Eötvös Loránd University

Faculty of Sciences

Pál Szeiler

Traffic prediction with embeddings

Thesis
Applied Mathematician MSc

Internal supervisor:
András Benczúr
ELKH SZTAKI

External supervisor:
Ferenc Béres
ELKH SZTAKI

Budapest, 2023

Contents

Acknowledgements

The research in this Thesis was supported by the European Union project RRF-2.3.1-21-2022-00004 within the framework of the Artificial Intelligence National Laboratory Grant no RRF-2.3.1-21-2022-00004.

1 Introduction

Forecasting traffic flow is a critical task in transportation and community transport. It involves predicting traffic speed in a road network using historical data. To do this, sensors measure and record the traffic speed on the chosen roads. However, this task is challenging because the sensors' dependencies cannot be explained only by their relative positions in the Euclidean space. Moreover, traffic speeds highly depend on the day and time. Over the weekend, there is no clearly identifiable period during the day when traffic would significantly increase. On weekdays, there are recurring events, such as rush hours, resulting in lower traffic speeds measured by sensors in the morning and evening and higher speeds during the night or midday. Figure 1 shows the speeds measured by different sensors in the PEMS-BAY dataset over a three-week span and highlights the difference between weekend and weekday traffic flow.

Research on traffic prediction has received significant attention in recent years. To address the abovementioned challenges, most models aim to

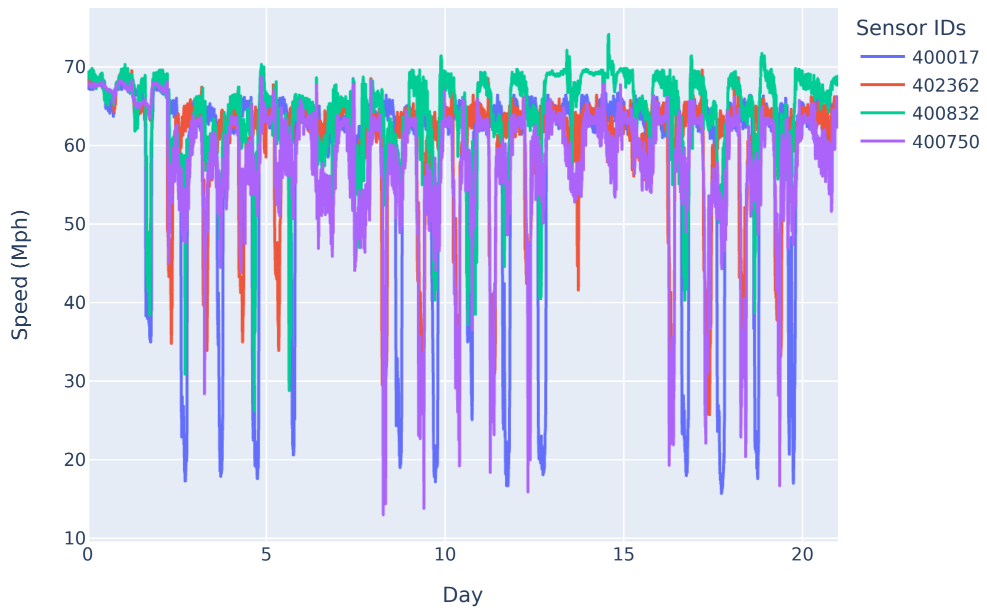


Figure 1: The measurements of different sensors during a 3-week span in the PEMS-BAY dataset

capture the interdependencies between sensors and utilize methods to identify weekly and daily patterns in the data. DCRNN [11], which leverages diffusion-convolution layers and recurrent layers to capture spatial information between sensors, was one of the pioneering models. Since then, many models have been developed[1], like MegaCRN [8], mostly based on graph neural networks (GNN).

In our experiments, we employed attributed and structural graph embeddings to capture node relationships. One of the advantages of using this form of graph representation is that the embedding can be trained in an unsupervised manner, even when node attributes are assigned.

1.1 The traffic flow prediction task

From the mathematical viewpoint, our task is to forecast a multivariate time series using measurements from multiple sensors. Let's denote the number of sensors by n, the number of features at each sensor by f, and the historical and prediction window by h and p, respectively. The problem formulation is as follows:

$$(G, X_{t-h+1}, \ldots, X_t) \to (X_{t+1}, \ldots, X_{t+p}).$$

Here, $X_t \in R^{n \times f}$ represents the features measured by each sensor at time t. G denotes the sensor graph. There are more ways to obtain useful information about the graph. Our main approach is graph embeddings:

$$G \to Z \in R^{n \times d}.$$

Here, each node in the graph corresponds to a d dimensional vector. Concatenating the vector representations to the features of the nodes, we get a new formulation of the problem:

$$(X^Z_{t-h+1}, \ldots, X^Z_t) \to (X^Z_{t+1}, \ldots, X^Z_{t+p}),$$

where $X^Z_t \in R^{n \times (f+d)}$. Since the embedding of network nodes, that are, in our case, the sensors, is stational in time, most of the features in X^Z_t stay the same for a given node.

We will use different types of neural networks, like a feedforward neural network, like Figure 2 to predict the traffic flow. In Section 3 we describe some of the embeddings we use to extract network information and we present our results in Section 5.

[1]https://paperswithcode.com/sota/traffic-prediction-on-metr-la

2 Neural networks

2.1 Multilayer Perceptron model

The multilayer perceptron (MLP) model is a type of neural network that has an input layer, one or more hidden layers, and an output layer. Each layer is composed of neurons, which are the basic processing units of the network. The layers are fully connected or dense, which means that each neuron in a layer is connected to all neurons in the next layer.

During training, the weights of the connections between neurons are adjusted in order to improve the network's performance. The weight matrix of the hidden layer i is denoted as W_i, the input of layer i is denoted as $x^{(i)}$, and the output of layer i is denoted as $y^{(i)}$. The output of layer i is computed as the application of the non-linear activation function f to the matrix multiplication of the weight matrix and the input vector of layer i. The output of layer i is then used as the input of layer $i+1$:

$$y^{(i)} = f(W_i \times x^{(i)} + b_i), \quad \text{and} \quad x^{(i+1)} = y^{(i)}.$$

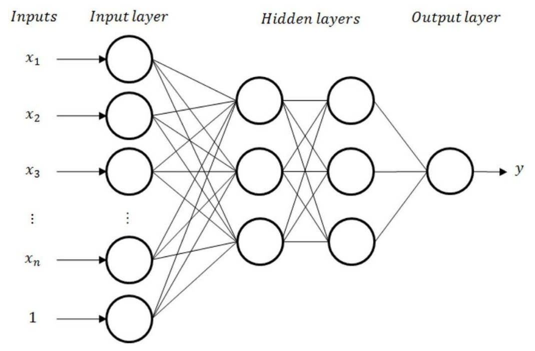


Figure 2: An MLP model with two hidden layers [21]

2.2 Activation functions

The non-linear activation function f allows the network to learn non-linear patterns in the data. Different activation functions are commonly used in different layers of the network. For example, in classification tasks, the activation function in the output layer is usually a softmax or log-softmax, while in regression tasks, the identity function is commonly used.

Below, we display some of the most frequently used activation functions.

Sigmoid

$$f(x) = \frac{1}{1+e^{-x}}$$

Rectified linear unit (ReLU)

$$f(x) = \begin{cases} x, \text{if } x \geq 0 \\ 0, \text{else} \end{cases}$$

ReLU has other variants, like **Leaky-ReLU**,

$$f(x) = \begin{cases} x, \text{if } x \geq 0 \\ 0.01x, \text{else} \end{cases}$$

or **SiLU** [5]:

$$f(x) = \frac{x}{1+e^{-x}}$$

The **tanh** function is also widely used in regression tasks:

$$f(x) = \frac{1-e^{-2x}}{1+e^{-2x}}$$

2.3 Backpropagation and loss functions

To measure the correctness of the predictions made by the neural network, a loss function must be defined. We denote the output of the neural network by $y^{(t)}$ and the target values by y. Given a loss function f, the loss of the prediction can be computed as

$$L = f(y, y^{(t)}).$$

In a multiclass classification problem, the output of the neural network are probabilities predicted for each class, and the target is a one-hot encoded vector representing the true class label. The most widely used loss function for classification is the **cross entropy** loss, where $y_i^{(t)}$ and y_i denote the i-th. coordinate of the output and the target vector, respectively:

$$L = -\sum_{i=1}^{n} y_i^{(t)} \log(y_i).$$

For regression tasks, one can use **mean square error (MSE)**

$$L = \frac{1}{n}\sum_{i=1}^{n}(y_i - y_i^{(t)})^2,$$

or **mean absolute error (MAE)**:

$$L = \frac{1}{n}\sum_{i=1}^{n}\left|y_i - y_i^{(t)}\right|.$$

The training of a neural network consists of two phases and it aims to reduce the loss. In the forward pass, the network calculates the loss value using the current weights. In the backward pass, the loss is propagated back through the network to update the weight matrices [19]. For a neural network with N layers, using the previous notations the gradients of the final layer can be calculated as follows:

$$\frac{\partial L}{\partial x^{(N-1)}} = \frac{\partial L}{\partial x^{(N)}}\frac{\partial x^{(N)}}{\partial x^{(N-1)}}.$$

Differentiating the loss with respect to the weights tells us how to change the weights in order to reduce the loss

$$\frac{\partial L}{\partial W_N} = \frac{\partial L}{\partial x^{(N)}}\frac{\partial x^{(N)}}{\partial W_N}.$$

These formulas are applicable for every layer, so we can go back from the last layer, compute the gradients, and update the weights.

2.4 Convolutional neural networks

Convolutional neural networks (CNN) are primarily used for processing image data for tasks such as image classification and object detection [20, 2]. They were first introduced in the 1990s [10], but gained popularity with the emergence of AlexNet [9] and other modern network architectures. In this context, we will introduce two-dimensional convolutional layers.

A convolutional layer consists of feature maps, each with the same height and width. Each feature map corresponds to a particular feature of the input data. For example, in RGB image data, each pixel has three values, one for the red, one for the blue, and one for the green color channels. Such an image will have three feature maps, each containing the values of its corresponding color channel.

The convolution operation

For each input channel, we define a kernel. A kernel k is (usually) a small matrix with height h and width w. It moves on its respective input feature map x, and for each position, it computes the value

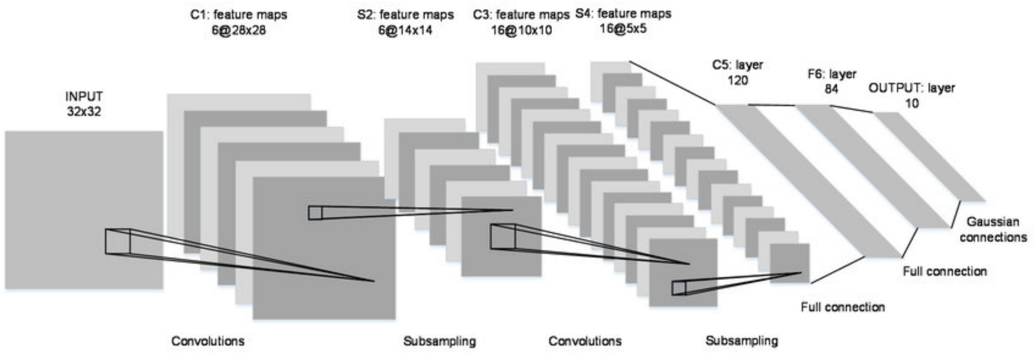


Figure 3: A CNN model [21]

$$r = \sum_{i=1}^{h} \sum_{j=1}^{w} x_{ij} k_{ij}.$$

A collection of kernels over the feature maps is referred to as a filter. Applying a filter to the data produces one feature map regardless of the number of feature maps the input data had. Therefore, the output is:

$$y_{kl} = \sum_{f=1}^{F} \sum_{i=1}^{h} \sum_{j=1}^{w} x_{k+i,l+j} k_{ij}^{f}.$$

If x was a $H \times W$ matrix, the resulting y is a $H' \times W'$ matrix, where

$$H' = H - h + 1 \quad \text{and} \quad W' = W - w + 1.$$

In practice, sometimes it is necessary to maintain the size of the input data, or on the contrary, we want to reduce its dimensionality. In the first case, padding can be used, and in the second case, we can apply pooling layers. A pooling layer has a kernel of size u by v and works almost the same way as the kernels in the convolutional layers. One main difference is that we apply pooling to each feature map of the input data and the number of feature maps remains the same after the operation. The output is:

$$y_{kl} = \operatorname*{pooling}_{i=1,j=1}^{i=u,j=v}(x_{ku+i,lv+j}),$$

where pooling is either the average or the max function.

2.5 Convolutional neural networks for time series

In a previous section, we discussed two-dimensional convolution layers, which are a great option for image data as they can capture the input's structure. However, time series are one-dimensional, though we may have multiple features at each time point. To handle this type of data, we can use one-dimensional convolutional layers, where the filters consist of $1 \times w$ kernels. Here, the parameter w represents how many time steps the network can simultaneously see.

Detecting patterns in the input data is crucial for the network, but it's possible to have shorter and long-term patterns simultaneously. A fixed kernel size may not be sufficient to handle this issue. To address this problem, one option is that we use multiple convolutional layers with different kernel sizes on the input data and then concatenate the extracted features. This approach enables the network to learn both short-term and long-term behaviors. After concatenation, applying a fully connected layer allows the network to learn the correlations between the outputs of the convolutions adaptively.

2.6 Recurrent neural networks

Recurrent neural networks (RNN) are a type of neural network architecture that is designed to process sequential data and detect patterns within it. This neural network architecture has revolutionized the field of speech recognition and machine translation [3]. Unlike feedforward neural networks, which only process inputs in one direction, recurrent neural networks feed the output of a hidden layer back to itself so that they can use the previously seen inputs to better process the current input. At time step t, the output of the hidden layer is the hidden state. This feedback loop allows the network to have a form of memory and learn from the previous inputs to better handle the current input, making it well-suited for sequential data.

An RNN has one hidden layer between the input and output layers. It can process data of any length. Here we denote the input of time step t as x_t, the hidden state as H_t, and the weight matrix of the hidden layer as W. H_t represents the memory of the network and we need a matrix W_h that regulates what information is relevant from the previous inputs. With these notations, we can compute H_t by

$$H_t = f(Wx_t + W_h H_{t-1})$$

where f is the activation function. Worth noting that W and W_h are independent of the current time step. RNNs can give output at every time step or with a given periodicity. In both cases, the computation of the output is similar to that of a feedforward network. Given U, the weight matrix of the final layer, and the activation function g, the output can be computed as follows:

$$o_t = g(UH_t).$$

RNNs are typically trained using backpropagation through time [22]. However, these recurrent networks often suffer from the vanishing gradient problem [6]. This means that the network can't handle long sequences. To address this issue, a new architecture, the LSTM [7] was proposed.

2.6.1 Long Short-Term Memory (LSTM)

Long Short-Term Memory (LSTM) is a type of recurrent neural network that was developed in 1997 [7]. In LSTM, each node is replaced by a complex node called a memory cell. In the memory cell, the input data of the current time step is combined with the information learned from the previous input, which is the hidden state, and the knowledge from all the preceding inputs, called the cell state. The memory cell takes these two states as input along with the data, updates the cell state, and produces the next hidden state. To achieve this, the memory cell has three gates:

- **forget gate**: Given the previous hidden state and the input data, it decides which part of the cell state is important
- **input gate**: Given the previous hidden state and the input data it decides which part of the input is important to the long-term memory of the network and should be added to the cell state
- **output gate**: Given the same inputs as above it filters the cell state and outputs the new hidden state. This gate decides which information is relevant and should be returned

Here we use the following notations: f_t for the forget gate, i_t for the input gate, and o_t for the output gate. Additionally, c_t and h_t represent the cell state and hidden state, respectively at time step t. The sigmoid function is denoted by σ. Now we can describe the operations in a memory cell as follows:

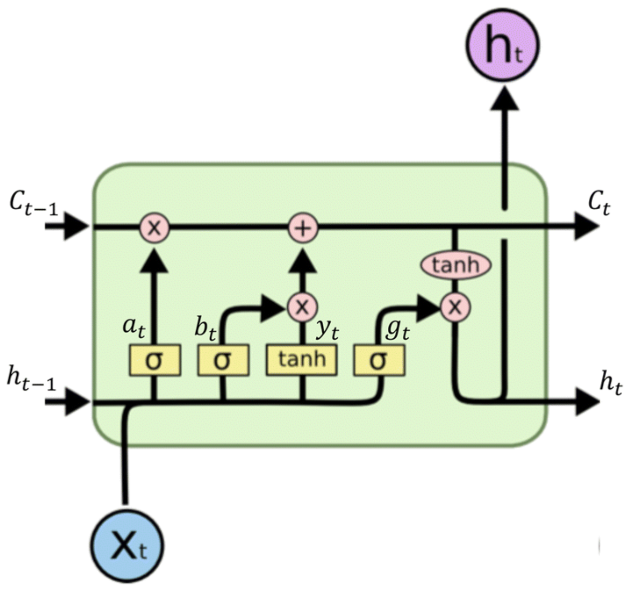


Figure 4: Illustration of an LSTM cell[2]

$$
\begin{aligned}
f_t &= \sigma(W_f x_t + W_{hf} h_{t-1}) \\
i_t &= \sigma(W_i x_t + W_{hi} h_{t-1}) \\
o_t &= \sigma(W_o x_t + W_{ho} h_{t-1}) \\
c_t &= f_t c_{t-1} + i_t \cdot \tanh(W_c x_t + U_c h_{t-1}) \\
h_t &= o_t \cdot \tanh(c_t).
\end{aligned}
$$

[2]`https://colah.github.io/posts/2015-08-Understanding-LSTMs/`

3 Graph embeddings

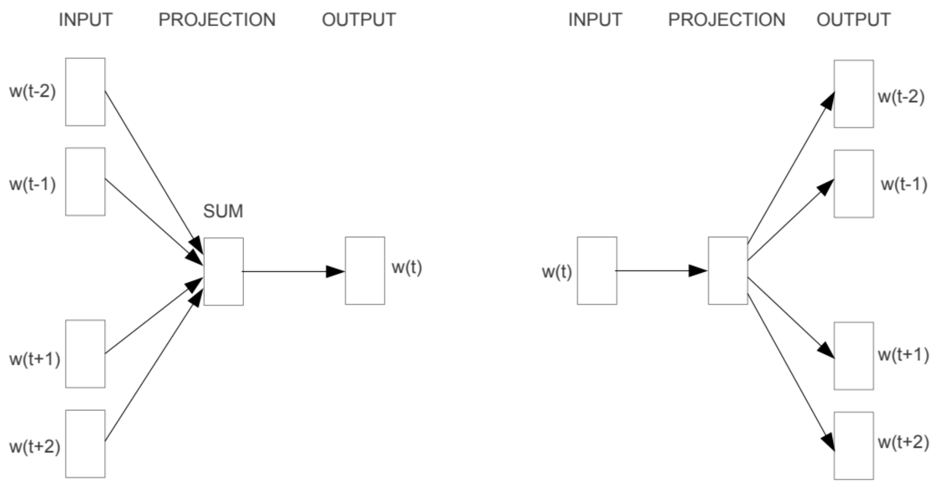
3.1 Introduction

When training neural networks, a lot of real-world data comes in the form of graphs. For example, in the traffic prediction problem, we may have a graph where the sensors are represented as nodes and the edges indicate that two sensors are close to each other. However, it is often easier if we do not have to store the entire graph but rather only a representation of it. To create such a representation, we can represent each node as a vector with d dimensions. In the past ten years, various node embedding algorithms have been developed, among others for knowledge graphs. Some algorithms explore the neighborhood of a node using random walks, like DeepWalk [13], while others use the factorization of the adjacency matrix like NetMF [15]. In this chapter, I will describe some of these node embedding algorithms and evaluate their performance on the METR-LA and PEMS-BAY traffic prediction datasets. My related results are discussed in Chapter 5.

3.2 Random walk-based algorithms and Word2Vec

Although most of the node embedding algorithms discussed in this section can be described as matrix factorization algorithms, we have included them here for completeness. The idea behind random walk-based embeddings is that we can view the graph as a vocabulary of nodes, where an edge between nodes u and v indicates that in a sentence, we can put u after v or v after u. Using this, a walk from a node is just a sentence, and generating several random walks from each node allows us to use Word2Vec [12] to embed the nodes into a vector space.

Word2Vec is a word embedding model that can be used to detect similarities between words. It has two approaches: In the **Continuous Bag-Of-Words (CBOW)** architecture, we predict the word based on its context. The order of the surrounding words does not matter, and it considers both preceding and following words within a fixed window size. The other approach is the **Skip-Gram** model, where the model takes a word as input and predicts its context. For both settings, Word2Vec has one hidden layer between the input and output layers. To train the model, we encode the words in the vocabulary as one-hot vectors and feed them into the network. The dimensionality of the hidden layer is the same as the dimensionality of the embedding, and after training, each word corresponds to a column in the weight matrix of the hidden layer.



(a) Continuous Bag-Of-Words (b) Skip-Gram

Figure 5: The two architectures of the Word2Vec model [12]

3.2.1 DeepWalk

Historically, DeepWalk [13] was one of the first graph embeddings. Originally developed to classify nodes and detect similarities in social networks, it outperformed many baseline models in these tasks at that time. DeepWalk consists of two main procedures. First, we generate several random walks from each node and then use these to train the Skip-Gram model. The difference between the bag-of-words and Skip-Gram models is that in the former, we have to compute the probability

$$P(w_0|(w_1, \dots w_k)),$$

and in the latter, we are interested in

$$P((w_0, \dots w_{k-1})|w_k).$$

When the walk length is large, it's computationally easier to use Skip-Gram than bag-of-words. So the objective function is

$$J = -\log P((w_0, \dots w_{k-1})|w_k).$$

3.2.2 Node2Vec

While exploring a graph with random walks, it's often necessary to use more complex strategies than simply choosing the next node at random. Node2Vec

[4] allows us to detect the local or far-away neighborhood of a node with the appropriate setup of the parameters, capturing latent representations of different scales. Given parameters p and q, Node2Vec computes edge weights π_{uv}, resulting in a biased random walk. Consider that we traversed the edge (t, u). Since $\forall w \in N(u), d(t, w) \leq 2$, we set the weight of the edge $(u, v), v \in N(u)$ as follows:

$$\pi_{uv} = \begin{cases} \frac{1}{p} & \text{if } d(t,v) = 0 \text{ or simply } v = t \\ 1 & \text{if } d(t,v) = 1 \\ \frac{1}{q} & \text{if } d(t,v) = 2. \end{cases}$$

With parameter p, we can influence the likelihood of going back to the previous node, and parameter q influences the likelihood of discovering nodes further away. If we compute these edge weights for every (t, u, v) tuple, we can efficiently set the weights at every node of the walk.

3.2.3 Walklets

Walklets [14] is an improvement over DeepWalk [13] in the sense that Walklets is capable of capturing representations of different scales. In our case, the measurements of a sensor on a highway may influence other sensors on the same route better than sensors from another street, even if they are closer to the chosen sensor. However, on smaller streets, only the sensors that are close to each other will influence each other. A fixed walk length cannot always capture these connections. But if we fix the length and skip some of the nodes, we can generate walks with different lengths, and these walks are able to explore both the closer neighborhood of a node and its connections far away.

Walklets generates random walks of different lengths and creates corpora $C_1, C_2, \ldots, C_k$. Given a corpus C_i and a walk $W \in C_i$, if u and $v \in W$, it means that u is reachable from v on a path of length at most i. Walklets uses the following objective function:

$$J = \sum_{u,v \in C_i} \log P(u|v),$$

where $P(u|v)$ denotes the probability of node u co-occuring with node v in a walk in C_i.

3.2.4 Role2Vec

Role2Vec [1] introduces the concept of attributed random walks. The algorithm first defines a function that maps each vertex to a type. Then, it

generates walks where consecutive vertices are adjacent and belong to the same type. Finally, Role2Vec uses Skip-Gram [12], similar to the previously mentioned embedding methods, to create embeddings based on the generated walks.
To create the vertex-to-type function Φ, we denote the vertices of the graph as $V = \{v_1, \ldots, v_n\}$ and the different types as $W = \{w_1, \ldots, w_m\}$. For every vertex $v \in V$ we denote its features as x_v. Thus the type of v is determined by

$$w(v) = \Phi(x_v).$$

After the types have been determined, we can calculate the transition probabilities in a very similar way to Node2Vec [4]. Specifically, we have

$$\pi_{uv} = \begin{cases} 0 & \text{if } w(v) \neq w(u) \\ \frac{l(uv)}{Zd_w(u)} & \text{if } w(v) = w(u), \end{cases}$$

where l denotes the edge weights in the graph, $d_w(u)$ denotes the number of neighbors of u with the same type as u and Z is the normalizing constant. In the unweighted case, $l \equiv 1$ and $Z = 1$.

3.2.5 Diff2Vec

Diff2Vec [18] is a random walk-based algorithm that generates a subgraph for every node and samples it for a sequence of vertices. Given a graph G, vertex v, and integer l, Diff2Vec generates G_v, a subgraph of G where $\left|V(G_v)\right| = l$. Starting with $V(G_v) = v$ and $E(G_v) = \emptyset$, Diff2Vec chooses a random vertex $u \in V(G_v)$, selects w from its neighbors in G at random then adds w to $V(G_v)$, and the edge (u, w) to $E(G_v)$. This process repeats until $\left|V(G_v)\right| = l$.

After generating G_v, each edge in G_v is duplicated to ensure that every node in the graph has an even degree. This results in a graph that has an Euler walk, and the node sequence corresponding to v is given by this walk. With this process, we generate a number of random walks W. Given W and a window w, we generate feature vectors for each vertex. The feature vectors corresponding to v are

$$\begin{aligned} f_v^{-1} &= (b_{u_i}, b_{u_2}, \ldots, b_{u_n}) \\ f_v^{1} &= (a_{u_i}, a_{u_2}, \ldots, a_{u_n}). \end{aligned}$$

Here, b_{u_i} and a_{u_i} represent the number of times the vertex u_i appeared in the walks before and after the vertex v, respectively, within a window of size w.

The MLP neural network used to generate the embedding follows the same approach as Word2Vec by setting the number of neurons in the hidden layer equal to the desired number of dimensions in the embedding. Specifically, we train the one-hot encoded vertices against their feature vectors using this MLP model and generate the embedding through the weight matrix of the hidden layer.

3.3 Graph embeddings based on matrix factorization

It has been shown that some of the random walk-based algorithms listed in the previous section are technically based on matrix factorization [15], that is, the embedding is produced by approximating a matrix using random walks and then performing factorization on that matrix. For example, DeepWalk [13] approximates the following matrix [15] as the walk length goes to infinity:

$$\log\Big(|E| \Big(\frac{1}{T}\sum_{r=1}^{T}(D^{-1}A)^r\Big)D^{-1}\Big) - \log(b),$$

where A is the adjacency matrix and D is a diagonal matrix containing the degrees of each vertex. The embeddings generated by DeepWalk [13] approximate the top-d singular values of this matrix.

There are embedding algorithms that factor either the adjacency matrix or the Laplacian of a given graph, but here we will only describe NetMF [15].

3.3.1 NetMF

The NetMF [15] algorithm has two variants, here we will describe a universal algorithm that works for all window sizes. Although for larger window sizes some computations become infeasible and an approximation is needed, while for small window sizes, we can compute the exact values. Given a graph G and its adjacency matrix A, the first step is to compute the Laplacian of A:

$$L = D^{-1/2}AD^{-1/2},$$

where D is diagonal and $D(i,i) = deg(u_i)$. We can write this matrix as $L = U\Lambda U^T$ using eigenvalue decomposition, where Λ is a diagonal matrix containing the eigenvalues. With Λ, we compute matrix M' as follows:

$$M = \frac{|E|}{b}D^{-1/2}U\Big(\frac{1}{T}\sum_{r=1}^{T}\Lambda^r\Big)U^T D^{-1/2}$$
$$M' = \max(M, 1).$$

If T is sufficiently large, the calculation of M becomes hard. To avoid this, we can approximate L as $L = U_h \Lambda_h U_h^T$ and use this approximation in the calculation of M. Furthermore, we use SVD to express M' as

$$M' = R \Sigma S^T.$$

For an embedding with d dimensions, we are only interested in the first d singular values. Therefore, the embedding vectors are calculated as

$$E = R_d \sqrt{\Sigma_d}.$$

3.4 Attributed graph embeddings

The graph embeddings described earlier only capture the structure of the graph, which means that nodes that are close to each other in the graph are mapped to nearby vectors. However, in many applications, nodes have additional features, and it is desirable that nodes with similar features are mapped close to each other in the embedding. In such cases, attributed graph embeddings are used.

3.4.1 Scalable Incomplete Network Embedding

One approach to attributed graph embeddings is SINE (Scalable Incomplete Network Embedding) [24]. SINE is a scalable graph embedding algorithm that can incorporate node attributes into the embedding. There are k different node attributes $a_1, \ldots, a_k$ and the input of the algorithm is the adjacency matrix and an attribute matrix $X \in R^{n \times k}$. To generate the embedding, SINE first generates a set of random walks from each node in the graph. It then applies a Skip-Gram style neural network to the random walks to learn the embedding.

The neural network used in SINE has one hidden layer, and the weight matrix of this layer gives the embedding. The network has two output layers: O_v and O_a, each with a different weight matrix. To learn the embedding, in every iteration SINE samples node pairs (v_i, v_j) with probability $\frac{1}{2}$ from the random walks and node-attribute pairs (v_i, a_j) with probability $\frac{1}{2}$. In the first case, SINE uses the output layer O_v to learn the context of the node, and in the second case, it uses O_a to learn the attributes of the node. The two objectives are:

$$\begin{aligned} J_1 &= -\log P(v_j | v_i) \\ J_2 &= -\log P(a_j | v_i). \end{aligned}$$

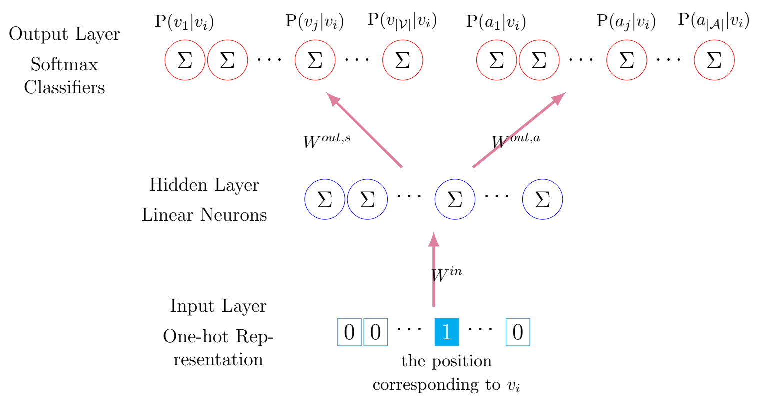


Figure 6: The model architecture of SINE [24]

3.4.2 FeatherNode

FeatherNode aims to compute each feature's distribution in the neighborhood of each node. The characteristic function of X for node u at point θ is

$$E[e^{i\theta X}|G,u] = \sum_{w\in V} P(w|u)\cdot e^{i\theta x_w} = \sum_{w\in V} P(w|u)\cos(\theta x_w) + i\cdot \sum_{w\in V} P(w|u)\sin(\theta x_w).$$

To describe the distribution of X on the nodes that are reachable from u by a random walk with r steps, the following equations were defined:

$$Re(E[e^{i\theta X}|G,u,r]) = \sum_{w\in V} P(v_{j+r}=w|v_j=u)\cos(\theta x_w) \quad (1)$$

$$Im(E[e^{i\theta X}|G,u,r]) = \sum_{w\in V} P(v_{j+r}=w|v_j=u)\sin(\theta x_w). \quad (2)$$

If we denote the normalized adjacency matrix by $\hat{A}$, then the r-step probabilities can be described by $\hat{A}^r$.

For a feature X and scales $\{1,2,\ldots,r\}$ we evaluate the real and imaginary part of the characteristic function at evaluation points $\{\theta^1,\theta^2,\ldots,\theta^r\}$ then concatenate the results, getting Z_{Re} and Z_{Im}. If there are more attributes, we repeat this process and concatenate the resulting Z_{Re} and Z_{Im} matrices, respectively. The final embedding will be the concatenation of Z_{Re} and Z_{Im}.

3.4.3 Attributed Embedding

Attributed Embedding (AE) [16] is a Skip-Gram-based embedding. Each node has an attribute set which will be denoted by F_v. The algorithm has three phases. In the first phase, we choose a starting node v_1 with probability proportional to node degree and sample a node sequence $(v_1, v_2, \dots, v_l)$ with a random walk starting from v_1. We split this sequence into two parts, the first $l-t$ nodes are the source nodes and the last t nodes are the target nodes. For each (source node, target node) pair (v_j, v_r) we add all the (v_r, f) and (v_j, f') pairs to the corpus C_r where $f \in F_{v_j}$ and $f' \in F_{v_r}$. We repeat this process s times.

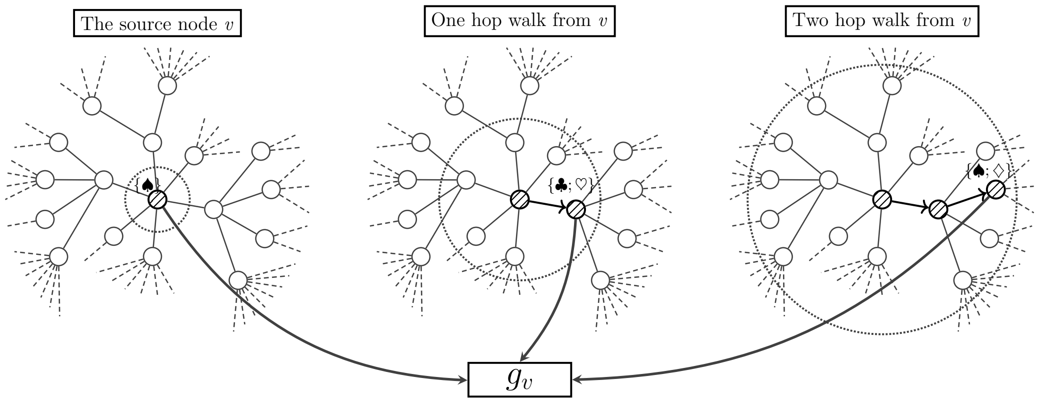
In the second phase, we create the corpus $C = \cup C_r$.

In the third phase, we train a Skip-Gram model on the corpus C to obtain a node embedding g and a feature embedding h.

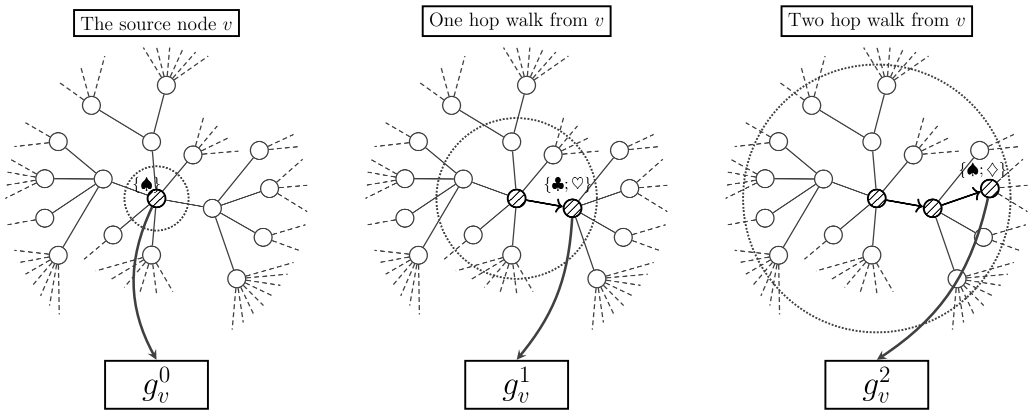
3.4.4 Multi-Scale Attributed Embedding

Multi-Scale Attributed Embedding (MUSAE) [16] is a variant of AE [16] modified to learn multi-scale dependencies. First, a window size t is chosen so that t divides d. MUSAE runs the first phase of AE to generate corpora C_r, $r \in \{1, \dots, t\}$.

In the second phase, the algorithm runs Skip-Gram on each corpus C_r to obtain t dimensional node embeddings g_r and feature embeddings h_r. Let g and h denote the concatenation of the embeddings g_r and h_r, respectively. The output of the algorithm is the pair (g, h).



(a) Pooled attributed embedding of node v.



(b) Multi-scale attributed embedding of node v.

Figure 7: In Figure 7a we learn a node embedding with the AE algorithm. In Figure 7b we learn the embedding with the MUSAE algorithm [16]

4 Datasets

There are two primary real-world datasets available for the task of traffic forecasting: METR-LA and PEMS-BAY[11]. The METR-LA dataset contains speed readings collected from 207 sensors located on highways in Los Angeles County between March 1st, 2012, and June 30th, 2012. In contrast, the PEMS-BAY dataset comprises data from 325 sensors in the Bay Area, covering a five-month period. Both datasets provide average speeds every five minutes. The data is divided into training, validation, and testing sets, with a fixed split of 70%, 10%, and 20%, respectively. In both cases, we can incorporate the time of day or the day of the week as additional information in the dataset. As shown in Figure 8, the standard deviation of the speeds measured by the sensors can be quite high, particularly in the case of the METR-LA dataset. Figure 10 highlights that the majority of the sensors detect rush hour traffic once a day, either in the morning or afternoon. The sensors are usually in pairs, which means they monitor opposite directions of the highway. This can lead to high differences in measurements despite the Euclidean distance between the two sensors being low, because in one direction the rush hours occur in the morning and in the other direction the rush hours occur in the evening.

Since the coordinates of the sensors are available in both datasets, it is possible to calculate pairwise distances between them. By treating each sensor as a node, a weighted graph of the road network can be constructed. The weight of the edge e between node v_i and v_j is determined as follows:

$$w_e = \begin{cases} e^{\frac{d(v_i v_j)^2}{\sigma^2}}, & \text{if this is} < \kappa \\ 0 & \text{otherwise,} \end{cases}$$

where $d(v_i, v_j)$ represents the distance between the nodes v_i and v_j, σ is the standard deviation of the distances and κ is a threshold value. Filtering out the small weights is required to keep the graph sparse.

Figure 10 compares the traffic speeds of a few sensors in two datasets. Besides periodicity, we observe that some sensors are uncorrelated. For example, in the PEMS-BAY dataset on weekdays, sensor 400017 measured lower speeds in the afternoon, while sensor 402362 measured lower speeds in the morning.

Figure 11 shows the measurements in a 3-week span. Especially in the PEMS-BAY dataset, we can easily detect the weekends and weekdays. On weekends, the rush hours rather shift to the middle of the day, and their magnitude is smaller.

Comparing the train and test sets of PEMS-BAY and METR-LA, we can see in Figure 8, that while in PEMS-BAY the mean and standard deviation of the sensors remains the same, in METR-LA, there's a minor shift in terms of the mean and a bigger shift in terms of the standard deviation. Although [11] does not elaborate on how the train/validation/test split was chosen, probably the test set contains the last 20% of the data. In this case, the measurements of the test set occur in June and the holidays and trips could explain this. The measurements of PEMS-BAY are less affected by the holiday season because it covers the period from Jan 1st, 2017 to May 31, 2017.

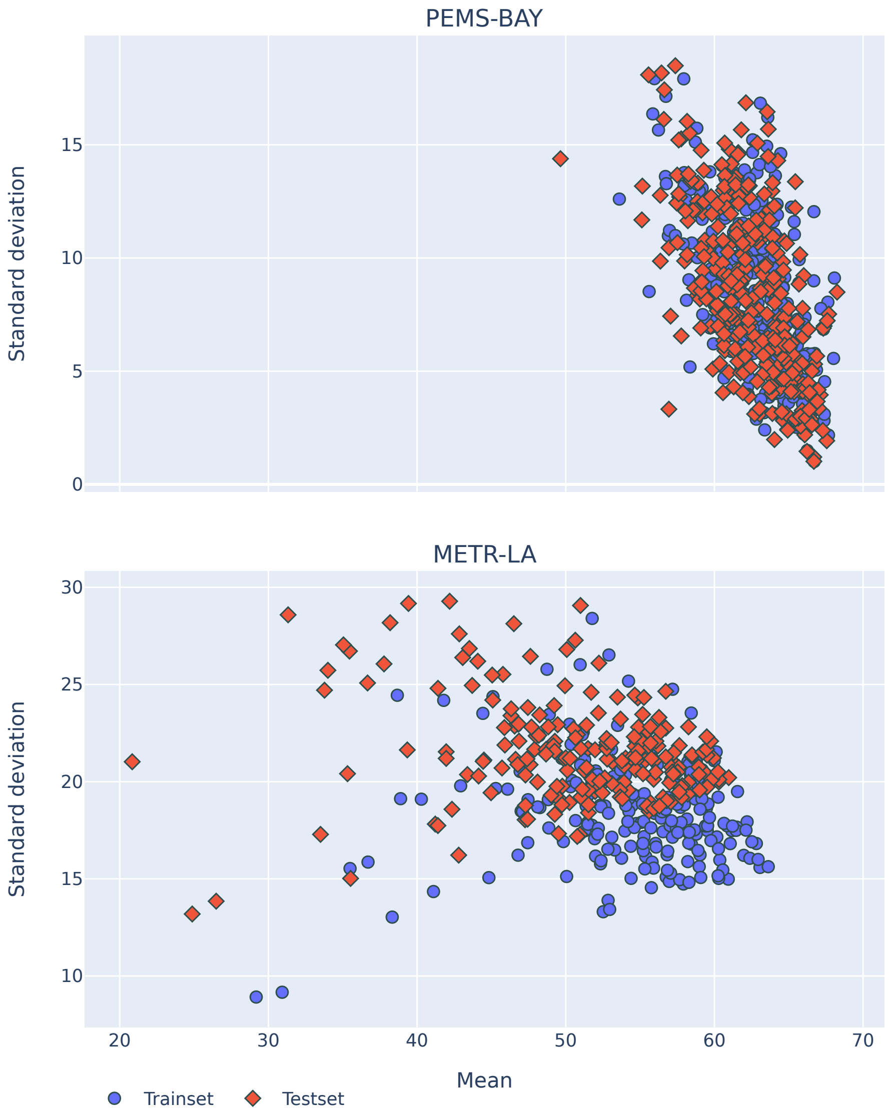
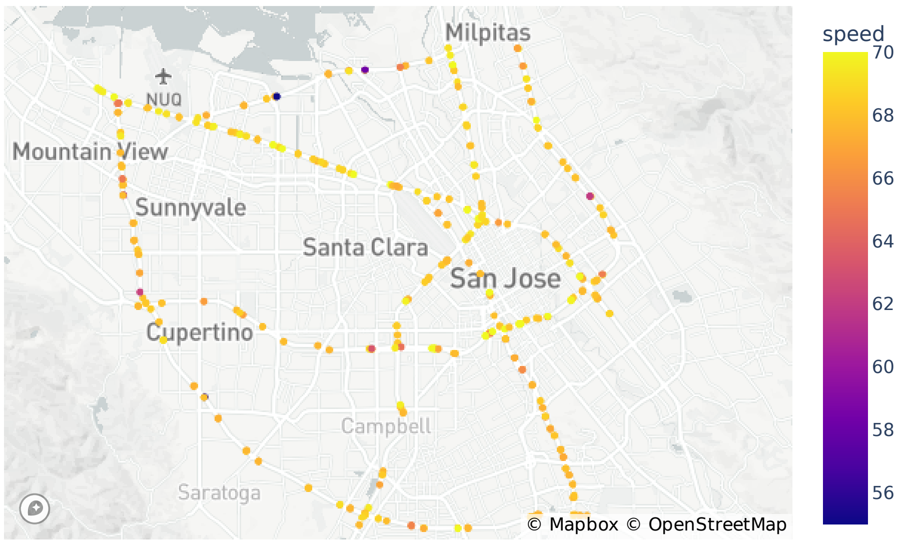
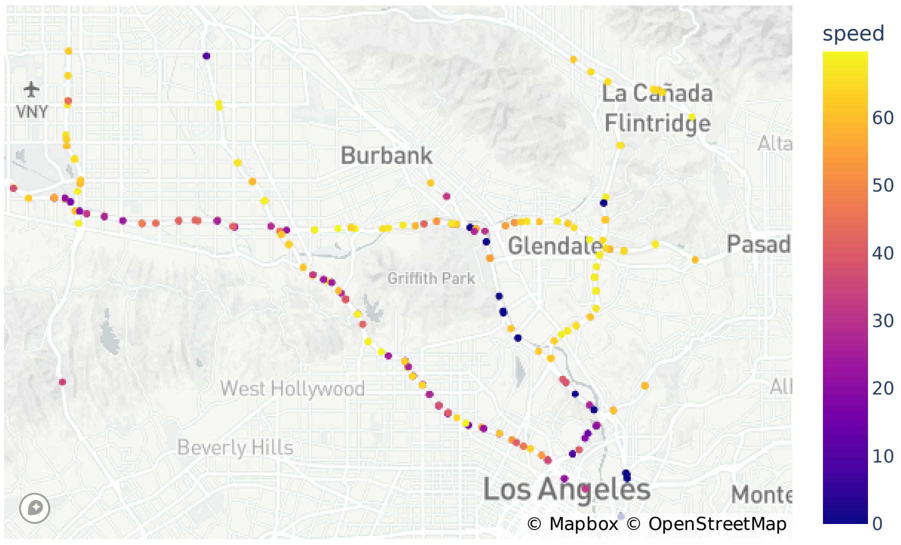


Figure 8: The mean and standard deviation of the sensors

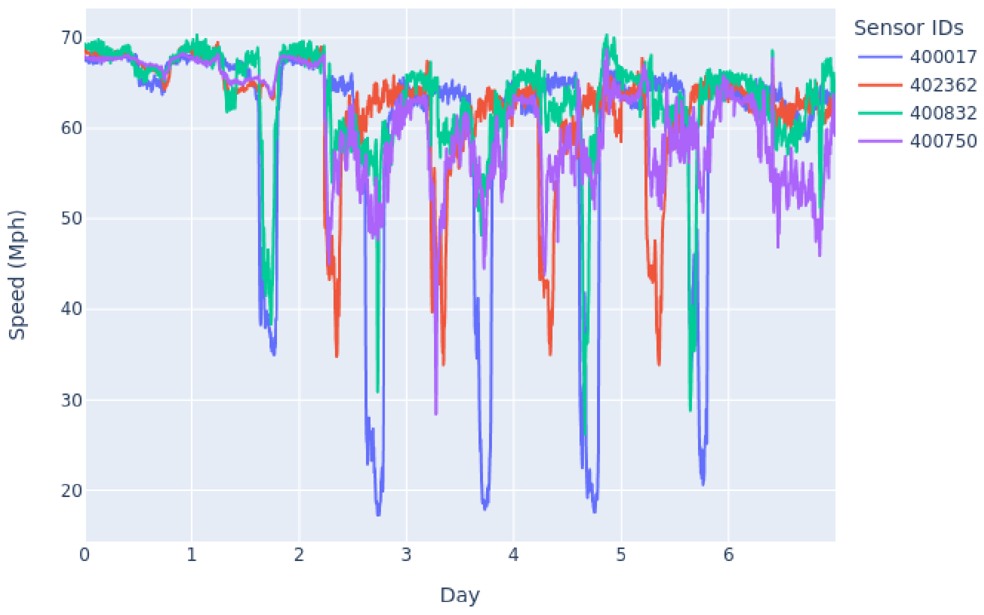


(a) PEMS-BAY

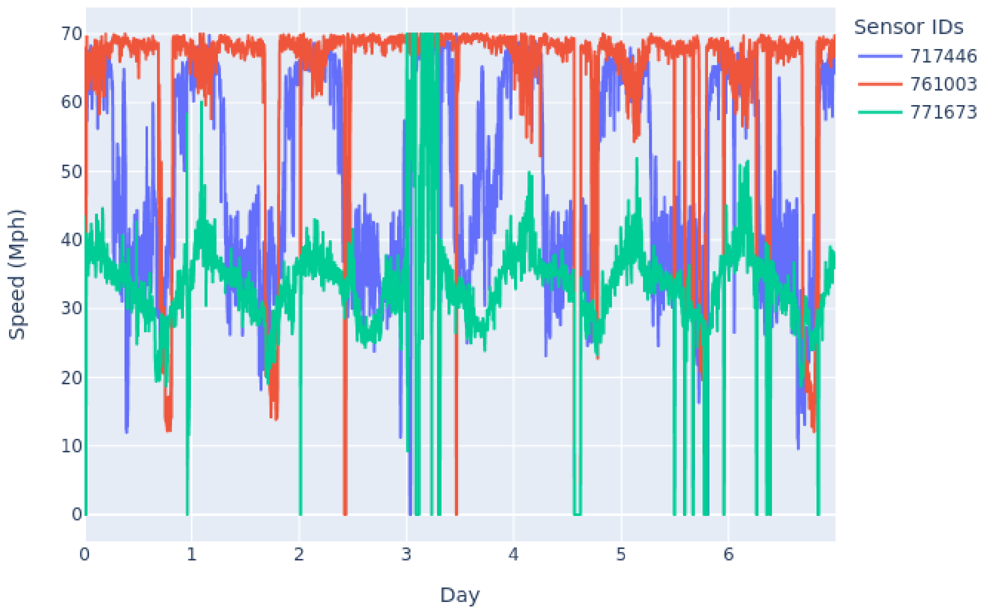


(b) METR-LA

Figure 9: The location of the sensors and the measured speeds at a given time.

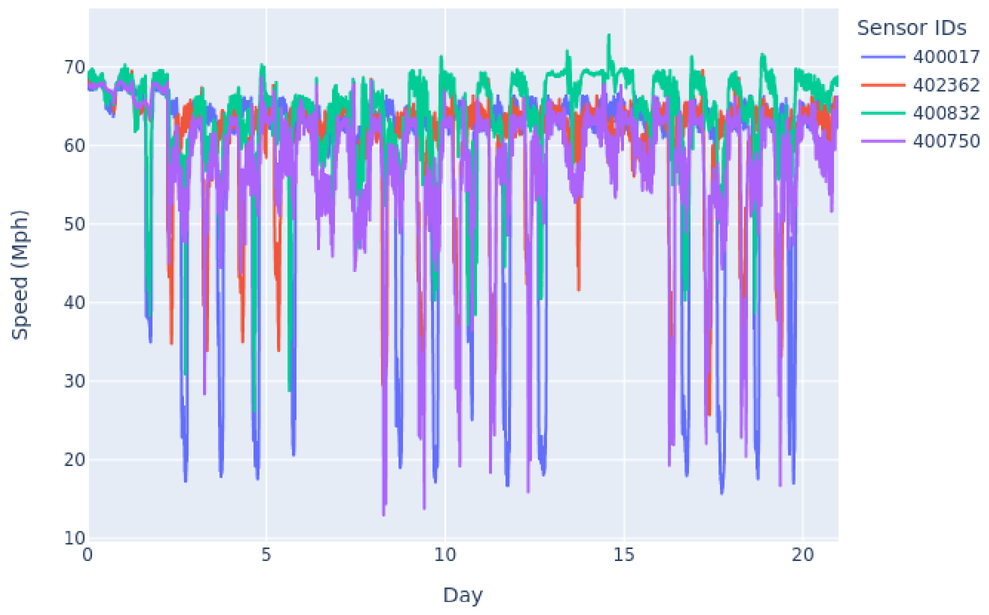


(a) PEMS-BAY

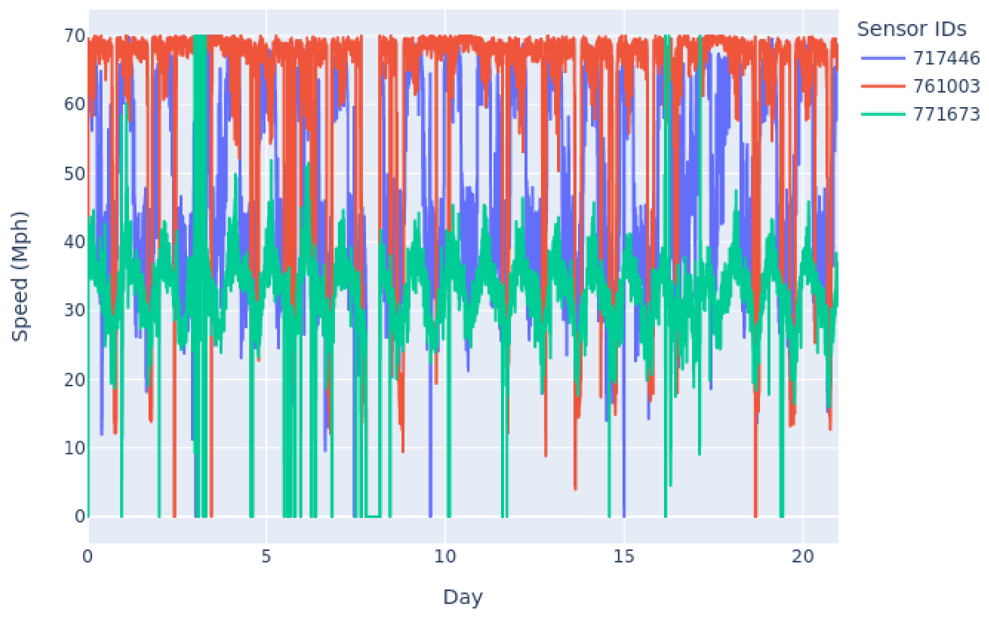


(b) METR-LA

Figure 10: The measurements of different sensors during a week.



(a) PEMS-BAY



(b) METR-LA

Figure 11: The measurements of different sensors during a 3-week span.

5 Results

We performed experiments on the two datasets introduced in Section 4 with various regression models and with more than ten node embeddings that extract features from the sensor proximity network. We used two different metrics to compare the embeddings and to compare our results to other GNN-based models:

- Mean Absolute Error (MAE)
- Rooted Mean Squared Error (RMSE)

5.1 Models

We did the experiments with two different neural networks:

- Multilayer Perceptron (MLP)
- Convolutional Neural Network (CNN)

As we formulated the traffic forecasting problem in Section 1, we have not only a time series for each sensor but a graph where each node represents a sensor and an edge between two nodes indicates that the respective sensors are close to each other.

In the MLP model, we used fully connected layers to extract the features from the embedding vectors, then a final dense layer to get the predictions for each node.

In the CNN model, we treated each node as a different univariate time series, where the additional features were the embedding vectors. To do this, we used $1 \times w$ dimensional kernels in the convolutional layers to extract the features given by the embedding vectors. Finally, we applied a linear layer to the results of the previous layers to learn the dependencies between the time series.

5.1.1 Feature extraction for the attributed node embeddings

We experimented with attributed node embeddings, such as SINE [24], FeatherNode [17], MUSAE [16] or AE [16]. This type of embedding requires a feature matrix containing the features for each node. In the traffic prediction task, the nodes don't have predefined attributes, so we tried to generate features from the time series.

One way to create features is to use all the speed readings from the training dataset as attributes for each node. However, this would result in too many features, making it difficult for the embeddings to learn effectively.

We considered two different approaches and experimented with their combination too. Our first attempt was to use the mean and standard deviation of the speed readings at each node in the training data as features. We denote this feature matrix as M_1. As shown in Figure 8, clustering the nodes by these two features is difficult, and we can't obtain meaningful clusters. Furthermore, there's a major difference between the training and test set mean and standard deviation of traffic speed for the METR-LA dataset.

In the second attempt, we assigned two binary attributes to the nodes based on the training dataset. We noticed that when a sensor measures low speeds, most of the measurements occur in either the morning or the afternoon. An example of this is Figure 10. Clustering the sensors by this fact allows us to separate sensor pairs monitoring the opposite directions of a road. As a result, we defined the two attributes as follows:

- A_1: "morning"
- A_2: "afternoon"

We applied Z-score normalization to the time series at each node and counted the very low values. If this number is smaller than a threshold value, we set both attributes to zero, representing that this sensor doesn't experience a rush hour. In the other case, we grouped them by the time of day and set the respective attribute to one. This feature matrix will be denoted by M_2.

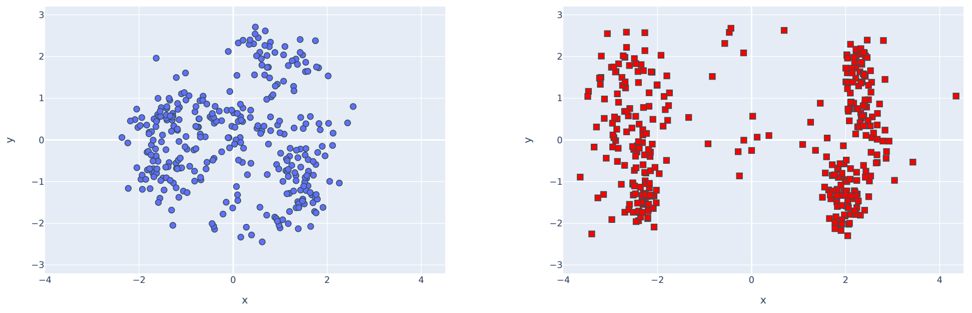
To demonstrate the difference between the two approaches, we trained the SINE [24] embedding with both feature matrices and applied PCA to reduce the dimensionality of the embedding. As illustrated in Figure 12, SINE is capable of capturing some form of relationship between the nodes using both the adjacency matrix and the assigned attributes with both feature matrices. The effect of the chosen features is more visible in the case of M_2, where the two significant clusters correspond to the two attributes.

For the third approach, we combined the first two feature matrices, denoted as M_3.

5.2 Experiments on the PEMS-BAY dataset

5.2.1 Structural embeddings

We experimented with many structural embeddings, and the results are shown in Figure 14, where we displayed some attributed embeddings for



(a) SINE with M_1 (b) SINE with M_2

Figure 12: The difference between the two feature matrices. We embedded the vectors into a higher dimensional vector space and used PCA on the embedding vectors.

comparison. Amongst the structural embeddings, Node2Vec and DeepWalk performed poorly with both the MLP and the CNN models. We experienced the highest variance in the test result at these two embeddings too. The possible causes of this are that DeepWalk explores the neighborhood of a node with a first-order random walk and Node2Vec uses biased second-order random walks. These are the simplest approaches among the tested embeddings and the difference between the chosen random walks in the experiments is high.

NetMF and Role2Vec had the same performance as the model without network information and the other embeddings had a slightly worse performance in terms of both MAE and RMSE with the MLP model.

Using the CNN model, the performance of Node2Vec increased significantly but was still below the model without network information. NetMF outperformed the model without network information and even the attributed embeddings.

5.2.2 Attributed embeddings

The attributed embeddings performed well overall, but compared to the structural embeddings in Figure 14, we experienced that NetMF outperformed every attributed embedding we tried. One of the reasons is, that the performance of the attributed embeddings heavily depends on the model and the feature matrix used. Using the MLP model, FeatherNode performed the same as the model without embeddings with each feature matrix. SINE, AE, and MUSAE performed poorly and their performance highly depended on the feature matrix used. In the case of SINE, there are huge differences between the feature matrices.

Using the more powerful CNN model, every embedding had a significantly improved performance and the performance difference of the feature matrices was reduced. Figure 13 illustrates the performance of SINE, FeatherNode, AE, and MUSAE with the three feature matrices using the CNN model on the PEMS-BAY test set. We can see, that in most cases, M_2 has a slightly better performance than the other attribute sets, especially in terms of RMSE, and each embedding outperformed the model without network information, except MUSAE. M_2 had the smallest variance among the feature matrices. Probably its cause is that the embeddings learn binary features easier. In the case of SINE, the worst-performing feature matrix with the MLP model, M_2, has the best performance with CNN.

MUSAE had the worst performance among the attributed embeddings. The reason for this is that MUSAE fits multiple low-dimensional embeddings and concatenates them. We used relatively small, 32-dimensional embeddings, and the embeddings of the different scales couldn't capture the relationships between the nodes. Although MUSAE is an improvement over AE, it falls short of AE for the same reason. FeatherNode also suffered from the reduction of the embedding dimension and the small number of features. FeatherNode has to evaluate the characteristic function on multiple points to have solid performance but this would increase the training time of both the embedding and the neural network. SINE had the best performance and it shows that even handcrafted features can be powerful enough to improve the worst embedding. SINE explores the neighborhood of a node like DeepWalk, but the nodes get embedded based on the graph structure and their attributes. This ensures that two sensors monitoring the opposite directions on the same road don't get embedded close to each other although the distance between them is low and they occur in a lot of random walks together.

5.3 Experiments on the METR-LA dataset

5.3.1 Structural embeddings

We compared the embeddings in Figure 15. As in the PEMS-BAY dataset, NetMF has the best performance, but surprisingly, Walklets and DeepWalk performed well compared to other structural embeddings. Every structural embedding except Grarep, Diff2Vec, and GLEE had a bit better performance than the model without the network information. But we have to note, that unlike on PEMS-BAY, where we were able to outperform methods such as Graph WaveNet [23], the performances on METR-LA were far from the graph neural network-based methods.

5.3.2 Attributed embeddings

We used the same feature matrices as with PEMS-BAY, and we found that using the time of day when the rush hours occur is again the best feature matrix, although it's harder to detect rush hours. It's best seen in Figure 10 and 18 as the measurements of the sensors in the Bay area are much cleaner than in Los Angeles. In Figure 18 we can't really talk about rush hours since the period where the speed of the traffic drops covers more than half of the day and this period also has some peaks. The average speed even drops to zero at a certain point. Using the mean and standard deviation of the measured speeds as features leads to poor performance. Taking a look at Figure 8, the range of the mean and standard deviation of the sensors in the METR-LA dataset is greater than in the PEMS-BAY dataset, and in METR-LA, there's a big difference between the train and test dataset.

Like on PEMS-BAY, SINE has the best performance among the attributed embeddings but in METR-LA, the difference between DeepWalk and SINE is smaller than in the case of PEMS-BAY, because the used feature matrices don't contain as much useful information as in PEMS-BAY. The performance of FeatherNode and MUSAE falls short of SINE for the same reasons as in the case of PEMS-BAY.

5.4 The effect of the embedding dimension

We compared the two best-performing embeddings, NetMF and SINE (with M_2) with a higher embedding dimension on PEMS-BAY. In Figure 16 we chose one specific run from each type and displayed how the MAE and RMSE on the validation set changed during the training on the PEMS-BAY and Figure 17 highlights the difference on the PEMS-BAY test set. With the embedding dimension set from 32 to 128, we achieved a little increase in performance in the case of both embeddings although the difference is more visible with NetMF. With other embeddings, like Role2Vec the high dimensions did not affect the performance. With the increased embedding dimensions we could compare the results to other methods, such as Graph WaveNet [23] and MegaCRN [8]. The best run of NetMF has the same MAE value as MegaCRN and a lower RMSE value as shown in Figure 17.

5.5 Node-level performances

We examined the results of the best-performing embedding, NetMF, for every node and displayed the results on the map in Figure 19. Due to the nature of the datasets, we used the 1-step prediction results here, which means that based on the last hour we predicted the average speed only for the next 5 minutes.

With the **PEMS-BAY** dataset, there was only one node with a MAE value higher than 4 and the majority of the nodes had a MAE value between 0.6 and 1.3.

On the **METR-LA** dataset the number of nodes with very small or very high MAE values was small, just like with PEMS-BAY.

With both datasets our model was able to capture the periodicity of the traffic and detect the beginning and the end of rush hours but especially with METR-LA, it couldn't detect the small peaks in the data. As Figure 19 highlights, plotting the mean absolute error of the prediction and the target values for each node, we get higher values mostly at bigger road crossings where the speeds highly depend on the other road and in the case of sensors which are isolated from the others so they can't get relevant information from their neighbors.

5.6 Runtime

We compared the embeddings based on the runtime of calculating the embedding vectors and the average time for an epoch too in Figures 20 and 21. In the experiments, we used 32 and 128-dimensional embeddings and the difference in the calculation times was minimal so we only show the results for the lower embedding dimension. The attributed embeddings were very slow compared to the structured embeddings, except FeatherNode. This is related to that we had to reduce the evaluation points of the characteristic function to get a low-dimensional embedding. The choice of the feature matrix also plays a minor role in the speed of the embedding. Our observation is that the embeddings learn easier the binary attributes, usually the feature matrix M_2 was the fastest. In Figure 20 we show the feature matrix M_3 because this matrix combines the other two and contains the most features.

The average epoch times were mostly influenced by the dimensionality of the embeddings and there's no significant difference between the embeddings. Figure 21 shows the results with 32-dimensional embeddings. Increasing the dimension to 128 would double the epoch times, although this would increase the performance as Figure 17 shows.

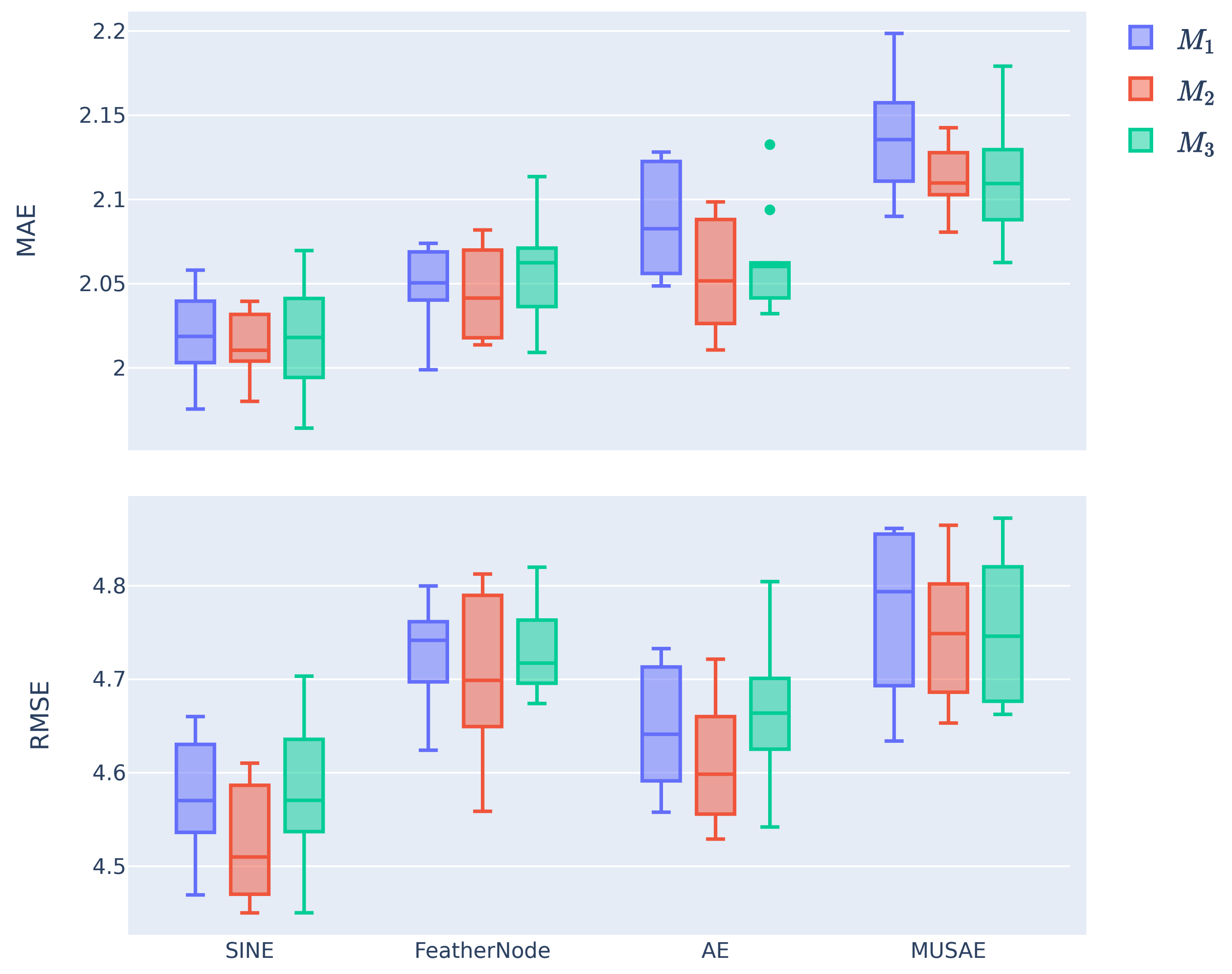


Figure 13: We compared the performance of the feature matrices on the PEMS-BAY test set with the CNN model. In most cases, if the embedding knows only the mean and standard deviation of the measured speeds at a node (blue), it performs worse than if it knows that the rush hour occurs in the morning or in the afternoon (red). The performance falls between the two previous approaches if the embedding knows all these features (green).

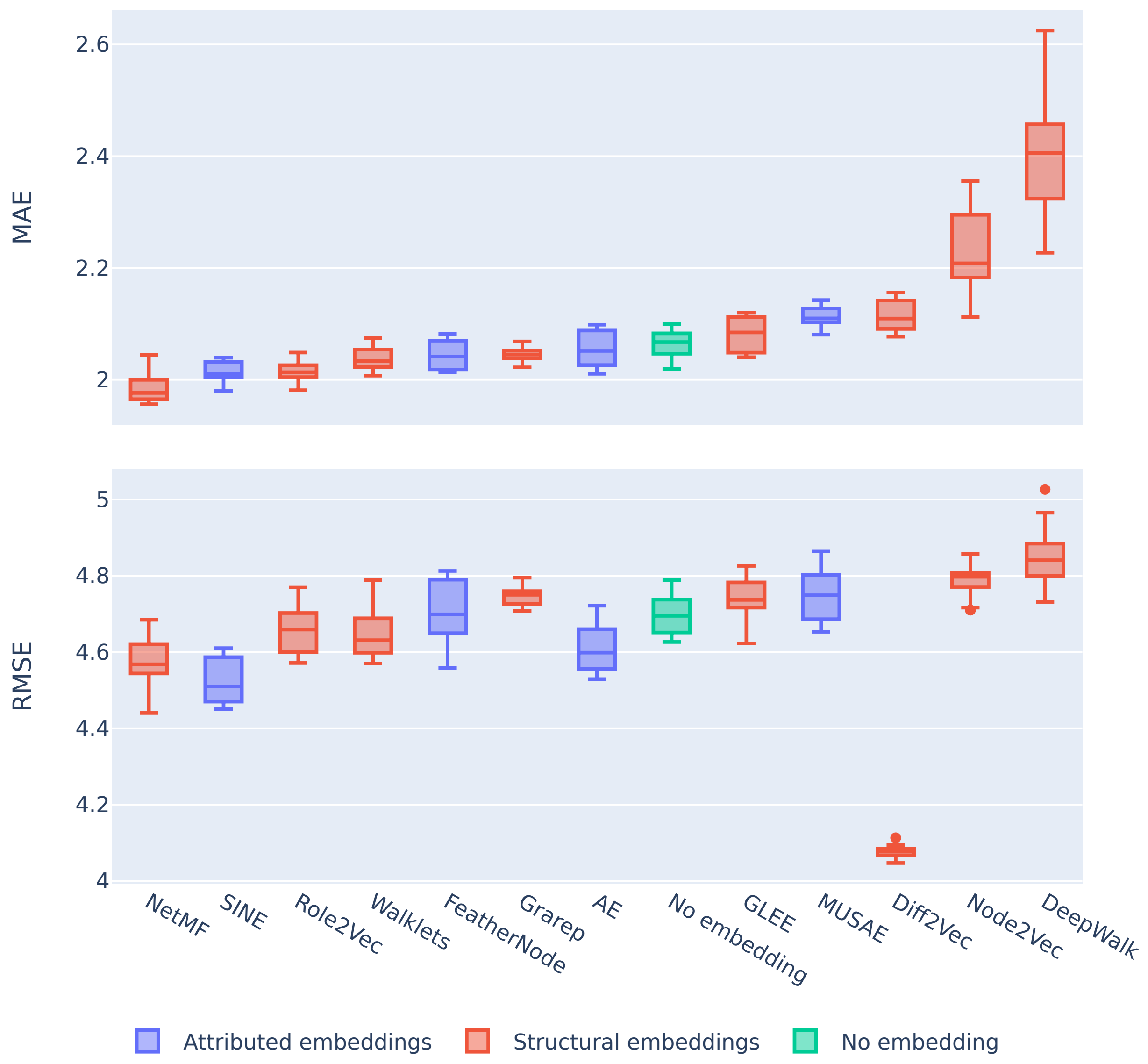


Figure 14: MAE and RMSE values with CNN on the PEMS-BAY test set. In the case of the attributed embeddings, the algorithm only knew the information if the rush hour occurs in the morning or the afternoon as we measured the best performances with this feature matrix.

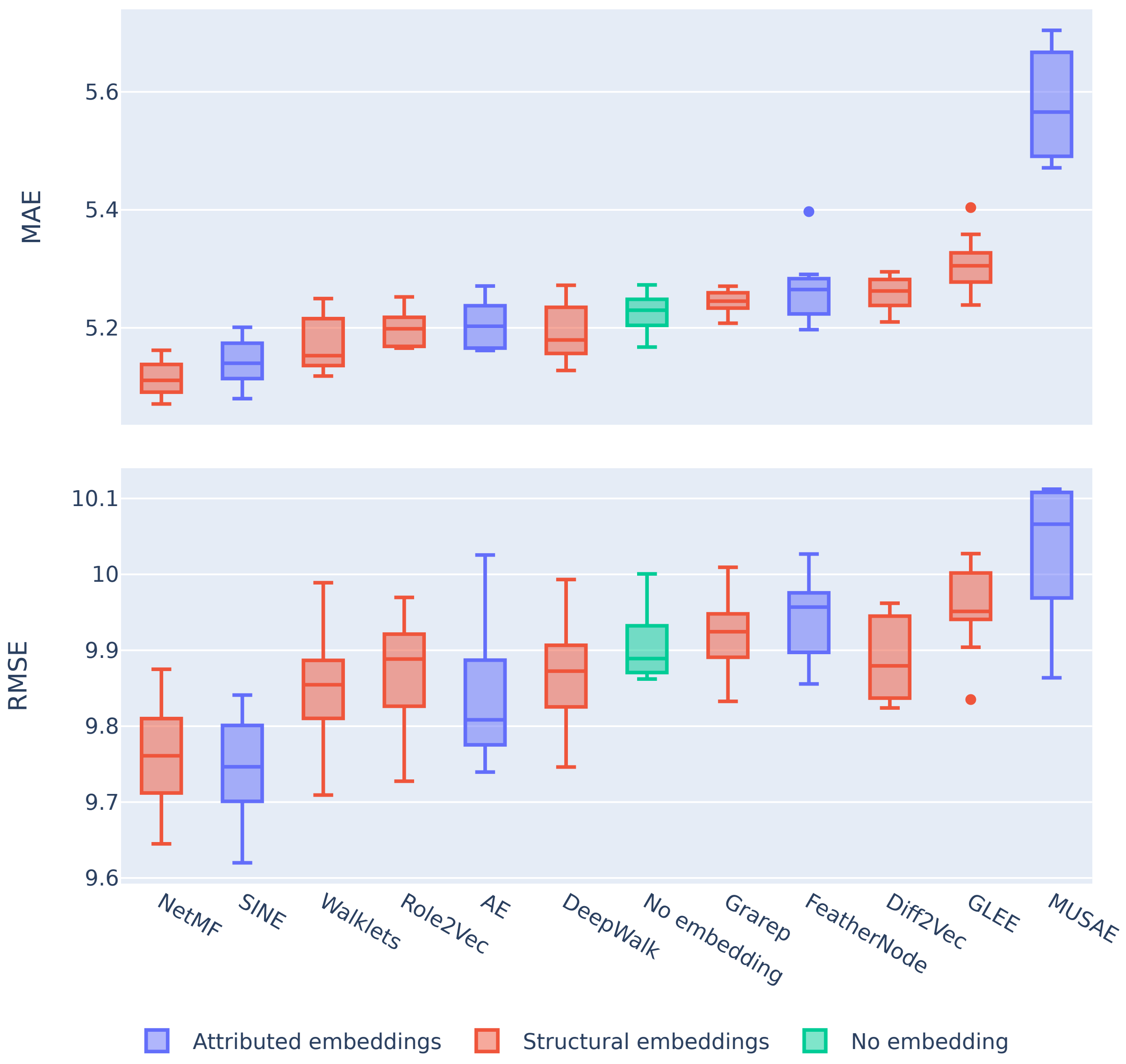


Figure 15: MAE and RMSE values with CNN on the METR-LA test set. For the attributed embeddings we chose the mean and standard deviation of the measured speeds as this feature matrix had the best performance.

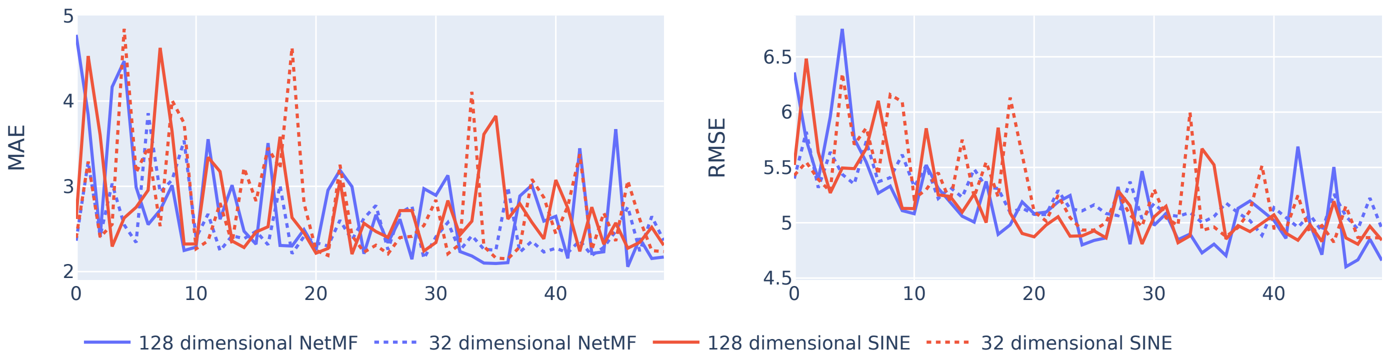


Figure 16: The performance of SINE and NetMF on the PEMS-BAY validation set with different embedding dimensions.

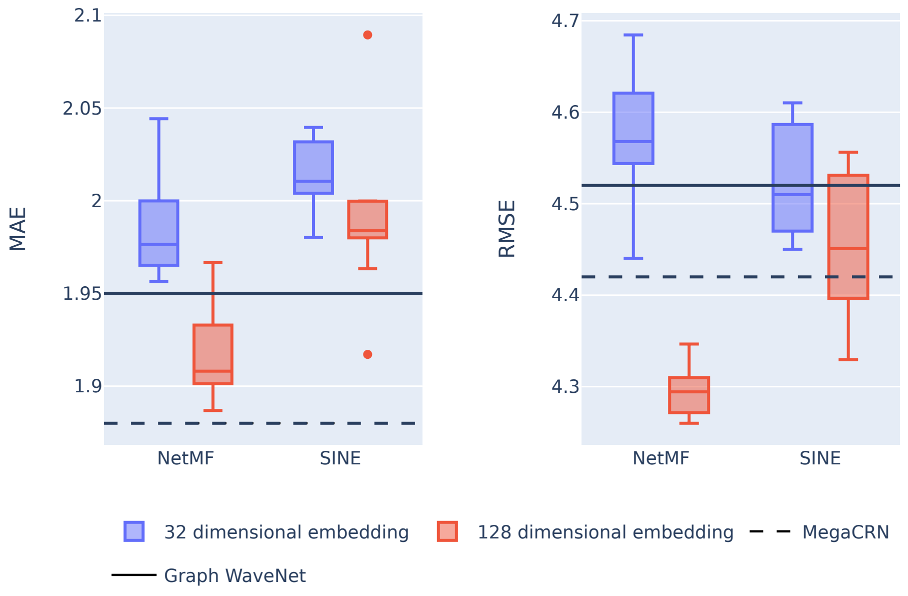


Figure 17: We compared SINE and NetMF on the PEMS-BAY test set with different embedding dimensions. The higher dimensionality increases the performance, it's especially true for NetMF.

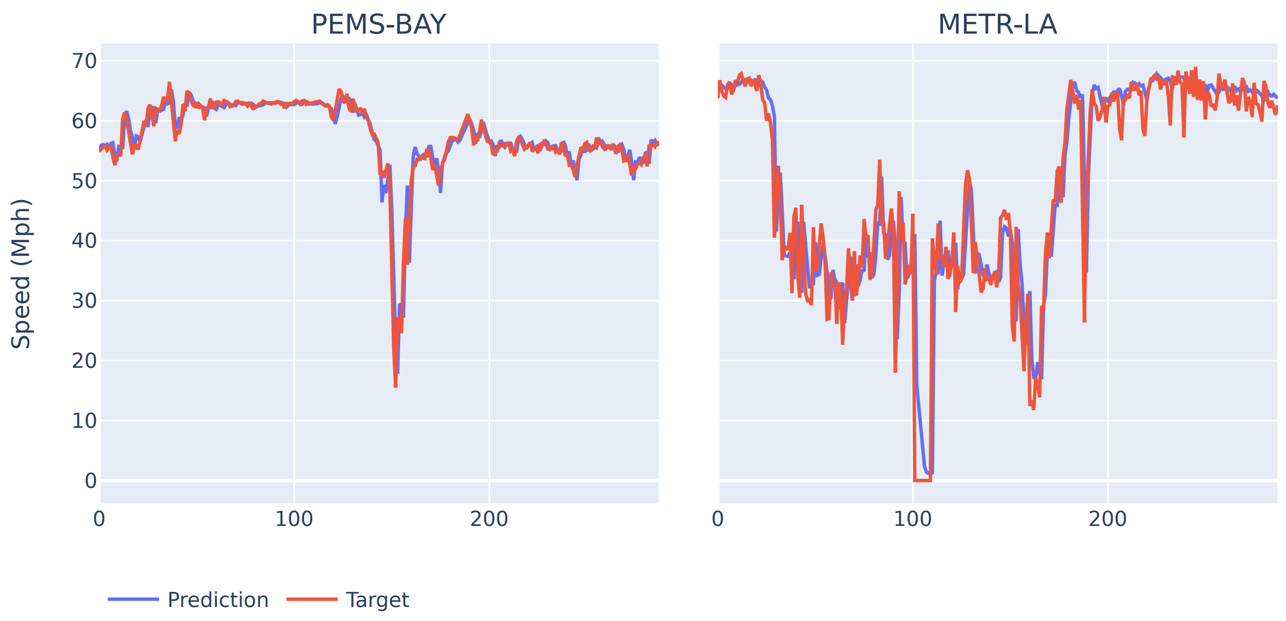


Figure 18: As NetMF had a solid performance on both datasets, we compared the NetMF-prediction and the target time series. This plot shows 288 consecutive target values in red (one day), and the forecast on the different datasets in blue. As seen in the images, on METR-LA our approach can't give back the smaller peaks.

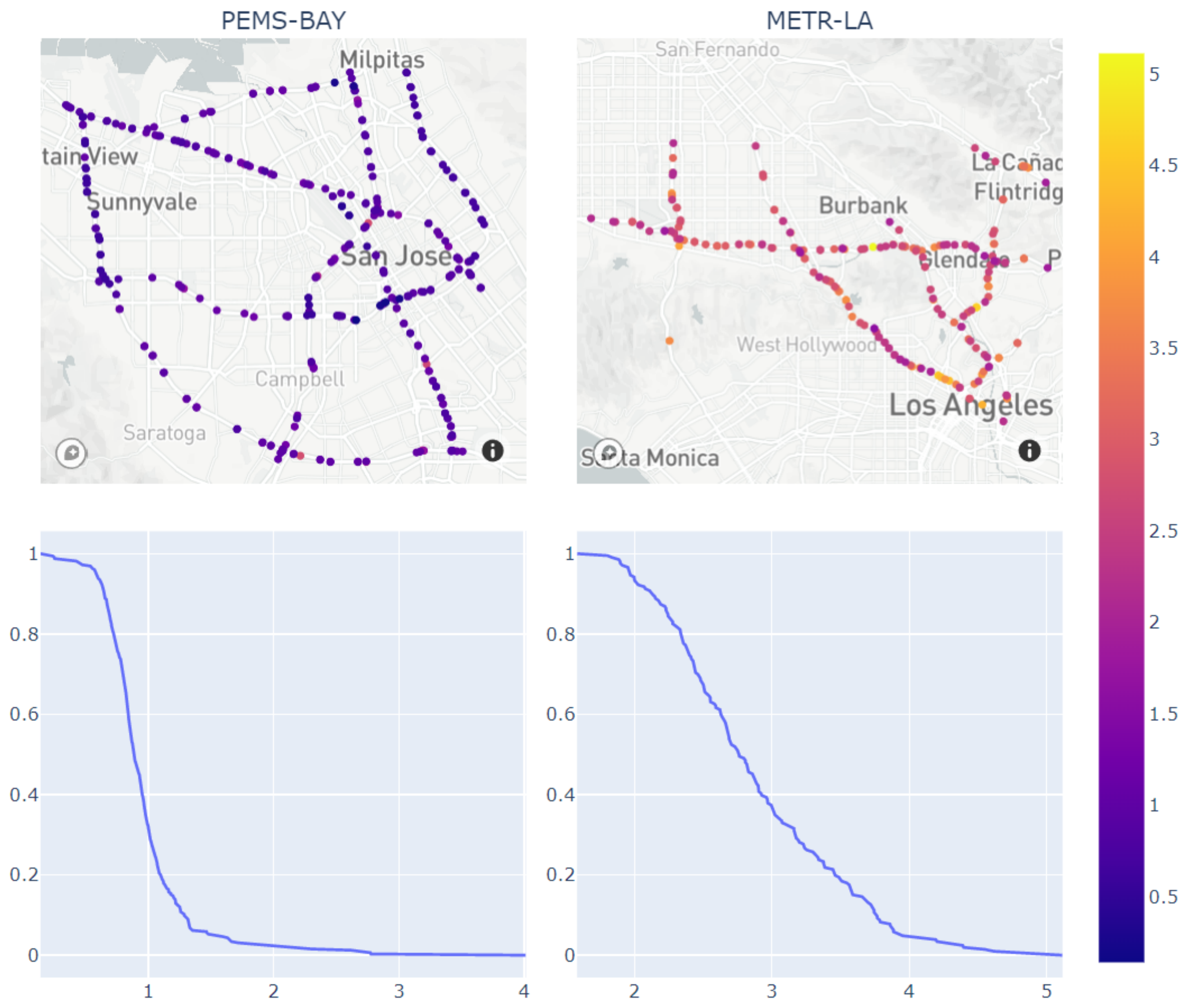


Figure 19: The two maps show the MAE values by node on the test sets of PEMS-BAY and METR-LA. The line plots show the distribution of the MAE values in the case of both datasets.

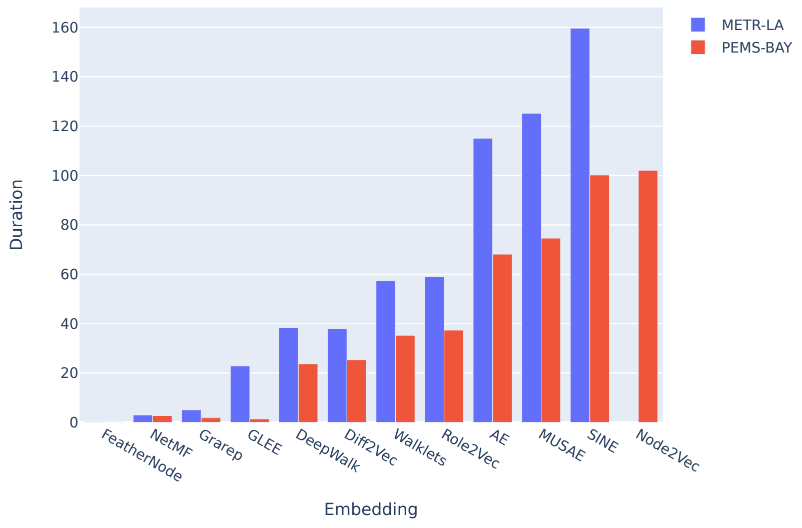


Figure 20: The average embedding times on the two datasets.

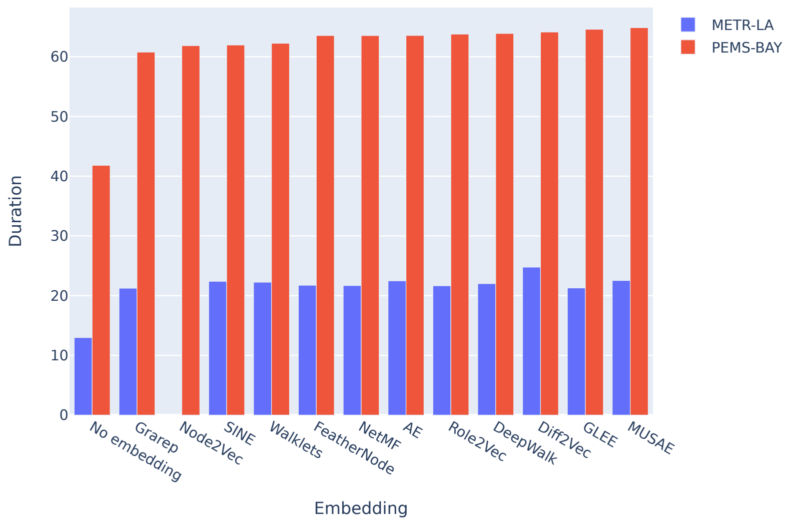


Figure 21: The average epoch times on the two datasets.

6 Summary

In this thesis, I examined the task of traffic forecasting. The two datasets that I used, METR-LA and PEMS-BAY [11], are considered benchmark datasets for this task. After formulating the problem as a multivariate time series forecasting task, my approach was to embed the underlying sensor adjacency matrix into a vector space using existing graph embedding algorithms and train neural networks with this representation of the sensor graph to make predictions.

Using the nature of the dataset, I defined features for the nodes, and with this, I was able to train attributed embeddings and compare them with structural embeddings. I found that this graph embedding-based approach is very dataset-dependent, this is especially true for the attributed embeddings because the features are hand-crafted and there's no guarantee that they are universal.

On the METR-LA dataset, this approach leads to bad results as the information about the sensor graph gives very little improvement. The structural embeddings can't capture meaningful relations between the nodes and the defined features are not enough to enhance the attributed embeddings. The two best-performing embeddings are SINE [24] and NetMF [15] but the results with them are far from other, graph neural network-based methods.

On the PEMS-BAY dataset, I achieved better results, the performance of SINE and NetMF are even comparable to methods like Graph WaveNet [23], a state-of-the-art algorithm from 2019. With increased embedding dimensions the results are even better, in terms of MAE, NetMF has a similar performance as MegaCRN [8] while the RMSE is lower.

In summary, there are algorithms among the existing embeddings which are capable of capturing the nature of a road network and are useful to embed this type of graphs. NetMF has a great performance and the algorithm only operates with the adjacency matrix while SINE needs additional features but as shown in Section 5, in some cases these features can be obtained from the dataset.

References

[1] Nesreen K. Ahmed et al. *Learning Role-based Graph Embeddings.* 2018. arXiv: 1802.02896 [stat.ML].

[2] Ross Girshick et al. *Rich feature hierarchies for accurate object detection and semantic segmentation.* 2014. arXiv: 1311.2524 [cs.CV].

[3] Alex Graves, Abdel rahman Mohamed, and Geoffrey Hinton. *Speech Recognition with Deep Recurrent Neural Networks.* 2013. arXiv: 1303.5778 [cs.NE].

[4] Aditya Grover and Jure Leskovec. *node2vec: Scalable Feature Learning for Networks.* 2016. arXiv: 1607.00653 [cs.SI].

[5] Dan Hendrycks and Kevin Gimpel. *Gaussian Error Linear Units (GELUs).* 2016. DOI: 10.48550/ARXIV.1606.08415. URL: https://arxiv.org/abs/1606.08415.

[6] Sepp Hochreiter. "The Vanishing Gradient Problem During Learning Recurrent Neural Nets and Problem Solutions". In: *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 6 (Apr. 1998), pp. 107–116. DOI: 10.1142/S0218488598000094.

[7] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-term Memory". In: *Neural computation* 9 (Dec. 1997), pp. 1735–80. DOI: 10.1162/neco.1997.9.8.1735.

[8] Renhe Jiang et al. *Spatio-Temporal Meta-Graph Learning for Traffic Forecasting.* 2023. arXiv: 2211.14701 [cs.LG].

[9] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Commun. ACM* 60.6 (2017), 84–90. ISSN: 0001-0782. DOI: 10.1145/3065386. URL: https://doi.org/10.1145/3065386.

[10] Yann LeCun et al. "Handwritten Digit Recognition with a Back-Propagation Network". In: *NIPS*. 1989.

[11] Yaguang Li et al. *Diffusion Convolutional Recurrent Neural Network: Data-Driven Traffic Forecasting.* 2018. arXiv: 1707.01926 [cs.LG].

[12] Tomas Mikolov et al. *Efficient Estimation of Word Representations in Vector Space.* 2013. arXiv: 1301.3781 [cs.CL].

[13] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. “DeepWalk”. In: *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2014. DOI: 10.1145/2623330.2623732. URL: https://doi.org/10.1145%2F2623330.2623732.

[14] Bryan Perozzi et al. *Don’t Walk, Skip! Online Learning of Multi-scale Network Embeddings*. 2017. arXiv: 1605.02115 [cs.SI].

[15] Jiezhong Qiu et al. “Network Embedding as Matrix Factorization”. In: *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*. ACM, 2018. DOI: 10.1145/3159652.3159706. URL: https://doi.org/10.1145%2F3159652.3159706.

[16] Benedek Rozemberczki, Carl Allen, and Rik Sarkar. *Multi-scale Attributed Node Embedding*. 2021. arXiv: 1909.13021 [cs.LG].

[17] Benedek Rozemberczki and Rik Sarkar. *Characteristic Functions on Graphs: Birds of a Feather, from Statistical Descriptors to Parametric Models*. 2020. arXiv: 2005.07959 [cs.LG].

[18] Benedek Rozemberczki and Rik Sarkar. *Fast Sequence-Based Embedding with Diffusion Graphs*. 2020. arXiv: 2001.07463 [cs.LG].

[19] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. “Learning representations by back-propagating errors”. In: *Nature* 323 (1986), pp. 533–536.

[20] Mingxing Tan and Quoc V. Le. *EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks*. 2020. arXiv: 1905.11946 [cs.LG].

[21] Viet Tra et al. “Bearing fault diagnosis under variable speed using convolutional neural networks and the stochastic diagonal levenberg-marquardt algorithm”. In: *Sensors* 17.12 (2017), p. 2834.

[22] P.J. Werbos. “Backpropagation through time: what it does and how to do it”. In: *Proceedings of the IEEE* 78.10 (1990), pp. 1550–1560. DOI: 10.1109/5.58337.

[23] Zonghan Wu et al. *Graph WaveNet for Deep Spatial-Temporal Graph Modeling*. 2019. arXiv: 1906.00121 [cs.LG].

[24] Daokun Zhang et al. *SINE: Scalable Incomplete Network Embedding*. 2018. arXiv: 1810.06768 [cs.SI].