Dissertation Type: research

DEPARTMENT OF COMPUTER SCIENCE

Coral Density Analysis using Deep Learning

Ainsley Rutterford

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Engineering in the Faculty of Engineering.

Monday 1st June, 2020

Declaration

This dissertation is submitted to the University of Bristol in accordance with the requirements of the degree of MEng in the Faculty of Engineering. It has not been submitted for any other degree or diploma of any examining body. Except where specifically acknowledged, it is all the work of the Author.

Ainsley Rutterford, Monday 1st June, 2020

Contents

Executive Summary

This project uses deep learning to automate the estimation of the amount of skeletal matter produced annually by corals. In particular, the U-Net architecture is successfully modified to automatically label the density bands present in two dimensional slices of coral. This is the first time that deep learning techniques have been used to process coral density data of any kind. To train the U-Net based network, a dataset containing ∼400 images is curated using slices extracted from four different three dimensional CT scans of coral skeletons. Various data augmentation techniques are used to artificially expand the dataset and improve the performance achieved. In order to effectively measure performance, an accuracy metric based off the average Euclidean distance between predicted and ground truth boundaries is conceived. Various ablation studies and rigorous quantitative and qualitative evaluation are applied to pinpoint performance and limitations. An ablated version of the U-Net architecture was found to perform better in this task and ultimately achieved a cross-validated accuracy of 77.8%.

The automatic labelling of the banding present in three dimensions is also explored. To this end, the U-Net architecture is modified to process three dimensional data, and a custom three dimensional data loader capable of online augmentation is implemented. In order to train this network, a larger dataset of manually labelled adjacent slices is curated, containing over 3,400 images extracted from four coral samples.

The densities of the coral skeletons and the distances between the density bands extracted by the network are automatically estimated using various classical image processing techniques. These estimates are used to automatically calculate the amount of skeletal matter produced annually and the final automated calculations are similar to the manual calculations reported in the literature.

Summary of Achievements

- I spent 25 hours manually labelling coral data.
- I generated a 2D dataset containing ∼400 images and a 3D dataset containing over 3,400 images using various scripts I wrote.
- Having not taken the "Applied Deep Learning" unit, I spent 20 hours researching and learning about the field of deep learning.
- I learned how to use the Keras and TensorFlow libraries in order to implement the networks used.
- I modified an existing 2D U-Net implementation and trained it on the curated dataset.
- I extended the U-Net architecture to process 3D data.
- I wrote a "data generator" from scratch in Python to perform "online" augmentation of 3D data as it is passed into the network since Keras does not have its own implementation.
- I implemented a custom accuracy metric in C and wrapped it in a Python function to achieve a ∼315 times speedup allowing the accuracy to be calculated in minutes rather than hours.
- I trained the network over 100 times whilst performing hyperparameter optimisation, in order to gain a better understanding of the model and to maximise the performance achieved.
- I successfully used ablation studies to find a simplified architecture that performs better than the original U-Net architecture at labelling the boundaries present in two dimensions.
- I cross-validated the results achieved and compared them to results achieved by other architectures.

Supporting Technologies

Various supporting technologies such as software packages, libraries, and computing resources were used throughout the project. These technologies are outlined below.

- Avizo[1]: the Avizo software package is a 3D visualisation program and was used to open and view the 3D CT data interactively. It was vital in the selection of the appropriate slices to label and use to create the dataset.
- GIMP[2]: the GNU Image Manipulation Program. GIMP is a free open-source cross-platform image editor and was used to manually label slices once they were extracted.
- Keras[3]: the Keras library is an open-source deep learning library written in Python. All architectures experimented with throughout the project were implemented using Keras.
- TensorFlow[4]: the Keras library makes use of a TensorFlow backend to allow the code to run on both CPUs and GPUs. The TensorFlow library was also utilised in order to implement the focal loss function [35] experimented with throughout the project.
- OpenCV[5]: the OpenCV library is an open-source computer vision library. Although many Python modules were used throughout the project, the OpenCV module was the main module used to read, write, and manipulate any images.
- BlueCrystal Phase 4[6]: in the early stages of the project, training and testing relied heavily on the GPU nodes of the BlueCrystal Phase 4 supercomputer.
- GW4 Isambard[7]: ultimately, the majority of training and testing took place on the phase 1 GPU nodes of the Isambard supercomputer.

[1] https://tiny.cc/avizo
[2] https://www.gimp.org
[3] https://keras.io
[4] https://tensorflow.org
[5] https://opencv.org
[6] https://www.acrc.bris.ac.uk/acrc/phase4.htm
[7] https://gw4.ac.uk/isambard

Acronyms

ANN	:	Artificial Neural Network
CCA	:	Connected Component Analysis
CNN	:	Convolutional Neural Network
CPU	:	Central Processing Unit
CT	:	Computed Tomography
GAN	:	Generative Adversarial Network
GIMP	:	GNU Image Manipulation Program
GPU	:	Graphics Processing Unit
ReLU	:	Rectified Linear Unit
SGD	:	Stochastic Gradient Descent

Acknowledgements

I am extremely grateful to Dr Tilo Burghardt whose enthusiasm, support, and guidance throughout this project has been invaluable. I will always be thankful for his contributions throughout my entire degree. I would like to extend my sincere thanks to Dr Erica Hendy and Dr Kenneth Johnson for their expertise and enthusiasm for the project. I am also grateful to Leonardo for the time and effort he put into labelling data for this project as well as his consistent support throughout.

Chapter 1

Contextual Background

This chapter outlines and motivates the main objectives of the project. First, the importance of the analysis of coral skeletons is discussed. Next, the motivations for an automated analysis process are outlined and the choice of a deep learning based solution is justified. The challenges involved in completing the project are highlighted, and related works in the analysis of coral skeletons are then briefly discussed. Finally, the main objectives of the project are summarised and listed.

1.1 Introduction

Coral polyps are tiny soft-bodied organisms. Through a process known as calcification, they use calcium and carbonate ions from the surrounding seawater to build themselves hard calcium carbonate skeletons. These polyps reproduce asexually and can form colonies of up to thousands of individual polyps all contributing to the same connected skeleton.

Using X-radiographs in 1972, Knutson et al. were able to confirm the presence of annual density bands in these calcium carbonate skeletons [29]. An annual density band consists of two portions: a high-density portion produced during the late summer and a low density portion formed during periods of seasonally lower water temperatures [19]. These bandings are parallel to the growth surface and orthogonal to the direction of growth; examples are shown in Figure 1.1. With knowledge of a coral sample's date of collection, these annual band pairs can be counted back through time to provide not only an age of the sample, but also a chronology of the coral's growth. More specifically, the annual banding can be used to measure three aspects of coral growth [36]:

1. The linear extension rate (mm y^{-1}): the distance that the coral grows per year.

2. The density (g cm^{-3}): the density of the skeletal matter being produced.

3. The calcification rate (g cm^{-2} y^{-1}): the mass of skeletal matter being produced per year, calculated by multiplying the linear extension rate with the density.

Analysis of these aspects of growth can be used to determine changes in a coral colony's environmental conditions as it was growing. Lough and Cooper highlight one significant potential use of the banding information contained within coral skeletons: "skeletal material contained deep within massive corals can be extracted to allow comparisons of present day growth rates with those pre-dating the industrial revolution and hence examine the consequences of climate and environmental changes on coral reefs" [36].

This project aims to automate the estimation of the linear extension rate which can then be used to estimate the calcification rate. Deep learning will be used to extract the boundaries of each annual density band, and the distance between these boundaries will then be estimated using a method based off of the Euclidean distances between points on the boundaries.

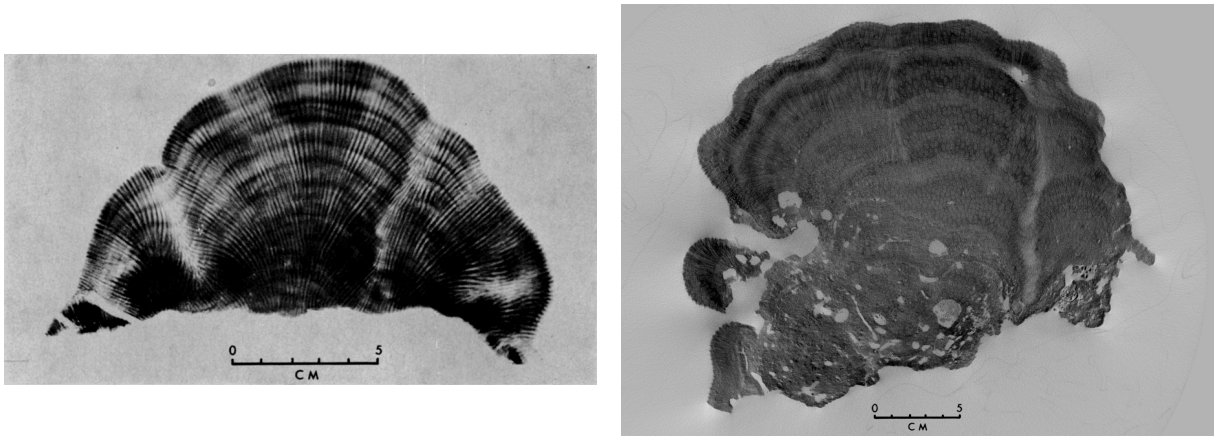


Figure 1.1: Examples of the annual density bands present in the skeletons of a genus of coral called Porites. **(left)** An X-radiograph of a Porites lobata coral sample presented by Buddemeier, Maragos, and Knutson in 1974: "[this is a] positive print of the X-ray negative; dark images correspond to high skeletal density, light images to low density" [8]. **(right)** An image taken from a CT scan of a Porites sample that exists in the dataset used throughout this project. The image channel has been inverted resulting in dark pixels corresponding to high density, and light pixels corresponding to low density.

1.2 Motivations

1.2.1 An Automated Solution

Although many methods for manually estimating the linear extension and calcification rates exist, an automated solution could be beneficial for multiple reasons.

Firstly, opinions on the positions of a particular boundary between high and low density bands can differ from person to person. It is important to note that an idealised annual density cycle is actually in the form of a sinusoidal wave, with the density gradually changing from high to low and back over the course of a year [37, p. 39]. Thus, an exact boundary between a high and low band does not actually exist. However, an effort can be made to consistently choose boundaries at the same point in the sinusoid each year. Naturally, people can disagree with each other when determining the positions of the boundaries; a researcher labelling the same slice for a second time might even disagree with their previous choices of boundary positions. This is not the case, however, for an artificial neural network. Once trained, a neural network is deterministic; given a particular input, it will always produce the same output.

Another benefit of the proposed automated solution is speed. An example of a manual solution is to manually label the density boundaries, use a tool to measure distances between multiple points along these boundaries, and average the distances measured to produce a single average linear extension rate. This process can be time consuming. The proposed solution could potentially estimate the calcification rate of a given slice in a matter of seconds. This would allow researchers to not only analyse the density banding quicker, but also ultimately enable them to process more data overall.

Finally, an automated solution has the potential to be more accurate than a manual solution. For example, an automated solution could measure hundreds of distances between two boundaries which could be used to produce a more accurate estimate than a manual solution in which only tens of distances were averaged.

1.2.2 Deep Learning

Feed-forward artificial neural networks have existed in some form since 1965 [49, 25], but the term "deep learning" was only coined in 2006 [49, 21, 20]. In the years since, the field of deep learning has seen a significant rise in both its popularity and practicality for a multitude of reasons and is now the state-of-the-art in many areas. This section outlines the justifications of the choice of a deep learning based solution. Deep learning is discussed in more technical detail in Section 2.1.

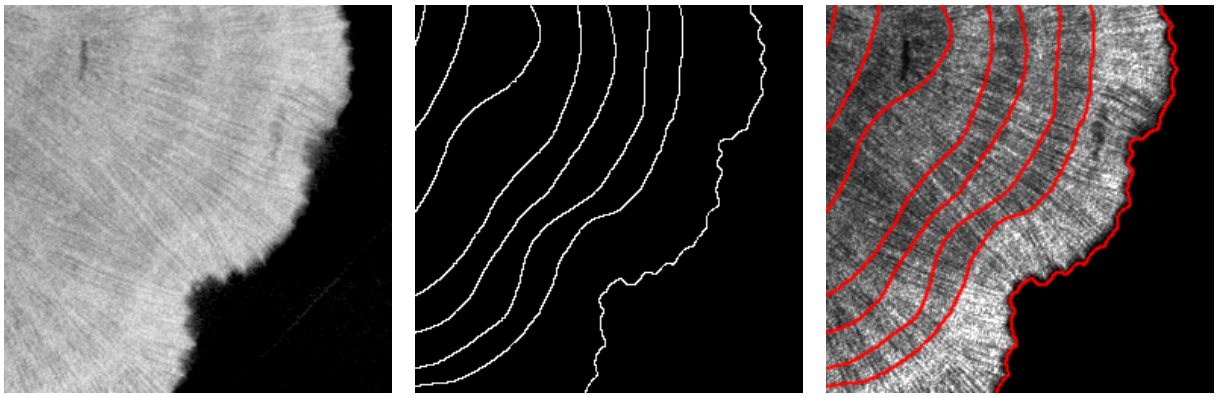


Figure 1.2: An example of a manually labelled 2D slice from the dataset used in this project. This represents the ideal semantic segmentation of a slice. **(left)** An unaltered section of a 2D slice taken from a 3D scan of a coral sample. The slice is a negative; brighter pixels correspond to high density and darker pixels correspond to low density. **(centre)** The manual labelling of the section. The white pixels correspond to the "part of a boundary" class, and the black pixels correspond to the "not part of a boundary" class. **(right)** The manual labelling overlaid with the corresponding section of the slice. The contrast and brightness of the slice has been altered to better show the annual banding, and the colour of the labelling has been changed for illustration purposes.

The Dataset

This project will make use of a dataset of more than 160 three dimensional computed tomography (CT) scans of unique coral skeletons from the Natural History Museum's collection. These three dimensional scans are stored as stacks of two dimensional `.tif` images. With each scan being represented by ∼2000 images, the whole dataset consists of hundreds of thousands of images. The initial dataset provided is unlabelled, so in order to train a supervised machine learning model (such as an artificial neural network) to extract the boundaries, some of the data must be manually labelled.

Although the labelling process is time consuming and will not be possible for the entire dataset, the hope is that researchers at the University of Bristol and the Natural History Museum can continue to label and re-train the proposed system even after this project is over. The abundance of data available makes a deep learning based solution a reasonable choice.

Semantic Segmentation

The task of extracting the boundaries between density bands can be presented as a semantic segmentation task. The volumetric CT data can be broken down into two dimensional images and from there, each pixel will be given a label from one of two classes: part of a boundary or not part of a boundary (see Figure 1.2).

Several forms of deep convolutional neural networks (CNNs) currently achieve the state-of-the-art in semantic segmentation[1] [11, 59]. For this project, pre-existing CNN architectures used for semantic segmentation that perform well on existing datasets are a reasonable starting point. Several CNN architectures will be repurposed and/or modified to process the skeletal density data.

The main architecture experimented with will be the U-Net architecture [45]. U-Net was initially used to perform semantic segmentation on biomedical images such as images of cells on glass that were recorded with differential interference contrast microscopy. The U-Net architecture has not yet been applied to coral density data and experimentation with the U-Net architecture and the dataset used in this project could yield interesting results. To the best of the author's knowledge, none of the architectures experimented with throughout this project have ever been applied to coral skeleton density data of any kind. This project aims to not only determine how well multiple architectures perform in segmenting the skeletal density data but also to determine if CNNs are currently a viable solution to this task.

[1] https://paperswithcode.com/task/semantic-segmentation/latest

1.3 Challenges

There are multiple challenges involved in the implementation of an automated solution. This section outlines and briefly discusses the most significant of these challenges.

1.3.1 Labelling

As previously discussed, the process of manually labelling the boundaries present in the data is time consuming. Labelling enough data for a CNN to perform well would require tens of hours of work. This process takes particularly long as appropriate slices must be extracted from the 3D CT data using a 3D visualisation program such as Avizo[2] before they can be labelled. Not only is this process time consuming, it is also challenging. This is especially the case for a person who has not dealt with coral skeleton density data before. Before samples can be manually labelled, an acceptable standard of labelling must be approved by an expert.

1.3.2 Class imbalance

Another challenge involved with this project is dealing with the inherent class imbalance of the boundary labels. In a typical classification problem, class imbalance occurs when one class contains significantly fewer samples than the other classes. Since semantic segmentation can be seen as per-pixel classification, class imbalance can be an issue in this task as well. Looking at Figure 1.2, it can be seen that the ratio of white to black pixels in the label is significantly low; there is a severe class imbalance between the "part of a boundary" and "not part of a boundary" classes. When a class imbalance exists within training data, supervised learning models will typically over-classify the "majority" groups due to their increased prior probabilities. As a result, the instances belonging to "minority" groups are misclassified more often than those belonging to the majority groups [26].

This class imbalance gives rise to another challenge when assessing the performance of a model in its segmentation of the density data. Since the boundary labels are only one pixel wide, a model that predicts boundary positions that are just one pixel off of the correct positions could potentially achieve a standard per-pixel accuracy of 0%. To solve this problem, a custom accuracy metric must be conceived, which rewards a model even when it predicts a boundary a few pixels away from the manually labelled position.

1.3.3 Loading Three Dimensional Data

When implementing an architecture capable of segmenting data in three dimensions, existing deep learning libraries—such as the Keras[3] library used throughout this project—make the implementation of a 3D architecture relatively straightforward. However, as of the time of writing, a 3D "data loader" capable of loading 3D data from a directory and performing online data augmentation does not yet exist for the Keras library and would have to be designed and implemented.

1.4 Related Work

Although many examples of coral species classification using deep learning exist, no machine learning techniques of any kind have been applied to CT scans of coral skeletons. The implementation of an automated system capable of extracting the density banding or calculating the linear extension and calcification rates has not yet been attempted or at least published.

Steffens et al. [52] make use of the DeeplabV3 [10] CNN to segment coral reef images into different types of substrates. However, the ImageCLEFcoral[4] dataset that they use to test their implementation consists of images taken of the surfaces of coral reefs. Alonso et al. [1] segment coral reef images of a similar kind from the Eilat Fluorescence Corals dataset [3] using the SegNet CNN architecture [2]. Again, this work attempts to segment images taken of the surfaces of coral reefs rather than any kind of data representing the internals of a coral skeleton.

[2] https://tiny.cc/avizo
[3] https://keras.io
[4] https://www.imageclef.org/2019/coral

Three dimensional implementations of the U-Net architecture do exist. For example, Çiçek et al. [12] modify the U-Net architecture to process 3D data by replacing all 2D operations with their 3D counterparts. They assess the performance of their model on a dataset of Xenopus kidney embryos and propose both semi-automated and fully-automated use cases of the model. It is worth noting that the segmentation task that they attempt does not give rise to a class imbalance, and the nature of the segmentation that this project attempts is dissimilar due to the sparsity of the coral density band boundaries.

1.5 Summary of Objectives

In summary, the main objective of this project is to automate the estimation of the calcification rate—the amount of skeletal matter produced annually by corals. This objective is broken down into multiple tasks:

1. Curate a dataset of image-label pairs from the initial unlabelled dataset provided.
2. Train existing two dimensional convolutional neural network architectures to extract the annual density boundaries present in the data.
3. Modify the architectures to process three dimensional data.
4. Conceive and implement an accuracy metric to assess the performance of different networks on the data.
5. Optimise hyperparameters of the chosen network architecture to maximise accuracy.
6. Perform ablation studies to gain a better understanding of the architectures used.
7. Make use of these extracted boundaries to automate the calculation of the annual skeletal matter produced.
8. Package the implementation in a form usable by researchers at the Natural History Museum and the School of Earth Sciences at the University of Bristol.

Chapter 2

Technical Background

As outlined in Chapter 1, the objective of this project is to use deep learning to automate the estimation of the amount of skeletal matter produced annually by corals. This chapter introduces and briefly describes the techniques that will be used in order to achieve this goal.

2.1 Deep Learning

Based loosely on the structure of the brain, artificial neural networks (ANNs) are computational models that have proved useful in a wide range of applications [33, 15, 22]—a notable example being the recent success of convolutional neural networks in the field of computer vision [57, 16]. Deep learning is a form of machine learning that concerns the use of ANNs with many layers—hence the name "deep" learning. The field has seen a significant increase in popularity in recent years since a network named AlexNet famously won the ImageNet Large Scale Visual Recognition Challenge in 2012[1], performing considerably better than the previous state-of-the-art [30].

A typical fully connected ANN is a layered network of "neurons" connected by a series of "weights". A neuron is simply an object that produces a weighted sum of some number of inputs. This weighted sum is often passed through a non-linear "activation" function and the final result is referred to as the "activation" of the neuron. When values are supplied to the input neurons of the network, these values are forward-propagated through the network, activating each layer of neurons which, in turn, activate the next layer. The resulting activations of the neurons in the final layer are the output of the network.

2.1.1 Convolutional Neural Networks

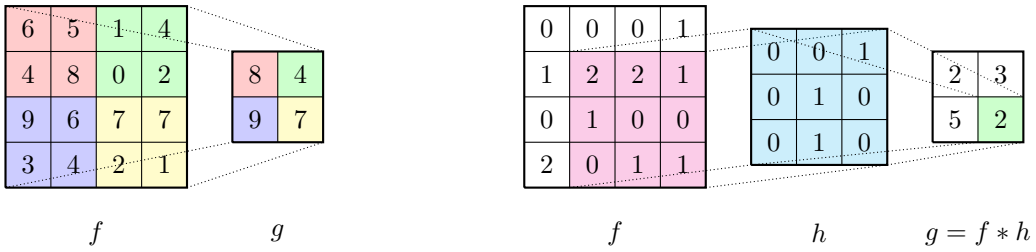
Many of the network architectures experimented with throughout the project will be convolutional neural networks (CNNs). A CNN is a type of neural network named after the discrete convolution operation that sets itself apart from typical fully connected ANNs. The convolution operation makes use of surrounding pixels in order to change the value of a central pixel. The 2D discrete convolution operation is defined as

$$g(x, y) = \sum_{i=1}^{m} \sum_{j=1}^{n} f(x - j, y - k)h(j, k) \tag{2.1}$$

where f is the input image, h is an $m \times n$ kernel, and g is the resulting image.

CNNs often contain multiple convolutional layers in which increasingly complex features of an image are detected by making use of a combination of simpler features detected in previous layers. A convolutional layer convolves the channels of an image with multiple learned kernels. A channel is simply a component of an image—an RGB image, for example, consists of three channels, whilst a greyscale image consists of only one. The output of these operations will be multiple modified images that are referred to as "feature channels" or "feature maps". These feature maps can then be used as the input channels to the next layer, and so on.

[1] http://www.image-net.org/challenges/LSVRC/2012/results.html



(a) 2D Max-pooling

(b) 2D Convolution

Figure 2.1: Illustrations of the 2D max-pooling and convolution operations (own figure). **(a)** The max-pooling operation shown takes an input image f and uses a 2×2 filter and stride of two to produce the output image g. Note that the maximum value in each coloured region of the input is the resulting value of the corresponding regions of the output. **(b)** The convolution operation shown takes an input image f and uses a 3×3 kernel h with a stride of one to produce the output image $g = f * h$, where $*$ denotes the convolution operation. Note that the kernel shown in the convolution operation is "flipped" in both the x and y axes before the element-wise multiplication and summation takes place.

As well as the convolution operation, the max-pooling operation is often used in CNNs and plays an important role in the CNN architectures that will be experimented with throughout this project. The max-pooling operation is used to decrease the dimensionality of the input in an attempt to force the network to "learn" features present in the input. Max-pooling layers are also used by models to achieve some translation invariance over spatial shifts in the input image [2]. The max-pooling and convolution operations are illustrated in Figure 2.1.

Another feature of CNNs that set them apart from their fully connected counterparts is the localised "receptive fields" of the neurons present in convolutional layers. Whilst neurons in a fully connected layer can receive input from every neuron in the previous layer, neurons present in a convolutional layer can only receive input from a small amount of neighbouring neurons in the previous layer. For example, if a 3×3 kernel is used, a neuron's activation can only be influenced by nine neurons in the previous layer. This area of input that can influence a neuron is called its receptive field. This is one of the reasons why during hyperparameter optimisation, kernel sizes from various layers are often tuned, in order to increase or decrease the sizes of neurons' receptive fields. Hyperparameter optimisation is discussed in Section 2.1.9.

2.1.2 Backpropagation

First popularised by Rumelhart et al. in 1986 [47], the backpropagation algorithm is still the main learning mechanism used in neural networks today. Once a loss function is defined to measure the performance of the network, backpropagation can be used to compute the derivative of the loss function with respect to each weight in the network using the chain rule. The derivatives are calculated one layer at a time, iterating backward from the output layer. Since the backpropagation algorithm makes use of the chain rule, the loss function must be differentiable. This must also be the case for the chosen activation functions for each neuron.

An optimisation algorithm can then use these computed derivatives to adjust each weight in order to minimise the chosen loss function. Loss functions used throughout this project are discussed in Section 2.1.4.

2.1.3 Optimisation Algorithms

Once the derivatives—or "gradients"—have been calculated, an optimisation algorithm is used to determine the exact value each weight should be updated to. Throughout the project, various optimisation algorithms are experimented with in order to improve the performance achieved. When training deep neural networks, some variant of the gradient descent algorithm is often used. Gradient descent [9, p. 536] is an iterative algorithm designed to find local minima of some differentiable function—in this case, the loss function. A visualisation of the gradient descent algorithm is shown in Figure 2.2. The gradient descent algorithm evaluates the loss function over the entire dataset before taking a "step" in the direction of the gradient computed. A step consists of updating the value of every weight in order to decrease

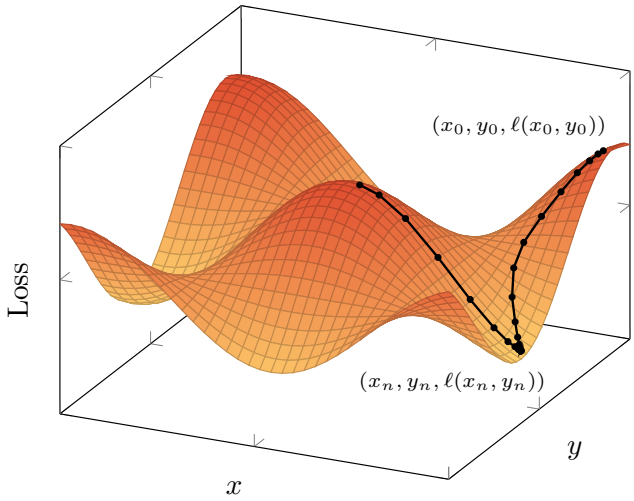


Figure 2.2: A diagram visualising the gradient descent algorithm for a model containing two trainable parameters, x and y (own figure). Given a random starting configuration with parameters x_0 and y_0, the average loss value that would be achieved when processing the entire dataset is given by $\ell(x_0, y_0)$. Say gradient descent takes n steps in the direction of steepest descent at each iteration until the loss value converges to some local minimum, x_n and y_n would be the final parameter values and the final loss would be $\ell(x_n, y_n)$. It is worth noting that even though the two starting configurations shown in the diagram both converge to the same local minimum, this might not always be the case. A starting configuration nearer to one of the other possible local minima would most likely not converge to the example local minimum $\ell(x_n, y_n)$ shown.

the average loss value that would be achieved by the network when processing the dataset. Evaluating the loss function over the entire dataset once is referred to as an "epoch".

The size of the weight update is not only determined by the gradient calculated, but also by a hyperparameter called the "learning rate". For example, gradient descent calculates the update for a single weight w as

$$w := w - \eta \nabla \ell(w) \tag{2.2}$$

where η is the learning rate and $\nabla \ell(w)$ is the derivative of the loss function ℓ with respect to w. Choosing an appropriate learning rate is challenging but is essential to allow an optimisation algorithm to converge to a local minimum. A learning rate that is too large can cause the loss function to fluctuate around the minimum, impeding convergence or even causing divergence [7]. The learning rate hyperparameter has a significant effect on training and is one of the most important parameters tuned in the hyperparameter optimisation process [4] (see Section 2.1.9).

Stochastic Gradient Descent

Issues arise when using gradient descent with larger datasets, as evaluating the loss function over the entire dataset before a step can be taken becomes increasingly computationally costly [46]. These issues have given rise to the popularity of the stochastic gradient descent (SGD) and the mini-batch gradient descent optimisation algorithms. SGD is a variant of the gradient descent algorithm that takes a step each time a sample is processed, rather than only once the entire dataset is processed. Mini-batch gradient descent is yet another variant of gradient descent in which weights are updated after evaluating the loss function over a subset—or "batch"—of the training samples.

Adam

First introduced by Kingma and Ba in 2014, Adam [28] is an optimisation algorithm that is also based off of SGD. However, rather than using the same learning rate across all parameters, Adam computes individual adaptive learning rates for each parameter. These individual learning rates are based off of estimates of the first moment (the mean) and second moment (the uncentered variance) of the gradients [46]. Kingma and Ba showed empirically that Adam works well in practice and compares favourably to other optimisation algorithms when optimising both fully connected ANNs and deep CNNs. Due to

its ability to train deep CNNs effectively, the Adam optimisation algorithm will be the main algorithm utilised throughout the project.

2.1.4 Loss functions

A loss function takes the ground truth label and the predicted label of a training sample, and outputs some measure of how well a model performed by producing that prediction. As mentioned in Section 2.1.2, backpropagation makes use of the derivative of the loss function with respect to each parameter in order to minimise the loss achieved. It is for this reason that loss functions must be differentiable. If the appropriate loss function is chosen, minimising the loss should improve the performance achieved by the network. The loss functions that will be experimented with throughout this project are outlined below.

Binary Cross-Entropy Loss

The binary cross-entropy loss function is used when classifying samples that can belong to two classes. Since the boundary extraction that this project attempts can be presented as a binary semantic segmentation task, the binary cross-entropy loss is a reasonable first choice of loss function. When performing binary image classification, a model will produce two probabilities—one for each class. Whereas, when performing binary semantic segmentation, a model effectively classifies each pixel; thus, two probabilities will be produced for each pixel. These are the model's predicted probabilities that a given image (or pixel) belongs to each of the two classes. Since these two classes are the only possible classes that the model can predict, the probabilities should sum to one. The binary cross-entropy loss is defined as:

$$CE(p, y) = \begin{cases} -\log(p) & \text{if } y = 1 \\ -\log(1-p) & \text{otherwise} \end{cases} \tag{2.3}$$

where $y \in \{+1, -1\}$ is the actual value (the ground-truth), and $p \in [0, 1]$ is the model's estimated probability for the class with label $y = 1$. This is the loss value achieved when classifying a single training sample (or a single pixel in the case of semantic segmentation).

Binary Focal Loss

Proposed by Lin et al. in 2018, the Focal Loss [35] is designed to address the "class imbalance" problem. In a typical classification problem, class imbalance occurs when one class contains significantly fewer samples than the other classes. Lin et al. highlight the fact that when using the cross-entropy loss function, even samples that are well-classified incur a loss with "non-trivial" magnitude. When summed over large numbers of samples from the easily classified majority class, these small loss values can overwhelm the loss resulting from the minority class samples [35]. As highlighted in Section 1.3, the boundary extraction task inherently contains a severe class imbalance; the focal loss function could be used to reduce the negative effects of this class imbalance on the performance of the networks used throughout the project.

In an attempt to address the class imbalance problem, the focal loss introduces two new parameters: a weighting factor $\alpha \in [0, 1]$ and a "focusing" parameter $\gamma \geq 0$. The binary focal loss is then defined as:

$$\mathrm{FL}(p, y) = \begin{cases} -\alpha(1-p)^{\gamma}\log(p) & \text{if } y = 1 \\ -\alpha(p^{\gamma})\log(1-p) & \text{otherwise} \end{cases} \tag{2.4}$$

where $y \in \{+1, -1\}$ is the actual value (the ground-truth), and $p \in [0, 1]$ is the model's estimated probability for the class with label $y = 1$. The weighting factor increases the loss produced by the misclassification of minority class samples whilst the focusing factor "reduces the contribution from easy examples and extends the range in which an example receives low loss" [35]. When $\gamma = 0$ and $\alpha = 1$, the focal loss is equivalent to the cross-entropy loss. The contribution of the γ value to the loss produced can be seen in Figure 2.3.

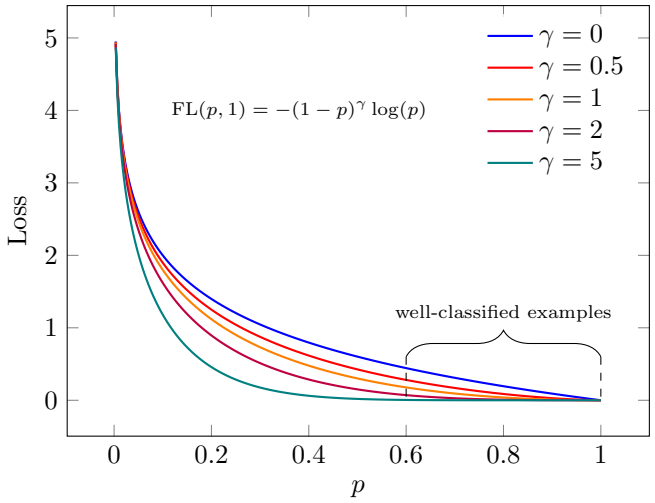


Figure 2.3: An illustration of the focal loss with varying values of γ. Recreated from [35, Fig. 1]: "setting $\gamma > 0$ reduces the relative loss for well-classified examples ($p > 0.5$), putting more focus on hard, misclassified examples". It can be seen that when $\gamma = 5$, the loss achieved by well-classified examples is far lower than when $\gamma = 0$.

2.1.5 Accuracy Metrics

Although loss functions can be used to measure performance, their main purpose is to be used to train the network. To quantify the performance achieved by a network, an accuracy metric should instead be used. Accuracy metrics do not play a direct role in training networks, though they can be used indirectly—for example, to determine whether or not to continue training. Choosing an appropriate accuracy metric for a given task is essential and is especially challenging for this project. As discussed in Chapter 1, due to the severe class imbalance present in the ground truth labels used in this project, a standard per-pixel accuracy would not be appropriate for this task, and a custom accuracy metric will have to be conceived.

2.1.6 Overfitting and Regularization

The term "overfitting" refers to the phenomenon when a model performs well on the data on which it is trained, yet poorly on data which it has not yet been exposed to. Overfitting is still a serious problem faced when training deep networks due to the large amount of parameters that must be learned [51, 13, 48]. Regularization is any technique that is used to reduce overfitting and allow a model to generalise better [31]. Due to the difficulties involved in manually labelling the coral data used in this project, obtaining enough labelled data proved challenging. With smaller amounts of labelled data to train on, overfitting becomes an even more significant problem. This section introduces some of the regularization techniques that will be applied throughout the project to combat this overfitting.

Data augmentation

A very common technique used to reduce overfitting is data augmentation. Data augmentation is the process of augmenting the labelled training data in some way in order to increase the amount of training data available. This augmentation can be performed "online" with each training sample being randomly augmented during the training process, or it can be performed "offline" with the augmentation taking place before training. The training data can be augmented in many ways. Common examples for 2D images include: random rotations within some predefined range, random changes to brightness levels, and random horizontal and vertical flips. Effective data augmentation allows for a dataset to be artificially expanded, enabling it to represent a more comprehensive set of possible samples. Due to the challenges involved in labelling the data used in this project, data augmentation is an important regularization technique and may significantly improve the performance achieved by both the 2D and 3D models experimented with.

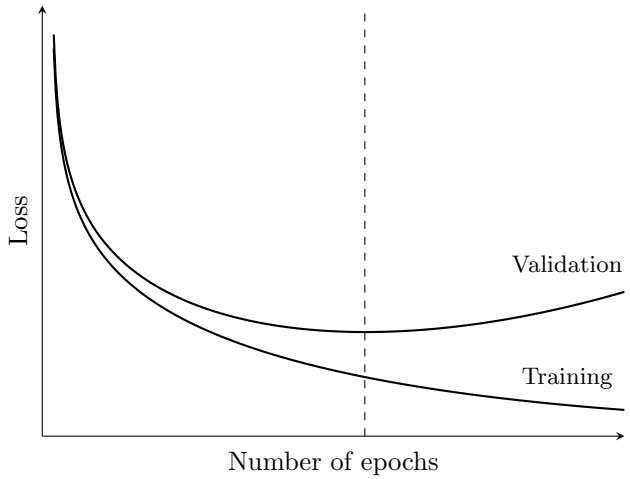


Figure 2.4: An illustration of idealised training and validation loss curves (own figure). Both the training and validation loss decrease until a point. However, from this point onward, although the training loss continues to decrease, the validation loss begins to increase and the model begins to overfit. This point at which the validation is lowest is the ideal time to stop the training process.

Dropout

Another technique often used to reduce overfitting is dropout. The use of dropout as a regularization technique when training neural networks was first popularised by Srivastava et al. [51] in 2014. Throughout the training process, subsets of randomly selected neurons and their connections are "dropped". Only the resulting reduced network is then trained on a particular training sample. A subset of dropped neurons will only remain dropped whilst processing a single training sample; the dropped neurons will then be added back to the network and a different subset of neurons will be dropped when processing the next training sample. The choice of which subset of neurons to drop for each training sample is random.

Dropout reduces overfitting by preventing neurons from "co-adapting" too much [51]. Co-adaptation between neurons refers to the phenomenon when one neuron relies too heavily on the input from another neuron. If one neuron relies too heavily on the input of another neuron, the receiving neuron's performance in producing an activation that would be beneficial to the performance of the network may be hindered by the one "bad" input—even if the inputs from all other incoming neurons were "good". In this case, the terms good and bad are used to describe whether or not an input contributes to the the optimal activation value calculated by backpropagation.

Early Stopping

Referred to as a "beautiful free lunch" by Geoffrey Hinton [17, p. 141], early stopping is a regularization technique in which the training process is stopped given some "stopping criteria" rather than after a set number of epochs. The simplest stopping criteria is to stop training as soon as the loss achieved on the validation set increases after an epoch, rather than decreases. However, when training complex models, the "loss curve" achieved on the validation set may contain many local minima [41]. A more common method is to stop training once the loss achieved on the validation set fails to decrease after a certain number of epochs. This method helps to prevent noise in the validation loss curve from stopping the training process prematurely. A simple example of early stopping for an ideal loss curve is shown in Figure 2.4.

2.1.7 Cross-Validation

When training and evaluating a model, the dataset is often split into three subsets: the training set, the validation set, and the test set. The training set is often the biggest subset and is used to fit the model. This is the set that the model "learns" from. The validation set is used to provide an unbiased evaluation of the model's performance on the dataset which can then be used to tune hyperparameters (discussed in Section 2.1.9). Note that the evaluation of the model's performance provided by the validation set becomes more biased each time the validation performance is used to tune hyperparameters. It is important to

keep the training and validation sets separate; the model must not be directly trained on any validation data. The test set is used to provide an unbiased evaluation of the final model. The test set is only used once the final model hyperparameters are chosen and the model has been trained. Evaluating the model's performance on a single test set may not be robust to "selection biases"—biases that arise from selecting a particular test set. Although curating a test set that represents a wide range of possible data samples can help reduce selection bias, some amount of bias is still inevitable.

In order to reduce the effects of selection bias, a cross-validation technique can be used. Cross-validation is designed to give a comprehensive measure of a model's performance throughout the entire dataset, not just a particular subset. Once the model's hyperparameters and training configuration have been finalised, multiple train/test splits can be used. For each new train/test split, the model can be re-trained and its performance will be evaluated on the new test set. This process can be repeated with multiple random train/test splits and the final reported performance will be the average of the performances achieved on each test set. In order to ensure that the final performance achieved is not biased, it is important to ensure that the model and its hyperparameters are not modified during the cross-validation process.

The nature of the density bands in the coral skeleton data used in this project varies significantly from skeleton to skeleton. The performance achieved on one skeleton does not necessarily reflect the performance achieved on others. Cross-validation will thus be necessary to ensure that the final performance reported is in fact representative of the network's performance on the entire dataset.

2.1.8 Ablation Studies

In the context of deep neural networks, the term "ablation study" is used to describe a procedure in which certain parts of the network are removed in order to gain a better understanding of the network's behaviour. Ablation studies can also be used as a powerful optimisation tool. For example, if a certain layer or group of layers is removed and the performance of the network remains relatively unchanged, the ablated network would contain fewer parameters resulting in shorter training and inference times.

2.1.9 Hyperparameter optimisation

A hyperparameter is a type of parameter which is explicitly set by hand as opposed to being learned via the training process. The learning rate and batch sizes are typical examples of hyperparameters, whereas neuron weights and the kernels used in convolutional layers are examples of learned parameters. The performance achieved by deep neural networks is known to depend critically on the identification of a good set of hyperparameters [34, 6].

The grid search technique has traditionally been used to find optimal hyperparameter configurations. A grid search is simply an exhaustive search through all of the possible combinations of a set of accepted hyperparameter values. In recent years, however, the grid search technique has often been replaced with a random search. Rather than exhaustively iterating over all combinations, random search iterates over random selections of combinations. Bergstra and Bengio [5] were able to show empirically that even over the same domain, random search is able to find hyperparameter configurations that are as good as or better than configurations found via grid search. Bergstra and Bengio also demonstrated that random search can find these optimal configurations within a small fraction of the computation time taken by grid search.

2.2 Network Architectures

This section introduces and details the various CNN architectures that will be experimented with throughout this project.

2.2.1 SegNet

In 2015, Badrinarayanan, Kendall, and Cipolla [2] introduced SegNet: a fully convolutional neural network architecture used for semantic pixel-wise segmentation that was primarily motivated by road scene understanding applications. An architecture is "fully convolutional" when it contains no fully connected layers. The architecture consists of an "encoder" and a "decoder" with some extra pass through of information from layers early in the encoder to later layers in the decoder.

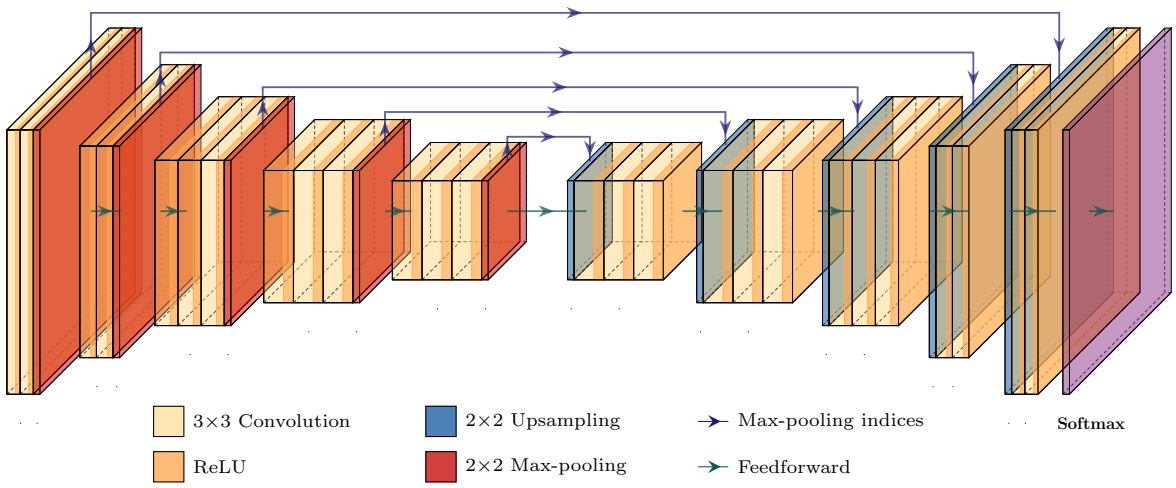


Figure 2.5: A diagram of the SegNet architecture (created using an open-source CNN architecture visualisation tool[2]). Each cube represents a layer. The green arrows represent the feedforward of information from one layer to the next, whereas the blue arrows represent the passing forward of the max-pooling indices for the upsampling layers to use.

An encoder is a model—or part of a model—that takes some input and "encodes" it into some lower dimensional representation. For example, a typical CNN architecture used for image classification is an example of an encoder, since the inputs are often high resolution two dimensional images, and the outputs are a small number of probabilities—one for each possible class that the image could be classified as. In fact, the encoder architecture used in SegNet is topologically identical to the convolutional layers used in the VGG16 architecture used for image classification [2, 50]. A decoder can be thought of as the opposite of an encoder; given some input, a decoder will "decode" the input back into some higher dimensional representation.

The information passed through from the encoder to the decoder is in the form of max-pooling "indices". The decoder uses pooling indices computed in the max-pooling step of corresponding encoder layers to perform non-linear upsampling. These indices are simply the locations of the maximum value in each pooling window. This approach eliminates the need for the architecture to "learn" the indices to use to upsample with, since the indices that it will use are already learned in the max-pooling layers of the encoder. A diagram of the SegNet architecture is shown in Figure 2.5.

2.2.2 U-Net

Presented by Ronneberger et al. in 2015 [45], U-Net is a CNN architecture that was initially used to perform image segmentation on biomedical images, but has since been used on a wider range of visual data. Due to its success in semantic segmentation tasks, it will be the main architecture experimented with throughout the project.

At a high level, the U-Net architecture consists of three sections: the contracting path, the bottleneck, and the expanding path. The contracting path follows the typical architecture of a convolutional network, in that the image is gradually downsampled via convolutional and max-pooling layers, and the number of feature channels is increased. The expansion path then gradually reconstructs the image via "up-convolutions". Throughout the expanding path, the feature channels from the corresponding contracting layers are appended to the feature channels in the expanding layers. This allows the features that are learned whilst contracting the image to also be used to reconstruct it. A diagram of the U-Net architecture is shown later in Chapter 3 when its internals are discussed in more detail.

The U-Net and SegNet architectures share multiple similarities. They are both fully convolutional architectures designed to perform semantic pixel-wise segmentation. Both architectures also consist of an encoder, a bottleneck, and a decoder, with the encoder following a typical CNN architecture. Note that when describing the U-Net architecture, Ronneberger et al. refer to the encoder and decoder as

[2] https://github.com/HarisIqbal88/PlotNeuralNet

the contracting and expanding paths respectively. Another similarity between the architectures is their passing of information from layers in the encoder to corresponding layers in the decoder. However, the passing of information is implemented differently in each architecture. Whilst U-Net passes information by concatenating feature maps from layers in the encoder to the feature maps of layers in the decoder, SegNet passes information in the form of pooling indices. These different methods of passing forward information can significantly affect the performance achieved in various semantic segmentation tasks, and will be experimented with later in the project.

2.2.3 Generative Adversarial Networks

A generative adversarial network (GAN) [42] is a framework in which two models are pitted against each other: a generative model G that captures the data distribution and a discriminative model D that estimates the probability that a sample came from the training data rather than G. Both models are trained simultaneously and in competition with each other, in that G is trained to maximise the probability of D making a mistake.

Many problems in image processing and computer vision involve "image-to-image translation": the translation of an input image into some corresponding output image. In recent years, adversarial networks have proved to be an effective general-purpose solution to image-to-image translation [14]. For example, the pix2pix model [24] demonstrates effective results in different computer vision tasks that had previously required special purpose models, including semantic segmentation and colourisation of black and white images. Since the boundary extraction task attempted in this project can also be thought of as an image-to-image translation task, the pix2pix model will briefly be experimented with later in the project.

2.3 Supporting Technologies

2.3.1 Keras

The Keras[3] library will be used to implement all architectures experimented with throughout this project. Keras is an open-source library written in Python that can be used to build and train deep learning models. Keras is capable of running on top of the TensorFlow[4] library allowing code to be run on both CPUs and GPUs.

Keras offers two APIs that can be used to define a model architecture: the sequential API and the functional API. The sequential API is easier to use and allows users to define a model architecture by simply passing a list of layer instances to the sequential constructor. However, the sequential API is limited in that it does not allow users to define architectures that share layers or have multiple inputs or outputs. Since the architectures implemented throughout this project contain layers that take inputs from more than one previous layer, the functional API must be used.

A simple example of how to define and train a fully connected ANN using the Keras library is shown in Listing 2.1.

[3] https://keras.io
[4] https://tensorflow.org

```
from keras.layers import Input, Dense
from keras.models import Model
from keras.optimizers import Adam

# Define the input layer
inputs = Input(shape=(784,))

# Define the hidden and output layers
linear1 = Dense(64, activation="relu")(input)
linear2 = Dense(64, activation="relu")(linear1)
outputs = Dense(10, activation="softmax")(linear2)

# Create the model
model = Model(inputs=inputs, outputs=outputs)

# Compile the model
model.compile(optimizer=Adam(lr=1e-4), loss="binary_crossentropy")

# Train the model on labelled data
model.fit(data, labels)
```

Listing 2.1: A small code snippet defining a simple fully connected ANN with an input of a 1D tensor of length 784, two hidden layers both containing 64 units with ReLU activation functions, and an output layer containing 10 units using the softmax activation function. The network is then trained using the Keras implementation of the Adam optimiser to minimise the binary cross-entropy loss.

Chapter 3

Implementation

This chapter details the implementation of the automated solution proposed in Chapter 1. First, an overview of the various components implemented throughout the project is given. The implementation of a deep network capable of automatically extracting annual density bands present in two dimensional data is then discussed. The attempts at implementing a network capable of automatically extracting the annual density bands present in three dimensions are also detailed. Then, the implementation of and reasoning behind the custom accuracy metric are discussed. Finally, the techniques used to automatically estimate the calcification rate from the extracted boundaries are described.

3.1 Overview

A system capable of calculating the calcification rate given only a small section of density data requires the implementation of many separate sub-components. Figure 3.2 provides an overview of these components and outlines the order in which they were implemented and will be discussed. Ultimately, the training of a 2D architecture to predict the positions of the boundaries between annual density bands was the main focus of the project. The steps taken to implement this component compose the majority of the content in this Chapter.

All aspects of the project relied heavily on the provided dataset. The processing and labelling of the data was a significant part of the implementation and will be discussed in detail. An example of a 3D scan present in the dataset is shown in Figure 3.1.

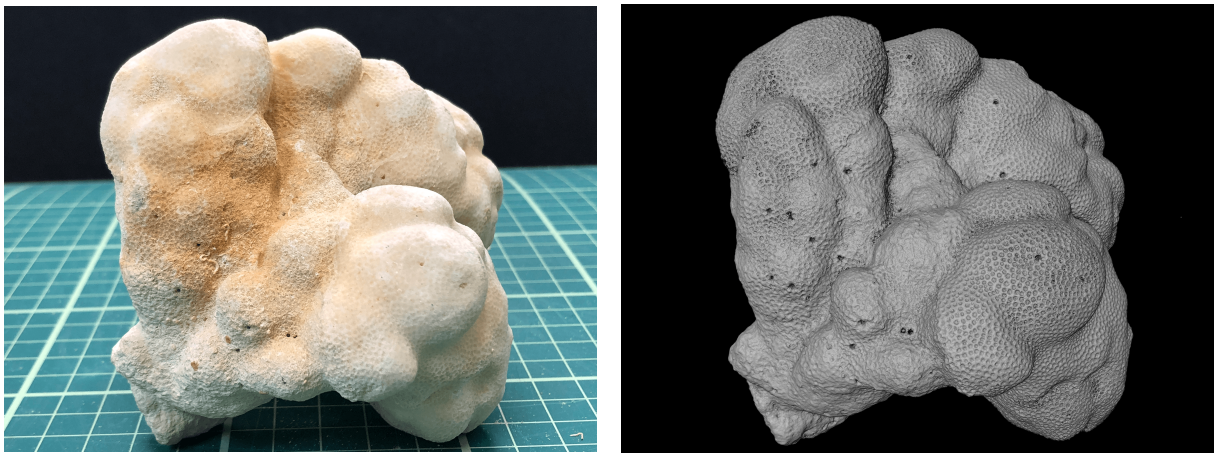


Figure 3.1: **(left)** An example of a Porites coral skeleton from the Natural History Museum's collection. This particular sample was collected from the Solomon Islands in 1974. In the initial dataset, the scan of this sample is represented by 1945×1508×1208 voxels. **(right)** A 3D volume rendering of the CT scan of the sample shown on the left. Created using the Avizo software package.

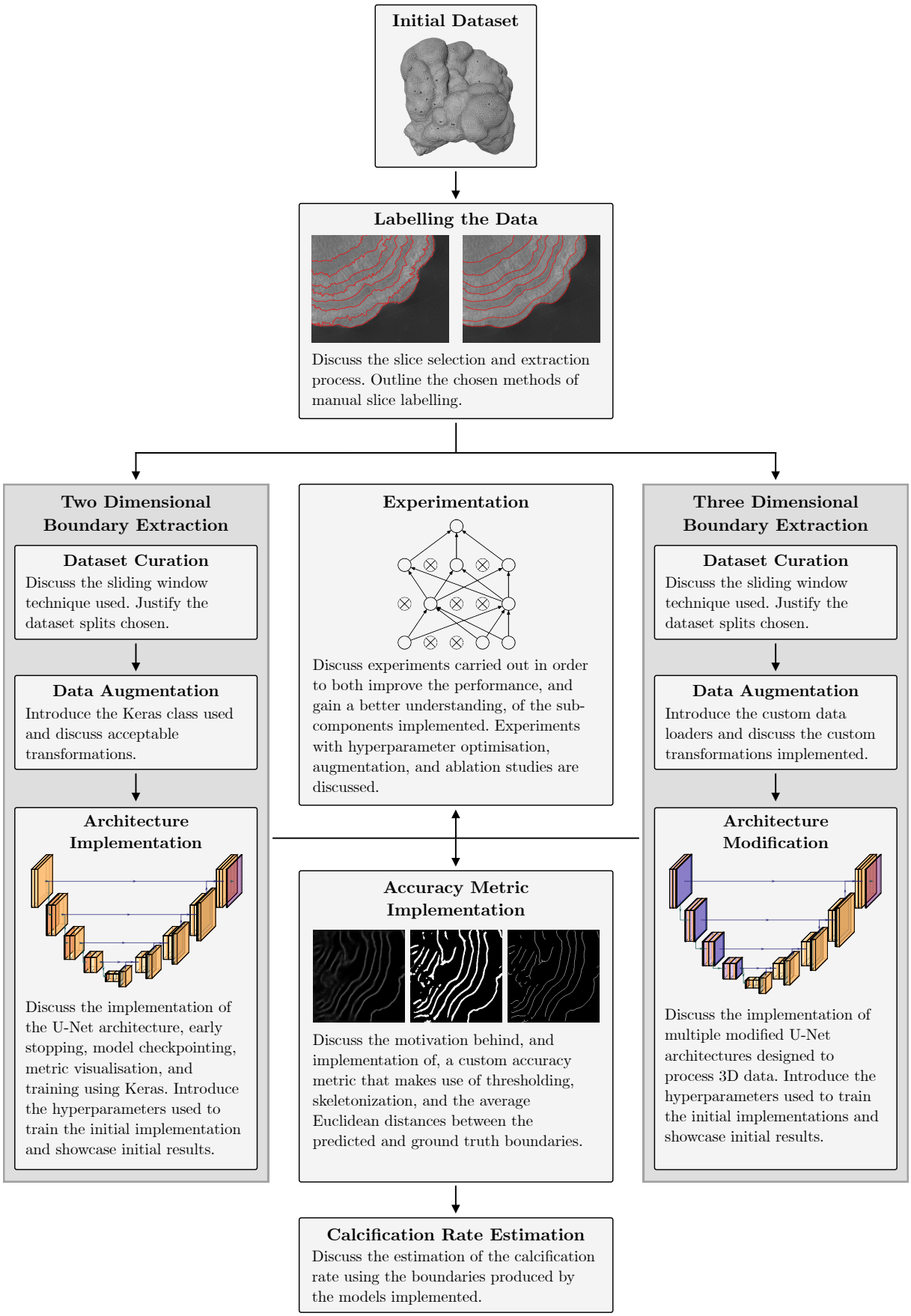


Figure 3.2: A diagram providing an overview of the components implemented and discussed throughout the project. The experimentation is discussed in Chapter 4. The dropout illustration in the experimentation section is taken from [51].

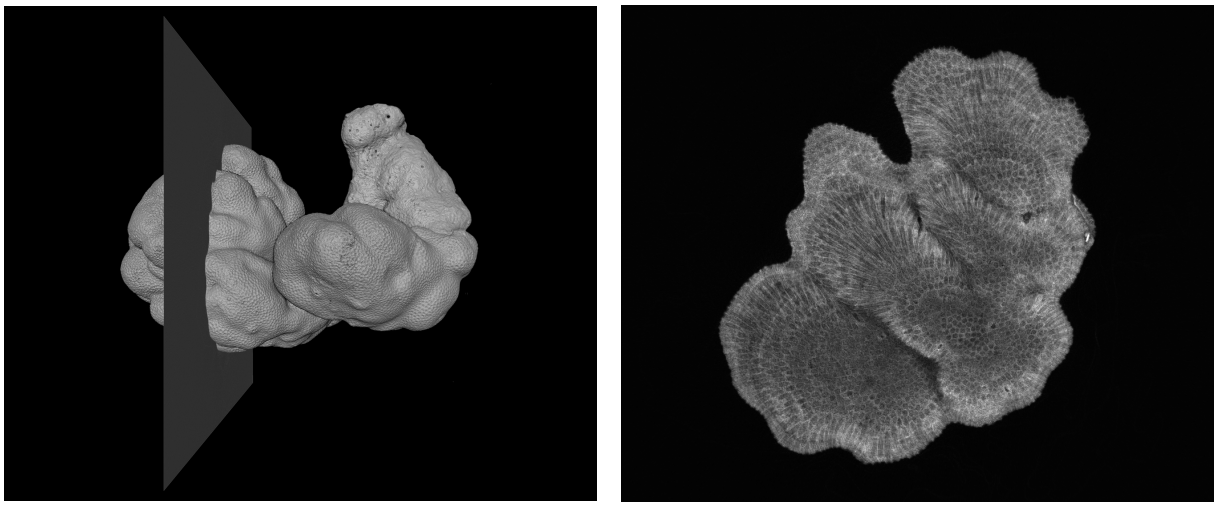


Figure 3.3: An illustration of the how the 3D CT data is composed of many 2D slices. **(left)** A 3D volume rendering of one of the CT scans present in the dataset. The grey rectangle "cuts" the scan and produces the slice shown on the right. It can also be seen that the scan on the left is represented by many slices similar to the one shown on the right. **(right)** A 2D slice extracted from the CT scan shown on the left. The slice is a negative; brighter pixels correspond to high density and darker pixels correspond to low density.

3.2 Two Dimensional Boundary Extraction

This section outlines the steps taken to implement and train a CNN capable of extracting the annual density banding present in two dimensional data.

3.2.1 Labelling the Data

In order for a supervised CNN to perform well, large amounts of labelled data must be available to train on. Since the dataset provided was initially unlabelled, a manual labelling process was devised and is described in this section.

The Initial Dataset

The dataset provided contains more than 160 three dimensional computed tomography (CT) scans of unique coral skeletons from the Natural History Museum's collection. Each individual 3D scan consists of stacks of thousands of 2D `.tif` images which will be referred to as "slices" (see Figure 3.3). This unlabelled dataset will be referred to as the "initial dataset". Although the initial dataset contains scans of ten different genera of coral, only scans of the Porites genus were considered for labelling and training the network with. There are 58 scans of Porites skeletons in total. An example of a Porites skeleton that is part of the dataset was shown previously in Figure 3.1. The Porites scans were chosen as they contain annual banding that can be more easily identified and labelled when compared to other coral genera.

A typical scan consists of ∼2000 slices that each have the same resolution of ∼2000×2000 pixels resulting in an overall 3D resolution of 2000×2000×2000 voxels. However, note that the 3D resolution, scale (e.g., the number of voxels used to represent a centimetre cubed), and density calibration (e.g., the voxel value used to represent a given density) of each scan varies.

Slice Selection and Extraction

Not all slices that compose a scan contain annual density banding that can confidently be identified. In order to find appropriate slices, each of the 58 Porites scans were opened and inspected manually using a 3D visualisation program called Avizo[1]. Avizo allows users to view slices orthogonal to the x, y, or z axis of a scan. A selection of six slices that could confidently be labelled were chosen. Since the density banding boundaries are orthogonal to the growth direction (or "concordant" with the growth surface [29]), the slices that present the most obvious density banding are normally parallel to the direction of growth.

[1] https://tiny.cc/avizo

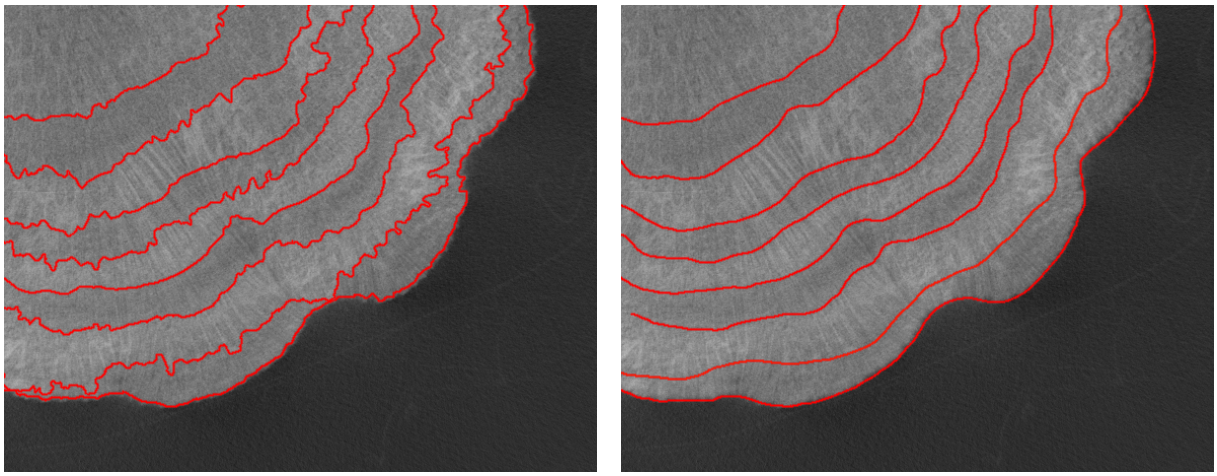


Figure 3.4: A comparison of the two methods of labelling that were initially considered. The boundary labels are coloured red and placed over the same original negative 2D slice. **(left)** A method of labelling in which the boundaries were placed at the sharpest change in density, resulting in complex boundaries whose positions are highly sensitive to noise in the image. **(right)** A method in which the chosen boundaries are "smooth" whilst also being placed as close as possible to the sharpest changes in density.

Once these slices were identified, a Python script was used to extract each slice and export them as greyscale `.png` images. The slices can be represented as greyscale images since they only require one channel to represent depth. Some of the surrounding slices were also extracted in order to curate the 3D dataset used later in the project (discussed in Section 3.3).

Slice Labelling

Once the slices were selected, a labelling process was established. Initially, two methods of labelling shown in Figure 3.4 were considered and shown to Dr Erica Hendy, a senior lecturer in biogeochemical cycles. Of the two methods, the "smooth" method was deemed as the more realistic choice. An idealised annual density cycle is actually in the form of a sinusoidal wave, with the density gradually changing from high to low and back over the course of a year [37, p. 39]. Thus, an exact boundary between a high and low band does not actually exist, making the "complex" method unrealistic both in terms of reproducibility, and in terms of biological accuracy.

Each selected slice was then manually labelled using the GNU Image Manipulation Program (GIMP)[2]. A one pixel wide white line is drawn at the beginning and end of each annual high density band and the rest of the image is black—no values other than black or white are present in the labels. In order to ensure that the decisions made regarding the boundaries of the annual bands were as consistent as possible, each slice was manually labelled three or more times, and the most common boundary positions were chosen. It is important for the labelling method to be consistent and reproducible as this enables the dataset to be more easily expanded in the future.

3.2.2 Dataset Curation

With only a small selection of slices successfully extracted and labelled, a larger dataset of image-label pairs was required for the model to train on.

Sliding Window

In order to expand the dataset, a sliding window technique was used. A script was written in Python that takes a slice, two 2D coordinates, a window size, and a "stride" as arguments. The pair of coordinates correspond to the top left corner and bottom right corner of the confidently labelled area of the slice. This is the area that the window can slide over in order to create the resulting "patches". The stride argument dictates how far the window should slide before each patch is produced, and the window size argument defines how large the resulting patches should be. This sliding window technique is visualised

[2] https://www.gimp.org

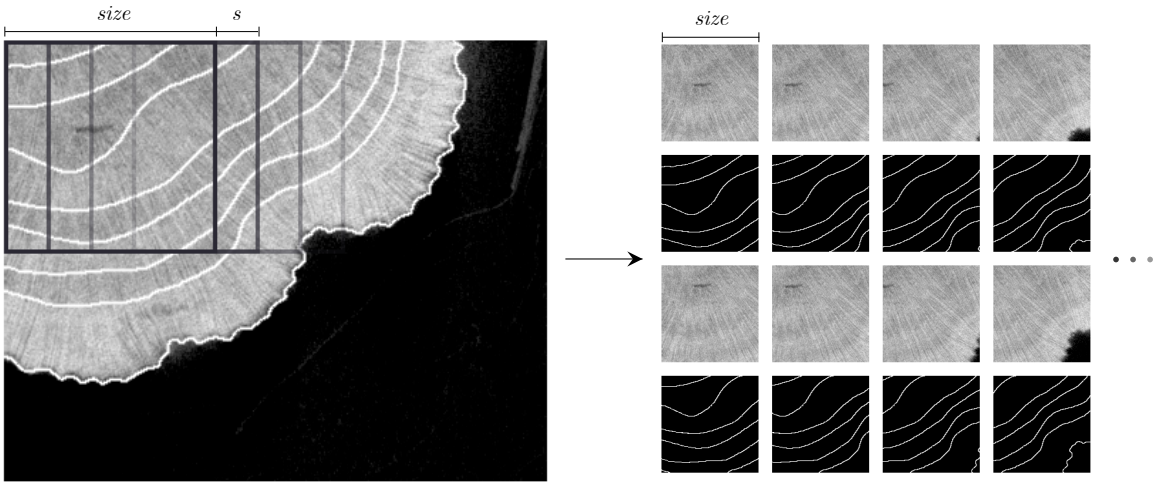


Figure 3.5: A visualisation of the sliding window technique used to produce multiple "patches" from a single slice. A portion of an original negative slice is shown with the manual labelling overlaid in white. The windows on the left show where the patches on the right would be produced from. *size* is the window size, and s is the stride value. The windows have a width and height of *size* and will be moved s pixels to the right before another patch is produced. Once the window reaches the right hand side of the image, the window will be placed back at the start and will be shifted downwards by s pixels before the process starts again.

in Figure 3.5. The patches produced in this project have a resolution of 256×256 pixels. The arguments used in order to produce the curated dataset used in this project are shown in Table 3.1. A total of 388 patches were produced from six slices taken from four different coral skeleton scans. This dataset of 388 256×256 patches and their corresponding labels were stored as greyscale `.png` images.

In order to prevent the dataset from representing certain coral samples better than others, an effort was made to equalise the number of patches produced per scan. Due to different slices containing varying sizes of confidently labelled areas, the stride argument was used to increase or decrease the number of patches produced. However, it was also important to produce as much data as possible to enable the network to perform well. Ultimately, some slices produced almost five times as many slices as others, resulting in some imbalance in the types of coral scans represented by the dataset. Although this could negatively affect the generalisability of a network trained on this dataset, cross-validation with carefully selected splits to represent each coral scan was ultimately used and is discussed in Chapter 4.

Splitting the Dataset

For the initial training and testing of the network, the curated dataset was split into a training set, a validation set, and a test set. Of the 388 patches, 322 were used for training, 56 for testing, and 10 for validation. The patches composing the test and validation sets were all produced from slices of a coral skeleton that was not part of the training set. This ensured that the network could not have been overfitting to the nature of the annual banding present in the skeleton used for testing. If this had been the case, the performance on the test data could have been positively skewed.

3.2.3 Data Augmentation

As discussed in Chapter 2, data augmentation is an important regularization technique. Since only 388 patches had been produced, extensive augmentation was used to artificially expand the dataset. This data augmentation later proved effective in improving the performance achieved by the network and is discussed further in Chapter 4.

The Keras `ImageDataGenerator` class

The Keras library allows users to easily implement augmentation using the `ImageDataGenerator` class[3]. In this case, the `flow_from_directory` method was used to perform online augmentation. This method first loads images from a directory into batches. A series of random transformations are then applied

[3] https://keras.io/preprocessing/image

Table 3.1: The arguments used with the Python script for each labelled slice. For each slice, a size argument of 256 was also specified. Note that some slices are represented by two rows as these slices contained two separate areas that could be confidently labelled. It was not possible to use one area that contains the two areas as this would result in multiple patches with no labelling being produced.

Slice	x_{top}	y_{top}	x_{bottom}	y_{bottom}	Stride (px)	Patches produced
RS0030_yz_0625.tif	1028	153	1350	530	20	36
RS0030_yz_0642.tif	1070	143	1390	463	20	18
RS0030_yz_0642.tif	895	400	1275	685	20	12
RS0116_0414.tif	666	1258	1560	1750	40	150
RS0116_0500.tif	970	1320	1320	1854	30	54
RS0116_0500.tif	1130	1250	1525	1730	30	56
RS0128_yz_0451.tif	490	258	790	690	20	32
RS0130_xz_0820.tif	513	1190	828	1485	10	30
Total						388

to each batch. The original batches are then replaced with these new randomly transformed versions. The `flow_from_directory` method then returns a Python generator that will yield randomly augmented batches of the data indefinitely. A network trained on the generator produced by this method will therefore be trained on randomly transformed samples as opposed to the original samples themselves. The batches can then be input to the model in the form of $(batch \times y \times x \times 1)$ tensors where $batch$ is the batch size, y and x are the heights and widths of the patches respectively, and 1 is the number of channels (since the patches are stored as greyscale images).

The `flow_from_directory` method only loads one stream of images from a directory. In order to load the corresponding labels from another directory, a `flow_from_directory` method from another instance of the `ImageDataGenerator` class must be used. The two generators produced can then be "zipped" into a single generator that yields image-label pairs.

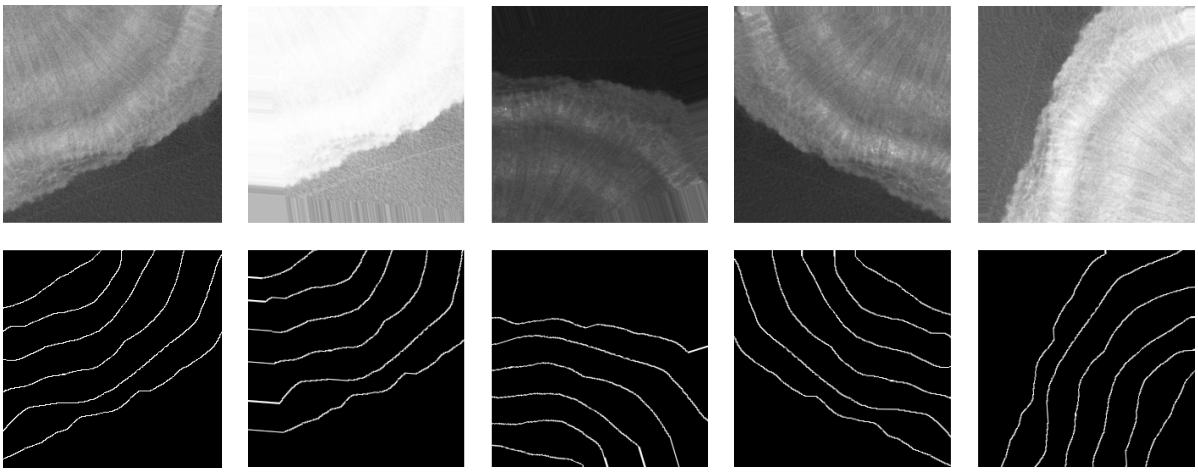
Since the `flow_from_directory` method applies random transformations to each image, the same random transformations must also be applied to the corresponding labels. This can be achieved by passing the same `seed` argument to the two `flow_from_directory` methods used to load the images and labels. These methods are also capable of shuffling the order in which images are loaded and trained with, so using the same `seed` value also ensures that the shuffling follows the same order for both the images and labels. The actual transformations to be used when augmenting the data are specified using various arguments passed to the `ImageDataGenerator` constructor.

Since the training generator yields randomly augmented image-label pairs indefinitely, a "classic" epoch will never be completed. Instead, when training the network, the `steps-per-epoch` argument defines how many batches should be processed before an epoch is considered complete. A simplified example demonstrating the implementation of the training generator and the transformations used is shown in Listing B.1.

Acceptable Augmentations

It was important to choose a range of possible transformations that would always produce a reasonable augmented image. The permitted augmentations that could be randomly chosen were:

- Rotations within a range of ± 2 degrees.
- Shifts in the x axis within a range of $\pm 2\%$ of the images' widths.
- Shifts in the y axis within a range of $\pm 2\%$ of the images' heights.
- Shears within a range of ± 2 degrees.
- Zooms in the x and y axes within a range of $\pm 2\%$ of the images' widths or heights.
- Brightness shifts within a range of $\pm 10\%$.
- Horizontal flips.
- Vertical flips.



(a) Original (b) Unrealistic (c) Realistic

Figure 3.6: Examples of augmented images created using both "unrealistic" and "realistic" sets of permitted transformations. The images are shown at the top and their corresponding labels are shown at the bottom. All labels are thresholded after augmentation to ensure that only white or black values are present. **(a)** The original image-label pair. **(b)** Two examples of image-label pairs augmented using a set of unrealistic permitted transformations resulting in images that do not represent the real coral data well. The first image's brightness has been shifted too far and the overly aggressive rotation and shifting has resulted in straight labels on the left-most edge which are not realistic. The second image's label has also been rotated too far resulting in a straight label on the right-most edge. **(c)** Two examples of image-label pairs augmented using the set of permitted transformations used in this project.

Using larger ranges of permitted transformations resulted in augmented images that did not represent the real data well. Examples of images created using both the ranges defined above and "unrealistic" ranges are shown in Figure 3.6. The shifts in brightness are particularly important as the calibration of the CT machine differs for each scan. As a result, a wide range of brightness distributions exist across different scans. This is discussed in further detail in Section 3.5.

3.2.4 U-Net

As discussed in Chapter 2, the U-Net architecture [45] is the main architecture experimented with. The implementation used throughout the project was initially based off of an implementation available on GitHub[4], though over the course of the project the majority of the code has been edited. It is written in Python and makes use of the Keras functional API. Recall that the original U-Net architecture consists of three sections: the contracting path, the bottleneck, and the expanding path. A high level diagram of the U-Net architecture is shown in Figure 3.7 and can be used to better understand the implementation details described.

The Contracting Path

The contracting path consists of multiple "blocks" with each block consisting of two convolutional layers followed by a 2×2 max-pooling layer with a stride of two. The convolutional layers both use 3×3 kernels and the ReLU activation function. Apart from the first block, the first convolutional layer of each contracting block doubles the number of feature channels. The implementation of the first contracting block is shown in Listing 3.1.

The `pool1` tensor defined in Listing 3.1 will be the input to the next contracting block, whose output will be the input to the next block, and so on. The first argument of the `Conv2D` function is the number of output feature channels. Since the number of feature channels doubles at each contracting block, the next contracting block would have two convolutional layers outputting 128 feature channels rather than 64. The second argument specifies the size of convolutional kernels to use, so 3×3 in this case. Note that the Keras `MaxPooling2D` function implicitly uses the same stride size as the pool size specified, so a stride of two in the x and y axes will be used.

[4] https://github.com/zhixuhao/unet

```
conv1 = Conv2D(64, 3, activation="relu", padding="same")(inputs)
conv1 = Conv2D(64, 3, activation="relu", padding="same")(conv1)
pool1 = MaxPooling2D(pool_size=(2, 2))(conv1)
```

Listing 3.1: The implementation of the first contracting block of the U-Net architecture using the Keras functional API.

```
up9 = UpSampling2D(size=(2, 2))(conv8)
up9 = Conv2D(64, 2, activation="relu", padding="same")(up9)
merge9 = concatenate([conv1, up9], axis=3)
conv9 = Conv2D(64, 3, activation="relu", padding="same")(merge9)
conv9 = Conv2D(64, 3, activation="relu", padding="same")(conv9)
```

Listing 3.2: The implementation of the last expanding block of the U-Net architecture using the Keras functional API.

The `"same"` padding argument specifies that the dimensions of the output feature channels should match the dimensions of the inputs. This differs from the implementation of the U-Net architecture used in the original paper. The original U-Net architecture would start with an oversized input. For example, to segment a 388×388 image, a 572×572 image would need to be used as input and the resulting output would have a resolution of 388×388. This decrease in resolution throughout the architecture is a result of no padding being used with each convolutional layer. In order to produce a larger image to be used as input to the network, the original image would be extrapolated via mirroring [45]. The use of the `"same"` padding allows the output image to be the same resolution as the input image preventing the need for any kind of extrapolation of the 256×256 patches.

The Bottleneck

The bottleneck contains only one block which also consists of two convolutional layers each using 3×3 kernels and the ReLU activation function, however no max-pooling layers are present. The bottleneck block is thus implemented similarly to the contracting block shown previously, only without the max-pooling layer. At this point, the number of output feature channels has reached 1024. Note that although the original U-Net architecture does not mention any dropout layers, a dropout layer is placed after the last block of the contracting path, and another dropout layer is placed after the bottleneck block. The dropout `rate` arguments that specify the probability that a given neuron is dropped during training were initially set to 0.5, but this value is experimented with and discussed further in Chapter 4.

The Expanding Path

The expanding path of the architecture described in the original U-Net paper consists of multiple blocks with each block containing a 2×2 "up-convolution" (or transposed convolution) layer followed by two convolutional layers each using 3×3 kernels and the ReLU activation function. A 2×2 transposed convolution using a stride of two in the x and y axes results in a doubling of the image size in both axes. The first convolutional layer of each expanding block halves the number of feature channels.

It is in these expanding blocks that the pass-forward of information from the contracting path takes place. The output feature channels from the corresponding blocks in the contracting path are appended to the feature channels that are produced by the transposed convolutions. This new set of feature channels containing both the feature channels from the contracting path, and the feature channels from the transposed convolution are then used as the input to the next convolutional layer in each expanding block. As discussed in Chapter 2, this allows the features that are learned whilst contracting the image to also be used to reconstruct it. The implementation of the last expanding block is shown in Listing 3.2.

Looking at Listing 3.2 it can be seen that the 2×2 transposed convolution used in the original U-Net architecture is replaced with a 2×2 upsampling layer (with an implicit 2×2 stride) followed by a regular 2×2 convolution. The original U-Net paper also does not mention an activation function being applied to the output of the transposed convolutions, whereas a ReLU activation is used here. These architectural design choices stem from the initial implementation used and although these do differ from the original

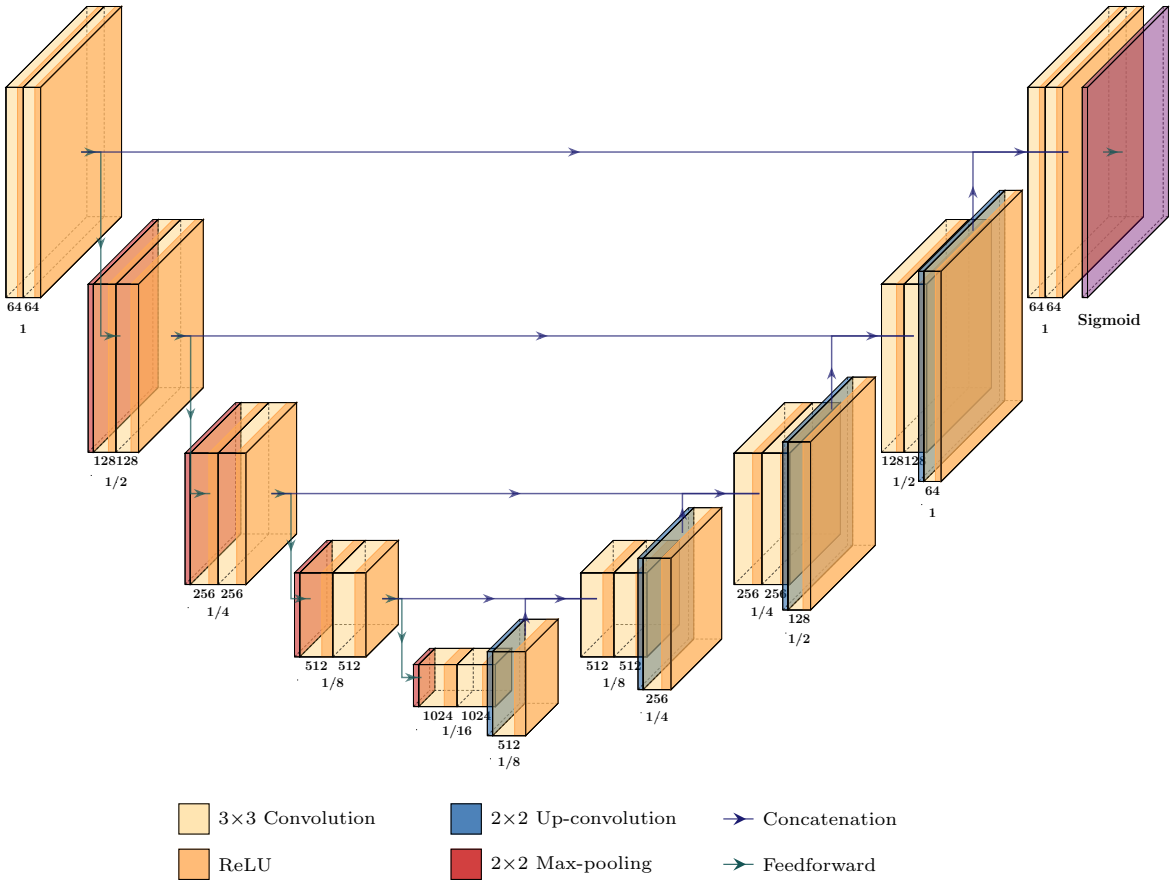


Figure 3.7: A diagram of the U-Net architecture (created using an open-source CNN architecture visualisation tool[5]). The architecture is named after the U-shape that it resembles when illustrated in this way. Each cube represents a layer with the number of output feature maps shown at the bottom of the cube. At the bottleneck there are 1024 feature maps that are each one sixteenth the size of the input image in the x and y dimensions. The green arrows represent the feedforward of information from one layer to the next, whereas the blue arrows represent the concatenation of the feature maps of one layer to another. Note that the max-pooling layers at the end of each contracting block are shown at the start of the next block for easier illustration of the pass forward of the feature channels. Similarly, the up-convolution layers are shown at the end of each contracting block rather than at the start of the next.

architecture, the effects on the performance achieved are negligible. Note that the output feature channels of the `up9` layer are appended to the feature channels of the `conv1` layer from the first contracting block using the `concatenate` function.

Output

After the last expanding block, a final 1×1 convolutional layer is present in the original U-Net architecture. This convolutional layer has two output feature channels, one corresponding to each possible class that a pixel could be classified as. Although the original paper does not mention an activation function of any kind, the sigmoid activation function is used in this implementation ensuring that the output values represent the probability that a given pixel is "part of a boundary" or "not part of a boundary". As mentioned earlier, a high level diagram of the U-Net architecture is shown in Figure 3.7 and can be used to better understand the implementation details described.

Finally, the model is stored in an instance of the Keras `Model` class. Both the input and output tensors are passed as arguments to the `Model` constructor and the resulting object contains all of the layers and their connections as attributes.

[5] https://github.com/HarisIqbal88/PlotNeuralNet

Table 3.2: A summary of the initial hyperparameter settings used.

Hyperparameter	Setting
Architecture	2D U-Net
Optimisation Algorithm	Adam
Learning rate	0.0001
Loss function	Cross-entropy
Epochs	20
Steps per epoch	500
Batch size	2

3.2.5 Training

Once the architecture and online data augmentation process had been implemented, the model was then compiled using the `compile` method from the `Model` class. The `compile` method is used to define which optimiser and loss function to use and which metrics to monitor. Subsequently, the `fit` method can be used to train the network.

Keras Callbacks

The Keras callbacks are objects that can perform actions at various stages of training (e.g., at the start or end of epochs)[6]. Callbacks were used to implement early stopping, model checkpointing, and metric visualisation using TensorBoard[7].

As discussed in Chapter 2, early stopping is a regularization technique in which the training process is stopped given some "stopping criteria" rather than after a set number of epochs. The Keras library allows users to implement early stopping using the `EarlyStopping` callback. Both the metric to monitor and a `patience` argument must be provided to the `EarlyStopping` callback. In this case, the loss achieved on the validation set is monitored. The `patience` argument defines the number of epochs that the validation loss can increase before the training is stopped.

Model checkpointing and TensorBoard visualisation are implemented with the `ModelCheckpoint` and `TensorBoard` callbacks respectively. The `ModelCheckpoint` callback simply saves the parameters of the entire network after each epoch to a `.hdf5` file. Using the `save_best_only=False` argument and a filename formatted using the current epoch number ensures that each epoch is saved as a new file, rather than the previous epoch's save being overwritten. Example usage of the callbacks discussed is shown in Listing B.2.

Initial Results

A few examples of the initial results achieved are shown in Figure 3.8. These initial results were output by a network trained using the Adam optimiser [28] with a learning rate of 0.0001 to optimise the binary cross-entropy loss function. The network was trained for 20 epochs with each epoch consisting of 500 augmented batches. A batch size of two was used so a total of $20 \times 500 \times 2 = 20{,}000$ augmented samples were seen by the network. These hyperparameters are summarised in Table 3.2. An average per-pixel accuracy of 96.7% was achieved. Note that this accuracy metric is misleading and is discussed in Section 3.4. The entire training process took ∼16 minutes on an Nvidia P100 "Pascal" GPU. Many more examples of boundary predictions resulting from networks trained with different hyperparameters are shown throughout Chapter 4. Listing B.2 shows a simplified example of the compilation of a network configured with these hyperparameters.

3.3 Three Dimensional Boundary Extraction

This section discusses the attempts to implement and train a modified U-Net architecture capable of extracting the annual density banding present in three dimensional data. An overview of the components implemented in this section is shown in Figure 3.9. As opposed to only utilising the density information

[6] https://keras.io/api/callbacks
[7] https://www.tensorflow.org/tensorboard

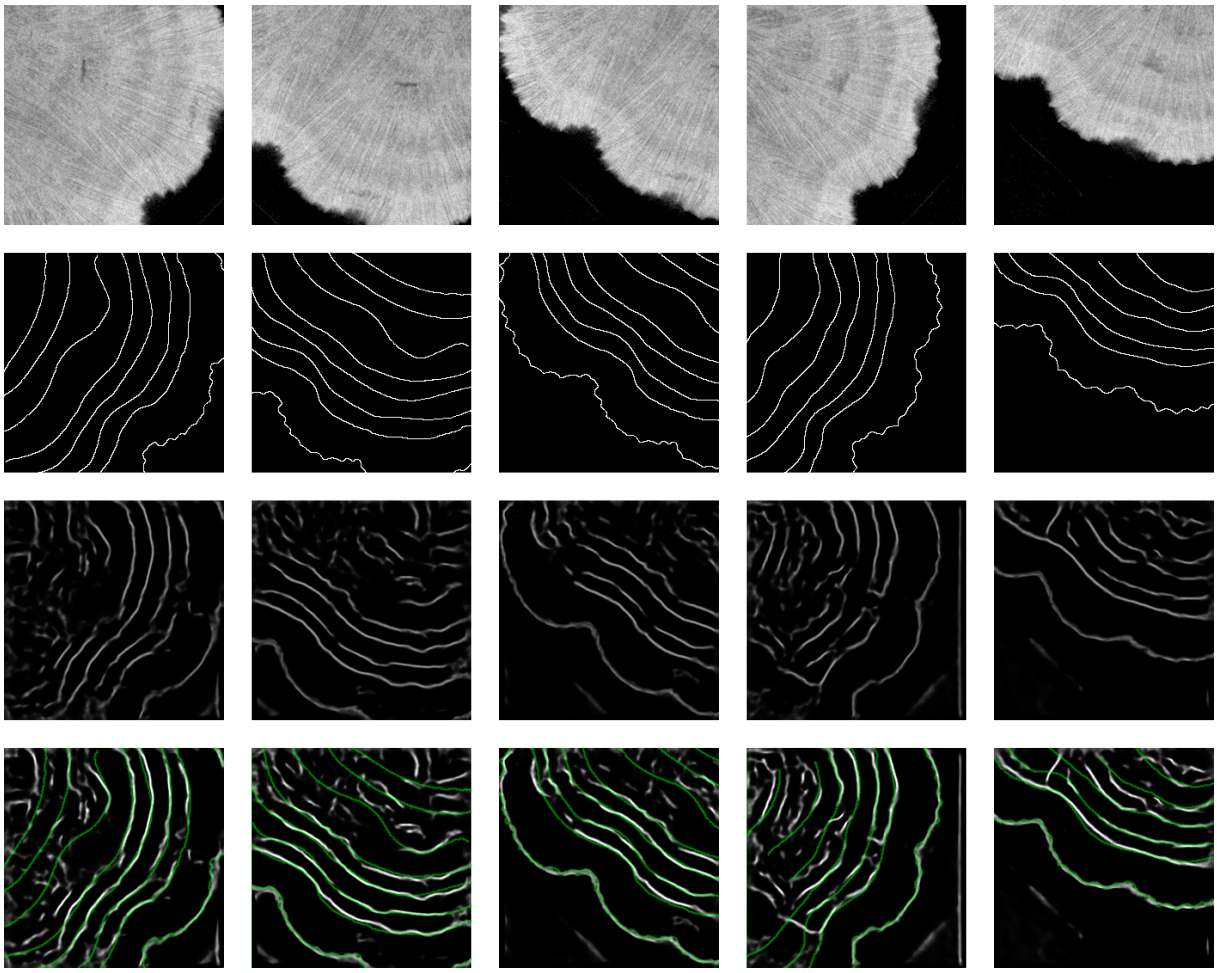


Figure 3.8: Initial results achieved by the network. Note that the hyperparameters chosen are not necessarily optimal and that performance may still be improved. **(top)** The input samples from the test set. **(centre-top)** The ground truth labels. **(centre-bottom)** The predictions output by the network. **(bottom)** The ground truth labels (in green) overlaid with the predictions.

available in a single slice, the modified architecture also makes use of the density information in surrounding slices when determining boundary positions. Due to the corallite structures present in the coral data, an architecture that could make use of a third dimension could potentially achieve better performance. The corallite structures are the thin bright lines visible in the coral images that are perpendicular to the growth surface of the coral. It can be seen in Figure 3.8 that the corallite structures are sometimes even predicted as density band boundaries by the two dimensional architecture. Whilst corallite structures vary significantly in adjacent slices due to their small size (often less than a millimetre in diameter), the annual density banding is very consistent across even tens of slices. An architecture that could make use of these commonalities across slices could thus perform significantly better.

3.3.1 Dataset Expansion

In order to train the modified architecture, the curated dataset discussed in Section 3.2.2 was expanded to include data from slices surrounding the slices initially labelled. Since the modified architecture makes use of up to nine slices (the central slice and four slices on either side), the eight slices surrounding the labelled slices also needed to be extracted. As mentioned in Section 3.2.1, when the sliding window Python script was used to create the 2D dataset, patches from surrounding slices were also produced. The resulting 3D dataset contains the same amount of samples (388) with each sample consisting of nine patches and the label consisting of the central labelled patch. The nine patches composing each sample are stored with the same name as the label apart from a suffix of `_0` to `_8` denoting the patches' positions relative to the central patch which has a suffix of `_4`.

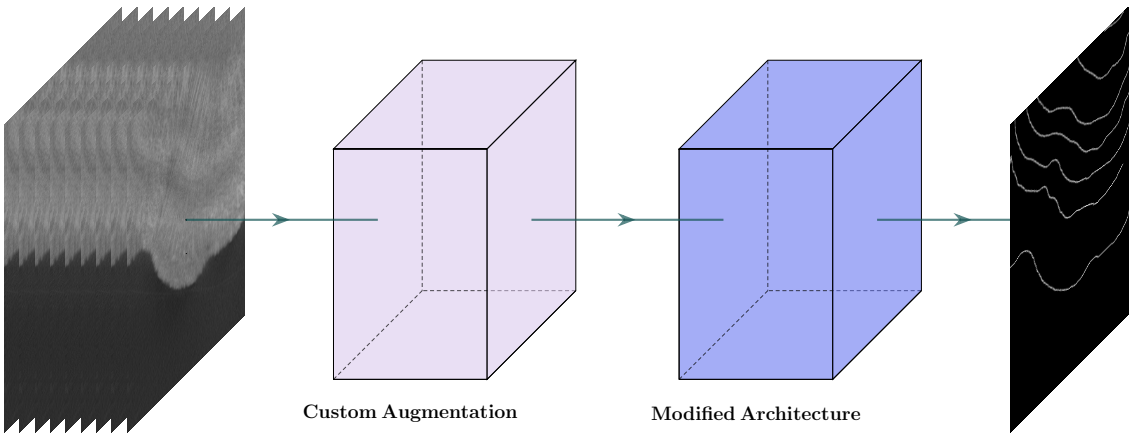


Figure 3.9: An overview of the components implemented in this section. Each training sample that the modified architecture trains on will consist of nine patches. Only the centre slice is labelled and the model attempts to extract the boundaries present in this central slice. A custom 3D "data loader" is implemented in order to perform online augmentation of the nine-patch training samples.

3.3.2 Data Loader Implementation

Whilst the 2D architecture made use of the `ImageDataGenerator` class to load 2D images into the model, a 3D version of the `ImageDataGenerator` class is not currently implemented in the Keras library. Although the `ImageDataGenerator` class could be used to simply load nine patches one after another and the Keras `stack` method could be used to stack the images to produce a $(9 \times y \times x \times 1)$ tensor, this method would not allow for online augmentation to be performed since different random transformations would be applied to all nine of the loaded patches. Note also that even if the same transformations were able to be applied to each 2D patch, transformations in the third dimension would not be possible. Thus, a custom data loader was written.

An `ImageDataGenerator3D` class and a `LabelDataGenerator2D` class were implemented. These were designed to be similar to the Keras `ImageDataGenerator` class to allow the same data augmentation arguments to be passed to their constructors. Recall that in order to produce 2D image-label pairs, a pair of generators created using the `flow_from_directory` method were used with the same `seed` to augment the image-label pairs with the same transformations. The online augmentation of the 3D data is designed to follow a similar structure; a pair of generators created using the `flow_from_directory` method of the `ImageDataGenerator3D` and `LabelDataGenerator2D` classes with the same `seed` argument are used to perform online augmentation to both the nine-patch sample and the single label patch. The `ImageDataGenerator3D` class is used to load and augment the 3D samples and the `LabelDataGenerator2D` class is used to load and augment the 2D labels.

The Custom `flow_from_directory` Methods

The `flow_from_directory` method of the `ImageDataGenerator3D` class first sorts the `.png` images from the sample directory alphanumerically. Nine images are then loaded, stacked, and "unsqueezed" resulting in a $(9 \times y \times x \times 1)$ tensor. The unsqueeze operation takes for example a $(y \times x)$ tensor and produces a $(y \times x \times 1)$ tensor. These stacks of nine images are then augmented using techniques discussed later in this section and are finally stacked into $(batch \times 9 \times y \times x \times 1)$ tensors where $batch$ is the batch size.

The `flow_from_directory` method of the `LabelDataGenerator2D` class also sorts the `.png` images from the label directory alphanumerically. These images are loaded and unsqueezed resulting in $(y \times x \times 1)$ tensors each containing a single label. The appropriate augmentation transformations are then applied and the resulting tensors are stacked into $(batch \times y \times x \times 1)$ tensors.

The two generators produced by the `flow_from_directory` methods of the `ImageDataGenerator3D` and `LabelDataGenerator2D` classes were then zipped together to create a single generator that yielded sample-label pairs with each sample consisting of nine patches, and each label consisting of only a central labelled patch. Similarly to the Keras `flow_from_directory` methods used when loading and augmenting the 2D data, these custom methods could also accept a `seed` argument ensuring that the same transformations

```
conv1 = Conv3D(8, 3, activation="relu", padding="same")(inputs)
conv1 = Conv3D(8, 3, activation="relu", padding="same")(conv1)
pool1 = MaxPooling3D(pool_size=(1, 2, 2))(conv1)
conv1 = Reshape((256, 256, 72))(conv1)
```

Listing 3.3: The implementation of the first contracting block of the modified U-Net architecture using the Keras functional API.

are applied to the sample-label pairs. The custom `flow_from_directory` method was also capable of shuffling the order in which samples and labels were loaded, and `seed` argument ensured that the samples and labels were shuffled in the same order.

Example usage of the `ImageDataGenerator3D` and `LabelDataGenerator2D` classes in order to perform online 3D augmentation is shown in Listing B.3.

Three Dimensional Transformations

The custom data loader enabled the following transformations:

- Random rotations in the x, y, and z dimensions within a range of ± 2 degrees.
- Random flips in the x, y, and z dimensions.
- Random brightness shifts within a range of $\pm 10\%$.

Although these are the ranges used to train this particular model, these transformations can use values within any given range, not just the ranges specified above.

3.3.3 Architecture Modification

Although various modified architectures were experimented with and are discussed in Chapter 4, only the initial modified architecture experimented with will be described here.

At a high level, the modification consisted of replacing all of the 2D operations in the contracting path with their 3D counterparts. However, issues arise when a tensor output from a 3D layer is passed to a 2D layer. Whereas the Keras `Conv3D` layer requires an input tensor with shape $(batch \times z \times y \times x \times channels)$ and produces a tensor with the same shape, the `Conv2D` layer inputs and outputs tensors with shape $(batch \times y \times x \times channels)$. Thus, in order to pass the resulting tensor to a 2D layer, the tensor of shape $(batch \times z \times y \times x \times channels)$ must be reshaped into a $(batch \times y \times x \times (z \times channels))$ tensor. A diagram of the modified U-Net architecture is shown in Figure 3.10.

The implementation of the first block of the modified contracting path is shown in Listing 3.3. It can be seen that the two 2D 3×3 convolutional layers have been replaced with 3D convolutional layers using 3×3×3 kernels (the kernel size is defined by the second argument of the `Conv3D` method). It can also be seen that although the max-pooling layer still uses a pool size and stride of two in the x and y axes, a value of one is instead used in the z axis. A pool size and stride of two cannot be used in the z axis as the input tensors only have a z dimension of nine. Since there are four max-pooling operations, the tensor would be reduced to a size of just one in the z dimension[8].

After the convolutional and max-pooling layers, the tensor is then reshaped to a $(batch \times 256 \times 256 \times 72)$ tensor allowing it to be concatenated to the feature channels of the 2D layers present in the expanding path. Note that there are only eight output feature channels. Since the number of output feature channels will be multiplied by nine, a number of feature channels was chosen in order to make the reshaped tensor as close as possible to the $(batch \times 256 \times 256 \times 64)$ tensor that was originally used in the 2D architecture. Various numbers of output channels were experimented with and are discussed in Chapter 4.

The remaining contracting blocks are implemented similarly to the first block shown in Listing 3.3. The bottleneck and expanding path are implemented as they were in the 2D architecture.

[8] Assuming that padding was used at each stage to pad the z dimension to an even number.

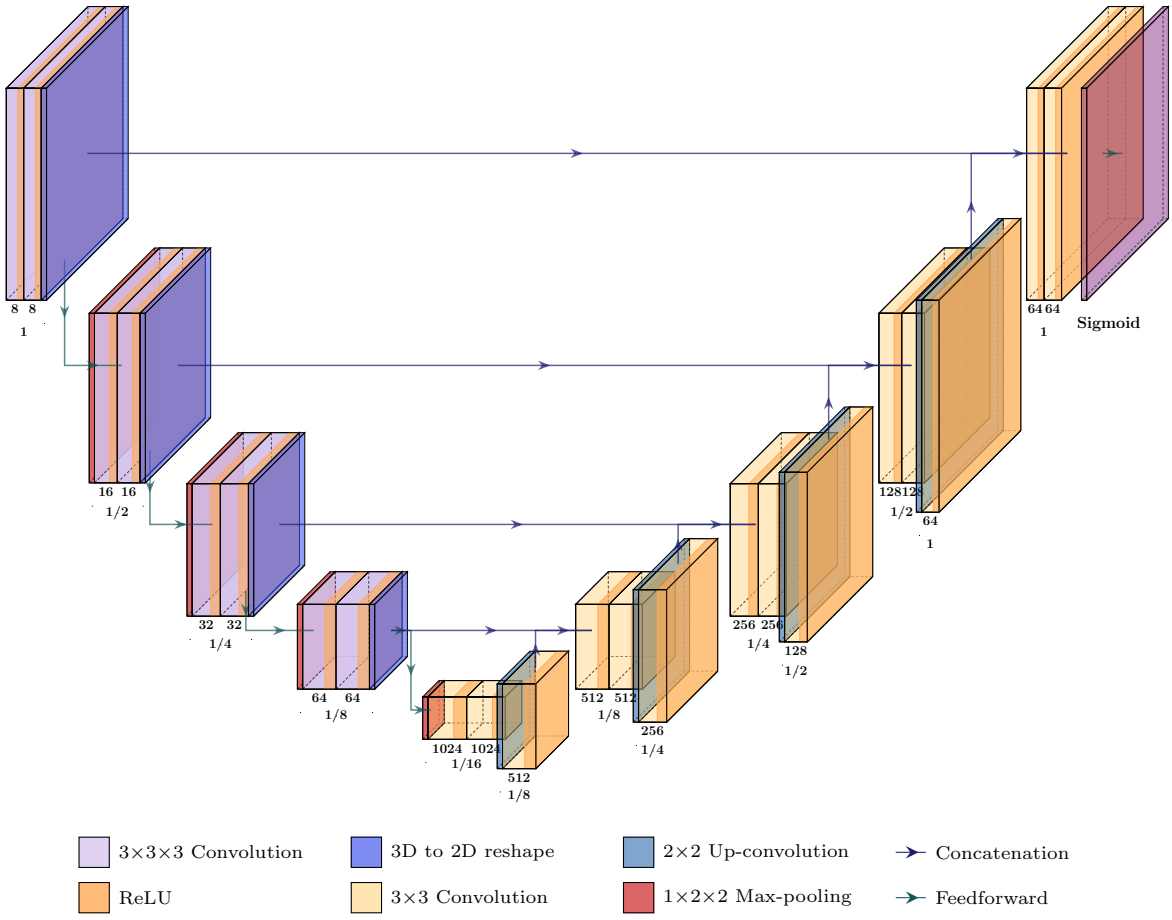


Figure 3.10: A diagram of the modified U-Net architecture (created using an open-source CNN architecture visualisation tool[5]). Each cube represents a layer with the number of output feature maps shown at the bottom of the cube. The green arrows represent the feedforward of information from one layer to the next, whereas the blue arrows represent the concatenation of the feature maps of one layer to another. Note that the max-pooling layers on the contracting blocks are three dimensional 1×2×2 max-pooling layers whereas the bottleneck max-pooling layer is simply a two dimensional 2×2 max-pooling layer.

3.3.4 Fully Three Dimensional Architecture

A fully 3D U-Net architecture was also implemented and experimented with. This network attempts to predict the boundaries present in all nine sample patches rather than just the central patch. In this case, the architecture modification consisted of replacing all of the 2D operations present in the network with their 3D counterparts. A pool size and stride of one was again used in the z axis of the max-pooling layers.

In order to train this fully 3D architecture, a new dataset containing nine-patch samples and nine-patch labels was curated using the same sliding window technique. Since only a small selection of slices were initially labelled to curate the original 2D dataset, a larger selection of slices adjacent to the initially selected slices were manually labelled. Ultimately, labelling enough data for this model to perform well proved challenging—curating a dataset with the same amount of samples used to train the 2D architecture would have required nine times the amount of labelled slices.

Both the experimentation with this architecture and the challenges involved in labelling 3D data are discussed further in Chapter 4.

3.3.5 Initial Results

Examples of the initial results achieved by the modified architecture that predicts the boundaries present in only the central patch are shown in Figure 3.11. These initial results were output by a network trained

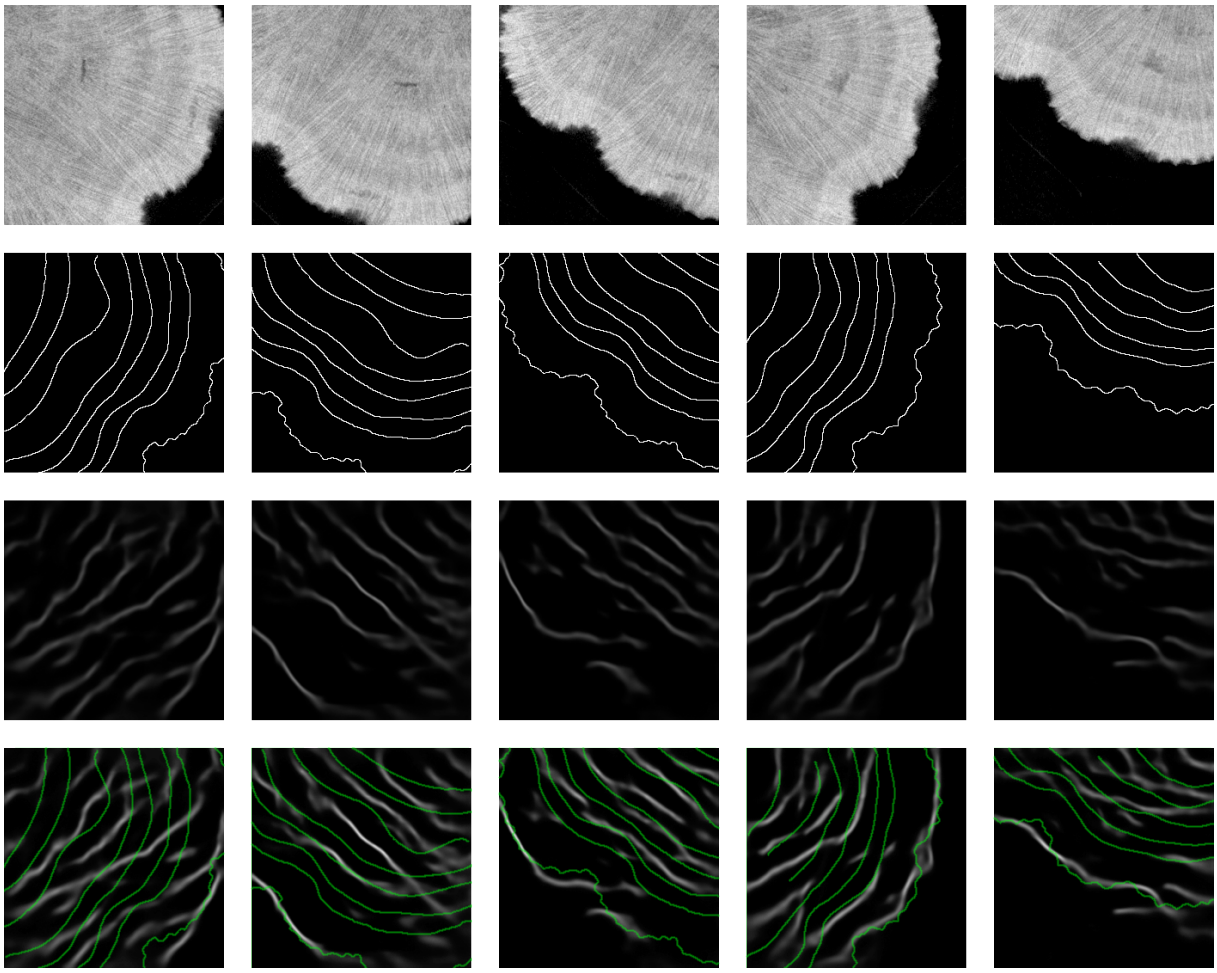


Figure 3.11: Initial results achieved by the modified network that predicts a central label only. Note that the hyperparameters chosen are not necessarily optimal and that performance may still be improved. **(top)** The input samples from the test set. **(centre-top)** The ground truth labels. **(centre-bottom)** The predictions output by the network. **(bottom)** The ground truth labels (in green) overlaid with the predictions.

using the Adam optimiser with a learning rate of 0.0001 to optimise the binary cross-entropy loss function. The network was trained for 20 epochs with each epoch consisting of 500 augmented samples. A batch size of two was used so a total of $20 \times 500 \times 2 = 20{,}000$ augmented nine-patch samples were seen by the network. The entire training process took ∼50 minutes on an Nvidia P100 "Pascal" GPU.

3.4 Accuracy Metric Implementation

As discussed briefly in Chapter 1, choosing an appropriate accuracy metric for the boundary extraction task was particularly challenging. This section discusses the need for a custom accuracy metric and then details its implementation.

3.4.1 Motivation

There are two major issues that arise when using a standard per-pixel accuracy metric to quantify the performance achieved in this task. The first issue stems from the severe class imbalance present in the labels. On average, ∼98% of a labelled patch is black—part of the "not part of a boundary" class. As a result, a model that learned to output a pure black image would achieve an average accuracy of 98%. The inherent class imbalance gives rise to another problem when assessing the performance of the model; since the boundary labels are only one pixel wide, a model that predicts boundary positions that are just one pixel off of the correct positions could potentially achieve an accuracy of 0%. The accuracy metric used must ensure that a model that predicts boundaries a few pixels away from the manually

(a) Original prediction (b) Thresholded prediction (c) Skeletonized prediction

Figure 3.12: An example of the skeletonization of a prediction output by the 2D architecture. The image is first thresholded using the Otsu method and then skeletonized using Zhang's method. Similarly to the ground truth labels, the resulting image contains white boundaries that are only one pixel wide.

labelled positions is still rewarded. Thus, a metric based off of the average Euclidean distances between the predicted and ground truth boundaries was conceived.

3.4.2 Methodology

There are multiple steps involved in the calculation of the custom accuracy. First, the prediction output by the model is thresholded using the Otsu method [32]. The Otsu method finds the threshold value that minimises the weighted within-class variance. Once this threshold value t is found, all pixels with values less than t are set to 0 and all pixels with values greater than t are set to 255. This thresholded image can then be "skeltonized". Skeltonization is the process of reducing the foreground regions of binary images to just one pixel wide representations. The scikit[9] implementation of Zhang's method [60] is used to skeletonize the image. Introduced in 1984, Zhang's method iteratively removes pixels present on the foreground region borders until no more pixels can be removed. An example of the skeletonization of a prediction using this method is shown in Figure 3.12.

Average Distance Calculation

Once the prediction is skeletonized, the "average distance" from the prediction to the ground truth label can be calculated. For each white pixel in the skeletonized prediction, the closest white pixel in the label is found. These distances are calculated as if the images were overlaid with each other. For example, a white pixel with position (24, 36) in the skeletonized prediction would have a Euclidean distance of zero from a white pixel also with position (24, 36) in the ground truth label.

Once these closest distances have been found for each pixel, they are averaged to calculate the average distance from the prediction to the label. This process is then repeated in order to calculate the average distance from the label to the prediction. These two distances are then averaged once again to calculate a final average distance between the two images. Note that the average distance from the prediction to the label can be significantly different from the average distance from the label to the prediction, and so averaging the two is important.

Since the metric relies on the average distance between white pixels, issues arise when a pure black image is predicted by the network. If a prediction contains no white pixels, it is not clear what the distance between the label and the prediction should be. It is important for a prediction of an all black image to achieve a low accuracy metric, so a pure black image is simply given a distance of ∞ which corresponds to an accuracy of 0%. This ensures that all samples receive an accuracy and can contribute to the average accuracy achieved.

[9] https://scikit-image.org

Accuracy Calculation

The average distance between the prediction and the label can be thought of as an error metric rather than an accuracy metric; the lower the distance, the better the model has performed in producing that prediction. Accuracy metrics are often represented as a percentage where a higher percentage value corresponds to better performance. In order to represent this accuracy metric as a percentage, the accuracy achieved is calculated as

$$\frac{d_{max} - d}{d_{max}} \times 100 \tag{3.1}$$

where d_{max} is the maximum distance that two images could be from each other, and d is the actual distance calculated. d_{max} would theoretically be achieved if the prediction contained a single white pixel in one of the corners, and the ground truth label contained a single white pixel in the opposite corner. This would result in a maximum theoretical distance of 362 for 256×256 patches. However, since the label patches present in the curated dataset must contain labelling across the entire patch, this scenario can never occur. The maximum distance achieved by any sample during training was actually 131 pixels. Ultimately, a maximum of 90 pixels was chosen, with errors above 90 pixels resulting in a truncated accuracy of 0%. This lower maximum (in comparison with the theoretical maximum of 362) was chosen to ensure that accuracy percentages over 90% correspond to predictions that are often considered acceptable when inspected visually.

The initial two dimensional U-Net implementation achieved a validation accuracy of ∼94% and the initial three dimensional implementation achieved a validation accuracy of ∼77%. The accuracy metric and its various strengths and weaknesses are discussed further in Chapter 4.

3.4.3 The `ctypes` Module

Since the entirety of the project is written in Python, the accuracy metric calculation was also written in Python, allowing it to be easily integrated into the training process. However, when written in Python, the calculation of the custom accuracy metric for a single sample took an average of 38 seconds. Calculating the average accuracy achieved across the entire training set would therefore take up to three hours. Since the distances from a white pixel in one image to all white pixels in the other must be calculated in order to find the minimum, the time complexity of the algorithm is $O(n^2)$ and this computation is unavoidable.

In order to reduce the time taken to calculate the accuracy metric, an implementation was written in C and the `ctypes` module was used to wrap the C implementation in Python. The C code is compiled into a `.so` library. Using the `ctypes` module, this library can be loaded and its methods can be accessed in Python.

Once implemented and wrapped in a Python function, the C implementation took an average of 0.12 seconds to calculate the accuracy metric for a single sample[10]. This provides a ∼315 times speed up compared to the pure Python implementation, and allows for the accuracy across the entire training set to be calculated in under a minute.

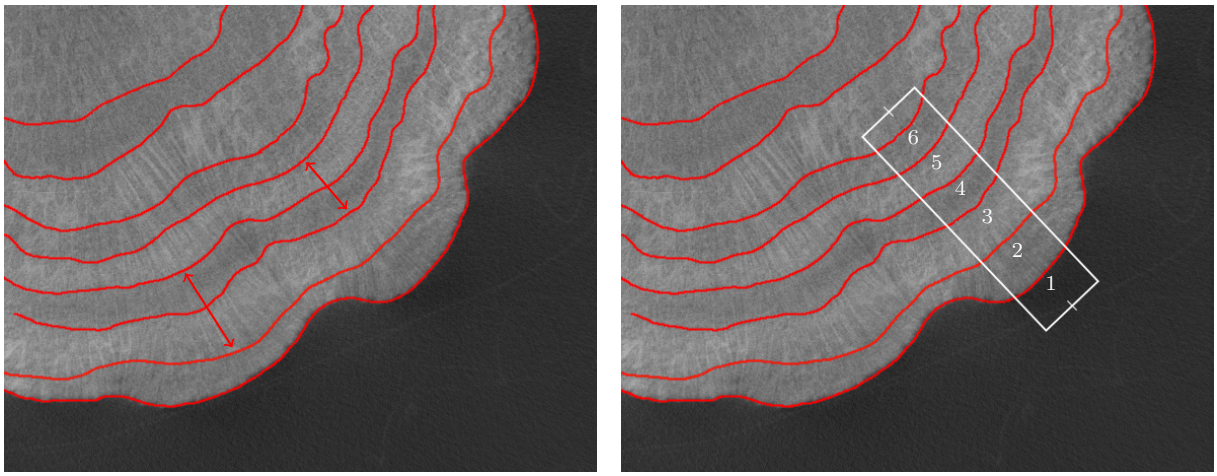
3.5 Calcification Rate Estimation

As outlined in Chapter 1, the objective of this project is to utilise the implemented networks to estimate the calcification rate—the mass of skeletal matter produced annually. In order to estimate the calcification rate, two values are required: the linear extension rate (the distance that the coral grows per year) and the density of the matter being produced. The calcification rate can then be calculated by multiplying these two values.

3.5.1 Linear Extension Rate Estimation

In order to estimate the linear extension rate, the average distance grown annually in terms of voxels must first be calculated. Since an idealised annual density cycle is in the form of a sinusoidal wave, with the

[10]The times achieved by the Python and C implementations are taken when using an Intel Core i5 9400 processor.



(a) (b)

Figure 3.13: **(a)** Examples of the linear extension rate. The distance grown in a year is the distance between every second boundary. The bottom arrow starts at the end of a high density band and finishes at the end of the next high density band, whereas the top arrow starts at the end of a low density band and finishes at the end of the next low density band. It can be seen that even though both of the arrows shown are valid measurements of the linear extension rate, the distances measured at different points in the skeleton can vary significantly. **(b)** An illustration of the rectangular area created given two points: one inside and one outside the skeleton. CCA is then used to uniquely label the boundaries from the outside in.

density gradually changing from high to low and back over the course of a year [37, p. 39], the distance grown in a year is actually the distance between every second boundary (see Figure 3.13a).

A Python script was written that takes an image and splits it into overlapping 256×256 patches using a sliding window technique similar to that utilised in Section 3.2.2. The boundaries present in these patches are then predicted by the network and the boundaries present in the entire image are reconstructed. During this reconstruction, only the inner areas of the predictions are taken into account since the predictions made at the edges of patches are often of a lower quality due to the lack of context available. For example, looking back at Figure 3.12, it can be seen that the outermost pixels of the boundaries are more blurry and tend to be less confidently predicted by the network. Once the boundaries across the entire image have been predicted and reconstructed, they are finally skeletonized.

A second script then takes the skeletonized predictions and two sets of coordinates. The two supplied sets of coordinates should correspond to two points on the image. The first should be a point outside the growth surface of the coral, and the second a point inside the coral skeleton. When a line is drawn between these supplied points, the line should be parallel to the direction of growth of the coral, as this ensures that the distances measured between the boundaries are a more accurate representation of the linear extension rate. Using these two supplied points, a rectangular area is created as shown in Figure 3.13b. This area is not only used to determine the linear extension rate, but also to calculate the average density of the skeletal matter later on.

The OpenCV `connectedComponents` method is then used to perform connected-component analysis (CCA). This method determines the positions of the skeletonized boundaries present in the rectangular area and supplies each with a unique label. Once these unique labels are determined, the boundaries are sorted by the order in which they occur relative to the outside of the skeleton (e.g., the growth surface first, followed by the outermost boundary, and so on). The distances between these boundaries are then calculated using an average of the minimum Euclidean distances from one boundary to the next (i.e., for each pixel in one boundary, the closest pixel in the next boundary is found, and the distance from the first boundary to the next is the average of these closest distances for each pixel).

Since the distance grown annually is the distance between every second boundary, the number of pixels grown per year can be estimated by multiplying the average distance between the boundaries by two. The resulting value represents the linear extension rate in terms of pixels per year. In order to convert this value to millimetres per year, it must be multiplied by the pixel size. The pixel size is the real distance that each pixel represents and is constant across all three dimensions (i.e., a voxel represents the same

distance in the x axis as it does in the z axis). A common pixel size for the scans composing this dataset is ~0.1 mm. Once the linear extension rate has been multiplied by this pixel size, the linear extension rate in terms of millimetres per year (mm y^{-1}) has been calculated.

3.5.2 Density Estimation

In order to calculate the density that each pixel value corresponds to, a calibration curve must be fit. Given the values of five densities that are known to correspond to five greyscale values, the non-linear relationship between a pixel's density and its corresponding greyscale value can be determined. This calibration curve is fit using `curve_fit`[11], the SciPy implementation of the non-linear least squares curve fitting method. In this case, a quadratic function is fit to the data. Once this function that maps a pixel's greyscale value to a corresponding density is found, it can be used to estimate each pixel's density in g cm^{-3}.

When estimating the average density of the skeletal matter produced, only pixels that lie within the rectangular area (seen in Figure 3.13b) are taken into account. This ensures that the same skeletal matter used to estimate the linear extension rate is used to estimate the average density. However, since this rectangular area contains both pixels that are within the skeleton and pixels that are not, two mean density values will exist. In order to find the mean that corresponds to the pixels that are within the coral skeleton, the Scikit-learn implementation of the K-means algorithm [38] is used[12]. It is assumed that only two means should be present—one corresponding to the skeletal pixels, and one corresponding to the other pixels—so the `n_clusters` argument is set to two. It is also assumed that the mean corresponding to the pixels within the coral skeleton should be higher than the mean corresponding to the pixels outside since coral skeletons are denser than air. Thus, the maximum of the two means discovered is taken to be the average density value of the skeletal matter that lies within the rectangular area.

With the linear extension rate (cm y^{-1}) and the density (g cm^{-3}) estimated, these two values can now simply be multiplied together to calculate the calcification rate (g cm^{-2} y^{-1}). The calcification rates estimated for various scans are discussed in Chapter 4.

3.6 Version Control

The Git version control system was used to backup and keep track of various iterations of the networks and other components implemented throughout the project. All of the code written in this project is publicly available on GitHub[13].

[11] https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.curve_fit.html
[12] https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html
[13] https://github.com/ainsleyrutterford/DeepC

Chapter 4

Results and Evaluation

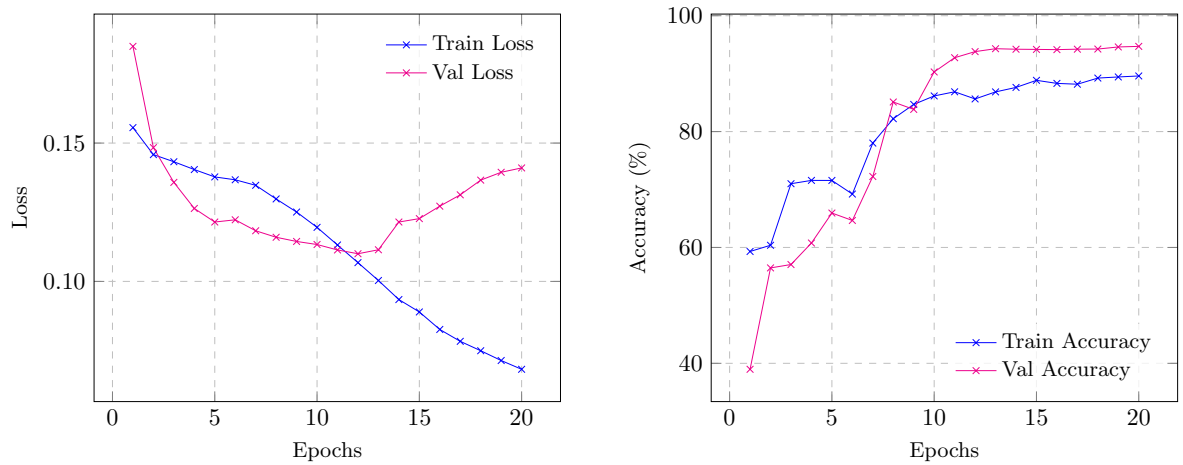
This chapter first discusses the experiments carried out in order to both improve the performance, and gain a better understanding, of the sub-components implemented. Experiments such as hyperparameter optimisation and ablation studies of the two dimensional architecture are discussed. Once the results are cross-validated, the final results achieved by the optimised models are then presented, interpreted, and compared to results achieved by alternative models. Finally, various aspects of the project are critically evaluated.

4.1 Two Dimensional Experimentation

This section reintroduces the initial results achieved by the "baseline" two dimensional architecture implemented in Chapter 3 and then outlines the experiments carried out in an attempt to improve the performance both qualitatively and quantitatively.

4.1.1 Initial Results

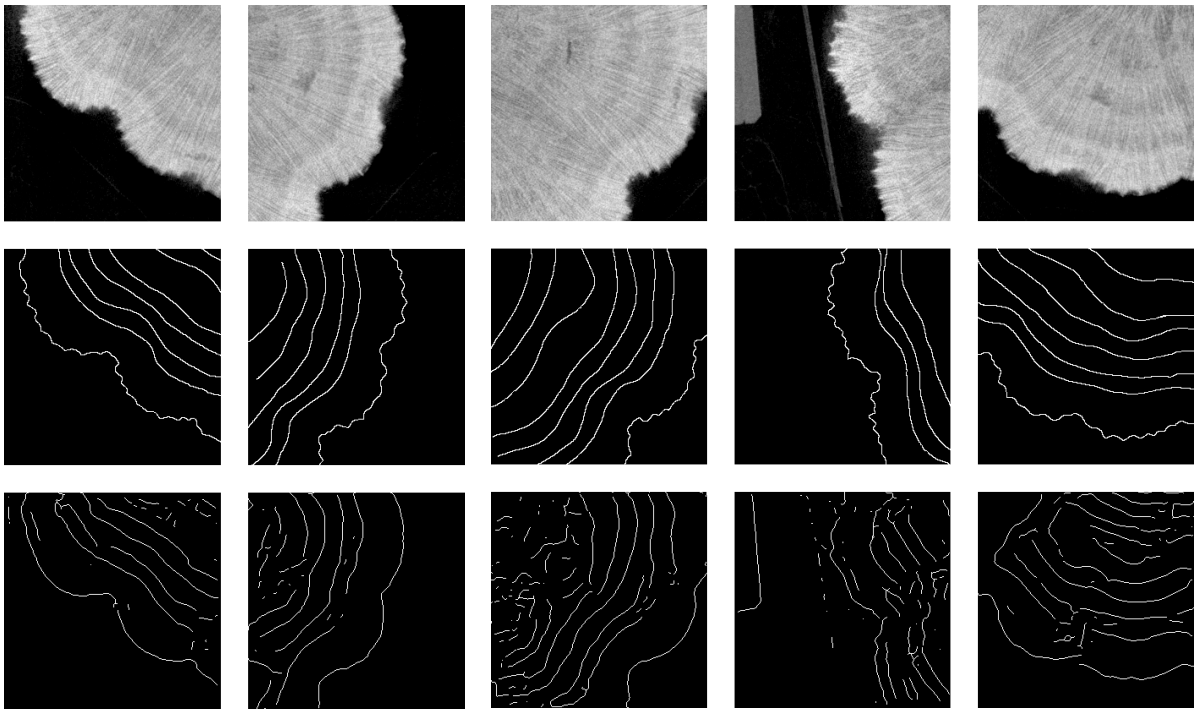
The initial results achieved by the baseline model can be discussed further in order to gain a better understanding of the strengths and weaknesses of the model. The loss and accuracy curves are shown in Figure 4.1, and example boundary predictions of both high and low "quality" are shown in Figure 4.2.



(a) Loss curve

(b) Loss curve

Figure 4.1: The loss and accuracy curves achieved when training the model over 20 epochs using the hyperparameters outlined in Table 3.2. It is important to note that although the validation loss begins to increase after epoch 13, the validation accuracy does not begin to decrease. Upon visual inspection of the boundary predictions, it could also be seen that the predictions were qualitatively better when the network was trained for 20 epochs rather than 13. The desirable qualitative features of a prediction are outlined in Figure 4.2.



(a) Higher accuracy: 96.8% and 96.5%

(b) Lower accuracy: 92.1%, 88.4%, and 91.5%

Figure 4.2: Examples of both high and low quality outputs from the network trained using the initial hyperparameters outlined in Table 3.2. The top images are patches from the validation set, the centre images are the corresponding ground truth labels, and the bottom images are the outputs of the network after they have been skeletonized. **(a)** Examples of high quality predictions. These examples achieved higher accuracies compared to the rest of the samples in the validation set. It can be seen that the boundaries contain comparatively less "noise" than the lower quality examples. **(b)** Examples of low quality predictions. The first example achieves a lower accuracy score due to the noise present in the inner section of the coral (on the left side of the image). The second example received a particularly low accuracy score due to the accidental classification of the object in the top left of the image as a boundary. This object is the platform that the coral sample is placed on in the CT machine and should perhaps have been cropped from the slices that these patches were produced from. The third example receives a low accuracy due to the accidental misclassification of a boundary perpendicular to the skeleton's surface. As discussed in Chapter 1, density banding boundaries should be parallel to the coral skeleton surface.

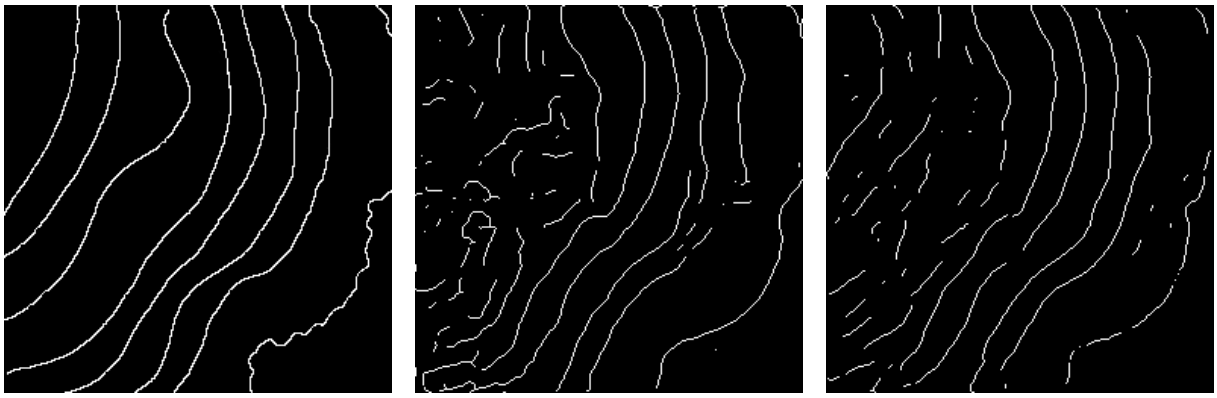
Their quality is assessed both via the accuracy metric achieved and via visual inspection. Looking at Figure 4.1, it can be see that the accuracy achieved on the validation set is noticeably lower than the accuracy achieved on the training set. This unusual behaviour may be due to the validation set containing "easier" examples in which the annual banding is more obvious. Although the performance reported on this validation set may potentially be positively skewed, the final performance achieved by the model will be reported on cross-validated results, and so the selection biases that arise from this dataset split should not affect the final reported performance.

4.1.2 Hyperparameter Optimisation

As discussed in Chapter 2, the performance achieved by deep neural networks is known to depend critically on the identification of a good set of hyperparameters [34, 6]. In this project, a manual form of grid search was used to discover optimised hyperparameter configurations. In this technique, all hyperparameters are fixed and only a single hyperparameter is varied at a time. Although this may not be the most efficient approach, it enables a better understanding of the model to be gained.

Learning Rate

Of all the hyperparameters relevant to deep learning models, the optimisation of the learning rate often has the biggest impact on the performance of a model [4]. The selection of a learning rate too large can cause an optimisation algorithm to take a step "over" minima, causing the loss to inadvertently



(a) Ground truth boundaries (b) $\eta = 0.0001$ prediction (c) $\eta = 0.00005$ prediction

Figure 4.3: An example of a patch that benefits from the a learning rate η of 0.00005. The predictions shown have been skeletonized using the technique discussed in Chapter 3. The $\eta = 0.0001$ prediction achieved an accuracy of 92.1%, whereas the $\eta = 0.00005$ prediction achieved an accuracy of 95.5%. The 0.0001 prediction achieves a lower accuracy score due to the noise present in the inner section of the coral on the left side of the image. The 0.00005 prediction has more realistic predictions in this area of the image with all boundaries almost parallel to the growth surface of the skeleton.

increase rather than decrease. The selection of a learning rate too small can also hinder performance as the optimisation algorithm may become permanently stuck in a suboptimal local minimum [18].

Since it is usually not possible to calculate an optimal learning rate a priori [43], some form of trial and error is required. A reasonable range of values to experiment with are given by Bengio in [4] and will be used as the basis of the values experimented with in this section. Bengio suggests an initial learning rate within the range of 10^{-6} to 1.

The accuracies achieved when training the network using various initial learning rates in this range are shown in Figure 4.4a. It can be seen that the final average accuracy achieved after 20 epochs is similar for all but the 0.00001 value. Although it may look like the accuracy for this learning rate could still improve with further training, the accuracy does not improve further even after 40 epochs. This plateau in the accuracy achieved suggests that the 0.00001 learning rate may be low enough to become stuck in suboptimal local minima.

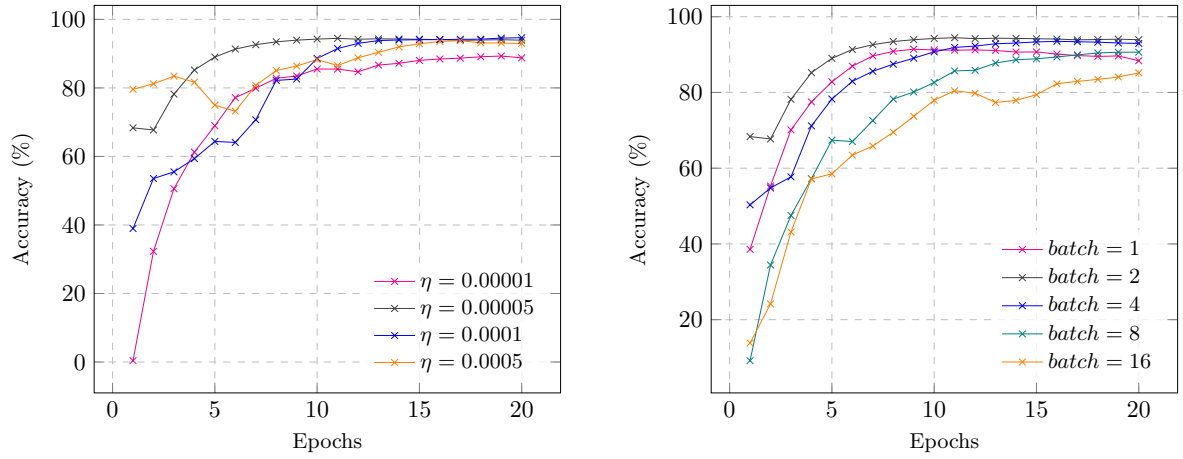
The training loss achieved by the 0.001 learning rate oscillated at $\sim$0.5 for the entire training process, even over multiple training attempts. In comparison, the 0.00005 learning rate achieved a training loss of 0.06 and a validation loss of 0.16. These oscillations are a common sign of a learning rate being too high and stepping "over" minima [7]. Pure black images were output for all samples in both the training and validation sets, resulting in an accuracy of 0% being achieved.

Although the 0.00005 and 0.0001 learning rates both achieved a similar accuracy of $\sim$95% after 20 epochs, the 0.00005 value was ultimately chosen for multiple reasons. First, the training and validation accuracy curves are smoother than the curves resulting from higher values, allowing the accuracy to be more reliably used as a stopping criteria for early stopping. Second, when inspecting the results visually, the 0.00005 learning rate results in predictions that are far clearer in the inner areas of the coral skeleton than any other learning rate. Although most of the learning rates were able to produce acceptable predictions nearer to the surface of the coral, only the 0.00005 learning rate was able to produce acceptable predictions in these inner areas. An example of a patch that benefited from the 0.00005 learning rate is shown in Figure 4.3.

Batch Size

The next parameter experimented with was the batch size. Ranging from a size of one up to a few hundreds in some applications [4], the batch size not only affects the performance achieved by a network, but can also affect training times significantly.

Small batches with sizes equal to powers of two were experimented with. The accuracies achieved when training using these various batch sizes are shown in Figure 4.4b. Looking at Figure 4.4b, it can already be seen that as the batch size increases, the accuracy achieved by the model decreases (with the exception



(a) Varying learning rate

(b) Varying batch size

Figure 4.4: The accuracies achieved on the validation set using various hyperparameter values. A TensorBoard smoothing value of 0.3 was used to improve visibility. **(a)** The accuracies achieved when training the network using various initial learning rate values η. Apart from varying the learning rate, the same hyperparameters used to achieve the initial results (outlined in Table 3.2) were used again here. **(b)** The accuracies achieved when training the network using various batch sizes. The hyperparameters outlined in Table 3.2 were used apart from a learning rate of 0.00005.

of a batch size of one). Batch sizes greater than 16 were also experimented with but the accuracy achieved only continued to decrease. It may seem that the accuracy achieved when $batch = 8$ could increase further given more training epochs. However, even after 40 epochs, the network trained with $batch = 8$ only achieved an accuracy of 91% compared to the accuracy of 95% achieved when $batch = 2$. Note that in order to ensure that the model was exposed to the same number of samples per epoch, with each increase in batch size, the `steps-per-epoch` argument of the `fit` method was also decreased. For example, if the batch size was doubled, the `steps-per-epoch` were halved. This is due to the `steps-per-epoch` argument specifying how many batches compose an epoch, rather than how many samples compose an epoch.

In practice, it has often been observed that larger batch sizes result in lower accuracy being achieved in deep learning models [27, 39, 58]. Keskar et al. propose that this decrease in performance may be attributed to larger batch sizes converging to "sharp" minima [27]. Whilst a "flatter" minimum can be described with low precision, a sharp minimum requires high precision. Keskar et al. cite the minimum description length (MDL) theory, which states that statistical models that require fewer bits to describe generalise better [44]. They argue that since flat minima can be specified with lower precision, they tend to have better generalisation performance. Note that the accuracy achieved on the validation set is a measure of how well the network generalises, since the network has not yet been trained on any slices present in the validation set.

Ultimately, a batch size of two was decided upon as the network trained with this size achieved the highest training and validation accuracy. Although a batch size of 16 provided a ∼1.4 times speed up in the training process, the model still trains in under 20 minutes when using a batch size of two[1]. For now, the improved performance provided by a smaller batch size outweighs the benefits provided by the training time saved when using larger sizes. If vast amounts of labelled data were available, a larger batch size would likely be necessary to keep training times reasonable.

Optimisation Algorithms

This section discusses the experimentation carried out with the stochastic gradient descent (SGD) optimisation algorithm. SGD was used successfully to train the original U-Net architecture to perform semantic segmentation on 2D greyscale biomedical data [45] and so is a reasonable alternative to the Adam optimiser used so far.

The Keras implementation of SGD using learning rates of 0.00005, 0.0001, 0.0005, 0.001, 0.005, and 0.01

[1] When trained using an Nvidia P100 "Pascal" GPU.

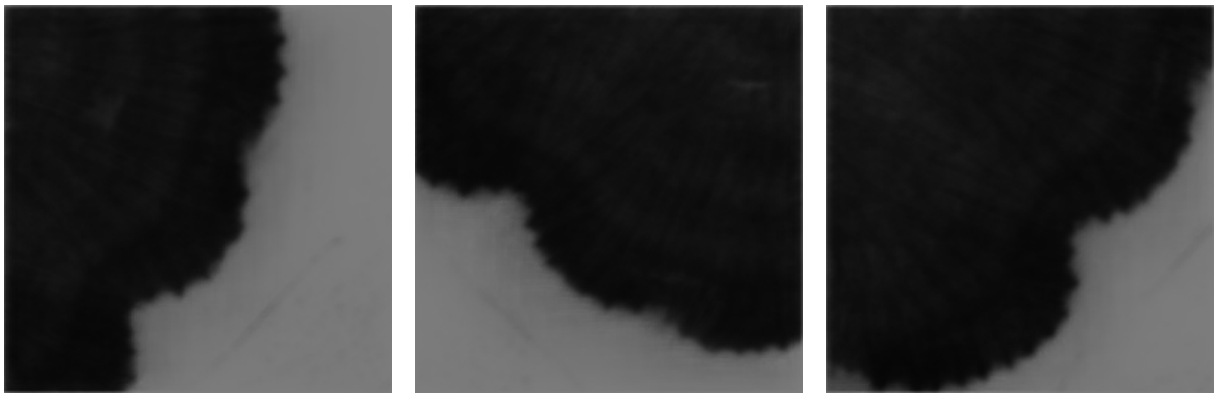


Figure 4.5: Example predictions output by the network when trained using the SGD optimisation algorithm using a learning rate of 0.0001. Predictions almost identical to these were output by all learning rates when a momentum value lower than 0.5 was used. It appears as though the input's channel has been inverted and the image has been blurred slightly.

were experimented with but in all cases no acceptable results were produced. When trained with each of these learning rates, the losses and accuracies for both the training and validation sets remained almost constant from the first epoch onwards and the produced predictions resembled the input images more than the corresponding labels. Examples of the predictions produced are shown in Figure 4.5.

In an attempt to improve the performance achieved by SGD, various levels of the Keras implementation of classical momentum [40] were experimented with. When momentum is enabled, the weight updates determined by SGD are calculated as a linear combination of the gradient and the previous updates made [53]. Momentum values of 0.1 to 0.9 in combination with all of the learning rates mentioned previously were experimented with but no acceptable results were produced once again. Although no acceptable results were produced, it is important to note that when using a momentum value above 0.5 in combination with a learning rate above 0.001, the validation loss was no longer constant after the first epoch, and a typical loss curve was produced. This suggests that the momentum implemented in Adam may play an important role in allowing the training loss to fall as the training process continues.

Although the SGD optimiser was utilised successfully in the original U-Net paper, the attempts to train this implementation of the architecture using SGD were not successful. Thus, the Adam optimiser continued to be used for the remainder of the experiments.

Loss Functions

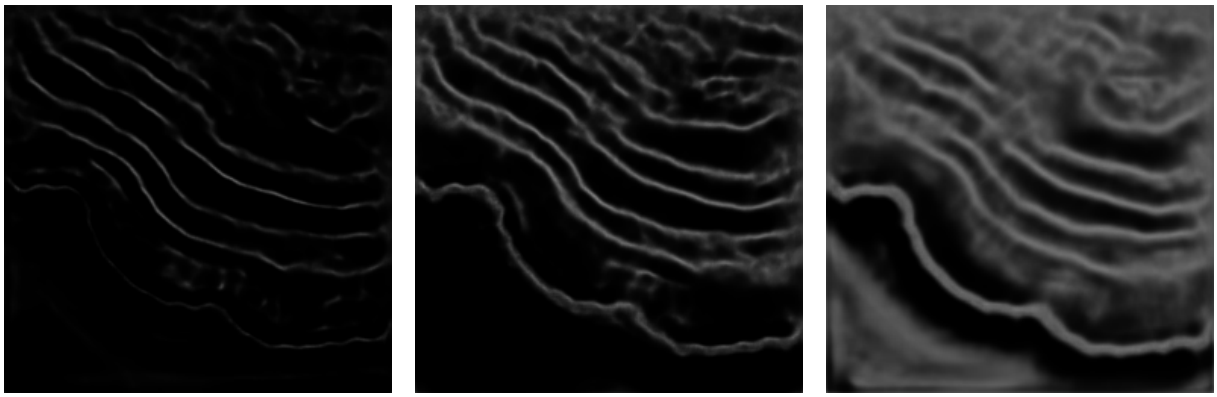
As outlined in Chapter 3, with ∼98% of an average labelled patch being black (part of the "not part of a boundary" class), the utilised labelling technique gives rise to a severe class imbalance. As a result, a model that learns to output a pure black image would achieve a low loss value for ∼98% of the image. Although this has proved to be a problem when assessing the accuracy of the model, the effects of this class imbalance on the network's ability to learn are not yet obvious. This section outlines the experiments carried out involving the focal loss function [35] designed specifically to address the class imbalance problem.

The focal loss introduces two tunable parameters: a weighting factor $\alpha \in [0, 1]$ used to increase the loss produced by the misclassification of minority class samples, and a "focusing" parameter $\gamma \geq 0$ used to reduce the contribution to the loss that "easy" examples have [35]. When the recommended default values of $\gamma = 2$ and $\alpha = 0.25$ were used, the network produced predictions comparable to those produced using cross-entropy both qualitatively and quantitatively with a validation accuracy of 93% being achieved. When varying the α and γ parameters, the best accuracy of 94% was achieved with $\alpha = 0.25$ and $\gamma = 1$.

When varying the γ parameter, an interesting phenomena was observed. Looking at Figure 4.6, it can be seen that an increase in γ results in more pixels being classified as the minority class—the "part of a boundary" class. Since the loss value produced per correctly classified black pixel is lower as γ increases, the contribution of an incorrectly classified white pixel to the overall loss is far higher. As a result, the network values the correct classification of a white pixel more than the correct classification of a black

Table 4.1: A summary of the training hyperparameters decided upon as a result of the various experiments run.

Hyperparameter	Setting
Architecture	2D U-Net
Optimisation Algorithm	Adam
Learning rate	0.00005
Loss function	Cross-entropy
Epochs	20
Steps per epoch	500
Batch size	2



(a) $\gamma = 0.1$ (b) $\gamma = 2$ (c) $\gamma = 5$

Figure 4.6: An illustration of how the focusing parameter γ of the focal loss function affects the predictions output by the network. Although Lin et al. suggest increasing α to 0.75 when γ is decreased to 0.1, α was set to a constant value of 0.25 for all values of γ to better illustrate how γ affects the performance of the network. It can be seen that as γ increases, the percentage of pixels that are predicted as the "part of a boundary" class also increases.

pixel, and so produces more white pixels overall.

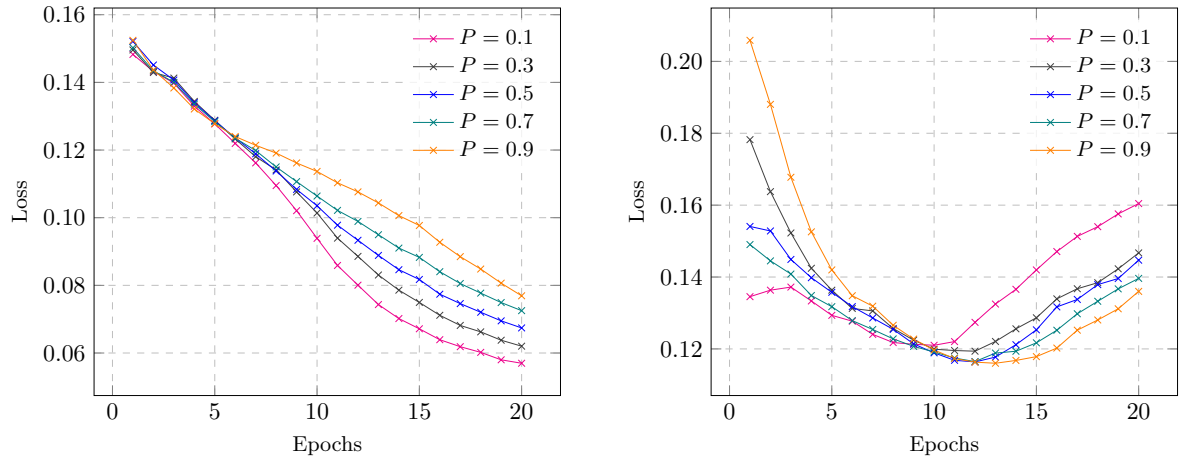
Since even the optimised parameterisation of the focal loss did not yield any improvements over the cross-entropy loss, the cross-entropy loss was ultimately decided upon. The hyperparameters used for the remainder of the experiments are summarised in Table 4.1.

Dropout Probability

Dropout is a regularization technique proposed by Srivastava et al. [51] in 2014. As mentioned in Chapter 3, although the original U-Net architecture does not mention any dropout layers, two dropout layers were placed in this implementation: one after the last contracting block and one after the bottleneck block. The experiments carried out in this section involve varying the dropout `rate` arguments that specify the probability that a neuron is dropped. In the initial implementation, these probabilities were set to 0.5.

The training and validation loss curves resulting from various dropout probabilities are shown in Figure 4.7. It can be seen that the dropout probability affects the training and validation losses in a similar manner to that discovered by Srivastava et al.; the higher the dropout probability, the higher the training loss and the lower the validation loss [51].

Similarly to the training loss, the training accuracy also improved as P decreased. However, the validation accuracy was not directly correlated with P; the validation accuracies achieved when P was equal to 0.1, 0.3, 0.5, 0.7, and 0.9, were 92%, 94%, 95%, 93%, and 91% respectively. Since the validation accuracy was highest when a dropout rate of 0.5 was used, this rate was decided upon.



(a) Training loss

(b) Validation loss

Figure 4.7: The loss achieved on the training and validation sets when training the network with varying dropout probabilities P. A TensorBoard smoothing value of 0.3 was used to improve visibility. It is important to note that although the validation losses begin to increase after ∼10 epochs for all dropout probabilities, the validation accuracies and qualitative performance actually continue to improve, and so early stopping before epoch 20 would result in a lower validation accuracy being achieved.

4.1.3 Augmentation

Due to the little amount of labelled training data available, augmentation proved to be an important regularization technique. The experimentation involving the transformations applied are discussed below. The network was again trained using the hyperparameters summarised in Table 4.1.

No Augmentation

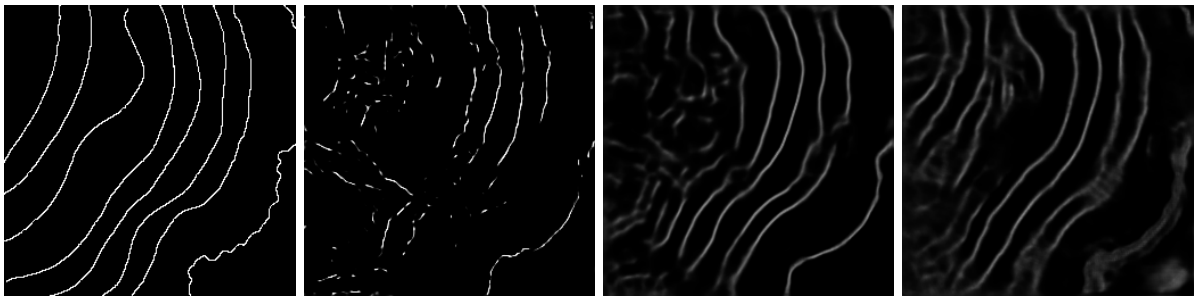
The first experiment involved removing the augmentation process altogether. The `flow_from_directory` method from the `ImageDataGenerator` class was still used to ensure that shuffling was still performed. The `steps_per_epoch` argument was also still set to 500 batches ensuring that the network was exposed to the same amount of samples throughout the training process. Since there were less than 500 training samples, some of these unaugmented samples were inevitably seen multiple times.

An example prediction output by the network when trained on unaugmented samples is shown in Figure 4.8b. Not only can it be seen that the performance is far worse qualitatively, but an average validation accuracy of 87% also makes evident the deterioration of the quantitative performance. The validation loss also reached a value of up to 0.4, far higher than the value of 0.15 reached when the network was trained using the initial transformation ranges. It is worth noting that a network trained on a larger dataset would not suffer as big of a deterioration in performance when no augmentation was used, as the dataset itself would already represent a much more comprehensive set of possible samples.

Augmentation Ranges

The ranges of acceptable augmentation transformations were also experimented with. The initial ranges outlined in Chapter 3 were scaled up and down, and the accuracies and losses achieved were monitored. A phenomena similar to that seen in Figure 4.7 (when the dropout probability was varied) was also observed in this case. The higher the ranges of acceptable transformations, the higher the training loss and the lower the validation loss achieved. The validation accuracy, however, was not directly correlated with the size of the transformation ranges. The highest validation accuracy was achieved using the initial ranges outlined in Chapter 3. Since the initial ranges were selected by hand to be as high as possible whilst also ensuring that the resulting images were "realistic", it is understandable that the best performance is achieved when using these ranges. Any higher ranges result in many of the artefacts outlined in Figure 3.6 being present. These artefacts result in augmented samples that no longer represent real coral data, and may be a contributing factor to the lower validation accuracy achieved.

Although higher transformation ranges did result in some valuable qualities being observed in the predictions, a trade-off with other valuable qualities was observed. For example, when looking at Figure 4.8,



(a) (b) (c) (d)

Figure 4.8: Example predictions produced by the network trained using various ranges of acceptable transformations. **(a)** The ground truth label. **(b)** A prediction output when no augmentation was used. The network misclassifies the majority of the boundaries, and the continuity within boundaries is often broken. It can also be seen that there are multiple wrongly classified boundaries that are perpendicular to the skeleton's surface. **(c)** A prediction output when the initial augmentation ranges outlined in Chapter 3 were used. **(d)** A prediction output when the initial ranges outlined in Chapter 3 were multiplied by five (e.g., rotations within a range of ± 10 degrees as opposed to ± 2 degrees). Although better predictions in the inner areas of the coral skeleton were produced, some degradation in the classification of the coral growth surface and the outermost density boundary is also observed.

it can be seen that although the highly augmented samples did produce better predictions in the inner areas of the coral skeleton, some degradation in the classification of the "easier" boundaries (such as the coral growth surface and the outermost density boundary) is also observed.

Ultimately, the initial ranges outlined in Chapter 3 continued to be used as the network trained with these ranges achieved the highest validation accuracy.

4.1.4 Ablation Studies

In the context of deep neural networks, the term "ablation study" is used to describe a procedure in which certain parts of the network are removed in order to gain a better understanding of the network's behaviour. This section discusses the various ablation studies performed.

Output Feature Channels Reduction

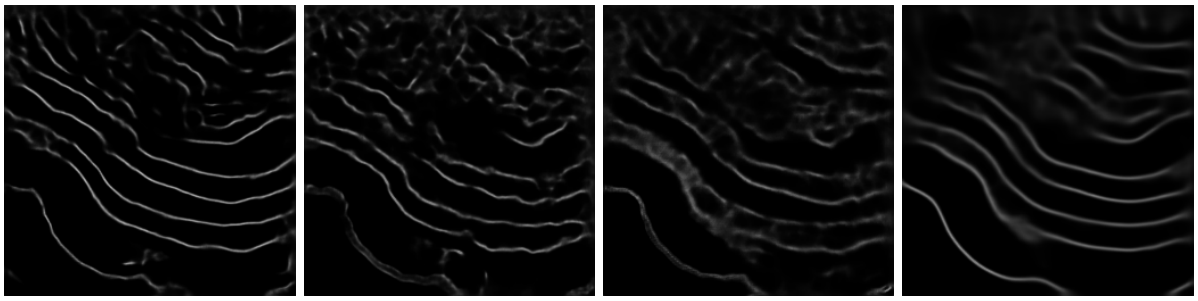
The first ablation study involved the removal of varying amounts of the feature channels output by each convolutional layer. When the number of output feature channels were halved for all convolutional layers in all blocks (e.g., the convolutional layer present in the first block now outputs 32 feature channels as opposed to the 64 output by the original architecture), the validation accuracy decreased by $\sim$2%. Despite this decrease only being minor, visual inspection of the resulting predictions showed a noticeable deterioration in their quality. As seen in Figure 4.9, the main qualitative issue observed was a lack of continuity within the predicted boundaries. Although the boundaries may have been predicted in the right place, many shorter boundaries were predicted rather than the desired long continuous boundaries. This small decrease in accuracy when there is a noticeable decrease in visual quality highlights an issue with the accuracy metric, where breaks in the continuity of boundaries are not punished sufficiently. This issue and other shortcomings of the accuracy metric are discussed further in Section 4.6.2.

Although reducing the output feature channels to half the original amount resulted in a $\sim$2.8 times speedup of the training process, the qualitative performance did deteriorate, so the full number of feature channels continued to be used.

Architecture Shortening

Another ablation study experimented with the removal of blocks throughout the architecture. In order to maintain the pass-forward of information, the removal of a contracting block also requires the removal of the corresponding expanding block, as this expanding block would otherwise have no contracting block from which information would be passed forward.

When the last contracting block and its corresponding expanding block were removed, the validation accuracy achieved decreased by $\sim$3% and the produced predictions were noticeably more noisy. This



(a) (b) (c) (d)

Figure 4.9: Example predictions produced by various ablated networks. **(a)** The prediction output by the unablated network for comparison. **(b)** A prediction output by the ablated network in which half of the convolutional layers' output feature channels are removed. It can be seen that the continuity within the bands is often broken. **(c)** A prediction output by the ablated network in which a contracting and expanding block were removed. The boundaries are noisy and blurred in parts. **(d)** A prediction output by the ablated network with no pass-forward of information. The produced boundaries are significantly smoother and the continuity of the boundaries is rarely broken.

shorter architecture does not contain as many layers and thus the outputs have not been processed by as many operations. The more layers that data has passed through, the "further" this data will be from the complex input images. Since the data has not been able to pass through as many layers in this case, the predictions have not been smoothed as much. An example of an output produced by this ablated architecture is shown in Figure 4.9.

Max-Pooling Removal

In order to assess the importance of downsampling in the performance achieved by this architecture, the max-pooling layers in the contracting path and corresponding up-sampling layers in the expanding path were removed. As a result, the feature channels throughout the architecture all have a resolution of 256×256. The ablated network did not perform well as no apparent boundaries were predicted. Blurry images vaguely resembling the input patches were produced and a validation accuracy of 0% was achieved. It is clear that the downsampling throughout the network is essential for the performance achieved. Since the convolutional layers all had to be applied to full size 256×256 images, training times increased by up to nine times and this ablated network was thus not a viable option anyway.

Pass-forward Removal

The final ablation study was used to determine the importance of the pass-forward of information from the contracting path to the expanding path that the U-Net architecture is known for. Other than the removal of all concatenation operations, the rest of the architecture was unchanged. With this pass-forward of information removed, the network performed surprisingly well. Although a decrease in the validation accuracy of ∼1% was observed, the predictions produced were qualitatively better in some respects. When looking at Figure 4.9 for example, it can be seen that the produced boundaries are significantly smoother and that the "continuity" of the boundaries is rarely broken. As discussed in Chapter 3, smooth boundaries are deemed as more realistic.

Smoother boundaries may be produced when there is no pass-forward of information from earlier in the network since the data must now pass through the entire network sequentially, and information from feature channels early in the architecture can not be used later on. The feature channels output by the blocks earlier in the architecture will be "closer" to the input image, since not as many operations have yet been performed on the data. These earlier feature maps contain images that have not yet been processed by many operations, and thus will not be as smooth as the feature maps output by the last block of the architecture which have had to pass through 26 layers. For example, the U-Net architecture with the pass-forward of information allows the last block of the network to make use of feature channels output by the first block, which have only been processed by two convolutional layers. Whereas, when the architecture does not pass-forward information, the last block can only make use of feature channels that have already passed through 24 layers.

Although smoother boundaries can be seen as a positive quality in this particular task, when perform-

Table 4.2: A summary of the performances achieved by the various ablated networks discussed throughout this section. Note that the unablated network took ∼16 minutes to train on an Nvidia P100 "Pascal" GPU.

Ablation	Trainable Parameters	Speedup	Validation Accuracy
None	31,031,685	×1.0	95%
Pass-forward Removal	27,898,245	×1.1	94%
Feature Channel Reduction	7,760,069	×2.8	93%
Architecture Shortening	7,697,285	×1.5	92%
Max-Pooling Removal	31,031,685	×0.1	0%

ing semantic segmentation with other data, a smoother segmentation is not necessarily more accurate. Note also that padding was used throughout this implementation of the architecture but was not used in the original implementation. This lack of padding in the original implementation by Ronneberger et al. significantly reduces the dimensionality of the bottleneck and could significantly affect the performance achieved when no pass-forward of information is present. Thus, although the lack of any pass-forward of information did not deteriorate the performance achieved in this task, the pass-forward may still be valuable when the architecture is implemented differently or utilised for different segmentation tasks.

The performances achieved by the various ablated networks are summarised in Table 4.2. Ultimately, due to the smoother continuous boundaries produced, the pass-forward of information was in fact removed and the resulting ablated network was used for the results shown in Section 4.4.

4.2 Alternative Three Dimensional Architectures

This section discusses the experiments carried out involving the structure of the three dimensional architecture implemented in Chapter 3.

4.2.1 Increased Convolutional Output Channels

As mentioned in Chapter 3, the number of feature channels output by the 3D convolutional layers were chosen in an attempt to replicate the original 2D U-Net architecture as closely as possible. Initially, the number of output channels was chosen to be eight times less than the number of output channels used in the 2D architecture (i.e., since the first block of the 2D architecture output 64 feature channels, only eight were output by the 3D architecture). The output feature channels must be reshaped in order to be used with 2D layers, resulting in the number of output channels being multiplied by nine. Thus, a factor of eight was the closest factor that divided evenly into the numbers of output feature channels used in the original paper (e.g., 64, 128, 256, etc.).

Whilst experimenting with the number of output feature channels, it was found that a higher number of feature channels (e.g., dividing by a factor of four rather than a factor of eight) produced significantly worse qualitative results and the validation accuracy dropped from ∼77% to ∼43%. The noticeably better performance achieved when using fewer output feature channels may be due to the reduction in the number of trainable parameters acting as a regularization technique.

4.2.2 Fully Three Dimensional Architecture

A fully three dimensional U-Net architecture was also experimented with. Similarly to the architectures discussed previously, this architecture also takes a chunk of nine patches as input. However, rather than only attempting to predict the boundaries present in the central slice, this architecture attempts to predict the boundaries present in all nine patches. Since the network is learning to label every input patch, the labels for these patches must also be provided.

Thus, in order to train the fully three dimensional architecture with the same amount of samples used previously, up to nine times as many slices would have to be labelled. This was not a realistic goal within the time frame of this project and only a small amount of adjacent slices were ultimately labelled. Two sets of nine slices that were all adjacent to each other were labelled, and the sliding window technique described in Chapter 3 was used once again to curate a small three dimensional dataset. Using the custom data loader detailed in Chapter 3, the three dimensional architecture was trained on various augmented

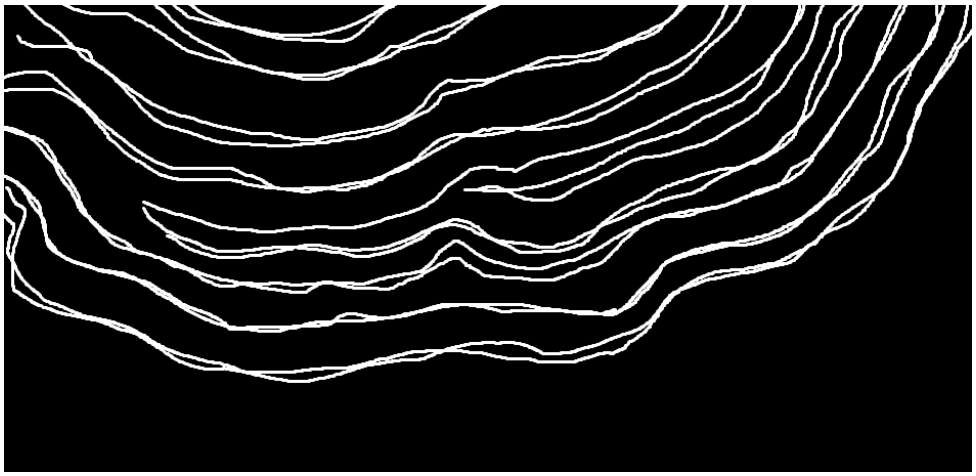


Figure 4.10: An example of two inconsistent adjacent labels. Lines corresponding to the same boundary in both slices can be up to eight pixels apart at some points. In reality, these boundaries should be no more than one pixel apart at any point.

versions of these three dimensional samples. Unfortunately, the architecture ultimately performed poorly, with the output predictions containing almost no visible boundaries.

Although this architecture performs poorly for various reasons, the lack of labelled three dimensional data may be the most significant. When looking at the training and validation loss curves, it can be seen that whilst the training loss continues to decrease for all 20 epochs, the validation loss begins to increase after only the second epoch. It appears as though the lack of data may be causing the network to severely overfit.

The quality of the labelled three dimensional data may be also be contributing to the poor performance achieved. As mentioned in Chapter 3, labelling the boundaries consistently is challenging. Looking at Figure 4.10, it can be seen that the adjacent slices may not have been labelled consistently. Although the two labels are clearly similar, it is important to note that the slices that these labels correspond to are near identical, since the two slices are adjacent and are each only $\sim$0.1 mm thick. The corresponding labels cannot realistically be more than one pixel apart from each other at any point. As the network was trained on these inconsistent nine-patch labels, it may have struggled to determine any relationships within the third dimension of the data. If the use of three dimensional architectures to extract these boundaries is revisited, consistent labelling of adjacent slices will be necessary if acceptable performance is to be achieved.

It is worth noting that although the hyperparameters outlined in Section 3.3 were initially used, multiple combinations of different learning rates, batch sizes, dropout rates, and kernel sizes were also experimented with, but none produced any visible boundaries. Ultimately, the original architecture whose implementation is discussed in Chapter 3 achieved the best performance of any of the three dimensional architectures experimented with.

4.3 Cross-validation

As outlined in Chapter 2, cross-validation techniques are used to give a comprehensive measure of a model's performance throughout the entire dataset, rather than just a particular subset. Cross-validation techniques reduce the effects of selection bias on the reported performance. Since the slices composing the test set and the slices composing the training set were extracted from different samples, some form of selection bias is inevitable and the performance achieved on the current test set does not necessarily represent the performance that would be achieved on the entire dataset.

4.3.1 Train/test Splits

In order to perform cross-validation, the dataset must be split into multiple training and testing sets known as "train/test" splits. Similarly to how the dataset was split in Chapter 3, the patches composing

Table 4.3: The accuracies achieved on the various cross-validation splits.

Test scan	Training samples	Test samples	Accuracy
RS0030	322	66	94.4%
RS0116	128	260	88.7%
RS0128	356	32	91.4%
RS0130	358	30	36.6%
Average			77.8%

the test sets must all be produced from slices of coral skeletons that are not part of the corresponding training sets. This ensures that in each train/test split, the network cannot be overfitting to the nature of the annual banding present in the skeletons used for testing. If this had been the case, the performance on each test set could have been positively skewed. For this reason, the train/test splits were manually selected, as opposed to randomly selected as they usually can be. It is important to note that no hyperparameters or augmentation settings were changed during the cross-validation process, as this could have been used to influence the final reported performance in some way.

Since all of the data was extracted from four coral skeletons, four splits were selected. In each split, the test set consists of every patch produced from one skeleton, and the training set consists of every patch produced from the rest of the available skeletons. The accuracies achieved for each train/test split are shown in Table 4.3.

4.4 Final Results

This section introduces and discusses the final results achieved by the ablated two dimensional architecture described in Section 4.1.4. The estimated calcification rates for each of the coral skeletons are also presented.

4.4.1 Final Boundary Extraction Results

Examples of higher and lower quality predictions produced by the final ablated two dimensional architecture are shown in Figures 4.11 and 4.12 respectively. The final cross-validated accuracy achieved was 77.8%.

Looking at Table 4.3, it can be seen that the split in which RS0030 is the test scan achieves the highest accuracy. This better performance achieved may be due to the validation set that was chosen to perform hyperparameter optimisation with. Since the slices composing the validation set were all extracted from the RS0030 scan, the hyperparameters were optimised in order to maximise the performance on this scan. However, it is also worth noting that the RS0030 scan contains arguably the most obvious annual banding, and so the better performance achieved may also be due to the RS0030 scan being inherently "easier". The banding present in this skeleton is more easily identified because the changes in density at the boundaries of two bands are often more abrupt—the density changes quickly over the space of only a few pixels. With other skeletons, however, the changes in density are often more gradual, and are thus harder to discern.

The slightly lower accuracy achieved on the RS0116 split may be due to the lack of training data available. Since the majority of the patches composing the curated dataset were extracted from this scan, only 128 training patches remained. Unfortunately, when classifying the RS0130 testing set, only the coral growth surfaces were ever successfully classified as a boundary. As a result, a significantly lower accuracy of 36.6% was achieved. Although it is unclear exactly as to why the network performed so poorly on this test set, it is worth noting that the annual banding present in the RS0130 scan was particularly hard to label manually. It was expected that the network would perform worse on this scan, nevertheless the lack of any boundaries being detected at all was a particularly poor result.

4.4.2 Final Calcification Rate Estimates

The density, linear extension rate, and calcification rate estimates made using the Python script described in Section 3.5 are listed in Table 4.4. The estimates made using both the ground truth boundaries and

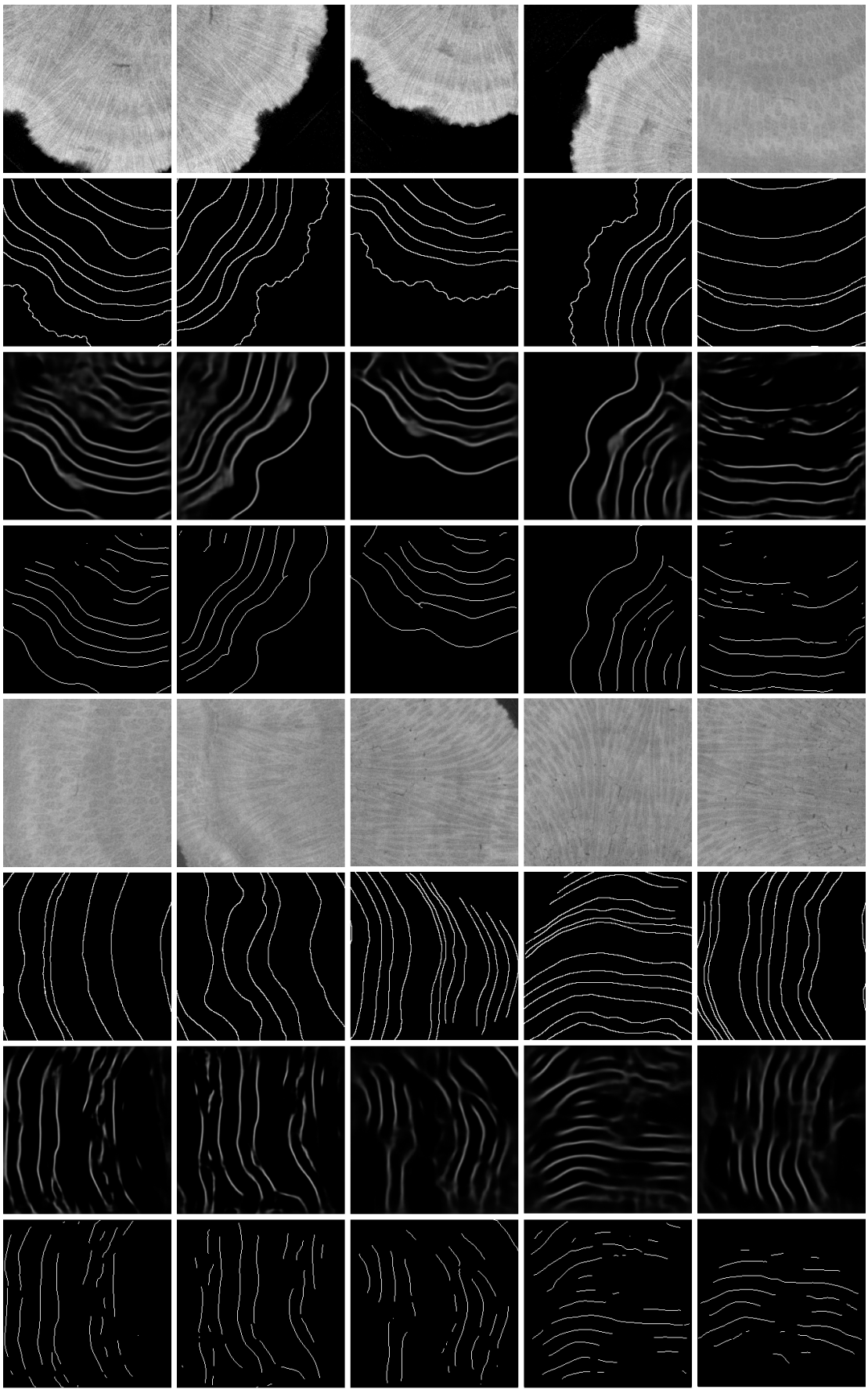


Figure 4.11: Higher quality examples of the final predictions produced by the network. From top to bottom: image, label, prediction, skeletonized prediction.

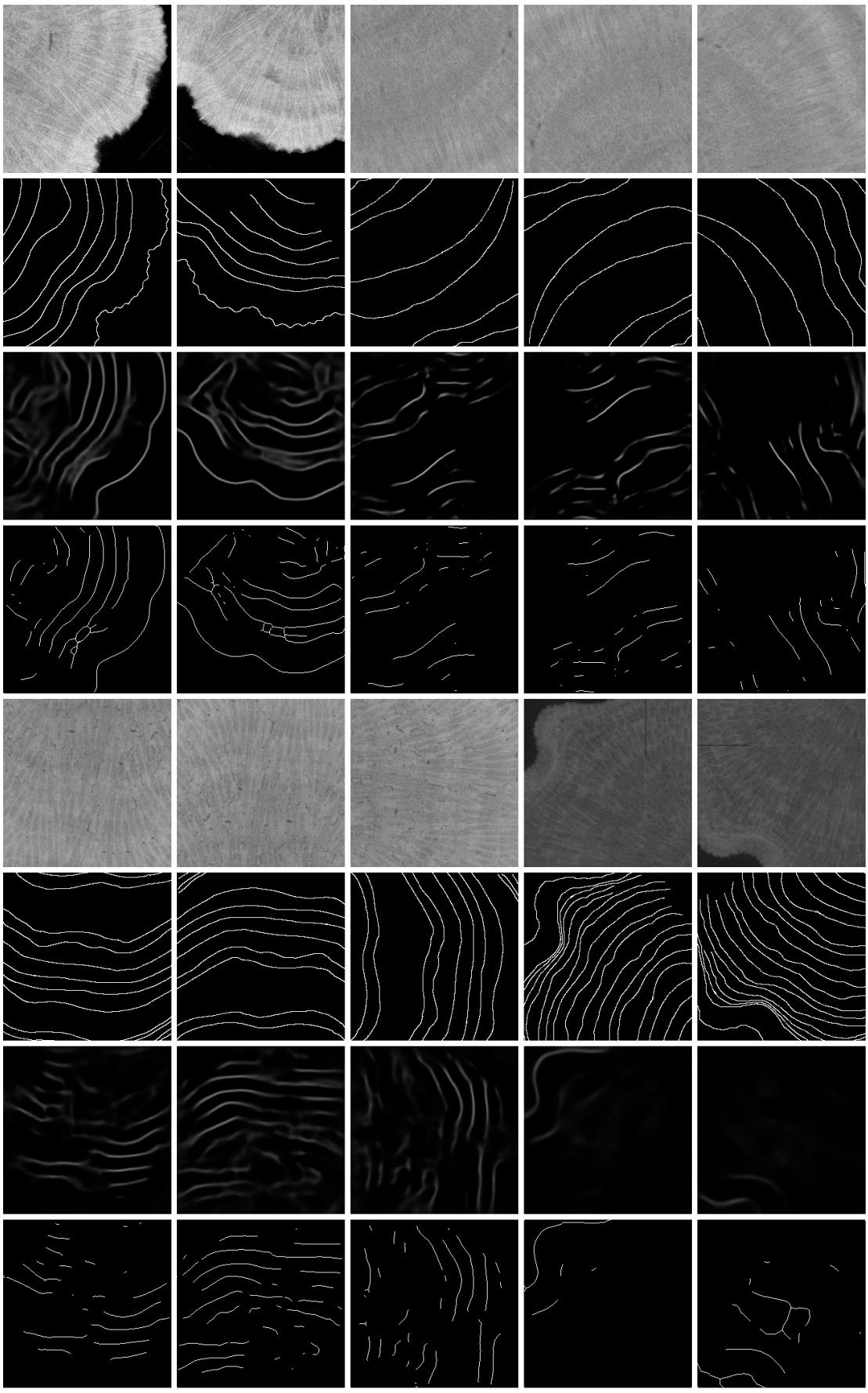


Figure 4.12: Lower quality examples of the final predictions produced by the network. From top to bottom: image, label, prediction, skeletonized prediction.

Table 4.4: The density, linear extension rate, and calcification rate estimates produced using the Python script described in Section 3.5. The estimates produced when using the ground truth boundaries are compared with the estimates produced when using the boundaries predicted by the final ablated two dimensional architecture. Note that the network did not produce any valid boundaries for the RS0130 scan so the corresponding estimates are left blank. The average linear extension rate and the average calcification rate estimated using the predicted boundaries are higher as a result.

	Density (g cm^{-3})	**Linear extension rate (mm y^{-1})**		**Calcification rate (g cm^{-2} y^{-1})**	
		Label	Prediction	Label	Prediction
RS0030	1.68	4.14	4.24	0.70	0.71
RS0116	1.52	5.26	5.15	0.80	0.79
RS0128	1.74	2.90	3.02	0.51	0.53
RS0130	0.94	2.08	–	0.20	–
Average	1.47	3.60	4.14	0.51	0.67

the predicted boundaries are shown.

These automatic estimates for Porites skeletons are in line with the manual estimates reported in the literature [56, 55]. The average calcification rate of 0.51 g cm^{-2} y^{-1} found using the components implemented in this project is also similar to the calcification rate manually estimated by researchers at the Natural History Museum[2].

4.5 Comparisons with Other Architectures

This section outlines the experimentation carried out involving various other architectures that can be utilised to perform semantic segmentation. Comparing the performances achieved by similar architectures can help one gain a better understanding of which aspects of the architectures affect performance the most.

4.5.1 SegNet

The first architecture experimented with was SegNet [2]. It was implemented using Keras, enabling it to be easily integrated into the code used in this project. The implementation is based heavily off of an implementation available on GitHub[3], which follows the architecture outlined in the original paper very closely. The learning rate and batch size were briefly experimented with, and in this case, a learning rate of 0.0001 and a batch size of two were found to produce the best results of any combination tested.

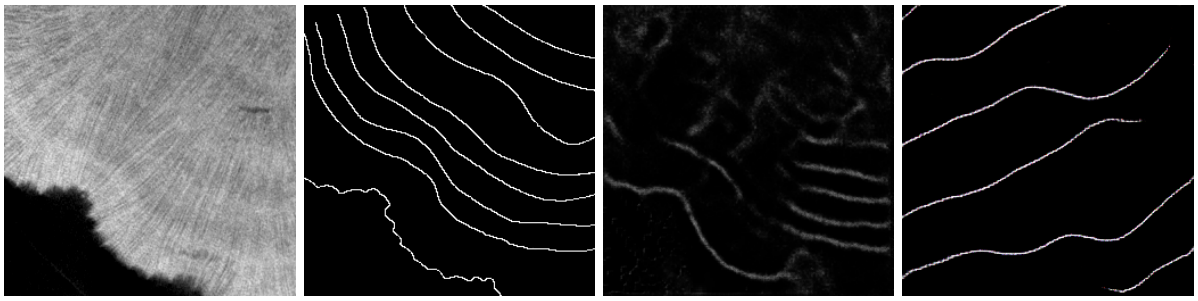
An example of a prediction produced by the SegNet implementation is shown in Figure 4.13c. It can be seen that although some banding was correctly identified in the bottom right of the prediction, the qualitative performance is poor overall. This is reflected by a validation accuracy of only 56% being achieved.

As mentioned in Chapter 2, the U-Net and SegNet architectures are remarkably similar overall. Both architectures follow some form of fully convolutional encoder-decoder structure and utilise some form of pass-forward of information from encoding blocks early in the architecture to decoding blocks later on. The similarity of the underlying fully convolutional encoder-decoder structures suggests that perhaps the difference in performance between the two architectures arises from the different methods of pass-forward used. Although the pass-forward of information is removed in the final architecture used in this project, the unablated U-Net architecture still performs significantly better than the SegNet architecture. The SegNet architecture passes information forward in the form of pooling indices. When this pass-forward of information was removed, the architecture performed better both qualitatively and quantitatively, with a validation accuracy of 75% now being achieved. This suggests that perhaps the passing forward of pooling indices is not as useful in this task as it is in other semantic segmentation tasks.

It is worth noting that the results discussed here may not be a true representation of the SegNet architecture's potential, as more time was spent optimising the hyperparameters used with U-Net architecture.

[2] According to values provided by Dr Kenneth Johnson, an average calcification rate of 0.48 g cm^{-2} y^{-1} was found on the samples that compose the dataset used to train the network.

[3] https://github.com/ykamikawa/tf-keras-SegNet



(a) (b) (c) (d)

Figure 4.13: Example predictions produced by the SegNet architecture and the pix2pix model. **(a)** The original coral skeleton patch. **(b)** The corresponding ground truth label. **(c)** A prediction output by the SegNet architecture. Although some banding is correctly identified in the bottom right of the prediction, the qualitative performance is poor overall. **(d)** A label generated by the pix2pix model. Although the generated boundaries are continuous, smooth, and sharp, they do not follow the same direction as the ground truth boundaries and are too far apart.

The hyperparameter settings used to achieve these results are summarised in Table C.1.

4.5.2 pix2pix

The pix2pix model [24] has shown some success whilst performing semantic segmentation and was an interesting model to experiment with. The experiments were carried out using a PyTorch[4] implementation provided by the original authors[5]. The generator was trained to generate boundary labels given some coral skeleton patches. The default hyperparameters specified by the authors were used and are summarised in Table C.2.

An example of a label generated is shown in Figure 4.13d. It appears as though the generator produces almost random predictions. Although at first glance these predictions look feasible, upon closer inspection it can be seen that the boundaries being predicted are completely wrong in many respects. In most of the labels produced, the predicted boundaries are not parallel to the growth surface and the spaces between them are significantly different than the spaces that exist between the ground truth labels. It is not obvious what data the generator is using to generate these labels, as no relationships between the patches and the generations can be found via visual inspection.

It is not clear as to why the pix2pix model performs so poorly at this task. Since the generator is only trained to maximise the probability that the discriminator makes a mistake, perhaps the discriminator is already not able to discern these fake labels from the real ones, and so the generator is not incentivised to generate labels that are any more realistic. Ultimately, the model achieved a validation accuracy of 37%. Note that the results discussed here may not be a true representation of the pix2pix model's potential, as more time could be spent optimising the hyperparameters used with the model and the performance achieved could improve.

4.6 Evaluation

4.6.1 Boundary Extraction

The majority of the project focused on the use of deep neural networks to extract the annual density band boundaries present in coral skeletons. This section aims to evaluate the decisions made throughout the project regarding this extraction process.

Labelling Method

It has been mentioned that the utilised labelling method gives rise to a severe class imbalance which introduces challenges when assessing performance. Although the goal of this project was to extract the boundaries between density bands, a labelling method in which the bands themselves were labelled might

[4] https://pytorch.org
[5] https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix

have been more a sensible choice. In this method, the possible classes would instead be "part of a high density band" and "part of a low density band". Since the high and low density bands are roughly equal in width, this labelling method would enable the number of pixels classified as a high band to be roughly equal to the number of pixels classified as a low band. As a result, the class imbalance within the labels would be significantly reduced and a standard per-pixel accuracy could be used.

Since the U-Net architecture is also capable of classifying more than two classes, the use of a third class to represent the area outside of a coral skeleton may have been useful. The current labelling method treats the growth surface of a skeleton as a density band boundary, even though this is technically not the case. The performance of the network in classifying banding boundaries may currently be hindered by the need to also classify the growth surface as a banding boundary, even though the nature of a growth surface is significantly different to the nature of an actual boundary.

Lack of Labelled Data

The experiments carried out in Section 4.1.3 involving data augmentation highlighted the effect that the lack of labelled data has on performance. An increase in the amount of labelled data would reduce the need for as many regularization techniques and would most likely increase the performance achieved. The lack of labelled data available is mostly due to the challenges involved in the manual labelling process. As mentioned in Chapter 3, upon visual inspection of over 50 Porites scans, only four scans contained slices that could be labelled confidently. Although more slices were labelled to be used by the three dimensional architecture, these slices were from the same scans and the labelling across adjacent slices was not consistent.

When this boundary extraction task is revisted, a larger amount of consistently labelled data will be key in improving the performance achieved.

Three Dimensional Architectures

Seeing as the two dimensional augmentation proved so important, it stands to reason that the performance achieved by an architecture making use of three dimensional data would also benefit significantly from appropriate augmentation. Although some forms of augmentation such as rotations, flips, and brightness shifts were implemented, the implementation of further transformations such as shifts, shearing, and zooming could have improved the performance achieved by the architectures that made use of a third dimension.

As mentioned in Section 4.2.2, the inconsistent labelling of the adjacent slices that compose the three dimensional data was a problem. In hindsight, perhaps each slice should have been labelled whilst constantly referencing the label of an adjacent slice, in order to ensure that the labelling is consistent in the third dimension. Although this labelling process would take appreciably more time per slice, it could potentially improve the performance achieved by both the two and three dimensional architectures significantly.

4.6.2 Accuracy Metric

The custom accuracy metric played a significant role in assessing the performance achieved by the various networks experimented with. Although the metric does solve many of the problems faced by a classic per-pixel accuracy metric, it is not without its issues.

The most obvious shortcoming highlighted in Section 4.1.4, is the fact that breaks in continuity are not punished enough. Breaks in continuity are indirectly punished to some extent, since a break results in the corresponding area of a ground truth boundary having no pixels close to it. However, the continuity of the boundaries is essential for the estimation of the calcification rate and should thus contribute to the accuracy achieved far more than it does currently.

Although the accuracy often reflects the qualitative performance well, there are examples in which visual inspection is still required to assess the performance of the network on this task. For example, although the predictions shown previously in Figure 4.9 vary noticeably in terms of quality, the accuracy achieved by all of the predictions were within $\sim$3% of each other.

It is important to note that despite its shortcomings, there is a still a clear correlation between the qualitative performance observed and the accuracy metric achieved. The custom accuracy metric is far

more useful than a standard per-pixel accuracy, and it has enabled useful quantitative comparisons to be made throughout the project.

Chapter 5

Conclusion

5.1 Contributions and Achievements

The main objective of this project was to automate the estimation of the calcification rate—the amount of skeletal matter produced annually by corals. This objective was broken down into multiple tasks and the majority of the project focused on the use of deep convolutional neural networks to extract the density band boundary positions. These positions were successfully extracted from two dimensional data with the network achieving a cross-validated accuracy of 77.8%, and predicting reasonable continuous boundaries for the majority of the coral samples experimented with. Through various ablation studies, a simplified version of the U-Net architecture was found, which both performed better and trained faster than the original U-Net. The network is packaged in the form of multiple Python scripts, which allow researchers to easily change hyperparameter values and train the model in under 30 minutes on a consumer grade GPU[1]. Appendix A contains instructions on how to train the networks implemented and on how to use the various Python scripts discussed throughout the project.

Although the attempts to extract the density band boundary positions present in three dimensional data were not successful due to the lack of data and inconsistent labelling, the insights into the three dimensional labelling process will enable researchers to label the data more accurately and potentially enable forms of three dimensional extraction in the future. Since it did not exist before, the implementation of a three dimensional data loader capable of online augmentation will also allow researchers to easily implement augmentation in their three dimensional Keras models.

The custom accuracy metric implemented is not without its issues, but is far more useful than a standard per-pixel accuracy in assessing the performance achieved in the boundary extraction task, and has enabled quantitative comparisons to be made throughout the project.

The tools implemented to estimate the density, linear extension rate, and calcification rate successfully produce estimates that are similar to the manual estimates reported in the literature. Once trained, the network is capable of extracting the boundaries present in an entire slice in under a minute, and the tools implemented are capable of estimating the statistics mentioned above in under ten seconds.

In summary, this project has highlighted the strengths and weaknesses of various ANN architectures whilst extracting the density band boundary positions, produced an optimised network capable of performing this task reliably, and provided researchers with a publicly available[2] semi-automated tool that is able to quickly and accurately estimate the density, linear extension rate, and calcification rate of coral samples.

[1] The final ablated U-Net architecture took ∼22 minutes to train on an Nvidia GTX 1070.
[2] https://github.com/ainsleyrutterford/DeepC

5.2 Future Work

5.2.1 Labelling Methods and Three Dimensional Architectures

As discussed in Chapter 4, the biggest limitation on the performance achieved appeared to be the lack of labelled data available. If provided with larger amounts of consistently labelled data, the architectures experimented with throughout could potentially achieve significantly better performance. The improved three dimensional labelling method outlined in Chapter 4 in which each is labelled whilst constantly referencing the label of an adjacent slice, may ensure that the labelling is consistent in the third dimension and could enable extraction of the boundaries present in three dimensions in the future.

The alternative labelling method outlined in Section 4.6.1 (in which the bands themselves are labelled as opposed to the boundaries between them) would also be an interesting avenue to pursue. Potentially solving the class imbalance faced and reducing the interference caused by the growth surfaces being classified as boundaries, this labelling method could improve the reliability of the boundary position predictions.

As mentioned in Chapter 3, due to the nature of the annual density banding, an architecture that could make use of a third dimension could perform significantly better. It appears as though the corallite structures present in the data may negatively affect the performance being achieved. Whilst corallite structures vary significantly in adjacent slices due to their small size (often less than a millimetre in diameter), the annual density banding is consistent across tens of scans. It seems likely that further research into the use of three dimensional architectures would yield improved results over those achieved by a two dimensional architecture.

5.2.2 Further Possible Experiments

Although various experiments including both the two and three dimensional architectures were carried out, it is worth noting that with more time and experiments, the performance achieved by these networks may still be improved. Since the lack of data currently remains an issue, experimentation with other regularization techniques such as spatial dropout [54], batch normalization [23], and L2 regularization could yield interesting results.

The original U-Net architecture was introduced almost five years ago [45], and various deep learning models have been proposed since that outperform U-Net in most semantic segmentation tasks [11, 59]. Experimentation with these alternative models may result in even better performance being achieved in the boundary extraction task tackled in this project.

Bibliography

[1] Inigo Alonso, Ana Cambra, Adolfo Munoz, Tali Treibitz, and Ana C Murillo. Coral-segmentation: Training dense labeling models with sparse ground truth. In *Proceedings of the IEEE International Conference on Computer Vision Workshops*, pages 2874–2882, 2017.

[2] Vijay Badrinarayanan, Alex Kendall, and Roberto Cipolla. Segnet: A deep convolutional encoder-decoder architecture for image segmentation. *IEEE transactions on pattern analysis and machine intelligence*, 39(12):2481–2495, 2017.

[3] Oscar Beijbom, Tali Treibitz, David I Kline, Gal Eyal, Adi Khen, Benjamin Neal, Yossi Loya, B Greg Mitchell, and David Kriegman. Improving automated annotation of benthic survey images using wide-band fluorescence. *Scientific reports*, 6:23166, 2016.

[4] Yoshua Bengio. Practical recommendations for gradient-based training of deep architectures. In *Neural networks: Tricks of the trade*, pages 437–478. Springer, 2012.

[5] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of machine learning research*, 13(Feb):281–305, 2012.

[6] James S Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*, pages 2546–2554, 2011.

[7] Christopher M Bishop et al. *Neural networks for pattern recognition*. Oxford university press, 1995.

[8] Robert W Buddemeier, James E Maragos, and David W Knutson. Radiographic studies of reef coral exoskeletons: rates and patterns of coral growth. *Journal of Experimental Marine Biology and Ecology*, 14(2):179–199, 1974.

[9] Augustin Cauchy. Méthode générale pour la résolution des systemes d'équations simultanées. *Comp. Rend. Sci. Paris*, 25(1847):536–538, 1847.

[10] Liang-Chieh Chen, George Papandreou, Florian Schroff, and Hartwig Adam. Rethinking atrous convolution for semantic image segmentation. *arXiv preprint arXiv:1706.05587*, 2017.

[11] Liang-Chieh Chen, Yukun Zhu, George Papandreou, Florian Schroff, and Hartwig Adam. Encoder-decoder with atrous separable convolution for semantic image segmentation. In *Proceedings of the European conference on computer vision (ECCV)*, pages 801–818, 2018.

[12] Özgün Çiçek, Ahmed Abdulkadir, Soeren S Lienkamp, Thomas Brox, and Olaf Ronneberger. 3d u-net: learning dense volumetric segmentation from sparse annotation. In *International conference on medical image computing and computer-assisted intervention*, pages 424–432. Springer, 2016.

[13] Michael Cogswell, Faruk Ahmed, Ross Girshick, Larry Zitnick, and Dhruv Batra. Reducing overfitting in deep networks by decorrelating representations. *arXiv preprint arXiv:1511.06068*, 2015.

[14] Antonia Creswell, Tom White, Vincent Dumoulin, Kai Arulkumaran, Biswa Sengupta, and Anil A Bharath. Generative adversarial networks: An overview. *IEEE Signal Processing Magazine*, 35(1):53–65, 2018.

[15] Andre Esteva, Alexandre Robicquet, Bharath Ramsundar, Volodymyr Kuleshov, Mark DePristo, Katherine Chou, Claire Cui, Greg Corrado, Sebastian Thrun, and Jeff Dean. A guide to deep learning in healthcare. *Nature medicine*, 25(1):24–29, 2019.

[16] Alberto Garcia-Garcia, Sergio Orts-Escolano, Sergiu Oprea, Victor Villena-Martinez, and Jose Garcia-Rodriguez. A review on deep learning techniques applied to semantic segmentation. *arXiv preprint arXiv:1704.06857*, 2017.

[17] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, 2019.

[18] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[19] Raymond C Highsmith. Coral growth rates and environmental control of density banding. *Journal of Experimental Marine Biology and Ecology*, 37(2):105–125, 1979.

[20] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

[21] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *science*, 313(5786):504–507, 2006.

[22] Julia Hirschberg and Christopher D Manning. Advances in natural language processing. *Science*, 349(6245):261–266, 2015.

[23] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[24] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A Efros. Image-to-image translation with conditional adversarial networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1125–1134, 2017.

[25] Alekseĭ Grigor'evich Ivakhnenko and Valentin Grigorévich Lapa. Cybernetic predicting devices, 1966.

[26] Justin M Johnson and Taghi M Khoshgoftaar. Survey on deep learning with class imbalance. *Journal of Big Data*, 6(1):27, 2019.

[27] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*, 2016.

[28] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[29] David W Knutson, Robert W Buddemeier, and Stephen V Smith. Coral chronometers: seasonal growth bands in reef corals. *Science*, 177(4045):270–272, 1972.

[30] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[31] Jan Kukačka, Vladimir Golkov, and Daniel Cremers. Regularization for deep learning: A taxonomy. *arXiv preprint arXiv:1710.10686*, 2017.

[32] Takio Kurita, Nobuyuki Otsu, and N Abdelmalek. Maximum likelihood thresholding based on population mixture models. *Pattern recognition*, 25(10):1231–1240, 1992.

[33] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[34] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *The Journal of Machine Learning Research*, 18(1):6765–6816, 2017.

[35] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection. In *Proceedings of the IEEE international conference on computer vision*, pages 2980–2988, 2017.

[36] Janice M Lough and Timothy F Cooper. New insights from coral growth band studies in an era of rapid environmental change. *Earth-Science Reviews*, 108(3-4):170–184, 2011.

[37] JM Lough and DJ Barnes. Intra-annual timing of density band formation of porites coral from the central great barrier reef. *Journal of Experimental Marine Biology and Ecology*, 135(1):35–57, 1990.

[38] James MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297. Oakland, CA, USA, 1967.

[39] Dominic Masters and Carlo Luschi. Revisiting small batch training for deep neural networks. *arXiv preprint arXiv:1804.07612*, 2018.

[40] Boris T Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17, 1964.

[41] Lutz Prechelt. Early stopping-but when? In *Neural Networks: Tricks of the trade*, pages 55–69. Springer, 1998.

[42] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

[43] Russell Reed and Robert J MarksII. *Neural smithing: supervised learning in feedforward artificial neural networks*. Mit Press, 1999.

[44] Jorma Rissanen. A universal prior for integers and estimation by minimum description length. *The Annals of statistics*, pages 416–431, 1983.

[45] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.

[46] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[47] David E Rumelhart, Geoffrey E Hinton, Ronald J Williams, et al. Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1, 1988.

[48] Shaeke Salman and Xiuwen Liu. Overfitting mechanism and avoidance in deep neural networks. *arXiv preprint arXiv:1901.06566*, 2019.

[49] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[50] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[51] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[52] Aljoscha Steffens, Antonio Campello, James Ravenscroft, Adrian Clark, and Hani Hagras. Deep segmentation: Using deep convolutional networks for coral reef pixel-wise parsing. *Volume*, 2380:9–12, 2019.

[53] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147, 2013.

[54] Jonathan Tompson, Ross Goroshin, Arjun Jain, Yann LeCun, and Christoph Bregler. Efficient object localization using convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 648–656, 2015.

[55] JJA Tortolero-Langarica, AP Rodríguez-Troncoso, JP Carricart-Ganivet, and AL Cupul-Magaña. Skeletal extension, density and calcification rates of massive free-living coral porites lobata dana, 1846. *Journal of experimental marine biology and ecology*, 478:68–76, 2016.

[56] José de Jesús A Tortolero-Langarica, Amílcar L Cupul-Magaña, Juan Carricart-Ganivet, Anderson B Mayfield, Alma P Rodríguez-Troncoso, et al. Differences in growth and calcification rates in the reef-building coral porites lobata: the implications of morphotype and gender on coral growth. *Frontiers in Marine Science*, 3:179, 2016.

[57] Athanasios Voulodimos, Nikolaos Doulamis, Anastasios Doulamis, and Eftychios Protopapadakis. Deep learning for computer vision: A brief review. *Computational intelligence and neuroscience*, 2018, 2018.

[58] Yang You, Igor Gitman, and Boris Ginsburg. Large batch training of convolutional networks. *arXiv preprint arXiv:1708.03888*, 2017.

[59] Yuhui Yuan, Xilin Chen, and Jingdong Wang. Object-contextual representations for semantic segmentation. *arXiv preprint arXiv:1909.11065*, 2019.

[60] TY Zhang and Ching Y. Suen. A fast parallel algorithm for thinning digital patterns. *Communications of the ACM*, 27(3):236–239, 1984.

Appendix A

Execution Instructions

A.1 Generating a Dataset

A.1.1 2D Datasets

In order to generate a 2D dataset, the provided `utils/sliding_window.py` script can be used. To see what command line arguments are available, run `python utils/sliding_window.py --help`.

The script can be used for each slice one at a time. The following arguments must be supplied: the slice file name, the top left and bottom right coordinates of the confidently labelled area, the window size, and the stride.

To generate patches from the `RS0030_yz_0625.tif` slice with the arguments specified in Table 3.1 for example, one would run:

```
python utils/sliding_window.py RS0030_yz_0625.tif 1028 153 1350 530 --stride 20
```

A.1.2 3D Datasets

In order to generate a 3D dataset, the provided `utils/sliding_window_3D.py` script can be used. To see what command line arguments are available, run `python utils/sliding_window_3D.py --help`.

The script can be used to generate patches from many adjacent slices at once. The number of images that compose a 3D sample can also be changed from the default of nine using the `--frames` argument. The following arguments must be supplied: the path of the directory containing the slices, the top left and bottom right coordinates of the confidently labelled area, the window size, and the stride.

To generate patches from many `RS0030_yz_XXXX.tif` slices with the arguments specified in Table 3.1 for example, one would run:

```
python utils/sliding_window_3D.py RS0030/ 1028 153 1350 530 --stride 20
```

if the `RS0030_yz_XXXX.tif` slices were stored in the `RS0030` directory.

A.2 Training a Network

In order to train the networks, the `train.py` script can simply be run. To see what command line arguments are available, run `python train.py --help`.

To train the ablated 2D U-Net architecture with a learning rate of 0.0001 and a batch size of four for example, one would run:

```
python train.py --ablated --lr 0.0001 --batch 4
```

To train the modified U-Net architecture using 3D data stored in the `data/3D` directory for example, one would run:

```
python train.py --model unet3D --dir data/3D
```

The model's learned parameters would be saved in a file named `checkpoint-20.hdf5` if the network was run for 20 epochs for example.

A.3 Assessing the Accuracy Achieved

The accuracy can be assessed using the `accuracy.py` script. To see what command line arguments are available, run `python accuracy.py --help`.

To assess the accuracy achieved by a standard 2D U-Net architecture trained over 30 epochs for example, one would run:

```
python accuracy.py --epochs 30
```

To assess the accuracy achieved by an ablated 2D U-Net architecture on a dataset in the `new_data/test` directory for example, one would run:

```
python accuracy.py --ablated --dir new_data/test
```

A.4 Estimating the Calcification Rate

In order to estimate the calcification rate of a given slice, the boundaries present in the slice must first be calculated using the `predict.py` script. To see what command line arguments are available, run `python predict.py --help`.

To predict the boundaries present in a slice named `slice.png` for example, one would run:

```
python predict.py --image slice.png
```

The image containing the skeletonized boundary positions will be saved in the `out.png` file. Next, the `utils/calcification.ipynb` script can be used. Open this script using a jupyter notebook[1] environment or something similar, and run the contained cells. Each cell will walk the user through the steps taken to estimate the density, linear extension rate, and calcification rate of the slice, and the final estimates will be printed by the last cell.

Working coordinates and density calibration values of the slices used in this project are provided in the script. If a user would like to estimate values for new slices, a `Slice` object must be defined with the following arguments: the slice image file name, the two sets of coordinates, and the density calibration values output by the CT machine.

In the future, this `.ipynb` script could easily be converted to a pure Python script and new slice information could be provided through the command line instead.

[1] https://jupyter.org

Appendix B

Implementation Listings

```
from keras.preprocessing.image import ImageDataGenerator
import random

# Specify the transformations allowed and store them in a dictionary
aug = dict(rotation_range=2,
           width_shift_range=0.02,
           height_shift_range=0.02,
           shear_range=2,
           zoom_range=0.02,
           brightness_range=[0.9,1.1],
           horizontal_flip=True,
           vertical_flip=True,
           fill_mode="nearest")

# Pass the contents of the dictionary as arguments to the ImageDataGenerator
# constructors
image_datagen = ImageDataGenerator(**aug)
label_datagen = ImageDataGenerator(**aug)

# Generate a random seed to be used for both the image and label generators
seed = random.randint(0, 100)

# Create the generators using the flow_from_directory method. The same seed
# value is passed to both methods ensuring the same transformations are applied.
image_generator = image_datagen.flow_from_directory(image_path, seed=seed, ...)
label_generator = label_datagen.flow_from_directory(label_path, seed=seed, ...)

# Zip the generators into one generator that yields image-label pairs
train_generator = zip(image_generator, label_generator)
```

Listing B.1: A simplified example of online augmentation implemented using the Keras `ImageDataGenerator` class.

```
from keras.callbacks import TensorBoard, ModelCheckpoint, EarlyStopping
from keras.models import Model
from keras.optimizers import Adam

# Create an instance of the Model class using the Model constructor
model = Model(inputs=..., outputs=...)

# Compile the model using the compile method
model.compile(optimizer=Adam(lr=1e-4),
              loss="binary_crossentropy",
              metrics=["accuracy"])

# Create a TensorBoard callback
tb = TensorBoard(log_dir="logs/")

# Create an EarlyStopping callback
es = EarlyStopping(monitor="val_loss", mode="min", patience=5)

# Create a ModelCheckpoint callback
mc = ModelCheckpoint(filepath="checkpoint-{epoch:02d}.hdf5",
                     monitor="loss",
                     save_best_only=False)

# Train the model using data provided by the train_generator defined earlier
model.fit_generator(train_generator,
                    steps_per_epoch=500,
                    epochs=20,
                    validation_data=val_generator,
                    validation_steps=10,
                    callbacks=[tb, mc, es])
```

Listing B.2: A simple example of compiling a Keras model using the `TensorBoard`, `EarlyStopping`, and `ModelCheckpoint` callbacks. The Keras implementations of the Adam optimiser, binary cross-entropy loss, and per-pixel accuracy are used. Once the model is compiled, it is then trained for 20 epochs using the `fit_generator` method.

```python
from generator import ImageDataGenerator3D, LabelDataGenerator2D
import random

# Specify the transformations allowed and store them in a dictionary
aug = dict(rotation_range=2,
           width_shift_range=0.02,
           height_shift_range=0.02,
           shear_range=2,
           zoom_range=0.02,
           brightness_range=[0.9,1.1],
           horizontal_flip=True,
           vertical_flip=True,
           fill_mode="nearest")

# Pass the contents of the dictionary as arguments to the ImageDataGenerator3D
# and LabelDataGenerator2D constructors
image_datagen = ImageDataGenerator3D(**aug)
label_datagen = LabelDataGenerator2D(**aug)

# Generate a random seed to be used for both the image and label generators
seed = random.randint(0, 100)

# Create the generators using the flow_from_directory method. The same seed
# value is passed to both methods ensuring the same transformations are applied.
image_generator = image_datagen.flow_from_directory(image_path, seed=seed, ...)
label_generator = label_datagen.flow_from_directory(label_path, seed=seed, ...)

# Zip the generators into one generator that yields image-label pairs
train_generator = zip(image_generator, label_generator)
```

Listing B.3: A simplified example of online 3D augmentation implemented using the `ImageDataGenerator3D` and `LabelDataGenerator2D` classes. Note the similarities with the 2D augmentation shown in Listing B.1.

Appendix C

Hyperparameter Tables

Table C.1: A summary of the hyperparameter settings used when training the SegNet architecture.

Hyperparameter	Setting
Architecture	2D SegNet
Optimisation Algorithm	Adam
Learning rate	0.0001
Loss function	Cross-entropy
Epochs	20
Steps per epoch	500
Batch size	2

Table C.2: A summary of the hyperparameter settings used when training the pix2pix model.

Hyperparameter	Setting
Generator Architecture	U-Net 256
Discriminator Architecture	Basic
Optimisation Algorithm	Adam
Learning rate	0.0002
Epochs	100
Batch size	1
Dropout	Enabled