Deep Learning with Principal Component Wavelet Networks

CS39440 Major Project Report

Author: Samuel Spurling (sas90@aber.ac.uk)

Supervisor: Dr Bernie Tiddeman. Project Supervisor (bpt@aber.ac.uk)

29th March 2022

Version 1.0 (Release)

This report is submitted as partial fulfilment of a BSc degree in
Computer Science and Artificial Intelligence (With Integrated Year in Industry) (GG47)

Department of Computer Science
Aberystwyth University
Aberystwyth
Ceredigion
SY23 3DB
Wales, UK

Declaration of originality

I confirm that:

- This submission is my own work, except where clearly indicated.
- I understand that there are severe penalties for Unacceptable Academic Practice, which can lead to loss of marks or even the withholding of a degree.
- I have read the regulations on Unacceptable Academic Practice from the University's Academic Registry (AR) and the relevant sections of the current Student Handbook of the Department of Computer Science.
- In submitting this work, I understand and agree to abide by the University's regulations governing these issues.

Name ……Samuel Spurling……

Date ……30/04/2022……

Consent to share this work

By including my name below, I hereby agree to this project's report and technical work being made available to other students and academic staff of the Aberystwyth Computer Science Department.

Name ……Samuel Spurling……

Date ……30/04/2022……

Acknowledgements

I am grateful to my project supervisor for his excellent guidance and advice, my flatmate for her constant support and encouragement, and my personal tutor for being a benevolent and reliable presence in my time at university.

Abstract

In this report we propose a novel application for Principal Component Wavelet Networks (PCWN) [1]. The PCWN decomposes an image using handcrafted invertible 2D wavelet filters-banks and 1x1 filters learnt through principal component analysis to control the size of decompositions. The reconstruction is accomplished using the inverted filter bank and an approximately inverted PCA. Previously the approach has shown competitive results for linear inverse problems in computer vision such as compressive sensing, super-resolution, and in-painting. The proposed novel application is image segmentation. This is a task where there are objects in an image and through segmentation the image is split into distinct components, or segments, where each segment represents an object in the image.

We use 2 variants of the PCWN, a fully convolutional and a fully connected approach. In experiments, there will be a comparison of the PCWN approach to the U-net model [2], a standard architecture for image segmentation, to benchmark the PCWN models performance. The results of the experiments demonstrate that PCWN has great potential when applied to the task of image segmentation. Possible future work looks at using PCWN as regularisation in the U-net architecture and replacing PCA with a different kind of invertible dimensionality reduction such as Kernel PCA or UMAP [3].

Figure 1: Image segmentation: Difference between semantic and instance [15] 11
Figure 2: Illustration of tanh and atanh 11
Figure 3: Wavelet decomposition of an image of a bird 12
Figure 4: PCA training process [1] 13
Figure 5: U-net architecture [2] 14
Figure 6: Example of high-resolution features being used to localize low-resolution features [2] 15
Figure 7: Atrous convolution: Atrous rate = 1 15
Figure 8: Atrous convolution: Atrous rate = 2 15
Figure 9: Image of a dog broken up into patches 16
Figure 10: screenshot of the Kanban board used for tracking tasks, using click-up https://app.clickup.com/ 17
Figure 11: Visualisation of IoU [30] 21
Figure 12: Fully connected PCWN architecture 24
Figure 13: Fully convolutional PCWN architecture 25
Figure 14: Original image of bird 26
Figure 15: Image of bird with mask applied 26
Figure 16: Inverted segmentation mask 26
Figure 17: Original segmentation mask 26
Figure 18: Proposed software architecture 27
Figure 19: Graph of training data size vs mean dice score 30
Figure 20: Graph comparing model performance on training and test data for training data size experiment .. 30
Figure 21: Graph of layer count vs mean dice score 31
Figure 22: Graph comparing model performance on training and test data for layer count experiment 31
Figure 23: Graph of keep percent vs mean dice score 32
Figure 24: Graph comparing model performance on training and test data for keep percent experiment 32
Figure 25: Graph of image resolution vs mean dice score 33
Figure 26: Graph comparing model performance on training and test data for image resolution experiment .. 33
Figure 27: Example output of PCWN on test data, dice score = 0.3 34
Figure 28: Example output of PCWN on train data, dice score = 0.2 34
Figure 29: Graph of training data size vs mean dice score 35
Figure 30: Graph comparing model performance on training and test data for training data size experiment .. 35
Figure 31: Graph of layer count vs mean dice score 36
Figure 32: Graph comparing model performance on training and test data for layer count experiment 36
Figure 33: Graph of keep percent vs mean dice score 37
Figure 34: Graph comparing model performance on training and test data for keep percent experiment 37
Figure 35: Graph of image resolution vs mean dice score 38
Figure 36: Graph comparing model performance on training and test data for image resolution experiment .. 38
Figure 37: Example output of PCWN on test data, dice score = 0.72 39
Figure 38: Example output of PCWN on train data, dice score = 0.81 39
Figure 39: Graph comparing model performance on training and test data for U-net model 40
Figure 40: Graph comparing model performance on training and test data for CNN model 40
Figure 41: Example CNN output on test data, dice score = 0.90 41
Figure 42: Example U-net output on test data, dice score = 0.89 41
Figure 43: Box plot showing every models performance on test and train data from bird dataset 42
Figure 44: Box plot showing every models performance on test and train data from pet dataset 42

Table 1: Table of features 17
Table 2: An experiment on the effect of training size on convolutional PCWN 29
Table 3: An experiment on the effect of layer count on convolutional PCWN 30
Table 4: An experiment on the effect of keep percent on convolutional PCWN 31
Table 5: An experiment on the effect of image resolution on convolutional PCWN 32
Table 6: An experiment to show the effect of training data size on connected PCWN 34
Table 7: An experiment to show the effect of layer count on connected PCWN 35
Table 8: An experiment to show the effect of keep percent on connected PCWN 36
Table 9: An experiment to show the effect of image resolution on connected PCWN 37
Table 10: Table comparing PCWN and deep learning approaches on test datasets 41

Contents

1 BACKGROUND, ANALYSIS & PROCESS 8

1.1 Background 8

1.2 Analysis 8
1.2.1 Fundamentals 8
1.2.2 PCWN 11
1.2.3 Deep Learning Architectures for Segmentation 14
1.2.4 Research Questions 16

1.3 Process 16
1.3.1 Overview 16
1.3.2 Iteration 0 - Setup 18
1.3.3 Iteration 1 18
1.3.4 Iteration 2 18
1.3.5 Iteration 3 19
1.3.6 Iteration 4 19

2 EXPERIMENT METHODS 20

2.1 Hypotheses 20

2.2 Hardware 20

2.3 Datasets 20

2.4 Metrics and Measurements 20
2.4.1 Dice Coefficient and Intersection Over Union (IoU) 20
2.4.2 Other Metrics 21

2.5 Measuring Performance 21
2.5.1 Repeated Experiments 21
2.5.2 Standard Deviation and Mean 21
2.5.3 Graphs 22

3 SOFTWARE DESIGN AND IMPLEMENTATION 23

3.1 Design 23
3.1.1 PCWN and Linear Least Squares 23
3.1.2 Deep Learning Methods 26
3.1.3 Overall Architecture 27

3.2 Implementation 28
3.2.1 Language 28
3.2.2 Development Software 28

4 RESULTS, TESTING AND CONCLUSIONS 28

4.1 Testing of system 28
4.1.1 Overall Approach to Testing 28

4.2 Issues with Experiments 29
4.2.1 Memory Errors 29

4.3 Determining best parameters for PCWN segmentation method 29
4.3.1 Convolutional Network Approach 29
4.3.2 Connected Network Approach 34

4.4 Training Deep learning networks 40

4.5 Experimental Results: PCWN vs Deep Learning 41

4.6 Conclusion 42

5 CRITICAL EVALUATION 43

5.1 Background and Research 43

5.2 Design 43

5.3 Implementation 43

5.4 Testing 44

5.5 Methodology 44

5.6 Time Management 45

5.7 Report 45

5.8 Reflection and Future Work 45

6 BIBLIOGRAPHY 46

7 APPENDICES 48

A. Third-Party Code and Libraries 48

B. Code Samples 49

1 Background, Analysis & Process

1.1 Background

The motivation for this project stems from an interest in computer vision, dimensionality reduction techniques, and a curiosity for wavelets. Image segmentation is an active field of research, with applications in medical imaging, satellite imagery and image-based search.

The project extends the Principal Component Wavelet Network (PCWN) method [1] to the task of image segmentation, so naturally PCWN was a key part of the background research. The motivation for this was to understand the novel approach to linear inverse problems and apply my findings to an image segmentation approach. Convolutional scattering wavelet networks [4] is another method which shares similarities to PCWN, especially on methods for mitigating the issue of large channels after multiple applications of wavelet filter banks. To understand the segmentation task, there was research of U-net [2], DeepLabV3 [5] and a comparison of segmentation methods [6].

1.2 Analysis

1.2.1 Fundamentals

1.2.1.1 Dimensionality Reduction: Principal Component Analysis and other Techniques

Principal Component Analysis (PCA) is a method of dimensionality reduction that calculates "principal components" and then orients the data using the principal components as basis vectors. The advantage of using PCA is that high dimensional data can be expressed in lower dimensions whilst retaining almost all the variance between data points. Consequently, the curse of dimensionality [7] can be avoided when using PCA in machine learning, where higher dimensions cause an increase in sparsity, causing data objects to appear dissimilar which impacts the algorithms' ability to learn.

Given an input matrix of data A, this is how to calculate the principal components. First the covariance matrix and the mean of the data is calculated. The mean is subtracted from the data to standardise the data.

$$\text{cov} = \sum_{A = A - \bar{x}} x_i^t x_i, \qquad \bar{x} = \frac{\sum x_i}{n}$$

The principal components are found by calculating the eigenvectors and eigenvalues of the covariance matrix, and then the eigenvectors are ordered using the eigenvalues. A n-dimensional dataset has n principal components. The sum of the eigenvalues of n principal components is the total variance, therefore we can calculate explained variance by dividing the eigenvalue of a component by the sum of the eigenvalues. The total explained variance is found by summing the explained variance for each component selected. The following is an example of eigenvectors v and eigenvalues λ.

$$v_1 = [0.5, 0.1],\ \lambda_1 = 1.5$$
$$v_2 = [0.2, 0.6],\ \lambda_2 = 0.8$$

The eigenvectors of the covariance matrix are the direction of the axes which have the highest variance, and the eigenvalues are the amount of variance in the respective components. The feature vector is formed using the eigenvectors like so.

$$B = \begin{bmatrix} 0.5 & 0.1 \\ 0.2 & 0.6 \end{bmatrix}$$

We are applying PCA to the feature maps produced by the filter banks. Finding the dot product of the feature maps (A) and the transpose of the feature vector (B) produces a dimensionally reduced feature map (C) that retains most of the information of the higher dimension feature map.

$$C = A \cdot B^T$$

PCA is an excellent technique to use on numerical and linearly separable data. For data where these conditions are not met, there are two dimensionality reduction approaches which can be used. The first is kernel PCA, which uses a function to map the data to a higher dimension before applying PCA. The reason for mapping the data to a higher dimension is that it may not be linearly separable in the original space but may be linearly separable in the projected space. If the reader is familiar with Support Vector Machines (SVM), the principle is very similar. Common kernels include polynomial and radial basis function (RBF). The second technique is Uniform Manifold Approximation and Projection (UMAP) [3]. This fascinating technique has roots in topology and has superior performance to T-Distributes Stochastic neighbourhood Embedding (TSNE) in many applications where preserving global structure is important. UMAP uses two high-level steps, approximating the manifold on which the data lies, and then approximately reconstructing that manifold using fuzzy simplicial steps. Further elaboration is beyond the scope of this report, however the original UMAP paper is incredibly interesting and worth reading.

1.2.1.2 Convolutional Neural Networks

An artificial neural network (ANN) is a directed and weighted graph where neurons are nodes and neuron connections (weights) are edges. ANNs are composed of layers, where every layer is a collection of neurons. The most common architecture is feed-forward. In this architecture every layer is connected to another layer, except the output layer, which only has a incoming connections. ANNs typically consist of three types of layers, input, output and hidden. The hidden layers of an ANN allow them to function as universal function approximators. That is to say, with the correct architecture and parameters an ANN can model any function to a small degree of error. Finding the correct architecture and parameters is a complex task with no general solution, and there are no guarantees of optimality. Typically, ANN's are trained using vast amounts of data and an error update algorithm that tweaks the parameters of the network towards a more optimal solution. Normally this algorithm is backpropagation [8].

There are many architectures other than feed-forward networks, such as Recurrent Neural Networks (RNNs), transformers, Self-Organising Maps (SOMs) and autoencoders. The architecture used for this project is Convolutional neural networks (CNNs).

CNNs are a commonly used architecture in computer vision. Popularised by Kunihiko Fukushima [9] and Yann LeCun [10] between 1980-1990, CNNs have a number of desirable qualities for computer vision tasks. CNN's work using convolutional filters, which are 2D matrices, or kernels, that slide over the image and "convolve" with patches on the image. A convolution measures the strength of a signal at a particular patch on the image. Multiple kernels can be used in parallel to generate deep feature maps, and the kernels can have any size. The size of the steps taken when sliding the kernel over an image is called the stride. A stride of 2 means that a pixel is skipped with every step. Below is the equation used to calculate the value of a 1D feature map, y, at location [m, n]. The variable x is the image we are convoluting, and h is the feature map. The $*$ operator is used as shorthand for a convolution operation.

$$y[m] = \sum_i x[i] \cdot h[m-i] = x[m] * h[m]$$

The kernel values are learnt through error update, the same as any other neural network. Typically, they are initialised randomly, although there are several handcrafted filters which can be used to initialise the kernels. For example, there are filters for edge detection and gaussian blurring.

The main advantage of using convolutions instead of a fully connected solution is the massive reduction in connections in the network relative to a standard densely connected network. In a feed-forward network, the number of connections scales exponentially with the resolution of the image. Every one of these connections requires an error update when training, which introduces a massive

computational overhead. The network will also struggle to generalise, resulting in overfitting. A CNN uses convolutions and pooling to minimise the connections between layers and lead to more efficient computation. Pooling is an operation which further reduces the size of feature maps. Max-pooling, for example, takes the largest value in a small window, discarding the other values.

1.2.1.3 Wavelets

Appropriate image representation is key to the success of an algorithm. There exist several different representations. The most familiar representation is the spatial domain. The spatial domain is how we view digital images, with resolution, colour channels and pixel intensity values. Other representations include the Fourier domain and wavelets.

Wavelets transforms are similar to Fourier transforms, except that whereas Fourier can only capture frequency information, wavelets can capture frequency and location information. Both Fourier and wavelet transforms measure signal strength. For Fourier transforms, this means measuring which frequencies have the largest amplitudes in the frequency domain of an image. For wavelet transforms, signal strength is a measure of how much of a wavelet is in a signal. This may sound like a convolution, and that's exactly what a wavelet transform is. This is shown by the discrete wavelet transform formula below, where W is a bank of wavelet filters, t is the location in the image, x is the image and y are the output feature map.

$$y[t] = \sum_{\varphi \in W} x(t) * \varphi_{m,n}(t)$$

Wavelets can be classified as discrete or continuous. Wavelets have been in use since the early 20th century, originating with Harr wavelets [11] which have enjoyed a wide range of applications, notably in Viola-Jones facial detection [12]. Harr wavelets are discrete wavelet transforms. There are many applications of wavelets, for example scattering wavelets are used in convolutional scattering wavelet networks [4]. The method in the paper uses sets of different scale Morlet wavelets, an approach which has desirable properties such as deformation, rotation, and scale invariance. Morlet wavelets do not have a simple inversion, requiring direct optimisation [13] or generative networks [14] to solve them. In PCWN the filter banks are based on approximations of gaussian derivatives.

1.2.1.4 Segmentation

Image segmentation is the computer vision task of finding a mask that divides an image into separate objects. A mask is a matrix of values, where every value corresponds to a different object, and every position in the matrix corresponds to a position in the image. There are numerous types of object segmentation, but the two main methods are as follows:

- Semantic segmentation: Segmenting images by class of objects, for example, identifying all people in an image.
- Instance segmentation: Segmenting images by instances of one class, for example, identifying every person in an image separately.

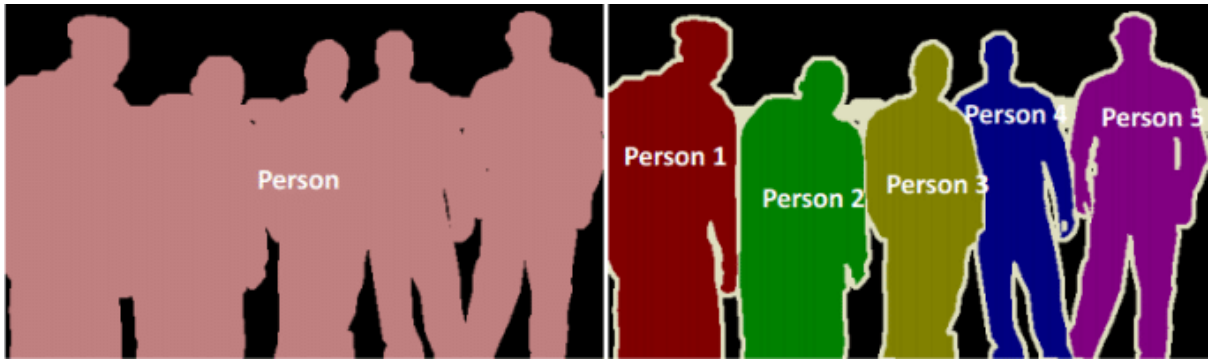

Figure 1: Image segmentation: Difference between semantic and instance [15]

The task PCWN is being used for is semantic segmentation. Semantic segmentation can be used to segment images into any number of classes, but for this use case there will only be 2 classes, this is called binary segmentation.

1.2.2 PCWN

The PCWN architecture is a hybrid of CNNs and wavelet transforms. The network consists of a contracting path, which decomposes an input, and an expanding path, which reconstructs the input. As previously stated, PCWN is meant for linear inverse tasks, where an image with missing data is reconstructed with the missing data predicted by the model. Segmentation is not a reconstruction, and as such the task requires an adaptation to PCWN that will be discussed in the experiment methods section.

1.2.2.1 Decomposition Step

The first step of decomposition is applying any activation functions. The network can work without them, but activations can be helpful for regularizing activity in the network. One activation function that may be of use is the hyperbolic tangent (tanh) and the inverse hyperbolic arctangent (atanh). This activation "squashes" the input between -1 and 1.

The formula for calculating tanh is:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

And the inverse is:

$$\operatorname{atanh}(x) = \frac{1}{2}\log\left(\frac{1+x}{1-x}\right) = \tan^{-1}(x)$$

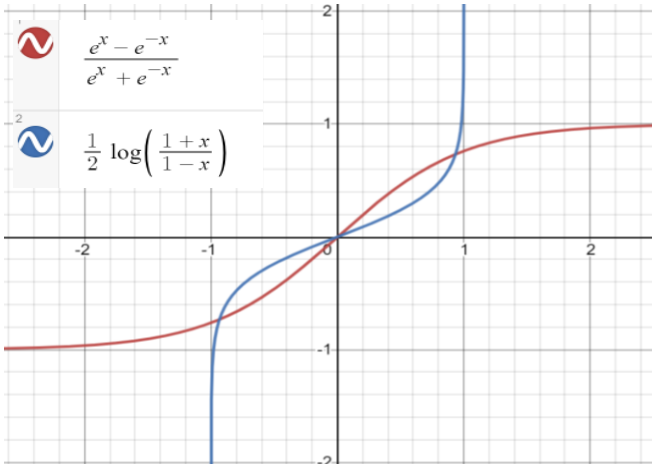

Figure 2: Illustration of tanh and atanh

The bank of wavelet filters used in PCWN are discrete and invertible. At every level of decomposition, each filter in the filter bank is applied to each channel in the partial decomposition. The filter bank used is based on approximations of the zeroth, first and second derivatives of the gaussian, applied along the x and y direction. Every input channel outputs 9 more channels in the next decomposition. This process is shown by the following formula:

$$t_{l+1}(9z + 3j + i) = G_i^x * G_j^y * s_l(z)$$

Where $t_{l+1}(9z + 3j + i)$ is the decomposition at tensor channel "$9z + 3j + i$" at level "$l + 1$" before PCA is applied. $s_l(z)$ is the decomposition tensor channel z at level l of the decomposition, after PCA has been applied, or alternatively it is the input. G_i^x and G_j^y are 1D Gaussian derivatives in order x and y of order i and j respectively, where $i \in 0,1,2$ and $j \in 0,1,2$.

Below is an example of one level of wavelet decomposition. The input is an image, which naturally has 3 channels, red, green, and blue. The number of output channels is 27, however many of these channels are redundant. Therefore, PCA is used, to reduce the number of channels, whilst retaining the variance. The full decomposition step applies the filter bank and then projects the output into the PCA subspace

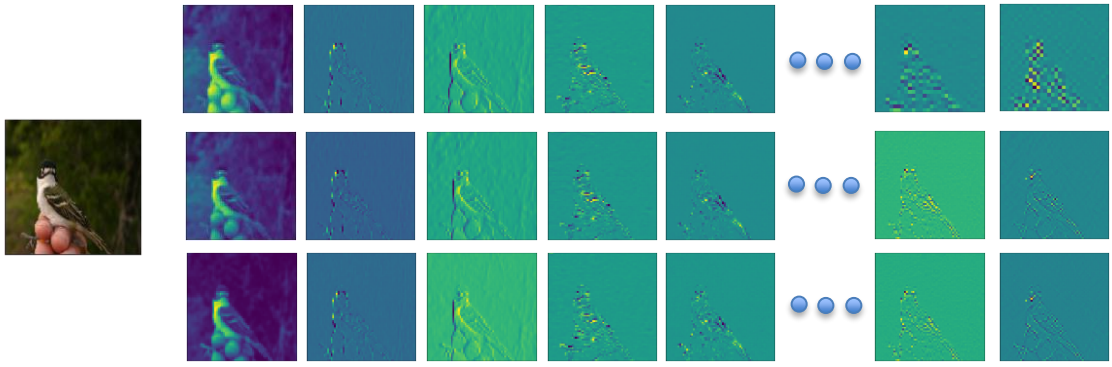

Figure 3: Wavelet decomposition of an image of a bird

To project the decompositions into the PCA subspace is simple once the principal components are calculated. The projection is calculated as a weighted sum of the channels in the decomposition.

$$s_l(z) = b_l(z) + \sum_{i=0}^{Z_l} W_l(z, i)t_l(i)$$

$s_l(z)$ is the PCA decomposition of channel z of the tensor. Z_l is the number of channels before projection into PCA subspace. $W_l(z, i)$ is the i^{th} component of the z^{th} principal component at level l in the decomposition. $b_l(z)$ is the bias term, which is calculated as the dot product of the negative of the mean and the principal component.

1.2.2.2 Training

A point of note is that PCWN does not use stochastic methods for learning such as backpropagation. This means the outcome of a PCWN is deterministic which has the advantage of repeatable results.

The PCWN trains sequentially, layer by layer. The training process loops over every image and feeds it through the filter bank. This creates a feature map, with many redundant channels. Every 2D position in this feature map is a vector, where the length of the vector is the number of channels in the

decomposition. The vectors are used to calculate the covariance matrix and the mean of the data. Using the covariance matrix and mean the principal components are calculated. The number of principal components kept can be a fixed number or the number required for a specific explained variance.

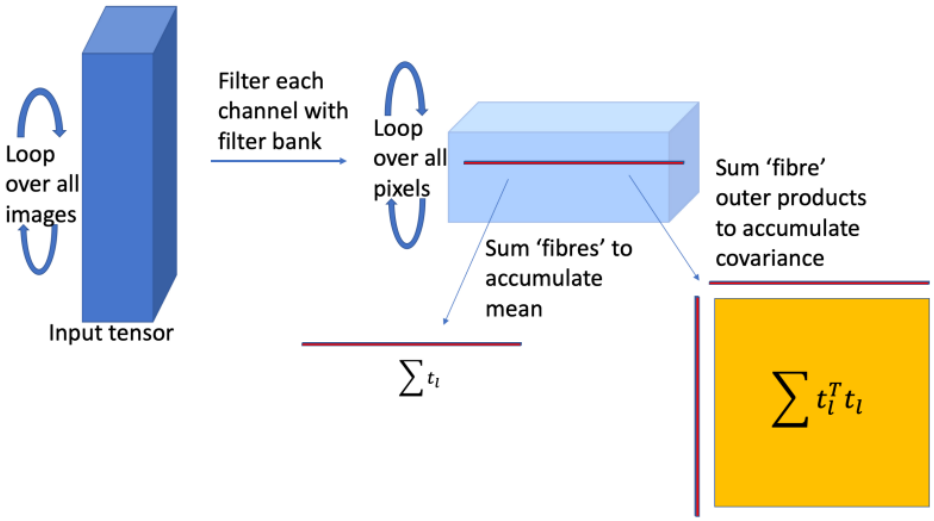

Figure 4: PCA training process [1]

A point to note is that the training process happens in every layer of the decomposition part of the network. This is why the PCA step is so important, as without it the number of channels would rapidly increase.

For every decomposition layer that is calculated, a corresponding reconstruction layer is also created

1.2.2.3 Reconstruction Step

Reconstructing the image is effectively the reverse of the decomposition steps. The first step is to apply the inverse of any activation functions to the decomposition. The result of this is the filtered images expressed in the PCA subspace. Reversing the PCA step is accomplished with the following equation.

$$t_l'(z) = m_l(z) + \sum_{i=0}^{T_l} W_l(i,z)s_l(i)$$

Where $m_l(z)$ is the mean for channel z or level l of the decomposition, T_l is the number of principal components used in level l, $W_l(i,z)$ is the z^{th} component of the i^{th} principal component at level l in the decomposition, and $s_l(i)$ is the filtered image in PCA subspace that we are reconstructing.

The next step is to reverse the wavelet filters. This step is accomplished using either the Moore-Penrose pseudo inverse or using reconstruction filters with compact support. Compact support means that the wavelet produces a value of 0 outside of some finite interval. The pseudo inverse method woks b constructing a filter matrix Φ which is the function of the wavelet filtering including the stride and border handling, and calculating the inverse of Φ using the formula:

$$\Phi' = (\Phi^t\Phi)^{-1}\Phi^t$$

Where Φ' is the pseudoinverse. The pseudoinverse is then applied to the decomposition, which is now in the filter subspace, after the PCA has been reversed. The result of multiplying Φ' with the

decomposition is the previous decomposition. If this is the last layer of the reconstruction, then the result is the reconstructed image.

1.2.3 Deep Learning Architectures for Segmentation

1.2.3.1 U-Net Architecture

The U-net architecture was conceived in 2015 and is so called because of its U-shaped architecture. The success of U-map comes from the use of skip connections. These are connections in the network that allow layer outputs to bypass layers of the network. The result is then concatenated or reintroduced later in the network. Skip connections, or residual connections, first appeared in highway networks [16]. Highway networks implemented skip connections with gates, like a long-short term memory (LSTM) [17] mechanism. Perhaps the most famous architecture to use skip-connections is ResNet [18], an image recognition network that is still used as a benchmark today.

Modern methods for image segmentation use multi-scale feature maps, which means the feature maps encodes information from small resolution features, such as a paw in an image of a cat, as well as large scale features, such as the background of the image. U-net is not an exception. Skip connections allow U-net to use multi-scale feature maps when calculating the segmentation mask. In a standard fully convolutional network there is a contracting path, which reduces the resolution of the feature map with every successive convolution and pooling operation. This means that feature information is lost, especially information regarding location of the object being segmented. Skip connections means that the network can retain information that would otherwise be lost. This is illustrated in the network architecture below.

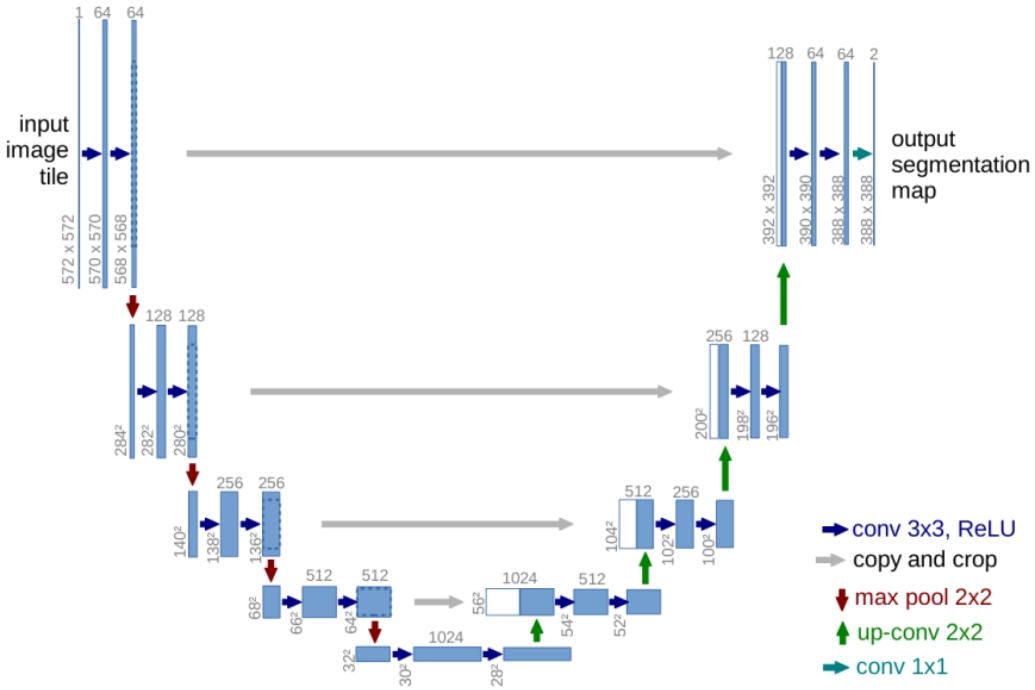

Figure 5: U-net architecture [2]

For image segmentation, the location of an image is important in constructing the mask. By using skip connections, it is possible to reuse high-resolution features to supplement up-sampled outputs from deeper layers, resulting in a more precise output. In the below example the blue box shows the location information provided by the high resolution feature to the low resolution feature in the yellow box.

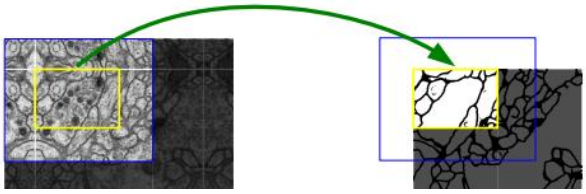

Figure 6: Example of high-resolution features being used to localize low-resolution features [2]

1.2.3.2 DeeplabV3 Architecture

The DeepLabV3 is state of the art for semantic segmentation. The approach proposes the use of atrous convolutions, used in a method called Atrous Spatial Pyramid Pooling (ASPP) [19]. ASPP resamples a feature map with multiple atrous filters of increasing dilation and combines the resultant map using a pooling operation. Atrous convolutions are convolutional filters with a dilation rate which determines the padding between values in the filter. Atrous convolutions are excellent for multi-scale features, capturing dense features whilst maintaining the resolution of resultant feature maps.

The formula for calculating a 1-dimensional atrous convolution is:

$$y[i] = \sum_{k} x[i + r \cdot k] \cdot w[k]$$

Where y is the output feature map, x is the image, i is a location in x and y, r is the atrous rate and w is the kernel. By increasing the atrous rate, we increase the number of zeros between consecutive values in the filter. For example, an atrous value of 1 is a conventional filter. An atrous rate of 2 pads one zero between every value in the filter.

1	2	1
2	3	2
1	2	1

Figure 7: Atrous convolution: Atrous rate = 1

1	0	2	0	1
0	0	0	0	0
2	0	3	0	2
0	0	0	0	0
1	0	2	0	1

Figure 8: Atrous convolution: Atrous rate = 2

The reason for including details on DeeplabV3 and atrous convolutions is that the ASPP method could be applied to future PCWN work. Atrous convolutions have been used with undecimated wavelet networks before successfully [20] and would be an interesting edit to the PCWN architecture, however ASPP was beyond the scope of this project.

1.2.3.3 Transformers

Transformers architectures [21] have seen widespread use in natural language processing, but recently have been applied to many computer vision tasks, including segmentation [22], with state-of-the-art results. Transformers are interesting architectures which use multi-headed attention to "attend" to different areas within some arbitrarily large window. For example, with images a transformer can attend to a patch in the top left corner, and another patch in the bottom right. This attending mechanism allows context to be captured over the entire window without information being lost. An attention head is a mechanism which combines 3 random vectors called key, query, and value using scaled dot product attention. Multi-headed attention means many attention heads are created and the outputs are concatenated, similar to pooling the output of multiple kernels being in a CNN.

Figure 9: Image of a dog broken up into patches

1.2.4 Research Questions

There were several initial questions posed:

- Can you apply the network to a novel task?
  - Image segmentation
  - Crowd counting
- Can another dimensionality reduction be used instead of PCA?
- What are the effects of changing network parameters?

Of these questions, 2 have been completed, with the remainder left for a future project.

1.2.4.1 Can you apply the network to a novel task? (Image Segmentation)

This task was the main focus of the project. There are a number of technical aspects to answering this question which were expected to take a long time, which is why the remaining questions have been delegated to future research.

1.2.4.2 What are the effects of changing network parameters?

This task is a natural extension of the segmentation task. Some experimentation would be required to determine the best parameters for segmentation. There are several parameters to vary through experiments. For the model, the parameters to edit include layer count, percentage of channels kept, different architectures and different activation functions. For the dataset, the editable parameters are image resolution and training data size.

1.3 Process

1.3.1 Overview

The project was an individual effort which meant an exact agile methodology was not an appropriate choice for the task. Waterfall planning was not appropriate either, as the requirements were not clear at the start of the project. The most sensible approach was to select aspects of agile to apply to account for the unknown requirements of the project and the consequent changes. Kanban boards were used to track progress in the project.

A using Kanban tools is most effective in a team. As the project was carried out by an individual there required some adaption to the techniques. The changes were mostly simplifications of the method, such as removing work in progress (WIP) limits. Lots of features are only beneficial when

working as a team, removing them streamlined development as less time was spent on features intended to improve collaboration.

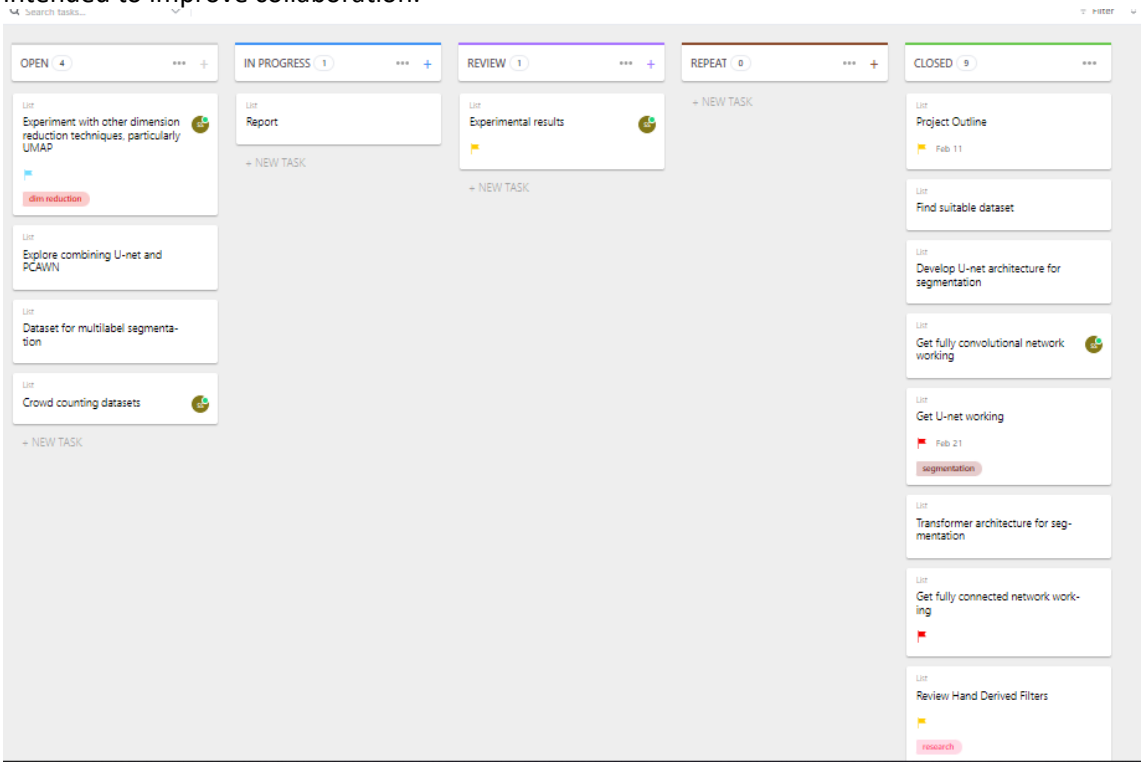

Figure 10: screenshot of the Kanban board used for tracking tasks, using click-up https://app.clickup.com/

Week-long iterations emerged naturally, as there were weekly meetings with the project supervisor. During these meetings there were discussions about current progress, and the next set of features to implement. This customer orientated, with the project supervisor as the customer, is consistent with feature driven development. The features were not agreed at the beginning of the project, rather they were the product of weekly meetings with the supervisor.

There were 4 iterations of the software, corresponding to roughly 5 weeks of development work. With 3 weeks used for research at the beginning of the project and the remaining time dedicated to writing the report.

Table 1: Table of features

Feature No	Feature Name	Complexity (1-10)	Priority	Description	Implemented
F1	Connected PCWN	7	High	Set up fully connected PCWN for image segmentation	Yes
F2	Convolutional PCWN	8	High	Set up convolutional PCWN for image segmentation	Yes
F3	U-net	6	High	Set up a U-net architecture for segmentation	Yes
F4	Deep CNN	5	Medium	Set up a deep autoencoding CNN for segmentation	Yes
F5	Loading bird dataset	2	High	Develop a method for easily loading the Caltech bird dataset into test-train splits	Yes

F6	Loading pet dataset	2	Medium	Develop a method for easily loading the Oxford pet dataset into test-train splits	Yes
F7	Image pre-processing	3	High	Processing and standardising the images to make it easier for models to learn	Yes
F8	Segmentation pre-processing	4	High	Processing and standardising the segmentation mask to make it easier for the models to learn	Yes
F9	Automated experiments for PCWN	5	Low	A notebook for testing numerous settings for PCWN models	Yes
F10	U-net combined with PCWN	9	Low	Adding skip-connections to PCWN, resulting in a U-net like architecture	No
F11	Multi-label segmentation	5	Low	Extending models to work with multi-label segmentation datasets	No
F12	PCA alternatives	9	Low	Replacing PCA in the network with a different dimension reduction technique	No

1.3.2 Iteration 0 - Setup

Iteration 0 corresponds to the research and spike work for this project. The first steps were to set up development software such as GitHub [23], Jupyter Notebooks [24], Spyder [25] and Anaconda [26]. This also included setting up the directory structure and cloning the GitHub repository with the original code (https://github.com/bptiddeman/PCWN.git), selecting the files that would be useful for the segmentation task.

1.3.3 Iteration 1

The first features implemented were the dataset loading and preprocessing features [F5,F6,F7&F8]. These were the first features to develop as it would be impossible to train any models without data in a sensible format.

The next feature to develop was the connected PCWN [F1]. This is a simpler architecture than the convolutional approach which is why it is the first to be developed.

1.3.4 Iteration 2

The next feature developed was the convolutional PCWN [F2]. The convolutional aspect added some difficulty, but overall the method was not vastly different to the connected PCWN. There was more time available in this iteration as lessons were learnt from developing the fully connected approach. This left time to develop the U-net architecture [F3].

1.3.5 Iteration 3

Iteration 3 was spent perfecting the U-net model and developing the autoencoding CNN model [F4] There was time dedicated to tweaking the PCWN models' parameters to see which parameters should be varied in the experiments and what kind of ranges should be used.

1.3.6 Iteration 4

Using the data collected from the manual experiments previously, an automated notebook [F9] was created to systematically test different model configurations such as keep percent, layers and image resolution. This iteration was dedicated to collecting data for analysis, from the PCWN and deep learning models.

2 Experiment Methods

2.1 Hypotheses

- PCWN will have similar results to a deep autoencoding CNN
- PCWN will improve with more layers
- U-net will be the better approach than PCWN

2.2 Hardware

Originally, the experiments were run on a HP laptop with 8GB of RAM, a quad-core intel I7-7500 CPU @ 2.7GHz with no GPU and an windows operating system. However, there were significant issues with memory management which incentivised upgrading the laptop to a newer model. AWS was considered, with an exploration of free tiers of elastic computation instances. Instead of AWS, an upgraded laptop was used, as the previous laptop was quite old and the elastic compute instances were expensive for the system requirements needed.

The new model was a Dell laptop with 16GB of RAM, an octo-core intel I7-11800H CPU @2.3GHz and a NVIDIA GeForce RTX 3060 GPU, running windows as an operating system. This upgrade solved the memory issues and sped up the experiments.

2.3 Datasets

There were two datasets used, California Tech Birds Dataset (2010) [27] and the Oxford-IIIT Pet Dataset [28]. These datasets were selected as it would be reasonably easy to turn the segmentation masks used in the datasets into binary segmentation masks using modular division. Both datasets have around 7000 images.

2.4 Metrics and Measurements

2.4.1 Dice Coefficient and Intersection Over Union (IoU)

Monitoring the training process and comparing the performance of different models relies on the use of metrics for measuring the accuracy of segmentation masks. The simplest way of doing this is pixel error, which is the number of pixel locations where the predicted mask and actual mask disagree. This method would be easy to implement however it's too sensitive to small displacements in the location of the boundaries of the masks.

One candidate metric to use is Intersection Over Union (IOU) [29] , or the Jaccard index. The IoU measures the intersection of the predicted and actual masks, divided by the union of both the masks. IoU can be visualized as shown here.

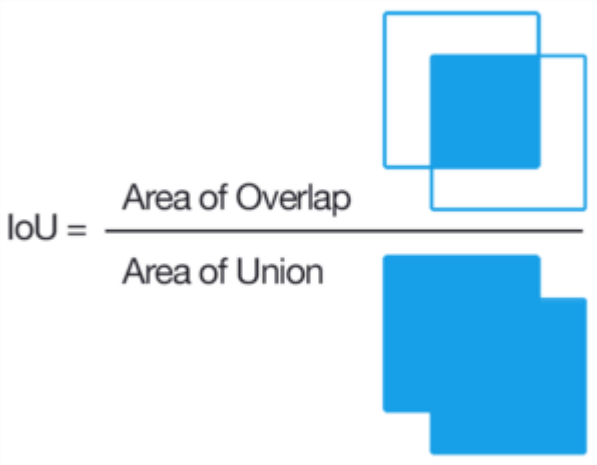

Figure 11: Visualisation of IoU [30]

The formula for calculating IoU is:

$$IOU = \frac{TP}{TP + FP + FN}$$

The second possible metric is the Dice coefficient [31, 32]. The Dice coefficient core is calculated the same way as the F1 score, or the harmonic mean, the formula is very similar to the Jaccard index, but the Dice coefficient weights true positives higher.

$$DiceScore = \frac{2TP}{2TP + FP + FN}$$

The Dice score and Jaccard index have very similar formulas. Dice score is used instead of the IoU as IoU is a better indicator of the worst-case performance whereas dice score is a better indicator of average performance [33]. Both methods do not cope well with examples where the segmentation mask is very small, however both the oxford pet dataset and the California tech bird dataset use segmentation mask that cover 40% of the image on average, so small masks should not be an issue.

2.4.2 Other Metrics

There are several other metrics that will be recorded, but that may not be used in the experiments. For example, training time will be recorded. This will be helpful in comparing the efficiency of models. The Dice coefficient will be calculated for both the training data and the test data, which will be an indicator of overfitting or underfitting in the PCWN or the deep learning approaches.

2.5 Measuring Performance

2.5.1 Repeated Experiments

The deep learning models take too long to train to repeat multiple times, and for PCWN the result is always the same as there is no stochastic element to the learning process, it is completely deterministic. This means that every model is trained once.

2.5.2 Standard Deviation and Mean

Mean dice coefficients and standard deviation are calculated by running the models over the dataset and recording values. The mean will be a good indicator of the performance of the models, and the standard deviation is an indicator of how stable the models' predictions are. The performance of the

models on individual instances of data from the test and train datasets will also be recorded and visualised as a box plot.

The mean IoU may also be recorded, but it won't be discussed in experiments.

2.5.3 Graphs

Seaborn and matplotlib will be used for graphing results. Seaborn is effective at producing boxplots which will be excellent for visualizing model performance on separate datasets. Matplotlib offers simple methods for plotting line graphs, which will be used for plotting the effect of changing PCWN parameters versus the result on the mean dice score, as well as the training curves of the deep learning models.

Pandas will be used for handling tabular data, and numpy and tensorflow will be used for handling large sequences of numerical data.

3 Software Design and Implementation

3.1 Design

3.1.1 PCWN and Linear Least Squares

3.1.1.1.1 PCWN and Linear Least Squares

PCWNs have worked excellently for linear inverse tasks in computer vision. For segmentation the PCWN architecture requires some edits. The method used for adapting PCWN segmentation is incredibly versatile and can be adapted to several tasks, such as crowd counting and point detection.

The method involves training 2 separate PCWN networks. One network trains on the input, in this case an image. The other network trains on the images with the segmented masks applied. Both networks are only able to reconstruct the inputs, or corrupted versions of them. To extend PCWN to the task of segmentation, there needs to be a method for mapping the decomposition of the image network to the decomposition of the segmentation network.

The mapping can be thought of as a linear equation, where y is the segmentation decomposition, x is the image decomposition, and matrix A and vector b are variables that once solved will map x to y.

$$y = Ax + b$$

Using linear least squares, it is possible to find a solution to this equation quite efficiently and simply. The aim is minimising the expression below:

$$min(\sum_i ||Ax_i + b - y_i||_2^2)$$

Where i is the number of elements in the dataset.

To do this requires finding the derivatives of each element with respect to A and b and setting them to zero. The derivation is as follows.

$$\sum_i ||Ax_i + b - y_i||_2^2 = \sum_i (\mathrm{Ax_i} + \mathrm{b} - \mathrm{y_i})(\mathrm{Ax_i} + \mathrm{b} - \mathrm{y_i})^{\mathrm{t}}$$

$$\frac{d}{dA} = 2\sum_i (\mathrm{Ax_i} + b - y_i)x_i^t = 0$$

$$\frac{d}{db} = 2\sum_i (\mathrm{Ax_i} + b - y_i) = 0$$

The derivatives are expanded and rearranged.

$$\sum_i (x_i^T y_i) = A\sum_i (x_i^T x_i) + b\sum_i (x_i^T)$$

$$\sum_i (y_i) = A\sum_i (x_i) + nb$$

The vector b is eliminated through substitution:

$$b = \frac{\sum_i (y_i) - A\sum_i (x_i)}{n}$$

The substitution results in the following equation:

$$A(\sum_i (x_i^T x_i) - \frac{\sum_i (x_i^T x_i)}{n}) = \sum_i (x_i^T y_i) - \frac{\sum_i (x_i^T y_i)}{n}$$

For the readers sake, the equation can be simplified by introducing X and Y as substitutions for $\sum_i (x_i^T x_i) - \frac{\sum_i (x_i^T x_i)}{n}$ and $\sum_i (x_i^T y_i) - \frac{\sum_i (x_i^T y_i)}{n}$ respectively, giving the following equation.

$$A \cdot X = Y$$

To find A, both sides of the equation are multiplied by the inverse of X. In practice, the Moore-Penrose pseudo-inverse [34, 35] is used, as the system of linear equations being solved may not have a unique solution.

$$A = X^{-1}Y$$

It is then trivial to find vector b by substituting A back into the original equation and rearranging:

$$b = \bar{y} - A\bar{x}$$

$\bar{y}$ is the mean of the segmentations and $\bar{x}$ is the mean of the image decompositions.

The method for calculating A and b can be accomplished in 2 ways, a fully connected method, or a fully convolutional method. The method of using linear least squares is the same, but the shape of the tensors is different.

3.1.1.1.2 Fully Connected Method

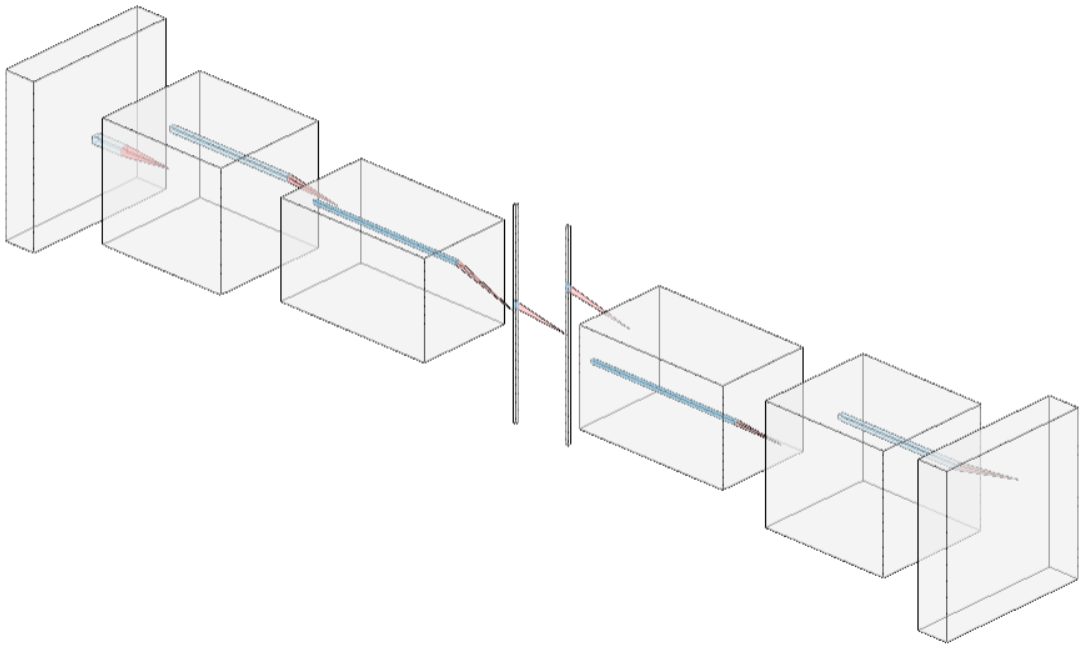

Figure 12: Fully connected PCWN architecture

In the fully connected method, the decompositions are flattened. This approach is location variant, and so is suited to images where the object of interest is in a consistent position, such as the centre of the image. When calculated using the fully connected method, the X and Y matrix can become very large. This can cause memory issues, especially when calculating the Moore-Penrose pseudoinverse. The actual size of A is the product of the product of the dimensions of each decomposition.

The method for solving the linear inverse for the fully connected PCWN is exactly as stated in the previous section. However, when using A and b to transform an image decomposition into a segmentation decomposition, the image decomposition must be flattened, and the output is then reshaped, where the shape is that of a valid segmentation decomposition.

The python code for this operation can be found in the appendix.

3.1.1.1.3 Fully convolutional method

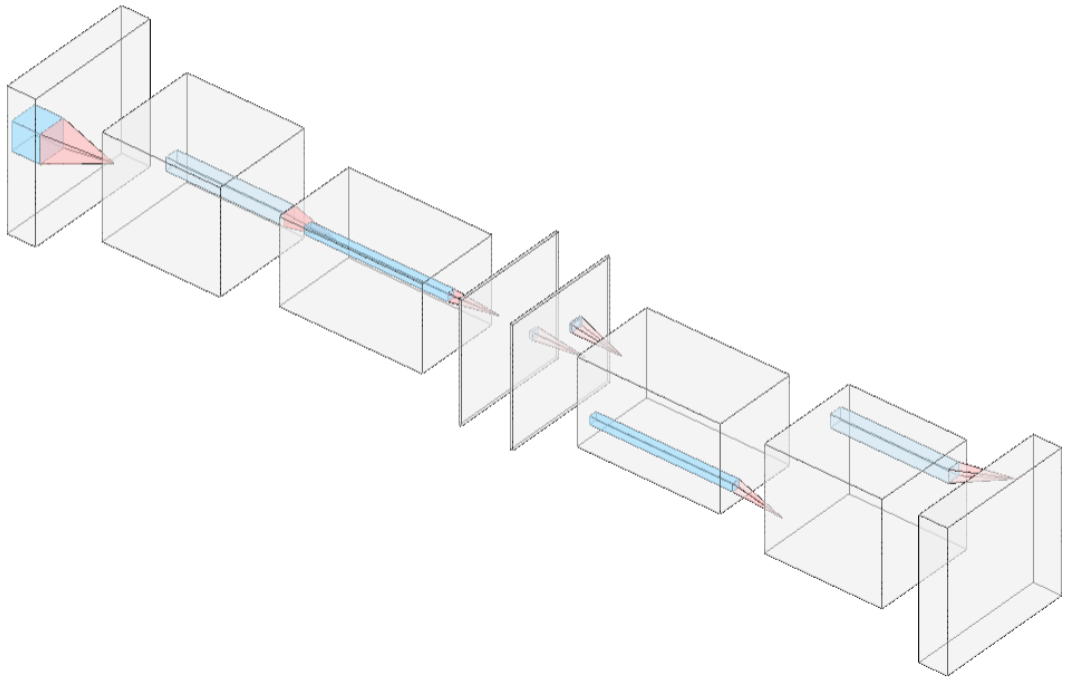

Figure 13: Fully convolutional PCWN architecture

The fully convolutional method is slightly more complex. This method is appropriate for images where the object of interest does not have a fixed position. The size of the matrix A is much smaller for a convolutional network approach, reducing computational complexity from calculations such as the pseudo-inverse. This means higher resolution images may be used, and deeper networks.
Solving the linear inverse for fully convolutional PCWN is slightly different to the fully connected method, as the convolutional approach uses matrices rather than vectors.

For the most part, the two approaches are very similar. When calculating the covariance between the decompositions, the matrix multiplication between a transposed decomposition and another decomposition is replaced with an outer product multiplication between decompositions. This is effectively the same operation but for a matrix.

The other difference is how the A matrix and b vector are applied when solving y=Ax+b. This method calculates a 1x1 convolutional filter, so A is reshaped as 1x1 filter and then applied to the image decomposition. The bias is added to the result. Again, this is effectively the same process as before but adapted for a matrix.

The python code for this operation can be found in the appendix.

3.1.1.2 Segmentation

Segmentation uses masks formed of nominal values. In binary segmentation, the mask has values of 0 or 1. The problem with using PCA for nominal data is that PCA makes numeric assumptions on the data, which break down when working with nominal data, causing the PCWN to incorrectly reconstruct the segmentation mask as a blurred point.

The workaround used to deal with this is applying the mask to the image, where the image is clipped to pixel values between 10-255, and the segmentation mask is formed of 0's across every channel. The reason for clipping the image pixel values is to increase the variance between pixels that are in

the image, and pixels that are in the segmentation mask, and reduce the chance of a false positive, where a very dark pixel is thought to be part of the mask when training.

The method for doing this takes the element-wise multiple of the image and the inverted mask. The zeros in the mask will set the object pixels to zero when multiplied together, and the ones in the mask will retain the background pixels.

Figure 17: Original segmentation mask

Figure 16: Inverted segmentation mask

Figure 14: Original image of bird

Figure 15: Image of bird with mask applied

To obtain the original mask, a simple thresholding is applied. The value used for this is 0.1, so any pixel that has a value below this is classified as a mask (note that pixel values are normalised between zero and one). For the deep learning methods the original segmentation mask is used as there are no numeric assumptions like with PCA that will cause issues.

3.1.2 Deep Learning Methods

Two deep learning methods were implemented. The first was the U-net architecture, the second was a deep autoencoding CNN. The reason for implementing two deep learning architectures was to provide some idea on how the skip connections impact the success of a segmentation model, as future PCWN work would look at using skip connections with PCWN. The deep CNN model was implemented to directly compare how a deep learning method compares to PCWN, as they both use a pyramid style decomposition and reconstruction. The U-net model is implemented as it is a standard algorithm used for segmentation.

The U-net architecture has been discussed previously, but the deep autoencoding CNN has not. An autoencoder is an ANN which compresses and encodes an object into some latent representation and reconstructs the object using the latent representation. The latent representation is often called a feature vector. The feature vector has several applications, from image search to data compression. An autoencoding CNN is simply an autoencoder which uses convolutional layers, making the architecture suited for images. The architecture is identical to PCWN, except that the autoencoding CNN learns the filters, whereas PCWN learns principal components, but the filters are fixed. This is the reason why the architecture will be a good comparison for PCWN, as they're identical apart from a different learning mechanism.

Both models will be made from scratch. This gives more control over the architectures, plus implementing U-net from scratch provides code and experience that would be helpful when converting PCWN to a U-net-like architecture.

The transformer model will be implemented, but possibly not tested. This is because two deep learning models are already being trained and having their results analysed. Adding another model to

experiments will take time, as the model needs to be optimised and the performance analysed. Furthermore, a vanilla vision transformer won't produce outstanding results without carefully editing the architecture.

3.1.3 Overall Architecture

The purpose of the software was to generate benchmark image segmentation models, and objectively compare the benchmark models to PCWN models.

The requirements of the software are as follows:

1) Implementation of PCWN architecture for segmentation
2) Implementation of one or more deep-learning methods for segmentation
3) Experiments to find best parameters for PCWN
4) Experiments to compare deep learning methods and PCWN
5) Dataset loading and pre-processing

The proposed architecture for accomplishing the software requirements is as follows:

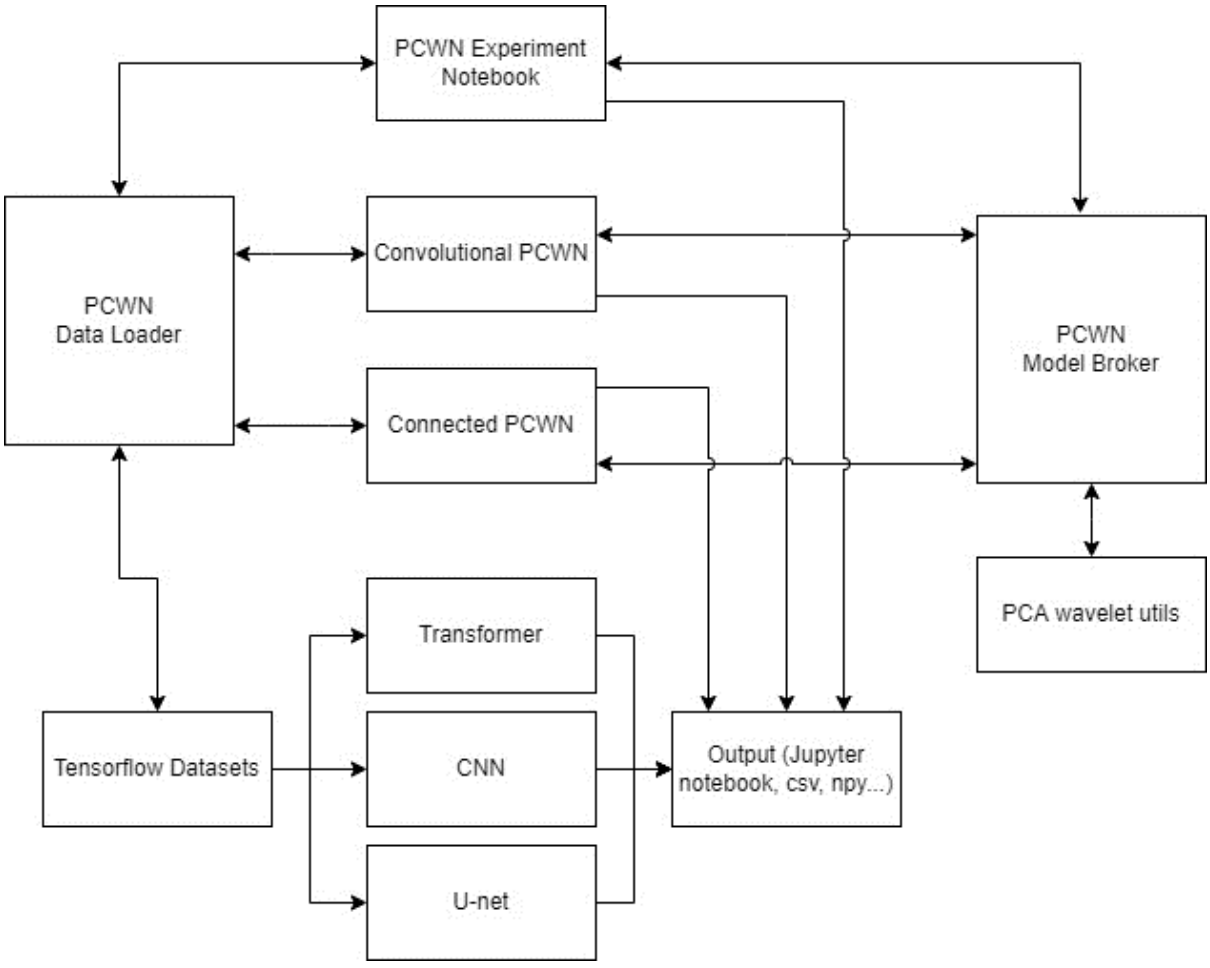

Figure 18: Proposed software architecture

The model above gives a high-level overview of the software components needed for this project.

- **PCWN Data Loader:** Handles the specific pre-processing needed for the PCWN networks, including the applying the segmentation mask to the image. There should be simple methods for loading different datasets with different resolutions and training data sizes
- **Tensorflow Datasets:** Downloads tensorflow datasets.

- **PCWN Experiment Notebook:** Interactive environment to systematically test both the convolutional and connected versions of PCWN.
- **Convolutional PCWN:** Trains a PCWN model, and then uses convolutional linear least squares to find the image decomposition to segmentation decomposition mapping.
- **Connected PCWN:** Trains a PCWN model, and then uses connected linear least squares to find the image decomposition to segmentation decomposition mapping.
- **Transformer:** Implements a transformer architecture for segmentation. Alternatively, another model could be used instead of a transformer. The model trains using tensorflow datasets
- **CNN:** Implements a deep autoencoding CNN, which is trained using tensorflow datasets
- **U-net:** Implements U-net, which is trained using tensorflow datasets
- **Output:** Stand in for the results output of the models. The data may be stored as .csv files, or .npy files. The cell outputs could be used too.
- **PCWN Model Broker:** Handles the loading of PCWN models, should have simple methods for building and loading the head and inverse head of PCWN models.
- **PCA wavelet utils:** Handles the building of PCWN models, this is the code that is provided by the authors

3.2 Implementation

3.2.1 Language

The language of choice was python. This is because the existing code had already been implemented in python, and the language has a plethora of machine learning and data processing libraries that can be used. Python also has GPU compatibility, however with the NVIDIA 3060 line of GPU's this set up is quite complex. With the time available in the project it may not be worth setting this up.

3.2.2 Development Software

Spyder and Jupyter Notebooks were the development environments used, with Anaconda used for managing packages and environments.

Spyder was used when developing modules as it has many useful static analysis features,

Jupyter notebooks were used as they are excellent for developing machine learning systems due to their interactivity, and ability to re-run cells. Notebooks are incredibly easy to test because of rerunnable cells.

4 Results, Testing and Conclusions

4.1 Testing of system

4.1.1 Overall Approach to Testing

Testing is included in the results section as good testing is required for reliable results. Testing was used to check models were working correctly and producing reasonable results.

Testing took a more manual route due to the interactive nature of Jupyter notebooks. Any time functionality needed testing, it was possible to isolate and test it using the cell like structure of the notebooks. This made development incredibly easy.

The use of testing was very important to make sure results were correct and experiments would not crash half-way through. To test the models, a debug flag was used in the data loader, which if set to true would limit the size of the dataset loaded. This meant most tests could be run quickly.

4.2 Issues with Experiments

4.2.1 Memory Errors

By far the most prevalent error was out of memory errors. The cause of this ranged from inefficient programming, insufficient RAM, and poor memory management. To solve the inefficient programming and memory management issues we removed unnecessary objects. The insufficient RAM issue was solved initially by using cloud-based platforms, and then by purchasing a new laptop, upgrading the RAM from 8GB to 16GB.

Memory errors meant initial experiments for the fully connected network were throttled at a 128x128 image resolution. This in turn limited the "count" parameter of the PCWN. This parameter controlled the number of layers in the head, and the inverse head. Every layer reduced the resolution of the composition by half. A 128x128 image would have a feature map with a 4x4 resolution after passing through the head component of a network with 5 layers. This caused an error with the convolutional filters used in reconstruction, as the resolution is too small.

The fully convolutional network had notably fewer memory problems compared to the fully connected version. The reason for this was the calculation of the A matrix, which uses computationally intense methods, such as calculating the pseudo-inverse. The A matrix for the fully connected method was larger than the fully convolutional matrix, meaning the memory usage was much higher for the fully connected variant.

4.3 Determining best parameters for PCWN segmentation method

This experiment tests the strength of variations of PCWN for segmentation. The results are sorted by the mean dice score. We can make several deductions from the results about what factors improve the performance of the models. 4 parameters were varied throughout the experiments, training data size, image resolution, keep percent and layer count. Keep percent and layer count relate to both the segmentation and the image PCWN models. If keep percent is set to 0.1 and layer count is set to 3, then both models use those parameters.

4.3.1 Convolutional Network Approach

4.3.1.1 Training Data Size

Table 2: An experiment on the effect of training size on convolutional PCWN

	Training data size	*count*	*Keep percent*	*resolution*	*Dice mean test*	*Dice mean train*	*Linear inverse train time*	*Image train time*
1	2000	3	0.1	128	0.433 ± 0.227	0.419 ± 0.234	11.633	19.644
2	4000	3	0.1	128	0.519 ± 0.212	0.506 ± 0.221	24.266	37.753
3	7049	3	0.1	128	**0.592 ± 0.196**	**0.585 ± 0.194**	46.681	69.625

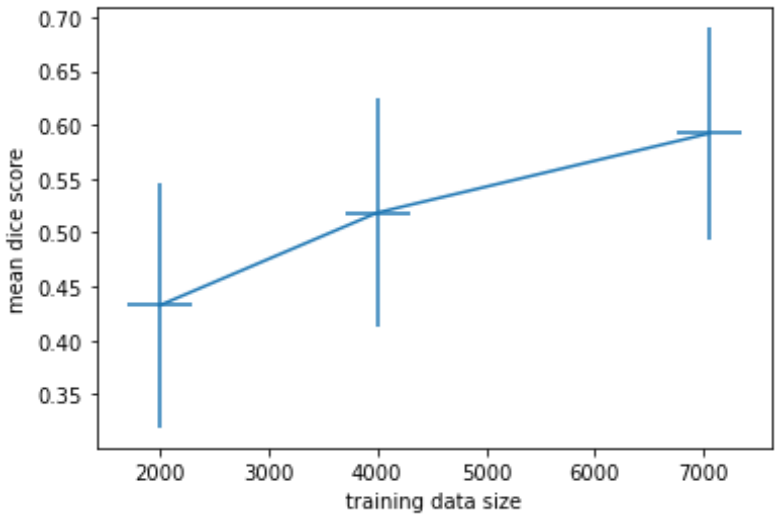

Figure 19: Graph of training data size vs mean dice score

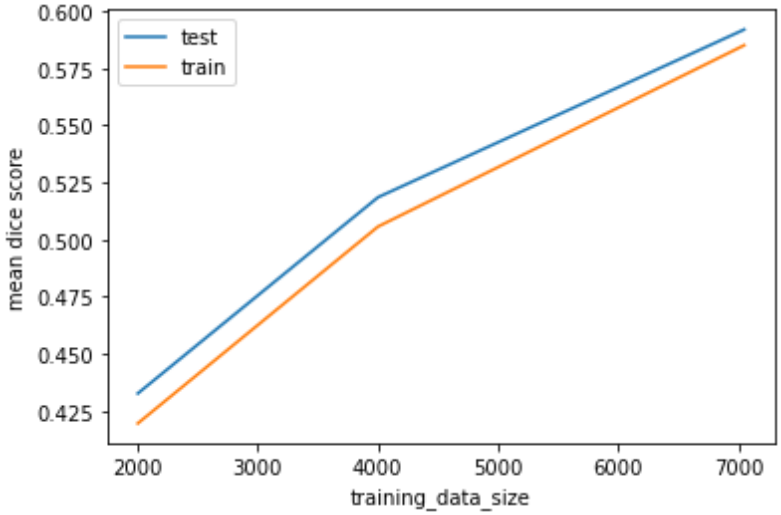

Figure 20: Graph comparing model performance on training and test data for training data size experiment

Training data size gives a significant improvement in the score. In the 2 lowest scoring experiments the reduced dataset is being used. Clearly adding more training data has a positive influence on the dice score.

4.3.1.2 Layer Count

Table 3: An experiment on the effect of layer count on convolutional PCWN

	Training data size	*count*	*Keep percent*	*resolution*	*Dice mean test*	*Dice mean train*	*Linear inverse train time*	*Image train time*
1	7049	3	0.1	128	0.592 ± 0.196	0.585 ± 0.194	48.890	73.204
2	7049	4	0.1	128	0.602 ± 0.182	0.595 ± 0.183	65.922	112.661
3	7049	5	0.1	128	**0.607 ± 0.178**	**0.597 ± 0.177**	86.203	202.736

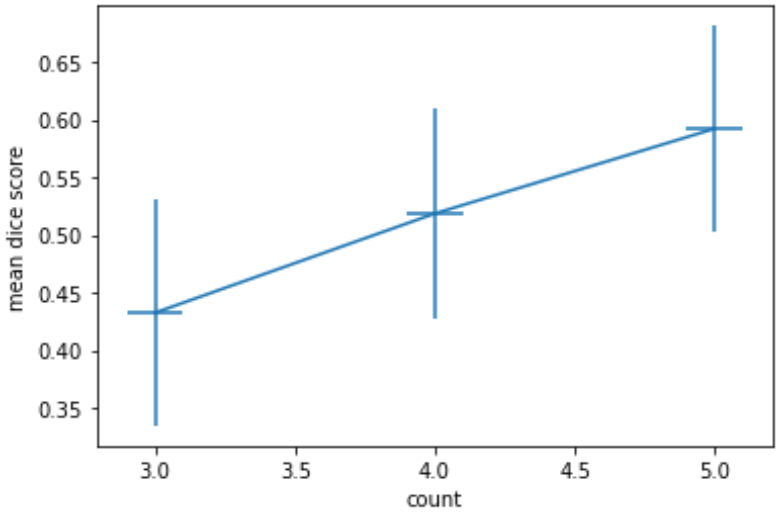

Figure 21: Graph of layer count vs mean dice score

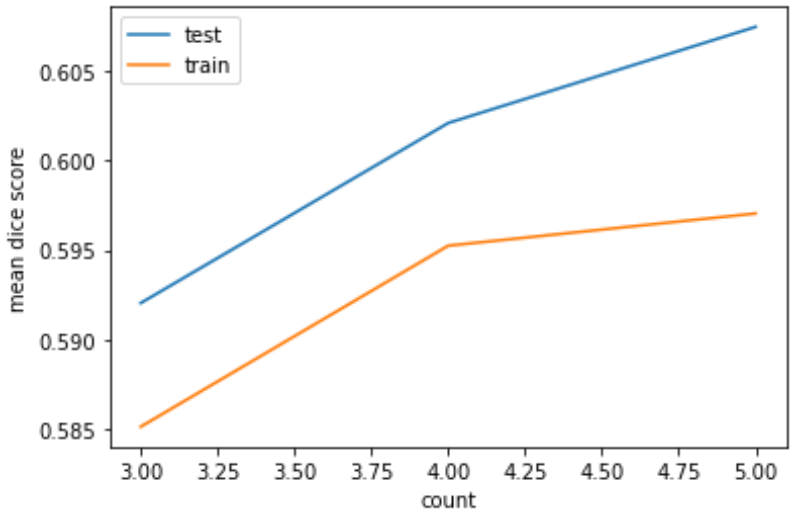

Figure 22: Graph comparing model performance on training and test data for layer count experiment

There is a strong relation between layer count and mean dice score. The more layers the better the performance. Adding more than 5 layers requires increasing the image resolution and keep percent to keep the decomposition size large enough that the model doesn't collapse.

4.3.1.3 Keep Percent

Table 4: An experiment on the effect of keep percent on convolutional PCWN

	Training data size	*count*	*Keep percent*	*resolution*	*Dice mean test*	*Dice mean train*	*Linear inverse train time*	*Image train time*
1	7049	3	0.8	128	0.356 ± 0.236	0.390 ± 0.242	88.184	115.369
2	7049	3	0.6	128	0.590 ± 0.196	0.583 ± 0.194	76.867	109.361
3	7049	3	0.4	128	0.591 ± 0.196	0.584 ± 0.194	69.401	92.353
4	7049	3	0.3	128	0.591 ± 0.196	0.584 ± 0.194	57.961	81.733
5	7049	3	0.2	128	**0.592 ± 0.196**	**0.585 ± 0.194**	56.610	80.350
6	7049	3	0.1	128	**0.592 ± 0.196**	**0.585 ± 0.194**	48.890	73.204

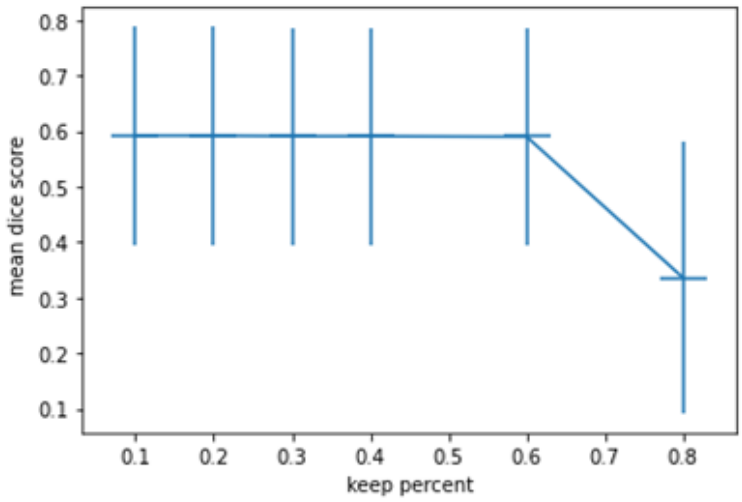

Figure 23: Graph of keep percent vs mean dice score

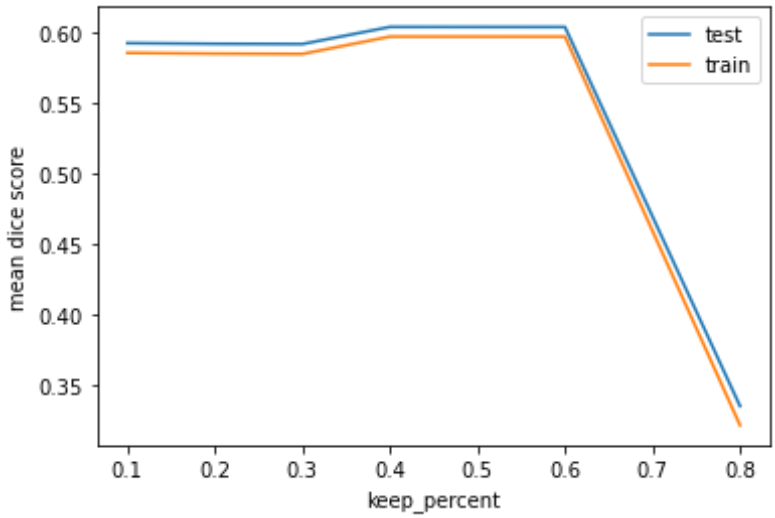

Figure 24: Graph comparing model performance on training and test data for keep percent experiment

Keep percent has a negligible effect on the dice score until higher values are reached. The training time is significantly improved by reducing the keep percent. When the keep percent is above 0.6 the dice score dramatically decreases, for both the test and train datasets. However, to increase the layer count there needs to be an increase in keep percent, otherwise the network can collapse as the decompositions become too small.

4.3.1.4 Resolution

Table 5: An experiment on the effect of image resolution on convolutional PCWN

	Training data size	*count*	*Keep percent*	*resolution*	*Dice mean test*	*Dice mean train*	*Linear inverse train time*	*Image train time*
1	7049	3	0.1	256	0.585 ± 0.200	0.577 ± 0.198	109.921	160.456
2	7049	3	0.1	128	0.592 ± 0.196	0.585 ± 0.194	46.681	69.625
3	7049	3	0.1	64	**0.600 ± 0.189**	**0.592 ± 0.188**	32.274	46.893

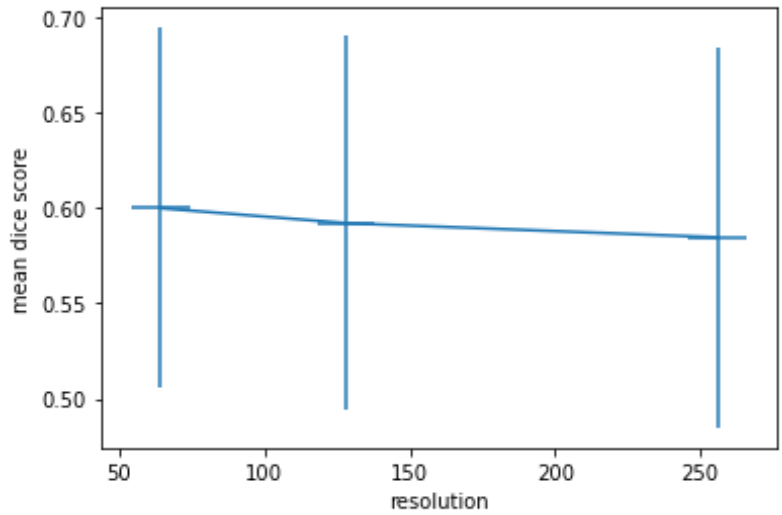

Figure 25: Graph of image resolution vs mean dice score

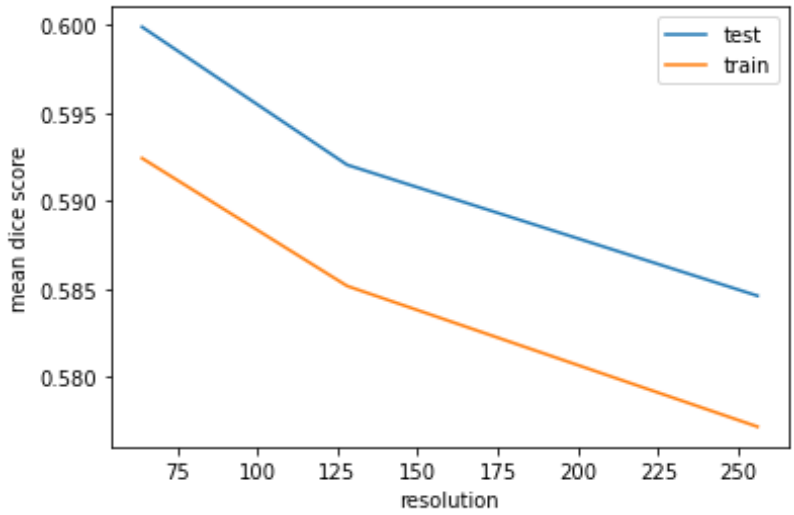

Figure 26: Graph comparing model performance on training and test data for image resolution experiment

Image resolution displays an interesting trend where the dice score decreases as the image resolution increases. This suggests convolutional PCWN works better for lower resolution images. This is an interesting result, as the model was expected to improve with more training data.

4.3.1.5 Example Result

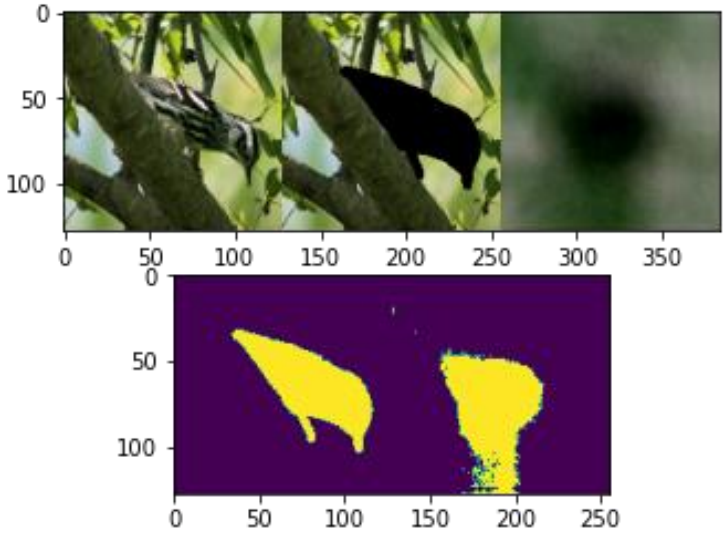

Figure 27: Example output of PCWN on test data, dice score = 0.3

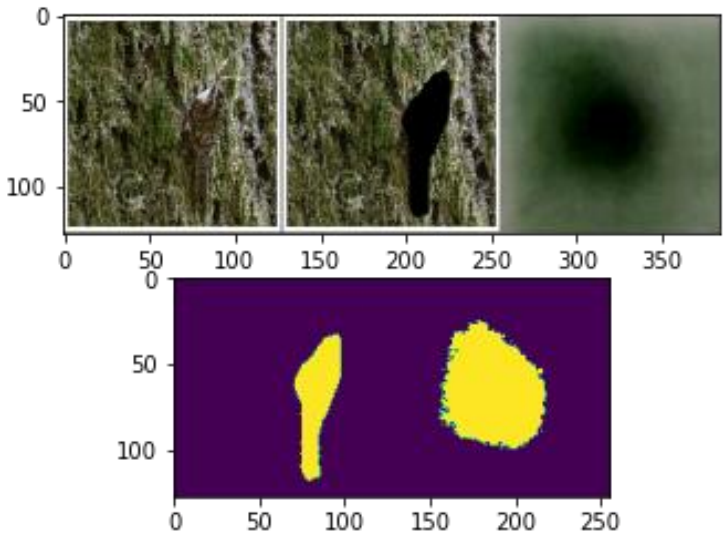

Figure 28: Example output of PCWN on train data, dice score = 0.2

The result above is an example of convolutional PCWN applied to the pet dataset. The image types are as follows, from top left: picture of dog, processed ground truth mask, predicted mask, ground truth mask, and predicted mask after thresholding. Convolutional PCWN shows very consistent results over test and train data. The predicted mask is very generic, a dark blur on a grey background. This indicates that the model is underfitting and is struggling to learn.

4.3.2 Connected Network Approach

4.3.2.1 Training Data Size

Table 6: An experiment to show the effect of training data size on connected PCWN

	Training data size	*count*	*Keep percent*	*resolution*	*Dice mean test*	*Dice mean train*	*Linear inverse train time*	*Image train time*
1	2000	3	0.1	128	0.437 ± 0.162	**0.924 ± 0.103**	323.406	21.657
2	4000	3	0.1	128	0.455 ± 0.164	0.870 ± 0.147	624.663	42.169
3	7049	3	0.1	128	**0.550 ± 0.161**	0.681 ± 0.181	1128.817	75.411

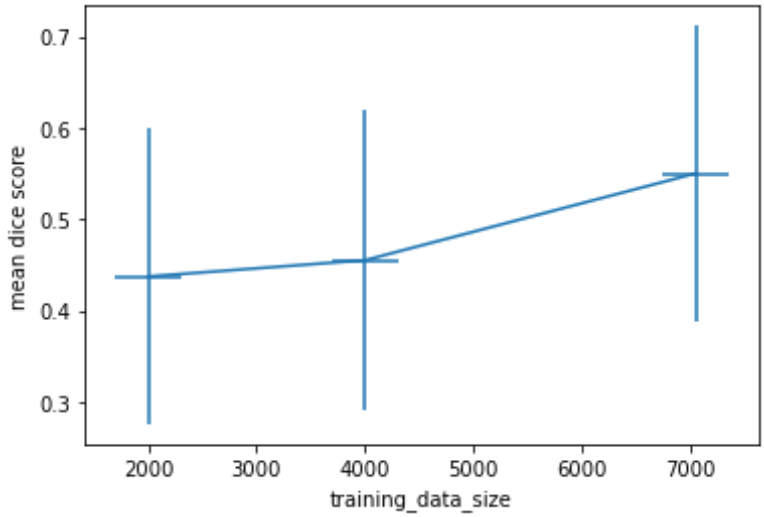

Figure 29: Graph of training data size vs mean dice score

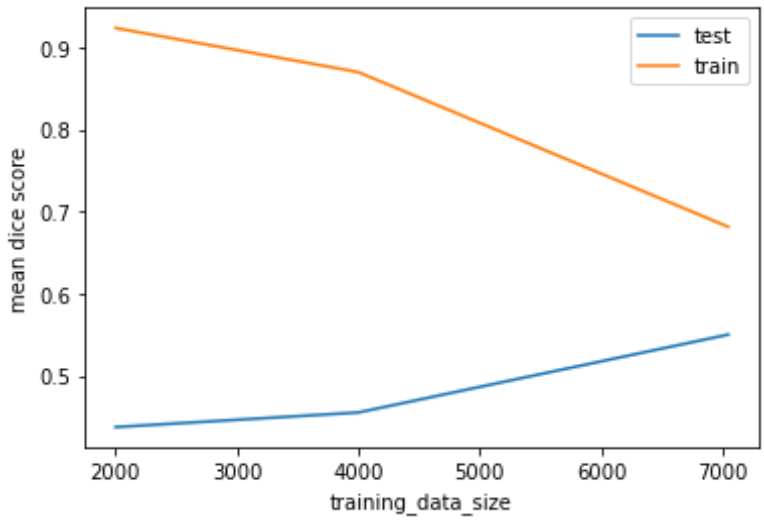

Figure 30: Graph comparing model performance on training and test data for training data size experiment

Training data has a positive effect on the mean dice score for the test data, but a negative effect on model performance when using the training data.

4.3.2.2 Layer Count

Table 7: An experiment to show the effect of layer count on connected PCWN

	Training data size	count	Keep percent	resolution	Dice mean test	Dice mean train	Linear inverse train time	Image train time
1	7049	3	0.1	128	0.550 ± 0.161	0.681 ± 0.181	1128.817	75.411
2	7049	4	0.1	128	**0.580 ± 0.171**	**0.707 ± 0.191**	1173.656	123.507
3	7049	5	0.1	128	0.569 ± 0.184	0.644 ± 0.176	1288.608	227.213

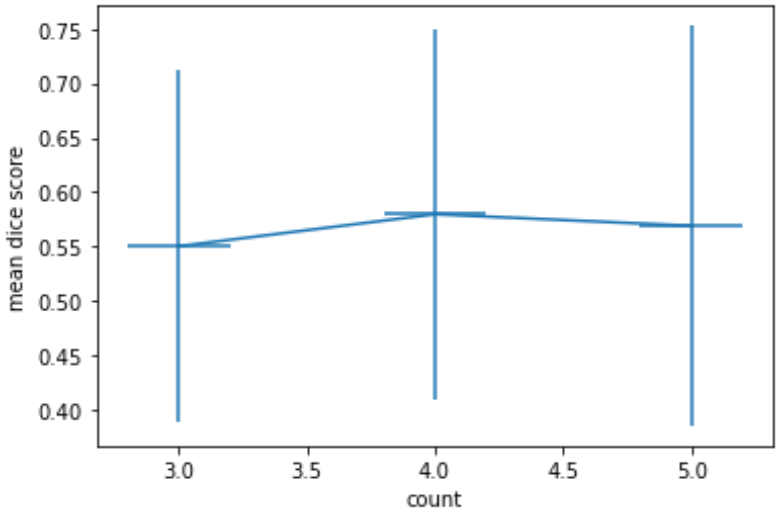

Figure 31: Graph of layer count vs mean dice score

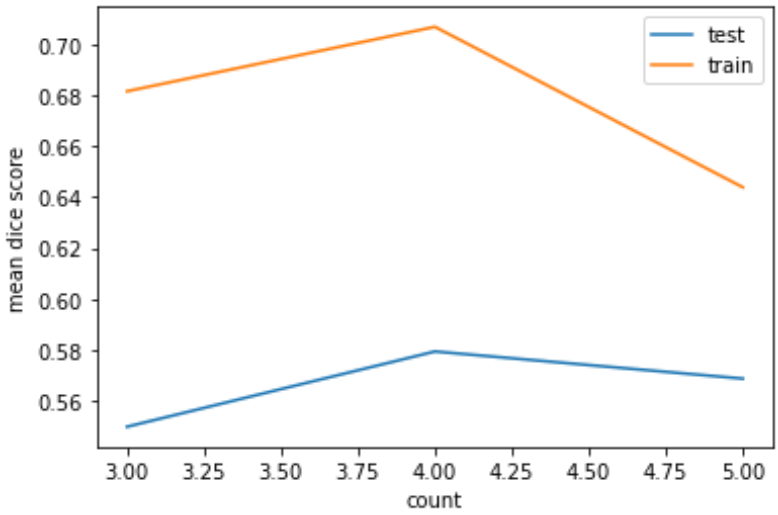

Figure 32: Graph comparing model performance on training and test data for layer count experiment

There is no clear benefit gained from increasing layer count with the fully connected approach. The model performance remains reasonably level as the layer count increases.

4.3.2.3 Keep Percent

Table 8: An experiment to show the effect of keep percent on connected PCWN

	Training data size	count	Keep percent	resolution	Dice mean test	Dice mean train	Linear inverse train time	Image train time
1	7049	3	0.1	128	**0.550 ± 0.161**	0.681 ± 0.181	1132.446	73.672
2	7049	3	0.2	128	0.516 ± 0.163	**0.744 ± 0.192**	5797.012	80.931
3	7049	3	0.3	128	0.526 ± 0.150	**0.744 ± 0.192**	15121.520	86.131
4	7049	3	0.4	128	0.508 ± 0.150	**0.744 ± 0.192**	23655.138	94.741

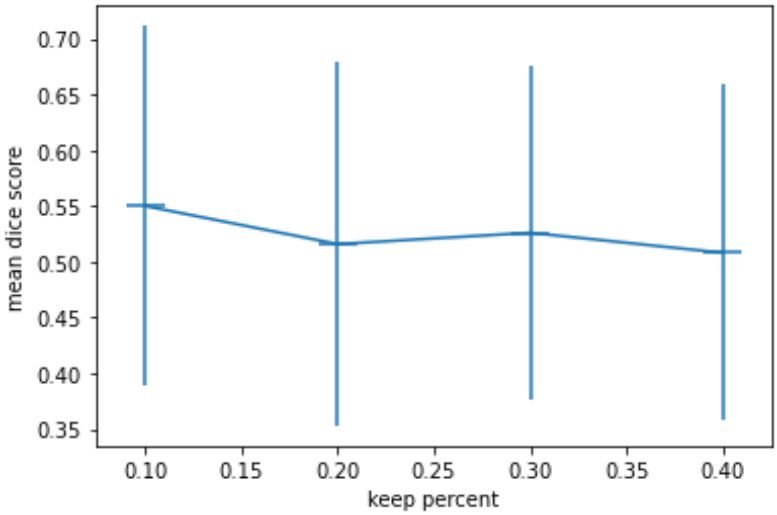

Figure 33: Graph of keep percent vs mean dice score

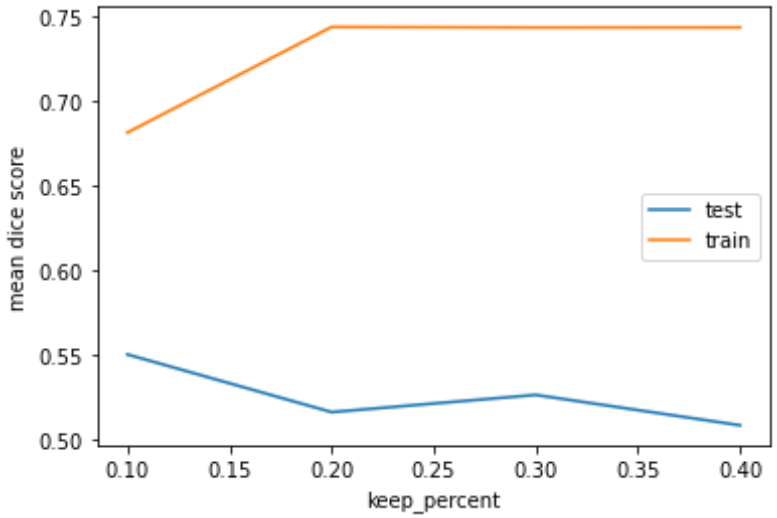

Figure 34: Graph comparing model performance on training and test data for keep percent experiment

Keep percent has little effect in the connected PCWN. In fact, there is a slight negative trend. Further experiments were planned but even with 16GB of RAM the computer would crash after 0.5 keep percent, as the size of the decomposition became too large. The linear inverse training time also increases significantly.

4.3.2.4 Resolution

Table 9: An experiment to show the effect of image resolution on connected PCWN

	Training data size	*count*	*Keep percent*	*resolution*	*Dice mean test*	*Dice mean train*	*Linear inverse train time*	*Image train time*
1	7049	3	0.1	64	**0.592 ± 0.172**	0.617 ± 0.177	110.172	57.156
2	7049	3	0.1	128	0.550 ± 0.161	0.681 ± 0.181	1128.817	75.411
3	7049	3	0.1	256	0.500 ± 0.164	**0.745 ± 0.193**	21648.167	166.170

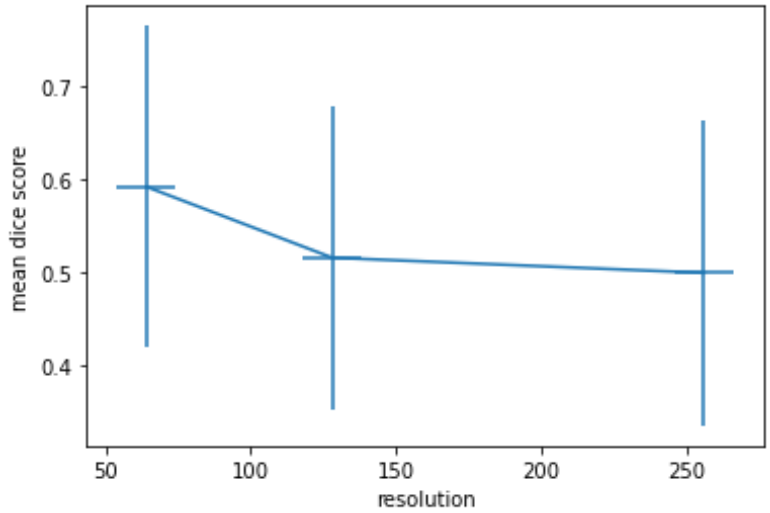

Figure 35: Graph of image resolution vs mean dice score

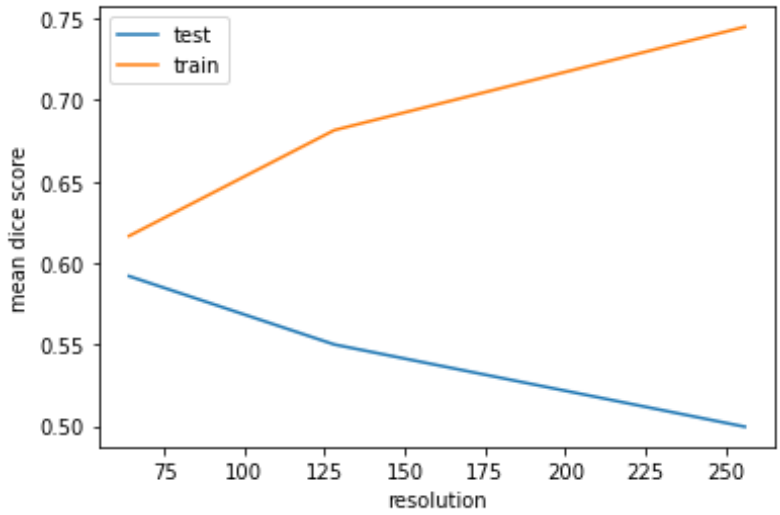

Figure 36: Graph comparing model performance on training and test data for image resolution experiment

Interestingly, the performance of the PCWN model showed completely different trends depending whether the model was run on the test or train set. For test data the model performed worse as the resolution increased, and for the train data model performance increased with image resolution.

4.3.2.5 Example Result

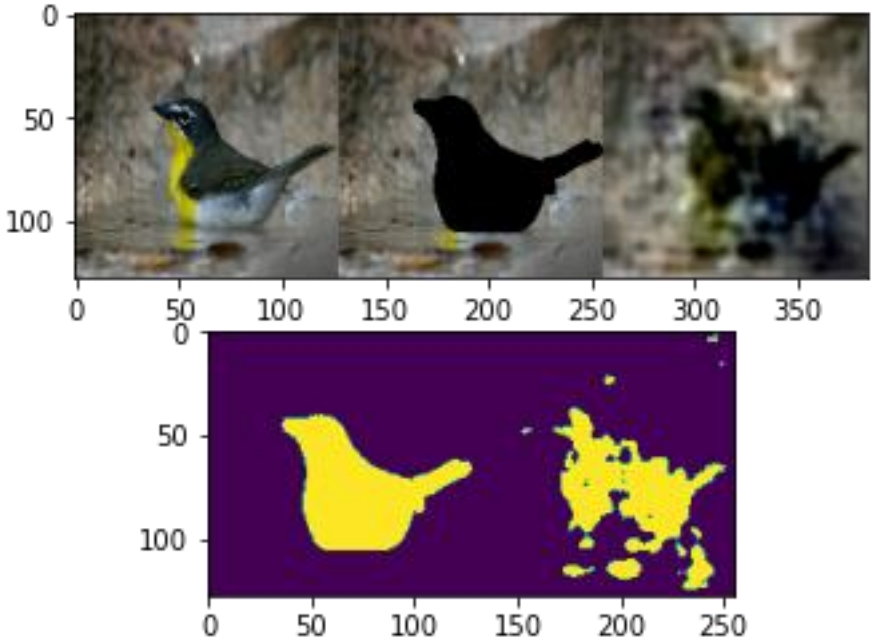

Figure 37: Example output of PCWN on test data, dice score = 0.72

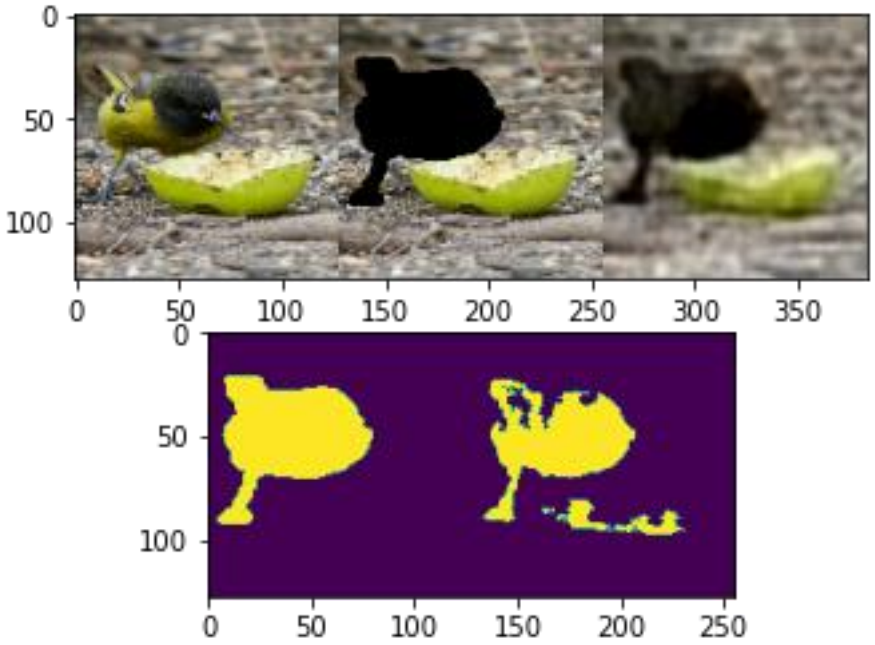

Figure 38: Example output of PCWN on train data, dice score = 0.81

These are some examples where PCWN has worked well, giving an acceptable mask for the bird. There is a clear gap in the reconstructions between the test and train instances, with the PCWN almost perfectly reconstructing the mask and the background when run on the training data but struggling with reconstructing the background with the test data. Much like the convolutional PCWN example images, these images are not representative of all the results. Many of the predicted segmentations for the test data were completely wrong.

4.4 Training Deep learning networks

Training the CNN and autoencoder took similar route. Once a suitable architecture had been built the models were trained until the average dice coefficient graph for the validation data started to plateau.
Both models were trained for 60-80 epochs (roughly 3 hours for the CNN, 4 hours for the U-net), with a keras callback saving the best model. The best model was the one that had the highest validation dice score.

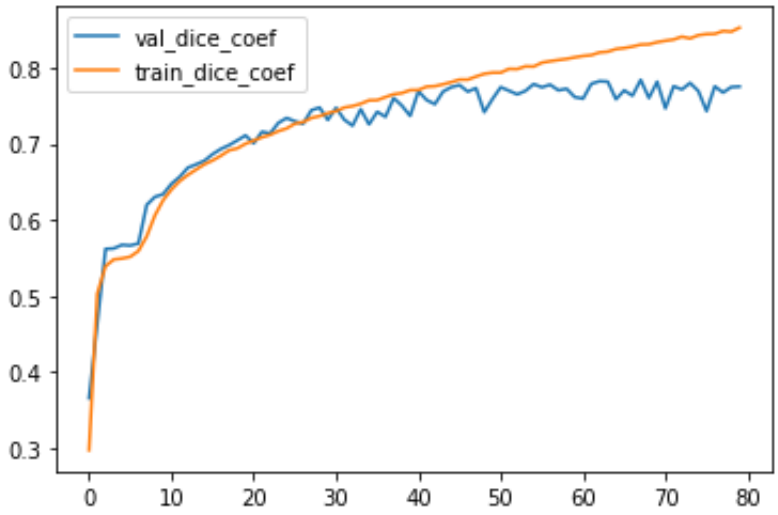

Figure 39: Graph comparing model performance on training and test data for U-net model

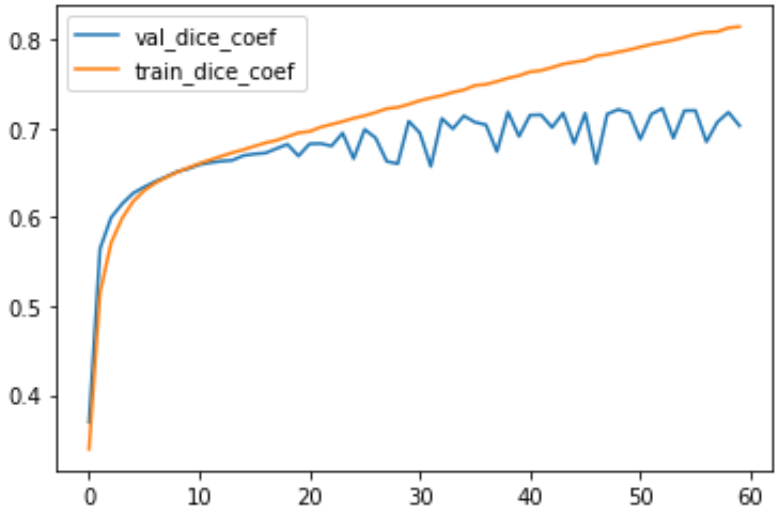

Figure 40: Graph comparing model performance on training and test data for CNN model

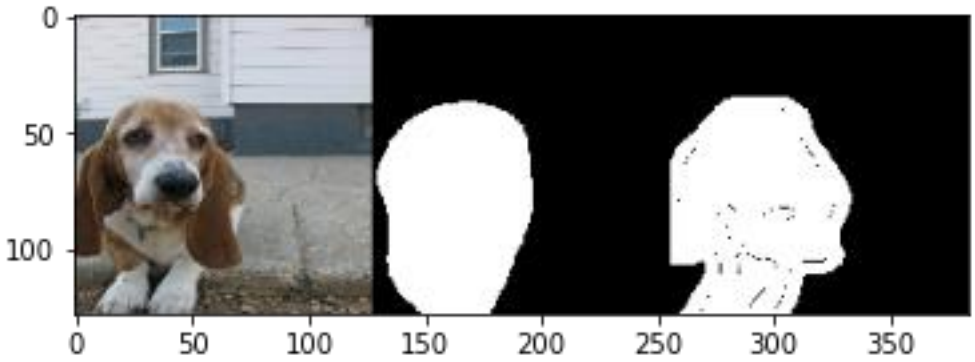

Figure 41: Example CNN output on test data, dice score = 0.90

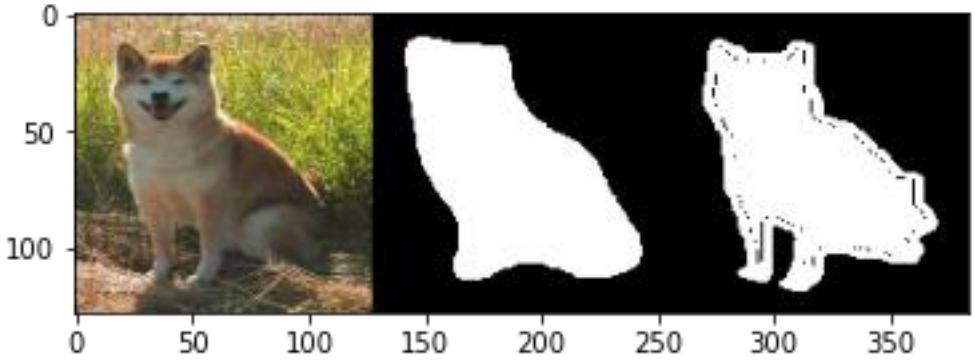

Figure 42: Example U-net output on test data, dice score = 0.89

The deep learning models both had excellent performances on the pet's dataset, and there was surprisingly little difference between the CNN and U-net approach. The models performed well on both the train and test datasets

4.5 Experimental Results: PCWN vs Deep Learning

All the models were trained using the full dataset from the Oxford Pets dataset and the California bird's dataset, with the images at a 128x128 resolution.

The PCWN settings used for both datasets were 4 layers and a 0.1 keep percent. For an unknown reason, any model that was larger than 4 layers would not work, even though in the previous experiments the exact same configurations had ran without a problem.

The recorded data is the performance of the models on a test set of size 300. The performance is measured using mean dice score and the standard deviation.

Table 10: Table comparing PCWN and deep learning approaches on test datasets

	Oxford Pets	***Caltech Birds***
U-net	**0.885 ± 0.067**	**0.698 ± 0.194**
Deep CNN	0.813 ± 0.107	0.690 ± 0.219
Convolutional PCWN	0.589 ± 0.178	0.335 ± 0.185
Connected PCWN	0.592 ± 0.196	0.209 ± 0.220

The results clearly indicate that U-net is the superior model, followed by the CNN approach. The PCWN models perform poorly on the test data, especially on the bird dataset. The convolutional model has the best performance on the pet's dataset, and the connected model performs best on the bird dataset.

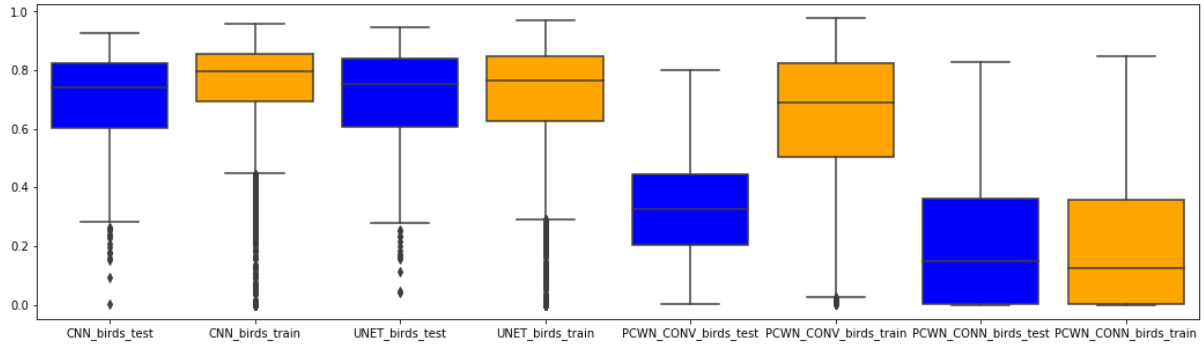

Figure 43: Box plot showing every models performance on test and train data from bird dataset

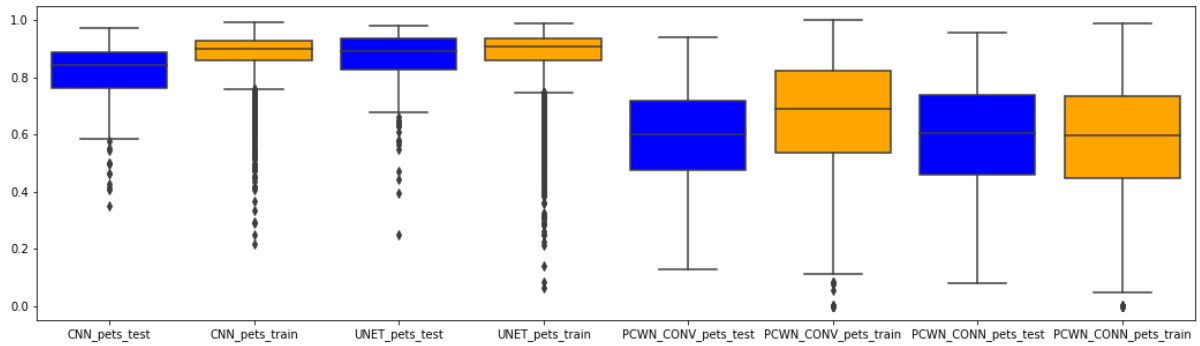

Figure 44: Box plot showing every models performance on test and train data from pet dataset

In the above plots, the first 4 box plots are deep learning models and the next 4 are the PCWN models, The deep learning models and the convolutional PCWN have reasonably consistent performances over the test and train datasets. In the bird dataset, the convolutional method has an excellent training data performance, but this does not translate to a good test data performance. The fully connected model has a poor performance for both the training and test splits. In the pet dataset, the results are much more promising. Both models show good results for both test splits, although the performance is not quite that of the deep learning models.

4.6 Conclusion

The PCWN segmentation technique clearly has some promise. The results from the pet dataset show that reasonable performance can be attained by this approach. There needs to be further investigation into why good results have been obtained for one dataset, yet for the other the results are much worse. The most likely cause is that there is a difference between segmenting pictures of pets and birds, maybe there's a larger variety of colours, or different shapes. There are many featural differences that could have caused a poor performance.

The experiments also indicate that PCWN parameters have to be carefully selected for every dataset. There was a large difference in the performance of the PCWN models between datasets, whereas the deep learning models were reasonably consistent. Furthermore, the PCWN model was very prone to crashing, or producing strange results, due to small changes in the four parameters tested (layer count, keep percent, training data size and image resolution).

5 Critical Evaluation

5.1 Background and Research

There was considerable research undertaken for this task, especially into wavelets, invertible networks, and deep learning techniques for segmentation. The project was more complex than anticipated and required specific design choices to make PCA based methods work for segmentation. The project was incredibly exploratory, taking 3 weeks for any concrete objectives for features to emerge. During this time there was a lot of work done in understanding PCWN architecture and segmentation networks. There was some pre-existing familiarity with PCA and deep learning techniques for computer vision, however there is a large difference between a basic understanding of the principles, to implementing them. Lots of spike work was done in this stage, taking the code from the paper, and debugging it was a relatively easy task, taking no more than a week.

The background research was possibly the favourite part of the project, it was incredibly invigorating to discover new developments in computer vision and the broader field of artificial intelligence. In hindsight too much time was spent on research, due to being overwhelmed with the possibilities and directions the project could take.

5.2 Design

The method for adapting segmentation masks to be amenable for PCA could probably be improved, or the PCA method could be changed to non—linear PCA. The approach taken to getting PCA to work for segmentation seemed non-optimal and unorthodox. With more research and better planning a better solution would have emerged, such as using non-linear PCA, or keeping the mask and image separate and applying a 2D gaussian distribution, with the mean at the centre of the mask. Not enough time was spent exploring pre-processing ideas such as this.

The linear least squares method was the appropriate algorithm for this report, as it was simple and a good general use approach. Given more time, it would have been interesting to explore different methods for mapping the image decomposition to the segmentation decomposition. Techniques such as polynomial regression, or even logistic regression could have been adapted for the task. However there was not enough time to implement and try these methods. A thorough analysis of the decompositions would have been incredibly insightful in assessing what model to use, and a highly recommended step for a future researcher of this project.

Image augmentation would be an interesting addition to the project. Rotating, flipping, and adding noise to the images to build a larger dataset of images would be beneficial to both PCWN models, which showed an increase in performance when more data is added.

The design proved to be an excellent first attempt at this problem, with lots of potential for future research. The report demonstrates that PCWN can be adapted for segmentation using the methods researched and devised in the design and analysis sections.

5.3 Implementation

The implementation also proved challenging. Once the code and paper were understood, getting the PCWN architecture to work was relatively easy. The main difficulty was writing the linear least squares method to map the image decomposition to the segmentation decomposition. This process was massively hampered by the hardware being used. Calculating the inverse was causing out of memory

errors or crashing python. There were also some issues around some of the linear algebra used. Matrix multiplication is non-commutative ($X^T X \neq X X^T$), and in the code there was confusion between which way round the matrices should have been. If the project was to be repeated, there would be more research into how linear least squares is derived before writing the code, to prevent confusion.

At times it seemed too much time was spent dealing with hardware issues and solving compatibility issues between the novel implementation and the code from the PCWN paper. However, this is the reality of working with software, so it came as no great surprise. If the project was to be done again, more concrete objectives early on would be very useful. A lot of time was spent deciding between directions to take the project, doing spike work and research for each option. If more concrete objectives had been set then there would be less time undertaking needless work, and more time focusing on producing the end product.

Ultimately, the main requirements of the research were met. The software produced could create PCWN models for segmentation, and these could be tested against deep learning models. Ideally, more work could have been done investigating changes to the architecture of PCWN, such as using dropout or another kind of regularisation. An investigation into adapting the network with non-linear components would have also been interesting.

5.4 Testing

It was very difficult to write a useful test. Agile approaches break down when it comes to notebooks as we can easily test any functionality through “execute and check”. In a way, there were many tests, the execute and check method was used countless times, mostly as sanity checks. The python files required some testing, but all the functionality of the files was used by the notebooks, where the tests could be run using execute and check again. Any further testing such as the use of external applications or dedicated test methods would have been contrived and a poor use of time.

5.5 Methodology

The methodology of feature driven development was helpful in guiding the development of software. This does not mean the approach was followed rigorously. The Kanban board was useful towards the beginning of the project, as it clearly presented the objectives ahead and assisted with planning and time management. This worked very well for a majority of the project.

Retrospectively, it would have been helpful to have more 1-on-1’s with the project supervisor, as many of the problems in the implementation were quickly resolved during these sessions. However, due to respect for the supervisors’ time, not many extra meetings were arranged.

Earlier in the project there was a considerable focus on using agile tools and adhering to agile methodologies. However this tailed off towards the end of the project, as the emphasis moved towards getting results and writing a report. In this sense, the methodology was effectively waterfall planning, as hardly any official documentation or report writing was done until the last 4 weeks. With this considered, a waterfall approach may have been more appropriate for this project.

For future researchers, an agile approach may be helpful in establishing the scope of the project, discovering what features are feasible and beneficial to the research output. After this initial discovery of objectives and sensible features to develop, a waterfall approach would be the optimal choice of methodology. This is because the features to implement are already decided and shouldn’t change over the course of the project.

5.6 Time Management

The time management for this project was well done and left plenty of time for reviewing and writing the report. By developing a routine of working on average around 20-30 hours a week, including Easter holidays, the project developed at a steady rate and accomplished all of the main objectives. There were a few days left at the end of the project which could have been used to implement more features, however this time seemed better spent checking the report and reviewing the quality of existing work. If some of the implementation issues had been solved earlier or had a more powerful laptop been used from the start, then more features could have been implemented.

5.7 Report

The report is satisfactory overall. Most of the sections are complete and well-structured, and clear enough to give a technical reader an understanding of the work accomplished. The thorough referencing provides further reading for users wishing to understand more of the project.

The project was not easy to write about in the style used, an academic paper style article would have been more appropriate. The report is certainly sufficient in terms of research and background, and the analysis of the problem is thorough enough to have dissected a complex technical problem and led to a satisfactory product.

5.8 Reflection and Future Work

Overall the project was a success, there were complex problems that needed to be addressed as PCWN is not an easy method to apply to segmentation. If the research was to be done again, there would not be an engineering element to the problem, as this seemed contrived and unnecessary for a research project, especially one undertaken as an individual effort. The research undertaken for the project was incredibly enjoyable, and an opportunity to explore many interesting fields.

PCWN is far from state of the art, for the task of image segmentation at least. Yet, there is potential to explore adaptions of the method, for image segmentation as well as other computer vision tasks. The implementation has left a lot of loose ends, applying only a fraction of the research completed. Given more time, a worthwhile endeavour would be exploring all the possible enhancements and applications for PCWN discussed in the background section. Some developments have already been discussed, such as combining U-net and PCWN, or using a different technique for dimensionality reduction. One of the problems with PCWN is the assumption of linear relationships in the data, which is only partially solved through use of non-linear activations. Further work into developing the method for non-linear data would be a good next step.

6 Bibliography

[1] B. Tiddeman and M. Ghahremani, "Principal Component Wavelet Networks for Solving Linear Inverse Problems," *Symmetry,* vol. 13, no. 6, p. 1083, 2021.

[2] R. Olaf, P. Fischer and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," in *International Conference on Medical image computing and computer-assisted intervention*, Cham, 2015.

[3] L. McInnes, J. Healy and J. Melville, "UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction," *ArXiv e-prints,* 2018.

[4] J. Bruna and S. Mallat, "Invariant Scattering Convolution Networks," *IEEE transactions on pattern analysis and machine intelligence,* vol. 35, no. 8, pp. 1872-1886, 2013.

[5] S. C. Yurtkulu, Y. H. Şahin and G. Unal, "Semantic Segmentation with Extended DeepLabv3 Architecture," in *2019 27th Signal Processing and Communications Applications Conference (SIU)*, 2019.

[6] S. Ghosh, et al, ""Understanding deep learning techniques for image segmentation."," *ACM Computing Surveys (CSUR),* vol. 52, no. 4, pp. 1-35, 2019.

[7] R. E. Bellman, Dynamic Programming, Princeton University Press, 1957.

[8] D. Rumelhart, G. Hinton and R. Williams, "Learning represntations by back-propagating errors," *Nature,* vol. 323, no. 9, pp. 523-536, 1986.

[9] K. Fukushima, "Neocognitron: A Self-organizing Neural Network Model," *Biological ,* vol. 36, pp. 193-202, 1980.

[10] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard and L. D. Jackel, "Backpropagation Applied to Handwritten Zip Code Recognition," AT&T Bell Laboratories, Holmdel, 1989.

[11] A. Haar, "Zur Theorie der orthogonalen Funktionensysteme," *Mathematische Annalen,* vol. 69, no. 3, pp. 331-371, 1910.

[12] P. Viola and M. Jones, "Rapid Object Detection using a Boosted Cascade of Simple," in *Computer Vision and Pattern Recognition*, 2001.

[13] E. Oyallon, S. Zagoruyko, G. Huang, N. Komodakis, S. Lacoste-Julien, M. Blaschko and E. Belilovsky, "Scattering networks for hybrid representation learning," *IEEE Trans. Pattern Anal. Mach. Intell.,* vol. 41, pp. 2208-2221, 2018.

[14] T. Angles and S. Mallat, "Generative networks as inverse problems with scattering transforms," in *ICLR*, 2018.

[15] V. Varatharasan, H.-S. Shin, A. Tsourdos and N. Colosimo, "Improving Learning Effectiveness For Object Detection and Classification in Cluttered Backgrounds," 2020.

[16] R. K. Srivastava, K. Greff and J. Schmidhuber, "Highway Networks," 2015.

[17] S. Hochreiter and S. Jurgen, "Long short-term memory," *Neural computation,* vol. 9, no. 8, pp. 1735-1780, 1997.

[18] K. He, X. Zhang, S. Ren and J. Sun, "Deep Residual Learning for Image Recognition," 2015.

[19] L.-C. Chen, G. Papandreou, I. Kokkinos, K. Murphy and A. L., "Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs," 2016.

[20] M. Holschneider, R. Kronland-Martinet, J. Morlet and P. Tchamitchian, "A real-time algorithm for signal analysis with the help of the wavelet transform," *Wavelets: Time-Frequency Methods and Phase Space,* pp. 289-297, 1989.

[21] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser and I. Polosukhin, "Attention is All You Need," *Advances in neural information processing systems,* vol. 30, 2017.

[22] H. Yan, C. Zhang and M. Wu, "Lawin Transformer: Improving Semantic Segmentation Transformer with Multi-Scale Representations via Large Window Attention," arXiv preprint arXiv:2201.01615, 2022.

[23] "Github Homepage," GitHub, Inc, [Online]. Available: http://github.com/. [Accessed 29 04 2022].

[24] "Jupyter Notebook Homepage," Jupyter, [Online]. Available: https://jupyter.org/. [Accessed 29 04 2022].

[25] Spyder, [Online]. Available: https://www.spyder-ide.org/. [Accessed 29 04 2022].

[26] "Anaconda Homepage," Anaconda, [Online]. Available: https://www.anaconda.com/. [Accessed 29 04 2022].

[27] P. Welinder, S. Branson, T. Mita, C. Wah, F. Schroff, S. Belongie and P. Perona, "Caltech-UCSD Birds 200," California Institute of Technology, 2010.

[28] O. M. Parkhi, A. Vedaldi, A. Zisserman and C. V. Jawahar, "Cats and Dogs," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2012.

[29] P. Jaccard, "THE DISTRIBUTION OF THE FLORA IN THE ALPINE ZONE," *New Phytologist,* vol. 11, no. 2, pp. 37-50, 1912.

[30] A. Rosebrock, "Intersection over Union (IoU) for object detection," pyimagesearch, 7 11 2016. [Online]. Available: https://pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/. [Accessed 28 04 2022].

[31] L. R. Dice, "Measures of the Amount of Ecologic Association Between Species," *Ecology,* vol. 26, no. 3, pp. 297-302, 1945.

[32] T. Sørensen, "A method of establishing groups of equal amplitude in plant sociology based on similarity of species and its application to analyses of the vegetation on Danish commons," *The Royal Danish Academy of Science and Letters,* vol. 5, no. 4, pp. 1-34, 1948.

[33] "F1/Dice-Score vs IOU," stackexchange, 12 1 2017. [Online]. Available: https://stats.stackexchange.com/questions/273537/f1-dice-score-vs-iou/276144#276144. [Accessed 25 4 2022].

[34] R. Penrose, "A generalized inverse for matrices," *Mathematical Proceedings of the Cambridge Philosophical Society,* vol. 51, pp. 406-413, 1955.

[35] E. Moore, "On the reciprocal of the general algebraic matrix," *Bulletin of the American Mathematical Society,* vol. 26, pp. 394-395, 1920.

7 Appendices

A. Third-Party Code and Libraries

Library	*Description*
keras	A submodule of tensorflow, used for building artificial neural networks using either a sequential or functional API
tensorflow	A module used for handling tensor operations and building neural networks
tensorflow_dataset	A module for loading well known datasets in an optimised format
numpy	A module for handling matrix operations
pandas	A module for handling data frame objects, tabular objects used for storing and processing multi-format data
matplotlib	A module for displaying data in a graphical format
seaborn	A module for displaying data in a prettier graphical format
os	A module for interacting with the system to allow for operations such as adding directories
sys	A module for allowing for CMD like commands for interacting with the underlying system
tqdm	A module used for monitoring the progress of iterations using a loading bar
MajorProject/Code/BaseCode directory	The original code from the PCWN authors, contains methods for running linear inverse experiments
pca_wavelet_utils	The module from the PCWN authors code that builds PCWN models

B. Code Samples

Least Squares: Fully Connected

Calculating matrix A and vector b for least squares. This method is used for the fully connected version, where the decompositions are flattened.

```
def connected_calculate_A_and_b(imghead, seghead, img_train,seg_train):
    imgflat = np.prod(imghead(next(iter(img_train))[0]).shape)
    segflat = np.prod(seghead(next(iter(seg_train))[0]).shape)
    end_shape = next(iter(seg_train))[0].shape
    n = 0.0

    xxt = np.zeros([imgflat])
    yxt = np.zeros([segflat])
    x = np.zeros([imgflat])
    y = np.zeros([segflat])

    bar = tqdm.notebook.tqdm(total = int(img_train.cardinality()))

    for item in iter(zip(img_train,seg_train)):

        bar.update(1)

        image = item[0][0]
        segmentation = item[1][0]

        imgdecom = imghead(image)
        segdecom = seghead(segmentation)

        mat = tf.reshape(imgdecom,[-1])
        segmat = tf.reshape(segdecom,[-1])

        cov = tf.matmul([mat],[mat],transpose_a=True)
        xxt += cov
        segcov = tf.matmul([mat],[segmat],transpose_a=True)
        yxt += segcov
        x+=mat
        y+=segmat
        n += 1

    xxt = xxt - tf.matmul([x],[x],transpose_a=True)/n
    yxt = yxt - tf.matmul([x],[y],transpose_a=True)/n
    inverse_xxt = tf.linalg.pinv(xxt)
    A = tf.linalg.matmul(inverse_xxt,yxt)
    b = (y - tf.linalg.matvec(A,x,transpose_a=True))/n
    return A,b
```

Least Squares: Fully Convolutional

Calculating matrix A and vector b for least squares. This method is used for the convolutional variant of PCWN, and reshapes the decompositions into 2D matrices.

```
def conv_calculate_A_and_b(imghead, seghead, img_train,seg_train):
    n = 0.0

    imdecom_shape = imghead(next(iter(img_train))[0]).shape
    img_channels = imdecom_shape[3] # shape_0
    imdecom_2d_shape = imdecom_shape[1]*imdecom_shape[2] #shape_1

    seg_channels = segdecom_shape[3] # shape_0
    segdecom_2d_shape = segdecom_shape[1]*segdecom_shape[2] #shape_1
    segdecom_shape = seghead(next(iter(seg_train))[0]).shape

    xxt = np.zeros([img_channels,img_channels])
    yxt = np.zeros([img_channels,seg_channels])
    x = np.ones([imdecom_2d_shape])
    x_m = np.zeros([img_channels])
    y = np.ones([segdecom_2d_shape])
    y_m = np.zeros([seg_channels])

    bar = tqdm.notebook.tqdm(total = int(img_train.cardinality()))

    for item in iter(zip(img_train,seg_train)):
        bar.update(1)
        image = item[0][0]
        segmentation = item[1][0]

        imgdecom = imghead(image)
        segdecom = seghead(segmentation)

        mat = tf.reshape(imgdecom,[-1,seg_channels])
        segmat = tf.reshape(segdecom,[-1,img_channels])

        cov = tf.tensordot(mat,mat,[0,0])
        xxt += cov
        #del cov

        segcov = tf.tensordot(mat,segmat,[0,0])
        yxt += segcov
        #del segcov

        x_m += tf.linalg.matvec(mat,x,transpose_a=True)
        y_m += tf.linalg.matvec(segmat,y,transpose_a=True)

        n += 1
    return A,b
```