Creating Bus Routes with Convolutional Neural Networks

Paschalis Skouzos

Master of Science
Artificial Intelligence
School of Informatics
University of Edinburgh
2019

Abstract

Due to the increased demand for bus service in recent years [28], bus agencies are forced to constantly improve their network through costly and time-consuming procedures. In our research, we propose two methodologies which, given an image of the available road network, automatically generate bus routes. The first method implements the Generative Adversarial Network (GAN) architecture of *pix2pix* [27]. Its target is to generate plausible images of bus routes given a road network image. The second method aims to determine which pixels in the road network image belong to a bus route using a convolutional auto-encoder called U-Net [37]. Our results show that the GAN is unable to create plausible routes due to the imbalance of foreground/background in the target images. The average travel time of random trips using the routes generated by U-Net is only about 1 minute longer (3% higher) of the average travel time using manually designed routes. Its test set precision is 19% and recall is 44%. In the process of creating our dataset of bus route images, we also produce the first clean bus route dataset in the General Transit Feed Specification (GTFS) format [6].

Acknowledgements

I would like to express my gratitude to my supervisor Dr Rik Sarkar for the instructive feedback and comments throughout the process but especially when I was stuck and thought that nothing was going to work. In addition, I am indebted to my parents, both literally and metaphorically, for the support they gave me throughout this year.

Table of Contents

1 Introduction **1**

2 Background **4**

2.1 General Transit Feed Specification 4

2.2 Convolutional Neural Networks 6

2.3 Generative Adversarial Networks 8

3 Dataset **11**

3.1 GTFS Feeds 11

3.2 Road Network and Bus Routes Images 13

4 Methodology **17**

4.1 Generative Adversarial Network 17

4.1.1 Generator 18

4.1.2 Discriminator 20

4.1.3 Training 20

4.1.4 Validation 21

4.2 U-Net 25

4.2.1 Architecture 25

4.2.2 Training 26

4.2.3 Validation 27

5 Experiments & Discussion **28**

5.1 Generative Adversarial Network 28

5.2 U-Net 30

6 Conclusion **37**

6.1 Summary 37

6.2 Future Work 38

Bibliography **39**

A Pixel to Coordinates Code **44**

Chapter 1

Introduction

In recent years, due to the increased congestion of road networks, there has been a rise in the demand of effective bus services [46]. For example in London, from 2000 to 2015, although the population grew by 19.9%, bus demand increased by 60.3% while car trips decreased by 13.9% [28]. Therefore, bus agencies are constantly required to improve their network's performance by adding new routes or redesigning existing ones. Transit network design is a long and costly process which involves analysing and predicting current and future demand. It requires being in constant liaison with other agencies, public and private institutions such as schools and hospitals, companies, the public, and any other source of demand [28]. In our research, we propose two approaches which automatically generate bus routes based on the available road network.

Due to the complexity of planning and operating a bus transport system, its design is split into four stages [15]:

1. **Strategic planning**, also known as transit network design, includes the selection of routes and stops for the network. The problem is usually defined on a graph $G = (N, E)$, where nodes N are intersections of the road network and edges E are road segments connecting the nodes. The input of the problem is an origin-destination matrix which provides the daily demand between nodes of the graph. An important advantage in our approach is that it does not require this matrix, enabling the selection of routes and stops with no prior demand knowledge. Another output of this stage is selecting the initial bus frequency in order to validate the selected routes and stops.

2. **Tactical planning** tackles the optimization of bus frequencies and timetabling.

Bus frequency is based on bus supply and demand, which may vary based on the time, the day, special events etc. Timetabling is the procedure of converting bus frequencies into the actual times that buses arrive at bus stops.

3. **Operational planning** is concerned with pairing operational resources, e.g. buses with routes or drivers with buses, taking into account constrains such as the number of available buses, bus capacity, and work regulations. Additional outputs may include bus maintenance schedule or parking slots assignment.

4. The final stage is called **real-time control** and it includes the actions taken by an agency during a disruption in the transit network service. Several factors such as adverse weather conditions, events, unavailable drivers, may obstruct normal service execution and the agency must be in a position to minimize the effects of such factors on the passengers.

Each stage requires as input the output of the previous stage and in our research we address strategic planning, the first stage of the pipeline. We test two methods. The first one utilizes GANs and based on an image of the road network it generates images of plausible bus routes. The second one performs image segmentation on the road network images and classifies road segments as bus routes or not. Although GANs fail to produce plausible bus routes, the image segmentation approach is able to generate bus routes which serve random trips with an average travel time of about 1 minute longer (3% higher) than the real routes of our test set. In the test set, the precision of our model is 19% and its recall is 44%. Routes generated from the image segmentation method are available at figure 1.1.

In a recent review of bus transport systems design [25], the authors identify two approaches for strategic planning. The first one is treating it as a discrete optimization problem using as input the origin-destination matrix of demand. The majority of the solutions for this approach utilize genetic algorithms. The second one substitutes the origin-destination matrix with a continuous demand function over the geographical service region. Our methods' novelty, compared to any available research on the topic, lies in three aspects:

1. It does not require any passenger demand knowledge as input.

2. It utilizes the knowledge of the field experts who designed the hundreds of routes we use to train our models. This is the first research to build upon previous route designs.

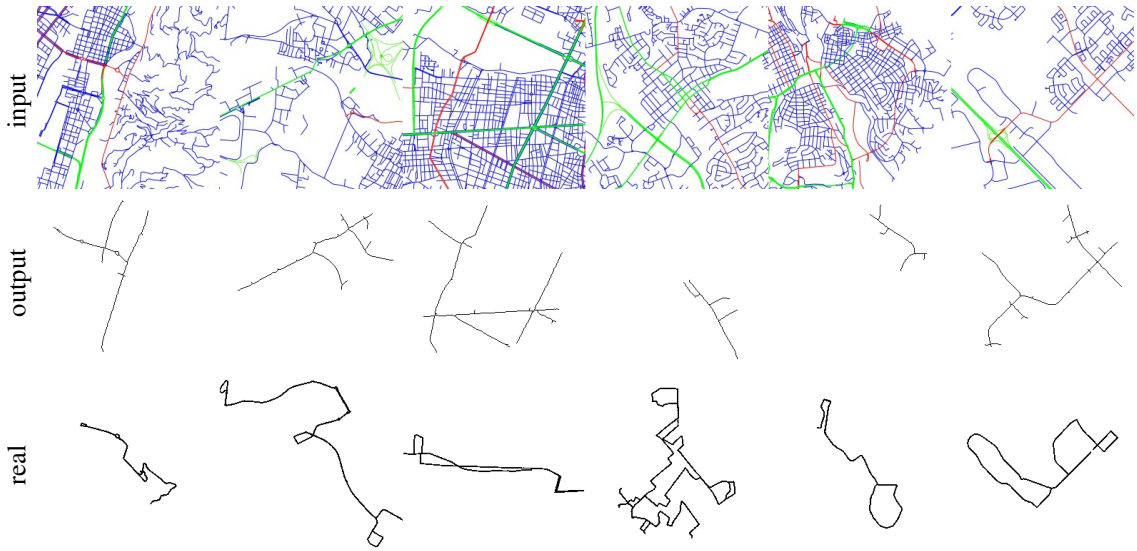


Figure 1.1: Sample routes generated using our method based on U-Net [37]

3. It transfers the problem of transit network design from the graph domain to the image domain.

In addition, we show that the GAN architecture of [27] is unable to produce images with a heavily imbalanced foreground/background. Another major by-product of our research is the first bus route dataset in the GTFS format which has no violations of the standard.

In chapter 2, we provide the required background regarding GTFS, Convolutional Neural Networks (CNNs), and GANs. In chapter 3, we describe how we obtain and clean GTFS feeds, and how we convert them to images of road networks and bus routes. In chapter 4, we present our two methodologies. In chapter 5 we detail our experiments, their results, and their analysis. We conclude with chapter 6, where we summarize our work and give directions for future research.

Chapter 2

Background

In this chapter, we provide the required background knowledge for GTFS, CNNs, and GANs.

2.1 General Transit Feed Specification

In 2006 Google developed GTFS as a way to standardize the publication of public transit data [38]. GTFS is a specification which defines a format in which public agencies can publicize their schedule. In essence, it defines a database schema which must be used in order to abide by the standard. The new standard spawned useful libraries which enable reading, manipulating, and analysing transit feeds from different agencies. GTFS feeds are the backbone of our approach since they are used in order to create our route images dataset and determine the distance between our stops. The agencies publicize their feed as one compressed file which includes several text files with comma separated values (CSV). Each compressed file can be thought of as a database with each CSV file being a table and each row in the files is a table entry. The standard defines some tables/columns as required and there are also optional tables/columns which an agency may provide for extra information [6]. Figure 2.1 shows an entity relationship diagram for GTFS, displaying only the tables and columns we use in our research.

- The two relevant columns of the **routes.txt** file are `route_id`, which acts as a primary key, and `route_type`, which defines the transport medium for each route, e.g. bus, train, ferry. Each route consists of at least one trip. It is therefore referenced from at least one row in the trips table. In the case of buses, each route has usually two trips, one for the inbound and one for the outbound direction.

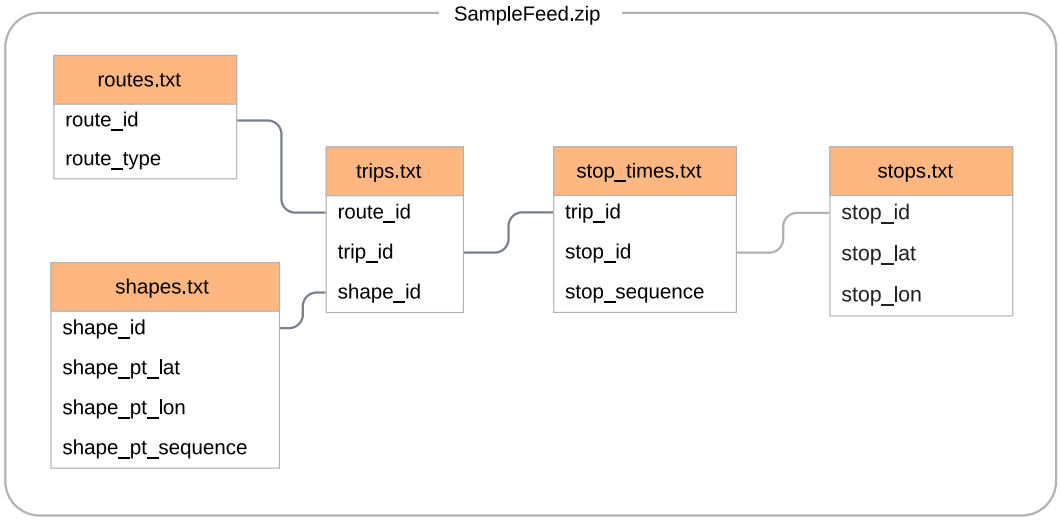


Figure 2.1: GTFS entity relationship diagram showing only the tables/columns relevant to our research

- In the **trips.txt** file, `trip_id` is the primary key and each trip belongs to a route with id `route_id`. In addition, we are interested in the `shape_id` which references one or more entries in the shapes file.

- The **shapes.txt** file is the one we use in order create the route images dataset. It defines the lines which are followed by trips in the feed. Each row contains a `shape_id`, `latitude-longitude` coordinates, and a `shape_pt_sequence` which defines the order of the lat-lon coordinates in the shape. The `shape_id` is shared among rows of the same shape and the ordered lat-lon coordinates create the lines of each shape. For example, suppose we have only one `shape_id` which consists of three rows as shown in figure 2.2. Those rows define two line segments on a map. The first one is the segment between the first and the second pair of coordinates (40.519456, -8.414631)-(40.575134, -8.448088), and the second segment is the one between the second and the third pair (40.575134, -8.448088)-(40.574958, -8.448066). A trip with `shape_id` of 1 would follow those segments, in that order.

- The **stop_times.txt** file contains the stop sequence which is performed by a trip. The `trip_id` column references a trip, the `stop_id` column links to a stop at the stops.txt file, and the `stop_sequence` defines the order in which the stops

shape_id	shape_pt_lat	shape_pt_lon	shape_pt_sequence
1	40.519456	-8.414631	1
1	40.575134	-8.448088	2
1	40.574958	-8.448066	3

Figure 2.2: Shapes.txt file example for one shape

are visited during a trip. This file also contains stop arrival and departure times, hence the name, but we did not include them in the diagram since we are not utilizing them.

- Finally, the **stops.txt** file includes information about the stops, such as latitude and longitude, with `stop_id` being the primary key. This file, in combination with stop_times.txt, is used to determine the distance of our stops in the generated routes.

2.2 Convolutional Neural Networks

CNNs are a special type of neural network which assumes images as input. To demonstrate the need for such a type of network assume we have a 256×256 RGB image of a digit and we attempt to classify it using a fully connected network. Since an RGB image has three channels, one cell in the first layer has $256 \times 256 \times 3 = 196608$ inputs and therefore 196609 weights, including the bias. As a result, the first layer has $196609 \times (\text{number of cells})$ weights which quickly becomes intractable even with one layer. To solve this issue we introduce convolutional layers which include several kernels/filters of small dimensions which we slide across the image. Each of them is able to detect a certain type of feature throughout the image [29]. For example, in figure 2.3 we have one convolutional layer with one kernel W of dimensions 3×3, without bias, operating on two-dimensional input X and outputting feature map H. The kernel's weights have been set so that it detects horizontal edges across input X [42]. The convolutional operation functions by sliding the kernel over the input and, at each step, performing element wise multiplication and then summation, $H_{k,l} = \sum_{i=k}^{k+2} \sum_{j=l}^{k+2} W_{k-i,k-j} X_{i,j}$. For instance, the highlighted 3 in H is produced by ap-

plying the kernel in the highlighted area of X. If our input was three-dimensional, e.g. RGB image, then the kernel would also have a depth of three, with different weights in each depth. The same operation as in our example would be performed in each pair of image channel and kernel depth. The final result of the three dimensions would then be summed in order to calculate the output, $H_{k,l} = \sum_{d=0}^{2} \sum_{i=k}^{k+2} \sum_{j=l}^{k+2} W_{k-i,k-j,d} X_{i,j,d}$. While kernels in the first layers can detect primitive features such as lines or circles, kernels in deeper layers are able to combine those primitive features into more abstract features. Because of the available combinations of features from previous layers, the number of detectable features increases in deeper layers and it is common practice to also increase the number of kernels as we go deeper [29]. The dimensions of the feature map H are calculated as $(H_w, H_h) = (X_w - W_w + 1, X_h - W_h + 1)$.

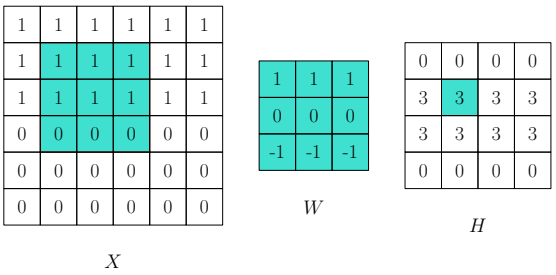


Figure 2.3: Horizontal edge detection kernel [42]

Three important concepts of CNNs we utilize are padding, stride, and max pooling. As shown in the equation which calculates feature map dimensions, even with a small kernel of size (2, 2) we are reducing the height and width of our input by one pixel. In order to avoid that, we can use **padding** which is the process of adding pixels of value 0 in the perimeter of our input in order to maintain the same dimensions in the output [29]. Using padding P, the feature map dimensions are $(H_w, H_h) = (X_w + P_w - W_w + 1, X_h + P_h - W_h + 1)$. Therefore, having $(P_w, P_h) = (W_w - 1, W_h - 1)$ results in equal input and output dimensions.

Stride is simply the number of pixels we slide the kernel over the input [29]. In our example of figure 2.3 we use a stride of $(w, h) = (1, 1)$, which means that in each step we move the kernel one pixel rightwards or downwards. If we want to downsample, we can increase the stride, e.g. to (2, 2), as demonstrated in figure 2.4. Since downsampling results in a reduced number of parameters, it can increase the training speed and also prevent overfitting. Using stride S and no padding, the feature map dimensions are calculated as $(H_w, H_h) = (\left\lfloor \frac{X_w - W_w + S_w}{S_w} \right\rfloor, \left\lfloor \frac{X_h - W_h + S_h}{S_h} \right\rfloor)$.

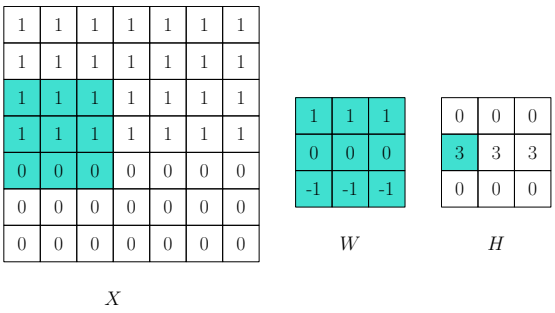


Figure 2.4: Horizontal edge detection kernel with stride (2, 2) [42]

Max pooling is another technique we can use to downsample after we have created feature map H [41]. The technique resembles convolutions in the sense that we still have a sliding window over our input but in that case the input is the feature map. We may also combine the operation with padding and stride. However, in contrast with the kernels used in the convolution step, this sliding window has no weights and at each step it simply picks the maximum element in its context. Figure 2.5 demonstrates max pooling with window size of (2, 2), stride of (1, 1), and no padding.

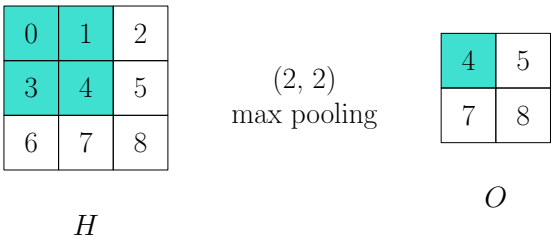


Figure 2.5: Max pooling of size (2, 2), stride of (1, 1), and no padding

2.3 Generative Adversarial Networks

The goal of our first approach is given an image of a road network to produce plausible, non-deterministic, bus routes. By striving for non-determinism, we can combine the generated routes and produce a complete bus network for the city. GANs where proposed in 2014 as a framework which aims to produce outputs similar to the training data [20].

The framework consists of two models which, as described by the authors[20], resemble counterfeiters and policemen. The generative model G, i.e. counterfeiters,

tries to produce outputs as close as possible to the actual data distribution while the discriminator model D, i.e. policemen, tries to distinguish between real data and data produced by the generative model. The input of the generative model is a random noise vector $\boldsymbol{z}$, and we want to learn the generator's distribution p_g over our targets $\boldsymbol{x}$. Therefore, the mapping from random noise to $\boldsymbol{x}$ is defined as $G(\boldsymbol{z};w_g)$ with w_g being the weights of our generative model. The discriminator model is defined as $D(\boldsymbol{x},w_d)$, w_d being its weights, and its output is the probability that input $\boldsymbol{x}$ came from actual data rather than $G(\boldsymbol{z};w_g)$. D is trained to maximize the probability of assigning the correct label to real and fake data while G is trained to minimize $\log(1-D(G(\boldsymbol{z})))$. The two models are players in a minimax game with the value function of equation 2.1. The training process of both models is outlined in algorithm 1 [20].

$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[1-\log D(G(\boldsymbol{z}))] \tag{2.1}$$

Algorithm 1: GAN training procedure using minibatch gradient descent

for *number of training epochs* **do**
  for *number of batches per epoch* **do**
    Select n noise samples $\boldsymbol{z}^1,\cdots,\boldsymbol{z}^n$
    Select n real samples $\boldsymbol{x}^1,\cdots,\boldsymbol{x}^n$
    Update discriminator's weights by ascending its gradient

$$\nabla_{w_d}\frac{1}{n}\sum_{i=1}^{n}[\log D(\boldsymbol{x}^i)+\log(1-D(G(\boldsymbol{z}^i)))]$$

    Update generator's weights by descending its gradient

$$\nabla_{w_g}\frac{1}{n}\sum_{i=1}^{n}\log(1-D(G(\boldsymbol{z}^i)))$$

  end
end

Since our input is the road network of a city, apart from random noise, the input of our generator must include the road network itself. This can be achieved by using a version of GANs called conditional GANs [35]. By conditioning both models with our road network $\boldsymbol{y}$, equation 2.1 can be rewritten as equation 2.2 with a similar training procedure.

$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x}|\boldsymbol{y})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[1-\log D(G(\boldsymbol{z}|\boldsymbol{y})|\boldsymbol{y})] \tag{2.2}$$

In [35], the authors created a conditional GAN in order to generate specific digits using the MNIST dataset [13]. The input for the generator was a random noise vector of size 100 as well as a specific digit in the form of one-hot encoded vector. The output was a two-dimensional greyscale image of the requested digit. The discriminator, apart from generated and real data, also receives the digit as a one-hot encoded vector, depicted in figure 2.6. Although using random noise in order to avoid deterministic outputs in conditional GANs worked in some cases [35, 47], in deeper architectures the generator is able to ignore the noise [34, 27]. In the latter cases, one approach to introduce noise in the model is to drop random neurons in each training iteration using dropout layers [27].

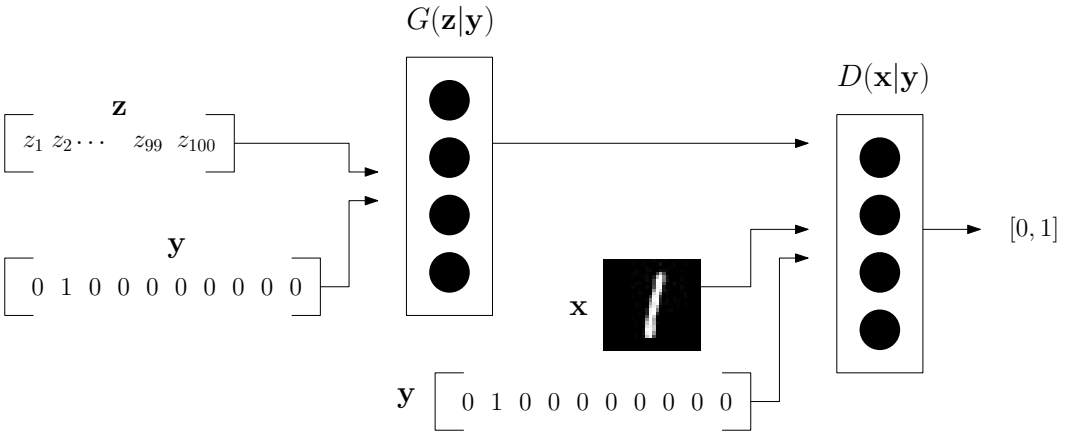


Figure 2.6: Conditional GAN architecture for MNIST digits

Chapter 3

Dataset

In this chapter we explain how we obtain and clean the GTFS feeds, creating the first GTFS dataset with no violations of the standard. In addition, we describe how we create images of road networks and bus routes.

3.1 GTFS Feeds

The first step in the GTFS dataset creation process is to download feeds from various agencies. Although each agency publicizes its feed in its own website, `https://transitfeeds.com/` is a website which collects and archives feeds from all over the world. It also provides an API for searching and downloading [4] feeds. Using their API, we search and download all available GTFS feeds with a total number of 842 and size 4,61GB.

The second step involves removing all non-bus routes. For the removal of irrelevant routes we use the library `partridge` [2] which allows filtering GTFS feeds by some value. We load all feeds with the filter `{'routes.txt': {'route_type': 3}}` which reads only feeds with `route_type` equal to 3, i.e. the numerical value for bus route types in the GTFS standard. Another helpful feature of the library is that it removes zombie entities. Namely, the ones relevant to the project are: routes with no trips, stop times with invalid trip id, and shapes with no trips. In addition, it removes stops with no stop times, however, care should be taken to manually disable this feature if one aims to create a valid GTFS dataset for the following reason.

The stops table has two columns we didn't mention, titled `location_type` and `parent_station`. An entity in the stops table may be one of five possible *types* including the type of 'parent station'. 'Parent station' is assigned to entities such as

large central stations with multiple platforms located in them. The platforms in those central stations are separate entities in the stops table and have their `parent_station` column filled with the stop_id of the parent/central station. If we do not disable the automatic removal of stops with no stop times then all parent stations will be removed since none of them is listed as an actual stop in the stop times table, only their 'children' stations are. As a result, our feed will be filled with 'children' platforms whose `parent_station` column points to invalid stop ids. The reason we didn't mention the above two columns in background section 2.1 is that they are irrelevant to our method. We are only interested in stops of type 'platform' and an invalid parent id would not affect us. However, a novel by-product of our project is the first valid bus GTFS feed dataset so for completeness we mention this bug here.

Since many agencies publicize their feeds without validating them, our dataset is filled with violations of the GTFS standard, e.g. undefined ids or invalid values. Therefore, the third step is cleaning the dataset. For this process we use the library `gtfstk` which provides a validation function that reports warnings and errors for the feeds [5]. In the first cleaning round we delete feeds who where left with empty tables because of the filtering procedure, e.g. feeds which included only train routes. In addition, we delete all columns which are not defined in the standard but were added by the agencies for special use, e.g. an agency may have included a column of average speed in their trips table. In the second round, we clean malformed entries, mainly in tables and columns we haven't mentioned, such as invalid agency language or invalid stop url. In the third round, we remove any entities with invalid foreign keys, e.g. trips whose `shape_id` column points to an invalid shape. In addition, we have to remove zombie entities again since the cleaning procedure may have reintroduced them in our dataset. For example when we remove trips with invalid `shape_id` we may create zombie routes with no trips. Lastly, we remove all optional tables apart from shapes.txt. The final output of the above cleaning rounds is a dataset with GTFS feeds, only for buses, which reports no errors from the `gtfstk` validation function.

On a final note regarding this dataset's creation, we notice that some feeds are identical because `https://transitfeeds.com/` contains duplicates. Therefore, in each zipped file/feed, we sort the text files in alphabetical order, then we concatenate the contents of the text files, and finally we use the MD5 hash algorithm in the concatenated string. Duplicate feeds with identical hash values are then removed. We are unable to hash the feed files directly since zip files contain metadata, such as creation date, which results in different hash values for identical content. The final dataset

contains 694 feeds, has a size of 2,91GB, and contains 72,580 routes.

3.2 Road Network and Bus Routes Images

Our next task is to create images of road networks and of the corresponding bus routes which are attached on those road networks. In general, map images are retrieved from map tile servers by specifying the longitude, the latitude, and the zoom level for the area we are interested in. Each map tile is usually 256×256 pixels, and if we need larger areas we have to combine multiple tiles. All the images must be of equal dimensions and of the same zoom level since the network has to learn in the same context for all routes. For instance, figure 3.1 displays three map images centered at longitude -3.187127 and latitude 55.944403 (Appleton Tower, Edinburgh) with different zoom levels. We can see that different roads are visible at different levels. Since we are interested in generating bus routes in cities, we have to pick a high zoom level, such as 15 or 16, in order to show even small roads.

(a) zoom $= 8$ (b) zoom $= 12$ (c) zoom $= 16$

Figure 3.1: Map images with dimensions 256×256, centered at Appleton Tower, Edinburgh, using the OpenStreetMaps-no labels server [8], at 3 different zoom levels

In the map images of figure 3.1 we used the OpenStreetMaps-no labels map tiles [8] which include noise such as map background, pedestrian roads, or points of interest. Our ideal map tiles would include only roads, and they would also distinguish between primary and non-primary roads. The road size distinction is useful since bus routes favour medium to large roads. Since such a tile set does not exist, we create our own using `https://www.maptiler.com/` [19]. Different colours can be set based on the type of the road in OpenStreetMaps. For the three largest types of roads, i.e. *motorway*, *trunk*, and *primary*, we choose green, (0, 255, 0) in RGB. For the next largest type,

secondary, we choose red (255, 0, 0), and for all remaining types we choose blue (0, 0, 255). Figure 3.2 shows a comparisson between the tiles used in figure 3.1 and our custom tiles, using the same center as before and zoom level of 15.

(a) OSM-no labels tiles (b) Custom tiles

Figure 3.2: Comparrison between OpenStreetMaps-no labels tiles [8] and our custom tile set at zoom level 15

Since we require a high zoom level of 15-16 in order to draw all types of roads, getting an image of dimensions 256×256 pixels will result in a very small area, as shown in sub-figure 3.1c. We will therefore be unable to use it for training since there are no routes which span only such a small geographical area. On the other hand, having large input images significantly increases the training parameters and reduces training speed. As a middle ground, we create images of dimensions 1024×1024 pixels at zoom level 15, and then resize them at 256×256. Having a simple tile set allows us to resize the images without losing information. Figure 3.3 demonstrates the 16 times larger geographical area we are able to obtain by creating 1024×1024 pixel images.

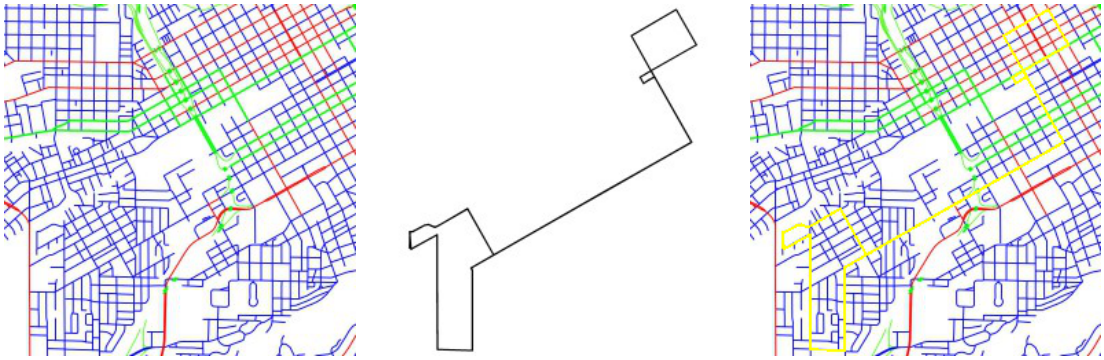
For the creation of map images at 1024×1024 pixels, zoom level 15, and centred at specific coordinates, we use the library `staticmap` [7]. We are able to use only the routes which can fit in the geographical area returned using the above number of pixels and zoom level. For example, bus routes between major cities need either more pixels or lower zoom level. In summary, during the image creation process, we iterate over all the routes in the dataset. Then, for each route, we calculate the centre of all its shapes. Afterwards, we create a map image of 1024×1024 pixels, at zoom level 15, centred at the previously calculated centre of the shapes. If the geographical boundaries of the image include all shapes, then we draw the route and the road network image. The code for turning pixel locations to GPS coordinates is available at appendix A. We

(a) 256×256 (b) 1024×1024 resized to 256×256

Figure 3.3: Geographical area coverage comparison by generating 256×256 images versus 1024×1024 at zoom level 15

draw routes and road networks in separate images since the former is our output and the latter our input. The process is understood easier through algorithm 2.

Due to the earth's shape, the boundaries have to be calculated for every location since map images centred closer to the equator encompass a larger area. For instance, using dimensions 1024×1024 and zoom level 15, the north bound at the equator is ≈ 0.02197 degrees from the image centre while in Edinburgh it is ≈ 0.01230. Figure 3.4 displays a sample of the network's input 3.4a, the target output/bus route 3.4b, and to demonstrate that the bus route is drawn at the correct location we place the bus route, using yellow for visibility, on top of the road network at 3.4c. Our image dataset has 3,613 pairs of road networks and bus routes.



(a) Road network (b) Bus route (c) Route merged with road

Figure 3.4: Sample input/road network (a), output/bus route (b), bus route on top of road network (c)

Algorithm 2: Road network and bus route image creation

```
foreach feed in the dataset do
    if shapes.txt file exists then
        foreach route in the feed do
            Get all shape ids from trips with route id equal to route
            if size of shape ids >1 then
                Calculate center lon/lat from all shapes
                Create image centered at the above center with size
                  1024 × 1024 and zoom 15
                Calculate coordinates of top-left and bottom right corner of
                  image
                if all shape coordinates are inside boundary coordinates then
                    Draw route
                    Draw image
                end
            end
        end
    end
end
```

Chapter 4

Methodology

In this chapter we explain the two methods we use in order to generate bus routes as well as the methodology we develop in order to externally validate the performance of our routes. The models we use are implemented as defined in their original papers [27, 37]. In this chapter and chapter 5 we specifically mention any different architectural or hyperparameter choices we make.

4.1 Generative Adversarial Network

The first method aims to create plausible bus routes using GANs. The architecture we use comes from [27] who found a configuration able to generalize across multiple domains. Figure 4.1 demonstrates the success of their model in several tasks.

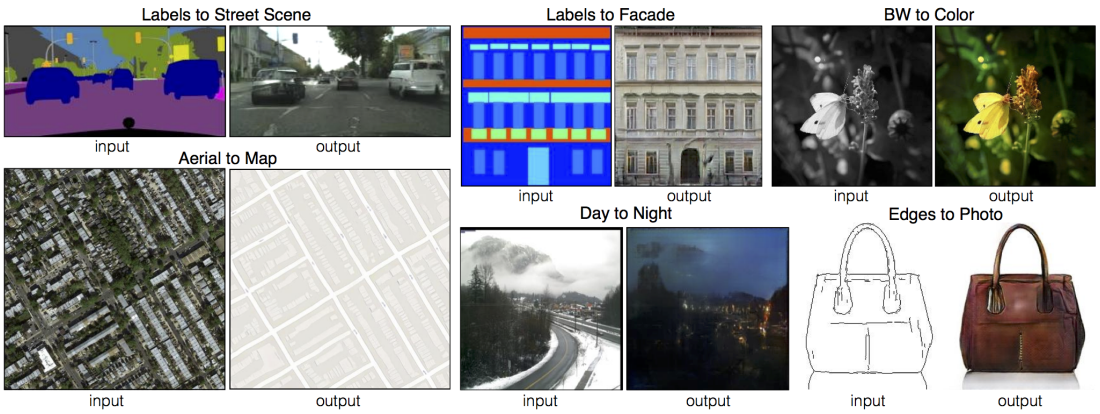


Figure 4.1: Examples from [27]

4.1.1 Generator

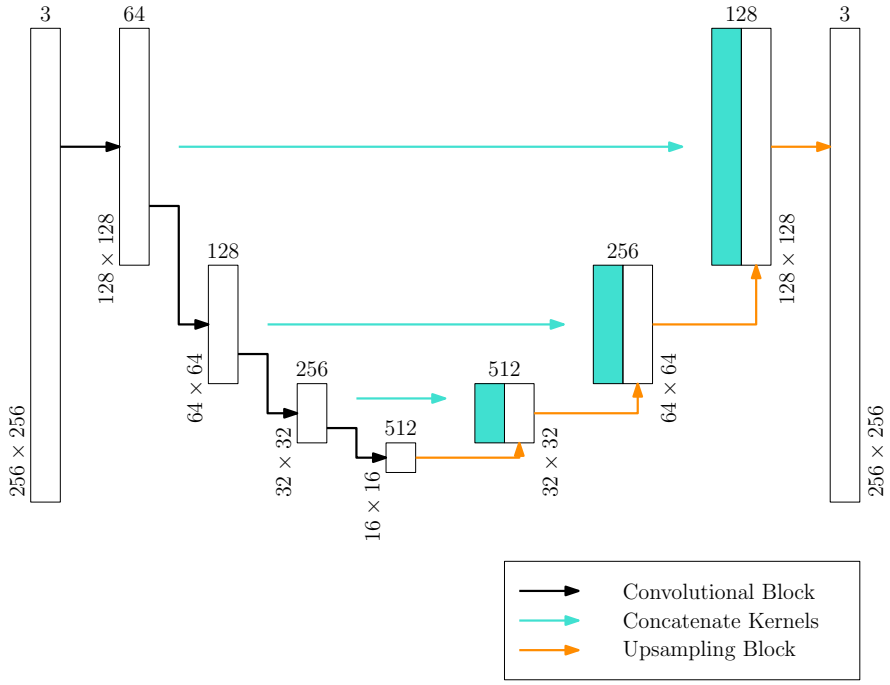
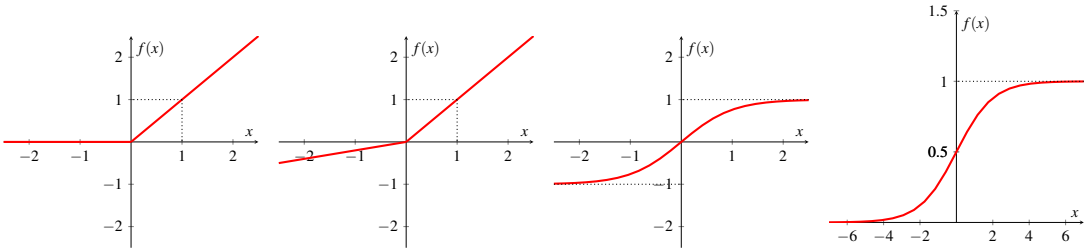


Figure 4.2: Generator architecture

The architecture of the generator is called U-Net which is an auto-encoder with skip connections [37]. Auto-encoders consist of two parts, the encoder and the decoder. While the encoder's job is to find a low dimensional representation of the input image, the decoder has to revert the process and rescale the image back to the original size [24]. Skip connections is the concatenation of convolutional layers in the contracting path of the encoder with layers in the expanding path of the decoder. They are used in order to pass high resolution features into the restructuring process. For instance, since we want the bus routes to be drawn exactly on top of some road segments, the road structure which is a high resolution feature, encoded in the initial convolutional layers, has to be passed directly to the decoder before getting lost in low dimensionality. The architecture we are implementing [27] follows the instructions of [36] who provide some guidelines for successfully training deep convolutional GANs.

The first advice is to downsample using strides instead of max-pooling. The reasoning is that max-pooling is a deterministic technique of downsampling while strides enable the model to learn its own spatial downsampling. The second advice is to use the Rectified Linear Unit (ReLU) activation function, $f(x) = \max(0,x)$ (Fig. 4.3a),

in all layers of the decoder except from the output which uses the hyperbolic tangent function (tanh), $f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ (Fig. 4.3c). The bounded activation function of tanh in the output layer speeds up the process by covering the colour range of our target distribution faster [36]. In addition, they propose the use of Leaky ReLU in the discriminator, $f(x) = \alpha x$ if $x < 0$ and $f(x) = x$ if $x \geq 0$ (Fig. 4.3b). The third suggestion is to use Batch Normalization [26] which scales the batch inputs of each neuron to zero mean and unit variance. This prevents the model from collapsing due to bad weight initialization and also facilitates the flow of gradients in deeper architectures [36].



(a) ReLU (b) Leaky ReLU with $\alpha = 0.2$ (c) Hyperbolic tangent (d) Sigmoid

Figure 4.3: Behaviour of our activation functions

The architecture of the generator with four convolutional blocks is displayed in figure 4.2. The numbers on top of the rectangular blocks indicate the number of channels/filters in each representation and the pixel dimensions are displayed in the bottom corner of each block. Keep in mind that the input is a road network image with three channels, RGB, and the target output is again an RGB image of a bus route. Each black arrow/convolutional block in the encoder consists of a convolutional layer with kernel=(4, 4) and stride=(2, 2), followed by a batch normalization layer, and a Leaky ReLU activation layer with $\alpha = 0.2$. Batch normalization is not used in the first convolutional block as was the case in [27, 36].

In each orange arrow/upsampling block we have an upsampling layer which doubles the dimensions, a convolutional layer with kernel=(4, 4), stride=(2, 2) and padding, a batch normalization layer, a ReLU activation layer, and then we concatenate with the corresponding block in the downsampling path. In the final upsampling block there is no batch normalization or concatenation and the activation is tanh. As we mentioned in section 2.3, we have to introduce noise in the generating process so following [27] we include a dropout layer in the first two upsampling blocks right before the ReLU activation layer. We set the dropout value to 0.5 which means that half of the kernels

are dropped.

4.1.2 Discriminator

The goal of the discriminator is given an RGB image to identify if it is a bus route or not. Its architecture is similar to the generator's encoder [27, 36]. All convolutional blocks in figure 4.4 include a convolutional layer with kernel=(4, 4), stride=(2, 2), a batch normalization layer, and a Leaky ReLU layer with $\alpha = 0.2$. Again, no batch normalization is used in the first block. After the last convolutional block we flatten the units in a fully connected layer and then add the final layer with one neuron and sigmoid activation, $f(x) = \frac{1}{1+e^{-x}}$ (Fig. 4.3d). An output close to 1 indicates that with high probability the input image is a bus route and an output close to 0 indicates the opposite.

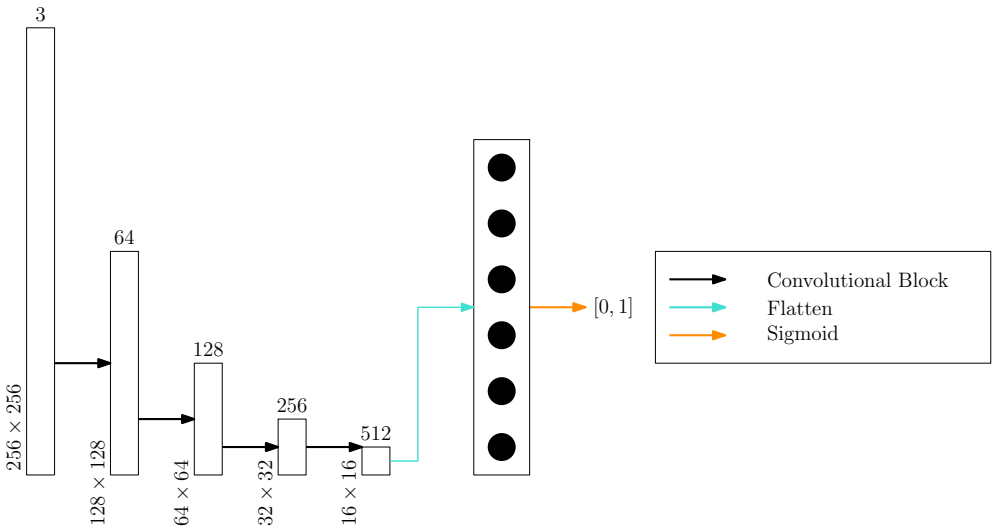


Figure 4.4: Discriminator architecture

4.1.3 Training

The training procedure is similar to the process we explained in background section 2.3 where discriminator and generator take turns training. However, mixing the conditional GAN loss function (Eq. 4.1) with other loss functions such as L1 (Eq. 4.2) or L2 produces better results. L1 is preferred since it reduces image blurring [27]. Therefore,

our total loss function is equation 4.3 with λ being the weight for the L1 loss.

$$\mathcal{L}_{cGAN}(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x}|\boldsymbol{y})] + \mathbb{E}_{\boldsymbol{z}\sim p_z(\boldsymbol{z})}[1 - \log D(G(\boldsymbol{z}|\boldsymbol{y})|\boldsymbol{y})] \tag{4.1}$$

$$\mathcal{L}_{L1}(G) = \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\|\boldsymbol{x} - G(\boldsymbol{z}|\boldsymbol{y})\|_1] \tag{4.2}$$

$$\mathcal{L} = \mathcal{L}_{cGAN}(D,G) + \lambda\mathcal{L}_{L1}(G) \tag{4.3}$$

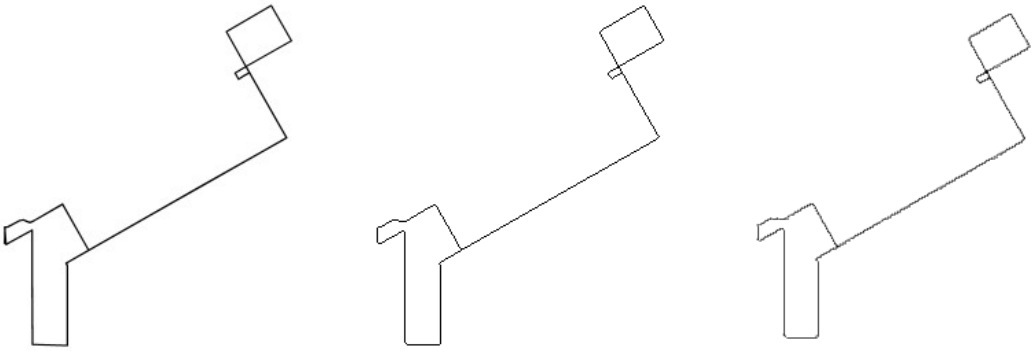
4.1.4 Validation

Validating the performance of GANs is an open problem [40] since neither of the loss functions we use are able to determine the quality of the generated bus route. The conditional GAN loss (Eq. 4.1) from a bad discriminator and bad generator may be equal to the loss from a well-trained discriminator and a well-trained generator. Since generator and discriminator are players in a minimax game, the loss function does not converge. Regarding the L1 loss (Eq. 4.2), it is possible that none of the bus route pixels in a generated image overlap the bus route pixels in the training image, resulting in high loss. However, the generated image may well represent a viable bus route given the road network. To solve this problem some researchers [27, 40] use Amazon Mechanical Turk where real and generated images are distributed to humans who try to distinguish between the two. However, since our task is to generate images of efficient bus routes we can transform the images back to routes in a graph, determine the bus stops, and then compare travel times of random trips using the generated bus stops versus the real ones.

4.1.4.1 Image to Graph Conversion

Our first task is to convert the generated bus route images into a graph where the edges are the road segments followed by the bus. The authors of [22] faced a similar problem when they used GANs to generate road networks and they had to convert the generated road network images back to graphs. Because the roads were too thick, they first used a skeletonization algorithm which iteratively removes pixels on object borders until no more pixels can be removed [48]. Afterwards, they created graph $G = (V,E)$ by adding node V_i to V for each skeleton pixel. For each V_i they checked its 8-neighbourhood pixels adding edges $E_{i,j} = (V_i, V_j)$ to each 8-neighbourhood pixel V_j which was part of the skeleton. We utilize the same technique and for demonstration purposes we use a bus

route image from the training set. We use the `skeletonize` method from `skimage` [43] and the process is displayed in figure 4.5. Before performing the skeletonization we convert the image to greyscale and set all pixels with intensity below 127 to 0 and the rest to 255. In case our generated route image contains more than one connected components, we keep the largest one. We observe that in the skeletonized figure 4.5b unnecessary black pixels from the original figure 4.5a have been removed and the conversion to a graph is successful since the graph figure 4.5c is identical to the skeletonized figure.



(a) Original route image (b) Route skeleton image (c) Route graph

Figure 4.5: Bus route image to graph procedure

4.1.4.2 Route Creation

Now that we have transformed the image to a graph $G = (V, E)$ we have to find a cycle which traverses all edges at least one time. This problem is known as the Chinese Postman Problem [44]. We add weight=1 to all vertical and horizontal edges and weight=$\sqrt{1^2 + 1^2}$ to all diagonal edges. To solve it we use the following steps [9, 11]:

1. Find vertices with odd degree
2. Find all possible pairings of odd degree vertices
3. Find the shortest distance path between all those pairings
4. Negate the weight of the shortest paths
5. Create graph using the shortest paths
6. Perform maximum weight matching in graph of step 5

7. Add the edges found in step 6 to the original graph

8. Find Euler circuit starting from random node

Maximum weight matching, mentioned in step 6, is the process of finding the largest possible set of edges, without common vertices, so that the sum of weights is maximized. This is why we negate the weights in step 4, in order to essentially perform minimum weight matching on the original weights since weights represent distance. The `networkx` [21] method used for this step implements the maximum weight matching algorithm from [18].

4.1.4.3 Bus Stop Creation

After determining the path of the bus, we have to choose the bus stops. To this end, we take advantage of our dataset and analyse the stop distances in all routes of our training set. During the bus route image creation process we log the ids of the routes we plot, as well as the GTFS feeds they come from, so we are able to check the corresponding feeds of images for stop distances. Using the complete dataset would yield noisy results since it includes country wide trips with longer stop distances. In addition, we must not use any routes from the validation or test set. We iterate over all our routes, and for each of their trips we calculate the great-circle distance of consecutive bus stops using the haversine formula (Eq. 4.4 - 4.6), where R is earth's radius in metres. We then calculate the average stop distance for each route.

$$a = \sin^2(\Delta\text{lat}/2) + \cos\text{lat}_1 \cdot \cos\text{lat}_2 \cdot \sin^2(\Delta\text{lon}/2) \tag{4.4}$$

$$c = 2\,\text{atan2}(\sqrt{a}, \sqrt{1-a}) \tag{4.5}$$

$$d = Rc \tag{4.6}$$

The average bus stop distance distribution statistics are available in table 4.1 and the histogram is displayed in figure 4.6. We checked the unexpected minimum value of 0 and it comes from a placeholder route which has only 1 bus stop and it is an on demand service for people with disabilities. We observe that large distances which may come from routes with a few stops, e.g. tourist sightseeing buses, create a right-skewed distribution with mean $417m$ and median $311m$. Our results, especially $Q_2(25\%) = 253.82$ and $Q_3(75\%) = 404.93$, verify [45] who states that in Europe bus stop distances range from 200 to $400m$. Therefore, we decide to set the bus stops for each route $300m$ apart. Using the code in appendix A we turn two adjacent nodes into coordinates and

with the haversine formula (Eq. 4.4 - 4.6) we can find the length of diagonal, vertical, and horizontal edges in metres. Then, beginning from the start of the route, we create stops every $300m$ until we are at a distance of less than $450m$ from the start. Figure 4.7 displays a sample route with its stops. We display a route from the training set with low amount of edge overlap for visibility.

samples	mean(m)	min(m)	25%(m)	50%(m)	75%(m)	max(m)
2712	417.88	0.00	253.82	311.53	404.93	4,010.04

Table 4.1: Statistics for the average bus stop distance from each route

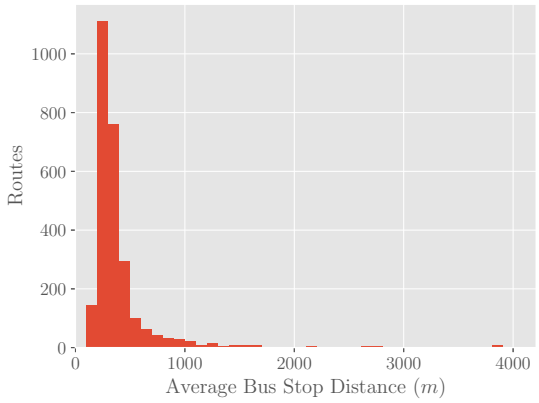


Figure 4.6: Average bus stop distance histogram

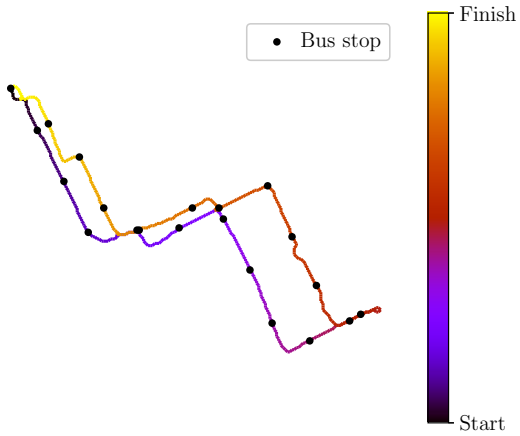


Figure 4.7: Sample route with stops

4.1.4.4 Compare Travel Times

Now that we have a method to convert generated route images to a sequence of bus stops we can compare our bus stop locations and the manually created bus stops using random trips. Our goal is to create a graph using walking time as edge weights, on top of which we connect our bus stops using bus riding time as weights. For each road network in the test set, we use `osmnx` [10] to download a directed graph of the network which we convert to undirected in order to avoid bus stops at dead ends. The edge weights are given in metres so we transform them to walking time by dividing with the average walking speed of $1.2m/s$ [17]. Afterwards, we place our bus stops to their closest nodes and use Dijkstra's algorithm [16] to find the shortest path from each bus stop to the next one. Then, we add a directed edge from each origin node

to its destination node with the weight of the path. The weight which is again in metres is converted to time by dividing with the average bus speed of $4.74 m/s$. The average bus speed was recorded from buses in London during 2018-2019 [1]. We do not use the paths we created before because we would have to match our route graph exactly on the road network graph. This may be impossible if some roads are missing. However, creating the route path is necessary in order to determine the bus stops and their sequence. Next, we add waiting time at each bus stop. To calculate the average waiting time we first calculate the bus headway by dividing the total route duration by the number of buses we have available. Then, we find the average waiting time by dividing the bus headway by 2. Finally, we generate random pairs of nodes and use Dijkstra's algorithm [16] in order to find the average travel time. Following the exact same procedure, but using the real bus stops from the GTFS feeds, we can compare the two results.

4.2 U-Net

4.2.1 Architecture

Our second approach simplifies the problem and uses a simple feed forward neural network. Since our target output is composed of only black and white pixels, it can be thought of as a mask with black pixels indicating the existence of a bus route and white pixels its non-existence. In the image processing literature this is known as image segmentation. U-Net, our GAN's generator architecture was originally built for such a purpose [37]. Instead of generating a 3 channel RGB image as output, we generate a two-dimensional mask with equal size to the input image. The value in each output cell is the probability that the corresponding input pixel is part of a bus route.

Figure 4.8 displays U-Net's architecture. Although it seems similar to our GAN's generator, the convolutional and upsampling blocks are implemented differently. As defined in the original paper [37], each convolutional block has a (2, 2) max-pooling layer, and two convolutional layers, each with kernel size=(3, 3), strides=(1, 1), padding, and ReLU activation. Max-pooling is not implemented in the first convolutional block and the last two convolutional blocks also have a dropout layer with dropout value of 0.5. A minor difference between the original paper [37] and our implementation is that they do not use padding. The differences between the GAN's encoder and U-Net's encoder are the usage of max-pooling instead of strides for downsampling, the usage of 2

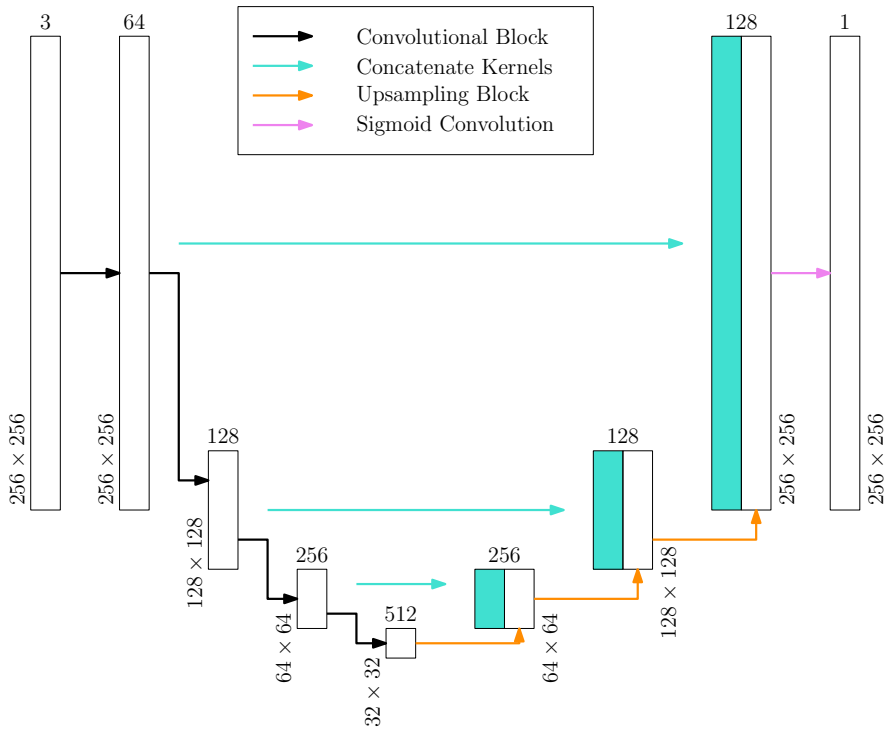


Figure 4.8: U-Net architecture

consecutive convolutional layers, the smaller kernel size of (3, 3), the ReLU activation instead of Leaky ReLU, and the dropout in the last two blocks.

In each upsampling block we have an upsampling layer which doubles the dimensions, and a convolutional layer with kernel size=(2, 2), strides=(1, 1), padding, and ReLU activation. We then merge with the corresponding layer in the encoder and follow with two convolutional layers identical to the pair in the encoder. In the final convolution we use kernel size=(1, 1) and a sigmoid activation in order to map the 128 features of each pixel in the previous layer into a value in the interval [0, 1]. Output values above 0.5 indicate that the corresponding pixel is part of a route.

4.2.2 Training

To train this model we first transform the target RGB output of bus route images into a two dimensional mask with a value of 1 in the black pixels and 0 in the white pixels. This is performed by converting the RGB image into an one channel greyscale image using the package `pillow` [3]. Afterwards, we turn each pixel with intensity below 220 into 1 and intensities above 220 into 0. We don't use 127 as the threshold because

some grey pixels get lost. The reverse procedure can be performed in order to go from a mask to an RGB image. The loss we use to train this model is binary cross-entropy (Eq. 4.7) where $y_{i,j}$ is the ground truth indicator $\{0, 1\}$ for pixel i, j and $p_{i,j}$ is the predicted probability that pixel i, j belongs to a bus route.

$$\mathcal{L}_{binaryCE} = -\frac{1}{256 * 256} \sum_{i=1}^{256} \sum_{j=1}^{256} (y_{i,j} \log(p_{i,j}) + (1 - y_{i,j}) \log(1 - p_{i,j})) \tag{4.7}$$

4.2.3 Validation

Validating the performance of the U-Net is simpler since its goal is to create a mask over the pixels of bus routes. Therefore, metrics such as binary cross entropy loss (Eq. 4.7), accuracy (Eq. 4.8), precision (Eq. 4.9), recall (Eq. 4.10), and F_1 score (Eq. 4.11), are able to determine its performance. In the equations, TP is true positives, TN is true negatives, FP is false positives, and FN is false negatives. In addition, by converting the mask in an RGB image we can also use the validation method from GANs, subsection 4.1.4, as an extrinsic method of evaluation. In our case, this extrinsic method of evaluation is the most relevant metric since it compares the average travel time from generated routes with the average travel time from manually designed routes.

$$\text{accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \tag{4.8}$$

$$\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{4.9}$$

$$\text{recall} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{4.10}$$

$$F_1 = 2\frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} \tag{4.11}$$

Chapter 5

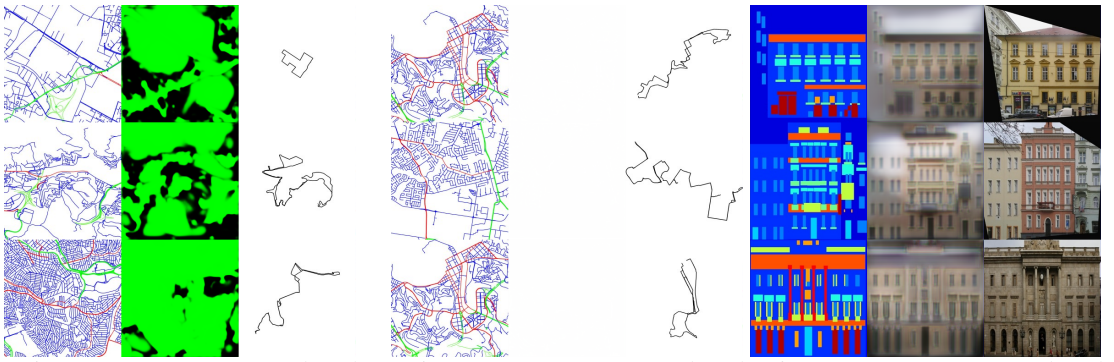
Experiments & Discussion

In this chapter we detail the experiments we perform, their results, and their analysis. We run all experiments on Google Cloud Platform using a virtual machine with 2 vCPUs, 13GB RAM, and NVIDIA Tesla K80. To log the configurations and results of each experiment we use the `sacred` framework [21] and store them in a mongoDB database. We implement the models using Keras [12] and the starting point of our code comes from [14] which we refactor to fit our needs. A limitation of our research is that, due to time constrains, we run each experiment with a single seed.

5.1 Generative Adversarial Network

To train the GAN model we follow the original paper [27] and use the Adam optimizer [30] with learning rate 0.0002, $\beta_1 = 0.5$, and $\beta_2 = 0.999$. The weights are also initialized as in [27] from a normal distribution with mean 0 and standard deviation 0.02. In the original paper, the authors used different training epochs and batch sizes for different tasks. We choose to train for 10 epochs with batch size of 4. The number of parameters prohibits a larger batch size because it does not fit into memory. In addition, we train with all possible combinations of the following sets: skip connections in generator = {True, False}, L_1 weight (λ) = {0, 10, 100}, Leaky ReLU in generator's encoder = {True, False}, number of convolutional blocks in generator/discriminator = {4, 8}. In total, we have 24 experiments. In [27], the authors use skip connections, test with $\lambda = \{0, 10\}$, and use Leaky ReLU. They use 8 convolutional blocks but we also try a shallower architecture of 4 because the level of detail required in our output is much simpler. The dataset of 3,613 pairs of road networks-bus route images is split 80/10/10 in training/validation/test set.

As we mentioned in subsection 4.1.4, neither loss of the GAN is indicative of its performance. Manual inspection of the generated images, using the validation set, reveals two types of output. The first type, displayed in subfigure 5.1a, comes from experiments which have L_1 weight 0, i.e. no L_1 loss. There is low similarity to the input structure with a random colour dominating over the images and none of them resemble our targets. Six out of eight experiments with no L_1 loss produce such type of output and the remaining two produce only white images. Images from configurations with L_1 weight either 10 or 100 also produce pure white images (subfigure 5.1b). Although those images are technically much closer to the target than the ones with no L_1, they are still far from our expected output. To ensure that we implemented the model correctly and without bugs, we train using the facades dataset of the original paper [27]. The images in subfigure 5.1c indicate that the code implementation is correct. As a result of the empty output, we are unable to use our validation methodology of 4.1.4 in our GAN approach. The ineffectiveness of our model is apparent purely from manual observation.



(a) L_1 weight 0 (b) L_1 weight $\{10, 100\}$ (c) Facades dataset

Figure 5.1: Generated images from our validation set (a), (b), and from the facades dataset (c) [27]

None of the other hyperparameters affect the style of our output. However, the increased number of convolutional blocks, and the use of skip connections, has an immediate effect in the training duration. Both of the above configurations increase the number of learnable parameters, which in turn increases training times. Table 5.1 displays the average training time, in minutes, using either 4 or 8 convolutional blocks, in combination with the use, or not, of skip connections. The average time of our experiments is just over one hour, 66 minutes.

The abstract output in 6/8 experiments without L_1 loss (subfigure 5.1a) is unex-

		Convolutional Blocks		
		4	8	Average
Skip Connections	*True*	60.8	98.2	79.5
	False	**41.3**	65.3	53.3
	Average	51.1	81.7	66.4

Table 5.1: GAN average training time (mins), for each combination of *skip connections* and *convolutional blocks*

pected since in the original paper [27] the authors are able to produce convincing results even without L_1. However, since the remaining two experiments produced more reasonable results in the form of all white images, we can conclude that the hyperparameter configurations which produce abstract images destabilize training. The two configurations which produce white images have no skip connections, 4 convolutional blocks, and the Leaky ReLU in the generator is both present and absent.

By using L_1 loss, the model learns to erase all roads from the input image and generate white output (subfigure 5.1b). This happens because, on average, each bus route image has only 1.61% black pixels. For comparison, in the MNIST digits dataset [13], the foreground is 13.23% of the image. As a result, the easiest way to reduce the L_1 loss, and therefore total loss, is to paint the image white, i.e. the background colour. Especially with L_1 weight in $\{10, 100\}$, the network is inclined to produce an image identical to the background since the contribution of L_1 in the total loss (Eq. 4.3) is much higher than the contribution of the GAN loss (Eq. 4.1). Therefore, we conclude that the L_1 loss is not fit for images with heavily imbalanced foreground/background.

5.2 U-Net

The configuration of the U-Net model is as described in subsection 4.2.1 and follows the original paper [37]. However, apart from starting with 64 kernels in the first layer, as in [37], we also experiment with 16 and 32 initial kernels, doubling them in each layer of the encoder and dividing them by 2 in each layer of the decoder. We initialize our weights with He normal initialization [23] as did the authors of [37]. He normal initialization is specifically designed with ReLU activations in mind and the values come from a normal distribution with zero mean and standard deviation $\sqrt{2/N}$, where

N is the number of incoming neurons from the previous layer. For example, if the previous layer has 32 filters, and we use kernel size $(2,2)$, then $N = 32 \cdot 2 \cdot 2 = 128$. To train we use the Adam optimizer [30] as in GANs, and we experiment with learning rates in $\{0.0005, 0.0001, 0.00005\}$, β_1 in $\{0.9, 0.75, 0.5\}$, and β_2 0.999. The authors of Adam suggest $\beta_1 = 0.9$ but we also try lower values because we have a similar encoder to our GAN where the authors found 0.9 being too large [27]. Instead of Adam, the authors of U-Net used simple stochastic gradient descent and they did not mention their learning rate. In total, we have 27 different configurations. We train with batch size 10, for a maximum of 100 epochs, but we also implement early stopping with patience 3. Early stopping stops the training procedure if the loss in the validation dataset has not improved for a number of epochs, in our case 3. It is used in order to prevent overfitting, but also save time and computational resources. The dataset is split 80/10/10 in training/validation/test set.

The minimum cross entropy loss (Eq. 4.7) of the validation set, from all configurations, is available in tables 5.2, 5.3, and 5.4. We observe that the lowest average loss comes from using 32 initial kernels. In addition, combining a high β_1 with a low learning rate, or a high learning rate with a low β_1, usually results in high loss. On average, the learning rate of 0.0001 provides the lowest loss.

Looking at the accuracy of our model in the validation set reveals that it is equal to 98.4% for all configurations. Observing the generated masks shows that all configurations classified zero pixels as part of a bus route and the output is a white only image, identical to our GANs with L_1 loss (subfigure 5.1b). As a result, precision is not defined (division by zero), and recall is zero for all configurations.

As is the case in the GAN methodology, our model suffers from the imbalanced target classes. Imbalanced datasets are considered one of the crucial problems in deep learning [31] and specifically in image segmentation [32]. In [39] the authors tested training with standard cross-entropy loss, as we did, versus training with a weighted cross-entropy loss which amplifies the error of the under-represented classes. The results show a significant reduction in performance using the standard loss. Since only 1.61% of our input belongs to the *bus route* class, our model is overfitting on the *non bus route* class and classifies all pixels as such. However, in this method we are able to lower the threshold of classifying a pixel as part of a route in order to obtain bus routes.

To that end, we use the saved model weights which produced the lowest validation set loss of 0.0569. Those weights come from training with learning rate of 0.0001, β_1

		Learning Rate			
		0.0005	0.0001	0.00005	Average
β_1	0.9	**0.0585**	0.0641	0.2429	0.1218
	0.75	0.0599	0.0666	0.2430	0.1232
	0.5	0.2529	0.0694	0.0890	0.1371
	Average	0.1238	0.0667	0.1916	0.1274

Table 5.2: Minimum validation set cross-entropy loss with **16 initial kernels**

		Learning Rate			
		0.0005	0.0001	0.00005	Average
β_1	0.9	0.0813	**0.0569**	0.2429	0.1270
	0.75	0.2550	0.0626	0.0587	0.1254
	0.5	0.0645	0.0576	0.0672	0.0631
	Average	0.1336	0.0590	0.1229	0.1052

Table 5.3: Minimum validation set cross-entropy loss with **32 initial kernels**

		Learning Rate			
		0.0005	0.0001	0.00005	Average
β_1	0.9	0.2550	0.0586	0.0618	0.1251
	0.75	0.2550	0.1095	0.0584	0.1410
	0.5	0.2550	0.0608	**0.0576**	0.1245
	Average	0.2550	0.0763	0.0593	0.1302

Table 5.4: Minimum validation set cross-entropy loss with **64 initial kernels**

of 0.9, and 32 initial kernels. To find the optimal threshold we perform our experiments in the validation set. The plots in figure 5.3 show the behaviour of precision, recall, F_1 score, and accuracy as we decrease the classification threshold from 0.5 to 0.01. Because our classes are imbalanced, precision, recall, and accuracy, are not useful. For example, as mentioned above, classifying all pixels as non-route pixels results in accuracy of 98.4%. Therefore, the most indicative measure is the F_1 score which takes into account both precision and recall. The highest F_1 value of 0.262 comes from threshold of 0.15. At that threshold, precision is 19%, which means that 1 in 5 pixels we classify as part of a bus route is indeed part of a bus route. Recall is 42%, which means that for every 5 pixels which belong in a bus route we are able to identify 2. Figure 5.3 displays the masks we obtain as we decrease the threshold level (t). Apart from the F_1 score, the masks also indicate that bus routes which span the largest set of connected black pixels at threshold $t = 0.15$ will perform better than using another threshold. For example at $t = 0.25$ we do not have enough connected black pixels, and at $t = 0.05$ the bus route will traverse the entire road network. In addition, we observe that our model gives higher probability of being part of a bus route to wide roads, i.e. red and green roads in our input.

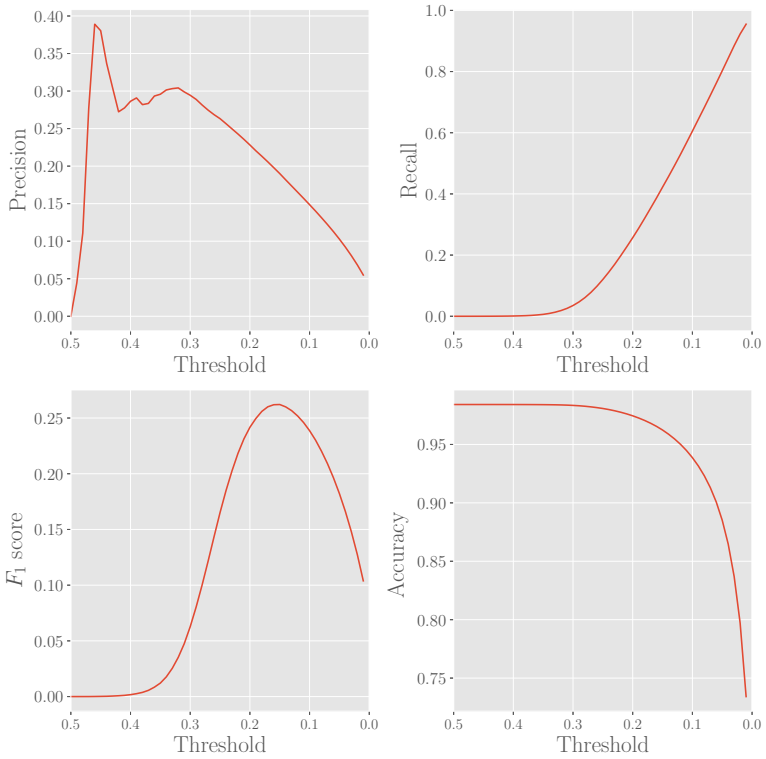


Figure 5.2: Behaviour of our metrics as we decrease the classification threshold

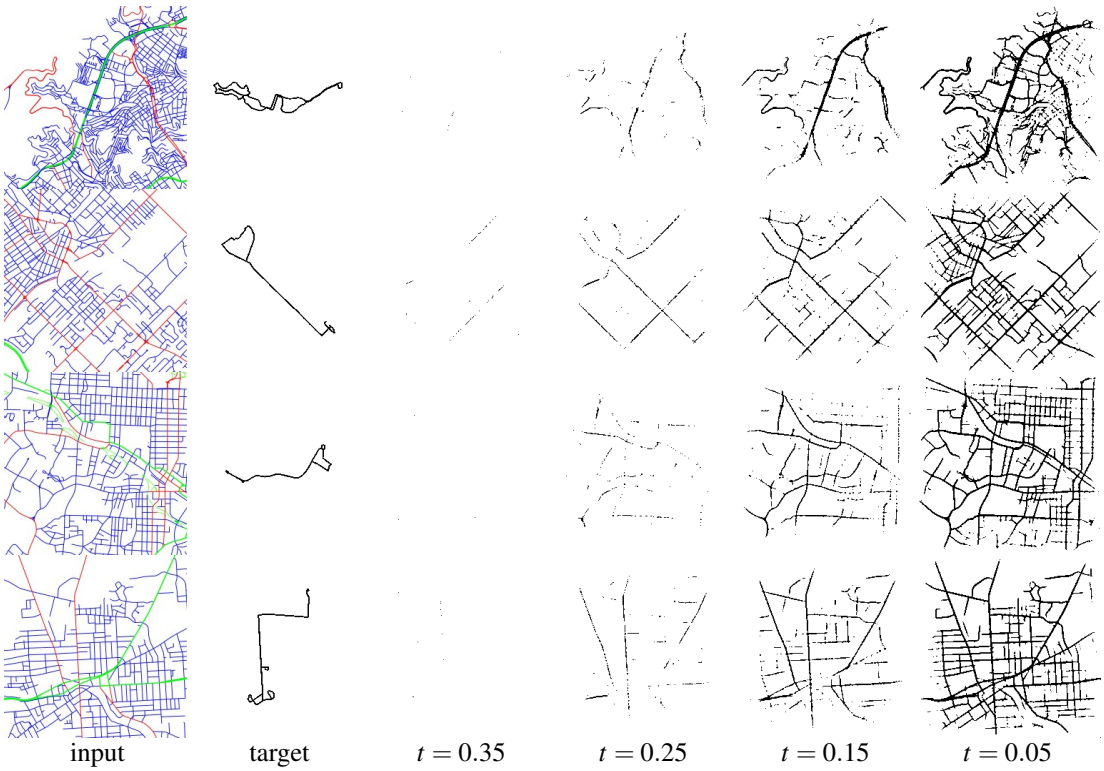


Figure 5.3: Bus route masks as we decrease the classification threshold (t)

For the aforementioned reasons, we report the results of the test set using $t = 0.15$. Test set F_1 score is 0.267, precision is 19%, recall is 44%, and accuracy is 96%. The binary cross entropy loss, which does not depend on the threshold, is 0.0559.

We also use the extrinsic evaluation method we created for the GAN in subsection 4.1.4 using 100 random trips for each route. We perform three simulations, each with a different number of concurrently running buses. The results of various statistics, averaged over all routes in our test set, are available at table 5.5. We report the average $\pm$ the standard error of the mean (SEM), which is given by $\sigma_{\bar{x}} = \frac{\sigma}{\sqrt{N}}$, where σ is the standard deviation of the values, and N is 362, the size of our test set. Figure 5.4 showcases samples of routes generated with classification threshold of 0.15 where we keep only the largest connected set of black pixels and use skeletonization as described in subsection 4.1.4.

In table 5.5 we observe that on average the service provided by the generated routes is only $55s$ slower (+3% average travel time) than the manually designed routes. The difference drops from $58s$ to $49s$ as we increase the number of concurrently running buses from 3 to 7. This occurs because the average generated route duration (time required for a bus to complete the route) is $3223 \pm 112.5s$ while the average manually

	Number of buses			
	3	5	7	Average
generated routes travel time (s)	1904 ± 20.9	1839 ± 20.7	1803 ± 20.8	1849 ± 12.1
real routes travel time (s)	1846 ± 20.2	1782 ± 19.4	1754 ± 19.4	1794 ± 11.4
generated routes wait time (s)	537 ± 18.7	322 ± 11.2	230 ± 8.0	363 ± 8.7
real routes wait time (s)	425 ± 12.7	255 ± 7.6	182 ± 5.5	287 ± 6.1
# of buses used in generated routes	25 ± 0.8	36 ± 0.8	42 ± 0.9	34 ± 0.5
# of buses used in real routes	31 ± 0.7	40 ± 0.8	45 ± 0.8	39 ± 0.5
# of common trips completed on foot	59 ± 0.7	47 ± 0.7	41 ± 0.7	49 ± 0.5

Table 5.5: Various statistics averaged over all routes of our test set $\pm$ SEM

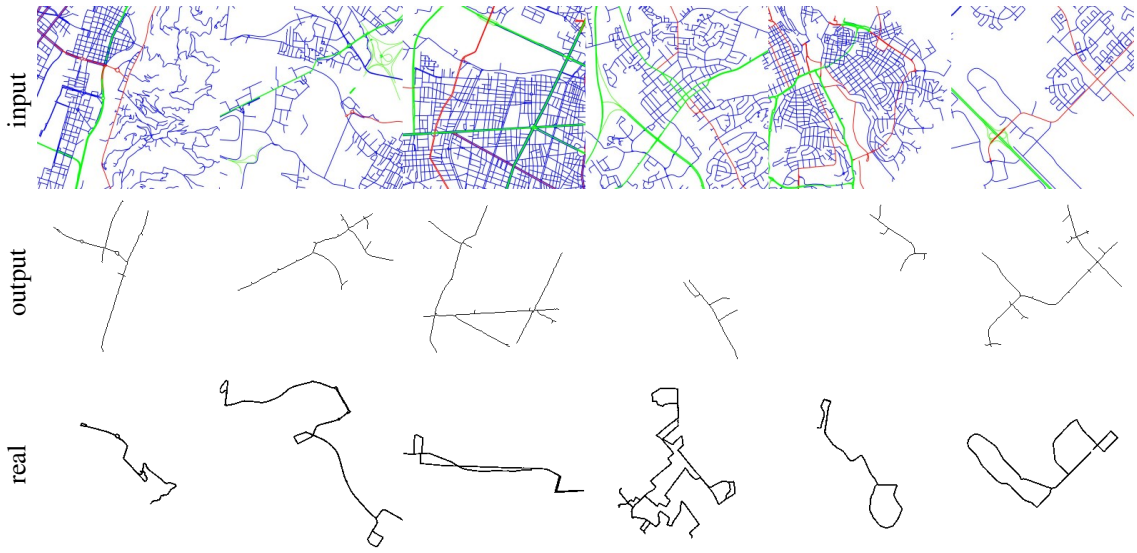


Figure 5.4: Sample routes generated during the external validation process

designed route duration is $2550 \pm 76.5s$. This difference impacts the drop of wait time at bus stops as we increase the number of buses. As a result of our longer duration, increasing the number of buses has a greater impact in our waiting time since it is calculated as $(\frac{\texttt{total route duration}}{\texttt{\# of buses}})/2$ (explained in 4.1.4.4). Therefore, as we increase the number of buses, the wait time in the generated routes drops by $307s$ while in the real routes it drops by $243s$. Furthermore, we have that out of the 100 random trips, on average 34 of them used a generated bus route and the remaining 66 where performed on foot since it was faster. Using 3 buses in the generated routes, 1/4 of the trips are using a bus while by using 7 buses the proportion is increased to 2/5. In the real routes, the average number of buses used is slightly higher at 39. On average, half of the random trips are completed on foot both in the generated and manually designed routes. This is expected since the real routes are designed to work in conjunction with other routes in the network in order to serve the whole region. Therefore, a large portion of random trips cannot be served by a bus. Since our model trains in the aforementioned real routes, the generated routes also suffer from the same problem.

A limitation of our approach is that the shortest paths followed by our buses are calculated in an undirected graph which means we assume all roads to be bidirectional. The shortest paths between stops in real routes are also calculated in the same undirected graph so it is still a fair comparison. However, the real bus stops were placed with directed roads in mind, so they may be sub-optimal in an undirected graph. Looking at figure 5.4 we observe that the shape of our routes is mainly composed of 2-3 main roads with branches on them. On the contrary, most real routes are traversing one main road without branches, back and forth, usually with a circular path in both ends in order to change direction. It is interesting that our unintuitive shape of routes with branches is able to perform equally well with the manually designed routes. However, having branches is prohibited in practice since the bus would have to perform U-turns at the end of each branch.

Chapter 6

Conclusion

In this final chapter we present a summary of our research and provide directions for future work on the topic.

6.1 Summary

In our project we identify the costly and continuous requirement of bus agencies to expand and improve their service. In order to automate the process of designing bus routes, we abolish the origin-destination matrix which was a prerequisite in all previous approaches. To that end, we build two deep convolutional models which act upon an image of the available road network and aim to produce an efficient bus route. The bus routes we use for training/evaluation are manually designed and they are collected in the form of GTFS feeds. After collecting all available feeds we perform a cleaning process which fixes any violations of the GTFS standard, resulting in the first clean GTFS dataset for bus routes. Our first network is a conditional GAN, called *pix2pix* [27], whose purpose is to output pictures of plausible routes. Due to class imbalance in our training bus routes, only 1.61% of the pictures depicts a route, this approach fails to generate any useful results. The second network is a convolutional auto-encoder, titled U-Net [37], and its goal is to generate bus routes by creating a mask over the road network. This method also suffers from the imbalance in our dataset. However, by lowering the classification threshold we are able to obtain efficient routes which, when tested on random trips, report only a 1 minute higher travel time (+3%) than the manually designed routes.

6.2 Future Work

As this is the first attempt to utilize images of manually designed bus routes in order to generate new ones, there is plenty of room for improvement. Regarding the usage of GANs, having images as output is an overkill since the target images have only two colours. Therefore, they can be processed as masks, like we did in the U-Net architecture. This would simplify the network, decreasing training time, but most importantly it would allow us to lower the threshold in case we have empty output. By performing the image segmentation approach using GANs we could get different routes for the same road network. As a result, we could combine those routes in order to form a complete bus network, with multiple routes, for a specific area.

Regarding the image segmentation technique with U-Net, a simple change which could yield better results is the usage of strides instead of max-pooling. As mentioned in section 4.1, strides allow the model to learn its own, non-deterministic downsampling. Furthermore, the generated routes, using threshold of 0.15, are longer than the manually designed routes and include branches, both of which impede its performance. In order to shorten the routes, and possibly get rid of the branches, we could slightly increase the classification threshold. The ideal approach would be, using the validation set, to perform our external evaluation method (subsection 4.1.4) with different threshold values and use the value which results in the lowest average travel time. In addition, we could also create a bus network, similar to the one proposed for the GAN, by shifting the road network image to some direction. For instance, if we want to create a bus service for a large geographical area, we can create partially overlapping images of its road network and create a route for each image. Since U-Net favours wide roads, some parts of the routes will probably overlap in wide roads which will conveniently serve for bus transfers.

A limitation of our research is that we did not design our networks with dataset imbalance in mind. Therefore, methods specifically created for such cases should be the focus of future research. Two loss functions designed to tackle dataset imbalance are weighted cross-entropy loss and focal loss [33]. Therefore, the GAN architecture should be used to create masks instead of images, and instead of L_1 the GAN loss should be combined with any of the aforementioned losses. Replacing the simple cross-entropy of the U-Net architecture with either of them is another promising approach.

Bibliography

[1] Buses performance data. URL: https://www.tfl.gov.uk/corporate/publications-and-reports/buses-performance-data.

[2] A fast, forgiving GTFS reader built on pandas DataFrames: Remix/partridge. URL: https://github.com/remix/partridge.

[3] The friendly PIL fork (Python Imaging Library). Contribute to python-pillow/Pillow development by creating an account on GitHub. URL: https://github.com/python-pillow/Pillow.

[4] OpenMobilityData - Public transit feeds from around the world. URL: https://transitfeeds.com/.

[5] A Python 3.6+ tool kit for analyzing General Transit Feed Specification (GTFS) data: Mrcagney/gtfstk. URL: https://github.com/mrcagney/gtfstk.

[6] Reference — Static Transit. URL: https://developers.google.com/transit/gtfs/reference/.

[7] A small, python-based library for creating map images with lines, markers and polygons.: Komoot/staticmap. URL: https://github.com/komoot/staticmap.

[8] Wmflabs OSM no labels. URL: https://tiles.wmflabs.org/osm-no-labels/$%7Bz%7D/$%7Bx%7D/$%7By%7D.png.

[9] Andrew Brooks. Graph Optimization with NetworkX in Python. URL: https://www.datacamp.com/community/tutorials/networkx-python-graph-tutorial.

[10] Geoff Boeing. OSMnx: New methods for acquiring, constructing, analyzing, and visualizing complex street networks. 65:126–139.

[11] Andrew Brooks. Graph optimization solvers for the Postman Problems: Brooksandrew/postman_problems. URL: https://github.com/brooksandrew/postman_problems.

[12] François Chollet et al. Keras. URL: https://keras.io.

[13] Gregory Cohen, Saeed Afshar, Jonathan Tapson, and André van Schaik. EMNIST: An extension of MNIST to handwritten letters. URL: http://arxiv.org/abs/1702.05373, arXiv:1702.05373.

[14] Thibault de Boissiere. Implementation of recent Deep Learning papers. Contribute to tdeboissiere/DeepLearningImplementations development by creating an account on GitHub. URL: https://github.com/tdeboissiere/DeepLearningImplementations.

[15] Guy Desaulniers and Mark D. Hickman. Chapter 2 Public Transit. In Cynthia Barnhart and Gilbert Laporte, editors, *Handbooks in Operations Research and Management Science*, volume 14 of *Transportation*, pages 69–127. Elsevier. URL: http://www.sciencedirect.com/science/article/pii/S0927050706140025, doi:10.1016/S0927-0507(06)14002-5.

[16] E. W. Dijkstra. A note on two problems in connexion with graphs. 1(1):269–271. doi:10.1007/BF01386390.

[17] Kay Fitzpatrick, Marcus A. Brewer, and Shawn Turner. Another Look at Pedestrian Walking Speed:. URL: https://journals.sagepub.com/doi/pdf/10.1177/0361198106198200104, doi:10.1177/0361198106198200104.

[18] Zvi Galil. Efficient algorithms for finding maximum matching in graphs. 18(1):23–38.

[19] Klokan Technologies GmbH. Mapping platform for quick publishing of zoomable maps online – MapTiler. URL: https://www.maptiler.com/.

[20] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative Adversarial Networks. URL: http://arxiv.org/abs/1406.2661, arXiv:1406.2661.

[21] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring Network Structure, Dynamics, and Function using NetworkX. In Gaël Varoquaux, Travis

Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11–15.

[22] Stefan Hartmann, Michael Weinmann, Raoul Wessel, and Reinhard Klein. StreetGAN: Towards Road Network Synthesis with Generative Adversarial Networks. URL: http://cg.cs.uni-bonn.de/aigaion2root/attachments/street_gan_2017.pdf.

[23] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. URL: http://arxiv.org/abs/1502.01852, arXiv:1502.01852.

[24] G. E. Hinton and R. R. Salakhutdinov. Reducing the Dimensionality of Data with Neural Networks. 313(5786):504–507. URL: https://science.sciencemag.org/content/313/5786/504, arXiv:16873662, doi:10.1126/science.1127647.

[25] O. J. Ibarra-Rojas, F. Delgado, R. Giesen, and J. C. Muñoz. Planning, operation, and control of bus transport systems: A literature review. 77:38–75. URL: http://www.sciencedirect.com/science/article/pii/S0191261515000454, doi:10.1016/j.trb.2015.03.002.

[26] Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. URL: http://arxiv.org/abs/1502.03167, arXiv:1502.03167.

[27] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A. Efros. Image-to-Image Translation with Conditional Adversarial Networks. URL: http://arxiv.org/abs/1611.07004, arXiv:1611.07004.

[28] Jonathan Roberts Consulting Ltd. Bus Planning Literature Review. URL: https://www.london.gov.uk/sites/default/files/bus_planning_literature_review_jrc_ltd.pdf.

[29] Andrej Karpathy. CS231n Convolutional Neural Networks for Visual Recognition. URL: http://cs231n.github.io/convolutional-networks/.

[30] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. URL: http://arxiv.org/abs/1412.6980, arXiv:1412.6980.

[31] Kyungmoon Lee and Heechul Jung. DavinciGAN: Unpaired Surgical Instrument Translation for Data Augmentation. URL: https://openreview.net/forum?id=rJxMvRFxeN.

[32] Zeju Li, Konstantinos Kamnitsas, and Ben Glocker. Overfitting of neural nets under class imbalance: Analysis and improvements for segmentation. URL: http://arxiv.org/abs/1907.10982, arXiv:1907.10982.

[33] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. Focal Loss for Dense Object Detection. URL: http://arxiv.org/abs/1708.02002, arXiv:1708.02002.

[34] Michael Mathieu, Camille Couprie, and Yann LeCun. Deep multi-scale video prediction beyond mean square error. URL: http://arxiv.org/abs/1511.05440, arXiv:1511.05440.

[35] Mehdi Mirza and Simon Osindero. Conditional Generative Adversarial Nets. URL: http://arxiv.org/abs/1411.1784, arXiv:1411.1784.

[36] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. URL: http://arxiv.org/abs/1511.06434, arXiv:1511.06434.

[37] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-Net: Convolutional Networks for Biomedical Image Segmentation. URL: http://arxiv.org/abs/1505.04597, arXiv:1505.04597.

[38] Wade Roush. Welcome to Google Transit: How (and Why) the Search Giant is Remapping Public Transportation. 2012/00/00. URL: https://trid.trb.org/view/1138348.

[39] D. Sakkos, E. S. L. Ho, and H. P. H. Shum. Illumination-Aware Multi-Task GANs for Foreground Segmentation. 7:10976–10986. doi:10.1109/ACCESS.2019.2891943.

[40] Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. Improved Techniques for Training GANs. URL: http://arxiv.org/abs/1606.03498, arXiv:1606.03498.

[41] Dominik Scherer, Andreas Müller, and Sven Behnke. Evaluation of pooling operations in convolutional architectures for object recognition. In *International Conference on Artificial Neural Networks*, pages 92–101. Springer.

[42] Paschalis Skouzos. *MLP Coursework 2: Exploring Convolutional Networks*. University of Edinburgh.

[43] Stéfan van der Walt, Johannes L. Schönberger, Juan Nunez-Iglesias, François Boulogne, Joshua D. Warner, Neil Yager, Emmanuelle Gouillart, and Tony Yu. Scikit-image: Image processing in Python. 2:e453. URL: `https://peerj.com/articles/453`, `doi:10.7717/peerj.453`.

[44] Victor Adamchik. The Chinese Postman Problem. URL: `https://www.cs.cmu.edu/~adamchik/21-127/lectures/graphs_4_print.pdf`.

[45] Jarrett Walker. *Human Transit: How Clearer Thinking about Public Transit Can Enrich Our Communities and Our Lives*. Island Press.

[46] Martin Wall. Is the increase in private buses due to ideology or demand? URL: `https://www.irishtimes.com/business/transport-and-tourism/is-the-increase-in-private-buses-due-to-ideology-or-demand-1.2802255`.

[47] Xiaolong Wang and Abhinav Gupta. Generative Image Modeling using Style and Structure Adversarial Networks. URL: `http://arxiv.org/abs/1603.05631`, `arXiv:1603.05631`.

[48] T. Y. Zhang and C. Y. Suen. A Fast Parallel Algorithm for Thinning Digital Patterns. 27(3):236–239. URL: `http://doi.acm.org/10.1145/357994.358023`, `doi:10.1145/357994.358023`.

Appendix A

Pixel to Coordinates Code

The method `px_to_coords` was written by us, the other four methods were already implemented in the `staticmap` [7] library.

staticmap.py

```
    def _lon_to_x(lon, zoom):
        """
         transform longitude to tile number
         :type lon: float
         :type zoom: int
         :rtype: float
         """
        if not (-180 <= lon <= 180):
            lon = (lon + 180) % 360 - 180

        return ((lon + 180.) / 360) * pow(2, zoom)


    def _lat_to_y(lat, zoom):
        """
         transform latitude to tile number
         :type lat: float
         :type zoom: int
         :rtype: float
         """
        if not (-90 <= lat <= 90):
            lat = (lat + 90) % 180 - 90

        return (1 - log(tan(lat * pi / 180) + 1 / cos(lat * pi / 180)) / pi) /
        ↪  2 * pow(2, zoom)


    def _y_to_lat(y, zoom):
        return atan(sinh(pi * (1 - 2 * y / pow(2, zoom)))) / pi * 180


    def _x_to_lon(x, zoom):
        return x / pow(2, zoom) * 360.0 - 180.0

```

```
34      def px_to_coords(px, img_width, img_height, zoom, center, tile_size = 256):
35          """
36           transform pixels to (lon, lat)
37
38           :param px: the pixel to transform to coordinates in (x, y)
    ↪ tuple with (0,0)
39           being the top left corner
40           :type px: tuple
41           :param img_width: the image width in px
42           :type img_width: int
43           :param img_height: the image height in px
44           :type img_height: int
45           :param zoom: zoom level of map
46           :type zoom: int
47           :param center: center of map in  (lon, lat)
48           :type center: tuple
49           :param tile_size: size of tiles in px
50           :type tile_size: int
51           :return: the (lon, lat) that maps to the input pixel
    ↪ location
52           :rtype: tuple
53           """
54          x_px = px[0]
55          y_px = px[1]
56          x = (x_px - img_width / 2) / tile_size + _lon_to_x(center[0], zoom)
57          y = (y_px - img_height / 2) / tile_size + _lat_to_y(center[1], zoom)
58          lon = _x_to_lon(x, zoom)
59          lat = _y_to_lat(y, zoom)
60          return lon, lat
```