

A APPENDIX / SUPPLEMENTAL MATERIAL

The Python implementation of our proposed framework and baseline methods is available at the following [Github Repository](#).

A.1 CELL CULTURE

Approximately 10^6 cells were plated on each Multielectrode Array. Neuronal cells were cultured either from the cortices of E15.5 mouse embryos or differentiated from human induced pluripotent stem cells via a dual SMAD inhibition (DSI) protocol or through a lentivirus-based NGN2 direct differentiation protocols as previously described (6). Cells were cultured until plating. For primary mouse neurons, this occurred at day-in-vitro (DIV) 0, for DSI cultures this occurred at between DIV 30 - 33 depending on culture development, for NGN2 cultures this occurred at DIV 3.

A.2 MEA SETUP AND PLATING

MaxOne Multielectrode Arrays (MEA; Maxwell Biosystems, AG, Switzerland) was used and is a high-resolution electrophysiology platform featuring 26,000 platinum electrodes arranged over an 8 mm². The MaxOne system is based on complementary meta-oxide-semiconductor (CMOS) technology and allows recording from up to 1024 channels. MEAs were coated with either polyethylenimine (PEI) in borate buffer for primary culture cells or Poly-D-Lysine for cells from an iPSC background before being coated with either 10 µg/ml mouse laminin or 10 µg/ml human 521 Laminin (Stemcell Technologies Australia, Melbourne, Australia) respectively to facilitate cell adhesion. Approximately 10^6 cells were plated on MEA after preparation as per (6). Cells were allowed approximately one hour to adhere to MEA surface before the well was flooded. The day after plating, cell culture media was changed for all culture types to BrainPhys™ Neuronal Medium (Stemcell Technologies Australia, Melbourne, Australia) supplemented with 1% penicillin-streptomycin. Cultures were maintained in a low O₂ incubator kept at 5% CO₂, 5% O₂, 36°C and 80% relative humidity. Every two days, half the media from each well was removed and replaced with free media. Media changes always occurred after all recording sessions.

A.3 DISHBRAIN PLATFORM AND ELECTRODE CONFIGURATION

The current DishBrain platform is configured as a low-latency, real-time MEA control system with on-line spike detection and recording software. The DishBrain platform provides on-line spike detection and recording configured as a low-latency, real-time MEA control. The DishBrain software runs at 20 kHz and allows recording at an incredibly fine timescale. This setup captured neuronal electrical activity and provided long-term, safe external electrical stimulation through biphasic pulses that elicited action potentials in neurons, as detailed in previous studies (34). There is the option of recording spikes in binary files, and regardless of recording, they are counted throughout 10 milliseconds (200 samples), at which point the game environment is provided with how many spikes are detected in each electrode in each predefined motor region as described below. Based on which motor region the spikes occurred in, they are interpreted as motor activity, moving the ‘paddle’ up or down in the virtual space. As the ball moves around the play area at a fixed speed and bounces off the edge of the play area and the paddle, the pong game is also updated at every 10ms interval. Once the ball hits the edge of the play area behind the paddle, one rally of pong has come to an end. The game environment will instead determine which type of feedback to apply at the end of the rally: random, silent, or none. Feedback is also provided when the ball contacts the paddle under the standard stimulus condition. A ‘stimulation sequencer’ module tracks the location of the ball relative to the paddle during each rally and encodes it as stimulation to one of eight stimulation sites. Each time a sample is received from the MEA, the stimulation sequencer is updated 20,000 times a second, and after the previous lot of MEA commands has completed, it constructs a new sequence of MEA commands based on the information it has been configured to transmit based on both place codes and rate codes. The stimulations take the form of a short square bi-phasic pulse that is a positive voltage, then a negative voltage. This pulse sequence is read and applied to the electrode by a Digital to Analog Converter (or DAC) on the MEA. A real-time interactive version of the game visualiser is available at <https://spikestream.corticallabs.com/>. Alternatively, cells could be recorded at ‘Rest’ in a Gameplay environment where activity was recorded to move the

paddle but no stimulation was delivered, with corresponding outcomes still recorded. Using this spontaneous activity alone as a baseline, the Gameplay characteristics of a culture were determined. Low level code for interacting with Maxwell API was written in C to minimize processing latencies-so packet processing latency was typically $<50 \mu s$. High-level code was written in Python, including configuration setups and general instructions for game settings. A 5 ms spike-to-stim latency was achieved, which was substantially due to MaxOne's inflexible hardware buffering. Fig. S1 illustrates a schematic view of Software components and data flow in the DishBrain closed loop system.

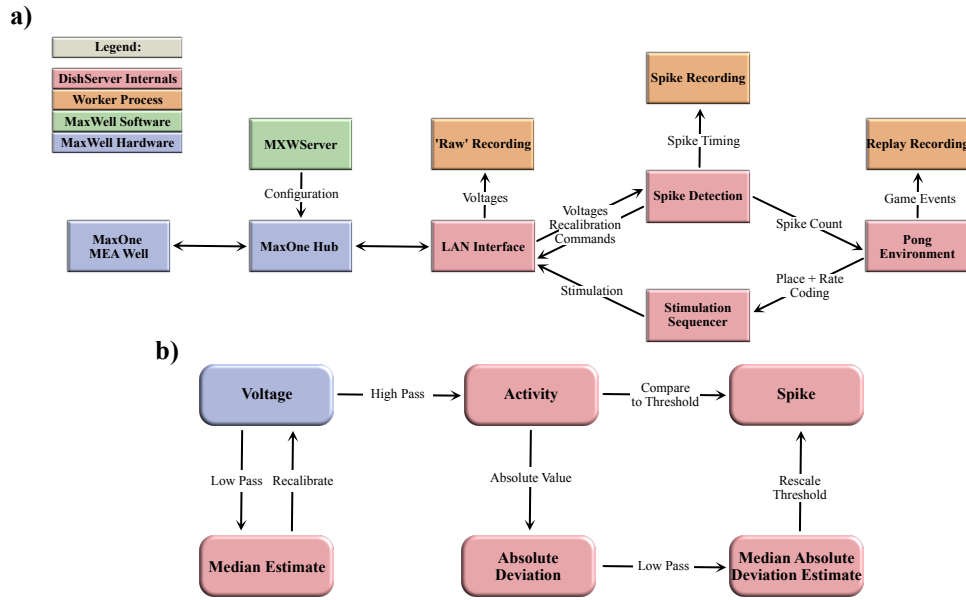


Figure S1: **a, b)** Schematics of software used for DishBrain. **a)** Software components and data flow in the DishBrain closed loop system. Voltage samples flow from the MEA to the ‘Pong’ environment, and sensory information flows from the ‘Pong’ environment back to the MEA, forming a closed loop. The blue rectangles mark proprietary pieces of hardware from MaxWell, including the MEA well which may contain a live culture of neurons. The green MXWServer is a piece of software provided by MaxWell which is used to configure the MEA and Hub, using a private API directly over the network. The red rectangles mark components of the ‘DishServer’ program, a high-performance program consisting of four components designed to run asynchronously, despite being run on a single CPU thread. The ‘LAN Interface’ component stores network state, for talking to the Hub, and produces arrays of voltage values for processing. Voltage values are passed to the ‘Spike Detection’ component, which stores feedback values and spike counts, and passes recalibration commands back to the LAN Interface. When the pong environment is ready to run, it updates the state of the paddle based on the spike counts, updates the state of the ball based on its velocity and collision conditions, and reconfigures the stimulation sequencer based on the relative position of the ball and current state of the game. The stimulation sequencer stores and updates indices and countdowns relating to the stimulations it must produce and converts these into commands each time the corresponding countdown reaches zero, which are finally passed back to the LAN Interface, to send to the MEA system, closing the loop. The procedures associated with each component are run one after the other in a simple loop control flow, but the ‘Pong’ environment only moves forward every 200th update, short-circuiting otherwise. Additionally, up to three worker processes are launched in parallel, depending on which parts of the system need to be recorded. They receive data from the main thread via shared memory and write it to file, allowing the main thread to continue processing data without having to hand control to the operating system and back again. **b)** Numeric operations in the real-time spike detection component of the DishBrain closed loop system, including multiple IIR filters. Running a virtual environment in a closed loop imposes strict performance requirements, and digital signal processing is the main bottleneck of this system, with close to 42 MB of data to process every second. Simple sequences of IIR digital filters is applied to incoming data, storing multiple arrays of 1024 feedback values in between each sample. First, spikes on the incoming data are detected by applying a high pass filter to determine the deviation of the activity, and comparing that to the MAD, which is itself calculated with a subsequent low pass filter. Then, a low pass filter is applied to the original data to determine whether the MEA hardware needs to be re-calibrated, affecting future samples. This system was able to keep up with the incoming data on a single thread of an Intel Core i7-8809G. Figures adapted from (6).

A.4 ADDITIONAL RESULTS

In this section, we present the learned representations of the three best performing windows in terms of the culture’s hit/miss ratios during Gameplay for two additional cultures in Figs. S2 and S3. The figures repeatedly demonstrate TAVRNN’s outperformance over the other baseline methods in identifying clusters of channels that belong to the same region on the HD-MEA.

Additionally, Fig. S4 represents t-SNE visualization of the learned representations of the three best and three worst windows based on hit/miss ratios ($High^{1,2,3}$ and $Low^{1,2,3}$) during Gameplay and Rest, as modeled by TAVRNN for all aggregated trials of an additional sample culture. These visualizations reveal an absence of distinguishable clusters during the rest state or during low-performing periods of gameplay. However, as we progress to time windows associated with higher performance levels in the game, distinct clustering patterns emerge. Notably, channels from the motor regions associated with *Up* and *Down* movements form distinct, cohesive clusters, despite the spatial separation of these channels (within each of the *Up* or *Down* subregions) on the HD-MEA. Similarly, channels from the *Sens* region group together into a separate cluster.

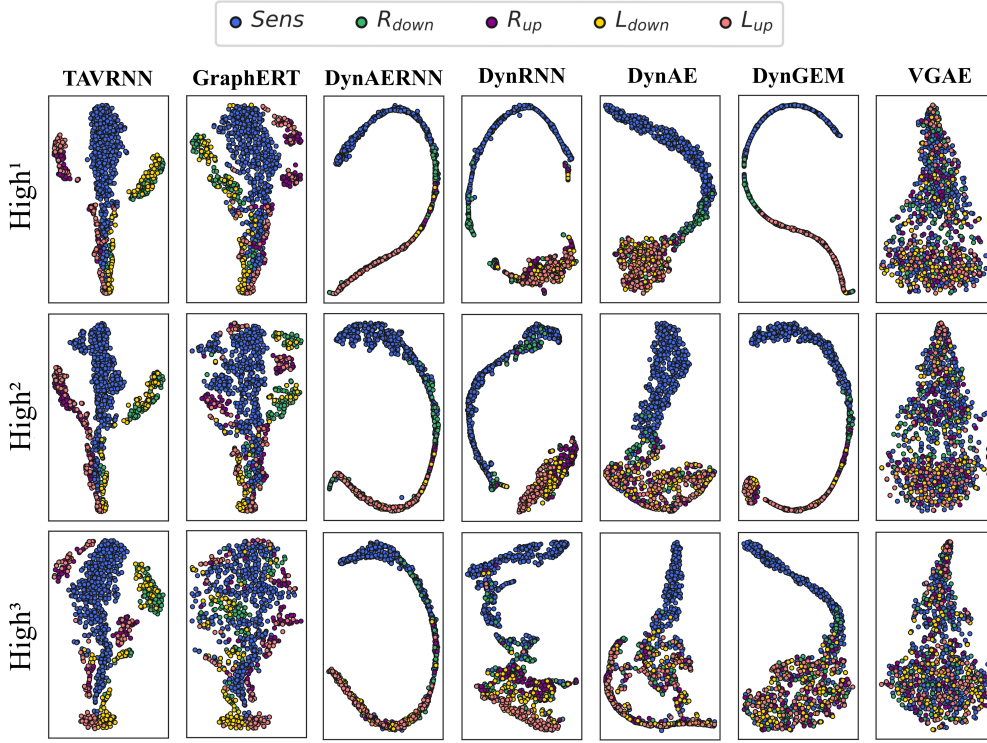


Figure S2: t-SNE visualization of the channels in the embedding space for $High^{1,2,3}$ windows of Gameplay using TAVRNN and all baseline methods for aggregated trials of an additional sample culture. Each channel is color-coded based on the predefined subregion it belongs to as shown in Fig. 1c.

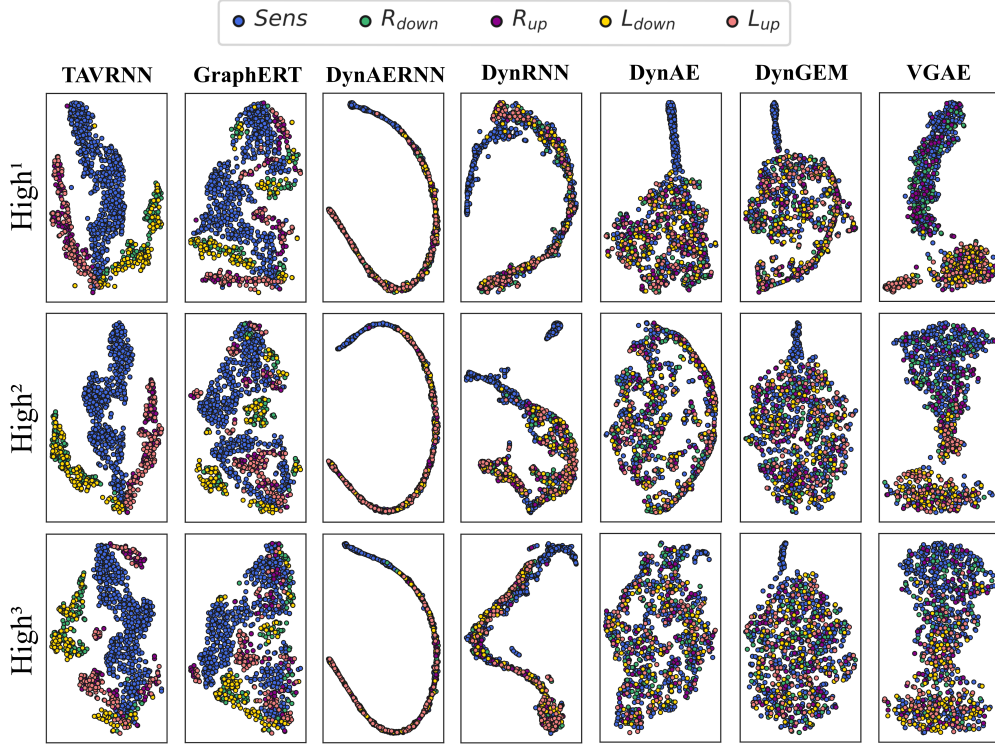


Figure S3: t-SNE visualization of the channels in the embedding space for $High^{1,2,3}$ windows of Gameplay using TAVRNN and all baseline methods for aggregated trials of another sample culture. Each channel is color-coded based on the predefined subregion it belongs to as shown in Fig. 1c.

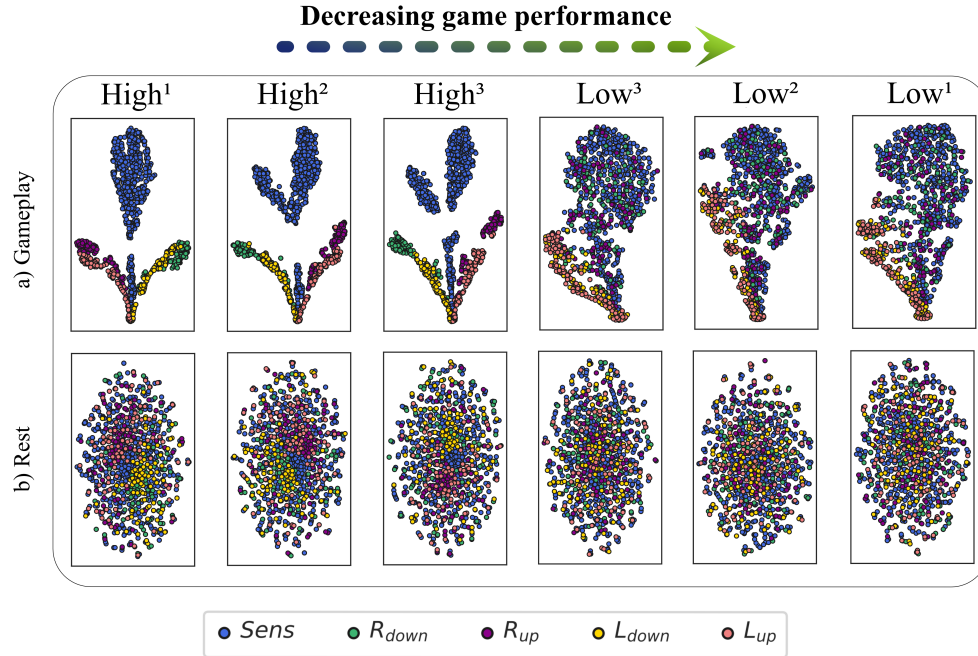


Figure S4: t-SNE visualization of the channels in the embedding space using TAVRNN during the top and bottom three windows ($High^{1,2,3}$ and $Low^{1,2,3}$) in terms of hit-miss-ratio during Gameplay and Rest for aggregated trials of a sample culture. Each channel is color-coded based on the predefined subregion it belongs to as shown in Fig. 1c.

A.5 CONNECTIVITY INFERENCE MECHANISMS

Methods for inferring connectivity are broadly categorized into two types: *model-free* and *model-based* approaches. Model-free methods rely on descriptive statistics and do not presuppose any specific underlying data generation mechanism, making them versatile for initial analyses. In contrast, model-based methods involve hypothesizing a mathematical model to elucidate the underlying biological processes by estimating its parameters and structure. Typically, these methods analyze time-series data, such as spike trains from individual neurons. However, recent advances have enabled studies to integrate spike inference with connectivity analysis directly from time-series data (35). In this work, we focus on utilizing the model-free methods.

Model-free methods do not presuppose any specific mechanisms underlying the observed data, offering a simpler alternative to model-based approaches. However, these methods do not facilitate the generation of activity data crucial for model validation or predictive analysis. Model-free techniques are primarily divided into two categories: those employing descriptive statistics such as Pearson correlation coefficient (PC) and cross-correlation (CC) and those utilizing information-theoretic measures such as Mutual information (MI), and Transfer entropy (TE) (35, 36, 37, 38, 39, 40, 41).

A.5.1 GRAPH KERNELS

In light of the diversity of connectivity inference methods discussed previously, each method can generate distinct graph representations from identical datasets. To extract meaningful insights from these varied representations, it is essential to employ a comparison methodology. However, graph comparison is computationally challenging. Ideally, one would verify if two graphs are exactly identical, a problem known as graph isomorphism, which is NP-complete (42). This complexity renders the task computationally prohibitive for large graphs.

To circumvent these difficulties, kernel methods offer a viable alternative. Kernels are functions designed to measure the similarity between pairs, enabling the transformation of objects into a high-dimensional space conducive to linear analysis methods. Graph kernels, specifically, facilitate the comparison of graphs by evaluating their structure, topology, and other attributes, thus proving instrumental in machine learning applications for graph data, such as clustering and classification (43, 44, 45).

Graph kernels vary in their approach to measuring similarity. Some rely on neighborhood aggregation, which consolidates information from adjacent nodes to form local feature vectors (46, 47, 48), while others utilize assignment and matching techniques to establish correspondences between nodes in different graphs (49). Additionally, some kernels identify and compare subgraph patterns (50), and others analyze walks and paths to capture structural nuances (51).

Here we concentrate on neighborhood aggregation methods, particularly pertinent for analyzing connectivity graphs derived from neuronal recordings, typically involving fewer than 1000 nodes without definitive node labels. These methods are also foundational for the graph neural network models. We exemplify this approach with the 1-dimensional *Weisfeiler-Lehman* (1-WL) algorithm (46), illustrating its application and effectiveness.

Weisfeiler-Lehman Algorithm The *Weisfeiler-Lehman* (WL) graph kernel is a sophisticated approach for computing graph similarities, which leverages an iterative relabeling scheme based on the *Weisfeiler-Lehman* isomorphism test. This method extends the basic graph kernel framework by incorporating local neighborhood information into the graph representation, making it particularly effective for graph classification tasks.

Consider a graph $G = (V, E, \ell)$, where V is the set of vertices, E is the set of edges, and $\ell : V \rightarrow \Sigma$ is a labeling function that maps each vertex to a label from a finite alphabet Σ . Initially, each vertex is assigned a label based on its original label or degree.

Define $\ell^0 = \ell$. At each iteration i , a new labeling ℓ^i is computed as follows:

$$\ell^{i+1}(v) = \text{HASH}(\ell^i(v), \{\!\!\{ \ell^i(u) \mid u \in N(v) \}\!\!\})$$

where $N(v)$ denotes the set of neighbors of vertex v and $\{\!\!\{ \cdot \}\!\!\}$ denotes a multiset, ensuring that the labels of neighboring vertices are considered without regard to their order. The function HASH maps the concatenated labels to a new, unique label. The algorithm continues iteratively, relabeling vertices until the labels converge or no new labels are produced (Fig. S5).

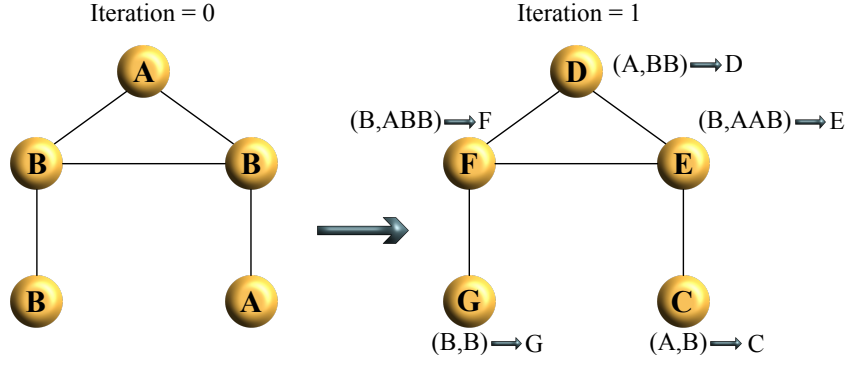


Figure S5: Illustration of the 1-dimensional *Weisfeiler-Lehman* (1-WL) algorithm. This diagram demonstrates how the 1-WL algorithm initially encounters overlapping node labels and, through one iteration, assigns unique labels to each node based on their positions within the graph.

After each iteration i , compute a feature vector $\phi^i(G)$ as the histogram of the labels across all vertices:

$$\phi^i(G) = (\#\{v \in V \mid \ell^i(v) = k\})_{k \in \mathcal{K}}$$

where \mathcal{K} is the set of all possible labels at iteration i .

The WL kernel between two graphs G and G' is defined as the sum of base kernel evaluations on the corresponding histograms at each iteration:

$$K(G, G') = \sum_{i=0}^h K_{\text{base}}(\phi^i(G), \phi^i(G'))$$

where K_{base} is typically chosen to be the linear kernel $K_{\text{base}}(\phi, \phi') = \phi \cdot \phi'$, and h is a predefined number of iterations, determining the depth of neighborhood aggregation.

In this study, we analyzed 437 recording sessions, comprising 262 Gameplay and 175 Rest sessions, to construct functional connectivity graphs. These graphs were derived using four distinct network inference algorithms: Zero-lag Pearson Correlations (PC), Cross-Correlation (CC), Mutual Information (MI), and Transfer Entropy (TE). For the PC analysis, connectivity matrices were thresholded at varying levels $t \in \{0, 20, 40, 60, 80\}\%$, retaining only the strongest connections as determined by their absolute correlation values. For both CC and TE, we explored delay values $d \in \{1, 2, 3, 4\}$. Each method produced 437 distinct networks.

Subsequently, a Weisfeiler-Lehman (WL) graph kernel with depth $h = 4$ was utilized to compute the kernel matrix \mathbf{K} , which was then employed in a Support Vector Machine (SVM) classifier to distinguish between Gameplay and Rest sessions. Classification effectiveness was evaluated through a 5-fold cross-validation on the *DishBrain* dataset, achieving the results summarized in Table S1. Notably, classification performance for CC and TE improved with increasing delay values, reflecting enhanced discriminative power of the graph kernels with longer embedding lengths. However, this increase in delay also introduced greater computational complexity, presenting challenges in scalability and traceability.

A.6 MARCHENKO-PASTUR DISTRIBUTION AND SHUFFLING PROCEDURE

In random matrix theory, the Marchenko-Pastur (MP) distribution describes the asymptotic behavior of the eigenvalues of large-dimensional sample covariance matrices. Consider a random matrix $\mathbf{A} \in \mathbb{R}^{p \times n}$, where p represents the number of variables (e.g., neurons or channels) and n represents the number of observations (e.g., time points). The sample covariance matrix is defined as:

$$\mathbf{C} = \frac{1}{n} \mathbf{A}^T \mathbf{A}$$

As both p and n grow large, while the ratio $\eta = \frac{p}{n}$ remains constant, the empirical distribution of the eigenvalues of \mathbf{C} converges to the Marchenko-Pastur distribution (28):

Table S1: Network inference method performance on *DishBrain* dataset

Network inference method	Avg. accuracy	Std. dev.
PC (t = 0%)	0.672	0.062
PC (t = 20%)	0.735	0.073
PC (t = 40%)	0.831	0.034
PC (t = 60%)	0.552	0.019
PC (t = 80%)	0.464	0.047
CC (d=1)	0.432	0.126
CC (d=2)	0.546	0.082
CC (d=3)	0.698	0.092
CC (d=4)	0.763	0.103
MI	0.722	0.057
TE (d=1)	0.657	0.073
TE (d=2)	0.688	0.112
TE (d=3)	0.731	0.028
TE (d=4)	0.794	0.063

$$\rho(\lambda) = \frac{\sqrt{(\lambda_+ - \lambda)(\lambda - \lambda_-)}}{2\pi\sigma^2\lambda\eta}$$

for $\lambda \in [\lambda_-, \lambda_+]$, where σ is the variance of the entries of matrix \mathbf{A} and:

$$\lambda_{\pm} = \sigma^2 (1 \pm \sqrt{\eta})^2$$

In the case where $\eta > 1$, which holds for our data (p is large relative to n), the MP distribution suggests that most of the eigenvalues will be close to zero. As a result, the sample covariance matrix is likely to be ill-conditioned, and hence unreliable for further analysis.

A.6.1 SHUFFLING PROCEDURE FOR CORRELATION ANALYSIS

To account for potential spurious correlations due to ill-conditioning of the sample covariance matrix, we perform a shuffling control procedure:

1. **Shuffle Time Points:** The time points of each channel are independently shuffled while maintaining the channel identity. This process destroys any temporal correlation, ensuring that the correlation between channels is not influenced by the original time structure.
2. **Multiple Iterations:** The shuffling procedure is repeated multiple times (e.g., we chose 1000 iterations) to build a null distribution of correlations for each pair of channels.
3. **Confidence Intervals:** Based on the null distribution obtained from the shuffled data, we compute confidence intervals for each pair of channels. Correlation values from the original data that lie outside of the 95% confidence interval are considered statistically significant.

This approach provides a robust method for identifying significant correlations in the presence of potential ill-conditioning of the sample covariance matrix.

A.7 UNSUPERVISED SEQUENTIAL VFE (sVFE) LOSS

In a Variational Graph Auto Encoder (VGAE), an encoder network is responsible for learning the latent embeddings $\{\mathbf{Z}_t\}_{t=0}^T$, which capture the representation of nodes in a reduced-dimensional space. The probability of an edge between nodes i and j in the reconstructed graph is determined by the inner product of their respective latent embeddings, $\mathbf{Z}_{t,i}$ and $\mathbf{Z}_{t,j}$. This process is usually accompanied by a sigmoid activation function to constrain the output values between 0 and 1:

$$\hat{a}_{t,ij} = \sigma(\mathbf{Z}_{t,i} \cdot \mathbf{Z}_{t,j}^T). \quad (\text{S1})$$

In this context, σ represents the sigmoid function, $\mathbf{Z}_{t,i}$ refers to the i th row of the matrix \mathbf{Z}_t , and $\hat{a}_{t,ij}$ corresponds to the (i, j) th element of the matrix $\hat{\mathbf{A}}_t$, indicating the predicted probability of an edge between nodes i and j at time t .

Considering that $\hat{a}_{t,ij}$ indicates the probability of an edge, the likelihood of the observed adjacency matrix \mathbf{A}_t based on the embeddings can be independently modeled for each edge using a Bernoulli distribution:

$$p_{\theta}(\mathbf{A}_t | \mathbf{Z}_{\leq t}, \mathbf{X}_{< t}, \mathbf{A}_{< t}) = \prod_{i,j=1}^N \hat{a}_{t,ij}^{a_{t,ij}} (1 - \hat{a}_{t,ij})^{1-a_{t,ij}}. \quad (\text{S2})$$

In this case, $a_{t,ij}$ represents the actual entry in the adjacency matrix \mathbf{A}_t , signifying the presence, absence, or weight (for weighted graphs) of an edge between nodes i and j .

The log-likelihood of the adjacency matrix, $\log p_{\theta}(\mathbf{A}_t | \mathbf{Z}_{\leq t}, \mathbf{X}_{< t}, \mathbf{A}_{< t})$, can be expressed as the negative binary cross entropy (BCE):

$$\mathcal{L}^{\text{BCE}}(\theta, \phi) = \sum_{i,j=1}^N \left[a_{t,ij} \log \hat{a}_{t,ij} + (1 - a_{t,ij}) \log(1 - \hat{a}_{t,ij}) \right]. \quad (\text{S3})$$

We approximate the first expectation term in the sequential VFE (sVFE) using Monte Carlo integration as follows:

$$\mathbb{E}_{q_{\phi}(z_t | x_{\leq t})} [\log p_{\theta}(\mathbf{A}_t | \mathbf{Z}_{\leq t}, \mathbf{X}_{< t}, \mathbf{A}_{< t})] = \frac{1}{M} \sum_{k=1}^M \mathcal{L}^{\text{BCE}}(\mathbf{Z}_t^k). \quad (\text{S4})$$

Here, k represents the particle index, and M refers to the number of particles, which may be set to 1 when the mini-batch size is sufficiently large (52).

Latent particles \mathbf{Z}_t^k are sampled from $q_{\phi}(\mathbf{Z}_t | \mathbf{X}_{\leq t}, \mathbf{A}_{\leq t}, \mathbf{Z}_{< t})$ as described by Eq. (7b), utilizing the reparameterization trick $\mathbf{Z}_t^k = \mu_t^{\text{enc}} + \sigma_t^{\text{enc}} \odot \epsilon_t^k$, where ϵ_t^k is drawn from $\mathcal{N}(0, I)$ and \odot represents the Hadamard (element-wise) product. Recurrent state particles \mathbf{H}_t^k are derived using Eq. (9), based on \mathbf{Z}_{t-1}^k and the previous time-step's state \mathbf{H}_{t-1}^k .

Additionally, an analytical solution for the Kullback-Leibler divergence D_{KL} in the sequential VFE Eq. (4) can be derived in closed form as:

$$D_{\text{KL}}(\theta, \phi) = \frac{1}{2} \sum_{i,j=1}^{N,D} \left[\frac{\sigma_{t,ij}^{\text{enc}2}}{\sigma_{t,ij}^{\text{prior}2}} - \log \frac{\sigma_{t,ij}^{\text{enc}2}}{\sigma_{t,ij}^{\text{prior}2}} + \frac{(\mu_{t,ij}^{\text{enc}} - \mu_{t,ij}^{\text{prior}})^2}{\sigma_{t,ij}^{\text{prior}2}} - 1 \right] \quad (\text{S5})$$

This KLD loss is deterministic, thereby eliminating the need for Monte Carlo approximation. It quantifies the statistical distance between the conditional prior as specified in Eq. (7a) and the approximate posterior in Eq. (7b). Optimizing this measure strengthens the causality within the latent space, as the prior Eq. (8a) focuses on the influence of preceding graphs and embeddings $\{\mathbf{X} < t, \mathbf{A} < t, \mathbf{Z} < t\}$.

By integrating Eq. (S4) and Eq. (S5) into Eq. (4), we formulate an unsupervised sVFE loss that forms the foundation of the proposed TAVRNN framework:

$$\begin{aligned}
\mathcal{L}^{\text{TAVRNN}}(\theta, \phi) &= \mathcal{L}^{\text{BCE}}(\theta, \phi) + \mathcal{D}^{\text{KL}}(\theta, \phi) \\
&= \underbrace{\frac{1}{M} \sum_{t=0}^T \sum_{k=1}^M \sum_{i,j=1}^N \left[a_{t,ij} \log \sigma \left(\mathbf{Z}_t^k \times \mathbf{Z}_t^{k^T} \right) + (1 - a_{t,ij}) \log \left(1 - \sigma \left(\mathbf{Z}_t^k \times \mathbf{Z}_t^{k^T} \right) \right) \right]}_{\mathcal{L}^{\text{BCE}}(\theta, \phi)} \\
&\quad + \underbrace{\frac{1}{2} \sum_{t=0}^T \sum_{i,j=1}^N \left[\frac{(\sigma_{t,ij}^{\text{enc}} + \epsilon)^2}{(\sigma_{t,ij}^{\text{prior}} + \epsilon)^2} - \log \frac{(\sigma_{t,ij}^{\text{enc}} + \epsilon)^2}{(\sigma_{t,ij}^{\text{prior}} + \epsilon)^2} + \frac{(\mu_{t,ij}^{\text{enc}} - \mu_{t,ij}^{\text{prior}})^2}{(\sigma_{t,ij}^{\text{prior}} + \epsilon)^2} - 1 \right]}_{\mathcal{D}^{\text{KL}}(\theta, \phi)}.
\end{aligned} \tag{S6}$$

A.8 TEMPORAL ATTENTION MECHANISM

The goal of this section is to present the mathematical details of the temporal attention mechanism for computing \mathbf{H}_t for $\hat{\mathbf{H}}_t$ and $\mathbf{H}_{t-1}, \mathbf{H}_{t-2}, \dots, \mathbf{H}_{t-w}$. Let the d_h dimensional row vector \bar{s}_i present the global state of the graph at time step i .¹ Also let $\bar{\mathbf{S}}$ be a $(w+1) \times (w+1)$ matrix that its i -th row is equal to $\bar{s}_{t-w-1+i}$. We compute the query vector q and the key matrix K as follows:

$$q = \bar{s}_t \times \mathbf{W}_q + b_q \tag{S7}$$

$$\mathbf{K} = \bar{\mathbf{S}} \times \mathbf{W}_k + b_k \tag{S8}$$

Here, the $d_h \times d_k$ matrices \mathbf{W}_q and \mathbf{W}_k , and also the d_k dimensional row vectors b_q and b_k are learnable parameters of our model. Then, the attention vector α , which is a $w+1$ dimensional row vector, will be defined as:

$$\alpha = \text{softmax} \left(\frac{q \times K^T}{\sqrt{d_k}} \right). \tag{S9}$$

Let us define the value matrices as follows:

$$\mathbf{V}_i = \mathbf{H}_{t-w-1+i} \times \mathbf{W}_v + b_v \quad \forall 1 \leq i \leq w, \tag{S10}$$

and

$$\mathbf{V}_{w+1} = \hat{\mathbf{H}}_t \times \mathbf{W}_v + b_v, \tag{S11}$$

where the $d_h \times d_h$ matrix \mathbf{W}_v and the d_h dimensional row vector b_v are the other learnable parameters of our model.

Finally, the state matrix \mathbf{H}_t will be computed as follows:

$$\mathbf{H}_t = \sum_{i=1}^w \alpha_i \times \mathbf{V}_i. \tag{S12}$$

A.9 TAVRNN MODEL TRAINING HYPERPARAMETERS

All the experiments were run on a 2.3 GHz Quad-Core Intel Core i5. PyTorch 1.8.1 was used to build neural network blocks.

We configured our TAVRNN model by employing graph-structured GRU-Attention with a single recurrent hidden layer consisting of 32 units. The window size w in the attention mechanism is set to the maximum possible for every time step, allowing the model to attend to all previous time steps, including the very first one. The functions φ_θ^x and φ_θ^z in Eqs. (8b) and (9) are implemented using a 32-dimensional fully-connected layer. For the function $\varphi_\theta^{\text{prior}}$ in Eq. (8a), we use two 32 and 8 dimensional fully-connected layers. To model μ_t^{enc} and Σ_t^{enc} we employ a 2-layer GCN with 32 and 8 layers, respectively. Our model is initialized using Glorot initialization (53). The learning rate for training is set to 0.01. Training is performed over 1000 epochs using the Adam SGD optimizer (54).

The implementation of our proposed model is available at the following [Github Repository](#).

¹For $i < t$, \bar{s}_i is equal to that row of \mathbf{H}_i which corresponds to the hypothetical node that is connected to all other nodes. Also, \bar{s}_t is equal to the corresponding row of $\hat{\mathbf{H}}_t$.

A.10 TIME COMPLEXITY ANALYSIS

In this section, we will compute the time complexity for each method. This analysis provides insights into the computational cost and efficiency of different methods for representation learning of temporal graph data. More specifically, we compute the time complexity of a forward pass on the entire set of the graph nodes in one snapshot for each method.

A.10.1 GRAPHERT:

GraphERT is a Transformer-based model for temporal graphs. It uses multiple random walks with different transition parameters p and q to capture the neighborhood structure around each node at specific time steps. These random walks are fed into a Transformer, which learns node-to-node interactions and their temporal relevance using multi-head attention.

Random Walks Generation:

For each graph snapshot, the algorithm generates $\gamma \times n \times |p| \times |q|$ random walks, where:

- γ is the number of random walks starting from each node for each pair of values assigned to p and q .
- n is the number of nodes in the graph.
- $|p|$ and $|q|$ are the number of different values for the hyperparameters p and q .

The time complexity for generating the random walks is:

$$\mathcal{O}(\gamma \times n \times |p| \times |q| \times L)$$

where L is the length of each random walk.

Transformer Processing:

Each random walk is processed by the Transformer. The time complexity of the Transformer is dominated by the self-attention mechanism, which scales quadratically with the sequence length and linearly with the number of attention heads.

For each random walk, the time complexity is:

$$\mathcal{O}(L^2 \times h_{\max} \times H \times k)$$

where:

- L is the random walk length.
- h_{\max} is the maximum dimensionality of the representation vectors used in different transformer layers. In the original implementation of GraphERT we have $h_{\max} = d$, but in general it can take any value larger than or equal to d .
- H is the number of attention heads.
- k is the number of layers in the Transformer.

Total Time Complexity:

The total number of random walks is $\gamma \times n \times |p| \times |q|$. Combining the time complexity for random walk generation and Transformer processing, the total time complexity for processing a single graph snapshot is:

$$\mathcal{O}(n \cdot \gamma \cdot |p| \cdot |q| \cdot (L + L^2 \cdot h_{\max} \cdot H \cdot k)) \in \mathcal{O}(n \cdot \gamma \cdot |p| \cdot |q| \cdot (L^2 \cdot h_{\max} \cdot H \cdot k))$$

We can assume that γ , $|p|$, q , H and k are constant values, because they can be fixed values, independent of the graph size (n) and the intended dimensionality of the final representations (d).

Therefore, we can simplify the total complexity as follows:

$$\mathcal{O}((\gamma \cdot |p| \cdot |q| \cdot H \cdot k) \cdot n \cdot L^2 \cdot h_{\max}) \in \mathcal{O}(n \cdot L^2 \cdot h_{\max})$$

However, it is worth noting that the constant value of this running time is large enough to make practical issues in real experiments. That is why GraphERT shows the most time complexity in Figure 3. Look at Table S2 for more details about the used values for the hyperparameters of this method.

Method	Hyperparameter	Description / Value
GraphERT	p (Return parameter)	Bias for random walks to return to previous node $\in [0.25, 0.5, 1, 2, 4]$
	q (In-out parameter)	Bias for random walks to explore outward $\in [0.25, 0.5, 1, 2, 4]$
	Random Walk Length (L)	Length of each random walk (32)
	Number of Random Walks (γ)	Number of random walks per node (10)
	Embedding Dimension (d)	Size of node embeddings (8)
	Attention Heads (H)	Number of attention heads (4)
	Transformer Layers (k)	Number of Transformer layers (6)
	Learning Rate	Learning rate for the Adam optimizer (1e-4)

Table S2: Hyperparameters for GraphERT

A.10.2 VGAE:

To compute the time complexity of a Variational Graph Autoencoder (VGAE) with n nodes, e edges, k Graph Convolutional Network (GCN) layers, and hidden dimensions h_1, h_2, \dots, h_k , where the final latent representation dimension is d , we need to analyze the time complexity at each layer of the GCN. This will account for both node-wise and edge-wise operations.

Step 1: GCN Layer Operations

A GCN layer applies a linear transformation followed by neighborhood aggregation. The complexity of a single GCN layer is typically determined by:

- **Node-wise operations:** These involve multiplying the node features by a weight matrix. This has a time complexity of $\mathcal{O}(n \cdot h_{\text{in}} \cdot h_{\text{out}})$, where h_{in} is the input dimension of the layer and h_{out} is the output dimension.
- **Edge-wise operations:** These involve aggregating the features of neighboring nodes through a message-passing operation over edges. This has a time complexity of $\mathcal{O}(e \cdot h_{\text{out}})$.

Step 2: Time Complexity of Each GCN Layer

For the i -th GCN layer:

- Let the input feature dimension be h_{i-1} and the output feature dimension be h_i .
- Node-wise multiplication has complexity $\mathcal{O}(n \cdot h_{i-1} \cdot h_i)$.
- Edge-wise aggregation has complexity $\mathcal{O}(e \cdot h_i)$.

Thus, the total time complexity of the i -th layer is:

$$\mathcal{O}(n \cdot h_{i-1} \cdot h_i + e \cdot h_i)$$

Step 3: Summing Over All GCN Layers

We have k GCN layers with dimensions h_0, h_1, \dots, h_k , where $h_0 = n$ is the input feature dimension and $h_k = d$ is the output dimension. Therefore, the total time complexity for all layers is:

$$T_{\text{GCN}} = \sum_{i=1}^k (\mathcal{O}(n \cdot h_{i-1} \cdot h_i + e \cdot h_i))$$

Step 4: VGAE Encoder and Decoder

- **Encoder:** The encoder, which maps node features to a latent representation space (mean and variance for the latent variables), has the same complexity as the GCN layers, so its complexity is T_{GCN} .
- **Decoder:** In VGAE, the decoder typically involves reconstructing the adjacency matrix from the latent space. The reconstruction (e.g., using a dot product between latent vectors) has a time complexity of $\mathcal{O}(n^2 \cdot d)$, as it involves calculating pairwise similarities between all node pairs.

Step 5: Total Time Complexity of VGAE

Summing up the time complexity of the GCN-based encoder and the decoder, we get the overall time complexity:

$$T_{\text{VGAE}} = T_{\text{GCN}} + \mathcal{O}(n^2 \cdot d)$$

This expands to:

$$T_{\text{VGAE}} = \sum_{i=1}^k (\mathcal{O}(n \cdot h_{i-1} \cdot h_i + e \cdot h_i)) + \mathcal{O}(n^2 \cdot d)$$

Conclusion

Let us denote $\max_{i=1}^k h_i$ by h_{\max} . We know that $n = h_0 \geq h_1 \geq \dots \geq h_k = d$. So, $h_{\max} = h_1$ and the time complexity of the VGAE is:

$$T_{\text{VGAE}} = \mathcal{O} \left(\sum_{i=1}^k (n \cdot h_{i-1} \cdot h_i + e \cdot h_i) + n^2 \cdot d \right) \in \mathcal{O}(n^2 \cdot h_{\max})$$

$s.t. \ h_0 = n, h_k = d$

This reflects the complexities of both the encoder (GCN layers) and the decoder (adjacency matrix reconstruction). The most significant term depends on the number of nodes, and the dimensions of the latent space. Hyperparameters of the VGAE model and the values assigned to them in the original paper are listed in Table [S2](#).

Method	Hyperparameter	Description / Value
VGAE	Latent Dimension (d)	Size of the latent space (dimension of node embeddings) (8)
	Graph Convolutional Layers (GCN)	Number of convolution layers to capture graph structure (2 layers)
	Learning Rate	Learning rate for the Adam optimizer (1e-2)
	Hidden Dimension (h)	Number of hidden units in the encoder GCN layers (32)

Table S3: Hyperparameters for Variational Graph Autoencoder (VGAE)

A.10.3 DYN GEM:

DynGEM uses a Multi-Layer Perceptron (MLP) autoencoder to generate low-dimensional embeddings for dynamic graphs at each snapshot. At time step $t = 1$, the model is trained on the first snapshot of the graph using a randomly initialized deep autoencoder. For subsequent time steps, embeddings and network parameters are initialized from the previous time step.

Given n nodes, k hidden layers with sizes h_1, h_2, \dots, h_k , and the latent representation dimension d , the time complexity of processing the input graph for each snapshot is:

$$\mathcal{O}(n \cdot (n \cdot h_1 + h_1 \cdot h_2 + \dots + h_{k-1} \cdot h_k + h_k \cdot d))$$

Conclusion

Let us denote $\max_{i=1}^{k+1} h_i$ by h_{\max} . We know that $n = h_0 \geq h_1 \geq \dots \geq h_{k+1} = d$. So, $h_{\max} = h_1$ and the time complexity of the DynGEM is:

$$T_{\text{DynGem}} = \mathcal{O}\left(\sum_{i=1}^{k+1} (n \cdot h_{i-1} \cdot h_i)\right) \in \mathcal{O}(n^2 \cdot h_{\max})$$

$s.t. h_0 = n, h_{k+1} = d$

Hyperparameters of this method and the assigned values to them can be found in Table [S4](#).

Method	Hyperparameter	Description / Value
DynGEM	Latent Dimension (d)	Size of the latent space (dimension of node embeddings) (8)
	Number of layers in the encoder/decoder	Autoencoder has 3 layers
	Layer Sizes (h_1, h_2)	Size of each layer in the autoencoder (500,300)
	L1 regularization coefficient (ν_1)	Encourages sparsity in the model's weights ($1e - 6$)
	L2 regularization coefficient (ν_2)	Encouraging weight values to remain small ($1e - 6$)
	Learning Rate	Learning rate ($1e - 4$)
	Reconstruction Loss Weight (β)	Weight for adjacency matrix reconstruction (5)

Table S4: Hyperparameters for DynGEM

A.10.4 DYNAE:

DynAE extends a static MLP autoencoder to handle temporal graphs. It uses l look-back adjacency matrices from past snapshots and feeds them into a deep autoencoder to reconstruct the current graph based on previous graphs.

Given an input size of $n \cdot l$ (where n is the number of nodes and l is the number of look-back snapshots), and k layers in the autoencoder, with the latent representation dimension d , the time complexity for the encoder is:

$$\mathcal{O}(n \cdot (n \cdot l \cdot h_1 + h_1 \cdot h_2 + \dots + h_k \cdot d))$$

Conclusion

Let us denote $\max_{i=1}^{k+1} h_i$ by h_{\max} . We know that $n \cdot l = h_0 \geq h_1 \geq \dots \geq h_{k+1} = d$. So, $h_{\max} = h_1$. In addition, l can be considered as a constant number, and the time complexity of the DynAE is:

$$T_{\text{DynAE}} = \mathcal{O} \left(\sum_{i=1}^{k+1} (n \cdot h_{i-1} \cdot h_i) \right) \in \mathcal{O}(n^2 \cdot h_{\max})$$

$$s.t. \ h_0 = n \cdot l, h_{k+1} = d$$

Hyperparameters of this method and the assigned values to them can be found in Table S5.

Method	Hyperparameter	Description / Value
DynAE	Look-back (l)	Number of previous snapshots used (2)
	Latent Dimension (d)	Size of the latent space (dimension of node embeddings) (8)
	Number of layers in the encoder/decoder	Autoencoder has 3 layers
	Layer Sizes (h_1, h_2)	Size of each autoencoder layer (500,300)
	L1 regularization coefficient (ν_1)	Encourages sparsity in the model's weights ($1e - 6$)
	L2 regularization coefficient (ν_2)	Encouraging weight values to remain small ($1e - 6$)
	Learning Rate	Learning rate ($1e - 4$)
	Reconstruction Loss Weight (β)	Weight for adjacency matrix reconstruction (5)

Table S5: Hyperparameters for DynAE

A.10.5 DYNRNN:

DynRNN is similar to DynAE, but it uses Recurrent Neural Networks (RNNs), specifically Long Short-Term Memory (LSTM) networks, to capture temporal dependencies across snapshots. Each node's neighborhood at each snapshot is passed into the LSTM.

The time complexity for LSTM step i on one node is:

$$\mathcal{O}(h_{i-1_{LSTM}} \cdot h_{i_{LSTM}} + h_{i_{LSTM}}^2)$$

Given n nodes, k_{LSTM} LSTM layers with sizes $h_{1_{LSTM}}, h_{2_{LSTM}}, \dots, h_{k_{LSTM}}$ and l snapshots, the total time complexity for one snapshot is:

$$\mathcal{O}(n \cdot (n \cdot l \cdot h_{1_{LSTM}} + h_{1_{LSTM}} \cdot h_{2_{LSTM}} + \dots + h_{k-1_{LSTM}} \cdot h_{k_{LSTM}} + h_{k_{LSTM}} \cdot d + h_{1_{LSTM}}^2 + \dots + h_{k_{LSTM}}^2 + d^2))$$

Conclusion

Let us denote $\max_{i=1}^{k+1} h_{i_{LSTM}}$ by h_{\max} . We know that $n \cdot l = h_{0_{LSTM}} \geq h_{1_{LSTM}} \geq \dots \geq h_{k+1_{LSTM}} = d$. So, $h_{\max} = h_{1_{LSTM}}$. In addition, l can be considered as a constant number, the time complexity of the DynRNN is:

$$T_{\text{DynRNN}} = \mathcal{O} \left(\sum_{i=1}^{k+1} (n \cdot (h_{i-1_{LSTM}} \cdot h_{i_{LSTM}} + h_{i_{LSTM}}^2)) \right) \in \mathcal{O}(n^2 \cdot h_{\max})$$

$$s.t. \ h_{0_{LSTM}} = n \cdot l, h_{k+1_{LSTM}} = d$$

Hyperparameters of this method and the assigned values to them can be found in Table S6.

Method	Hyperparameter	Description / Value
DynRNN	Look-back (l)	Number of previous snapshots used (2)
	Latent Dimension (d)	Size of the latent space (dimension of node embeddings) (8)
	Number of RNN Layers	Number of stacked LSTM layers (3)
	Hidden State Size	Number of hidden units in LSTM (500,300)
	L1 regularization coefficient (ν_1)	Encourages sparsity in the model's weights ($1e - 6$)
	L2 regularization coefficient (ν_2)	Encouraging weight values to remain small ($1e - 6$)
	Learning Rate	Learning rate ($1e - 4$)
	Reconstruction Loss Weight (β)	Weight for adjacency matrix reconstruction (5)

Table S6: Hyperparameters for DynRNN

A.10.6 DYNAERNN:

DynaERNN combines the autoencoder from DynAE with the LSTM-based RNN from DynRNN. The encoder compresses the neighborhood vectors of l snapshots into a low-dimensional space, which the LSTM processes across time to capture temporal dependencies.

The total time complexity for DynAERNN is the sum of the autoencoder and LSTM complexities:

$$\mathcal{O}(n \cdot (n \cdot l \cdot h_1 + h_1 \cdot h_2 + \dots + h_{k-1} \cdot h_k) + \mathcal{O}(n \cdot (h_k \cdot h_{1_{LSTM}} + h_{1_{LSTM}} \cdot h_{2_{LSTM}} + \dots + h_{k-1_{LSTM}} \cdot h_{k_{LSTM}} + h_{k_{LSTM}} \cdot d + h_{1_{LSTM}}^2 + \dots + h_{k_{LSTM}}^2 + d^2)))$$

Conclusion

Let us denote $\max(\max_{i=1}^k h_i, \max_{i=1}^{k+1} h_{i_{LSTM}})$ by h_{\max} . We know that $n \cdot l = h_0 \geq h_1 \geq \dots \geq h_k = h_{0_{LSTM}} \geq h_{1_{LSTM}} \geq \dots \geq h_{k+1_{LSTM}} = d$. So, $h_{\max} = h_1$. In addition, l can be considered as a constant number time complexity of the DynRNN is:

$$T_{\text{DynAERNN}} = \mathcal{O} \left(\sum_{i=1}^k (n \cdot h_{i-1} \cdot h_i) + \sum_{i=1}^{k+1} (n \cdot (h_{i-1_{LSTM}} \cdot h_{i_{LSTM}} + h_{i_{LSTM}}^2)) \right) \in \mathcal{O}(n^2 \cdot h)$$

s.t. $h_0 = n \cdot l, h_{0_{LSTM}} = h_k, h_{k+1_{LSTM}} = d$

Hyperparameters of this method and the assigned values to them can be found in Table [S7](#).

A.10.7 TAVRNN:

The time complexity of the TAVRNN framework is driven by several components, including GNN layers, GRU operations, and an attention mechanism. Below, we break down the total complexity into the time complexity of each component.

1. GNN and GRU Layers:

At each time step t , the model processes the graph using a combination of GNN layers and a GRU-based RNN. The time complexity for these operations can be broken down as follows:

- **Low-dimensional Embedding:** first of all, each n -dimensional neighborhood vector is mapped to a h_{GRU} -dimensional embedding using a one layer feed forward network. The time complexity of this part will be:

$$\mathcal{O}(n^2 \cdot h_{GPU})$$

Method	Hyperparameter	Description / Value
DynAERNN	Look-back (l)	Number of previous snapshots used (2)
	Latent Dimension (d)	Size of the latent space (dimension of node embeddings) (8)
	Autoencoder Layer Sizes	Size of each autoencoder layer (500,300)
	Number of RNN Layers	Number of stacked LSTM layers (3)
	LSTM Hidden State Size	Number of hidden units in LSTM (500,300)
	L1 regularization coefficient (ν_1)	Encourages sparsity in the model's weights ($1e - 6$)
	L2 regularization coefficient (ν_2)	Encouraging weight values to remain small ($1e - 6$)
	Learning Rate	Learning rate ($1e - 4$)
	Reconstruction Loss Weight (β)	Weight for adjacency matrix reconstruction (5)

Table S7: Hyperparameters for DynAERNN

- **Graph Convolution (GNN):** Similar to the VGAE mentioned above, the time complexity of the GNN layer is:

$$T_{\text{GNN}} = \sum_{i=1}^k (\mathcal{O}(n \cdot h_{i-1} \cdot h_i + e \cdot h_i))$$

- **GRU Operation:** Since the inner functions of our GPU cell is implemented by GCN layers, the dominant term in the time complexity of the GPU cell in each time step is equal to:

$$\mathcal{O}(n \cdot h_{GRU}^2 + e \cdot h_{GRU})$$

2. Temporal Attention Mechanism:

The attention mechanism aggregates past hidden states over a window of size w . The attention of the model into the last w snapshots is computed in:

$$\mathcal{O}(w \cdot h)$$

where w is the attention window size and h is the hidden dimension. The time complexity of computing the weighted average vectors for all the n node according to these computed attentions is:

$$\mathcal{O}(n \cdot w \cdot h)$$

3. Reconstruction: Similar to VGAE, the reconstruction process in TAVRNN is through computing the inner product of the final representation of each pair of the nodes, and its time complexity is:

$$\mathcal{O}(n^2 \cdot d)$$

4. Overall Time Complexity for Each Time Step:

The overall time complexity at each time step is a combination of the initial projection to a low-dimensional space using a feedforward layer, GNN and GRU computations, attention mechanism, and reconstruction:

$$\mathcal{O}(n \cdot (h_1 + h_1 \cdot h_2 + \dots + h_k \cdot d) + e \cdot (h_1 + \dots + h_k) + n \cdot h_{GRU}^2 + e \cdot h_{GRU} + (n+1) \cdot w \cdot h + n^2 \cdot d)$$

Conclusion

Let us denote $\max(\max_{i=1}^{k+1} h_i, h_{GRU}, h)$ by h_{\max} . We know that $n \cdot l = h_0 \geq h_1 \geq \dots \geq h_k + 1 = d$. So, $h_{\max} = h_1$. We can infer that the time complexity of TAVRNN is:

$$T_{\text{TAVRNN}} = \mathcal{O}\left(\sum_{i=1}^{k+1}(n \cdot h_{i-1} \cdot h_i + e \cdot h_i) + n \cdot h_{GRU}^2 + e \cdot h_{GRU} + n \cdot w \cdot h + n^2 \cdot d\right) \in \mathcal{O}(n^2 \cdot h_{\max} + n \cdot w \cdot h)$$

$$s.t. h_0 = 1, h_{k+1} = d$$

The summary of the time complexities for different methods is shown in Table S8.

Table S8: One forward pass time complexity for one time window (i.e. snapshot).

Method	Complexity
VGAE	$\mathcal{O}\left(\sum_{i=1}^k(n \cdot h_{i-1} \cdot h_i + e \cdot h_i) + n^2 \cdot d\right) \in \mathcal{O}(n^2 \cdot h_{\max})$
DynGEM	$\mathcal{O}\left(\sum_{i=1}^{k+1}(n \cdot h_{i-1} \cdot h_i)\right) \in \mathcal{O}(n^2 \cdot h_{\max})$
DynAE	$\mathcal{O}\left(\sum_{i=1}^{k+1}(n \cdot h_{i-1} \cdot h_i)\right) \in \mathcal{O}(n^2 \cdot h_{\max})$
DynRNN	$\mathcal{O}\left(\sum_{i=1}^{k+1}(n \cdot (h_{i-1\text{LSTM}} \cdot h_{i\text{LSTM}} + h_{i\text{LSTM}}^2))\right) \in \mathcal{O}(n^2 \cdot h_{\max})$
DynAERNN	$\mathcal{O}\left(\sum_{i=1}^k(n \cdot h_{i-1} \cdot h_i) + \sum_{i=1}^{k+1}(n \cdot (h_{i-1\text{LSTM}} \cdot h_{i\text{LSTM}} + h_{i\text{LSTM}}^2))\right) \in \mathcal{O}(n^2 \cdot h_{\max})$
GraphERT	$\mathcal{O}((\gamma \cdot p \cdot q \cdot H \cdot k) \cdot n \cdot L^2 \cdot h_{\max}) \in \mathcal{O}(n \cdot L^2 \cdot h_{\max})$
TAVRNN	$\mathcal{O}\left(\sum_{i=1}^{k+1}(n \cdot h_{i-1} \cdot h_i + e \cdot h_i) + n \cdot h_{GRU}^2 + e \cdot h_{GRU} + n \cdot w \cdot h + n^2 \cdot d\right) \in \mathcal{O}(n^2 \cdot h_{\max} + n \cdot w \cdot h_{\max})$