Temporal Graph Neural Networks for Epidemiological Forecasting

Rekha Bhupatiraju
Dept. of Electrical Engineering and Computer Science
University of Tennessee, Knoxville
Knoxville, TN, USA
rbhupati@vols.utk.edu

Owen Queen
Dept. of Electrical Engineering and Computer Science
University of Tennessee, Knoxville
Knoxville, TN, USA
oqueen@vols.utk.edu

Sai Thatigotla
Dept. of Electrical Engineering and Computer Science
University of Tennessee, Knoxville
Knoxville, TN, USA
sthatigo@vols.utk.edu

***Abstract*—In this project, we looked into modeling a synthetic dataset made from an agent-based model for simulating opioid addiction in a social network. The dataset contains sequential dynamic graphs as nodes has different classes and edges disappear and are created over time. We also looked into a pre-existing benchmark dataset to help develop and compare static and spatiotemporal GNNs within and between these datasets. We demonstrate that our synthetic dataset could potentionally serve as a benchmark as trivial static and spatiotemporal GNNs are shown ineffetive, but the task is more receptive to more complex dynamic/spatiotemporal GNNs.**

***Index Terms*—Graph Neural Networks, temporal, dynamic, benchmark**

I. Introduction

Graph Neural Networks (GNNs) are an emerging field in machine learning that is growing rapidly with numerous applications from biology, chemistry, sociology, e-commerce, and more. Many sources of data and problems in the real-world can be represented as graph structures. Being able to learn over these structures can present interesting opportunities to develop effective models. And as with many other types of data, graphs do not exist in vacuums either and can change over time. Thus, there is a need to develop GNNs that can learn in this additional dimension. In this regard, we have a synthetic, dynamic, spatiotemporal dataset based on an agent-based model of opioid addiction. Thus, our motivation for this work is to explore GNNs on our dataset as well as consider its effectiveness as a potential benchmark at the same time. In addition, such models could be used as powerful tools for predictions in epidemiology, where many researchers utilize social networks to study populations.

II. Previous Work

One of the first methods developed to deal with GNNs on dynamic graphs was the temporal graph network [1], which adds memory and modifies the message passing function to account for preceding time steps. One state of the art architectures we studied was RetaGNN [2], which handles the problem of sequential recommendation, or the recommendation of a list of items based on current accessed items. This problem comes with unique challenges in how often users arrive and interact and as such needs learning that is transferable without retraining. It considers local subgraphs in pairs and uses a relational attentive layer to learn features. Heterogenous graph attention networks [3] are another recent approach developed to address "heterogenous" graphs with varying types of nodes and links. They use node level attention combined with semantic level attention. The latter learns about "meta-paths" which are a type of relation between two object. These two attention values are then used to learn node embeddings by finding optimal combinations.

Final project for COSC 525 - Deep Learning - with Dr. Amir Sadovnik at the University of Tennessee, Knoxville, Spring 2021.

III. Technical Approach

A. Graph Neural Networks

Graphs are data structures consisting of (V, E) where V is the set of nodes, or vertices, in the graph and E is the set of edges connecting those nodes, representing relationships between the nodes they connect. Graphs arise as a natural formulation for data in many applications, including social network analysis, recommendation systems, and knowledge graphs. Because of the rich knowledge encoded within graphs, learning on these data structures can yield powerful models that are able to capture the complex relationships encoded by graphs.

Because of the non-Euclidean properties of graphs, developing vectorized representations of graphical data poses some unique challenges. Graphs can contain complex relationships between nodes that may deviate from the sequential nature of time series and natural language data. In addition, the topology of graphs are often not fixed, even across the same application domain, unlike image or grid-based data where deep learning models can leverage the grid-like properties

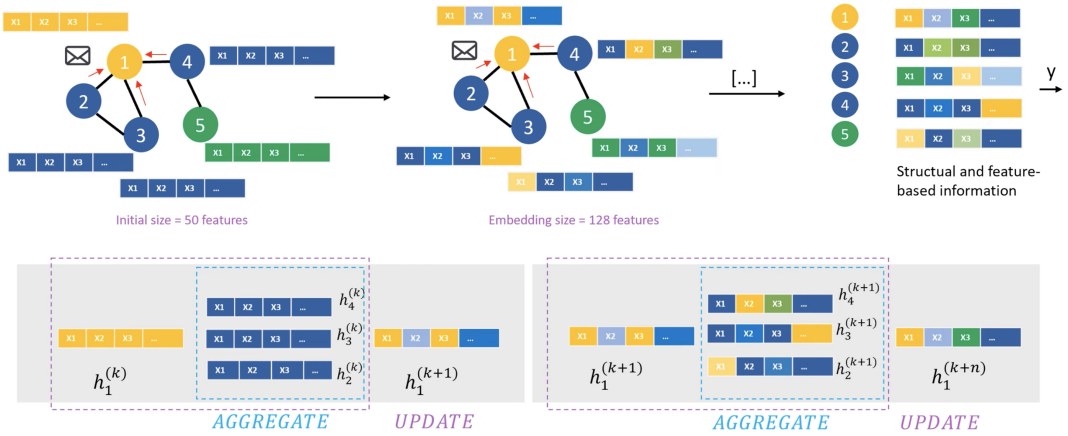


Fig. 1. Message Passing diagram, taken from [4]. Successive message-passing layers result in a mixture of states of surrounding nodes within the embedding of each node in the graph. Update and Aggregate functions work together to produce a final node embedding.

of the data. In addition, common representations of graphs such as adjacency matrices do not suffice to holistically represent the graph in a commonly-known framework such as convolutional neural networks. An isomorphism of the graph may produce a differing adjacency matrix, and the ordering of rows and columns in the adjacency matrix is arbitrary when representing the graph. Therefore, there could be many permutations of the adjacency matrix to represent the same graph, and convolutional networks as well as fully-connected networks are not able to capture the equivalency of these isomorphic structures.

Therefore, in order to build systems that can generate representations of graphs, methods adopt strategies from both convolutional networks as well as sequential networks such as recurrent networks and transformers. Graph neural networks, or GNNs, are a family of deep learning frameworks that seek to develop representations of graphs in order to perform some learning task on that graph. GNNs are constructed in an encoder-decoder architecture; however, most of the time, the encoder is followed by fully-connected layers that perform predictions on the embeddings. Thus, the bulk of the innovation in GNNs is in the encoder.

In order to embed graphs, GNNs make use of so-called message-passing layers. In a graph (V, E), we say that each $u \in V$ has an embedding h_u; initially, this is often the vector encoding of the node properties from the training data. This h_u embedding will be updated successively across message-passing layers, where $h_u^{(k)}$ denotes the embedding of node u at time step k. In message-passing layers, we define two functions, Γ and Φ. Φ is known as the *aggregator* function, and it combines all embeddings $\{h_v^k, \forall v \in \mathcal{N}(u)\}$ where $\mathcal{N}(u)$ is the neighborhood of nodes directly connected to node u through u's edges. The Γ function is then known as the *update* function, combining $h_u^{(k)}$ with the output from Φ. The embedding obtained from Γ is the stored as $h_u^{(k+1)}$, the successive embedding for node u. Therefore, one message-passing layer can be written:

$$h_u^{(k+1)} = \Gamma\Big(h_u^{(k)}, \Phi(\{h_v^k, \forall v \in \mathcal{N}(u)\})\Big)$$

As seen in Figure 1, each embedding of the graph is updated iteratively by stacking multiple message-passing layers. Thus, the idea of a "time step" in this context refers to this iterative updating of node embeddings. Additionally, one can add functionality in Φ for incorporating edge features, but this is not explored in our project due to the structure of our data. This same process is repeated for every node in the graph until an embedding is obtained for every node. This procedure can be repeated for multiple time steps by adding more message passing layers. However, there is a balance between how many message-passing layers to include in a model; too few layers will not capture long-term dependencies through multiple neighbors, but too many layers will lead to a phenomenon known as *oversmoothing* where the embedding for nodes in the graph become indistinguishable from one another. At the end of the message passing layers, a pooling layer is used to downsample all embeddings into a singular vector that can then be input into the decoder, which is often made up of one or more fully-connected layers.

In general, different types of GNNs simply alter the method by which the Γ and Φ functions are learned. One popular paper from Kipf and Welling [5] simply aggregates neighbors and the node u by taking a normalized sum of the neighbor neighbors; the function to compute the embedding is then learned. In other work, the attention mechanism has been applied to both aggregation and updating [6]. The choice of layer often depends on the task at hand. There are three main types of GNN problems: node-level predictions, edge-level predictions, and graph-level predictions. In this project, we will focus on graph-level predictions, where we try to predict some property of the entire graph itself rather than individual portions of the graph.

B. Temporal Graph Neural Networks

One problem which has not seen very much attention in the literature until recently is the idea of applying GNNs to

dynamic graphs. A dynamic graph features a time component, with changing node and edge features throughout time. One example of a dynamic graph would be a social network anaylyzed at snapshots over time, where individuals change their relationships with other individuals in the network, and the properties for each individual itself may also change. In order to learn dynamic graphs, the idea of GNNs must be extended to learn representations over time, treating the dynamic graph as a sequence. These types of GNNs are known as temporal graph neural networks. This idea will be discussed throughout the report.

C. Graph Neural Network Models

1) Static GNNs: As a first step in the project, we ran several static GNNs - models that do not have a temporal component - in order to evaluate how well we could perform when ignoring the temporal nature of the synthetic data. The types of static GNN architectures utilized in this study are shown below:

- Graph Convolutional Network [5]: This model can be likened to the AlexNet [7] moment for graph neural networks and graph representation learning. Graph convolutional layers extend the idea of convolutions to graphs. We can think of an image as being a grid-based graph that performs the a convolution on a local region of the image; likewise, graph convolutions perform operations on a node and its neighbors, using local information to inform construction of embeddings. Successive convolutional layers end in embeddings that are more mixed throughout the graph, expanding the receptive field of each node. This model offers the basis for other models in graph representation learning, and it has an easy-to-use implementation in Pytorch Geometric.
- Graph SAGE: The Graph SAGE operator was first introducted in [8]. The original intention for the operator was for large dynamic graphs. Generation of a low-dimensional embedding of a large graph is useful for various reasons, but most existing methods required that the entire graphs is used to generate the embedding for a node - thus being *transductive*. Graph SAGE is an inductive framework that generates embeddings for new data by by learning a function that creates embeddings by looking at the node's neighborhood. Pytorch Geometric offers this operator as a layer, which we then used to create a very simple architecture with a Graph SAGE (or SAGEConv as it's called in Pytorch Geometric) layer before a linear layer.
- Graph Attention Network: Graph Attention Networks [6] use masked self-attention layers based on those used in NLP problems. They attend to the features of a node's neighborhood to add a trainable weight for those features. This approach also allows for inductive learning as well as better performance on heterogenous graphs. This model is also available in Pytorch Geometric as a GATConv layer. The number of attentional heads can be varied.

2) Temporal GNNs: In addition to static GNNs, we ran several temporal GNNs in order to try and improve on accuracy from the static models. We experimented with the following models:

- Diffusion Convolutional Gated Recurrent Unit GNN (DCGRU): The DGRNN was introduced in [9] to tackle the problem of Traffic forecasting which is an example of a complex spatiotemporal forecasting problem due to the spatial natures of road networks, temporal dynamics from weather and other factors, and so on. They created an encoder-decoder network and use other steps to deal with this problem, but our (and Pytorch Geometric Temporal's) takeaway is the DCGRU, or the GRU portion of the network in the paper. The DCGRU is a GRU with the matrix multiplications replaced by the diffusion convolution. The diffusion convolution is based on the diffusion process and is characterized by a bidirectional random walk on the graph. To learn more, please refer to [9].
- Chebyshev Graph Convolutional Gated Recurrent Unit Cell (GCGRU): The GCGRU was introduced in [10] which was a paper that tried to generalize RNNs and CNNs to graphs. They generalized the convolutional LSTM (and any vanilla RNN) model by replacing the 2D convolution operator by the graph convolution operator. The Chebychev coefficients are a parameter for the GRU. To learn more, refer to [10].

IV. Dataset and Implementation

A. Synthetic Dataset

We use a synthetic dataset in this project that is taken from Owen's research in mathematical modeling. The dataset is built from an agent-based model that models opioid addiction on a social network. The model generates a dynamic social network, where individuals in a population change addiction status, and edges in the network are altered as individuals die (removed) and are born (introduced) into the network. The model is highly stochastic, with state transitions occurring via a Poisson process. The model consists of five mutually exclusive compartments to which individuals are partitioned into:

- **Susceptible**: Individuals who are not addicted to either prescription opioids or heroin and who are not currently prescribed any opioids.
- **Prescribed**: Individuals who are currently prescribed opioids from a healthcare provider.
- **Addicted to prescription opioids**: Individuals whose level of misuse of opioids can be categorized as addiction. This class is also simply known as "Addicted".
- **Addicted to Heroin**: Individuals who are abusing illicit heroin. This class is also simply known as "Heroin".
- **Recovered**: Individuals who have completed a treatment program and are no longer addicted to either prescription opioids or heroin.

The social networks in the model are built on one of three theoretical frameworks for constructing random graphs: Erdos-Renyi [11], Barabasi-Albert [12], and Watts-Strogatz [13]. The

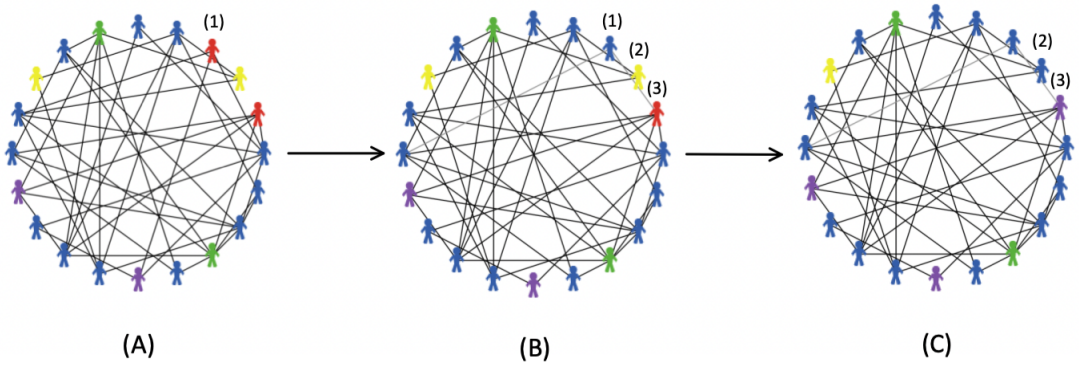


Fig. 2. The synthetic dataset is shown in this figure. Arrows denote a transition in time step, therefore (A) is at time step t, (B) at $t+1$ and (C) at $t+2$. Note two important properties of the network are shown here: dynamic edge changes and node attribute changes. In (A), individual (1) is Addicted (red), but in (B), that turtle has died and been reintroduced into the network as a Susceptible (blue) individual. Note that (1)'s edges are changed as a result of this death from (A) to (B). From (B) to (C), both individuals (2) and (3) change classes, from Prescribed (yellow) to Susceptible (blue) and from Addicted (red) to Heroin addicted (purple), respectively.

underlying stochastic model performed on the network stays constant throughout the differing network topologies.

The manner of the transitions between compartments are driven by the arrangement of connections in the network. For example, an individual has a higher probability of becoming addicted to heroin if that individual has several neighbors who are also addicted to heroin. Therefore, while there is a high level of stochasticity in the model, there is a clear underlying model architecture and patterns that arise from said architecture. We hypothesize, that this pattern can be learned by a machine learning system.

For this project, we chose to fix the parameters of the mathematical model, varying only the proportions of class populations and the structure of the underlying network. From this, we recorded the final proportion of Heroin individuals after 100 time steps, and this was the label that we tried to predict from the dataset. Each node is modeled as having a one-hot encoded class, with each index in the one-hot vector corresponding the to the 5 classes in the model.

Note that we are not publishing the code to generate these data. The model is currently in the process of preparing for publication, and we do not wish to publish the contents of said model until it has been submitted for review. If you have questions about the model, please contact Owen (oqueen@vols.utk.edu).

B. Chicken Pox Dataset

We also looked at a benchmark dataset that was available in the Pytorch Geometric Temporal library [15] - the *Chickenpox Cases in Hungary* Dataset [14]. The dataset is a time series dataset of weekly reported chickenpox cases in Hungarian counties from January 2005 to January 2015. So, there are over 500 entries for each county without any data missing. Representing this data as a graph gives a 20 (for each county + the capital) vertex graph with 61 edges between the nodes which describes the spatial relation of the counties. This dataset contains properties that make it a good benchmark according to Rozemberczki et. al. including: temporal and spatial autocorrelation - the current case counts are based on the past and a node's neighbors, heteroskedasticity - standard deviation is not constant, seasonality, counties have very different population levels, zero inflation as some counties reported no cases at certain times, and the time horizon of the dataset allows for long-term changes in population and trends. Figure 3 provides a visualization of these properties.

C. Hyperparameters

For every model that we tried in this study, we varied the hyperparameters in order to find a set that optimized the results from the models. These hyperparameter choices are explained below:

1) Fixed Hyperparameters: A list of fixed hyperparameters are shown below:

- **Loss Function**: Mean squared error was used as the loss function across all learning tasks.
- **Optimizer**: The Adam optimizer [16] was used as the optimizer for every task.
- **Epochs**: The number of iterations on the dataset that we train the model. For our task, this number was largely limited by availability of computational resources.
- **Performance Metric**: In order to evaluate the accuracy/effectiveness of our methods, we used scikit-learn's coefficient of determination score [17].

2) Static GNN Hyperparameters: The static GNNs consisted of a few hyperparameters that we were able to experiment with:

- **Number of Hidden Channels**: The dimension of the vector holding the intermediate states of the network. This number is analogous to the number of nodes in the output of hidden layers in a neural network.
- **Learning Rate**: The learning rate of the Adam optimizer used to train the networks.
- **Time Step**: The time step at which we chose for representing the static network. Since we ran the synthetic

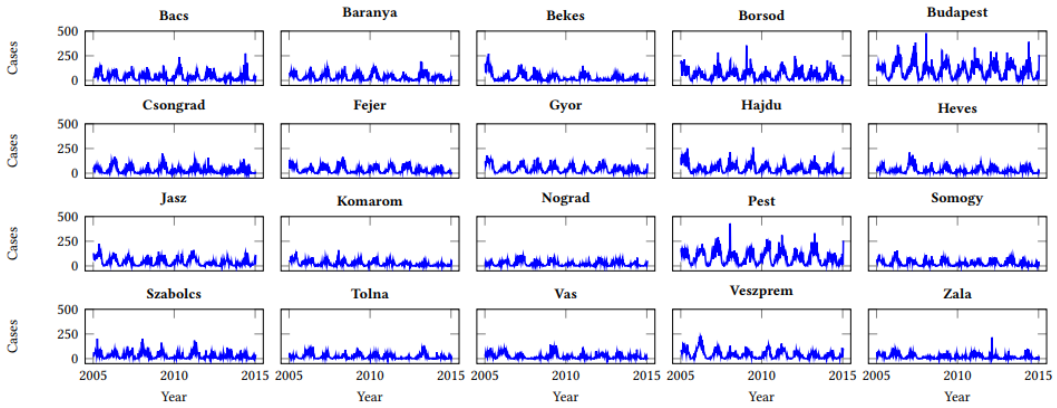


Fig. 3. The time series of Chickenpox cases in Hungary are show above with the x-axis representing weeks from January 2005 to January 2015 and the y-axis the case count for that week. This figure comes from [14].

dataset for 100 time steps, this could have been any time step in $t_0, t_1, ..., t_{99}$.

- **Number of Layers**: The number of message-passing layers used to construct the models. In the introduction to graph neural networks, we discussed the trade-off between using lots of layers in the model - too many can lead to oversmoothing but too few may reduce learning capacity of the network.

3) Temporal GNN Hyperparameters: The temporal networks featured some additional hyperparameters:

- **Time Steps**: This is the number of time steps that we run the model before making a prediction. In the synthetic dataset, we run the model for the first 50 time steps, i.e. time steps $t_0, t_1, ..., t_{49}$. For the Chickenpox dataset, there are 8 time steps (weeks) of data preceding the next time step which is the target.
- **Number of recurrent layers**: Number of recurrent layers stack on one another in the model. For the synthetic and chickenpox dataset, we vary this hyperparameter.
- **K**: Filter size for the first layer.

For the synthetic dataset experiments, we held the following hyperparameters constant due to a lack of computational resources and time to test the hyperparameters: time steps, learning rate, and number of hidden channels. Note that batching was not performed on the synthetic dataset due to a lack of native batching for our type of data in Pytorch Geometric Temporal.

For the chickenpox dataset, we held the following hyperparameters constant due to lack of computational resources and time: time steps (8), learning rate (0.01), number of hidden channels (32).

D. Technology

1) Software: We use the NetworkX [18] library for preprocessing of graph data. All of our deep learning models are built on the Pytorch framework and packages extending the framework. We use Pytorch Geometric [19] for building the static GNNs and assisting in building the temporal GNNs. We also use Pytorch Geometric Temporal [15] for building the temporal GNNs. Pytorch Geometric Temporal is a very new package (released around April 15, 2021), so there were some challenges in integrating our data into the frameworks used by the package.

Most models were constructed in Jupyter Notebooks. Sharing of source code and data was performed through Github and Google Drive, utilizing Google Colab to run notebooks and share code. In addition, we attempted to set up a virtual machine on the Google Cloud Platform on which to run our code, but we were unable to gain access to GPUs for administrative reasons.

2) Hardware: To implement these networks, we utilized both CPU and GPU architectures for various portions of the project. GPUs were provided through Google Colab. **Portion about GPUs**. Because of our difficulties with accessing GPUs on GCP and the limitations of Google Colab, development of the models was slow. For most of the models, we were forced to use the CPU for model training.

V. Experiments and Results Analysis

A. Analysis of Static Graph Neural Networks on Sequence Data

We conducted an analysis of GNN architectures designed for static graphs. A general outline of the experiments performed is shown below:

1) GCN Results: : For the GCN, we see that varying hyperparameters did not seem to change the accuracy of the model by much. A smaller learning rate leads to slower convergence, which is an expected result due to the nature of the learning rate. In addition, a larger number of hidden channels lead to a faster convergence rate. One important result is that varying the time step on which we gathered a network to predict the final proportion of Heroin individuals in the model did not produce significantly different results. We hypothesized that even the GCN with time step 99, the time step right before the final time step, would be able to predict the final proportion well, but this was not the case. This is

most likely indicative of how poorly graph neural networks, particularly static models, perform on this synthetic dataset. See Figure 14 in the Appendix for graphical results of the GCN.

2) GraphSAGE Results: : The GraphSAGE model showed a similar lack of accuracy despite hyperparameter modification. Convergence happened most quickly with a learning rate of 0.1 and a network with three layers. Keeping hidden channels at 64 seemed to produce more stable results. An example of the results is below, with the results of more hyperparameter settings in figure 15 of the Appendix.

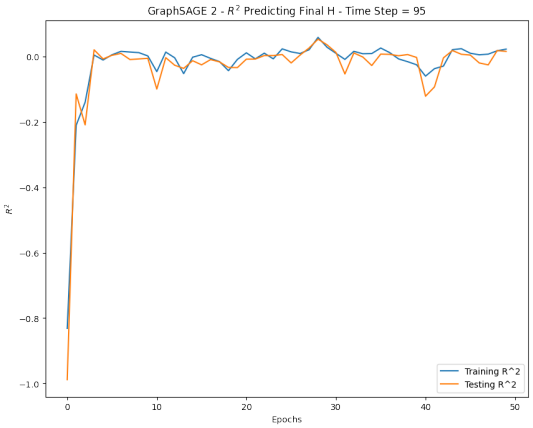


Fig. 4. GraphSAGE results with 3 layers, 64 hidden channels, and learning rate of 0.01

3) GAT Results: : The results using graph attention layers echoed those of the other static networks. Smaller learning rates showed better performance, and 64 channels produced the most stable results. Below is an example of results and more comprehensive results can be found in figure 16 of the Appendix.

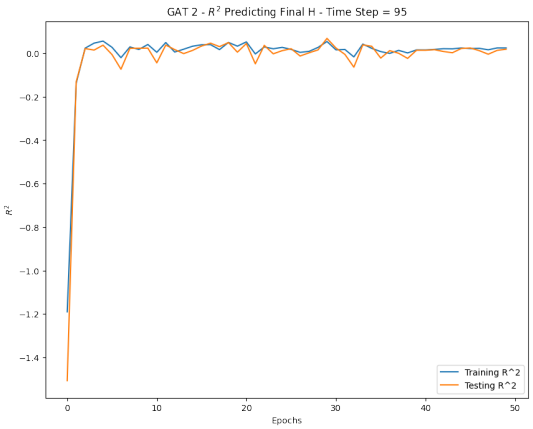


Fig. 5. GAT results with 2 layers, 64 hidden channels, and learning rate of 0.01

B. Temporal Graph Neural Networks

1) Chickenpox Dataset: For the Chickenpox dataset, we used 2 recurrent, spatiotemporal layers: the GCGRU and DCGRU to predict the current cases for every county based on the past 8 weeks of data. The hyperparameters tested were different amount of stacked layers (1-3), as well as the number of filters (1 and 4). The data was split 80/20 into train and test sets. The training and testing loss were recorded as well as the R2 train and test accuracy at every epoch. The epochs varied from 150-200 depending on if the computer (a local machine) could run the model without the Jupyter Notebook kernel crashing. Note that the case counts were standardized.

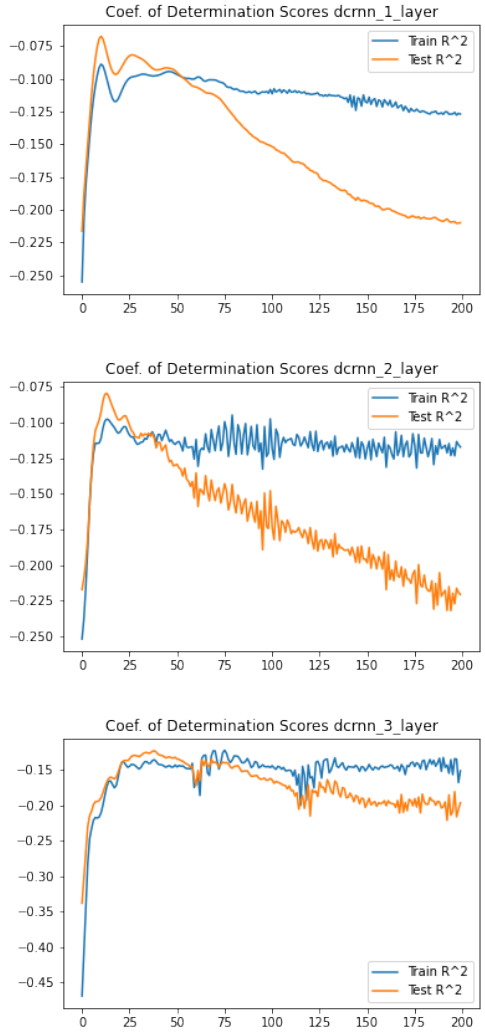
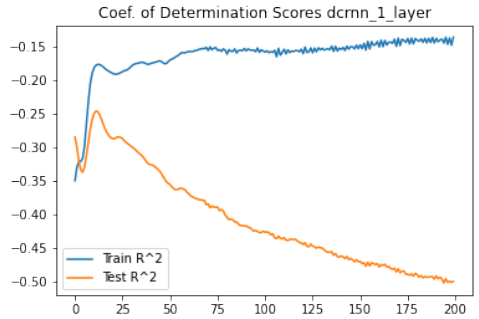
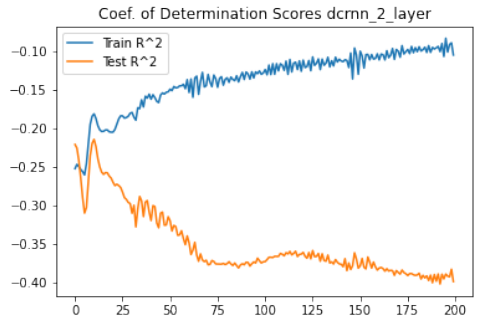


Fig. 6. DCRNN Curves of Coefficient of Determination on the chickenpox dataset for 1,2, and 3 layer and filter size 1.

Figure 6 and 7 seems to suggest that increasing the filter size to 4 didn't help much with final test accuracy or made it slightly worse. Figure 8 and 9 have instances were the lower filter size (1) was more effective while the there were instances that the higher filter size (4) was more effective. From this, it's hard to conclude if changing the filter size from 1 to 4 was meaningful. Unfortunately, testing different filter sizes not computationally feasible given our time and resources.

Figures 6, 7, 8, 9 all show a negative coefficient of deter-





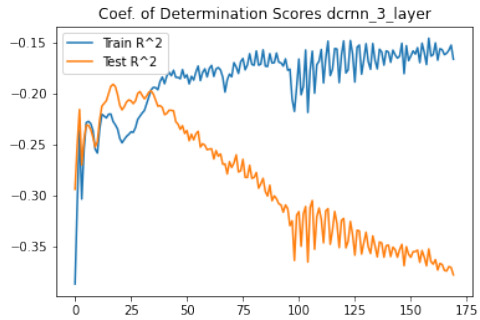
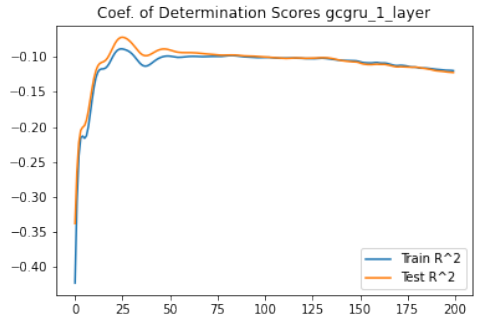
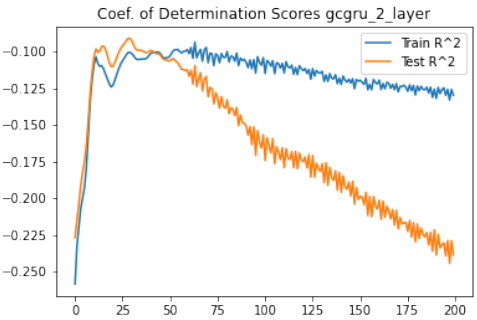


Fig. 7. DCRNN Curves of Coefficient of Determination on the chickenpox dataset for 1,2, and 3 layer and filter size 4.





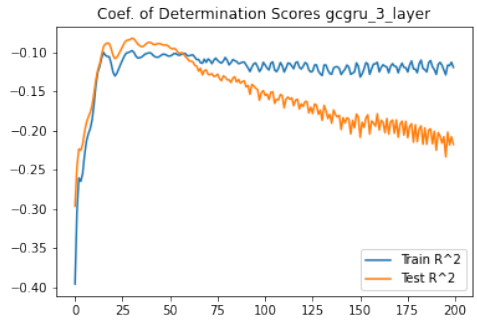


Fig. 8. GCGRU Curves of Coefficient of Determination on the chickenpox dataset for 1,2, and 3 layer and filter size 1.

mination indicating poor performance of all the models and hyperparameters. Also, in almost every one of the charts in the figures show that models starts diverging and overfitting pretty early on and the test accuracy starts decreasing as well after a certain point. While this seems to suggest that simply stacking recurrent layers is inadequate - it is hard to fully conclude without trying to cover the hyperparameter space more broadly.

2) Synthetic Dataset: The two temporal frameworks, GCGRU and DGRNN, were tested on the synthetic dataset to attempt to predict final proportion of Heroin individuals. The only hyperparameter that was tested was the number of stacked recurrent layers included in the model.

In the DGRNN, a smaller number of layers seemed to produce more accurate results. This is most likely due to the oversmoothing effect; adding more layers most likely made the embeddings less distinct, decreasing the accuracy of the method. However, the DGRNN does seem to perform better overall than the static networks, which matches our intuition about temporal models working better on this dynamic data.

There seems to be less distinction between results in the GCGRU than in the DGRNN. However, adding layers has almost the opposite effect where more layers produce more accurate results. Most likely due to model construction, this model is not as sensitive to oversmoothing as DGRNN seems to be on the synthetic dataset.

Figures 12 and 13 show an interesting result. Increasing the filter size from 1 to 8 showed a drastic increase in performance. This suggested that the receptive field was too small with only a filter size of 1 and wasn't extracting, at least, spatial features as well. This seems to be in contrast with the chickenpox dataset results, but it is important to note that the chickenpox dataset only tried up until filter size of 4. Of course that might not matter, since the synthetic dataset has larger graphs (100 nodes vs 20 in chickenpox).

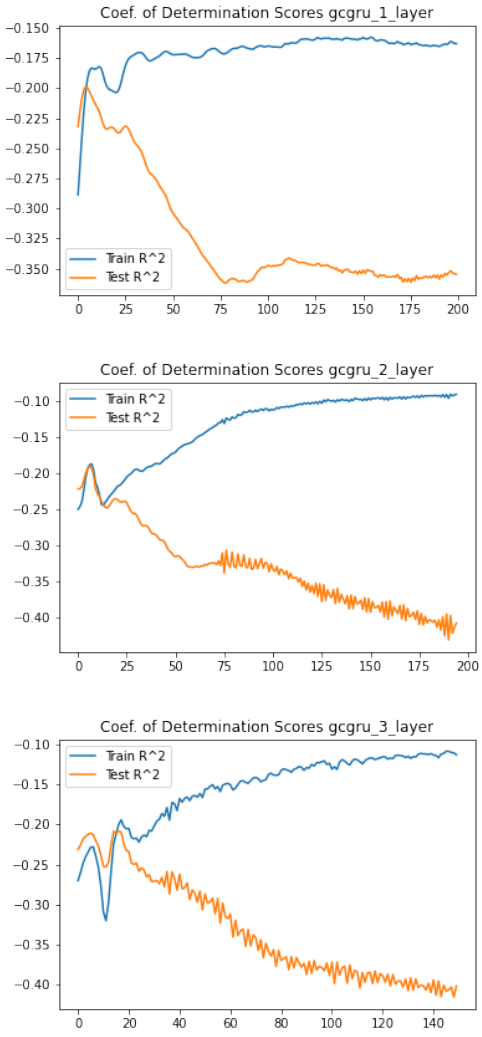


Fig. 9. GCGRU Curves of Coefficient of Determination on the chickenpox dataset for 1,2, and 3 layer and filter size 4.

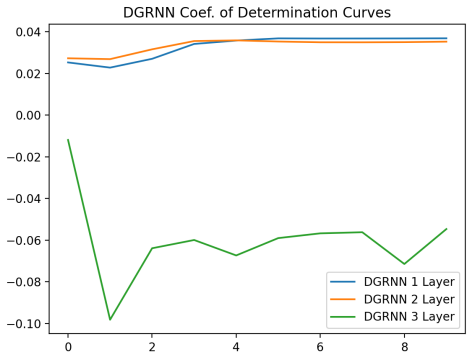


Fig. 10. DGRNN Curves of Coefficient of Determination on the synthetic dataset with default parameters. A significant difference is seen between models with 1, 2 recurrent layers and then 3 recurrent layers.

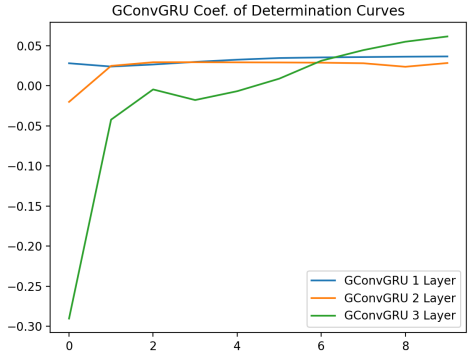


Fig. 11. GCGRU Curves of Coefficient of Determination on the synthetic dataset with default parameters.

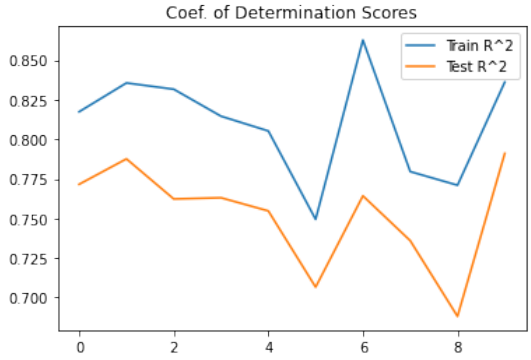


Fig. 12. DCRNN Curves of Coefficient of Determination on the synthetic dataset for 1 layer and filter size 8.

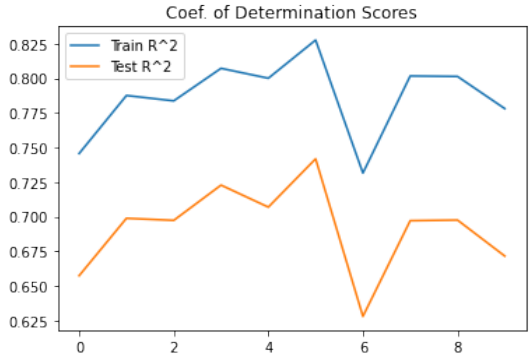


Fig. 13. GCGRU Curves of Coefficient of Determination on the synthetic dataset for 1 layer and filter size 8.

VI. Conclusion

Through the utilization of graph neural networks, we were able to test a variety of architectures for both static and temporal graph frameworks, analyzing hyperparameter choices in these models while using them on both the synthetic dataset and other benchmark datasets. This project was a learning experience for every member in the group, particularly because nobody had any prior experience with graph neural networks.

We were able to gain experience with developing deep learning models in Pytorch and graph neural networks in Pytorch Geometric. We also experienced the limitations of graph neural networks first hand, and this project has given us valuable insight into machine learning problems involving graphs.

We have also shown that our dataset presents a clear challenge for GNNs in that static networks alone cannot learn on it. It was only when we experimented with more sophisticated temporal models that we were able to achieve beter performance on the dataset. We hypothesize that these properties are due to the stochastic nature of the network; the temporal component is essential to learning the meta-patterns in this data. This dataset could serve as an example of a challenge of GNNs and machine learning more broadly, being able to learn when the signal between variables is rather weak. This problem may also be a question of whether our dataset presents a reasonably-learnable task.

There are several modifications that could have been made to this study in order to improve the results:

- **Modifications to data generation**: More controls could have been implemented in the data generation process. We did not control for network structure, but this could have been done in order to analyze whether varying network topologies contribute to the inaccuracies experienced in the results. We could have also controlled the network to prevent it from changing edges; this would have created a quasi-static graph where only node attributes change across time steps. There seems to be more literature available on analyzing these quasi-static graphs, so we could have leveraged this wider diversity of literature to develop state-of-the-art models. Other variables could have also been altered such as size and time steps for the model runs.
- **Trying new datasets**: This project only included the analysis of a few datasets, and this could have been expanded to further explore the capabilities of the GNNs that we were able to try.
- **Different architectures**: In this project, we stuck to using previously-implemented architectures upon which to build our models. However, we could have explored expanding our implementations to more recent models, such as those mentioned in our proposal.
- **Running models on GCP**: We could have set up a virtual machine earlier with the hope of running our models on GCP. However, since we got a late start on the project, we were unable to fully take advantage of this resource that could have sped up development.

Overall, we all learned a lot from this project by getting our hands dirty with these models, and it was a nice culmination to the course. Graph neural networks are an exciting new domain in deep learning, and we all look forward to exploring them more in our own research.

REFERENCES

[1] E. Rossi, B. Chamberlain, F. Frasca, D. Eynard, F. Monti, and M. Bronstein, "Temporal graph networks for deep learning on dynamic graphs," 2020.

[2] C. Hsu and C.-T. Li, "Retagnn: Relational temporal attentive graph neural networks for holistic sequential recommendation," 2021.

[3] X. Wang, H. Ji, C. Shi, B. Wang, P. Cui, P. Yu, and Y. Ye, "Heterogeneous graph attention network," 2021.

[4] DeepFindr, ""understanding graph neural networks — part 2/3 - gnns and it's variants"."

[5] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[6] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," *arXiv preprint arXiv:1710.10903*, 2017.

[7] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, pp. 1097–1105, 2012.

[8] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," *arXiv preprint arXiv:1706.02216*, 2017.

[9] Y. Li, R. Yu, C. Shahabi, and Y. Liu, "Diffusion convolutional recurrent neural network: Data-driven traffic forecasting," 2018.

[10] Y. Seo, M. Defferrard, P. Vandergheynst, and X. Bresson, "Structured sequence modeling with graph convolutional recurrent networks," 2016.

[11] P. Erdős and A. Rényi, "On the evolution of random graphs," *Publ. Math. Inst. Hung. Acad. Sci*, vol. 5, no. 1, pp. 17–60, 1960.

[12] A.-L. Barabási and R. Albert, "Emergence of scaling in random networks," *Science*, vol. 286, no. 5439, pp. 509–512, 1999.

[13] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world'networks," *Nature*, vol. 393, no. 6684, pp. 440–442, 1998.

[14] B. Rozemberczki, P. Scherer, O. Kiss, R. Sarkar, and T. Ferenci, "Chickenpox Cases in Hungary: a Benchmark Dataset for Spatiotemporal Signal Processing with Graph Neural Networks," 2021.

[15] B. Rozemberczki, P. Scherer, Y. He, G. Panagopoulos, M. Astefanoaei, O. Kiss, F. Beres, N. Collignon, and R. Sarkar, "PyTorch Geometric Temporal: Spatiotemporal Signal Processing with Neural Machine Learning Models," 2021.

[16] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[17] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[18] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using networkx," in *Proceedings of the 7th Python in Science Conference* (G. Varoquaux, T. Vaught, and J. Millman, eds.), (Pasadena, CA USA), pp. 11 – 15, 2008.

[19] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch Geometric," in *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

VII. APPENDICES

A. Code Design

1) Processing the Synthetic Dataset: The synthetic data was generated from software written one of Owen's research groups, and it was stored in NetworkX objects and Pickled in order to transfer it between systems. The original benchmark synthetic dataset we used is located in code included with this report and in the Github. After loading in this data to our notebooks, there are also a few processing tools to transfer it into the relevant format for either Pytorch Geometric or Pytorch Geometric Temporal. In Pytorch Geometric, we use a DataLoader object to hold the data; this automatically performs batching in the training loops. In Pytorch Geometric Temporal, we use their custom-built loader known as a Signal; the signal allows you to represent every time step of one run of the network model. Therefore, in order to run multiple time series samples, we had to store signals in a list and iteratively process each of these. We had to resort to this rather primitive form of processing data because Pytorch Geometric Temporal

does not yet have the robust data processing infrastructure like more mature packages such as Pytorch Geometric.

2) Running Pytorch Models: The models were run inside Jupyter Notebooks in order to code more interactively. Each model was definied by its own class with its own init() function as well as a forward function to define the forward propogation of the model. There are some quirks with Pytorch compared to TensorFlow with the user having to indicate when the model was training through a "model.train()" and when it was in inference model with "model.eval()" so Pytorch can know whether to stop certain training-only aspects of the network such as Dropout.

B. Workload Distribution

- **Owen** worked on gathering the synthetic data and setting up the processing code so that we could all run models in Pytorch. He also built and performed analysis of the Graph Convolutional Network, and he applied the temporal models on the synthetic dataset. In addition, he wrote the introduction on graph neural networks, and he wrote the descriptions of the synthetic dataset.
- **Sai** worked on building the Graph SAGE model, and both the temporal models which he then used to model the chickenpox dataset. He also ran the experiment for comparing the larger filter size for the synthetic dataset. He also worked on the writing the abstract, introduction, and the description of the chickenpox dataset.
- **Rekha** worked on building the Graph Attention Network model, and ran more tests with that and the Graph SAGE model to assess performance with a variety of parameters. She wrote about the results for that test, as well as the previous work section. She also experimented with some combined static models.

C. Additional Figures

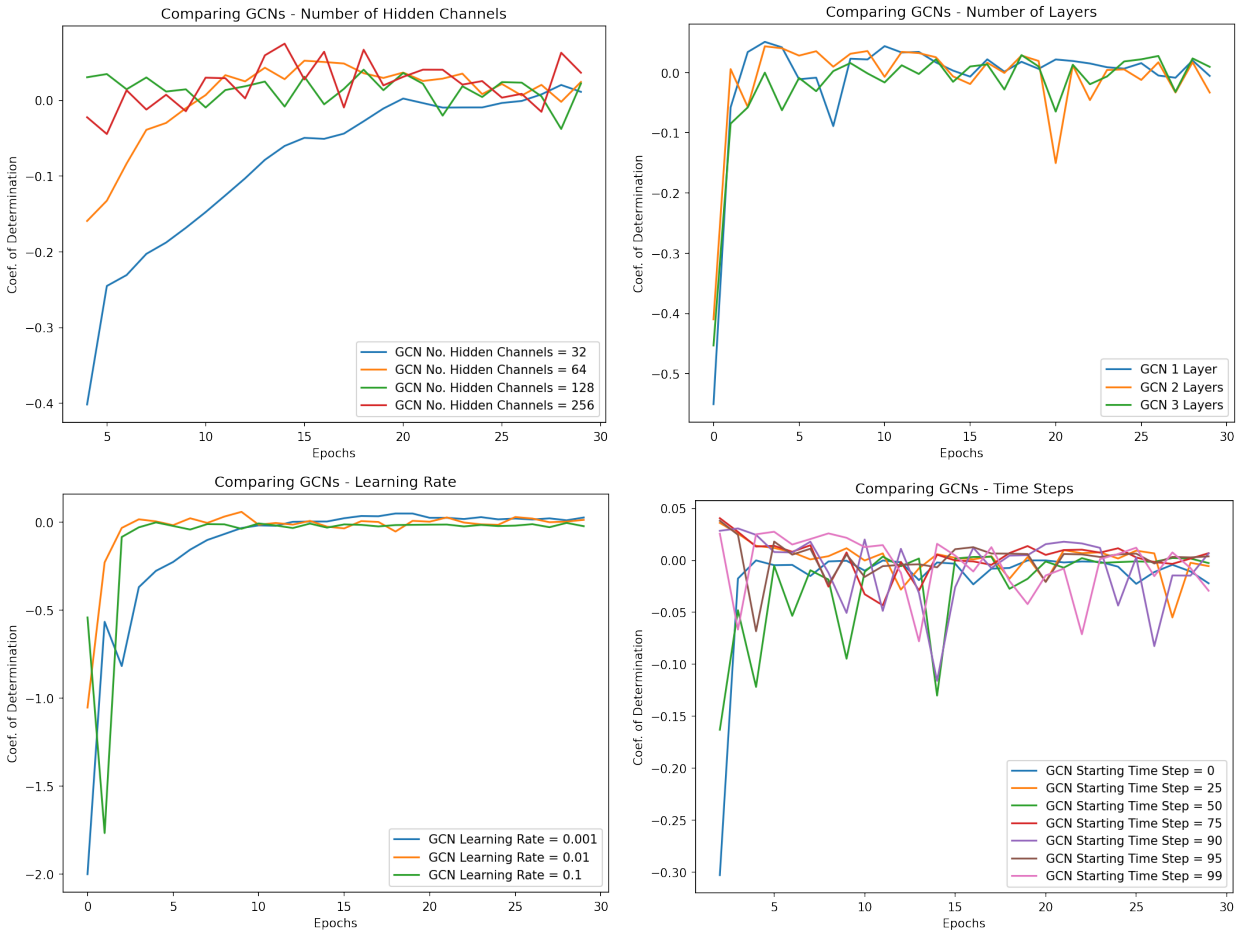


Fig. 14. Graph convolutional network results on the synthetic dataset when varying hyperparameters. Unless stated, we fixed parameters at: 64 hidden channels, 2 layers, 0.01 learning rate, and time step = 95.

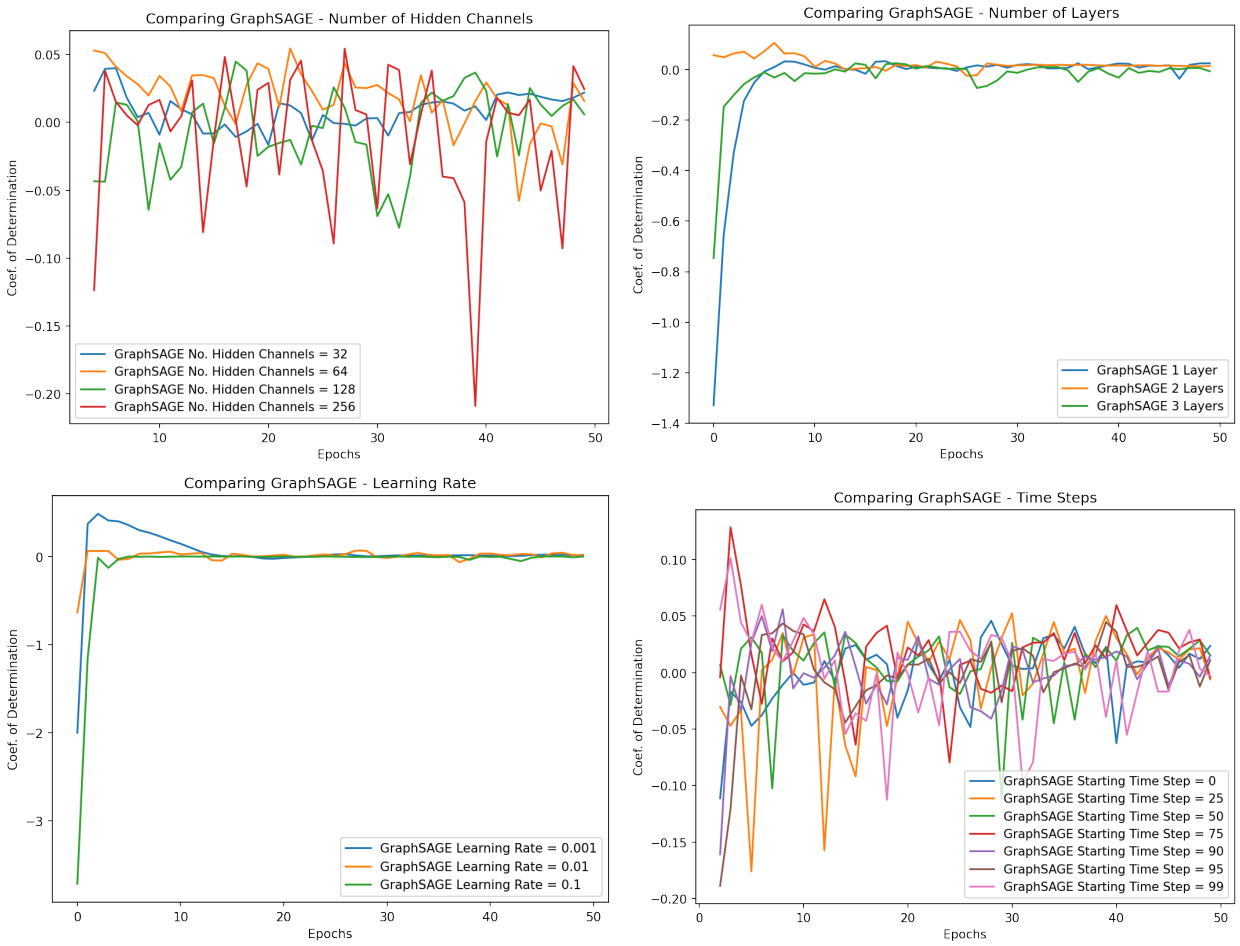


Fig. 15. GraphSAGE network results on the synthetic dataset when varying hyperparameters. Unless stated, we fixed parameters at: 64 hidden channels, 3 layers, 0.01 learning rate, and time step = 95.

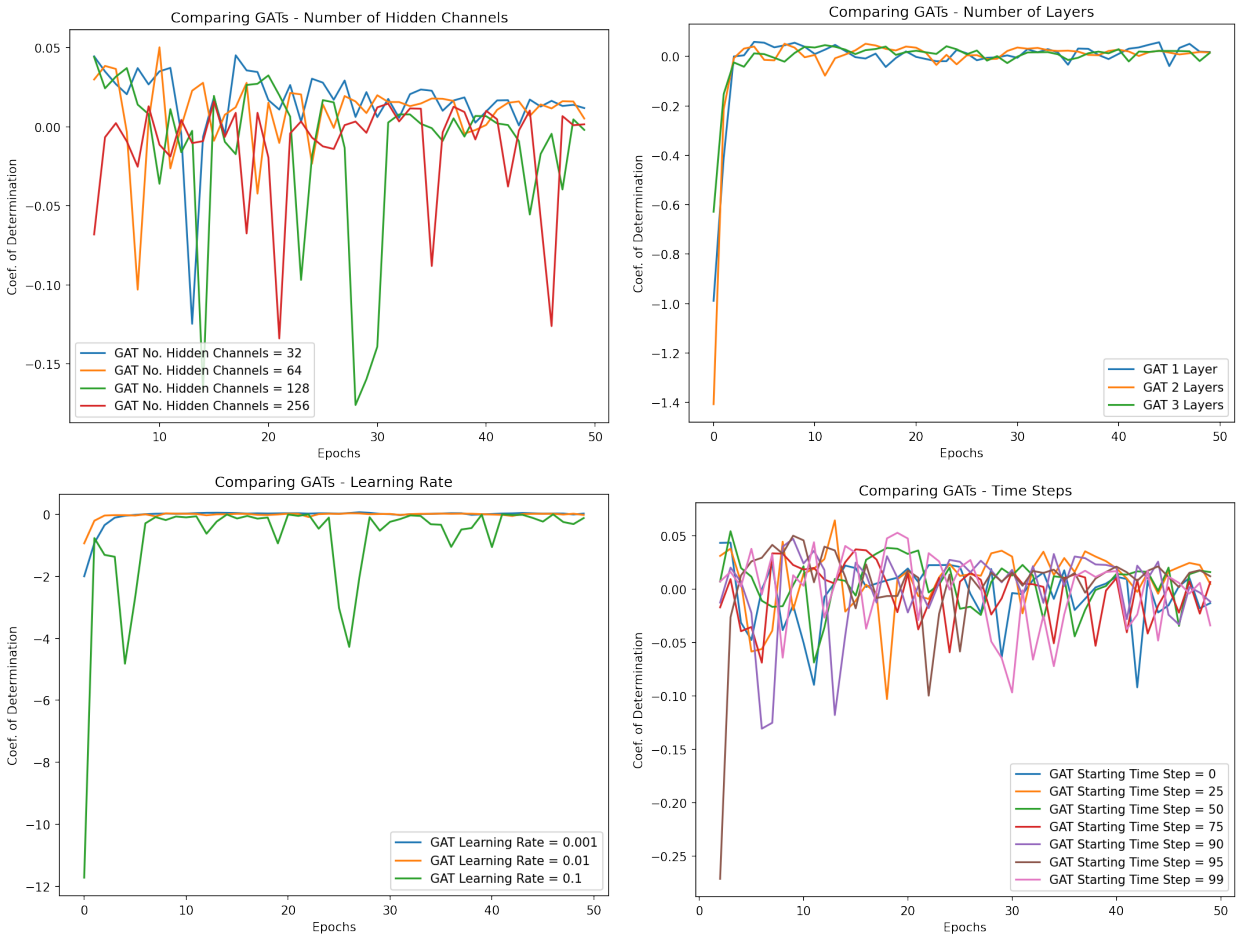
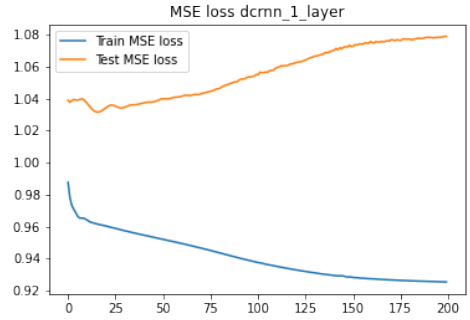
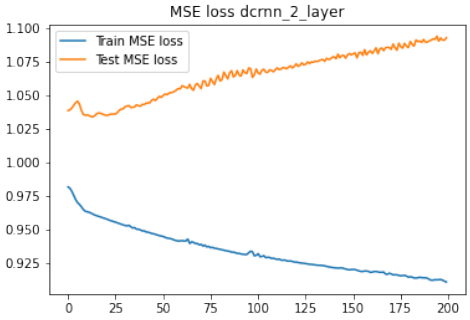


Fig. 16. Graph Attention network results on the synthetic dataset when varying hyperparameters. Unless stated, we fixed parameters at: 64 hidden channels, 3 layers, 0.01 learning rate, and time step = 95.





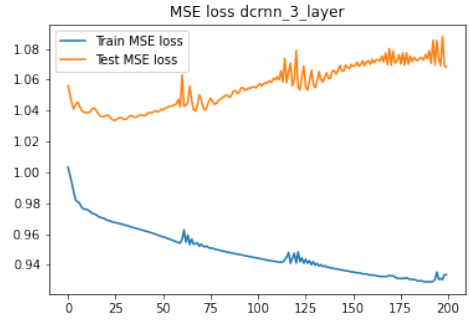
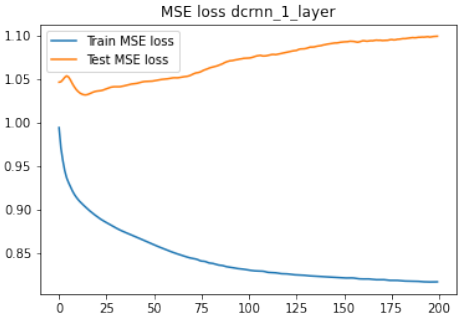
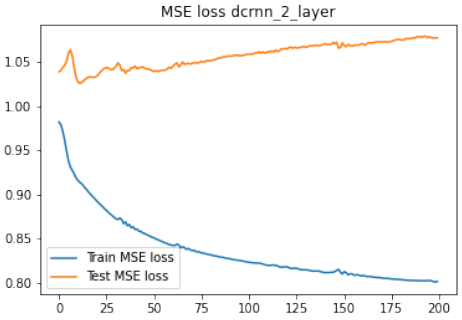


Fig. 17. DCRNN MSE loss on the chickenpox dataset for 1,2, and 3 layer and filter size 1.





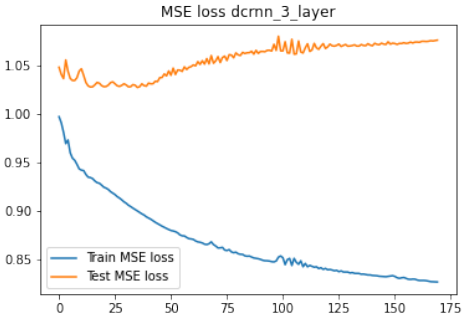
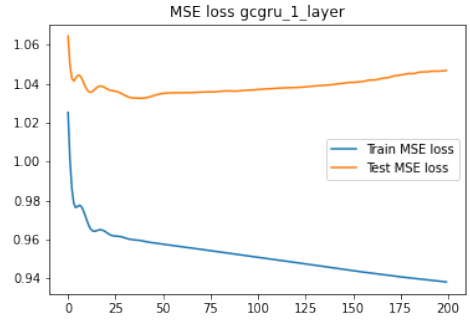
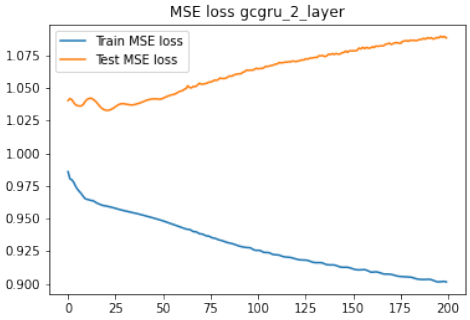


Fig. 18. DCRNN MSE loss on the chickenpox dataset for 1,2, and 3 layer and filter size 4.





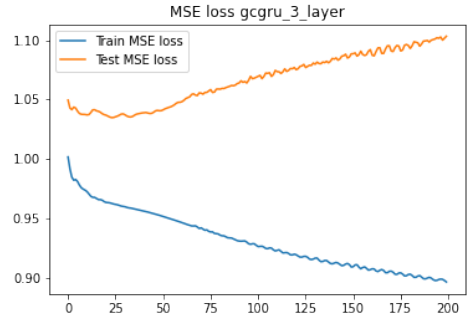
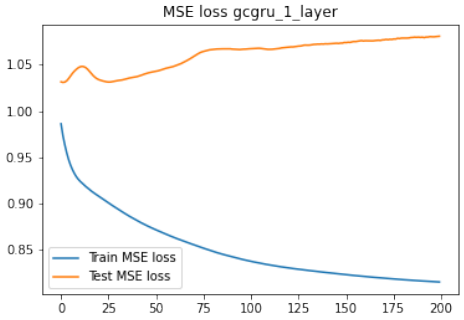
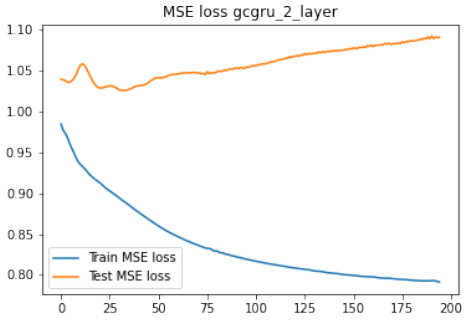


Fig. 19. GCGRU MSE loss on the chickenpox dataset for 1,2, and 3 layer and filter size 1.





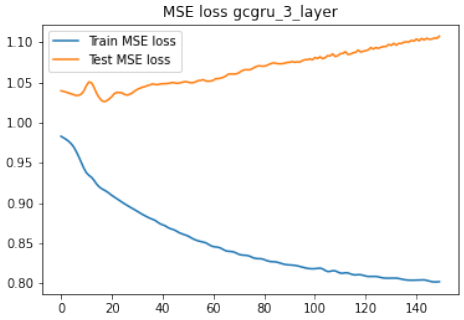


Fig. 20. GCGRU MSE loss on the chickenpox dataset for 1,2, and 3 layer and filter size 4.