Chapter 15
Dynamic Graph Neural Networks

Seyed Mehran Kazemi

Abstract The world around us is composed of entities that interact and form relations with each other. This makes graphs an essential data representation and a crucial building-block for machine learning applications; the nodes of the graph correspond to entities and the edges correspond to interactions and relations. The entities and relations may evolve; e.g., new entities may appear, entity properties may change, and new relations may be formed between two entities. This gives rise to dynamic graphs. In applications where dynamic graphs arise, there often exists important information within the evolution of the graph, and modeling and exploiting such information is crucial in achieving high predictive performance. In this chapter, we characterize various categories of dynamic graph modeling problems. Then we describe some of the prominent extensions of graph neural networks to dynamic graphs that have been proposed in the literature. We conclude by reviewing three notable applications of dynamic graph neural networks namely skeleton-based human activity recognition, traffic forecasting, and temporal knowledge graph completion.

15.1 Introduction

Traditionally, machine learning models were developed to make predictions about entities (or objects or examples) given only their features and irrespective of their connections with the other entities in the data. Examples of such prediction tasks include predicting the political party a social network user supports given their other features, predicting the topic of a publication given its text, predicting the type of the object in an image given the image pixels, and predicting the traffic in a road (or road segment) given historical traffic data in that road.

Seyed Mehran Kazemi
Borealis AI, e-mail: mehran.kazemi@borealisai.com

In many applications, there exist relationships between the entities that can be exploited to make better predictions about them. As a few examples, social network users that are close friends or family members are more likely to support the same political party, two publications by the same author are more likely to have the same topic, two images taken from the same website (or uploaded to social media by the same user) are more likely to have similar objects in them, and two roads that are connected are more likely to have similar traffic volumes. The data for these applications can be represented in the form of a graph where nodes correspond to entities and edges correspond to the relationships between these entities.

Graphs arise naturally in many real-world applications including recommender systems, biology, social networks, ontologies, knowledge graphs, and computational finance. In some domains the graph is static, i.e. the graph structure and the node features are fixed over time. In other domains, the graph changes over time. In a social network, for example, new edges are added when people make new friends, existing edges are removed when people stop being friends, and node features change as people change their attributes, e.g., when they change their career assuming that career is one of the node features. In this chapter, we focus on the domains where the graph is dynamic and changes over time.

In applications where dynamic graphs arise, modeling the evolution of the graph is often crucial in making accurate predictions. Over the years, several classes of machine learning models have been developed that capture the structure and the evolution of dynamic graphs. Among these classes, extensions of graph neural networks (GNNs) (Scarselli et al, 2008; Kipf and Welling, 2017b) to dynamic graphs have recently found success in several domains and they have become one of the essential tools in the machine learning toolbox. In this chapter, we review the GNN approaches for dynamic graphs and provide several application domains where dynamic GNNs have provided striking results. The chapter is not meant to be a full survey of the literature but rather a description of the common techniques for applying GNNs to dynamic graphs. For a comprehensive survey of representation learning approaches for dynamic graphs we refer the reader to (Kazemi et al, 2020), and for a more specialized survey of GNN-based approaches to dynamic graphs we refer the reader to (Skarding et al, 2020).

The rest of the chapter is organized as follows. In Section 15.2, we define the notation that will be used throughout the chapter and provide the necessary background to follow the rest of the chapter. In Section 15.3, we describe different types of dynamic graphs and different prediction problems on these graphs. In Section 15.4, we review several approaches for applying GNNs on dynamic graphs. In Section 15.5, we review some of the applications of dynamic GNNs. Finally, Section 18.8 summarizes and concludes the chapter.

15.2 Background and Notation

In this section, we define our notation and provide the background required to follow the rest of the chapter.

We use lowercase letters z to denote scalars, bold lowercase letters $\boldsymbol{z}$ to denote vectors and uppercase letters $\boldsymbol{Z}$ to denote matrices. $\boldsymbol{z}_i$ denotes the i element of $\boldsymbol{z}$, $\boldsymbol{Z}_i$ denotes a column vector corresponding to the i row of $\boldsymbol{Z}$, and $\boldsymbol{Z}_{i,j}$ denotes the element at the i row and j column of $\boldsymbol{Z}$. $\boldsymbol{z}$ denotes the transpose of $\boldsymbol{z}$ and $\boldsymbol{Z}$ denotes the transpose of $\boldsymbol{Z}$. $(\boldsymbol{z}\boldsymbol{z}') \in \mathbb{R}^{d+d'}$ corresponds to the concatenation of $\boldsymbol{z} \in \mathbb{R}^{d}$ and $\boldsymbol{z}' \in \mathbb{R}^{d'}$. We use to represent an identity matrix. We use $\odot$ to denote element-wise (Hadamard) product. We represent a sequence as $[e_1, e_2, \ldots, e_k]$ and a set as $\{e_1, e_2, \ldots, e_k\}$ where e_is represent the elements in the sequence or set.

In this chapter, we mainly consider attributed graphs. We represent an *attributed graph* as $G = (V, \boldsymbol{A}, \boldsymbol{X})$ where $V = \{v_1, v_2, \ldots, v_n\}$ is the set of vertices (aka nodes), $n = |V|$ denotes the number of nodes, $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ is an *adjacency matrix*, and $\boldsymbol{X} \in \mathbb{R}^{n \times d}$ is a feature matrix where $\boldsymbol{X}_i$ represents the features associated with the i node v_i and d denotes the number of features. If there exists no edge between v_i and v_j, then $\boldsymbol{A}_{i,j} = 0$; otherwise, $\boldsymbol{A}_{i,j} \in \mathbb{R}_+$ represents the weight of the edge where $\mathbb{R}_+$ represents positive real numbers.

If G is *unweighted*, then the range of $\boldsymbol{A}$ is $\{0,1\}$ (i.e. $\boldsymbol{A} \in \{0,1\}^{n \times n}$). G is *undirected* if the edges have no directions; it is *directed* if the edges have directions. For an undirected graph, $\boldsymbol{A}$ is symmetric (i.e. $\boldsymbol{A} = \boldsymbol{A}$). For each edge $\boldsymbol{A}_{i,j} > 0$ of a directed graph, we call v_i the source and v_j the target of the edge. If G is *multi-relational* with a set $R = \{r_1, \ldots, r_m\}$ of relations, then the graph has m adjacency matrices where the i adjacency matrix represents the existence of the i relation r_i between the nodes.

15.2.1 Graph Neural Networks

In this chapter, we use the term *Graph Neural Network (GNN)* to refer to the general class of neural networks that operate on graphs through message-passing between the nodes. Here, we provide a brief description of GNNs.

Let $G = (V, \boldsymbol{A}, \boldsymbol{X})$ be a static attributed graph. A GNN is a function $f : \mathbb{R}^{n \times n} \times \mathbb{R}^{n \times d} \to \mathbb{R}^{n \times d'}$ that takes G (or more specifically $\boldsymbol{A}$ and $\boldsymbol{X}$) as input and provides as output a matrix $\boldsymbol{Z} \in \mathbb{R}^{n \times d'}$ where $\boldsymbol{Z}_i \in \mathbb{R}^{d'}$ corresponds to a hidden representation for the i node v_i. This hidden representation is called the *node embedding*. Providing a node embedding for each node v_i can be viewed as dimensionality reduction where the information from v_i's initial features as well as the information from its connectivity to other nodes and the features of these nodes are captured in a vector $\boldsymbol{Z}_i$. This vector can be used to make informed predictions about v_i. In what follows, we describe two example GNNs namely *graph convolutions networks* and *graph attention networks* for undirected graphs.

Graph Convolutional Networks: Graph convolutional networks (GCNs) (Kipf and Welling, 2017b) stack multiple layers of graph convolution. The l layer of GCN for an undirected graph $G = (V, \boldsymbol{A}, \boldsymbol{X})$ can be formulated as follows:

$$\boldsymbol{Z}^{(l)} = \sigma(\boldsymbol{D}^{-\frac{1}{2}}\tilde{\boldsymbol{A}}\boldsymbol{D}^{-\frac{1}{2}}\boldsymbol{Z}^{(l-1)}\boldsymbol{W}^{(l)}) \tag{15.1}$$

where $\tilde{\boldsymbol{A}} = \boldsymbol{A}+$ corresponds to the adjacency matrix with self-loops, $\boldsymbol{D}$ is a diagonal degree matrix with $\boldsymbol{D}_{i,i} = \tilde{\boldsymbol{A}}_i\mathbf{1}$ ($\mathbf{1}$ represents a column vector of ones) and $\boldsymbol{D}_{i,j} = 0$ for $i \neq j$, $\boldsymbol{D}^{-\frac{1}{2}}\tilde{\boldsymbol{A}}\boldsymbol{D}^{-\frac{1}{2}}$ corresponds to a row and column normalization of $\tilde{\boldsymbol{A}}$, $\boldsymbol{Z}^{(l)} \in \mathbb{R}^{n\times d^{(l)}}$ and $\boldsymbol{Z}^{(l-1)} \in \mathbb{R}^{n\times d^{(l-1)}}$ represent the node embeddings in layer l and $(l-1)$ respectively with $\boldsymbol{Z}^{(0)} = \boldsymbol{X}$, $\boldsymbol{W}^{(l)} \in \mathbb{R}^{d^{(l-1)}\times d^{(l)}}$ represents the weight matrix at layer l, and σ is an activation function.

The l layer of a GCN model can be described in terms of the following steps. First, it applies a linear projection to the node embeddings $\boldsymbol{Z}^{(l-1)}$ using the weight matrix $\boldsymbol{W}^{(l)}$, then for each node v_i it computes a weighted sum of the projected embeddings of v_i and its neighbors where the weights for the weighted sum are specified according to $\boldsymbol{D}^{-\frac{1}{2}}\tilde{\boldsymbol{A}}\boldsymbol{D}^{-\frac{1}{2}}$, and finally it applies a non-linearity to the weighted sums and updates the node embeddings. Notice that in a L-layer GCN, the embedding for each node is computed based on its L-hop neighborhood (i.e. based on the nodes that are at most L hops away from it).

Graph Attention Networks: Instead of fixing the weights when computing a weighted sum of the neighbors, attention-based GNNs replace $\boldsymbol{D}^{-\frac{1}{2}}\tilde{\boldsymbol{A}}\boldsymbol{D}^{-\frac{1}{2}}$ in equation 15.1 with an attention matrix $\hat{\boldsymbol{A}}^{(l)} \in \mathbb{R}^{n\times n}$ such that:

$$\boldsymbol{Z}^{(l)} = \sigma(\hat{\boldsymbol{A}}^{(l)}\boldsymbol{Z}^{(l-1)}\boldsymbol{W}^{(l)}) \tag{15.2}$$

$$\hat{\boldsymbol{A}}^{(l)}_{i,j} = \frac{\boldsymbol{E}^{(l)}_{i,j}}{\sum_k \boldsymbol{E}^{(l)}_{i,k}}, \quad \boldsymbol{E}^{(l)}_{i,j} = \tilde{\boldsymbol{A}}_{i,j}\exp\big(\alpha(\boldsymbol{Z}^{(l-1)}_i, \boldsymbol{Z}^{(l-1)}_j; \theta^{(l)})\big) \tag{15.3}$$

where $\alpha : \mathbb{R}^{d^{(l-1)}} \times \mathbb{R}^{d^{(l-1)}} \to \mathbb{R}$ is a function with parameters $\theta^{(l)}$ that computes *attention weights* for pairs of nodes. Here, $\tilde{\boldsymbol{A}}$ acts as a mask that ensures $\boldsymbol{E}^{(l)}_{i,j} = 0$ (and consequently $\hat{\boldsymbol{A}}^{(l)}_{i,j} = 0$) if v_i and v_j are not connected. The exp function in the computation of $\boldsymbol{E}^{(l)}_{i,j}$ and the normalization $\frac{\boldsymbol{E}^{(l)}_{i,j}}{\sum_k \boldsymbol{E}^{(l)}_{i,k}}$ correspond to a (masked) softmax function of the attention weights. Different attention-based GNNs can be constructed with different choices of α. In graph attention networks (GATs) (Veličković et al, 2018), $\theta^{(l)} \in \mathbb{R}^{2d^{(l)}}$ and α is defined as follows:

$$\alpha(\boldsymbol{Z}^{(l-1)}_i, \boldsymbol{Z}^{(l-1)}_j; \theta^{(l)}) = \sigma\big(\theta^{(l)}(\boldsymbol{W}^{(l)}\boldsymbol{Z}^{(l-1)}_i \mid\mid \boldsymbol{W}^{(l)}\boldsymbol{Z}^{(l-1)}_j)\big) \tag{15.4}$$

where σ is an activation function. The formulation in equation 15.2 corresponds to a *single-head* attention-based GNN. A *multi-head* attention-based GNN computes multiple attention matrices $\hat{\boldsymbol{A}}^{(l,1)}, \ldots, \hat{\boldsymbol{A}}^{(l,\beta)}$ using equation 15.3 but with different weights $\theta^{(l,1)},\dots,\theta^{(l,\beta)}$ and $\boldsymbol{W}^{(l,1)},\dots,\boldsymbol{W}^{(l,\beta)}$ and then replaces equation 15.2 with:

$$\boldsymbol{Z}^{(l)} = \sigma(\hat{\boldsymbol{A}}^{(l,1)}\boldsymbol{Z}^{(l-1)}\boldsymbol{W}^{(l,1)} \mid\mid \dots \mid\mid \hat{\boldsymbol{A}}^{(l,\beta)}\boldsymbol{Z}^{(l-1)}\boldsymbol{W}^{(l,\beta)}) \tag{15.5}$$

where β is the number of heads. Each head may learn to aggregate the neighbors differently and extract different information.

15.2.2 Sequence Models

Over the years, several models have been proposed that operate on sequences. In this chapter, we are mainly interested in neural sequence models that take as input a sequence $[\boldsymbol{x}^{(1)},\boldsymbol{x}^{(2)},\dots,\boldsymbol{x}^{(\tau)}]$ of observations where $\boldsymbol{x}^{(t)} \in \mathbb{R}^d$ for all $t \in \{1,\dots,\tau\}$, and produce as output hidden representations $[\boldsymbol{h}^{(1)},\boldsymbol{h}^{(2)},\dots,\boldsymbol{h}^{(\tau)}]$ where $\boldsymbol{h}^{(t)} \in \mathbb{R}^{d'}$ for all $t \in \{1,\dots,\tau\}$. Here, τ represents the length of the sequence or the timestamp for the last element in the sequence. Each hidden representation $\boldsymbol{h}^{(t)}$ is a *sequence embedding* capturing information from the first t observations. Providing a sequence embedding for a given sequence can be viewed as dimensionality reduction where the information from the first t observations in the sequence is captured in a single vector $\boldsymbol{h}^{(t)}$ which can be used to make informed predictions about the sequence. In what follows, we describe *recurrent neural networks*, *Transformers*, and *convolutional neural networks* for sequence modeling.

Recurrent Neural Networks: Recurrent neural networks (RNNs) (Elman, 1990) and its variants have achieved impressive results on a range of sequence modeling problems. The core principle of the RNN is that its output is a function of the current data point as well as a representation of the previous inputs. Vanilla RNNs consume the input sequence one by one and provides embeddings using the following equation (applied sequentially for t in $[1,\dots,\tau]$):

$$\boldsymbol{h}^{(t)} = RNN(\boldsymbol{x}^{(t)},\boldsymbol{h}^{(t-1)}) = \sigma(\boldsymbol{W}^{(i)}\boldsymbol{x}^{(t)} + \boldsymbol{W}^{(h)}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}) \tag{15.6}$$

where $W^{(.)}$s and $\boldsymbol{b}$ are the model parameters, $\boldsymbol{h}^{(t)}$ is the hidden state corresponding to the embedding of the first t observations, and $\boldsymbol{x}^{(t)}$ is the t observation. One may initialize $\boldsymbol{h}^{(0)} = \boldsymbol{0}$, where $\boldsymbol{0}$ is a vector of 0s, or let $\boldsymbol{h}^{(0)}$ be learned during training. Training vanilla RNNs is typically difficult due to gradient vanishing and exploding.

Long short term memory (LSTMs) (Hochreiter and Schmidhuber, 1997) (and *gated recurrent units (GRUs)* (Cho et al, 2014a)) alleviate the training problem of vanilla RNNs through gating mechanism and additive operations. An LSTM model consumes the input sequence one by one and provides embeddings using the following equations:

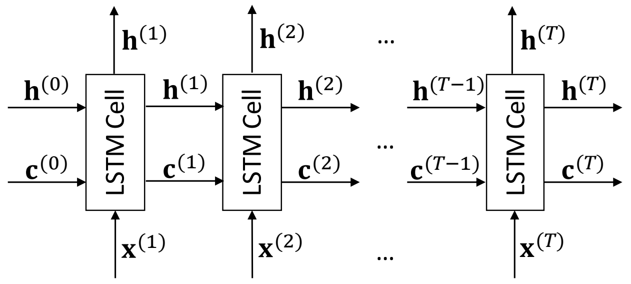


Fig. 15.1: An LSTM model taking as input a sequence $\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(\tau)}$ and producing hidden representations $\boldsymbol{h}^{(1)}, \boldsymbol{h}^{(2)}, \ldots, \boldsymbol{h}^{(\tau)}$ as output. Equations 15.7-15.11 describe the operations in *LSTM Cells*.

$$\boldsymbol{i}^{(t)} = \sigma\left(\boldsymbol{W}^{(ii)}\boldsymbol{x}^{(t)} + \boldsymbol{W}^{(ih)}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}^{(i)}\right) \tag{15.7}$$

$$\boldsymbol{f}^{(t)} = \sigma\left(\boldsymbol{W}^{(fi)}\boldsymbol{x}^{(t)} + \boldsymbol{W}^{(fh)}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}^{(f)}\right) \tag{15.8}$$

$$\boldsymbol{c}^{(t)} = \boldsymbol{f}^{(t)} \odot \boldsymbol{c}^{(t-1)} + \boldsymbol{i}^{(t)} \odot Tanh\left(\boldsymbol{W}^{(ci)}\boldsymbol{x}^{(t)} + \boldsymbol{W}^{(ch)}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}^{(c)}\right) \tag{15.9}$$

$$\boldsymbol{o}^{(t)} = \sigma\left(\boldsymbol{W}^{(oi)}\boldsymbol{x}^{(t)} + \boldsymbol{W}^{(oh)}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}^{(o)}\right) \tag{15.10}$$

$$\boldsymbol{h}^{(t)} = \boldsymbol{o}^{(t)} \odot Tanh\left(\boldsymbol{c}^{(t)}\right) \tag{15.11}$$

Here $\boldsymbol{i}^{(t)}$, $\boldsymbol{f}^{(t)}$, and $\boldsymbol{o}^{(t)}$ represent the input, forget and output gates respectively, $\boldsymbol{c}^{(t)}$ is the memory cell, $\boldsymbol{h}^{(t)}$ is the hidden state corresponding to the embedding of the sequence until t observation, σ is an activation function (typically Sigmoid), *Tanh* represents the hyperbolic tangent function, and $\boldsymbol{W}^{(\cdot\cdot)}$s and $\boldsymbol{b}^{(\cdot)}$s are weight matrices and vectors. Similar to vanilla RNNs, one may initialize $\boldsymbol{h}^{(0)} = \boldsymbol{c}^{(0)} = \mathbf{0}$ or let them be vectors with learnable parameters. Figure 15.1 shows an overview of an LSTM model.

A bidirectional RNN (BiRNN) (Schuster and Paliwal, 1997) is a combination of two RNNs one consuming the input sequence $[\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(\tau)}]$ in the forward direction and producing hidden representations $[\overrightarrow{\boldsymbol{h}}^{(1)}, \overrightarrow{\boldsymbol{h}}^{(2)}, \ldots, \overrightarrow{\boldsymbol{h}}^{(\tau)}]$ as output, and the other consuming the input sequence backwards (i.e. $[\boldsymbol{x}^{(\tau)}, \boldsymbol{x}^{(\tau-1)}, \ldots, \boldsymbol{x}^{(1)}]$) and producing hidden representations $[\overleftarrow{\boldsymbol{h}}^{(\tau)}, \overleftarrow{\boldsymbol{h}}^{(\tau-1)}, \ldots, \overleftarrow{\boldsymbol{h}}^{(1)}]$ as output. These two hidden representations are then concatenated producing a single hidden representation $\boldsymbol{h}^{(t)} = (\overrightarrow{\boldsymbol{h}}^{(t)}\overleftarrow{\boldsymbol{h}}^{(t)})$. Note that in RNNs, $\boldsymbol{h}^{(t)}$ is computed only based on observations at or before t whereas in BiRNNs, $\boldsymbol{h}^{(t)}$ is computed based on observations at, before, or after t. BiLSTMs Graves et al (2005) are a specific version of BiRNNs where the RNN is an LSTM.

Transformers: Consuming the input sequence one by one makes RNNs not amenable to parallelization. It also makes capturing long-range dependencies difficult. To solve these issues, the *Transformer* model Vaswani et al (2017) allows processing a sequence as a whole. The central operation in Transformer models is the self-attention mechanism. Let $\boldsymbol{H}^{(l-1)}$ be an embedding matrix in layer $(l-1)$ such that its t row $\boldsymbol{H}_t^{(l-1)}$ represents the embedding of the first t observations. The self-attention mechanism at each layer l can be described similar to equation 15.2 and equation 15.3 for attention-based GNNs by defining $\tilde{A}$ in equation 15.3 as a lower triangular matrix where $\tilde{A}_{i,j} = 1$ if $i \leq j$ and $\tilde{A}_{i,j} = 0$ otherwise, replacing $\boldsymbol{Z}^{(l)}$ and $\boldsymbol{Z}^{(l-1)}$ with $\boldsymbol{H}^{(l)}$ and $\boldsymbol{H}^{(l-1)}$, and defining the α function in equation 15.3 as follows:

$$\alpha(\boldsymbol{H}_t^{(l-1)}, \boldsymbol{H}_{t'}^{(l-1)}; \theta^{(l)}) = \frac{\boldsymbol{Q}_t \boldsymbol{K}_{t'}}{\sqrt{d^{(k)}}}, \boldsymbol{Q} = \boldsymbol{W}^{(l,Q)} \boldsymbol{H}^{(l-1)}, \boldsymbol{K} = \boldsymbol{W}^{(l,K)} \boldsymbol{H}^{(l-1)} \tag{15.12}$$

where $\theta^l = \{\boldsymbol{W}^{(l,Q)}, \boldsymbol{W}^{(l,K)}\}$ are the weights with $\boldsymbol{W}^{(l,Q)}, \boldsymbol{W}^{(l,K)} \in \mathbb{R}^{d^{(l-1)} \times d^{(k)}}$. The matrices $\boldsymbol{Q}$ and $\boldsymbol{K}$ are called the *query* and *key* matrices[1]. $\boldsymbol{Q}_t$ and $\boldsymbol{K}_{t'}$ represent column vectors corresponding to the t and t' th row of $\boldsymbol{Q}$ and $\boldsymbol{K}$, respectively. After L layers, the hidden representations $\boldsymbol{H}^{(L)}$ contain the sequence embeddings with $\boldsymbol{H}_t^{(L)}$ corresponding to the embedding of the first t observations (denoted as $\boldsymbol{h}^{(t)}$ for RNNs). The lower-triangular matrix $\tilde{A}$ ensures that the embedding $\boldsymbol{H}_t^{(L)}$ is computed based only on the observations at and before the t observation. One may define $\tilde{A}$ as a matrix of all 1s to allow $\boldsymbol{H}_t^{(L)}$ to be computed based on the observations at, before, and after the t observation (similar to BiRNNs).

In equation 15.12, the embeddings are updated based on an aggregation of the embeddings from the previous timestamps, but the order of these embeddings is not modeled explicitly. To enable taking the order into account, the embeddings in the Transformer model are initialized as $\boldsymbol{H}_t^{(0)} = \boldsymbol{x}^{(t)} + \boldsymbol{p}^{(t)}$ or $\boldsymbol{H}_t^{(0)} = (\boldsymbol{x}^{(t)} \,||\, \boldsymbol{p}^{(t)})$ where $\boldsymbol{H}_t^{(0)}$ is the t row of $\boldsymbol{H}^{(0)}$, $\boldsymbol{x}^{(t)}$ is the t observation, and $\boldsymbol{p}^{(t)}$ is a positional encoding capturing information about the position of the observation in the sequence. In the original work, the positional encodings are defined as follows:

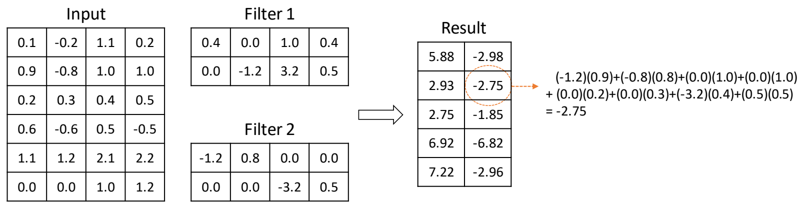
$$\boldsymbol{p}_{2i}^{(t)} = \sin(t/10000^{2i/d}), \quad \boldsymbol{p}_{2i+1}^{(t)} = \sin(t/10000^{2i/d} + \pi/2) \tag{15.13}$$

Note that $\boldsymbol{p}^{(t)}$ is constant and does not change during training.

Convolutional Neural Networks: Convolutional neural networks (CNNs) (Le Cun et al, 1989) have revolutionized many computer vision applications. Originally, CNNs were proposed for 2D signals such as images. They were later used for 1D signals such as sequences and time-series. Here, we describe 1D CNNs. We start with describing 1D convolutions. Let $\boldsymbol{H} \in \mathbb{R}^{n \times d}$ be a matrix and $\boldsymbol{F} \in \mathbb{R}^{u \times d}$ be a convolution filter. Applying the filter $\boldsymbol{F}$ on $\boldsymbol{H}$ produces a vector $\boldsymbol{h}' \in \mathbb{R}^{n-u+1}$ as follows:

$$\boldsymbol{h}'_i = \sum_{j=1}^{u} \sum_{k=1}^{d} \boldsymbol{H}_{i+j-1,k} \boldsymbol{F}_{j,k} \tag{15.14}$$

[1] For readers familiar with Transformers, in our description the *values* matrix corresponds to the multiplication of the embedding matrix with the weight matrix $\boldsymbol{W}^{(l)}$ in equation 15.2.



Input

0.1	-0.2	1.1	0.2
0.9	-0.8	1.0	1.0
0.2	0.3	0.4	0.5
0.6	-0.6	0.5	-0.5
1.1	1.2	2.1	2.2
0.0	0.0	1.0	1.2

Filter 1

0.4	0.0	1.0	0.4
0.0	-1.2	3.2	0.5

Filter 2

-1.2	0.8	0.0	0.0
0.0	0.0	-3.2	0.5

Result

5.88	-2.98
2.93	-2.75
2.75	-1.85
6.92	-6.82
7.22	-2.96

Fig. 15.2: An example of a 1D convolution operation with two convolution filters.

It is also possible to produce a vector $\boldsymbol{h}' \in \mathbb{R}^n$ (i.e. a vector whose dimension is the same as the first dimension of $\boldsymbol{H}$) by padding $\boldsymbol{H}$ with zeros. Having d' convolution filters, one can generate d' vectors as in equation 15.14 and stack them to generate a matrix $\boldsymbol{H}' \in \mathbb{R}^{(n-u+1)\times d'}$ (or $\boldsymbol{H}' \in \mathbb{R}^{n\times d'}$). Figure 15.2 provides an example of 1D convolution.

The 1D convolution operation in equation 15.14 is the main building block of the 1D CNNs. Similar to equation 15.12, let us assume $\boldsymbol{H}^{(l-1)}$ represents the embeddings in the l layer with $\boldsymbol{H}_t^{(0)} = \boldsymbol{x}^{(t)}$ where $\boldsymbol{H}_t^{(0)}$ represents the t row of $\boldsymbol{H}^{(0)}$ and $\boldsymbol{x}^{(t)}$ is the t observation. 1D CNN models apply multiple convolution filters to $\boldsymbol{H}^{(l-1)}$ as described above and produce a matrix to which activation and (sometimes) pooling operations are applied to produce $\boldsymbol{H}^{(l)}$. The convolution filters are the learnable parameters of the model. Hereafter, we use the term CNN to refer to the general family of 1D convolutional neural networks.

15.2.3 Encoder-Decoder Framework and Model Training

A deep neural network model can typically be decomposed into an encoder and a decoder module. The encoder module takes the input and provides vector-representations (or embeddings), and the decoder module takes the embeddings and provides predictions. The GNNs and sequence models described in Sections 15.2.1 and 15.2.2 correspond to the encoder modules of a full model; they provide node embeddings $\boldsymbol{Z}$ and sequence embeddings $\boldsymbol{H}$, respectively. The decoder is typically task-specific. As an example, for a node classification task, the decoder can be a feed-forward neural network applied on a node embedding $\boldsymbol{Z}_i$ provided by the encoder, followed by a softmax function. Such a decoder provides as output a vector $\hat{\boldsymbol{y}} \in \mathbb{R}^{|C|}$ where C represents the classes, $|C|$ represents the number of classes, and $\hat{\boldsymbol{y}}_j$ shows the probability of the node belonging to the j class. A similar decoder can be used for sequence classification. As another example, for a link prediction problem, the decoder can take as input the embeddings for two nodes, take the sigmoid of a dot-product of the two node embeddings, and use the produced number as the probability of an edge existing between the two nodes.

The parameters of a model are learned through optimization by minimizing a task-specific loss function. For a classification task, for instance, we typically assume having access to a set of ground-truth labels $\boldsymbol{Y}$ where $\boldsymbol{Y}_{i,j} = 1$ if the i example belongs to the j class and $\boldsymbol{Y}_{i,j} = 0$ otherwise. We learn the parameters of the model by minimizing (e.g., using stochastic gradient descent) the cross entropy loss defined as follows:

$$L = -\frac{1}{|\boldsymbol{Y}_{i,j}|} \sum_i \sum_j \boldsymbol{Y}_{i,j} log(\hat{\boldsymbol{Y}}_{i,j}) \tag{15.15}$$

where $|\boldsymbol{Y}_{i,j}|$ denotes the number of rows in $\boldsymbol{Y}_{i,j}$ corresponding to the number of labeled examples, and $\hat{\boldsymbol{Y}}_{i,j}$ is the probability of the i example belonging to the j class according to the model. For other tasks, one may use other appropriate loss functions.

15.3 Categories of Dynamic Graphs

Different applications give rise to different types of dynamic graphs and different prediction problems. Before commencing the model development, it is crucial to identify the type of dynamic graph and its static and evolving parts, and have a clear understanding of the prediction problem. In what follows, we describe some general categories of dynamic graphs, their evolution types, and some common prediction problems for them.

15.3.1 Discrete vs. Continues

As pointed out in (Kazemi et al, 2020), dynamic graphs can be divided into discrete-time and continuous-time categories. Here, we describe the two categories and point out how discrete-time can be considered a specific case of continuous-time dynamic graphs.

A discrete-time dynamic graph (DTDG) is a sequence $[G^{(1)}, G^{(2)}, \dots, G^{(\tau)}]$ of graph snapshots where each $G^{(t)} = (V^{(t)}, \boldsymbol{A}^{(t)}, \boldsymbol{X}^{(t)})$ has vertices $V^{(t)}$, adjacency matrix $\boldsymbol{A}^{(t)}$ and feature matrix $\boldsymbol{X}^{(t)}$. DTDGs mainly appear in applications where (sensory) data is captured at regularly-spaced intervals.

Example 15.1. Figure 15.3 shows three snapshots of an example DTDG. In the first snapshot, there are three nodes. In the next snapshot, a new node v_4 is added and a connection is formed between this node and v_2. Furthermore, the features of v_1 are updated. In the third snapshot, a new edge has been added between v_3 and v_4.

A special type of DTDGs is the spatio-temporal graphs where a set of entities are spatially (i.e. in terms of closeness in space) and temporally correlated and data is captured at regularly-spaced intervals. An example of such a spatio-temporal graph is traffic data in a city or a region where traffic statistics at each road are computed at regularly-spaced intervals; the traffic at a particular road at time t is correlated with

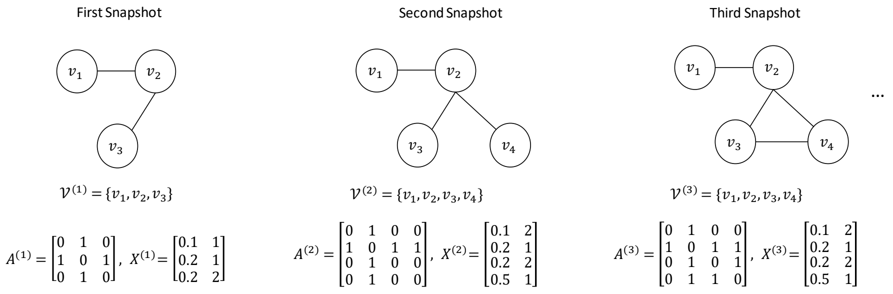

First Snapshot
Second Snapshot
Third Snapshot
v_1
v_2
v_3
v_4
...
$\mathcal{V}^{(1)} = \{v_1, v_2, v_3\}$
$\mathcal{V}^{(2)} = \{v_1, v_2, v_3, v_4\}$
$\mathcal{V}^{(3)} = \{v_1, v_2, v_3, v_4\}$
$A^{(1)} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}, X^{(1)} = \begin{bmatrix} 0.1 & 1 \\ 0.2 & 1 \\ 0.2 & 2 \end{bmatrix}$
$A^{(2)} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}, X^{(2)} = \begin{bmatrix} 0.1 & 2 \\ 0.2 & 1 \\ 0.2 & 2 \\ 0.5 & 1 \end{bmatrix}$
$A^{(3)} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}, X^{(3)} = \begin{bmatrix} 0.1 & 2 \\ 0.2 & 1 \\ 0.2 & 2 \\ 0.5 & 1 \end{bmatrix}$

Fig. 15.3: Three snapshots of an example DTDG. In the first snapshot, there are 3 nodes. In the second snapshot, a new node v_4 is added and a connection is formed between this node and v_2. Moreover, the features of v_1 are updated. In the third snapshot, a new edge has been added between v_3 and v_4.

the traffic at the roads connected to it at time t (spatial correlation) as well as the traffic at this roads and the ones connected to it at previous timestamps (temporal correlation). In this example, the nodes in each $G^{(t)}$ may represent roads (or road segments), the adjacency matrix $\boldsymbol{A}^{(t)}$ may represent how the roads are connected, and the feature matrix $\boldsymbol{X}^{(t)}$ may represent the traffic statistics in each road at time t.

A *continuous-time dynamic graph (CTDG)* is a pair $(G^{(t_0)}, O)$ where $G^{(t_0)} = (V^{(t_0)}, \boldsymbol{A}^{(t_0)}, \boldsymbol{X}^{(t_0)})$ is a static graph[2] representing an initial state at time t_0 and O is a sequence of temporal observations/events. Each observation is a tuple of the form $(event\ type, event, timestamp)$ where *event type* can be a node or edge addition, node or edge deletion, node feature update, etc., *event* represents the actual event that happened, and *timestamp* is the time at which the event occurred.

Example 15.2. An example of a CTDG is a pair $(G^{(t_0)}, O)$ where $G^{(t_0)}$ is the graph in the first snapshot of Figure 15.3 and the observations are as follows:

$$O = [(add\ node, v_4, \text{20-05-2020}), (add\ edge, (v_2, v_4), \text{21-05-2020}),$$
$$(Feature\ update, (v_1, [0.1, 2]), \text{28-05-2020}), (add\ edge, (v_3, v_4), \text{04-06-2020})]$$

where, e.g., $(add\ node, v_4, \text{20-05-2020})$ is an observation corresponding to a new node v_4 being added to the graph at time 20-05-2020.

At any point $t \geq t_0$ in time, a snapshot $G^{(t)}$ (corresponding to a static graph) can be obtained from a CTDG by updating $G^{(t_0)}$ sequentially according to the observations O that occurred before (or at) time t. In some cases, multiple edges may have been added between two nodes giving rise to multi-graphs; one may aggregate the edges to convert the multi-graph into a simple graph if required. Therefore, a DTDG can be viewed as a special case of a CTDG where only some regularly spaced snapshots of the CTDG are available.

[2] Note that we can have $V^{(t_0)} = \{\}$ corresponding to a graph with no nodes. We can also have $\boldsymbol{A}_{i,j}^{(t_0)} = 0$ for all i, j corresponding to a graph with no edges.

Example 15.3. For the CTDG in Example 15.2, assume t_0 = 01-05-2020 and we only observe the state of the graph at the first day of each month (01-05-2020, 01-06-2020 and 01-07-2020 for this example). In this case, the CTDG will reduce to the DTDG snapshots in Figure 15.3.

15.3.2 Types of Evolution

For both DTDGs and CTDGs, various parts of the graph may change and evolve. Here, we describe some of the main types of evolution. As a running example, we use a dynamic graph corresponding to a social network where the nodes represent *users* and the edges represent connections such as *friendship*.

Node addition/deletion: In our running example, new users may join the platform resulting in new nodes being added to the graph, and some users may leave the platform resulting in some nodes being removed from the graph.

Feature update: Users may have multiple features such as *age*, *country of residence*, *occupation*, etc. These features may change over time as users become older, move to a new country, or change their occupation.

Edge addition/deletion: As time goes by, some users become friends resulting in new edges and some people stop being friends resulting in some edges being removed from the graph. As pointed out in (Trivedi et al, 2019), the observations corresponding to events between two nodes may be categorized into *association* and *communication* events. The former corresponds to events that lead to structural changes in the graph and result in a long-lasting flow of information between the nodes (e.g., the formation of new friendships in social networks). The latter corresponds to events that result in a temporary flow of information between nodes (e.g., the exchange of messages in a social network). These two event categories typically evolve at different rates and one may model them differently, especially in applications where they are both present.

Edge weight updates: The adjacency matrix corresponding to the friendships may be weighted where the weights represent the strength of the friendships (e.g., computed based on the duration of friendship or other features). In this case, the strength of the friendships may change over time resulting in edge weight updates.

Relation updates: The edges between the users may be labeled where the label indicates the type of the connection, e.g., *friendship*, *engagement*, and *siblings*. In this case, the relation between two users may change over time (e.g., it may change from *friendship* to *engagement*). One may see relation update as a special case of edge evolution where one edge is deleted and another edge is added (e.g., the *friendship* edge is removed and an *engagement* edge is added).

15.3.3 Prediction Problems, Interpolation, and Extrapolation

We review four types of prediction problems for dynamic graphs: node classification/regression, graph classification, link prediction, and time prediction. Some of these problems can be studied under two settings: interpolation and extrapolation. They can also be studied under a transductive or inductive prediction setting. In what follows, we will describe each prediction problem. We let be a (discrete-time or continuous-time) dynamic graph containing information in a time interval $[t_0, \tau]$.

Node classification/regression: Let $V^{(t)} = \{v_1, \ldots, v_n\}$ represent the nodes in at time t. Node classification at time t is the problem of classifying a node $v_i \in V^{(t)}$ into a predefined set of classes C. Node regression at time t is the problem of predicting a continuous feature for a node $v_i \in V^{(t)}$. In the extrapolation setting, we make predictions about a future state (i.e. $t \geq \tau$) and the predictions are made based on the observations before or at t (e.g., forecasting the weather for the upcoming days). In the interpolation setting, $t_0 \leq t \leq \tau$ and the predictions are made based on all the observations (e.g., filling the missing values).

Graph classification: Let $\{1, 2, \ldots, \text{k}\}$ be a set of dynamic graphs. Graph classification is the problem of classifying each dynamic graph i into a predefined set of classes C.

Link prediction: Link prediction is the problem of predicting new links between the nodes of a dynamic graph. In the case of interpolation, the goal is to predict if there was an edge between two nodes v_i and v_j at timestamp $t_0 \leq t \leq \tau$ (or a time interval between t_0 and τ), assuming that v_i and v_j are in at time t. The interpolation problem is also known as the *completion* problem and can be used to predict missing links. In the case of extrapolation, the goal is to predict if there is going to be an edge between two nodes v_i and v_j at a timestamp $t > \tau$ (or a time interval after τ) assuming that v_i and v_j are in the at time τ.

Time prediction: Time prediction is the problem of predicting when an event happened or when it will happen. In the case of interpolation (sometimes called *temporal scoping*), the goal is to predict the time $t_0 \leq t \leq \tau$ when an event occurred (e.g., when two nodes v_i and v_j started or ended their connection). In the extrapolation case (sometimes called *time to event prediction*), the goal is to predict the time $t > \tau$ when an event will happen (e.g., when a connection will be formed between v_i and v_j).

Transductive vs. Inductive: The above problem definitions for node classification/regression, link prediction, and time prediction correspond to a transductive setting in which at the test time, predictions are to be made for entities already observed during training. In the inductive setting, information about previously unseen entities (or entirely new graphs) is provided at the test time and predictions are to be made for these entities (see (Hamilton et al, 2017b; Xu et al, 2020a; Albooyeh et al, 2020) for examples). The graph classification task is inductive by nature as it requires making predictions for previously unseen graphs at the test time.

15.4 Modeling Dynamic Graphs with Graph Neural Networks

In Section 15.2.1, we described how applying a GNN on a static graph G provides an embedding matrix $\boldsymbol{Z} \in \mathbb{R}^{n \times d'}$ where n is the number of nodes, d' is the embedding dimension, and $\boldsymbol{Z}_i$ represents the embedding for the i entity v_i and can be used to make predictions about it. For dynamic graphs, we wish to extend GNNs to obtain embeddings $\boldsymbol{Z}^{(t)} \in \mathbb{R}^{n_t \times d'}$ for any timestamp t, where n_t is the number of nodes in the graph at time t and $\boldsymbol{Z}_i^{(t)}$ captures the information about the i entity at time t. In this section, we review several such extensions of GNNs. We mainly describe the encoder part of the models for dynamic graphs as the decoder and the loss functions can be defined similarly to Section 15.2.3.

15.4.1 Conversion to Static Graphs

A simple but sometimes effective approach for applying GNNs on dynamic graphs is to first convert the dynamic graph into a static graph and then apply a GNN on the resulting static graph. The main benefits of this approach include simplicity as well as enabling the use of a wealth of GNN models and techniques for static graphs. One disadvantage with this approach, however, is the potential loss of information. In what follows, we describe two conversion approaches.

Temporal aggregation: We start with describing temporal aggregation for a particular type of dynamic graphs and then explain how it extends to more general cases. Consider a DTDG $[G^{(1)}, G^{(2)}, \dots, G^{(\tau)}]$ where each $G^{(t)} = (V^{(t)}, \boldsymbol{A}^{(t)}, \boldsymbol{X}^{(t)})$ such that $V^{(1)} = \cdots = V^{(\tau)} = V$ and $\boldsymbol{X}^{(1)} = \cdots = \boldsymbol{X}^{(\tau)} = \boldsymbol{X}$ (i.e. the nodes and their features are fixed over time and only the adjacency matrix evolves). Note that in this case, the adjacency matrices have the same shape. One way to convert this DTDG into a static graph is through a weighted aggregation of the adjacency matrices as follows:

$$\boldsymbol{A}^{(agg)} = \sum_{t=1}^{\tau} \phi(t, \tau) \boldsymbol{A}^{(t)} \tag{15.16}$$

where $\phi : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ provides the weight for the t adjacency matrix as a function of t and τ. For extrapolation problems, a common choice for ϕ is $\phi(t, \tau) = \exp(-\theta(\tau - t))$ corresponding to exponentially decaying the importance of the older adjacency matrices (Yao et al, 2016). Here, θ is a hyperparameter controlling how fast the importance decays. For interpolation problems where a prediction is to be made for a timestamp $1 \leq t' \leq \tau$, one may define the function as $\phi(t, t') = \exp(-\theta|t' - t|)$ corresponding to exponentially decaying the importance of the adjacency matrices as they move further away from t'. Through this aggregation, one can convert the DTDG above into a static graph $G = (V, \boldsymbol{A}^{(agg)}, \boldsymbol{X})$ and subsequently apply a static GNN model on it to make predictions. It is important to note that the aggregated adjacency matrix is weighted (i.e. $\boldsymbol{A}^{(agg)} \in \mathbb{R}^{n \times n}$) so one can only use the GNN models that can handle weighted graphs.

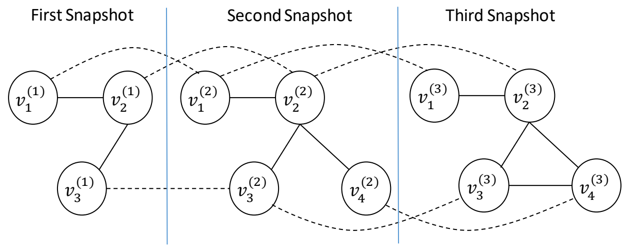


Fig. 15.4: An example of converting a DTDG into a static graph through temporal unrolling. Solid lines represent the edges in the graph at different timestamps and dashed lines represent the added edges. In this example, each node is connected to the node corresponding to the same entity only in the previous timestamp (i.e. $\omega = 1$).

In the case where node features also evolve, one may use a similar aggregation as in equation 15.16 and compute $\boldsymbol{X}^{(agg)}$ based on $[\boldsymbol{X}^{(1)}, \boldsymbol{X}^{(2)}, \dots, \boldsymbol{X}^{(\tau)}]$. In the case where nodes are added and removed, one possible way of aggregation is as follows. Let $V^{(s)} = \{v \mid v \in V^{(1)} \cup \dots \cup V^{(\tau)}\}$ represent the set of all the nodes that existed throughout time. We can expand every $\boldsymbol{A}^{(t)}$ to a matrix in $\mathbb{R}^{|V^{(s)}| \times |V^{(s)}|}$ where the values for the rows and columns corresponding to any node $v \notin V^{(t)}$ are all 0s. The feature vectors can be expanded similarly. Then, equation 15.16 can be applied on the expanded adjacency and feature matrices. A similar aggregation can be done for CTDGs by first converting it into a DTDG (see Section 15.3.1) and then applying equation 15.16.

Example 15.4. Consider a DTDG with the three snapshots in Figure 15.3. We let $V^{(s)} = \{v_1, v_2, v_3, v_4\}$, add a row and a column of zeros to $\boldsymbol{A}^{(1)}$, and add a row of zeros to $\boldsymbol{X}^{(1)}$. Then, we use equation 15.16 with some value of θ to compute $\boldsymbol{A}^{(agg)}$ and $\boldsymbol{X}^{(agg)}$. Then we apply a GNN on the aggregated graph.

Temporal unrolling: Another way of converting a dynamic graph into a static graph is unrolling the dynamic graph and connecting the nodes corresponding to the same object across time. Consider a DTDG $[G^{(1)}, G^{(2)}, \dots, G^{(\tau)}]$ and let $G^{(t)} = (V^{(t)}, \boldsymbol{A}^{(t)}, \boldsymbol{X}^{(t)})$ for $t \in \{1, \dots, \tau\}$. Let $G^{(s)} = (V^{(s)}, \boldsymbol{A}^{(s)}, \boldsymbol{X}^{(s)})$ represent the static graph to be generated from the DTDG. We let $V^{(s)} = \{v^{(t)} \mid v \in V^{(t)}, t \in \{1, \dots, \tau\}\}$. That is, every node $v \in V^{(t)}$ at every timestamp $t \in \{1, \dots, \tau\}$ becomes a new node named $v^{(t)}$ in $V^{(s)}$ (so $|V^{(s)}| = \sum_{t=1}^{\tau} |V^{(t)}|$). Note that this is different from the way we constructed $V^{(s)}$ for temporal aggregation: here every node at every timestamp becomes a node in $V^{(s)}$ whereas in temporal aggregation we took a union of the nodes across timestamps. For every node $v^{(t)} \in V^{(s)}$, we let the features of $v^{(t)}$ in $\boldsymbol{X}^{(s)}$ to be the same as its features in $\boldsymbol{X}^{(t)}$. If two nodes $v_i, v_j \in V^{(t)}$ are connected according to $\boldsymbol{A}^{(t)}$, we connect the corresponding nodes in $\boldsymbol{A}^{(s)}$. We also connect each node $v^{(t)}$ to $v^{(t')}$ for $t' \in \{max(1, t-\omega), \dots, t-1\}$ so a node corresponding to an entity at time t becomes connected to the nodes corresponding to the same entity at the previous ω timestamps, where ω is a hyperparameter. One may assign different weights to these temporal edges in $\boldsymbol{A}^{(s)}$ based on the difference between t and t' (e.g., exponentially decaying the weight). Having constructed the static graph $G^{(s)}$, one may apply a GNN model on it and, e.g., use the resulting embedding for $v^{(t)}$s (i.e. the nodes corresponding to the t timestamp of the DTDG) to make predictions about the nodes.

Example 15.5. Figure 15.4 provides an example of temporal unrolling for the DTDG in Figure 15.3 with $\omega = 1$. The graph has 11 nodes overall and so $\boldsymbol{A}^{(s)} \in \mathbb{R}^{11\times 11}$. The node features are set according to the ones in Figure 15.3, e.g., the feature values for $v_1^{(2)}$ are 0.1 and 2.

15.4.2 Graph Neural Networks for DTDGs

One natural way of developing models for DTDGs is by combining GNNs with sequence models; the GNN captures the information within the node connections and the sequence model captures the information within their evolution. A large number of the works on dynamic graphs in the literature follow this approach – see, e.g., (Seo et al, 2018; Manessi et al, 2020; Xu et al, 2019a). Here, we describe some generic ways of combining GNNs with sequence models.

GNN-RNN: Let be a DTDG with a sequence $[G^{(1)},\dots,G^{(\tau)}]$ of snapshots where $G^{(t)} = (V^{(t)}, \boldsymbol{A}^{(t)}, \boldsymbol{X}^{(t)})$ for each $t \in \{1,\dots,\tau\}$. Suppose we want to obtain node embeddings at some time $t \leq \tau$ based on the observations at or before t. For simplicity, let us assume $V^{(1)} = V^{(2)} = \dots = V^{(\tau)} = V$, i.e. the nodes are the same throughout time (in cases where the nodes change, one may use a similar strategy as in Example 15.4).

We can apply a GNN to each of the $G^{(t)}$s and obtain a hidden representation matrix $\boldsymbol{Z}^{(t)}$ whose rows correspond to node embeddings. Then, for the i node v_i, we obtain a sequence of embeddings $[\boldsymbol{Z}_i^{(1)}, \boldsymbol{Z}_i^{(2)}, \dots, \boldsymbol{Z}_i^{(\tau)}]$. These embeddings do not yet contain temporal information. To incorporate the temporal aspect of the DTDG into the embeddings and obtain a temporal embedding for v_i at time t, we can feed the sequence $[\boldsymbol{Z}_i^{(1)}, \boldsymbol{Z}_i^{(2)}, \dots, \boldsymbol{Z}_i^{(t)}]$ into an RNN model defined in equation 27.1 by replacing $\boldsymbol{x}^{(t)}$ with $\boldsymbol{Z}_i^{(t)}$ and using the hidden representation of the RNN model as the temporal node embedding for v_i. The temporal embedding for other nodes can be obtained similarly by feeding their sequence of embeddings produced by the GNN model to the same RNN model. The following formulae describe a variant of the GNN-RNN model where the GNN is a GCN (defined in equation 15.1), the RNN is an LSTM model, and the LSTM operations are applied to all nodes embeddings at the same time (the formulae are applied sequentially for t in $[1,2,\dots,\tau]$).

$$\boldsymbol{Z}^{(t)} = GCN(\boldsymbol{X}^{(t)}, \boldsymbol{A}^{(t)}) \tag{15.17}$$

$$\boldsymbol{I}^{(t)} = \sigma\left(\boldsymbol{Z}^{(t)}\boldsymbol{W}^{(ii)} + \boldsymbol{H}^{(t-1)}\boldsymbol{W}^{(ih)} + \boldsymbol{b}^{(i)}\right) \tag{15.18}$$

$$\boldsymbol{F}^{(t)} = \sigma\left(\boldsymbol{Z}^{(t)}\boldsymbol{W}^{(fi)} + \boldsymbol{H}^{(t-1)}\boldsymbol{W}^{(fh)} + \boldsymbol{b}^{(f)}\right) \tag{15.19}$$

$$\boldsymbol{C}^{(t)} = \boldsymbol{F}^{(t)} \odot \boldsymbol{C}^{(t-1)} + \boldsymbol{I}^{(t)} \odot Tanh\left(\boldsymbol{Z}^{(t)}\boldsymbol{W}^{(ci)} + \boldsymbol{H}^{(t-1)}\boldsymbol{W}^{(ch)} + \boldsymbol{b}^{(c)}\right) \tag{15.20}$$

$$\boldsymbol{O}^{(t)} = \sigma\left(\boldsymbol{Z}^{(t)}\boldsymbol{W}^{(oi)} + \boldsymbol{H}^{(t-1)}\boldsymbol{W}^{(oh)} + \boldsymbol{b}^{(o)}\right) \tag{15.21}$$

$$\boldsymbol{H}^{(t)} = \boldsymbol{O}^{(t)} \odot Tanh\left(\boldsymbol{C}^{(t)}\right) \tag{15.22}$$

where, similar to equations 15.7-15.11, $\boldsymbol{I}^{(t)}$, $\boldsymbol{F}^{(t)}$, and $\boldsymbol{O}^{(t)}$ represent the input, forget and output gates for the nodes respectively, $\boldsymbol{C}^{(t)}$ is the memory cell, $\boldsymbol{H}^{(t)}$ is the hidden state corresponding to the node embeddings for the first t observation, and $\boldsymbol{W}^{(..)}$s and $\boldsymbol{b}^{(.)}$s are weight matrices and vectors. In the above formulae, when we add a matrix $\boldsymbol{Z}^{(t)}\boldsymbol{W}^{(.i)} + \boldsymbol{H}^{(t-1)}\boldsymbol{W}^{(.h)}$ with a bias vector $\boldsymbol{b}^{(.)}$, we assume the bias vector $\boldsymbol{b}^{(.)}$ as added to every row of the matrix. $\boldsymbol{H}^{(0)}$ and $\boldsymbol{C}^{(0)}$ can be initialized with zeros or learned from the data. $\boldsymbol{H}^{(t)}$ corresponds to the temporal node embeddings at time t and can be used to make predictions about them. We can summarize the equations above into:

$$\boldsymbol{Z}^{(t)} = GCN(\boldsymbol{X}^{(t)}, \boldsymbol{A}^{(t)}) \tag{15.23}$$

$$\boldsymbol{H}^{(t)}, \boldsymbol{C}^{(t)} = LSTM(\boldsymbol{Z}^{(t)}, \boldsymbol{H}^{(t-1)}, \boldsymbol{C}^{(t-1)}) \tag{15.24}$$

In a similar way, one can construct other variations of the GNN-RNN model such as GCN-GRU, GAT-LSTM, GAT-RNN, etc. Figure 15.5 provides an overview of the GCN-LSTM model.

RNN-GNN: In cases where the graph structure is fixed through time (i.e. $\boldsymbol{A}^{(1)} = \dots = \boldsymbol{A}^{(\tau)} = \boldsymbol{A}$) and only node features change, instead of first applying a GNN model and then applying a sequence model to obtain temporal node embeddings, one may apply the sequence model first to capture the temporal evolution of the node features and then apply a GNN model to capture the correlations between the nodes. We can create different variations of this generic model by using different GNN and sequence models (e.g., LSTM-GCN, LSTM-GAT, GRU-GCN, etc.). The formulation for a LSTM-GCN model is as follows:

$$\boldsymbol{H}^{(t)}, \boldsymbol{C}^{(t)} = LSTM(\boldsymbol{X}^{(t)}, \boldsymbol{H}^{(t-1)}, \boldsymbol{C}^{(t-1)}) \tag{15.25}$$

$$\boldsymbol{Z}^{(t)} = GCN(\boldsymbol{H}^{(t)}, \boldsymbol{A}) \tag{15.26}$$

with $\boldsymbol{Z}^{(t)}$ containing the temporal node embeddings at time t. Note that RNN-GNN is only appropriate if the the adjacency matrix is fixed over time; otherwise, RNN-GNN fails to capture the information within the evolution of the graph structure.

GNN-BiRNN and BiRNN-GNN: In the case of GNN-RNN and RNN-GNN, the obtained node embeddings $\boldsymbol{H}^{(t)}$ contain information about the observations at

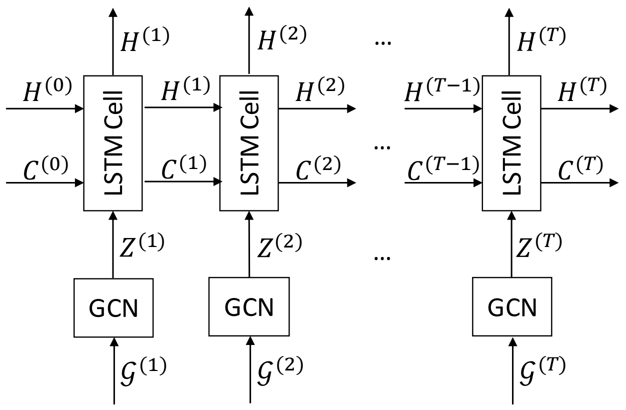


Fig. 15.5: The GCN-LSTM model taking a sequence $G^{(1)}, G^{(2)}, \ldots, G^{(\tau)}$ as input and producing hidden representations $\boldsymbol{H}^{(1)}, \boldsymbol{H}^{(2)}, \ldots, \boldsymbol{H}^{(\tau)}$ as output. The operations in *LSTM Cells* are described in equations 15.18-15.22. The GCN modules have shared parameters.

or before time t. This is appropriate for extrapolation problems. For interpolation problems (e.g., when we want to predict missing links between edges at a timestamp $t \le \tau$), however, we may want to use the observations before, at, or after time t. One possible way of achieving this is by combining a GNN with a BiRNN so that the BiRNN provides information from not only the observations at or before time t but also after time t.

GNN-Transformer: Combining GNNs with Transformers can be done in a similar way as in GNN-RNNs. We apply a GNN to each of the $G^{(t)}$s and obtain a hidden representation matrix $\boldsymbol{Z}^{(t)}$ whose rows correspond to node embeddings. Then for the i entity v_i, we create a matrix $\boldsymbol{H}^{(0,i)}$ such that $\boldsymbol{H}_t^{(0,i)} = \boldsymbol{Z}_i^{(t)} + \boldsymbol{p}^{(t)}$ (or $\boldsymbol{H}_t^{(0,i)} = \boldsymbol{Z}_i^{(t)} \boldsymbol{p}^{(t)}$) where $\boldsymbol{p}^{(t)}$ is the positional encoding vector for position t. That is, the t row of $\boldsymbol{H}^{(0,i)}$ contains the embedding $\boldsymbol{Z}_i^{(t)}$ of v_i obtained by applying the GCN model on $G^{(t)}$, plus the positional encoding. The 0 superscript in $\boldsymbol{H}^{(0,i)}$ shows that $\boldsymbol{H}^{(0,i)}$ corresponds to the input of a Transformer model in the 0 layer. Once we have $\boldsymbol{H}^{(0,i)}$, we can apply an L-layer Transformer model (see equations 15.2, 15.3 and 15.12) to obtain $\boldsymbol{H}^{(L,i)}$ where $\boldsymbol{H}_t^{(L,i)}$ corresponds to the temporal embedding of v_i at time t. For extrapolation, the matrix $\tilde{\boldsymbol{A}}$ in equation 15.3 is a lower triangular matrix with $\tilde{\boldsymbol{A}}_{i,j} = 1$ if $i \le j$ and 0 otherwise; for interpolation, $\tilde{\boldsymbol{A}}$ is a matrix of all 1s. The GCN-Transformer variant of the GNN-Transformer model can be described using the following equations:

$$\boldsymbol{Z}^{(t)} = GCN(\boldsymbol{X}^{(t)},\ \boldsymbol{A}^{(t)})\ for\ t \in \{1,2,\ldots,\tau\} \tag{15.27}$$

$$\boldsymbol{H}_t^{(0,i)} = \boldsymbol{Z}_i^{(t)} + \boldsymbol{p}^{(t)}\ for\ t \in \{1,2,\ldots,\tau\},\ i \in \{1,2,\ldots,|V|\} \tag{15.28}$$

$$\boldsymbol{H}^{(L,i)} = Transformer(\boldsymbol{H}^{(0,i)},\ \tilde{\boldsymbol{A}})\ for\ i \in \{1,2,\ldots,|V|\} \tag{15.29}$$

GNN-CNN: In a similar way as GNN-RNN and GNN-Transformer, one can combine GNNs with CNNs where the GNN provides $[\boldsymbol{Z}^{(1)}, \boldsymbol{Z}^{(2)}, \ldots, \boldsymbol{Z}^{(t)}]$, then the embeddings $[\boldsymbol{Z}_i^{(1)}, \boldsymbol{Z}_i^{(2)}, \ldots, \boldsymbol{Z}_i^{(t)}]$ for each node v_i are stacked into a matrix $\boldsymbol{H}^{(0,i)}$ similar to the GNN-Transformer model, and then a 1D CNN model is applied on $\boldsymbol{H}^{(0,i)}$ (see Section 15.2.2) to provide the final node embeddings.

Creating Deeper Models: Consider the GCN-LSTM model in Figure 15.5. The output of the GCN module is a sequence $[\boldsymbol{Z}^{(1)}, \boldsymbol{Z}^{(2)}, \ldots, \boldsymbol{Z}^{(\tau)}]$ and the outputs of the LSTM module is a sequence of hidden representation matrices $[\boldsymbol{H}^{(1)}, \boldsymbol{H}^{(2)}, \ldots, \boldsymbol{H}^{(\tau)}]$. Let us call the output of the GCN module as $[\boldsymbol{Z}^{(1,1)}, \boldsymbol{Z}^{(1,2)}, \ldots, \boldsymbol{Z}^{(1,\tau)}]$ and the output of the LSTM module as $[\boldsymbol{H}^{(1,1)}, \boldsymbol{H}^{(1,2)}, \ldots, \boldsymbol{H}^{(1,\tau)}]$ where the added superscript 1 indicates that these are the hidden representations created at layer 1. One may consider each $\boldsymbol{H}^{(1,\,t)}$ as the new node features for the nodes in $G^{(t)}$ and run a GCN module (with separate parameters from the initial GCN) again to obtain $[\boldsymbol{Z}^{(2,1)}, \boldsymbol{Z}^{(2,2)}, \ldots, \boldsymbol{Z}^{(2,\tau)}]$. Then, another LSTM module may operate on these matrices to produce $[\boldsymbol{H}^{(2,1)}, \boldsymbol{H}^{(2,2)}, \ldots, \boldsymbol{H}^{(2,\tau)}]$. Stacking L of these GCN-LSTM blocks produces $[\boldsymbol{H}^{(L,1)}, \boldsymbol{H}^{(L,2)}, \ldots, \boldsymbol{H}^{(L,\tau)}]$ as output. These hidden matrices can then be used for making predictions about the nodes. The l layer of this model can be formulated as below (the formulae are applied sequentially for t in $[1,\ldots,\tau]$):

$$\boldsymbol{Z}^{(l,\,t)} = GCN(\boldsymbol{H}^{(l-1,\,t)}, \boldsymbol{A}^{(t)}) \tag{15.30}$$

$$\boldsymbol{H}^{(l,\,t)}, \boldsymbol{C}^{(l,\,t)} = LSTM(\boldsymbol{Z}^{(l,\,t)}, \boldsymbol{H}^{(l,\,t-1)}, \boldsymbol{C}^{(l,\,t-1)}) \tag{15.31}$$

where $\boldsymbol{H}^{(0,t)} = \boldsymbol{X}^{(t)}$ for $t \in \{1,\ldots,\tau\}$. The above two equations define what is called a *GCN-LSTM block*. Other blocks can be constructed using similar combinations.

15.4.3 Graph Neural Networks for CTDGs

Recently, developing models that operate on CTDGs without converting them to DTDGs (or converting them to static graphs) has been the subject of several studies. One class of models for CTDGs is based on extensions of the sequence models described in Section 15.2.2, especially RNNs. The general idea behind these models is to consume the observations sequentially and update the embedding of a node whenever a new observation is made about that node (or, in some works, about one of its neighbors). Before describing GNN-based approaches for CTDGs, we briefly describe some of the RNN-based models for CTDGs.

Consider a CTDG with $G^{(t_0)} = (V^{(t_0)}, \boldsymbol{A}^{(t_0)}, \boldsymbol{X}^{(t_0)})$ with $\boldsymbol{A}_{i,j}^{(t_0)} = 0$ for all i, j (i.e. no initial edges) and observations O whose only type is edge additions. Since the only observation types are edge additions, for this CTDG, the nodes and their features are fixed over time. Let $\boldsymbol{Z}^{(t-)}$ represent the node embeddings right before time t (initially, $\boldsymbol{Z}^{(t_0)} = \boldsymbol{X}^{(t_0)}$ or $\boldsymbol{Z}^{(t_0)} = \boldsymbol{X}^{(t_0)}\boldsymbol{W}$ where $\boldsymbol{W}$ is a weight matrix with learnable parameters). Upon making an observation $(AddEdge, (v_i, v_j), t)$ corresponding to a new directed edge between two nodes $v_i, v_j \in V$, the model developed in (Kumar et al, 2019b) updates the embeddings for v_i and v_j as follows:

$$\boldsymbol{Z}_i^{(t)} = RNN_{source}((\boldsymbol{Z}_j^{(t-)} \mid\mid \Delta t_i \mid\mid \boldsymbol{f}),\ \boldsymbol{Z}_i^{(t-)}) \tag{15.32}$$

$$\boldsymbol{Z}_j^{(t)} = RNN_{target}((\boldsymbol{Z}_i^{(t-)} \mid\mid \Delta t_j \mid\mid \boldsymbol{f}),\ \boldsymbol{Z}_j^{(t-)}) \tag{15.33}$$

where RNN_{source} and RNN_{target} are two RNNs with different weights[3], Δt_i and Δt_j represent the time elapsed since v_i's and v_j's previous interactions respectively[4], $\boldsymbol{f}$ represents a vector of features corresponding to edge features (if any), $\mid\mid$ indicates concatenation, and $\boldsymbol{Z}_i^{(t)}$ and $\boldsymbol{Z}_j^{(t)}$ represent the updated embeddings at time t. The first RNN takes as input a new observation $(\boldsymbol{Z}_j^{(t-)} \mid\mid \Delta t_i \mid\mid \boldsymbol{f})$ and the previous hidden state of a node $\boldsymbol{Z}_i^{(t-)}$ and provides an updated representation (similarly for the second RNN). Besides learning a temporal embedding $\boldsymbol{Z}^{(t)}$ as described above, in (Kumar et al, 2019b) another embedding vector is also learned for each entity that is fixed over time and captures the static features of the nodes. The two embeddings are then concatenated to produce the final embedding that is used for making predictions.

In Trivedi et al (2017), a similar strategy is followed to develop a model for CTDGs with multi-relational graphs in which two custom RNNs update the node embeddings for the source and target nodes once a new labeled edge is observed between them. In Trivedi et al (2019), a model is developed that is similar to the above models but closer in nature to GNNs. Upon making an observation $(AddEdge, (v_i, v_j), t)$, the node embedding for v_i is updated as follows (and similarly for v_j):

$$\boldsymbol{Z}_i^{(t)} = RNN((\boldsymbol{z}_{\mathcal{N}}(v_j)\Delta t_i),\ \boldsymbol{Z}_i^{(t-)}) \tag{15.34}$$

where $\boldsymbol{z}_{\mathcal{N}(v_j)}$ is an embedding that is computed based on a custom attention-weighted aggregation of the embeddings of v_j and its neighbors at time t, and Δt_i is defined similarly as in equation 15.32. Unlike equation 15.32 where the RNN updates the embedding of v_i based on the embedding of v_j alone, in equation 15.34 the embedding of v_i is updated based on an aggregation of the embeddings from the first-order neighborhood of v_j which makes it close in nature to GNNs.

Many of the existing RNN-based approaches for CTDGs only compute the node embeddings based on their immediate neighboring nodes (or nodes that are 1-hop

[3] The reason for using two RNNs is to allow the source and target nodes of a directed graph to be updated differently upon making the observation $(AddEdge, (v_i, v_j), t)$. If the graph is undirected, one may use a single RNN.

[4] If this is the first interaction of v_i (or v_j), then Δt_i (or Δt_j) can be the time elapsed since t_0.

away from them) and do not take into account the nodes that are multi-hops away. We now describe a GNN-based model for CTDGs named *temporal graph attention networks (TGAT)* and developed in (Xu et al, 2020a) that computes node embeddings based on the k-hop neighborhood of the nodes (i.e. based on the nodes that are at most k hops away). Being a GNN-based model, TGAT can learn embeddings for new nodes that are added to a graph and can be used for inductive settings where at the test time, predictions are to be made for previously unseen nodes.

Similar to the Transformer model, TGAT removes the recurrence and instead relies on self-attention and an extension of positional encoding to continuous time encoding named Time2Vec. In Time2Vec (Kazemi et al, 2019), time t (or a delta of time as in equation 15.32 and equation 15.34) is represented as a vector $\boldsymbol{z}^{(t)}$ defined as follows:

$$\boldsymbol{z}_i^{(t)} = \begin{cases} \omega_i t + \varphi_i, & \text{if } \ i = 0. \\ \sin(\omega_i t + \varphi_i), & \text{if } \ 1 \leq i \leq k. \end{cases} \tag{15.35}$$

where ω and φ are vectors with learnable parameters. TGAT uses a specific case of Time2Vec where the linear term is removed and the parameters φ are fixed to 0s and $\frac{\pi}{2}$s similar to equation 15.13. We refer the reader to Kazemi et al (2019); Xu et al (2020a) for theoretical and practical motivations of such a time encoding.

Now we describe how TGAT computes node embeddings. For a node v_i and timestamp t, let $\mathcal{N}_i^{(t)}$ represent the set of nodes that interacted with v_i at or before time t and the timestamps for the interaction. Each element of $\mathcal{N}_i^{(t)}$ is of the form (v_j, t_k) where $t_k \leq t$. The l layer of TGAT computes the embedding $\boldsymbol{h}^{(t,l,i)}$ for v_i at time t in layer l using the following steps:

1. For any node v_i, $\boldsymbol{h}^{(t,0,i)}$ (corresponding to the embedding of v_i in the 0 layer in time t) is assumed to be equal to $\boldsymbol{X}_i$ for any value of t.
2. A matrix $\boldsymbol{K}^{(t,l,i)}$ with $|\mathcal{N}_i^{(t)}|$ rows is created such that for each $(v_j, t_k) \in \mathcal{N}_i^{(t)}$, $\boldsymbol{K}^{(t,l,i)}$ has a row $(\boldsymbol{h}^{(t_k,l-1,j)} \mid\mid \boldsymbol{z}^{(t-t_k)})$ where $\boldsymbol{h}^{(t_k,l-1,j)}$ corresponds to the embedding of v_j in layer $(l-1)$ at the time t_k of its interaction with v_i and $\boldsymbol{z}^{(t-t_k)}$ is an encoding for the delta time $(t-t_k)$ as in equation 15.35. Note that each $\boldsymbol{h}^{(t_k,l-1,j)}$ is computed recursively using the same steps outlined here.
3. A vector $\boldsymbol{q}^{(t,l,i)}$ is computed as $(\boldsymbol{h}^{(t,l-1,i)} \boldsymbol{z}^{(0)})$ where $\boldsymbol{h}^{(t,l-1,i)}$ is the embedding of v_i at time t in layer $(l-1)$ and $\boldsymbol{z}^{(0)}$ is an encoding for a delta of time equal to 0 as in equation 15.35.
4. $\boldsymbol{q}^{(t,l,i)}$ is used to determine how much v_i should attend to each row of $\boldsymbol{K}^{(t,l,i)}$ corresponding to the representation of its neighbors[5]. Attention weights $\boldsymbol{a}^{(t,l,i)}$ are computed using equation 15.12 where the j element of $\boldsymbol{a}^{(t,l,i)}$ is computed as $\boldsymbol{a}_j^{(t,l,i)} = \alpha(\boldsymbol{q}^{(t,l,i)}, \boldsymbol{K}_j^{(t,l,i)}; \theta^{(l)})$.
5. Having the attention weights, a representation $\tilde{\boldsymbol{h}}^{(t,l,i)}$ is computed for v_i using equation 15.2 where the attention matrix $\hat{\boldsymbol{A}}^{(l)}$ is replaced with the attention vector $\boldsymbol{a}^{(t,l,i)}$.

[5] For simplicity, here we describe a single-head attention-based GNN version of TGAT; in the original work, a multi-head version is used (see equation 15.5 for details.)

6. Finally, $\boldsymbol{h}^{(t,l,i)} = FF^{(l)}(\boldsymbol{h}^{(t,l-1,i)}\tilde{\boldsymbol{h}}^{(t,l,i)})$ computes the representation for node v_i at time t in layer l where $FF^{(l)}$ is a feed-forward neural network in layer l.

An L-layer TGAT model computes node embeddings based on the L-hop neighborhood of a node.

Suppose we run a 2-layer TGAT model on a temporal graph where v_i interacted with v_j at time $t_1 < t$ and v_j interacted with v_k at time $t_2 < t_1$. The embedding $\boldsymbol{h}^{(t,2,i)}$ is computed based on the embedding $\boldsymbol{h}^{(t_1,1,j)}$ which is itself computed based on the embedding $\boldsymbol{h}^{(t_2,0,k)}$. Since we are now at 0 layer, $\boldsymbol{h}^{(t_2,0,k)}$ in TGAT is approximated with $\boldsymbol{X}_k$ thus ignoring the interactions v_k has had before time t_2. This may be suboptimal if v_k has had important interactions before t_2 as these interactions are not reflected on $\boldsymbol{h}^{(t_1,1,j)}$ and hence not reflected on $\boldsymbol{h}^{(t,2,i)}$. In (Rossi et al, 2020), this problem is remedied by using a recurrent model (similar to those introduced at the beginning of this subsection) that provides node embeddings at any time based on their previous local interactions, and initializing $\boldsymbol{h}^{(t,0,i)}$s with these embeddings.

15.5 Applications

In this chapter, we provide some examples of real-world problems that have been formulated as predictions over dynamic graphs and modeled using GNNs. In particular, we review applications in computer vision, traffic forecasting, and knowledge graphs. This is by no means a comprehensive list; other application domains include recommendation systems Song et al (2019a), physical simulation of object trajectories Kipf et al (2018), social network analysis Min et al (????), automated software bug triaging Wu et al (2021a), and many more.

15.5.1 Skeleton-based Human Activity Recognition

Human activity recognition from videos is a well-studied problem in computer vision with several applications. Given a video of a human, the goal is to classify the activity performed by the human in the video into a pre-defined set of classes such as *walking*, *running*, *dancing*, etc. One possible approach for this problem is to make predictions based on the human body skeleton as the skeleton conveys important information for human action recognition. In this subsection, we provide a dynamic graph formulation of this problem and a modeling approach based mainly on (a simplified version of) the approach of (Yan et al, 2018a).

Let us begin with formulating the skeleton-based activity recognition problem as reasoning over a dynamic graph. A video is a sequence of frames and each frame can be converted into a set of n nodes corresponding to the key points in the skeleton using computer vision techniques (see, e.g., (Cao et al, 2017)). These n nodes each have a feature vector representing their (2D or 3D) coordinates in the image frame. The human body specifies how these key points are connected to each other. With

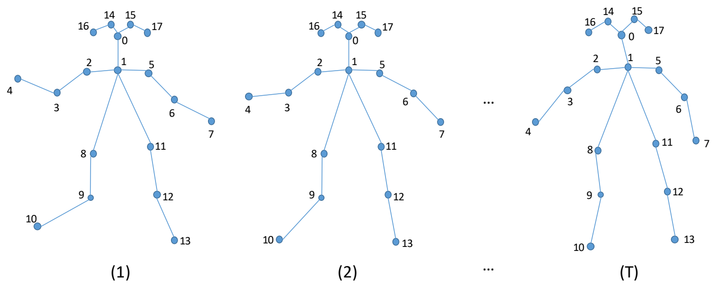


Fig. 15.6: The human skeleton represented as a graph for each snapshot of a video. The nodes represent the key points and the edges represent connections between these key points. The t graph corresponds to the human skeleton obtained from the t frame of a video.

this description, we can formulate the problem as reasoning over a DTDG consisting of a sequence $[G^{(1)}, G^{(2)}, \dots, G^{(\tau)}]$ of graphs where each $G^{(t)} = (V^{(t)}, \boldsymbol{A}^{(t)}, \boldsymbol{X}^{(t)})$ corresponds to the t frame of a video with $V^{(t)}$ representing the set of key points in the t frame, $\boldsymbol{A}^{(t)}$ representing their connections, and $\boldsymbol{X}^{(t)}$ representing their features. An example is provided in Figure 15.6. One may notice that $V^{(1)} = \dots = V^{(\tau)} = V$ and $\boldsymbol{A}^{(1)} = \dots = \boldsymbol{A}^{(\tau)} = \boldsymbol{A}$, i.e. the nodes and the adjacency matrices remain fixed throughout the sequence because they correspond to the key points and how they are connected in the human body. For instance, in the graphs of Figure 15.6, the node numbered as 3 is always connected to the nodes numbered as 2 and 4. The feature matrices $\boldsymbol{X}^{(t)}$, however, keep changing as the coordinates of the key points change in different frames. The activity recognition can now be cast as classifying a dynamic graph into a set of predefined classes C.

The approach employed in (Yan et al, 2018a) is to convert the above DTDG into a static graph through temporal unrolling (see Section 15.4.1). In the static graph, the node corresponding to a key point at time t is connected to other key points at time t according to the human body (or, in other words, according to $\boldsymbol{A}^{(t)}$) as well as the nodes representing the same key point and its neighbors in the previous ω timestamps. Once a static graph is constructed, a GNN can be applied to obtain embeddings for every joint at every timestamp. Since activity recognition corresponds to graph classification in this formulation, the decoder may consist of a (max, mean, or another type of) pooling layer on the node embeddings to obtain a graph embedding followed by a feed-forward network and a softmax layer to make class predictions.

In the l layer of the GNN in (Yan et al, 2018a), the adjacency matrix is multiplied element-wise to a mask matrix $\boldsymbol{M}^{(l)}$ with learnable parameters (i.e. $\boldsymbol{A} \odot \boldsymbol{M}^{(l)}$ is used as the adjacency matrix). $\boldsymbol{M}^{(l)}$ can be considered a data-independent attention map that learns weights for the edges in $\boldsymbol{A}$. The goal of $\boldsymbol{M}^{(l)}$ is to learn which connections are more important for activity recognition. Multiplying by $\boldsymbol{M}^{(l)}$ only allows for changing the weight of the edges in $\boldsymbol{A}$ but it cannot add new edges. Connecting the key points according to the human body may arguably not be the best choice as, e.g., the connection between the hands is important in recognizing the *clapping* activity. In (Li et al, 2019e), the adjacency is summed with two other matrices $\boldsymbol{B}^{(l)}$ and $\boldsymbol{C}^{(l)}$ (i.e. $\boldsymbol{A}+\boldsymbol{B}^{(l)}+\boldsymbol{C}^{(l)}$ is used as the adjacency) where $\boldsymbol{B}^{(l)}$ is a data-independent attention matrix similar to $\boldsymbol{M}^{(l)}$ and $\boldsymbol{C}^{(l)}$ is a data-dependent attention matrix. Adding two matrices $\boldsymbol{B}^{(l)}$ and $\boldsymbol{C}^{(l)}$ to $\boldsymbol{A}$ allows for not only changing the edge weights in $\boldsymbol{A}$ but also adding new edges.

Instead of converting the dynamic graph to a static graph through temporal unrolling and applying a GNN on the static graph as in the previous two works, in Shi et al (2019b), (among other changes) a GNN-CNN model is used. One can use other combinations of a GNN and a sequence model (e.g., GNN-RNN) to obtain embeddings for joints at different timestamps. Note that activity recognition is not an extrapolation problem (i.e. the goal is not to predict the future based on the past). Therefore, to obtain the joint embeddings at time t, one may use information not only from $G^{(t')}$ where $t' \leq t$ but also from timestamps $t' > t$. This can be done by using, e.g., a GNN-BiRNN model (see Section 15.4.2).

15.5.2 Traffic Forecasting

For urban traffic control, traffic forecasting plays a paramount role. To predict the future traffic of a road, one needs to consider two important factors: spatial dependence and temporal dependence. The traffics in different roads are spatially dependent on each other as future traffic in one road depends on the traffic in the roads that are connected to it. The spatial dependence is a function of the topology of the road networks. There is also temporal dependence for each road because the traffic volume on a road at any time depends on the traffic volume at the previous times. There are also periodic patterns as, e.g., the traffic in a road may be similar at the same times of the day or at the same times of the week.

Early approaches for traffic forecasting mainly focused on temporal dependencies and ignored the spatial dependencies (Fu et al, 2016). Later approaches aimed at capturing spatial dependencies using convolutional neural networks (CNNs) (Yu et al, 2017b), but CNNs are typically restricted to grid structures. To enable capturing both spatial and temporal dependencies, several recent works have formulated traffic forecasting as reasoning over a dynamic graph (DTDGs in particular).

We first start by formulating traffic forecasting as a reasoning problem over a dynamic graph. One possible formulation is to consider a node for each road segment and connect two nodes if their corresponding road segments intersect with each other. The node features are the traffic flow variables (e.g., speed, volume, and density). The edges can be directed, e.g., to show the flow of the traffic in one-way roads, or undirected, showing that traffic flows in both directions. The structure of the graph can also change over time as, e.g., some road segments or some intersections may get (temporarily) closed. One may record the traffic flow variables and the state of the roads and intersections at regularly-spaced time intervals resulting in a DTDG. Alternatively, one may record the variables at different (asynchronous) time intervals resulting in a CTDG. The prediction problem is a node regression problem as we require to predict the traffic flow for the nodes, and it is an extrapolation problem as we need to predict the future state of the flow. The problem can be studied under a transductive setting where a model is trained based on the traffic data in a region and tested for making predictions about the same region. It can also be studied under an inductive setting where a model is trained based on the traffic data in multiple regions and is tested on new regions.

In (Zhao et al, 2019c), a model is proposed for transductive traffic forecasting in which the problem is formulated as reasoning over a DTDG with a sequence $[G^{(1)}, G^{(2)}, \dots, G^{(\tau)}]$ of snapshots. The graph structure is considered to be fixed (i.e. no changes in road or intersection conditions) but the node features, corresponding to traffic flow features, change over time. The proposed model is a GCN-GRU model (see Section 15.4.2) where the GCN captures the spatial dependencies and the GRU captures the temporal dependencies. At any time t, the model provides a hidden representation matrix $\boldsymbol{H}^{(t)}$ based on the information at or before t; the rows of this matrix correspond to the node embeddings. These embeddings can then be used to make predictions about the traffic flow in the next timestamp(s). Assuming $\hat{\boldsymbol{Y}}^{(t+1)}$ represents the predictions for the next timestamp and $\boldsymbol{Y}^{(t+1)}$ represents the ground truth, the model is trained by minimizing an L2-regularized sum of the absolute errors $||\hat{\boldsymbol{Y}}^{(t+1)} - \boldsymbol{Y}^{(t+1)}||$.

As explained in Section 15.2.2, RNN-based models (e.g., the GCN-GRU model above) typically require sequential computations and are not amenable to parallelization. In (Yu et al, 2018a), the temporal dependencies are captured using CNNs instead of RNNs. The proposed model contains multiple blocks of CNN-GNN-CNN where the GNN is a generalization of GCNs to multi-dimensional tensors and the CNNs are gated.

The two works described so far consider the adjacency matrix to be fixed in different timestamps. As explained earlier, however, the adjacency matrix may change over time, e.g., due to accidents and roadblocks. In (Diao et al, 2019), the change in the adjacency matrix is taken into account through estimating the change in the topology of the roads based on the short-term traffic data.

15.5.3 Temporal Knowledge Graph Completion

Knowledge graphs (KGs) are databases of facts. A KG contains a set of facts in the form of triples (v_i, r_j, v_k) where v_i and v_k are called the subject and object entities and r_j is a relation. A KG can be viewed as a directed multi-relational graph with nodes $V = \{v_1, \dots, v_n\}$, relations $R = \{r_1, \dots, r_m\}$, and m adjacency matrices where the j adjacency matrix corresponds to the relations of type r_j between the nodes according to the triples.

A *temporal knowledge graph (TKG)* contains a set of temporal facts. Each fact may be associated with a single timestamp indicating the time when the event specified by the fact occurred, or a time interval indicating the start and end timestamps.

The facts with a single timestamp typically represent communication events and the facts with a time interval typically represent associative events (see Section 15.3.2)[6]. Here, we focus on facts with a single timestamp for which a TKG can be defined as a set of quadruples of the form (v_i, r_j, v_k, t) where t indicates the time when (v_i, r_j, v_k) occurred. Depending on the granularity of the timestamps, one may think of a TKG as a DTDG or a CTDG.

TKG completion is the problem of learning models based on the existing temporal facts in a TKG to answer queries of the type $(v_i, r_j, ?, t)$ (or $(?, r_j, v_k, t)$) where the correct answer is an entity $v \in V$ such that (v_i, r_j, v, t) (or (v, r_j, v_k, t)) has not been observed during training. It is mainly an interpolation problem as queries are to be answered at a timestamp t based on the past, present, and future facts. Currently, the majority of the models for TKG completion are not based on GNNs (e.g., see (Goel et al, 2020; García-Durán et al, 2018; Dasgupta et al, 2018; Lacroix et al, 2020)). Here, we describe a GNN-based approach that is mainly based on the work in (Wu et al, 2020b).

Since TKGs correspond to multi-relational graphs, to develop a GNN-based model that operates on a TKG we first need a relational GNN. Here, we describe a model named *relational graph convolution network (RGCN)* (Schlichtkrull et al, 2018) but other relational GNN models can also be used (see, e.g., (Vashishth et al, 2020)). Whereas GCN projects all neighbors of a node using the same weight matrix (see Section 15.2.1), RGCN applies relation-specific projections. Let $\hat{R}$ be a a set of relations that includes every relation in $R = \{r_1, \dots, r_m\}$ as well as a self-loop relation r_0 where each node has the relation r_0 only with itself. As is common in directed graphs (see, e.g., (Marcheggiani and Titov, 2017)) and specially for multi-relational graphs (see, e.g., (Kazemi and Poole, 2018)), for each relation $r_j \in R$ we also add an auxiliary relation r_j^{-1} to $\hat{R}$ where v_i has relation r_j^{-1} with v_k if and only if v_k has relation r_j with v_i. The l layer of an RGCN model can then be described as follows:

$$\boldsymbol{Z}^{(l)} = \sigma\Big(\sum_{r \in \hat{R}} \boldsymbol{D}^{(r)^{-1}} \boldsymbol{A}^{(r)} \boldsymbol{Z}^{(l-1)} \boldsymbol{W}^{(l,r)}\Big) \tag{15.36}$$

where $\boldsymbol{A}^{(r)} \in \mathbb{R}^{n \times n}$ represents the adjacency matrix corresponding to relation r, $\boldsymbol{D}^{(r)}$ is the degree matrix of $\boldsymbol{A}^{(r)}$ with $D_{i,i}^{(r)}$ representing the number of incoming relations of type r for the i node, $D^{(r)^{-1}}$ is a normalization term[7], $\boldsymbol{W}^{(l,r)}$ is a relation-specific weight matrix for layer l, $\boldsymbol{Z}^{(l-1)}$ represents the node embeddings in the (l-1) layer, and $\boldsymbol{Z}^{(l)}$ represents the updated node embeddings in the l layer. If initial features $\boldsymbol{X}$ are provided as input, $\boldsymbol{Z}^{(0)}$ can be set to $\boldsymbol{X}$. Otherwise, $\boldsymbol{Z}^{(0)}$ can either be set as 1-hot encodings where $\boldsymbol{Z}_i^{(0)}$ is a vector whose elements are all zeros except in the

[6] This, however, is not always true as one may break a fact such as $(v_i, \mathit{LivedIn}, v_j)$ with a time interval $[2010, 2015]$ (meaning from 2010 until 2015) into a fact $(v_i, \mathit{StartedLivingIn}, v_j)$ with a timestamp of 2010 and another fact $(v_i, \mathit{EndedLivingIn}, v_j)$ with a timestamp of 2015.

[7] One needs to handle the cases where $D_{i,i}^{(r)} = 0$ to avoid numerical issues.

i position where it is 1, or it can be randomly initialized and then learned from the data.

In (Wu et al, 2020b), a TKG is formulated as a DTDG consisting of a sequence of snapshots $[G^{(1)}, G^{(2)}, \ldots, G^{(\tau)}]$ of multi-relational graphs. Each $G^{(t)}$ contains the same set of entities V and relations R (corresponding to all the entities and relations in the TKG) and contains the triples (v_i, r_j, v_k, t) from the TKG that occurred at time t. Then, RGCN-BiGRU and RGCN-Transformer models are developed (see Section 15.4.2) that operate on the DTDG formulation of the TKG where the RGCN model provides the node embeddings at every timestamp and the BiGRU and Transformer models aggregate the temporal information. Note that in each $G^{(t)}$ there may be several nodes with no incoming and outgoing edges (and also no features since TKGs typically do not have node features). RGCN does not learn a representation for these nodes as there exists no information about them in $G^{(t)}$. To handle this, special BiGRU and Transformer models are developed in (Wu et al, 2020b) that handle missing values.

The RGCN-BiGRU and RGCN-Transformer models provide node embeddings $\boldsymbol{H}^{(t)}$ at any timestamp t. To answer a query such as $(v_i, r_j, ?, t)$, one can compute the plausibility score of (v_i, r_j, v_k, t) for every $v_k \in V$ and select the entity that achieves the highest score. A common approach to find the score for an entity v_k for the above query is to use the TransE decoder Bordes et al (2013) according to which the score is $-||\boldsymbol{H}_i^{(t)} + \boldsymbol{R}_j - \boldsymbol{H}_k^{(t)}||$ where $\boldsymbol{H}_i^{(t)}$ and $\boldsymbol{H}_k^{(t)}$ correspond to the node embeddings for v_i and v_k at time t (provided by the RGCN) and $\boldsymbol{R}$ is a matrix with learnable parameters which has $m = |R|$ rows each corresponding to an embedding for a relation. TransE and its extensions are known to make unrealistic assumptions about the types and properties of the relations Kazemi and Poole (2018), so, alternatively, one may use other decoders that has been developed within the knowledge graph embedding community (e.g., the models in (Kazemi and Poole, 2018; Trouillon et al, 2016)).

When the timestamps in the TKG are discrete and there are not many of them, one can use a similar approach as above to answer queries of the form $(v_i, r_j, v_k, ?)$ through finding the score for every t in the set of discrete timestamps and selecting the one that achieves the highest score (see, e.g., (Leblay and Chekol, 2018)). Time prediction for TKGs has been also studied in an extrapolation setting where the goal is to predict when an event is going to happen in the future. This has been mainly done using temporal point processes as decoders (see, e.g., (Trivedi et al, 2017, 2019)).

15.6 Summary

Graph-based techniques are emerging as leading approaches in the industry for application domains with relational information. Among these techniques, graph neural networks (GNNs) are currently among the top-performing approaches. While GNNs and other graph-based techniques were initially developed mainly for static graphs, extending these approaches to dynamic graphs has been the subject of several recent studies and has found success in several important areas. In this chapter, we reviewed the techniques for applying GNNs to dynamic graphs. We also reviewed some of the applications of dynamic GNNs in different domains including computer vision, traffic forecasting, and knowledge graphs.

Editor's Notes: In the universe, the only thing unchanged is "change" itself, so do networks. Hence extending techniques for simple, static networks to those for dynamic ones is inevitably the trend while this domain is progressing. While there is a fast-increasing research body for dynamic networks in recent years, much more efforts are needed in order for substantial progress in the key issues such as scalability and validity discussed in Chapter 5 and other chapters. Extensions of the techniques in Chapters 9-18 are also needed. Many real-world applications radically speaking, requires to consider dynamic network, such as recommender system (Chapter 19) and urban intelligence (Chapter 27). So they could also benefit from the technique advancement toward dynamic networks.