Generating City Layouts using an Artificial Neural Network and Procedural Generation

Feliks Jakimow

2021

BSc (Hons) Computer Game Applications Development

School of Design and Informatics
Abertay University
Dundee

Contents

1 Introduction **7**
1.1 Aim 8
1.2 Objectives 8
1.3 Research Question 8
1.4 Hypothesis 8
1.5 Overview 9

2 Literature Review **10**
2.1 Generating Roads 10
2.1.1 Using Procedural Generation 10
2.1.2 Using Artificial Neural Networks 11
2.2 Generating Plots 13
2.2.1 Polygon Detection 14
2.2.2 Polygon Subdivision and Scaling 16

3 Methodology **17**
3.1 Preparing Training Data 17
3.2 Neural Network 18
3.3 Generating Roads 19
3.4 Plots 20
3.5 Performance Evaluation 21
3.5.1 Visualisation 22
3.6 Progress Bar 22

4 Results **23**
4.1 Generated Graphs 24
4.2 Generated Graph Statistics 25
4.3 Generated Plots 28
4.3.1 Verification Of Shortest Path 30

5 Discussion **31**
5.1 Training Sequence 31
5.2 Network Structure 32
5.3 Activation Functions 32
5.4 More Diverse Data Sets 33
5.5 Plot Detection 33

5.6 Plot Subdivision 34
5.7 Evaluation Methods 35
5.8 Project Effectiveness 35

6 Conclusion **37**

7 References **38**

8 Appendices **40**
8.1 A: Quad Graph Polygons 40
8.2 B: Bounding Box Coordinates Used 41
8.3 C: Python Packages Used 42

List of Figures

1 CityEngine is used to generate roads on Manhattan Island, New York, United States (top), versus the real-life road layout (Parish and Müller, 2001). 10
2 Ordnance Survey (n.d.) plot map, with house plots (left), tenement plots (right) in Edinburgh, Scotland. 13
3 Example output from Keir's (2020) *Procedural City Generator*. 14
4 Diagram of the ANN for road generation. 19
5 Bounding boxes used to query roads from. Retrieved using Racicot's (2012) bboxfinder. 23
6 The graphs used as input to to the generator to their respective networks. 24
7 Resulting output given the input in Figure 6 for the respective networks at their saved 10, 20, and 30 epoch stages. 24
8 Diversity metric (percentage of nodes that overlap with the ground truth map) for the generated maps and ground truth. . 25
9 Graph density of the ground truth versus the generated graphs at various epochs. 26
10 Road density distribution of the ground truth vs the generated graphs at various epochs. 26
11 Junction connectivity distribution of the ground truth vs the generated graphs at various epochs. 27
12 Transport convenience distribution of the ground truth vs the generated graphs at various epochs. 27
13 Bounding boxes used for road retrieval in 14. Retrieved using bboxfinder (Racicot, 2012). 28
14 Plot detection road input graphs, as retrieved from OSM. Numbers shown are OSM node ID's. 29
15 Detected plots, as retrieved from OSM. Numbers shown are OSM node ID's. 29
16 Three by Three quad lattice graph. 30

Thank you to Christopher Acornley for his guidance, advice, and supervision regarding this dissertation throughout the academic year.

Abstract

Creating city layouts manually can be difficult and time consuming. Automating this process to be simple and versatile would benefit many users, such as game designers and artists who might otherwise spent a lot of time studying road layouts manually. Existing implementations use Procedural Generation, which often puts the burden onto the user to capture the rules of the city. Artificial Neural Networks can be used to 'learn' the distribution of data automatically, given that it is correctly configured and a lot of data is available.

This project aims to implement a combination of Artificial Neural Network and Procedural Generation methods to generate city layouts and evaluate its effectiveness.

The study implements a variety of evaluation techniques which are to be performed on generated roads, to compare its effectiveness in relation to the source map. The roads are generated by a Recurrent Neural Network with Gated Recurrent Unit. This Artificial Neural Network is trained on data extracted from OpenStreetMap. Plots are then detected using the Dijkstra algorithm and evaluated.

The road generation Artificial Neural Network was unable to capture the source road map effectively. The plot detection algorithm was achieved, however due to time constraints plot generation using polygon slicing and subdivision was not implemented.

The study offers a basis for future development of similar problems, especially as no source code for the used Neural Turtle Graphics approach is available. Further development is required in regards to road generation and plot subdivision, especially regarding using an embedding layer instead of manual one-hot encoding to reduce memory usage.

Acronyms

ANN Artificial Neural Network. 3, 5, 7, 8, 10, 11, 17–19, 31, 32, 35

API Application Programming Interface. 17

BFS Breadth First Search. 12, 19

Encoder-Decoder Encoder-Decoder. 11, 12, 18, 19

GRU Gated Recurrent Unit. 5, 11, 12, 18, 33

LSTM Long Short-Term Memory. 12

NTG Neural Turtle Graphics. 5, 11, 12, 15, 17, 19, 22, 35, 37

OSM OpenStreetMap. 3, 5, 7, 17, 29, 35

PG Procedural Generation. 5, 7, 8, 10, 12, 14, 17, 33

RNN Recurrent Neural Network. 5, 11, 12, 33

1 Introduction

Creating a layout of a city from scratch can be difficult. Frequently, the city may be based off real world designs. Due to this, consideration to the original style must be given to achieve a believable layout. This can be a difficult task due to the various parameters that one must consider: road length, angle, distribution, pattern, connectivity, building locations etc. Capturing the style accurately may prove to be difficult, as doing so requires extensive analyses of maps. This is time consuming and tedious.

Methods to automate this process have been devised. From Procedural Generation, which requires the user to manually identify rules, or patterns to follow (Parish and Müller, 2001); to using Artificial Neural Network, which must be optimised for identifying them (Chu et al., 2019). This is due to an Artificial Neural Network typically learning the distribution from the data provided. This can be done through a number of methods such as supervised training. In supervised training the network receives the input and the expected output. This data is given to the network and activation's tracked, to be later adjusted (back-propagated) by an optimiser typically using gradient descent. As a consequence, Artificial Neural Networks tend to rely on large amounts of specifically formatted data which can be processed and trained on. This training data must be sourced from an appropriate place, while being arranged into a suitable format and staying general enough (i.e. not 'overfitted') to be used outside the training sequence. Due to the nature of an Artificial Neural Network, the resulting model may be difficult to comprehend as it has often went through hundreds of thousands iterations.

Maps can be found easily online, using services such as OpenStreetMap (Curran, Crumlish and Fisher, 2012). The data is frequently provided in a simplified format that can be represented in an undirected graph. This graph can contain information about the nodes and edges it houses. Analysis can be performed on this data to generate conclusions, but also training data for an Artificial Neural Network (Chu et al. 2019).

An Artificial Neural Network can be used in conjunction with Procedural Generation, which is frequently used for more specialised tasks, however does not require as much input data. These two methods could be fused together to help game designers and artists when designing a city-based environment for a video game, or perhaps create new, procedural gameplay experiences in realistic city environments.

1.1 Aim

This dissertation aims to implement and evaluate a method of generating city layouts using a combination of Artificial Neural Network and Procedural Generation. The use of an Artificial Neural Network for road generation should be more flexible over a Procedural Generation approach due to the ability to 'learn' layouts autonomously and on a range of source layouts. In conjunction with Procedural Generation, plots can be found for where buildings can go, and help to outline shapes and sizes for buildings.

1.2 Objectives

- Develop a method of extracting data from real world maps.
- Convert the extracted map data into a suitable format for generating a training sequence.
- Implement a way of generating training data for the network.
- Research and evaluate approaches to generating roads using Artificial Neural Networks.
- Evaluate the performance of the created Artificial Neural Network.
- Research a method of generating plots using Procedural Generation.
- Evaluate the performance of the plot generation method.
- Provide recommendations on improving the chosen approach.

1.3 Research Question

How effective is the combination of an Artificial Neural Network with Procedural Generation at generating city layouts accurately?

1.4 Hypothesis

The Artificial Neural Network will generate roads styled after the input map, and Procedural Generation will be able to generate realistic looking plots.

1.5 Overview

Following this section the literature review will discuss current approaches to generating roads and plots. Next, the methodology will describe the taken approaches to acquiring training data, network structure, and plot generation. Continuing, the results will show the quantitative and qualitative data, and the discussion will interpret the data, and suggest improvements to the approach. Finally the conclusion will summarise the effectiveness of the approach.

2 Literature Review

The layout of a city can be considered to consist of two items: roads and plots. Buildings (or other structures such as parks) are placed on these plots, and they are divided by roads which interconnect the plots. The shape of the roads and plots vary between cities. These are important qualities to consider when creating a city layouts. Generating roads and plots can be considered as two separate procedures, despite being interlinked with each other.

2.1 Generating Roads

The generation of roads can be approached using Procedural Generation (PG), Artificial Neural Networks (ANNs), or a combination of both. Both of the approaches require some initial data to guide the algorithm, or for the user to determine how close the output is to the desired effect.

2.1.1 Using Procedural Generation

Figure 1: CityEngine is used to generate roads on Manhattan Island, New York, United States (top), versus the real-life road layout (Parish and Müller, 2001).

Typically roads have rules they tend to follow, therefore using a rule based approach can be used to generate fairly convincing cities. An example of this given by Parish and Müller (2001) with their CityEngine approach, which uses extended L-systems to determine the successor road based on a combination of factors: preceding road, global goals, and local rules. Global goals consider factors such population density, general road patterns such as radial, checker, and branching; local rules consider whether the new piece would be in water, overlapping another piece, or whether some criteria with highway roads are met.

This approach generates believable cities, however it involves creating the rules for the generator manually, proving tedious (Chu et al., 2019). In addition, cities which do not follow rules (such as Istanbul, Turkey), have different rule sets, or lack planning. This makes them difficult to adequately describe with rules, causing generation performance to suffer (Chu et al., 2019).

2.1.2 Using Artificial Neural Networks

Artificial Neural Networks (ANNs) have been used for various applications, such as predicting values, generating text, and classifying images. Various architectures are used for different problems. In the case of generating roads, two prominent examples Neural Turtle Graphics (NTG) and GraphRNN have used the Recurrent Neural Network (RNN) architecture to facilitate the generation of roads.

A RNN is a type of neural network that *remembers* the previous state of the network, through time steps. Every time a new input value is given to the input, another version of the network is created and connected to the previous one, generating a new time step. Typically the final time step is used to predict the next value in the chain.

A common use of an RNN is the Encoder-Decoder architecture, which uses two separate RNNs. The encoder network encodes the input data into a fixed-sized intermediary vector, which is then decoded by the decoder counterpart. This allows the input to be a different size from the output.

In order to achieve the interconnection between other time steps, a specific architecture of the *neurons* must be picked that supports this. A common architecture is the Gated Recurrent Unit (GRU). This architecture typically contains two inputs, one which determines how much of the incoming data should be passed to future ones, and the other determines how much of the

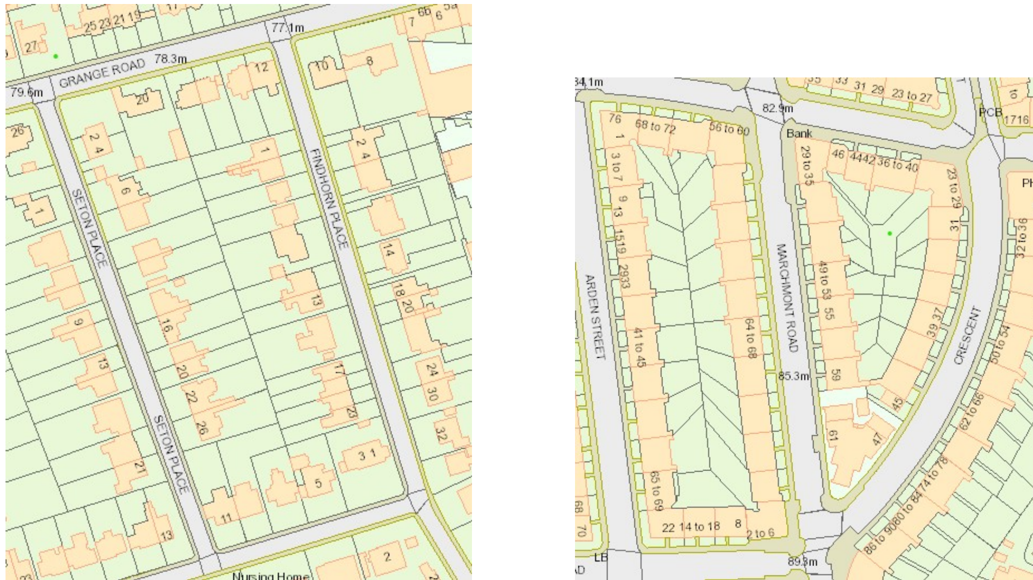
data should be forgotten. This data is then passed to other *neurons*, and the *neuron* in a future time step. GRU is typically used over Long Short-Term Memory units as it improves on the vanishing gradient descent problem (Dey and Salem, 2017).

The GraphRNN approach, as described by You et al. (2018), uses a GRU Encoder-Decoder RNN architecture to predict the next road. The roads are stored in an undirected graph format. The first network encodes the state of the graph, whereas the second network generates the next node. The state of the graph is encoded using an adjacency matrix, which is then predicted from the encoded graph.

Chu et al. (2019) describes their approach, NTG, which works similarly to a graphics turtle (a *turtle* is given commands to move). It has similarities with GraphRNN, where an undirected graph is used to store the nodes, and an GRU Encoder-Decoder RNN is used. However, instead of using an adjacency matrix generate represent and generate new nodes, the relative location change to each other is used. A node is generated by using forty previous nodes, sorted by Breadth First Search (BFS).

Overall, NTG achieves better generation performance than GraphRNN as it fails to capture the city style due to its lack of positional encoding in adjacent nodes and ability to encode complex graphs (Chu et al., 2019; You et al., 2018). For this reason, the NTG approach was chosen. In addition, the network identifies the rules its self, unlike PG, increasing versatility to almost any city big enough.

2.2 Generating Plots



(a) Real world plot data from around Cumin Place, Edinburgh, UK.

(b) Real world plot data from around Marchmont Road, Edinburgh, UK.

Figure 2: Ordnance Survey (n.d.) plot map, with house plots (left), tenement plots (right) in Edinburgh, Scotland.

Building plots are frequently designated around roads inside of cities. Due to abundance of roads inside of the city, these plots are surrounded by roads (see figure 2). For this reason, the detection of areas between roads was selected. As a result, a plot can be represented as a cyclic path or polygon. These polygons should later be subdivided and scaled to form usable plots, similar to ones shown in figure 2.

Generally, the size of the plot corresponds with the size of the building, and buildings may be connected together. In other cases plots may be used to determine parks. In the context of a plot generation algorithm, sometimes large spaces may be left undivided.

Keir's (2020) *Procedural City Generator*[1] approach (Figure 3) is designed to create "[North] American grid based city" layouts. This incorporates a

[1]The project title differs between the *GitHub* and *itch.io* or *maps.probabletrain.com*

Figure 3: Example output from Keir's (2020) *Procedural City Generator*.

PG approach for both road and plot generation, with some context for parks and unfilled plots. The following sections discuss Keir (2020) approach to detection and the subdivision/scaling process of polygons.

2.2.1 Polygon Detection

An implementation of plot generation has been devised by Keir (2020) in their approach using tensors. This method is based on finding the areas between roads generated by the road generator. The plot detection algorithm attempts to find a three or four sided polygon. Firstly the comparison vector created. The comparison vector is defined between the penultimate and last visited nodes. A second *adjacent* vector is made between the positions of the last visited and adjacent node. Subsequently, the angle between the comparison and adjacent vector is calculated using $\text{atan2}(y_1 - y_2, x_1 - x_2)$ where x_1, y_1 is the comparison vector, and x_2, y_2 is the adjacent vector. This process is repeated for every adjacent node, and the node with the 'rightmost'

(documentation page) listings. The *itch.io* title has been used as the name for this approach.

angle is picked. Each rightmost node is added to a visited queue, from which a polygon is formed and stored.

This detection method is unable to backtrack if it finds its self stuck in a dead end due to the angle calculated. This approach works in conjunction with the road generator designed by Keir (2020) as it generates quads with no dead ends. Therefore this method unsuitable for real world maps where dead ends frequently exist.

As the picked road generation approach, NTG by Chu et al. (2019), represents maps using an undirected graph, a path finding algorithm can be used. Two popular graph-based algorithms A* (implemented by Hart, Nilsson and Raphael, (1968)) and Dijkstra (implemented by Dijkstra (1959)) can be used for finding paths.

Both the A* and Dijkstra algorithms 'flow out' towards the start point from the destination, and then (if the starting point is found) work their way back by finding the shortest path using the calculated flow out values. The flow out process involves associating a value with each node, which is defined by the weight of the edge between the two nodes (usually number of jumps from the destination). The flow out process continues until either the start point is found or the whole graph is visited. A node can only be visited once. A* differs from Dijkstra as it determines the next node to perform the flow out calculation on by estimating the distance away from the start node. The heuristic on an edge-weighted undirected graph, as suggested by Hart, Nilsson and Raphael (1968) is calculating the flow out values on the lowest weighted edge.

Once the starting point is reached, the both of the algorithms attempt to make their way to the destination point based on the calculated flow out values. A path is mapped from the starting node, by iterating through adjacent nodes with the lowest flow out values. This results in the shortest possible path.

In the case of the NTG map graph, an edge's weight is defined by the euclidean distance (or equivalent for the geographic coordinate system) between the two nodes that it connects (as each of the nodes contain a positional value). Moreover, this heuristic can be modified to consider which node is closest to the starting point, by calculating the euclidean distance between it and the start point.

Dijkstra, unlike A* calculates the values of the whole graph until it reaches the destination node. In the case of the suggested application (plot detection), this may be beneficial as less flow out operations must be performed

given that the result of the previous flow out is cached. This may decrease the amount of calculation and reduce the programs' complexity by reducing code. Due to these reasons, Dijkstra will be chosen as the preferred method.

2.2.2 Polygon Subdivision and Scaling

In Keir's (2020) *Procedural City Generator* approach, after detecting plot polygons, they are subdivided to form building lots. These lots may later be used by other parts of the program to form buildings and create 3D renders.

Polygons are first shrunk to ensure that every side is an equal distance from the surrounding roads. Subsequently, the polygon is recursively subdivided until a new polygon with a defined minimum area is produced. This subdivision process is repeated for most nodes, though includes a probability that some spaces will not be subdivided. The subdivision is done with the help of the library PolyK. This library uses a line intersection algorithm to define create two new polygons.

The drawback of this approach is that it results with plots which are inaccessible to roads (see Figure 2). However, this subdivision process may be modified as polygon slicing may be used on multi angle polygons. This will be the preferred approach due to its flexibility and simplicity.

3 Methodology

The NTG approach was chosen for the generation of roads due to the generation performance described by Chu et al. (2019). NTG requires multiple acyclic input paths to be provided. This data should then capture the layout of the city accurately once trained. Consequently, the map data is represented using an undirected graph.

The project would be ultimately aimed towards artists and designers. However, as this application will be a prototype to test the feasibility of generating city layouts using ANN and PG, a graphical user interface will not be provided. Though performance is not critical, improvements to reduce the overall execution time are welcome.

3.1 Preparing Training Data

The map data was sourced using OpenStreetMap (OSM), as it contains a map of the world with an open API allowing the use of the stored data easily. Initially, roads are fetched from the database (entries tagged as `highway`) from a defined bounding box. The information, namely the nodes' longitude and latitude, ID numbers, and connections is then parsed into an undirected graph, called the *city graph*. The connections between the nodes are represented as edges between nodes. The `NetworkX` library (Hagberg, Schult and Swart, 2008) was used for representing the undirected graph in the implementation.

Due to one-hot encoding (also commonly referred to as cardinality), any nodes that are more than one hundred meters apart, are split to ensure that they can be correctly encoded for training. This method takes into account each of the node's geographic coordinate system (i.e. longitude and latitude).

A training sequence is then generated based on the city graph. The input sequence consists of up to fifty acyclic paths leading to a node, each of which contain twenty path nodes (half of the NTG specification, due to reasons mentioned later). There are up to twenty individual sets of fifty paths per node. A path is generated by getting various paths to all of the nodes up to twenty nodes away. The prediction is up to six nodes.

The nodes positions are then encoded into Δx and Δy between the node and its successor (or in the case of the prediction, the current node and its successors). This is to ensure that the nodes are not dependant on the predictions position (Chu et al., 2019). Additionally, any paths that do not

have the full twenty nodes have their change in position marked as zero in their path. Likewise, if the node does not have twenty previous paths leading to the node, path(s) with no positional changes are added in order to ensure that the length of the training sequences are consistent. All empty positions and paths are added to the start of the training sequence, as information closer to the end of the training sequence is inputted last, therefore has more impact on the output (Chu et al., 2019). Moreover, the padding also helps to ensure that the sequence is a consistent size, hopefully making the ANN learn the data pattern.

The prediction goes through an almost identical process, except there are only up to six nodes that can be predicted. Most nodes (except a dead end) will have at least one node to predict. Unlike the input, the nodes are not shifted 'forward', as the input is most likely to be 'remembered' from the start.

Lastly, all the changes in distances are encoded using one-hot encoding. This encoding associates each number into its own index,. Any non-whole number is rounded. The 'number of features' must be twice the size of the maximum absolute value of a distance (in this case, one hundred meters) in order to account for negative values, as the `keras.utils.to_categorical` function does not create indices for negative values. Later, during decoding half of the maximum possible value is added.

This process is used for up to a thousand nodes, in order to not exceed hardware limits on memory. Likewise, the `batch size` has been set to the maximum number of input paths. This is in order for whole paths to be processed before the back propagation/gradient descent occurs (Keras Team, no date). Otherwise, accuracy may be affected negatively if a whole sequence is not trained on.

The training process uses the ADAM optimizer, as described by Kingma and Ba (2014). This optimizer was picked due to the low memory requirements, and computational efficiency. This in turn would allow a greater number of samples to be trained on due to the increased space in memory.

3.2 Neural Network

The ANN is implemented using the Keras library. This is as the Keras library provides an easy to use implementation of the required GRU units, as well as training capabilities etc.

The ANN (Figure 4) uses an Encoder-Decoder architecture, as per the

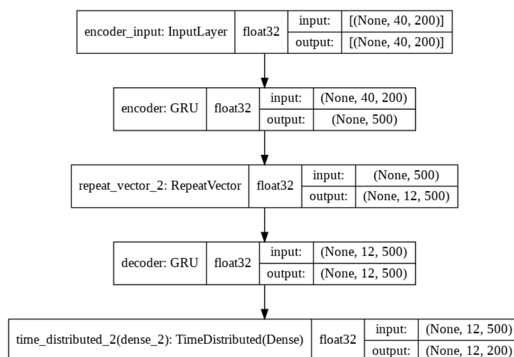


Figure 4: Diagram of the ANN for road generation.

NTG approach described by Chu et al. (2019). One of the key differences include that there is no attribute vector for the name of the city being trained, not allowing the training on multiple networks. Both the encoder and decoder part of the network use five hundred units, as descibed by Chu et al. (2019), as they achieved a high generation accuracy.

The one shot model architecture was picked for the model. This is where the network is given a set of incoming paths that are then used to predict an output. Other network Encoder-Decoder architectures were considered, such as the recursive model, however Keras does not support the implementation of such architectures (Brownlee, 2017).

Another design decision made was creating the one-hot encoding of the data manually, instead of an Embedding layer due to initial concerns over training performance. This topic is further discussed in the discussion section.

The tanh activation function was used on the encoder and decoder. This was as the values are more aggressively shifted to 1 or -1, unlike Sigmoid, which does not shift values as aggressively. This is in addition to performance benefits (Nwankpa et al., 2018).

3.3 Generating Roads

After training, a section of the source city graph is extracted. The number of nodes extracted, as well as the starting node can be configured. This is as due to the ANN used learns a 'language', which requires existing inputs to generate new ones. This map extract is generated using BFS, where the

initial node is located at (0, 0), and subsequent nodes positions are calculated by their change in x, and y. The use of a Cartesian graph was done to simplify the addition of new nodes, as otherwise the generated coordinates would have to been changed to a geographic coordinate system. Such an approach would've been more complicated to implement than the extraction of the map.

A stack of nodes which have yet to be processed is generated, with each of the nodes added to the stack. A training sequence is generated for each of the nodes. The node at the start is popped off the stack, the input sequence to the network is generated (albeit with a different distance function to account for the map extract graph using Cartesian coordinates), and passed to the network. The network then predicts the outgoing node(s).

As the network is made to return all generated time steps (via the `return_sequences=True` option when defining the decoder), the last time step is used to base the prediction. This was kept for diagnostic and debugging purposes.

The generated prediction is a set of up to six nodes. These nodes are ordered in a two dimensional shape, where the first dimension is the number of nodes multiplied by the number of dimensions (2D), and the second one the number of features. These distances are decoded by reversing the one-hot encoding (finding index with the largest value), then subtracting one hundred (i.e. half of number of features/cardinality) to restore the encoded negative values. These values are then converted into Δx, Δy pairs and returned.

These values are added to the map, given that all the generated positions are not zero (as this is an indication that there are no new adjacent nodes). Any nodes generated are added to the stack for later processing. This continues until either the stack runs out of nodes, or a limit (in this case five hundred) is reached.

3.4 Plots

The implementation of plot generation only features plot extraction. This is due to time constraints. Further developments on plot generation are mentioned in the discussion section. Plots are represented as areas inside roads. These areas can be represented as polygons. As the polygons contain an arbitrary number of sides depending on the generation, a path finding algorithm was used to find them.

The Dijkstra algorithm was chosen as it finds all possible paths, which can later be cached for finding later paths. Though the utilized library

`return_sequences=True`which this algorithm was used does not necessarily guarantee this. The path finding algorithm is then applied on the undirected graph to find the shortest cyclic paths.

The start point is set as one of the adjacent nodes, and the destination point as the current node. Subsequently, the edge that connects the two nodes is temporarily removed, to ensure that the next path found is not directly back to the start node. The generated path is effectively the shortest cyclic path. Then, this path is stored in a collection of paths given that the path does not exist in another ordered permutation, which may be mirrored. The check involves rotating the path, as well as mirroring the rotated path to see if it already exists in the collection. This is computationally expensive as if a node does not exist, the worse case scenario of $O(n \times n)$ is reached. As the collection of polygons must be iterable, it was implemented as a `list`. The visited adjacent node is then added, and will not be visited again. This detection process is repeated for every node in the graph to ensure that every polygon will be found.

3.5 Performance Evaluation

Performance was evaluated by implementing similar evaluation to Chu et al. (2019), as it provided a comprehensive overview of the road generation performance with both qualitative and quantitative techniques. For plot generation qualitative analysis was chosen, as due to the nature of plot data, it cannot be necessarily retrieved as easily or potentially require a subscription to access the API.

The performance metrics for road generation are included as follows:

- Road Density - Calculated using the `networkx.classes.function.density()` function, which uses the following equation: $density = \frac{2m}{n(n-1)}$ where n is number of nodes, and m is number of edges.
- Junction Connectivity - Counting the number of adjacent nodes each node in the graph has.
- Road Length - Measuring each of the edge's lengths.
- Diversity - Number of overlapping nodes with in 10 meters of the ground truth.

- Transport Convenience - Length of shortest path from all nodes to 25% of all nodes, to reduce calculation time.

This deviates from Chu et al. (2019) NTG approach and the initial proposal by:

- Changing road density calculations to use graph statistics instead of road density in 100m, 200m, and 300m neighbourhoods as the generated sequences did not have enough roads.

- Changing transport convenience to be calculated by all nodes to random nodes, as the generated map may not be 500m wide.

- Not using a neural network to gauge perceptual similarity, as this was considered out of scope.

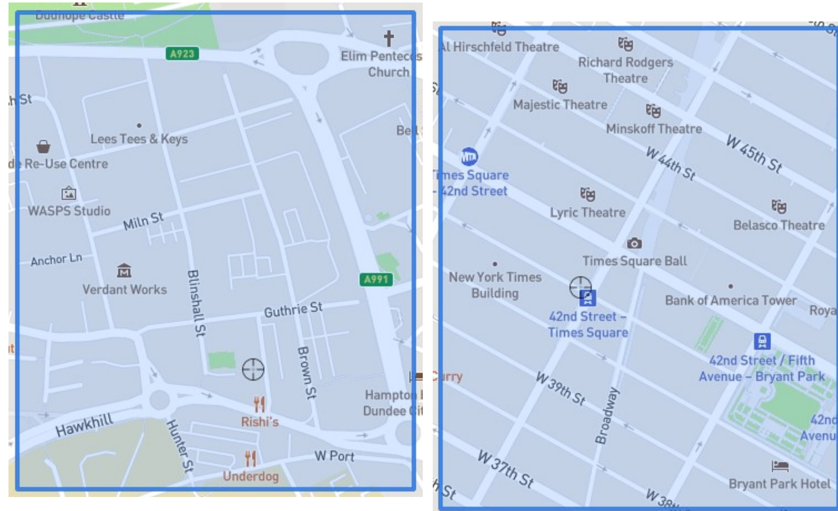
3.5.1 Visualisation

The `NetworkX` library provides visualisation capabilities. The `Matplotlib` library is used by `NetworkX` as the back end to render the maps. This allows easy rendering of nodes at specific positions with the connecting edges. However, for an unidentified reason the edges between nodes may not be rendered correctly, despite existing in the graph.

3.6 Progress Bar

The ProgressBar2 library (van Hattem, 2020) was used to indicate progress of various operations in order to provide an indication of the stage of the program.

4 Results



(a) Dundee Central, UK bounding box.

(b) New York, US bounding box.

Figure 5: Bounding boxes used to query roads from. Retrieved using Racicot's (2012) bboxfinder.

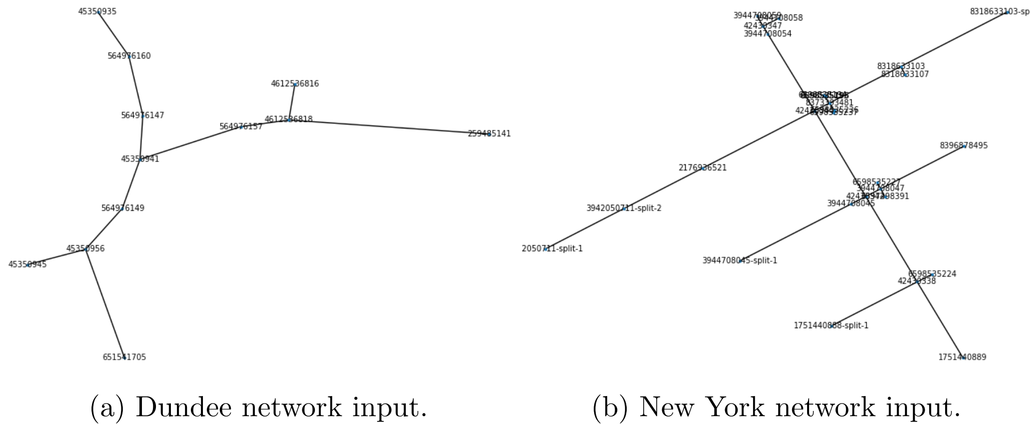
The evaluation was conducted on two unique areas:

- Dundee City Center, Dundee, UK around Dundee Central Mosque.
- New York, US around Times Square.

The two areas were picked as they had relatively unique styles. New York contains a grid layout with a line running through it. Whereas, the Dundee area is less ordered.

The roads from these areas were extracted, converted to a graph format, and made into a training sequence. The networks were then trained on this data. Two separate networks were created for each of the areas, and trained up to thirty epochs, and saved every ten, resulting in three networks at ten, twenty, and thirty epochs. These were then compared versus the ground truth.

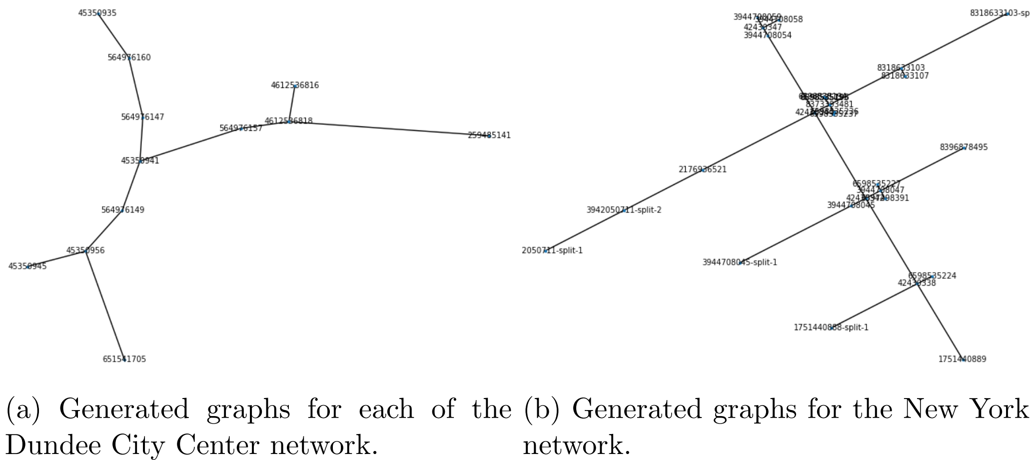
4.1 Generated Graphs



(a) Dundee network input. (b) New York network input.

Figure 6: The graphs used as input to to the generator to their respective networks.

The starting node (and by extent the input graph) was selected as it was around the beginning of the node collection, which guaranteed their inclusion in the training set.



(a) Generated graphs for each of the Dundee City Center network.

(b) Generated graphs for the New York network.

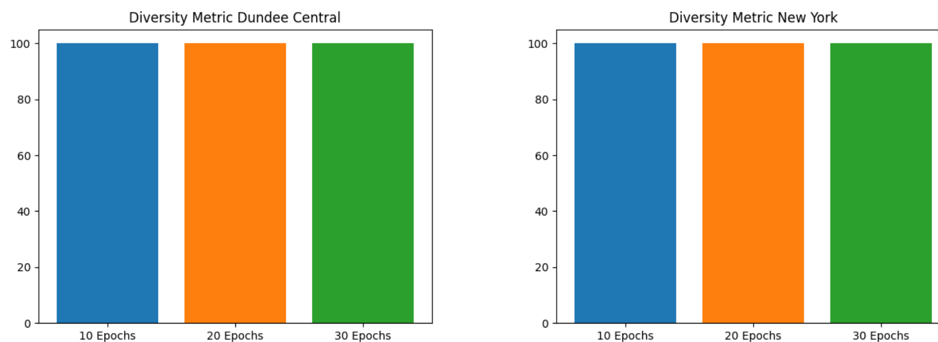
Figure 7: Resulting output given the input in Figure 6 for the respective networks at their saved 10, 20, and 30 epoch stages.

As Figure 7 shows, the networks did not generate anything given the

inputs. Additional starting maps were tried, with different starting input graphs, however the output was the same.

4.2 Generated Graph Statistics

The qualitative analysis was still performed despite the lack of generation. This means that all generated graphs will be identical, with the exception of the ground truth.

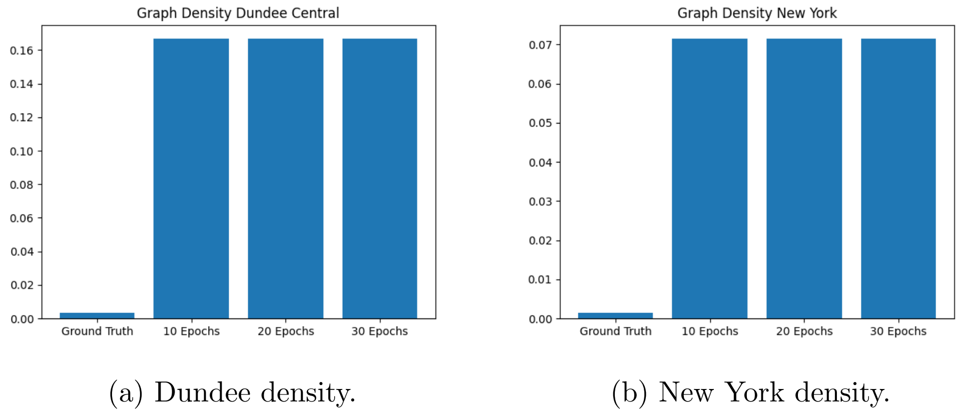


(a) Dundee diversity metric.

(b) New York diversity metric.

Figure 8: Diversity metric (percentage of nodes that overlap with the ground truth map) for the generated maps and ground truth.

As the generated graph only contained the input graph, which is a subset of the ground truth graph, all nodes present overlap therefore resulting in no diversity.

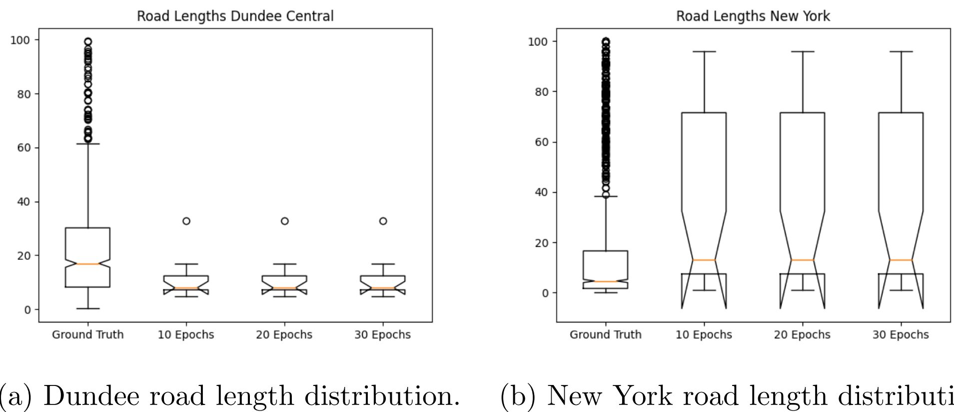


(a) Dundee density.

(b) New York density.

Figure 9: Graph density of the ground truth versus the generated graphs at various epochs.

Each generated graph's densities have a high disparity between the ground truth, which remains identical through the various epochs. This is due to the lack of generated nodes, and much lower amount of nodes and edges in comparison to the ground truth.



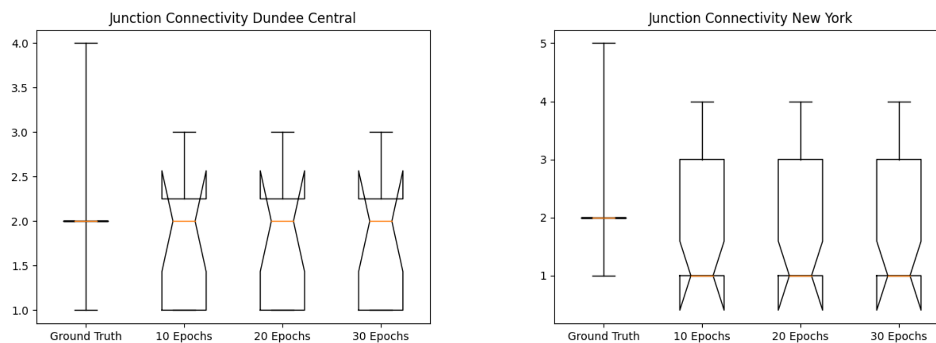
(a) Dundee road length distribution.

(b) New York road length distribution.

Figure 10: Road density distribution of the ground truth vs the generated graphs at various epochs.

The distribution of road lengths was not captured by any generated network. The present distribution is only a subset of the ground truth. They remain identical throughout the epochs as no new nodes were generated. Due

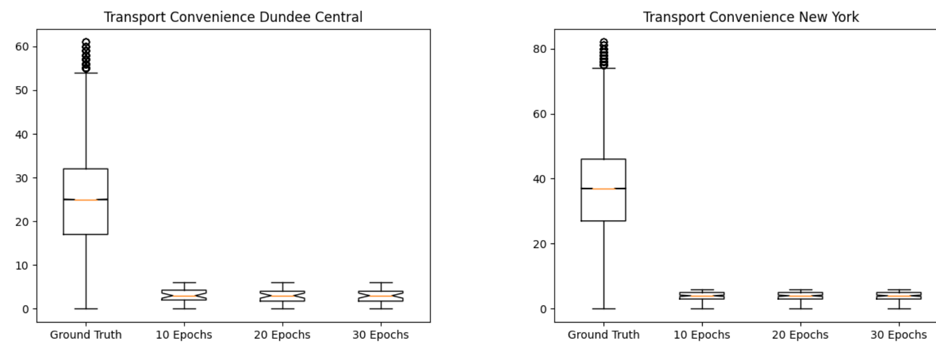
to the lack of data, the box and whisker plot was not accurately conveyed. Though edges in Dundee Central have a tendency to be mostly longer than New York, however New York has more outliers and some very long distances.



(a) Dundee junction connectivity. (b) New York junction connectivity.

Figure 11: Junction connectivity distribution of the ground truth vs the generated graphs at various epochs.

Similarly to Figure 10, there was too little data to generate an accurate graph of the distribution. Though, most nodes have at least two junctions, which indicates that most roads in New York are not junctions. Dundee Central has a similar distribution, though some outliers may more junctions.



(a) Dundee transport convenience. (b) New York transport convenience.

Figure 12: Transport convenience distribution of the ground truth vs the generated graphs at various epochs.

The transport convenience between the different epochs because no new nodes were generated. The ground truths are slightly different from each other, where generally the number of number of nodes travelled is lower in Dundee, however the the results are more consistent across New York.

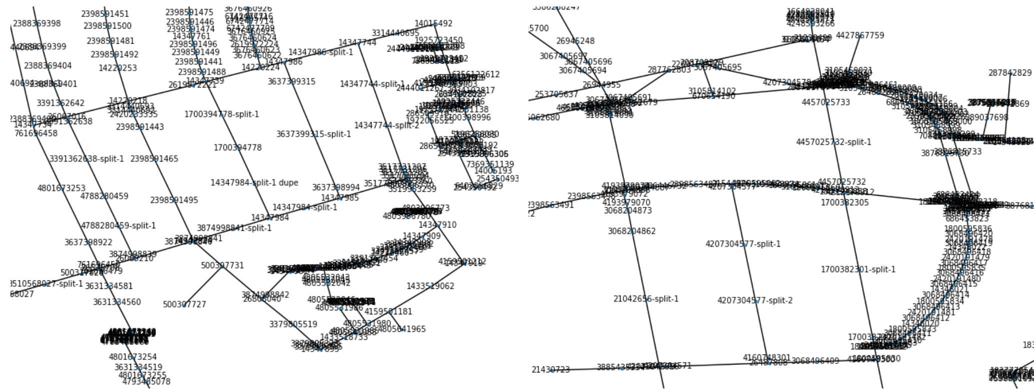
4.3 Generated Plots

Plot detection was performed on the same areas as Figure 2. This is to be able to provide a direct comparison to real world plots.

(a) Bounding box around Cumin Place, Edinburgh, UK.

(b) Bounding box around Marchmont Road, Edinburgh, UK

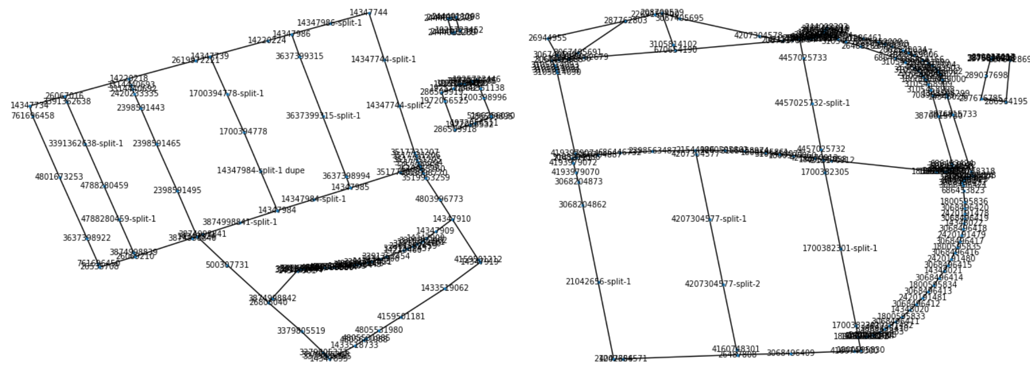
Figure 13: Bounding boxes used for road retrieval in 14. Retrieved using bboxfinder (Racicot, 2012).



(a) Around Cumin Place, Edinburgh, UK.

(b) Around Marchmont Road, Edinburgh, UK

Figure 14: Plot detection road input graphs, as retrieved from OSM. Numbers shown are OSM node ID's.



(a) Detected plots from around Cumin Place, Edinburgh, UK.

(b) Detected plots from around Marchmont Road, Edinburgh, UK

Figure 15: Detected plots, as retrieved from OSM. Numbers shown are OSM node ID's.

For Cumin Place (Figure 15a) a total of ten polygons were found. For Marchmont Road (Figure 15b) thirteen polygons were found. Two of the retrieved polygons towards the bottom of the graph, from Figure 14a do not contain the interior roads, as expected. All closed polygons were retrieved. The triangular polygons towards the top of Figure 15b appear to be from the

paths in the park above. While a closed polygon, the independent polygon at right of figure 15a was created by a road which had a path at the end.

4.3.1 Verification Of Shortest Path

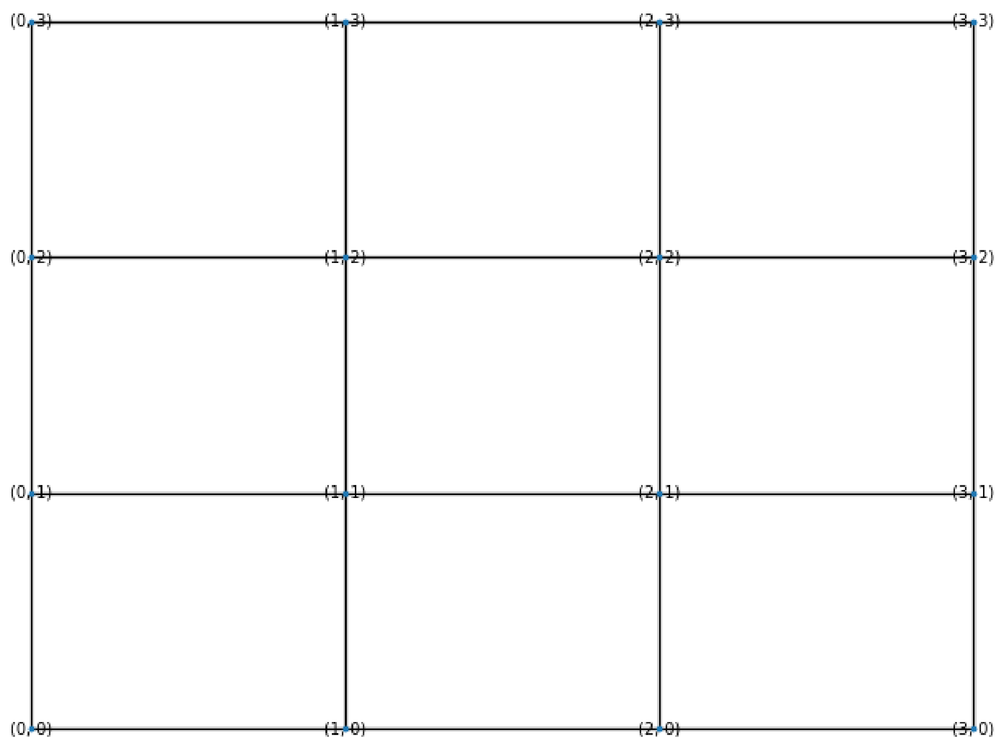


Figure 16: Three by Three quad lattice graph.

To verify that the found polygons are actually of the shortest paths, and do not contain loops that generate similar shapes, a test case of detecting polygons on a graph of quads was performed. The input graph consists of a lattice of quads.

The number of polygons found were nine, and all had four components. This is the expected result. The results may be viewed in section 8.1.

5 Discussion

The neural networks were unable to capture the essence of the input graph. This is evident throughout the results section, most notably on 7 where none of the networks, at any training epoch generated nodes. This could be due to a variety of reasons, which will be discussed in this section. Additionally, the plot detection algorithm has found all possible closed polygons, which was a success. This section will also discuss potential implementations of plot subdivision and review how the current plot generation performs.

5.1 Training Sequence

The training data has fifty input paths paths per node, with each path consisting of twenty nodes, and each node has two dimensions of two hundred features, encoded as a `float32`. This resulted in a very large amount of training data, especially once one-hot encoded. Additionally, due to the padding required to get the data consistently sized, a lot of the data was filled with zeros.

Due to the amount of data the initial training sequence for the New York network was over 12.2GB in size on disk. This is in contrast to the non encoded training data (i.e. before padding certain parts of the sequence and performing one-hot encoding), which was just 16MB on disk while including the prediction data. This resulted in little real training data to be passed in to the network to train on, and as it requires to learn the 'vocabulary' of the map, many sequences may be required.

This may be resolved by using an embedding layer instead of manually one-hot encoding, as it would reduce the number of space taken up by the training sequence substantially. Another solution would be to encode the data just before training, however an embedding layer provides near identical functionality, while requiring less code. While training may take longer as all of the encoding would take place before being passed into the encoder, the ability to train on more samples before exceeding memory capacity limits may prove more beneficial as the ANN could capture the input graph more accurately. Additionally, this would allow increasing the number of previous nodes and paths to numbers specified by Chu et al. (2019) - potentially improving generation performance to their levels.

The currently described approach may be more beneficial on machines with large amounts of memory (as the training sequence must fit in memory

for training to be performed) by speeding up the network training. This would also allow more experimentation with adapting the network structure to achieve better generation performance.

Another method which may aid in improving the network's capabilities is by shuffling the sequences, so the possibility of the network over fitting on a certain node and path set is reduced. A potential pitfall of the current training strategy is that the network learned to expect twenty sets of fifty paths per node. This could be mitigated by shuffling the sets of nodes in a random order to prevent this from happening.

Due to the way the training data was stored, the ability to accurately gauge loss and accuracy was negatively affected. The training process required iterating through the training data in sets of twenty nodes, and further iterated by the number of epochs. The issues regarding the unnecessary iteration can be solved by resizing the data, however due to the method of one-hot encoding, memory limitations were met when trying to reshape the data. This could be resolved by changing to an embedding layer.

5.2 Network Structure

The network structure picked may be another culprit of why the ANN was unable to capture the structure of the input graph. Brownlee (2017) suggests trying a different structure for language model. Especially as they may help reduce the dependence on the encoder to capture the state of the graph (Brownlee, 2017). Though, if the suggested approach of feeding the input back into the network is to be taken, additional consideration would have to be given onto how the data should be summarised during the training sequence. Moreover, the two described approaches would require specific knowledge of TensorFlow to create a suitable network. This is in contrast to the current 'one shot' approach described by Brownlee (2017) which is easy to implement. Overall, consideration may be given to this if changes to the training data would not suffice.

5.3 Activation Functions

The current activation function is Tanh. According to Nwankpa et al. (2018), Tanh suffers from vanishing gradient descent, which may prevent it reaching the local minima. Moreover, the function's maximum value of 1 is only

achieved when the input is zero. This would mean that not every GRU unit is being used. Such limitations can be rectified by using ReLu.

ReLu is a development on the Tanh function, though RNNs are not listed as a common use, it is used for similar applications such as speech recognition (Nwankpa et al., 2018). Additionally, the ReLU activation function offers better performance, which may offset some of the performance impact of switching to an embedding layer. Nevertheless, ReLU is prone to over fitting and gradient descent issues (Nwankpa et al., 2018). Though, further investigation into this topic would have to be performed before reaching a definite conclusion.

5.4 More Diverse Data Sets

The input graphs were based on areas in the UK and US. More diverse data sets may be tried from different parts of the world. This would give a more accurate gauge of performance across different cities, but also provide more options for the intended end users, whom may want to set their game in a city within another place in the world.

5.5 Plot Detection

The plot detection algorithm found all expected polygons, while ensuring that they are the shortest (see Figure 16). This forms a good basis for later subdividing the plots into usable areas, and meets the objective of generating plots using PG. Some additional work may be beneficial for achieving better plots. Improvements such as detecting dead ends as to know where not to place plots, or to subdivide the larger polygon where a dead end is detected, if plots are to be generated in a similar fashion to Figure 2.

An area that it falls behind in is when detecting areas in between roads, such as towards the right in figure 15b. These plots may not be suitable for buildings, and should be marked as such. This issue however, should be tackled by the plot subdivision algorithm when determining which plots are marked as such, or even to expand the subdivision approach to identify areas.

The detection algorithm could be improved by reducing the number of nodes which are required to visit. The current algorithm has a potential worse case performance of $O(n-1)$ (where n is number of nodes), as every node has to be visited, excluding itself. This could be reduced by only performing

the polygon detection on points which have more than two edges, as nodes with only two edges can be considered a straight path. The time impact would be lessened, as the current plot detection algorithm is computationally expensive.

The data structure for the currently stored polygons could use optimisation. It currently uses the `list` collection, which requires iteration through every element to see if it exists (resulting in worse case $O(n)$ performance). This could be solved by using a `dict` which instead uses a hash function, that gives near linear ($O(1)$) performance during look ups.

In conclusion, the plot detection algorithm achieves its goal of finding the cyclic shortest paths also called polygons. The implementation would benefit from some optimisations to speed it up and some future improvements to benefit plot subdivision.

5.6 Plot Subdivision

Plot subdivision was not implemented due to time constraints. However, Keir's (2020) City Map Generator approach to generating plots would be used. This would build upon the current plot generation code to make new polygons.

Unlike the City Map Generator, the plot subdivision would be carried out differently, to bring it closer to how real world plots are designed in the UK (see Figure 16). This would be done by splitting the polygon at its shortest side, then evenly subdividing the following polygons. A different approach would have to be carried out on non simple polygons, such as ones found in figure 15b, where polygons which are rounded on their edge(s) would result in uneven plots, which is unlike the real world example found in figure **??** which appears to use a radial pattern. In these cases, a radial subdivision pattern may be used.

As this part of the solution is procedurally generated, more user control is required than over the network. The subdivision process would benefit from configurable parameters that would allow the user to generate their desired plot. Some configuration in terms of maximum and minimum plot size, where the plots shouldn't be generated would allow the user to get the desired effect.

Given that this subdivision process was implemented, while able to generate plots that would be seen in places such as the UK, it would need further work for different places with less robust building regulations. This is as

in other places of the world, buildings can be built outside of plots or even less robustly so. Moreover, as mentioned earlier some plots would have to be discounted for generation of buildings. This could be informed by the implementation of the user's parameters.

The plot identification could take place by creating a classification ANN, which would determine whether a plot is the correct size and shape to have a building on it. This could be approached by using OSM and classifying areas with street addresses as such. Though, this approach would not be universal as not all of the world's addresses are indexed nor consistent(OpenStreetMap Wiki Contributors, n.d.). This could be resolved by allowing the user to manually designate areas as ones they wish for plots to be generated on.

5.7 Evaluation Methods

The implemented evaluation methods attempt to capture the present data in a graph qualitatively. Overall, the road generation statistics provide an accurate picture of the key information for a graph. While usable statistics were only retrieved from the ground truth across most figures, there was enough difference to discern between Dundee and New York, for example through road length, junction connectivity, and transport convenience.

The plot evaluation was largely qualitative, which while not a problem on its own, could have used some measurements of the generated plot's area, aspect ratio, and number. While this data could not be as easily compared to the ground truth due to the plot data not being as freely available online, it would have provided an insight to the user on their plot generation configuration.

In general, more testing would be welcome, especially if the road generation aspect is more resolved to be able to compare the output to the NTG approach. While some modification to the graph density calculations would have to take place to ensure that it is directly comparable to the finding listed by Chu et al. (2019), generally the current ones represent a comprehensive picture of the generated graphs.

5.8 Project Effectiveness

This prototype, while unable to generate roads has provided a basis on which a fully featured working model can be built upon. This is important, as there are no implementations of NTG available online. Suggestions have been

provided on how to improve the model for future users. The implemented plot detection capabilities build upon existing procedural work by adding a unique way of finding plots in real world scenarios, that could prove useful for other researchers. Evaluation methods have been devised for both, which is helpful in visualising and describing the generated items.

6 Conclusion

This project provided a basis for approaching similar problems related to city layout generation in the future by simplifying future research into using techniques like Neural Turtle Graphics, as of the date of writing, no source code is publicly available. While the network was unsuccessful in capturing road networks' structure, a number of suggestions to improve the generation have been provided. This includes: replacing the one-hot encoding with a embedding layer to save memory, changing training data order to ensure that the network is not overfitting, and changing the activation function to improve performance and accuracy.

The plot generation was partly implemented due to time constraints. The implemented plot detection algorithm is better than other approach described as it works on multi-sided polygons. A solution and an approach were proposed to implementing plot subdivision, which involved building upon the work of Keir (2020), involving adding more complex polygon slicing logic to achieve results closer to real world plots. Furthermore, more research and evaluation would have to be performed into ensuring that the suggested approach works across different cities, as the current tests were performed on only two cities across two countries.

In relation to the research question, the effectiveness of the approach is mixed. The road generator requires more research and modification, and further investigation into methods of subdividing plots may be necessary. Consequently, the hypothesis has been disproven as the network did not generate roads, and more work is required to implement realistic plots.

7 References

Brownlee, J. (2017) How to develop a Seq2Seq model for neural machine translation in keras, Machinelearningmastery.com. Available at: https://machinelearningmastery.com/define-encoder-decoder-sequence-sequence-model-neural-machine-translation-keras/.

Cho, K. et al. (2014) 'Learning phrase representations using RNN encoder-decoder for statistical machine translation', arXiv [cs.CL]. Available at: http://arxiv.org/abs/1406.1078.

Chu, H. et al. (2019) 'Neural Turtle Graphics for modeling city road layouts', arXiv [cs.CV]. Available at: http://arxiv.org/abs/1910.02055.

Curran, K., Crumlish, J. and Fisher, G. (2012) 'OpenStreetMap', International journal of interactive communication systems and technologies, 2(1), pp. 69–78.

Dey, R. and Salem, F. M. (2017) 'Gate-variants of Gated Recurrent Unit (GRU) neural networks', arXiv [cs.NE]. Available at: http://arxiv.org/abs/1701.05923.

Dijkstra, E. W. (1959) 'A note on two problems in connexion with graphs', Numerische mathematik, 1(1), pp. 269–271.

Hagberg, A. A., Schult, D. A. and Swart, P. J. (2008) 'Exploring network structure, dynamics, and function using NetworkX"', in Vaught, T. and Millman, J. (eds). Pasadena, CA USA, pp. 11–15,.

Hart, P., Nilsson, N. and Raphael, B. (1968) 'A formal basis for the heuristic determination of minimum cost paths', IEEE transactions on systems science and cybernetics, 4(2), pp. 100–107.

van Hattem, R. (2020) ProgressBar2, Pypi.org. Available at: https://pypi.org/project/progressbar2/.

Keir (2020) City Map Generator. Available at: https://github.com/ProbableTrain/MapGenerator https://maps.probabletrain.com https://probabletrain.itch.io/city-generator.

Keras Team (no date) Model training APIs, Keras.io. Available at: https://keras.io/api/models/model_training_apis/.

Kingma, D. P. and Ba, J. (2014) 'Adam: A method for stochastic optimization', arXiv [cs.LG]. Available at: http://arxiv.org/abs/1412.6980.

Nwankpa, C. et al. (2018) 'Activation functions: Comparison of trends in practice and research for deep learning', arXiv [cs.LG]. Availableat:http://arxiv.org/abs/1811.03378.

OpenStreetMap Wiki Contributors (no date) Addresses, Openstreetmap.org. Available at: https://wiki.openstreetmap.org/wiki/Addresses.

OrdnanceSurvey (no date) 'Edinburgh Planning Permission Applications (Map View)'. Houses: Grid Ref NT 25908 71937, Tenaments: Grid Ref NT 25469 72234. Scale: N/A. Available at: https://citydev-portal.edinburgh.gov.uk/idoxpa-web/spatialDisplay.do?action=display&searchType=Application.

Parish, Y. I. H. and Müller, P. (2001) 'Procedural modeling of cities', in Proceedings of the 28th annual conference on Computer graphics and interactive techniques - SIGGRAPH '01. New York, New York, USA: ACM Press.

Racicot, A. (2012) bbox finder, Bboxfinder.com. Available at: http://bboxfinder.com/.

You, J. et al. (2018) 'GraphRNN: Generating realistic graphs with deep auto-regressive models', arXiv [cs.LG]. Available at: http://arxiv.org/abs/1802.08773.

8 Appendices

8.1 A: Quad Graph Polygons

The following output was generated after finding polygons. Note that each of the polygons only contain four nodes.

```
[[(0, 0), (0, 1), (1, 1), (1, 0)],
[(0, 1), (0, 2), (1, 2), (1, 1)],
[(0, 2), (0, 3), (1, 3), (1, 2)],
[(1, 0), (1, 1), (2, 1), (2, 0)],
[(1, 1), (1, 2), (2, 2), (2, 1)],
[(1, 2), (1, 3), (2, 3), (2, 2)],
[(2, 0), (2, 1), (3, 1), (3, 0)],
[(2, 1), (2, 2), (3, 2), (3, 1)],
[(2, 2), (2, 3), (3, 3), (3, 2)]]
```

8.2 B: Bounding Box Coordinates Used

Formatted as EPSG:4326, in Latitude/Longitude coordinate format.

- Around Times Square, New York, US: 40.752247,-73.990324,40.759269,-73.982642
- Dundee City Center around Central Mosque, UK: 56.459124,-2.985106,56.464221,-2.977424
- Around Cumin Place, Edinburgh, UK: 55.932760,-3.188041,55.936739,-3.179866
- Around Marchmont Road, Edinburgh, UK: 55.936132,-3.198298,55.939653,-3.192515

8.3 C: Python Packages Used

- OSMPythonTools
- networkx
- Tensorflow
- keras
- numpy
- ProgressBar