **0.76** | **0.72** |
| Uber (Post COVID-19) | LAS | 0.71 | 0.66 |
| | **PED-GNN** | **0.79** | **0.75** |
| Yellow Taxi (Post COVID-19 | LAS | 0.70 | 0.61 |
| | **PED-GNN** | **0.75** | **0.70** |

We conclude our experimental study by evaluating the prediction intervals associated with our model. As with Chapter 3, We restrict our comparisons to the only competitive baseline model (LAS), and we limit ourselves to just the one-step-ahead prediction for this analysis. As shown in Table 6, our model (PED-GNN) is more powerful than the LAS model in capturing the ground truth with the prediction intervals that are constructed. The coverage probability ($CP$) of PED-GNN exceeds that of the LAS by 9%, 10%, 8%, and 9% for the Uber dataset pre- and post-pandemic at 95% and 90% confidence levels respectively (Figures 31, 32). The test on the taxi datasets demonstrates similar superiority of PED-GNN over LAS across all comparison levels.

## 4.5 Summary & Conclusions

In this chapter, we propose a probabilistic encoder-decoder graphical neural network (PED-GNN) for demand flow network prediction. The novelty of our approach comes from the fact that we are feeding the model with a time-series composed of demand networks instead of numeric vectors of fixed size. The challenging aspect of feeding this type of data to a ML model is the arbitrary size of demand networks, which can be a huge hurdle for running gradient decent and back-propagation algorithms to optimize the weights of the neural network. These algorithms were designed to handle input and output of fixed size, which is not the case with graphs. For this reason, in the design of our architecture, we make use of a graph convolution network to learn the topological structure of a demand flow network and to understand the interactions between micro geographical nodes of the network. The GNN units enable us to overcome this challenge since they are capable of reading graphs and mapping them to a fixed size embedding, as explained earlier. We also apply LSTM layers on the resulting graph embedding to capture the interactions in the temporal dimension. The choice of an encoder-decoder framework enables one to map the historical time-series composed of demand flow networks into a latent representation. This latent representation is then decoded to generate forecasts for multiple steps ahead in the forecast horizon, and the teacher-forcing policy is deployed to limit the impact of inaccurate predictions on subsequent forecasts in the prediction horizon.

We find that incorporating exogenous factors enhances the learning process, and improves the model performance. Monte Carlo (MC) dropout also allows us to construct prediction intervals, which quantify our confidence in the generated forecasts. Our experimental study demonstrates that our model outperforms other advanced learning models on two real-world datasets from both before and after the surge in the COVID-19 pandemic.

We believe the next step would be to expand this work and incorporate supply data to build reliable pricing models and more comprehensive matching algorithms. Another promising path is to explore using a variant of our model to capture the dynamics of other types of networks to build useful forecasting models for other application domains such as supply chains and cloud computing.

**Algorithm 3** Training of PED-GNN

```
Input: training data $\{\mathbf{G}_1, ...., \mathbf{G}_N\}$, $\{\mathbf{Q_1}, ..., \mathbf{Q_N}\}$, lookback hyper-parameter $k$, Forecast
    horizon length $F$
Output: PED-GNN model $\mathcal{M}(.)$
 1: Initialize the encoder $\zeta(.)$, and decoder $\vartheta(.)$
 2: while stopping criteria not met do
 3:   for $t =$ k, ... ,N do
 4:     $\mathbf{k}_t, \mathbf{v}_t = \zeta(G_{t-k}, ..., G_t)$ latent representation produced by encoder
 5:     Decoder:
 6:     for $\tau = t, ..., t + F$ do
 7:       $$G_\tau = \begin{cases} \hat{G}_\tau & \text{w.p. } \alpha \\ G_\tau & \text{otherwise} \qquad \text{teacher forcing case} \end{cases}$$
 8:       $\delta_\tau = \omega(G_\tau)$, where $\omega(.)$ is a graph embedding block
 9:       Perform the forward pass for each LSTM cell (LSTM outputs: o, h)
          $$\{d_\tau, h_\tau\} = \begin{cases} lstm(\delta_\tau, \mathbf{h}^l_{\tau-1}, \mathbf{o}_{\tau-1}), \text{ if } l = 1 \\ lstm(\mathbf{d}^{l-1}_\tau, \mathbf{h}^l_{\text{initialized}}, \mathbf{0}), \text{ if } \tau = 1 \\ lstm(\mathbf{d}^{l-1}_\tau, \mathbf{h}^l_{\tau-1}, \mathbf{o}_{\tau-1}), \text{ otherwise} \end{cases}$$
10:       $i_\tau = concatenate[\mathbf{o}_\tau, Q_\tau]$
11:       $\hat{G}_\tau = MLP(\{i\}_\tau)$
12:     end for
13:     perform backpropagation to update weights of $\zeta(.)$, and $\vartheta(.)$ to minimize the gap
        between predicted values and ground truth values $\mathcal{L}(\hat{\mathbf{G}}_t, \mathbf{G}_t)$
14:   end for
15: end while
```

**Algorithm 4** Inference of PED-GNN

```
Input: testing data $\{\mathbf{G}_t\}$; trained PED-GNN model $\mathcal{M}(.)$, exogenous factors vector $\mathbf{Q_t}$,
    dropout rate $r$, number of epochs $N$
Output: predicted values mean $\mu_{\mathbf{G}_t}$ and standard deviation $\xi_{\mathbf{G}_t}$
1: for $i = 1, \ldots, N$ do
2:     $\hat{\mathbf{G}}_t^i = Dropout(\mathcal{M}(\mathbf{G}_t, Q_t), r)$
3: end for
4: $\mu_{\mathbf{G}_t} = \frac{1}{N}\sum_{i=1}^{N} \hat{\mathbf{G}}_t^i$
5: $\xi^2_{\mathbf{G}_t} = \frac{1}{N}\sum_{i=1}^{N} (\hat{\mathbf{G}}_t^i - \mu_{\mathbf{G}_t})^2$
6: return $\mu_{\mathbf{G}_t}, \xi_{\mathbf{G}_t}$
```

# 5.0 A Novel Hybrid Deep Learning Model For Stock Price Forecasting

In this chapter we study the extension of our model framework and test its applicability to a different application domain, namely finance. The stock market is an essential component of any open economy and a major indicator for gauging the economic health of a country. Many factors affect the behavior of the stock market, including political & social conditions, the state of the global economy, and financial performance of enterprises. Generally speaking, in the finance community, investing strategies can be divided into three categories: fundamental, technical and quantitative. Investors who follow the fundamental strategy focus mainly on the performance of an individual company and its track record to value a stock, while technical analysis considers only historical time series data to discover patterns. The core of the quantitative strategy is to use mathematical and statistical modeling to map historical time-series, along with other external factors, to the future stock price. Forecasting stock prices is a challenging task in the finance world due to its complexity and the dynamics associated with stock prices. We propose an approach that is similar to technical analysis in that we use time-series data to make forecasts.

Most prior work in the area of time-series stock price forecasting has used classical models such as Auto-Regressive Moving Average (ARMA) [98], linear regression [12], and even simple moving averages [24, 55]. These models are based on parameterizing pre-defined equations to fit a mathematical model to a sequence of historical data. The major drawback of these models is that they lack the ability to capture the natural non-linear dynamics in the data. Also, these models cannot incorporate external factors (e.g., interactions between different companies) since they only consider univariate time-series. This limitation largely affects the ability to generalize one model based on historical time-series data of a specific stock, to other stocks. Apart from the classical models, traditional machine learning algorithms such as support vector machines (SVM), random forest (RF), and artificial neural networks (ANNs) have also been used for financial time-series forecasting and these generally yield higher forecast accuracy [10, 46, 18]. However, these models still assume a pre-determined non-linear mathematical form, which may not capture the true underlying nonlinear relationship. Another line of research uses recent advances in deep learning for stock price forecasting [51, 7, 34, 41, 72, 77, 53, 94, 85]. However, most of the work has focused only on deterministic next step prediction, or classification of the stock price direction.

In this study, we investigate the problem of obtaining reliable forecasts for multi-step-ahead stock prices for a target company by using deep learning. The technical challenges to overcome include complex and nonlinear interactions, combining time-series with exogenous factors in financial sequence modeling, achieving acceptable performance, and quantifying the uncertainty associated with the model's output in a probabilistic setting. We believe that forecasting multi-step ahead stock price for a particular company is essential in order to make better informed trading decisions. We also believe that in addition to the past performance record of the target company, the future stock price is dependent on both the impact of and the correlation with stock prices of other companies. Therefore, our framework will incorporate this along with other technical and macroeconomic indicators.

In order to address the challenges mentioned above, we propose an end-to-end feature learning framework with a novel architecture for multi-step stock price forecasting. Similar to the concept of converting a natural language sentence to a word vector embedding, we conjecture that finding an appropriate embedding for the stock price history will enhance the learning process. Hence, we first encode a time-series sequence of historical records for the target company. This is then decoded to forecast the multi-step ahead closing prices. More specifically, in the encoder, the sequence of historical records are passed through a temporal convolutional network to extract the stock price's hidden characteristics. We then employ an attention mechanism to encapsulate the portions of the input sequence on which we should focus for each time-step in the forecast horizon. Concurrently, exogenous factors are mapped to a lower dimensional latent representation by passing them through an auto-encoder. For uncertainty estimation, we exploit Monte Carlo (MC) dropout layers and keep them activated during inference.

The reminder of this chapter is organized as follows. Section 5.1 discusses background. Section 5.2 provides the problem definition and details of our proposed framework. Section 5.3 covers the experimental study, including the settings and results. Finally, in Section 5.4, we provide a summary and conclusions.

## 5.1 Background

### 5.1.1 Representation Learning

It is widely believed that learning representations is one of the main factors associated with the success of deep neural networks. In fact, the performance of machine learning algorithms can be greatly affected by the choice of data representation. Finding good representations for complex input data helps algorithms better understand the data and do the necessary processing. The computational aspect of this is also crucial because mapping large, complex data to a lower-dimensional space contributes to the algorithm's computational efficiency. The lower dimensional representation can also help avoid over-fitting and yield better generalization to unseen data. Principal component analysis (PCA) is a fundamental technique in dimensionality reduction, where the raw data with $p$ features is projected linearly to a $q$ dimensional vector where $q <= p$ [69]. There are many equivalent mathematical ways for deriving the principal components. The first principal component is the direction which explains a maximal amount of variance in the data (the eigenvector corresponding to the largest eigenvalue of the covariance matrix for the data). The $k^{th}$ component is the variance-maximizing direction orthogonal to the previous $k - 1$ components. The orthogonality condition ensures this component is uncorrelated with the preceding components.

Another learning representation method that is widely used in deep learning is Restricted Boltzmann Machines (RBM). RBM is a shallow artificial neural network consists of two layers: visible and hidden. These two layers are connected by a fully bipartite graph, where no two nodes within the same layer are connected, while every node in the visible layer is connected to every node in the hidden layer and vice versa. There are three primary steps in the learning process: 1) forward pass, 2) backward pass, and 3) reconstruction error calculation. We pass the input to the hidden layer, and then reconstruct the input from the hidden layer. The loss function computes the distance between the generated output and the original input. Thus, RBM is an unsupervised learning algorithm that leverages back-propagation to find the best representation that can be used in order to later reconstruct the input. Auto-encoders have the same concept but with a deep network instead of a shallow one.

### 5.1.2 Dropout in deep learning

In deep learning, dropout layers are used for regularization in order to avoid overfitting [75]. Reduction in generalization error can be achieved by randomly dropping neurons of the network during training. This technique can be further expanded to quantify uncertainty [25]. The generated predictions of the network can be non-deterministic if Monte Carlo (MC) dropout is used. The main distinction is that MC dropout layers remain active during inference for multiple runs. Thus, for the same input, running the trained model multiple times will generate multiple predictions. The predictions generated can be viewed as samples from a probabilistic distribution. There are two main advantages of this framework: 1) it provides uncertainty estimation without any additional computational effort, and 2) it does not require any change to the network architecture.

## 5.2 Proposed Framework

In this section we introduce the notation used in this study, and then describe the details of our framework.

### 5.2.1 Notation and Problem Definition

Looking into the past for $t$ days, we define $\mathbf{S} = \{\mathbf{S}_1, \mathbf{S}_2, ..., \mathbf{S}_j\}$ as the stock price time-series for the target company we are interested in, where $\mathbf{S}_j = \{H_j, L_j, O_j, C_j, V_j\}$. Here, $\mathbf{S}_t$

represents the stock price vector for the target company at time $j$, and is composed of the highest ($H_j$), lowest ($L_j$), opening ($O_j$) and closing ($C_j$) prices as well as the volume ($V_j$) of traded stocks for the company. We also use $\mathbf{X}_j$ to represent the external factors at time $j$, which includes stock prices of other companies as well as some relevant macroeconomic indicators. The ground truth output is the closing price for the next $F$ days $\mathbf{C} = C_{t+1}, ...., C_{t+F}$, while the predicted value is $\hat{\mathbf{C}} = \hat{C}_{t+1}, ...., \hat{C}_{t+F}$. The aim of our model is to learn a non-linear mapping $\sigma$ to the multi-step ahead closing price:

$$\hat{\mathbf{C}} = \sigma(\mathbf{S}_1, \mathbf{S}_2, ..., \mathbf{S}_t, \mathbf{X}_t) \tag{5.1}$$

**Algorithm 5** Auto-Encoder training algorithm

```
Input: Dataset $\mathbf{X}_1, ...., \mathbf{X}_n$
Output: Encoder $\zeta$, Decoder $\vartheta$
1: Initialize parameters $\zeta(.) = \mathbf{W}, \mathbf{b}$ & $\vartheta(.) = \mathbf{W}', \mathbf{b}'$
2: repeat
3:   for $i = 1, \dots, N$ do
4:     $\mathcal{L} = \sum_{i=1}^{N} ||\mathbf{X}_i - \sigma'(\mathbf{W}'(\sigma(\mathbf{W}\mathbf{X}_i + \mathbf{b})) + \mathbf{b}')||^2$ calculate sum of reconstruction errors
5:     $\zeta(.)\vartheta(.) \longleftarrow$ update encoder decoder parameters
6:   end for
7: until convergence of parameters $\zeta(.)$ & $\vartheta(.)$
```

### 5.2.2 Model

Our deep learning model has two sub-models that require sequential training. The aim of the first is to map the exogenous factors into a lower-dimensional space. Then, we learn a latent representation for the stock price history and concatenate it to the output of the first model to forecast future stock price. We now describe each sub-model in details.

**Algorithm 6** Training of CAED-TCN

```
Input: training data $\{\mathbf{S}_1, ...., \mathbf{S}_N\}$, $\{\mathbf{X_1}, ..., \mathbf{X_N}\}$, lookback window size $k$, Forecast window $F$, trained auto-encoder $\zeta(.)$ & $\vartheta(.)$
Output: CAED-TCN model $\mathcal{M}(.)$
 1: Initialize the encoder $\mathcal{G}(.)$, and decoder $\mathcal{F}(.)$ in Fig. ??
 2: while stopping criteria not met do
 3:   for $t =$ k, . . . ,N do
 4:     $\mathbf{k}_t, \mathbf{v}_t = \mathcal{G}(S_{t-k}, ..., S_t)$ encoder output
 5:     Decoder:
 6:     for $\tau = t, ..., t + F$ do
 7:       $$C_\tau = \begin{cases} \hat{C}_\tau & \text{w.p. } p \\ C_\tau & \text{otherwise} \qquad \text{teacher forcing case} \end{cases}$$
 8:       $\nu_\tau = \phi(C_\tau)$, where $\phi(.)$ is a linear layer
 9:       For each LSTM cell and hidden state(h and r are the two outputs of the LSTM)
          $$\{d_\tau, h_\tau\} = \begin{cases} lstm(\nu_\tau, \mathbf{h}^l_{\tau-1}, \mathbf{r}_{\tau-1}), \text{ if } l = 1 \\ lstm(\mathbf{d}^{l-1}_\tau, \mathbf{h}^l_{\text{initialized}}, \mathbf{0}), \text{ if } \tau = 1 \\ lstm(\mathbf{d}^{l-1}_\tau, \mathbf{h}^l_{\tau-1}, \mathbf{r}_{\tau-1}), \text{ otherwise} \end{cases}$$
10:       use $\mathbf{k}_t, \mathbf{v}_t$ to compute the attention vector $a_\tau$
11:       $Z_\tau = \zeta(X_{\tau-1})$
12:       $i_\tau = concatenate[a_\tau, \mathbf{r}_\tau, Z_\tau]$
13:       $\hat{C}_\tau = MLP(\{i\}_\tau)$
14:     end for
15:     perform backward passes to update parameters of $\mathcal{G}(.)$ and $\mathcal{F}(.)$ by minimizing the loss function $\mathcal{L}(\hat{\mathbf{C}}_t, \mathbf{C}_t)$
16:   end for
17: end while
```

**5.2.2.1 Learning Embedding of Exogenous Factors** The main task of this portion of the model is to learn a suitable embedding for the exogenous factors (Algorithm 5). To achieve this, we deploy an auto-encoder architecture consisting of encoder and decoder stages. The auto-encoder is different from PCA since the orthogonality condition is relaxed in addition to stacking multiple linear layers, with a nonlinear activation function. In the encoder stage, we compress the exogenous factors vector (discussed later in the experimental study section) into a lower-dimensional vector by passing it through three linear layers with a rectified linear units (ReLU) activation function. Then, in the decoder stage, we reconstruct the original input from the compressed representation by passing it through similar stacked layers in reverse order. Hence, the architecture presents a bottleneck in the middle, from which the reconstruction of the input data is implemented. Figure 33 shows the details of our auto-encoder. More formally, we split the network into two segments: the encoder $\zeta$ and the decoder $\vartheta$.

$$\begin{aligned} \zeta &: \mathbf{X} \rightarrow \mathbf{Z} \\ \vartheta &: \mathbf{Z} \rightarrow \mathbf{X}' \end{aligned} \tag{5.2}$$

The encoder and decoder can be represented by a standard neural network, where $\mathbf{X}$, $\mathbf{X}'$, $\sigma$ and $\mathbf{Z}$ are the exogenous factors vector, the reconstructed vector, the activation function and the bottleneck representation, respectively. Let $\mathbf{W}$, $\mathbf{b}$, $\mathbf{W}'$, and $\mathbf{b}'$ represent the encoder and decoder weights and biases. Then

$$\begin{aligned} \mathbf{Z} &= \sigma(\mathbf{W}\mathbf{X} + \mathbf{b}) \\ \mathbf{X}' &= \sigma'(\mathbf{W}'\mathbf{Z} + \mathbf{b}') \\ Objective function &: \min_{\zeta,\vartheta} ||\mathbf{X} - (\zeta \circ \vartheta)\mathbf{X}|| \end{aligned} \tag{5.3}$$

The loss function of the auto-encoder is the squared distance between the original input and the constructed one:

$$\mathcal{L} = ||\mathbf{X} - \mathbf{X}'||^2 = ||\mathbf{X} - \sigma'(\mathbf{W}'(\sigma(\mathbf{W}\mathbf{X} + \mathbf{b})) + \mathbf{b}')||^2 \tag{5.4}$$

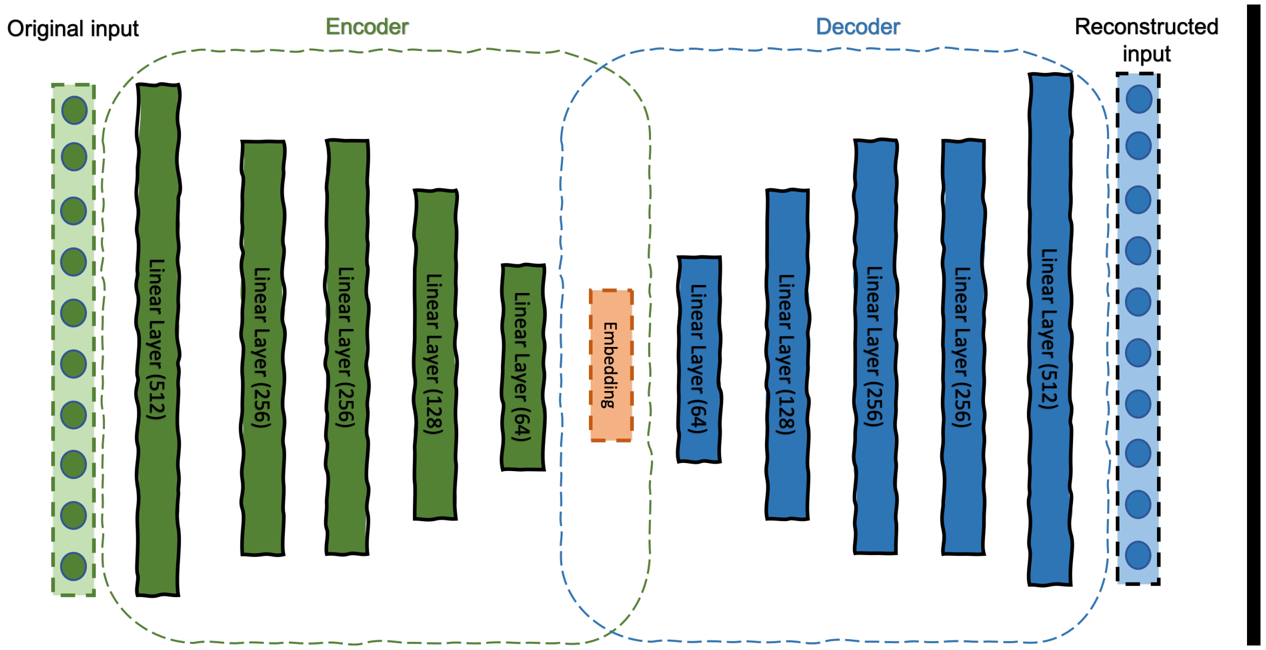


Figure 33: Learning exogenous factors embedding

**5.2.2.2 Time-series encoder decoder** We leverage the encoder decoder framework to understand time-series history and forecast future stock prices (Algorithm 6). The main assumption here is that an appropriate embedding for the stock price history can result in more accurate forecasts. The advantage of this approach is that it removes noise from the original input and captures only important local features. Hence, the main objective of the encoder in our framework is to map the history of stock price $\mathbf{S} = \{\mathbf{S}_1, \mathbf{S}_2, ..., \mathbf{S}_{t-1}\}$ to a latent representation of a fixed dimension vector. The vector is then passed to the decoder to generate the predictions. Our model is shown in Figure 34. The encoder usually consists of stacked Long Short-Term Memory (LSTM) or Gated Recurrent Unit (GRU) layers; however, we use a Temporal Convolution Network (TCN) architecture in our encoder. The encoder receives the financial sequence $\mathbf{S}_t$ and passes it through the TCN to extract local characteristics of the time series. The TCN reduces the total number of trainable parameters through the concept of parameter sharing, and by leveraging local connectivity of convolution layers. The architecture of TCN can only handle input-output sequences

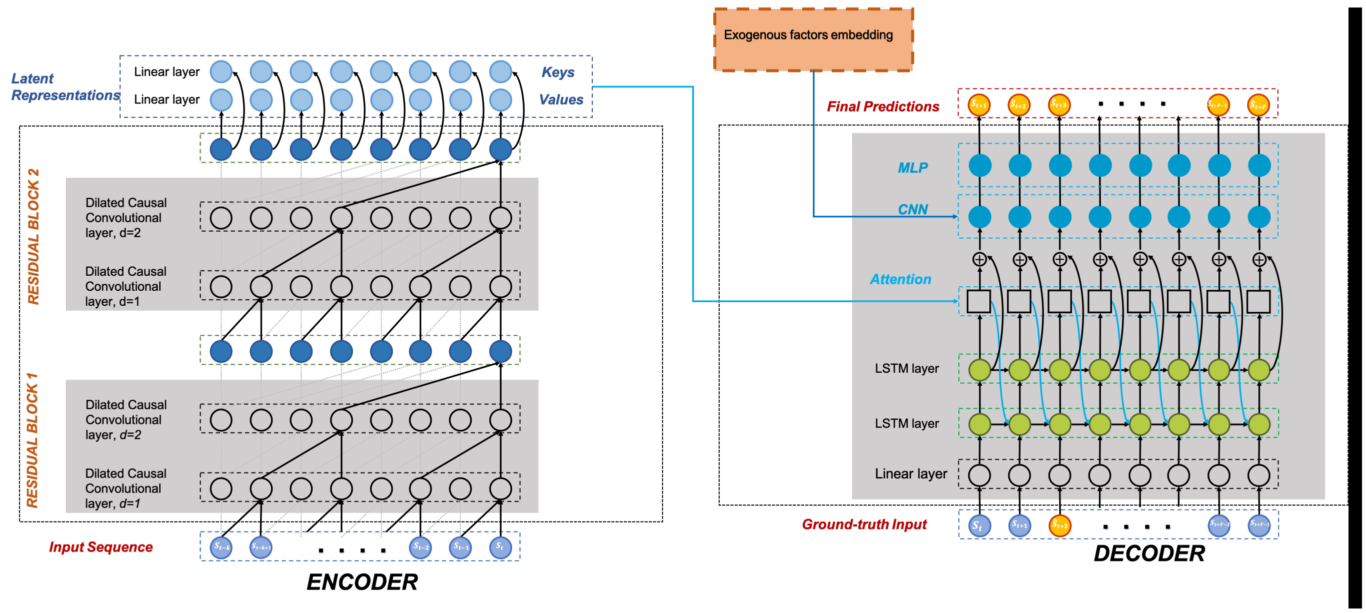


Figure 34: Encoder-Decoder Model

of the same length. Another important characteristic of TCN architecture is the causality condition for the convolutions, i.e. an output $y_t$ at time $t$ is convolved only with elements $y_\tau$ in the previous layer, where $\tau \leq t$. In other words, future data points in the sequence cannot be leaked and used in the convolution [9].

Simple causal convolutions are less preferred to dilated convolutions due to the ability of the latter to allow the receptive field to grow exponentially as we increase the number of layers. Mathematically, given an input sequence, $\mathbf{x} \in \mathbb{R}^k$ and a kernel function $\psi(.) : \{0, \dots, k-1\} \longrightarrow \mathbb{R}$, the dilated convolution $\mathbf{C}(.)$ on element $q$ of the sequence is defined as

$$\mathbf{C}(s) = (\psi \circledast_\delta \mathbf{x})(q) = \sum_{i=0}^{k-1} \psi(i) \, . \, \mathbf{x}_{s-\delta i}, \tag{5.5}$$

where $\delta = 2^\eta$ is the dilation factor, $\eta$ is the depth of the network, $N$ is the kernel size, and $q - \delta i$ accounts for the direction of the past [9]. For example, consider Figure 34 (i.e. input sequence $\mathbf{x} = \mathbf{S}$), and suppose the index $\mathbf{q} = 8$. Then the dilated convolution $\mathbf{C}(.)$ of factor

$d = 1$ and kernel size $N = 2$ will be:

$$\mathbf{C}(q) = \sum_{i=0}^{1} \psi(i) \,.\, \mathbf{S}_{8-q.i} = \psi(0) \,.\, \mathbf{S}_8 + \psi(1) \,.\, \mathbf{S}_7 \tag{5.6}$$

The encoder consists of two stacked residual blocks each consisting of two dilated convolution layers. The kernels of both layers are normalized and passed through ReLU activation function, followed by a dropout layer for regularization purposes. The problem of exploding/vanishing gradients is handled by these blocks. The encoder produces the latent representation in the form of two parallel linear layers, `keys` and `values`, as shown in Figure 34. This specific form of output is necessary in the implementation of the attention mechanism in our model.

The most recent ground truth stock price vector (with probability $p$) or the most recent prediction generated (with probability $(1 - p)$) is used to initialize the decoder; we used a value equal to 0.1. The decoder consists of two linear layers, followed by two LSTM layers, an attention model, a CNN layer, and a two-layer MLP (multi-layer perceptron). The decoder forecasts the future stock closing price $\hat{\mathbf{C}}$ for the target company. For random instances in the forecast horizon, with predetermined probability $p$, we feed the decoder with the forecast generated at the previous step instead of the associated ground truth. This technique helps with training the model to avoid the propagation of any inaccurate prediction to subsequent forecasts in the forecast horizon.

We also integrate the general attention mechanism in our model, which is a component of our network's architecture, and is in charge of quantifying the interdependence between the input and the output [13]. It helps the model focus on the most important segments of the input sequence at each time step in the forecast horizon. More precisely, for each time step in the forecast horizon, we pass the current output of the last LSTM layer along with the output of the encoder, `keys` and `values`, to the attention function. Then, through batch matrix multiplication, we obtain the attention context, which identifies the significant input segment at the current time step.

## 5.3 Experimental Study

In this section, we evaluate our model and compare it with a set of baseline models from the literature. We start by describing the dataset used in our study along with the experimental settings and then provide an analysis of the results.

### 5.3.1 Dataset

We consider two datasets to train and evaluate our framework. Amazon is the target company in the first dataset, while we use Apple in the second. The selection is based on the large trading volumes of these enterprises, which affects the entire market and the S&P 500 companies in particular, since we aim to study the effect of market indices (e.g. S&P 500) on individual stocks. We fetched the historical multivariate time-series data from the *Yahoo Finance* website for the last 10 years. The time-series includes the daily open, close, low, and high prices in addition to the traded volume. For the exogenous factors, we obtained the closing price for the rest of the S&P 500 companies, as well as the NASDAQ, S&P 500, and Dow Jones indices. We also used VIX, *a.k.a* the Chicago Board Options Exchange (CBOE) index, which is a real-time market index that captures the market's expectation of 30-day forward-looking volatility. In addition, we performed feature engineering to obtain technical indicators such as the moving average based on different rolling values, the difference in traded volumes over the previous two days, as well as the time-series seasonality, trend and residual for the target company. Furthermore, we normalized all inputs to ensure learning stability and efficiency [67].

### 5.3.2 Experimental Setup

We evaluate the performance of our confident, attentive, encoder-decoder model with TCN architecture (CAED-TCN), and compare it to the performance of the following baselines, where each is implemented with with the necessary fine-tuning.

- MLP: A multi layer perceptron composed of stacked linear layers
- Vanila LSTM: A single LSTM layer followed by an output layer

Table 7: Performance of the Stock Price Prediction Models

| Forecast window | | One-step | | | Three-step | | | Five-step | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Dataset | Model | $RMSE$ | $sMAPE$ | $sL1$ | $RMSE$ | $sMAPE$ | $sL1$ | $RMSE$ | $sMAPE$ | |
| AMZN | MLP | 0.068 | 0.0033 | 0.057 | 0.073 | 0.0032 | 0.057 | 0.074 | 0.0036 | 0.059 |
| | Vanilla LSTM | 0.069 | 0.0034 | 0.065 | 0.073 | 0.0031 | 0.058 | 0.074 | 0.0037 | 0.059 |
| | multi-LSTM | 0.067 | 0.0033 | 0.055 | 0.067 | 0.0032 | 0.054 | 0.068 | 0.0033 | 0.054 |
| | Enc-Dec LSTM | 0.064 | 0.0030 | 0.052 | 0.067 | 0.0031 | 0.054 | 0.067 | 0.0032 | 0.053 |
| | modified LAS | 0.048 | 0.0011 | 0.043 | 0.048 | 0.0014 | 0.037 | 0.049 | 0.0015 | 0.037 |
| | **CAED-TCN** | **0.026** | **0.0004** | **0.021** | **0.035** | **0.0010** | **0.034** | **0.042** | **0.007** | **0.028** |
| AAPL | MLP | 0.044 | 0.0010 | 0.039 | 0.058 | 0.0017 | 0.051 | 0.064 | 0.0021 | 0.059 |
| | Vanilla LSTM | 0.043 | 0.0012 | 0.039 | 0.047 | 0.0011 | 0.039 | 0.050 | 0.0012 | 0.041 |
| | multi-LSTM | 0.039 | 0.0007 | 0.033 | 0.040 | 0.0009 | 0.034 | 0.046 | 0.0011 | 0.041 |
| | Enc-Dec LSTM | 0.062 | 0.002 | 0.058 | 0.064 | 0.0021 | 0.060 | 0.074 | 0.0029 | 0.070 |
| | modified LAS | 0.025 | 0.0003 | 0.023 | 0.040 | 0.0008 | 0.033 | 0.044 | 0.0009 | 0.038 |
| | **CAED-TCN** | **0.017** | **0.0001** | **0.0142** | **0.035** | **0.0006** | **0.031** | **0.035** | **0.0008** | **0.029** |

- Multi-LSTM: Three LSTM layers followed by an output layer.
- Encoder-Decoder LSTM: An encoder and a decoder, each with two LSTM layers.
- Modified LAS (Chan, William 2015[13]): Listen attend and spell (LAS) is an encoder-decoder sequence model with attention. The structure of the encoder consists of 3 pyramidal bidirectional LSTM (pBLSTM) layers and the decoder is composed of 2 BLSTM layers.

The sliding window size for our input sequence is 50 days, which we used to forecast over three different forecast horizon lengths (one, three and five steps ahead). We also evaluated the impact of exogenous factors and the correlation with other companies on the performance of the model. The evaluation metrics used in our analysis are: Root Mean Square Error ($RMSE$), Symmetric Mean Absolute Percentage Error ($sMAPE$), and Smooth L1 ($sL1$). Similar to the previous chapter, although learner sensitivity to outliers is not the focus of our work and can be further investigated in future studies, the effect of anomalies are better captured by RMSE and sMAPE. The last measure, also called Huber loss, is less sensitive to outliers and uses a squared term only if the absolute error is under 1.

$$RMSE = \frac{1}{T}\sqrt{\sum_{t=1}^{T}(C_t - \hat{C}_t)^2} \tag{5.7}$$

$$sMAPE = \frac{1}{T}\sum_{t=1}^{T}\frac{|C_t - \hat{C}_t|}{(|C_t| - |\hat{C}_t|)/2 + 1} \tag{5.8}$$

$$sL1 = \frac{1}{T}\sum_{t=1}^{T} z_t \tag{5.9}$$

where $z_t$ is given by:

$$z_t = \begin{cases} 0.5 * (C_t - \hat{C}_t)^2, & \text{if } |C_t - \hat{C}_t| < 1 \\ |C_t - \hat{C}_t| - 0.5, & \text{otherwise} \end{cases} \tag{5.10}$$

During inference, for uncertainty estimation in our forecasts, we focus on the one-step-ahead ($o-s-a$) forecasts and run the model for 1000 epochs while keeping the MC dropout activated (Algorithm 7). For the same input, this process generates 1000 different $o-s-a$ forecasts for each period, which in turn allows us to build a confidence interval for the forecast for each period. We constructed both 90% & 95% confidence intervals, and we then used the coverage probability ($CP$) metric to evaluate the uncertainty estimation. Here $CP$ is the proportion of the confidence intervals constructed that contain the ground truth being forecast.

**Algorithm 7** Inference of CAED-TCN

```
Input: testing data $\{\mathbf{S}_t\}$; trained CAED-TCN model $\mathcal{M}(.)$, exogenous factors vector $\mathbf{X_t}$,
    dropout probability $p$, number of iterations $N$
Output: prediction mean $\mu_{\mathbf{J}_t}$ and uncertainty $\xi_{\mathbf{J}_t}$
1: for $i = 1, \ldots, N$ do
2:     $\hat{\mathbf{C}}_t^i = Dropout(\mathcal{M}(\mathbf{C}_t, X_t), p)$
3: end for
4: $\mu_{\mathbf{C}_t} = \frac{1}{N} \sum_{i=1}^{N} \hat{\mathbf{C}}_t^i$
5: $\xi^2_{\mathbf{C}_t} = \frac{1}{N} \sum_{i=1}^{N} (\hat{\mathbf{C}}_t^i - \mu_{\mathbf{C}_t})^2$
6: return $\mu_{\mathbf{C}_t}, \xi_{\mathbf{C}_t}$
```

### 5.3.3 Results

Table 7 shows the performance, based on our three metrics, of our model and the other five models, evaluated over the entire test dataset for both the AMZN and the AAPL datasets for three different forecast horizon lengths. As expected, the performance of all models starts to degrade as we increase the length of the forecast window (except in a couple of cases, with respect to sL1, which treats errors selectively). This observation agrees with the fundamental forecasting concept where forecast accuracy is lower as we look further into the future.

In looking at the baseline models, although Vanilla LSTM is designed to capture long-term relationships, MLP actually performs as well or slightly better. This might be attributable to the fact that the depth of MLP can provide a more meaningful representation. However, both MLP and vanilla LSTM have the poorest performance metrics relative to the other models, indicating that simple models do not efficiently capture stock market dynamics. Modifying the vanilla LSTM model by stacking multiple LSTM layers (multi-LSTM) improves the performance since the model now is deep and with LSTM components. The model performs even better with the adoption of the encoder-decoder framework. The key to this performance enhancement is the latent representation learning performed within this framework. Finally, the modified LAS outperforms all previous models with the adoption of an attention mechanism. The main advantage of adopting the attention mechanism is to

Table 8: Incorporation of Additional Information & Attention in the Stock Price Prediction Model

| Dataset | Model | $RMSE$ | $sMAPE$ | $sL1$ |
|---|---|---|---|---|
| AMZN | univariate | 0.048 | 0.0014 | 0.039 |
| | multivariate | 0.042 | 0.0012 | 0.034 |
| | **multivariate & exogenous factors** | **0.035** | **0.0007** | **0.028** |
| | multivariate & exogenous factors w/o attention | 0.041 | 0.0010 | 0.034 |
| AAPL | univariate | 0.071 | 0.0029 | 0.064 |
| | multivariate | 0.051 | 0.0013 | 0.045 |
| | **multivariate & exogenous factors** | **0.035** | **0.0008** | **0.029** |
| | multivariate & exogenous factors w/o attention | 0.047 | **0.0011** | 0.042 |

exploit the long-term associations between inputs and outputs of the dataset.

As Table 7 illustrates, our model, CAED-TCN, is superior to the baseline models with respect to all metrics and all forecast horizons. In fact, even the five-steps-ahead forecasts from our model outperform the one-step-ahead forecasts from all of the baseline models. There are two major reasons for this: 1) the adoption of the TCN architecture for the encoder, and 2) the incorporation of the learned representation for exogenous factors. In addition to the enhancement provided by the attention mechanism, teacher forcing and the representation learning of the exogenous factors, the TCN encoder exploits local connectivity and parameter-sharing to provide more efficient and stable learning. Although the Modified LAS approach provides the closest performance to our model, it requires learning $13,680,500$ parameters while our model only has $1,362,688$ learnable parameters. This is an essential contribution of our work, where we are able to reduce the learning computational complexity by approximately 90%.

Next, we focus on the five-steps ahead forecasts and illustrate the importance of incorporating exogenous factors as well as the multivariate time-series data in our model. The univariate time-series input contains only closing price history, while the multivariate time-series includes all values in $\mathbf{S}_t$. Clearly, learning the latent representation for the multivariate

Table 9: Coverage Probability of the Stock Price Prediction Model

| Dataset | Model | CP at 99% | CP at 95% |
|---|---|---|---|
| AMZN | Modified LAS | 0.71 | 0.57 |
| | **CAED-TCN** | **0.89** | **0.83** |
| AAPL | Modified LAS | 0.87 | 0.71 |
| | **CAED-TCN** | **0.98** | **0.87** |

time-series feeds the decoder with more information that better explains the past behavior of the target company stock. For AMZN this modification to the stock history encoding yields a reduction of around 12%, 14% and 13% for the $RMSE$, $sMAPE$ and $sL1$ respectively, as shown in Table 8. Furthermore, the representation learning of the external factors discussed in the previous section advanced the performance by 16% for the $RMSE$, 41% for $sMAPE$ and 17% for the $sL1$. Similarly, for AAPL stock, the incorporation of multivariate time-series reduced the RMSE by 28%, while the incorporation of exogenous factors led to a further 16% reduction. The same general trend can be observed for the other evaluation metrics. Therefore, the analysis here suggests that including exogenous factors and the more comprehensive encoding for the stock price history enhances the learning of our model. Furthermore, the table illustrates the importance of the attention related aspect in our architecture for the learning enhancement process.

Finally, we end our discussion by examining the uncertainty estimation associated with our model to provide some measure of confidence in our forecasts. Once again, we restrict our attention to the one-step ahead prediction, and the comparison is limited to our CAED-TCN model and the only other truly competitive baseline model (modified LAS). Table 9 shows the coverage probability (CP) for both models for both datasets. The $CP$ of our CAED-TCN is higher than the $CP$ of modified LAS for both datasets across different confidence levels. More precisely, at a 95% confidence level, the $CP$ for CAED-TCN exceeds that of the modified LAS by 26% and 16% for AMZN and AAPL respectively. The same pattern is observed at a 99% confidence level with a 27% advantage for the AMZN dataset and 11%

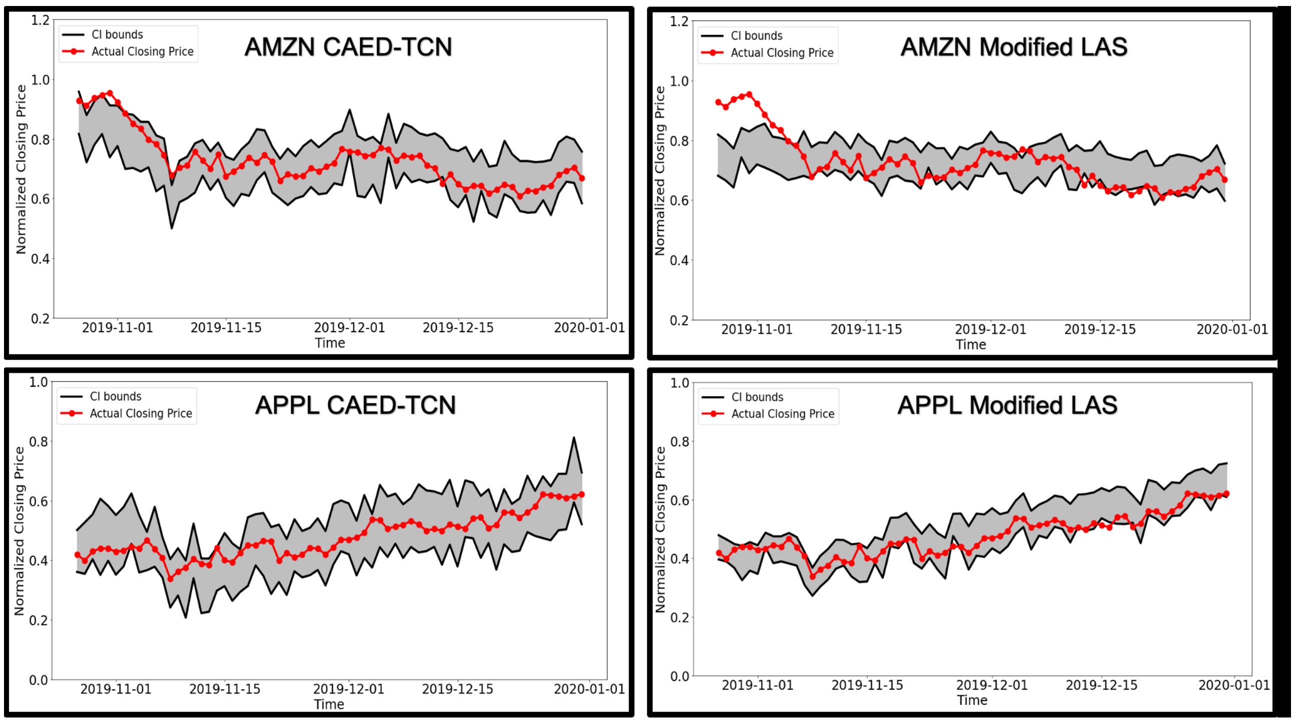


Figure 35: 99% Confidence intervals

for AAPL. Our model was able to capture the movement of the closing price with narrower confidence intervals for both stocks (Figures 35 & 36). These figures provide more support to the conclusion that CAED-TCN is superior to the modified LAS model.

## 5.4 Conclusion

In this chapter we have proposed a confident, attentive, encoder-decoder with TCN (CAED-TCN) model, a novel deep-learning based approach for stock closing price predictions. We address several technical challenges such as representation learning for exogenous factors, latent representation for multivariate time-series, model robustness, and forecast uncertainty estimation. We first design the auto-encoder to learn how to represent exogenous factors. Then, we leverage the encoder-decoder framework to map the historical records of the target stock to a latent representation. Our design for the encoder consists of a temporal convolution network (TCN) structure, while the decoder has two stacked LSTM layers followed by a convolution layer and an MLP. A general attention mechanism is deployed, and a teacher-forcing policy helps the model to learn how to recover from early mistakes in the forecast horizon.

We learn that representation learning with exogenous factors, and additional historical time-series data incorporation enhance the performance of our model. We quantify uncertainty estimation by designing the architecture with Monte Carlo dropout layers. This step is done during inference by running the model multiple times to build a confidence interval instead of relying on a single forecast. Our experimental study on AMZN and AAPL stocks demonstrates that our model outperforms other advanced models.

The next step would be to extend the model to perform online learning and predict stock prices in real time. It can also be deployed as an input model to design a more comprehensive automated trading system. Another direction for this work might be to tailor this model to fit other application domains with relatively similar underlying structures.

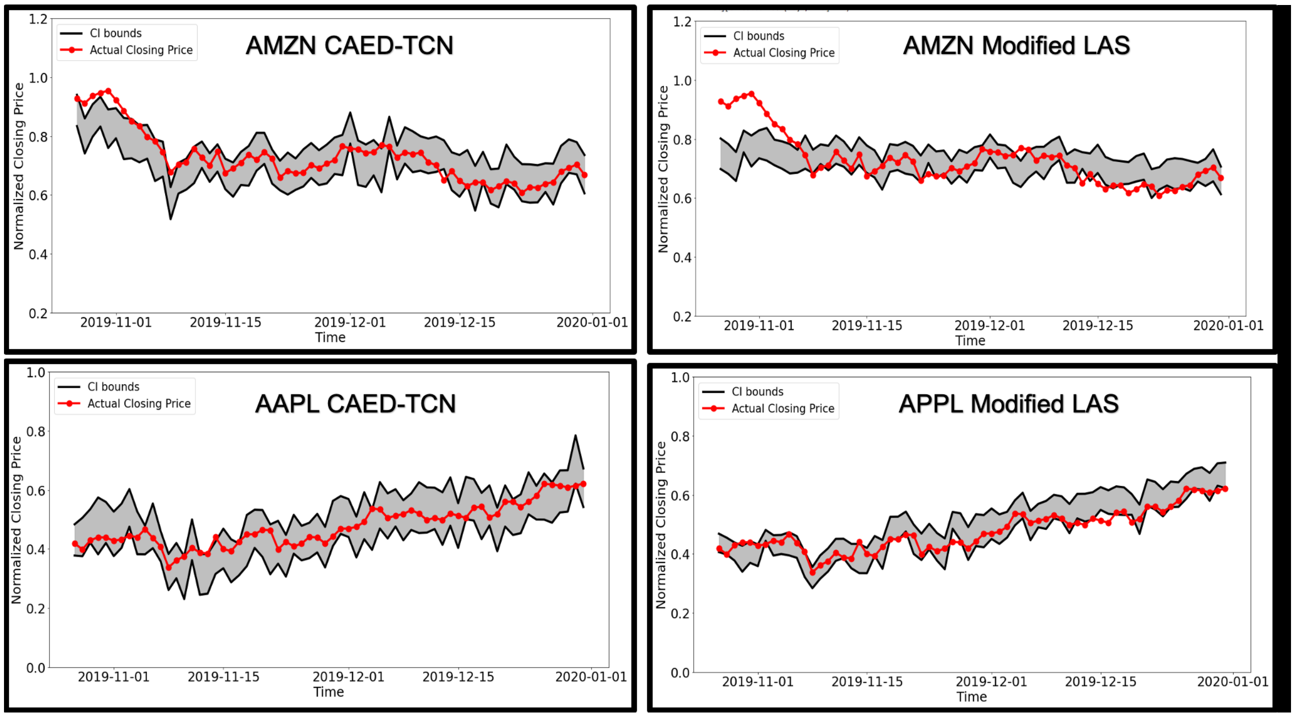


Figure 36: 95% Confidence intervals

# 6.0 Summary and Conclusions

The work discussed in this dissertation has been in the realm of harnessing advancements in deep learning to build reliable time-series forecasting models for real-life applications. Our models for the three applications studied share three fundamental characteristics: 1) the ability to generate demand for multiple steps in the future, 2) the ability to produce forecast distributions instead of single estimates, and 3) the flexibility to incorporate exogenous factors. In Chapters 3 & 4, we focus on forecasting travel demand for both taxi & ride-sharing companies. This is vital to develop efficient and effective dispatch systems that can optimize important business metrics. The main task of Chapter 3 was to design a novel, reliable region-based prediction framework that is capable of providing the following information: 1) Expected real-time demand originating from each micro-geographical region within a specific time interval for multiple steps in the future, and 2) An uncertainty estimate for each forecast. Our contribution in this chapter can be described as following:

- We introduce a two-stage novel deep neural network (MSPN-TCN). Our network utilizes the encoder-decoder framework with TCN architecture and MC dropout layers to generate the desired distributions.
- We deploy transfer learning in the second stage to incorporate exogenous factors (e.g., time-series trend and seasonality, and temporal clustering) for further learning enhancement.
- Our experimental study shows the superiority of our model over other advanced baselines on two datasets for NYC Taxi and Uber services.
- The number of learnable parameters are significantly lower with the deployment of TCN and transfer learning when compared to modified-LAS (the most competing model).

In Chapter 4, the scope of our approach was extended to include destination information in our forecasting. This results in a time series of graphs, and we forecast not just demand but the flow of demand. Demand flow network forecasting is essential, especially in a world that is moving increasingly towards a shared economy. Recent innovative ride-sharing products rely heavily on prior knowledge of demand with similar routes. The availability of such information can greatly improve the quality of supply-demand matches. The desired outcomes of our model are the same as in Chapter 3, but for the demand flow network instead of pick-up zones based demand. The incorporation of destination in the modeling substantially affects the nature of data we deal with, and changes the view point of demand from one involving isolated data points to one involving relationships and interactions between geographical nodes. Our contributions in Chapter 4 can be summarized as following:

- We introduce an end-to-end encoder-decoder framework capable of understanding the topological characteristics of travel demand flow networks and generating stochastic travel demand networks for multiple step ahead in the future.
- We test our model on two real-world datasets and the results confirm the advantage our model has over other common advanced baselines based on three different evaluation metrics.
- We test our model on data prior and post COVID-19 pandemic, and the results support the robustness of our model.

Finally, in Chapter 5, we studied how the overall structure of our deep learning approach might generalize to other domains, by applying it to another complex area, namely finance. The structure of the model is similar to the models described in Chapters 3 & 4 where the encoder-decoder framework is used for representation learning. Also, the output of the model is in the form of prediction intervals for multiple steps in the future. The goal in Chapter 5 is to build a deep-learning based model for stock price distribution forecasting for multiple steps ahead. Our contributions in this chapter are twofold:

- We propose an end-to-end feature learning framework with a novel architecture for multi-step stock price forecasting. We use an auto-encoder for the unsupervised learning part where we learn an embedding for the external factors that include the stock price of other companies in the market along with important market indices. We also leverage the encoder-decoder framework to map the historical time-series into a latent representation and then decode it to generate future forecasts. The architecture of our model incorporates MC dropout for future stock price uncertainty estimation.

- We validate our model on two real datasets for AMZN and AAPL stocks and our experiments demonstrate the performance improvement obtained by our model when compared to other baseline methods.

# Appendix A Glossary

| | | |
|---|---|---|
| **AI** | Artificial Intelligence. | 1 |
| **ANN** | Artificial Neural Network. | 9 |
| **BNN** | Bayesian Neural Network. | 29 |
| **CNN** | Convolutional Neural Network. | 3 |
| **CP** | Coverage Probability. | 98 |
| **CRF** | Conditional Random Field. | 27 |
| **DL** | Deep Learning. | 1 |
| **GCRF** | Gaussian Conditional Random Field. | 27 |
| **GPU** | Graphical Processing Unit. | 1 |
| **GRU** | Gated Recurrent Unit. | 28 |
| **IoT** | Internet of Things. | 2 |
| **ITS** | Intelligent Transportation Systems. | 3 |
| **LAS** | Listen Attend Spell. | 28 |
| **LSTM** | Long Short Term Memory. | 17 |
| **ML** | Machine Learning. | 1, 24 |
| **MLP** | Multi Layer Perception. | 6 |
| **NLP** | Natural Language Processing. | 1 |
| **RF** | Random Forest. | 83 |
| **RNN** | Recurrent Neural Network. | 3 |
| **SVM** | Support Vector Machine. | 2 |
| **TCN** | Temporal Convolution Network. | 19 |

# Appendix B Region Based Model

Data loading:

```
class MyDataset(data.Dataset):
    def __init__(self, X, window=50):
        self.X = X
        self.window=window

    def __len__(self):
        return len(self.X)-2*self.window

    def __getitem__(self,index):
        X = torch.from_numpy(self.X[index:index+self.window,:-22]).float()

        X_ext=torch.from_numpy(self.X[index+self.window,-22:]).float()
        Y = torch.from_numpy(self.X[index+self.window:index+2*self.window,:-22]).float() #.reshape(522)
        Y_2=torch.from_numpy(self.X[index+self.window+1,:-22]).float()
        return X,Y,X_ext,Y_2

num_workers = 8 if cuda else 0
batch_size=32
# Training
train_dataset = MyDataset(train)
train_loader_args = dict(shuffle=True, batch_size=batch_size, num_workers=num_workers, pin_memory=True)
if cuda else dict(shuffle=True, batch_size=batch_size)
train_loader = data.DataLoader(train_dataset, **train_loader_args)
# Testing
test_dataset = MyDataset(test)

test_loader_args = dict(shuffle=False, batch_size=batch_size, num_workers=num_workers, pin_memory=True)
if cuda else dict(shuffle=False, batch_size=batch_size)
test_loader = data.DataLoader(test_dataset, **test_loader_args)
```

TCN:

```
class Chomp1d(nn.Module):
    def __init__(self, chomp_size):
        super(Chomp1d, self).__init__()
        self.chomp_size = chomp_size

    def forward(self, x):
        return x[:, :, :-self.chomp_size].contiguous()

class TemporalBlock(nn.Module):
    def __init__(self, n_inputs, n_outputs, kernel_size, stride, dilation, padding, dropout=0.2):
        super(TemporalBlock, self).__init__()
        self.conv1 = weight_norm(nn.Conv1d(n_inputs, n_outputs, kernel_size,
                                           stride=stride, padding=padding, dilation=dilation))
        self.chomp1 = Chomp1d(padding)
        self.relu1 = nn.ReLU()
        self.dropout1 = nn.Dropout(dropout)
        self.conv2 = weight_norm(nn.Conv1d(n_outputs, n_outputs, kernel_size,
                                           stride=stride, padding=padding, dilation=dilation))
        self.chomp2 = Chomp1d(padding)
        self.relu2 = nn.ReLU()
        self.dropout2 = nn.Dropout(dropout)

        self.net = nn.Sequential(self.conv1, self.chomp1, self.relu1, self.dropout1,
                                 self.conv2, self.chomp2, self.relu2, self.dropout2)
        self.downsample = nn.Conv1d(n_inputs, n_outputs, 1) if n_inputs != n_outputs else None
        self.relu = nn.ReLU()
        self.init_weights()

    def init_weights(self):
        self.conv1.weight.data.normal_(0, 0.01)
      self.conv2.weight.data.normal_(0, 0.01)
        if self.downsample is not None:
           self.downsample.weight.data.normal_(0, 0.01)

    def forward(self, x):
        out = self.net(x)
        res = x if self.downsample is None else self.downsample(x)
        return self.relu(out + res)

class TemporalConvNet(nn.Module):
    def __init__(self, num_inputs, num_channels, kernel_size=2, dropout=0.2):
        super(TemporalConvNet, self).__init__()
        layers = []
        num_levels = len(num_channels)
        for i in range(num_levels):
            dilation_size = 2 ** i
            in_channels = num_inputs if i == 0 else num_channels[i-1]
            out_channels = num_channels[i]
            layers +=
            [TemporalBlock(in_channels, out_channels, kernel_size, stride=1, dilation=dilation_size,
                                     padding=(kernel_size-1) * dilation_size, dropout=dropout)]

        self.network = nn.Sequential(*layers)

    def forward(self, x):
        return self.network(x)

class TCN(nn.Module):
    def __init__(self, input_size, output_size, num_channels, kernel_size, dropout):
        super(TCN, self).__init__()
        self.tcn = TemporalConvNet(input_size, num_channels, kernel_size=kernel_size, dropout=dropout)

    def forward(self, inputs):
        """Inputs have to have dimension (N, C_in, L_in)"""
        """N: """
        """C_in: """
        """L_in: seq_len"""
        y1 = self.tcn(inputs)  # input should have dimension (N, C, L)
        return y1
```

Attention:

```
class Attention(nn.Module):
  def __init__(self):
    super(Attention, self).__init__()
  def forward(self, query, key, value):
    '''
    :param query :(N,context_size) Query is the output of LSTMCell from Decoder
    :param key: (N,key_size) Key Projection from Encoder per time step
    :param value: (N,value_size) Value Projection from Encoder per time step
    :return output: Attended Context
    :return attention_mask: Attention mask that can be plotted
    '''
    energy = torch.bmm(key.permute(1,0,2), query.unsqueeze(2))
    energy=energy.squeeze(2).to(device)
    attention = nn.functional.softmax(energy, dim=1)
    context = torch.bmm(attention.unsqueeze(1), value.permute(1,0,2)).squeeze(1)
    return context, attention
```

Encoder:

```
class Encoder(nn.Module):
  def __init__(self, input_dim, hidden_dim, value_size=256,key_size=256):
    super(Encoder, self).__init__()
    self.tcn = TCN(input_size=input_dim,output_size=hidden_dim,num_channels=[32, 528, 50],
            kernel_size=2,dropout=0.1)
    self.key_network = nn.Linear(50, value_size)
    self.value_network = nn.Linear(50, key_size)

  def forward(self,x):
    x = x.permute(0,2,1)
    x = self.tcn(x)
    keys = self.key_network(x.permute(0,2,1))
    value = self.value_network(x.permute(0,2,1))
    return keys, value
```

## Decoder

```
class Decoder(nn.Module):
  def __init__(self, input_dim, hidden_dim, value_size=256, key_size=256,  isAttended=False):
    super(Decoder, self).__init__()
    self.linear1=nn.Linear(input_dim,hidden_dim)
    self.lstm1 = nn.LSTMCell(input_size=hidden_dim+value_size, hidden_size=hidden_dim)
    self.lstm2 = nn.LSTMCell(input_size=hidden_dim, hidden_size=key_size)
    self.isAttended = isAttended
    if(isAttended):
      self.attention = Attention()
    self.linear2 = nn.Linear(key_size+value_size,input_dim)
    self.relue=nn.ReLU()

  def forward(self, keys, values, y=None, train=True,teacher_forcing_rate=0.8):
    keys=keys.permute(1,0,2) #seq_len, batch, keys_size
    values=values.permute(1,0,2) #seq_len, batch, values_size
    batch_size = keys.shape[1]
    if(train):
      max_len =  y.shape[1]
      out1 = self.linear1(y)
    else:
      max_len = 50
    predictions_list = []
    hidden_states = [None, None]
    prediction = torch.zeros(batch_size,1).to(device)
    for i in range(max_len):
      '''
      teacher forcing technique
      '''
      if(train):

        if y ==None:
            teacher_forcing_rate=0
        if i==0:
            teacher_forcing_rate=1
        teacher_forcing = True if random.random() < teacher_forcing_rate else False
        if teacher_forcing:
            downsampled_input = out1[:,i-1,:]
        else:
            print("i",i,np.shape(prediction))
            downsampled_input=self.linear1(prediction)
      else:
        downsampled_input = self.linear1(prediction)
      if i ==0:
        context=torch.zeros(batch_size,values.size(-1)).to(device)
      inp = torch.cat([downsampled_input,context], dim=1)
      hidden_states[0] = self.lstm1(inp,hidden_states[0])

      inp_2 = hidden_states[0][0]
      hidden_states[1] = self.lstm2(inp_2,hidden_states[1])
      output = hidden_states[1][0]
      query=output
      value=values[:,:,:]
      if (self.isAttended):
        context,attention=self.attention(query, keys, value)
      else:
        context=value[-1,:,:]
      prediction = self.linear2(torch.cat([output, context], dim=1))
      predictions_list.append(prediction.unsqueeze(1))
    return torch.cat(predictions_list, dim=1)
```

## Seq2seq:

```
class Seq2Seq(nn.Module):
  def __init__(self,input_dim,hidden_dim,value_size=256, key_size=256,isAttended=False):
    super(Seq2Seq,self).__init__()
    self.encoder = Encoder(input_dim, hidden_dim)
    self.decoder = Decoder(input_dim, hidden_dim)
  def forward(self,x, y=None,train=True):
    key, value = self.encoder(x)
    if(train):
      predictions = self.decoder(key, value, y)
    else:
      predictions = self.decoder(key, value, y=None, train=False)
    return predictions
```

Decoder (stage II):

```
class Stage2(nn.Module):
  def __init__(self, input_dim, hidden_dim,output_size, value_size=256,extenal_size=22):
    super(Stage2, self).__init__()
    self.encoder = Encoder(input_dim, hidden_dim)
    self.linear1 = nn.Linear(value_size+extenal_size, hidden_dim*2)       #269=hid_dim+ext_dim
    self.linear2 = nn.Linear(hidden_dim*2, hidden_dim)
    self.linear3 = nn.Linear(hidden_dim*2, hidden_dim)
    self.linear_output = nn.Linear(hidden_dim, output_size)
    self.relue=nn.ReLU()
    self.bn=nn.BatchNorm1d(hidden_dim*2)
    self.bn2=nn.BatchNorm1d(hidden_dim)

  def forward(self,x,x_ext):
    batch_size=x.size(0)
    zz,enc_output=self.encoder(x)
    out=F.dropout(enc_output, p=0.3, training=True)
    inp=torch.cat([out[:,-1,:].reshape(batch_size,-1),x_ext], dim=1)
    del zz
    del enc_output
    inp = self.linear1(inp)
    inp=self.relue(inp)
    inp=F.dropout(inp, p=0.2, training=True)
    inp = self.linear2(inp)
    inp=self.relue(inp)
    inp=F.dropout(inp, p=0.2, training=True)
    linear_output= self.linear_output(inp)
    return linear_output
```

Training Function:

```
def train(model,train_loader, criterion,criterion2,criterion3, optimizer,i):
  model.train()
  loss_RMSE = 0
  loss_L1_smooth=0
  loss_sMAPE=0
  start_time = time.time()
  for batch_num, (x,y,X_ext,Y_2) in enumerate(train_loader):
      x = x.to(device) #(batch,seq_len=130,input_dim=522)
      y = y.to(device)
      predictions = model(x,y)
      loss = torch.sqrt(criterion(predictions[:,0:264], y[:,0:264]))
      loss2 = criterion2(predictions[:,0:264], y[:,0:264])
      loss3 = criterion3(predictions[:,0:264], y[:,0:264])
      loss.backward()
      optimizer.step()
      loss_RMSE+=loss.item()
      loss_L1_smooth+=loss2.item()
      loss_sMAPE+=loss3.item()
      del predictions
      del loss
      del x,y
  loss_RMSE /= len(train_loader)
  loss_L1_smooth/= len(train_loader)
  loss_sMAPE/= len(train_loader)
  end_time=time.time()
  return loss_RMSE,loss_L1_smooth,loss_sMAPE
```

Validation function:

```
def val(model,val_loader, criterion,criterion2,criterion3, optimizer,i):
  with torch.no_grad():
    model.eval()
    loss_RMSE = 0
    loss_L1_smooth=0
    loss_sMAPE=0
    start_time = time.time()
    output=[]
    for batch_num, (x,y,X_ext,Y_2) in enumerate(val_loader):
        x = x.to(device)
        y = y.to(device)
        predictions = model(x,y,train=True)
        # print("predictions",predictions.size())
        loss = torch.sqrt(criterion(predictions[:,0:264], y[:,0:264]))
        loss2 = criterion2(predictions[:,0:264], y[:,0:264])
        loss3 = criterion3(predictions[:,0:264], y[:,0:264])
        output.append(predictions)
        loss_RMSE+=loss.item()
        loss_L1_smooth+=loss2.item()
        loss_sMAPE+=loss3.item()
        del predictions
        del loss
        del x,y

    loss_RMSE /= len(val_loader)
    loss_L1_smooth/= len(val_loader)
    loss_sMAPE/= len(val_loader)
    end_time=time.time()
    return loss_RMSE,loss_L1_smooth,loss_sMAPE,output
```

# Appendix C Demand Flow Prediction

Graph generation:

```
def get_complete_adj(g,edge_idx,zones=270):
  matrix=np.zeros((zones,zones))
  att_list=list(nx.get_edge_attributes(g,'demand').values())
  keys=list(nx.get_edge_attributes(g,'demand').keys())
  for i in range(len(keys)):
    r,c=int(keys[i][0]),int(keys[i][1])
    matrix[r-1,c-1]=att_list[i]
  return matrix

def graph_list(data,weather):
  data=data.dropna()
  GRAPHS=[]
  external=[]
  adj=[]
  edge_idx=[]
  min_time=pd.to_datetime(data['Pickup_DateTime'].min()).floor('h')
  max_time=pd.to_datetime(data['Pickup_DateTime'].max()).floor('h')+pd.DateOffset(hours=1)
  time_hour_list=pd.date_range(min_time,max_time, freq='H')
  for t in time_hour_list:
    t=str(t)
    print(t)
    window=data[(data['Pickup_DateTime'] >= t) & (data['Pickup_DateTime'] <
        str(pd.to_datetime(t)+pd.DateOffset(hours=1)))]
    print(len(window))
    data_window=pd.DataFrame(window.groupby(['PUlocationID','DOlocationID']).count())
                ['Pickup_DateTime'].reset_index()
    data_window.rename({'Pickup_DateTime': 'demand'}, axis=1, inplace=True)
    g=graph_generation(data_window)
    ext=get_external_factors(t,weather)
    external.append(ext)
    GRAPHS.append(g)
    edge_idx.append(g.edges(data=True))
    adj_matrix=get_complete_adj(g,edge_idx)
    adj.append(adj_matrix)
  return GRAPHS,np.array(adj),edge_idx,external
```

Data loading:

```
class MyDataset(data.Dataset):
    def __init__(self, adj,ext,window=5,pred_window=5):
        self.adj = adj
        self.ext=np.array(ext)
        self.window=window
        self.pred_window=pred_window
    def __len__(self):
        return len(self.adj)-2*self.window
    def __getitem__(self,index):
        adj=torch.from_numpy(self.adj[index:index+self.window]).float()
        x=torch.eye(self.adj.shape[1]).float()
        x = x.repeat(self.window,1,1)
        y=torch.eye(self.adj.shape[1]).float()
        y = y.repeat(self.pred_window,1,1)
        adj_y=torch.from_numpy(self.adj[index+self.window:index+self.window+self.pred_window]).float()
        external=torch.from_numpy(self.ext[index+self.window:index+self.window+self.pred_window]).float()
        return x,adj,y,adj_y,external
num_workers = 8 if cuda else 0
batch_size=32
# Training
train_dataset = MyDataset(train,ext_train)
train_loader_args = dict(shuffle=True, batch_size=batch_size, num_workers=num_workers, pin_memory=True)
if cuda  else dict(shuffle=True, batch_size=batch_size)
train_loader = data.DataLoader(train_dataset, **train_loader_args)
# # Testing
test_dataset = MyDataset(test,ext_test)
test_loader_args = dict(shuffle=False, batch_size=batch_size, num_workers=num_workers, pin_memory=True)
if cuda else dict(shuffle=False, batch_size=batch_size)
test_loader = data.DataLoader(test_dataset, **test_loader_args)
```

Encoder:

```
class Graph_embedding(torch.nn.Module):
    '''
    Graph Conv to get embeddings

    '''

    def __init__(self, in_channels, out_channels):
        super(Graph_embedding, self).__init__()
        self.conv1 = DenseGCNConv(in_channels, out_channels, bias=True)  #in_channel=B*N*N
        self.conv2 = DenseGCNConv(out_channels, in_channels,  bias=True)

    def forward(self, x, adj, add_loop=True):
        x=self.conv1(x, adj)
        x = F.relu(x)
        return x

class Encoder(torch.nn.Module):
    '''
    Encode history of demand graphs

    '''
    def __init__(self, in_channels, out_channels):
        super(Encoder, self).__init__()
        self.graph_emb=Graph_embedding(in_channels=1,out_channels=hidden_dim)
        self.Conv2D=nn.Conv2d(1, 1, kernel_size=16, padding=1,dilation=10) #input_size=270
        self.Conv2D2=nn.Conv2d(1, 1, kernel_size=16, padding=1,dilation=5)
        self.Linear=nn.Linear(2401,512)
        self.lstm = nn.LSTM(input_size=512, hidden_size=1024)

    def forward(self, x, adj,):
        batch_size,seq_len=adj.shape[0],adj.shape[1]
        E=[] #Graph Embeddings
        for i in range(seq_len):
          e=self.graph_emb(x[:,i,:,:],adj[:,i,:,:])
          e=self.Conv2D(e.unsqueeze(1))
          e=self.Conv2D2(e)
          e=torch.flatten(e,start_dim=1)
          e=self.Linear(e)
          E.append(e)
        E=torch.stack(E)
        output_lstm,latent =self.lstm(E)
        return output_lstm,latent
```

Decoder:

```
class Decoder(nn.Module):
  def __init__(self, in_channels, out_channels, nodes=270,   isAttended=False):
    super(Decoder, self).__init__()
    self.graph_emb=Graph_embedding(in_channels=1,out_channels=hidden_dim)
    self.Conv2D=nn.Conv2d(1, 1, kernel_size=16, padding=1,dilation=10) #input_size=270
    self.Conv2D2=nn.Conv2d(1, 1, kernel_size=16, padding=1,dilation=5)
    self.linearg=nn.Linear(2401,512)
    self.lstm1 = nn.LSTMCell(input_size=512, hidden_size=1024)
    self.lstm2 = nn.LSTMCell(input_size=1024, hidden_size=1024)
    self.isAttended = isAttended
    if(isAttended):
      self.attention = Attention()
    self.linear1 = nn.Linear(1024,2048)
    self.linear2 = nn.Linear(2048+60,nodes*nodes)
    self.linear3 = nn.Linear(4096,nodes*nodes)
    self.relu=nn.ReLU()

  def forward(self, latent, external,y=None,adj_y=None, train=True,teacher_forcing_rate=0.8):
    '''
    :param key :(T,N,key_size)=(time/seq_len,batch,key_size) Output of the Encoder Key projection layer
    :param values: (T,N,value_size) Output of the Encoder Value projection layer
    :param text: (N,seq_len) Batch input of seq with text_length
    :param train: Train or eval mode
    :return predictions: Returns the character perdiction probability
    '''

    batch_size = y.shape[0]
    if(train):
      max_len =  y.shape[1]
    else:
      max_len = 3
    predictions_list = []
    hidden_states = [None, None]
    prediction = torch.zeros(batch_size,270*270).to(device)
    for i in range(max_len):
      '''
      Here you should implement Gumble noise and teacher forcing techniques
      '''
      if (train):
        if y ==None:
            teacher_forcing_rate=0
        if i==0:
            teacher_forcing_rate=1
        teacher_forcing = True if random.random() < teacher_forcing_rate else False
        if teacher_forcing:
          dec_input_x,dec_input_adj=y[:,i,:,:],adj_y[:,i,:,:]
        else:
          print("i",i,np.shape(prediction))
          dec_input_x=torch.eye(self.adj_y.shape[1]).float()
          dec_input_x= dec_input_x.repeat(batch_size,1,1).to(device)
          dec_input_adj=prediction
      else:
        dec_input_x=torch.eye(self.adj_y.shape[1]).float()
        dec_input_x= dec_input_x.repeat(batch_size,1,1).to(device)
        dec_input_adj=prediction
      if i ==0:
        latent_hidden,latent_out=latent[0].squeeze(0),latent[1].squeeze(0).to(device)
      e=self.graph_emb(dec_input_x,dec_input_adj)
      e=self.Conv2D(e.unsqueeze(1))
      e=self.Conv2D2(e)
      e=torch.flatten(e,start_dim=1)
      e=self.linearg(e)
      hidden_states[0] = self.lstm1(e,(latent_hidden,latent_out))
      inp_2 = hidden_states[0][0]
      hidden_states[1] = self.lstm2(inp_2,hidden_states[1])
      latent_hidden,latent_out = hidden_states[1][0],hidden_states[1][1]
      output=latent_out
      output=self.linear1(output)
      output=F.dropout(output, p=0.2, training=True)
      output=torch.cat([output,external.reshape(batch_size,-1)],dim=1)
      output=self.linear2(output)
      prediction=self.relu(output)
      predictions_list.append(prediction.unsqueeze(1))
    predictions_list=torch.stack(predictions_list).to(device)
    return predictions_list
```

Seq2seq:

```
class Seq2Seq(nn.Module):
  def __init__(self, input_size, hidden_dim, value_size=270, key_size=270, isAttended=False):
    super(Seq2Seq, self).__init__()

    self.encoder = Encoder(in_channels=input_size, out_channels=hidden_dim)
    self.decoder = Decoder(in_channels=input_size, out_channels=hidden_dim)
  def forward(self, x, adj, external=None, y=None, adj_y=None, train=True):
    _, latent= self.encoder(x, adj)
    if(train):
      predictions = self.decoder(latent, external, y, adj_y)
    else:
      predictions = self.decoder(latent, external, y=None, train=False)
    return predictions
```

Training function:

```
def train(model, train_loader, criterion, criterion2, criterion3, optimizerr, i=0):
  model.train()
  loss_RMSE = 0
  loss_L1_smooth=0
  loss_sMAPE=0
  start_time = time.time()
  for batch_num, (x, adj, y, adj_y, external) in enumerate(train_loader):
      x = x.to(device) #(batch, N=nodes, feature_size=1)
      adj = adj.to(device) #(batch, N, N)
      y=y.to(device)
      adj_y=adj_y.to(device)
      external=external.to(device)
      adj_hat = model(x, adj, external, y, adj_y).to(device)
      adj_y=torch.flatten(adj_y, start_dim=2).squeeze(2).to(device)
      adj_hat=adj_hat.squeeze(2)
      adj_hat=adj_hat.permute(1,0,2).to(device)
      mask=(adj_y!=0).float()
      adj_y=adj_y*mask
      adj_hat=adj_hat*mask
      loss=torch.sqrt(criterion(adj_y, adj_hat)).to(device)
      loss2=criterion2(adj_y, adj_hat).to(device)
      loss3=criterion3(adj_hat, adj_y).to(device)
      loss.backward()
      optimizer.step()
      loss_RMSE+=loss.item()
      loss_L1_smooth+=loss2.item()
      loss_sMAPE+=loss3.item()
      del adj_y, adj_hat, loss, loss2, loss3, x, y, external
  loss_RMSE /= len(train_loader)
  loss_L1_smooth/= len(train_loader)
  loss_sMAPE/= len(train_loader)
  end_time=time.time()
  print('Epoch', i+1, ' Training RMSELoss: ', loss_RMSE, ' Training loss_L1_smooth: ', loss_L1_smooth,
        ' Training loss_sMAPE: ', loss_sMAPE, 'Time: ', end_time - start_time, 's')
  return loss_RMSE, loss_L1_smooth, loss_sMAPE
```

Validation function:

```
def val(model,val_loader,  criterion,criterion2,criterion3, optimizer,i=0):
  with torch.no_grad():
    model.eval()
    loss_RMSE = 0
    loss_L1_smooth=0
    loss_sMAPE=0
    start_time = time.time()
    output=[]
    for batch_num, (x,adj,y,adj_y,external) in enumerate(train_loader):
        x = x.to(device) #(batch,N=nodes,feature_size=1)
        adj = adj.to(device) #(batch,N,N)
        y=y.to(device)
        adj_y=adj_y.to(device)
        external=external.to(device)
        adj_hat = model(x, adj,external,y,adj_y).to(device)
        adj_y=torch.flatten(adj_y,start_dim=2).squeeze(2).to(device)
        adj_hat=adj_hat.squeeze(2)
        adj_hat=adj_hat.permute(1,0,2).to(device)
        mask=(adj_y!=0).float()
        adj_y=adj_y*mask
        adj_hat=adj_hat*mask
        loss=torch.sqrt(criterion(adj_y,adj_hat)).to(device)
        loss2=criterion2(adj_y,adj_hat).to(device)
        loss3=criterion3(adj_hat,adj_y).to(device)
        loss_RMSE+=loss.item()
        loss_L1_smooth+=loss2.item()
        loss_sMAPE+=loss3.item()
        output.append(adj_hat)
        del adj_y, adj_hat, loss, loss2, loss3, x, y
    loss_RMSE /= len(val_loader)
    loss_L1_smooth/= len(val_loader)
    loss_sMAPE/= len(val_loader)
    end_time=time.time()
    print('Epoch',i+1,' Testing RMSELoss: ', loss_RMSE,' Testing loss_L1_smooth: ', loss_L1_smooth,
          ' Testing loss_sMAPE: ', loss_sMAPE, 'Time: ',end_time - start_time, 's')
  return loss_RMSE,loss_L1_smooth,loss_sMAPE,output
```

## Appendix D Stock Price Prediction

Data loading:

```
class MyDataset(data.Dataset):
    def __init__(self, X, window=50,f=1):
        self.X = X
      self.window=window
        self.f=f

    def __len__(self):
        return len(self.X)-2*self.window

    def __getitem__(self,index):
        X = torch.from_numpy(self.X[index:index+self.window,-12:]).float()
        X_ext=torch.from_numpy(self.X[index+self.window,:-12]).float()
        Y = torch.from_numpy(self.X[index+self.window:index+self.window+self.f,-12:]).float()
        Y_2=torch.from_numpy(self.X[index+self.window+self.f,-12:]).float()
        return X,Y,X_ext,Y_2
num_workers = 8 if cuda else 0
batch_size=32
# Training
train_dataset = MyDataset(train)
train_loader_args = dict(shuffle=True, batch_size=batch_size, num_workers=num_workers,pin_memory=True,
            drop_last=True) if cuda else dict(shuffle=True, batch_size=batch_size)
train_loader = data.DataLoader(train_dataset, **train_loader_args)
# Testing
test_dataset = MyDataset(test)
test_loader_args = dict(shuffle=False, batch_size=batch_size, num_workers=num_workers, pin_memory=True)
if cuda else dict(shuffle=False, batch_size=batch_size)
test_loader = data.DataLoader(test_dataset, **test_loader_args)
```

Auto-encoder:

```
class Encoder(nn.Module):
  def __init__(self, seq_len, n_features, embedding_dim=64):
    super(Encoder, self).__init__()
    self.seq_len, self.n_features = seq_len, n_features
    self.embedding_dim, self.hidden_dim = embedding_dim, 2 * embedding_dim
    self.encoder1 = nn.Linear(in_features=self.n_features,
                             out_features=self.hidden_dim)
    self.activation1 = nn.ReLU(self.hidden_dim)
    self.encoder2 = nn.Linear(in_features=self.hidden_dim,
                             out_features=self.embedding_dim)
    self.activation2 = nn.ReLU(self.embedding_dim)
    self.encoder3 = nn.Linear(in_features=self.embedding_dim,
                             out_features=128)
    self.activation3 = nn.ReLU(self.embedding_dim/2)
    self.encoder4 = nn.Linear(in_features=128,
                             out_features=64)
    self.activation4 = nn.ReLU(self.embedding_dim/4)
    self.encoder5 = nn.Linear(in_features=64,
                             out_features=32)
    self.activation5 = nn.ReLU(self.embedding_dim/8)

  def forward(self, x):

    x = self.encoder1(x)
    x = self.activation1(x)
    x = self.encoder2(x)
    x = self.activation2(x)
    x = self.encoder3(x)
    x = self.activation3(x)
    x = self.encoder4(x)
    x = self.activation4(x)
    x = self.encoder5(x)
    x = self.activation5(x)
    return x

  def __init__(self, seq_len, input_dim=64, n_features=1):
    super(Decoder, self).__init__()
    self.seq_len, self.input_dim = seq_len, input_dim
    self.hidden_dim, self.n_features = 2 * input_dim, n_features
    self.decoder1 = nn.Linear(in_features=32,
                             out_features=64)
    self.activation1 = nn.ReLU(self.input_dim/4)
    self.decoder2 = nn.Linear(in_features=64,
                             out_features=128)
    self.activation2 = nn.ReLU(self.input_dim/2)
    self.decoder3 = nn.Linear(in_features=128,
                             out_features=self.input_dim)
    self.activation3 = nn.ReLU(self.input_dim)
    self.decoder4 = nn.Linear(in_features=self.input_dim,
                             out_features=self.input_dim)
    self.activation4 = nn.ReLU(self.input_dim)
    self.decoder5 = nn.Linear(in_features=self.input_dim,
                             out_features=self.hidden_dim)
    self.activation5 = nn.ReLU(self.hidden_dim)
    self.output_layer = nn.Linear(in_features=self.hidden_dim,
                                  out_features=self.n_features)

  def forward(self, x):
    x = self.decoder1(x)
    x = self.activation1(x)
    x = self.decoder2(x)
    x = self.activation2(x)
    x = self.decoder3(x)
    x = self.activation3(x)
    x = self.decoder4(x)
    x = self.activation4(x)
    x = self.decoder5(x)
    x = self.activation5(x)
    return self.output_layer(x)
```

Encoder:

```
class Encoders(nn.Module):
  def __init__(self, input_dim, hidden_dim, value_size=256,key_size=256):
    super(Encoders, self).__init__()
    inpu_size=12
    self.tcn = TCN(input_size=input_dim,output_size=hidden_dim,
             num_channels=[32, inpu_size, 50],kernel_size=2,dropout=0.1)
    self.key_network = nn.Linear(50, value_size)
    self.value_network = nn.Linear(50, key_size)

  def forward(self,x):
    x = x.permute(0,2,1)
    x = self.tcn(x)
    keys = self.key_network(x.permute(0,2,1))
    value = self.value_network(x.permute(0,2,1))
    return keys, value
```

Decoder:

```
class Decoders(nn.Module):
  def __init__(self, input_dim, hidden_dim, value_size=256, key_size=256,  isAttended=True):
    super(Decoders, self).__init__()
    self.linear1=nn.Linear(input_dim,hidden_dim)
    self.lstm1 = nn.LSTMCell(input_size=hidden_dim+value_size, hidden_size=hidden_dim)
    self.lstm2 = nn.LSTMCell(input_size=hidden_dim, hidden_size=key_size)
    self.isAttended = isAttended
    if(isAttended):
      self.attention = Attention()
    self.conv=nn.Sequential(nn.Conv1d(in_channels=1, out_channels=1,kernel_size=3,stride=1),
             nn.BatchNorm1d(num_features=1), nn.ReLU(inplace=True))
    self.inp = ((key_size+value_size+32)-3)//1+1
    self.mlp = nn.Linear(self.inp,input_dim) # original

  def forward(self, keys, values,x_ext, y=None, train=True,teacher_forcing_rate=0.8):
    '''
    :param key :(T,N,key_size)=(time/seq_len,batch,key_size) Output of the Encoder Key projection layer
    :param values: (T,N,value_size) Output of the Encoder Value projection layer
    :param text: (N,seq_len) Batch input of text with seq_length
    :param train: Train or eval mode
    :return predictions: Returns the character perdiction probability
    '''
    keys=keys.permute(1,0,2) #seq_len,batch,keys_size
    values=values.permute(1,0,2) #seq_len,batch,values_size     batch_size = keys.shape[1]
    if(train):
      max_len =  y.shape[1]
      out1 = self.linear1(y)
    else:
      max_len = 50
    predictions_list = []
    hidden_states = [None, None]#, None]
    prediction = torch.zeros(batch_size,1).to(device)

    for i in range(max_len):
      '''
      teacher forcing technique
      '''
      if(train):
        if y ==None:
            teacher_forcing_rate=0
        if i==0:
            teacher_forcing_rate=1

        teacher_forcing = True if random.random() < teacher_forcing_rate else False
        if teacher_forcing:
            downsampled_input = out1[:,i-1,:]
        else:
            print("i",i,np.shape(prediction))
            downsampled_input=self.linear1(prediction)

      else:
        downsampled_input = self.linear1(prediction)

      if i ==0:
        context=torch.zeros(batch_size,values.size(-1)).to(device)
      inp = torch.cat([downsampled_input,context], dim=1)
      hidden_states[0] = self.lstm1(inp,hidden_states[0])
      inp_2 = hidden_states[0][0]
      hidden_states[1] = self.lstm2(inp_2,hidden_states[1])
      output = hidden_states[1][0]
      query=output
      value=values[:,:,:]
      if (self.isAttended):
        context,attention=self.attention(query, keys, value)
      else:
        context=value[-1,:,:]
      prediction = self.conv(torch.cat([output, context,x_ext], dim=1).unsqueeze(1))
      prediction = self.mlp(prediction.squeeze())
      predictions_list.append(prediction.unsqueeze(1))
    return torch.cat(predictions_list, dim=1)
```

## Seq2seq

```
class Seq2Seq(nn.Module):
  def __init__(self,input_dim,hidden_dim,value_size=256, key_size=256,isAttended=False):
    super(Seq2Seq,self).__init__()

    self.encoders = Encoders(input_dim, hidden_dim)
    self.decoders = Decoders(input_dim, hidden_dim)
  def forward(self,x,x_ext, y=None,train=True):
    key, value = self.encoders(x)
    if(train):
      predictions = self.decoders(key, value,x_ext, y)
    else:
      predictions = self.decoders(key, value,x_ext, y=None, train=False)
    return predictions
```

## Training function:

```
def train(model,train_loader, criterion,criterion2,criterion3, optimizer,i,extractor):
  model.train()
  loss_RMSE = 0
  loss_L1_smooth=0
  loss_sMAPE=0
  start_time = time.time()
  for batch_num, (x,y,X_ext,y_2) in enumerate(train_loader):
      torch.autograd.set_detect_anomaly(True)
      x = x.to(device) #(batch,seq_len=130,input_dim=522)
      y = y.to(device) #(batch,seq_len=130,input_dim=522)
      X_ext = X_ext.to(device)
      _,latent = extractor(X_ext)
      predictions = model(x,latent,y)
      loss = torch.sqrt(criterion(predictions[:,:,-6], y[:,:,-6]))
      loss2 = criterion2(predictions[:,:,-6], y[:,:,-6])
      loss3 = criterion3(predictions[:,:,-6], y[:,:,-6])
      loss.backward()
      optimizer.step()
      loss_RMSE+=loss.item()
      loss_L1_smooth+=loss2.item()
      loss_sMAPE+=loss3.item()
      del predictions
      del loss
      del x,y
  loss_RMSE /= len(train_loader)
  loss_L1_smooth/= len(train_loader)
  loss_sMAPE/= len(train_loader)
  end_time=time.time()
  print('Epoch',i+1,' Training RMSELoss: ', loss_RMSE,' Training loss_L1_smooth: ',
        loss_L1_smooth,' Training loss_sMAPE: ', loss_sMAPE, 'Time: ',end_time - start_time, 's')
  return loss_RMSE,loss_L1_smooth,loss_sMAPE
```

Validation function:

```
def val(model,val_loader , criterion ,criterion2 ,criterion3 , optimizer ,i ,extractor ):
  with torch.no_grad():
    model.eval()
    loss_RMSE = 0
    loss_L1_smooth=0
    loss_sMAPE=0
    start_time = time.time()
    output=[]
    for batch_num, (x,y,X_ext,y_2) in enumerate(val_loader):
    # for batch_num, (x,y,X_ext,X_ext_2d,y_2) in enumerate(val_loader):

        x = x.to(device) #(batch,seq_len=130,input_dim=522)
        y = y.to(device) #(batch,seq_len=130,input_dim=522)
        X_ext = X_ext.to(device)
        _,latent = extractor(X_ext)
        predictions = model(x,latent,y,train=True)
        loss = torch.sqrt(criterion(predictions[:,:,-6], y[:,:,-6]))
        loss2 = criterion2(predictions[:,:,-6], y[:,:,-6])
        loss3 = criterion3(predictions[:,:,-6], y[:,:,-6])
        output.append(predictions)
        loss_RMSE+=loss.item()
        loss_L1_smooth+=loss2.item()
        loss_sMAPE+=loss3.item()
        del predictions
        del loss
        del x,y
    loss_RMSE /= len(val_loader)
    loss_L1_smooth/= len(val_loader)
    loss_sMAPE/= len(val_loader)
    end_time=time.time()
    print('Epoch',i+1,'_Val_RMSELoss:_', loss_RMSE,'_Val_loss_L1_smooth:_', loss_L1_smooth,
    '_Val_loss_sMAPE:_', loss_sMAPE, 'Time:_',end_time - start_time, 's')
    print("="*20)
    return loss_RMSE,loss_L1_smooth,loss_sMAPE,output
```

# Bibliography

[1] Cs224w: Machine learning with graphs stanford. http://web.stanford.edu/class/cs224w/slides/01-intro.pdf. Accessed: 2022-02-20.

[2] Cs224w: Machine learning with graphs stanford. http://web.stanford.edu/class/cs224w/slides/06-GNN1.pdf. Accessed: 2022-02-20.

[3] Cs224w: Machine learning with graphs stanford. http://web.stanford.edu/class/cs224w/slides/01-intro.pdf. Accessed: 2022-02-20.

[4] Deep learning (part 1) - feedforward neural networks (fnn). https://training.galaxyproject.org/training-material/topics/statistics/tutorials/FNN/tutorial.html. Accessed: 2022-04-23.

[5] Encoder-decoder seq2seq models, clearly explained!! https://medium.com/analytics-vidhya/encoder-decoder-seq2seq-models-clearly-explained-c34186fbf49b. Accessed: 2022-04-23.

[6] Taxi and ridehailing usage in new york city. https://toddwschneider.com/dashboards/nyc-taxi-ridehailing-uber-lyft-data/. Accessed: 2021-11-11.

[7] Andrés Arévalo, Jaime Niño, German Hernández, and Javier Sandoval. High-frequency trading strategy based on deep neural networks. In *International conference on intelligent computing*, pages 424–436. Springer, 2016.

[8] Vijay Badrinarayanan, Alex Kendall, and Roberto Cipolla. Segnet: A deep convolutional encoder-decoder architecture for image segmentation. *IEEE transactions on pattern analysis and machine intelligence*, 39(12):2481–2495, 2017.

[9] Shaojie Bai, J. Zico Kolter, and Vladlen Koltun. An empirical evaluation of generic convolutional and recurrent networks for sequence modeling, 2018.

[10] Michel Ballings, Dirk Van den Poel, Nathalie Hespeels, and Ruben Gryp. Evaluating multiple classifiers for stock price direction prediction. *Expert Systems with Applications*, 42(20):7046–7056, 2015.

[11] George EP Box and David A Pierce. Distribution of residual autocorrelations in autoregressive-integrated moving average time series models. *Journal of the American statistical Association*, 65(332):1509–1526, 1970.

[12] Yahya Eru Cakra and Bayu Distiawan Trisedya. Stock price prediction using linear regression based on sentiment analysis. In *2015 international conference on advanced computer science and information systems (ICACSIS)*, pages 147–154. IEEE, 2015.

[13] William Chan, Navdeep Jaitly, Quoc Le, and Oriol Vinyals. Listen, attend and spell: A neural network for large vocabulary conversational speech recognition. In *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4960–4964. IEEE, 2016.

[14] Gang Chen. A gentle tutorial of recurrent neural network with error backpropagation. *CoRR*, abs/1610.02583, 2016.

[15] Long Chen, Piyushimita Vonu Thakuriah, and Konstantinos Ampountolas. Short-term prediction of demand for ride-hailing services: A deep learning approach. *Journal of Big Data Analytics in Transportation*, pages 1–21, 2021.

[16] Michele Cocca, Douglas Teixeira, Luca Vassio, Marco Mellia, Jussara M Almeida, and Ana Paula Couto da Silva. On car-sharing usage prediction with open socio-demographic data. *Electronics*, 9(1):72, 2020.

[17] Boyd Cohen and Pablo Muñoz. Sharing cities and sustainable consumption and production: towards an integrated framework. *Journal of cleaner production*, 134:87–97, 2016.

[18] Shom Prasad Das and Sudarsan Padhy. Support vector machines for prediction of futures prices in indian stock market. *International Journal of Computer Applications*, 41(3), 2012.

[19] Dingxiong Deng, Cyrus Shahabi, Ugur Demiryurek, Linhong Zhu, Rose Yu, and Yan Liu. Latent space model for road networks to predict time-varying traffic. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1525–1534, 2016.

[20] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning. *arXiv preprint arXiv:1603.07285*, 2016.

[21] Michael Dusenberry, Ghassen Jerfel, Yeming Wen, Yian Ma, Jasper Snoek, Katherine Heller, Balaji Lakshminarayanan, and Dustin Tran. Efficient and scalable bayesian neural nets with rank-1 factors. In *International conference on machine learning*, pages 2782–2792. PMLR, 2020.

[22] Alex Ezeh, Oyinlola Oyebode, David Satterthwaite, Yen-Fu Chen, Robert Ndugwa, Jo Sartori, Blessing Mberu, GJ Melendez-Torres, Tilahun Haregu, Samuel I Watson, et al. The history, geography, and sociology of slums and the health problems of people who live in slums. *The lancet*, 389(10068):547–558, 2017.

[23] Hassan Ismail Fawaz, Germain Forestier, Jonathan Weber, Lhassane Idoumghar, and Pierre-Alain Muller. Deep learning for time series classification: a review. *Data Mining and Knowledge Discovery*, 33(4):917–963, 2019.

[24] SGM Fifield, DM Power, and DGS Knipe. The performance of moving average rules in emerging stock markets. *Applied Financial Economics*, 18(19):1515–1532, 2008.

[25] Yarin Gal and Zoubin Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *international conference on machine learning*, pages 1050–1059, 2016.

[26] Zhenhao Gan, Chaoshun Li, Jianzhong Zhou, and Geng Tang. Temporal convolutional networks interval prediction model for wind speed forecasting. *Electric Power Systems Research*, 191:106865, 2021.

[27] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *International conference on machine learning*, pages 1263–1272. PMLR, 2017.

[28] Vinayaka Gude, Steven Corns, and Suzanna Long. Flood prediction and uncertainty estimation using deep learning. *Water*, 12(3):884, 2020.

[29] Ge Guo and Wei Yuan. Short-term traffic speed forecasting based on graph attention temporal convolutional networks. *Neurocomputing*, 410:387–393, 2020.

[30] Ge Guo and Tianqi Zhang. A residual spatio-temporal architecture for travel demand forecasting. *Transportation Research Part C: Emerging Technologies*, 115:102639, 2020.

[31] Siyu Hao, Der-Horng Lee, and De Zhao. Sequence to sequence learning with attention mechanism for short-term passenger flow prediction in large-scale metro system. *Transportation Research Part C: Emerging Technologies*, 107:287–300, 2019.

[32] José Miguel Hernández-Lobato and Ryan Adams. Probabilistic backpropagation for scalable learning of bayesian neural networks. In *International Conference on Machine Learning*, pages 1861–1869, 2015.

[33] Alexander Hess, Stefan Spinler, and Matthias Winkenbach. Real-time demand forecasting for an urban delivery platform. *Transportation Research Part E: Logistics and Transportation Review*, 145:102147, 2021.

[34] M Hiransha, E Ab Gopalakrishnan, Vijay Krishna Menon, and KP Soman. Nse stock market prediction using deep-learning models. *Procedia computer science*, 132:1351–1362, 2018.

[35] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[36] Kazuki Irie, Rohit Prabhavalkar, Anjuli Kannan, Antoine Bruguier, David Rybach, and Patrick Nguyen. Model unit exploration for sequence-to-sequence speech recognition. *arXiv preprint arXiv:1902.01955*, 2019.

[37] Guangyin Jin, Yan Cui, Liang Zeng, Hanbo Tang, Yanghe Feng, and Jincai Huang. Urban ride-hailing demand prediction with multiple spatio-temporal information fusion network. *Transportation Research Part C: Emerging Technologies*, 117:102665, 2020.

[38] Jintao Ke, Siyuan Feng, Zheng Zhu, Hai Yang, and Jieping Ye. Joint predictions of multi-modal ride-hailing demands: A deep multi-task multi-graph learning-based approach. *Transportation Research Part C: Emerging Technologies*, 127:103063, 2021.

[39] Jintao Ke, Xiaoran Qin, Hai Yang, Zhengfei Zheng, Zheng Zhu, and Jieping Ye. Predicting origin-destination ride-sourcing demand with a spatio-temporal encoder-decoder residual multi-graph convolutional network. *Transportation Research Part C: Emerging Technologies*, 122:102858, 2021.

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

[41] Kaustubh Khare, Omkar Darekar, Prafull Gupta, and VZ Attar. Short term stock price prediction using deep learning. In *2017 2nd IEEE International Conference on Recent Trends in Electronics, Information & Communication Technology (RTEICT)*, pages 482–486. IEEE, 2017.

[42] Yoon Kim. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*, 2014.

[43] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[44] Eleftheria Kontou, Venu Garikapati, and Yi Hou. Reducing ridesourcing empty vehicle travel with future travel demand prediction. *Transportation Research Part C: Emerging Technologies*, 121:102826, 2020.

[45] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[46] Manish Kumar and M Thenmozhi. Forecasting stock index movement: A comparison of support vector machines and random forest. In *Indian institute of capital markets 9th capital markets conference paper*, 2006.

[47] Gaoyang Li, Li Yang, Chi-Guhn Lee, Xiaohua Wang, and Mingzhe Rong. A bayesian deep learning rul framework integrating epistemic and aleatoric uncertainties. *IEEE Transactions on Industrial Electronics*, 2020.

[48] Lingbo Liu, Zhilin Qiu, Guanbin Li, Qing Wang, Wanli Ouyang, and Liang Lin. Contextualized spatial–temporal network for taxi origin-destination demand prediction. *IEEE Transactions on Intelligent Transportation Systems*, 20(10):3875–3887, 2019.

[49] Yongqi Liu, Hui Qin, Zhendong Zhang, Shaoqian Pei, Zhiqiang Jiang, Zhongkai Feng, and Jianzhong Zhou. Probabilistic spatiotemporal wind speed forecasting based on a variational bayesian deep learning model. *Applied Energy*, 260:114259, 2020.

[50] Zhizhen Liu, Hong Chen, Yan Li, and Qi Zhang. Taxi demand prediction based on a combination forecasting model in hotspots. *Journal of Advanced Transportation*, 2020, 2020.

[51] Wen Long, Zhichen Lu, and Lingxiao Cui. Deep learning-based feature engineering for stock price movement prediction. *Knowledge-Based Systems*, 164:163–173, 2019.

[52] Xiaojiao Mao, Chunhua Shen, and Yu-Bin Yang. Image restoration using very deep convolutional encoder-decoder networks with symmetric skip connections. In *Advances in neural information processing systems*, pages 2802–2810, 2016.

[53] Sidra Mehtab and Jaydip Sen. A time series analysis-based stock price prediction using machine learning and deep learning models. *arXiv preprint arXiv:2004.11697*, 2020.

[54] John Miller and Moritz Hardt. When recurrent models don't need to be recurrent. *CoRR*, abs/1805.10369, 2018.

[55] Terence C Mills. Technical analysis and the london stock exchange: Testing trading rules using the ft30. *International Journal of Finance & Economics*, 2(4):319–331, 1997.

[56] Bahman Moghimi, Abolfazl Safikhani, Camille Kamga, Wei Hao, and Jiaqi Ma. Short-term prediction of signal cycle on an arterial with actuated-uncoordinated control using sparse time series models. *IEEE Transactions on Intelligent Transportation Systems*, 20(8):2976–2985, 2018.

[57] George O Mohler, Martin B Short, P Jeffrey Brantingham, Frederic Paik Schoenberg, and George E Tita. Self-exciting point process modeling of crime. *Journal of the American Statistical Association*, 106(493):100–108, 2011.

[58] Luis Moreira-Matias, Joao Gama, Michel Ferreira, Joao Mendes-Moreira, and Luis Damas. Predicting taxi–passenger demand using streaming data. *IEEE Transactions on Intelligent Transportation Systems*, 14(3):1393–1402, 2013.

[59] Alireza Nejadettehad, Hamid Mahini, and Behnam Bahrak. Short-term demand forecasting for online car-hailing services using recurrent neural networks. *Applied Artificial Intelligence*, 34(9):674–689, 2020.

[60] Graham Neubig. Neural machine translation and sequence-to-sequence models: A tutorial. *CoRR*, abs/1703.01619, 2017.

[61] Kun Niu, Chao Wang, Xinjie Zhou, and Tong Zhou. Predicting ride-hailing service demand via rpa-lstm. *IEEE Transactions on Vehicular Technology*, 68(5):4213–4222, 2019.

[62] John Paisley, David Blei, and Michael Jordan. Variational bayesian inference with stochastic search. *arXiv preprint arXiv:1206.6430*, 2012.

[63] Pallavi Pant and Roy M Harrison. Estimation of the contribution of road traffic emissions to particulate matter concentrations from field measurements: a review. *Atmospheric environment*, 77:78–97, 2013.

[64] Nicolas Papernot and Patrick McDaniel. Deep k-nearest neighbors: Towards confident, interpretable and robust deep learning. *arXiv preprint arXiv:1803.04765*, 2018.

[65] Andrija Petrović, Mladen Nikolić, Miloš Jovanović, and Boris Delibašić. Gaussian conditional random fields for classification. *arXiv preprint arXiv:1902.00045*, 2019.

[66] Rohit Prabhavalkar, Kanishka Rao, Tara N Sainath, Bo Li, Leif Johnson, and Navdeep Jaitly. A comparison of sequence-to-sequence models for speech recognition. In *Interspeech*, pages 939–943, 2017.

[67] M. Puheim and L. Madarász. Normalization of inputs and outputs of neural network based robotic arm controller in role of inverse kinematic model. In *2014 IEEE 12th International Symposium on Applied Machine Intelligence and Informatics (SAMI)*, pages 85–89, 2014.

[68] Xinwu Qian, Satish V Ukkusuri, Chao Yang, and Fenfan Yan. Short-term demand forecasting for on-demand mobility service. *IEEE Transactions on Intelligent Transportation Systems*, 2020.

[69] Markus Ringnér. What is principal component analysis? *Nature biotechnology*, 26(3):303–304, 2008.

[70] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[71] David E Rumelhart, Richard Durbin, Richard Golden, and Yves Chauvin. Backpropagation: The basic theory. *Backpropagation: Theory, architectures and applications*, pages 1–34, 1995.

[72] Sreelekshmy Selvin, R Vinayakumar, EA Gopalakrishnan, Vijay Krishna Menon, and KP Soman. Stock price prediction using lstm, rnn and cnn-sliding window model. In *2017 international conference on advances in computing, communications and informatics (icacci)*, pages 1643–1647. IEEE, 2017.

[73] Kiam Tian Seow, Nam Hai Dang, and Der-Horng Lee. A collaborative multiagent taxi-dispatch system. *IEEE Transactions on Automation science and engineering*, 7(3):607–616, 2009.

[74] Alex Sherstinsky. Fundamentals of recurrent neural network (RNN) and long short-term memory (LSTM) network. *CoRR*, abs/1808.03314, 2018.

[75] Nitish Srivastava. Improving neural networks with dropout. *University of Toronto*, 182(566):7, 2013.

[76] Yuxing Sun, Biao Leng, and Wei Guan. A novel wavelet-svm short-time passenger flow prediction in beijing subway system. *Neurocomputing*, 166:109–121, 2015.

[77] Md Arif Istiake Sunny, Mirza Mohd Shahriar Maswood, and Abdullah G Alharbi. Deep learning-based stock price prediction using lstm and bi-directional lstm model. In *2020 2nd Novel Intelligent and Leading Emerging Sciences Conference (NILES)*, pages 87–92. IEEE, 2020.

[78] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[79] Jinjun Tang, Jian Liang, Fang Liu, Jingjing Hao, and Yinhai Wang. Multi-community passenger demand prediction at region level based on spatio-temporal graph convolutional network. *Transportation Research Part C: Emerging Technologies*, 124:102951, 2021.

[80] Yongxin Tong, Yuqiang Chen, Zimu Zhou, Lei Chen, Jie Wang, Qiang Yang, Jieping Ye, and Weifeng Lv. The simpler the better: a unified approach to predicting original taxi demands based on large-scale online platforms. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1653–1662, 2017.

[81] Aaron Van Oord, Nal Kalchbrenner, and Koray Kavukcuoglu. Pixel recurrent neural networks. In *International conference on machine learning*, pages 1747–1756. PMLR, 2016.

[82] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

[83] Dongjie Wang, Yan Yang, and Shangming Ning. Deepstcl: A deep spatio-temporal convlstm for travel demand prediction. In *2018 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2018.

[84] Senzhang Wang, Jiannong Cao, and Philip Yu. Deep learning for spatio-temporal data mining: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 2020.

[85] Xing Wang, Yijun Wang, Bin Weng, and Aleksandr Vinel. Stock2vec: A hybrid deep learning framework for stock market prediction with representation learning and temporal convolutional network. *arXiv preprint arXiv:2010.01197*, 2020.

[86] Matthew R Williams and Terrance D Savitsky. Uncertainty estimation for pseudo-bayesian inference under complex sampling. *International Statistical Review*, 89(1):72–107, 2021.

[87] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[88] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems*, 32(1):4–24, 2020.

[89] Jun Xu, Rouhollah Rahmatizadeh, Ladislau Bölöni, and Damla Turgut. Real-time prediction of taxi demand using recurrent neural networks. *IEEE Transactions on Intelligent Transportation Systems*, 19(8):2572–2581, 2017.

[90] Xiang Yan, Xinyu Liu, and Xilei Zhao. Using machine learning for direct demand modeling of ridesourcing services in chicago. *Journal of Transport Geography*, 83:102661, 2020.

[91] Jianbo Yang, Minh Nhut Nguyen, Phyo Phyo San, Xiao Li Li, and Shonali Krishnaswamy. Deep convolutional neural networks on multichannel time series for human activity recognition. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

[92] Huaxiu Yao, Fei Wu, Jintao Ke, Xianfeng Tang, Yitian Jia, Siyu Lu, Pinghua Gong, Jieping Ye, and Zhenhui Li. Deep multi-view spatial-temporal network for taxi demand prediction. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

[93] Xueyan Yin, Genze Wu, Jinze Wei, Yanming Shen, Heng Qi, and Baocai Yin. A comprehensive survey on traffic prediction. *arXiv preprint arXiv:2004.08555*, 2020.

[94] Pengfei Yu and Xuesong Yan. Stock price prediction based on deep neural networks. *Neural Computing and Applications*, 32(6):1609–1628, 2020.

[95] Nicholas Jing Yuan, Yu Zheng, Liuhang Zhang, and Xing Xie. T-finder: A recommender system for finding passengers and vacant taxis. *IEEE Transactions on knowledge and data engineering*, 25(10):2390–2403, 2012.

[96] Chizhan Zhang, Fenghua Zhu, Yisheng Lv, Peijun Ye, and Fei-Yue Wang. Mlrnn: Taxi demand prediction based on multi-level deep learning and regional heterogeneity analysis. *IEEE Transactions on Intelligent Transportation Systems*, 2021.

[97] Dapeng Zhang, Feng Xiao, Minyu Shen, and Shaopeng Zhong. Dneat: A novel dynamic node-edge attention network for origin-destination demand prediction. *Transportation Research Part C: Emerging Technologies*, 122:102851, 2021.

[98] G Peter Zhang. Time series forecasting using a hybrid arima and neural network model. *Neurocomputing*, 50:159–175, 2003.

[99] Yiling Zhang, Yan Yang, Wei Zhou, Hao Wang, and Xiaocao Ouyang. Multi-city traffic flow forecasting via multi-task learning. *Applied Intelligence*, pages 1–19, 2021.

[100] Feifei Zhao, Weiping Wang, Huijun Sun, Hongming Yang, and Jianjun Wu. Station-level short-term demand forecast of carsharing system via station-embedding-based hybrid neural network. *Transportmetrica B: Transport Dynamics*, pages 1–19, 2021.

[101] Yu Zhao, Rennong Yang, Guillaume Chevalier, Ximeng Xu, and Zhenxing Zhang. Deep residual bidir-lstm for human activity recognition using wearable sensors. *Mathematical Problems in Engineering*, 2018, 2018.

[102] Lingxue Zhu and Nikolay Laptev. Deep and confident prediction for time series at uber. In *2017 IEEE International Conference on Data Mining Workshops (ICDMW)*, pages 103–110. IEEE, 2017.

[103] Xiexin Zou, Shiyao Zhang, Chenhan Zhang, JQ James, and Edward Chung. Long-term origin-destination demand prediction with graph deep learning. *IEEE Transactions on Big Data*, 2021.