The School of Mathematics

THE UNIVERSITY
of EDINBURGH

Using Machine Learning and Character Recognition Tools to Extract Text from Historic Maps

by

Kai Williams

Dissertation Presented for the Degree of
MSc in Computational Applied Mathematics

August 2020

Supervised by
Dr Stuart King

Abstract

Historic maps provide us with vital information such as the evolution of structures and landscapes over time. The National Library of Scotland owns a rare and extensive collection of historic maps. With the help of vectorisation, a wide range of valuable analyses can be undertaken on these maps. The effective vectorisation of historic structures is often inhibited by the overlapping of text over these features of interest. Therefore, to resolve this issue, we strive to formulate an automated end-to-end pipeline for: automated data set generation; text bounding box detection; pixel-level text masks and text recognition. This was initially approached via the utilisation of template matching, which resulted in a detailed representation of the density of greenspace within historic maps of Edinburgh. In addition, we demonstrated a novel method for generating a custom dataset to be used in conjunction with existing OCR methods and a Mask R-CNN and UNet deep learning model to successfully detect, recognise and segment instances of text. The Mask R-CNN model was able to accurately encapsulate 87.72% of all text inside bounding boxes whilst identifying 99.12% of all instances of text in an image. In conclusion, we show that the utilisation of our end-to-end pipeline could be used successfully to aid a wide range of future projects in the removal of text to aid vectorisation of historic maps.

Acknowledgments

Firstly, I'd like to thank my supervisor Dr Stuart King for his constant support and guidance throughout the course of the project. I would like to extend my gratitude to Christopher Fleet and the National Library of Scotland for supplying the fantastic historic maps that made this research project possible. I would also like to thank EDINA for help and supplying computing facilities for this project. I wish to express my deepest gratitude to my parents and sister for their love and unconditional support. I would also like to thank my partner and best friend, Nathalie Feeley, for being with me every step of the way.

Own Work Declaration

I declare that the work done in this project is my own, except where stated otherwise.

Contents

1 Introduction **1**

2 Digital Images and Image Pre-processing **4**
2.1 Digital Representation of Images . . . 4
2.2 Image File Formats . . . 4
2.3 Colour Quantisation . . . 4
2.3.1 K-Means Clustering . . . 5
2.4 Image Binarization . . . 6
2.4.1 Converting to Grayscale . . . 6
2.4.2 Manual Threshold Identification . . . 7
2.4.3 Otsu Method . . . 8
2.5 Removing Noise using Filters . . . 9
2.5.1 Linear Convolution . . . 9
2.5.2 Gaussian Blur . . . 10
2.5.3 Median Filter . . . 10

3 Artificial Neural Networks **11**
3.1 The History of Artificial Neural Networks . . . 11
3.2 Artificial Neural Networks Architecture . . . 11
3.3 Forward Propagation . . . 12
3.4 Activation Functions . . . 13
3.5 Loss vs Cost functions . . . 14
3.6 Gradient Decent . . . 15
3.7 Backpropagation . . . 16

4 Convolutional Neural Networks **19**
4.1 CNN Layers . . . 19
4.1.1 Convolutional Layers . . . 19
4.1.2 Activation Layers . . . 20
4.1.3 Pooling Layers . . . 20
4.1.4 Normalization Layers . . . 21
4.1.5 Dropout Layers . . . 21
4.2 Classification . . . 21
4.3 Object Localisation . . . 22
4.3.1 Intersection over Union . . . 24
4.4 Object Detection . . . 24
4.4.1 Non-max Suppression . . . 24
4.4.2 Sliding Window . . . 25
4.4.3 Faster R-CNN . . . 26
4.5 Image Segmentation . . . 27
4.5.1 UNet . . . 28
4.5.2 Mask R-CNN . . . 29

5 Optical Character Recognition **31**
5.1 Structured Text . . . 31
5.1.1 Template Matching . . . 31
5.1.2 Tesseract . . . 31
5.2 Unstructured Text . . . 32
5.2.1 Efficient and Accurate Scene Text Detector . . . 33

6 Results **35**
6.1 Template Matching 35
6.2 Pre-trained Text Detection Models 41
6.3 Creating A Dataset 45
6.4 Mask R-CNN 47
6.4.1 Accuracy Comparison 57
6.4.2 Text Recognition with Mask R-CNN 57
6.4.3 CNN for Text Rotation Prediction 60
6.5 UNet 60
6.5.1 Text Recognition with UNet and Mask R-CNN 64

7 Conclusions **66**

8 Appendix **72**
8.1 Computer Specifications 72
8.2 Custom Dataset Generator Class 72
8.3 Create Masks from XML files 80
8.4 UNet 81
8.5 Text Rotation Classifier 84
8.6 Template Matching 86

List of Tables

1 Table of different activation functions . . . 14
2 Table of a generated image's proposed texts and corresponding overlap scores . . 47
3 Table of deep learning model predictive text accuracy . . . 57

List of Figures

1 Historic map of Jamaica, by James Robertson 1804, with additional circled sugar plantations [27]. . . . 1
2 Historic map and satellite image of Leith . . . 3
3 Historic map RGB colour split plot . . . 4
4 RGB colour space of a 3200x3200 sampled image of a historic map. . . . 5
5 Example of different grayscale methods . . . 7
6 Grayscale histogram plot . . . 8
7 Illustration of a matrix convolution . . . 9
8 Gaussian blurred historic map example . . . 10
9 Diagram of a single-layer perceptron. . . . 11
10 Diagram of an artifical neural network . . . 12
11 An illustration of gradient decent. . . . 16
12 Diagram illustrating layer dependence in backpropagation . . . 18
13 A diagram an example convolutional neural network. . . . 19
14 A diagram of max pooling . . . 21
15 Object localiser example images . . . 24
16 Diagram of the sliding window and image pyramid object detection method . . . 25
17 Fully convolutional implementation of the sliding window method . . . 26
18 Illustration of the Faster R-CNN architecture . . . 27
19 Semantic vs instance segmentation example image . . . 28
20 Illustration of the UNet segmentation model architecture . . . 29
21 Illustration of the Mask R-CNN architecture . . . 30
22 Tesseract OCR engine applied to map image . . . 32
23 An example of an unstructured text image . . . 33
24 Illustration of the EAST architecture . . . 33
25 Example of a quadrangle and score map used in EAST detector . . . 34
26 Cropped tree symbol templates used in the greenspace template matching. . . . 35
27 Applied template matching to an image with lots of trees . . . 36
28 Before and after non-maximum suppression of the template matches . . . 37
29 Plot of 2D Gaussian distribution . . . 38
30 Guassian greenspace desnity plot . . . 39
31 Euclidean distance greenspace plot . . . 40
32 CharNet applied to a binary historic map image . . . 42
33 CharNet applied to a clustered historic map image . . . 43
34 EAST applied to a clustered historic map image . . . 44
35 Example image and mask generated for dataset . . . 46
36 Mask R-CNN loss function plots for epochs 1-43 . . . 48
37 Mask R-CNN prediction after trained only on the heads of the networks . . . 49
38 Mask R-CNN loss function plots for epochs 1-53 . . . 50
39 Mask R-CNN prediction after trained on all layers . . . 51
40 Large Mask R-CNN prediction before NMS . . . 52
41 Large Mask R-CNN prediction after NMS . . . 53
42 Large retrained Mask R-CNN prediction after NMS . . . 55
43 Mask R-CNN difficult image prediction . . . 56
44 Minimum area rectangle approach on STREET text . . . 58

45 Minimum area rectangle approach on BROUGHTON text 58
46 OCR applied to Mask R-CNN + minimum area rectangle approach 59
47 Example of rotational text training data 60
48 Dice loss plot for UNet 61
49 UNet predictions on large unseen image 62
50 UNet predictions on large unseen image at different rotations 63
51 UNet retrained for larger overlapping text 64
52 OCR applied to Mask R-CNN + UNet + minimum area rectangle approach . . . 65
53 Inpainting using partial convolutions 67

1 Introduction

Historic maps are vital windows into the past which, amongst their many uses, aid us in the development of landscapes of the future. Their features and depictions of urbanization provide us with invaluable information such as the structure and history of the landscape and the arrangement of buildings within it. Through construction, war and even natural disasters such as earthquakes, land has been thoroughly altered over the course of time. Historic maps in particular provide us with a wealth of information about the development and change of land use of particular areas over time. Historic maps are also vital in construction and civil engineering as they provide information about geotechnical hazards such as disused mines and quarries [48]. This aids in the process of re-landscaping, which involves the careful determination and identification of safe foundations to build upon. This highlights the huge economic benefits of digitizing and automating feature detection within historic maps for their significant use in construction and civil engineering [4].

In addition to the clear economic benefits, there are also societal benefits to digitizing and automating feature detection in historic maps. For example, in 2016, Fleet summarises the Legacies of British Slave-ownership project carried out at University College London. The project, with the help of The National Library of Scotland, has worked towards documenting all of the slave plantation estates in the British Caribbean from 1763-1833. The final map of all of the sugar plantations in Jamaica can be seen in figure 1. To do this, they have utilised georeferenced maps of Jamaica and Scotland obtained from the extensive database of digitised historic maps provided by The National Library of Scotland [8]. Projects such as this have a huge impact on society's understanding of the history, significance and scale of slavery across the globe.

Figure 1: Historic map of Jamaica, by James Robertson 1804, with additional circled sugar plantations [27].

To be able to perform further and more efficient image analysis for future projects, such as the Legacies of British Slave-ownership project, it can be beneficial to convert an image made up of individual pixels, otherwise known as a raster image, to an image defined in terms of points connected by lines and curves, otherwise known as a vector image. In 2016, Iosifescu, Tsorlini and Hurni [18] carried out the vectorization of historic buildings in maps, this is the process

of converting a raster image into a vector image. Converting raster historic maps into vector form greatly helps our ability to automatically quantify spatial changes in a region over time. A significant problem faced when vectorizing historic structures is the occurrence of text labels overlapping buildings and other features of interest in historic maps [18]. This prevents features underneath the text from being correctly vectorized. This issue was encountered, in the study by Iosifescu, Tsorlini and Hurni: "An important problem that can be already identified concerns the labels which remain in the resulted images". Through the utilisation of deep learning, this project aims to localise and extract text from historic maps in attempt to rectify the issue noted by Iosifescu, Tsorlini and Hurni.

In this project, deep learning was chosen to complete the task of text localisation and extraction. Deep learning, an example of artificial intelligence, is able to learn without human supervision and is capable of learning things such as object recognition and detection. Deep learning was chosen to be utilised in this project due to its significant progression in recent years. The evolution of deep learning has come about due to the increase in readily available computational power, and exponential increase in data generation. Deep learning architectures have shown, in some cases, to transcend human-level accuracy in tasks such as image classification and object detection. These models, however, are often limited by the amount of available training data. This is typically because these data sets have to be custom made which is extremely time consuming and expensive.

In the project at hand, we strive to formulate an automated end-to-end pipeline for: automated data set generation; text bounding box detection; pixel-level text masks and text recognition. By doing this, we aim to selectively extract pixels classified as text within map images. As a result, this will minimise the extent of changes made to the image as a consequence of removing the text. Consequently, this will minimise the need to repair the image following text extraction and will form a final image suitable for the vectorization process. To achieve this, we are utilising historic maps of Edinburgh, kindly supplied by the National Library of Scotland. In this project, we focus specifically on the 1892-1914 OS 25-inch historic maps, each map being of roughly 16800×12300 in size with associated georeference metadata. This georeference metadata contains the degrees, minutes, seconds (DMS) coordinates for all four corners of the historic maps, along with their centers. An example of using this information to visualise a historic map using modern day satellite imagery can be seen in figure 2.

(a)

(b)

Figure 2: (a) Historic map of Leith. (b). Used the georeference from the historic map of Leith in combination with the Google GMaps API to overlay the four corners (as seen as red markers) of the historic map onto a modern day satellite image of Leith, Edinburgh.

2 Digital Images and Image Pre-processing

Digital images are the means of storing and manipulating an image on a computer. They are a representation of an image often in the form of a grid of pixels each with a colour intensity value [2]. Digital images are often pre-processed before being inputted into an algorithm to perform automated analysis. Pre-processing is a way of standardizing across sets of images. This can be useful as image pre-processing can help a variety of different deep learning models to effectively learn the desired features, such as text within maps. Pre-processing can be done via removing image noise, reducing the number of distinct colours present in a digital image or even reducing the number of colour channels.

2.1 Digital Representation of Images

A digital image, such as a historic map, on a computer is often represented by a matrix of size $W \times H \times C$, where W & H are the width and height of the digital image respectively, and C is the number of colour channels. An example of the separate colour channels of a Red Green Blue (RGB) image can be seen in figure 3. Each element in the digital image matrix represents a pixel intensity value as an integer in the range $[0, 2^k - 1]$, where 0 signifies no intensity (or black) and $2^k - 1$ signifies full intensity [2]. Using this representation, the image can be represented digitally by k bits storing the intensity values for each pixel using a binary representation for each colour channel. In an RGB colour space, using three 8-bit RGB pixels, also known as 24-bit colour, allows the option of $2^8 \cdot 2^8 \cdot 2^8 = 2^{24} = 16,777,216$ distinct colours.

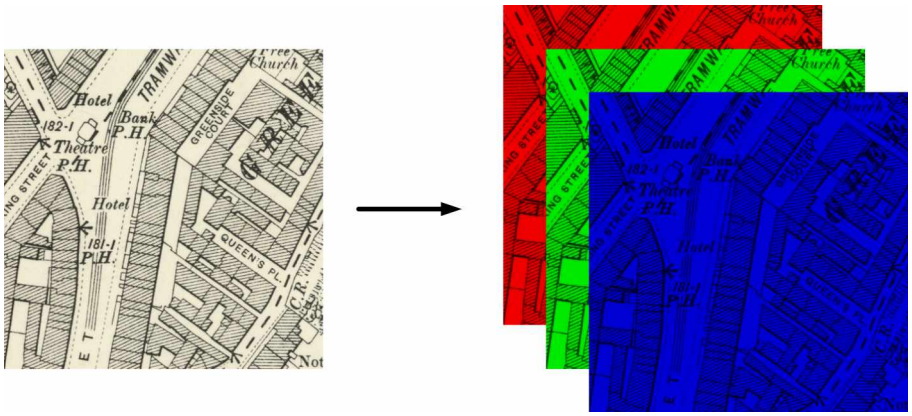


Figure 3: The separate RGB colour channels of a digital colour image of a map.

2.2 Image File Formats

There are many ways to store a digital image, such as a map, on a computer, depending on the use of the image [2]. The historic maps we are working with in this project use the uncompressed Tagged Image File Format, or TIFF, file format. More specifically, the historic maps are saved using a GeoTIFF file format. This file format contains the extra metadata necessary to locate the map's position on the globe, such as the geolocation coordinates and coordinate system. Other file types include PNG and GIF which both use lossless compression. They differ in that GIF is limited to 256 distinct colours, whereas PNG, much like a TIFF file, can handle up to 16.7 million. Another popular choice is JPEG. JPEG offers the ability to use and control the amount of lossy compression. Lossy compression can be used to greatly reduce the size of an image file at the cost of some loss of detail.

2.3 Colour Quantisation

Colour quantisation is the process of reducing the number of distinct colours in a digital image, whilst still retaining visual similarity to the original image [2]. A common application of colour

quantisation is digital image compression [54]. This can be achieved by combining similar pixel values. This reduces the number of distinct colours in an image which each require encoding. For example, in 24-bit RGB colour space [84,15,157] and [85,15,156] represent different colours, but these will be very hard to distinguish visually. Reducing the need to store many colours which are nearly indistinguishable allows an image to be stored using less memory with little impact on the image's appearance.

Interestingly, the reduction of distinct colours present in an image can have further uses than simply reducing file size. Particularly, in the case of historic maps we might choose to utilise colour quantisation to reduce the size of the colour palette present in the digitised images. As seen in figure 4, there exists a wide range of distinct but very similar colours. Unifying some of these colours may result in us being able to more easily classify image features as well as tackle issues such as pixelation. In the initial construction of the map a small colour palette would have been used, and many of the subtle colour variations captured in the digital image of the map are not intended by the map's maker but products of small variations in printing, ageing or digital imaging of the original map. This presents us with a further reason why it might be desirable to use colour quantisation to reduce the colour palette in the digital map image.

Typically, in colour quantisation, we try to group a set of colours together based on their position in a 3-dimensional RGB colour space. Therefore, the most standard approach is to treat colour quantisation as a clustering problem.

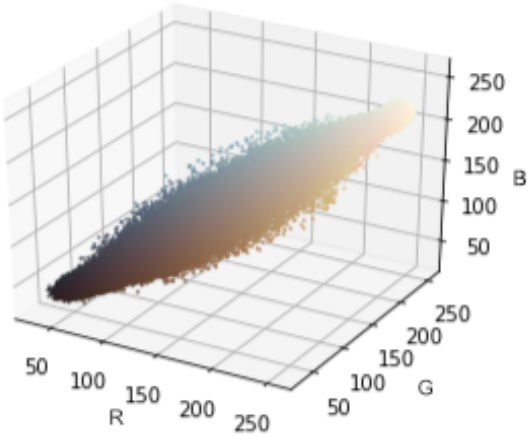


Figure 4: RGB colour space of a 3200x3200 sampled image of a historic map.

2.3.1 K-Means Clustering

K-means clustering is a type of unsupervised machine learning algorithm [21]. The algorithm minimises the distance between the different clusters and their centroids, denoted as the set $\{\mu_1, ..., \mu_K\}$. K-means clustering achieves this by randomly selecting K points to be the starting centroids and then grouping points based on their distance to the closest centroid. New centroids are then calculated by averaging the pixels in the cluster. This process is iterativerly repeated until there is no change between the newly calculated centroids and the previous centroids.

Suppose $\{x^{(1)}, ..., x^{(m)}\}$ is a set of pixel points in n-dimensional colour space, where $x^{(i)} \in \mathbb{R}^n$ is the pixel intensity vector for some point i and $m \in \mathbb{Z}$ is the number of pixels. Then

mathematically we can describe k-means clustering as trying to minimise the objective function

$$J = \sum_{i=1}^{m} \sum_{k=1}^{K} w_{ik} ||x^{(i)} - \mu_k||, \tag{2.1}$$

where

$$w_{ik} = \begin{cases} 1, & \text{if } k = argmin_j ||x^{(i)} - \mu_j||^2 \\ 0, & \text{otherwise} \end{cases}$$

and

$$\mu_k = \frac{\sum_{i=1}^{m} w_{ik} x^{(i)}}{\sum_{i=1}^{m} w_{ik}}.$$

Noting that $\sum_{i=1}^{m} w_{ik} \neq 0$ for any $k \in K$, since each cluster cannot be empty. An algorithmic implamentation of this can be seen in algorithm 1.

K-means functions by iteratively partitioning a group of points into K distinct sub-groups or clusters. It is therefore beneficial if the number of clusters is known prior to training the algorithm. On some images this can be easily noticed. However, if the number of clusters is not known then a hyperparameter search or trial and error approach may be utilised to find the ideal number of clusters.

Algorithm 1: KMeans Clustering Algorithm

Input : $\mu \in \mathbb{R}^{K \times n}$, $x \in \mathbb{R}^{m \times n}$
Output: $\mu \in \mathbb{R}^{K \times n}$

```
randomly select K data points to be μ_1, ..., μ_K
initialise μ̂ ∈ R^{K×n} \ {μ}
while μ̂_k ≠ μ_k   ∀k ∈ K do
    set μ̂ = μ
    initialise w = 0 ∈ R^{(m×K)}
    foreach i ∈ m do
        set k = argmin_j ||x^(i) − μ_j||^2
        set w_ik=1
    end
    foreach k ∈ K do
        set μ_k = (Σ_{i=1}^m w_ik x^(i)) / (Σ_{i=1}^m w_ik)
    end
end
```

2.4 Image Binarization

An alternative to colour quantization is binarization. Binarization can be used to convert k-bit grayscale images (images with a single colour channel) into 1-bit binary images. This not only greatly reduces the digital image file size, but is also extremely effective at differentiating between objects, such as text and background pixels.

2.4.1 Converting to Grayscale

Before we can apply binarization to a digital colour image, we must first convert the image into grayscale. This can be achieved by applying a grayscale function mapping $\mathcal{G}$ to the digital image. This function mapping can be defined as $\mathcal{G}\colon \mathbb{R}^{w \times h \times c} \longrightarrow \mathbb{R}^{w \times h}$ where w and h are the width and height of the digital image, respectively, and c is the number of colour channels [20]. We assume that our image is represented in RGB colour space, $c = 3$. There are a number of

commonly used methods we can utilise to convert the colour image into grayscale.

The simplest method to convert a RGB colour image into grayscale is the *Intensity* method [20]. This method creates new grayscale pixel intensity values using the average of each pixel's colour channel values. This can be defined as

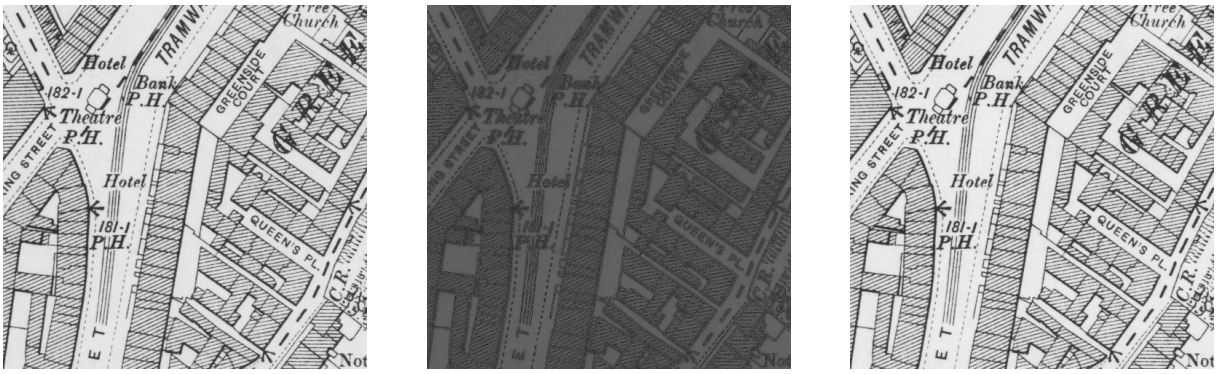
$$\mathcal{G}_{\text{Intensity}} \leftarrow \frac{1}{3}(R + G + B). \tag{2.2}$$

Another method is called the *Lightness* method. This method functions by averaging the most and the least dominant colour channel pixel. This method works best if colour channels are fully saturated [20]. This can be defined as

$$\mathcal{G}_{\text{Lightness}} \leftarrow \frac{1}{2}(\max(R, G, B) + \min(R, G, B)).$$

One of the most frequently used methods for grayscale conversion is the *Luminosity* [35] method. This method accounts for the human eye being less sensitive to visible light which is comprised of longer (redder) or shorter (bluer) wavelengths. The optimal wavelength of colour recognition by the human eye sits at around a yellow-green colour [35]. To adjust for this, *Luminosity* uses a weighted average to give more priority to the channels that the human eye is more receptive to. This is defined as

$$\mathcal{G}_{\text{Luminosity}} \leftarrow 0.212R + 0.7152G + 0.0722B.$$



(a) (b) (c)

Figure 5: (*a*) *Intensity*, (*b*) *Lightness* and (*c*) *Luminosity* grayscale methods applied to the map image in figure 3.

2.4.2 Manual Threshold Identification

Once an image has been converted to grayscale, it can then be converted to a binary image via manual threshold identification. During manual threshold identification, a pixel brightness threshold is chosen by eye based on its adequate separation of features from the background of an image. All of the pixel values that lie above this threshold are set to 1 and the rest are set 0. By choosing an effective threshold, we can often highlight interesting features, such as buildings and text, from undesired image noise.

There are a variety of categories of thresholding methods. Some categories include *Local*, *Clustering*-based, *Entropy*, *Spatial* methods [45]. We however, focus on a *histogram*-based approach.

First, consider a k-bit grayscale image $I \in \mathbb{R}^{m \times n}$ where $I(u, v) \in [0, 2^k - 1]$. Then the image

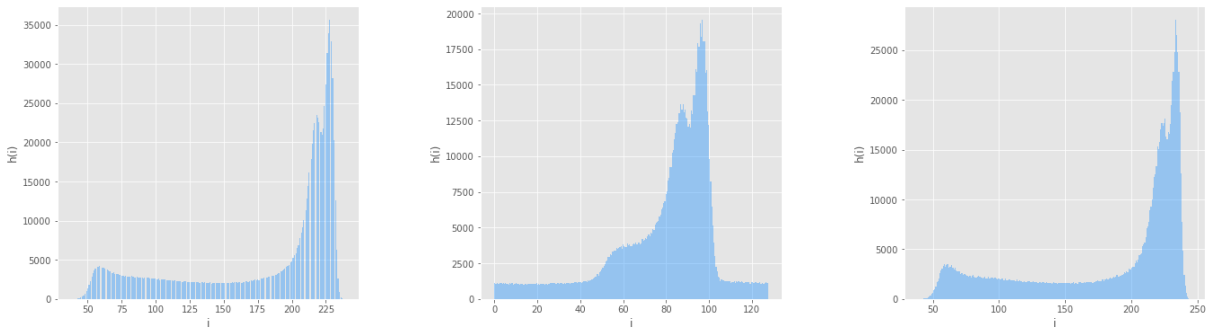
pixel histogram, h, is defined as

$$h(i) = |\{(u,v)|I(u,v) = i\}| \tag{2.3}$$

for all $0 \leq i < 2^k = K$ [2] [1]. To convert our image I into the desired binarised image $B \in \{0 \cup 1\}^{m \times n}$ we must choose a threshold value T such that

$$B(u,v) = \begin{cases} 1, & \text{if } I(u,v) > T \\ 0, & \text{otherwise} \end{cases} \tag{2.4}$$

$\forall u, v \in I$. The choice of this threshold is imperative to limiting information loss. It can be seen from figure 6 that it is not always straight forward which threshold value should be chosen, especially if the distribution is not bimodal.



(a) (b) (c)

Figure 6: Corresponding histogram plot of the figures in figure 5 with $K = 256$ bins for (a) *Intensity*, (b) *Lightness* and (c) *Luminosity* grayscale methods.

2.4.3 Otsu Method

Ideally we don't want to have to manually choose the most appropriate threshold. One method to help tackle this problem is Otsu's method [31]. This method aims to automatically find the threshold that minimises the intra-class variance, σ_W^2, of two partitions of an image histogram. This is defined by

$$\sigma_W^2(t) = q_1(t)\sigma_1^2(t) + q_2(t)\sigma_2^2(t) \tag{2.5}$$

where

$$q_1(t) = \sum_{i=0}^{t-1} p(i), \quad q_2(t) = \sum_{i=t}^{K-1} p(i), \quad \mu_1(t) = \sum_{i=0}^{t-1} \frac{ip(i)}{q_1(i)}, \quad \mu_2(t) = \sum_{i=t}^{K-1} \frac{ip(i)}{q_2(i)},$$
$$\sigma_1^2(t) = \sum_{i=0}^{t-1} [i - \mu_1(t)]^2 \frac{p(i)}{q_1(i)}, \quad \sigma_2^2(t) = \sum_{i=t}^{K-1} [i - \mu_2(t)]^2 \frac{p(i)}{q_2(i)}$$

for some threshold $t \in [0, K-1]$ and $p(i)$ is the probability of pixel value i being chosen at random. From this, we can calculate the threshold by computing σ_w^2 over all values of t whilst recording the minimum intra-class variance. A more efficient method, however, is to instead define and use the between class variance, σ_B^2, as

$$\sigma_B^2 = \sigma^2 - \sigma_W^2 = q_1(\mu_1 - \mu)^2 + q_2(\mu_2 - \mu)^2 \tag{2.6}$$

[1] $|.|$ denotes the cardinality of the set

[31] then substituting $\mu = q_1\mu_1 + q_2\mu_2$ in equation (2.6) and use the fact that $q_1 + q_2 = 1$ to simplify. From this, we obtain

$$\sigma_B^2 = q_1 q_2 [\mu_1 - \mu_2]^2 \tag{2.7}$$

where σ^2 and μ are the variance and mean of the image histogram, respectively. Since σ^2 is constant, from equation (2.6) we can see that minimising σ_W^2 is equivalent to maximising σ_B^2 which is far more efficient to compute. This maximisation gives us the desired threshold value for the image binarisation.

Otsu's method has some clear limitations however. It assumes that the image histogram is bimodal, it therefore works best when there is a strong contrast between the image background and its content. Because of this, Otsu's method is less applicable on images with varying illumination and results can suffer when the two optimal classes are significantly different.

2.5 Removing Noise using Filters

Noise in digital images is often produced by the image sensors utilised to digitise the image. Image sensors often cause random pixels of varying brightness. A common approach to rectify this is to assume that the true values of these pixels are similar to the pixel values of the neighbouring pixels. Under this assumption, a filter is applied to the image via an operation called convolution.

2.5.1 Linear Convolution

A convolution is a mathematical operation, usually denoted $(*)$, which takes two functions of the same dimensionality and returns a single function [2]. In image processing, these two functions are usually an input image $I \in \mathbb{R}^{w \times h \times c}$ and a filter, or kernel, $H \in \mathbb{R}^{n \times m \times c}$ where w & n are the widths of the input image and kernel, respectively, whilst h & m are their respective heights, and c is the number of colour channels. Once the convolution operation has been applied, each pixel (x, y) in the new image $I' \in \mathbb{R}^{(w+n-1)\times(h+m-1)\times c}$ is the sum of the product of a kernel matrix and a subsection of the image matrix. The subsection of the image matrix must be the same size as the kernel and is centered at the point (x, y). We can define the matrix convolution as

$$I'_{(x,y,c)} = \sum_u \sum_v I_{(u,v,c)} H_{(x-u+1,y-v+1,c)}$$

where u and v range over all legal subscripts for $I_{(u,v,c)}$ and $H_{(x-u+1,y-v+1,c)}$ [23, 2]. An illustration of this can be seen in figure 7. Intuitively, the matrix convolution function can be seen as measuring the amount of overlap between two images [23].

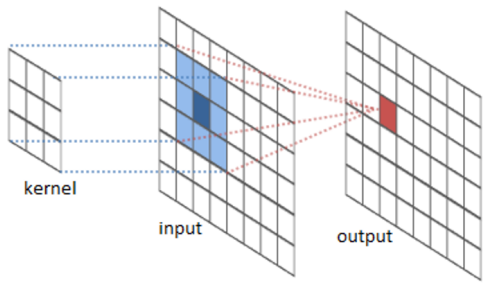


Figure 7: Visualisation of matrix convolution. This shows the kernel sliding over the input image. The resulting output image pixel is the sum of the element wise multiplication at each kernel position [51].

2.5.2 Gaussian Blur

An example of a specific type of filter used to remove noise in a digital image, which can be applied by linear convolution, is the Gaussian blur filter. This filter uses a two dimensional Gaussian function defined by

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{2.8}$$

where x and y are the center of the Gaussian distribution and σ is its standard deviation. This gives higher weightings to pixels located near the new convolved pixel. An example of a Gaussian filter could be the 5×5 normalised Gaussian Blur kernel with $\sigma = 0.8$ and centered at $[3, 3]$:

$$H_{blur} = \begin{bmatrix} 0.0004809 & 0.0050111 & 0.0109454 & 0.0050111 & 0.0004809 \\ 0.0050111 & \mathbf{0.052217} & 0.1140541 & \mathbf{0.052217} & 0.0050111 \\ 0.0109454 & 0.1140541 & \mathbf{0.249117} & 0.1140541 & 0.0109454 \\ 0.0050111 & \mathbf{0.052217} & 0.1140541 & \mathbf{0.052217} & 0.0050111 \\ 0.0004809 & 0.0050111 & 0.0109454 & 0.0050111 & 0.0004809 \end{bmatrix}$$

It can be seen from this matrix that the largest value of the kernel is in the center, with the element values symmetrically decreasing as we move away from this point. Also, note that we must normalise the kernel by dividing each element by the sum of all the kernel elements. This ensures that the average pixel in the convolved image will be as bright as the average pixel in the unmodified image. If the kernel summed up to more/less than one, then this would have the same effect as applying a Gaussian Blur filter followed by a brightness/darkness filter.

Other than removing noise from digital images, Gaussian Blur filters can also be used to enhance images at different scales. We can achieve this by viewing smaller scale details in images as noise we wish to remove and thus drawing more attention to objects at larger scales. As seen in figure 8.

(a) (b)

Figure 8: A 21×21 Gaussian Blur filter with $\sigma = 5$ applied to a 487×386 subsection of a historic map of Edinburgh.

2.5.3 Median Filter

Median filter is another example of a filter, often used to also remove noise from an image. As opposed to a Gaussian filter, a median filter is non-linear. A benefit to using median filters is that it removes noise whilst keeping edges sharp. For this reason, it is a powerful tool when trying to remove salt-and-pepper noise, or speckle noise. Salt-and-pepper noise, or speckle noise, is the occurrence of sparsely distributed black and white pixels in a digital image. The application of a median filter is similar to that of a convolution filter over an image. However, instead of taking the product of the image window with a kernel, we simply take the median value of pixels contained in the window. This median value then becomes the pixel value in the new image.

3 Artificial Neural Networks

In this section we will introduce the concept of Artificial Neural Networks (ANNs) as a stepping stone towards the topic of Convolutional Neural Networks (CNNs) which we later heavily rely on for text detection in our historic maps.

3.1 The History of Artificial Neural Networks

The story of artificial neural networks famously began in 1943, when neurophysiologist, Warren McCulloch and logician, Walter Pitts released a paper proposing a model for how neurons function [29]. They theorised that neuron function can be explained by algorithms called threshold logic. Following this proposal, a large amount of research has been published building on McCulloch and Pitts' work. A further significant publication, by Frank Rosenblatt in the late 1950's, described the development of the perceptron; the oldest type of neural network still in use today. As seen in figure 9, the perceptron passes a weighted sum of inputs through a step activation function to return a binary value. This was found to be useful in the classification of different data points. Research in the field slowed down soon after this, and in 1962 it was proven that the perceptron was incapable of some simple logic operators [34]. Specifically, the perceptron was unable to to apply the XOR logic operator. This is because the resulting output classes of the XOR are not linearly separable. It was eventually realised that this problem could be overcome by adding another layer of perceptrons. However, a new problem shortly came to light, involving how to efficiently produce a set of weights for the network. This was solved in the 1986 paper entitled "Learning representations by back-propagating errors" [42] by David Rumelhart and Geoffrey Hinton where they suggested using a differentiable activation function, and thus the networks could be trained via gradient descent. Due to the computational power required to solve these networks and the limited computer power available at the time, research progressed slowly until the 2000's. Since then, with the tremendous increase of computational power and surge in access to readily available data, the field of deep learning has experienced dramatic growth.

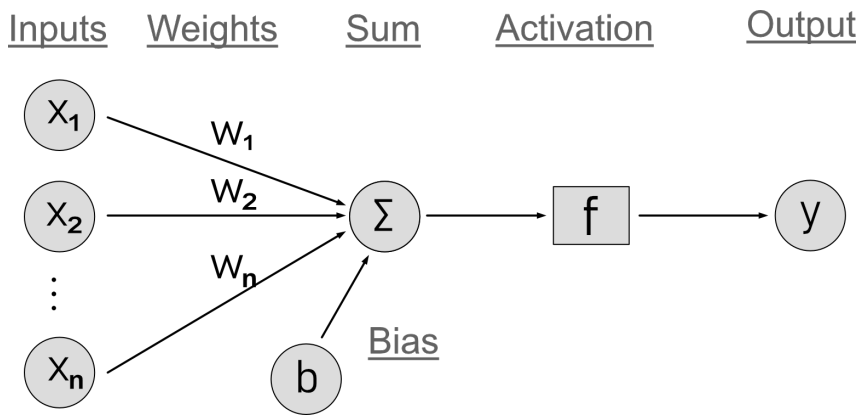


Figure 9: Diagram of a single-layer perceptron.

3.2 Artificial Neural Networks Architecture

Building on top of the single-layer perceptron, the architecture of the modern day "vanilla" neural network is comprised of:

- **Input layer:** Point of entry for the data you wish to pass through the neural network. The number of input neurons depends on the shape of the data.
- **Hidden layer(s):** Located between the input and output layers of the network, the hidden layers take in weighted inputs and produce an output by passing the weights through an activation function.

- **Output layer:** The outputs from the previous layer final layer of neurons are weighted and passed through a final activation function to produce an output for the model.

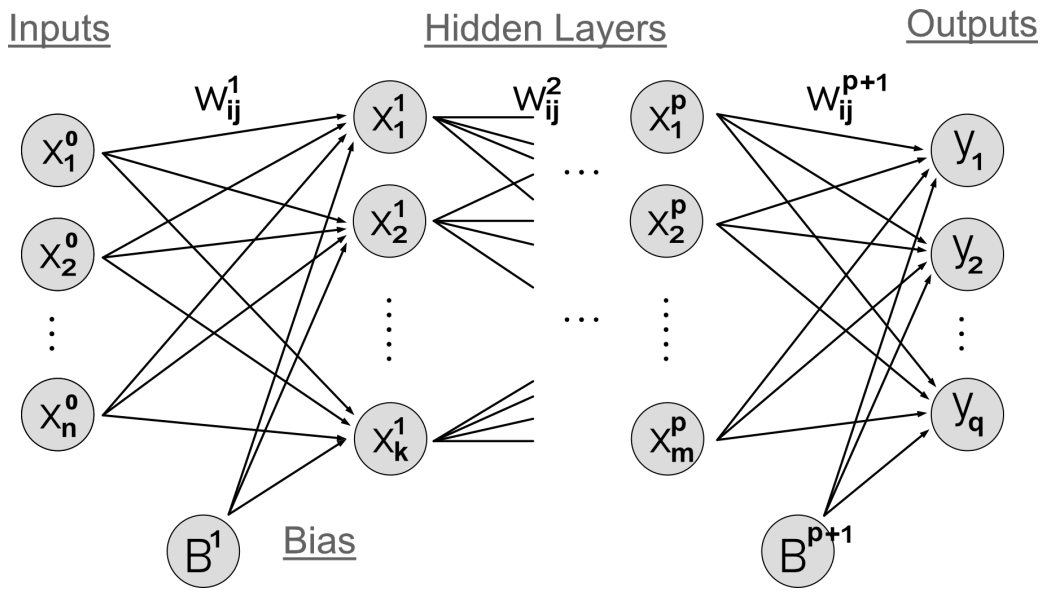


Figure 10: Artificial neural network or multilayer perceptron with P hidden layers, $P+1$ weight matrices and bias vectors, n inputs and q outputs.

3.3 Forward Propagation

Forward propagation is the process of propagating the input data through the layers of the network to obtain an output. For the single-layer perceptron, we formulate our data in the form of an input vector $X = [x_1, x_2, ..., x_n]^T \in \mathbb{R}^n$. The dot product is then taken with X and some weight vector $W = [w_1, w_2, ..., w_n]^T \in \mathbb{R}^n$ to obtain the pre-activation value $z \in \mathbb{R}$. This is defined as

$$z = \sum_{i=1}^{n} w_i x_i + b = W^T X + b \tag{3.1}$$

where $b \in \mathbb{R}$ is some bias value. This is a trainable constant which allows the networks to move the activation function left or right. The pre-activation value is then passed through some, usually non-linear, activation function, f, to obtain the final output

$$y = f(z). \tag{3.2}$$

Plain artificial neural networks are simply multiple perceptrons stacked together via fully connected layers, (each neuron in layer m is connected to each neuron in layer $m+1$ via some weight) as seen in figure 10. This makes the process of forward propagation very similar to that of the single-layer perceptron, but instead of a single weight vector and bias value we now have a weight matrix and bias vector

$$W = \begin{bmatrix} w_{1,1} & w_{1,2} & \dots & w_{1,k} \\ w_{2,1} & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ w_{n,1} & \dots & \dots & w_{n,k} \end{bmatrix} \in \mathbb{R}^{n \times k}, \quad B = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_k \end{bmatrix} \in \mathbb{R}^k$$

where k is the number of neurons in the following hidden layer. Similarly to in the perceptron, we now compute the matrix vector product

$$Z^{(m)} = \begin{bmatrix} \sum_{i=1}^{n} w_{i1}^{(m)} x_i^{(m-1)} + b_1^{(m)} \\ \vdots \\ \sum_{i=1}^{n} w_{ik}^{(m)} x_i^{(m-1)} + b_k^{(m)} \end{bmatrix} = W^{(m)T} X^{(m-1)} + B^{(m)} \tag{3.3}$$

which is then passed through an activation function

$$X^{(m)} = f(Z^{(m)}) = \begin{bmatrix} f(z_1^{(m)}) \\ f(z_2^{(m)}) \\ \vdots \\ f(z_k^{(m)}) \end{bmatrix} \tag{3.4}$$

to obtain the outputs for that layer, $X^{(m)} \in \mathbb{R}^k$. This is taken as the input vector for the next layer with a new trainable weight matrix and bias vector and then propagated forward through the network until we reach the final output(s), Y, for the model. The output shape depends on the predefined architecture of the network. If this were a simple binary classifier model, the output would be a single output neuron returning a value between 0 and 1 for the predicted class.

3.4 Activation Functions

The activation function is a way of mathematically mimicking the activation potential of neurons in the brain; it is responsible for determining whether specific neurons fire. Activation functions are usually always non-linear. Non-linearity is crucial for networks to be able to learn the mappings of complex non-linear data. Without introducing non-linearity, no matter how many hidden layers you add to the neural network, the overall model would be linearly equivalent to that of a single-layer perceptron.

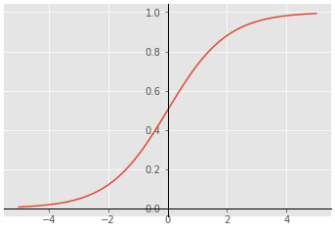
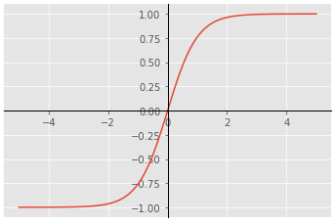
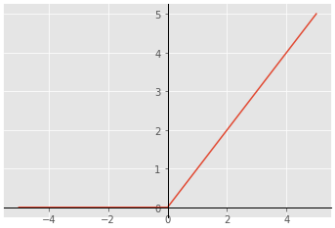
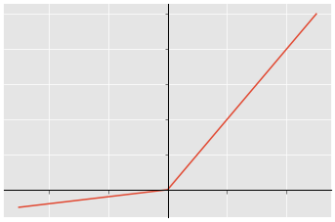
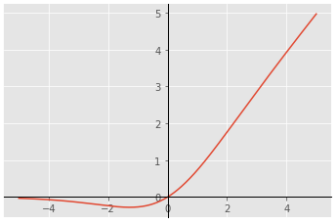
There are several commonly used activation functions to consider when creating a deep learning model, each with their own advantages and disadvantages [30, 28]. A popular selection can be seen in table 1. The Rectified Linear Unit (ReLU) activation function, as seen in table 1, is currently one of the most prominently used activation functions. It has situational advantages over activations such as the Sigmoid and Hyperbolic Tangent (tanh) functions because of its ability to introduce sparsity and tackle an issue called the vanishing gradient problem associated with the aforementioned functions. The vanishing gradient problem is a problem that affects the weights in earlier layers in a network causing them to change incredibly slowly, or in worst cases not at all. This originates from the way we train the networks via a process called backpropagation, discussed in section 3.7. In short, how we change the weights in the earlier layers depend on gradients in the later layers. With activation functions such as Sigmoid and tanh, as they approach their asymptotes, the gradient at these points approach zero. Consequently, multiplying several of these gradients together as we propagate backwards throughout the network results in vanishingly small gradients to train the weights with. ReLU, on the other hand, has a constant gradient which helps tackle this problem at the cost of introducing another possible problem of exploding gradients.

Despite its positive attributes, one issue with ReLU, known as the "dying ReLU" problem, allows for the possibility of all gradients becoming zero. Due to this, the network cannot update and thus the network dies. Variants of ReLU help to solve this problem. Variants include those such as Leaky ReLU and Parametric ReLU. Leaky ReLU utilises $f(x) = 0.01x$ when $x < 0$ instead of 0 in the case of ReLU. Similarly, Parametric ReLU utilises $f(x) = \alpha x$ when $x < 0$ for some constant α. Both variants rectify the 'dying ReLU' problem by removing the zero part of

the function, at the cost of losing the ability to introduce sparsity.

One of the more recently proposed activation functions to tackle the vanishing gradient problem and the dying ReLU problem is the Swish function [36], table 1. Its asymptotic convergence as $x \to -\infty$ allows large negative weights to become zero whilst still retaining some information in smaller negative weights. Similarly to ReLU, Swish is unbounded from above. However, unlike ReLU, Swish is a smooth function which results in a smooth gradient landscape which can aid convergence. Swish was shown to improve some top-1 [2] classification accuracies by $0.6 - 0.9\%$ [36].

Table 1: A table of different activation functions including their equations, derivatives, function ranges and equation plots.

Name	Plot	Equation	Derivative	Range
Sigmoid (Logistic)		$f(x) = \frac{1}{1+e^{-x}}$	$f'(x) = f(x)(1 - f(x))$	$(0, 1)$
Tanh		$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	$f'(x) = 1 - f(x)^2$	$(-1, 1)$
ReLU		$f(x) = \max(0, x)$	$f'(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$	$[0, \infty)$
Leaky ReLU		$f(x) = \begin{cases} 0.01x & x < 0 \\ x & x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0.01 & x < 0 \\ 1 & x \geq 0 \end{cases}$	$(-\infty, \infty)$
Swish		$f(x) = \frac{x}{1+e^{-x}}$	$f'(x) = \frac{1+e^{-x}+xe^{-x}}{(1+e^{-x})^2}$	$[-0.278, \infty)$

3.5 Loss vs Cost functions

As we will shortly describe, most neural networks train via gradient-based optimization. This involves adjusting the trainable parameters of the network based on the current performance of

[2] Top-n accuracy is when the correct class label appears in the n top predicted labels.

the model. Quantifying how close the model predictions are to the true output is done by the use of a cost function. By minimising this cost function, we aim to reduce the errors between the predicted and true values. Sometimes, this function is instead called the loss function.

In the field of deep learning, the loss and cost functions often get used interchangeably [13]. In this thesis we will also do the same. It is worth noting however, that in some publications the term loss function is used as the measure of penalty between a single predicted label and the true output. Whereas, the term cost function is used as the total penalty (or cost) associated with all the predicted labels and the true outputs. For example, under this definition, a common loss function used in linear regression is the squared error loss which can be defined as

$$l(y_i, y_i^p) = (y_i - y_i^p)^2 \tag{3.5}$$

where y_i is the expected true outcome and y_i^p is the prediction from the model. Whereas the cost function could be defined as the mean squared error

$$C(y, y^p) = \frac{1}{n}\sum_{i=1}^{n}(y_i - y_i^p)^2 \tag{3.6}$$

where n is the total number of labels.

3.6 Gradient Decent

In order to train a neural network via backpropagation, we must first understand how to minimise a cost function. We can do this by utilizing an optimization algorithm called gradient descent. Gradient descent is a first order numerical method for finding the location of a local minimum of a differentiable function. Starting at an initial guess for the location of a local minimum, gradient decent attempts to find the true location of the local minimum by using the derivative of the function to iteratively take steps in the direction of the negative gradient. Therefore iteratively stepping 'down hill' to a minimum. This process is shown diagrammatically in figure 11. We can define gradient descent as

$$x_{n+1} = x_n - \gamma \nabla F(x_n) \tag{3.7}$$

where x_n is some discrete point in the function space F and γ is a constant called the learning rate. It can be seen in figure 11 that the speed and type of convergence of the gradient descent algorithm is heavily dependent on the learning rate. If the learning rate is too low, then convergence can be incredibly slow and inefficient. If it is too fast, then we may not converge to the local minimum, or even worse, not converge to anything at all. To help combat this, we may choose to use a scheduled or adaptive learning rate. For example, we might choose to use a larger learning rate at the beginning, when we are far away from a minimum, and then gradually decrease the learning rate overtime to aid convergence. This is called a step schedule scheme.

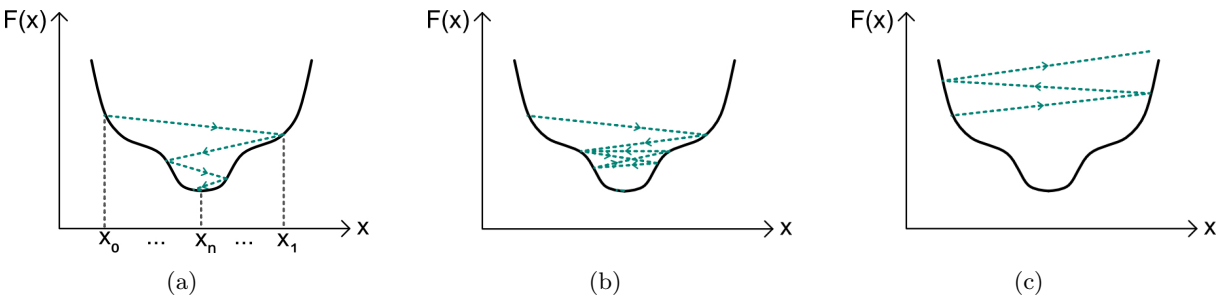


Figure 11: Examples of gradient descent with varying learning rates. (a) Ideal learning rate where method converges quickly to the local minimum. (b) Slightly too high learning rate and the method converges but not to the local minimum. (c) Too high learning rate and the method starts to diverge.

There are many ways to implement a learning rate method, but the most common types are:

Time-Based Decay: Decreases the learning rate at each iteration depending on the learning rate in the previous iteration. The equation for this is

$$\gamma_{n+1} = \frac{\gamma_n}{1 + dn}$$

where γ is the learning rate, d is the decay rate and n is the current iteration number.

Step-Based Decay: Decreases the learning rate at each predefined number of iterations. The formula for this scheme is given by

$$\gamma_n = \gamma_0 d^{\lfloor \frac{1+n}{r} \rfloor}$$

where γ_0 is the initial learning rate, d is the percentage decay (0.5 corresponds to halving the learning rate each time), r is the decay rate (5 corresponds to a reduction every 5 iterations) and n is the current iteration number.

Exponential Decay: Exponential decay, as the name suggests, is scheme that decreases the learning rate based on a decreasing exponential function. This can be defined as

$$\gamma_n = \gamma_0 e^{-dn}$$

where d is the decay parameter and n is the current iteration number.

Adaptive Learning Rates: An issue with learning rate schemes is that they are sensitive hyperparameters that have to be predefined at the beginning of each training session. Adaptive learning rates help solve this by adapting to the current state of the session. Current popular adaptive learning rate methods include: RMSProp Optimizer; Adagrad Optimizer; Adam Optimizer.

3.7 Backpropagation

Backpropagation is the process of propagating the predictions of labelled data back through a neural network in order to adjust the trainable parameters, with the aim of minimising the overall error or cost of the network. In order to adjust a trainable parameter, e.g the weight values, such that the cost function is decreased, we need to be able to measure the cost functions sensitivity to these parameters. Mathematically, this can be achieved by finding the gradient of

the cost function with respect to the trainable parameters. This is defined as

$$\frac{\partial C_0}{\partial w_{ij}^{(m)}}, \quad \frac{\partial C_0}{\partial b_j^{(m)}}. \tag{3.8}$$

for some weight, $w_{ij}^{(m)}$, or bias $b_j^{(m)}$ value. How to compute these derivatives is not immediately clear. However, in section [3.3] we can observe that if we made a change in $w^{(m)}$, then this would change the value of $z^{(m)}$ which would in turn change the output of that layer $x^{(m)}$ and thus change the overall cost function. It perhaps becomes more clear that if we take derivatives with respect to the appropriate variables, then we can construct the gradients from the above equation (3.8) via the chain rule. Doing this gives

$$\frac{\partial C_0}{\partial w_{ij}^{(m)}} = \frac{\partial z_j^{(m)}}{\partial w_{ij}^{(m)}} \frac{\partial x_j^{(m)}}{\partial z_j^{(m)}} \frac{\partial C_0}{\partial x_j^{(m)}}, \quad \frac{\partial C_0}{\partial b_j^{(m)}} = \frac{\partial z_j^{(m)}}{\partial b_j^{(m)}} \frac{\partial x_j^{(m)}}{\partial z_j^{(m)}} \frac{\partial C_0}{\partial x_j^{(m)}} \tag{3.9}$$

which can computed given some initial conditions. For example, assuming that we're using a mean squared error cost function, as seen in equation (3.6), and wish to calculate the gradient w.r.t the cost for some weight value $w_{ij}^{(m)}$ in between the final output layer $x^{(m)} \in \mathbb{R}^k$ and hidden layer $x^{(m-1)} \in \mathbb{R}^n$. From equation (3.9) we can compute

$$\begin{aligned}
\frac{\partial C_0}{\partial x_j^{(m)}} &= \frac{\partial \frac{1}{k} \sum_{l=1}^{k} (x_l^{(m)} - y_l)^2}{\partial x_j^{(m)}} = \frac{2}{k}(x_j^{(m)} - y_j) \\
\frac{\partial x_j^{(m)}}{\partial z_j^{(m)}} &= f'(z_j^{(m)}) \\
\frac{\partial z_j^{(m)}}{\partial w_{ij}^{(m)}} &= x_i^{(m-1)}
\end{aligned}$$

since $z_j^{(m)} = w_{ij}^{(m)} x_i^{(m-1)} + b_j^{(m)}$ and $x_j^{(m)} = f(z_j^{(m)})$ for some activation function f. Combining these equations we obtain the desired gradient

$$\frac{\partial C_0}{\partial w_{ij}^{(m)}} = \frac{2}{k}(x_j^{(m)} - y_j) f'(z_j^{(m)}) x_i^{(m-1)}. \tag{3.10}$$

This can be generalised for any weight $w^{(m)}$ as

$$\frac{\partial C_0}{\partial w^{(m)}} = \frac{2}{k} x^{(m-1)} \left[(x^{(m)} - y) \cdot f'(z^{(m)}) \right]^T. \tag{3.11}$$

Building on this, if we instead wish to calculate this derivative for some weight value $w_{ij}^{(m-1)}$ in the second to last layer, then by using equation (3.9) we obtain

$$\frac{\partial C_0}{\partial w_{ij}^{(m-1)}} = \frac{\partial z_j^{(m-1)}}{\partial w_{ij}^{(m-1)}} \frac{\partial x_j^{(m-1)}}{\partial z_j^{(m-1)}} \frac{\partial C_0}{\partial x_j^{(m-1)}}. \tag{3.12}$$

The only thing we have to consider differently in this equation compared to the first, is that the amount $x_j^{(m-1)}$ affects the cost is split up by the weights leading into all the output nodes in layer $x^{(m)}$ rather than leading directly into the loss function as seen previously. An illustration of this can be seen in figure 12. To solve this issue, we apply the chain rule once more and sum

over all connected nodes in $x^{(m)}$ to obtain

$$\frac{\partial C_0}{\partial x_j^{(m-1)}} = \sum_{i=1}^{n} \frac{\partial z_i^{(m)}}{\partial x_j^{(m-1)}} \frac{\partial x_i^{(m)}}{\partial z_i^{(m)}} \frac{\partial C_0}{\partial x_i^{(m)}}. \tag{3.13}$$

Hence, using equations (3.12) and (3.13) we are able to recursively compute the gradient of the cost function w.r.t any and all weight values. A similar method can be applied to the bias vectors or any other trainable parameters in the network. Once we have these parameter gradients, we can use gradient descent to take steps in the direction towards the local minimum and hence calculating the best parameter values that minimise the cost.

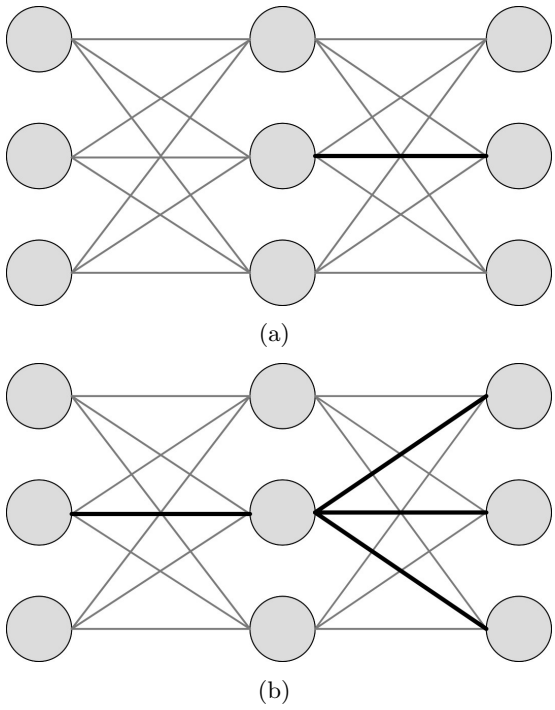


Figure 12: In backpropagation: (a) Each weight in a final layer does not depend on any other weight values. (b) A weight in the second to final layer depends on some of the weights (as seen in bold) in the previous layer.

4 Convolutional Neural Networks

Convolutional neural networks are a subclass of neural networks which contain at least one convolution layer. Rather than learning the pixel-level relationship between the input image and the label as an ANN might, CNNs learn local features of an image by repeatedly applying trainable convolution filters, as seen in section 2.5.1, at different scales. These local features can then be passed through some model, e.g a vanilla neural network, to predict the output label of the class. A major break-through in the field of computer vision happened in the 1990s when Yann LeCun et al. proposed in their paper entitled "Gradient-Based Learning Applied to Document Recognition" [24] the first CNN model. They demonstrated that their CNN model could take simpler input features and produce complex classifiers which were used to successfully recognise handwritten characters. This style of using filters to extract local image features over a vanilla ANN approach is massively more efficient for learning speed, memory consumption and the amount of training data required. It's still possible to use ANNs for some small image classification tasks such as on the MNIST dataset (dataset of handwritten digits created in 1999 by LeCun et al.). However, for larger images it becomes impossible very fast. For example, if you wanted an ANN to take a 1024×1024 sized 8-bit greyscale image as an input, followed by a fully connected hidden layer that is twice the size as the input layer, then this would consume $1024^2 * 2 * 1024^2 * 8 = 17,592,186,044,416$ bits or $16,384$ gigabytes of memory for the first weight matrix alone. Needless to say, for larger images, such as the historic maps utilised in this project, using a vanilla ANN alone is an infeasible solution.

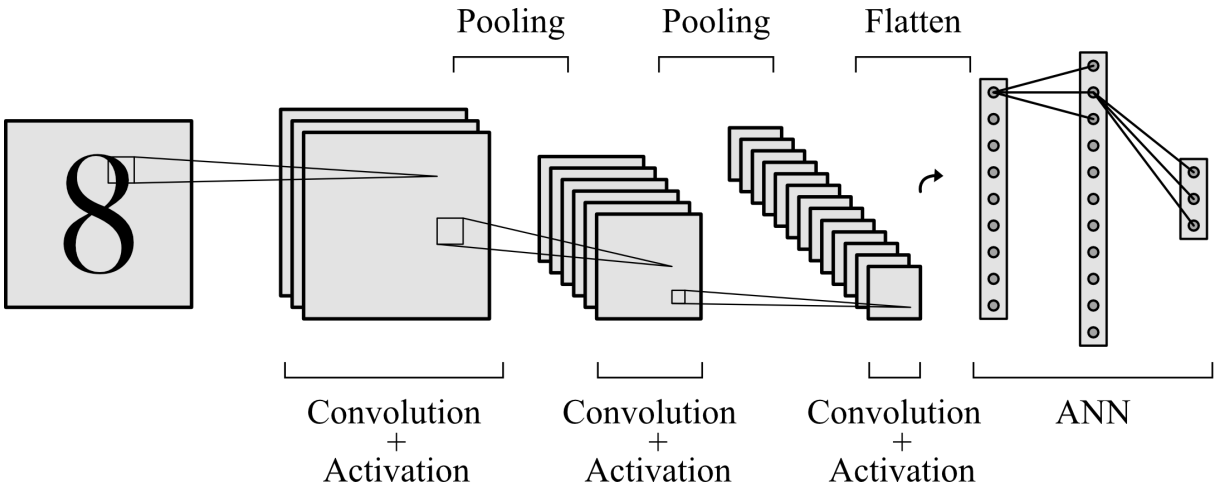


Figure 13: A diagram an example convolutional neural network.

4.1 CNN Layers

The type of architecture used to construct convolutional neural networks is highly dependent its purpose. There are a wide range of different layers commonly used, each with their own use. In this section we will discuss a small, but important, selection of these layers which we later utilise to construct some of our own deep learning models.

4.1.1 Convolutional Layers

The convolutional layer consists of trainable spatial filters, or kernels, that are convolved along the spatial dimensions of an input multidimensional matrix, or tensor [21]. The convolved points have a bias value added and are then passed through an activation layer, much like a node of a fully connected ANN with a lot fewer connections. The resulting tensor is a collection of feature maps containing specific information of the input tensor conditional on the filter's kernel operation. These filters are trainable parameters which can be optimized through gradient descent

based optimization.

In convolutional layers, multiple trainable filters are passed over an input image to obtain a collection of feature maps. The number of resulting feature maps is a hyperparameter [3] and depends on the number of kernel blocks. For example, suppose we have an input image of size $Width \times Height \times C1$, where $C1$ is the number of colour channels. Then if we pass a 4D tensor of filters of size $C1 \times k \times K \times C2$, where $C2$ is the number of output feature maps and K is the kernel size, we would obtain $Width \times Height \times C2$ features maps (if using same padding).

One problem with applying a simple convolution to an image, as seen in section 2.5.1, is that resulting feature map is smaller than the input image. The size of the resulting feature map can be calculated by

$$\text{Output size}_{\text{width}} = \frac{N_{\text{width}}}{S_{\text{width}} + 1}, \quad \text{Output size}_{\text{height}} = \frac{N_{\text{height}}}{S_{\text{height}} + 1} \tag{4.1}$$

where N and K are the sizes of the input image and kernel, respectively and S is the stride size. Often this change in size is undesirable. One common approach to keep the same feature map size is to apply zero padding to the input image before the convolution is applied. The amount of padding, P, required is given by

$$\text{Output size} = \frac{N - K + 2P}{S} + 1 \implies P = \frac{S(\text{Output size} - 1) + K - W}{2}. \tag{4.2}$$

4.1.2 Activation Layers

Convolutional layers are very similar to fully connected ANN layers except they have a lot less weights. Similarly to fully connected ANN layers, convolutional layers still require the addition of bias value and to be passed through a non-linear activation function to allow them to respond in a non-linear fashion to data. Therefore, activation layers are usually applied after a convolution layer to achieve this. The type of activation function used in the activation layer is often similar to one used in a fully connected ANN layer, as discussed in section 3.4.

4.1.3 Pooling Layers

Pooling layers reduce the spatial dimensions of the input whilst trying to retain import features. They achieve this by passing a filter with a stride over the feature maps, similar to a convolutional layer, and performs an operation which summarises the information in this filter as a single value. An example can be seen in figure 14. Its common to see models having some sort of pooling layer after one or two convolutional layers. This not only reduces the memory requirements and computational complexity of the model, but by combing several values it reduces the likelihood of overfitting and allows the next set of smaller convolution filters to be passed over the images at a higher scale.

There are two main types of pooling uses in convolutional neural networks:

- **Max Pooling:** The maximum of the filter values is taken to be the new pixel value.
- **Average Pooling:** The average of the filter values is taken to be the new pixel value.

One alternative to pooling, however, is simply using strided convolutions. This has the benefit of allowing the model to learn which properties are most import at the cost of increasing the number of trainable parameters. In [47], Jost T. Springenberg et al. demonstrate that max-pooling could be replaced by strided convolutions without loss in accuracy on several image recognition benchmarks.

[3] Parameters that cannot be directly learned during training.

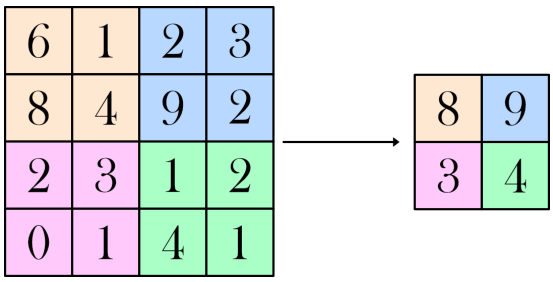


Figure 14: An example of applying a 2×2 max pooling on a 4×4 image matrix.

4.1.4 Normalization Layers

In the 2015 paper "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift" [17], Sergey Ioffe and Christian Szegedy explain that in the training of deep neural networks the weights of a layers are updated given the prior distribution of the weights in the later layers of the network. These distributions, however, have likely changed during that pass of backpropagation, and thus, the training process is forever chasing a distribution which is changing throughout each training pass. To help combat this, [17] introduces the concept of batch normalization to change the distribution of inputs during training. They showed using batch normalization allowed them to use much higher learning rates which in turn allowed them to achieve the same prediction accuracy using a state-of-the-art classification model with 14 fewer training steps.

The equation for normalization can be defined as

$$y_i = \frac{\gamma(x_i - \mu)}{\sigma} + \beta \tag{4.3}$$

where x is the input, y is the output, γ is the scale factor, μ is the mean, σ is the standard deviation and β is the offset factor [49]. Thus, for a set of input values x we can normalize some batch $\hat{x} \subset x$ by finding the mean and standard deviation of $\hat{x}$ and setting the new normalised value x_i^* to be y_i for all $\hat{x}_i \in \hat{x}$.

4.1.5 Dropout Layers

Dropout layers aim to help prevent overfitting. They achieve this by disconnecting a random percentage of the network nodes at some layer for the current iteration. These nodes are temporarily disabled and do not participate in the either forward propagation or the training via backpropagation. In turn, this forces the other nodes to mathematically pick up the slack, preventing a selection of trainable parameters dominating the model predictions and overfitting the model.

4.2 Classification

During the process of object classification, the neural network takes an image input and uses it to predict a class label, or a list of class probabilities for the image. Classification is a type of supervised learning problem. In order to train a classifier, large quantities of data labelled according to desired classes are required. Note, a classifier will only be able to predict labels for classes on which it has been trained. For example, if you trained a dog/cat binary classifier, then inputting a picture of a person would still result in the classifier returning the label of cat or dog.

As described in section 3.5, neural networks require a cost function to train the model relative to its current performance. One of the most common ways to train a classification CNN is by

cross-entropy loss. This is defined as

$$CE = -\sum_{i}^{C} t_i log(s_i) \tag{4.4}$$

where C is the number of classes, s_i is the predicted probability of class i being the true class and t_i is 1 if i is the true class and 0 otherwise. To obtain the predicted probability of each class, the final activation layer uses a sigmoid activation function if the problem is binary, this is defined as

$$s_i = f(x_i) = \frac{1}{1 + e^{-x_i}}. \tag{4.5}$$

otherwise, if the problem is not binary, the softmax activation function is utilised. The softmax function is defined as

$$s_i = f(x_i) = \frac{e^{x_i}}{\sum_j^C e^{x_j}}, \tag{4.6}$$

where x_i is one of C final outputs of the model. Both these activation functions return the probability that each label is the true class. In the case of the sigmoid activation function, because the classification is binary, s_2 is described by $1 - s_1$ and thus only one probability is required to describe the other. For this reason, it is common to see equation (4.4) written as

$$CE_{\text{binary}} = -\sum_{i=1}^{2} t_i log(s_i) = -t_1 log(s_1) - (1 - t_1) log(1 - s_1) \tag{4.7}$$

since $t_1 + t_2 = 1$ and $s_1 + s_2 = 1$.

4.3 Object Localisation

Often in computer vision it is useful to not only classify an image depending on the object within it, but also identify the position of the object. The object's location is frequently encompassed by a bounding box [11]. Although, there are other variations of object identifiers such as ellipses or facial keypoints [52, 33]. It is important to highlight the differences between object localisation and object detection. Object localisation is an extension of a classifier. It predicts the class of an image and returns the coordinates of the object's bounding box. Unlike an object detector, an object localiser will not return multiple object bounding boxes per class.

In order to extend a single label, multi-class classification model to become a bounding box object localisation model, we must first increase the number of output nodes. Typically, this is achieved by adding five more nodes to the output layer of the CNN. One of these five nodes is a binary node, p_c, used to indicate if an object is present in the image or not. The other four nodes are used to define the coordinates of the bounding box. The coordinates that are normally used are (b_x, b_y, x_w, x_h) where b_x, b_y are either the center coordinates or the coordinates of the top-left corner. These coordinates are used instead of the common approach to defining a box in computer graphics, given by $(b_{x_1}, b_{y_1}, b_{x_2}, b_{y_2})$ where b_{x_i}, b_{y_i} are the top-left and bottom-right corners for $i = 1, 2$, respectively. By choosing the bounding box coordinate system given in the first case, this avoids the possibility that the predicted coordinates result in a negative bounding box, for example if $b_{x_1} > b_{x_2}$ (or $b_{y_1} > b_{y_2}$).

After adding the additional five nodes, the output layer for the object localiser becomes

$$y^p = \begin{bmatrix} p_c \\ b_x \\ b_y \\ b_w \\ b_h \\ C_1 \\ \vdots \\ C_n \end{bmatrix} \tag{4.8}$$

where $C_1, ..., C_n$ are the object category predictions. In order to train the network, however, we must define a loss function. Contrary to the categorical loss function, the object localisation loss function requires three separate loss functions for each of the binary & category classifications and the bounding box regression parts of the output [9]. An example of a loss function for these could be to use binary cross-entropy and categorical cross-entrop loss for the classification losses and mean squared error for the bounding box regression loss. These losses are summed together to form the total loss for the model. It is important to note, however, the loss function should not include $L_{\text{categorical}}$ and L_{bbox} when the there are no objects in the image, i.e when $y_0 = 0$. This is because, in this case, we only need to adjust the model based on how well it classifies whether there are no objects in the image. The total loss function could therefore be written as

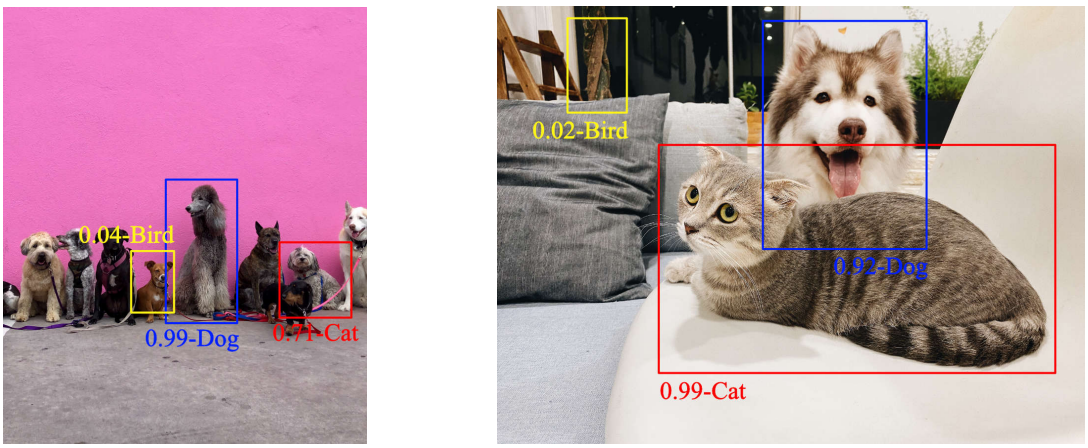
$$L_{\text{total}} = \alpha L_{\text{binary}} + y_0 \big[\beta L_{\text{categorical}} + \gamma L_{\text{bbox}}\big] \tag{4.9}$$

where α, β, γ are hyperparameters. Tuning these hyperparameters shifts the priorities of the CNN loss.

It is possible to extend this model for multiple class labels. This model will return exactly one bounding box per class. For example, exactly one bounding box for the dog class and exactly one for the cat class will be returned, independent of whether either are present [43], figure 15. Extending the model for multiple class labels can be achieved by simply adding a bounding box coordinate set $b_i = [b_{x_i}, b_{y_i}, b_{w_i}, b_{h_i}]^T$ to the output layer for each category, thus giving

$$y^p = \begin{bmatrix} p_c \\ b_1 \\ \vdots \\ b_n \\ C_1 \\ \vdots \\ C_n. \end{bmatrix} \tag{4.10}$$

We must also make a couple of changes to the loss function. Firstly, because we are allowing for multiple labels to be present at once, a softmax activation function no longer makes sense. Instead a sigmoid function is required, which treats each classes' presence in the image as binary. Next, similarly to how we do not include the categorical loss (now binary losses) and regression loss if $y_0 = 0$, we will not include each classes' regression loss unless the class is truly present in the image.



(a) (b)

Figure 15: Two example output images with predicted bounding boxes and probability scores for a multi-label object localiser, with class names: $\{Cat, Dog, Bird\}$

4.3.1 Intersection over Union

As useful as taking the mean squared error of predicted bounding box coordinates is, in many situations it is desirable to have a more understandable metric. A common metric for evaluating the accuracy of the bounding box prediction or pixel-wise mask compared to the ground truth is the Intersection over Union (IoU), also known as Jaccard index. IoU is a metric that calculates the prediction accuracy by computing a ratio of the predicted pixels, P, correctly overlapping the ground truth, G, to the total number of pixels that cover the union of both.

$$IoU = \frac{|P \cap G|}{|P \cup G|} = \frac{|P \cap G|}{|P| + |G| - |P \cap G|}$$

4.4 Object Detection

Object detection is the process of localising multiple objects of a given class in an image [6, 55]. This is an important process for the detection of text within the historic maps utilised in this study.

4.4.1 Non-max Suppression

A common consequence of many object recognition models is that the model will return multiple bounding boxes for the same object. Clearly, this is not desirable. One frequently utilised way of removing the extra bounding boxes is a method called non-max suppression (NMS) [40]. Non-max suppression begins by choosing the bounding box with the highest predicted probability of containing an object. The algorithm then iteratively (or greedily) looks at the rest of the bounding boxes and removes any boxes with an intersection over union greater than some threshold λ_{thresh}. For object detectors with multiple classes, NMS is applied to the predicted bounding boxes for each object class.

Algorithm 2: Non-max suppression algorithm

Input : B, S proposed bounding boxes and scores and IoU threshold λ_{thresh}.
Output: $\hat{B}$ filtered proposed bounding boxes.

```
B̂ = ∅
foreach b_i ∈ B do
    discard = False
    foreach b_j ∈ B do
        if IoU(b_i, b_j) > λ_thresh and S_j > S_i then
            discard = True
            break
        end
    end
    if discard = False then
        B̂ = B̂ ∪ b_i
    end
end
```

4.4.2 Sliding Window

The sliding window method is perhaps one of the most intuitive ways to tackle object detection. To use this method you iteratively slide a window over the whole image, as seen in figure 16. At each window position,the windowed sub-image gets resized and fed into a pre-trained object classifier or object localiser model. The resulting confidence scores and bounding boxes are saved with their associated class and non-max suppression is applied.

It is clear that the method so far is highly dependent on the initial window size. Pick a window that is too small and the model will fail to identify objects at a larger scale. However, if the window is too big, the localisation model might not detect all of the different objects. An effective method for tackling this problem is to pick a smaller sliding window, iterate it through the image, then once finished with the iterations, we shrink the size of the image by some amount and repeat. This method is called image pyramids [3] and will allow the model to detect objects on multiple scales.

Even though the sliding window with image pyramids method can work effectively on some problems, the model's performance will suffer when tasked with finding objects that are very close together. On top of this, computing the forward pass for the model at each iteration for

Figure 16: An example of the sliding window and image pyramid object detection method applied to the Lenna image; a standard test image in the field of image processing

multiple scales is extremely computationally expensive. We can see in figure 16 that with each forward pass of the sliding window, a lot of computational resources are wasted recomputing

large amounts of the same information. To prevent this, we can reformulate the sliding window method as a convolutional implementation [44] which shares this information between regions. This method replaces the sliding window steps for a pooling layer and the fully connected layers in the original bounding box regression for 1×1 convolution layers. Doing this, allows for the whole image to be passed into the CNN at once, allowing it to share the information between windows. This can be seen in figure 17.

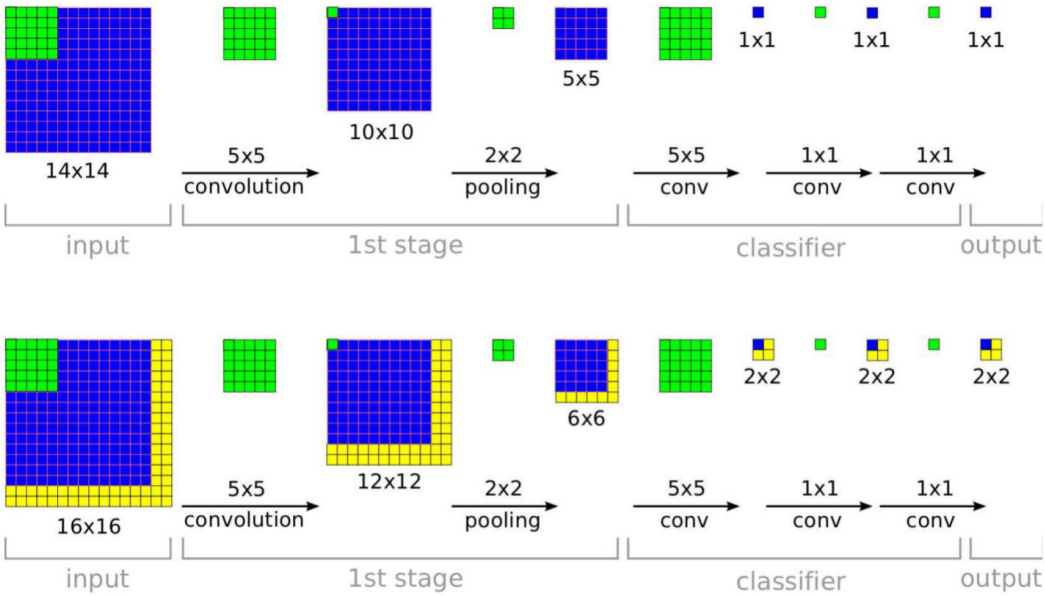


Figure 17: Image taken from [44] demonstrating the ConvNet applied to the sliding window in the first row and then consequentially applied to the whole image. Since all layers are applied convolutionally, the extra computational resources required for the larger image are limited to the yellow pixels [44].

4.4.3 Faster R-CNN

A model known as Faster Region-proposal Convolutional Neural Network (R-CNN) [38], published in 2015, is the third in a series of R-CNN, object detection papers, the previous being R-CNN and Fast R-CNN [12, 10]. An interesting aspect of R-CNN which ignited the idea behind the first publication, was the use of an algorithm called Selective Search which carried out clustering methods on the image in order to propose regions of interest (RoI). These RoI get fed into a CNN where they are classified and a new bounding box is predicted. This is similar to the sliding window approach in section 4.4.2. However, the number of generated RoI (typically $2,000$) is much less than the number required by the sliding windows method. Similar to the sliding window method, the R-CNN method was still very slow and required large amounts of memory to store the region proposals. These issues were addressed in the Fast R-CNN model which used a convolutional implementation of sliding windows, section 4.4.2, to classify the image in one forward pass. Soon after this, Faster R-CNN was proposed, this was the first of the models which was fully differentiable.

The Faster R-CNN architecture uses a trainable Region Proposal Network (RPN) in the network to generate RoI. The RPN takes an image as an input and uses the shared set of convolutional layers to return a group of bounding box object proposals and their associated scores. These predictions then go through a process called region of interest pooling to extract features corresponding to objects. These features are then fed into the classifier which labels the regions and adjusts their bounding box coordinates [21]. This can be seen in figure 18.

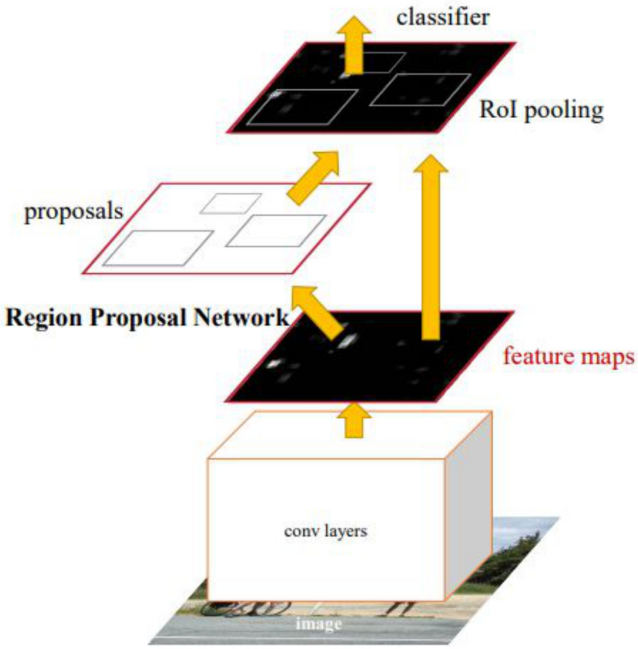


Figure 18: Illustration of the Faster R-CNN architecture [38].

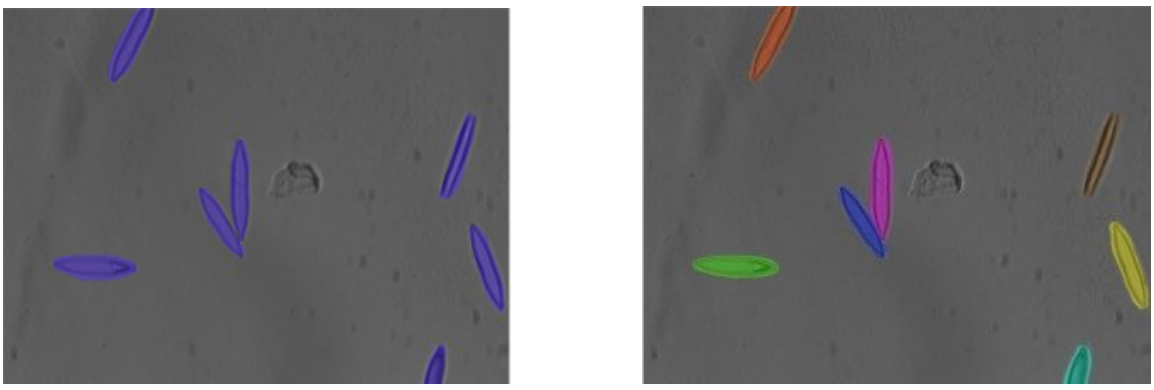
4.5 Image Segmentation

In many tasks in computer vision, we may require more information about an object in an image than just simply its bounding box. The next logical step is to define a method to instead locate the objects per pixel mask. Being able to extract the masks of objects plays a crucial role in tasks such as tumour detection [37] and autonomous driving [22]. In the case of historic maps, we can see that this category of methods could be extremely useful for both locating and finding a per character mask of the text. These masks could then be used to remove the text to aid vectorization.

In deep learning there are two types of segmentation: *Semantic* segmentation and *Instance* segmentation.

- **Semantic Segmentation:** Detects the object masks for each known category.
- **Instance Segmentation:** Detects each objects mask instance for each known category.

As seen in figure 19, the semantic segmentation model labels pixels in the image based solely on their object class. Whereas, for instance segmentation, the pixels are also labelled per instance of the object. Therefore, this shows that an instance segmentation model could theoretically be utilised to locate each instance of text. This could aid us in the problem of unstructured text recognition. Whereas, for the problem of finding per-character masks to be used in the removal of the text, both a semantic and instance segmentation model could be considered.



(a) (b)

Figure 19: Example of (a) a semantic segmentation mask vs (b) an instance segmentation mask tasked on the segmentation of algae [41].

4.5.1 UNet

UNet is a semantic segmentation deep learning neural network which was originally developed for bio medical image segmentation [39]. UNet essentially consists of two paths. The first path is an encoder which takes an image as an input and captures image context by applying a combination of convolutional and pooling layers until you are left with multiple small feature maps. These maps are then up-sampled and combined with feature maps from different scales to aid localisation until you reach a final output image which is the same width and height as the input with K different channels for each predicted class. This architecture allows the model to make a prediction as to whether each pixel constitutes an object class. An illustration of a UNet model can be seen in figure 20.

In order to train the model, in the original paper of UNet, Ronneberger, Fischer and Brox [39] proposed to use a pixel-wise soft-max with cross entropy loss. For some input image $I \in \mathbb{Z}^{N \times M \times 3}$ and its corresponding ground truth mask $\Omega \in \mathbb{Z}^{N \times M}$ we can define the soft-max as

$$p_k(x) = \frac{\exp\left(a_k(x)\right)}{\sum_{k'=1}^{K} \exp\left(a_{k'}(x)\right)} \tag{4.11}$$

where $a_k(x)$ is the activation value of the feature channel k at the pixel position $x \in \Omega$. The cross-entropy loss can then defined as

$$L = \sum_{x \in \Omega} w(x) \log(p_{l(x)}(x) \tag{4.12}$$

where $l(x)$ is the true class label of x and $w : \Omega \to \mathbb{R}$ is a weight map introduced to give certain pixels a higher importance.

Another common loss function used to train segmentation models is the dice loss function. The dice loss function is similar to the IoU metric and can be defined as

$$D = \frac{2|P \cap G|}{|P| + |G|}$$

where P are the predicted pixels and G are the ground truth pixels. Dice loss has an advantage over cross-entropy losses because of its robustness to class imbalance [7].

In order to utilised a UNet deep learning model for a task, we first need a dataset of images and corresponding ground truth masks to train the model on. In the case of text detection

and extraction in historic maps, we would be required to create masks of all the text. Often for image segmentation tasks this has to be done by manually drawing a mask outline for each object, or character in our case, in an image. Since the text in historic maps consist of mostly the same colour, one idea is to cluster the maps to make the text a uniform colour. Then create a computer program to allow the user to click on a character and use a flood-fill style algorithm to, in one click, obtain the mask of the character. This would be a hugely more efficient approach to extract most of the text masks. For characters that overlap features of the same colour, however, we would still need to draw the mask ourselves.

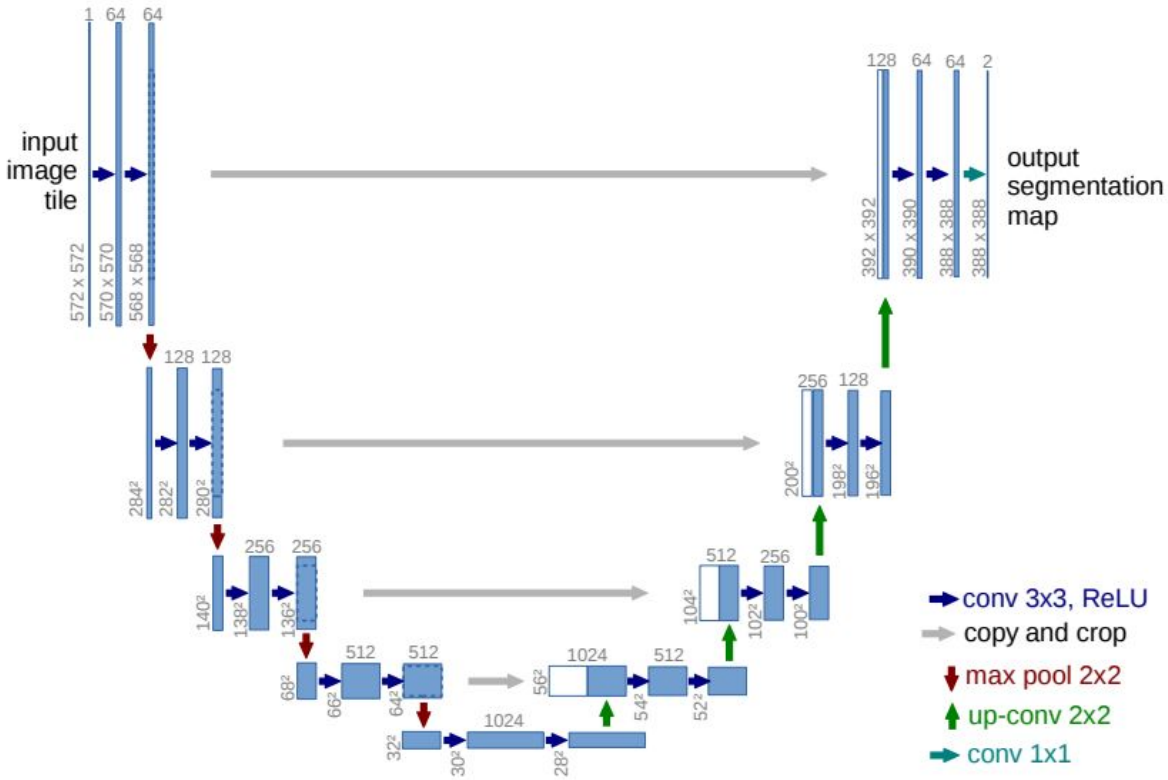


Figure 20: Example UNet architecture where each blue box corresponds to a a multi-channel feature map and the white box corresponds to a concatenated feature maps [39].

4.5.2 Mask R-CNN

Mask R-CNN is a deep learning neural network which is both an object detection and object instance segmentation model. Unlike UNet, Mask R-CNN is capable of locating and segmenting objects on a per-instance basis. This style of deep learning model could be extremely useful in the task of text detection in historic maps. The difficulty with using these styles of neural networks is that they requires vast amounts of training data to get reliable and accurate results. Specifically, in the case of the Mask R-CNN model, we not only require bounding boxes of objects labelled by class, but also the objects true instance segmentation mask. This information is extremely time consuming to obtain, and in some cases, may not be possible due to the limited variety of data.

The architecture of a Mask R-CNN model is an adaptation of the Faster R-CNN model, as discussed in section 4.4.3. The first step in the Mask R-CNN model is the predictions of region of interests via the region proposal network. Following this, instead of the region of interest pooling step, as seen in the Faster R-CNN implementation, the predicted bounding boxes go through a RoI Align step [16]. This step uses bilinear interpolation to tackle the issue of region proposals that lie between integer pixel coordinates. Despite this small change, introducing RoI

Align was shown to increase segmentation accuracy between 10% and 50%, [16]. The resulting regions of interest and feature maps are then fed into a final output layer containing a classifier, bounding box and mask predictor. An illustration of a Mask R-CNN architecture can be seen in figure 21.

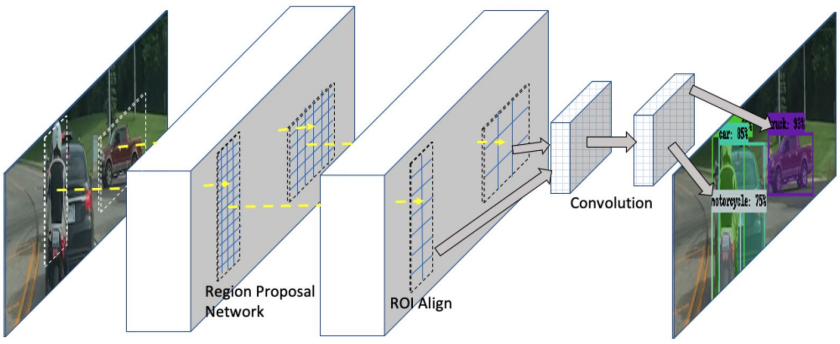


Figure 21: Illustration of an input image fed into an example Mask R-CNN architecture [21].

5 Optical Character Recognition

Optical Character Recognition, or OCR, is a category of methods designed to extract text from digital images. Being able to automate this, otherwise time-consuming, task is extremely important. This is especially true in areas such as the sorting of postal mail and digitising real world documents, which could otherwise take countless hours to perform. There are two main types of image classes which OCR methods are applied. These are structured and unstructured text images.

5.1 Structured Text

Structured text is text that is assumed to have certain features such as standard fonts, easily distinguishable rows and structures with upright orientations. Examples of this could be pages of a book or a scanned page of a passport.

5.1.1 Template Matching

One method used in optical character recognition is a pattern matching approach called template matching [19]. In template matching, an image known as a template is moved across an image in a sliding window approach and a correlation coefficient is calculated for each position. The position or positions with the highest values signify a match. In order to do this, we must have a similarity function.

One of the simplest similarity functions involves taking the sum of absolute differences (SAD) between each image pixel value and the overlapping template pixel value. This can be defined as

$$SAD(I_1, I_2) = \sum_{(u,v)\in I} |I_1[u,v] - I_2[u,v]| \tag{5.1}$$

where I_1 is the template image and I_2 is the base image window. Another similar method is to instead use the sum of the squared differences (SSD) which can be defined as

$$SSD(I_1, I_2) = \sum_{(u,v)\in I} (I_1[u,v] - I_2[u,v])^2. \tag{5.2}$$

Using the squared difference over the absolute can have the benefit of emphasizing the spread of the errors, which in turn can make thresholding for the templates easier. One final method is the zero-mean normalised cross correlation (ZNCC) method. This will return a value between -1 and 1 which corresponds to the two images ranging from negatively correlated to positively correlated. This can be defined as

$$ZNCC(I_1, I_2) = \frac{\sum_{(u,v)\in I} I_1[u,v] \cdot I_2[u,v]}{\sqrt{\sum_{(u,v)\in I} I_1^2[u,v] \cdot \sum_{(u,v)\in I} I_2^2[u,v]}}. \tag{5.3}$$

Whilst template matching can be an effective approach, if the base image's colours are modified in any way, or the image itself has undergone some transformation, then the templates predictive accuracy will significantly decrease.

5.1.2 Tesseract

Tesseract [50] was originally developed in the 1980's as part of a PhD research project at Hewlett Packard (HP) labs in Bristol [46]. Development of the Tesseract engine continued at HP between 1985 and 1994 and was eventually made open source in 2005. Since then, Tesseract has become a very popular and powerful open source optical character recognition package. Using the python wrapper for Tesseract, called pytesseract, means that converting an image of structured text to string can be done in only a few lines of code. This can be seen below.

```
from PIL import Image
import pytesseract
pytesseract.pytesseract.tesseract_cmd = r"C:\ProgramFiles\Tesseract-OCR\
                                         tesseract.exe"

image = Image.open('my_image.png')
text = pytesseract.image_to_string(image, lang='eng')
```

Needless to say, this is a very useful tool for easily extracting structured text from an image. However, in the case of the historic maps, most of the text is at different orientations and surrounded by noise so it would not be possible to apply the OCR method without some level of preprocessing. In figure 23, we can see that the resulting predictions of the Tesseract OCR engine can be massively dependent on orientation. In the case of the last two images in figure 23, there is only a 5 degree rotational different between the two images and yet this is enough to result in one image being almost correctly classified whilst the other not being recognised at all. Therefore, to recognise the text within the maps we require completion of a preliminary step prior to using the Tesseract OCR method. This involves the utilization of techniques such as those used in natural scene text detection.

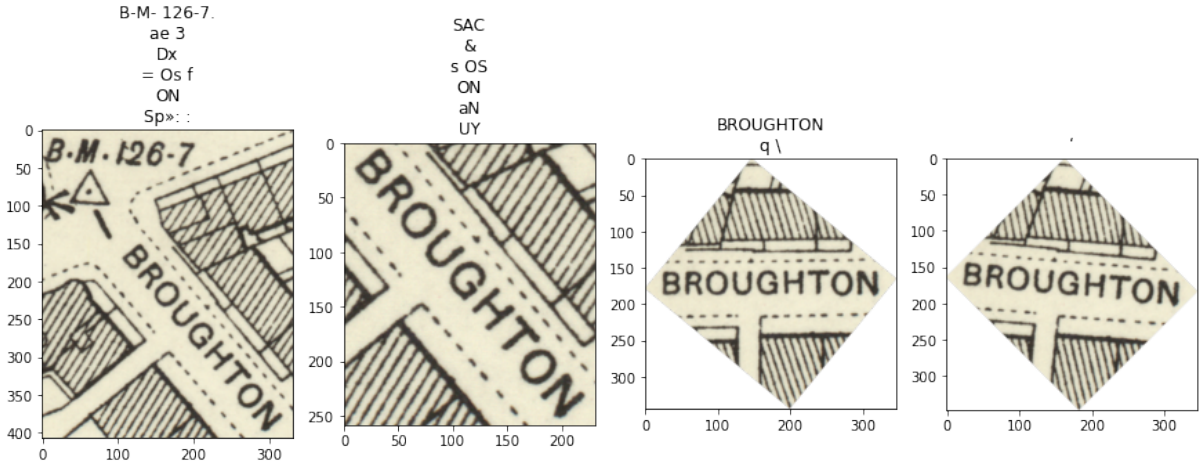


Figure 22: The Tesseract OCR engine applied to a selection of example map images containing text where the predicted text is labelled above each image. The last two images are rotated versions of the second image with angles of 50 and 45 degrees, respectively.

5.2 Unstructured Text

Unstructured text, commonly known as natural scene text, comprises of text in images where there is no prior knowledge of its structure. An example of this can be seen in figure **??**. Unstructured text is a much harder problem to solve than structured text. This is because we cannot simply apply classical OCR methods due to the methods having an fundamental assumption that the text is structured. Therefore, a common approach is to use convolutional neural networks to find bounding boxes of the text and then perform image processing before deploying a classical OCR method.

Figure 23: An example of an image containing unstructured text where the example predicted text bounding boxes are in red.

5.2.1 Efficient and Accurate Scene Text Detector

Efficient and Accurate Scene Text detection [56], most commonly known as EAST, is a deep neural network model designed for scene text detection. The EAST detector has three major parts, a feature extractor, a feature merger and prediction layers. Since the sizes of word regions vary significantly, a text detector needs to take into account that detecting large words requires features from deep within the feature extractor after multiple pooling and convolutional layers. Whereas, for detecting smaller words, the detector requires the features sooner, before too many pooling layers are applied, and the information is lost. Therefore, to tackle this, EAST detector adopts the approach of UNet [39] to gradually merge feature maps at different levels in the feature-merging branch. Figure 24 demonstrates this. After the feature-merging step, the resulting features are fed into an output layer consisting of a score map and either a rotated box (RBOX) or a quadrangle (QUAD) geometry text identifier [56].

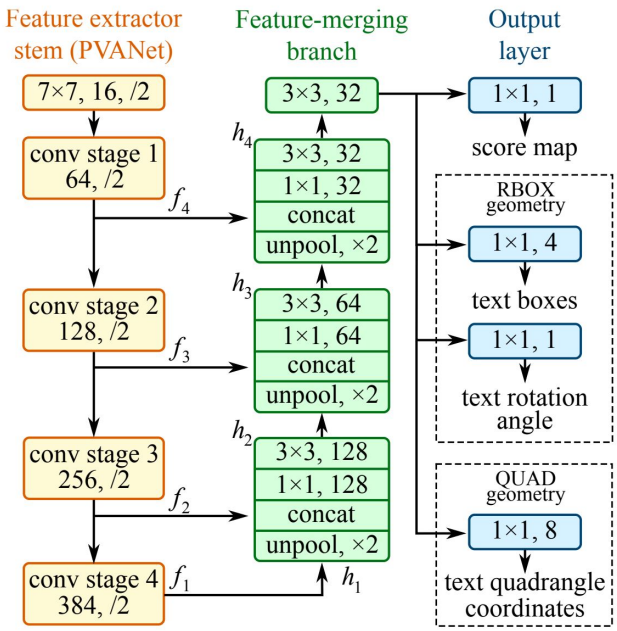


Figure 24: A diagram from [56] showing the structure of the EAST detection pipeline.

(a) (b)

Figure 25: (a) A ground truth text quadrangle (dashed) and the shrunken box (solid); (b) The corresponding score map who's pixel values range from $[0, 1]$ [56].

Assuming a rotated bounding box approach, then loss function can be formulated by the sum

$$L = L_s + \lambda_r L_r \tag{5.4}$$

where L_s and L_r are the loss functions for the score map and geometry losses, respectively, and λ_r controls the importance between the two. The score map uses a balanced cross-entropy loss, given by

$$L_s = -\beta Y \log(\hat{Y}) - (1-\beta)(1-Y)\log(1-\hat{Y}) \tag{5.5}$$

where $\hat{Y}$ is the predicted score map and Y is the ground truth. The parameter β is called the balancing factor and is given by

$$\beta = 1 - \frac{\sum_{y \in Y} y}{|Y|}. \tag{5.6}$$

Similarly, the rotated box loss is comprised of two loss functions given by

$$L_r = L_b + \lambda_\theta L_\theta \tag{5.7}$$

where L_b is the box overlap loss which utilises the intersection over union loss applied to the predicted box, $\hat{R}$ and ground truth, R, as defined by

$$L_b(\hat{R}, R) = -\log IoU(\hat{R}, R) = -log\frac{|\hat{R} \cap R|}{|\hat{R} \cup R|} \tag{5.8}$$

and L_θ is the loss function for the angle of rotation as given by

$$L_\theta(\hat{\theta}, \theta) = 1 - \cos(\hat{\theta} - \theta) \tag{5.9}$$

where $\hat{\theta}$ is the predicted angle of rotation, θ is the ground truth and λ_θ is some constant.

6 Results

6.1 Template Matching

The text within the map comes in many different fonts, sizes and rotations. We are therefore initially unable to carry out template matching to detect characters without any pre-processing. Without pre-processing, we would have to carry out template matching on an infeasible amount of template styles at different scales and rotations. This would also inevitably result in many false positive matches.

One specific feature of the maps that is ideal for a template matching approach, is the symbols for trees and shrubs. This is because there are only a few variations, and all of them appear on the map in the same rotation. By looking at the positive matches, we can then infer where the greatest amounts of greenspace lie on the map, and cross-correlate this with older maps to gauge changes in greenspace over time.

In order to carry out the template matching, we must first acquire the templates. This was done by locating and cropping out different symbols for trees from the original map and placing them in a folder. The tree templates that were used can be seen in figure 26. Following this, each template in the folder is passed over the entire image using a zero-mean normalised cross correlation template matching method. The bounding boxes resulting from this method that score higher than a given threshold are recorded. A threshold of 0.6 was experimentally found to result in a great balance of correctly matching a tree that contains noise, and also not finding many false positives e.g. buildings and text. An example of the applied template matching can be seen in figure 27 and 28 (a). In figure 27 especially, we can visualise the success of this method. It shows that all but a small number of trees that run along the water are found with no false matches.

Figure 26: Cropped tree symbol templates used in the greenspace template matching.

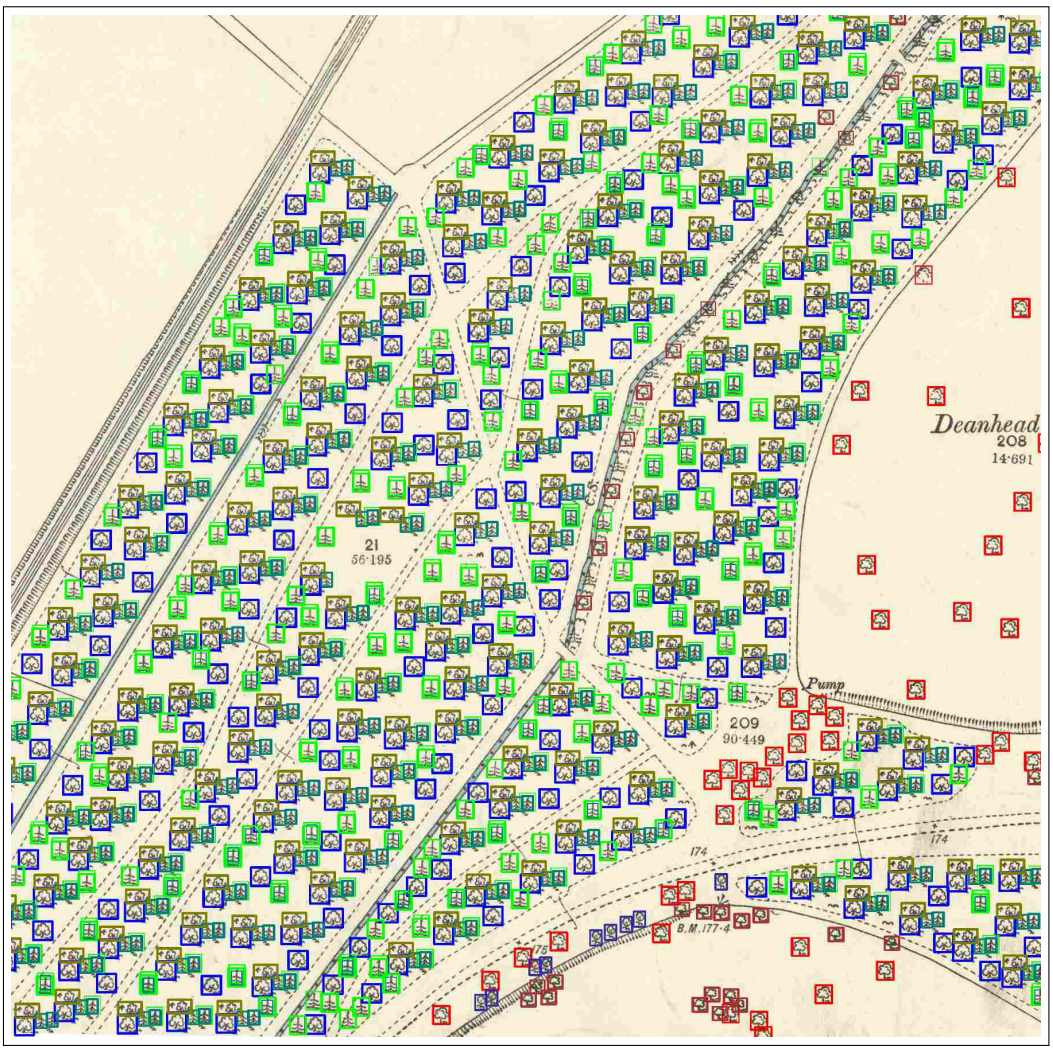
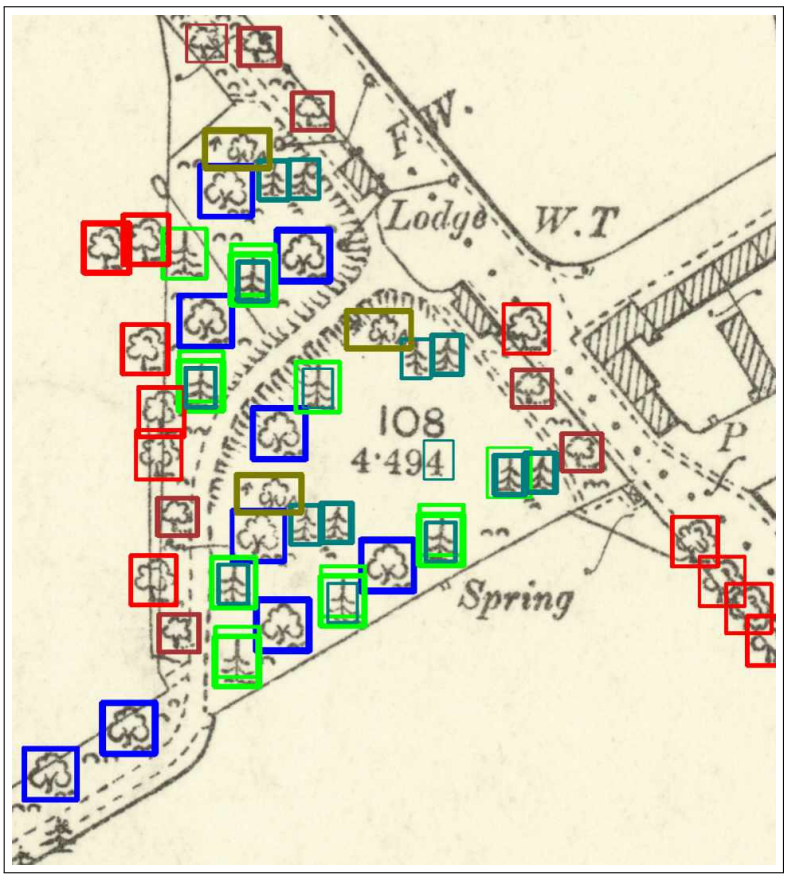
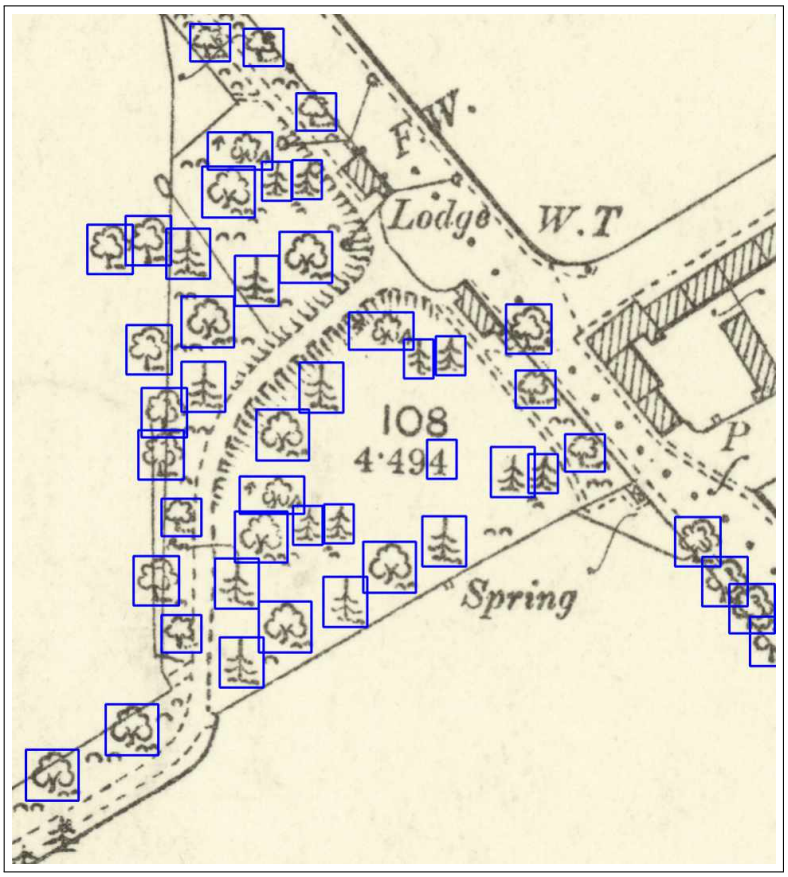


Figure 27: Tree template matching applied to a subsection of map with the templates from figure 26 and a threshold of 0.6 where each box represent a match and the box colour represents the different template it matches with.

In order to calculate greenspace density, we require a single bounding box per tree. However, as can be seen in figure 28 (a), the template matching method often picks up multiple of the same tree with the same or different templates. In order to combat this, we apply non-max suppression to our boxes with an IoU threshold of 0.7. This threshold is small enough to remove templates matching the same tree, but not so small that it removes the templates of different trees that are close together. The effect of the non-max suppression can be seen on the lower-right road in figure 28. The threshold of 0.7 was determined experimentally to effectively remove duplicate templates whilst also retaining individual templates that are situated close together. For a given map segment this parameter may require investigation to determine an optimal value.



(a)



(b)

Figure 28: (*a*) Resulting template matches for the ZNCC template matching method with a threshold of 0.6. (*b*) The suppressed template matches using non-maximum suppression with a IoU threshold of 0.7.

The aforementioned template matching method was then applied to a highly urban 12320×16863 map of Leith. After performing non-max suppression, 1497 template matches were found, with only 34 of these being false positive matches, equivalent to 2.27%. These false positive matches can be easily removed manually in a few minutes by outputting the indexes of the boxes as text on the boxes themselves, then recording and removing the negative results from the template box list. However, if necessary, further investigation could be carried out to optimize the template threshold. This could help refine the number of errors. Interestingly, 27 of the 34 false positive matchings were with text and therefore these errors could be eliminated by firstly removing the text, as best as possible, via utilisation of a pre-processing step.

By locating template matches, we aim to obtain a more visual perspective of the density of greenspace in the map. One useful description of the greenspace occuring on a map segment, is simply to count how many templates fall within a square or circular neighbourhood of each pixel of the map. For efficient calculation this can be reformulated into summing over a matrix centred on a pixel of interest where the matrix contains a one at each template position and a zero elsewhere. This however, is highly dependent on the size of the specified neighbourhood (either radius of the circle, or side of the square), and gives equal greenspace weightings throughout that whole region. For a more descriptive measure of greenspace local to each pixel in the map, it is more intuitive to have the centre of the template match contribute a higher weighting to the overall score and have this weighting decrease as you move away from this centre. For this we can use a weighting which follows a 2D Gaussian distribution, as seen in figure 29. This has the benefit that its size and standard deviation could be based on theory or experimental findings, such as for air pollution. Alternatively, a different Gaussian can be applied for each type of symbol. An example of a Gaussian density plot can be seen in figure 30.

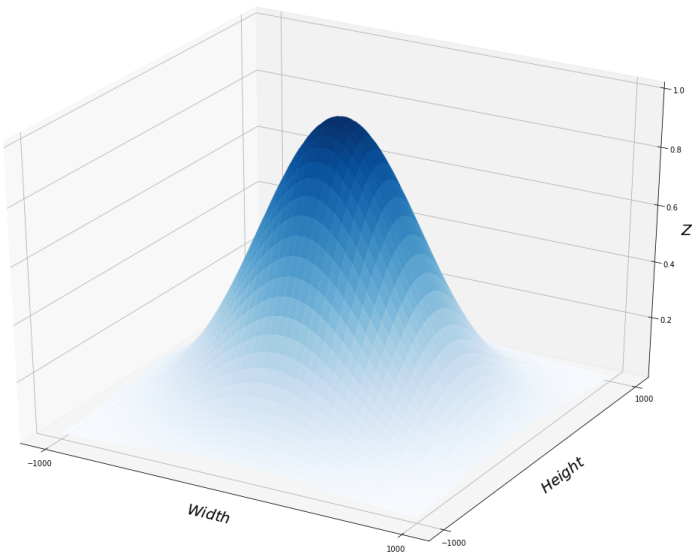
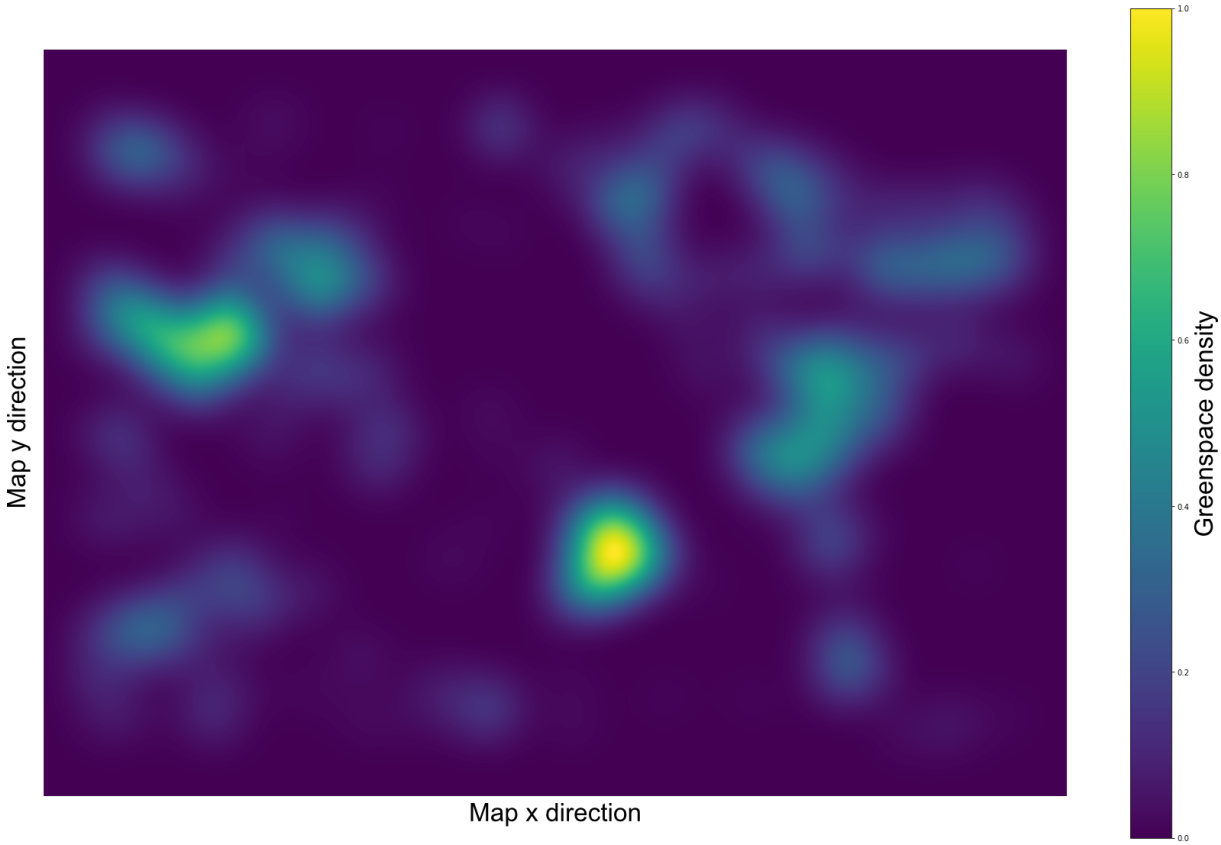
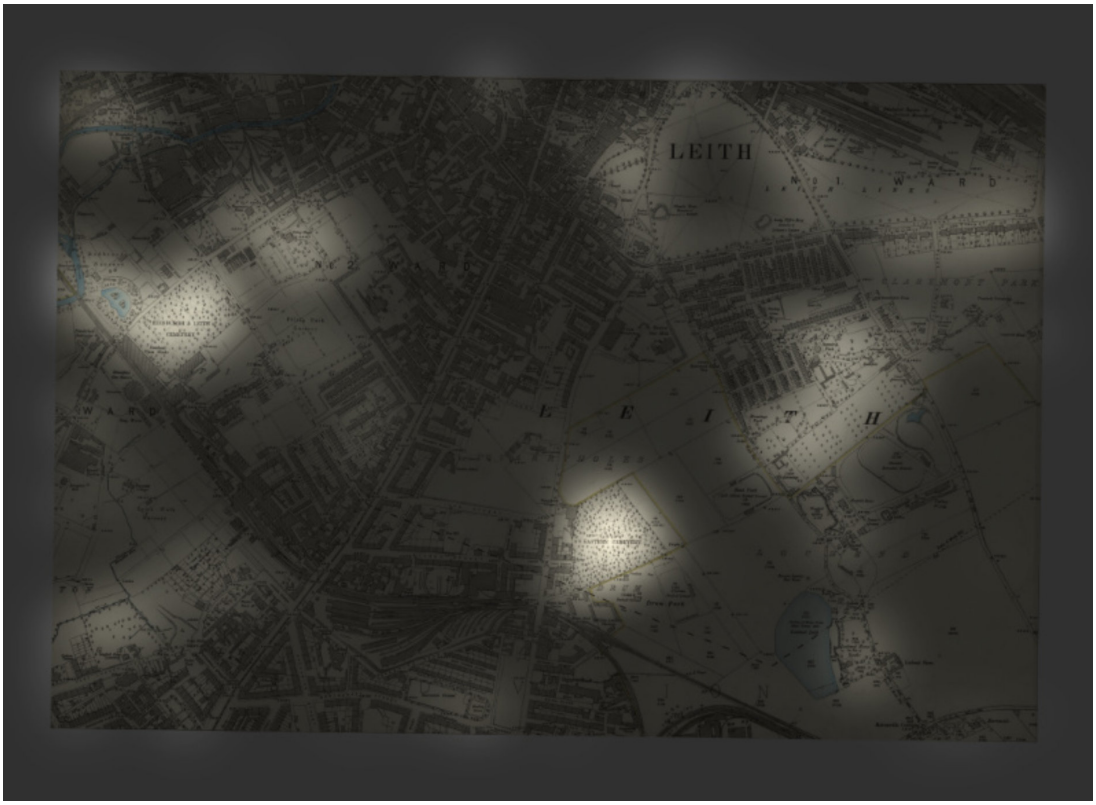


Figure 29: Plot of a 2D Gaussian distribution with $\mu = 0$ and $\sigma = 400$.

Another possible way of evaluating the greenspace is at each pixel record the Euclidean distance between itself and the nearest template match. This allows us to better visualise the different subregions of greenspace, as seen in figure 31.

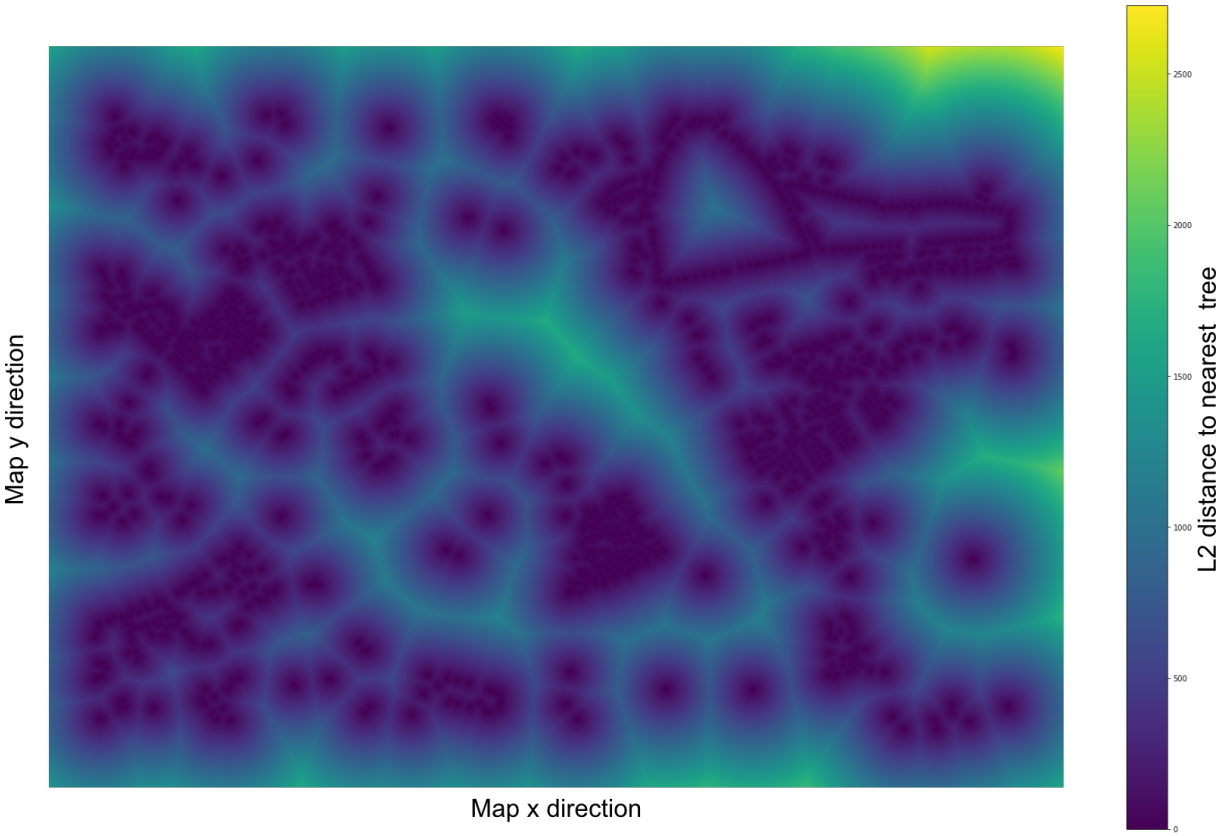


(a)

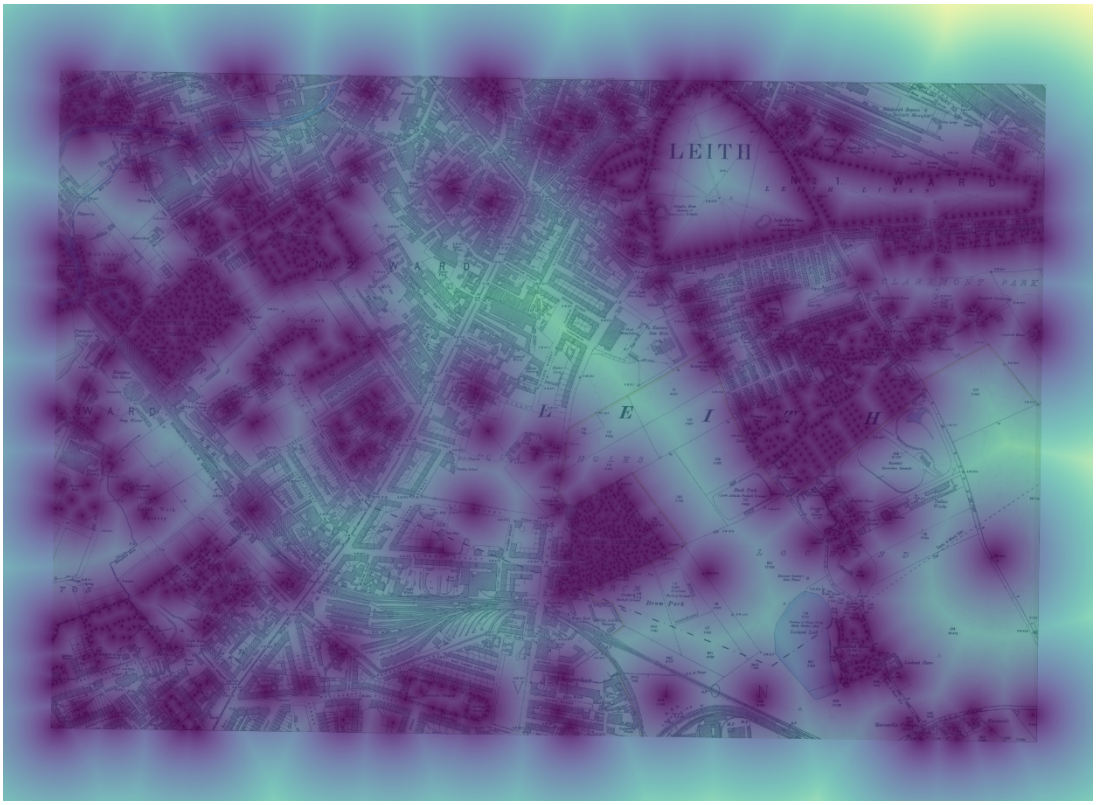


(b)

Figure 30: Gaussian density plot (a) where a Gaussian distribution with $\sigma = 400$ is added to a zero matrix matrix at each template center. An example overlay of the resulting Gaussian density plot can be seen in (b).



(a)



(b)

Figure 31: (a) a plot where each pixel value is the Euclidean distance between itself and the nearest template center. (b) is an overlay of the Euclidean template distance plot with the original map image.

6.2 Pre-trained Text Detection Models

To create a reference point, we tested out the performance of two pre-trained deep learning models. One was OpenCV's implementation of EAST text detector, trained on the 2013 and 2015 ICDAR dataset and the other, a current state-of-the-art scene text detector, CharNet [53], also trained on the ICDAR 2015 dataset. We applied CharNet to both a clustered and a binarized image, as seen in figures 33 & 32. The model correctly detects and recognises some of the text in the images without predicting any false positive cases. Interestingly, CharNet identifies far more text on the clustered image than the binary image. The EAST detector on the other hand, as seen in figure 34, performs much worse than CharNet. EAST struggles to decipher between the buildings and general image noise from the text. This leads to it predicting multiple false regions.

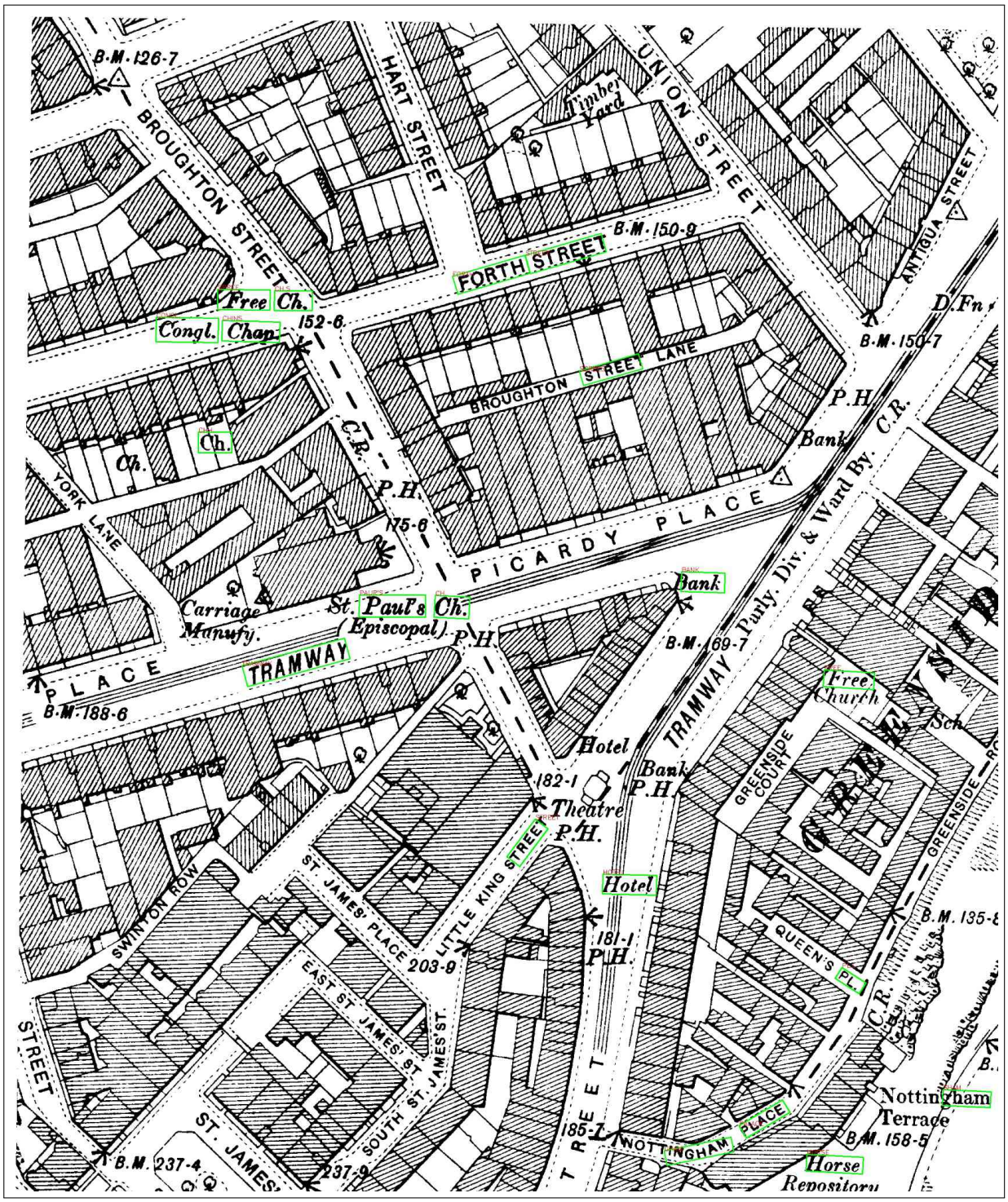


Figure 32: CharNet trained on the ICDAR 2015 dataset applied to a black and white map sub-image.

Figure 33: CharNet trained on the ICDAR 2015 dataset applied to a $k = 3$ clustered map sub-image.

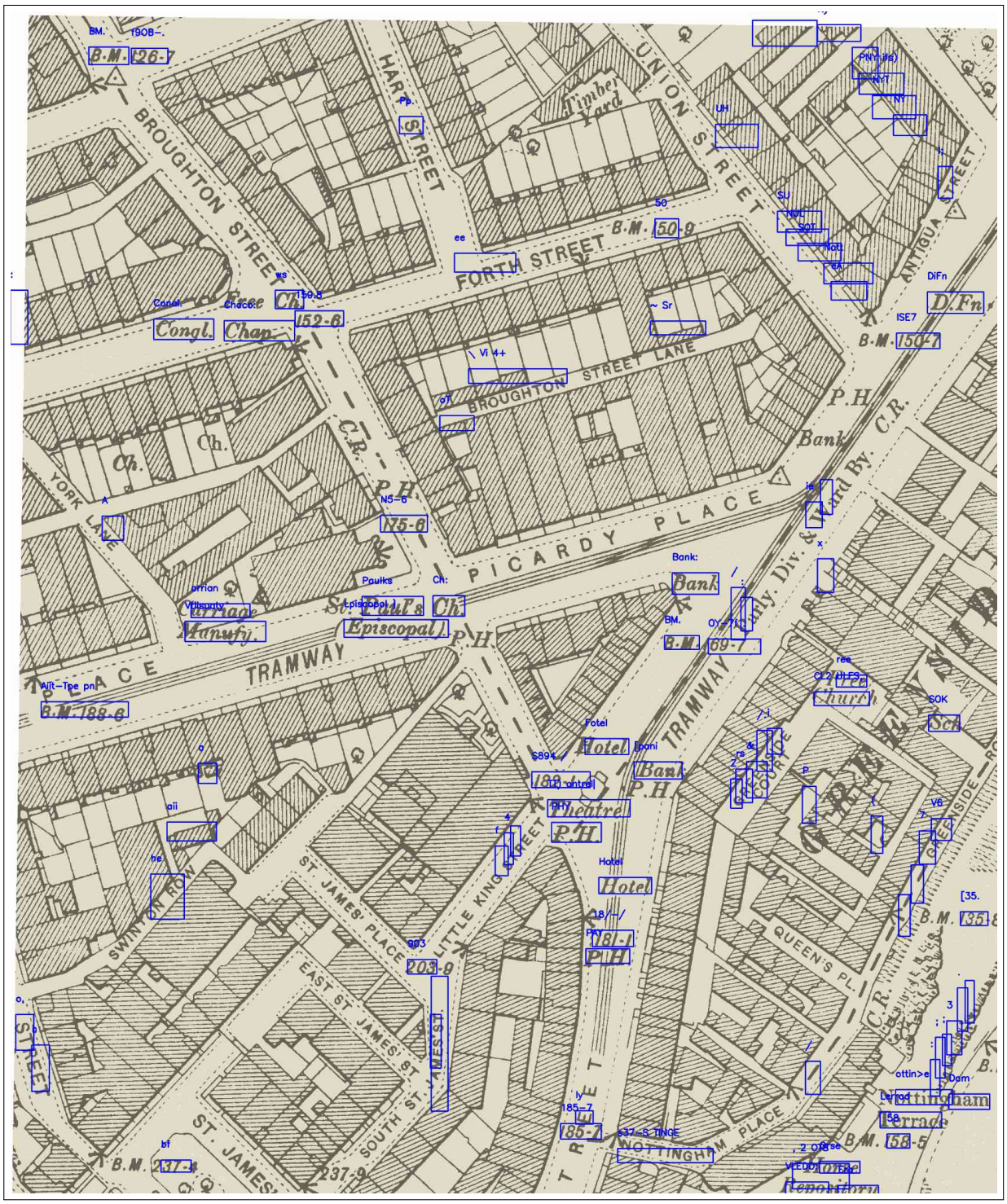


Figure 34: A pretrained EAST detector model applied to a $k = 3$ clustered map sub-image.

6.3 Creating A Dataset

To train an object detection or segmentation model, we need a data set of images, ground truth bounding boxes and per-pixel mask labels to train the model with. There are some open-source projects for generating scene text data such as SynthText or pre-made data sets like COCO-Text. However, we propose a way of creating a bespoke data set using the available map data. This gives us more control of adjusting the data set images to resemble the text in the maps more closely. This, we predicted, would lead to obtaining better predictive accuracy for the models trained on it.

One common way to create a data set for an instance segmentation model is to take images, or in our case, subsections of the map images, and carefully draw masks and bounding boxes around each instance of text and every individual character. This is very time consuming and, depending on how many maps you do this for, the images may not contain a great enough variety of text for the model to generalise the data without overfitting it. We instead propose a way to automatically generate a suitable data set by thoughtfully overlaying synthetic text onto sub-samples of the maps:

1. We applied pre-processing to the map images. A KMeans K= 3 clustering algorithm was trained on a subsection of a map, and then the colour of each pixel on the entire map were transformed to be the nearest of the resulting centroids. This played a part in not only ensuring that the new overlayed text looks as close as it can to the original, but also to remove pixelation about the characters to increase the effectiveness of the text recognition and segmentation in the prediction stage.

2. Next, we created map images absent of any text for the new text to be overlayed upon. This was done by using the graphics software Affinity Designer to roughly paint over all the text in the clustered maps. Other software such as Photoshop can also be used.

3. A dictionary of texts and fonts to be randomly sampled was then created. To do this, 5,000 randomly generated numbers and characters sets were appended to a 25,000-word dictionary. In this case, 16 similar fonts to the fonts used in the maps were identified and chosen.

4. For each data set image, we took a subsection of one of the filled in maps. Then, we proposed randomly sampled and positioned text and fonts which could be placed onto the map at a later stage.

5. For each proposed text, we calculated their bounding box coordinates and assigned it an overlap score. This score is the amount of pixel overlap with text and black in the map for the proposed position, divided by the total number of pixels in the text. This gives a measure of text overlap which helps to pick the most appropriate text placement.

6. We then sorted the proposed texts by score and discarded texts with a greater score than some threshold. A threshold of 0.17 was experimentally found to give a good balance of text overlap and clarity.

7. The text with the lowest score was chosen and added to a new list of chosen texts. For each text in the sorted list, the text is also appended if its bounding box does not overlap with any of the other bounding boxes in the new list. Therefore, the text that fits the background best without overlapping any other text is chosen.

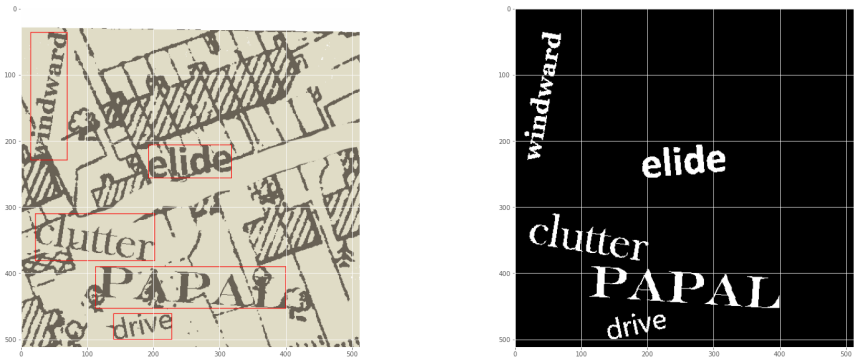
8. The chosen text is then overlayed onto the sub-image and an XML file containing the information of the all the texts, such as their score, bounding boxes and flattened masks is created. An example file can be seen below.

Dataset XML file

```
<annotation>
  <Filename>0_00000001.png</Filename>
  <object>
    <Mask>0000000011100010000... </Mask>
    <Text>Apple<\Text>
    <Rotation>-29.1932<\Rotation>
    <OverlapScore>0.0182<\OverlapScore>
    <bndbox>
        <xmin>72<\xmin>
        <ymin>129<\ymin>
        <xmax>105<\xmax>
        <ymax>157<\ymax>
    </bndbox>
  </object>
  <object>
    ...
  </object>
  ...
</annotation>
```

9. This method is then repeated for the desired number of data samples. To greatly increase the speed at which the samples are generated, we ran this in parallel by utilizing a different background image instance on each core.

The resulting data set generation method was found to be reasonably efficient. Running on all 12 threads of an i7-8700K CPU with 40 text proposals, I was able to generate a training set of $30,000$ images of sizes 1024×1024, and a testing set of $9,000$ images along with their corresponding XML data files in just over 20 minutes. Thus, averaging over 32 1024×1024 images and their XML data files per second.



(a) (b)

Figure 35: An example text image and its corresponding text masks, where the red box describes the ground true text instance bounding box.

Table 2: Table of associating overlap scores for the proposed texts for figure 35 with the chosen texts in black and the discarded texts in gray.

Score	Text
0.01122334455667789	drive
0.03049370764762827	clutter
0.04121687929342493	get
0.054093567251461985	Braniff
0.06630581867388363	windward
0.06751054852320675	bolt
0.1037444177258674	Montenegrin
0.11217323449083105	matchmaker
0.11384524162466647	flowchart
0.12375690607734807	G
0.129764801297648	papal
0.13317847464188928	elide
0.15611146295123496	lugubrious
0.16441251056635672	wiener
0.17479674796747968	ashman
0.17483363581367212	nibble

6.4 Mask R-CNN

A Mask R-CNN was trained on our generated dataset. We chose a Mask R-CNN because of its ability to do both object detection and instance segmentation. Theoretically, this would allow us to locate the bounding box of the text and could be used in conjunction with classical OCR methods to recognise the text. This also gives a pixel-wise mask of the text to easily allow us to remove it. This aids vectorization of the map buildings.

The Matterport implementation of Mask R-CNN [1] was used. This is based on a Feature Pyramid Network with a ResNet101 [15] feature extractor backbone and trained on the MS COCO dataset [25]. In an attempt to increase the speed of the models convergence, transfer learning was used. This was achieved by initialising the model with the weights of the pre-trained network and setting the trainable layers to be the heads of the region proposal network, classifier and mask generator. We then proceeded to train the model on our custom dataset with a learning rate of 0.009, on $1,000$ 1024×1024 images per epoch and a batch size of 2 for 19 epochs. Since the neural network is detecting small instances of text in each image, we set the region proposal network anchor scales to be sizes $(8, 16, 32, 64, 128)$. Each epoch took approximately 21 minutes running on a RTX 2070 graphics card. The graphs of the loss functions can be seen in figure 36. After the 19th epoch, the loss function appeared to plateau and thus we decrease it by a factor of ten and carried on the training process for another 25 epochs. It can be seen in figure 36 that this decrease largely helped the convergence of the mask loss function. An image of this trained network applied to real text, on an unseen map, not used for any data generation, can be seen in figure 37.

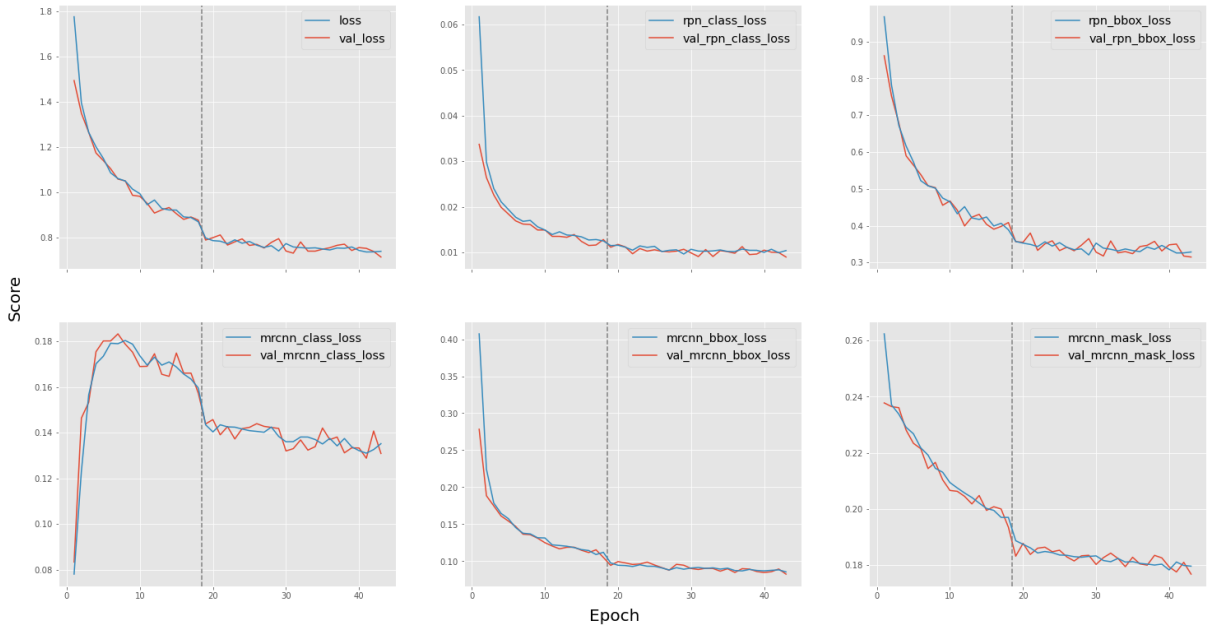


Figure 36: Graphs of the different loss functions for the region proposal network (top-middle & top-right), the Mask R-CNN prediction layers (bottom row) and the total loss function (top-left) for the Mask R-CNN model during the training process. The training losses can be seen in blue, while the validation loss is visible in red. In the plot figure legends class and bbox are short for classifier and bounding box, respectively. The dash line at epoch 19 indicates when the learning rate was changed from 0.009 to 0.0009 to aid convergence.

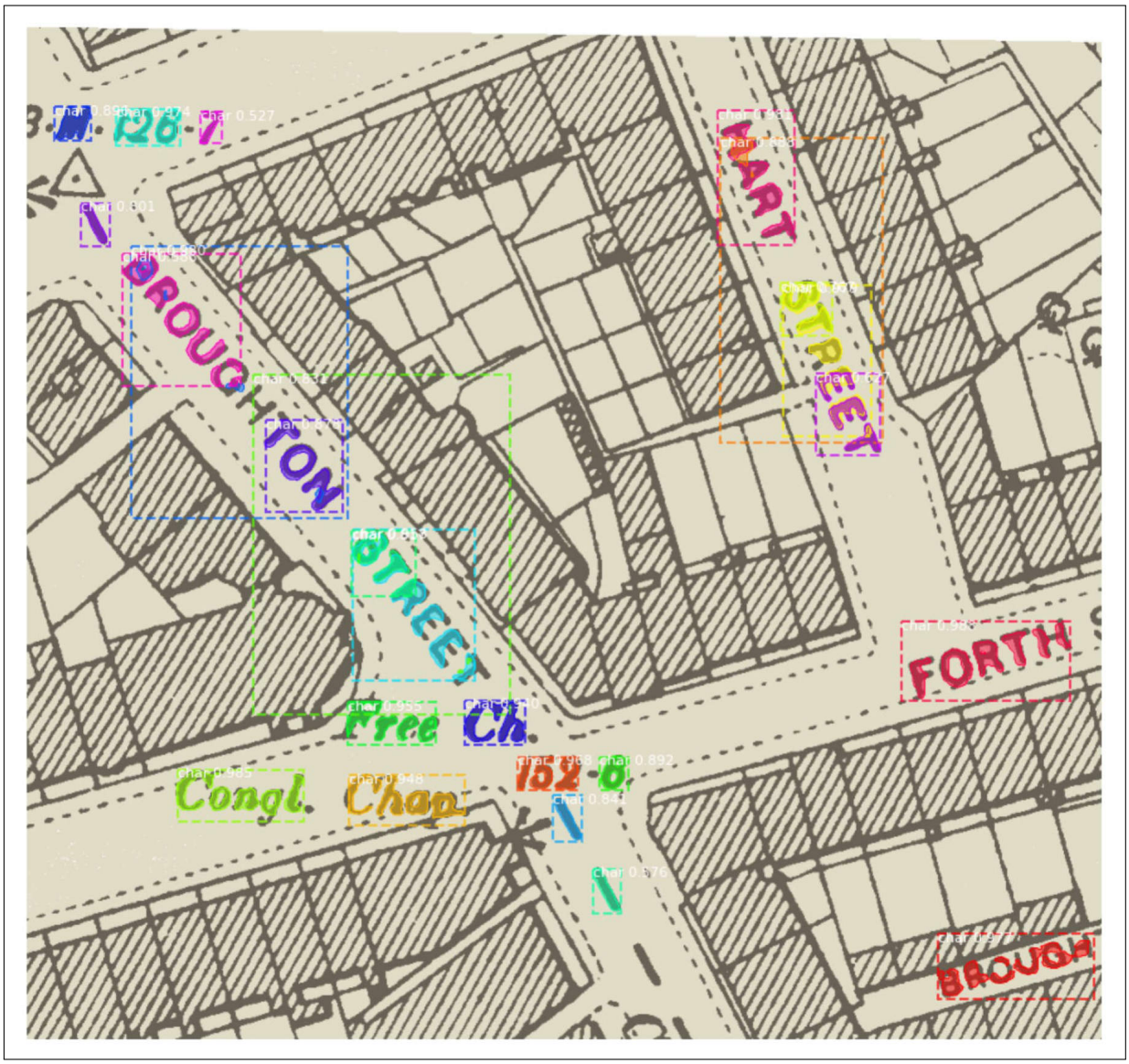


Figure 37: A $K3$ clustered 1024×1024 unseen map image fed into our Mask R-CNN model trained only on the top layers of the RPN, Classifier and Mask generator. Each colour of bounding box and mask represents a different prediction of a text instance.

In figure 37, we can see that the model was able to identify all fully visible text in the image whilst predicting impressive pixel-wise masks for most characters. One issue, however, is that the model double predicts a lot of text which would make it very difficult to piece the appropriate text back together in order to apply OCR methods. Thus, we initialised the model with the newly trained weights and allowed the whole model to be trainable as opposed to only the heads. We lowered the learning rate by a further factor of five and continued the training process for 10 more epochs. The resulting change in loss functions can be seen in figure 38. One of the more significant changes in loss was the bounding box prediction for the region proposal network. This allowed the model to better predict where each instance of the the text is before it is classified. This improvement is clearly evident in figure 39 where the neural network no longer double predicts the same instance of text. One thing that is clear from figure 39, however, is that it struggles to fully predict the masks of longer words. We tried a variety of different learning rates over numerous more epochs but were not able to successfully solve this problem.

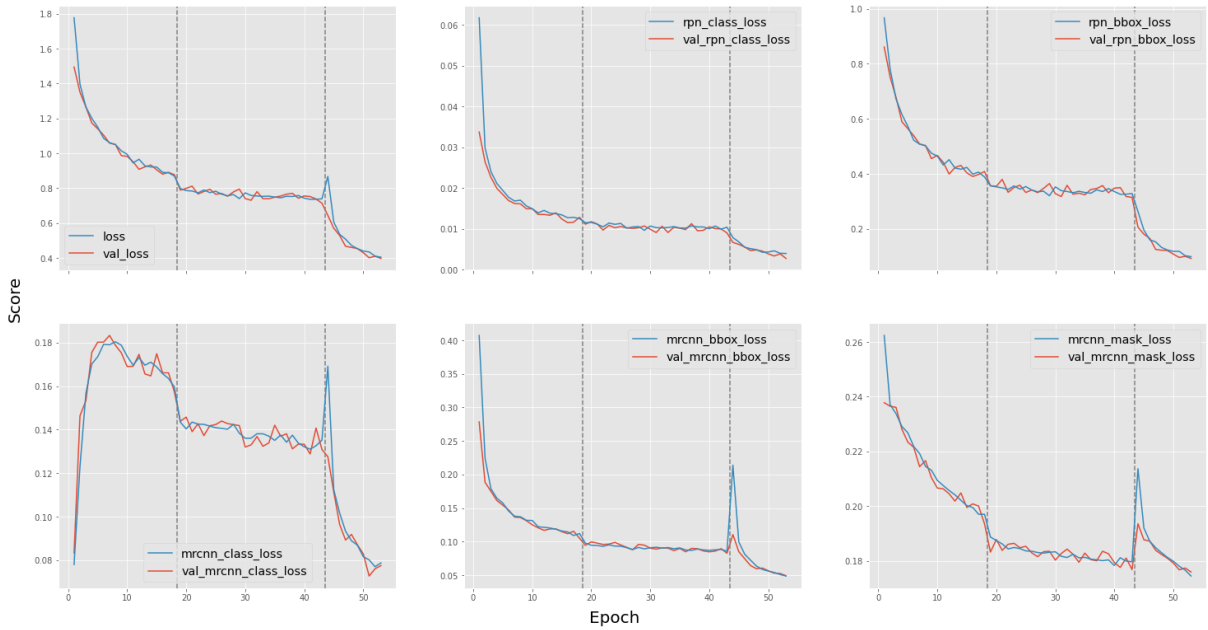


Figure 38: Graphs of the different loss functions for the region proposal network (top-middle & top-right), the Mask R-CNN prediction layers (bottom row) and the total loss function (top-left) for the Mask R-CNN model during the training process. The training losses can be seen in blue, while the validation loss is visible in red. In the plot figure legends class and bbox are short for classifier and bounding box, respectively. The first dashed line at epoch 19 indicates when the learning rate was was changed from 0.009 to 0.0009 to aid convergence. The second dashed line at epoch 43 indicates when the all of the Mask R-CNN layers were changed to be trainable.

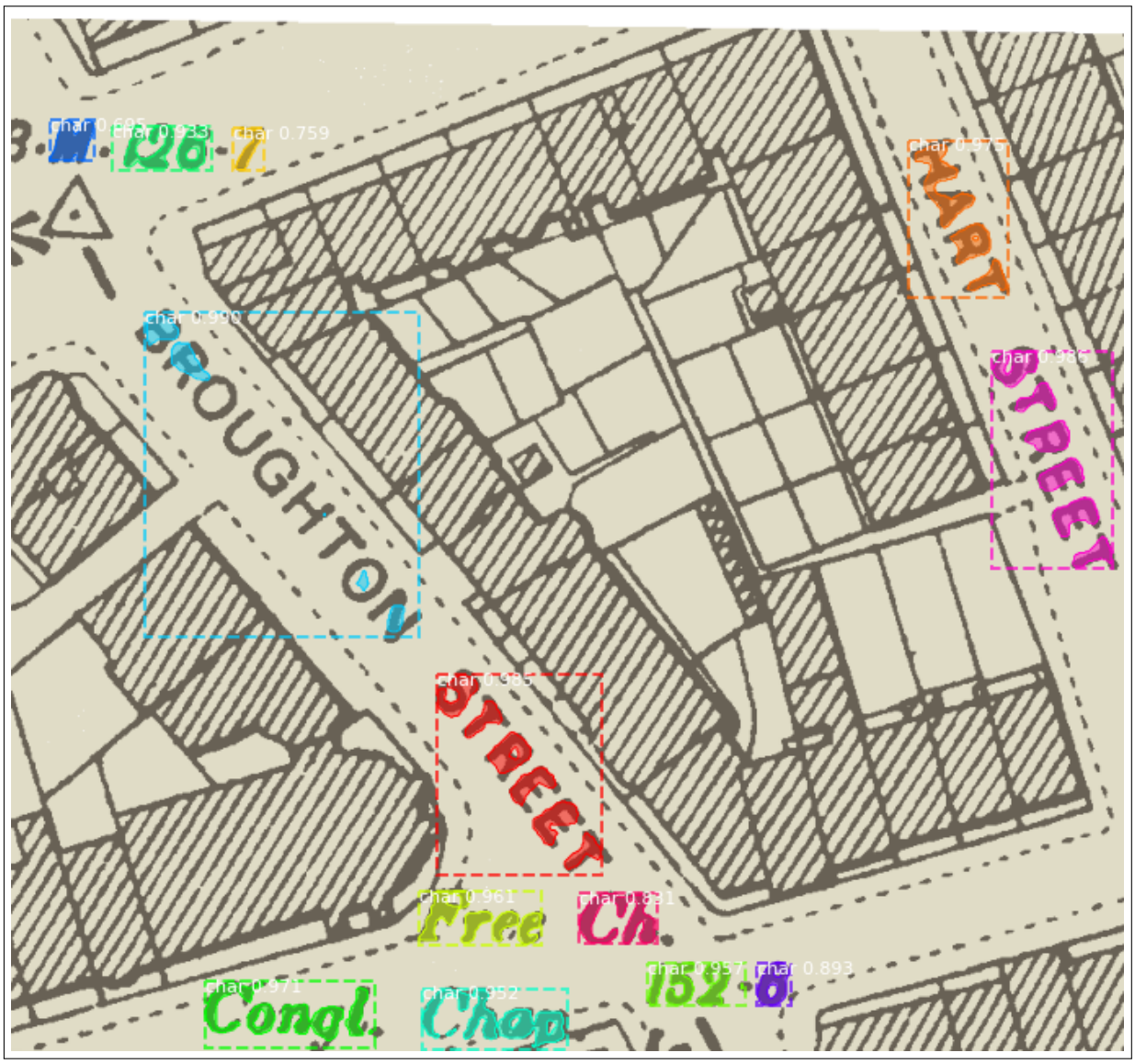


Figure 39: A $K3$ clustered 1024×1024 unseen map image fed into our Mask R-CNN model trained on all layers. Each colour of bounding box and mask represents a different prediction of a text instance.

Due to the Mask R-CNN object detector having been trained exclusively on sub-sampled images of size 1024×1024, we cannot simply take a larger image, resize it and feed it to the neural network and expect the same results. Therefore, in order to use the Mask R-CNN on the whole map image, we employed a windowing method, section 4.4.2. As seen in figure 39, some of the text will inevitably lie half-way between the windows. To account for this problem, we must allow for some overlap in the windows, then apply a method such as non-maximum suppression to remove the duplicate predictions. An example of this method, applied to a large image, can be seen before non-max suppression in figure 40 and then afterwards in figure 41.

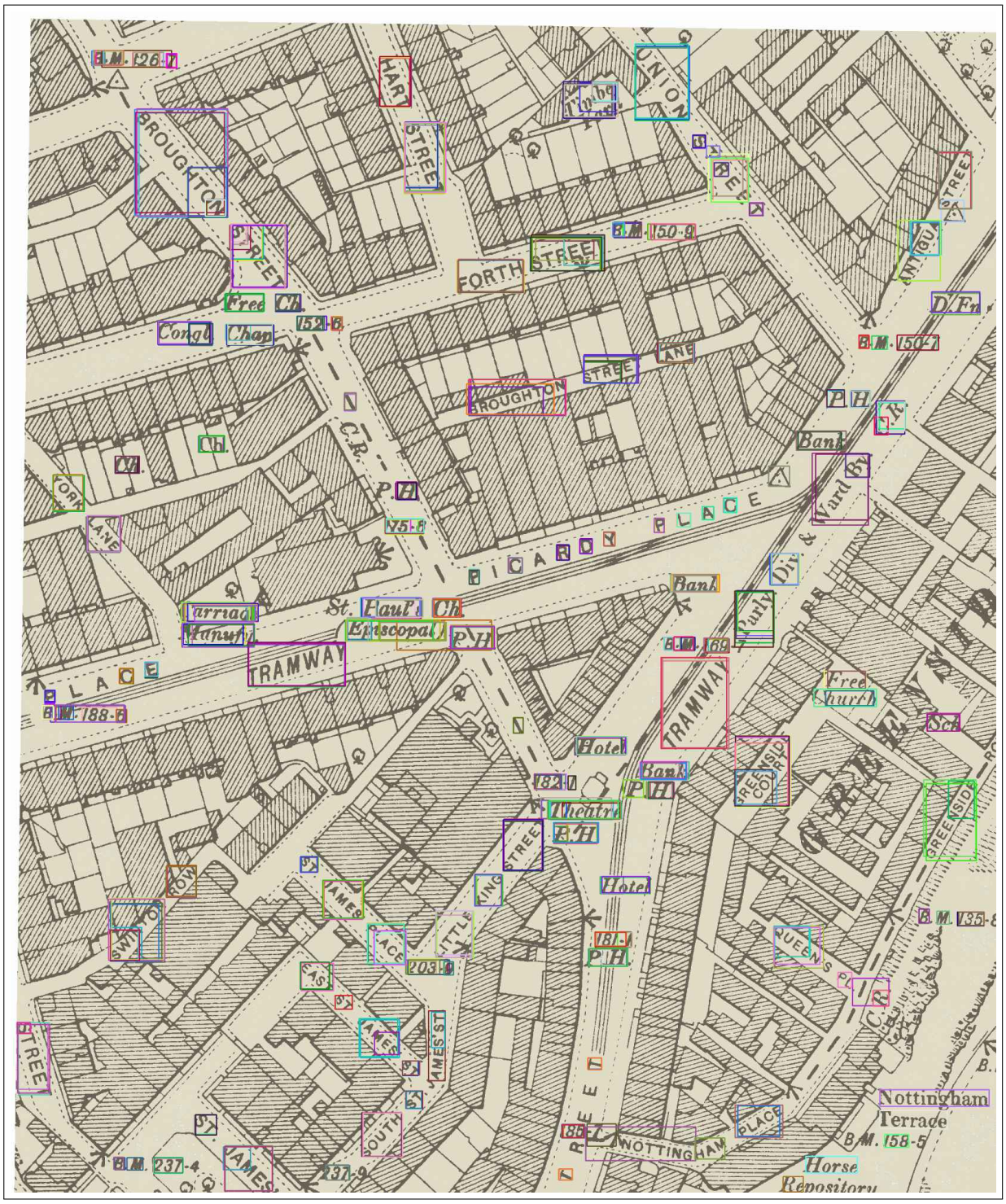


Figure 40: The resulting bounding boxes before non-max suppression for the sliding window method involving the trained Mask R-CNN model applied to a larger unseen image.

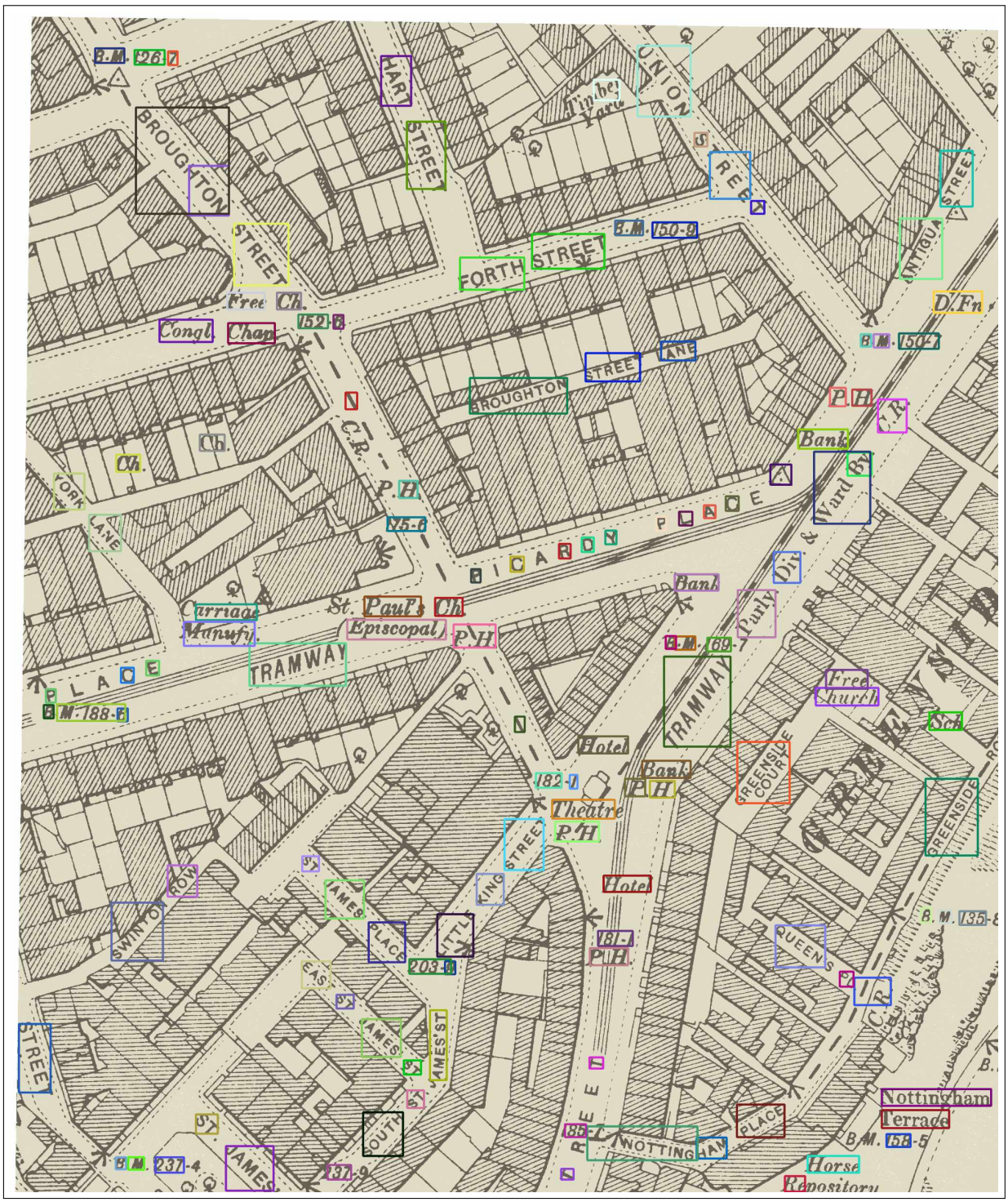


Figure 41: The resulting bounding boxes after non-max suppression for the sliding window method involving the trained Mask R-CNN model applied to a larger unseen image.

We can see from figure 41 that this method works very effectively at detecting most of the text on the map. Some instances of text that the neural network does struggle with, however, is the large "GREENSIDE" font, and text where the characters are spaced out, such as, for example the word "PICARDY". Also, there are some problems with non-maximum suppression suppressing a correct bounding box, such as the "Timber Yard" text at the top-middle section of the map of figure 41.

In an attempt to combat a few of the issues with the previous model, we altered the dataset and changed the random word dictionary such that every 5th word had a single space between its

characters and every 7th word had two. On top of this, we also increased the overlap threshold to 0.23 and added some larger font styles. The Mask R-CNN model was then retrained using the previously trained weights on this adapted data set. The comparative results can be seen in figure 42. The model shows strong signs that it is able to infer from the image the text's rotation. It better predicts where each instance of spaced text is and even combines street names which are situated close together in the same orientation.

To demonstrate the strength of the model, we applied the retrained network to an image which contained text surrounded by noise and overlapping buildings. We can see from both the predicted bounding boxes after non-max suppression and the generated masks from figure 43 that the neural network performs well even on these difficult cases.

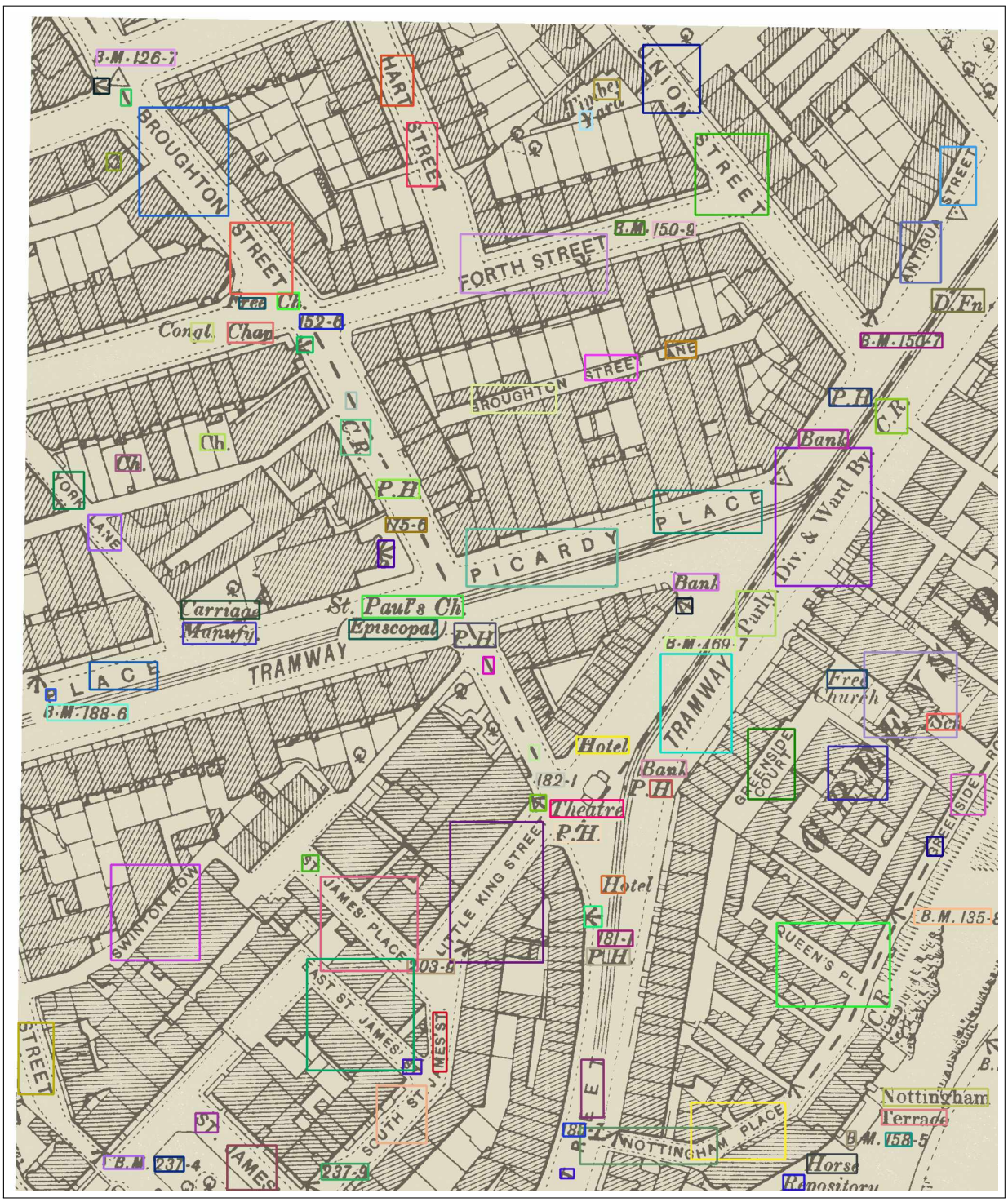
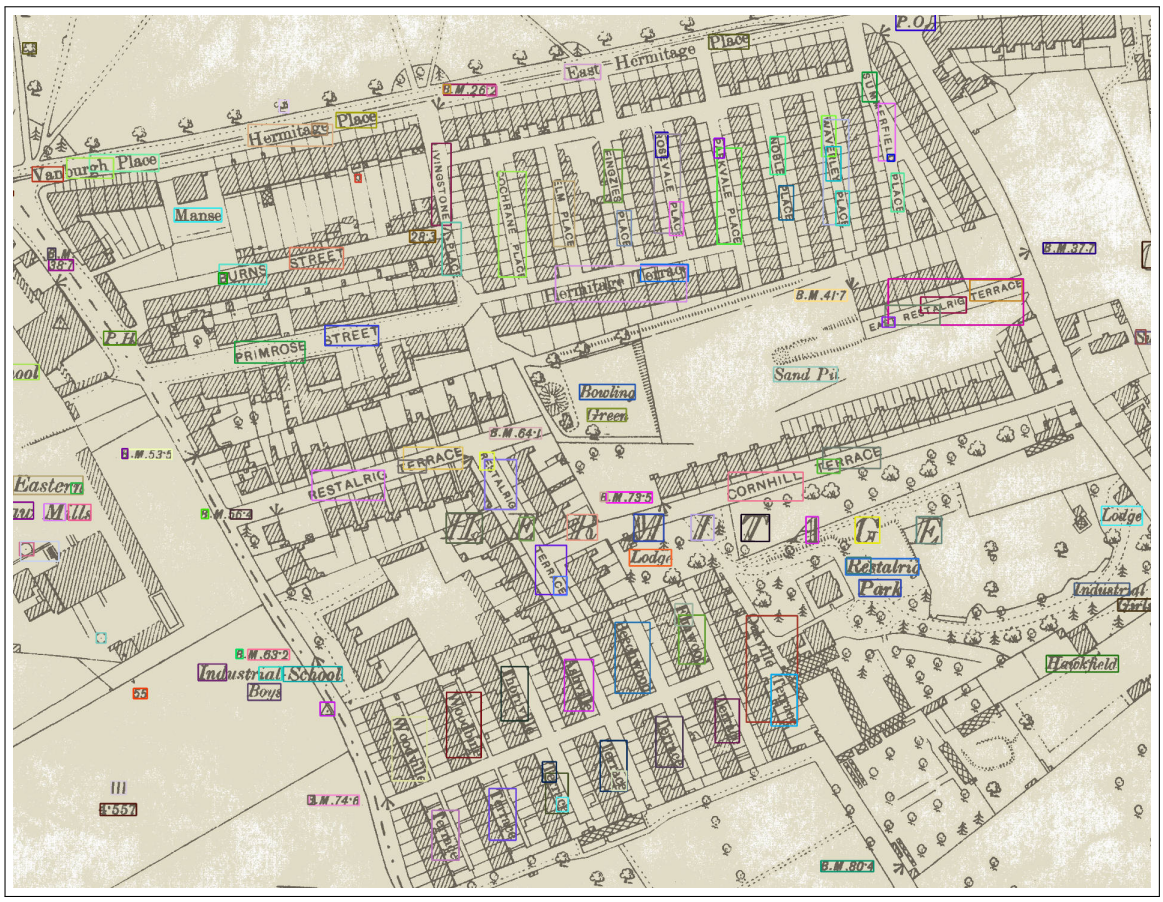
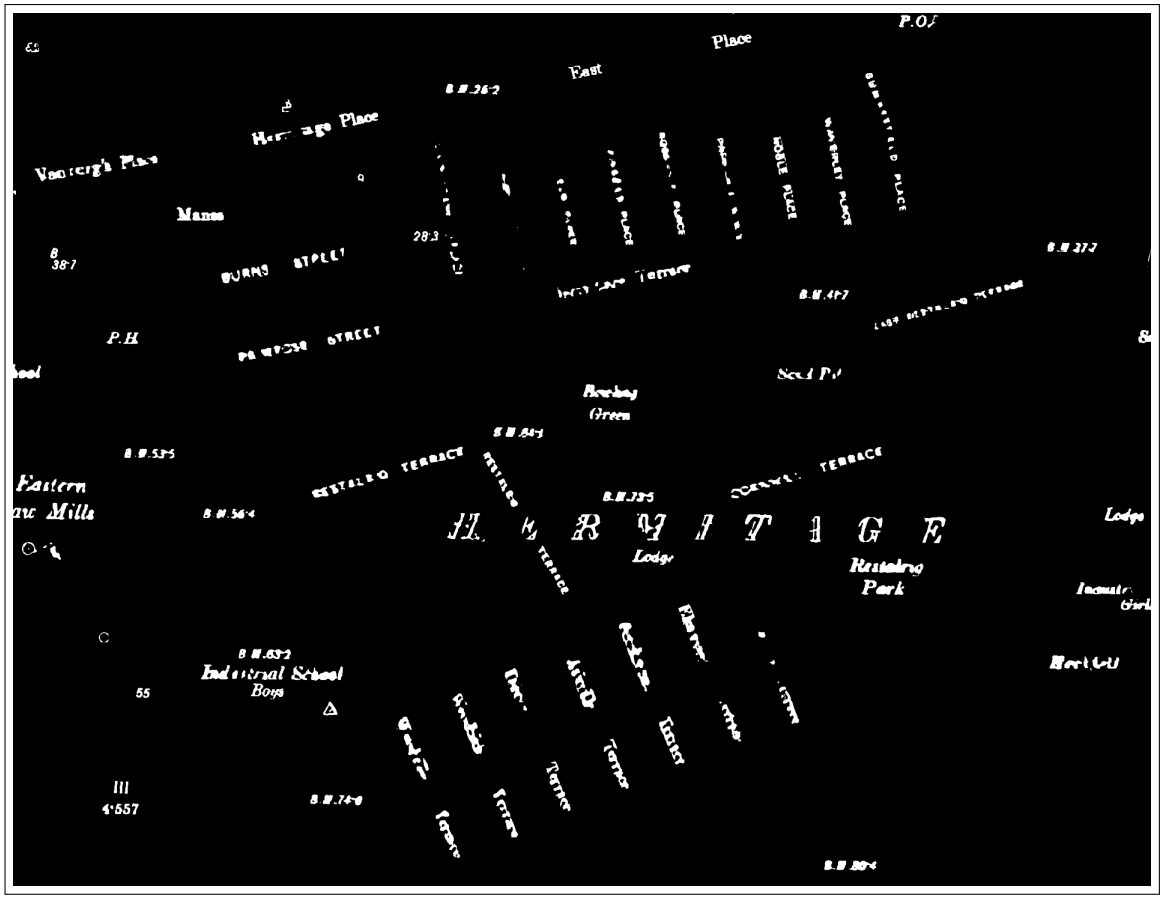


Figure 42: The resulting bounding boxes after non-max suppression for the sliding window method involving the retrained Mask R-CNN model applied to the image in figure 41.



(a)



(b)

Figure 43: The resulting bounding boxes (a) and masks (b) for the sliding window method involving the retrained Mask R-CNN model applied to a difficult subsection of a map.

6.4.1 Accuracy Comparison

To be able to quantify the predictive bounding box accuracy of the models, we used two separate accuracy measures. Accuracy 1 defines the percentage of words which are correctly situated in a bounding box, predicted by the model. Whereas, Accuracy 2 defines the percentage of words which are partially identified by one or more bounding boxes. The results of the CharNet model; the Mask R-CNN model, before the retraining for words with spaces (Model 1) and the retrained Masked R-CNN (Model 2) can be seen in table 3.

Table 3: Table of predictive bounding box accuracy measured on a 4000×4000 subsection of a historic map image for the deep learning models. These models include CharNet, the pre-retrained Mask R-CNN model (Model 1) and the post-retrained Mask R-CNN model (Model 2), both before and after non-maximum suppression (NMS). Accuracy 1 defines the percentage of words which are correctly situated in a bounding box predicted by the model. Accuracy 2 defines the percentage of words which are partially identified by one or more bounding boxes. Missed defines the percentage of words which were not recognised at all.

Model	Accuracy 1 (%)	Accuracy 2 (%)	Missed (%)
CharNet	35.96	44.74	55.26
Model 1 before NMS	77.19	95.61	4.39
Model 1 after NMS	73.68	95.61	4.39
Model 2 before NMS	87.72	99.12	0.88
Model 2 after NMS	83.33	99.12	0.88

We can see in table 3 that both Model 1 & 2 more than doubles the performance of the pre-trained state of the art text detector CharNet for both accuracy measures. In addition to this, there is also a significant difference in the Accuracy 1 percentage obtained by the Mask R-CNN model, retrained on the adapted dataset, and the prior Mask R-CNN model. This highlights the increase in model performance that could be obtained by optimizing the dataset generator's parameters for this specific problem.

It is also noticeable in table 3 that in both Models 1 and 2, applying non-maximum suppression decreases the Accuracy 1 percentage by approximately 4%. As seen in section 6.4, this is due to the non-maximum suppression algorithm suppressing the true bounding box for the text in favour of a different bounding box. This could be resolved by training the Mask R-CNN for longer so that it predicts the true boxes with a higher confidence. Further investigation could also be carried out to find a more complimentary method of combining multiple prediction bounding boxes.

6.4.2 Text Recognision with Mask R-CNN

After detecting the mask and bounding box of the text, the next step is to attempt to recognise the text. In order to effectively utilise a optical character recognition method, we first require the text to be orientated in the correct direction with as little noise around the text as possible.

One idea we had to adjust for the text orientation and noise was to exploit the information from the predicted mask. Even though the neural network doesn't give a perfect pixel-wise segmentation of the text, it holds vital information about the location of the individual characters in the text. Thus, from the generated mask we could find the minimum area rectangle which enclosed this mask, as seen in figure 44. Due to the text being straight, this method gives us the orientation of the text in the image and hence allows us to straighten the text by using the rectangle's rotation. Unfortunately, the angle at which we must rotate the text depends on if the text is angled upwards or downwards. We must rotate $-\theta$ if angled downwards or $-90 - \theta$ if angled upwards. To help tackle this, we can rotate the text using the orientation of the rectangle's longest side. This solves the problem for text of a length longer than 2 characters,

however, for single characters this method often results in the text being rotated in an incorrect direction, as seen in figure 46. Therefore, in instances where the minimum area rectangle's sides are of similar length, it might be more appropriate to use a different method, such as a pattern matching approach, to determine the correct orientation. From the correctly orientated image we can once again utilise the mask and its minimum area rectangle to remove a lot of noise both above and below the bounding box. The resulting image can then be fed into an OCR method to more accurately determine the text.

Figure 44: The cropped STREET text bounding box and mask predicted by the Mask R-CNN model where the red box is the minimum bounding rectangle. The cropped box is rotated using the minim bounding boxes rotation and image noise is removed above and below the mask.

This method of using the bounding box to correctly orientate the image performs well even on longer instances of text of which, in figure 39, it can be seen that the neural network struggles with predicting the masks. This is because it still returns spots of the mask for both ends of the text which is enough information to gauge its orientation. One instance of this can be seen in figure 45. The only additional thing that may need to be considered is giving an extra amount of padding to the resulting bounding rectangle before the noise is removed. This would help to prevent segments of the text being removed as noise, as can be seen in figure 45.

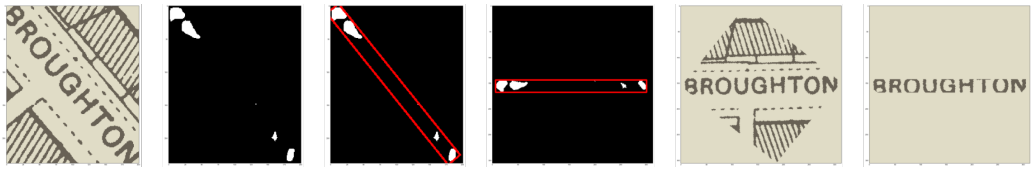


Figure 45: The cropped BROUGHTON text bounding box and mask predicted by the Mask R-CNN model where the red box is the minimum bounding rectangle. The cropped box is rotated using the minim bounding boxes rotation and image noise is removed above and below the mask.

The above method was applied to the map image in figure 39 and the results can be seen in figure 46.

Figure 46: The OCR method applied to the predicted text bounding boxes from figure 39. The first column is the predicted text masks. The second is the cropped predicted bounding boxes. The third column shows the images after their orientations are adjusted and noise is removed. Their predicted text results using pytesseract can be seen above each final image.

6.4.3 CNN for Text Rotation Prediction

As seen in the previous subsection, using the minimum area rectangle to find the rotation of the text can either be extremely accurate, or be inaccurate by 90 degrees. Another method of finding the rotation of the text was considered. This method involve training a CNN classifier to predict the text rotation. In section 6.3, we create a dataset of images with corresponding annotation XML files which included each texts bounding box and rotation. Thus, to make a training dataset for the classifier, we used the bounding boxes to crop out text images and record their rotations. The images were then rescaled to fit in a 256×256 image whilst preserving the same aspect ratio. An example of some training images can be seen in figure 47.

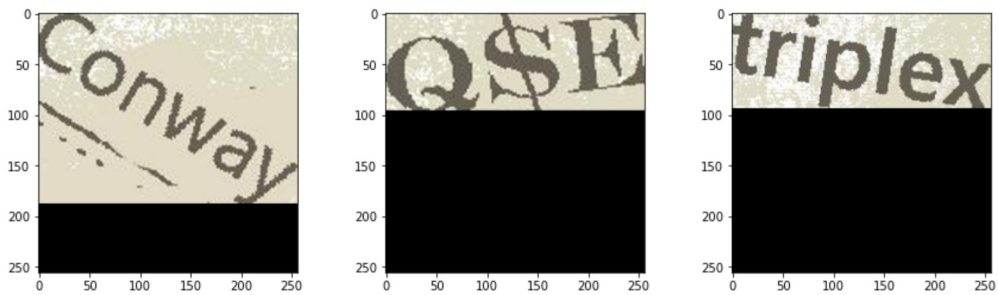


Figure 47: Example of rotated text from our generated images cropped using the dataset annotation XML file and resizes to 256×256 whilst keeping the same aspect ratio.

We created a model which utilised: a base ResNet50 [15] feature extractor; a batch normalization layer; a global average pooling layer; a 0.6 dropout layer and finally a single output neuron with a tanh activation function. We then proceeded to train the model using the EAST rotational loss function as seen in section 5.2.1 which is defined as

$$L_\theta(\hat{\theta}, \theta) = 1 - \cos(\hat{\theta} - \theta) \tag{6.1}$$

where $\hat{\theta}$ is the predicted angle of rotation, θ is the ground truth.

The fully trained model performed fairly well on a test dataset, with an average absolute difference of 13.36 degrees between the true and predicted rotations. Furthermore, if we were to use the sign of the predicted rotation to infer the direction of the text, the model had an accuracy of 84.79%. Unfortunately, this is not high enough to surpass the performance of solely using the minimum area rectangle approach. Although, more work could be done to improve upon the model and target the cases where the minimum area rectangle approach fails most.

6.5 UNet

We have seen in section 6.4 that the Mask R-CNN performs very effectively at detecting instances of text and predicting enough mask information to be able to manipulate the image to perform optical character recognition. However, the Mask R-CNN model doesn't predict enough of the mask for it to be effective at removing the entirety of the text from the image. Due to this, we trained a UNet semantic segmentation model on the same generated data set of 1024×1024 custom images. We trained the model using a dice loss function and a learning rate ranging from 10^{-6} to 10^{-8}. These learning rates were used because the model's convergence on the larger 1024×1024 images was highly dependent on having a small learning rate. A plot of the loss function can be seen in figure 48.

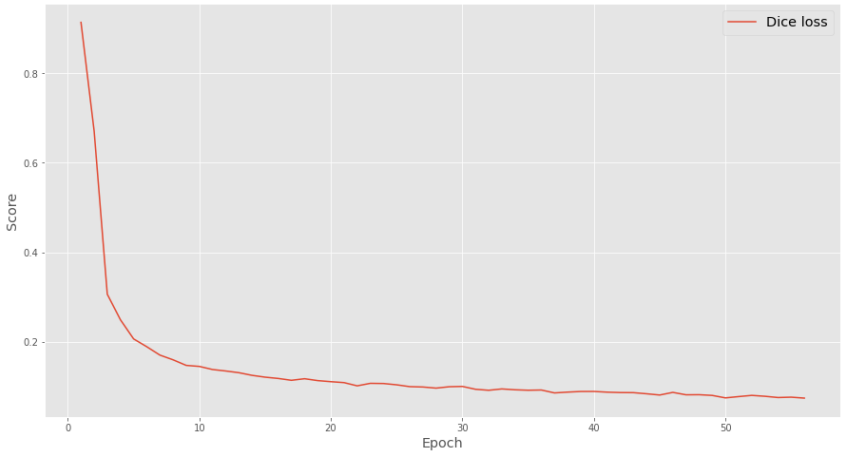


Figure 48: Plot of the dice loss function for the first 56 epochs for the training of a UNet segmentation model.

Once again, we must employ a sliding window method to the trained model to predict larger images with the same accuracy. We can see in figure 49 the trained UNet model greatly outperforms the Mask R-CNN model when it comes to the accuracy of the pixel masks. Similarly to the Mask R-CNN model, however, is that it struggles to recognise the larger overlapping text fonts. Although, this could most likely be addressed by further manipulating the data set.

One way to help the model increase the number of predicted results is to rerun the sliding windowed UNet model on the image at different rotations. An example of rerunning the model at different image rotations can be seen in figure 50. Doing this helps the model pick up a lot of the missed characters at the cost of masking some of the smaller objects with an on-map fixed orientation, such as the trees. Although, this could be adjusted for by utilising different image rotations in the training data.

Figure 49: The sliding window method involving the trained UNet model predictions applied to a larger unseen image.

Figure 50: The combined sliding window method involving the trained UNet model predictions applied to a larger unseen image at different image rotations of -30, -15, 0, 15, 30 degrees.

In order to attempt to remove the larger overlapping text, we modified the data set generator to have a minimum overlap score of 0.23 and a new maximum overlap score of 0.6. We used ten hand-picked similar fonts to the font used for the large word "GREENSIDE" in figure 51, and created a data set of smaller 512×512 images. The UNet model was then trained on this data set. A predicted mask for the large text can be seen in figure 51. Although the retrained model does not fully predict all of the characters pixel masks, it can be seen to perform much better than before. Hence, this shows that with further training and altering of the data set, we can adapt the model to tackle some of the more difficult map text segmentation tasks.

Figure 51: The retrained UNet model for larger overlapping text fonts applied to the difficult GREENSIDE historic text.

6.5.1 Text Recognision with UNet and Mask R-CNN

In section 6.4.2 we showed that the Mask R-CNN model, although struggles to predict masks for large images, predicted enough information to be able to correctly orientate most detected text. This lack of mask accuracy, however, detrimentally affects the image text when we attempt to remove the extra image noise from both above and below the minimum area rectangle. It is difficult to implement a general amount of padding to tackle this problem without, in some cases, introducing more noise.

We can instead use our trained UNet model to more accurately predict the characters of text. This is achieved by cropping the predicted text bounding box, generated from the Mask R-CNN model, out of the predicted mask from the UNet model. We can see in figure 52 that this method improves upon the previous method of just using the mask generated from the Mask R-CNN model. It is also clear from figure 52 that it would be advantageous to combine the masks of both models.

Figure 52: An adaption of figure 46. The fourth column shows the text masks as predicted by the trained UNet model. The fifth column shows the predicted images after their orientations are adjusted and extra noise is remove using the new predicted masks combined with their new predicted text using pytesseract.

7 Conclusions

In this project, we aimed to generate a dataset able to train an object detector and segmentation deep learning model, to aid text recognition. By using a novel approach of generating a custom dataset of historic map text, we have successfully addressed the outlined goals and were able to train multiple deep learning models. The efficiency of this approach is evidenced in section 6.4.1, which shows that the Mask R-CNN model was able to not only identify 99.12% instances of text in an image but also correctly encapsulate 87.72% of all text within the image inside bounding boxes. Programmatically generating a dataset in this manner allows us to train a vast variety of deep learning models, with a comparatively minute amount of manual labour and minimal risk of overfitting the training data. In addition to this, because of how the dataset is generated, the parameters, such as the style of fonts and text sizes, could be further refined or expanded to target specific or more general text. This highlights the adaptability and flexibility of this approach. Future work could also be carried out to identify the best choice of words to sample from a list. Although in our case using an English dictionary based approach worked extremely successfully, the dictionary may not contain the same distribution of characters such as vowels, consonants and numbers as can be seen in historic maps. A possible improvement to this could be to utilise a sample list of words created based on a comprehensive list of old and new street names which is readily available online. This may improve the number of accurate bounding boxes and mask predictions.

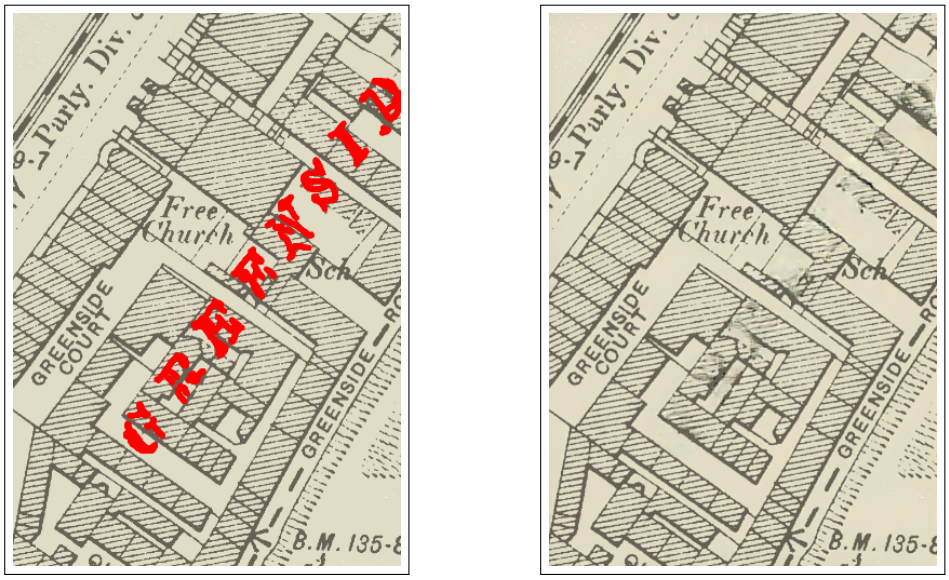
The choice of a Mask R-CNN model worked extremely effectively. Not only did it localise text instances with a very high degree of accuracy, it also correctly predicted sections of the text masks. Even though the model did not identify the entirety of the text masks, especially for longer words, the masks still held enough information such that we could utilise a minimum area rectangle approach to find the rotation of the text's bounding box. Future work may benefit from further training of the Mask R-CNN model to try and improve on the model's mask prediction. It may also be beneficial to utilise a different feature extractor. This is because the ResNet feature extractor utilised by the Mask R-CNN in this project, starts by applying a 7×7 convolutional layer. This is a very large convolutional filter and using it may result in a large amount of information loss at the beginning of the feature extraction process. Therefore a different feature extractor may be able to overcome this issue of information loss and be better suited for historical map image segmentation.

In addition to improving mask prediction of the Mask R-CNN, we identify that being able to correctly refine the predicted bounding boxes is also an important task. Bounding boxes that do not encapsulate words in their entirety result in only segments of words being recognised and located. This research showed that using solely non-maximum suppression on the predicted bounding boxes sometimes resulted in the most accurate bounding box being suppressed. Instead, a method of merging predicted boxes could be investigated. One idea would be to apply the minimum area rectangle method to establish the text's rotation, then apply a clustering method to merge those bounding boxes with similar spatial and rotational coordinates. This method may also be useful for combining bounding boxes for street name text in particular. This is because they are typically situated close together in the same orientation.

Compared to the Mask R-CNN, the trained UNet model showed exceptional accuracy in the prediction of most text masks. These masks were then shown to further aid in the removal of non-text noise in the bounding box images. This process increases the effectiveness of the Tesseract OCR method. Once again, it is likely that by refining the dataset generator we could obtain higher segmentation accuracy. This is supported by our attempt in section 6.5 which showed that, by using larger font types, we were able to greatly improve the predictive accuracy of the UNet model on the difficult task of predicting the masks of large text fonts overlapping buildings. In this case however, predicting the masks of large text when it overlaps important map features is the most difficult problem to overcome for text segmentation in historic maps.

This is because large text can sometimes be difficult to differentiate from other structures such as buildings. Therefore, it may be beneficial to complement our dataset by additionally, manually creating a selection of real large text mask images or by restricting the dictionary of words to only contain text of the large fonts.

As well as being a tool to aid text detection, the predicted UNet text masks could also be used for text removal. Removing the text by using a pixel-level mask minimises the amount of damage, and thus, the amount of repair needed for the historic map once text has been removed. Investigation into text removal via inpainting, may be the next logical step following on from the work done in this project. Inpainting is the method of removing items from a digital image and conserving damaged areas by filling them in, with the aim of presenting a complete image [5]. An investigation into trainable inpainting methods, such as the method of Partial Convolutions [26], is recommended. This is because, we can utilise our dataset generator to obtain text images and text masks to train the inpainting method to remove text from maps whilst preserving any underlying features. Although inpainting is outside of the scope of the project at hand, a demonstration of using an online implementation of a Partial Convolutions inpainting method [14], trained on the ImageNet dataset, can be seen in figure 53. Even though this method was trained on a dataset of natural scene images, this method of shows signs of being able to preserve the fundamental image structure of the historic maps.



(a) (b)

Figure 53: (*a*) The GREENSIDE text map image with the text mask manually draw onto the image in red. (*b*) The resulting inpainted image via the Partial Convolutions online implementation [14].

Another aspect of the project that proved to be successful was using a template pattern matching approach to calculate the density of greenspace, in a historic map, based on tree symbols. In section 6.1 we show by utilising template matching in a historic map image that 1497 template matches were found and only 34 of these were falsely identified. Even though this an error of only 2.27%, further investigation could be carried out to train a CNN classifier to target the removal of false positive matches. This could be carried out by slightly modifying our dataset generator to overlay tree symbols instead of text. However, we would still require a treeless historic map image to overlay the tree symbols onto. Instead of manually removing the tree symbols, much like we did with the text, we could lower the acceptance threshold of the template matching algorithm such that it matches all trees (plus false positives), then sample our training set from the rest of the map image. In addition to improving the already impressive accuracy of the template matching algorithm, it would be interesting to find the greenspace

density across different historic maps in the same area. Thus, building a visualisation of the rate and change in urbanisation.

In conclusion, we have constructed a novel method for automatically generating a dataset of, but not limited to, historic map text. By using the generated dataset, we have successfully trained a Mask R-CNN and UNet deep learning models to be able to both detect text and their masks. We have also demonstrated that this information can be used in conjunction with existing OCR methods to recognise the text, as well as potentially aiding a wide range of future projects in the removal of text.

References

[1] Waleed Abdulla. *Mask R-CNN for object detection and instance segmentation on Keras and TensorFlow*. https://github.com/matterport/Mask_RCNN. 2017.

[2] Wilhelm Burger and Mark J. Burge. *Digital Image Processing: An Algorithmic Introduction Using Java*. 2nd. Springer Publishing Company, Incorporated, 2016. ISBN: 1447166833.

[3] B K Choudhary et al. "Pyramid Method in Image Processing". In: (Jan. 2012).

[4] W De Breuck and P Van Burm. "The use of historic maps for the preparation of engineering geological maps in Flanders (Belgium)". In: *Bulletin of the International Association of Engineering Geology-Bulletin de l'Association Internationale de Géologie de l'Ingénieur* 21.1 (1980), pp. 99–103.

[5] Omar Elharrouss et al. "Image Inpainting: A Review". In: *Neural Processing Letters* 51.2 (Dec. 2019), pp. 2007–2028. ISSN: 1573-773X. DOI: 10.1007/s11063-019-10163-0. URL: http://dx.doi.org/10.1007/s11063-019-10163-0.

[6] P. F. Felzenszwalb et al. "Object Detection with Discriminatively Trained Part-Based Models". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 32.9 (2010), pp. 1627–1645.

[7] Lucas Fidon et al. "Generalised Wasserstein Dice Score for Imbalanced Multi-class Segmentation Using Holistic Convolutional Networks". In: *Lecture Notes in Computer Science* (2018), pp. 64–76. ISSN: 1611-3349. DOI: 10.1007/978-3-319-75238-9_6. URL: http://dx.doi.org/10.1007/978-3-319-75238-9_6.

[8] Chris Fleet. *Mapping Slavery*. Nov. 2016. URL: https://blog.nls.uk/mapping-slavery/ (visited on 08/18/2020).

[9] Aurélien Géron. *Hands-on machine learning with Scikit-Learn and TensorFlow : concepts, tools, and techniques to build intelligent systems*. Sebastopol, CA: O'Reilly Media, 2017. ISBN: 978-1491962299.

[10] Ross Girshick. *Fast R-CNN*. 2015. arXiv: 1504.08083 [cs.CV].

[11] Ross Girshick et al. *Rich feature hierarchies for accurate object detection and semantic segmentation*. 2013. arXiv: 1311.2524 [cs.CV].

[12] Ross Girshick et al. *Rich feature hierarchies for accurate object detection and semantic segmentation*. 2013. arXiv: 1311.2524 [cs.CV].

[13] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. The MIT Press, 2016. ISBN: 0262035618.

[14] Mathias Gruber. *FixMyPhoto.ai*. URL: https://www.fixmyphoto.ai/' (visited on 08/01/2020).

[15] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV].

[16] Kaiming He et al. *Mask R-CNN*. 2017. arXiv: 1703.06870 [cs.CV].

[17] Sergey Ioffe and Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. arXiv: 1502.03167 [cs.LG].

[18] Ionut Iosifescu, Angeliki Tsorlini, and Lorenz Hurni. "Towards a comprehensive methodology for automatic vectorization of raster historical maps". In: *e-Perimetron* 11 (June 2016), pp. 57–76.

[19] D. Kalina and R. Golovanov. "Application of Template Matching for Optical Character Recognition". In: *2019 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus)*. 2019, pp. 2213–2217.

[20] Christopher Kanan and Garrison Cottrell. "Color-to-Grayscale: Does the Method Matter in Image Recognition?" In: *PloS one* 7 (Jan. 2012), e29740. DOI: 10.1371/journal.pone.0029740.

[21] Krishnendu Kar. *Mastering Computer Vision with TensorFlow 2.x. Build advanced computer vision applications using machine learning and deep learning techniques.* Packt Publishing Ltd, 2020. ISBN: 1838827064.

[22] Çağrı Kaymak and Ayşegül Uçar. "A Brief Survey and an Application of Semantic Image Segmentation for Autonomous Driving". In: Aug. 2018, p. 1. DOI: 10.1007/978-3-030-11479-4_9.

[23] F. Keller. *Convolutions and Kernels.* Feb. 2010. URL: http://www.inf.ed.ac.uk/teaching/courses/cfcs1/lectures/cfcs_l15.pdf (visited on 08/16/2020).

[24] Y. Lecun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[25] Tsung-Yi Lin et al. *Microsoft COCO: Common Objects in Context.* 2014. arXiv: 1405.0312 [cs.CV].

[26] Guilin Liu et al. *Image Inpainting for Irregular Holes Using Partial Convolutions.* 2018. arXiv: 1804.07723 [cs.CV].

[27] University College London. *Legacies of British Slave-ownership.* URL: https://www.ucl.ac.uk/lbs/maps/caribbean/jamaica#zoom=10&lng=-77.746124&lat=18.181083 (visited on 08/18/2020).

[28] MachineCurve. *ReLU, Sigmoid and Tanh: today's most used activation functions.* Sept. 2019. URL: https://www.machinecurve.com/index.php/2019/09/04/relu-sigmoid-and-tanh-todays-most-used-activation-functions (visited on 07/28/2020).

[29] Warren Mcculloch and Walter Pitts. "A Logical Calculus of Ideas Immanent in Nervous Activity". In: *Bulletin of Mathematical Biophysics* 5 (1943), pp. 127–147.

[30] missinglink.ai. *7 Types of Neural Network Activation Functions: How to Choose?* URL: https://bit.ly/39GWkg3 (visited on 07/28/2020).

[31] N. Otsu. "A Threshold Selection Method from Gray-Level Histograms". In: *IEEE Transactions on Systems, Man, and Cybernetics* 9.1 (1979), pp. 62–66.

[32] Henryk Palus. "Color Image Segmentation". In: Oct. 2006, pp. 103–128. DOI: 10.1201/9781420009781.ch5.

[33] Je-Kang Park and Dong Kang. "Unified convolutional neural network for direct facial keypoints detection". In: *The Visual Computer* (May 2018). DOI: 10.1007/s00371-018-1561-3.

[34] *Perceptrons: An Introduction to Computational Geometry.* The MIT Press, 1969. ISBN: 0262630222.

[35] William K. Pratt. *Digital Image Processing: PIKS Scientific Inside.* USA: Wiley-Interscience, 2007. ISBN: 0471767778.

[36] Prajit Ramachandran, Barret Zoph, and Quoc V. Le. "Searching for Activation Functions". In: *CoRR* abs/1710.05941 (2017). arXiv: 1710.05941. URL: http://arxiv.org/abs/1710.05941.

[37] Neha Rani and Sharda Vashisth. "Brain Tumor Detection and Classification with Feed Forward Back-Prop Neural Network". In: *International Journal of Computer Applications* 146 (July 2016), pp. 1–6. DOI: 10.5120/ijca2016910738.

[38] Shaoqing Ren et al. *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks.* 2015. arXiv: 1506.01497 [cs.CV].

[39] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. *U-Net: Convolutional Networks for Biomedical Image Segmentation.* 2015. arXiv: 1505.04597 [cs.CV].

[40] Rasmus Rothe, Matthieu Guillaumin, and Luc Van Gool. "Non-Maximum Suppression for Object Detection by Passing Messages between Windows". In: vol. 9003. Apr. 2015. DOI: 10.1007/978-3-319-16865-4_19.

[41] Jesus Ruiz-Santaquiteria et al. "Semantic versus instance segmentation in microscopic algae detection". In: *Engineering Applications of Artificial Intelligence* 87 (Jan. 2020), p. 103271. DOI: 10.1016/j.engappai.2019.103271.

[42] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors". In: *Nature* 323.6088 (Oct. 1986), pp. 533–536. URL: http://dx.doi.org/10.1038/323533a0.

[43] Olga Russakovsky et al. *ImageNet Large Scale Visual Recognition Challenge*. 2014. arXiv: 1409.0575 [cs.CV].

[44] Pierre Sermanet et al. "OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks". In: *International Conference on Learning Representations (ICLR) (Banff)* (Dec. 2013).

[45] M. Sezgin and Bulent Sankur. "Survey over image thresholding techniques and quantitative performance evaluation". In: *Journal of Electronic Imaging* 13 (Jan. 2004), pp. 146–168. DOI: 10.1117/1.1631315.

[46] Ray Smith. "An Overview of the Tesseract OCR Engine". In: *ICDAR '07: Proceedings of the Ninth International Conference on Document Analysis and Recognition*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 629–633. ISBN: 0-7695-2822-8. URL: http://www.google.de/research/pubs/archive/33418.pdf.

[47] Jost Tobias Springenberg et al. *Striving for Simplicity: The All Convolutional Net*. 2014. arXiv: 1412.6806 [cs.LG].

[48] P. Syms. *Land, Development and Design*. 2nd ed. John Wiley Sons, 2010.

[49] TensorFlow. *Normalizations*. June 2020. URL: https://www.tensorflow.org/addons/tutorials/layers_normalizations (visited on 08/01/2020).

[50] *tesseract*. URL: https://github.com/tesseract-ocr/tesseract (visited on 08/11/2020).

[51] J. Vitor. *Evaluation of a Python algorithm for parallel convolution*. July 2017. URL: https://jeanvitor.com/convolution-parallel-algorithm-python/ (visited on 08/16/2020).

[52] Zeya Wang et al. "Ellipse Detection of Optic Disc-and-Cup Boundary in Fundus Images". In: Apr. 2019, pp. 601–604. DOI: 10.1109/ISBI.2019.8759173.

[53] Linjie Xing et al. *Convolutional Character Networks*. 2019. arXiv: 1910.07954 [cs.CV].

[54] Chen-Kuei Yang and Wen-Hsiang Tsai. "Color Image Compression Using Quantization, Thresholding, and Edge Detection Techniques All Based on the Moment-Preserving Principle". In: *Pattern Recogn. Lett.* 19.2 (Feb. 1998), pp. 205–215. ISSN: 0167-8655. DOI: 10.1016/S0167-8655(97)00166-9. URL: https://doi.org/10.1016/S0167-8655(97)00166-9.

[55] Zhong-Qiu Zhao et al. *Object Detection with Deep Learning: A Review*. 2018. arXiv: 1807.05511 [cs.CV].

[56] Xinyu Zhou et al. *EAST: An Efficient and Accurate Scene Text Detector*. 2017. arXiv: 1704.03155 [cs.CV].

8 Appendix

8.1 Computer Specifications

CPU - Intel i7 Six Core Processor i7-8700k (3.7GHz) 12MB Cache
GPU - 8G RTX 2070
Motherboard - ASUS ROG STRIX Z370-E GAMING
Memory (RAM) - 32GB Corsair VENGEANCE DDR4 3000MHz (4 x 8GB)
Operating System - Windows 10 Professional 64 Bit

8.2 Custom Dataset Generator Class

```
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image, ImageDraw, ImageFont, ImageOps
Image.MAX_IMAGE_PIXELS = None
import os
import glob
import random
import string
import requests
import xml.etree.cElementTree as ET
from joblib import Parallel, delayed
import multiprocessing
import warnings


class DataGen:

    def __init__(self, imageFolderPath, newFolderPath, newFolderName,
                                            subImageSize, overlapThreshold=0.23,
                                             numTextProposals=50 ):
        """A class for a custom text dataset generator.

         Parameters:
          imageFolderPath (str): Path to an image folder containing K3 clustered
                                                 maps with text removed.
          newFolderPath (str): Path  to the desired location of the generated
                                                dataset.
          newFolderName (str): Name of the desired generated dataset folder.
          subImageSize (int,int): Size of each subsampled image.
          overlapThreshold (float): Maximum amount of text-building overlap.
          numTextProposals (int): Number of proposed text instances.
        """
        self.imgFoldPath = imageFolderPath
        self.newFoldPath = newFolderPath + newFolderName
        self.subImgSize = subImageSize
        self.threshold = overlapThreshold
        self.numProposals = numTextProposals
        self.fonts = ['helveticaneue medium_2.ttf', 'century.ttf','alger.ttf','
                                              brlnsr.ttf','brlnsdb.ttf','
                                              schlbkb.ttf',
                      'CALIBRIB.ttf','PERB____.TTF', 'BELLB.TTF','GOUDOSB.TTF','
                                                        BASKVILL.TTF','
                                                        georgiab.ttf','
                                                        georgia.ttf',
                      'seguisb.ttf','segoeui.ttf','engr.ttf', 'IMPRISHA.TTF', '
                                                        CASTELAR.TTF', '
                                                        UBUNTU-M.TTF','
                                                        UBUNTU-MI.TTF',
                      'UBUNTU-B.TTF', 'ARLRDBD.TTF', 'GILLUBCD.TTF']

        self.CheckFonts()
        self.TrainOrTest = 'Train'
```

```
        self.OpenDictionary()


    def CheckFonts(self):
      """Checks the avaible of system fonts. Fonts are removed if not found."""

        availableFonts = []
        for font in self.fonts:
            try:
                ImageFont.truetype('c:/windows/fonts/'+ font, 1)
                availableFonts.append(font)
            except:
                warnings.warn('Font {0} not found so was removed from the list
                                                        of avaible fonts.'.
                                                        format(font))
        if not availableFonts:
            raise Exception('No avaible fonts found.')

        self.fonts = availableFonts


    def OpenDictionary(self):
      """Creates a list of words to be sampled from."""
        word_site = "http://svnweb.freebsd.org/csrg/share/dict/words?view=co&
                                                content-type=text/plain"
        response = requests.get(word_site)
        WORDS = response.content.splitlines()
        self.WORDS = [word.decode('utf-8') for word in WORDS]
        self.WORDS += [' '.join(self.WORDS[i]) for i in range(0,25000, 5) ]
        self.WORDS += ['  '.join(self.WORDS[i]) for i in range(0,25000, 8) ]
        self.WORDS += list(string.ascii_letters + string.digits)
        self.WORDS += [''.join(random.choices(string.digits, k=random.randint(1,
                                                5))) for _ in range(4000)]
        self.WORDS += [''.join(random.choices(string.ascii_letters + string.
                                                digits, k=random.randint(1,5)))
                                                for _ in range(2000)]




    def CreatePaths(self):
      """Creates the test/train folders."""
        for path in [r'\Train', r'\Test', r'\TrainAnn', r'\TestAnn']:
            for ex in [r'\Char',r'\Map']:
                if not os.path.exists(self.newFoldPath+path+ex):
                    os.makedirs(self.newFoldPath+path+ex)


    def getRandomCropBox(self, W, H):
      """Gets a random sub-box within a larger box.

      Parameters:
        W (int): Width of the larger box.
        H (int): Height of the larger box.

      Returns:
        Box coordinates for the random box.
      """
        left = random.randint(0,W-self.subImgSize[0])
        top = random.randint(0,H-self.subImgSize[1])
        return [left, top, left+self.subImgSize[0], top+self.subImgSize[1]]


    def checkIntersects(self, a, b):
      """Checks if two boxes interect.
```

```
    Parameters:
      a (int,int,int,int): Box coordinates of the first box.
      b (int,int,int,int): Box coordinates of the second box.

    Returns:
      Boolean value of if the boxes overlap.
    """
      x1,y1,xx1,yy1 = a
      x2,y2,xx2,yy2 = b

      return not (xx1 < x2 or x1 > xx2 or yy1 < y2 or y1 > yy2)


  def getBoundingBox(self, mask):
    """Returns the bounding box of an input max.

    Parameters:
      mask (ndarray): 2D array integer of mask where 0 represents no mask.

    Returns:
      Returns the bounding box coordinates of the mask and the mask restricted
                                          to this bounding box.
    """
      indexes = np.where(mask>0)
      mask_ = mask.copy()
      mask_[indexes] = 1

      y, x =  indexes
      xmin, xmax = (x.min(), x.max())
      ymin, ymax = (y.min(), y.max())

      # return mask of bounding box
      return mask_[ymin:ymax, xmin:xmax], xmin, ymin, xmax, ymax


  def createXML(self, wordTupArray, filename):
      """Creates an XML file containing containing the annotation information
                                                of the generated images.

  Parameters:
    wordTupArray (ndarray, (int,int,int,int), float, string, float, ndarray, (
                                      int,int,int,int)): Tuple
      list containing the text mask, text bounding box, text rotation, text
                                      string, overlap score, restricted
      text mask.
    filename (str): Name of the XML file.
      """
      root = ET.Element("annotation")
      ET.SubElement(root, "Filename").text = filename

      for wordTup in wordTupArray:
          self.createObject(root, wordTup)

      tree = ET.ElementTree(root)
      tree.write(self.newFoldPath + '\\'+self.TrainOrTest+'Ann\\Char\\' +
                                                filename.split('.')[0] + '.xml')


  def createObject(self, root, wordTup):
    """Creates an object in the root XML file for a text instance.

    Parameters:
    root (cElementTree): XML element tree.
    wordTupArray (ndarray, (int,int,int,int), float, string, float, ndarray, (
                                      int,int,int,int)): Tuple
```

```
        containing the text mask, text bounding box, text rotation, text string,
                                        overlap score, restricted
        text mask.
    """
        _, _, rot, text, score, mask, bbox = wordTup
        xmin, ymin, xmax, ymax = bbox

        obj = ET.SubElement(root, "object")

        mask_flat = mask.astype(np.uint8).flatten()
        mask_str = ''.join(map(str, mask_flat))

        ET.SubElement(obj, "Mask").text = mask_str
        ET.SubElement(obj, "Text").text = text
        ET.SubElement(obj, "Rotation").text = str(rot)
        ET.SubElement(obj, "OverlapScore").text = str(score)

        bb = ET.SubElement(obj, "bndbox")

        ET.SubElement(bb,"xmin").text = str(xmin)
        ET.SubElement(bb,"ymin").text = str(ymin)
        ET.SubElement(bb,"xmax").text = str(xmax)
        ET.SubElement(bb,"ymax").text = str(ymax)


    def proposeWords(self, img):
        """Proposes text instances to be overlayed onto an image.

        Parameters:
        img (PIL.Image): Image.

        Returns:
        A tuple of information about the proposed generated images and text.
        """

        width, height = img.size
        img = np.array(img)
        masks = []

        for i in range(self.numProposals):
            font = random.choice(self.fonts)
            fontSize = random.randint(35,95) # was 55
            fnt = ImageFont.truetype('c:/windows/fonts/'+ font, fontSize)
            word = random.choice(self.WORDS)
            rot = random.uniform(-89, 89) #np.clip(np.random.normal()*40, -80,
                                                    80)

            w,h = ImageDraw.Draw(Image.new('RGB', (0,0), (0,0,0))).textsize(word
                                                    , fnt)
            text_mask = Image.new('RGBA', (w,h), (0,0,0,0))
            d = ImageDraw.Draw(text_mask)


            w, h = d.textsize(word, font=fnt)
            d.text((0, 0), word, font=fnt, fill=tuple([104, 97, 85]))
            text_mask = np.array(text_mask.rotate(rot, expand=True))

            text_mask[text_mask[:,:,3]>=[50]]=[104,97,85,255]
            text_mask[text_mask[:,:,3]<[50]]=[0,0,0,0]

            text_mask = Image.fromarray(text_mask)
            w,h = text_mask.size

            if w > width or h > height:
                continue
```

```
            xcord = random.randint(0,width-w)
            ycord = random.randint(0,height-h)

            text_mask = np.array(text_mask)

            temp = img[ycord:ycord+h,xcord:xcord+w,:]
            score = np.sum((text_mask[:,:,0]==104)*(temp[:,:,0]==104))/np.sum((
                                                    text_mask[:,:,0]==104))

            binary_mask, xmin, ymin, xmax, ymax = self.getBoundingBox(text_mask[
                                                    :,:,0])

            text_mask = Image.fromarray(text_mask)
            masks.append((text_mask, (xcord, ycord, w, h), rot, word, score,
                                                    binary_mask,
                        (xmin+xcord,ymin+ycord,xmax+xcord,ymax+ycord,) ))

        if len(masks)==0:
            raise ValueError('No generated text could fit on image. Try
                                                    decreasing fontsize or
                                                    limiting word length.')

        return masks


    def refineWords(self, masks):
      """Refines the text proposals by comparing overlap scores and text box
                                            overlapings.

      Parameters:
        masks (ndarray, (int,int,int,int), float, string, float, ndarray, (int,
                                            int,int,int)): Tuple
        containing the text mask, text bounding box, text rotation, text string,
                                        overlap score, restricted
        text mask.

    Returns:
        A tuple of information about the refined generated images and text.
      """
        masks = sorted(masks, key=lambda x: x[4],reverse=False)
        masks = [masks[0]] + [masks[i] for i in range(1,len(masks)) if masks[i][
                                                4]<self.threshold]
        final_masks = []

        for i, mask in enumerate(masks):
            if i==0:
                final_masks.append(mask)
                continue
            overlaps = False
            for f_mask in final_masks:
                if self.checkIntersects(mask[6], f_mask[6]):
                    overlaps = True
            if not overlaps:
                final_masks.append(mask)

        return final_masks


    def GenerateSample(self, img):
      """Generates overlays text on a sample image.

      Parameters:
        img (PIL.Image): Image.
      """
        img_ = img.copy()
        masks = self.proposeWords(img_)
```

```
        masks = self.refineWords(masks)
        seg = np.zeros(img_.size)
        plt.style.use('ggplot')

        for mask, cords, rot, word, score, text_mask, bbox in masks:

            img_.paste(mask,cords[:2],mask)
            d = ImageDraw.Draw(img_)
            d.rectangle(bbox, outline='red' )
            seg[bbox[1]:bbox[3], bbox[0]:bbox[2]][text_mask>0] = 1

        plt.figure(figsize=(10,10))
        plt.imshow(img_)
        plt.show()
        plt.figure(figsize=(10,10))
        plt.imshow(seg,cmap='gray')
        plt.show()


    def OverlayCharacters(self, img, filename, generateXML=True):
      """Overlays text onto an image and generates the annotation XML file.

      Parameters:
        img (PIL.Image): Image.
        filename (string): String name of the image file.
        generateXML (Bool): True/False should the image annotation XML file be
                                                made.

      Returns:
        Returns the input image with the overlayed text on it.
      """
        masks = self.proposeWords(img)
        masks = self.refineWords(masks)

        if generateXML:
            self.createXML(masks, filename)

        for mask, cords, _, _, _, _, _ in masks:
            img.paste(mask,cords[:2],mask)

        return img


    def Generate(self, imagePath, num):
      """Generates a number of image data for a given map image.

      Parameters:
        imagePath (string): The large map image file path.
        num (int): Large image reference number used for parallel data
                                                generation.
      """
        Image.MAX_IMAGE_PIXELS = None
        mapC = 0
        charC = 0

        img = Image.open(imagePath)
        w,h = img.size

        # generate no text images
        for _ in range(self.numPerImage):
            break # comment out for negative textless sample generations
            filename = str(num)+'_'+str(mapC).zfill(8) + '.png'
            savePath = self.newFoldPath + r'\\{0}\Map\\{1}'.format(self.
                                                        TrainOrTest, filename)
            img.crop(self.getRandomCropBox(w,h)).save(savePath)
            mapC += 1
```

```
        # generate images with text overlay
        for _ in range(self.numPerImage):
            filename = str(num)+'_'+str(charC).zfill(8) + '.png'
            savePath = self.newFoldPath + r'\\{0}\Char\\{1}'.format(self.
                                                    TrainOrTest, filename)
            newImg = img.crop(self.getRandomCropBox(w,h))
            self.OverlayCharacters(newImg, filename ).save(savePath)
            charC += 1


    def GenerateDataSet(self, numImages, TrainOrTest='Train'):
      """Generates a dataset.

      Parameters:
        numImages (int): Number of image samples required for the dataset.
        TrainOrTest (string): Determines if the images go into the test or train
                                                directory
      """
        self.TrainOrTest = 'Train'
        if TrainOrTest.lower() == 'test':
            self.TrainOrTest = 'Test'

        imagePaths = glob.glob(self.imgFoldPath+r'\*')
        numPaths = len(imagePaths)
        self.numPerImage = numImages//numPaths

        for num in range(numImages):
            self.Generate(imagePaths[num], num)


    def ParallelGenerateDataSet(self, numImages, TrainOrTest='Train', num_cores=
                                                0):
      """Generates the dataset in parallel utilising a different large map image
                                                instance on each core.

      Parameters:
        numImages (int): Number of image samples required for the dataset.
        TrainOrTest (string): Determines if the images go into the test or train
                                                directory
        num_cores (int): The number of cores you wish to utilise.
      """
        self.TrainOrTest = 'Train'
        if TrainOrTest.lower() == 'test':
            self.TrainOrTest = 'Test'

        imagePaths = glob.glob(self.imgFoldPath+r'\*')

        if num_cores==0:
            num_cores = multiprocessing.cpu_count()

        # look at the same image on different cores without introducing image
                                                    bias
        wastedCores = num_cores // len(imagePaths)
        imagePaths = imagePaths * wastedCores

        self.numPerImage = numImages // len(imagePaths)

        Parallel(n_jobs=num_cores)(delayed(self.Generate)(path,n) for n, path in
                                                    enumerate(imagePaths))


if __name__=='__main__':
  IFP = '../maps/RemovedText'
  NFP = '../maps'
```

```
DG = DataGen(IFP, NFP, '/Dataset_folder', (1024,1024))
DG.CreatePaths()
DG.ParallelGenerateDataSet(30000, 'Train', 12)
DG.ParallelGenerateDataSet(9000, 'Test', 12)
```

8.3 Create Masks from XML files

```
from glob import glob
from xml.etree import ElementTree
from PIL import Image
import cv2
import matplotlib.pyplot as plt
import numpy as np

def extract_boxes(filename):
    # load and parse the file
    tree = ElementTree.parse(filename)
    # get the root of the document
    root = tree.getroot()

    # extract each bounding box
    boxes = list()
    masks = list()
    for obj in root.findall('.//object'):
        xmin = int(obj.find('.//bndbox//xmin').text)
        xmax = int(obj.find('.//bndbox//xmax').text)
        ymin = int(obj.find('.//bndbox//ymin').text)
        ymax = int(obj.find('.//bndbox//ymax').text)
        coors = [xmin, ymin, xmax, ymax]
        boxes.append(coors)
        masks.append(np.array(list(obj.find('Mask').text)).astype(np.uint8))

    return boxes, masks

def combine_masks(path, boxes, masks, size=(512,512)):
    h, w = size
    mask = np.zeros([h, w], dtype='uint8')

    for i in range(len(boxes)):
        box = boxes[i]
        row_s, row_e = box[1], box[3]
        col_s, col_e = box[0], box[2]

        mask[row_s:row_e, col_s:col_e] = masks[i].reshape(row_e-row_s,col_e-
                                                    col_s)

    return mask

annotations_dir = '\\maps\\DataSet512\\TrainAnn\\Char'
masks_dir = r'\\maps\\DataSet512\\TrainMasks\\Char'
for f in glob(annotations_dir+r'\*'):
    boxes, masks = extract_boxes(f)
    mask = combine_masks(f, boxes, masks)
    Image.fromarray(mask.astype(np.uint8), 'L').save(masks_dir+ '//' + f.split('
                                                \\')[-1].split('.')[0] + '.png')
```

8.4 UNet

```
import numpy as np
import os
import skimage.io as io
import skimage.transform as trans
import numpy as np
from keras.models import *
from keras.layers import *
from keras.optimizers import *
from keras.callbacks import ModelCheckpoint, LearningRateScheduler
from keras import backend as keras
from keras.preprocessing.image import ImageDataGenerator
import tensorflow as tf

config = tf.ConfigProto()
config.gpu_options.allow_growth=True
sess = tf.Session(config=config)


def dice_coef(y_true, y_pred):
    y_true_f = K.flatten(y_true)
    y_pred_f = K.flatten(y_pred)
    intersection = K.sum(y_true_f * y_pred_f)
    return (2. * intersection + 1.) / (K.sum(y_true_f) + K.sum(y_pred_f) + 1.)


def dice_coef_loss(y_true, y_pred):
    return 1-dice_coef(y_true, y_pred)

def unet(pretrained_weights = None,input_size = (256,256,1)):
    inputs = Input(input_size)
    conv1 = Conv2D(64, 3, activation = 'relu', padding = 'same',
                                        kernel_initializer = 'he_normal')(
                                        inputs)
    conv1 = Conv2D(64, 3, activation = 'relu', padding = 'same',
                                        kernel_initializer = 'he_normal')(
                                        conv1)
    pool1 = MaxPooling2D(pool_size=(2, 2))(conv1)
    conv2 = Conv2D(128, 3, activation = 'relu', padding = 'same',
                                        kernel_initializer = 'he_normal')(
                                        pool1)
    conv2 = Conv2D(128, 3, activation = 'relu', padding = 'same',
                                        kernel_initializer = 'he_normal')(
                                        conv2)
    pool2 = MaxPooling2D(pool_size=(2, 2))(conv2)
    conv3 = Conv2D(256, 3, activation = 'relu', padding = 'same',
                                        kernel_initializer = 'he_normal')(
                                        pool2)
    conv3 = Conv2D(256, 3, activation = 'relu', padding = 'same',
                                        kernel_initializer = 'he_normal')(
                                        conv3)
    pool3 = MaxPooling2D(pool_size=(2, 2))(conv3)
    conv4 = Conv2D(512, 3, activation = 'relu', padding = 'same',
                                        kernel_initializer = 'he_normal')(
                                        pool3)
    conv4 = Conv2D(512, 3, activation = 'relu', padding = 'same',
                                        kernel_initializer = 'he_normal')(
                                        conv4)
    drop4 = Dropout(0.5)(conv4)
    pool4 = MaxPooling2D(pool_size=(2, 2))(drop4)

    conv5 = Conv2D(1024, 3, activation = 'relu', padding = 'same',
                                        kernel_initializer = 'he_normal')(
                                        pool4)
```

```
conv5 = Conv2D(1024, 3, activation = 'relu', padding = 'same',
                                        kernel_initializer = 'he_normal')(
                                        conv5)
drop5 = Dropout(0.5)(conv5)

up6 = Conv2D(512, 2, activation = 'relu', padding = 'same',
                                        kernel_initializer = 'he_normal')(
                                        UpSampling2D(size = (2,2))(drop5))
merge6 = concatenate([drop4,up6], axis = 3)
conv6 = Conv2D(512, 3, activation = 'relu', padding = 'same',
                                        kernel_initializer = 'he_normal')(
                                        merge6)
conv6 = Conv2D(512, 3, activation = 'relu', padding = 'same',
                                        kernel_initializer = 'he_normal')(
                                        conv6)

up7 = Conv2D(256, 2, activation = 'relu', padding = 'same',
                                        kernel_initializer = 'he_normal')(
                                        UpSampling2D(size = (2,2))(conv6))
merge7 = concatenate([conv3,up7], axis = 3)
conv7 = Conv2D(256, 3, activation = 'relu', padding = 'same',
                                        kernel_initializer = 'he_normal')(
                                        merge7)
conv7 = Conv2D(256, 3, activation = 'relu', padding = 'same',
                                        kernel_initializer = 'he_normal')(
                                        conv7)

up8 = Conv2D(128, 2, activation = 'relu', padding = 'same',
                                        kernel_initializer = 'he_normal')(
                                        UpSampling2D(size = (2,2))(conv7))
merge8 = concatenate([conv2,up8], axis = 3)
conv8 = Conv2D(128, 3, activation = 'relu', padding = 'same',
                                        kernel_initializer = 'he_normal')(
                                        merge8)
conv8 = Conv2D(128, 3, activation = 'relu', padding = 'same',
                                        kernel_initializer = 'he_normal')(
                                        conv8)

up9 = Conv2D(64, 2, activation = 'relu', padding = 'same',
                                        kernel_initializer = 'he_normal')(
                                        UpSampling2D(size = (2,2))(conv8))
merge9 = concatenate([conv1,up9], axis = 3)
conv9 = Conv2D(64, 3, activation = 'relu', padding = 'same',
                                        kernel_initializer = 'he_normal')(
                                        merge9)
conv9 = Conv2D(64, 3, activation = 'relu', padding = 'same',
                                        kernel_initializer = 'he_normal')(
                                        conv9)
conv9 = Conv2D(2, 3, activation = 'relu', padding = 'same',
                                        kernel_initializer = 'he_normal')(
                                        conv9)
conv10 = Conv2D(1, 1, activation = 'sigmoid')(conv9)

model = Model(input = inputs, output = conv10)

model.compile(optimizer = Adam(lr = 1e-6), loss= dice_coef_loss, metrics=[
                                        dice_coef])


if(pretrained_weights):
    model.load_weights(pretrained_weights)

return model
```

```
image_datagen = ImageDataGenerator()
mask_datagen = ImageDataGenerator()
seed = 1

images_dir = '../maps/DataSet512/Train' # Train image folder
annotations_dir ='../maps/DataSet512/TrainMasks' # Train mask images made from
                                        annotation XML file
images2_dir = '../maps/DataSet512/Test' # Test image folder
annotations2_dir ='maps/DataSet512/TestMasks' # Test mask images made from
                                        annotation XML file

batchsize=3
w = 256*2
image_generator = image_datagen.flow_from_directory(images_dir, target_size=(w,w
                                        ), classes=['Char'], class_mode=None,
                                        seed=seed, batch_size=batchsize,
                                        color_mode='rgb')
mask_generator = mask_datagen.flow_from_directory(annotations_dir, target_size=(
                                        w,w), classes=['Char'], class_mode=None,
                                         seed=seed, batch_size=batchsize,
                                        color_mode='grayscale')
train_generator = zip(image_generator, mask_generator)

hist = model.fit_generator(train_generator, steps_per_epoch=2000//batchsize,
                                        epochs=50)

x,y = 1100,800
w = 512
img = np.asarray(Image.open('../maps/map.tif').crop((x,y,x+w,y+w)).resize((512,
                                        512)))

pred_masks = model.predict(img[np.newaxis,...])
index = 0
plt.figure(figsize=(10,10))
plt.imshow(pred_masks[index,:,:,0],cmap='gray')
plt.show()
plt.figure(figsize=(10,10))
plt.imshow(img)
```

8.5 Text Rotation Classifier

```
import glob
from PIL import Image
from xml.etree import ElementTree
from keras.models import load_model
from sklearn.datasets import load_files
from keras.utils import np_utils
from keras import applications
from keras.preprocessing.image import ImageDataGenerator
from keras import optimizers
from keras.models import Sequential,Model,load_model
from keras.layers import Dense, Dropout, Flatten, Conv2D, MaxPool2D,
                                        GlobalAveragePooling2D, Dense,
                                        BatchNormalization
from keras.callbacks import TensorBoard,ReduceLROnPlateau,ModelCheckpoint
import keras
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf

config = tf.ConfigProto()
config.gpu_options.allow_growth=True
sess = tf.Session(config=config)


img_path = 'maps\\DataSet\\Test\\Char'
an_path = 'maps\\DataSet\\TestAnn\\Char'

class textgen:

    def __init__(self, img_path, ann_path, batchsize=64):
        self.imgs = glob.glob(img_path + r'\*')
        self.ann_path = ann_path
        self.nfiles = len(self.imgs)
        self.bs = batchsize
        self.current = 0

    def generator(self):

        def init():
            return 0

        i = init()
        while True:
            if i + self.bs >= self.nfiles:
                i = init()
                self.current = 0


            imgs = []
            rotations = []
            count  = 0
            for j in range(self.bs):
                if count >= self.bs:
                    i += j
                    break

                image = Image.open(self.imgs[i+j])
                boxes, rots = self.extract_boxes(self.ann_path + '\\' + self.
                                                    imgs[i+j].split('\\')[-1
                                                    ].split('.')[0] + '.xml'
                                                    )

                for index, box in enumerate(boxes):
                    count +=1
```

```
                    n_img = image.crop(box)
                    x = max(n_img.size)
                    n_img = n_img.resize((int(n_img.size[0]/x*256),
                                          int(n_img.size[1]/x*256)))

                    padded = Image.new('RGB', (256,256))
                    padded.paste(n_img, (0,0))

                    imgs.append(np.array(padded))#.resize((256,256)))
                    rotations += [rots[index]]

                    if count >= self.bs:
                        break
            #i += self.bs

            yield (np.array(imgs), np.array(rotations)/180)


    def extract_boxes(self, filename):

        # load and parse the file
        tree = ElementTree.parse(filename)
        # get the root of the document
        root = tree.getroot()

        # extract each bounding box
        boxes = list()
        rotations = list()
        for obj in root.findall('.//object'):
            xmin = int(obj.find('.//bndbox//xmin').text)
            xmax = int(obj.find('.//bndbox//xmax').text)
            ymin = int(obj.find('.//bndbox//ymin').text)
            ymax = int(obj.find('.//bndbox//ymax').text)
            rotations.append(float(obj.find('.//Rotation').text))
            coors = [xmin, ymin, xmax, ymax]
            boxes.append(coors)

        return boxes, rotations # width, height, masks

base_model = applications.resnet50.ResNet50(weights= None, include_top=False,
                                            input_shape= (256,256,3))
x = base_model.output
x = BatchNormalization()(x)
x = GlobalAveragePooling2D()(x)
#x = Flatten()(x)
x = Dropout(0.6)(x)
predictions = Dense(1, activation= 'tanh')(x)
model = Model(inputs = base_model.input, outputs = predictions)

def customLoss(yTrue,yPred):
    return 1-keras.backend.cos(yPred-yTrue)

adam = Adam(lr=0.0001)
model.compile(optimizer= adam, loss=customLoss, metrics=['accuracy'])

gen = textgen(img_path, an_path, 16).generator()
model.fit_generator(gen, epochs = 100,steps_per_epoch=5000//16)
```

8.6 Template Matching

```
import cv2 as cv
import glob
import numpy as np
from matplotlib import pyplot as plt
import random


img_rgb = cv.imread('map.tif') # load in map image
img_gray = cv.cvtColor(img_rgb, cv.COLOR_BGR2GRAY) # convert to grayscale
bboxes = []
templates = glob.glob('templates/*.tif') # load in tree template image paths
colors = [[random.randint(0,255) for i in range(3)] for j in range(len(templates
                                        ))]

for i, template_path in enumerate(templates):
    template = cv.imread(template_path, 0) # load template image
    w, h = template.shape[::-1]
    res = cv.matchTemplate(img_gray,template,cv.TM_CCOEFF_NORMED) # perfrom
                                                  template matching
    threshold = 0.6
    loc = np.where( res >= threshold) # threshold
    for pt in zip(*loc[::-1]):
        bboxes.append([pt[0],pt[1], w, h])
        cv.rectangle(img_rgb, pt, (pt[0] + w, pt[1] + h), colors[i], 2) # draw
                                                     template match on image

cv.imwrite('templateNoNMS.tif', img_rgb) # save image

thresh_boxes = non_max_suppression(boxes, 0.7) # apply a non-max suppression
                                        algorithm
boxes = thresh_boxes.copy()



# Gaussain Density Mask
rows,cols = img_gray.shape
new = np.zeros((rows,cols)) # initialise gaussian mask

kernel =  cv.getGaussianKernel(2001, 400)
gauss = kernel*kernel.T # create 2D gaussian kernel

for i, box in enumerate(thresh_boxes):
    if i %100==0:
        print(i)

    boxw = box[2] - box[0]
    boxh = box[3] - box[1]

    boxCenter = (box[1]  + boxh//2, box[0] + boxw//2) # center gaussian at box
                                                  center
    radius = 1000
    new_box = np.zeros((2*radius+1, 2*radius+1))

    x1 = max(0, boxCenter[1]-radius)
    x2 = min(boxCenter[1]+radius, cols)
    y1 = max(0, boxCenter[0]-radius)
    y2 = min(boxCenter[0]+radius, rows)

    dx = x2 - x1
    dy = y2 - y1
    # add gaussian to mask
    new[y1:y2+1, x1:x2+1] += gauss[-1*(y1==0)*(boxCenter[0]-radius):dy-1*(y1==0)
                                                  *(boxCenter[0]-radius)+(y2!=rows),-1
                                                  *(x1==0)*(boxCenter[1]-radius):dx-1*
```

```
                                                    (x1==0)*(boxCenter[1]-radius)+(x2!=
                                                    cols)]



# Euclidean Mask
centers_y = boxes[:,1]+(boxes[:,3]-boxes[:,1])//2
centers_x = boxes[:,0]+(boxes[:,2]-boxes[:,0])//2

rows,cols = img_gray.shape
density = np.zeros((rows,cols))

for i in range(rows):
    if i % 100==0 and i is not 0:
        print(i)
    for j in range(cols):
        density[i,j] = np.sqrt(np.min((centers_y - i)**2+(centers_x - j)**2 ))
                                                    # Euclidean distance

plt.figure(figsize=(30,20))
plt.imshow(density)
plt.axis('off')
plt.colorbar()
```