



វិទ្យាស្ថានបច្ចេកវិទ្យាកម្ពុជា
INSTITUTE OF TECHNOLOGY OF CAMBODIA

GRADUATE SCHOOL
MASTER'S DEGREE OF ENGINEERING
IN
DATA SCIENCE

**TEMPORAL GRAPH LEARNING WITH
APPLICATION TO LARGE-SCALE FLIGHT TRAFFIC
PREDICTION**

A THESIS SUBMITTED BY

HOR Hang

UNDER CO-SUPERVISION OF

Asst. Prof. Dr. PHAUK Sokkhey, Dr. HAS Sothea,

Prof. BENEDEK Gabor, Dr. NEANG Pheak

JULY 2024

TEMPORAL GRAPH LEARNING WITH APPLICATION TO LARGE-SCALE FLIGHT TRAFFIC PREDICTION

A THESIS PRESENTED BY

HOR Hang

TO
THE INSTITUTE OF TECHNOLOGY OF CAMBODIA

IN PARTIAL FULFILMENT OF THE
REQUIREMENTS FOR THE AWARD OF
MASTER'S DEGREE OF ENGINEERING

SPECIALIZATION: *Data Science*

SUPERVISOR: **Asst. Prof. Dr. PHAUK Sokkhey**

CO-SUPERVISOR: **Dr. HAS Sothea**

CO-SUPERVISOR: **Prof. BENEDEK Gabor**

CO-SUPERVISOR: **Dr. NEANG Pheak**

EXAMINATION COMMITTEE

Asst. Prof. Dr. SIM Tepmony , Director of Graduate School, ITC	<i>Chair</i>
Asst. Prof. Dr. VALY Dona , Lecturer and Researcher, GIC/RIC, ITC	<i>Reviewer</i>
Mr. NHIM Malai , Lecturer/Researcher, AMS, ITC	<i>Reviewer</i>
Mrs. TANN Chantara , Lecturer/Researcher, AMS, ITC	<i>Reporter</i>
Asst. Prof. Dr. PHAUK Sokkhey , Head of Master's Data Science, ITC	<i>Member</i>

PHNOM PENH, JULY 18, 2024



ក្រសួងអប់រំ យុវជន និងកីឡា
វិទ្យាស្ថានបច្ចេកវិទ្យាកម្ពុជា



និក្ខេបបទបរិញ្ញាបត្រជាន់ខ្ពស់
របស់និស្សិត ហោ ហាង

កាលបរិច្ឆេទការពារ ៖ ១៨ កក្កដា ២០២៤

អនុញ្ញាតឱ្យការពារនិក្ខេបបទ

នាយកវិទ្យាស្ថាន ៖ _____

ថ្ងៃទី ខែ ឆ្នាំ ២០២៤


ប្រធានបទ ៖ វាស់ស៊ីនសិក្សាលើក្រាហ្វិចត្តន៍អមជាមួយការអនុវត្តន៍លើការ

ព្យាករណ៍បរាចរណ៍ជើងហោះហើរនៃសំណុំទិន្នន័យខ្នាតធំ

ស្ថាប័នទទួលកម្មសិក្សា ៖ មន្ទីរពិសោធន៍ស្រាវជ្រាវ និងវិភាគទិន្នន័យ

នាយកសាលាក្រោយបរិញ្ញាបត្រ ៖ បណ្ឌិត ស៊ីម ទេពមុនី _____

សហស្រ្តាចារ្យណែនាំ ៖ ១. សាស្រ្តាចារ្យជំនួយបណ្ឌិត ភោគ សុខឌី _____

២. បណ្ឌិត ហាស សុផា _____ 

៣. សាស្រ្តាចារ្យ BENEDEK Gabor _____ 

៤. បណ្ឌិត នាង ភីក _____

រាជធានីភ្នំពេញ ឆ្នាំ២០២៤



**MINISTRY OF EDUCATION,
YOUTH AND SPORT**



INSTITUTE OF TECHNOLOGY OF CAMBODIA

MASTER'S THESIS

Of Mr. HOR Hang

Defense Date: July 18, 2024

PERMISSION TO DEFEND THE THESIS

Director of Institute: _____

Phnom Penh, _____, 2024

Thesis's Title: Temporal Graph Learning with Application to Large-Scale Flight Traffic Prediction

Host Institution: Research and Data Analytics Laboratory

Director of Graduate School: Dr. SIM Tepmony _____

Supervisors: 1. Asst. Prof. Dr. PHAUK Sökkhey _____

2. Dr. HAS Sothea _____ 

3. Prof. BENEDEK Gabor _____ 

4. Dr. NEANG Pheak _____

PHNOM PENH, CAMBODIA

ACKNOWLEDGEMENTS

My deepest gratitude goes to my advisor, **Asst. Prof. Dr. PHAUK Sokkhey**. His generosity, brilliance, and unwavering support were instrumental in completing this thesis. Dr. Sokkhey consistently provided invaluable guidance, fostering a spirit of creativity, flexibility, and meticulousness throughout the research process. His dedication to my success, including the countless hours spent mentoring me, is deeply appreciated. Without his expertise and persistent encouragement, this dissertation would not have been possible.

We gratefully acknowledge Lynx Analytics for providing access to the powerful Lynx Azure Cluster, which significantly aided our research. We would also like to extend our thanks to co-advisor **Prof. Gabor BENEDEK** for his invaluable guidance in establishing a dependable scale-up environment and for his training session on graph visualization using Lynkite.

Likewise, I would like to express my sincere gratitude and a deep sense of thanks to my co-advisor, **Dr. HAS Sothea**, Centre National de la Recherche Scientifique, Université Paris Cité and Ecole Polytechnique, and **Dr. NEANG Pheak**, Department of Applied Mathematics and Statistics, for their guidance, motivation, empathy, and as well as for his keen interest on the research topic and experiment to give color to this research. His prompt inspirations, timely suggestions with kindness, enthusiasm, and dynamism have enabled me to write up this thesis completely.

Furthermore, I would also like to thank **Asst. Prof. Dr. SIM Tepmony**, Dean of the Graduate School of ITC, who has prepared the most relevant course and systematic study program for the students. He always supports and puts all efforts from his best to strengthen and broaden scholarship opportunities and exchange programs.

Finally, I would like to acknowledge to the Department of Applied Mathematics and Statistics (AMS) for providing the necessary documentation, guidelines, and template for this thesis preparation. All model coding and optimizing works were supported by the Master Program of Data Science (M-DAS) of the Institute of Technology of Cambodia (ITC).

អត្ថបទសង្ខេប

រចនាសម្ព័ន្ធនៃបណ្តាញ ពីបណ្តាញសង្គមជាមួយនឹងសក្តានុពលនៃទំនាក់ទំនងដែលផ្លាស់ប្តូរ រហូតទៅដល់ប្រព័ន្ធដឹកជញ្ជូនជាមួយនឹងជាមួយនឹងការវិវត្តន៍ឥតឈប់ឈរនៃបណ្តាញ គឺជាទម្រង់ចាំបាច់ដែលអាចចាប់យកលក្ខណៈនៃការវិវត្តន៍របស់បណ្តាញ។ ដើម្បីស្វែងយល់ពីប្រព័ន្ធវិវត្តន៍ទាំងនេះ យើងត្រូវការគំរូម៉ាស៊ីនស្វ័យសិក្សា ដែលអាចចាប់យកទាំងសណ្ឋាននៃបណ្តាញ និងការវិវត្តន៍តាមពេលវេលាផងដែរ។ ការស្រាវជ្រាវនេះស្វែងយល់ពីម៉ាស៊ីនស្វ័យសិក្សាលើក្រាហ្វវិវត្តន៍ Temporal Graph Learning (TGL) ដែលជាផ្នែករងជំនាន់មានអានុភាពរបស់ម៉ាស៊ីនស្វ័យសិក្សា (Machine Learning) ដែលបានរចនាឡើងសម្រាប់គោលបំណងនេះ។ គំរូដែលមានស្រាប់សម្រាប់សិក្សាលើក្រាហ្វស្ថិតិវន្ត (Static Graph) មិនអាចអនុវត្តជាមួយបណ្តាញដែលវិវត្តន៍តាមពេលវេលាបានទេ។ គំរូក្រាហ្វវិវត្តន៍ TGL ដោះស្រាយបញ្ហាដែលប្រឈមនេះបានដោយដាក់បញ្ចូលភាពអាស្រ័យលើពេលវេលារួមជាមួយព័ត៌មានរចនាសម្ព័ន្ធ។ ទោះជាយ៉ាងណាក៏ដោយ ការបណ្តុះបណ្តាលគំរូទាំងនេះនៅលើសំណុំទិន្នន័យនៃបណ្តាញខ្នាតយក្ស ត្រូវការចំណាយធនធានយ៉ាងច្រើន។

ការស្រាវជ្រាវនេះស្នើវិធីសាស្ត្រដើម្បីពន្លឿនការអភិវឌ្ឍន៍គំរូ TGL ។ តាមរយៈការប្រើប្រាស់យុទ្ធសាស្ត្រគំរូប៉ាន់ (Subsampling) អនុវត្តទៅលើទិន្នន័យហោះហើរ tgbl-flight ខ្នាតធំ ធ្វើអោយយើងទទួលបានការយល់ដឹងអំពីចរិតលក្ខណៈនៃសំណុំទិន្នន័យកាន់តែច្បាស់ និងផ្តល់នូវភាពងាយស្រួលដើម្បីបណ្តុះបណ្តាលនូវគំរូ Temporal Graph Network (TGN) យ៉ាងមានប្រសិទ្ធភាពនៅលើផ្នែករឹង GPUs ធម្មតា។ នេះអនុញ្ញាតឱ្យមានការពិសោធន៍នីមួយៗបានយ៉ាងរហ័ស និងការវាយតម្លៃគំរូ មុនពេលចំណាយធនធានយ៉ាងច្រើនដើម្បីធ្វើការពង្រីកមាត្រដ្ឋានទៅលើផ្នែករឹង GPUs ដែលមានអនុភាពខ្ពស់ និងតម្លៃថ្លៃ។ លើសពីនេះទៀត គ្មានគំរូណាមួយអាចប្រសើរជាងគេគ្រប់ប្រភេទនៃសំណុំទិន្នន័យបានទេ។ ដើម្បីសម្រេចបាននូវការព្យាករណ៍អនាគតដ៏ល្អប្រសើរលើទិន្នន័យហោះហើរមួយនេះ យើងបានធ្វើការវាយតម្លៃយ៉ាងម៉ត់ចត់លើគំរូ TGL ទំនើបនិងមានអនុភាពបំផុតចំនួន 10 គំរូ។ តាមរយៈការពិសោធន៍ និងការវិភាគ យើងបានកំណត់អត្តសញ្ញាណគំរូដែលស័ក្តិសមបំផុតទៅនឹងសំណុំទិន្នន័យហោះហើរនេះ ដោយបង្កើនសក្តានុពលរបស់វាសម្រាប់ការទស្សន៍ទាយត្រឹមត្រូវលើសំណុំទិន្នន័យពេញលេញ (Full Dataset)។

លើសពីនេះ យើងណែនាំម៉ូដែល TGN-ST ដែលជាវ៉ារ្យង់ជាមួយកម្មវិធីអ៊ុនកូដពេលវេលាស្ថិតិវន្ត។ បានបំផុសគំនិតដោយការស្រាវជ្រាវលើស្ថានភាពនៃការបណ្តុះបណ្តាល TGN-ST ផ្តល់នូវភាពប្រសើរឡើងគួរឱ្យកត់សម្គាល់លើ vanilla TGN ។ នៅលើសំណុំទិន្នន័យគំរូរង ម៉ូដែល TGN-ST របស់យើង មិនត្រឹមតែមានភាពល្អប្រសើរជាង vanilla TGN 7.25% ប៉ុណ្ណោះទេ ប៉ុន្តែថែមទាំងសម្រេចបាននូវការបណ្តុះបណ្តាល (Training) លឿនលឿនជាងមុន 25% និងលឿននៃការវាយ

តម្លៃ (Validation) លឿនជាង 4.34x ផងដែរ។ នៅលើសំណុំទិន្នន័យពេញលេញ TGN-ST សម្រេចបាននូវលទ្ធផលចុងក្រោយបង្អស់ថ្មី ដែលលើសពី TGN វ៉ានីល ដោយការកែលម្អដាច់ខាត 1.99% ។ លើសពីភាពជាក់លាក់ TGN-ST បង្ហាញពីប្រសិទ្ធភាពគួរឱ្យកត់សម្គាល់។ ការប្រើប្រាស់ torch.compile API របស់កញ្ចប់ PyTorch និងការណែនាំអំពីក្បួនដោះស្រាយការបញ្ជូនបន្ត (Forward Pass Algorithm) ដ៏មានប្រសិទ្ធភាពខ្ពស់ TGN-ST សម្រេចបាននូវពេលវេលាហ្វូរវ៉ាត់លឿនជាង 15% និងសុពលភាពលឿនជាង 5 ដងបើប្រៀបធៀបទៅនឹងគំរូ TGN vanilla។ ការកើនឡើងជាប់លាប់ក្នុងការធ្វើតេស្ត MRR ដោយ TGN-ST និងការធ្វើឱ្យប្រសើរឡើងនូវប្រសិទ្ធភាពយ៉ាងច្រើននៅទូទាំងទាំងគំរូរង និងសំណុំទិន្នន័យពេញលេញផ្តល់នូវភស្តុតាងរឹងមាំសម្រាប់ប្រសិទ្ធភាពនៃគ្រោង (Framework) ដែលបានស្នើឡើង និងគំរូ TGN-ST ។ នេះត្រូវសម្រាយផ្លូវសម្រាប់ការអភិវឌ្ឍន៍ និងការដាក់ឱ្យដំណើរការកាន់តែលឿននៃគំរូ TGL នៅលើបណ្តាញថាមវន្តខ្ពស់។

ABSTRACT

The fabric of real-world networks, from social media with its ever-shifting communication dynamics to transportation systems with their constantly changing flows, is inherently dynamic, necessitating models that can capture their evolving nature. To understand these dynamic systems, we need models that capture not only their network topology but also how they change over time. This research explores Temporal Graph Learning (TGL), a powerful subfield of machine learning designed for this purpose. Traditional static graph models fall short when dealing with evolving networks. TGL models address this challenge by incorporating temporal dependencies alongside structural information. However, training these models on large-scale networks can be computationally expensive.

This work proposes a novel approach to accelerate the development of TGL models. By leveraging a customized subsampling strategy on the massive tgbl-flight dataset, we gain insight into the datasets and efficiently train state-of-the-art Temporal Graph Network (TGN) models on regular GPUs. This allows rapid experimentation and model evaluation before scaling to more powerful hardware. No single model reigns supreme across all datasets. To achieve optimal future prediction on this specific data, we conducted a rigorous evaluation of 10 state-of-the-art models. By experimenting with various models, we determined the optimal one for our dataset, achieving high prediction accuracy while efficiently utilizing computational resources.

Furthermore, we introduce the TGN-ST model, a variant with a Static Time Encoder. Inspired by research on training stability, TGN-ST offers a significant improvement over vanilla TGN. On subsample dataset, the propose TGN-ST not only surpass vanilla TGN by 7.25% absolute improvement but also achieves a 25% faster training and 4.34x faster validation. On the full dataset, TGN-ST achieves a new state-of-the-art result, exceeding vanilla TGN by 1.99% absolute improvement. Beyond accuracy, TGN-ST demonstrates remarkable efficiency. By utilizing the PyTorch’s *torch.compile* framework and introducing a highly efficient forward pass algorithm, TGN-ST achieves a 15% faster training time and 5x faster validation compared to the vanilla TGN model. The consistent gain in Test MRR by TGN-ST and the substantial efficiency improvements across the subsample and full dataset provide strong evidence for the effectiveness of the proposed framework and TGN-ST model. This paves the way for faster development and deployment of TGL models on large-scale dynamic networks.

LIST OF ABBREVIATIONS

AdamW	Adaptive moment with decoupled Weight decay
AI	Artificial Intelligent
BERT	Bidirectional Encoder Representations from Transformers
BFS	Breadth First Search
CAWN	Causal Anonymous Walk Network
CAWs	Causal Anonymous Walks
CNN	Convolutional Neural Network
CUDA	Compute Unified Device Architecture
DNN	Deep Neural Networks
DyGFormer	Dynamic Graph transFormer
DyREP	Dynamic REPresentation
ETA	Estimated Time of Arrival
FFN	Feed Forward Network
GAT	Graph Attention Network
GCN	Graph Convolutional Network
GNN	Graph Neural Network
GPT-3	Generative Pre-trained Transformer
GPU	Graphical Processing Unit
GraphSAGE	Graph Sampling and Aggregation
GRU	Gated Recurrent Unit
ILSVRC	ImageNet Large Scale Visual Recognition Challenge
JAX	Just After eXecution
JIT	Just-in-Time
JODIE	Joint Dynamic User-Item Embedding
LLMs	Large Language Model
LPA	Label Propagation Algorithm
LSTM	Long Short-Term Memory
MRR	Mean-Reciprocal-Rank
ML	Machine Learning
MLE	Maximum Likelihood Estimation
MLP	Multi-Layer Perceptron
MRR	Mean-Reciprocal-Rank

NCE	Noise Contrastive Estimation
NCoE	Neighbor Co-occurrence Encoding
NLL	Negative Log Loss
NLP	Natural Language Processing
NNs	Neural Networks
NS	Negative Sampling
OOM	Out-Of-Memory
PDF	Probability Density Function
PReLU	Parametric ReLU
ReLU	Rectified Linear Unit function
RNN	Recurrent Neural Network
SAM	Self-Attention Mechanisms
SGD	Stochastic Gradient Descent
TCL	Transformer-based Dynamic Graph Modelling via Contrastive Learning
TEA	Temporal Edge Appearance
TET	Temporal Edge Traffic
TGAT	Temporal Graph Attention
TGB	Temporal Graph Benchmark
TGL	Temporal Graph Learning
TGN	Temporal Graph Network
TGN-ST	Temporal Graph Network with Static Time encoder
TGNNs	Temporal Graph Neural Networks
UniMP	Unified Message Passing

TABLE OF CONTENTS

ACKNOWLEDGEMENTS	i
អត្ថបទសង្ខេប	ii
ABSTRACT	iv
LIST OF ABBREVIATIONS	v
TABLE OF CONTENTS	vii
LIST OF FIGURES	ix
LIST OF TABLES	xii
CHAPTER I INTRODUCTION	1
1.1. The success and pitfalls of deep learning	2
1.2. The Rise of Temporal Graph Neural Networks	3
1.3. Research Questions and Contributions	6
CHAPTER II LITERATURE REVIEW	9
2.1. Mathematical Notation	9
2.2. Essentials	11
2.2.1. Multi-Layer Perceptron	11
2.2.2. Learning Neural Network Parameters (Optimization and Regularization)	13
2.2.3. Recurrent Neural Networks	17
2.2.4. Transformer	19
2.2.5. Neural Networks Optimization	21
2.3. Graph Data Structure	21
2.4. Learning Settings on Static Graphs	22
2.5. An Overview of Graph Neural Networks	22
2.6. Temporal Graph Learning	26
2.6.1. Temporal Graph Data Structure and Learning on Temporal Graphs	26
2.6.2. Random Walk on Temporal Graphs	27
2.6.3. Negative Sampling in Temporal Graphs	28
2.6.4. Existing Temporal Graph Learning Architecture	29
2.7. Contrastive Learning	35
CHAPTER III METHODOLOGY	38
3.1. Dataset Insight	38
3.2. Evaluation Metrics	40
3.3. Temporal Graph Learning Pipeline	40
3.4. Temporal Graph Model Selection Strategy	43
3.5. TGN Model Architecture Insight	46

3.6. Graph Capture and Compilation	47
3.7. TGN-ST forward pass algorithm	49
3.8. Experimental Settings and Hyperparameter Tuning	52
CHAPTER IV RESULTS AND DISCUSSIONS	57
CHAPTER V CONCLUSIONS AND FUTURE DIRECTIONS	61
5.1. Conclusions.....	61
5.2. Reproducible Research	62
5.3. Future Directions	62
REFERENCES	63

LIST OF FIGURES

Figure 1.1.1 (Top): Cat images from ImageNet dataset. (Bottom Left): An input image (of the tabby cat class). A deep neural network (VGG-16 (Simonyan & Zisserman, 2015)) pre-trained on ImageNet can extract meaningful representations in both its shallow (bottom middle) and deep (Bottom right) layers. The shallower layers typically extract rudimentary features (such as separating the foreground from the background) while, at the deeper stages of processing, the network highlights distinct “cat-like” properties, such as its eyes, ears and fur.	2
Figure 1.2.1 (Pan, Chen, Long, Zhang, & Yu, 2019) (left) 2D Convolution. Analogous to a graph, each pixel in an image is taken as a node where neighbors are determined by the filter size. The 2D convolution takes the weighted average of pixel values of the red node along with its neighbors. The neighbors of a node are ordered and have a fixed size. (right) Graph Convolution. To get a hidden representation of the red node, one simple solution of the graph convolutional operation is to take the average value of the node features of the red node along with its neighbors. Different from image data, the neighbors of a node are.....	5
Figure 2.2.1 Left: A single perceptron, whose input vector \mathbf{x} , weight \mathbf{w} , bias b and layer-wise activation function σ . Right: A standard multi-layer perceptron with one hidden layer.	11
Figure 2.2.2 Plot of all (a) Activation functions and (b) its corresponding gradient function used throughout the thesis.....	12
Figure 2.2.3 Polynomial degree of 2, 14, and 20 fit to 21 data point	15
Figure 2.2.4 An illustration of early stopping with patience of 3 epochs.	16
Figure 2.2.5 (Srivastava, Hinton, Krizhevsky, Sutskever, & Salakhutdinov, 2014)The effect of a single iteration of dropout on a feed forward network. (a) standard neural network, (b) after applying dropout.....	16
Figure 2.2.6 (Zhang A., 2020) Computational graph for an MLP with input \mathbf{x} , hidden layer \mathbf{h} , output \mathbf{o} , loss function $L = \mathbf{lo}, y$, and l_2 regularizer s on the <i>weights</i> , and total loss $J = L + s$	17
Figure 2.2.7 Gate Recurrent Unit (GRU) (Zhang A., 2020).....	18
Figure 2.2.8 Transformer architecture (Vaswani, et al., 2017). While BERT’s backbone employs Encoder block, GPT’s backbone employs Decoder block.	19

Figure 2.2.9 Attention visualizations (Vaswani, et al., 2017) for a sequence of words. Many of the attention heads attend to a distant dependency of the verb ‘making’, completing the phrase ‘making...more difficult’. Attentions here shown only for the word ‘making’	20
Figure 2.2.10 An illustration of directed graph.....	21
Figure 2.5.1 (Hamilton W. L., 2020) An illustration of 2-hop $k = 2$ neighborhood aggregation by unfolding the neighborhood around the target node. For instance, node A is aggregated information from node B , C , and D . The corresponding neighborhood of B , C , and D recursively aggregated information from its neighborhood.	23
Figure 2.5.2 Plot of edge’s attention weight on sub-graph of Cora’s dataset.	25
Figure 2.5.3 An illustration of temporal graph network.	26
Figure 2.6.1 (Poursafaei, Huang, Pelrine, & Rabbany, 2022) Negative edge sampling strategies during evaluation for dynamic link prediction; (a) random sampling, (b) historical sampling, and (c) inductive sampling	28
Figure 2.7.1 A taxonomy of temporal graph learning methods	37
Figure 3.1.1 Graph subsampling visualization of 100 sample nodes and their corresponding edges. The left-hand side is TET plot, and the right-hand side is the time series of averaged node degrees. (Note: the last timestamp is not one month fully aggregated, so the chart is relatively low).	39
Figure 3.1.1 Illustration of graph subsampling visualization pipeline.....	40
Figure 3.3.1 An illustration of ur temporal graph learning pipeline	41
Figure 3.4.1 (top left) average test mrr results of various model benchmarking. (top right) GPU memory usage in bytes. (bottom left) training time per epoch. (bottom right) validation time per epoch.....	45
Figure 3.4.2 An illustration of attention-based message passing aggregator.....	46
Figure 3.5.1 An illustration of link predictor module of TGN network.	47
Figure 3.6.1 An illustration of FX graph of static time encoder (a) user code and (b) complied and optimized code	48
Figure 3.7.1 An illustration of Hyperparameter sweeping filtered on optimizer by sgd only. (top left) visualize the best validation mrr of each run. (top right) the correlation of the best mrr with each hyperparameter. (bottom) all hyperparameter filtered by sgd optimizer, i.e. top transparent rectangular on optimizer column.	52
Figure 3.8.2 Hyperparameter settings of best_val_mrr greater than 79% filtered by AdamW optimizer.	53

Figure 3.8.3 Hyperparameter sweeping filtered by top performance (shown in square transparent rectangular on best_val_mrr column). Notice that y_relu stands for leaky_relu. .54	
Figure 3.8.4 An example of using cosine annealing with warm restart and its corresponding loss function during model training.55	
Figure 3.8.5 FX Graph details of TGN embedding modules.....56	
Figure 3.8.1 Illustration of training, validation and test result bechmarking between TGN-ST and vanilla TGN model. (a) Training loss result. (b) validation mean reciprocal rank (MRR). (c) training time per epoch. (d) validation time per epoch. (e) test MRR. (f) wall clock time.59	

LIST OF TABLES

Table 1 List of Mathematical Notations.....	9
Table 2. Graph statistic of TGB dataset over 100 sample nodes	39
Table 3. Hyperparameter settings of 200 sweeping on TGN-ST models	55
Table 4. Result summary of 100 sample nodes and full dataset benchmarking between vanilla TGN and TGN-ST model.	58

CHAPTER I

INTRODUCTION

Early successes in data science, particularly in areas such as feature engineering and model selection, relied heavily on handcrafted approaches. This involved manually defining rules and transformations to extract meaningful features from raw data. Similarly, model selection often focuses on choosing algorithms with interpretable parameters and clear decision boundaries. This symbolic approach, akin to knowledge bases in Artificial Intelligence (AI), achieved some initial wins. However, its limitations became apparent as data complexity grew. Feature engineering, for example, could become computationally expensive and require significant domain expertise. Additionally, manually defining rules often struggled to capture the non-linear relationships and intricate patterns within data. This inherent challenge in capturing complex data patterns led to a paradigm shift towards data-driven approaches, which called for the standardization of benchmarking computer vision tasks.

A standard computer vision task like the ImageNet dataset (Deng, et al., 2009) (**Figure 1.1.1 (top)**), exemplifies this challenge. ImageNet is a giant collection of over 14 million images categorized into thousands of everyday objects. While humans can effortlessly recognize objects in these images, translating this intuition into a set of classification rules is extremely difficult. Real-world data exhibits inherent variability – a cat image can appear in countless poses, lighting conditions, and scales. Capturing this variability through hand-crafted rules becomes impractical. Even minor changes in the image, like rotations or shifts, can easily fool such a system, highlighting the limitations of this handcrafted approach. This is where the ImageNet task emerged as a pivotal benchmark. By providing a massive, richly labeled dataset encompassing a vast array of everyday objects, ImageNet challenged researchers to develop algorithms capable of robust image classification. The task's design goes beyond just evaluating classification accuracy. In the realm of artificial intelligence, computer vision strives to enable machines to “see” and “understand” the world. Therefore, it pushes the boundaries of how well machines can learn to handle the inherent variability within real-world data, understand action, language semantic meaning, and visual the world. It is therefore paving the way for more robust and automated approaches in data science.

Unlike such a rule-based classifier, humans have been exposed to a significant number of animals, not just limited to cat, during their lifetimes and have therefore learnt the complex

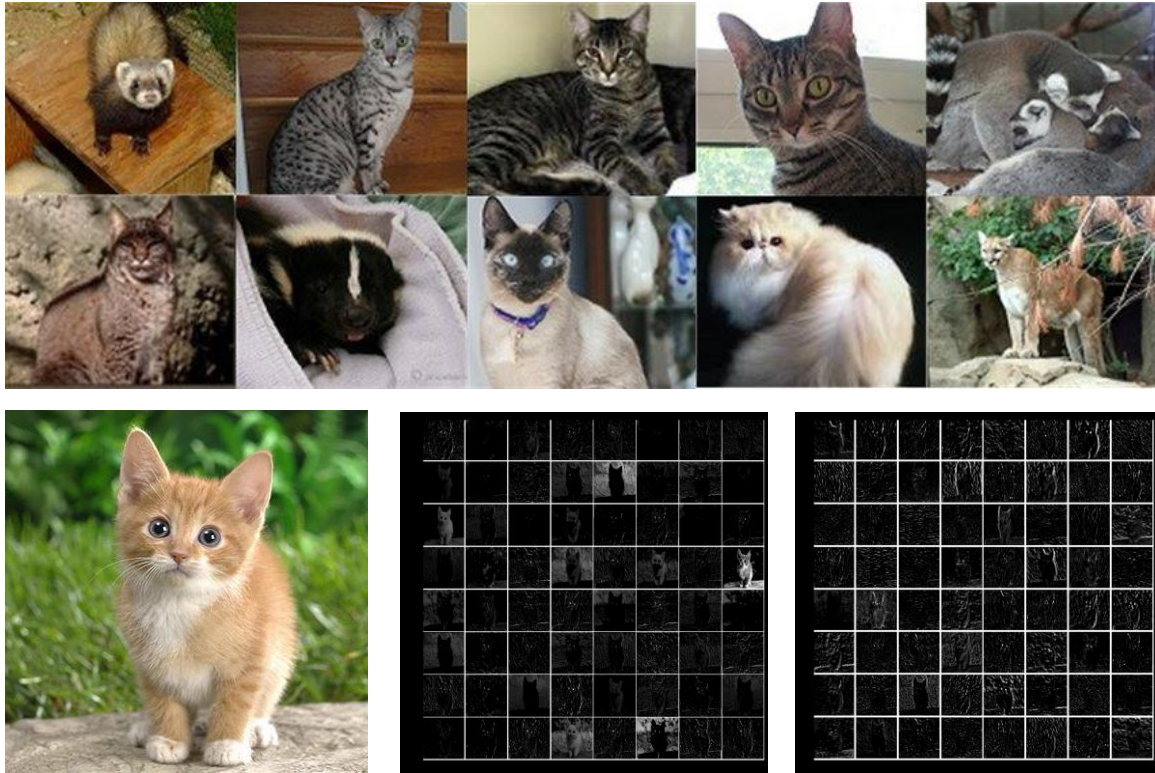


Figure 1.1.1 (Top): Cat images from ImageNet dataset. **(Bottom Left):** An input image (of the tabby cat class). A deep neural network (VGG-16 (Simonyan & Zisserman, 2015)) pre-trained on ImageNet can extract meaningful representations in both its shallow (**bottom middle**) and deep (**Bottom right**) layers. The shallower layers typically extract rudimentary features (such as separating the foreground from the background) while, at the deeper stages of processing, the network highlights distinct “cat-like” properties, such as its eyes, ears and fur.

features that makeup of the animals such as cats, without ever needing to rigorously define a set of rules for doing so. Machine Learning (ML), at its core, aims to capture exactly this—systems capable of generalizing from past experiences. It has, therefore, become the dominant approach to artificial intelligence in recent years; especially through deep neural networks (commonly known as deep learning (LeCun, Bengio, & Hinton, Deep learning, 2015) in this context).

1.1. The success and pitfalls of deep learning

Deep neural networks (DNNs) tackle the problem from a representation learning (Bengio, Courville, & Vincent, Representation learning: A review and new perspectives., 2013) perspective; they are composed of a stack of feature-extracting layers, with the first layer processing raw inputs, and each subsequent layer receiving the output of the previous layer. As

such, DNNs build up gradually more complex representations of the inputs and eliminate the need for hand-crafted feature extraction. Given a sufficiently large training set of input/output examples, the network iteratively adjusts its parameters (typically using a first-order method (LeCun Y. , Bottou, Bengio, & Haffner, 1998) such as gradient descent) to optimize the error of the network’s predictions on those examples compared to the ground-truth output. Consequently, the network gradually acquires better representations of the data as well, with the “shallower” layers specializing in detecting simple input features (such as edges and contours in an image), with “deeper” layers detecting more complex clues. **Figure 1.1.1 (bottom)** provides a visualization of this phenomenon in the case of image classification on the ImageNet dataset (where it is typically easiest to conceptualize). This method assumes a direction, which is in stark contrast to the symbolic approaches; minimal assumptions about the specifics of the task are given, relying on the model to automatically infer them through observing the training data. In conjunction with specialized architectures that use parameter sharing to exploit simple redundancies and invariances in grid-structured data—such as convolutional neural networks (CNNs) (Krizhevsky, Sutskever, & Hinton, 2012) for images and recurrent neural networks (RNNs) (ELMAN, 1990; Hochreiter, 1997; Jordan, 1997) for sequences and Transformer for both images (Dosovitskiy, et al., 2021) and sequences (Vaswani, et al., 2017) —these techniques currently hold the state-of-the-art result on many challenging tasks of interest.

1.2. The Rise of Temporal Graph Neural Networks

The phenomenal success of deep learning across various domains can be attributed to several key factors: the rapidly developing computational resources (e.g., GPU), the availability of big training data, and the effectiveness of deep learning to extract latent representations from Euclidean data (e.g., images, text, and videos). Consider image data for instance, we can represent an image as a regular grid in the Euclidean space. A convolutional neural network (CNN) can exploit the shift-invariance, local connectivity, and compositionality of image data (Bronstein, Bruna, LeCun, Szlam, & Vandergheynst, 2017). Consequently, CNNs can extract the local meaningful features shared with the entire data sets for various image analyses.

Despite deep learning's remarkable achievements in computer vision (e.g. (LeCun Y. , Bottou, Bengio, & Haffner, 1998; Krizhevsky, Sutskever, & Hinton, 2012), natural language processing (e.g., (Bengio, Ducharme, & Vincent, 2000; Vaswani, et al., 2017)), game playing

(e.g., (Volodymyr Mnih, Graves, Antonoglou, Wierstra, & Riedmiller, 2013; Silver, et al., 2016; Louppe, Cho, Becot, & Cranmer, 2019)), and in the natural sciences (e.g., Dahl et al., 2014; Louppe et al., 2019), deep neural networks still struggle with general abilities like relational and causal reasoning, conceptual abstraction, and other fundamental human cognitive skills.

A central problem in machine learning revolves around inductive bias (Mitchell, 1980): how can we build models that learn the right representations, abstractions, and skills that could allow them to generalize to novel and unforeseeable circumstances? Inductive bias in deep neural networks manifests in various ways: The chosen model architecture, training objective, optimization procedure, and even the way training data is presented (e.g., data augmentation), all significantly impact generalization to unseen data. The surge in deep learning's popularity was partly fueled by the design of a specific architectural inductive bias – Convolutional Neural Networks (CNNs) (LeCun Y. , Bottou, Bengio, & Haffner, 1998; Krizhevsky, Sutskever, & Hinton, 2012)

CNNs leverage a specialized architecture where model parameters are tied or shared across image locations, exploiting the fact that image feature statistics are often translation invariant. This parameter sharing equips the model with a powerful inductive bias – "filters" in the CNN model only need to learn about local features and therefore generalize well across locations in the image. A similar inductive bias exists in Recurrent Neural Networks (RNNs), which share parameters across time steps, leading to better generalization on stationary time series and other sequential data.

While deep learning effectively captures hidden patterns of Euclidean data, there is an increasing number of applications where the data are represented as graphs. With modern technology and the rise of the internet, graphs are everywhere. For instance, in traffic networks, the roads can be segmented into small lines whose end points can modeled as nodes, and connection between those endpoints as the graph edge is used to predict the traffic flow (Derrow-Pinion, et al., 2021) (i.e. estimating the time of arrival ETA). In e-commerce, a graph-based learning system can exploit the interactions between users and products to make highly accurate recommendations. In chemistry, molecules are modeled as graphs, and their bioactivity needs to be identified for drug discovery. In a citation network, papers are linked to each other via a citation link. It is required to be categorized into different groups. However, the inherent complexity of graph data has presented significant challenges to existing machine

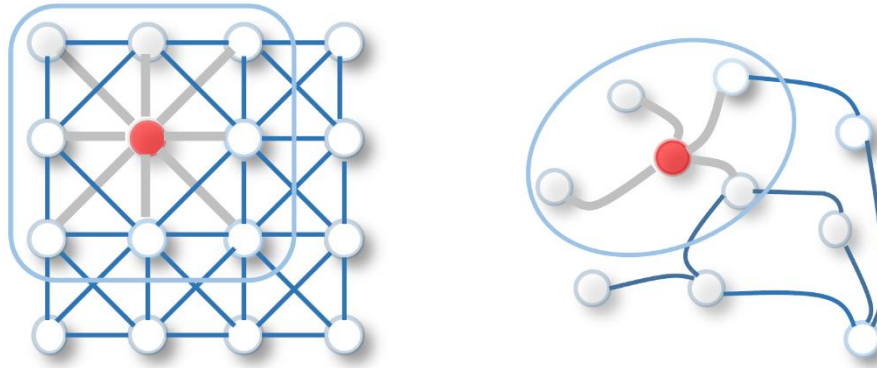


Figure 1.2.1 (Pan, Chen, Long, Zhang, & Yu, 2019) **(left)** 2D Convolution.

Analogous to a graph, each pixel in an image is taken as a node where neighbors are determined by the filter size. The 2D convolution takes the weighted average of pixel values of the red node along with its neighbors. The neighbors of a node are ordered and have a fixed size. **(right)** Graph Convolution. To get a hidden representation of the red node, one simple solution of the graph convolutional operation is to take the average value of the node features of the red node along with its neighbors. Different from image data, the neighbors of a node are aggregated.

learning algorithms. As graphs can be irregular, a graph may have a variable size of unordered nodes, and nodes from a graph may have a different number of neighbors, resulting in some important operations (e.g., convolutions) being easy to compute in the image domain but difficult to apply to the graph domain. Additionally, a core assumption of existing machine learning algorithms is that instances are independent of each other. This assumption crumbles for graph data because each node (instance) is intricately linked to others by various connections, such as citations, friendships, and interactions.

Recognizing the limitations of traditional methods for graph data, researchers are actively exploring ways to extend deep learning approaches to this domain. Inspired by CNNs, RNNs, and autoencoders from deep learning, new generalizations and definitions of important operations have been rapidly developed over the past few years to handle the complexity of graph data. For instance, graph convolution generalizes the concept of 2D convolution from images. As illustrated in **Figure 1.2.1**, an image can be viewed as a special case of a graph where pixels are connected to their adjacent neighbors. Similar to 2D convolution, graph convolutions can be performed by taking the weighted average of a node's neighborhood information.

One way to achieve this is by structuring the representations and computations in a deep neural network in the form of a graph, leading to a class of models named *graph neural networks* (Kipf & Welling, 2017; Marco Gori, 2005; Scarselli, Gori, Tsoi, Hagenbuchner, & Monfardini, 2009; Justin, S., F., Oriol, & E., 2017; Battaglia, et al., 2018). While GNN is powerful and could achieve state-of-the-art results on many aspects of learning on graphs, the real-world network is inherently not static. The increasing prevalence of dynamic networks, where connections and node properties change over time, has demanded new methods beyond static graph neural networks. This has led to the rise of temporal graph learning (TGL) or temporal graph neural networks (TGNNs), which effectively combine the strengths of GNNs with techniques from recurrent neural networks or other methods. TGNNs can learn how the characteristics of nodes and edges evolve within a network, unlocking superior performance in tasks like link prediction, node classification, and anomaly detection specifically within temporal contexts. The continual growth of large-scale temporal graph datasets, advancements in computational power, and the surge in applications requiring analysis of dynamic network behaviors (like traffic forecasting, social network analysis, and disease spread modeling) have all fueled the rise of TGNNs. As a rapidly evolving field, TGNN research constantly develops new architectures and techniques to address challenges like handling intricate temporal patterns, open-ended node types, and scenarios with limited historical data. In conclusion, temporal graph neural networks represent a substantial leap forward in network analysis, empowering researchers and practitioners to harness the power of GNNs for understanding and making predictions on dynamic and ever-changing systems.

1.3. Research Questions and Contributions

Extending Graph Neural Networks (GNNs) from static graphs into temporal graph learning (TGL) presents a significant challenge. TGL models must account for the continuous evolution of nodes, edges, and features within the network. However, this advancement comes at a computational cost. Training models on large temporal graphs can be computationally expensive and time-consuming. For instance, utilizing a cloud computing platform like Google Compute Engine costs \$3.67 per hour, translating to a monthly expense of approximately \$2,682. Therefore, exploring methods to optimize computational efficiency becomes paramount when working with TGL on large-scale datasets. Having provided the necessary background overview of the importance of structural inductive biases, surveyed the wide landscape of ongoing related work, and computational challenges in temporal graph learning, I will now formulate four research questions that I seek to answer within this thesis, along with

the specific contributions made towards achieving them (and where they may be found in this document).

Question 1: *Efficient Analysis of Large-Scale Dynamic Flight Traffic Networks:*

Considering a large-scale dataset of international flight traffic represented as a temporal graph with millions of edges, what are efficient computational techniques that can extract valuable insights, given a limited computational budget?

To facilitate experimentation on a wider range of computational resources, a scalable temporal graph pipeline was developed in **Section 3.1** and **3.3**. This pipeline leverages graph subsampling and discretization techniques to create smaller, more manageable representations of the original temporal graph. Modern visualization techniques, such as Temporal Edge Appearance (TEA) plots and Temporal Edge Traffic (TET) plots, are then employed on the subsampled graph to extract valuable insights from the dataset.

Question 2. *TGL Model Selection Strategy and Evaluation Method:* *Building upon the insights gained from the temporal graph learning pipeline and dataset analysis in Question 1, the next step is identifying the most suitable model architecture for our specific data. Furthermore, appropriate evaluation metrics must be established to assess the performance of the chosen model.*

Informed by the comprehensive literature review presented in **Section 2.6.4**, ten models were selected for evaluation on the subsampled graph. The evaluation methodology is detailed in **Section 3.2**. **Section 3.4** analyzes the model selection process and the corresponding configuration of the chosen model. Finally, a deep dive into the selected model's performance is provided in **Section 3.5**.

Question 3. *Optimizing Performance of the Selected Temporal Graph Learning Model (TGL Model):*

Building upon the model selection process in Question 2, this section explores strategies to enhance the performance of the chosen temporal graph learning (TGL) model. Specifically, we will investigate techniques for optimizing the hyperparameter settings of the model to achieve optimal results on our dataset.

To address the well-documented issue of instability in learnable time encodings within Temporal Graph Neural Networks (TGNs), we propose an efficient

temporal graph network with static time encoder (TGN-ST) model that leverages graph capture and compilation framework discussed in **Section 3.6** and vectorized forward pass algorithms discussed in **Section 3.7**. This model aims to improve stability and efficiency. Additionally, to optimize performance for our specific dataset, we conducted a comprehensive hyperparameter sweep with 200 iterations to identify the most suitable settings as discussed in **Section 3.8**.

Question 4. Generalizability Assessment and Performance on Full Traffic Flight Dataset.

While the model was trained on a subsampled graph for computational efficiency, it is crucial to assess how the learned patterns and insights translate to the complete dataset. This evaluation will reveal the model's ability to perform effectively on unseen data, a critical factor for real-world applications.

With a fully functional end-to-end temporal graph learning pipeline, including graph subsampling, model selection, and optimization, we can now investigate the potential performance gains on the complete traffic flight dataset, i.e. TGB's tgbl-flight dataset (Huang, et al., 2023). **CHAPTER IV** focuses on evaluating the generalizability and performance of the chosen model on the full traffic flight dataset. This will reveal whether the model trained on the subsampled data exhibits improved performance when applied to the full dataset.

Aside from exposing my primary research contributions—in the two chapters, **CHAPTER III** and **CHAPTER IV**, outlined above—this thesis also contains (in **CHAPTER II**) a comprehensive overview of background information on machine learning with deep neural networks. Emphasis is given to providing the essential mathematical details of the relevant models with structural inductive biases (going from RNNs and Transformer towards graph convolutional networks and temporal graph neural network methods). I will also provide concluding thoughts, with particular emphasis on future work directions, in **CHAPTER V**.

CHAPTER II

LITERATURE REVIEW

This chapter lays the foundation for the thesis by introducing essential topics, background concepts, and notation that will be used extensively throughout the following chapters. Any additional background information will be introduced as needed in later sections. Specifically,

- **Section 2.1** present mathematical notions used throughout this thesis
- **Section 2.2** provides a primer on essential elements of deep learning
- **Section 2.3** introduces graph data structures.
- **Section 2.4** discusses learning settings on static graphs.
- **Section 2.5** covers graph neural networks.
- **Section 2.6** explores temporal graph learning methods.
- **Section 2.7** concludes with an introduction to contrastive learning.

2.1. Mathematical Notation

This section establishes a reference for the key notation used throughout this thesis. Subsequent chapters will introduce any additional notation specific to their content.

Table 1 List of Mathematical Notations

Example	Explanation
I	The identity matrix
x	A lowercase italic letter typically denotes a scalar or scalar-valued random variable
\mathbf{x}	A lowercase bold letter typically denotes a vector or vector-valued random variable.
\mathbf{X}	An uppercase bold letter typically denotes a matrix or matrix-valued random variable.
\mathcal{X}	Calligraphic letters typically denote sets. An exception is \mathcal{L} , which we use to denote scalar-valued objective functions
\mathbb{R}^N	The N-dimensional real space
x_i	The i -th element of vector \mathbf{x}
$X_{i,j}$	The i, j -th element of matrix \mathbf{X}

$f_{\boldsymbol{\theta}}(\cdot)$ $f(\cdot; \boldsymbol{\theta})$	Parameter dependency of functions is typically made explicit with a Greek letter $\boldsymbol{\theta}$ or $\boldsymbol{\phi}$
$p(\cdot), q(\cdot)$	Probability density functions (PDFs) or in other words distributions are denoted by the lower-case letters $p(\cdot)$ and $q(\cdot)$, where $q(\cdot)$ is typically reserved for <i>variational</i> distributions. We use the same notation for probability mass functions (PMFs) of discrete random variables. The same letter will be used for marginals, joint distributions, and conditionals of the same probabilistic model.
$f: \mathbb{X} \mapsto \mathbb{Y}$	The function f taking the elements of set \mathbb{X} mapping to set \mathbb{Y}
$f(x)$	Function f applied to input x . f is applied element-wise if x is a tensor
$\log(x)$	The natural logarithm of x
$\exp(x)$	The natural exponential of x
$\ \mathbf{x}\ $	The L_2 norm of vector \mathbf{x}
$\mathbf{x} \parallel \mathbf{y}$	Concatenation of \mathbf{x} and \mathbf{y}
\mathbf{X}^T	Transposed matrix \mathbf{X}
$\mathbf{x} \odot \mathbf{y}$	The Hadamard product of vector \mathbf{x} and \mathbf{y}
$\nabla_{\boldsymbol{\theta}} \mathbf{y} _{\boldsymbol{\theta}=\boldsymbol{\theta}_0}$	The gradient of \mathbf{y} with respect to $\boldsymbol{\theta}$, evaluated at $\boldsymbol{\theta} = \boldsymbol{\theta}_0$
$\mathcal{N}(\cdot; \boldsymbol{\mu}, \boldsymbol{\Sigma})$	A multivariate normal (or Gaussian) distribution with a vector of means $\boldsymbol{\mu}$ and a covariance matrix $\boldsymbol{\Sigma}$.
$\mathbf{X} \subseteq \mathbf{Y}$	\mathbf{X} is a subset of \mathbf{Y}
$\mathcal{O}(\cdot)$	The limiting behavior of a function is when an argument tends towards a particular value or infinity.
$\mathbf{x}_u^{(k)}$	The representation vector of element u at layer k
$\bar{\mathbf{X}}$	The set of all elements excludes the set \mathbf{X}
$\mathbf{X}[:, i]$	Get all the elements of rows of \mathbf{X} at column indexed by i

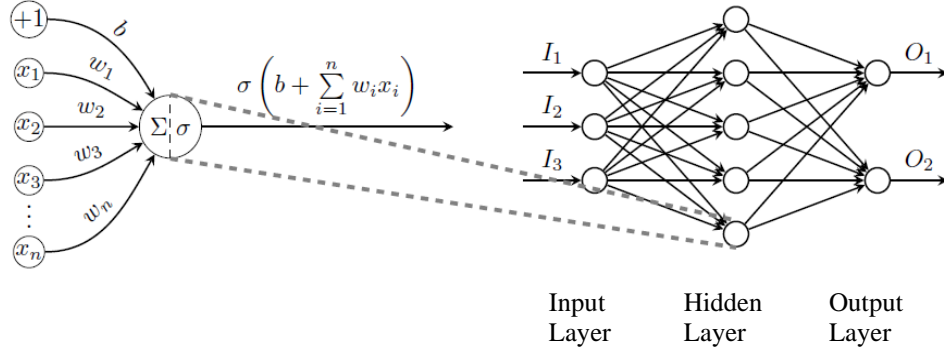


Figure 2.2.1 Left: A single perceptron, whose input vector \mathbf{x} , weight \mathbf{w} , bias b and layer-wise activation function σ . **Right:** A standard multi-layer perceptron with one hidden layer.

2.2. Essentials

2.2.1. Multi-Layer Perceptron

Multi-layer perceptron (MLP), also known as deep feedforward networks, are the quintessential deep learning models. It will be used within all the settings in this thesis. The goal of an MLP is to approximate some function f^* . The universal approximation theorem (Hornik, Stinchcombe, & White, 1989) states that an MLP with a linear output layer and at least one hidden layer with any “squashing” activation function can approximate any Borel measurable function from one finite-dimensional space to another with any desired nonzero amount of error, provided that the network is given enough hidden units. The output of MLP can be written as:

$$\mathbf{h}^{(k)} = \sigma(\mathbf{W}^{(k)} \mathbf{h}^{(k-1)} + \mathbf{b}^{(k)}) \quad (\text{Eq. 1})$$

Where $\mathbf{W}^{(k)}$ and $\mathbf{b}^{(k)}$ are the learnable weight and bias at layer k , respectively.

It is worth mentioning a special kind of MLP that is widely used ubiquitously in many various domains such as NLP like Transformer (Vaswani, et al., 2017), GNN like GAT (Velickovic, et al., 2018), TGL like TGAT (Xu, Ruan, Korpeoglu, Kumar, & Achan, 2020) defined as:

$$FFN(x) = \max(0, xW^{(2)} + b_1)W^{(2)} + b^{(2)} \quad (\text{Eq. 2})$$

The activation functions of interest for this thesis.

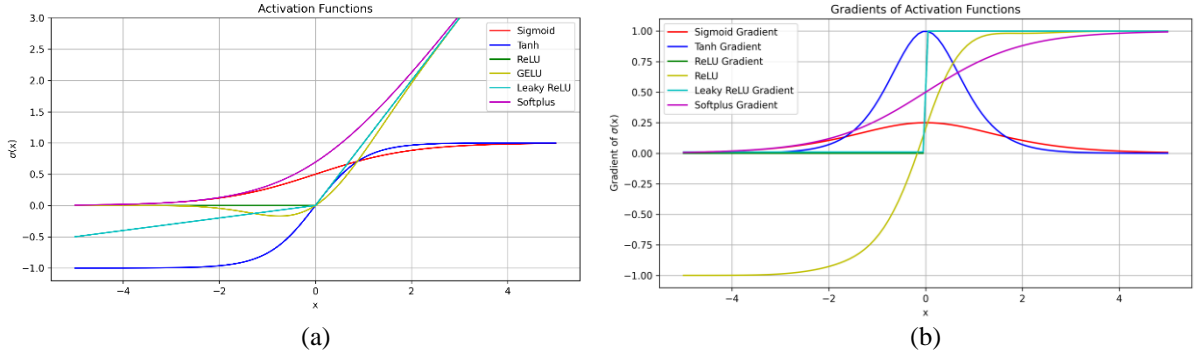


Figure 2.2.2 Plot of all (a) Activation functions and (b) its corresponding gradient function used throughout the thesis.

- The logistic sigmoid function and its derivative:

$$\sigma(x) = \frac{1}{1 + e^{-x}}; \sigma'(x) = \sigma(x)(1 - \sigma(x)) \quad (\text{Eq. 3})$$

which monotonically maps real numbers to the $[0; 1]$ range, making it suitable for modeling Bernoulli random variables (e.g. for binary classification).

- The hyperbolic tangent function and its derivative:

$$\sigma(x) = \tanh(x); \sigma'(x) = 1 - (\sigma(x))^2 \quad (\text{Eq. 4})$$

which can model general real outputs (given its $[-1; 1]$ range and sigmoidal shape).

- The rectified linear function (ReLU) (Glorot, Bordes, & Bengio, 2011) and its derivative:

$$\sigma(x) = \max(0, x); \sigma'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases} \quad (\text{Eq. 5})$$

- The GeLU and its derivative:

$$\sigma(x) = x\Phi(x); \sigma'(x) = \Phi(x) + xP(X = x) \quad (\text{Eq. 6})$$

Where $\Phi(x)$ is the cumulative distribution function for Gaussian distribution and $P(X = x)$ is the value of the probability density function (PDF) at x .

- The Leaky ReLU, with a give negative slot, α :

$$\sigma(x) = \begin{cases} x & \text{for } x \geq 0 \\ \alpha x & \text{for } x < 0 \end{cases}; \sigma'(x) = \begin{cases} 1 & x \geq 0 \\ \alpha & x < 0 \end{cases} \quad (\text{Eq. 7})$$

Called the parametric ReLU (PReLU) (Kaiming He, 2015) when α is a learnable parameter.

- The Softplus function, with a give β :

$$\sigma = \frac{1}{\beta} \log(1 + \exp(\beta x)); \sigma'(x) = \text{sigmoid}(\beta x) \quad (\text{Eq. 8})$$

SoftPlus is a smooth approximation to the ReLU function and can be used to constrain the output of a machine to always be positive.

2.2.2. Learning Neural Network Parameters (Optimization and Regularization)

In this section, we discuss how to fit deep neural networks (DNNs) to data. Given the input and label couple $(\mathbf{x}_n, \mathbf{y}_n)$, the standard approach to use maximum likelihood estimation, by minimizing the Negative Log Loss (NLL):

$$\mathcal{L}(\boldsymbol{\theta}) = -\log p(\mathcal{D}|\boldsymbol{\theta}) = -\sum_{n=1}^N \log p(\mathbf{y}_n|\mathbf{x}_n; \boldsymbol{\theta}) \quad (\text{Eq. 9})$$

For DNN, the loss function, \mathcal{L} , commonly optimized using mini-batch *stochastic gradient descent* (SGD). At each iteration, a mini-batch, \mathcal{B} of m training example is selected, and the backpropagation algorithm (Rumelhart, Hinton, & Williams, 1986) is applied to them to efficiently compute the gradient of the loss function, $\nabla_{\boldsymbol{\theta}} \mathcal{L}_{\mathcal{B}}$ with respect to the network's parameters, $\boldsymbol{\theta}$. This computation involved repeatedly applying the chain rule for partial derivatives, proceeding progressively backward from the output layer toward the input layer of the network. The parameters are then updated by doing a single step of gradient descent:

$$\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t - \eta_t \nabla_{\boldsymbol{\theta}} \mathcal{L}_{\mathcal{B}}|_{\boldsymbol{\theta}=\boldsymbol{\theta}_t} \quad (\text{Eq. 10})$$

Where η_t is the *learning rate*. η_t can be set to a constant value or change throughout training epochs. The choice of learning rate is critical to convergence rate and stability. While choosing a too-high learning rate can lead to faster convergence it can overshoot the minimum region, choosing a too-low learning rate leads to slow convergence. Training can take a very long time to reach an acceptable level of performance. The optimal learning rate can vary greatly depending on the specific problem and the landscape of the loss function. Some problems may have a smooth, well-defined minimum, while others may have a more rugged landscape with many local minima. A learning rate that works well for one problem might not

be suitable for another.

Rather than choosing a single constant learning rate, a *learning rate schedule* can be used, in which the step size is adjusted over time. A related approach known as stochastic gradient descent with warm restarts, was proposed in (Loshchilov & Hutter, 2017). A new warm-started run or restart of SGD once T_i epochs are performed, where I is the index of the run. Importantly, the restarts are not performed from scratch but rather emulated by increasing the learning rate η_t while the old value of x_t is used as an initial solution. The amount of this increase controls to which extent the previously acquired information is used. Within the i -th run, the learning rate decays with a *cosine annealing* for each batch as follows:

$$\eta_t = \eta_{min}^i + \frac{1}{2}(\eta_{max}^i - \eta_{min}^i) \left(1 + \cos \left(\frac{T_{cur}}{T_i} \pi \right) \right) \quad (\text{Eq. 14})$$

where η_{min}^i and η_{max}^i are the ranges for the learning rate, and T_{cur} accounts for how many epochs have been performed since the last restart.

However, recent advances in adaptive optimizers—that dynamically adjust the learning rate based on historical gradient updates—have substantially reduced this issue. One of the most popular variants of SGD that we are going to use throughout this research is adaptive momentum, known as Adam (Kingma & Ba, 2015). Letting $\mathbf{g}_t = \nabla_{\theta} \mathcal{L}_{\mathcal{B}}|_{\theta=\theta_t}$ be the gradient of loss function \mathcal{L} with respect to the parameter θ evaluated at time t , the algorithm maintains exponential moving averages of both \mathbf{g}_t and $\mathbf{g}_t^2 = \mathbf{g} \odot \mathbf{g}$:

$$\mathbf{m}_{t+1} = \beta_1 \mathbf{m}_t + (1 - \beta_1) \mathbf{g}_t; \mathbf{s}_{t+1} = \beta_2 \mathbf{s}_t + (1 - \beta_2) \mathbf{g}_t^2 \quad (\text{Eq. 11})$$

The default values of the mixing constants are $\beta_1 = 0.9$ and $\beta_2 = 0.999$. This heavily biases the averages towards zero early on, so the following bias correction term is applied.

$$\hat{\mathbf{m}}_{t+1} = \frac{\mathbf{m}_t}{1 - \beta_1^t}; \hat{\mathbf{s}}_{t+1} = \frac{\mathbf{s}_t}{1 - \beta_2^t} \quad (\text{Eq. 12})$$

Then we perform the following parameter update:

$$\theta_{t+1} \leftarrow \theta_t - \eta_0 \frac{\hat{\mathbf{m}}_t}{\sqrt{\hat{\mathbf{s}}_t} + \varepsilon} \quad (\text{Eq. 13})$$

Where η_0 is the initial learning rate (which may now be set far more leniently), and ε is a small constant (usually 10^{-8}) design to protect against running into the numerical issue of dividing by zero.

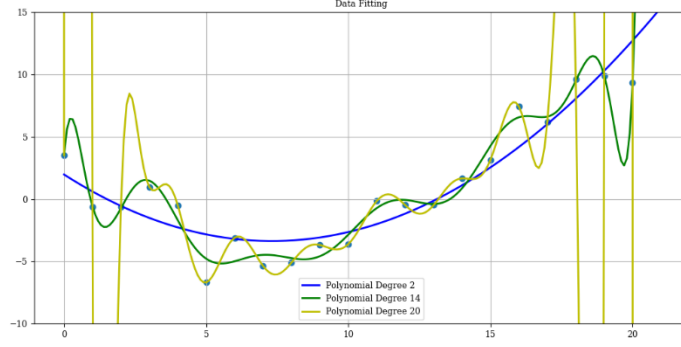


Figure 2.2.3 Polynomial degree of 2, 14, and 20 fit to 21 data point

A fundamental problem with MLE is that it will try to pick parameters that minimize loss on the training set, but this may not result in a model that has low loss on the future data. This is called *overfitting* (see **Figure 2.2.3** for an example). To protect neural networks against overfitting, *regularization* is usually the first line of defense. Broadly speaking, it requires the neural network to optimize for the same problem, under additional constraints that somehow restrict the set of parameters available to it during training—hopefully, in a way that discourages memorization. For the models deployed within this dissertation, three regularization techniques have been utilized, and I will describe them in turn, across separate paragraphs.

L_2 -Regularization. This regularizer imposes a prior belief (it is standard to use Gaussian prior for the weights $\mathcal{N}(\mathbf{w}|\mathbf{0}, \alpha^2 \mathbf{I})$ and biases, $\mathcal{N}(\mathbf{b}|\mathbf{0}, \beta^2 \mathbf{I})$) that the weights and biases of the network should not deviate too far from zero. Aside from constraining the model complexity, this may also yield clear numerical benefits. It is implemented as a weight penalty term—based on the L_2 norm—which is added to the loss function as follows:

$$\tilde{\mathcal{L}} = \mathcal{L} + \frac{\lambda}{2} \|\boldsymbol{\theta}\|^2 \quad (\text{Eq. 15})$$

where λ is a constant that controls the importance of penalizing the weights compared to optimizing the loss function. It should be chosen carefully. With this L_2 regularization, the gradient of the standard SGD at time t is modified to $\mathbf{g}_t = \nabla_{\boldsymbol{\theta}} \mathcal{L}_{\mathcal{B}}|_{\boldsymbol{\theta}=\boldsymbol{\theta}_t} + \lambda \boldsymbol{\theta}_{t-1}$, which is proved to be equivalent to SGD with weight decay, whose the parameter update of standard SGD is expressed as $\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t - \eta_t \nabla_{\boldsymbol{\theta}} \mathcal{L}_{\mathcal{B}}|_{\boldsymbol{\theta}=\boldsymbol{\theta}_t} - \eta_t \lambda \boldsymbol{\theta}_t$. However, in case of adaptive learning SGD such as Adam, L_2 regularization is not equivalent to weight decay since the gradient of regularizer gets scaled along with the gradient of the loss function $\mathcal{L}_{\mathcal{B}}$. Adaptive moment with decoupled weight decay (AdamW) is proposed by (Loshchilov & Hutter, 2019) to improve

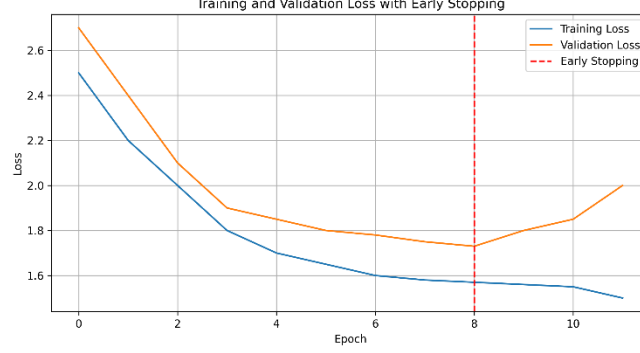


Figure 2.2.4 An illustration of early stopping with patience of 3 epochs.

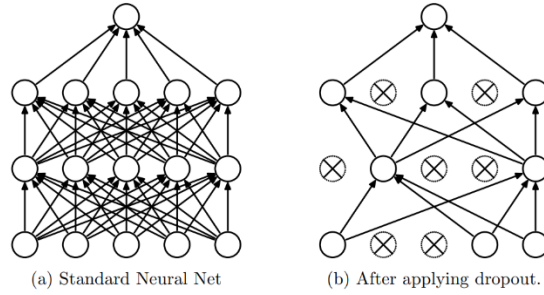


Figure 2.2.5 (Srivastava, Hinton, Krizhevsky, Sutskever, & Salakhutdinov, 2014) The effect of a single iteration of dropout on a feed forward network. (a) standard neural network, (b) after applying dropout

generalization ability of Adam, known as AdamW. To get the parameter update of AdamW, the parameter update of Adam in (Eq. 13) is modified to:

$$\theta_{t+1} \leftarrow \theta_t - \eta_t \left(\eta_0 \frac{\hat{m}_t}{\sqrt{\hat{s}_t + \varepsilon}} + \lambda \theta_t \right) \quad (\text{Eq. 16})$$

To improve the anytime performance of AdamW, (Loshchilov & Hutter, 2019) proposes to extend AdamW with the warm restarts introduced in (Loshchilov & Hutter, 2017) as discussed previously.

Early Stopping. Perhaps, it is the simplest way to prevent overfitting. It is referred to the heuristic of stopping the training procedure when the error on the validation set starts to increase (see **Figure 2.2.4** for an example). This method works because we are restricting the ability of the optimization algorithm to transfer information from the training examples to the parameters.

Dropout. (Srivastava, Hinton, Krizhevsky, Sutskever, & Salakhutdinov, 2014) This form of

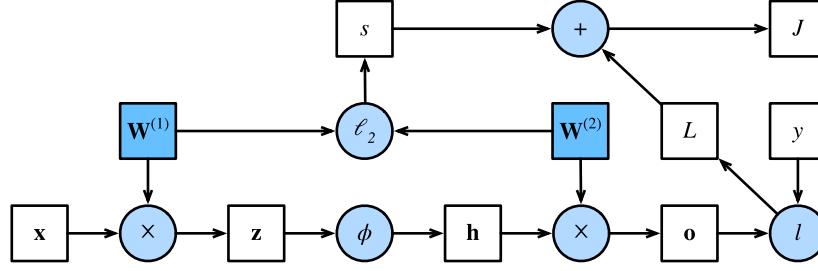


Figure 2.2.6 (Zhang A., 2020) Computational graph for an MLP with input \mathbf{x} , hidden layer \mathbf{h} , output \mathbf{o} , loss function $L = l(\mathbf{o}, y)$, and l_2 regularizer s on the weights, and total loss $J = L + s$.

regularization technique is achieved by randomly turning off all the outgoing connections from each neural with probability, p (see **Figure 2.2.5** for an example). Dropout can dramatically reduce overfitting and is very widely used. Intuitively, the reason dropout works well is that it prevents complex co-adaptation of the hidden units. In other words, each unit must learn to perform well even if some of the other units are missing at random. This prevents units from learning complex, but fragile, dependencies on each other.

Computational Graph. MLPs are a simple kind of DNN in which each layer feeds directly into the next, forming a chain structure. However, modern DNN can combine differentiable components in much more complex ways, to create a computation graph, analogous to how programmers combine elementary functions to make more complex ones. During data fitting, it is required to compute the gradient for backpropagation to update the DNN parameters. To avoid this repeated computation by working in reverse topological order, modern packages such as *PyTorch*, *TensorFlow*, and *JAX* allow the user to define the “forward” pass, which is the computational graph as illustrated in **Figure 2.2.6**. Given a computational graph corresponding to an MLP with one hidden layer with weight decay. More precisely, the model computes the linear pre-activation $\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x}$, the hidden activations $\mathbf{h} = \phi(\mathbf{z})$, the linear outputs $\mathbf{o} = \mathbf{W}^{(2)}\mathbf{h}$, the loss $L = l(\mathbf{o}, y)$, the regularizer $s = \frac{\lambda}{2} (\|\mathbf{W}^{(1)}\|_F^2 + \|\mathbf{W}^{(2)}\|_F^2)$, and the total loss $J = L + s$. Building this computational graph allows a package such as *Pytorch* to compute the gradient of a function with respect to the inputs by using automatic differentiation.

2.2.3. Recurrent Neural Networks

Another class of neural networks that deal with sequence models is the Recurrent Neural Network (RNN) (ELMAN, 1990). RNN takes an input space of sequences mapped to an output space of sequence in a stateful way. That is, the prediction of output y_t depends not

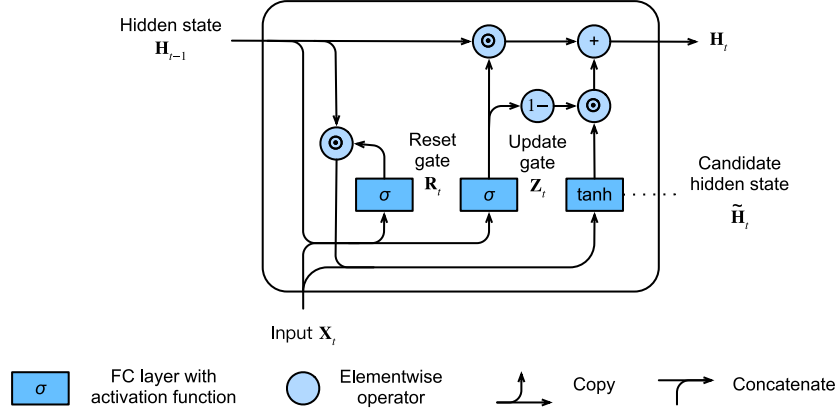


Figure 2.2.7 Gate Recurrent Unit (GRU) (Zhang A., 2020)

only on the input x_t , but also on the hidden state of system, h_t , which gets updated over time, as the sequence is processed. The update function f is usually given (Eq. 16) by

$$h_{t+1} = \varphi(W_{xh}[x, y_t] + W_{hh}h_t + b_h) \quad (\text{Eq. 16})$$

where W_{hh} are the hidden-to-hidden weights, W_{xh} are the input-to-hidden weights, and b_h are the biases terms.

Unfortunately, the activations in an RNN can decay or explode as we go forward in time since we multiply by the weight matrix W_{hh} at each time step. Similarly, the gradients in an RNN can decay or explode as we go backward in time since we multiply the Jacobians at each time step. To overcome these numerical problems, the LSTM architecture (Hochreiter, 1997) introduced error carousal and gating techniques that allow LSTM to scale to deep layers to learn long sequences. Due to rapidly growing popularity during the 2010s, a large number of researchers began to experiment with simplified architectures in hopes of retaining the key idea of incorporating an internal state and multiplicative gating mechanisms to speed up computation. The gated recurrent unit (GRU) (Cho, et al., 2014) offered a streamlined version of the LSTM memory cell that often achieves comparable performance but with the advantage of being faster to compute (Chung, Gulcehre, Cho, & Bengio, 2014). The key idea of GRU is to learn when to update the hidden state, by using a gating unit. This can be used to selectively “remember” important pieces of information when they are first seen. The model can also learn when to reset the hidden state, and thus forget things that are no longer useful. A GRU as depicted in **Figure 2.2.7** can be expressed as follows.

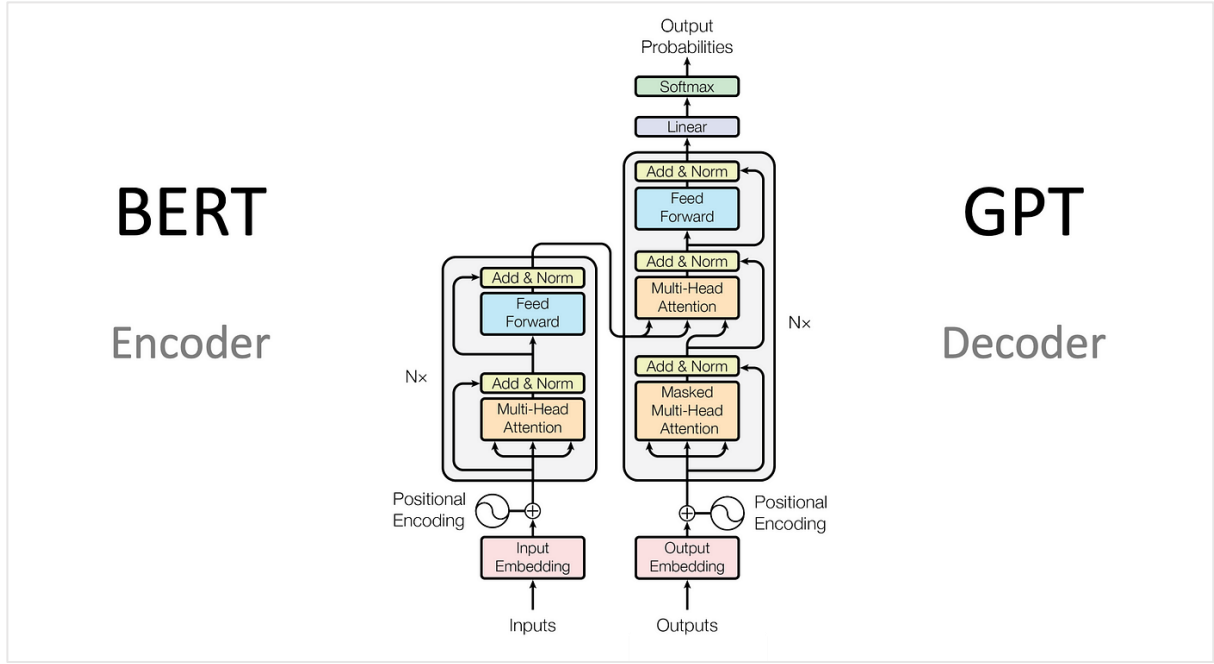


Figure 2.2.8 Transformer architecture (Vaswani, et al., 2017). While BERT’s backbone employs Encoder block, GPT’s backbone employs Decoder block.

$$\mathbf{R}_t = \sigma(\mathbf{X}_t \mathbf{W}_{xr} + \mathbf{H}_{t-1} \mathbf{W}_{hr} + \mathbf{b}_r) \quad (\text{Eq. 17})$$

$$\mathbf{Z}_t = \sigma(\mathbf{X}_t \mathbf{W}_{xz} + \mathbf{H}_{t-1} \mathbf{W}_{hz} + \mathbf{b}_z) \quad (\text{Eq. 18})$$

$$\tilde{\mathbf{H}}_t = \tanh(\mathbf{X}_t \mathbf{W}_{xh} + (\mathbf{R}_t \odot \mathbf{H}_{t-1}) \mathbf{W}_{hh} + \mathbf{b}_h) \quad (\text{Eq. 19})$$

$$\mathbf{H}_t = \mathbf{Z}_t \odot \mathbf{H}_{t-1} + (1 - \mathbf{Z}_t) \odot \tilde{\mathbf{H}}_t \quad (\text{Eq. 20})$$

2.2.4. Transformer

Motivated by neural machine translation problem, (Vaswani, et al., 2017) introduces the Transformer architecture, which is undoubtedly the main backbone behind the success of modern Large Language Models (LLMs) such as BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding (Devlin, Chang, Lee, & Toutanova, 2019), Generative Pre-trained Transformer (GPT-3) (Brown, et al., 2020), and Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context (Georgiev, et al., 2024).

The main core module of the Transformer architecture is the self-attention mechanism. It scales the dot-product attention (Bahdanau, Cho, & Bengio, 2015) by $1/\sqrt{d_k}$ which is called scale-dot product attention. It can be expressed by:

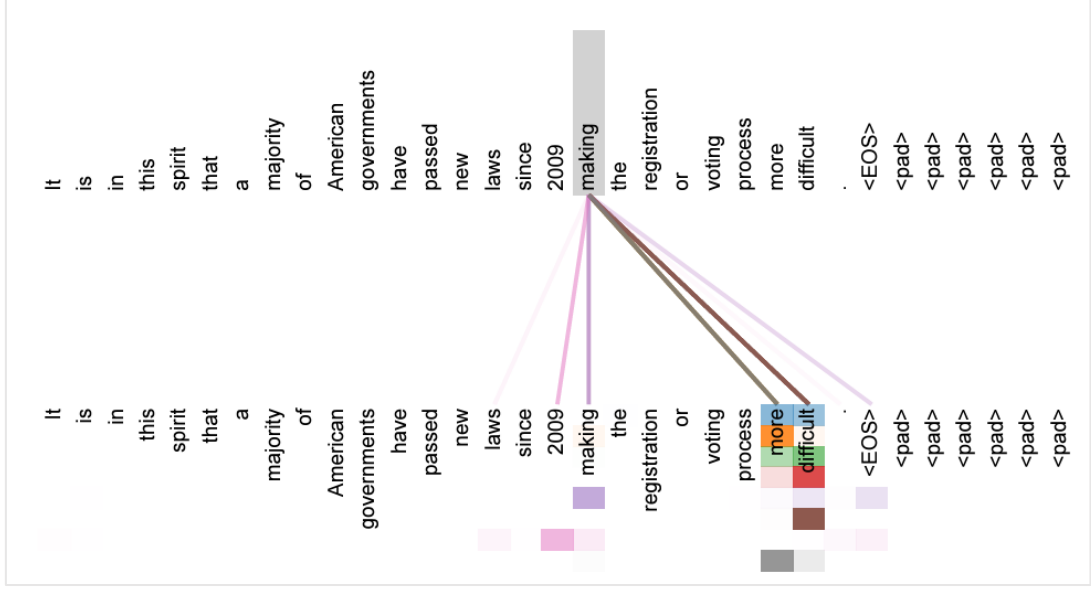


Figure 2.2.9 Attention visualizations (Vaswani, et al., 2017) for a sequence of words. Many of the attention heads attend to a distant dependency of the verb ‘making’, completing the phrase ‘making...more difficult’. Attentions here shown only for the word ‘making’.

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right) \mathbf{V} \quad (\text{Eq. 21})$$

where the “softmax” function is a function that maps inputs to a probability distribution over the C possible output labels. It is defined as:

$$\text{softmax}(\mathbf{x}) \triangleq \left[\frac{e^{x_1}}{\sum_{c'=1}^C e^{x_{c'}}}, \dots, \frac{e^{x_C}}{\sum_{c'=1}^C e^{x_{c'}}} \right] \quad (\text{Eq. 22})$$

This maps \mathbb{R}^C to $[0,1]^C$, and satisfies the constraints that $0 \leq \text{softmax}(\mathbf{x})_c \leq 1$ and $\sum_{c=1}^C \text{softmax}(\mathbf{x})_c = 1$.

The success of the Transformer also relies on its interpretability. As illustrated in **Figure 2.2.9**, each word in the sequence attends differently for every word in the sequence.

In analogous to Graph Neural Networks (GNN) which will be discussed in **Section 2.4**, it can be seen that the Transformer is actually a GNNs (Joshi, 2020). In fact, the basic transformer layer is exactly equivalent to a GNN layer using multi-headed attention if we assume that the GNN received fully connected graph as input (Hamilton, Ying, & Leskovec, 2017). Concretely, we can consider each word in the sequence as the node and the attention weight to each other word within the sequence as the edge that passes the message from the destination node the source node.

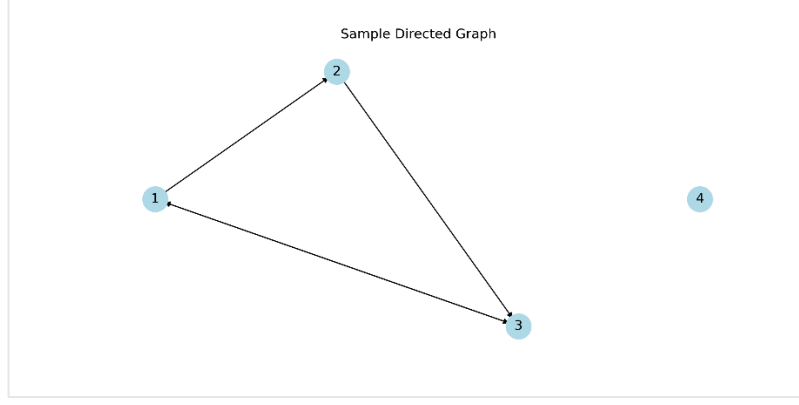


Figure 2.2.10 An illustration of directed graph.

To improve the model generalization and training stability, single head attention is further extended to multi-head attention which can be expressed as below:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \text{head}_2, \dots, \text{head}_h)W^O \quad (\text{Eq. 23})$$

Where $\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$

Where the projections are parameter matrices $W_i^Q \in \mathbb{R}^{(d_{\text{model}} \times d_k)}$, $W_i^K \in \mathbb{R}^{(d_{\text{model}} \times d_k)}$, $W_i^V \in \mathbb{R}^{(d_{\text{model}} \times d_k)}$, and $W_i^O \in \mathbb{R}^{(d_{\text{model}} \times d_k)}$.

2.2.5. Neural Networks Optimization

2.3. Graph Data Structure

Before starting the discussion about *Graph Neural Networks (GNN)*, it is useful to provide a formal definition of graph data.

Definition 2.3.1. (Graph Data) A graph $G = (V, E)$ is defined by a set of nodes \mathcal{V} and a set of edge $E \subseteq V \times V$ between these nodes. An edge going from $v_i \in V$ to $v_j \in V$ is denoted as $(v_i, v_j) \in E$. A graph may have node attributes \mathbf{X} , where $\mathbf{X} \in \mathbb{R}^{(n \times d)}$ is a node feature matrix with $\mathbf{x}_v \in \mathbb{R}^d$ representing the feature vector of a node v . Meanwhile, a graph may have associated edge attributes \mathbf{X}^e , where $\mathbf{X}^e \in \mathbb{R}^{(m \times c)}$ is an edge feature matrix with $\mathbf{x}_{(v,u)}^e \in \mathbb{R}^c$ representing the feature vector of an edge (v, u) .

There are two ways to construct a graph. The first way is to construct the graph through *Adjacency Matrix* $\mathbf{A} \in \mathbb{R}^{(V \times V)}$. To represent the graph data with the adjacency matrix, nodes are organized into ordered indices of the column and row of the matrix. The presence of an edge within the graph is captured by the value of i -th row and j -th column in the matrix, i.e.

$A_{ij} = 1$ if $(v_i, v_j) \in E$ and $A_{ij} = 0$ otherwise. Concretely, the graph with a set of nodes $\{\textcircled{1}, \textcircled{2}, \textcircled{3}, \textcircled{4}\}$.

illustrated in **Figure 2.2.10**. can be represented by the below matrix (Eq. 24):

$$A = \begin{matrix} & \begin{matrix} \textcircled{1} & \textcircled{2} & \textcircled{3} & \textcircled{4} \end{matrix} \\ \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \end{matrix} \quad (\text{Eq. 24})$$

An alternative and efficient way to represent the graph is through a *sparse matrix* $A \in \mathbb{R}^E$ since the adjacency representation required quadratic in the number of nodes $\mathcal{O}(V^2)$. This representation can reduce the memory requirements to $\mathcal{O}(E)$. This is achieved by pairing the source and destination nodes in separate rows of the matrix. Concretely, the alternative sparse representation can be represented by a tuple of source and destination nodes as below equation:

$$E = ((1,2), (2,3), (1,3), (3,1)) \quad (\text{Eq. 25})$$

2.4. Learning Settings on Static Graphs

The machine learning literature distinguishes two types of learning. First, *Transductive learning*, in which the training algorithm has access to the entirety of the input graph’s features, including the test nodes. Second, *Inductive learning*, in which the algorithm will not have access to all nodes upfront. While transductive learning deals with an unchanged graph and may be seen as a semi-supervised task of propagating labels from training nodes to the remainder of the graph, inductive learning deals with an evolving graph wherein test nodes are incrementally added or, more generally, there exist disjoint and unseen test graph. Inductive learning is a substantially harder learning problem, which requires generalizing across arbitrary graph structures, and many transductive graph learning problems will be theoretically inappropriate in the transductive setting (Veličković, 2019).

2.5. An Overview of Graph Neural Networks

Graph Neural Networks (GNN) represent a powerful learning paradigm that has achieved great success (Scarselli, Gori, Tsoi, Hagenbuchner, & Monfardini, 2008; Akoglu, Chandy, & Faloutsos, 2013; Hamilton, Ying, & Leskovec, 2017; Veličković, et al., 2018; Xu, Hu, Leskovec, & Jegelka, 2020; You, Ying, & Leskovec, 2020; Derrow-Pinion, et al., 2021; Ying, et al., 2018; Zhu, et al., 2019). Among these models, message-passing GNNs, such

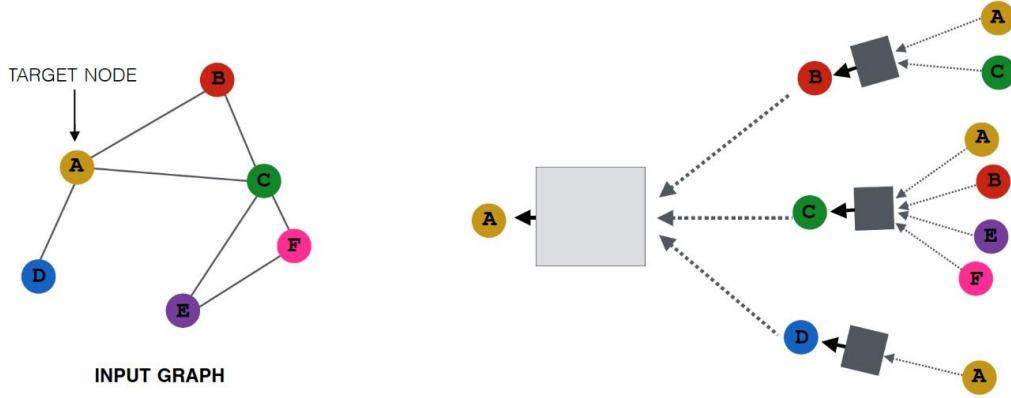


Figure 2.5.1 (Hamilton W. L., 2020) An illustration of 2-hop ($k = 2$) neighborhood aggregation by unfolding the neighborhood around the target node.

For instance, node **A** is aggregated information from node **B**, **C**, and **D**. The corresponding neighborhood of **B**, **C**, and **D** recursively aggregated information from its neighborhood.

as Graph Convolutional Networks (GCN) (Kipf & Welling, 2017), GraphSAGE (Hamilton, Ying, & Leskovec, Inductive Representation Learning on Large Graphs, 2017), Graph Attention Network (GAT) (Velickovic, et al., 2018), are dominantly used today due to their simplicity, efficiency and strong performance in a real-world application. The central idea behind message-passing GNN is to learn a *hidden embeddings* $\mathbf{h}_u^{(k)}$ corresponding to each node $u \in V$ is updated according to information aggregated from u 's graph neighborhood $\mathcal{N}(u)$ illustrated **Figure 2.5.1**. The message-passing GNN update can be expressed (Hamilton W. L., Graph Representation Learning, 2020) as below:

$$\mathbf{h}_u^{(k+1)} = \text{UPDATE}^{(k)}\left(\mathbf{h}_u^{(k)}, \text{AGGREGATE}^{(k)}\left(\mathbf{h}_v^{(k)}, \forall v \in \mathcal{N}(u)\right)\right) \quad (\text{Eq. 26})$$

$$= \text{UPDATE}^{(k)}\left(\mathbf{h}_u^{(k)}, \mathbf{m}_{(\mathcal{N}(u))}^{(k)}\right) \quad (\text{Eq. 27})$$

Where *UPDATE* and *AGGREGATE* are arbitrary differentiable functions and $\mathbf{m}_{\mathcal{N}(u)}$ is the “message” that is aggregated from u 's graph neighborhood $\mathcal{N}(u)$.

One of the most popular GNN baselines with a strong theoretical foundation and vast application is Graph Convolutional Network (GCN) (Kipf & Welling, 2017). The GCN is a graph-based neural networks model $f(X, A)$ with message-passing operations that can be motivated as first-order (i.e. linear) approximation to spectral graph convolutions, follow by a

non-linear activation function. Let $h_u^{(k)} \in \mathbb{R}^{(d_k)}$ be the hidden representation vector of node $i \in V$ with dimensionality d_k after the k th-layer message passing step, then a single message passing step in GCN model takes the following form:

$$h_u^{(k+1)} = \sigma \left(\mathbf{W}_0^{(k)T} h_u^{(k)} + \sum_{v \in \mathcal{N}(u)} \underbrace{\frac{\mathbf{W}_1^{(k)T} h_v^{(k)}}{\sqrt{|\mathcal{N}(u)| |\mathcal{N}(v)|}}}_{=\mathbf{m}_{\mathcal{N}(u)}^{(k)}} \right) \quad (\text{Eq. 28})$$

Where $\mathbf{W}_0^{(k)}$ and $\mathbf{W}_1^{(k)}$ are learnable parameters at layer k . The *AGGREGATE* or $\mathbf{m}_{\mathcal{N}(u)}^{(k)}$ operator is corresponding to the sum over the neighborhood nodes normalized by the node degree, called symmetric normalization. The *UPDATE* operator in is corresponding to apply layer-wise non-linearity σ to, such as the ReLU activation function, the self-loop cumulated with the $\mathbf{m}_{\mathcal{N}(u)}^{(k)}$. It is worth mentioning that symmetric normalization allows for the emergence of more interesting dynamics and can be related to a particular approximation of applying the convolution theorem to signal defined on graph (Bruna, Zaremba, Szlam, & LeCun, 2014; Defferrard, Bresson, & Vandergheynst, 2016). This causes the layer to be simple, powerful, and robust to overfitting – however, it is in theory not applicable to inductive problems, as knowledge of neighborhood sizes implies that the learned filters rely on knowing the entire graph structure upfront.

To overcome the domain dependency and storage limitations of GCN, Graph Sampling and Aggregation (GraphSAGE) (Hamilton, Ying, & Leskovec, Inductive Representation Learning on Large Graphs, 2017) extends the GCN framework and proposes a general *inductive* framework that leverages node features and information to efficiently generate node embeddings for previously unseen nodes. Instead of averaging signals from all one-hop neighbors, GraphSAGE sample fixed d -neighborhoods $\mathcal{N}_d(v)$ for each node v . This removes the strong dependency on fixed information from nodes sampled from the neighborhood. GraphSAGE propagation expression can be written as:

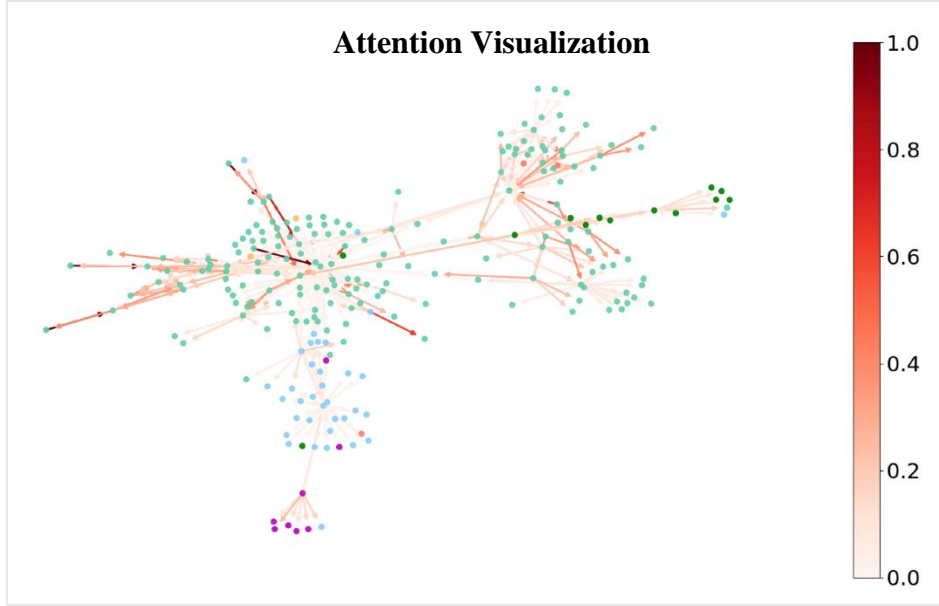


Figure 2.5.2 Plot of edge's attention weight on sub-graph of Cora's dataset.

$$\mathbf{h}_{\mathcal{N}_d(v)}^{(k)} = \text{AGGREGATE} \left(\mathbf{h}_u^{(k-1)} \middle| \forall u \in \mathcal{N}_d(v) \right) \quad (\text{Eq. 29})$$

$$\mathbf{h}_v^{(k)} = \sigma \left(\mathbf{W}^{(k)} \cdot \left[\mathbf{h}_v^{(k-1)} \parallel \mathbf{h}_{\mathcal{N}_d(v)}^{(k)} \right] \right) \quad (\text{Eq. 30})$$

Where the *AGGREGATE* function can be any permutation invariant operator such as mean pooling or max pooling.

Attention mechanisms discussed in **Section 2.2.4** have been successfully used in language models where they allow models to identify relevant parts of the long sequence inputs. Inspired by their success, similar ideas have been proposed for graph convolution networks. Such graph-based attention models learn to focus their attention on important neighbors during the message-passing step via parametric patches which are learned on top of node features. This provides more flexibility in inductive settings, compared to methods that rely on fixed weights such as GCN (Kevin, 2022).

The Graph Attention Networks (GAT) model is an attention-based version of GCNs. At every GAT layer, it attends over the neighborhood of each node, as illustrated in **Figure 2.5.2**, and learns to selectively pick nodes that lead to the best performance for some downstream tasks.

Unified Message Passing (UniMP) (Shi, et al., 2021) unifies the message-passing Graph Neural Network (GNN) and Label Propagation Algorithm (LPA), a traditional algorithm

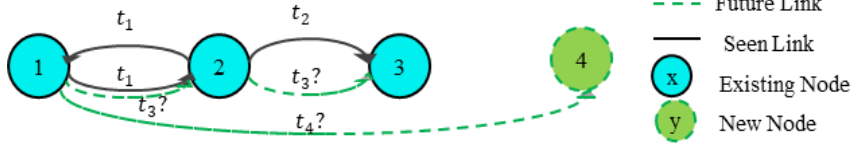


Figure 2.5.3 An illustration of temporal graph network.

that only utilizes labels and relations between nodes to predict by assuming the similarity between the connected nodes. UniMP adopts vanilla Transformer’s multi-head attention into graph learning with considering the case of edges features. Each head of attention-based embedding update $h_u^{(k+1)}$ can be expressed as below:

$$h_u^{(k+1)} = W_1 h_u^{(k)} + \sum_{v \in \mathcal{N}(u)} \alpha_{(u,v)} (W_2 h_v^{(k)} + W_6 e_{(uv)}) \quad (\text{Eq. 31})$$

Where the attention coefficient $\alpha_{(u,v)}$ are computed via:

$$\alpha_{(u,v)} = \text{softmax} \left(\frac{(W_3 h_u^{(k)})^T (W_4 h_v^{(k)} + W_6 e_{(uv)})}{\sqrt{d}} \right) \quad (\text{Eq. 32})$$

2.6. Temporal Graph Learning

2.6.1. Temporal Graph Data Structure and Learning on Temporal Graphs

As we gain an understanding of the static graph data, we extend it into general temporal graph (or dynamic graph) data.

Definition 2.5.1 (Temporal Graph or Dynamic Graph) A temporal graph is defined as a sequence of non-decreasing chronological interactions $G = (u_1, v_1, t_1), (u_2, v_2, t_2), \dots, (u_N, v_N, t_N)$ with $0 \leq t_1 \leq t_2 \leq \dots \leq t_N$, where $u_i, v_i \in V$ denote the source node and destination node of the i -th link at timestamp t_i . V is the set of all the nodes in the graph. A graph may have associated node attributes \mathbf{X} , where $\mathbf{X} \in \mathbb{R}^{(n \times d)}$ is a node feature matrix with $\mathbf{x}_v \in \mathbb{R}^d$ representing the feature vector of a node v . Meanwhile, a graph may have associated edge attributes $\mathbf{X}^{e_{uv}}$, where $\mathbf{X}^{e_{uv}} \in \mathbb{R}^{(m \times c)}$ is an edge feature matrix with $\mathbf{x}_{(v,u)}^e \in \mathbb{R}^c$ representing the feature vector of an edge $(v, u) \in E_t$.

Temporal graph network as illustrated in **Figure 2.5.3** is a continuous time-directed graph whose edges and nodes are added or deleted at any given time. It can be extended to an

undirected graph if there exist two connections in both directions. The goal of learning in temporal graphs consists of predicting the future interaction for *link prediction* tasks and node properties change for *node prediction* tasks as the nodes, edges, and properties continually change over time. For our purpose of study, we provide the formal definition below.

Definition 2.5.2 (Dynamic Link Prediction) *Given the source node u , destination node v , at timestamp t , and historical interactions before time t , i.e., $\{u', v', t'\} | t' < t\}$, the task is to design a model to learn time-aware representation $\mathbf{h}_u^t \in \mathbb{R}^d$ and $\mathbf{h}_v^t \in \mathbb{R}^d$ for node u and node v with d as the output dimension for predicting the future interaction between node u and node v .*

2.6.2. Random Walk on Temporal Graphs

Definition 2.5.3 (Random Walk) *Given a graph $G = (V, E)$ where V is a set of nodes, and E is a set of edges connecting those nodes. A random walk of length l on graph G starting at node v , if the i -th node is u , then the $(i + 1)$ -th node is chosen at random uniformly and independently among the neighbors of node u .*

Definition 2.5.4 (Temporal Graph Motif) *Given the set of temporal edge $E_{v,t} = \{(e, t') \in E | t' < t, v \in e\}$ to include the links attached to a node v before certain time t . A walk W (reverse over time) on temporal networks can be represented as*

$$W = ((w_0, t_0), \dots, (w_m, t_m), t_0 > \dots > t_m, (\{w_{i-1}, w_i\}, t_i) \in E \text{ for all } i. \quad (\text{Eq. 33})$$

Each walk that connects subgraphs by links appearing within a restricted time range is called the temporal graph motif.

An *anonymous walk*, which was first considered by (Micali & Zhu, 2016) to study the reconstruction of a Markov process, a kind of random walk that has anonymized the node identities by the orders of their appearance in each walk, which is termed relative node identities in AW and defined it as

$$I_A W(w, W) \triangleq |v_0, v_1, \dots, v_{k^*}| \text{ where } k^* \text{ is the smallest index s.t } v_{k^*} = w \quad (\text{Eq. 34})$$

This anonymization is proved to be an *inductive* property of the models (Wang, Chang, Liu, Leskovec, & Li, 2021).

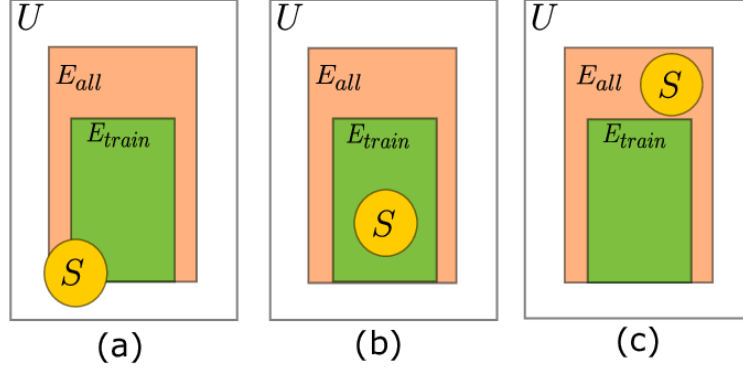


Figure 2.6.1 (Poursafaei, Huang, Pelrine, & Rabbany, 2022) Negative edge sampling strategies during evaluation for dynamic link prediction; (a) random sampling, (b) historical sampling, and (c) inductive sampling

2.6.3. Negative Sampling in Temporal Graphs

To evaluate the performance of the algorithm learning in different settings, i.e. transductive and inductive tasks discussed in **Section 2.4**, a proper negative edge, an edge that does not exist in the graph, i.e. $e_{uv} \notin E_{all}$, negative sampling (NS) strategy must be selected. It can be categorized into 3 categories as below:

- **Random Negative Sampling.** The edges are randomly sampled from almost all possible node pairs of the graphs (Kumar, Zhang, & Leskovec, 2019; Rossi, et al., 2020; Trivedi, Farajtabar, Biswal, & Zha, 2019; Wang, Chang, Liu, Leskovec, & Li, 2021; Xu, Ruan, Korpeoglu, Kumar, & Achan, 2020), as illustrated in **Figure 2.6.2** (a). For instance, at each timestamp, there exists a set of positive edges consisting of a source node u and a destination node v together with edge timestamps t and edge feature e_{uv}^t . To generate negative samples, the standard procedure is to keep the timestamps, features, and source nodes of the positive edges while choosing destination nodes randomly from all nodes.
- **Historical Negative Sampling.** In historical NS, the negative edges are sampled from the set of edges that have been observed during previous timestamps E_{train} but are absent in the current step \bar{E}_t . The objective of this strategy is to evaluate whether a given method can predict which timestamps an edge would reoccur, rather than, for example, naively predicting it always reoccurs whenever it has been seen once. Therefore, in historical NS, for a given time step t , we sample from the edges $e = (u, v) \in (E_{train} \cap \bar{E}_t)$, illustrated in **Figure 2.6.2** (b).

- **Inductive Negative Sampling.** While in historical NS the observed training edges is mainly focused, in inductive NS, our focus is to evaluate whether a given method can model the reoccurrence pattern of edges only seen during test time. At test time, after observing the edges that were not seen during training, the model is asked to predict if such edges exist in future steps of the test phase. Therefore, in the inductive NS, we sample from the edges $e \in (E_{test} \cap E_{train} \cap E_t)$ at time step t , illustrated in **Figure 2.6.2 (c)**. As these edges are not observed during training, they are considered inductive edges. Note that in both historical and inductive NS if the number of available negative edges of the given type is less than the number of positive edges, the remaining negative edges are sampled by the random NS strategy.

2.6.4. Existing Temporal Graph Learning Architecture

In this section, we are interested in extending the graph neural network from static to dynamic graph. Neighbor aggregation is one of the most used in various graph models as a crucial ingredient, where the embeddings of its neighbors. Dynamic graph neighborhood aggregation can be viewed as a straightforward extension of classic graph convolution layers (Kipf & Welling, 2017; Velickovic, et al., 2018) with additional time or sequence information discussed in 2.3. In the context of a temporal graph, the neighbor aggregation layer can be defined as follows (Souza, Mesquita, Kaski, & Garg, 2022; Xu, Ruan, Korpeoglu, Kumar, & Achan, 2020; Rossi, et al., 2020; Chen, et al., 2023).

$$\mathbf{m}_u^{(k)} = AGGREGATE^{(k)} \left(\left(\mathbf{h}_v^{(k-1)}(t), \Phi(e_{uv,t'}) \right) \middle| e_{uv,t'} \in E_t \right) \quad (\text{Eq. 35})$$

$$\mathbf{h}_u^{(k)} = UPDATE^{(k)} \left(\mathbf{h}_u^{(k-1)}, \mathbf{m}_u^{(k)} \right) \quad (\text{Eq. 36})$$

Where t denotes the current time. $e_{uv,t'}$ denotes the event between node u and node v at time $t'(\leq t)$. $\Phi(e_{uv,t'})$ encode the event features, as well as its related time and sequence information, which may include the current time, the event time, the current number of interactions node u and node v , etc. Like static graphs, $AGGREGATE^{(k)}$ and $UPDATE^{(k)}$ denotes arbitrary learnable functions, which are typically shared at layer k . $\mathbf{m}^{(k)}$ is the message function that collects the information from neighbors at the $(k-1)$ -th layer, which $\mathbf{h}_u^{(k)}$ is the new embedding of node u at layer k , combining its current embedding at the $(k-1)$ -th layer with the message information $\mathbf{m}_u^{(k)}$.

Joint Dynamic User-Item Embedding (JODIE). (Kumar, Zhang, & Leskovec, 2019) is designed specifically for bipartite graphs of user-item interactions. When an interaction between user v_i and item v_j , JODIE update the temporal embedding of user $x_i(t)$ (also update items $x_j(t)$ separately) using RNN by

$$x_i(t) = RNN(x_i(t_i^-), x_j(t_j^-), e_{ij}^t, t - t_i^-) \quad (\text{Eq. 37})$$

Where t_i^- is the latest interaction before current timestamp t .

Then the dynamic embedding of node v_i at time t_0 is computed by

$$h_i(t_0) = (1 + (t_0 - t^-)W) \cdot x_i(t^-) \quad (\text{Eq. 38})$$

Finally, the prediction on any user-item pair interaction at time t_0 is computed by $FFN([h_i(t_0) \parallel h_j(t_0)])$ where \parallel denote concatenation operator.

Temporal Graph Attention (TGAT). (Xu, Ruan, Korpeoglu, Kumar, & Achan, 2020) leverage Brochner’s theorem to derive the translation invariant learnable functional time encoder by converting the problem of kernel learning to distribution learning. The resulting formula can be expressed as (Eq. 53).

$$t \mapsto \Phi_d(t) := \sqrt{\frac{1}{d}} [\cos(\omega_1 t), \sin(\omega_1 t), \dots, \cos(\omega_d t), \sin(\omega_d t)] \quad (\text{Eq. 39})$$

Thanks to this time-translation invariance time encoder function, TGAT can use relative time $t - t_1, \dots, t - t_N$ as the interaction times since $|t_t - t_j| = |(t - t_i) - (t - t_j)|$, integrate it with previous layer embedding $\tilde{h}_0^{(k-1)}, \dots, \tilde{h}_N^{(k-1)}$ to obtain the current layer entity-temporal feature matrix as.

$$\mathbf{Z}(t) = [\tilde{h}_0^{(k-1)}(t) \parallel \Phi_{d_T}(0), \dots, \tilde{h}_N^{(k-1)}(t) \parallel \Phi_{d_T}(N)] \quad (\text{Eq. 40})$$

This feature matrix $\mathbf{Z}(t)$ are then projected to three “query”, “key”, and “value” matrix:

$$\mathbf{q}(t) = [\mathbf{Z}(t)]_0 \mathbf{W}_Q, \mathbf{K}(t) = [\mathbf{Z}(t)]_{1:N} \mathbf{W}_K, \mathbf{V}(t) = [\mathbf{Z}(t)]_{1:N} \mathbf{W}_V \quad (\text{Eq. 41})$$

In line with vanilla Transformer discussed in 2.2.4 and GAT/UniMP discussed in 2.6.4, this key, value, and value matrix can be pass forward into multi-head attention and feed forward network (FFN) obtaining the target node embedding $\tilde{h}_0^{(k)}$.

Temporal Graph Network (TGN). (Rossi, et al., 2021) provides a generic, scalable, and efficient framework to model dynamic graphs. It addresses the challenges of dynamic graphs by incorporating memory modules and a message-passing mechanism that can capture both structural and temporal information. The memory modules store historical node embeddings, while the message-passing mechanism allows nodes to exchange information with their neighbors. TGN can efficiently handle dynamic graphs by employing a combination of attention mechanisms and temporal aggregators to learn node representations that capture the local and global temporal dependencies.

GraphMixer (Cong, et al., 2023). Motivated by the importance of simpler model architecture, GraphMixer proposes a simple yet powerful architecture for link prediction that mainly consists of 3 modules which are link encoder, node encoder, and link classifier. Instead of using trainable time encoder functions like TGAT (Xu, Ruan, Korpoglu, Kumar, & Achan, 2020) and TGN (Rossi, et al., 2021), GraphMixer proposes to use a static time encoder function that is proved to be more stable during model training than the learnable time encoder function. Despite of strong performance of RNN-based or Self-Attention Mechanisms (SAM) based TGL method, GraphMixer proves that neither of them is necessary. Inspired by computer vision architecture’s MLP Mixer (Tolstikhin, et al., 2021), GraphMixer uses a 1-layer MLP Mixer to summarize the temporal link information. Specifically, it first encodes the relative timestamp using a static time function $\cos(\omega(\mathbf{t}_0 - \mathbf{t}))$ then concatenate it with its corresponding link features \mathbf{e}_{uv}^t . Since the node in the graph may have a different number of temporal neighborhoods, to obtain a matrix $\mathbf{T}_2(t_0)$ by stacking each node’s temporal link information, it uses zero padding to match the dimension. Meanwhile, it is also crucial to capture the interaction frequency by the number of zero-padded. The result node’s temporal link information $\mathbf{h}_2(t_0) = \text{Mean}(\mathbf{H}_{channel})$ whose $\mathbf{H}_{channel}$ is computed by the below expressions.

$$\mathbf{H}_{token} = \mathbf{H}_{input} + \mathbf{W}_{token}^{(2)} \text{GeLU} \left(\mathbf{W}_{token}^{(1)} \text{LayerNorm}(\mathbf{T}_2(t_0)) \right) \quad (\text{Eq. 42})$$

$$\mathbf{H}_{channel} = \mathbf{H}_{token} + \text{GeLU} \left(\text{LayerNorm}(\mathbf{H}_{token}) \mathbf{W}_{channel}^{(1)} \right) \mathbf{W}_{channel}^{(2)} \quad (\text{Eq. 43})$$

EdgeBank. (Poursafaei, Huang, Pelrine, & Rabbany, 2022) proposes a simple baseline that is purely based on a memory approach without trainable parameters for transductive dynamic link prediction. It stores the observed interactions in the memory unit and updates the memory through various strategies. An interaction will be predicted as positive if it is retained in the

memory, and negative otherwise. There are two variants of EdgeBank: EdgeBank_∞ with unlimited memory to store all the observed edges; EdgeBank_{tw}, which only remembers edges within a fixed-size time window from the immediate past.

DyGFormer. Jointly learn the sequence of the first-hop historical neighborhood of pair nodes, S_u^t and S_v^t within the interaction by computing neighbor co-occurrences $\mathbf{C}_u^t \in \mathbb{R}^{|S_u^t| \times 2}$ and $\mathbf{C}_v^t \in \mathbb{R}^{|S_v^t| \times 2}$ to capture the correlations of the pair nodes. It is then applied FFN to obtain neighbor co-occurrence encoding (NCoE) of node u (also node v):

$$\mathbf{X}_{u,c}^t = \text{FFN}(\mathbf{C}_u^t[:, 0]) + \text{FFN}(\mathbf{C}_u^t[:, 1]) \in \mathbb{R}^{|S_u^t| \times d_c} \quad (\text{Eq. 44})$$

This NCoE $\mathbf{X}_{u,c}^t$, node u 's neighbor feature $\mathbf{X}_{u,N(u)}^t \in \mathbb{R}^{|S_u^t| \times d_N}$, relative time encoded feature as used by TGAT $\mathbf{X}_{u,T}^t \in \mathbb{R}^{S_u^t \times d_T}$, and interaction feature $\mathbf{X}_{u,E}^t \in \mathbb{R}^{S_u^t \times d_E}$ are then chunked into multiple patches of length $l_u^t = \left\lfloor \frac{S_u^t}{P} \right\rfloor$ whose P is patch size. This patching technique allows DyGFormer to scale to long sequences and memorize longer histories of interactions. It is then applied linear transformations of those 4 patched matrices \mathbf{M}_u^t and concatenate together to obtain the feature encoding \mathbf{Z}_u^t (also \mathbf{Z}_v^t) feeding into the encoder block of the Transformer (Vaswani, et al., 2017) with GeLU (Gimpel, 2016) in place of ReLU (Nair & Hinton, 2010) and applying LayerNorm (Ba, Kiros, & Hinton, 2016) before each block rather than after.

$$\mathbf{H}_t = \text{Transformer}([\mathbf{Z}_u^t \parallel \mathbf{Z}_v^t]) \in \mathbb{R}^{(l_u^t + l_v^t) \times 4d} \quad (\text{Eq. 45})$$

The time-aware node representations of node u and node v at timestamp t are derived by averaging the related representations in \mathbf{H}^t with an output layer projection

$$\mathbf{h}_u^t = \text{MEAN}(\mathbf{H}^t[:, l_u^t, :])\mathbf{W}_{out} + \mathbf{b}_{out} \in \mathbb{R}^{d_{out}} \quad (\text{Eq. 46})$$

$$\mathbf{h}_v^t = \text{MEAN}(\mathbf{H}^t[l_u^t: l_u^t + l_v^t, :])\mathbf{W}_{out} + \mathbf{b}_{out} \in \mathbb{R}^{d_{out}} \quad (\text{Eq. 47})$$

Where $\mathbf{W}_{out} \in \mathbb{R}^{4d \times d_{out}}$ and $\mathbf{b}_{out} \in \mathbb{R}^{d_{out}}$ are learnable parameters with d_{out} as the output dimension.

DyRep (Trivedi, Farajtabar, Biswal, & Zha, 2019) proposes a novel modeling framework for dynamic graphs that posits representation learning as a *latent mediation process* bridging two observed processes, namely dynamics of the network (realized as topological evolution) and dynamics on the network (realized as activities between nodes). Concretely, DyREP proposes a two-time scale deep temporal point process model, one of the stochastic processes that can

be equivalently represented as a counting process, $N(t)$, that captures the interleaved dynamics of the observed processes. The conditional intensity function $\lambda_{(u,v),k}(t)$ that is used to model the occurrence of event (u, v, t, k) , where $k \in 0, 1$, between nodes u and node v at time t can be written as:

$$\lambda_{(u,v),k}(t) = \psi_k \left(\log \left(1 + \exp \left(\frac{\omega_k \cdot [\mathbf{h}_u(t^-) \parallel \mathbf{h}_v(t^-)]}{\psi_k} \right) \right) \right) \quad (\text{Eq. 48})$$

Where ω_k serves as the model parameter that learns time-scale specific compatibility and ψ_k is a dynamic parameter that corresponds to the rate of events arising from a corresponding process.

This intensity function $\lambda_{(u,v),k}$ is then associated with long-term associations $A(t^-)$, and attention $S(t^-)$ to obtain the temporal point process parameters update via:

$$\mathbf{S}_{uv}(t) = \text{UPDATER}(\mathbf{A}(t^-), \mathbf{S}(t^-), \lambda_k(t))$$

Where *UPDATER* is implemented in Algorithm 1 (Trivedi, Farajtabar, Biswal, & Zha, 2019).

DyREP further uses a Temporal Point Process-based Attention Mechanism that uses temporal information to compute the attention coefficient for a structural edge between nodes. These coefficients are then used to compute the aggregate quantity \mathbf{h}_{struct} required for embedding. With this setup, for any event at a time t , the update of a node u can be written as

$$\mathbf{h}_u(t) = \sigma(\mathbf{W}_{struct} \mathbf{h}_{struct}^v(t^-) + \mathbf{W}_{rec} \mathbf{h}_v(t^-) + \mathbf{W}_t(t - t_u^-)) \quad (\text{Eq. 49})$$

Where $\mathbf{W}_{struct}, \mathbf{W}_{rec} \in \mathbb{R}^{d \times d}$ and $\mathbf{W}_t \in \mathbb{R}^d$ are learnable parameters.

Causal Anonymous Walk Network (CAWN) (Wang Y. , Chang, Liu, Leskovec, & Li, 2021) is a mixer of RRN- and Self-Attention Mechanism-based methods that propose to represent network dynamics by extracting temporal network motifs using temporal random walks with probability proportional to $\exp(\alpha(t - t_0))$, where t, t^- are the timestamps of this link and the link previously sampled respectively. α is a non-negative hyperparameter for time preference. Given a link u_0, v_0 and a time t_0 , Causal Anonymous Walk (CAWs) collect m -step walks starting from both u_0 and v_0 recording into S_u and S_v . CAWs replace node identities with the hitting counts of the nodes based on a set of sampled walks to establish the correlation between motifs S_u and S_v as $g(w, S_u)$ and $g(w, S_v)$. Each walk is anonymized as

$$\widehat{W} = ((I_{CAW}(w_0), t_0), \dots, (I_{CAW}(w_m), t_m)) \quad (\text{Eq. 50})$$

Where $I_{CAW}(w) \triangleq \{g(w, S_u), g(w, S_v)\}$

Then, CAWN neural networks can be conveniently leveraged to extract their structural and fine-grained temporal information by encoding CAWs. the extracted motifs are fed into RNNs to encode each walk jointly with a relatively learnable time encoder as a representation

$$enc(\widehat{W}) = RNN(\{f_1(I_{CAW}(w_i)) \parallel f_2(t_{i-1} - t_i)\}_{i=0,1,\dots,m}) \text{ where } t_{-1} = t_0 \quad (\text{Eq. 51})$$

Where $f_1(I_{CAW}(w_i)) = MLP(g(w, S_u)) + MLP(g(w, S_v))$.

After encoding each walk in $S_u \cup S_v$, $enc(\widehat{W})$, CAWN uses a self-attention mechanism to aggregate the representations of multi-walks into a single vector and add a feed-forward network for final link prediction, written as

$$h_{uv} = FFN\left(\frac{1}{2M} \sum_{i=0}^{2M} softmax\left(enc(\widehat{W}_i)^T \mathbf{Q}_1 enc(\widehat{W}_j)_{1 \leq i \leq n}\right)\right) enc(\widehat{W}_i) \mathbf{Q}_2 \quad (\text{Eq. 52})$$

Where $\mathbf{Q}_1, \mathbf{Q}_2 \in \mathbb{R}^{d \times d}$ are two learnable parameter matrices.

Transformer-based Dynamic Graph Modelling via Contrastive Learning (TCL) (Wang L. , et al., 2018) TCL approach employs a rule-based graph traversal algorithm, specifically a breadth-first search (BFS) algorithm, to extract the subgraph and obtain the node embedding matrix \mathbf{E}_u^{node} . This matrix is then enriched with information about graph depth encoded by positional embeddings, i.e. \mathbf{E}_u^{depth} , and relative time encoded \mathbf{E}_u^{time} by a simple linear projection. The resulting input embedding matrices are formulated as

$$\widehat{\mathbf{E}}_u = \mathbf{E}_u^{node} + \mathbf{E}_u^{depth} + \mathbf{E}_u^{time} \quad (\text{Eq. 55})$$

TCL subsequently utilizes a transformer to independently extract representations of the temporal neighborhoods associated with the two interacting nodes

$$\mathbf{h}_u^b = Transformer(\widehat{\mathbf{E}}_u); \mathbf{h}_v^b = Transformer(\widehat{\mathbf{E}}_v) \quad (\text{Eq. 56})$$

To model the semantic interdependencies between these nodes, a co-attentional transformer is introduced

$$\mathbf{h}_u^b = \text{Transformer}_{\text{cross}}(\tilde{\mathbf{h}}_u); \mathbf{h}_v^b = \text{Transformer}_{\text{cross}}(\tilde{\mathbf{h}}_v) \quad (\text{Eq. 57})$$

Here the $\text{Transformer}_{\text{cross}}(\cdot)$ is transformer encoding operator leverages cross-attention mechanisms, where the source node embedding serves as the query, and the destination node embedding serves as both the key and value. Finally, to enhance model performance, contrastive learning is employed to maximize the mutual information between the predicted representations of future interacting nodes.

We can synthesize the previously discussed methods into a four-column, five-row table visualized in **Figure 2.7.1**. The columns represent how each method learns the graph structure by representing nodes. Conversely, the rows depict how each method aggregates and accumulates information across the network as it evolves. Notably, while graph neural networks encompass a majority of the methods, attention mechanisms are the most prevalent technique employed.

2.7. Contrastive Learning

Contrastive learning describes a class of methods for learning representations by contrasting pairs of related data examples against pairs of unrelated data examples. This approach naturally fits graph-structured data, as relations are given by the edges in the graph. We can cast this problem in the context of energy-based learning (LeCun, Chopra, Hadsell, Ranzato, & Huang, 2006), where we associate a scalar energy $E(f_\theta(\mathbf{x}), f_\theta(\mathbf{y}))$ for pairs of data points $(\mathbf{x}, \mathbf{y}) \in D \times D$ from a dataset D , where $f_\theta(\cdot)$ is an encoder function that maps an observed data point \mathbf{x} to its hidden representation $\mathbf{h}_\mathbf{x}$. We will use Neural Networks (NNs) for $f_\theta(\cdot)$ in practice. Training is carried out by optimizing an objective that encourages low energies for positive (related) pairs and higher energies for negative (unrelated) pairs. Variants of this approach include noise contrastive estimation (NCE) (Gutmann & Hyvärinen, 2010; Mnih & Teh, 2012), negative sampling (Mikolov, Sutskever, Chen, Corrado, & Dean, 2013), and deep metric learning (Chopra, Hadsell, & LeCun, 2005; Hadsell, Chopra, & LeCun, 2006). In NCE and negative sampling, the objective is a binary cross-entropy loss, which specifically for negative sampling takes the following form:

$$\mathcal{L} = -\left(\frac{1}{|\mathcal{T}|}\right) \sum_{(\mathbf{x}, \mathbf{y}, c)} c \log(l(s(\mathbf{x}, \mathbf{y}))) + (1 - c) \log(1 - l(s(\mathbf{x}, \mathbf{y}))) \quad (\text{Eq. 53})$$

Where we define the score for a pair as $s(\mathbf{x}, \mathbf{y}) = -E(f_{\theta}(\mathbf{x}), f_{\theta}(\mathbf{y}))$. \mathcal{T} is a set that contains all positive pairs and many negative pairs – usually k is a hyperparameter. $l(x) = 1/(1 + e^{-x})$ is the logistic sigmoid function and c is an indicator variable that is 1 for positive pairs and 0 for negative samples. A common technique for obtaining negative samples is by corrupting a positive pair, e.g., by replacing one data example in the pair with a random other data example. In this context, the energy function is often chosen to be the negative inner product between hidden representations $E(\mathbf{h}_x, \mathbf{h}_y) = -\mathbf{h}_x^T \mathbf{h}_y$, but other choices are possible.

The related NCE objective differs slightly from and can be used to learn an (asymptotically) unbiased model of the underlying data distribution — see (Dyer, 2014) for details. We will make use of contrastive learning for unsupervised representation learning on graphs and link prediction in CHAPTER VI using an objective based on.

TGL Methods				
	GNN based	Sequence based	Random walk based	None
Memory based	JODIE		CAWN	EdgeBank**
Self-attn based	TGAT	DyGFormer		
	TCL			
	GAT*			
Self-attn & Memory base	DyREP			
	TGN			
MLP		GraphMixer		
None	GraphSAGE*			
	GCN*			

(*) Static graph method; (**) EdgeBank is purely memorized method.

Figure 2.7.1 A taxonomy of temporal graph learning methods

CHAPTER III

METHODOLOGY

3.1. Dataset Insight

The tgbl-flight dataset is one of the TGB datasets (Huang, et al., 2023). It is a large-scale crowd-sourced international flight network from 2019 to 2022 consisting of 67M edges, 18K nodes, and 1385 timestamps. From this dataset, the main objective is predicting the future occurrence of a given flight route on a specific day, which is specified as a link prediction task, discussed in **Section 2.6.1**. Due to large-scale datasets, any graph insight operations are computationally expensive. It is required to perform graph subsampling and discretization for visualization, and graph statistic computation. Therefore, we introduced temporal graph subsampling by random node sampling. By first extracting first-hop neighborhood nodes and corresponding interactions, we further collect those edges projecting into monthly graph snapshots, called discretization. Finally, we can calculate graph statistics and perform visualization to gain insight into the dataset. From Temporal Edge Appearance or TEA (Poursafaei, Huang, Pelrine, & Rabbany, 2022), Fig. 3.1.1 shows the historical seen edges versus new occurrence edges. From the figures, it can be observed that despite discretization into the monthly period, the portion of the new occurrence edge is still not sufficiently high. So, the temporal graph learning model shall be able to learn temporal dependency over a long period. To further quantify the pattern of the plot, EdgeBank (Poursafaei, Huang, Pelrine, & Rabbany, 2022) provides a novel graph statistical indicator that can extract insight from the temporal relation as below:

$$\text{re-occurrence} = \frac{|E_{\text{train}} \cap E_{\text{test}}|}{|E_{\text{train}}|} \quad (\text{Eq. 54})$$

$$\text{surprise} = \frac{|E_{\text{test}} \setminus E_{\text{train}}|}{|E_{\text{test}}|} \quad (\text{Eq. 55})$$

$$\text{novelty} = \frac{1}{T} \sum_{t=1}^T \frac{|E^t \setminus E_{\text{seen}}^t|}{|E^t|} \quad (\text{Eq. 56})$$

where $E^t = \{(s, d, t_e) \mid t_e = t\}$ and $E_{\text{seen}}^t = \{(s, d, t_e) \mid t_e < t\}$

Where E_{train} and E_{test} is a set of edges that are seen during train and test time, respectively.

Table 2. Graph statistic of TGB dataset over 100 sample nodes

	Re-occurrence	Novelty	Surprise	Average Degree
Sample Node	0.305	0.36	0.244	12.25

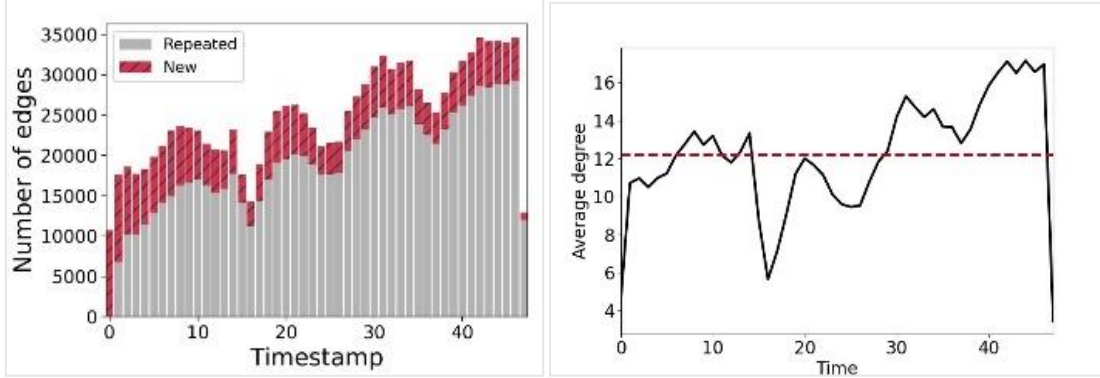


Figure 3.1.1 Graph subsampling visualization of 100 sample nodes and their corresponding edges. The left-hand side is the TEA plot, and the right-hand side is the time series of averaged node degrees. (Note: the last timestamp is not one month fully aggregated, so the chart is relatively low).

The surprise index indicates the proportion of new nodes that have ever been seen during the training phase. This is crucial for selecting the model that can not only the transductive learning but also inductive learning. These graph statistics provide us with valuable guidance in selecting the suitable model architecture for learning from this dataset. The model architecture will be discussed in detail in **Section 2.3**.

The average node degree indicates the sparsity of the graph. This metric is important for graph evaluation methods. From **Table 1**, the average node degree is only 12.25, which indicates that the graph is very sparse. Since most of the nodes are not connected within the graph, simple negative edge prediction, always predicting negative edge or non-interaction, could provide sufficiently high accuracy that inflates the real performance of the model. A proper evaluation metric is crucial for evaluating the real performance of such a graph. **Section 2.6** discusses the evaluation metrics in detail.

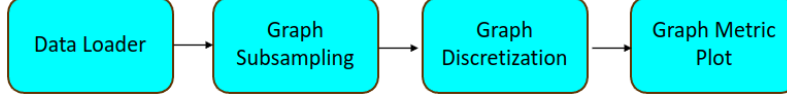


Figure 3.1.1 Illustration of graph subsampling visualization pipeline

3.2. Evaluation Metrics

As we gained insight from the dataset discussed in **Section CHAPTER III**, a more robust evaluation method is required for evaluating the model performance. Using one of negative edge, an edge that has not been seen before, contrasting one positive edge as binary classification is considered the standard evaluation method. However, TGB (Shenyang, et al., 2023) argues that this evaluation method inflates the model performance due to the sparsity of the graph. TGB proposes to treat the evaluation method as a ranking problem, by contrasting one positive edge per multiple negative edges and using Mean Reciprocal Rank (MRR) as an evaluation metric.

$$MRR = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{rank_i} \quad (\text{Eq. 57})$$

The intuition behind this metric is to rank the most relevant items at the top of the list based on probability scores. The higher the MRR, the better the model can find the most relevant items within the lists. In the link prediction task specifically, the model should be able to list the positive edge, an edge that exists in the graph, on the top of the list among multiple negative edges, an edge that does not exist in the graph.

To test the TGL model’s ability to selectively memorize historical edges and the inductive ability of unseen nodes, both historical negative edges, and random negative edges are sampled in link prediction evaluation.

3.3. Temporal Graph Learning Pipeline

One of the significant hurdles in working with real-world temporal graphs is their sheer size. Traditional approaches can become computationally expensive and time-consuming, hindering the efficiency of the training and evaluation process. To address this challenge, we have developed scalable pipelines specifically designed for handling large-scale temporal graphs.

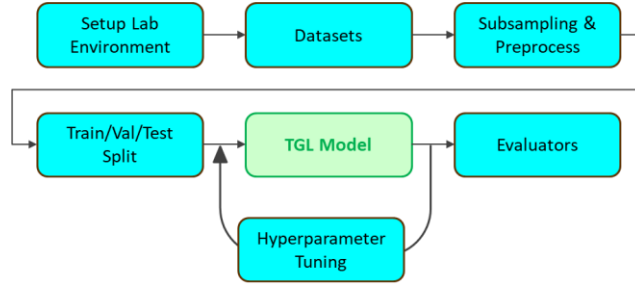


Figure 3.3.1 An illustration of ur temporal graph learning pipeline

These pipelines leverage a key technique called *graph subsampling*. By extracting a smaller, manageable representation of the original graph, we can significantly reduce computational costs and speed up the evaluation iterations of our Temporal Graph Learning (TGL) models. This is particularly beneficial in the early stages of development, where experimenting with different parameter settings and troubleshooting potential bugs in the code is crucial. Spending excessive time and resources training on the full dataset can be wasteful, especially if the model configuration is incorrect.

Therefore, our scalable pipelines offer a compelling solution. They enable us to train and evaluate TGL models on readily available local machines with limited resources, such as those with less than 2GB of GPU memory. This democratizes access to powerful graph learning techniques, making them more accessible to researchers with limited computational budgets. Furthermore, these pipelines are inherently scalable – designed to handle datasets of any size. By strategically partitioning and processing the data, the pipeline can seamlessly adapt to increasingly complex temporal graphs.

The overall structure of this scalable pipeline is depicted in **Figure 3.3.1**. It consists of several critical modules, each playing a vital role in efficient TGL model development for large-scale temporal graphs. We provided the details of those modules as follows:

Setup Lab Environment. Here, we outline the steps to set up a virtual environment for the temporal graph learning (TGL) project. While the process works on both Windows and Linux machines, we recommend using Linux. Typically, the latest software packages become available on Linux before Windows.

- Hardware and Software Requirements:
 - Operating System: Linux (recommended) or Windows
 - Python Version: 3.10 or higher
 - GPU: NVIDIA GPU with CUDA compatibility (version 7.0 or higher)

- Software Libraries:
 - PyTorch (version 2.0 or higher)
 - PyTorch Geometric (version 2.5 or higher)

Subsampling. While there are many methods feasible for sampling the graph. We sampled the nodes from the whole graph. The corresponding destination nodes are also extracted by the edges that connect those nodes. For metrics evaluating purposes, we also need to randomly select the destination node serving as the negative edges, the edges that do not exist within the temporal graph. To test the transductive as well as inductive ability of the model (discussed in **Section 2.4**), the negative edges are sampled 50% from “historical negative sampling” and 50% from “inductive negative sampling” discussed in **Section 2.6.3**. This pre-process subgraph and obtained negative sampled edges are serialized and stored in *.pkl* format for later fast retrieval. Meanwhile, deterministically sampling by region, for instance, continence or iso-region sampling, is also supported by our pipeline.

Train/validation/test split. (Extracting Relevant Edges) We focus on the temporal edges between the sampled nodes and their immediate neighbors (first-hop neighborhood). These edges represent the core interactions within the subsample and are crucial for tasks like link prediction or understanding the evolution of local network structures. (Chronological Split) To ensure the training process encounters a realistic distribution of temporal information, we split the extracted edges based on their chronological order. Here is our approach:

- 70% Train Set: This set contains the majority of the edges and is used to train the model on the temporal relationships within the subsample.
- 15% Validation Set: This set is used to evaluate the model's performance during training and fine-tune hyperparameters before deploying the final model.
- 15% Test Set: This unseen set is used for the final evaluation of the model's generalizability on unseen temporal data.

TGL Model. This module serves as the placeholder for the specific TGL model that is ready to plug in. The details of our proposed TGN-ST model architecture will be discussed in **Section 3.5- 3.7**.

Hyperparameter tuning. A crucial step in developing a TGL model is finding the optimal hyperparameters. To streamline this process, we leverage the Weights & Biases (*wandb*) framework. *wandb* acts as a central hub, allowing us to effortlessly track and visualize all

aspects of our experiments throughout the training phase. This includes logging hyperparameter settings for each run, metric results, and even training gradients for deeper analysis. Furthermore, *wandb* facilitates efficient hyperparameter sweeps, where we can automatically try hundreds of different configurations to find the best-performing hyperparameter settings. This eliminates the risk of losing track of settings and experimental outcomes, a common challenge when conducting numerous experiments. Ultimately, *wandb*'s comprehensive tracking capabilities empower us to significantly accelerate the hyperparameter tuning process and achieve optimal performance for our TGL model.

3.4. Temporal Graph Model Selection Strategy

After setting up the pipeline, we are ready to experiment with various TGL models. The recent and state-of-the-art models are selected including JODIE, TCL, EdgeBank, TGN, TGAT, DyGFomer, GraphMixer, CAWN, and DyREP. The details of each architecture and method are discussed details in **Section 2.6.4**.

Implementation details. Our experiment is conducted on an Ubuntu 22.04 machine equipped with Intel Core i7 14700k @3.4GHz with 20 cores (8 P-cores + 12 E-cores). The GPU device is NVIDIA RTX 4080 Super (320 tensor cores) with 16GB memory.

Model configurations. All models are optimized by Adam Optimizer (Kingma & Ba, 2015) and use supervised binary cross-entropy loss with logit as the objective function. We train the models for 100 epochs with the use of the early stopping strategy with a patience of 5. We select the model that achieves the best performance on the validation set for testing. We set the learning rate and batch size to 0.0001 and 200 for all the methods. We run the methods three times with seeds from 1 to 3 and report the average performance to eliminate deviations. Each model's hyperparameter settings are detailed below.

- **JODIE (Kumar, Zhang, & Leskovec, 2019):**
 - Dimension of time encoding: 100
 - Dimension of node memory: 100
 - Dimension of output representation: 100
 - Memory updater: vanilla recurrent neural network
- **DyRep (Trivedi, Farajtabar, Biswal, & Zha, 2019):**
 - Dimension of time encoding: 100
 - Dimension of node memory: 100

- Dimension of output representation: 100
- Number of graph attention heads: 2
- Number of graph convolution layers: 1
- Memory updater: vanilla recurrent neural network
- **TGAT (Xu, Ruan, Korpeoglu, Kumar, & Achan, 2020):**
 - Dimension of time encoding: 100
 - Dimension of output representation: 100
 - Number of graph attention heads: 2
 - Number of graph convolution layers: 2
- **TGN (Rossi, et al., 2021):**
 - Dimension of time encoding: 100
 - Dimension of node memory: 100
 - Dimension of output representation: 100
 - Number of graph attention heads: 2
 - Number of graph convolution layers: 1
 - Memory updater: gated recurrent unit
- **CAWN (Wang Y. , Chang, Liu, Leskovec, & Li, 2021):**
 - Dimension of time encoding: 100
 - Dimension of position encoding: 100
 - Dimension of output representation: 100
 - Number of attention heads for encoding walks: 8
 - Length of each walk (including the target node): 2
 - Time scaling factor α : $1e-6$
- **TCL (Wang L. , et al., 2018):**
 - Dimension of time encoding: 100
 - Dimension of depth encoding: 100
 - Dimension of output representation: 100
 - Number of attention heads: 2
 - Number of Transformer layers: 2
- **GraphMixer (Cong, et al., 2023):**
 - Dimension of time encoding: 100
 - Dimension of output representation: 100
 - Number of MLP-Mixer layers: 2

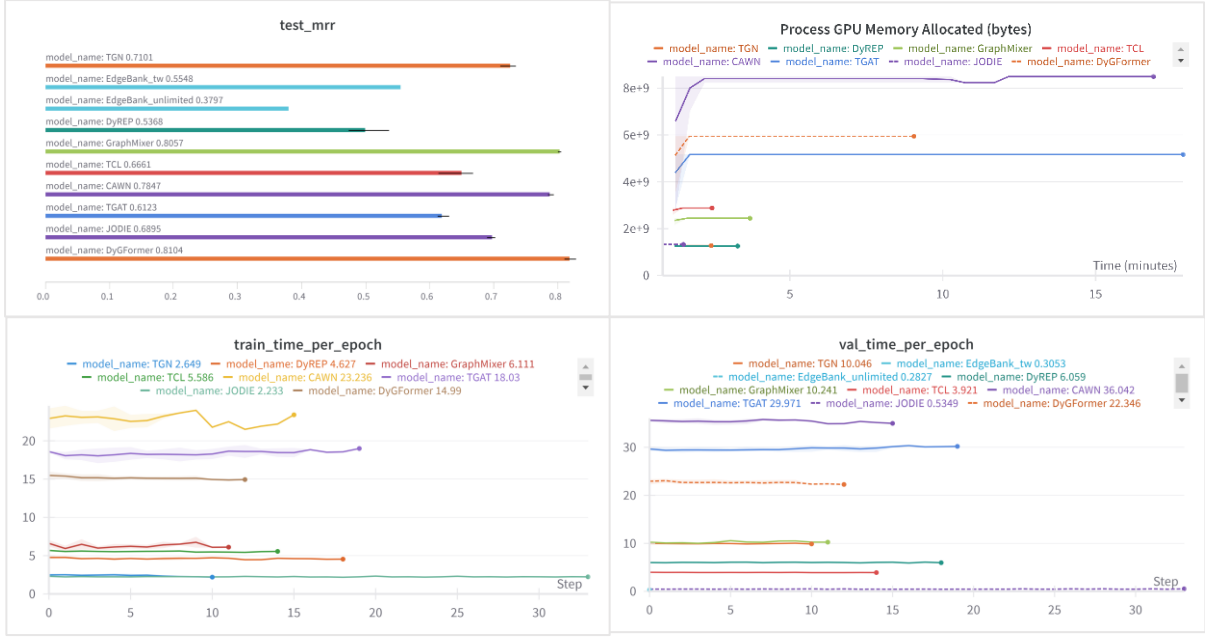


Figure 3.4.1 (top left) average test mrr results of various model benchmarking. (top right) GPU memory usage in bytes. (bottom left) training time per epoch. (bottom right) validation time per epoch.

- Time gap T : 2000
- **DyGFormer (Yu, Sun, Du, & Lv, 2023):**
 - Dimension of time encoding d_T : 100
 - Dimension of neighbor co-occurrence encoding d_C : 50
 - Dimension of aligned encoding d : 50
 - Dimension of output representation d_{out} : 100
 - Number of attention heads I : 2
 - Number of Transformer layers L : 2

Experiment Results and Analysis. The result of important experimental metrics is depicted in **Figure 3.4.1**. The top 5 test MRR models are DyGFormer (81.04%), GraphMixer (80.57%), CAWN (78.47%), TGN (71.01%), and JODIE (68.95%). We might select the top performance model to scale the model to the full dataset. However, those top 3 methods have shown high computational resources which is multiple times compared to TGN and JODIE models. Specifically, DyGFormer is reported to be the top performance among all the models in our experiment. The training time and validation time of DyGFormer are 5.7 times and 2.2 times respectively, compared to the TGN model. We shall also be concerned about the GPU memory usage as well since the high memory consumption model might not fit into our experimental environment that consists of only 16GB. However, the patching technique of DyGFormer

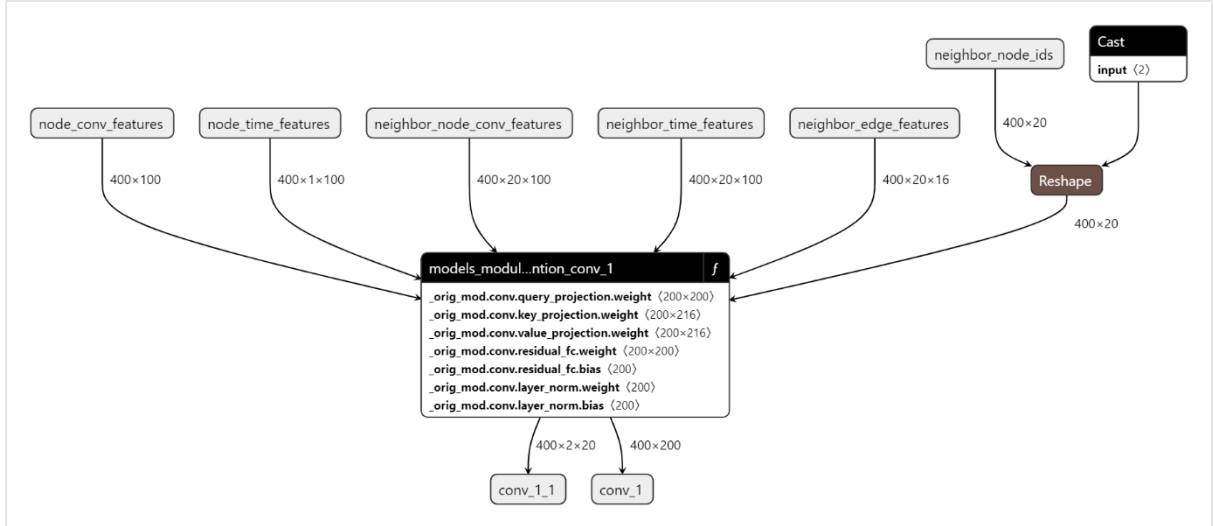


Figure 3.4.2 An illustration of attention-based message passing aggregator.

allows the model to tackle the out-of-memory (OOM) problem. While CAWN is the second-best performance in terms of test MRR, CAWN is 8.8 and 3.6 times slower than TGN training time and validation time, respectively. CAWN also consumes much more GPU memory than TGN by 6.6 times. It is worth noting that the inefficiency of CAWN comes from their inefficient construction of neighbor structural features. The third performance is GraphMixer whose training time and GPU memory consumption is 2.3 times slower and 1.7 times bigger than the TGN model, respectively.

Since we need to scale the model to the large-scale dataset, we are not only concerned about the model performance but also concerned about the efficiency of the model as well. As a result, the TGN model architecture is selected for further deep dive and optimization which will be discussed in the next section.

3.5. TGN Model Architecture Insight

Based on the insights from the dataset, as discussed in section 2.1, we can select a suitable model architecture for the specified dataset in this study. As a result, we select the TGN framework (Rossi, et al., 2021) as our base model architecture. In the present study, we are interested in studying three core modules:

Memory Updater. Upon each node interaction, the memory module encodes the relative time using trainable learning parameters before updating the state of involved nodes into the compressed format using recurrent neural network (RNN) (ELMAN, 1990) such as gate recurrent unit (GRU) (Chung, Gulcehre, Cho, & Bengio, 2014) or Long Short-Term Memory (LSTM) (Hochreiter & Schmidhuber, 1997). The original paper uses a learnable time encoder

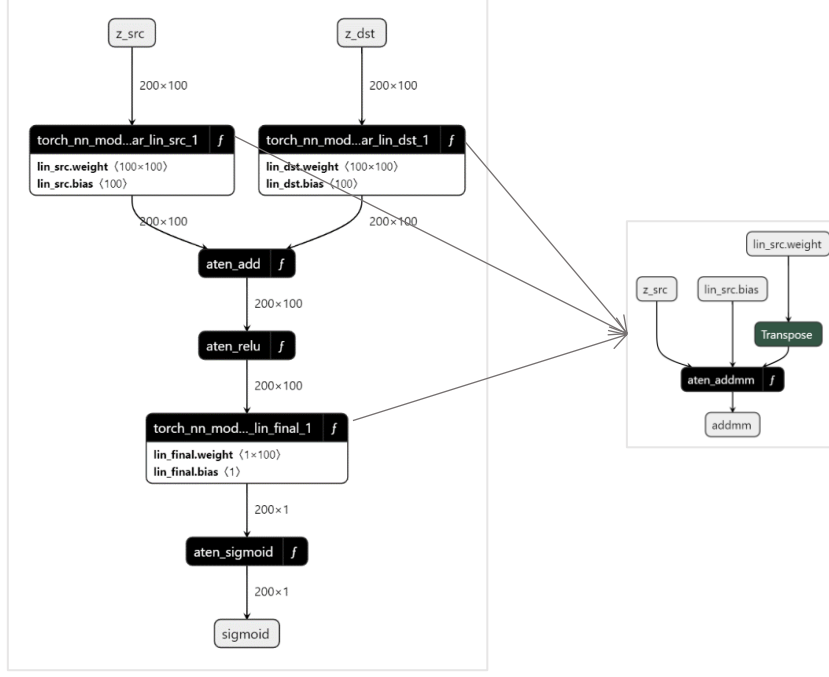


Figure 3.5.1 An illustration of link predictor module of TGN network.

(Kazemi, et al., 2019; Da Xu, 2020) which is proven to be unstable to train (Cong, et al., 2023). As a result, we propose a temporal graph networks using static-time encoder function $\cos(t\omega)$ TGN-ST which utilizes the feature $\omega = \{\alpha^{-(i-1)/\beta}\}_{i=1}^d$ to encode each timestamp into a d-dimensional vector.

Embedding. This module is used to generate the temporal embedding of the node at any time t . It uses a multi-head self-attention mechanism (Justin, S., F., Oriol, & E., 2017; Vaswani, et al., 2017) to attend to each k-hop neighborhood. This is crucial as in our example dataset, the transit hub of an airport shall know which flight causes another flight to depart from this hub airport. See **Figure 3.4.2** for the overview of the module. (Details depicted at **Figure 3.8.5**)

Decoder/Link Predictor. **Figure 3.5.1** use a two-layer perceptron with a non-linearity activation function to project the source and destination node hidden state into proper dimension before feeding into a sigmoid function to calculate edge probability for the prediction task.

3.6. Graph Capture and Compilation

As we need to train the model on the large-scale dataset, code optimization is required to speed up the wall clock time and save computational resources. Therefore, we implement *torch.compile* (Ansel, et al., 2024) wrapped around this embedding and decoder module to

knowledge to write compute-unified device architecture CUDA code on NVIDIA GPU. This gave rise to the popularity of OpenAI Triton (Tillet, Kung, & Cox, 2019), an open-source Python-like programming language that enables researchers with no CUDA experience to write highly efficient GPU code—most of the time on par with what an expert would be able to produce. However, it still requires a few folds of code to generate the same function. Thanks to the PyTorch extension of TorchInductor that allows the code to compile to Triton just with a simple *torch.compile* API call.

3.7. TGN-ST forward pass algorithm

From the experiment of model selection from **Section 3.4**, we observed that the validation time is slower than the training time per epoch basic. This led us to a further investigation of the high computational operation. We can track the runtime of each operation of the forward pass to obtain the time spent on any suspect operations. Before performing this tracking, it is important to understand the **Algorithm 1** that will be described as follows.

1. **Initialization:** Three variables are initialized, starting with the fresh memory of each node, the embedding and link predictor’s parameters are newly initialized (step 3).
2. **Batch Loop:** The algorithm iterates over batches of temporal data (step 4).
3. **Negative Sampling:** Within the loop, a "*negative dst node*" (destination node) is queried by loading the negative sampled edge that was pre-processed by the pipeline (step 5).
4. **Neighbor Query:** The neighbors of a specific node are queried. This involves retrieving information about the nodes connected to the current node (step 5).
5. **Node Loop:** It is iterated over the negative neighbors of each node in the batches.
6. **Memory Updater with Time Encoder:** An "implicit trainable time encoder" is used to incorporate with the "*memory_updater*" module to update the node representation (step 7).
7. **Node Embedding:** The nodes and their neighbors involved in the current interaction are extracted and used for updating the current interaction using the "*embedding*" module (step 9).
8. **Predictor:** The resulting embedded source and destination nodes are concatenated and projected by the "*link_predictor*" module to obtain the interaction occurrence probability (step 10).
9. **MRR Calculation:** After obtaining the predicted state either present "True/1" or

“False/0”, the algorithm can now rank the positive edge contrasting the negative edge to calculate the MRR metrics (step 11).

While the Mean Reciprocal Rank (MRR) metric is primarily used to evaluate the model's performance on the validation or test set, the predicted outcomes themselves can also play a role in the training process. Specifically, these predicted outcomes can be fed into a binary cross-entropy (BCE) with a logit loss function during the training phase. This loss function then guides the model's parameter updates, helping it learn to generate more accurate predictions.

Algorithm 1 describes the model's forward pass. This process involves two loops: (1) An outer loop that iterates over each batch of data points (training examples) in the training dataset. An inner loop that iterates over all the negative edges generated for each node within a particular batch to calculate the MRR metrics. This implementation is not efficient since it must loop through each positive node within the batches and then query the negative neighbor batches of such node, in line 7 of algorithm 1. The resulting queried negative neighbor batch is then combined with each corresponding positive node passing to the memory updater, embedding, and link predictor in lines 8 – 10 of Algorithm 1.

Algorithm 1 highlights a potential bottleneck in the standard TGN forward pass (lines 7-10). This bottleneck arises from the loop that iterates through each negative neighbor query, memory updater, embedding, and link predictor for every node. To address this and reduce computational cost, we propose the TGN-ST forward pass algorithm in Algorithm 2. TGN-ST leverages the capabilities of modern GPUs with parallel processing architectures. Unlike the standard approach, TGN-ST avoids the need for individual loops for each node's negative neighbors (lines 6-9 in Algorithm 2). Instead, it exploits the GPU's ability to perform large matrix operations simultaneously. In line 6 of **Algorithm 2**, TGN-ST queries all the negative neighbor batches for all nodes within the batch simultaneously. This eliminates the need for separate loops and significantly improves computational efficiency. Following the simultaneous negative neighbor query, TGN-ST combines these negative neighbor batches with the positive nodes within the batch (line 6). This combined data is then passed through the memory updater, embedding, and link predictor modules simultaneously, further maximizing the utilization of the GPU's parallel processing power. In essence, TGN-ST restructures the forward pass to take advantage of modern hardware capabilities, resulting in a more efficient and faster training process.

Algorithm 1. Temporal Graph Network (TGN) forward pass algorithm

Algorithms 1: TGN Forward Pass

```

1  Input:  $X \in \mathbb{R}^{2 \times N \times D}$ ;  $NS \in \mathbb{R}^{N \times S}$   $\triangleright S$  size of neighbors;  $D$  input size
2  Output:  $Y \in \mathbb{R}^{N \times 1}$ ;  $MRR \in \mathbb{R}$   $\triangleright N$  total number of edges
3  Initialization: mem_updater, embedding, link_predictor
4  For each batch  $X_i \in X$  do
5       $NS_i \leftarrow$  Query negative dst node corresponds to each  $x_i \in X_i$   $\triangleright X_i = [X_{src_i}, X_{dst_i}]$ 
6      For each  $x_i \in X_i$  contrasts  $ns_i \in NS_i$  do
7           $NBR_i \leftarrow$  Query the neighbor of  $\{x_i, ns_i\}$ 
8           $Z_i \leftarrow mem\_updater(\{x_i, NBR_i\})$   $\triangleright$  implicit learnable time encoder
9           $Z_i \leftarrow embedding(Z_i)$ 
10          $Y_i \leftarrow link\_predictor(Z_{src_i} \in Z_i, Z_{dst_i} \in Z_i)$   $\triangleright Z_{src_i}$  embedding of source nodes
11          $MRR_i \leftarrow$  Compute the metrics MRR
12     End For
13 End For
14 Return  $Y$ ,  $MRR$ 

```

Algorithm 2. Temporal Graph Network with Static Time Encoder (TGN-ST) forward pass algorithm

Algorithms 2: TGN-ST Forward Pass

```

1  Input:  $X \in \mathbb{R}^{2 \times N \times D}$ ;  $NS \in \mathbb{R}^{N \times S}$   $\triangleright S$  size of neighbors;  $D$  input size
2  Output:  $Y \in \mathbb{R}^{N \times 1}$ ;  $MRR \in \mathbb{R}$   $\triangleright N$  total number of edges
3  Initialization: mem_updaterst, torch.compile({embedding, link_predictor})
4  For each batch  $X_i \in X$  do
5       $NS_i \leftarrow$  Query negative dst node corresponds to  $X_i$   $\triangleright X_i = [X_{src_i}, X_{dst_i}]$ 
6       $NBR_i \leftarrow$  Query the neighbor of  $\{X_i, NS_i\}$ 
7       $Z_i \leftarrow mem\_updater_{st}(X_i, NBR_i)$   $\triangleright$  implicit static time encoder
8       $Z_i \leftarrow embedding(Z_i)$ 
9       $Y_i \leftarrow link\_predictor([Z_{src_i} \in Z_i, Z_{dst_i} \in Z_i])$   $\triangleright Z_{src_i}$  embedding of source nodes
10     For each  $x_i \in X_i$  contrasts  $ns_i \in NS_i$  do
11          $MRR_i \leftarrow$  Compute the metrics MRR
12     End For
13 End For
14 Return  $Y$ ,  $MRR$ 

```



Figure 3.7.1 An illustration of Hyperparameter sweeping filtered on optimizer by sgd only. (top left) visualize the best validation mrr of each run. (top right) the correlation of the best mrr with each hyperparameter. (bottom) all hyperparameter filtered by sgd optimizer, i.e. top transparent rectangular on optimizer column.

3.8. Experimental Settings and Hyperparameter Tuning

To evaluate the performance and efficiency benchmarking between our proposed TGN-ST versus vanilla TGN model, we conducted the experimentation on the same environment as discussed in **Section 3.4**, i.e. on Ubuntu 22.04 (LTS) machine with an RTX 4080 Super GPU with 16GB of memory. For the vanilla TGN experiment, all configurations are adopted from the TGB baseline (Huang, et al., 2023). For our configuration of the TGN-ST model, we run hyperparameter sweeping using *wandb*¹ as discussed in detail below.

Selecting hyperparameters is a tedious task. No single hyperparameter settings that suit all datasets and the machine learning models. Careful parameter selection can help the model

¹ <https://wandb.ai/>

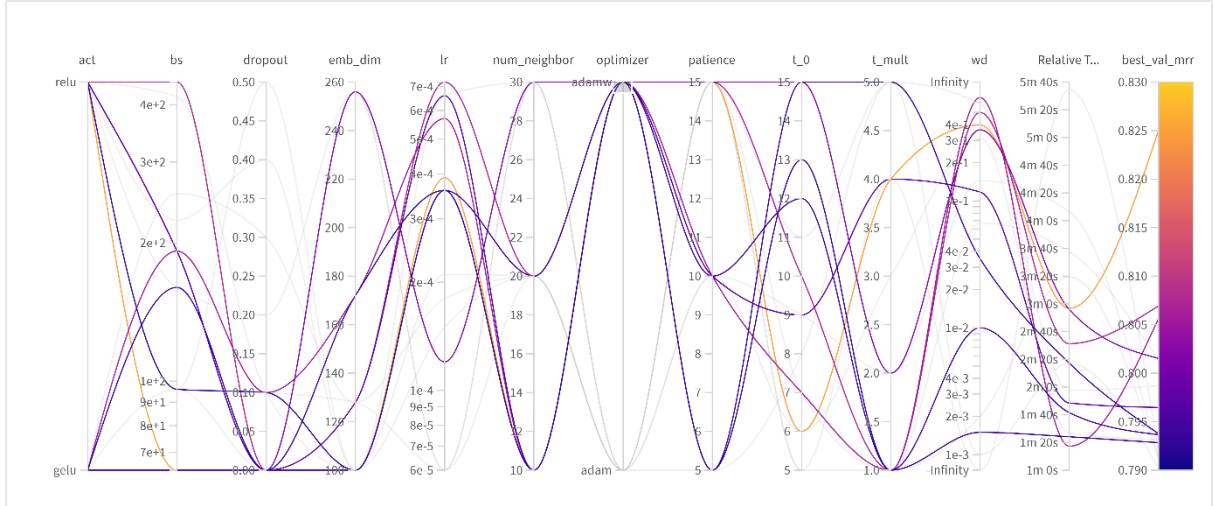


Figure 3.8.2 Hyperparameter settings of best_val_mrr greater than 79% filtered by AdamW optimizer.

to converge to a good optimal solution with reasonable computational resources. Therefore, we automate hyperparameter sweeping using *wandb* package, a platform designed for experiment tracking and interactive visualization. Specifically, we run 200 hyperparameter random search experiments on 100 sampling nodes based on FFN’s activation function (act), batch size (bs), dropout rate and patience, learning rate (lr), number of iteration of the first restart of learning rate t_0 and the multiplier after first restart t_{mult} , optimizer, number of neighbors (num_neighbor), weight decay rate (wb). All hyperparameter settings are listed in **Table 3**, by random search methods subject to validation mean reciprocal rank (MRR) maximization.

The resulting of 200 runs of hyperparameter sweep filtered by top performance hyperparameter setting is obtained and depicted in **Figure 3.7.1**. Each line in the graph corresponds to a set of hyperparameter settings and its corresponding loss results in the optimization over 50 epochs. We can filter and obtain the impact of each hypermeter on the model performance MRR as illustrated in **Figure 3.7.1**. It is mainly observed that the majority of the poor MRR is coming from the SGD optimizer. Except for one run, the performance of TGN-ST MRR is below 70%. We can conclude that running our model using SGD should be avoided. It is not only achieving poor performance but also it is not efficient to train as it takes longer time to converge. Another important observation is that all the top performance is either coming from the ReLU or GELU activation. So, the LeakyReLU activation hurts the performance and should not be used for scaling up the model.

Tuning hyperparameters is crucial for achieving optimal performance and efficiency in TGL models. Large batch size and low patience can hurt the performance while the small batch

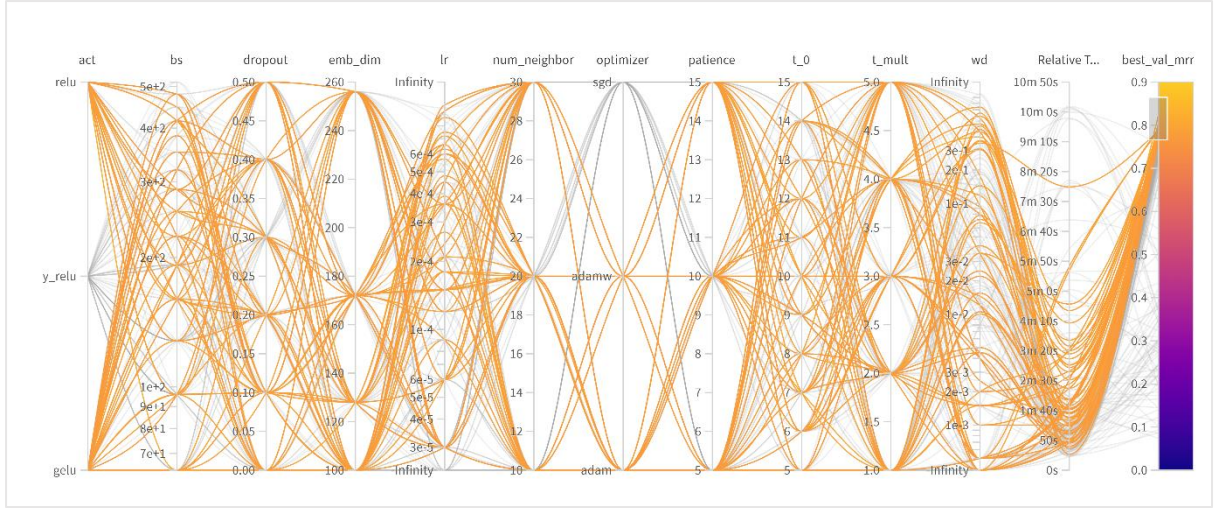


Figure 3.8.3 Hyperparameter sweeping filtered by top performance (shown in square transparent rectangular on best_val_mrr column). Notice that y_relu stands for leaky_relu.

size and high patience lead to slow training and inference. A trade-off between these values shall be selected to achieve optimal performance and efficiency. A large dropout rate can lead to higher generalization but may result in losing too much information during the training. Meanwhile, the high embedding dimension and large number of neighbors could capture the information but would sacrifice the memory storage. As a result, the trade-off between efficiency and performance shall be selected. The bottom row of the hyperparameter setting is selected to scale our model to large datasets.

We narrowed down the hyperparameter settings by focusing on those with the highest validation MRR (greater than 79%). This left us with only 13 settings out of the original 200. Interestingly, AdamW with cosine annealing with warm restart (see **Figure 3.8.4** as an example for the scheduler and its corresponding loss function over the training epoch) outperformed the standard Adam optimizer (without weight decay) in this group. It is worthwhile to note that since AdamW already includes weight decay as a regularization technique, a high dropout rate might not be necessary as they achieve similar effects.

The setting with the best validation MRR also used the smallest batch size (64) and the longest early stopping patience (15 epochs). While this approach led to good validation performance, it resulted in a much slower training time (around 3 minutes compared to other top settings). The learning rate for these top settings seems to be concentrated around a range of $3.5e-4$ to $7e-4$, suggesting this is a good starting point for further optimization.

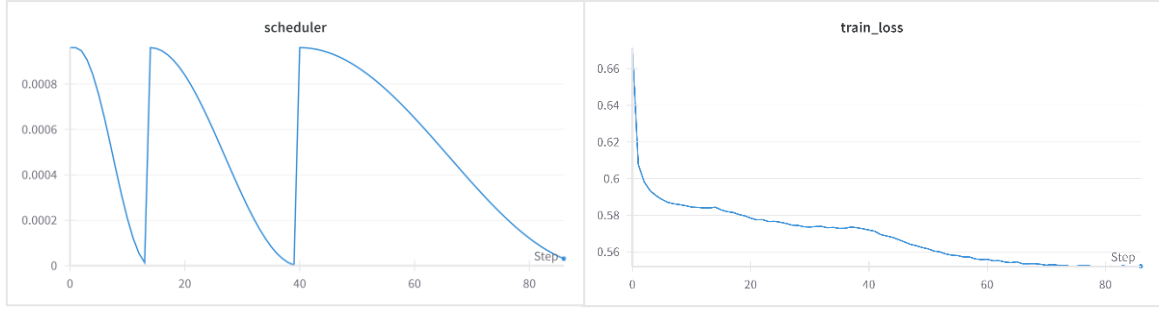


Figure 3.8.4 An example of using cosine annealing with warm restart and its corresponding loss function during model training.

Table 3. Hyperparameter settings of 200 sweeping on TGN-ST models

	Act	bs	dropo ut	emb_d im	lr	num_n eighbo r	optimizer	pati ence	t_0	t_{mult}	wd
distrib ution	rand_sel	q_log_unif orm_value s	int_u niform	rand_sel	q_log_unif orm_value s	rand_sel	rand_sel	rand_sel	int_u niform	int_u niform	q_log_unif orm_value s
range	[relu, leaky_relu, gelu]	[32,512]	[0,0.5]	[100,1 72,256]	[0.00001, 0.0001]	[10,20, 30]	[sgd, adam, adamw]	[5,1 0,15]	[5,15]	[1,5]	[0.0001, 1]
step	-	32	0.1	-	0.00003	-	-	-	1	1	0.0005
final selecti on	ReLU	288	0.1	100	0.00072	10	AdamW	5	14	4	0.01292

From the above hyperparameter settings analysis, we can select the hyperparameter setting that balances those factors such as a sufficiently large batch for faster training without sacrificing much performance, and we can obtain a good optimizer that can be generalized very well in the test set. A hyperparameter setting is selected and listed in the last row of **Table 3**. These hyperparameter settings have shown encouraging performance on random sample nodes. We hypothesized that it would also perform well on the full dataset as well.

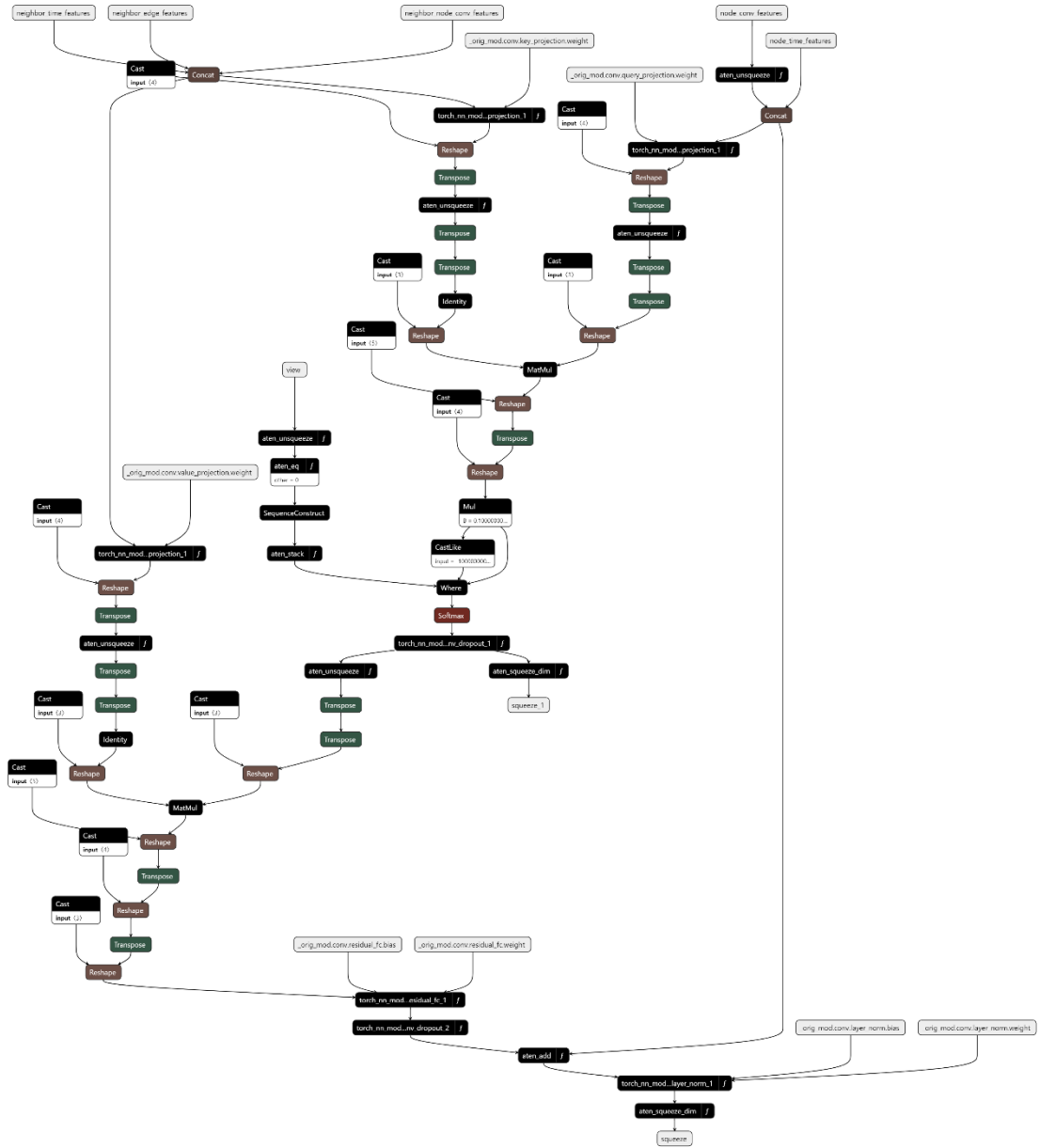


Figure 3.8.5 FX Graph details of TGN embedding modules.

CHAPTER IV

RESULTS AND DISCUSSIONS

This section analyzes the performance and efficiency gains achieved by the TGN-ST model compared to the vanilla TGN model. The generalization capabilities and factors that influence the performance of our proposed TGN-ST framework are also discussed.

We employed hyperparameter sweeping to identify optimal configurations for both models. These settings were then used to run three experiments per model on a subset of 100 sample nodes from the tgbl-flight datasets.

Figure 3.8.1 (e) and (f) depict the resulting training and validation Mean Reciprocal Rank (MRR) scores, respectively. We configured the experiments with an early stopping mechanism that terminates training if there is no further improvement in validation MRR for five consecutive epochs. As evident in the figures, the TGN model suffers from instability and terminates early due to this criterion. In contrast, TGN-ST outperforms the vanilla TGN throughout the training process, demonstrating superior performance in terms of both loss and MRR.

The TGN-ST model surpasses the vanilla TGN model not only in terms of test MRR but also in training and validation efficiency. **Figure 3.8.1** (c) and (d) illustrates the significant reduction in training and validation time per epoch achieved by TGN-ST. The average test MRR across three runs for TGN-ST shows a 7.25% improvement over the TGN model. Furthermore, summarizes the efficiency gains, highlighting a 25% reduction in training time and a remarkable 4.34x reduction in validation time per epoch for TGN-ST.

These positive results on the sub-sampled dataset provided strong justification for applying the chosen model architecture, temporal graph pipeline, forward pass algorithm and hyperparameter settings to the full tgbl-flight dataset. Consequently, the TGN-ST model achieved a state-of-the-art test MRR of 72.49%, surpassing the vanilla TGN by 1.99%. Additionally, TGN-ST demonstrated a 15% improvement in training time per epoch. Most notably, the evaluation process on the full dataset benefited from a significant 5x speedup compared to the vanilla TGN model.

To gain insight into the generalizability of our TGL framework, below are the key observations from the obtained result below **Table 4**:

Table 4. Result summary of 100 sample nodes and full dataset benchmarking between vanilla TGN and TGN-ST model.

		Test MRR (%)	Training Time per Epoch (s)	Validation Time per Epoch (s)
100 Sample Nodes	Vanilla TGN	72.94	2.3743	9.8546
	TGN-ST	80.19	1.7763	2.2687
	Gain	7.25	0.598 (25%)	7.586 (4.34x)
Full Dataset	Vanilla TGN	70.50	2800	9485
	TGN-ST	72.49	2366	1887
	Gain	1.99	434 (15%)	7598 (5.02x)

- **Test MRR:** The Test MRR is higher for both models on the subsample compared to the full dataset. This suggests that the subsample might have captured a slightly "easier" subset of link prediction tasks, leading to inflated performance. However, TGN-ST still demonstrates a consistent gain in Test MRR (around 7% on subsample and 2% on full dataset) compared to the vanilla TGN model. This indicates that TGN-ST has a general advantage in link prediction regardless of the specific data subset.
- **Training and Validation Time:** The significant efficiency gains (reduction in training and validation time) observed for TGN-ST on the subsample hold true even on the full dataset. This suggests that the core algorithmic improvements of TGN-ST, like the static time encoder, graph capture and compilation, and the forward pass algorithm, translate well to larger and more complex datasets.

The reasons that could lead to performance gap between subsample graph and full can be caused by many factors whose some reasons are listed below:

- **Sampling Bias:** There's a chance the subsampling method might not perfectly capture the entire distribution of nodes, edges, and temporal patterns present in the full dataset. This can lead to the model performing well on the specific characteristics of the subsample but struggling to generalize to the broader range of data points in the full graph.

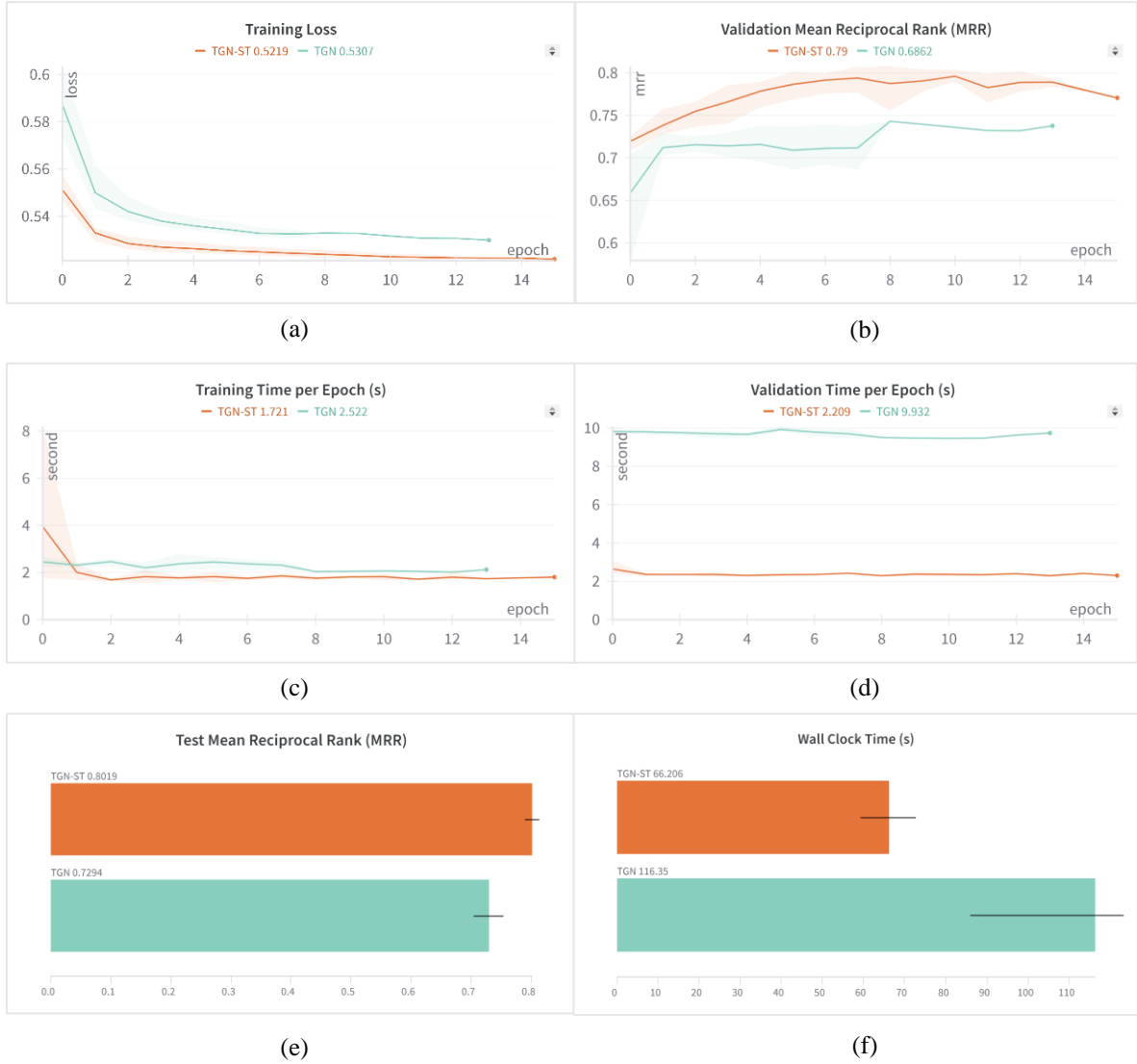


Figure 3.8.1 Illustration of training, validation and test result benchmarking between TGN-ST and vanilla TGN model. (a) Training loss result. (b) validation mean reciprocal rank (MRR). (c) training time per epoch. (d) validation time per epoch. (e) test MRR. (f) wall clock time.

- **Limited Representativeness:** A subsample, by definition, is a smaller representation of the full dataset. Even a well-designed subsampling technique might not fully capture the complexity of the relationships and temporal dynamics within the entire graph. This can lead to inflated performance metrics on the subsample that don't translate directly to the full dataset.
- **Statistical Fluctuations:** Training and evaluating on a smaller subsample introduce statistical noise. Since only a portion of the data is used, the performance metrics (like MRR and training time) might fluctuate more compared to the full dataset. This can cause slight discrepancies in the results even if the model is performing consistently.

In essence, the gap between the subsample and full dataset results highlights the limitations of subsampling but doesn't necessarily negate the overall findings. The consistent gain in Test MRR by TGN-ST and the substantial efficiency improvements across both the subsample and full dataset provide strong evidence for the effectiveness of the proposed framework and TGN-ST model.

CHAPTER V

CONCLUSIONS AND FUTURE DIRECTIONS

5.1. Conclusions

This study has laid the groundwork for significant advancements in temporal graph learning (TGL). We have introduced a scalable and reproducible framework that empowers researchers and practitioners to efficiently develop and evaluate TGL models. This framework incorporates essential tools like graph subsampling and discretization, comprehensive graph visualization techniques, statistical analysis, and hyperparameter tuning, streamlining the entire development process.

Furthermore, we addressed limitations in existing TGN models by proposing the TGN-ST model with a novel static time encoder. This innovation enhances the model's stability during training, leading to superior performance. The TGN-ST model achieves state-of-the-art performance on the benchmark tgbl-flight dataset (72.49% test MRR) while demonstrating significant improvements in training and validation speed (15% and 5x faster, respectively) compared to the vanilla TGN. This efficiency allows for faster and more comprehensive model evaluation.

Our main contributions are summarized twofold:

Scalable Framework: The proposed framework facilitates efficient TGL development by enabling effective training and evaluation on large-scale datasets with minimal computational resources. This broadens the accessibility and applicability of TGL techniques.

Enhanced TGN Model: The TGN-ST model with its static time encoder brings greater stability and robustness to the training process. This translates to a more reliable and generalizable model suitable for real-world applications.

Finally, the comprehensive evaluation process, incorporating hyperparameter tuning and robust performance metrics, ensures the generalizability and practical value of the TGN-ST model. This study paves the way for further advancements in TGL, enabling deeper insights into the ever-evolving nature of real-world temporal graphs.

5.2. Reproducible Research

To ensure the reproducibility of my research, all code, data, and relevant documentation are openly accessible on [GitHub](https://github.com/HorHang/TGN-ST)². This repository provides a comprehensive record of the data preprocessing, visualization, model development, and evaluation processes. By sharing these materials, I aim to enhance transparency, facilitate collaboration, and enable independent verification of my findings. Researchers can access and execute the code to replicate the experiments, potentially leading to further insights and advancements in the field.

5.3. Future Directions

In future work, we aim to scale the temporal graph learning pipeline to multi-GPU or multi-node clusters, leveraging distributed algorithms to further improve computational efficiency and model performance. This advancement will facilitate the deployment of our models in real-world applications that require high-speed processing of vast amounts of temporal data. Another interesting area of research is to explore different subsampling and summarization strategies such as random walk or sample by connected components to preserve structural information as much as possible from the full dataset. Inspired by our literature review and provided a taxonomy of temporal graph learning methods, while most of the TGL model leverages the GNN method, there are more areas to research on random walk and sequence-based TGL models.

² <https://github.com/HorHang/TGN-ST>

REFERENCES

- Akoglu, L., Chandy, R., & Faloutsos, C. (2013). Opinion fraud detection in online reviews by network effects. *AAAI Conference on Artificial Intelligence (AAAI)*.
- Ansel, J., Yang, E., He, H., Gimelshein, N., Jain, A., Voznesensky, M., . . . Will. (2024). PyTorch2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. *ASPLOS*.
- Ba, J. L., Kiros, J. R., & Hinton, G. E. (2016). Layer normalization. *arXiv preprint arXiv:1607.06450*.
- Bahdanau, D., Cho, K., & Bengio, Y. (2015). Neural Machine Translation by Jointly Learning to Align and Translate. *International Conferences on Representation Learnings (ICLR)*.
- Battaglia, P. W., Hamrick, J. B., Bapst, V., Sanchez-Gonzalez, A., Zambaldi, V., Malinowski, M., . . . A. (2018). Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*.
- Bengio, Y., Courville, A., & Vincent, P. (2013). Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, (pp. 35(8):1798–1828).
- Bengio, Y., Ducharme, R., & Vincent, P. (2000). A Neural Probabilistic Language Model. *Advances in Neural Information Processing Systems 13 (NIPS 2000)*.
- Bronstein, M. M., Bruna, J., LeCun, Y., Szlam, A., & Vandergheynst, P. (2017). Geometric deep learning: going beyond euclidean data. *IEEE Signal Processing Magazine*, (pp. vol. 34, no. 4, pp. 18–42).
- Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., . . . Da. (2020). Language Models are Few-Shot Learners. *34th Conference on Neural Information Processing Systems (NeurIPS 2020)*. Vancouver, Canada.
- Bruna, J., Zaremba, W., Szlam, A., & LeCun, Y. (2014). Spectral networks and locally connected networks on graphs. *International Conference on Learning Representations*.

- Chen, Y., Zeng, A., Huzhang, G., Yu, Q., Zhang, K., Yuanpeng, C., . . . Zhou, Z. (2023). Recurrent Temporal Revision Graph Networks. *Advances in Neural Information Processing Systems (NeurIPS)*.
- Cho, K., Merriënboer, B. v., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014). Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. *Empirical Methods in Natural Language Processing (EMNLP)*, (pp. 1724–1734). Doha, Qatar.
- Chopra, S., Hadsell, R., & LeCun, Y. (2005). Learning a similarity metric discriminatively, with application to face verification. *Conference on Computer Vision and Pattern Recognition (CVPR)*.
- Chung, J., Gulcehre, C., Cho, K., & Bengio, Y. (2014). Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. *NIPS*.
- Cong, W., Zhang, S., Kang, J., Yuan, B., Wu, H., Zhou, X., . . . Mahdavi, M. (2023). Do We Really Need Complicated Model Architectures For Temporal Networks? *International Conference on Learning Representations (ICLR)*.
- Defferrard, M., Bresson, X., & Vandergheynst, P. (2016). Convolutional neural networks on graphs with fast localized spectral filtering. *Advances in Neural Information Processing Systems*, (pp. 3844–3852).
- Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., & Fei-Fei, L. (2009). ImageNet: A large-scale hierarchical image database. *IEEE Conference on Computer Vision and Pattern Recognition*.
- Derrow-Pinion, A., She, J., Wong, D., Lange, O., Hester, T., Perez, L., . . . Veličković, P. (2021). ETA Prediction with Graph Neural Networks in Google Maps. *CIKM '21*.
- Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *NAACL-HLT 2019*, (pp. 4171–4186). Minneapolis, Minnesota.
- Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., . . . Houlsby, N. (2021). An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. *International Conference on Learning Representation (ICLR)*.

- Dyer, C. (2014). Notes on noise contrastive estimation and negative sampling. *arXiv preprint arXiv:1410.8251*.
- ELMAN, J. L. (1990). Finding Structure in Time. *Cognitive Science*.
- Georgiev, G. T., Lei, V. I., Burnell, R., Bai, L., Gulati, A., Tanzer, G., . . . P, C. (2024). Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. <https://arxiv.org/pdf/2403.05530>.
- Gimpel, D. H. (2016). Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*.
- Glorot, X., Bordes, A., & Bengio, Y. (2011). Deep sparse rectifier neural networks. *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, (pp. 315–323).
- Gutmann, M., & Hyvärinen, A. (2010). Noise-contrastive estimation: A new estimation principle for unnormalized statistical models. *the Thirteenth International Conference on Artificial Intelligence and Statistics, PMLR*, (pp. 9:297-304).
- Hadsell, R., Chopra, S., & LeCun, Y. (2006). Dimensionality reduction by learning an invariant mapping. *Conference on Computer Vision and Pattern Recognition (CVPR)*.
- Hamilton, W. L. (2020). *Graph Representation Learning*. Synthesis Lectures on Artificial Intelligence and Machine Learning.
- Hamilton, W. L., Ying, R., & Leskovec, J. (2017). Inductive Representation Learning on Large Graphs. *Neural Information Processing Systems (NIPS)*.
- Hamilton, W., Ying, Z., & Leskovec, J. (2017). Inductive representation learning on large graphs. *Advances in Neural Information Processing Systems (NeurIPS)*.
- Hochreiter, S. &–1. (1997). Long Short-Term memory. *Neural computation*, (pp. pp. 9(8):1735–1780).
- Hornik, K., Stinchcombe, M., & White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*, 359 - 366.
- Huang, S., Danovitch, F. P., Fey, M., Hu, W., Rossi, E., Leskovec, J., . . . Rabbany, R. (2023). Temporal Graph Benchmark for Machine Learning on Temporal Graphs. *Advances in Neural Information Processing Systems*.

- James Reed, Z. D. (2022). Torch.fx: practical program capture and transformation for deep learning in python. *Proceedings of Machine Learning and Systems*, (pp. Vol. 4, 638–651).
- Jordan, M. I. (1997). Serial order: A parallel distributed processing approach. *Advances in psychology* (pp. volume 121, pages 471–495). Elsevier.
- Joshi, C. K. (2020). *Transformers are Graph Neural Networks*. Retrieved from NTU Graph Deep Learning Lab: <https://graphdeeplearning.github.io/post/transformers-are-gnns/>
- Justin, G., S., S. S., F., R. P., Oriol, V., & E., D. G. (2017). Neural Message Passing for Quantum Chemistry. *International Conference on Machine Learning*. Sydney, Australia.
- Kaiming He, X. Z. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *IEEE international conference on computer vision*, (pp. 1026–1034).
- Kevin, M. K. (2022). Graph Embeddings. In M. K. Kevin, *Probabilistic Machine Learning* (p. 762). The MIT Press.
- Kingma, D. P., & Ba, J. (2015). Adam: A method for stochastic optimization. *International Conference on Learning Representations*.
- Kipf, T. N., & Welling, M. (2017). Semi-supervised classification with graph convolutional networks. *International Conference on Learning Representations (ICLR)*.
- Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. *Conference on Neural Information Processing Systems*.
- Kumar, S., Zhang, X., & Leskovec, J. (2019). Predicting Dynamic Embedding Trajectory in Temporal Interaction Networks. *ACM SIGKDD*.
- LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*, 521(7553):436.
- LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-Based Learning Applied to Document. *Proc. of the IEEE*.

- LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, (pp. 86(11):2278–2324).
- LeCun, Y., Chopra, S., Hadsell, R., Ranzato, M., & Huang, F. J. (2006). A Tutorial on Energy-Based Learning. *The Courant Institute of Mathematical Sciences, New York University*. MIT Press.
- Loshchilov, I., & Hutter, F. (2017). SGDR: Stochastic Gradient Descent with Warm Restarts. *International Conference on Learning Representations (ICLR)*.
- Loshchilov, I., & Hutter, F. (2019). Decoupled weight decay regularization. *International Conference on Learning Representations (ICLR)*.
- Louppe, G., Cho, K., Becot, C., & Cranmer, K. (2019). QCDaware recursive neural networks for jet physics. *Journal of High Energy Physics*.
- Marco Gori, G. M. (2005). A new model for learning in graph domains. *International Joint Conference on Neural Networks*. IEEE.
- Micali, S., & Zhu, Z. A. (2016). Reconstructing Markov processes from independent and anonymous experiments. *Discrete Applied Mathematics*.
- Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., & Dean, J. (2013). Distributed representations of words and phrases and their compositionality. *Advances in Neural Information Processing Systems (NIPS)*.
- Mitchell, T. M. (1980). The need for biases in learning generalizations. *Department of Computer Science, Laboratory for Computer Science Research, Rutgers Univ. New Jersey*.
- Mnih, A., & Teh, Y. W. (2012). A Fast and Simple Algorithm for Training Neural Probabilistic Language Models. *International Conference on Machine Learning (ICLR)*, (pp. 1751-1758).
- Nair, V., & Hinton, G. E. (2010). Rectified linear units improve restricted boltzmann machines. *International Conference on Machine Learning (ICLR)* (pp. 807–814). Omnipress.
- Pan, Z. W., Chen, F., Long, G., Zhang, C., & Yu, P. S. (2019). A Comprehensive Survey on Graph Neural Networks. *arXiv:1901.00596*.

- Poursafaei, F., Huang, S., Pelrine, K., & Rabbany, R. (2022). Towards Better Evaluation for Dynamic Link Prediction. *Neural Information Processing Systems (NeurIPS)*.
- Rossi, E., Chamberlain, B., Frasca, F., Eynard, D., Monti, F., & Bronstein, M. (2021). Temporal Graph Networks for Deep Learning on Dynamic Graphs. *ICLR*.
- Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagation errors. *Nature*, 323(6088):533.
- Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M., & Monfardini, G. (2009). The graph neural network model. *IEEE Transactions on Neural Networks*.
- Shi, Y., Huang, Z., Feng, S., Zhong, H., Wang, W., & Sun, Y. (2021). Masked Label Prediction: Unified Message Passing Model for Semi-Supervised Classification. *Thirtieth International Joint Conference on Artificial Intelligence (IJCAI-21)*.
- Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Driessche, G. v., . . . Tim. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*, 529, pages484–489.
- Simonyan, K., & Zisserman, A. (2015). Very Deep Convolutional Networks for Large-Scale Image Recognition. *International Conference on Learning Representations (ICLR)*.
- Souza, A., Mesquita, D., Kaski, S., & Garg, V. (2022). Provably expressive temporal graph networks. *NeurIPS*, (pp. 35:32257–32269).
- Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research* 15, 1929-1958.
- Tillet, P., Kung, H. T., & Cox, D. (2019). Triton: an intermediate language and compiler for tiled neural network computations. *Association for Computing Machinery*.
- Tolstikhin, I., Houlsby, N., Kolesnikov, A., Beyer, L., Zhai, X., Unterthiner, T., . . . Dosovitskiy, A. (2021). MLP-Mixer: An all-MLP Architecture for Vision. *Neural Information Processing Systems (NeurIPS)*.
- Trivedi, R., Farajtabar, M., Biswal, P., & Zha, H. (2019). Dyrep: Learning representations over dynamic graphs. *International conference on learning representations*.

- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., . . . Polosukhin, I. (2017). Attention Is All You Need. *31st Conference on Neural Information Processing Systems (NIPS 2017)*. Long Beach, CA, USA.: Neural Information Processing Systems.
- Veličković, P. (2019). *The resurgence of structure in deep neural networks*. Doctor of Philosophy Dissertation.
- Velickovic, P., Cucurull, G., Casanova, A., Romero, A., Lio, P., & Bengio, Y. (2018). Graph attention networks. *International Conference on Learning Representations (ICLR)*.
- Volodymyr Mnih, K. K., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.
- Wang, L., Chang, X., Li, S., Chu, Y., Li, H., Zhang, W., . . . Yang, H. (2018). TCL: Transformer-based Dynamic Graph Modelling via Contrastive Learning. *arXiv preprint arXiv:2105.07944*.
- Wang, Y., Chang, Y.-Y., Liu, Y., Leskovec, J., & Li, P. (2021). Inductive Representation Learning in Temporal Networks via Causal Anonymous Walks. *International Conference on Learning Representations (ICLR)*.
- Xu, D., Ruan, C., Korpeoglu, E., Kumar, S., & Achan, K. (2020). Inductive Representation Learning on Temporal Graphs. *International Conference on Learning Representations*.
- Xu, K., Hu, W., Leskovec, J., & Jegelka, S. (2020). How Powerful are Graph Neural Networks? *International Conference on Learning Representations (ICLR)*.
- Ying, R., He, R., Chen, K., Eksombatchai, P., Hamilton, W. L., & Leskovec, J. (2018). Graph Convolutional Neural Networks for Web-Scale Recommender Systems. *KDD'18*.
- You, J., Ying, R., & Leskovec, J. (2020). Design Space for Graph Neural Networks. *Advances in Neural Information Processing Systems (NeurIPS)*.
- Yu, L., Sun, L., Du, B., & Lv, W. (2023). Towards Better Dynamic Graph Learning: New Architecture and Unified Library. *Conference on Neural Information Processing Systems (NIPS)*.
- Zhang A., L. Z. (2020). *Dive into deep learning*. <https://d2l.ai/>.

Zhu, R., Zhao, K., Yang, H., Lin, W., Zhou, C., Ai, B., . . . Zhou, J. (2019). AliGraph: A Comprehensive Graph Neural Network Platform. *VLDB Endowment*, (pp. 2094 - 2105).

GRADUATE SCHOOL

INSTITUTE OF TECHNOLOGY OF CAMBODIA

Russian Federation Boulevard, P.O. Box 86 Phnom Penh, Cambodia

Email: graduate@itc.edu.kh