Università
Ca'Foscari
Venezia

Master's Degree in
Global Development
& Entrepreneurship

Final Thesis

International Trade Modelling with Recurrent Graph Neural Networks

Supervisor
Ch. Prof. Massimo Warglien

Graduand
Claudio
Casellato
861553

Academic Year
2020 / 2021

TABLE OF CONTENTS

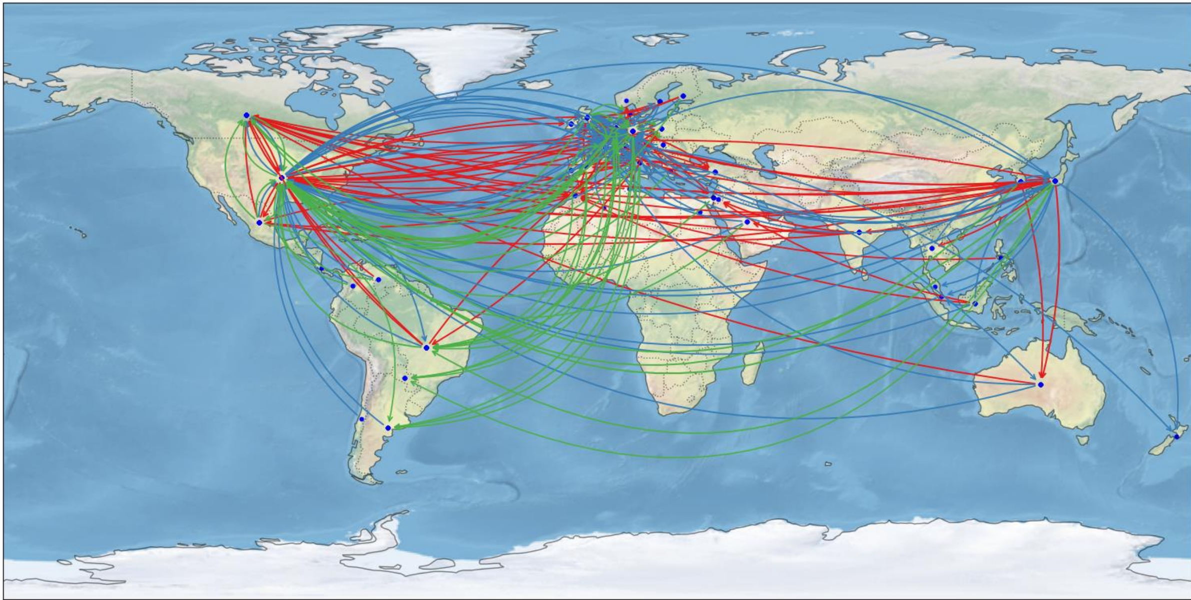
Table of contents 2
Abstract 4
0 Introduction 5
1 Introduction to Graphs 9
1.1 Undirected and Directed Graphs 9
1.2 Stochastic Graphs 10
1.3 Dynamic Graphs 11
2 Explorative Analysis: Network Measures 12
2.1 Node Degree 12
2.2 Closeness Centrality 13
2.3 Link Density 14
2.4 Jaccard Similarity 14
3 Static Graph Encoder 15
3.1 Artificial Neural Network 15
3.1.1 Sigmoid Activation 17
3.1.2 Relu Activation 17
3.1.3 Softplus Activation 18
3.1.4 Leaky-Relu Activation 18
3.1.5 Tanh Activation 18
3.2 Autoencoder Model 19
3.3 Training a neural network 20
3.4 Graph Neural Networks 20
3.4.1 Graph Attention Network 21
4 Static Graph Decoder 24
5 Recurrent temporal encoder 25
5.1.1 Recurrent Neural Network 26
5.1.2 Gated Recurrent Unit 28
5.1.3 Training with Back Propagation Trough Time (BPTT) 29
6 Loss Function 30
7 Aleatoric and Epistemic Uncertainty estimation 32
7.1 Aleatoric Uncertainty 32
7.2 Epistemic Uncertainty with Dropout 33

8 Model Evaluation 37
8.1 Regression 38
8.1.1 Mean Absolute Error 38
8.1.2 Root Mean Squared Error 38
8.1.3 Accuracy 38
8.1.4 Explained Variance 39
8.1.5 Edge Distribution 39
8.2 Classification 40
8.2.1 ROC curve 40
8.2.2 Precision Recall Curve 41
8.2.3 Confusion Matrix 42
8.2.4 Expected Calibration Error 42
9 Embedding Visualization 43
10 Dataset 44
11 Results Analysis 45
12 Conclusion 61
13 REFERENCES 62
14 Appendix 67

ABSTRACT

Graph Neural Networks (GNN) are a powerful technique to model data on non-eucledian domains with neural network universal function approximator. They are mainly used on static networks where nodes and edges do not change over time. To overcome this issue new models extended the GNN model to incorporate temporal data and the resulting model is defined as a Dynamic Graph Neural Networks (DGNN). We use this technique to model the bilateral trade evolution of the International Trade Network (ITN) where nodes in the network represent the countries, encoded as a feature vector and the edges represent the trade relations between two countries. We analyze the topological and statistical properties of the estimated model and visualize the evolution of relations between countries. We then evaluate the model predictive performance on link prediction and reconstruction capabilities.

0 INTRODUCTION



The aim of this thesis is to develop an algorithm that can predict the time evolution of the international trade network for different commodities. The analysis of networks has long been of interest for many the research communities as many real-world problems arise from a network structured data. However, the focus has usually been on graphs that do not change over time, known as static graphs (Chen & Chen, 2017). Examples of some field of study that necessitate the use of graph algorithms are neuroscience, where the neurons can be viewed as nodes in the graph and the synapses as the edge connecting two nodes (Bessadok, Mahjoub, & Rekik, 2021); chemistry, where the atoms of a molecule represent the nodes of the graph and edge bonds between atoms represent the edges (Hannes, 2021) (Bonginia & Bianchini, 2021); traffic prediction where the nodes are sampled points on a road-map and the edges are the roads connecting the nodes (Yu, Yin, & Zhu, 2018) (Zhao, et al., 2020) (Zheng, Fan, Wang, & Qi, 2020); knowledge graphs, where nodes are the entities and the connections are the relations between entities (Wang & He, 2019) (Ji, Pan, & Cambria, 2021), social network analysis where the nodes are people and the edges represent a friendship (Hoff, Raftery, & Handcock); financial transaction where the sender and receiver of the transaction are the nodes and the transaction amount represent the edge between nodes (Dan, Jiajing, Qi, & Zibin, 2020) (Wei, Zhang, & Liu, 2021).

The focus of this thesis is however on dynamic graphs applied to the International Trade Network (ITN) where both nodes' attributes and the resulting edges between the nodes change over time, thereby adding a temporal dimension to the graph.

The problem with having a graph as input is that the graph is that a graph is an irregular domain since not all the nodes are connected, therefore the neighborhood of a node, defined as the set of nodes connected to a node, changes for different nodes. For example, images can be thought as a regular graph where the size on the neighborhood of a pixel in the image is the same for every pixel. Instead, in the ITN the neighborhood size of nodes changes depending on the country. Therefore, one needs to design an algorithm that can process a variable input size.

Many techniques have been developed to allow a statistical model to process graph structured data. One stream of research focuses on modelling the graph through Bayesian methods by putting a prior probability distribution on the edge of the network (Caimo & Friel, 2011) and estimating the posterior of the parameters of the model though Monte Carlo Markov Chain (MCMC) sampling. A more advanced model was developed by (Billio, Casarin, Kaufmann, & Iacopini) where they generalize the VAR models to dynamic tensors and estimate the parameters with MCMC. Another stream of research approaches the problem from a dimensionality reduction stand-point, called Latent Factor Models (LFM), where one tries to reconstruct the network from the latent characteristics of the nodes and assumes that the network structure can be fully defined by the node features (Hoff, Raftery, & Handcock) (Kim, Lee, Xue, & Niu, 2018). A generalization of latent factor models to multiple dimensions can be defined via tensor decomposition techniques (Bader, Harshman, & Kolda, 2007) where a Three-way DEDI-COM model is used to decompose the dynamic graph, however one downside is that the model cannot be used in an inductive setting. Another stream of research focuses on the topological structure of the network and take a statistical mechanics approach of estimating the distribution of the edges (Tiziano, Fagiolo, & Diego, 2011). Another stream of research focuses on the evolution of edges by applying a generative model of the network inspired by the Ecology literature with a preferential attachment model (García-Algarra, Mouronte-López, & Galeano, 2019). Another stream of research approaches the problem through a graph theory perspective. The two main branches are spectral based methods (Sandryhaila & Moura, 2013) (Bruna, Zaremba, Szlam, & LeCun,

2014) (Gavili & Zhang, 2017) (N. & Welling, 2017) and spatial based methods (Petar, et al., 2018) (Dwivedi & Bresson, 2021). The spectral methods analyze the spectrum of the Laplacian matrix by the eigenvalue-eigenvector decomposition. Due to the intense computational requirements of computing the eigenvectors of the Laplacian (Defferrard, Bresson, & Vandergheynst, 2017) introduced an approximation of the spectral filters by the Chebyshev expansion of the graph Laplacian. The spatial methods define the convolutions directly on the graph instead of first transforming the graph into the spectral domain. The neural network based architecture call *"Graph Attention Network"* (Petar, et al., 2018) build on a previous technique used in the sequential modelling domain called the *"Attention Mechanism"* (Vaswani, Shazeer, & Parmar, 2017) and applies it to the graph domain. The advantage of this technique is that it does not require a fixed size neighborhood, can be efficiently parallelizable and can be used in inductive learning problems where one does not need to recompute the parameters of the model when adding a new node to the graph.

Most of the previous techniques focus on encoding a static graph. For a more general model, one would need to estimate the effect that previous node interactions have on future nodes interactions. The next paragraph will introduce different techniques for modelling a dynamic graph.

The model from (Singer, Guy, & Radinsky, 2019) first computes the representation of the nodes by a biased random walk through each snapshot of the dynamic graph, with a technique called node2vec (Grover & Leskovec, s.d.). Then tries to align the consecutive timesteps node embeddings by minimizing the distance between timesteps embedding with a rotation matrix that rotates the future node embedding in the direction of the previous node embedding. The final node representations are fed through a LSTM recurrent neural network and LSTM hidden state is used for classification. This model however is not end-to-end differentiable since the node2vec algorithm is a sampling based algorithm. Furthermore, the model is limited to model the node representations of the node embeddings and does not predict the evolution of the embeddings nor the adjacency matrix.

The model from (Pareja & Domeniconi, 2019) first utilizes a Graph Convolutional Neural Network (GCN) to get the node embeddings for each node in the graph, the utilizes a recurrent neural network model to directly predict the weights of the GCN.

The model from (Zhao, et al., 2020) called T-GCN is similar to the previous model but instead of feeding to the recurrent model the weights of the GCN, the recurrent model takes as inputs the node embeddings previously generated. This thesis will build on the architecture from this model to encode the node embeddings which will be used as factors to reconstruct the graph. The main differences between this model and the one which will be presented is that we utilize a different architecture called Graph Attention Network to encode the nodes of the graph, as the GCN cannot process directed graphs. Furthermore the T-GCN in the paper focuses on temporal graphs which are graph with a fixed adjacency matrix and evolving node features. The model developed in this thesis instead accepts different adjacency matrices for each time step prediction.

The model from (Chen & Chen, 2017) defines a bilinear latent factor model to get the graph reconstruction from the nodes features and estimates the factors by defining the auto-cross-covariance matrices at lag h between the column of the factor matrix. This thesis will utilize the bilinear model to reconstruct the graph from the node embeddings found with the T-GCN model.

The outline of thesis is as follow: the second chapter introduces the notion of a graph and the different types of graphs, the third chapter introduces some network measures to analyze the statistics of the nodes, the fourth chapter introduces the encoder building block, the fifth chapter introduces the decoding building block and the sixth chapter introduces the recurrent temporal modelling block, the seventh chapter defines the loss function utilized to estimate the parameters of the model and the eighth chapter introduces two uncertainty measures related to the data and the model.

1 INTRODUCTION TO GRAPHS

A graph $G \in \{N, E\}$ is defined as the collection of a set of nodes $N \in \{n_0, \dots, n_k\}$, where $k = |N|$ is the total number of nodes, and a set of edges $E \in \{e_0, \dots, e_m\}$ connecting two nodes, where $m = |E|$ is the total number of edges. The edges $e_{ij} = (n_i, n_j) \in E$ of a graph are usually indexed by the indices of the source node i and the target node j. The graph can be of three types depending on the values of the edges. The first type is the binary graph where the edges are either 0 or 1: $e_{ij} \in 0{,}1$. The second type is the weighted graph where the value of the edges are real numbers: $e_{ij} \in \mathbb{R}$. The weighted graph is a generalization of a binary graph. The third type of graph is defined as an attributed graph, where the edges have a vector as the value: $e_{ij} \in \mathbb{R}^d$. The attributed graph is a generalization of a weighted graph.

1.1 UNDIRECTED AND DIRECTED GRAPHS

The graph can be furthermore split in two categories. The first category is the undirected and thereby symmetric graph where the value of edges between the source node and the target node is the same: $e_{ij} = e_{ji}$.

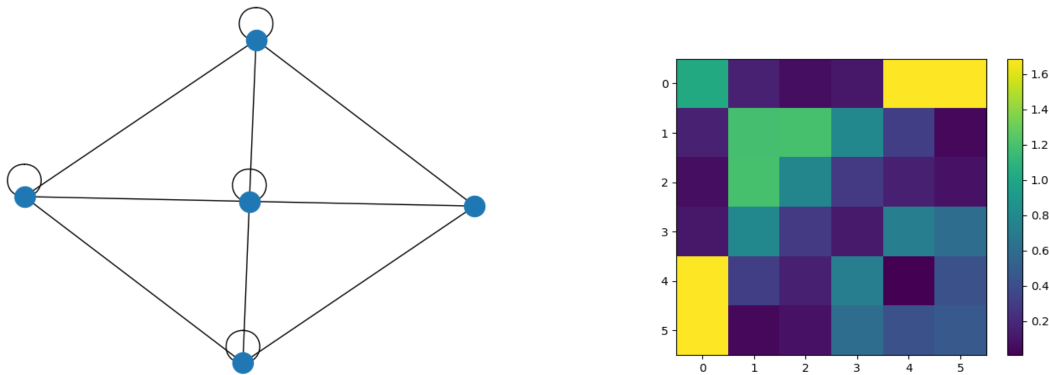


Figure 1 Example of an Undirected Graph and undirected adjacency matrix

The second category is the directed and thereby asymmetric graph where the value of edges between the source node and the target node may not be the same. Since the transaction between countries has a directed structure as not all edges are mirrored and the magnitude of the exchange differs whether the country is small or big, we utilize this graph structure to build the model.

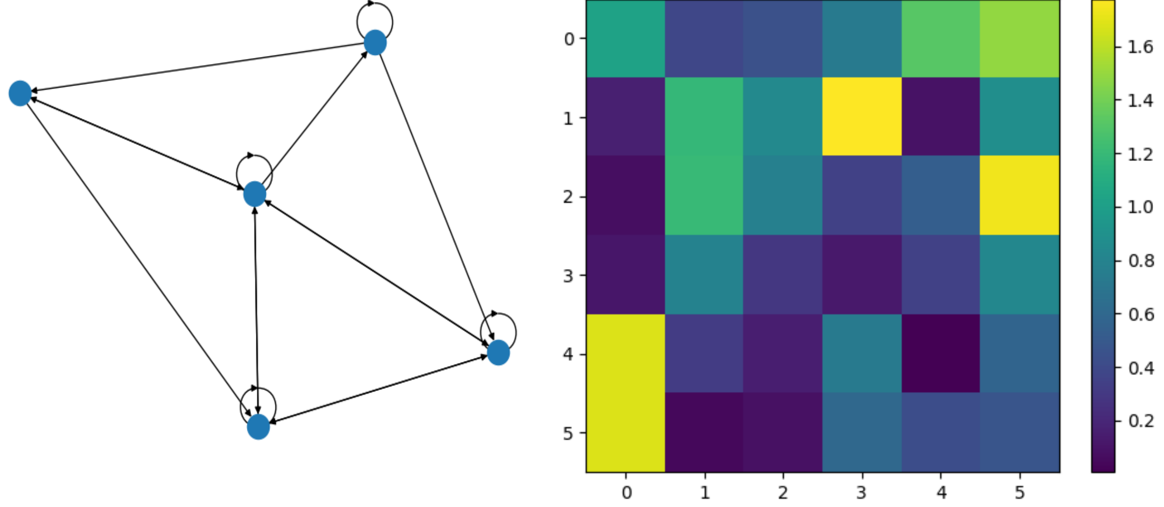


Figure 2 Example of a Directed Graph and a directed adjacency matrix

The adjacency matrix $A \in \mathbb{R}^{kxk}$ is a square matrix the matrix used to represent the graph where values a_{ij} are the edge weights. If the graph if undirected then $A = A^T$.

1.2 Stochastic Graphs

A stochastic graph is a graph $G \in \{N, E, X\}$ where N is the set of nodes, E is the set of edges and X is the set of features associated with each node, is sampled from a probability distribution defined over the space of graphs: $G \sim P(G)$. Since the model developed in this thesis belongs to a latent factor model, it assumes that the graph structure can be fully characterized by the features of the nodes. We can define the probability distribution over the graph as conditional to the node features:

$$G \sim P(G|X)$$

Since the ITN has a weighted directed graph structure one needs to model both whether and edge is present between two nodes according to their features and, conditional on the presence of an edge, the weight of the edge. The joint probability of the binary and weighted edges can be decomposed as a probability distribution over binary edges B conditioned on the node features and a probability distribution over the edge weights W given the existence of an edge and the node features:

$$G \sim P(W|X, B)P(B|X)$$

Where $P(W|X, B)$ is a joint probability over the weighted edges of the graph and can be any parametric continuous distribution:

$$P(W|X,B) = \prod_i \prod_j P(W_{ij}|x_i, x_j, B_{ij})$$

$$W_{ij} \sim P(W_{ij}|x_i, x_j, B_{ij}) = LogNormal(W_{ij} \mid x_i, x_j, B_{ij}; \mu_{ij}, \sigma_{ij})$$

Since it is known that a log-normal distribution approximates quite well the value of the transaction of the ITN (Aitchison & Brown, 1957), we choose this distribution to model the weighted adjacency matrix, parametrized by the mean μ and variance σ:

$P(B|X)$ is a joint distribution with independent Bernoulli components parametrized by p_{ij}.

$$P(B|X) = \prod_i \prod_j P(B_{ij}|x_i, x_j)$$

$$P(B_{ij}|x_i, x_j) = Bernoulli(B_{ij} \mid x_i, x_j ; p_{ij})$$

1.3 DYNAMIC GRAPHS

A dynamic graph is a graph where the nodes features, and the adjacency matrix change over time. Formally a dynamic graph can be defined as a collection of graph snapshots $G^{dyn} \in \{G^0, \ldots, G^t\}$ where each snapshot $G^t \in \{N^t, E^t, X^t\}$ is the graph associated with a time stamp indexed by t. The node features are vectors $\boldsymbol{x^t}_{n_i} \in \mathbb{R}^d$ associated with the characteristics of the node. The full nodes feature matrix for timestamp $X^t \in \mathbb{R}^{k \, x \, d}$ is the matrix containing all the stacked node features. If the nodes do not have features, then the graph is called a featureless graph. To model a featureless graph the node features are set as a one-hot encoded vector where the vector has value 1 corresponding to the index of the node and zero everywhere else. The vector is therefore the indicator function $\mathbb{1}_i$.

The transition dynamics of a dynamic graph evolve according to a transition function that take the node features at the previous time step and the current node features to predict the next time step node features:

$$X_{t+1} = f(X_t, X_{t-1})$$

The function can either be a deterministic function or a stochastic function. In this thesis the transition function will be a deterministic recurrent neural network.

2 EXPLORATIVE ANALYSIS: NETWORK MEASURES

To understand and visualize the structure of each snapshot of the dynamic graph we utilize a diverse set of node similarity metrics which measure different statistics of the nodes of the graph. For large graph the direct visualization of nodes and edges becomes cumbersome since the plot would be too crowded to analyze. The different metrics on the vertices can capture different aspects of different type connectivity of the graph. We first introduce the simplest and most common metrics for directed graphs which are the in-degree and out-degree of the nodes, the closeness between two nodes, the link density and the Jaccard Similarity. We then introduce more sophisticated ones such as the eigen-vector centrality and the Laplacian Clustering. (Zafarani, Abbasi, & Liu, 2014). We utilized the Networkx library (Hagberg, Schult, & Swart, 2008) to compute the following metrics.

2.1 NODE DEGREE

The node degree centrality counts the number of nodes adjacent nodes j to a node i. The degree D is therefore the size of the neighborhood of node i:

$$D(n_i) = |\mathcal{N}_i|$$

Where $\mathcal{N}_i = \{n_j | e_{ij} \in E\}$ is the set of nodes j that have a connecting edge with node i. One can easily compute the node degree by summing alone one axis of the adjacency matrix.

For directed graph one needs to distinguish the in-degree and the out-degree on the node. For node n_i, the in-degree $D^-(n_i)$ measures the number of incoming edges for node and the out-degree $D^+(n_i)$ measures the out-going edges. The total degree is the sum of the in-degree and out-degree. When combining the in/out degree one ignores the edge direction, and the result will be the degree of an undirected graph:

$$D^-(n_i) = \sum_j A_{ij}$$

$$D^+(n_i) = \sum_i A_{ij}$$

$$D^{tot}(n_i) = D^+(n_i) + D^-(n_i)$$

To see which noes are more central, the degree measures can be ranked to see which are the most important nodes. If the in-degree is high, it signifies that a node has high prestige in the network. If the out-degree is high, it signifies that a node has high gregariousness, therefore having high influence in the network.

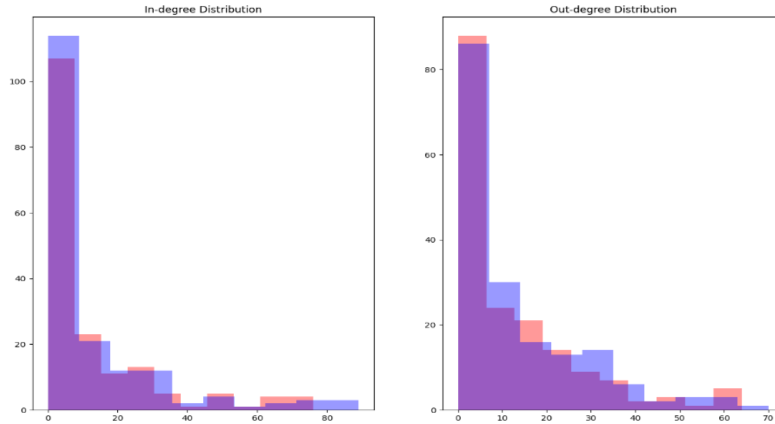


Figure 3 Node Degrees for the True graph (blue color) and Predicted graph (red color)

2.2 Closeness Centrality

The closeness between two nodes is a measure of centrality which calculates the inverse of the shortest path between two nodes. The shorter the path distance between two nodes the higher the closeness:

$$C(n_i, n_j) = \frac{1}{d(n_i, n_j)}$$

The shortest path can be computed by the A* search algorithm. The distance matrix computed by the closeness centrality can be utilized for computing the closeness centrality measure which sums the closeness between node i and all the other nodes excluding itself:

$$C(n_j) = \frac{n-1}{\sum_{1}^{k-1} d(n_k, n_j)}$$

2.3 LINK DENSITY

The link density measures the ratio between the existing edges and the total possible edges. It allows to understand how much the graph is connected compared to a complete graph where every node has a connection with every other node. For directed graph the formula is:

$$Link\ Density\ Directed = \frac{total\ edges}{total\ possible\ edges} = \frac{|E|}{k(k-1)}$$

For undirected graph one need to account for the fact that every edge is reciprocated, thereby dividing by two the total edges:

$$Link\ Density\ Directed = \frac{total\ edges}{total\ possible\ edges} = \frac{|E|}{k(k-1)/2}$$

2.4 JACCARD SIMILARITY

The Jaccard Similarity measures the overlap between the neighborhoods of two distinct nodes. The measure is defined as the size of the intersection between two node's neighborhoods over the union of the two neighborhoods:

$$J(n_i, n_j) = \frac{|\mathcal{N}_i \cap \mathcal{N}_j|}{|\mathcal{N}_i \cup \mathcal{N}_j|}$$

3 STATIC GRAPH ENCODER

3.1 ARTIFICIAL NEURAL NETWORK

Artificial Neural Networks (ANN) (Goodfellow, Bengio, & Courville, 2016) is technique that can model nonlinear relations between the input and the output. It is a biologically brain-inspired function where each node of the network receives the input from the sender and chooses how much of the signal to forward to subsequent nodes. Each layer of the neural network is usually comprised of two steps: the composition of a linear function with a nonlinear function. Considering the layer l, the first function in the network layer projects the input features $X \in \mathbb{R}^{k \times d}$, where k is the number of sampled points and d is the original input size, by a linear projection matrix weight matrix $W^l \in \mathbb{R}^{d \times o}$, where o is the size of the output. In addition, one can use a bias term $b \in \mathbb{R}^{k \times 1}$ shared across the output dimensions, which acts as a translation the output space:

$$z^l{}_k = x_k^{\mathrm{T}} \cdot W^l = \sum_d x_{kd} \cdot W^l{}_{do} + b_k$$

$$z^l{}_k \in \mathbb{R}^o$$

Which in matrix notation is

$$Z^l = X \cdot W^l + b^l$$

$$Z^l \in \mathbb{R}^{k \times o}$$

The hidden representation Z is then transformed by a nonlinear function $\sigma(\cdot)$ selected from a set of available functions. The non-linearity is applied element wise for each output dimension of the vector z_k. The result of the nonlinear transformation is called the activation of a neuron:

$$a_{ko}^l = \sigma(z^l{}_{ko})$$

$$A^l = \sigma(Z^l)$$

Chaining these operations will result in a deep neural network with multiple layers:

$$DNN^L(X) = \sigma^L\left(Z^L \ldots \left(\sigma^0\left(Z^0(W^lX)\right)\right)\right) = \sigma^L \circ Z^L \circ W^L \circ \ldots \circ X$$

Where L is the number of layers.

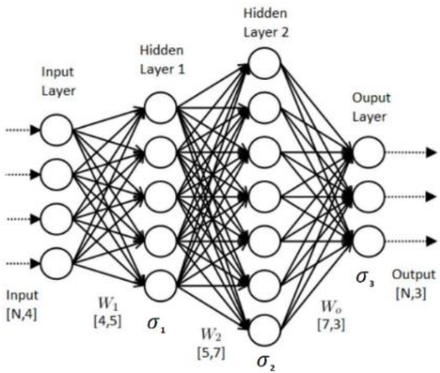


Figure 4 Diagram of a Deep Neural Network (https://stackoverflow.com/, 2021)

The above diagram represents the output units as nodes and the weights connections as directed edges.

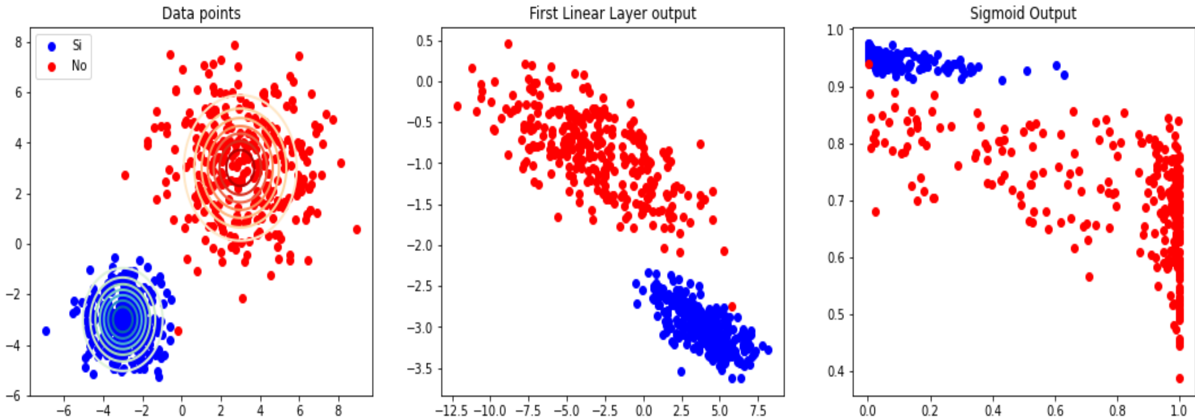


Figure 5 Visualization data transformed by the hidden layer and activation layer

3.1.1 Sigmoid Activation

One common activation function is the logistic function also called sigmoid. This function has the property of mapping the domain of the function onto the range contained between (0,1).

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

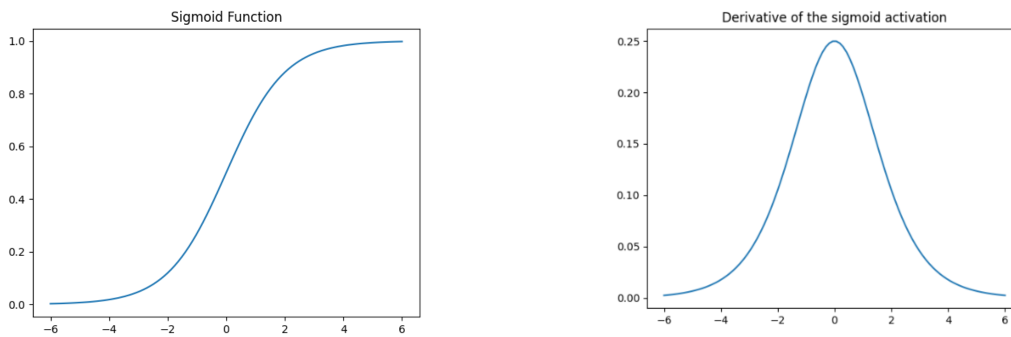


Figure 6 Sigmoid Function and its derivative

This activation can be used as either the hidden activation or for the final layer for binary classification. One issue with this activation function is that the derivative saturates for large values of the input, thereby impeding the learning process since the derivative update will be close to zero.

3.1.2 Relu Activation

The most common activation function is the relu function which is $\sigma_{relu}(x) = \max(0, x)$. This activation is very common and performs empirically well because it solves the gradient saturation problem (Xu, Wang, Chen, & Li, 2015) because the gradient is constant independently from the values of the input. One problem with this activation is that the gradient is a step function. For values less than zero the gradient update will be zero.

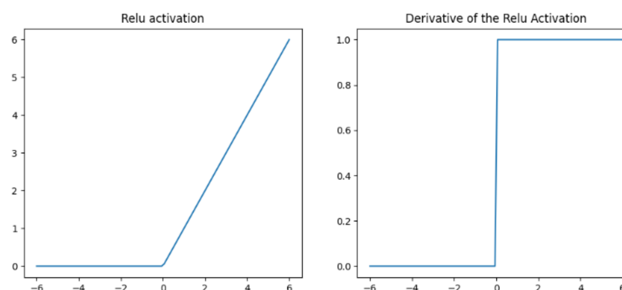


Figure 7 Relu Function and its derivative

3.1.3 Softplus Activation

The softplus activation function is an improvement over the relu function as it smoothens the boundary at 0. Therefor the gradient still exists for values near zero.

$$\sigma_{softplu}(x) = \ln{(1 + e^x)}$$

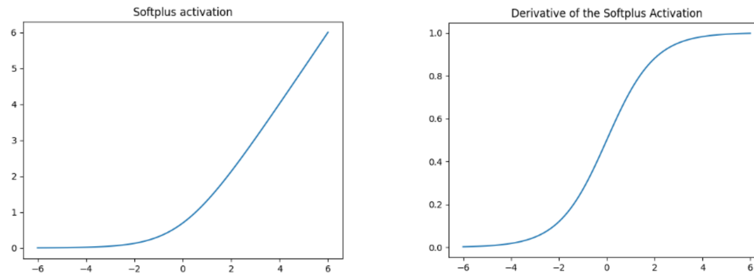


Figure 8 Softplus Activation function and its derivative

3.1.4 Leaky-Relu Activation

The Leaky-Relu activation function is similar to the Relu activation function however it has a small nonzero gradient for values less than 0 controlled by the parameter α.

$$\sigma_{leaky-relu}(x) = max(\alpha x, x)$$

$$\alpha \in [0,1]$$

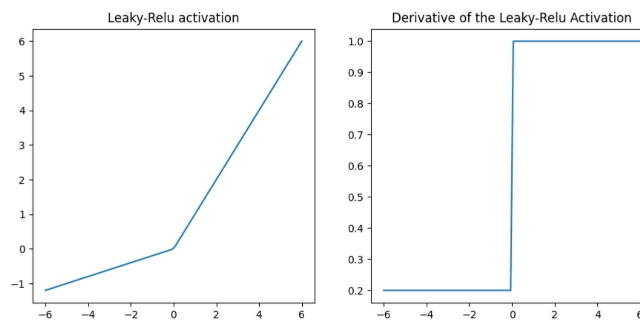
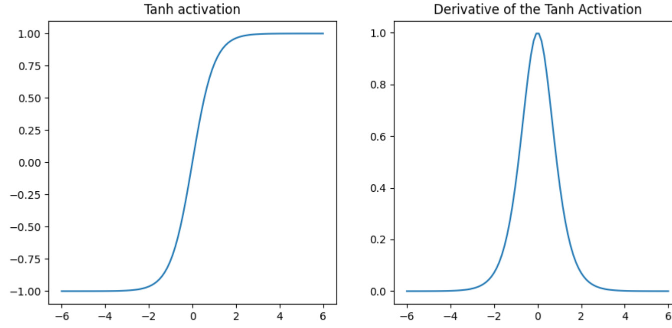


Figure 9 Leaky-Relu Function and gradient

3.1.5 Tanh Activation

The Tanh activation maps the input domain onto the range (-1, 1) and is often used as a gating function in recurrent neural network models.



3.2 AUTOENCODER MODEL

A deep neural network can be though as an encoding step which maps the inputs to a lower dimensional space called hidden space. If a decoding step which maps the hidden back to the original input space is present, the resulting architecture is defined as an autoencoder, as it tries to automatically encode the input features by non-linearly projecting them to a lower dimensional space and then trying to reconstruct the original input features in the original space. Formally, an autoencoder is defined as:

$$DNN_{encoder} : X \rightarrow X'$$

$$DNN_{decoder} : X' \rightarrow X$$

Where $X \in \mathbb{R}^{k \times d}$ and $X' \in \mathbb{R}^{k \times 0}$, and $o < d$. Then one needs to define how close the reconstruction is from the true data points:

$$L(X, X') = ||X, \hat{X}||^2 = ||X - DNN_{decoder}(DNN_{encoder}(X))||^2$$

$$= ||X - \sigma^L\left(Z^L \ldots \left(\sigma^0\left(Z^0(X)\right)\right)\right)||^2$$

Where $L(X, X')$ is the L_2 norm reconstruction loss between the input and the predicted output of the neural network. Many different loss functions can be utilized based on the task at hand, which will be explained more in details in chapter 7. The autoencoder model is a general model which can be used with many different architectures such as Convolutional Neural Networks (CNN) (LeCun, Haffner, Bottou, & Bengio, 1999), Recurrent Neural Networks (RNN) (David, Geoffrey, & Ronald, 1985) and GNNs.

3.3 TRAINING A NEURAL NETWORK

The parameters of the neural network are optimized with gradient descent. At every step of the iteration, called epoch, the weights of the NN are updated by the gradient of the weights with respect to the loss function. The gradient is multiplied by a parameter that controls how much the weights will be updated:

$$\theta_{t+1} = \theta_t - \nabla_\theta L(X, \hat{X})$$

To calculate the gradient w.r.t the loss function, the back-propagation algorithm is used (Rumelhart, Hinton, & Williams, 1986). This algorithm is based on the chain rule of derivation and efficiently computes the gradient for every layer. Here we show an example of computing a single weight partial derivative w.r.t the loss function as layer l.

$$\frac{\partial L(X,\hat{X})}{\partial W_{ij}^{(l)}} = \frac{\partial L(X,\hat{X})}{\partial \sigma_i^{(L)}} \cdot \frac{\partial \sigma_i^{(L)}}{\partial z_i^{(L)}} \cdot \frac{\partial z_i^{(L)}}{\partial W_{ij}^{(L)}} \cdot \frac{\partial W_{ij}^{(L)}}{\partial \sigma_i^{(L-1)}} \cdots \frac{\partial z_i^{(L-1)}}{\partial W_{ij}^{(L-1)}}$$

3.4 GRAPH NEURAL NETWORKS

The main difference between an ANN and a Graph Neural Network (GNN) is that the GNN operates on a set of nodes features with a dependency structure based on the adjacency matrix, instead of treating the nodes as an independent sampled point. The static graph encoder is a neural network function that maps the node features of the graph to a latent representation. The main idea behind a graph neural network is that the node features, and therefore the information related to a node, is shared between adjacent nodes based on the adjacency matrix. Therefore, the GNN takes as input both the node features and the adjacency matrix. The procedure of sharing messages is called the *message passing* step. Formally a GNN is a function defined as $GNN: (X, A) \rightarrow X'$. All GNN models so far developed can be casted into a message passing framework (Bronstein, Bruna, Cohen, & Veličković, 2021).

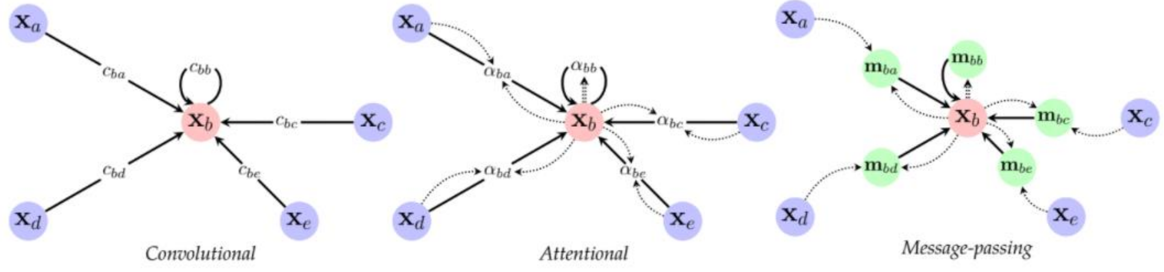


Figure 10 Message Passing Representation (Bronstein, Bruna, Cohen, & Veličković, 2021)

The node features x_i are updated by weighting the incoming nodes features x_j of the neighbors. The node features are then aggregated via a permutation invariant operator such as the sum, mean or max operator.

For a convolutional neural network, the values of the weights depend directly on the weights of the adjacency matrix.

For the graph attention network, the weights are implicitly computed based on the agreement between node features of the incoming neighbors and the receiver node via the attention mechanism. The message passing framework is the superset of the two methods where the message between two nodes is computed by a learnable function. In the following paragraph the graph attention model will be explained more in details.

3.4.1 Graph Attention Network

The graph attention network (GAT) (Petar, et al., 2018) was developed to increase the flexibility of the GCN by having learnable filter graph which learns the weight to assign a node neighbor in order to efficiently learn to propagate the signal through the graph. For each time step of the dynamic graph we have a set of node features $X = \{x_0, \dots, x_k\} \in \mathbb{R}^{k \times d}$. The GAT model will product a new set of node features X' as its output.

The initial set is to first multiply the initial features by a matrix $W \in \mathbb{R}^{d \times f}$ which will project the initial node features to the space of dimension f:

$$H = XW$$

Then the transformed node features H will be utilized to get the edge attention matrix A. First a scoring function will be used to get the score of the tuple of nodes (i,j) features.

The scoring is done by concatenating the edge features of the two nodes and multiplying the resulting vector by a learnable kernel, followed by an activation function:

$$e_{ij} = Relu(k^T[h_i||h_j])$$

To efficiently calculate the edge scores one can split the kernel $k \in \mathbb{R}^{2f\times1}$ by half, where one half will score the so-called self-attention, and the other half will score the neighbor. The result of the two scores $h_{slef} \in \mathbb{R}^{k\times1}$ and $h_{neighb} \in \mathbb{R}^{k\times1}$ can be summed together by taking advantage of broadcasting operations of mathematical libraries to get the edge score matrix $E \in \mathbb{R}^{k\times k}$.

Since the score matrix E will be dense as the attention scores every node tuple, many nodes that are not in the neighborhood of the aggregating node will have influence in the message aggregation step. To avoid this issue the authors proposed to apply a mask to every edge which in not in the edge set.

$$e_{ij} = e_{ij} + mask_{ij}$$

$$mask_{ij} = \mathbb{1}_{e_{ij}\in E}$$

Where the values of the mask are 0 for existing edges and $-10e^8$ for non-existing edges.

The high negative value in the mask is necessary because the edge scores will be normalized by a softmax function, therefore if the mask values were not high enough there would be still some weight given to that specific edge attention. The edge scores are then normalized as follows:

$$a_{ij} = \frac{\exp(e_{ij})}{\sum_{k\in\mathcal{N}_i}\exp(e_{ik})}$$

The attention weights will then be used to compute a linear combination of the features by multiplying the attention matrix A with the node features X, followed by an activation function:

$$X' = \sigma(\mathrm{A}\cdot\mathrm{X})$$

$$x_i' = \sigma\left(\sum_{j\in\mathcal{N}_i} a_{ij}^k h_j\right)$$

To stabilize the training and encoding different graph views the authors proposed to multiple kernels for the attention mechanism $\{k_i, \dots, k_K\}$ and either concatenate the resulting output node features:

$$x_i' = \left[\sigma\left(\sum_{j \in \mathcal{N}_i} a_{ij}^0 h_j\right) || \dots || \, \sigma\left(\sum_{j \in \mathcal{N}_i} a_{ij}^K h_j\right)\right]$$

or average output node features:

$$x_i' = \frac{1}{k} \sum_k \sigma\left(\sum_{j \in \mathcal{N}_i} a_{ij}^0 h_j\right)$$

This architecture can be castes into a message passing framework as:

$$\mathbf{h}_u = \phi\left(\mathbf{x}_u, \bigoplus_{v \in \mathcal{N}_u} a(\mathbf{x}_u, \mathbf{x}_v)\psi(\mathbf{x}_v)\right)$$

Figure 11 Message passing GAT (Bronstein, Bruna, Cohen, & Veličković, 2021)

Where h_u is the final node representation, x_u is the initial node representation, the $\oplus$ is the aggregation operator that aggregates messages from the neighbors, which in this case is just the sum operator, ψ is an initial node embedding transformation function which in the case of GAT is a linear transformation, $a(x_u, x_v)$ is a function that scores the nodes, which in this case is the attention mechanism and ϕ is a final transformation which in this case is the final activation function.

4 STATIC GRAPH DECODER

Now that we have defined the framework to encode the original input nodes into a latent embedding dimension, we can now focus on how to reconstruct the graph adjacency matrix for a single time step. Formally, given the resulting transformed node features outputted by the GAT model, we would like to design a function such that takes as input two nodes and returns a value which will be the parameters of the predictive distribution:

$$Decoder : X' \times X' \rightarrow A$$

We will need to implement three separate decoders, one for each parameter of the distribution. The first decoder will be used to parametrize the joint Bernoulli distribution to predict the existence of and edge, the second decoder will be used to parametrize the mean of the Lognormal joint distribution and the third will be used to parametrize the variance of the Lognormal join distribution.

There are two main ways to reconstruct a stochastic graph parameter from node features. The first one is known as the inner-product decoder, the second one is a generalization of the inner-product decoder and support asymmetric matrices.

The inner-product decoder takes two nodes feature matrices and computes the inner-product of each pair of nodes:

$$Decoder_{inn}(X) = XX^T$$

The resulting matrix will have size $k \times k$ where k is the size of the set of nodes. The inner product measures the unnormalized cosine distance between two points and can be though of as a measure of similarity between two embeddings. The issue with this type of decoder is that the inner-product is a symmetric operation and therefore cannot approximate a directed graph.

To overcome this issue one simple method is to introduce a square matrix with the same dimension of the node features dimension to make the score asymmetric. This type of decoder is called a bilinear decoder (Chen & Chen, 2017):

$$Decoder_{bil}(X) = XRX^T$$

Where $R \in \mathbb{R}^{f \times f}$ and $X \in \mathbb{R}^{k \times f}$. One can realize that if the R matrix was an identity matrix with ones as diagonal and zero everywhere else, the bilinear product would reduce to an inner-product. One way to justify to the bilinear product formula is to first linearly transform the features of the left embedding matrix with a square parameter matrix: $X \cdot R_0$, and the same for the right embedding matrix: $X \cdot R_1$. Then we perform the inner product between the two transformed matrices: $(X \cdot R_0)\,(X \cdot R_1)^T = (X \cdot R_0)(R_1^T X^T) = XRX^T$.

5 RECURRENT TEMPORAL ENCODER

Having defined how to transform the initial features into a latent embedding via a graph neural network encoding and a way to reconstruct a graph from the node embeddings, we will now focus on how to model the dynamics of the node to exploit the temporal structure of the dynamic graph. The models we will use are recurrent neural network models (RNN). These family neural networks are used to process sequential data and the parameters of the neural network are shared across time. These models take the either the output of the model from the previous time step, the input at the current time step or the hidden state from the previous time step which holds information from the past and carries it forward in the future. One advantage of using such architectures to model temporal data is that they have an inductive bias to model sequential structured data and do not necessarily need an explicit time dimension to understand the passage of time. However there are two main drawback of using an RNN. The first is that the computation runs sequentially and can be slow to run for many time stamps. The second major drawback is that they suffer from the so-called *vanishing gradient problem* (Hochreiter, 1998). This problem is inherent from the gradient-based optimization algorithm used, which is called Back Propagation Through Time (BPTT). One way to alleviate the vanishing gradient problem is to introduce some gating functions that modulate the signal going forward and allow better control over the gradient in the backward optimization step. The more advanced recurrent models such as the Long Short Term Memory (LSTM) network and the Gated Recurrent Units (GRU) utilize gating functions and achieve better performance.

5.1.1 Recurrent Neural Network

A recurrent neural network is the simplest RNN for of recurrent network. The model holds a hidden state for each time step which is used to carry information from the past forward into the future. At each time step, the model first encodes the current time step input and maps it to a hidden space. The model then merges the hidden state from the previous time step and the encoded input and it updates the current hidden state. Finally the model decodes the hidden state to get a prediction output which will be wither used for downstream tasks or as the actual output of the model.

Formally a neural network is a function parametrized by the set of parameters θ of the type:

$$h_t = f(x_t, h_{t-1}; \theta)$$

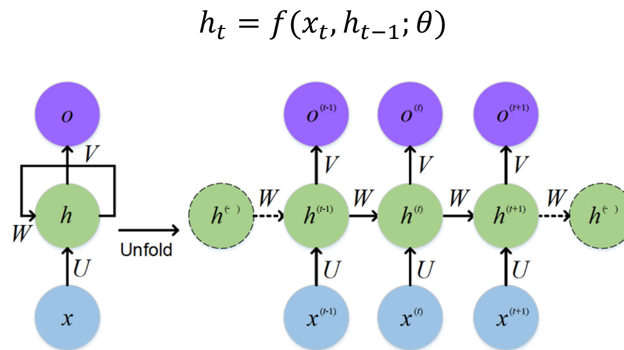


Figure 12 Computation diagram of RNN. The left graph is the representation of the computational flow of the RNN. The right graph is the "unrolled computational graph" where the computational steps are made explicit (Weijiang, Naiyang, Yuan, AU, & Zhigang, 2017)

The computation of the functions is done recursively via a feedback loop that feeds back the hidden state to the function. The forward propagation of the state is done by the following update equations:

$$a = b + Wh_{t-1} + Ux_t$$

$$h_t = \sigma(a)$$

$$o = b_o + Vh_t$$

Where $h_{t-1} \in \mathbb{R}^d$ is the hidden state vector, $W \in \mathbb{R}^{d \times d}$ is the hidden state parameter used to for the hidden state transition, $U \in \mathbb{R}^{i \times d}$ is a weight matrix which maps the input $x_t \in \mathbb{R}^i$ at time to the hidden state space, $b \in \mathbb{R}^d$ is a bias term and σ is the tanh activation function. To get the final output a further linear projection matrix V is used

with a bias term b_o. The initial state h_0 is set to the zero-vector indicating that the sequence is staring.

This simple recurrent architecture can be adapted to process the temporal dependence of the dynamic graph. First, instead of a vector, the input to the RNN will be the encoded node feature matrix $X_t \in \mathbb{R}^{k \times f}$ for the current time step. To get the encoded node feature matrix we will use the GAT encoder with the initial features and current adjacency matrix. The hidden state dimension changes from a vector to a matrix of the same shape as the encoded node feature matrix, therefore each node will have its own state. The recurrent graph architecture is implemented by the following equations:

$$X'_t = GAT(A_t, X_t)$$

$$H_t = b + W_h H_{t-1} + W_x X'_t$$

$$H'_t = \sigma(H_t)$$

These equations implement a GAT-RNN cell. Having updated the previous hidden state with the current inputs, we can then use the new state at the current time step to decode the graph parameters utilized by the joint distribution of the edges with the bilinear decoder model followed by an activation function. Each parameter of the distribution is computed by an independent GAT-RNN cell, thereby decoupling the dynamics of different parameters:

$$P_t = \sigma_{sigmoid}(Decoder_{bil}(H_t^p))$$

$$W_t = \sigma_{relu}(Decoder_{bil}(H_t^w))$$

$$V_t = \sigma_{softplus}(Decoder_{bil}(H_t^v)) + 1^{-3}$$

The activation function for the parameters used to parameterize the Bernoulli distribution P_t is the sigmoid function, since the output of the activation will be bounded between (0,1) and therefore can model del probability parameter. The activation function used for the edge weights W_t is the relu function since the transactions amount between countries must be greater than zero. The matrix W_t will be used to parametrize the mean of the LogNormal distribution. The activation function for the standard deviation parameter is the softplus function with an added a small constant to prevent the numerical instabilities. As previously mentioned, this model suffers from the

vanishing gradient problem. One model that deals with this issue is the Gated Recurrent Unit recurrent model, which will be introduced in the following paragraph.

5.1.2 Gated Recurrent Unit

The Gated Recurrent Unit (GRU) (Cho, Merrienboer, Bahdanau, & Bengio, 2014) (Chung, Gulcehre, Cho, & Bengio, 2014) is conceptually similar to the simple RNN but it's more suitable for longer time sequences. We chose to implement a GRU instead of the LSTM because it has less parameter to estimate compared to the LSTM and empirically performs better. To avoid the vanishing gradient problem the, function implements the so called gated units which allow the function to forget and update part the hidden state. This functionality allows the gradient to create shortcuts path that bypass multiple time steps. The two novel states compared to the RNN are the reset gate and the update gate. First a candidate state C_t is proposed by gating the previous state H_{t-1} with a reset function that output the reset values R_t:

$$X_t' = GAT(X_t, A_t)$$

$$R_t = \sigma_{sigmoid}(W_r X_t' + U_r H_{t-1} + b_r)$$

$$C_t = \sigma_{tanh}(W_c X_t' + U_c(R_t \odot H_{t-1}) + b_c)$$

Where $\odot$ is the element wise product. W_r and U_r are the weight matrices of the reset gate and b_r is the bias term. W_c and U_c are the weight matrices of the candidate state and b_c is the bias. The new hidden state is the result of a linear interpolation between the previous state H_{t-1} and the candidate state C_t. The interpolation coefficient is controlled by the update gate function U_t:

$$U_t = \sigma_{sigmoid}(W_u X_t' + U_u H_{t-1} + b_u)$$

$$H_t = (1 - U_t) \odot H_{t-1} + U_t \odot C_t$$

We then predict the parameters of the distribution of the edges based on the updated hidden state H_t. As in the RNN case, each bilinear decoder will receive the output from a different GAT-GRU encoding block, since sharing the hidden state between parameters is detrimental for the learning process.

5.1.3 Training with Back Propagation Trough Time (BPTT)

To optimize the weights of a recurrent neural network one needs to backpropagate the error gradient to each previous step of the computations. This is because the hidden state of the current time step depends on all the previous hidden states. The technique to propagate the error gradient to the previous time steps is call Back Propagation Through Time (BPTT) (Werbos, 1990). The method is passed on the backpropagation methods, which is simply the chain rule. However, instead of limiting the gradient update to just one step, the chain of gradients is computed up to the initial time step for each time step of the sequence. The following derivation of the algorithm is applied to the simpler RNN case and only for the parameter that multiplies the hidden state. The same logic applies to the more complex GRU and the other parameters.

For each time step, a loss function is computed, and the total loss function will be the sum of the losses divided by the number of time steps:

$$L_{tot} = \frac{1}{T}\sum_{t} L_t(y_t, \hat{y_t})$$

Where y is the desired output and $\hat{y_t}$ is the predicted output. The derivative w.r.t the weights that multiply the hidden state at time t will be (Hu, s.d.):

$$\frac{\partial L}{\partial w_h} = \frac{1}{T}\sum_{t} \frac{\partial L_t(y_t, \hat{y_t})}{\partial w_h} = \frac{1}{T}\sum_{t} \frac{\partial L_t(y_t, \hat{y_t})}{\partial w_h} = \frac{1}{T}\sum_{t} \frac{\partial L_t(y_t, \hat{y_t})}{\partial o_t}\frac{\partial o_t}{\partial h_t}\frac{\partial h_t}{\partial w_h}$$

The problem now stems from the fact that $\frac{\partial h_t}{\partial w_h}$ depends on the previous h_{t-1} according to the update equation of the RNN: $h_t = f(x_t, h_{t-1}; w_h)$. To compute the partial derivative of the hidden state w.r.t the weight w_t we need a recursive formula that computes the partial derivative of the previous hidden state to the current hidden state for each time step (Murat, s.d.):

$$\frac{\partial h_t}{\partial w_h} = \left(\prod_{t=1}^{t-1} \frac{\partial h_{t+1}}{\partial h_t}\right)\frac{\partial h_t}{\partial w_h}$$

$$\Rightarrow \frac{\partial L}{\partial w_h} = \frac{1}{T}\sum_{t} \frac{\partial L_t(y_t, \hat{y_t})}{\partial o_t}\frac{\partial o_t}{\partial h_t}\left(\prod_{t=1}^{t-1} \frac{\partial h_{t+1}}{\partial h_t}\right)\frac{\partial h_t}{\partial w_h}$$

Since the formula for the derivative contains the product of the partial derivative of the hidden state up to time t-1, this will lead to instability during training as the gradient

would either vanish or blow up. To mitigate this effect the one can truncate gradient computation to T-k time steps. The summation will thereby begin on the k time step and the gradient will then backpropagated up to k time steps. Thanks to modern auto differentiation frameworks the recursive gradient updates is calculated automatically.

6 LOSS FUNCTION

The following loss function is based on the maximization of the log-likelihood of the distribution parameterizing the network edges. The likelihood of a function is the probability of the data being generated by the parameter of the distribution:

$$\mathcal{L}(\theta|\boldsymbol{x}) = P(x_0, \dots, x_i|\theta) = \prod_i P(x_i; \theta)$$

Where, in the example above, $\boldsymbol{x}$ is a vector containing independently sampled points generated by the true parameter θ^*. $\mathcal{L}$ is the likelihood function of the parameter θ. Since the points are independently sampled, the join probability of the points is the product of their individual probability distribution parametrized by the parameter θ. We would therefore like to find the parameter of the distribution that maximize the likelihood:

$$argmax_{\,\theta}\mathcal{L}(\theta|\boldsymbol{x})$$

The likelihood of a graph corresponding to a single time step if defined by the edge distribution. Since the network adjacency matrix is a sparse matrix with many more missing edges that existing edges, we chose the zero-inflated lognormal distribution to model the edge probability. The zero inflated lognormal distribution is a mixed discrete-continuous distribution.

One component of the zero-inflated lognormal is the lognormal distribution. Samples from a lognormal distribution are distributed according to exponentiated gaussian samples and vice versa, the natural log of samples from the lognormal distribution will be distributed as a gaussian distribution:

$$x \sim Gaussian(\mu, \sigma)$$

$$e^x = y \sim Lognormal(e^\mu, e^\sigma)$$

The lognormal is characterized by the following pdf parametrized by μ and σ:

$$PDF(x) = \frac{1}{x\sigma\sqrt{2\pi}} e^{-\frac{\ln(x)-\mu}{2\sigma^2}}$$

The discrete component of a zero-inflated distribution is the Bernoulli distribution:

$$\{0,1\} \sim Bernulli(p)$$

The pdf of the zero-inflated lognormal distribution is (Vandal, Kodra, Dy, & Ganguly, 2018):

$$PDF(x) = f(x; \mu, \sigma, p) = \begin{cases} 1 - p & x = 0 \\ p \dfrac{1}{x\sigma\sqrt{2\pi}} e^{-\frac{\ln(x)-\mu}{2\sigma^2}}, & x > 0 \end{cases}$$

Since the likelihood function is a product of individual probability density functions, it would not be computational feasible to directly optimize it because of the product rule of differentiation. One common way to obviate the problem is maximizing the log-likelihood. The log is a monotonically increasing function. The same value will maximize both the likelihood and the log-likelihood. Taking the log of the products will result in a sum over log-densities functions for each possible edge in the graph. The log likelihood of join distribution of the edges for one time step will be:

$$Log(\mathcal{L}(\theta)) = \ln(f(E\,;\theta)) = \frac{1}{k^2}\sum_i \sum_j \mathbb{1}_{e_{ij}>0} \cdot P_{ij} + (1 - \mathbb{1}_{e_{ij}>0})(1 - P_{ij})$$

$$-\frac{1}{2k^2}\sum_{i,j \in E} V_{ij}^{-2} \,||\log(E_{ij}) - W_{ij}||^2 + \log(V_{ij}^2)$$

Where $\theta = \{P, W, V\}$ is the set on weight matrices that will parametrize the distribution and k^2 is the total number of edges. P is the decoded probability matrix, W is the decoded mean matrix and V is the decoded standard deviation matrix. From the formula one can recognize that the first part is the cross-entropy binary loss classification loss, and the second part is the regression loss with an adjustment for the variance. If the variance was constant the regression loss would reduce to the squared error loss. The

loss function takes as input the adjacency of the t+1 time step and the parameters of the distribution over edges predicted at time step t, which will be used to compute the likelihood for the adjacency matrix.

To optimize the weights of the network, for each time step we update the parameter weights based on the gradient of the negative log-likelihood function, so that the optimization process will minimize the loss function:

$$\theta_{t+1} = \theta_t - \alpha \nabla_\theta Log(\mathcal{L}(\theta_t))$$

Where α is the learning rate hyper-parameter that controls how much the weights are updated. The problem with the maximum likelihood estimation is that it gives a point estimate of the optimal parameters and not the full posterior distribution. The following chapter will present how to approximate the full posterior and the difference between model and data uncertainty

7 Aleatoric and Epistemic Uncertainty estimation

The total uncertainty can be divided into a model uncertainty part called epistemic uncertainty, which represent the uncertainty over the model parameters, and the aleatoric uncertainty, which represent the inherent variability of the data. The epistemic uncertainty is also known as reducible uncertainty because as the number of data points increases, the lower the uncertainty over the parameter of the model. For points close to the training input data points the epistemic uncertainty will be low and will progressively be higher for point outside the training distribution.

7.1 Aleatoric Uncertainty

The expectation and variance of the distribution is (Vandal, Kodra, Dy, & Ganguly, 2018):

$$\mathbb{E}[E_s] = P_s \, exp(W_s + \frac{1}{2} V)$$

$$\mathbb{V}[E_s] = P_s^2 \, exp(2W_s + \, 2V)$$

The variance of $\mathbb{V}[E_s]$ represents the uncertainty of the data, also known as aleatoric and irreducible uncertainty.

7.2 EPISTEMIC UNCERTAINTY WITH DROPOUT

Dropout (Srivastava, Hinton, & Krizhevsky, 2014) is a simple technique to inject noise into every layer of the neural network. The resulting output will be stochastic, and the variance of the output will represent the model uncertainty over the input data. Dropout samples, at every layer, either a binary mask or a continuous valued mask from either a Bernoulli distribution or a gaussian distribution and multiplies the mask to the layer activations outputs.

$$\boldsymbol{m}^l \sim Bernuoulli(1-p)$$

$$\sigma_l'(z) = \sigma_l(z) * \boldsymbol{m}^l$$

$$\sigma_{l+1}(z) = \sigma_l'(z) W_l$$

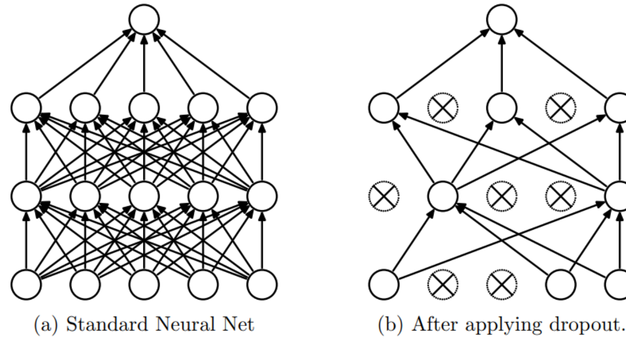


Figure 13 Dropout visualization (Srivastava, Hinton, & Krizhevsky, 2014)

Where σ if the activation function of the units in a single layer l and $\boldsymbol{m}^l \in \mathbb{R}^{|z|}$ is the mask of the same size of the activation output and p is the probability of retention. This will result on some activations to be dropped from the network. During training, only the resulting subnetwork weights are updated, however all the neural network weights will be update in expectation during many epochs of training. This can be though as model averaging. Dropout is also used as a function regularizer which allows the model to generalize better to unseen data points and leads to faster parameter convergence.

As demonstrated by (Gal, 2016), any neural network optimized with dropout is equivalent to a form of approximate inference in a probabilistic interpretation of the model. This result is important because it shows that the optimal weights found by training a neural network with dropout will be the same as the ones found by optimizing a Bayesian Neural Network with variational inference. Therefore, this means that a

neural network trained with dropout already is a Bayesian neural network. The prior distribution on the neural network weights is related to the amount of noise of the mask.

One can get the estimates of the parameter uncertainty by doing T forward passes with T different sampled masks and then estimate the variance of the output of the model. This variance will be the model uncertainty or epistemic uncertainty. The total variance of the model will therefore be the epistemic variance plus the aleatoric variance.

At inference time one can decide whether to sample the dropout activation mask to make the output of the network stochastic. If the dropout is not used during inference, the weight matrices of the network must be scaled by the dropout probability p:

$$W_{test} = pW_{train}$$

To make a concrete example, the following figures represent a regression problem on a toy dataset where the independent variable follows a heteroskedastic gaussian distribution with linear mean and nonlinear variance. The first figure shows the training data point and the prediction from the estimated model.

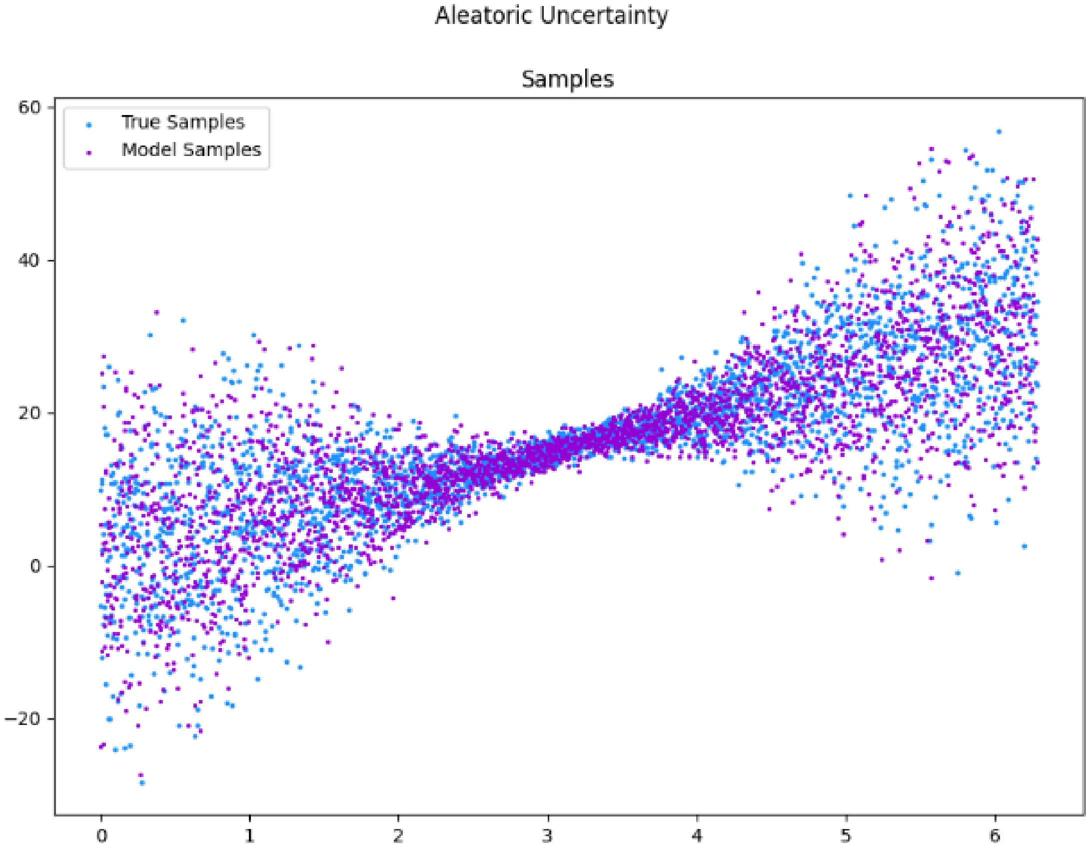


Figure 14 True and Predicted samples

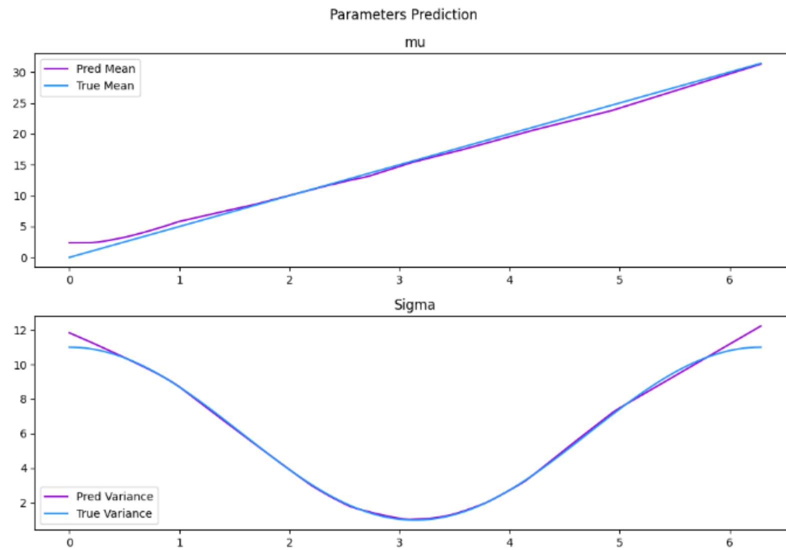


Figure 15 True and Predicted mean and variance without dropout during inference

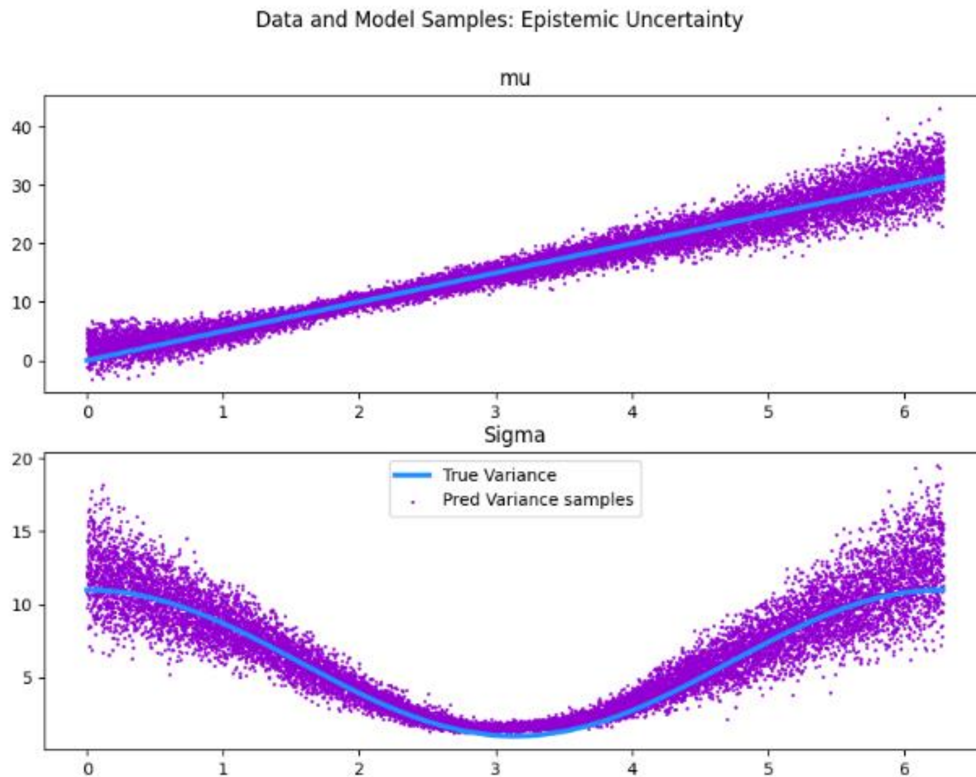


Figure 16 True and Predicted mean and variance with dopout

The second and third figures show the estimated and true parameter of the distribution. The function used to parametrize the mean and variance of the gaussian distribution is a deep neural network parametrized by the weight matrices and biases $\theta = \{W^l, b^l\}$. The figure on the left shows the uncertainty over the parameters of the model. For each

forward pass of the model a different set of parameters (μ, σ) are sampled which will then be used to parametrize a gaussian distribution:

$$\mu, \sigma \sim DNN(x;\ \theta)$$

$$y \sim N(x; \sigma, \mu)$$

The figure on the right instead does not use dropout during the inference stage and therefore the parameters of the output distribution are deterministic. As one can see in the left figure, the higher the variance of the data the more uncertain the parameters values are, however the mean of the distribution closely approximates the true mean of the parameters.

To get the total uncertainty estimate, we need to sample S_p number of parameters which will parametrize the zero-inflated distribution. Then one can proceed in two ways. The first is to sample S_d points and compute the mean and variance of the data points for each sampled coefficient.

$$W_{s_p}, V_{s_p}, P_{s_p} \sim Model(A_t, X_t; \theta, M)$$

$$E_{s_d} \sim Lognormal(W_{s_p}, V_{s_p}, P_{s_p})$$

$$\mathbb{E}[E] = \frac{1}{S_{tot}} \sum_{S_{s_d}} E_{s_d}$$

$$\mathbb{V}[E_s] = \frac{1}{S_{tot}} \sum_{S_d} \left(E_{s_d} - \mathbb{E}[E]\right)^2$$

Where $S_{tot} = S_p * S_d$.

The second method is to compute the expected mean and variance of the final distribution by taking a Monte Carlo estimate of the mean and variance of the aleatoric uncertainty:

$$\mathbb{E}[E] = \frac{1}{S_p} \sum_{S_p} P_{s_p}\ exp(W_{s_p} + \frac{1}{2} V_{s_p})$$

$$\mathbb{V}[E] = \frac{1}{S_p} \sum_{S_p} P_{s_p}^2\ exp(2W_{s_p} + 2V_{s_p})$$

The second method is the preferred method since it requires a smaller number of samples (Vandal, Kodra, Dy, & Ganguly, 2018).

8 MODEL EVALUATION

To assess the model performance, one needs to define a diverse set of metrics that can capture the predictive capabilities of the model. These metrics can either be computed during the training phase to check whether the optimization process is running correctly or can be computed during the evaluation phase there the optimization process is stopped, and the model is evaluated on unseen data.

The usual way to evaluate a model is to split the dataset in three blocks. The first set which will contain most of the data is the training set. The model will be optimized on this training set. The second set is called the validation set where in each iteration of the optimization procedure the model will be evaluated on this set to check whether the model is overfitting to the training set. The third set is the test set which will be used to evaluate the model on completely unseen data. The difference between the validation set and the test set is that if the modeler tunes some hyperparameters of the model, for example the learning rate, the number of layers or the activation function, the model could still overfit to the test set even though the model itself has not been trained on it. Therefore, the test set is used to evaluate the performance of the model on completely unseen data. One can then check the discrepancy between the evaluation metrics computed on the different dataset splits and check whether the model overfits to the training set and how well it generalizes to unseen data.

Since the input data of the model is an ordered set of adjacency matrices, to implement the training, validation split we mask some of the edges of each input adjacency matrix A_t and output adjacency matrix A_{t+1} by sampling a binary matrix with probability p:

$$M_t^{train} \sim Bernoulli(p_{train}) \ \ and \ \ M_t^{test} = 1 - M_t^{train}$$

$$A_t^{train} = A_t \odot M_t^{train} \ \ and \ \ A_t^{test} = A_t \odot M_t^{test}$$

We sample $T - t_{test}$ binary masks and leave the next t_{test} time steps for the test set.

The following paragraphs will introduce two types of evaluation metrics. The first type of measures is related to the regression task, where the analysis is focused on the

distance between the true edge weight and the sampled edge weight. The second type of measures are related to the classification task, where the analysis if focused on the predictive distribution of the existence of an edge.

8.1 REGRESSION

The regression metrics measure the distance between the predicted value and the true value. The closer the forecasted values are from the actual outcome the lower the distance and the better the performance of the model. Since the values of the edges are lognormally distributed, for each metric we take the logarithm of $A_{ij} = \log(A_{ij})$

8.1.1 Mean Absolute Error

The mean absolute error (MAE) measures the absolute distance between the values of the true adjacency matrix A_{ij} and the predicted adjacency matrix $\hat{A}_{ij}$. Every difference between the values will have the same weight.

$$MAE_t = \frac{1}{k^2}\sum_i \sum_j |A_{ij} - \hat{A}_{ij}|$$

8.1.2 Root Mean Squared Error

The root mean squared error (RMSE) measures the squared distance between the values of the true adjacency matrix A_{ij} and the predicted adjacency matrix $\hat{A}_{ij}$. Large values of the error will have proportionally larger impact in the score than small error values.

$$RMSE_t = \sqrt{\frac{1}{k^2}\sum_i \sum_j \left(A_{ij} - \hat{A}_{ij}\right)^2}$$

8.1.3 Accuracy

The accuracy measures the precision of the regression. The higher the accuracy the higher the model performance. In the numerator the Frobenius norm is used to score the difference between the true and predicted adjacencies. The Frobenius norm is the square root of the sum of the absolute squared values of the errors:

$$ACC_t = 1 - \frac{||A - \widehat{A}\,||_F}{||A||_F} \quad where \quad ||A||_F = \sqrt{\sum_i \sum_j |A_{ij}|^2}$$

8.1.4 Explained Variance

The explained variance calculates the percentage of the variance that is explained by the model. The higher the score the better the model captures the variability in the data.

$$VAR = 1 - \frac{\mathbb{V}[A_{ij} - \widehat{A}_{ij}]}{\mathbb{V}[A_{ij}]}$$

8.1.5 Edge Distribution

The distribution of edge weights is a simple way to visualize the discrepancy between the predicted values and the true values. From this plot one can see whether the model has a bias and how well the model approximates the true probability distribution of the data.

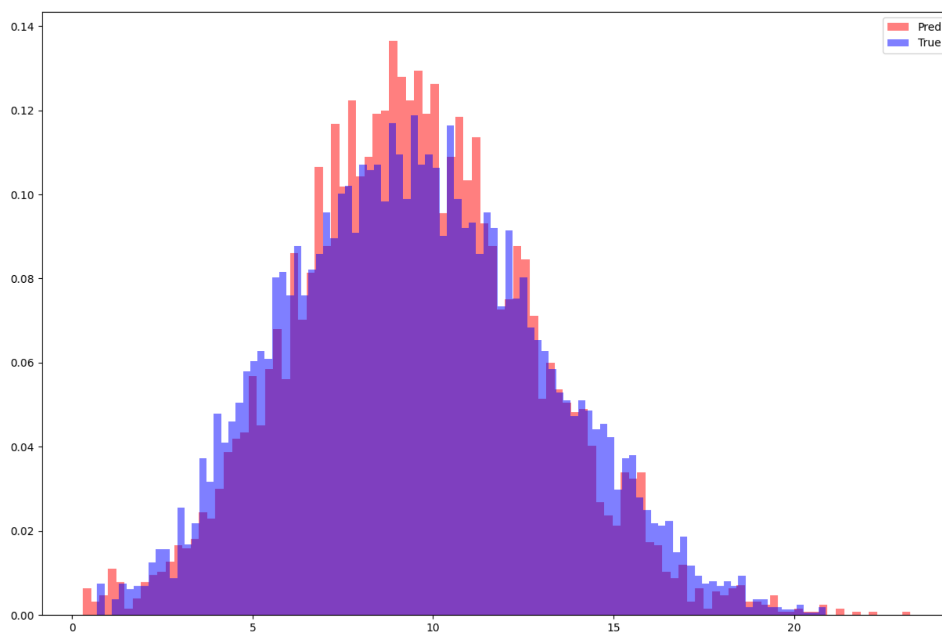


Figure 17 Example of true and predicted trade

8.2 CLASSIFICATION

Classification metrics evaluate different aspects of a classifier. Since the model developed has a component that predicts the probability of the existence of an edge between two nodes, the model can be considered a binary classifier.

8.2.1 ROC curve

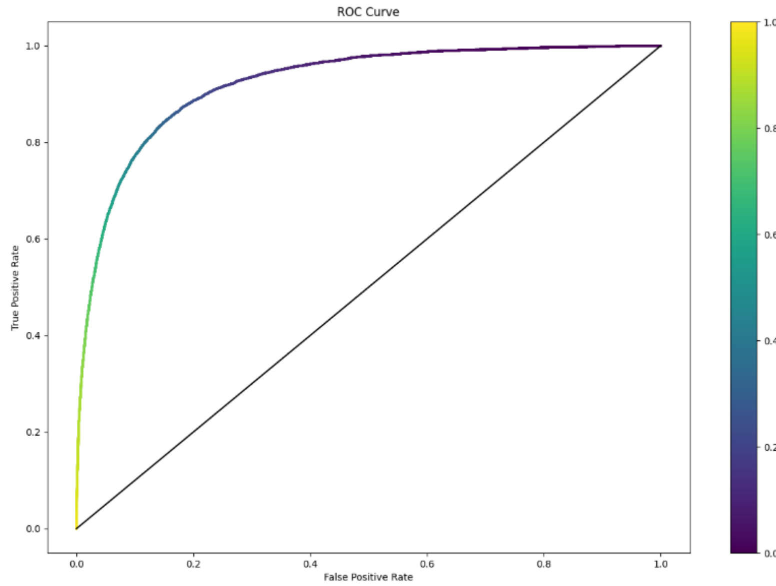


Figure 18 Example of ROC curve

The Receiver Operating Characteristic (ROC) curve is a chart that evaluates the performance of a binary classifier by varying the classification threshold. The False Positives Rate (FPR) is plotted on the x axis and the True Positive Rate (TPR) is plotted on the y axis. The TPR and FPR are defined as:

$$TPR = \frac{True\ Positive}{Total\ Positive\ Prediction} = \frac{True\ Positive}{True\ Positive + False\ Negative}$$

$$FPR = \frac{False\ Positive}{Total\ Negative\ Prediction} = \frac{False\ Positive}{False\ Positive + True\ Negative}$$

Every point on the curve corresponds to a different classification threshold. The higher the threshold the lower the TPR will be since only the data points classified with a high probability will be considered and many true positives will be excluded. The higher the classification threshold the lower the FPR will be, since only points on which the classifier is sure of the prediction will be included in the set. The opposite effect is true for low threshold values.

The optimal point on the curve is the point minimizing the distance between (0,1), which is the point where the FPR is zero and the TPR is 1 (Unal, 2017). For each point the distance between the point and the optimal value is computed and the point associated with the lowest distance is returned.

8.2.2 Precision Recall Curve

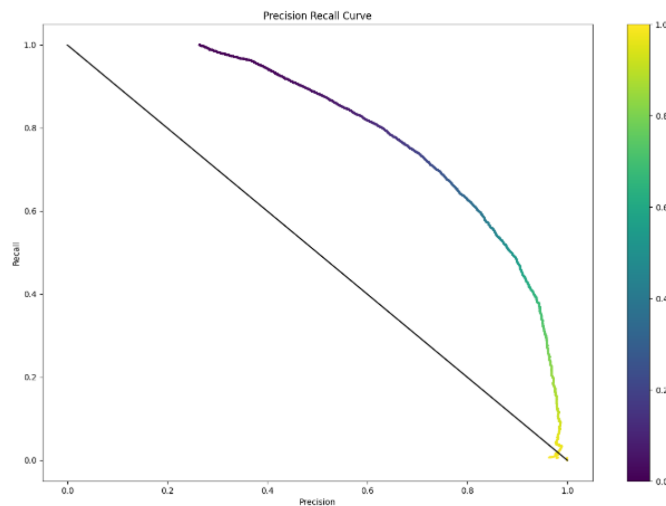


Figure 19 Example of Precision and Recall Curve

The Precision-Recall Curve shows the tradeoff between the precision score and the recall score for different level of classification thresholds. The precision score is the number of true positives i.e. the number of points correctly classified as positive, divided by the total number of positives i.e. the total number of points the classifier classified as positive:

$$Precision = \frac{TP}{TP + FP}$$

The recall measures the number of true positives divided by the number of actual positives. It measures how many items from the total set the classifier managed to identify:

$$Recall = \frac{TP}{TP + FN}$$

8.2.3 Confusion Matrix

The confusion matrix shows the number of true positives, true negative, false positive and false negatives. The off-diagonal elements are misclassified points. The y axis represents the true label of the point and the x axis represent the predicted label from the model. The upper left element of the confusion matrix represents the number of True Negative training points, which are the missing edges which were correctly classified as missing edges. The lower right element represents the number of True Positive edges, which are the edges that were correctly classified as belonging to the graph. The upper right elements represent the False Positive edges, which are the edges that were classified as present but are missing from the graph. The lower right elements represent the number of False Negatives edges, which are edges that were classified as missing but are present in the graph.

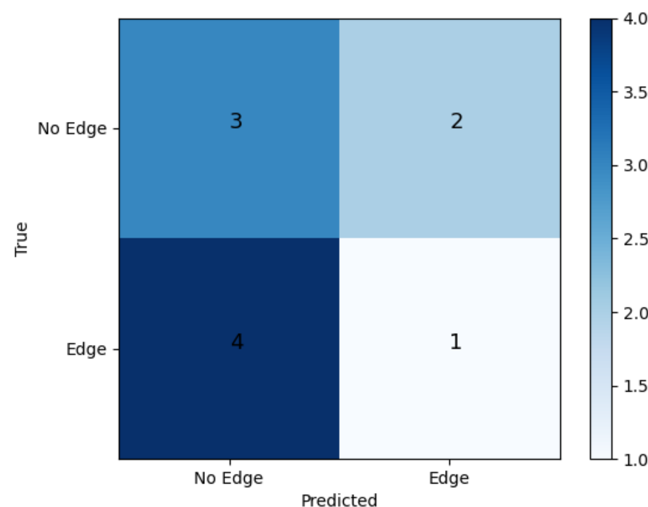


Figure 20 Confusion matrix example

8.2.4 Expected Calibration Error

The expected calibration error measures how much the probability estimated from the model match the actual frequency distribution of the accuracy of the model (Guo, Pleiss, & Weinberger, 2017). This metric measures the distance from the confidence and accuracy of the probability output of the model. If the model makes predictions with high confidence but the accuracy for those predictions is low, then the model is said to be mis calibrated. The expected calibration error will then take this measure in expectation. One can estimate this expectation by taking the mean of the difference

between the probability output and the accuracy relative to each probability. Formally the expected calibration error is defined as:

$$\mathbb{E}_P\left[\left|\mathbb{P}\left(\hat{Y} = Y \middle| \hat{P} = p\right) - p\right|\right]$$

In practice the first is estimated by counting the accuracy how many of the predicted points which fall in a small interval on the probability support where correct. The second term is the average confidence within a small interval bin.

9 EMBEDDING VISUALIZATION

To visualize the high dimensional node embeddings, representing the countries characteristics, coming from the RNN model we utilize a technique called T-Distributed Neighbor Stochastic Embedding (t-SNE) (Maaten & Hinton, 2008). This dimensionality reduction technique tries to minimize the LK-divergence between distribution of the high dimensional inputs x_i and lower dimensional outputs y_i:

$$p_{i|j} = \frac{\exp\left(-||x_i - x_j||^2/2\sigma_i^2\right)}{\sum_{k \neq i} \exp\left(-||x_i - x_k||^2/2\sigma_i^2\right)}$$

$$q_{i|j} = \frac{\exp\left(-||y_i - y_j||^2/2\sigma_i^2\right)}{\sum_{k \neq i} \exp\left(-||y_i - y_k||^2/2\sigma_i^2\right)}$$

$$C = \sum_i KL(P_i||Q_i) = \sum_i \sum_j p_{i|j} \log\left(\frac{p_{i|j}}{q_{i|j}}\right)$$

Where $p_{i|j}$ and $q_{i|j}$ are the conditional probabilities of the higher and lower dimensional input respectively. We utilized the Scikit-Learn python library (Pedregosa, 2011) to get the lower dimensional embeddings.

We then utilized the k-means algorithm clustering algorithm (X. & J., 2011) to cluster the embedded low dimensional nodes. K-means first randomly distributes some centroid points which are the points which represent the center of the clusters. Then for each iteration it calculates the distance between each of the points to the centroid and assigns a cluster to each point based on the closets cluster. It then updates the centroid based on the average position of the points assigned to the cluster and iterates until convergence. The final result will be the centroid and the points associated to the centroid.

10 DATASET

The source of the data used for the analysis comes from the UN Comtrade database (United Nations. UN Comtrade , s.d.). UN Comtrade is the most used database for annual data on international merchandise trade statistics detailed by commodity and partner. From the databased we can extract the amount of trade by dollar value and the quantity exchanged by the imported and exporter. In the database the countries can have either an exporter role or an importer role. Since the export of one country is the import of the other the two values should match. In realty this is not the case since the imports are measured at CIF, which is the Cost, Insurance and Freight Import Value. This value includes import charges and customs fees. Exports are instead measured as FOB, which is the Free On-Board values. This value includes loading and transportation costs but does not include customs fees and import charges (Methodology Guide for UN Comtrade User on UN Comtrade Upgrade 2019, 2019). The merchandise in the database follows two different reporting standard codes, the HS standard, the SITC standard and the BEC standard. The HS and SITC standards have different codes based on the year revisions. We utilized the SITC standard with revision 1 because it allows to gather data from 1964 to 2019. The countries codes follow the ISO3 naming convention (https://www.iso.org/, 2020). One issue with this dataset is that the dataset is not homogenized, and the actual trade values may be different than the ones reported. Furthermore, the values of the trade below e^5 are missing from the years before the year 2000. This leads to inaccurate estimates of the level of trade and higher variance in the prediction.

11 RESULTS ANALYSIS

In this section we will first explore the ITN structure with the network measures previously introduced and compare the measures of the predicted network estimated from the model and the actual network measures. We will the follow with the analysis of the predictive performance of the model by visualizing the regression and classification metrics.

To understand the main players in the international trade network we plot the total sum of import and export of each country in the dataset, from 1995 to 2019, and rank the countries from the biggest to the lowest trader.

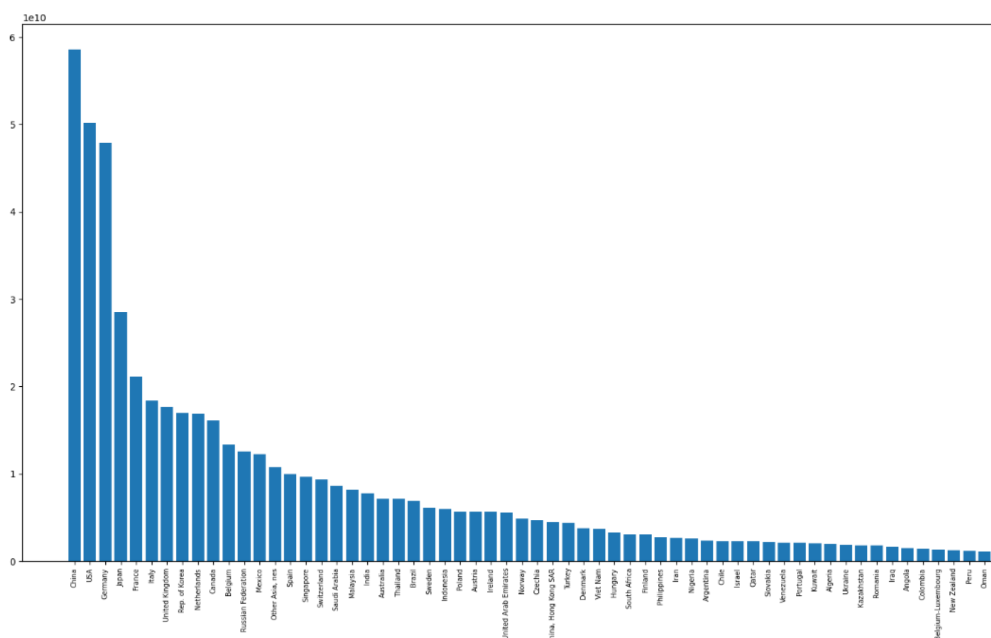


Figure 21 Total sum of trade ranked by country

From this plot we can see that China has surpassed the United States in terms of total goods traded. In the second place we find the United States, followed by Germany, Japan, France, Italy and the United Kingdom. From the graph we can see that the total trade follows an exponential curve. We will now show true and predicted adjacency matrix from corresponding to a randomly sampled product from the dataset. We will analyze more in details the performance of the model in the subsequent paragraphs.

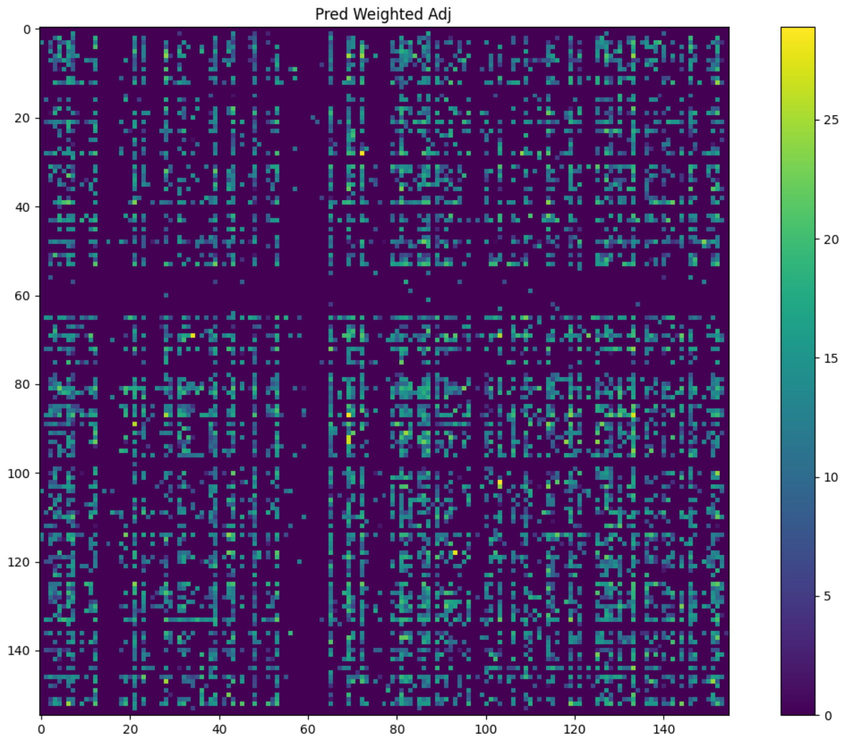


Figure 22 Predicted Adjacency Matrix

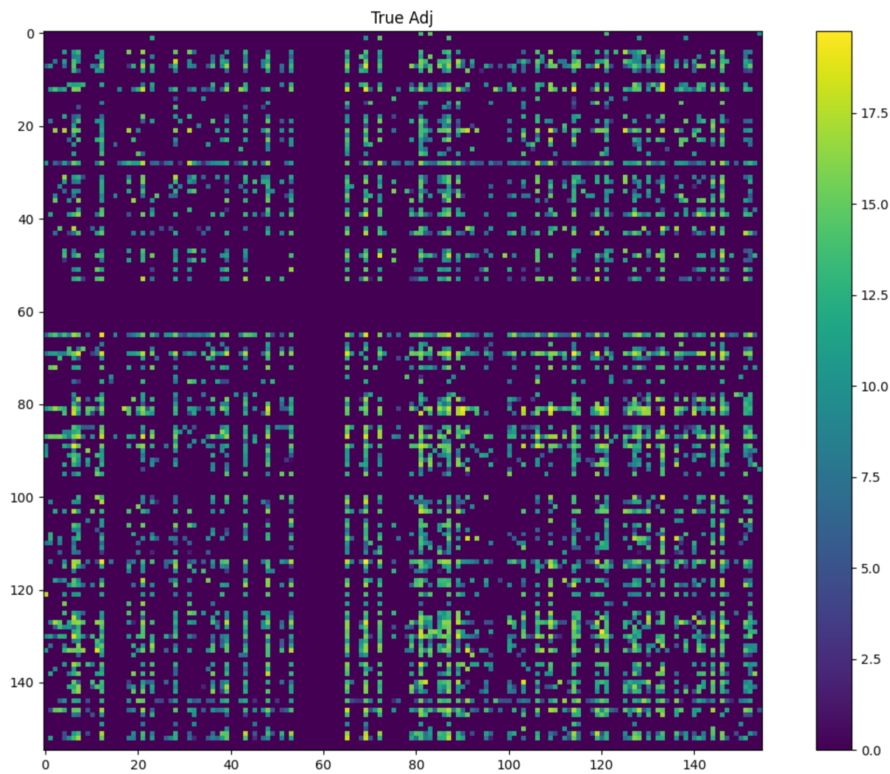


Figure 23 True Adjacency Matrix

From the two images we can see that the model is able to reconstruct the adjacency matrix of the next time steps. The zero lines in the middle that cross the plot are countries that no ceded to exist. However, the model still sampled some points from the missing countries because we did not explicitly model the entrance or exit probability of the countries.

The following plot shows the distribution of the in-degree, out-degree and closeness centrality of the network, relative to different years, and how they evolve through time.

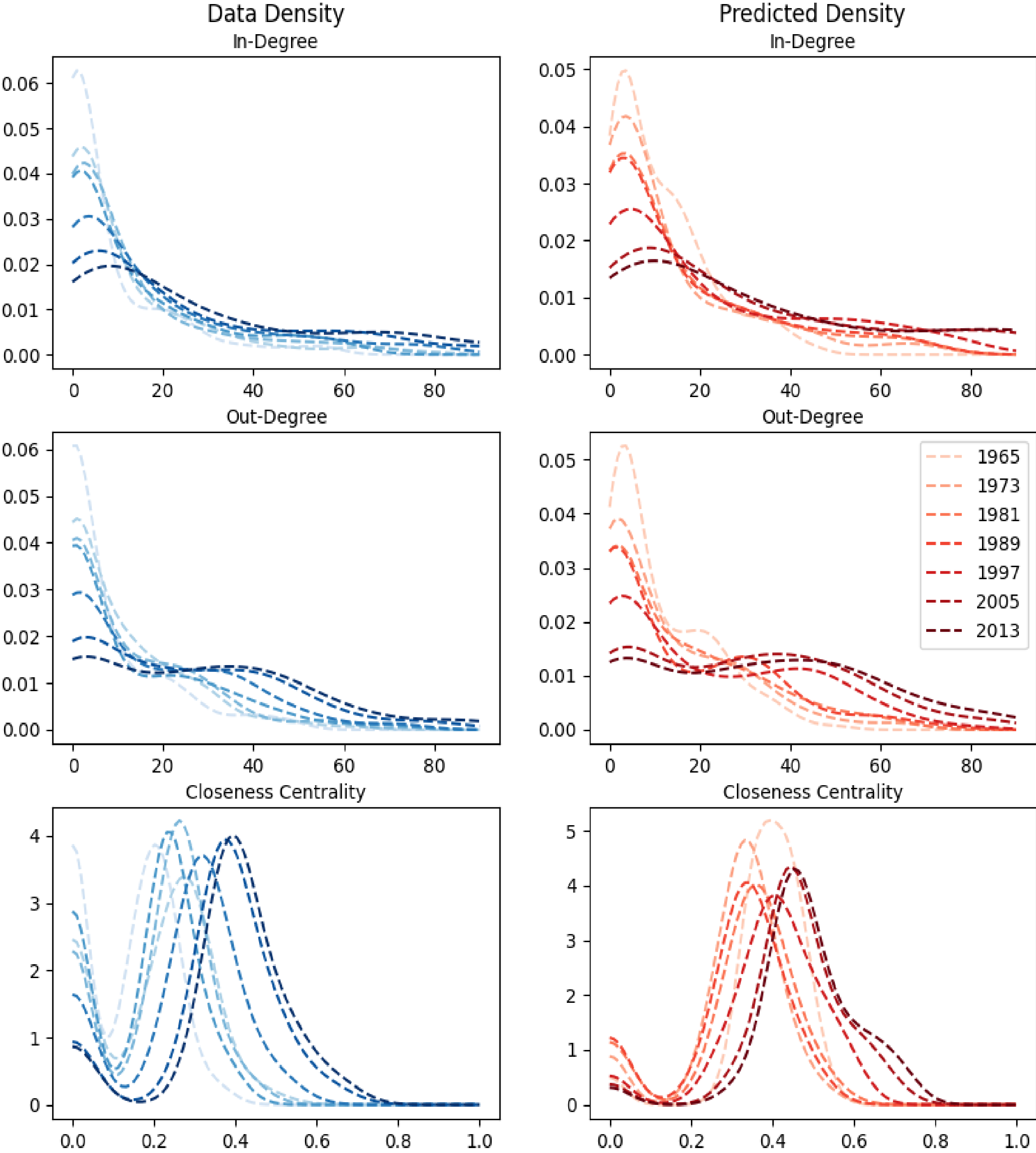


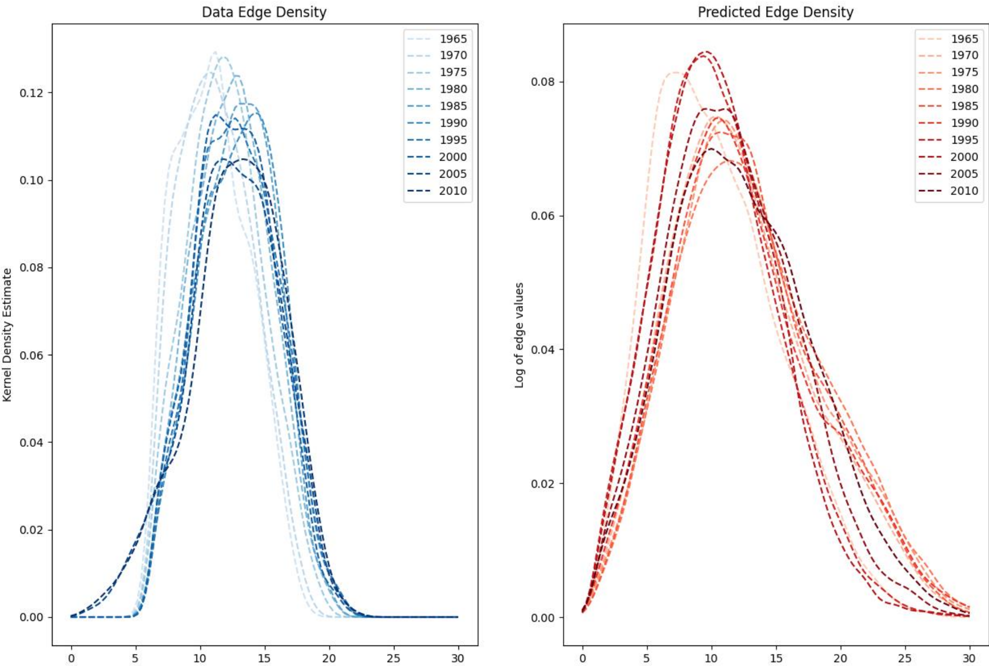
Figure 24 In Degree, Out Degree and Closeness Centrality from the true data and model samples

As we can see from the plots, all statistics tend to shift to the left, indicating the increase of each measure through time. For example, the increase in in-degree shows that the countries of the network tend to increase the amount of product they import.

The same reasoning applies to the out-degree measure. The closeness centrality shows that the countries become more integrated through time and the path from one country to the other gets shorter in the space of the network, even though the position of the country stays the same in the spatial coordinates. This signifies that the trade increases through time as the countries become more interconnected with each other. The closeness in 1965 was 0.2 and in 2013 was 0.4 on average. This means that the countries were twice as close in 2013 as they were in 1965.

We can see that the measures derived from the sampled adjacency matrix follow closely the actual node statistics.

Another statistic which shows the evolution of the international trade network is the evolution of the edge distribution. The following plot shows the edge distribution 1965 to 2010.



From the plot we ca see that the mean of the distribution shifts to the right, indicating that on average the value of trade increases. One can also notice that the data from the

Comtrade dataset does not have the values below e^5 before the year 2000. This induces a bias in the model as the distribution is truncated.

The following plots show the evolution of the Jaccard centrality measure which measures the similarity between two nodes in the network by comparing the neighborhoods of the nodes.

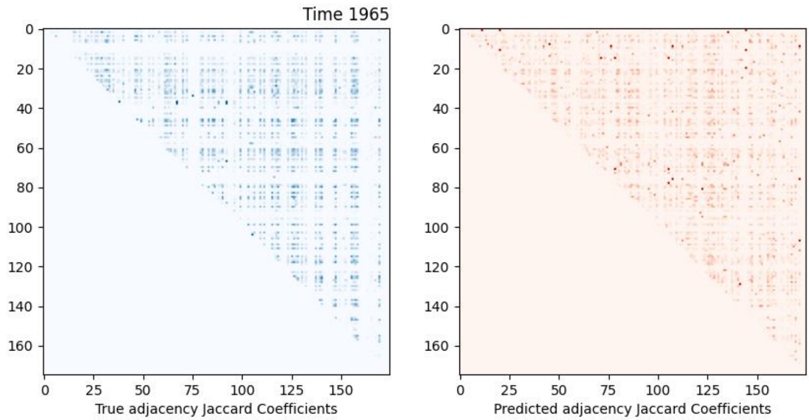


Figure 25 Jaccard Centrality for 1965 for Comtrade raw data

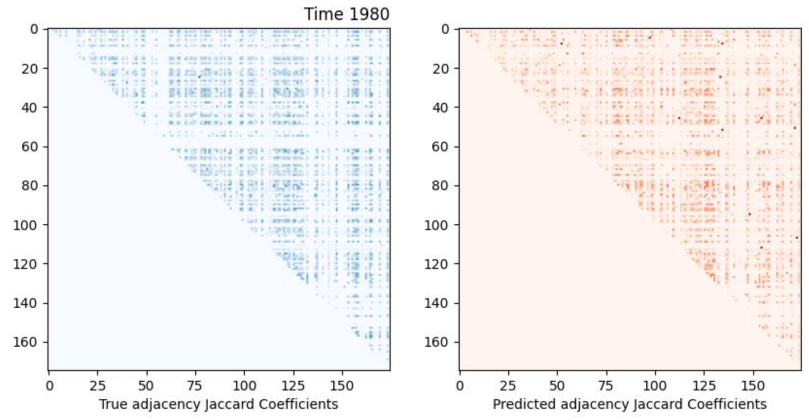


Figure 26 Jaccard Centrality for 1980 Comtrade raw data

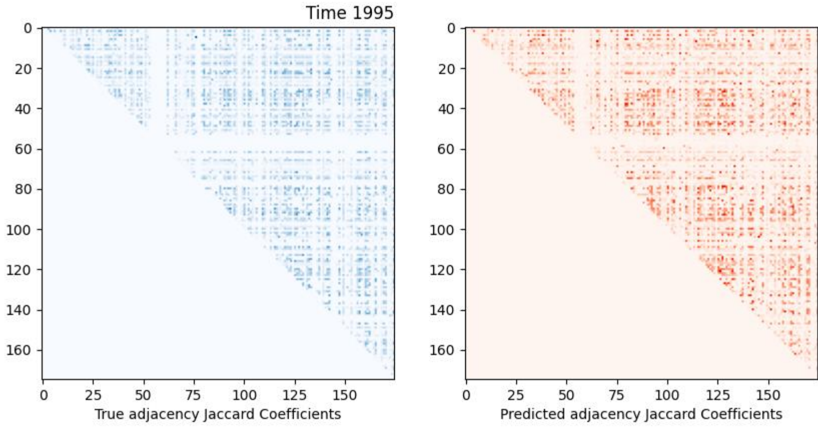


Figure 27 Jaccard Centrality for 1995 Comtrade raw data

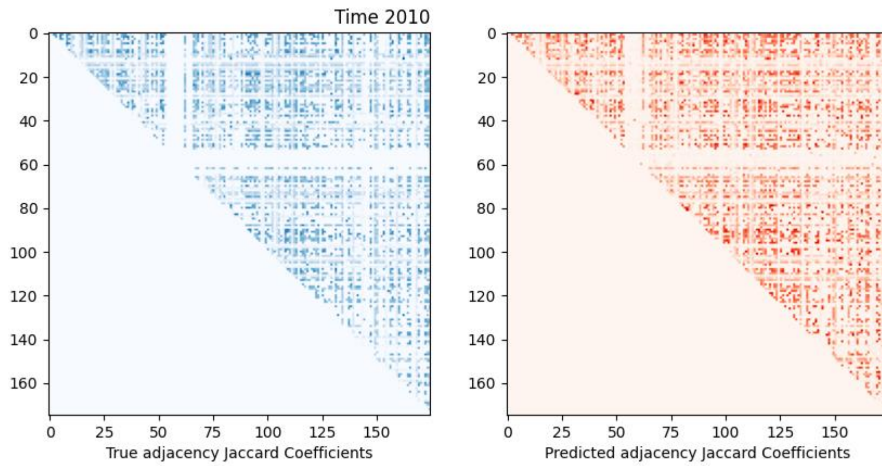


Figure 28 Jaccard Centrality for 2010 Comtrade raw data

The x and y axes represent the indices of the countries. Each cell of the matrix represents the similarity score between the two countries.

The more intense colors signify a higher centrality measure. By visually inspecting the plots we can see that as time progresses, the nodes in the network become more similar. This is because the increase in connection of the networks will inherently lead to neighbors from different countries to interact with each other, thereby increasing the similarity score between the corresponding neighbors. We can also see that the model captures these interactions quite well as the more strongly colored cells in the true network are closely matches with the predicted network.

The next plot represents the increase on the network connectivity through time, validating the argument that the trade between countries increases as time progresses.

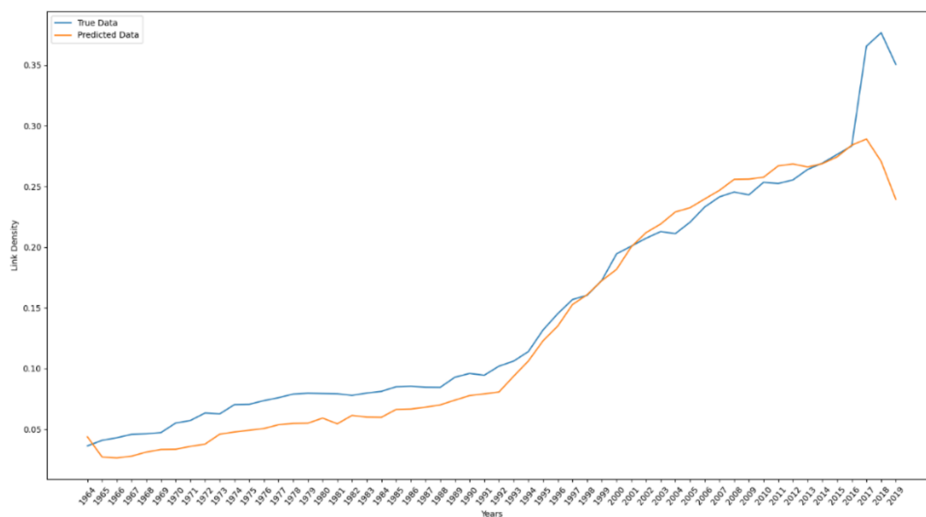


Figure 29 Link Density evolution for predicted and true data

This plot shows the link density measure and how it changes through time. We can see that the model approximates well this metric. However, in the last three years the two measures diverge. This is because we trained the model up to 2012 and the following years were taken as the validation set. We can also see that the model underestimates the link density up to 1997. This is because the adjacency matrix for the year leading to the beginning of the millennium were extremely sparse and therefore the network head that acts as the classifier of the presence of the edges between two nodes is biased to predict zero edges. We will later show this effect when analyzing the classification performance of the model. One can attribute the increase of linkages to the decrease of transaction cost of trading between distant countries, the lower transportation costs due to the economy of scale of the international trade and the technological advancement (Maluck & Donner).

Since each country in the network is represented by a high dimensional vector, we applied the t-SNE dimensionality reduction described in the 9th chapter to project the high dimensional embeddings to a 2-dimensional plane. From the plot we can see countries belonging to the same continent are clustered together. This is relevant because it signifies that the algorithm understood the structure of the network and the distance between countries without being fed that information as input. For reference, we first show the initial embedding corresponding to an untrained model and then visualize the embeddings from the trained model.

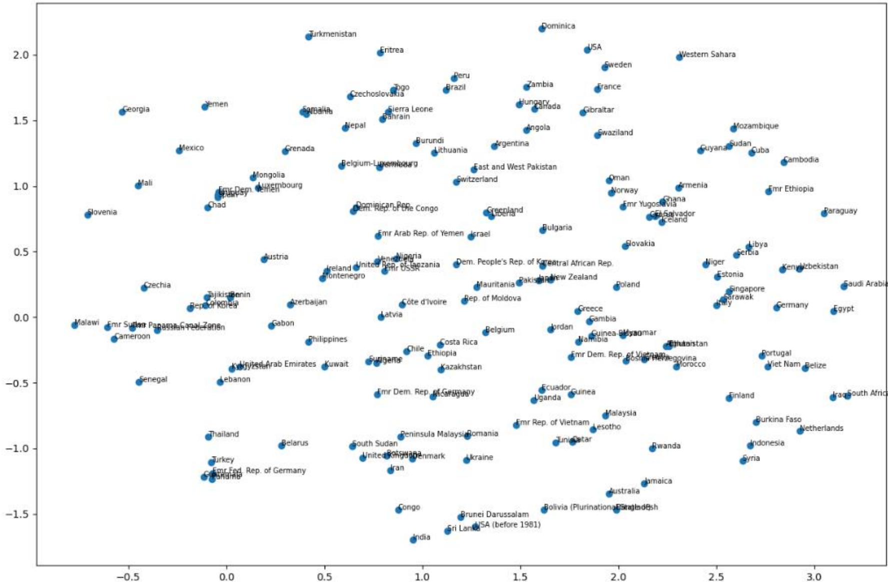


Figure 30 Embeddings from the Untrained Model

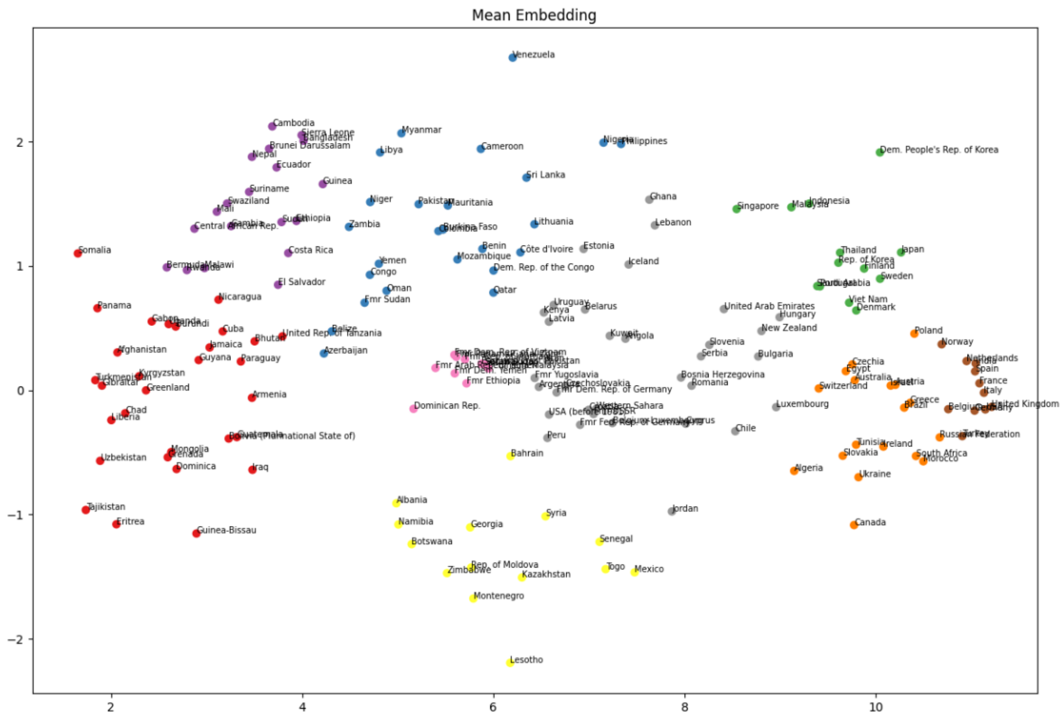


Figure 31 Node Embeddings of Countries

The following paragraphs will focus more in detail the prediction performance of the algorithm. We will show the metrics related to the training and test set. Then we will show the regression and classification performance. To finish we will present the uncertainty over the parameters of the predictive distribution for a random subset of edges.

We first begin by showing the training curve of the model for every training epoch.

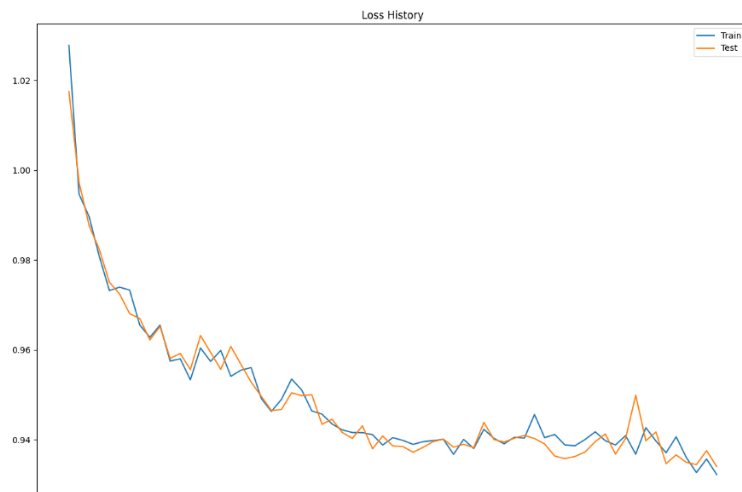
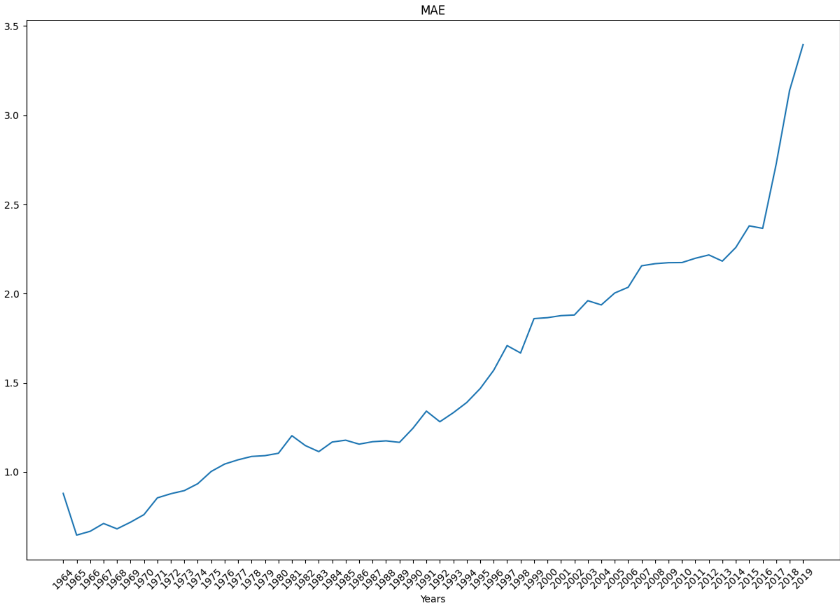
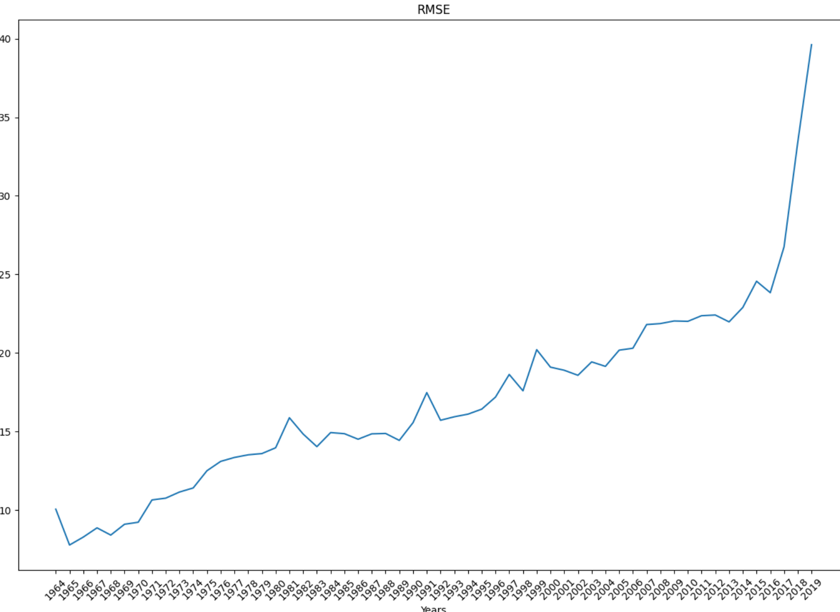


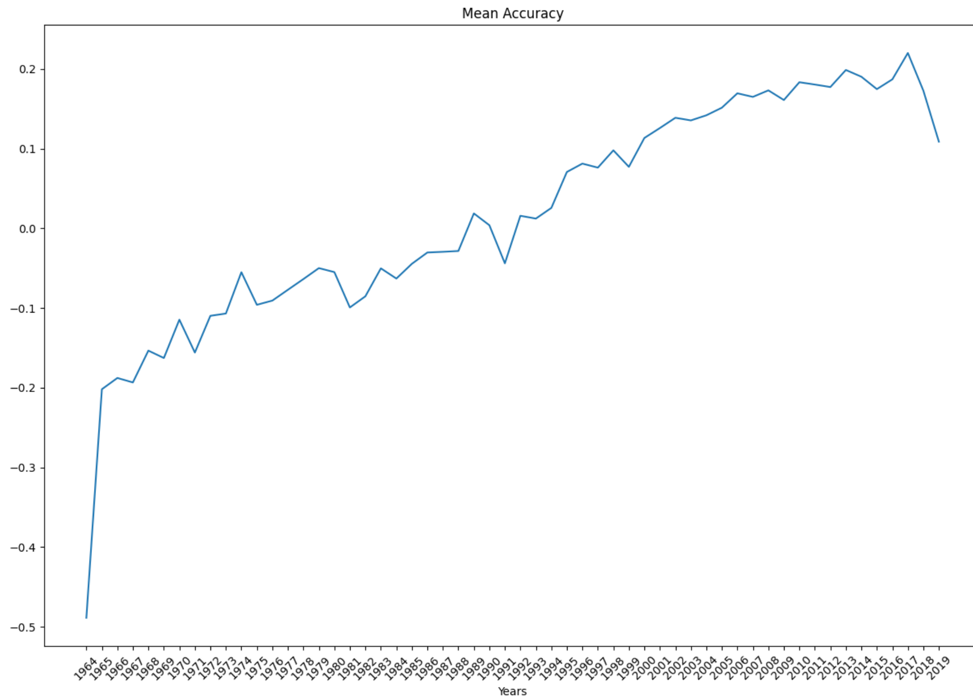
Figure 32 Loss history for training and test set

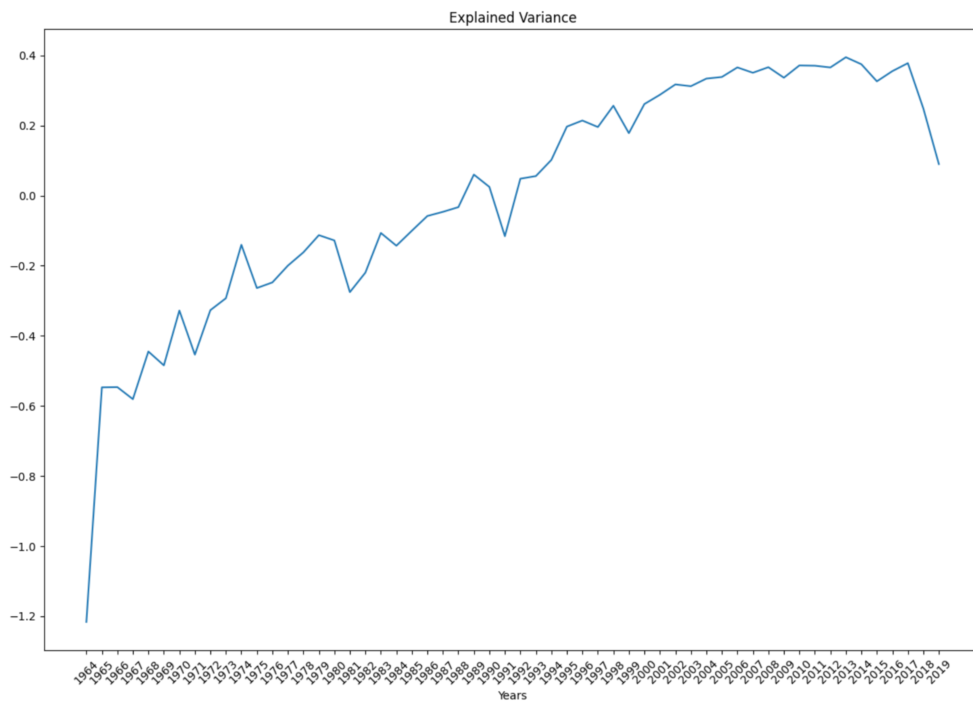
The plot above shows the loss function value for each training epoch corresponding to the training and test set. We can see that the loss decreases as the network updates its parameters. For this particular training setup we utilized a 50% split between the training and test edges. We sampled a mask of ones and zeros with probability 50% for each true adjacency matrix and masked the values the values of the matrix entries. We can see that the two values of the loss function match closely, which signifies that the model can generalize well on unseen edges, thereby reconstructing well the original adjacency matrix and predicting the missing edges. We trained for a total of 60 epochs.

We will now present the regression metrics for the edge values through time.



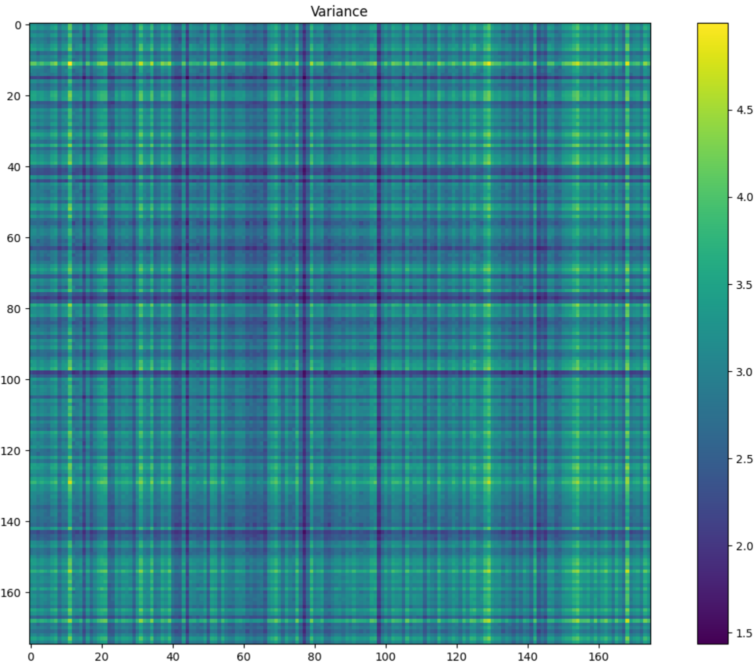




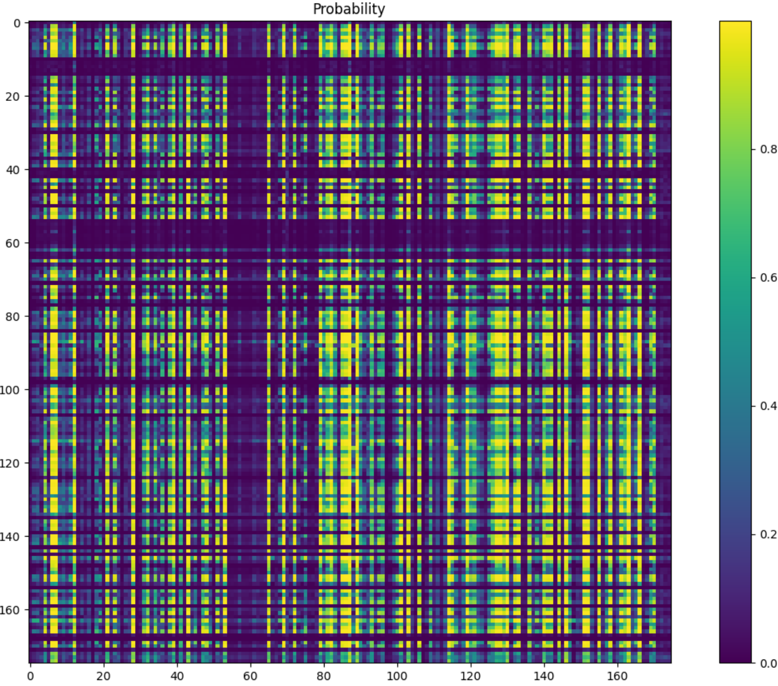


From the plots we can see that the RMSE and the MAE increase over the years. This is because as the non-zero edges increase, there is a higher probability for the regression to predict wrong values. However, we can also notice that the mean accuracy and the explained variance increase over the years.

We then evaluated the performance of the classifier and regressor on the validation set.



The above plot displays the variance estimated for each edge of the network. We can see that missing countries have low estimated variance since they do not transact with any country. The following plot shows the predicted probability matrix for the last time step.



The following plots show the ROC curve and the Precision curve for the last time period. We can see that the area under the curve og the ROC curve is high, hoverver the optimal classification thershold is lower than 50%. This signifies that the model underestimate the presence of links between countries . The optimal point is shown in red.

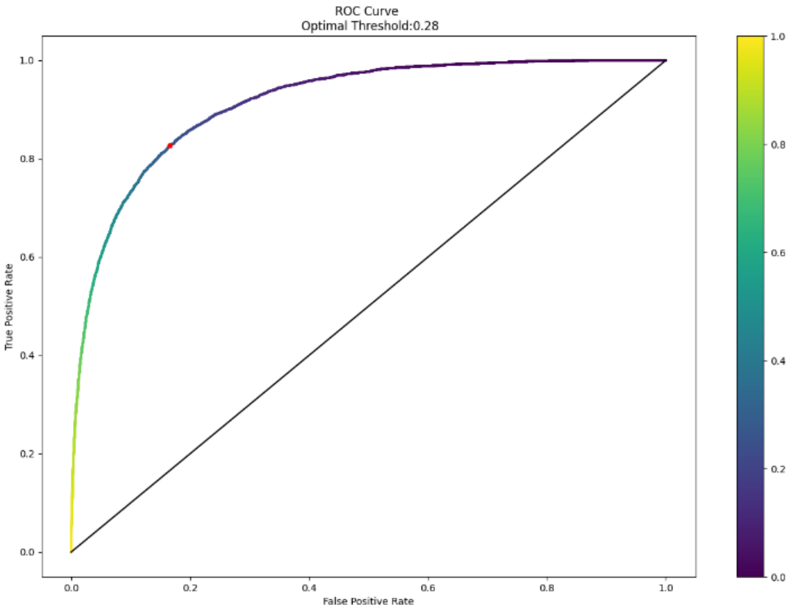


Figure 33 ROC Curve for Comtrade raw data

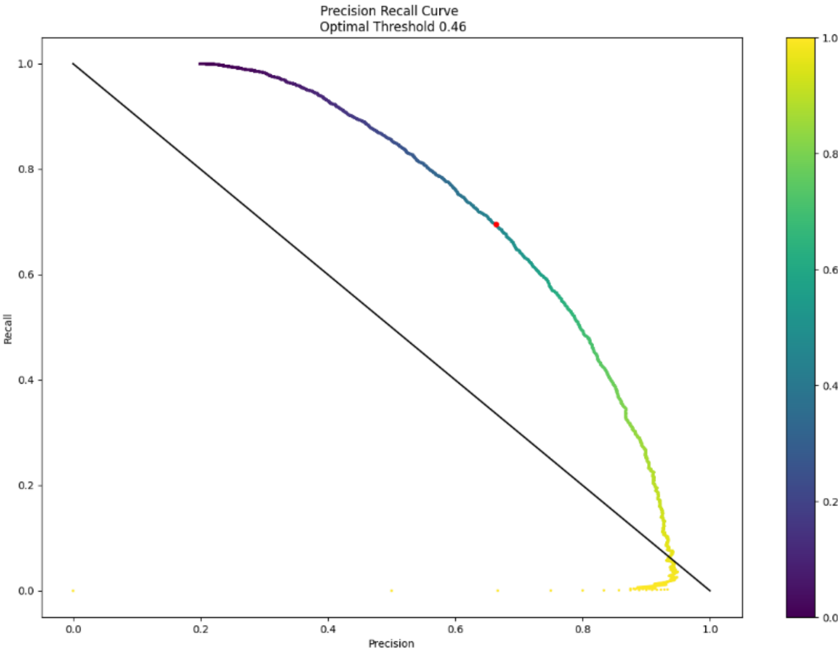
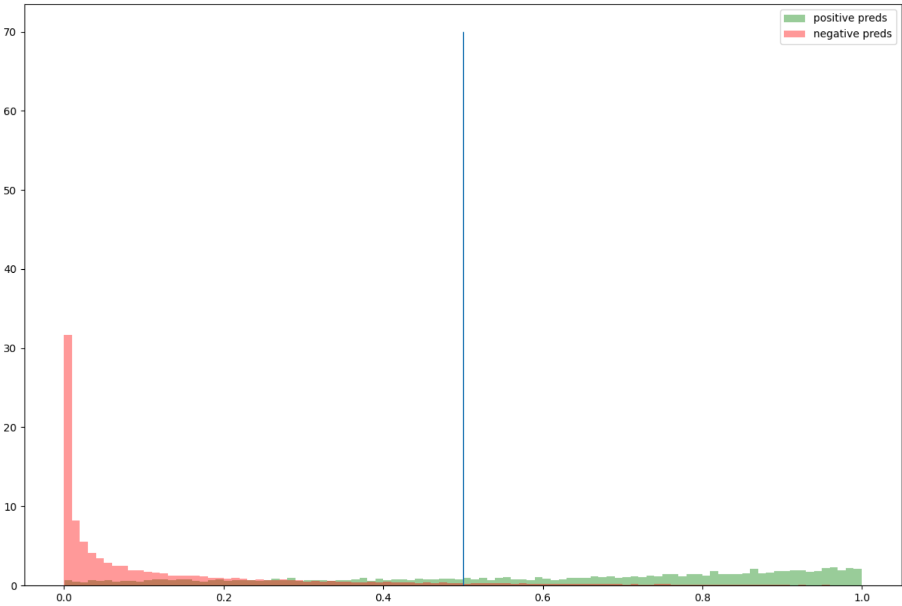
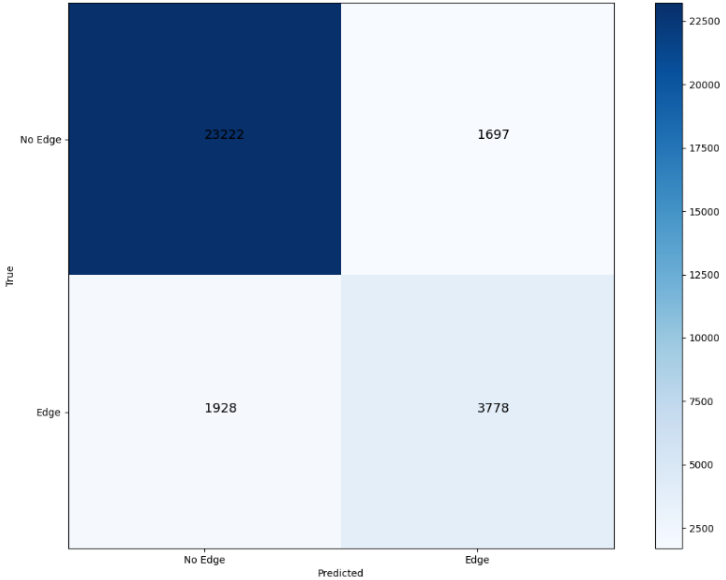


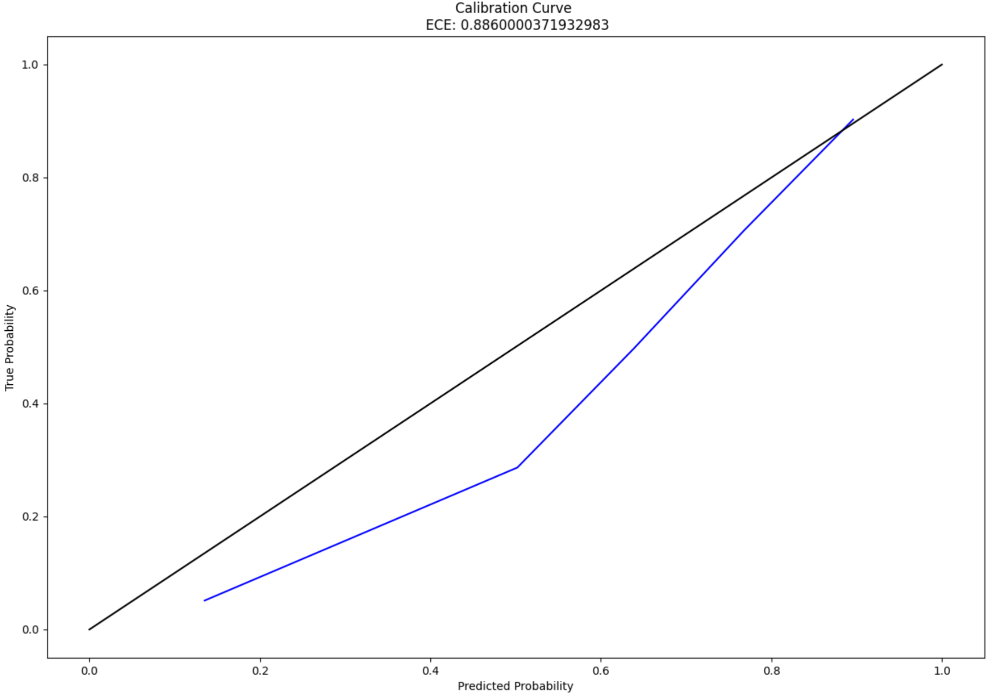
Figure 34 Precision Recall curve for Comtrade raw Data

The following two plots show the frequency distribution of the probabilities predicted from the classifier for the negative edges and for the positive edges. The overlap between the two frequency distributions are the misclassified examples. We can see that the classifier does not have high certainty on the positive edges. This is due to the sparsity of the adjacency matrix.





The following plot shows the calibration curve for the classifier. The closer the curve is to the diagonal line the more the classifier is calibrated. We can see that the classifier is not well calibrated and the average accuracy does not resamble the average predicted probability.



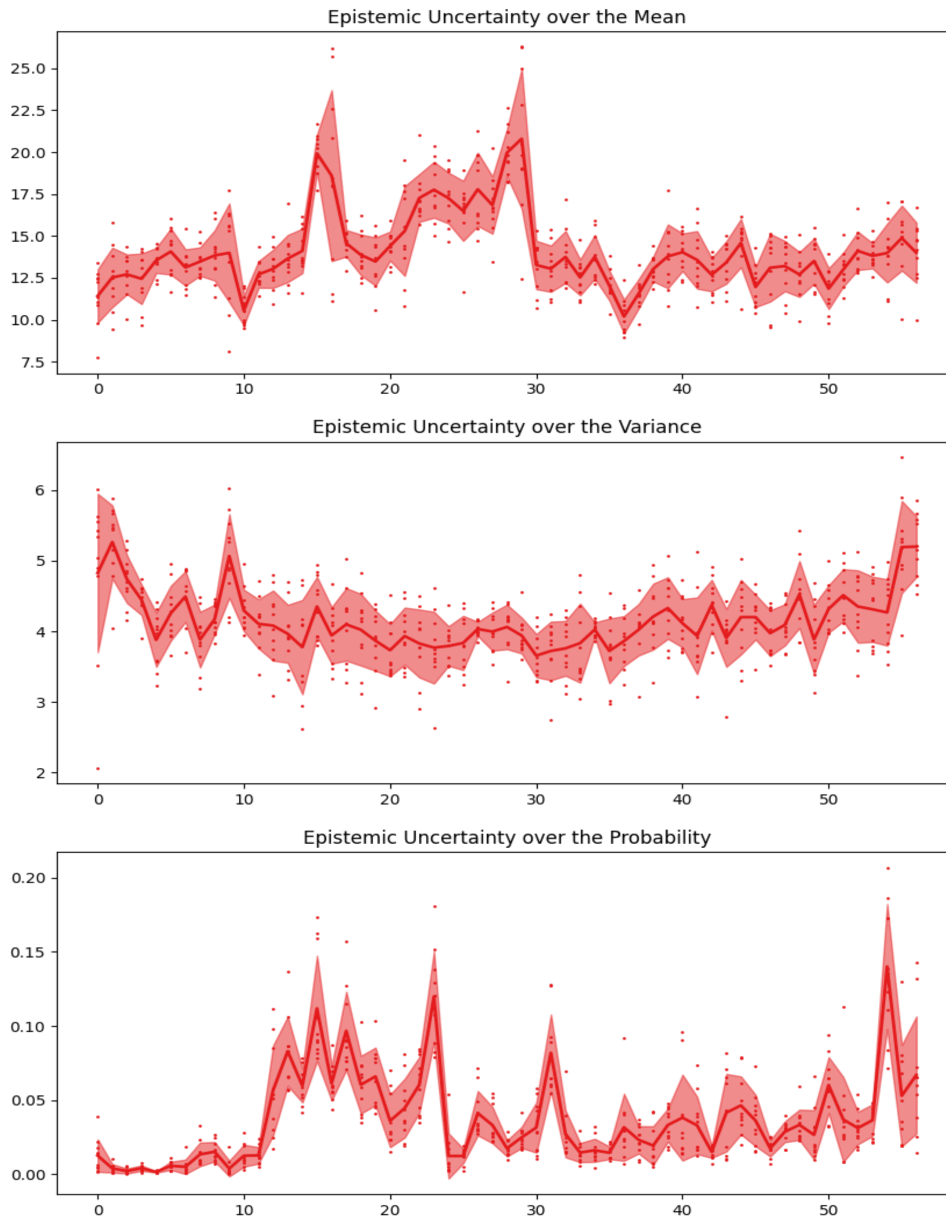


Figure 35 Uncertainty over parameters

The plot above shows the mean and uncertainty distribution for the mean, variance and probability parameters for a random sample of edges. The confidence interval is one standard deviation wide.

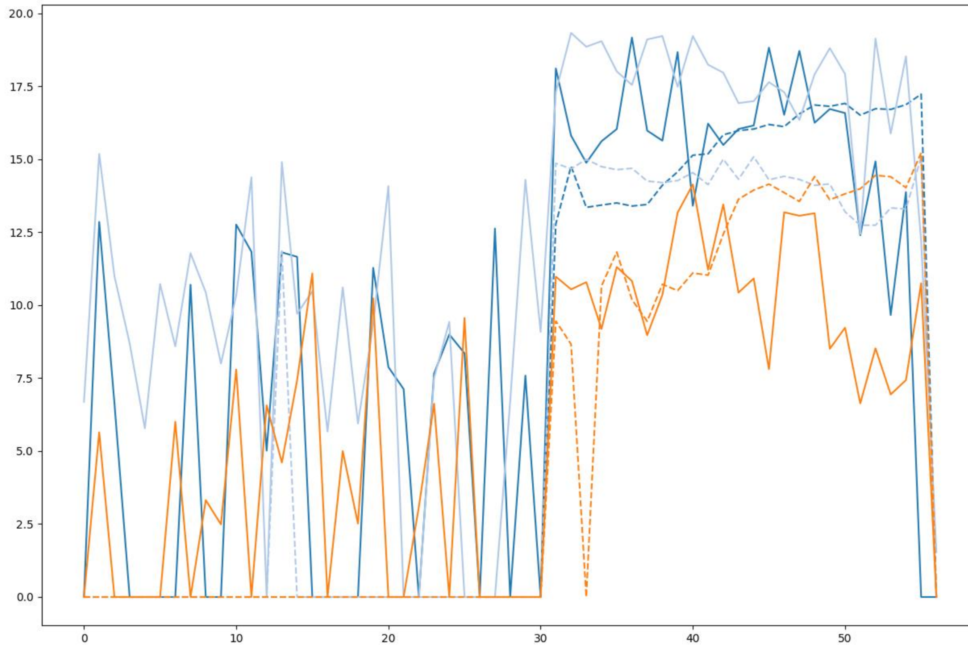


Figure 36 Prediction Samples

This plot shows a prediction sample from the model compared to the actual data. The actual data is plotted with the dashed lines and the model sample with the continuous lines. We can see that there is high variability in the prediction from the model. This can stem from the fact that the data itself has high variability.

12 Conclusion

The aim of this thesis was to develop an algorithm that could predict the dynamically changing trade network over time. Our empirical results showed that the algorithm was able to reproduce the network statistics and reconstruct the adjacency matrix even when partially observed. This algorithm however showed a bias towards the zero edged because the binary data for the classifier was not balanced and the adjacency matrix was sparse. Care should be taken to account for such an unbalance data. None the less the combination between the graph neural network encoder, the recurrent neural network and the bilinear decoder was successfully able to reconstruct the network. From the estimated network measures, we were able to tell that the international trade network is becoming more and more integrated as countries trade with each other, thereby creating relationships and strong dependencies. Furthermore, we showed that the embeddings of the countries could represent the physical proximity of those countries. This fact is interesting since the network structure is closely related to the distance between the countries.

A future research direction might be to include many factors as initial node features and not only the indicator function of the country. This would lead to a more complete model and better embeddings. Furthermore, one could implement more sophisticated techniques that require the recurrent neural network to be stochastic thereby improving the dynamics of the network.

13 REFERENCES

A.Rodrigueza, M., & JoshuaShinavierb. (20210). Exposing multi-relational networks to single-relational network analysis algorithms. *Journal of Informetrics, 4*(1), 29-41. doi:https://doi.org/10.1016/j.joi.2009.06.004

Aitchison, J., & Brown, J. A. (1957). The Lognormal Distribution, With Special Reference to Its Uses in Economics. *Journal of the Royal Statistical Society*, 228-230.

Bader, B. W., Harshman, R. A., & Kolda, T. G. (2007). Temporal Analysis of Semantic Graphs Using ASALSAN.

Bessadok, A., Mahjoub, M. A., & Rekik, I. (2021, 6 7). *Graph Neural Networks in Network Neuroscience.* Retrieved from https://arxiv.org: /pdf/2106.03535.pdf

Billio, M., Casarin, R., Kaufmann, S., & Iacopini, M. (n.d.). Bayesian Dynamic Tensor Regression.

Bonginia, P., & Bianchini, M. (2021, 05 27). *Molecular Generative Graph Neural Networks for Drug Discovery*. Retrieved from https://arxiv.org/: abs/2012.07397

Bronstein, M. M., Bruna, J., Cohen, T., & Veličković, P. (2021). *Geometric Deep Learning: Grids, Groups, Graphs, Geodesics, and Gauges*. Retrieved from https://arxiv.org: /abs/2104.13478

Bruna, J., Zaremba, W., Szlam, A., & LeCun, Y. (2014, 05 21). *Spectral Networks and Deep Locally Connected Networks on Graphs*. Retrieved from https://arxiv.org/: abs/1312.6203

Caimo, A., & Friel, N. (2011). Bayesian inference for exponential random graph models. *Social Networks, 33*(1), 41-55. doi:https://doi.org/10.1016/j.socnet.2010.09.004

Chen, E. Y., & Chen, R. (2017). Factor Models for High-Dimensional Dynamic Networks: with Application to International Trade Flow Time Series 1981-2015. *arxiv: Methodology*.

Cho, K., Merrienboer, B. v., Bahdanau, D., & Bengio, Y. (2014). *On the properties of neural machine translation: Encoder-decoder approaches*. Retrieved from https://arxiv.org: /abs/1409.1259

Chung, J., Gulcehre, C., Cho, K., & Bengio, Y. (2014). *Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling*. Retrieved from https://arxiv.org/: abs/1412.3555

Dan, L., Jiajing, W., Qi, Y., & Zibin, Z. (2020). T-EDGE: Temporal WEighted MultiDiGraph Embedding for Ethereum Transaction Network Analysis. *Frontiers in Physics*. doi:10.3389/fphy.2020.00204

David, R., Geoffrey, H., & Ronald, W. (1985). Learning internal representations by error propagation.

Defferrard, M., Bresson, X., & Vandergheynst, P. (2017, 02 05). *Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering*. Retrieved from https://arxiv.org/: abs/1606.09375

Dengliang, S. (2017, 05 24). *Clarifing a misunderstanding of back-propagation through time method*. Retrieved from https://dengliangshi.github.io: /2017/05/24/clarifing-a-misunderstanding-of-back-propagation-through-time-method.html

Dwivedi, V. P., & Bresson, X. (2021, Jan 24). *A Generalization of Transformer Networks to Graphs*. Retrieved from https://arxiv.org: /abs/2012.09699

Gal, Y. (2016). *Uncertainty in Deep Learning.*

García-Algarra, J., Mouronte-López, M. L., & Galeano, J. (2019). A stochastic generative model of the World Trade Network. *Scientific Reports*.

Gavili, A., & Zhang, X.-P. (2017). On the Shift Operator, Graph Frequency and Optimal Filtering in Graph Signal Processing. *IEEE Transactions on Signal Processing* , 6303 - 6318.

Goodfellow, I. J., Bengio, Y., & Courville, A. (2016). *Deep Learning.* Cambridge, MA, USA: MIT Press.

Grover, A., & Leskovec, J. (n.d.). *node2vec: Scalable Feature Learning for Networks*. Retrieved from https://arxiv.org: /abs/1607.00653

Guo, C., Pleiss, G., & Weinberger, Y. S. (2017 , August). On calibration of modern neural networks. *ICML'17: Proceedings of the 34th International Conference on Machine Learning, 70*, 1321–1330. doi:10.5555/3305381.3305518

Hagberg, A. A., Schult, D. A., & Swart, P. J. (2008). Exploring network structure, dynamics, and function using NetworkX. *Proceedings of the 7th Python in Science Conference (SciPy2008)*.

Hammond, D. K., Vandergheynst, P., & Gribonval, a. R. (2011.). Wavelets on graphs via spectral graph theory. *Applied and Computational Harmonic Analysis*, 129–150.

Hannes, S. (2021). *Self-Supervised learning for small Molecular Graphs*.

Hochreiter, S. (1998). The Vanishing Gradient Problem During Learning Recurrent Neural Networks. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 107-116.

Hoff, P. D., Raftery, A. E., & Handcock, M. S. (n.d.). Latent Space Approaches to Social Network Analysis.

https://stackoverflow.com/. (2021). Retrieved from https://i.stack.imgur.com/j2qa7.png: questions/65600387/when-to-use-a-neural-network-with-just-one-output-neuron-and-when-with-multiple

https://www.iso.org/. (2020). Retrieved from Codes for the representation of names of countries and their subdivisions: obp/ui/#iso:std:iso:3166:-1:ed-4:v1:en

Hu, R. (n.d.). *8.7. Backpropagation Through Time*. Retrieved from https://d2l.ai/: chapter_recurrent-neural-networks/bptt.html#full-computation

Ji, S., Pan, S., & Cambria, E. (2021, 3 1). *A Survey on Knowledge Graphs: Representation, Acquisition and Applications*. Retrieved from https://arxiv.org: /abs/2002.00388

Jin, D., Kim, S., Rossi, R. A., & Koutra, D. (2020, 9 21). *From Static to Dynamic Node Embeddings*. Retrieved from https://arxiv.org: /abs/2009.10017

Kim, B., Lee, K. H., Xue, L., & Niu, X. (2018). A review of dynamic network models with latent variables. *Statistics Survey, 12*, 105-135.

LeCun, Y., Haffner, P., Bottou, L., & Bengio, Y. (1999). Object Recognition with Gradient-Based Learning. In *Shape, Contour and Grouping in Computer Vision* (p. 319).

Maaten, L. v., & Hinton, G. (2008). Visualizing Data using t-SNE. *Journal of Machine Learning Research 9*, 2579-2605.

Maluck, J., & Donner, R. V. (n.d.). A Network of Networks Perspective on Global Trade. doi:10.1371/journal.pone.0133310

Methodology Guide for UN Comtrade User on UN Comtrade Upgrade 2019. (2019). Retrieved from https://comtrade.un.org/: data/MethodologyGuideforComtradePlus.pdf

Murat, M. (n.d.). *Backpropagation Through Time for Recurrent Neural Network*. Retrieved from https://mmuratarat.github.io: /2019-02-07/bptt-of-rnn

N., K. T., & Welling, M. (2017). Semi-Supervised Classification with Graph Convolutional Networks. ICLR. Retrieved from https://arxiv.org/abs/1609.02907

Pareja, A., & Domeniconi, G. (2019, 11 18). *EvolveGCN - Evolving Graph Convolutional Networks for Dynamic Graphs*. Retrieved from https://arxiv.org: /abs/1902.10191

Pedregosa, F. a. (2011). Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 2825-2830.

Petar, V., Guillem, C., Arantxa, C., Adriana, R., Pietro, L., & Yoshua, B. (2018, 02 18). *Graph Attention Networks*. Retrieved from https://arxiv.org/: abs/1710.10903

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, pages533–536.

Sandryhaila, A., & Moura, J. M. (2013). Discrete Signal Processing on Graphs. *IEEE Transactions on Signal Processing*, 1644-1656. doi:10.1109/TSP.2013.2238935

Sigmoid function. (n.d.). Retrieved from https://en.wikipedia.org/wiki/: Sigmoid_function#/media/File:Logistic-curve.svg

Singer, U., Guy, I., & Radinsky, K. (2019, 3 21). *Node Embedding over Temporal Graphs*. Retrieved from arxiv.org/: abs/1903.08889

Squartini, T., Fagiolo, G., & Garlaschelli, D. (2011, 11 2). *Randomizing world trade. II. A weighted network analysis*. Retrieved from https://arxiv.org: /abs/1103.1249

Srivastava, N., Hinton, G., & Krizhevsky, A. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 1929–1958.

Tiziano, S., Fagiolo, G., & Diego, G. (2011, 11 2). Randomizing world trade I A binary network analysis. Retrieved from https://arxiv.org: /abs/1103.1243

Unal, I. (2017). Defining an Optimal Cut-Point Value in ROC Analysis: An Alternative Approach. *Computational and Mathematical Methods in Medicine*. doi:https://doi.org/10.1155/2017/3762651

United Nations. UN Comtrade . (n.d.). Retrieved from https://comtrade.un.org: /data/

Vandal, T., Kodra, E., Dy, J., & Ganguly, S. (2018). Quantifying Uncertainty in Discrete-Continuous and Skewed data with Baeysian Deep Learning. *18: Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2377–2386. doi:https://doi.org/10.1145/3219819.3219996

Vaswani, A., Shazeer, N., & Parmar, N. (2017). Attention is all you need. *NIPS'17: Proceedings of the 31st International Conference on Neural Information Processing Systems*, 6000–6010.

Visualizing networks. (n.d.). Retrieved from http://www.mkivela.com/pymnet/visualizing.html.

Wang, X., & He, X. (2019, Jun 8). *KGAT: Knowledge Graph Attention Network for Recommendation*. Retrieved from https://arxiv.org: /abs/1905.07854

Wei, W., Zhang, Q., & Liu, L. (2021). Bitcoin Transaction Forecasting With Deep Network Representation Learning. *IEEE Transactions on Emerging Topics in Computing, 9*. doi:10.1109/TETC.2020.3010464

Weijiang, F., Naiyang, G., Yuan, L., AU, Z. X., & Zhigang, L. (2017). Audio visual speech recognition with multimodal recurrent neural networks. In 2. I. Networks (Ed.).

Werbos, P. J. (1990). Backpropagation Through Time: What It Does and How to Do It. *Proceedings of the IEEE, 78*, 1550 - 1560.

X., J., & J., H. (2011). K-Means Clustering. *Encyclopedia of Machine Learning*. doi:https://doi.org/10.1007/978-0-387-30164-8_425

Xu, B., Wang, N., Chen, T., & Li, M. (2015). Empirical Evaluation of Rectified Activations in Convolutional Network. *CoRR*. Retrieved from http://arxiv.org/abs/1505.00853

Yu, B., Yin, H., & Zhu, Z. (2018, 07 12). *Spatio-Temporal Graph Convolutional Networks: A Deep Learning Framework for Traffic Forecasting*. Retrieved from https://arxiv.org: /abs/1709.04875

Zafarani, R., Abbasi, M. A., & Liu, H. (2014). *Social Media Mining.* Cambridge University.

Zhao, L., Song, Y., Zhang, C., Liu, Y., Wang, P., Lin, T., . . . Li, H. (2020). T-GCN - A Temporal Graph Convolutional Network for Traffic Prediction. *IEEE Transactions on Intelligent Transportation Systems, 21*, 3848 - 3858. doi:10.1109/TITS.2019.2935152

Zheng, C., Fan, X., Wang, C., & Qi, J. (2020). GMAN: A Graph Multi-Attention Network for Traffic Prediction. *AAAI*, 1234--1241.

Zignago, G. G. (2010). *BACI: International Trade Database at the Product-Level. The 1994-2007 Version.* CEPII. Retrieved from http://www.cepii.fr/CEPII/en/publications/wp/abstract.asp?NoDoc=2726

14 APPENDIX

The following three classes are the class for the Bilinear decoder, the GRU-GAT cell and full model definition. The full implementation can be found at https://github.com/claCase/Master-Thesis.

```
import tensorflow as tf
from tensorflow.keras import layers as l
from tensorflow.keras import activations
from tensorflow.keras import initializers
import tensorflow.keras.backend as k

class BatchBilinearDecoderDense(l.Layer):
    """
    inputs:
        - X of shape batch x N x d
        - A of shape batch x N x N
    outputs: A of shape batch x N x N
    """

    def __init__(self, activation="relu", qr=True, regularizer="l2"):
        super(BatchBilinearDecoderDense, self).__init__()
        self.activation = activation
        self.regularizer = regularizer
        self.qr = qr

    def build(self, input_shape):
        x = input_shape
        self.R = self.add_weight(
            shape=(x[-1], x[-1]),
            initializer="glorot_normal",
            regularizer=self.regularizer,
            name="bilinear_matrix",
        )

    def call(self, inputs, *args, **kwargs):
        x = inputs
        if self.qr:
            Q, W = tf.linalg.qr(x, full_matrices=False)
            W_t = tf.einsum("...jk->...kj", W)
            Q_t = tf.einsum("...jk->...kj", Q)
            Z = tf.matmul(tf.matmul(W, self.R), W_t)
            A = tf.matmul(tf.matmul(Q, Z), Q_t)
            A = activations.get(self.activation)(A)
            return tf.matmul(Q, W), A
        else:
            x_t = tf.einsum("...jk->...kj", x)
            mat_left = tf.matmul(x, self.R)
            A = activations.get(self.activation)(tf.matmul(mat_left, x_t))
            return x, A
```

```
class GRUGAT(l.Layer):
    def __init__(self, hidden_size=10, attn_heads=10, dropout=0.2,
hidden_activation="relu", rc_gat=False,
                 temporal_smoothness=""):
        super(GRUGAT, self).__init__()
        self.gnn_u = GATConv(channels=hidden_size // 2,
```

```
attn_heads=attn_heads, concat_heads=True,
                             activation=hidden_activation,
dropout_rate=dropout, kernel_regularizer="l2")
        self.rc_gat = rc_gat
        if self.rc_gat:
            self.gnn_r = GATConv(channels=hidden_size // 2,
attn_heads=attn_heads, concat_heads=True,
                                 activation=hidden_activation,
dropout_rate=dropout, kernel_regularizer="l2")
            self.gnn_c = GATConv(channels=hidden_size // 2,
attn_heads=attn_heads, concat_heads=True,
                                 activation=hidden_activation,
dropout_rate=dropout, kernel_regularizer="l2")

        self.hidden_activation = hidden_activation
        self.hidden_size = (hidden_size // 2) * attn_heads
        self.drop = l.Dropout(dropout)
        self.state_size = self.hidden_size
        self.output_size = self.hidden_size
        self.temporal_smoothness = temporal_smoothness
        if self.temporal_smoothness:
            self.tmp_smooth = TemporalSmoothness(0.5,
self.temporal_smoothness)

    def get_initial_state(self, inputs=None, batch_size=None, dtype=None):
        x, a = inputs
        return tf.zeros(shape=(*x.shape[:-1], self.hidden_size))

    def build(self, input_shape):
        self.b_u = self.add_weight(shape=(self.hidden_size,),
initializer="glorot_normal", name="b_u")
        self.b_r = self.add_weight(shape=(self.hidden_size,),
initializer="glorot_normal", name="b_r")
        self.b_c = self.add_weight(shape=(self.hidden_size,),
initializer="glorot_normal", name="b_c")
        self.W_u = self.add_weight(shape=(self.hidden_size * 2,
self.hidden_size), initializer="glorot_normal",
                                   name="W_u")
        self.W_r = self.add_weight(shape=(self.hidden_size * 2,
self.hidden_size), initializer="glorot_normal",
                                   name="W_r")
        self.W_c = self.add_weight(shape=(self.hidden_size * 2,
self.hidden_size), initializer="glorot_normal",
                                   name="W_c")

    def call(self, inputs, state, training, *args, **kwargs):
        x, a = inputs
        # Encoding
        if state is None:
            h = self.get_initial_state(inputs)
        else:
            h = state

        conv_u = self.gnn_u(inputs, training=training)  # B x N x d
        if self.rc_gat:
            conv_r = self.gnn_r(inputs, training=training)  # B x N x d
            conv_c = self.gnn_c(inputs, training=training)  # B x N x d
        else:
            conv_r = conv_u
            conv_c = conv_u
        # Recurrence
```

```
        u = tf.nn.sigmoid(self.b_u + tf.concat([conv_u, h], -1) @ self.W_u)
        r = tf.nn.sigmoid(self.b_r + tf.concat([conv_r, h], -1) @ self.W_r)
        c = tf.nn.tanh(self.b_c + tf.concat([conv_c, r * h], -1) @
self.W_c)
        h_prime = u * h + (1 - u) * c
        h_prime = self.drop(h_prime, training=training)
        return h_prime
```

```
class GRUGATLognormal(m.Model):
    def __init__(self, hidden_size=4, attn_heads=4, dropout=0.2,
hidden_activation="relu", temporal_smoothness=""):
        super(GRUGATLognormal, self).__init__()
        # Encoders
        self.GatRnn_p = GRUGAT(hidden_size=hidden_size,
attn_heads=attn_heads, dropout=dropout,
                               hidden_activation=hidden_activation,
temporal_smoothness=temporal_smoothness)
        self.GatRnn_mu = GRUGAT(hidden_size=hidden_size,
attn_heads=attn_heads, dropout=dropout,
                                hidden_activation=hidden_activation,
temporal_smoothness=temporal_smoothness)
        self.GatRnn_sigma = GRUGAT(hidden_size=hidden_size,
attn_heads=attn_heads, dropout=dropout,
                                   hidden_activation=hidden_activation,
temporal_smoothness=temporal_smoothness)

        # Decoders
        self.decoder_mu = BatchBilinearDecoderDense(activation=None,
qr=False)
        self.decoder_sigma = BatchBilinearDecoderDense(activation=None,
qr=False)
        self.decoder_p = BatchBilinearDecoderDense(activation=None,
qr=False)

    def call(self, inputs, states, training=None, mask=None):
        # Encoding
        h_prime_p = self.GatRnn_p(inputs, states[0])
        h_prime_mu = self.GatRnn_mu(inputs, states[1])
        h_prime_sigma = self.GatRnn_sigma(inputs, states[2])

        # Decoding
        x_p, p = self.decoder_p(h_prime_p)
        p = tf.expand_dims(p, -1)
        x_mu, mu = self.decoder_mu(h_prime_mu)
        mu = tf.expand_dims(mu, -1)
        x_sigma, sigma = self.decoder_sigma(h_prime_sigma)
        sigma = tf.expand_dims(sigma, -1)
        logits = tf.concat([p, mu, sigma], -1)
        return logits, h_prime_p, h_prime_mu, h_prime_sigma
```