Graph Generation

Paul Dreczkowski
University of St Andrews
200031862

Table of Contents

Table of Figures

Acknowledgements

I wish to express my sincerest gratitude to my supervisor Ruth Hoffmann for all the help, support and guidance she has given me throughout the project.

I would also like to show how thankful I am to my best friend and partner, Ashley McGeechan.

Abstract

In this dissertation, the researcher has proposed the use of graph generation to generate graphs of different types and output them in a file format which can be used for visualisation and neural network training.

This paper will begin by discussing the different kinds and types of graphs that can exist and what kind of graphs this paper will be using along with which form of machine learning would be best for this task.

This paper will further elaborate on using a neural network model in order to try and learn from a dataset of randomly generated graphs for the purpose of trying to determine the properties and type of an inputted random graph.

Finally, the dissertation finishes by discussing the results of the newly created neural network and graph generator, and assesses their suitability as a method of graph generation and classification.

Declaration

I declare that the material submitted for assessment is my own work except where credit is explicitly given to others by citation or acknowledgement. This work was performed during the current academic year except where otherwise stated. The main text of this project report is 10,268 words long, including project specification and plan.

In submitting this project report to the University of St Andrews, I give permission for it to be made available for use in accordance with the regulations of the University Library.

I also give permission for the title and abstract to be published and for copies of the report to be made and supplied at cost to any bona fide library or research worker, and to be made available on the World Wide Web. I retain the copyright in this work.

1. Introduction

1.1. What is a Graph

Graphs are a natural data structure which can be used to show relational and structural information for systems such as social networks, transport routing, and molecule structures. These graphs are made up from dots and lines that can connect to each other. The dots are known as “Nodes” or “Vertices” and the lines are known as “Edges”.

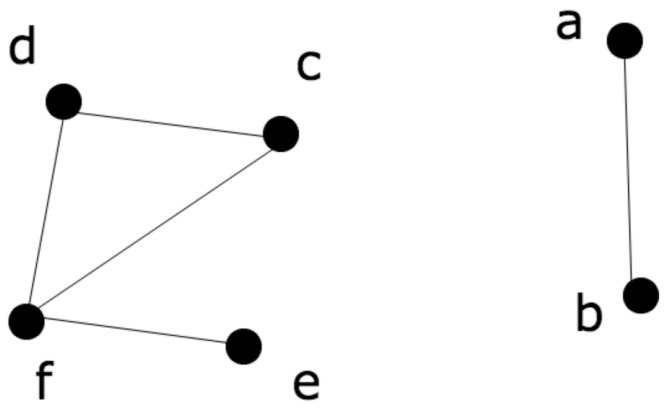

Figure 1.1 – Diagram showing an undirected graph with 6 nodes and 5 edges

By utilising these graph data structures, we are able to generate them with specific features for applications such as mapping molecular structures to help with designing drugs for the pharmaceutical industry (You et al., 2019), help push research in network science fields (Leskovec et al., 2010) with even more applications for technologies like social networking.

1.2. Purpose

The purpose behind this research is to explore the feasibility behind applying Deep Learning and Neural Network technologies to graph generation and graph theory problems and create a basis for others to work upon. This kind of work has had more specialised research done for subjects like network science, or drug manufacturing, but less research has been done on training models for more general use using randomly generated graphs. A successful model would be able to derive useful properties from any input graph without domain specific knowledge of what the graph represents and a graph generation tool would allow users to generate a dataset to the specifications they need at any size and scale. This would be useful due to the sheer

variety of applications graphs have and the number of industries they are used in. If general Neural Network models can learn to derive the properties of graphs from raw data it would give individual researchers and programmers access to large scale graph analysis capabilities with minimal experience in Graph Theory and data processing.

1.3. Aim

The original aims of the project were to create a tool that will generate random graphs with specified parameters, train a model that will use image classification to try and classify the outputted graphs and outputted graphs will be stored in an appropriate file type for easy parsing for visual data. As mentioned previously, due to how graphs are formed, generating a graph will produce nodes and edges, but this data can be visualised in a multitude of different ways. With this in mind, due to further research conducted on graph visualisation, building an image classifier around a data structure that can be visualised in multiple different ways makes building an image classifier extremely difficult and potentially fruitless.

With this new information in hand, we decided to alter the project such that the main aims with this project; to create a tool that can generate random graphs using a set of input parameters to generate a dataset, and then train and learn a machine learning model in order to try and classify a random graph for properties such as connectivity, degree sequences and planarity, whilst being able to store said graphs as an appropriate file type so that the information from it can be parsed easily, including making visualisations of the samples. The project accomplished:

The creation of a random graph generation tool.

With the use of NetworkX as a base for generating a random graph by itself, a tool was created around the use of its functions such that the user is able to input parameters in order to either generate a dataset of random directed graphs with however many nodes or just a random graph with however many nodes and edges a user may want. This is done with a mix of the NetworkX framework and code based around it.

The creation of Neural Network models.

This program trains Neural Network models based on a set of hyperparameters using Grid Search and evaluated using Cross Validation. A

final set of optimized models were created based on thorough experimentation and saved for live predictions.

Training Neural Networks on multiple graph properties.

Whenever a graph is randomly generated, it will store the node data, edge data, the Connectivity of the graph, the Degree Sequence of the graph, and the Planarity of the graph. In this project we were able to create models that predicted on all three of these properties with significant success.

Evaluating the accuracy of Neural Network predictions on graph data.

As impressive as it would be for the model to be 100% accurate and therefore be perfect, this is not necessary for the model to still provide useful and worthwhile predictions. The model's purpose is such that it is able to make as accurate as possible predictions based on previous data that it has learned such that it can be worthwhile for making predictions on our generated graphs. The models detailed in the reports were capable of achieving up to 89% accuracy on unseen test data for Connectivity, 95% accuracy for Planarity, and approached 100% for Degree Sequences.

2. Context Survey

2.1. Graph Neural Network

In 2018, a written review on the applications and methods for Graph Neural Networks was published (Zhou et al., 2018). A graph neural network is a specific type of deep learning method that operates on a graph domain and due to its accuracy and learning abilities, GNN's have been used more often in graph analysis recently. With how accurate some of these GNN models can be, there has been some level of skepticism in the reproducibility of them for all kinds of different datasets and real world applications. With this in mind, further research that focused on Semi-supervised node classification concludes that different dataset splits lead to a vastly different ranking of models where even simpler models could outperform more complex ones (Shchur, Mumme, Bojchevski and Günnemann, 2019). As noted, the progress and power of these GNN models has been increasing exponentially as more research is done around them for machine learning tasks in the graph domain with more flexible models and more optimised training algorithms. The researchers went on to state that the main issues with the current state of Graph Neural Networks is robustness, interpretability, pretraining and complex structure modelling. If general Neural Network models are capable of learning from graph data as is the subject of this project, they may be able to outperform the specialised GNN models in these key areas due to their lower complexity.

2.2. Learning deep neural networks for node classification

Whilst deep neural networks have been able to make much progress in fields like image classification and text prediction, node classification still proposes significant challenges (Li and Pi, 2019). Node classification is the problem of predicting the label of an unlabelled node. The research in this paper outlines a solution for this problem through training a framework for Deep Neural Networks using supervised learning to guide the model using prelabeled node data with high levels of performance on real-world datasets. The framework itself was able to overcome the issue of current suboptimal learning methods for node classification based on the network embedding and classifier whilst taking advantage of deep neural networks to obtain feature representation of the data, whilst encoding nonlinear structural and semantic information (Li and Pi, 2019). With this project being a combination of graph generation and node classification to make prediction, this paper offered useful information such as using a more supervised approach for deep neural networks instead of an unsupervised approach which lead to

suboptimal solutions using clustering and visualisation as documented in a previous paper (Cao, Lu and Xu, 2016).

2.3. Graph networks for molecular design

Research done in 2020 commences by creating a new molecular design platform and begins by comparing it with 6 different GNN models (Mercado et al., 2021). What's most interesting about the molecular design platform is that the tool itself does not contain any manually inputted rules regarding chemistry or molecule structure, but this information is learned from the inputted dataset. This shows that graph-based generative models are worth looking into for molecular graph generation as there are more advantages for working with the graph representation of a molecule than that of a string-based representation. The most important thing to recognise from this research is that every molecular subgraph is interpretable which is not the case for all molecular substrings. Overall, this research shows that, at least, that graph-based generative models are much more flexible whilst having more promising tools for addressing the current and future challenges in molecular design (Mercado et al., 2021). This research shows the benefits for using graph-based generative models for molecular design to therefore perform predictions based on graphs. In this project we will be using deep neural networks for graph classification to further research this area using random graph generation and more general Neural Network models.

3. Ethics

There are no ethical complications due to the nature of the project. Any and all data used in the datasets for the neural network models are generated through the use of the graph generation tool in the project.

4. Design

There are two main categories of graphs, Undirected and Directed. In Figure 1.1 we see an Undirected Graph. In these graphs the relationship between two connected nodes is not constrained by direction, they are mutually connected to each other. This is shown by the lack of arrows on the connected edges that would indicate this relationship. If one were to model the connections between users in a social network like Facebook it would take the form of an Undirected Graph, as all connections on the platform are mutual

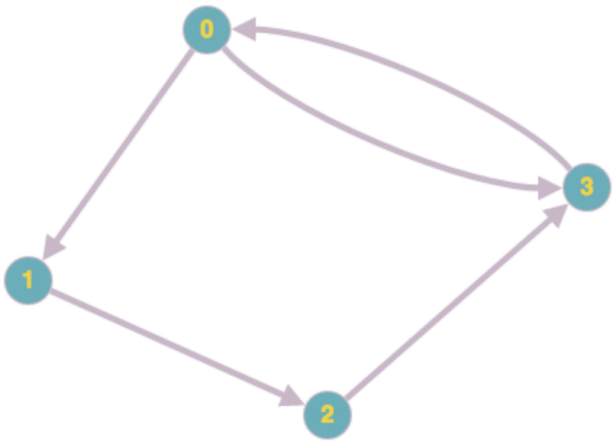

Figure 4.1 – Diagram showing a directed graph with 4 nodes and 5 edges

If one were to model another social network like Twitter however, where one user may choose to follow another without the other following them back, an Undirected Graph would not fully encode this relationship. This requires the use of the second type of graph, the Directed Graph. A Directed Graph encodes one-way relationships between nodes, indicated visually as pointed arrows on the edges. As shown in Figure 4.1, each edge for every node has an arrow with a head and a tail. Mutual relationships can still exist in Directed Graphs but they exist as two distinct edges between the two nodes with their own properties, as seen between node 0 and node 3 in the diagram. Therefore Undirected Graphs can be represented by Directed Graphs. There are many domains which necessitate Directed Graphs and so in order to achieve the goal of making the model applicable to a wide range of problems they will be the primary graph type explored in this project.

4.1. Connectivity

The connectivity of a graph refers to the ways in which the nodes in a graph can be connected to each other. Upon generation, each generated graph will have a certain level of connectivity depending on the nodes and edges that are given. For a rail transportation system, where the nodes are stations and the edges are the rails between them, connectivity can describe what journeys are possible in that system.

There are 4 levels of connectivity that will be tested; Disconnected graphs are said to be disconnected when there exist two nodes for which there is no path between them. Using the rail example, if an accident were to occur near the end of a rail line, the stations past that point would have no way of connecting to the other stations in the system, making the system disconnected. As shown in Figure 4.2 below, there is no edge connecting either node 0 or node 1 to node 2 or node 3, thus this is a disconnected graph.

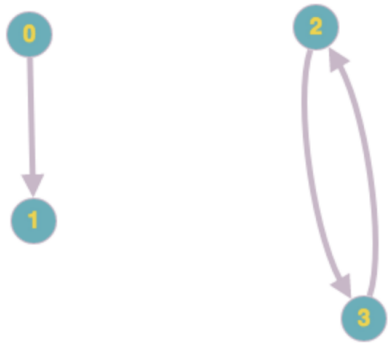

Figure 4.2 – Diagram showing a disconnected graph with 4 nodes and 2 edges

The next level of connectivity is a Weakly Connected graph. A Weakly Connected graph is said to have that connectivity if there exists a path to and from every node in the network, only if direction is ignored. In the real world people account for connectivity for rail systems. Trains are always able to travel both directions along a rail so they cannot be weakly connected. As shown in Figure 4.3 below, since there are no outgoing edges from node 1 there only exists a path from it to the other nodes if direction is ignored, making the graph weakly connected.

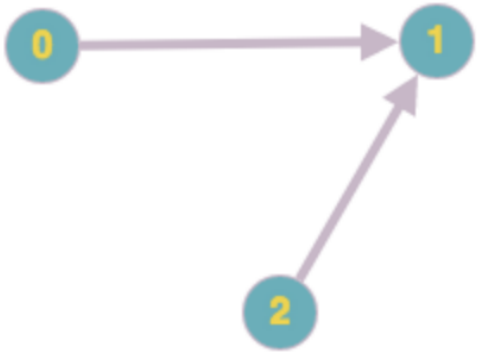

Figure 4.3 – Diagram showing a weakly connected graph with 3 nodes and 2 edges

A strongly connected graph is a graph where each node has a path that connects itself to every other node. In rail systems the goal is for the network to be strongly connected such that each station can be reached from every other station. The routes do not have to be efficient, a loop where you can only travel one way is strongly connected, as long as all nodes are reachable from all other nodes. As shown below, there is a path from each node to every other node which makes it strongly connected.

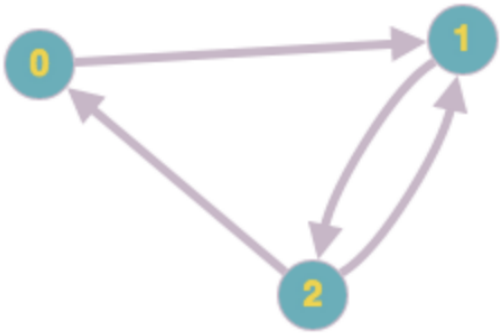

Figure 4.4 – Diagram showing a strongly connected graph with 3 nodes and 4 edges

The final level of connectivity is for a graph to be completely connected. A directed complete graph will have every node with an edge connected to every other node. Using the rail system example, a fully connected rail line would have every station with train lines to every other station. This is obviously infeasible and would likely be more dangerous as well due to the many rail lines crossing over each other. Overseas shipping systems however could be considered completely connected. As shown below in Figure 4.5, each node has an edge connecting it directly to every other node

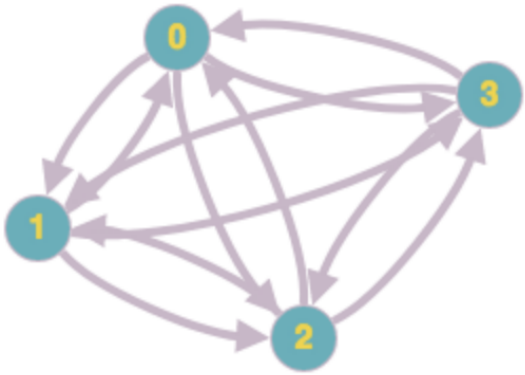

Figure 4.5 – Diagram showing a completed graph with 4 nodes and 12 edges

4.2. Planarity

Planarity is a way to describe a graph in graph theory that can be drawn in a plane such that the edges between nodes do not intersect or cross-over. By this definition, a graph where the edges cross-over and intersect one another are nonplanar graphs. As mentioned before, a notable use of graphs would be in the modelling for transportation systems, A node would be displaying a city and each edge being a transport line between them, planarity would allow us to make a graph of these cities and make sure the transportation lines do not intersect which would be more efficient.

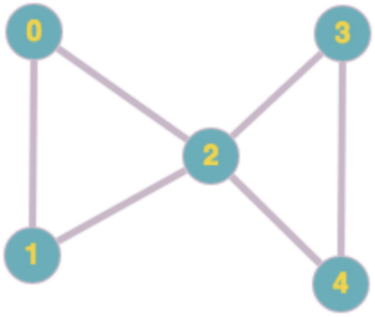

Figure 4.6 – Diagram showing a planar graph where none of the edges intersect

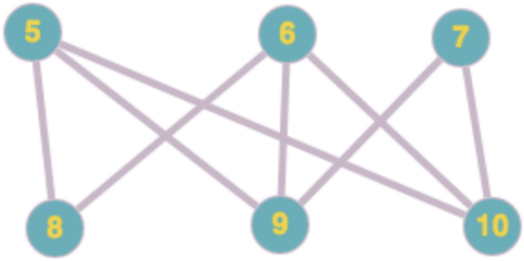

Figure 4.7 – Diagram showing a nonplanar graph where many of the edges intersect

4.3. Degree Sequence

The degree sequence of a directed graph shows a list of how many edges are connected to each node. This can further be separated into a list of outgoing and incoming edges. A large train station can have many train lines connected to it due to it being a station with lots of traffic coming from many places, but a smaller train station will only need a few train lines due to how few people will be using it. The degree sequence for the graph in Figure 4.6 would be (2, 2, 4, 2, 2) while the sequence for the graph in Figure 4.7 would be (3, 3, 2, 2, 3, 3). Some degree sequences are unique while others can be shared by many, very different graphs. Generating graphs solely from a degree sequence is a very useful and non-trivial task.

4.4. NetworkX

NetworkX is a network analysis library with support for creating and analysing directed graphs, which will be the focus of this project. As such it forms the basis for our random graph generation tool. More specifically the gnm_random_graph function which produces a random graph from a set of user-specified parameters (Daniel et al., 2008). Our program will handle the user input and convert that data to a form usable by NetworkX to generate a graph or dataset of graphs. We will then convert that object to a human-readable JSON format for saving. NetworkX also provides helper functions for determining the properties of graphs that we can use to label our dataset.

4.5. Basic Neural Network

Artificial neural networks (often called just neural networks) are an approach to artificial intelligence and machine learning that is based upon human cognition and the neurons and synapses in the brain. A neural network is made up of an input layer, an output layer, and a number of hidden layers. These layers contain artificial neurons which take the weighted sum of the outputs of the last layer and apply a bias term and an activation function to it to produce an output. An optimisation algorithm adjusts the values of the weights and biases during training so that the final output matches as closely to the true expected value.

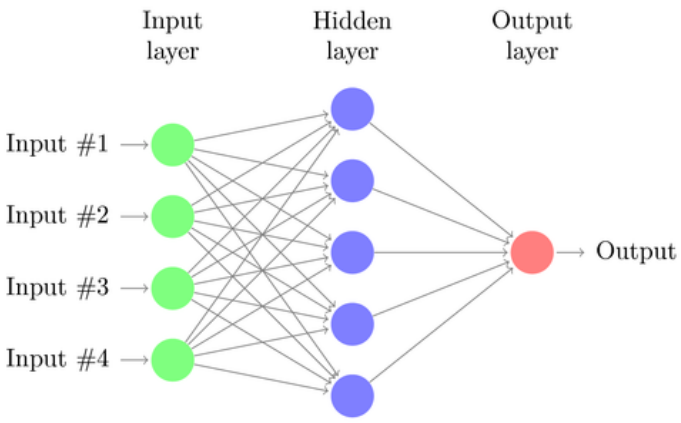

Figure 4.8 – Diagram showing a standard neural network (Fauske, 2006)

The primary model used in this project is the Feed Forward Model shown in Figure 4.8. Every neuron in each hidden layer connects to every neuron in the layer after it and they combine and alter the inputs into a form that the output layer then scales to match the scale of the true values. These kinds of networks use many hidden layers and so are referred to as Deep Neural Networks and the field of training these networks is called Deep Learning.

The challenge in designing a Feed Forward neural network involves determining an appropriate depth for the model, that being the number of hidden layers, and the width of each layer, that being the number of neurons. Increasing the width and the depth increases the network's capacity for remembering data and its ability to perform transformations on the input data. However high capacity networks are harder to train and can encounter overfitting, where the model uses that capacity to memorise the training data rather than learning a representation that allows it to generalise to new examples. Training involves finding a combination of weight and bias values for all the neurons and connections in the network. These are the parameters of the neural network and determine how the inputs are combined and transformed to produce the output. The goal of the neural network is to approximate a function that produces the correct output for each input by optimising these weights and biases.

We initialise the weights to random values and use an optimisation algorithm that utilises backward propagation to iteratively adjust the weights in ways that improve the network’s predictions on the training inputs. We keep track of the accuracy of the classifications being made by the network and view how it changes over time to determine how well the network is learning and relate that to the selected parameters. We can then make adjustments to the design of the network based on the results. To ensure the model is learning the correct features some of the data is withheld from the training set and

used after training is complete to evaluate how the network predicts on unseen data. Finally, when the model can no longer improve, or we are happy with the performance, the weights can be saved and the network can be deployed to make live predictions on any new data as needed.

Neural networks have a number of hyperparameters that affect the learning process. The learning rate controls how large an adjustment the algorithm can make to each individual parameter at each training step. The batch size controls how many samples the model is shown between each training step. Epochs control how many total iterations through the dataset are made. There are many other hyperparameters that all affect the network in different ways and it is the job of the designer to find values for these hyperparameters that produce the best training results.

One of the primary goals of the project is to evaluate whether neural networks can be trained to understand graphs. The more properties that the models can be shown to learn and how successful they are at learning them will act as the evidence to support our conclusion on that question. The three main factors the neural network will be learning from the dataset of graphs to predict are connectivity, planarity, and degree sequences.

5. Implementation

5.1. Graph Generation

The first objective of this project was to implement a tool for generating random graphs. This tool had to be able to create large quantities of graphs and determine their various properties so that they could be used as a training dataset for our predictive models. This is primarily accomplished through the NetworkX library in our Graph module. The Graph module has two modes that can be chosen through the command line interface. The first mode generates a single graph while the second generates a dataset.

The first mode is generally used for testing and visualisation purposes. The user enters a number of nodes and edges they want the graph to contain and can specify the names of the nodes or any edges they would like included. Two graphs are generated, one containing the specified nodes and edges, and one containing the remaining nodes with no edges. These are composed to form one graph and random edges are then generated from the set of non-edges until the graph matches the specifications. Using Matplotlib the graph is plotted using a circular layout and in the case of nodes that are connected bi-directionally, the corresponding edges are curved so as to not overlap.

The second mode is slightly modified to facilitate the easier creation of a full dataset. The user enters a desired number of graphs for the script to generate. Nodes cannot be named and edges cannot be specified. Nodes are named sequential integers to ensure consistency in the dataset and to reduce the amount of preprocessing needed to make it suitable for our Neural Network. For each graph a number between the minimum and the maximum number of nodes specified by the user is randomly generated. The max number is stored in a config file for use in preprocessing the data for the Neural Network. The max number of edges is derived from the number of nodes multiplied by the number of nodes minus one. Each graph is given a number of edges corresponding to a random number in the range between one and the max. This will create an imbalanced dataset. For example a graph with five nodes has twenty possible edges, meaning only 5% of the graphs will be fully connected. We will address this problem in the preprocessing section.

Graph data is stored in a dictionary containing its nodes, edges, connectivity type, degree sequence, and planarity. Nodes are stored as a list of integers.

Edges are stored as a list of tuples, the first value being the origin node and the second being the destination node. Connectivity type is stored as an integer with 0 representing a disconnected graph, 1 representing a weakly connected graph, 2 representing a strongly connected graph, and 3 representing a completely connected graph. Degree sequences are stored as a list of tuples, the first value in the tuple being the name of the node and the second being its degree. Planarity is stored as a 0 or 1, representing a non-planar graph and a planar graph respectively. Each graph dictionary is stored separately as a JSON file named sequentially and stored in the 'graphs' folder in the active directory.

5.2. Deep Learning Model

The second objective of the project was to implement a Deep Learning model capable of taking in a graph as input and determining its properties. This is achieved through the Tensorflow package and the Keras frontend that allow for the creation of simple and efficient deep learning models using common layers, activation functions, and optimizers. Like the graph generation module, this module has multiple modes chosen through the command line interface. The three modes correspond to the three properties of the graph that the model can be trained to predict, connectivity, degree sequence, and planarity.

The script begins by reading in all of the data contained in the graphs folder created by the graph generation script and prepares it for preprocessing. The number of nodes is read from the config file and this number is used to determine the maximum number of edges. Tensors must conform to a consistent shape. The first entry in a tensor must have the same number of values as the second entry and so on. When we consider the contents of a graph this raises a number of issues. Nodes are represented as a one-dimensional array of values while edges are two-dimensional pairs of values. There are two possible solutions for transforming this data to be compliant with the rules of tensors.

The first is to make the node data implicit. That is to say if all of the graphs have the same number of nodes then there is no need to include them in the data, only the edges would be necessary. This is the simpler of the two solutions but it has two major drawbacks. The first is that a model that can only predict on graphs with a set number of nodes is not very flexible. While there are certainly problems in which such a model could be useful most problems involve graphs with varying numbers of nodes. Secondly it makes some aspects of the model trivial. If there are a set number of nodes then any graph missing one of those nodes in its edges is disconnected, while

any graph that has the max number of edges is fully connected. In a model that supports varying numbers of nodes it will have to derive what nodes are missing and what the max number of edges are for that graph based on the node data. Since this solution is easy to implement and reason on it was used initially for testing before the second solution was implemented.

This solution is to use padding. Padding is using a dummy value to fill in for missing values in the data. By including the node data at the start of the tensor and padding graphs with nodes fewer than the max number with a value of -1, a value that would never naturally occur in the graphs, the tensor shape of each graph can be kept equal. Likewise the edge data can be padded so that it matches the max number of edges, based on the max number of nodes. The way this is implemented particularly is by first creating a tensor of the max shape filled with values of -1. Then each graph is iterated through and the data replaces the values in its corresponding place. Edges are naturally represented as pairs, the first value being the origin node of the edge and the second value being the destination. If the node data was to be packaged similarly, [0, 1], [2, 3] and so on, they would be indistinguishable from edges. In theory structure like this does not matter for simple Dense networks, but structure is integral to more advanced models like Convolutional Neural Networks and Recurrent Neural Networks. To maintain this structure edges retain their two-value structure and node values are succeeded by a -1. For example a graph with two nodes and two edges between these nodes would be structured as [0, -1], [1, -1], [0, 1], [1,0] with values of [-1, -1] padding adding after the node data and edge data to bring them to the correct shape.

After this comes the issue of class imbalance. As discussed earlier, graphs are far more likely to be weakly or strongly connected than they are to be disconnected or fully connected and this imbalance rises with the number of nodes. Similarly graphs are less likely to be planar the more edges they have, which correlates with the number of nodes. Class imbalance is very dangerous when dealing with Neural Networks. Neural Networks optimise based on adjusting weights in the direction that best improves the loss value. In a dataset with a severe class imbalance the initial path of fastest improvement would be found by predicting high-frequency classes and ignoring low-frequency classes. Even if the model learns to differentiate between these high-frequency classes it may find itself trapped in a local minima. Even though performance can be theoretically improved the learning rate may be too small to allow any weight changes to result in a lower loss value. The network can no longer learn to predict those low-frequency classes. There are many ways to counteract the problem of unbalanced classes and since we are able to generate as much data as we want we have

a considerable number of options. The primary way we will address class imbalance is through undersampling. Undersampling is when samples of majority classes are removed until every class has an equal number of samples, matching that of the least frequent class. This way we can generate a large sample size and evaluate the model based on the regular and undersampled dataset and clearly observe the effects of class imbalance. This is applied when attempting to train on either Connectivity or Planarity.

When dealing with Degree Sequences, which are represented by multiple values each continuous between zero and the number of other nodes, it is infeasible to generate enough data to model this as a classification problem except for graphs with a very small quantity of nodes. Since the values being predicted correspond to real numbers and not categories it makes the most sense to model the problem as a Regression problem. While Regression models can suffer due to lack of data, their ability to predict values they haven't seen directly means they don't experience class imbalance in the same way. So while lower order nodes will have more data available and be biased towards smaller numbers, given enough data the model should still be able to approximate the relationship between the edges and the degree sequence.

As a last measure of preprocessing we shuffle the dataset and labels using the same seed to ensure that the link between a sample and it's label is not broken while sufficiently randomizing the dataset. This is where a training and test split would be done if we were training a single model but since we are at this stage using a Grid Search with Cross-Validation this is unnecessary.

Next we define our deep learning model. The building of this model is stored as a function that will allow us to adjust the hyperparameters programmatically. In this case the create_model function takes three arguments, the learning rate, the number of neurons in each layer, and the number of layers. The model is sequential, meaning data passes through each layer in sequence. The first layer simply reshapes the data to ensure it has the structure discussed earlier. Then for each layer specified in the arguments we add a Dense layer and a Dropout layer.

A Dense layer is the most common layer used in Neural Networks. A node in a Dense layer is connected to all the nodes in the layer before it and the layer after it. For each node in the layer before it, it stores a weight in a matrix called the kernel. The node computes the dot product of the input values and the kernel, adds a bias value, and applies an activation function. The output is then sent to all the nodes in the next layer. The optimizer adjusts

the values of these weights and biases to change the output values and in turn get an improved final answer. The activation function is ReLU which outputs zero when the value is zero or below and outputs the value itself when it is positive. The ReLU function is widely used as it is fast to compute and its sparse activation, where 50% of nodes initialize to an output of zero, make training very efficient. There are alternatives to ReLU which will be explored in the evaluation stage.

The Dropout layer selects a specified percentage of nodes to be disabled during each step in training. The nodes are selected randomly and change each step. While disabled they do not output any value and their weights are not adjusted. Dropout is set to 20% for every Dense layer except the output layer. Dropout aims to prevent the model from becoming too dependent on any one node and counteract overfitting, where the model memorises the input data at the cost of its ability to predict on data it hasn't seen. This is particularly a concern for high capacity models. After that is a Flatten layer which simply reshapes the data again, this time to make it one dimensional.

Finally there is a Dense output layer with a number of nodes equal to the number of expected outputs for the data it is predicting. This means four nodes for the four Connectivity categories, one node for the binary Planarity, and a number of nodes equivalent to the max nodes in the graph data for the Degree Sequence. This output layer has no activation function and so is purely the weighted sum of the inputs plus the bias term.

Each of these prediction types also uses a different loss function which is selected here. These are Sparse Categorical Cross Entropy, Binary Cross Entropy, and Mean Squared Error respectively. Cross Entropy is a measure of how many extra bits are needed to correctly represent a set of inputs as one of the output categories. For example if a model is 100% certain that a set of inputs belong to a category then no extra bits are needed but if the model is only 50% sure for one category and 50% sure for another then an extra bit would be needed. Even more would be needed if the model is categorising the inputs incorrectly. Sparse Categorical Cross Entropy is used for multiclass categorisations like Connectivity while Binary Cross Entropy is used for binary categorisations like Planarity. Mean Squared Error measures the square of the difference between the predicted values and the true values. Squaring the error means that the effect incorrect values have on the loss value scales exponentially with how incorrect it is. An error of two contributes four times as much to the loss value as an error of one.

The last thing we must specify here is the optimiser, the algorithm that will calculate the gradient of the weights and adjust them. For this we use the

Adam optimizer. The Adam optimizer has documented success on many deep learning problems and is less dependent on hyperparameter tuning than other optimizers like RMSProp or SGD. There are several alternative versions of Adam that can offer better results under certain circumstances which again will be explored in the evaluation section.

We then compile the model and wrap it in either a KerasClassifier or KerasRegressor wrapper to make it compatible with Scikit Learn's Grid Search module. We specify the verbosity as two so we can see the progress of the models as they train and set shuffling to true, which randomises the order of samples in each batch. Grid Searching allows us to define a set of values for each hyperparameter and have models trained on all the combinations of those hyperparameters and evaluated on which has the best validation loss over multiple cross-validation splits. We can then iterate on those results, centering the grid search on the best parameters until improvements become too sparse. To search over multiple parameters we first define those parameters in a dictionary. Batch Size controls the amount of data the model is shown at each training step and Epochs controls the number of iterations through the entire dataset. Some models benefit from smaller batch sizes while others benefit from higher. Models should get better with a greater number of epochs although improvements will begin to plateau and some can see an increase in validation loss if the model begins overfitting, in which case it makes sense to reduce the number of epochs. Since these hyperparameters are hugely influential in how long a model takes to train it makes sense during initial exploration, where the number of models is high and many will be dead-ends, to prioritise speed of training when setting their values. This means a low number of epochs, enough for them to get past the initial burst of large improvements per epoch, and high batch sizes. Our initial values for Learning Rate will be centered at 0.001, the recommended default value for the Adam optimizer, going up to 0.01 and down to 0.0001. The number of neurons will have values of 8, 16, and 32, and the number of layers will be 1, 2, or 3.

We create the Grid Search object using the model and the parameter grid and set Cross Validation to two. Grid Search uses k-fold cross validation which means the dataset will be split into two folds. The model will be trained on half the data and validated on the other half, then repeated with the opposite split. At higher values like five for example, the model would be trained on 4/5ths of the data and validated on the remaining 1/5th, repeated five times. Like earlier the decision to split the data only two ways is a time consideration during the early stages of training. We then fit the models using the dataset and save the results of the Grid Search to a JSON file.

Included with this training script is a second script that trains the models using the best found hyperparameters and saves the model itself rather than simply saving the parameters and results of the training. These models train using a standard 80/20 training to test split and a further 80/20 training to validation split. This script uses much of the same preprocessing and model training code from the primary script but also includes graph plots of the Accuracy and Loss over the training epochs.

5.3. Classifier

Finally there is a Classifier script which loads the trained models and a specified graph or folder of graphs, and predicts the property requested by the user. The prediction and the true value are displayed for comparison. For Degree Sequences the predicted sequence includes predicted padded values.

6. Evaluation

Firstly we must elaborate about the evaluation methodology we will use that has been mentioned briefly in previous sections. This project is largely experimental. There is no set expectation for what the final model should look like nor what performance or limitations are acceptable. The goal of this project is primarily to determine if general neural network models are capable of learning to reason about graph data. It is our secondary objective to expand and optimise these models as much as possible as a means to determine the feasibility of using neural networks to tackle real world graph theory problems. For that purpose we have used an iterative Grid Search based method to train many different models of varying speeds and size for each graph property individually. Once a pattern has been established for how different combinations of parameters affect the quality of the model we selected the best performers and manually adjusted them to create a set of ideal models.

6.1. Connectivity

Connectivity is a Classification problem. There are 4 classes based on the ability to travel from one node to the others through their connecting edges. The initial test batch for each property was run with the same set of parameters.
Those are:

- Batch Size: 4096
- Epochs: 100
- Layers : 1, 2, 3
- Learning Rate: 0.01, 0.001, 0.0001
- Neurons: 8, 16, 32

The results of that initial test are tabulated in Figure 6.1 with the Accuracy of each model being the averaged accuracy of the two models trained with those parameters across their respective validation sets.

Batch Size	Epochs	Layers	Learning Rate	Neurons	Accuracy
4096	100	1	0.01	8	0.891
4096	100	1	0.01	16	0.889
4096	100	1	0.01	32	0.893
4096	100	1	0.001	8	0.882
4096	100	1	0.001	16	0.885
4096	100	1	0.001	32	0.89
4096	100	1	0.0001	8	0.713
4096	100	1	0.0001	16	0.779
4096	100	1	0.0001	32	0.811
4096	100	2	0.01	8	0.883
4096	100	2	0.01	16	0.892
4096	100	2	0.01	32	0.886
4096	100	2	0.001	8	0.878
4096	100	2	0.001	16	0.885
4096	100	2	0.001	32	0.889
4096	100	2	0.0001	8	0.663
4096	100	2	0.0001	16	0.781
4096	100	2	0.0001	32	0.782
4096	100	3	0.01	8	0.883
4096	100	3	0.01	16	0.888
4096	100	3	0.01	32	0.888
4096	100	3	0.001	8	0.88
4096	100	3	0.001	16	0.885
4096	100	3	0.001	32	0.891
4096	100	3	0.0001	8	0.669
4096	100	3	0.0001	16	0.753
4096	100	3	0.0001	32	0.793

Figure 6.1 – Table of results for the initial Connectivity test models

The dataset fed to these models were undersampled from the full dataset to have a perfect class balance. This results in a dataset of roughly 10,000 samples out of the original 100,000 being used. Due to the Cross Validation this is further reduced to around 5,000 during the training stage.

In Figure 6.2 we see the model sorted by Accuracy, revealing a number of very clear trends.

Batch Size	Epochs	Layers	Learning Rate	Neurons	Accuracy
4096	100	1	0.01	32	0.893
4096	100	2	0.01	16	0.892
4096	100	1	0.01	8	0.891
4096	100	3	0.001	32	0.891
4096	100	1	0.001	32	0.89
4096	100	1	0.01	16	0.889
4096	100	2	0.001	32	0.889
4096	100	3	0.01	16	0.888
4096	100	3	0.01	32	0.888
4096	100	2	0.01	32	0.886
4096	100	1	0.001	16	0.885
4096	100	2	0.001	16	0.885
4096	100	3	0.001	16	0.885
4096	100	2	0.01	8	0.883
4096	100	3	0.01	8	0.883
4096	100	1	0.001	8	0.882
4096	100	3	0.001	8	0.88
4096	100	2	0.001	8	0.878
4096	100	1	0.0001	32	0.811
4096	100	3	0.0001	32	0.793
4096	100	2	0.0001	32	0.782
4096	100	2	0.0001	16	0.781

4096	100	1	0.0001	16	0.779
4096	100	3	0.0001	16	0.753
4096	100	1	0.0001	8	0.713
4096	100	3	0.0001	8	0.669
4096	100	2	0.0001	8	0.663

Figure 6.2 – Table of results for the initial Connectivity test models sorted by Accuracy

Learning rate appears to be a key factor in determining the model's learning capability with all of the 0.0001 learning rate models displaying the lowest accuracy levels. The other parameters in this data are not as helpful however. Models with larger quantities of neurons consistently outperformed those with only eight, except in the case of the one layer eight neuron model which ranked third. There is also no clear relationship between the number of layers and the performance of the model. One layer models have the highest average placing but the difference between the first place model with one layer and the fourth place model with three layers is only 0.002. None of the models are able to reach an Accuracy of 0.9 although many are very close. If we consider that there are four categories an unintelligent model, one that predicted at random or only ever predicted one category, would have a baseline performance of 0.25. All of the models presented here clearly beat this metric. The neural network is capable of learning the property of connectivity and deriving it from the data to some degree. However the difficulty of predicting each category, or at least the perceived difficulty of predicting each category, is not equal. A graph in which a node is missing from the list of edges is clearly disconnected. A graph which has the max number of edges for its number of nodes is clearly fully connected. While the former does not describe every disconnected graph, and the latter is made more difficult by the existence of padding, the majority of the complex behaviour we are looking for the model to learn is in discerning between weakly and strongly connected graphs. A model that could perfectly discern disconnected and fully connected graphs but guessed for weakly and strongly connected graphs would have an accuracy of 0.75. It is very promising then that all models except for the 0.0001 learning rate, eight neuron models have an accuracy higher than 0.75. We iterated on this test by removing 0.0001 from the list of learning rates and eight from the list of neurons, and then decreasing the batch size and increasing the number of epochs. The results of that test are recorded in Figure 6.3.

Batch Size	Epochs	Layers	Learning Rate	Neurons	Accuracy
2048	200	1	0.01	16	0.889
2048	200	1	0.01	32	0.888
2048	200	1	0.001	16	0.891
2048	200	1	0.001	32	0.893
2048	200	2	0.01	16	0.891
2048	200	2	0.01	32	0.891
2048	200	2	0.001	16	0.891
2048	200	2	0.001	32	0.893
2048	200	3	0.01	16	0.89
2048	200	3	0.01	32	0.892
2048	200	3	0.001	16	0.892
2048	200	3	0.001	32	0.889

Figure 6.3 – Table of results for the secondary Connectivity test models

The results tighten up, with only a 0.005 difference between the best and worst model, but the lower batch size and increased training time has not had any effect on the overall performance of the models, they still fail to surpass an accuracy of 0.893. We can make a more extreme adjustment and see if this is a pattern that holds. As such we reduced the batch size significantly, to 128, increased the number of epochs to 500, and replaced the 16 neuron parameter with a 48 neuron parameter.

Batch Size	Epochs	Layers	Learning Rate	Neurons	Accuracy
128	500	1	0.01	32	0.884
128	500	1	0.01	48	0.884
128	500	1	0.001	32	0.893
128	500	1	0.001	48	0.89
128	500	2	0.01	32	0.886
128	500	2	0.01	48	0.882
128	500	2	0.001	32	0.891

128	500	2	0.001	48	0.886
128	500	3	0.01	32	0.887
128	500	3	0.01	48	0.892
128	500	3	0.001	32	0.888
128	500	3	0.001	48	0.888

Figure 6.4 – Table of results for the tertiary Connectivity test models

figure 6.4 seems to confirm that something is limiting the Accuracy of the model that is not related to these hyperparameters. As a final test of this theory we trained a low capacity model with the new batch rate and epoch parameters seen in Figure 6.5.

Batch Size	Epochs	Layers	Learning Rate	Neurons	Accuracy
128	500	2	0.001	8	0.892

Figure 6.5 – Table of results for low capacity Connectivity test model

Hyperparameter tuning appears not to be the issue with this early convergence. To study the model more closely we trained the model again using a standard training split with 20% of the samples being split into an unseen testing set and 20% of the remaining training samples being used for validation. This model achieved a training accuracy of 0.893, a validation accuracy of 0.885, and a testing accuracy of 0.890. Examining the results of the models predictions on the test data we can observe a sample that the model has categorised incorrectly to determine if there are any issues with the data. In this example sample 8 of the test data is weakly connected but is erroneously predicted as strongly connected. Looking at the data in Figure 6.6 we can see there exists no edge originating from node 3, meaning the graph is in fact weakly connected. After repeating this for a number of samples we determined there are no issues with the data which would be damaging the model's ability to learn.

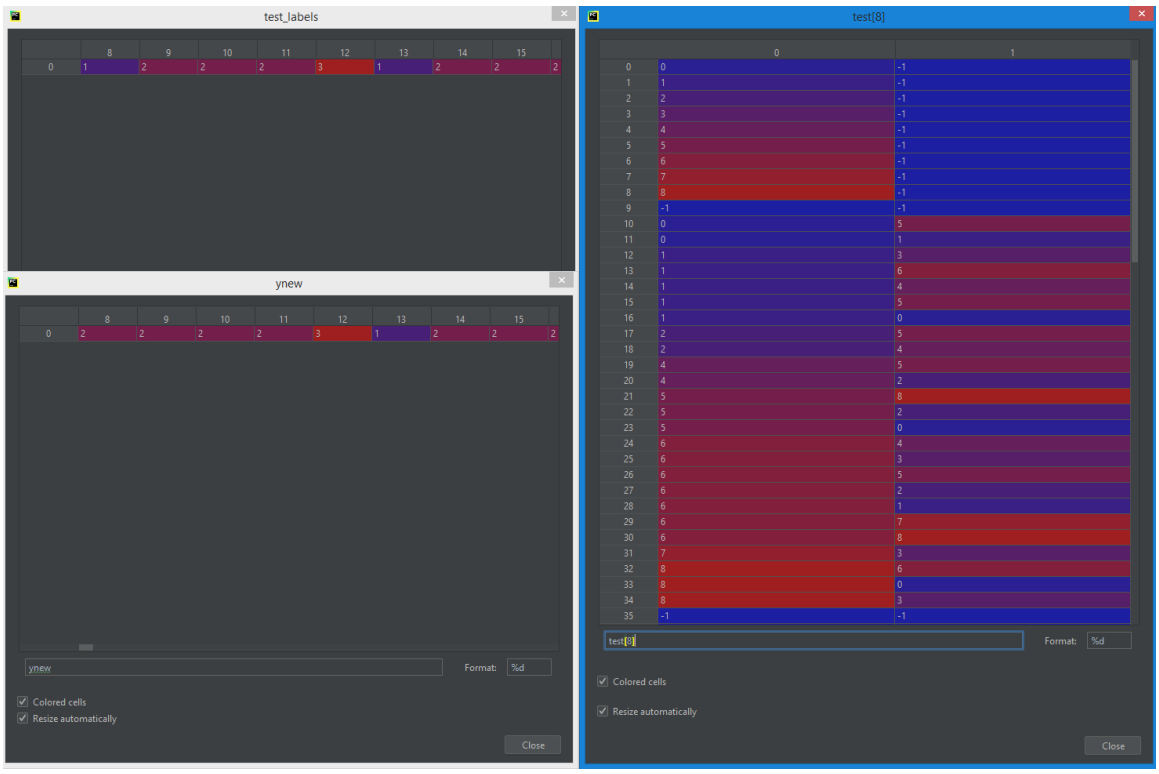

Figure 6.6 – Sample data from the Test set. The top left window displays the true values. The bottom left window displays the predicted values. The right window displays the contents of the sample being viewed.

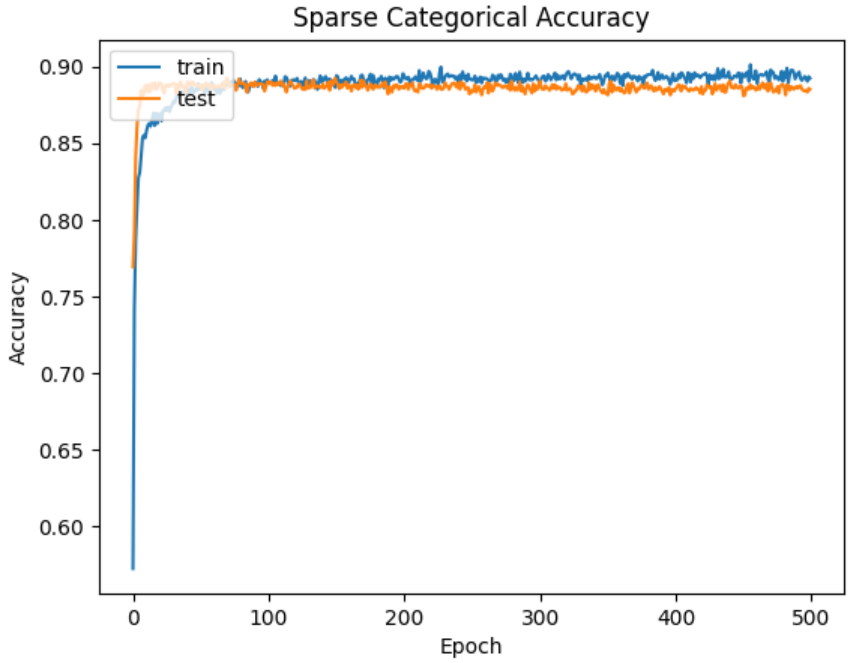

Figure 6.7 – Accuracy over time graph for the low capacity Connectivity model

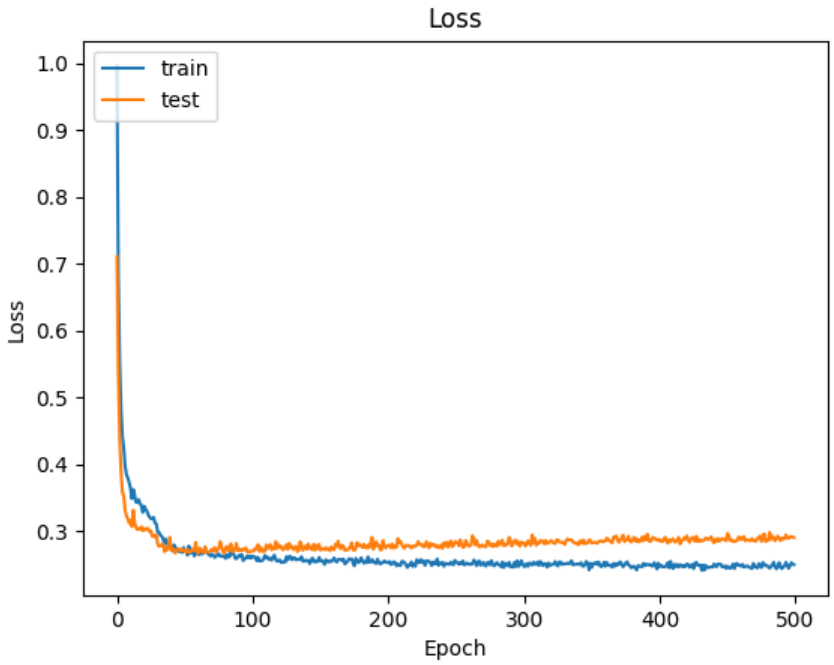

Figure 6.8 – Loss over time graph for the low capacity Connectivity model

In Figure 6.7 and Figure 6.8 we see plots of the Accuracy and Loss over time. The steep initial curve and the jagged plateau might imply that the Learning Rate is in fact too high. The model also shows evidence of overfitting despite the low capacity and presence of Dropout layers. One possibility is that the Adam optimizer's built in learning rate decay is not aggressive enough and as such the model is failing to converge. To test this theory we trained the model again with a learning rate of 0.00001 and 3000 epochs of training time. This model achieved a training accuracy of 0.880, a validation accuracy of 0.894, and a test accuracy of 0.891. This gives a slightly higher validation accuracy and test accuracy but lower training accuracy. Importantly it does not address the issue of early convergence. To test whether this was due to the Adam optimizer the model was trained using the SGD optimizer with the same hyperparameters as those in figure X, with a decay rate of 0.001, and a momentum of 0.9, resulting in a training accuracy of 0.864, a validation accuracy of 0.887 and a test accuracy of 0.882.

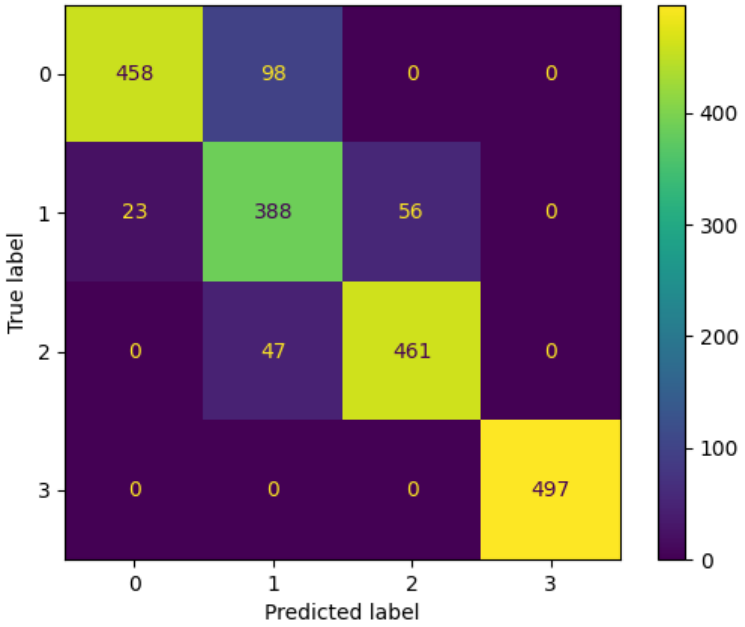

Figure 6.9 – Confusion Matrix of the low capacity Connectivity model

The reason for the premature convergence may simply be an issue with the data. While we have checked the samples for errors, neural networks are very sensitive to the quantity and quality of the data provided. Therefore it is possible that there are not enough samples, especially of the type that the model has difficulty with, for the model to learn to differentiate them fully. Plotting the Confusion Matrix of the model's predictions on the test data, shown in Figure 6.9, confirms that the model has difficulty differentiating disconnected graphs from weakly connected graphs, and weakly connected graphs from strongly connected graphs, as expected. Disabling the undersampling step and letting the model train on a much larger dataset gives the following Confusion Matrix.

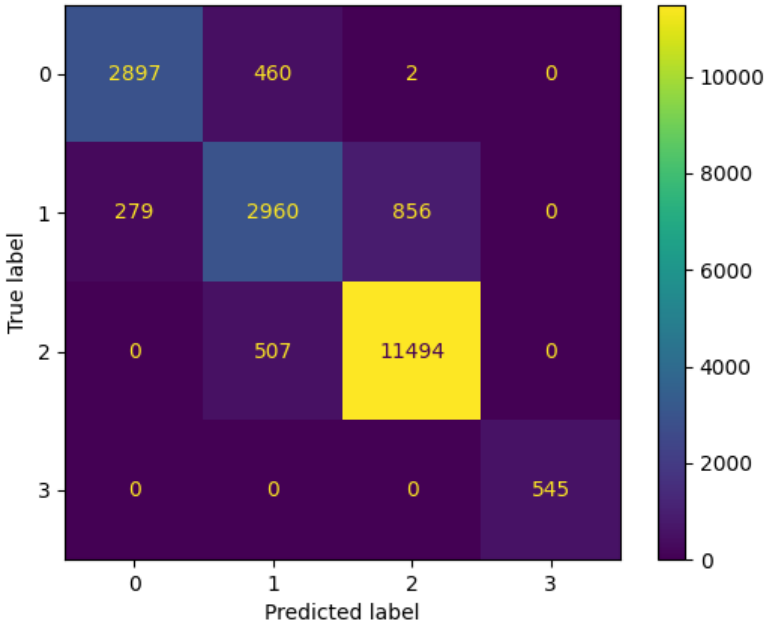

Figure 6.10 – Confusion Matrix of the low capacity Connectivity model trained with Undersampling disabled

This shows minor improvements over the Undersampled model. For example the percentage of misclassified disconnected graphs reduces from 17.6% to 13.7%. However it also displays worse behaviour in predicting weakly connected graphs, going from a 16.9% false negative rate to 27.7%. We can further augment the data by removing any duplicates from the graph data as seen in Figure 6.11. This most affects completely connected graphs since there is only a single unique completely connected graph for each number of nodes. Since graphs produced by NetworkX differ slightly in the ordering of edges we see 4 completely connected graphs in the test data and, as expected, the network is not able to correctly classify such an infrequent category. Removing these graphs slightly improves the accuracy for strongly connected graphs but has negative effects on disconnected and weakly connected graphs.

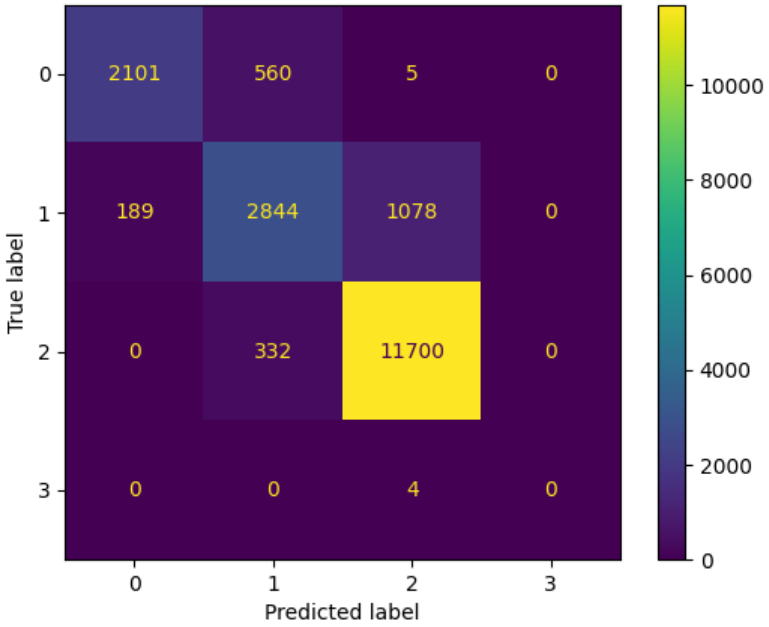

Figure 6.11 – Confusion Matrix of the low capacity Connectivity model trained with only unique graphs

By specifically undersampling so that the dataset comprises only 10,000 weakly and strongly connected graphs, the program produces the Confusion Matrix in Figure 6.12.

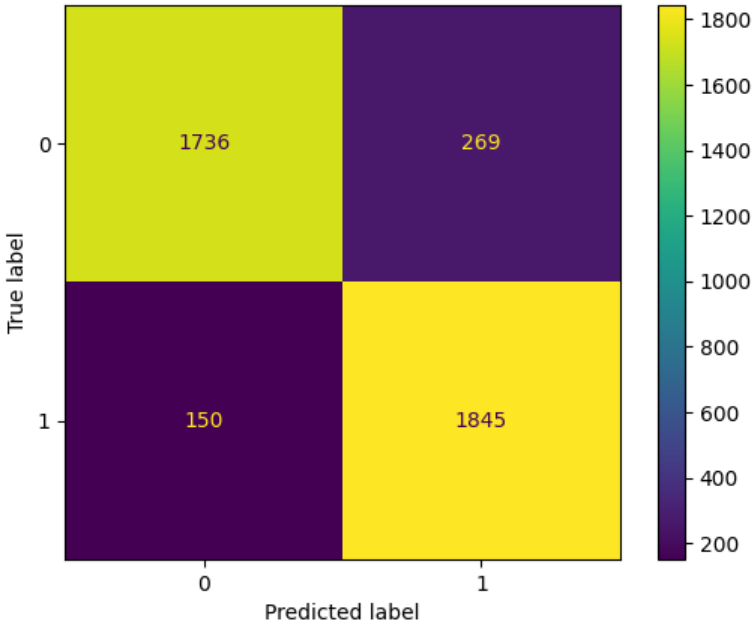

Figure 6.12 – Confusion Matrix of the low capacity Connectivity model trained with only Weakly and Strongly Connected Graph data

Here the false negative rate for weakly connected graphs is only 13.4% and 7.5% for strongly connected. This is solid evidence that the models are

capable of learning to identify the differences between graphs in a more intelligent way and that the best approach may not be to model the problem as a multiclass classification problem but multiple binary classification problems.

In all, the neural networks tested here display some complex understanding of graph connectivity that allow them to approach a 90% Accuracy. They also show some evidence of being able to go past that limit given some combination of better data collection and more complex or specialised network architecture.

6.2 Degree Sequences

The initial test batch for each property was run with the same set of parameters.
Those are:

- Batch Size: 4096
- Epochs: 100
- Layers : 1, 2, 3
- Learning Rate: 0.01, 0.001, 0.0001
- Neurons: 8, 16, 32

Batch Size	Epochs	Layers	Learning Rate	Neurons	Mean Squared Error
4096	100	1	0.01	8	1.3
4096	100	1	0.01	16	1.26
4096	100	1	0.01	32	1.217
4096	100	1	0.001	8	1.543
4096	100	1	0.001	16	1.46
4096	100	1	0.001	32	1.384
4096	100	1	0.0001	8	3.903
4096	100	1	0.0001	16	3.191
4096	100	1	0.0001	32	2.683
4096	100	2	0.01	8	1.265
4096	100	2	0.01	16	1.179
4096	100	2	0.01	32	1.119
4096	100	2	0.001	8	1.48
4096	100	2	0.001	16	1.354
4096	100	2	0.001	32	1.226
4096	100	2	0.0001	8	4.358
4096	100	2	0.0001	16	2.891
4096	100	2	0.0001	32	2.301
4096	100	3	0.01	8	1.156

4096	100	3	0.01	16	1.13
4096	100	3	0.01	32	1.039
4096	100	3	0.001	8	1.606
4096	100	3	0.001	16	1.334
4096	100	3	0.001	32	1.238
4096	100	3	0.0001	8	4.215
4096	100	3	0.0001	16	2.85
4096	100	3	0.0001	32	2.289

Figure 6.13 – Table of results for the initial Degree Sequence test models

Due to being a Regression problem and not a Classification problem the metric used for evaluating these models is Mean Squared Error, where lower values are better, rather than Accuracy. The models were fed the entire 100,000 sample dataset, split into two 50,000 sample training sets for Cross Validation, with the Mean Squared Error being the averaged Mean Squared Error of each model's performance on the other model's training set. Sorting the models by Mean Squared Error gives the table in Figure 6.14.

Batch Size	Epochs	Layers	Learning Rate	Neurons	Mean Squared Error
4096	100	3	0.01	32	1.039
4096	100	2	0.01	32	1.119
4096	100	3	0.01	16	1.13
4096	100	3	0.01	8	1.156
4096	100	2	0.01	16	1.179
4096	100	1	0.01	32	1.217
4096	100	2	0.001	32	1.226
4096	100	3	0.001	32	1.238
4096	100	1	0.01	16	1.26
4096	100	2	0.01	8	1.265
4096	100	1	0.01	8	1.3
4096	100	3	0.001	16	1.334

4096	100	2	0.001	16	1.354
4096	100	1	0.001	32	1.384
4096	100	1	0.001	16	1.46
4096	100	2	0.001	8	1.48
4096	100	1	0.001	8	1.543
4096	100	3	0.001	8	1.606
4096	100	3	0.0001	32	2.289
4096	100	2	0.0001	32	2.301
4096	100	1	0.0001	32	2.683
4096	100	3	0.0001	16	2.85
4096	100	2	0.0001	16	2.891
4096	100	1	0.0001	16	3.191
4096	100	1	0.0001	8	3.903
4096	100	3	0.0001	8	4.215
4096	100	2	0.0001	8	4.358

Figure 6.14 – Table of results for the initial Degree Sequence test models sorted by Mean Squared Error

From this data we see the models clearly prefer a higher learning rate with only two models not following this pattern. The two that are out of order are the two and three layer, 32 neuron, 0.001 learning rate models. A relationship between performance and layer depth or neuron count is much clearer in these results than those of Connectivity which serve to explain these two models. Of the top six models only one has 8 neurons, in which case it has 3 layers, and only one has 1 layer, in which case it has 32 neurons. There is a clear bias towards models with a greater capacity. For the second iteration of model testing we halved the batch size, doubled the number of epochs, removed the 1 layer and 0.0001 learning rate parameters, and replaced the 8 neuron parameter with a 64 neuron parameter.

Batch Size	Epochs	Layers	Learning Rate	Neurons	Mean Squared Error
2048	200	3	0.01	64	0.398
2048	200	3	0.01	32	0.514

2048	200	2	0.01	64	0.641
2048	200	3	0.01	16	0.667
2048	200	3	0.001	64	0.767
2048	200	2	0.01	32	0.873
2048	200	2	0.001	64	0.899
2048	200	2	0.01	16	0.966
2048	200	3	0.001	32	0.969
2048	200	2	0.001	32	1.022
2048	200	3	0.001	16	1.138
2048	200	2	0.001	16	1.165

Figure 6.15 –Table of results for the secondary Degree Sequence test models sorted by Mean Squared Error

The effect of the lower batch size and increased training time has had a significant effect with even the worst model of this iteration improving by 0.189 over the last iteration. Layer size and learning rate again appear to be the main factors in model performance with the top 6 models all having either 3 layers or a 0.01 learning rate, or both as is the case for the top 2 models. Training the best model again, this time with the training split described in the Connectivity sections, allows us to observe the loss graph before we make further decisions.

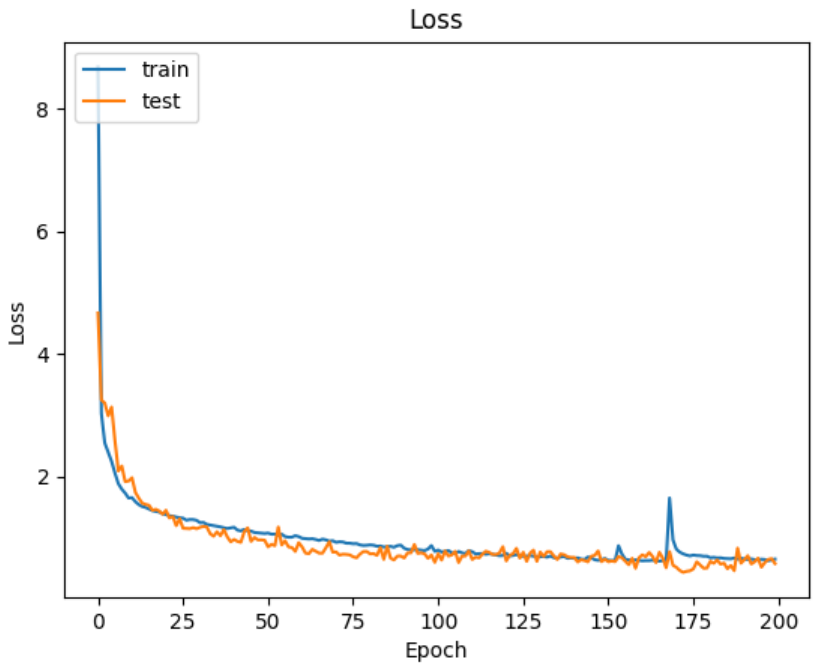

Figure 6.16 – Loss over time graph for the 2048 batch size - 3 layer - 64 neuron - 0.01 learning rate Degree Sequence model

There are two main points we can learn from Figure 6.16. Firstly, the steep curve could indicate that our learning rate is too high. Secondly there are small spikes in the training loss that may be the result of having too high a batch size. The random shuffling of the batches can cause an imbalanced distribution of data that result in poor choices of weight updates. Therefore we adjusted the batch size to 512 and reduced the learning rate to 0.001.

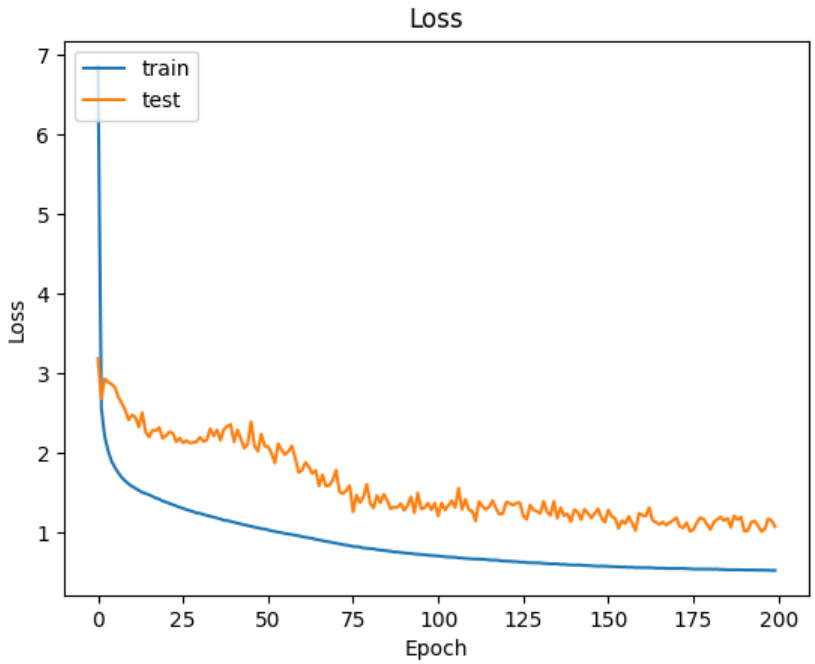

Figure 6.17 – Loss over time graph for the 512 batch size - 3 layer - 64 neuron - 0.001 learning rate Degree Sequence model

From Figure 6.17 we see the training loss has significantly smoothed and is now well below 1.0 Mean Squared Error. The validation error however remains very similar to how it looked in the previous model. This indicates a problem with the regularisation. The model's primary form of regularisation is the Dropout layers. Dropout layers activate during training time but not at testing time. The validation loss could be a result of the Dropout rate being too low, resulting in poor regularisation and generalisation, or it could be that the model is already generalising well and that the Dropout rate is too high. As such we adjusted these layers in both directions, training one model with a 50% Dropout rate and another with a 0% Dropout rate.

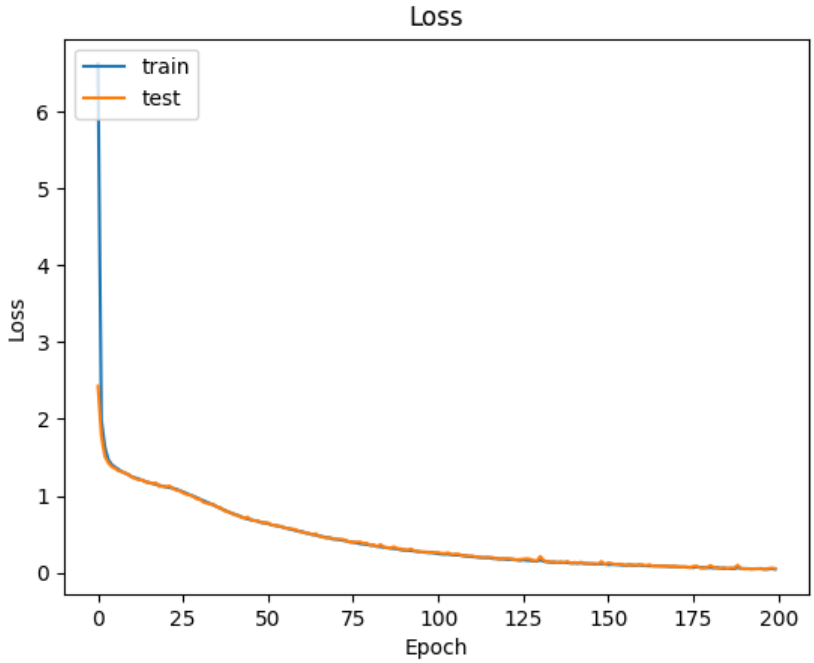

Figure 6.18 - Loss over time graph for the 512 batch size - 3 layer - 64 neuron - 0.001 learning rate - 0% Dropout rate Degree Sequence model

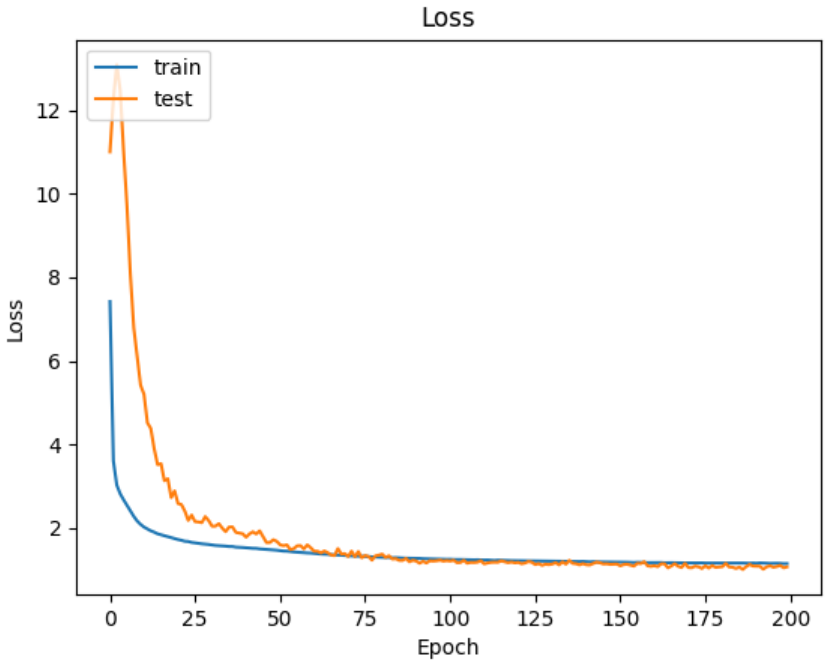

Figure 6.19 – Loss over time graph for the 512 batch size - 3 layer - 64 neuron - 0.001 learning rate - 50% Dropout rate Degree Sequence model

Both of these options resulted in better generalisation. As we see in Figure 6.18 and Figure 6.19 the gap between training loss is very small in both cases. However the model with 0% dropout, on the left, had a much lower Mean Squared Error than the 50% dropout model, indicating that the

reduction in capacity from the Dropout layers has had a negative effect on the model's ability to learn.

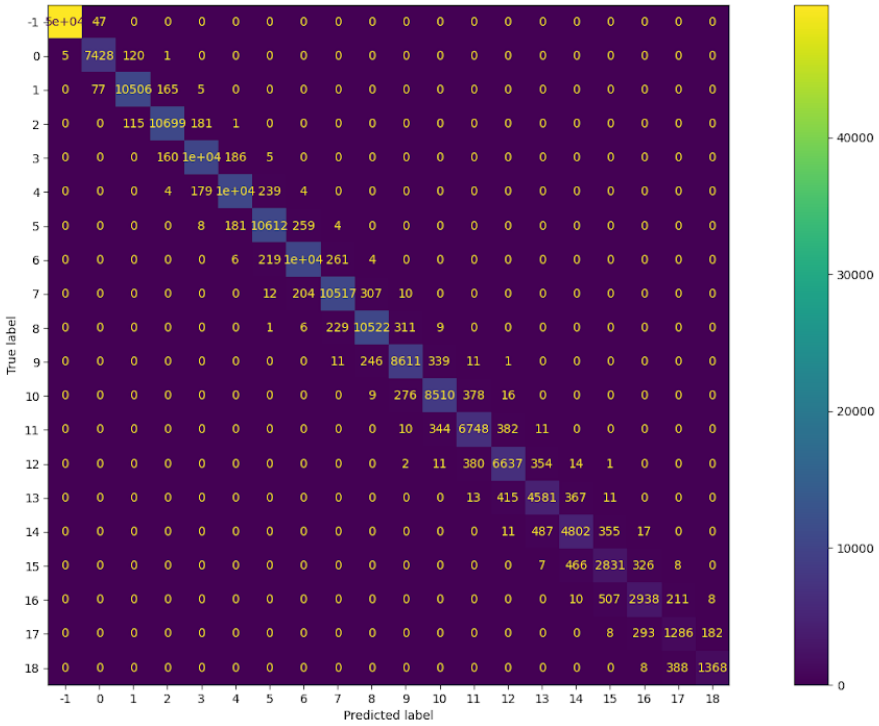

Figure 6.20 – Confusion Matrix for the 512 batch size - 3 layer - 64 neuron - 0.001 learning rate - 0% Dropout rate Degree Sequence model

From Figure 6.20 we can see the maximum error for any value in the test set is only three, for most values it is two and for some it is only one. The predicted values are rounded to the nearest whole number and any values below -1 or greater than 18 are brought to those respective values. The model understandably has a lower error rate for smaller values in the degree sequence as they are more common, and is biased towards overestimating until it reaches values of 12 and above, where it becomes biased to underestimation. With no visible indications of problems in the loss graph we further reduced the batch size to 128 and retrained the model alongside a 48 neuron and 80 neuron model, as capacity seems to be the most significant factor.

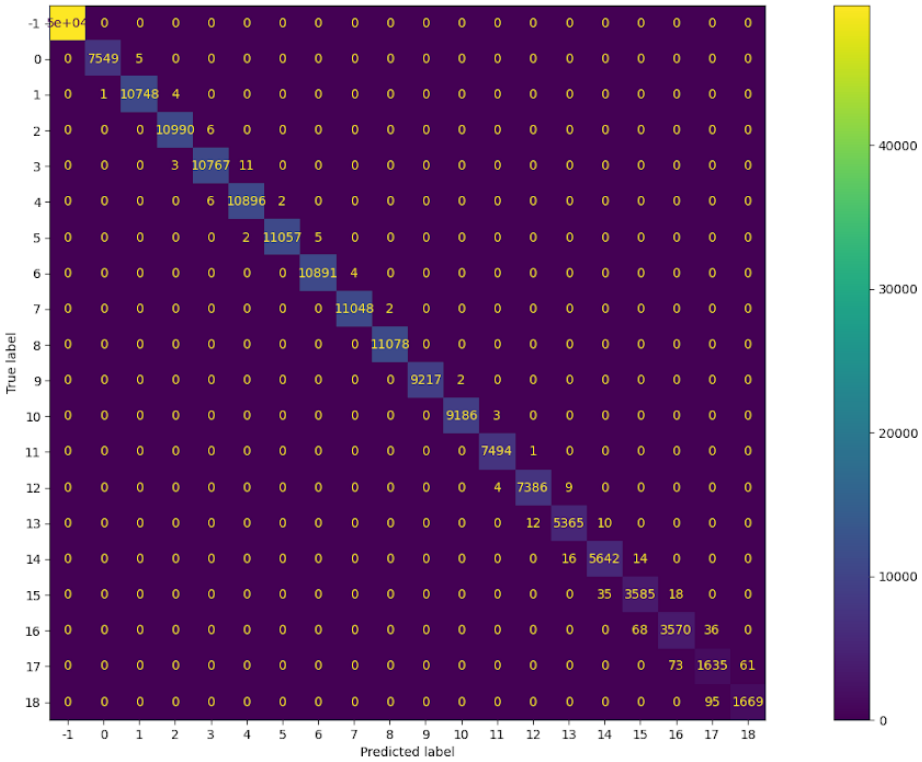

Figure 6.21 – Loss over time graph for the 128 batch size - 3 layer - 64 neuron - 0.001 learning rate - 0% Dropout rate Degree Sequence model

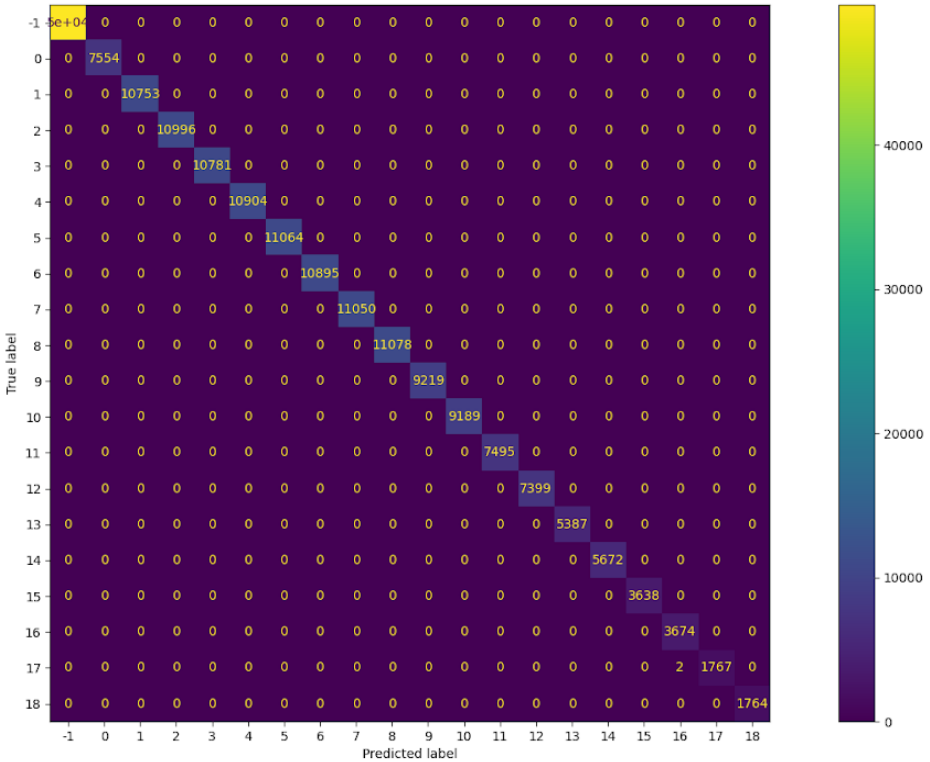

Figure 6.22 – Loss over time graph for the 512 batch size - 3 layer - 80 neuron - 0.001 learning rate - 0% Dropout rate Degree Sequence model

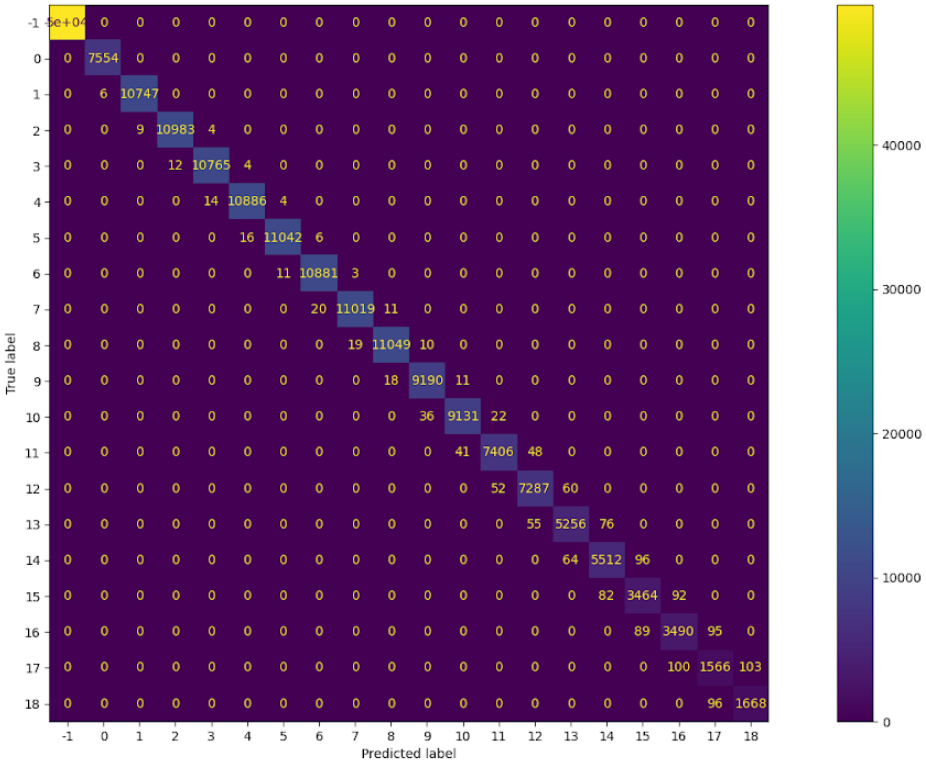

Figure 6.23 – Loss over time graph for the 512 batch size - 3 layer - 48 neuron - 0.001 learning rate - 0% Dropout rate Degree Sequence model

All of the models manage to reduce the maximum error for a value down to one in all cases but the 80 neuron model in particular manages to make only 2 wrong classifications across the entire testing set of 20,000 samples and 200,000 values.

6.3. Planarity

The initial test batch for each property was run with the same set of parameters.
Those are:

- Batch Size: 4096
- Epochs: 100
- Layers : 1, 2, 3
- Learning Rate: 0.01, 0.001, 0.0001
- Neurons: 8, 16, 32

Batch Size	Epochs	Layers	Learning Rate	Neurons	Accuracy
4096	100	1	0.01	8	0.947
4096	100	1	0.01	16	0.947
4096	100	1	0.01	32	0.948
4096	100	1	0.001	8	0.947
4096	100	1	0.001	16	0.947
4096	100	1	0.001	32	0.947
4096	100	1	0.0001	8	0.947
4096	100	1	0.0001	16	0.945
4096	100	1	0.0001	32	0.946
4096	100	2	0.01	8	0.948
4096	100	2	0.01	16	0.948
4096	100	2	0.01	32	0.949
4096	100	2	0.001	8	0.947
4096	100	2	0.001	16	0.947
4096	100	2	0.001	32	0.948
4096	100	2	0.0001	8	0.941
4096	100	2	0.0001	16	0.946
4096	100	2	0.0001	32	0.947
4096	100	3	0.01	8	0.947
4096	100	3	0.01	16	0.949

4096	100	3	0.01	32	0.948
4096	100	3	0.001	8	0.947
4096	100	3	0.001	16	0.947
4096	100	3	0.001	32	0.947
4096	100	3	0.0001	8	0.934
4096	100	3	0.0001	16	0.948
4096	100	3	0.0001	32	0.946

Figure 6.24 – Table of results for the initial Planarity test models

The dataset for these models, like that used for the Connectivity test, was undersampled, resulting in roughly 80,000 of the 100,000 samples being used, split into two 40,000 sample training sets. It does not seem necessary to order these results by Accuracy. Every model was easily able to reach an Accuracy of 0.93 but none were able to reach 0.95. As this situation is so similar to the one encountered in the Connectivity experiment we chose to approach the problem in a similar way, starting by training a simple model and examining it more closely. The parameters of a 128 batch size, 50 epochs of training time, 2 layers of 8 neurons, and a learning rate of 0.0001, were chosen. Only 50 epochs were needed for the model to converge as shown in the following plots.

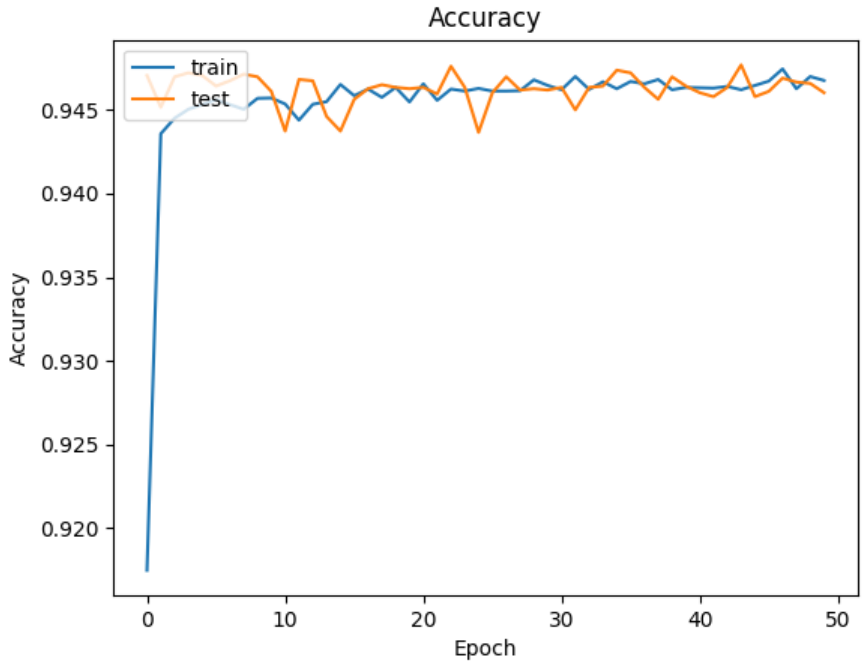

Figure 6.25 – Accuracy over time graph for the 128 batch size - 2 layer - 8 neuron - 0.0001 learning rate Planarity model

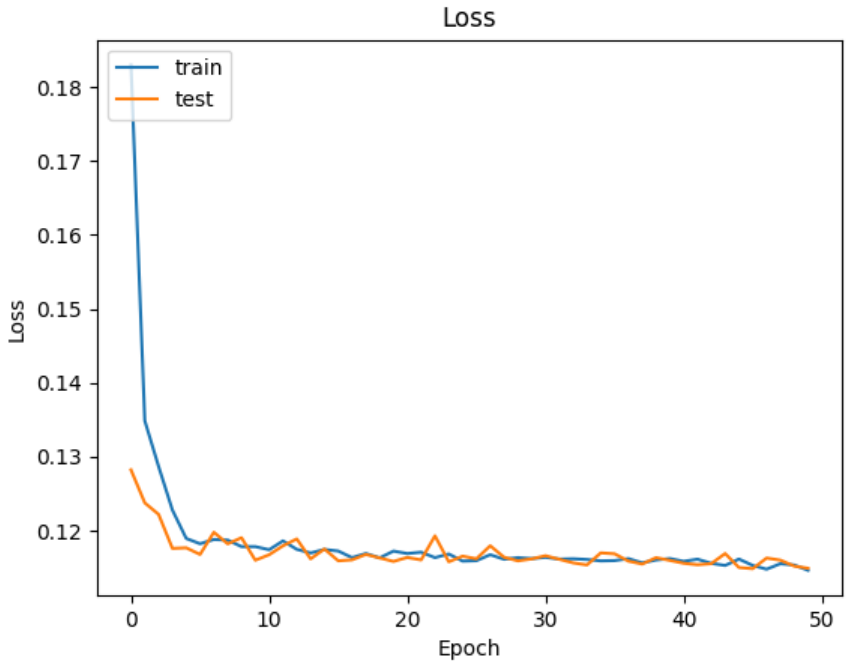

Figure 6.26 – Loss over time graph for the 128 batch size - 2 layer - 8 neuron - 0.0001 learning rate Planarity model

As with the Connectivity models, many variations on the network architecture, hyperparameters, and data processing were attempted but none were able to significantly push the model past this roadblock. Neural Networks take the path of fastest descent when optimizing the loss and this can cause them to converge to a sharp local minima where it cannot escape.

Using a 3 layer 64 neuron model with a learning rate of 0.001 and a batch size of 4096 we performed a Grid Search on the Regularisation parameters, these being the Dropout rate in the Dropout layers, the L1 Regularisation of the Dense layers, and the L2 Regularisation of the Dense layers.

Dropout	L1	L2	Accuracy
0	0.00001	0.00001	0.949
0	0.00001	0.0001	0.948
0	0.00001	0.001	0.947
0	0.0001	0.00001	0.948
0	0.0001	0.0001	0.948
0	0.0001	0.001	0.948

0	0.001	0.00001	0.947
0	0.001	0.0001	0.947
0	0.001	0.001	0.947
0.2	0.00001	0.00001	0.948
0.2	0.00001	0.0001	0.948
0.2	0.00001	0.001	0.947
0.2	0.0001	0.00001	0.948
0.2	0.0001	0.0001	0.948
0.2	0.0001	0.001	0.948
0.2	0.001	0.00001	0.947
0.2	0.001	0.0001	0.947
0.2	0.001	0.001	0.947
0.5	0.00001	0.00001	0.946
0.5	0.00001	0.0001	0.946
0.5	0.00001	0.001	0.948
0.5	0.0001	0.00001	0.947
0.5	0.0001	0.0001	0.947
0.5	0.0001	0.001	0.948
0.5	0.001	0.00001	0.946
0.5	0.001	0.0001	0.946
0.5	0.001	0.001	0.947

Figure 6.27 – Table of results for the Regularisation evaluation Planarity models

Figure 6.27 clearly shows that despite varying levels of Regularisation all of the tested models converged at similar points.

Again using a 3 layer 64 neuron model with a learning rate of 0.001 and a batch size of 4096 we performed a Grid Search on the Activation functions of the Dense layers, the Kernel Initializer of the Dense layers, and the Optimization Algorithm of the model.

Activation	Initializer	Optimizer	Accuracy
relu	glorot_normal	Adam	0.948
relu	glorot_normal	SGD	0.942
relu	glorot_normal	RMSProp	0.947
relu	glorot_normal	Adadelta	0.531
relu	glorot_normal	Adamax	0.948
relu	glorot_normal	Adagrad	0.946
relu	glorot_normal	Nadam	0.948
relu	glorot_normal	Ftrl	0.5
relu	glorot_uniform	Adam	0.948
relu	glorot_uniform	SGD	0.942
relu	glorot_uniform	RMSProp	0.948
relu	glorot_uniform	Adadelta	0.584
relu	glorot_uniform	Adamax	0.948
relu	glorot_uniform	Adagrad	0.945
relu	glorot_uniform	Nadam	0.947
relu	glorot_uniform	Ftrl	0.5
relu	he_normal	Adam	0.947
relu	he_normal	SGD	0.947
relu	he_normal	RMSProp	0.947
relu	he_normal	Adadelta	0.898
relu	he_normal	Adamax	0.948
relu	he_normal	Adagrad	0.948
relu	he_normal	Nadam	0.947
relu	he_normal	Ftrl	0.5
relu	he_uniform	Adam	0.947
relu	he_uniform	SGD	0.946
relu	he_uniform	RMSProp	0.947
relu	he_uniform	Adadelta	0.746

relu	he_uniform	Adamax	0.948
relu	he_uniform	Adagrad	0.948
relu	he_uniform	Nadam	0.946
relu	he_uniform	Ftrl	0.704
elu	glorot_normal	Adam	0.948
elu	glorot_normal	SGD	0.944
elu	glorot_normal	RMSProp	0.948
elu	glorot_normal	Adadelta	0.893
elu	glorot_normal	Adamax	0.947
elu	glorot_normal	Adagrad	0.947
elu	glorot_normal	Nadam	0.948
elu	glorot_normal	Ftrl	0.5
elu	glorot_uniform	Adam	0.949
elu	glorot_uniform	SGD	0.945
elu	glorot_uniform	RMSProp	0.948
elu	glorot_uniform	Adadelta	0.88
elu	glorot_uniform	Adamax	0.947
elu	glorot_uniform	Adagrad	0.946
elu	glorot_uniform	Nadam	0.948
elu	glorot_uniform	Ftrl	0.5
elu	he_normal	Adam	0.951
elu	he_normal	SGD	0.947
elu	he_normal	RMSProp	0.949
elu	he_normal	Adadelta	0.917
elu	he_normal	Adamax	0.95
elu	he_normal	Adagrad	0.949
elu	he_normal	Nadam	0.95
elu	he_normal	Ftrl	0.5
elu	he_uniform	Adam	0.95

elu	he_uniform	SGD	0.95
elu	he_uniform	RMSProp	0.95
elu	he_uniform	Adadelta	0.907
elu	he_uniform	Adamax	0.949
elu	he_uniform	Adagrad	0.949
elu	he_uniform	Nadam	0.95
elu	he_uniform	Ftrl	0.5
selu	glorot_normal	Adam	0.947
selu	glorot_normal	SGD	0.946
selu	glorot_normal	RMSProp	0.945
selu	glorot_normal	Adadelta	0.897
selu	glorot_normal	Adamax	0.947
selu	glorot_normal	Adagrad	0.947
selu	glorot_normal	Nadam	0.947
selu	glorot_normal	Ftrl	0.5
selu	glorot_uniform	Adam	0.947
selu	glorot_uniform	SGD	0.946
selu	glorot_uniform	RMSProp	0.942
selu	glorot_uniform	Adadelta	0.904
selu	glorot_uniform	Adamax	0.947
selu	glorot_uniform	Adagrad	0.947
selu	glorot_uniform	Nadam	0.947
selu	glorot_uniform	Ftrl	0.5
selu	he_normal	Adam	0.948
selu	he_normal	SGD	0.948
selu	he_normal	RMSProp	0.949
selu	he_normal	Adadelta	0.922
selu	he_normal	Adamax	0.948
selu	he_normal	Adagrad	0.948

selu	he_normal	Nadam	0.947
selu	he_normal	Ftrl	0.928
selu	he_uniform	Adam	0.949
selu	he_uniform	SGD	0.949
selu	he_uniform	RMSProp	0.949
selu	he_uniform	Adadelta	0.927
selu	he_uniform	Adamax	0.95
selu	he_uniform	Adagrad	0.948
selu	he_uniform	Nadam	0.949
selu	he_uniform	Ftrl	0.915

Figure 6.28 – Table of results for the Activation, Initialization, and Optimization evaluation Planarity models

The most significant finding of these experiments were the models that managed to break into the 0.95 accuracy range, all of which came as a result of the Elu activation function and the He Uniform or He Normal initialization algorithms. This supports the idea that further improvement is possible through the use of more specialised components.

6.4. Final Models

Based on the total findings of the evaluation the following final exemplar models were generated.

For Connectivity the selected hyperparameters are:

- Batch Size: 8
- Epochs: 100
- Layers: 2
- Learning Rate: 0.00001
- Neurons: 32
- Activation Function: Relu
- Optimizer: Adam
- Initializer: Glorot Uniform
- Dropout Rate: 0%
- L1 Regularisation: 0
- L2 Regularisation: 0

This model achieves a 0.902 training accuracy, a 0.887 validation accuracy and a 0.899 test accuracy. The confusion matrix for this model is shown in Figure 6.29.

For Degree Sequences the selected hyperparameters are:

- Batch Size: 128
- Epochs: 200
- Layers: 3
- Learning Rate: 0.001
- Neurons: 80
- Activation Function: Relu
- Optimizer: Adam
- Initializer: Glorot Uniform
- Dropout Rate: 0%
- L1 Regularisation: 0
- L2 Regularisation: 0

This model achieves a 0.005 training mean squared error, and a 0.002 validation mean squared error. The confusion matrix for this model is shown in Figure 6.30.

For Planarity the selected hyperparameters are:

- Batch Size: 4096
- Epochs: 500
- Layers: 3
- Learning Rate: 0.00001
- Neurons: 64
- Activation Function: Elu
- Optimizer: Adam
- Initializer: He Normal
- Dropout Rate: 20%
- L1 Regularisation: 0
- L2 Regularisation: 0

This model achieves a 0.945 training accuracy, a 0.949 validation accuracy and a 0.950 test accuracy. The confusion matrix for this model is shown in Figure 6.31.

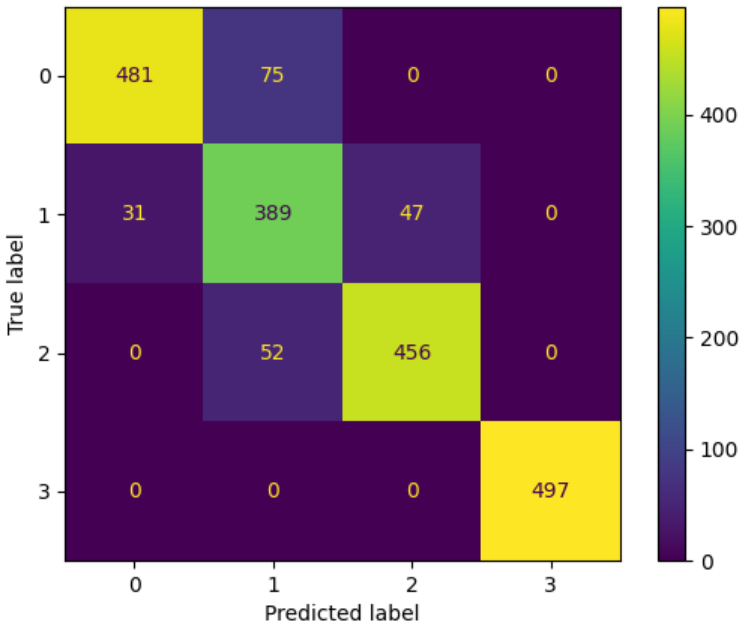

Figure 6.29 – Confusion Matrix for the exemplar Connectivity model

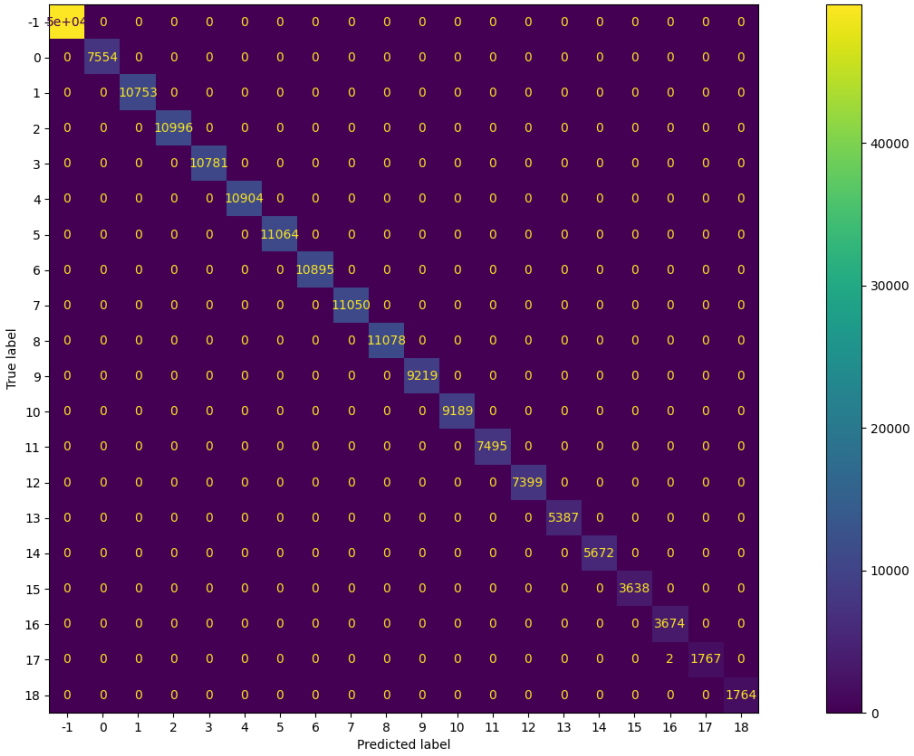

Figure 6.30 – Confusion Matrix for the exemplar Degree Sequence model

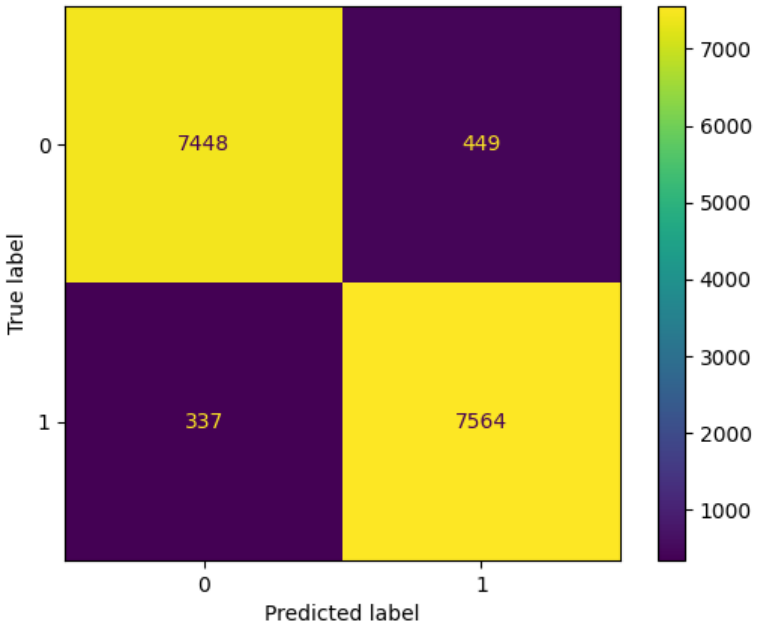

Figure 6.31 – Confusion Matrix for the exemplar Planarity model

7. Conclusion

The main goal of this project was to investigate and evaluate the capabilities of Deep Learning Neural Networks in learning to derive useful properties from graph data. To achieve this goal we have created a robust Graph Generation tool which produces viable datasets of randomized graph data based on user-specified parameters. We have produced varying datasets with different levels of scale and complexity that record useful properties for training a Neural Network. We have built a framework for training sets of Neural Networks with diverse hyperparameters on graph data and recording the results in an understandable and reproducible manner. Finally we have produced a set of pretrained Neural Network models capable of performing live predictions on unseen graph data with documented quantitative assessments on their performance.

We believe the data gathered in this report supports the view that general Neural Networks are capable of learning to derive properties from graph data. However the limitations of such methods must be made clear. While the properties of Connectivity and Planarity can be predicted at relatively high accuracies of roughly 89% and 95% respectively, no path was found to improve the models beyond these thresholds despite the wide range of hyperparameters searched. The speed at which these accuracies can be reached implies that the majority of graphs are trivial for the Neural Network to predict, but those that are non-trivial represent a major hurdle in learning. The slight variations in model loss suggest that the Neural Networks can learn from these more complex graphs but it may require more advanced data selection techniques and network architectures that are beyond the scope of this project. Degree Sequences, which represent a strict mathematical transformation rather than the more abstract classifications of Connectivity and Planarity, showed massive success with almost perfect accuracy being achieved.

In conclusion we feel the work represented in this project acts as a solid basis for future research into general Neural Networks and their applications in Graph Theory problems.

References

You, J., Liu, B., Ying, R., Pande, V. and Leskovec, J., 2019. *Graph Convolutional Policy Network for Goal-Directed Molecular Graph Generation v3*. [online] arXiv.org. Available at: <https://arxiv.org/abs/1806.02473> [Accessed 25 July 2021].

Leskovec, J., Chakrabarti, D., Kleinberg, J., Faloutsos, C. and Ghahramani, Z., 2010. Kronecker Graphs: An Approach to Modeling Networks. *Journal of Machine Learning Research 11*, pp.985-1042.

Gu, X., 2019. Explore Deep Graph Generation. [online] Snap.stanford.edu. Available at: <http://snap.stanford.edu/class/cs224w-2019/project/26420251.pdf> [Accessed 25 July 2021].

Fauske, K, M. (2006) Example: Neural Network. [online] Available at: http://www.texample.net/tikz/examples/neural-network/ [Accessed 26 July 2021].

Parahar, M., 2021. Connected vs Disconnected Graphs. [online] Tutorialspoint.com. Available at: <https://www.tutorialspoint.com/connected-vs-disconnected-graphs> [Accessed 27 July 2021].

GeeksforGeeks. 2021. Check if a graph is Strongly, Unilaterally or Weakly connected. [online] Available at: <https://www.geeksforgeeks.org/check-if-a-graph-is-strongly-unilaterally-or-weakly-connected/> [Accessed 1 August 2021].

Aric A. Hagberg, Daniel A. Schult and Pieter J. Swart, "Exploring network structure, dynamics, and function using NetworkX", in Proceedings of the 7th Python in Science Conference (SciPy2008), Gäel Varoquaux, Travis Vaught, and Jarrod Millman (Eds), (Pasadena, CA USA), pp. 11–15, Aug 2008

Zhou, J., Cui, G., Zhang, Z., Yang, C., Liu, Z. and Sun, M., 2018. Graph Neural Networks: A Review of Methods and Applications. [online] ResearchGate. Available at: <https://www.researchgate.net/publication/329841448_Graph_Neural_Networks_A_Review_of_Methods_and_Applications> [Accessed 8 August 2021].

Shchur, O., Mumme, M., Bojchevski, A. and Günnemann, S., 2019. Pitfalls of Graph Neural Network Evaluation. [online] arXiv. Available at: <https://arxiv.org/abs/1811.05868> [Accessed 8 August 2021].

Li, B. and Pi, D., 2019. Learning deep neural networks for node classification. [online] ScienceDirect. Available at: <https://www.sciencedirect.com/science/article/pii/S0957417419304853> [Accessed 8 August 2021].

Mercado, R., Rastemo, T., Lindelöf, E., Klambauer, G., Engkvist, O., Chen, H. and Bjerrum, E., 2021. Graph networks for molecular design. [online] IOPScience. Available at: <https://iopscience.iop.org/article/10.1088/2632-2153/abcf91> [Accessed 9 August 2021].

Cao, S., Lu, W. and Xu, Q., 2016. deep neural network for learning graph representations. [online] ResearchGate. Available at: <https://www.researchgate.net/publication/303495864_deep_neural_network_for_learning_graph_representations> [Accessed 10 August 2021].