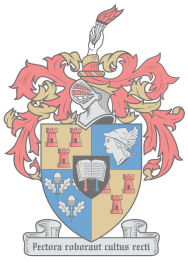
Investigating Fully Convolutional Networks for Bio-image Segmentation

by

Stiaan Wiehman



Thesis presented in partial fulfilment of the requirements for the degree of Master of Science in Computer Science in the Faculty of Science at Stellenbosch University

Supervisor: Dr. R.S. Kroon
Co-supervisor: Dr. H.A.C. De Villiers

March 2018

Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Date: March 2018 .....

Abstract

Investigating Fully Convolutional Networks for Bio-image Segmentation

S. Wiehman

Computer Science Division,
Department of Mathematical Sciences,
University of Stellenbosch,
Private Bag X1, Matieland 7602, South Africa.

Thesis: M.Sc. (Computer Science)

March 2018

Bio-image analysis is a useful tool for life science researchers with a wide variety of potential applications. A specific area of interest is applying semantic segmentation methods to bio-images, which is challenging due to the typically small data sets in this application area. Neural networks have shown great promise in both general image segmentation problems, as well as bio-image segmentation problems. A recently developed class of neural networks, Fully Convolutional Networks (FCNs), have shown state-of-the-art performance on various semantic segmentation tasks.

This thesis provides a thorough investigation into FCN architectures and their use in the semantic segmentation of two bio-image data sets. FCNs have been shown to provide improved performance over regular convolutional neural networks (CNNs). This work starts by comparing these two classes of networks by applying a CNN and three FCNs on the Broad Institute's *Caenorhabditis elegans* data set. We showed that the three FCNs performed better on the task of semantic segmentation and provide key insights into the difference in their performance.

Recent FCNs can be characterized by two main design aspects: the number of pooling steps in the architecture, and the presence or absence of skip connections. In existing literature, these hyperparameters are typically used without a detailed analysis of their effects. We build on this work by investigating these design aspects and determine their contribution towards the overall performance of the network. Using the recently presented U-net architecture and the accompanying nerve cell membrane data set, this investigation revealed

that: (1) increasing the depth of the network by adding additional pooling steps could improve performance up to a (hypothesized) domain-specific saturation point (assuming the inclusion of the necessary skip connections), and (2) each skip connection in the architecture appears to make a different contribution towards the behavior of the network, with some skip connections being more important than others. These findings could provide a better understanding on how to construct new FCN architectures for future applications.

We complete this investigation by exploring the possibility of performing end-to-end unsupervised learning as a pre-training technique, and test the resulting models on both fully labeled bio-image data and artificially created partially labeled bio-image data. We proposed a novel augmentation to FCN architectures which allows them to undergo end-to-end unsupervised pre-training. We showed that our unsupervised pre-training approach provides a significant reduction in the variance of the performance of the models. We then applied the supervised version and the pre-trained version of the U-net model on various amounts of partially labeled data, and found that the FCNs are capable of reaching competitive performance with as little as 0.2% of the original pixel labels.

The results generated in this thesis provide the foundation for further research into a more sophisticated unsupervised pre-training approach. Such an approach might reduce the need for fully annotated bio-image data, consequently reducing the time and financial resources required to perform the annotations.

Uittreksel

'n Ondersoek van Volledig Konvolusionele Netwerke vir Biobeeldsegmentasie

("Investigating Fully Convolutional Networks for Bio-image Segmentation")

S. Wiehman

Afdeling Rekenaarwetenskap,
Departement van Wiskundige Wetenskap,
Universiteit van Stellenbosch,
Privaatsak X1, Matieland 7602, Suid Afrika.

Tesis: M.Sc. (Rekenaarwetenskap)

Maart 2018

Biobeeldanalise is 'n handige tekniek middel vir navorsers in die lewenswetenskappe met 'n wye verskeidenheid van potensiële toepassings. 'n Spesifieke area van belangstelling is om semantiese segmentasiemetodes toe te pas op biobeelde, wat veral uitdagend is as gevolg van die tipies klein datastelle in hierdie toepassingsgebied. Neurale netwerke het besonderse belofte getoon in beide algemene beeldsegmentasieprobleme, sowel as biobeeldsegmentasieprobleme. 'n Onlangs ontwikkelde klas van neurale netwerke, Volledig Konvolusionele Netwerke (VKNe), het baanbrekerprestasie getoon op verskeie semantiese segmentasietake.

Hierdie tesis onderneem 'n deeglike ondersoek van VKN argitekture en die gebruik daarvan in die semantiese segmentering van twee biobeeld datastelle. VKNe het al verbeterde prestasie getoon oor gewone konvolusionele neurale netwerke (KNNe). Hierdie werk begin deur dié twee klasse van netwerke te vergelyk met die toepassing van 'n KNN en drie VKNe op die Broad Instituut se *Caenorhabditis elegans* datastel. Ons wys dat die drie VKNe beter presteer op hierdie semantiese segmentasietaak en verskaf belangrike insigte ten opsigte van die verskille in hul prestasies.

Onlangse VKNe kan gekarakteriseer word deur twee hoof ontwerpsaspekte: die aantal vernouingstappe in die argitektuur en die teenwoordigheid of afwesigheid van oorslaanverbindings. In bestaande literatuur word hierdie hiperparameters tipies gebruik sonder 'n gedetailleerde analise van hul effekte. Ons bou op hierdie werk deur hierdie ontwerpsaspekte en hul bydrae tot die

algehele prestasie van die netwerk te ondersoek. Met die gebruik van die U-net argitektuur en die meegaande senuweeselmembraan datastel, het hierdie ondersoek die volgende twee bevindinge aan die lig gebring: (1) die verdieping van die netwerk deur addisionele vernouingsstappe by te voeg kan prestasie verbeter tot 'n (vermoedelik) domein-spesifieke versadigingspunt (met die veronderstelling dat die nodige oorslaanverbindings teenwoordig is), en (2) elke oorslaanverbinding in die argitektuur lewer 'n unieke bydrae tot die algehele gedrag van die netwerk, met somige oorslaanverbindings meer belangrik as ander. Hierdie bevindinge kan 'n beter begrip verskaf oor hoe om nuwe VKN argitekture te bou vir toekomstige toepassings.

Ons voltooi hierdie ondersoek deur die moontlikheid van punt-tot-punt onbegeleide afrigtingte ondersoek as 'n vooraf-afrigtingstegniek, en toets die voortspruitende modelle op beide volledig geannoteerde biobeelde en kunsmatige gedeeltelik geannoteerde biobeelde. Ons ontwikkel 'n nuwe uitbreiding tot VKN argitekture wat hul toelaat om punt-tot-punt onbegeleide afrigting te ondergaan. Ons wys dat ons onbegeleide vooraf-afrigtingstegniek lei na 'n beduidende vermindering in die variansie van die prestasie van die modelle. Ons het toe beide die begeleide weergawe en die vooraf-afgerigte weergawe van die U-net model toegepas op verskeie vlakke van gedeeltelik geannoteerde data, en bevind dat die VKNe bereik byna baanbrekersprestasie met so min as 0.2% van die oorspronklike data etikette.

Die resultate bevat in hierdie tesis vorm 'n basis vir verdere ondersoek na 'n meer gesofistikeerde onbegeleide vooraf-afrigtingstegniek. So 'n tegniek kan die behoefte aan volledig geannoteerde biobeelde verminder en gevolglik ook die tyd en finansiële hulpbronne wat benodig word vir data annoteer verminder.

Acknowledgements

I would like to express my sincere gratitude to the following people and organizations:

- My (now) co-supervisor, Dr. H.A.C. de Villiers, for your encouraging emails, fun conversations and allowing me to freely explore the field of neural networks.
- My supervisor, Dr. R.S. Kroon, for your extensive guidance towards completing this thesis.
- My friends and family, for your help and support, and distracting me when I needed it most.
- The National Research Foundation (NRF), for their financial support during the course of this research.
- The CSIR-SU Centre for Artificial Intelligence Research, for their financial support during the course of this research.

Dedications

This thesis is dedicated to my mother.

Contents

List of Figures

List of Tables

Nomenclature

Acronyms and Abbreviations

BBBC	Broad Bioimage Benchmark Collection
C. elegans	*Caenorhabditis elegans*
CNN	Convolutional Neural Network
ConvSoftmax	convolutional layer using softmax activation
EM	Electron Microscopy
FCN	Fully Convolutional Network
FN	False Negative
FP	False Positive
GPU	Graphics Processing Unit
ISBI	International Symposium on Biomedical Imaging
MSE	Mean Squared Error
PReLU	Parametric Rectified Linear Unit
ReLU	Rectified Linear Unit
RMSProp	Root Mean Square Propagation
TN	True Negative
TP	True Positive

Chapter 1

Introduction

Bio-image analysis is a useful tool for researchers to quickly and accurately process the vast amount of image data that is obtained from high-throughput imaging technology. There are a wide variety of applications for bio-image analysis, some of which include being able to distinguish between different cell types, to identify specific pathogens or disease, or to track certain cellular organisms or organelles. These applications are especially useful in the healthcare sector and pharmaceutical industry. A review of these applications is provided in Peng (2008).

A specific area of interest is applying semantic segmentation methods on bio-images. By providing a semantic label for every pixel in an image, it is possible to not only determine the location of a desired biological object with high accuracy, but also to classify the object. Examples of this include differentiating between cell types in a tissue sample (Khan *et al.*, 2014) or determining whether a microscopic organism is alive or dead. One important challenge with bio-images that does not occur with the more general image data sets like ImageNet (Deng *et al.*, 2009) or the PASCAL VOC Challenge (Everingham *et al.*, 2015), is that the ground truth for bio-images needs to be annotated by an expert of the respective field or through use of specialized analytical labeling techniques which can be expensive and time consuming (Kraus *et al.*, 2016). This usually results in fairly small data sets (a couple of hundred images or less) with incomplete or partial ground truth.

Neural networks have shown great promise in semantic segmentation tasks, delivering state-of-the-art performance on a number of segmentation data sets such as PASCAL VOC (Long *et al.*, 2015), SIFT Flow (Pinheiro and Collobert, 2014) and Stanford Background (Pinheiro and Collobert, 2014; Socher *et al.*, 2011). A particular class of neural networks that are an elegant solution to performing semantic segmentation is the recently developed Fully Convolutional Networks (FCNs) (Long *et al.*, 2015).

The elegance of an FCN lies in its architecture: it is mainly composed of convolutional layers and contains no fully connected layers. The lack of fully connected layers allow the network to be applied to input images of

arbitrary size, provided that the input image is large enough to perform valid convolutions throughout the network. This gives FCNs the unique capability to produce full segmentation maps as output, rather than single pixel labels in the case of conventional Convolutional Neural Networks (CNNs). This capability allows FCNs to process an entire image in one pass, thus greatly reducing the image processing time compared to CNNs.

1.1 Problem Statement and Motivation

Over the years, neural networks have expanded from containing a few thousand parameters 30 years ago up to the hundreds of billions of parameters today. Even though the neural networks grew in size and sophistication over the years, not much is yet understood on how they perform the tasks they are trained to do. This is what is referred to when claiming that neural networks are 'black boxes', in that the function they calculate cannot be motivated or explained in a meaningful way. Furthermore, there is no consensus on the methodology of building a new neural network architecture for any specific task. The common approach is to use a proven architecture from one domain and apply it to a different domain, with minor architectural and parameter adjustments through trial and error. The same is true for FCNs, and given their recency, there is less work in the literature to fall back on.

FCNs have been shown to provide improved performance on semantic segmentation tasks compared to regular CNNs, but not much is yet understood on how to construct an FCN for a particular task. This thesis aims to investigate FCNs for the purpose of semantic segmentation on two bio-image data sets.

This work will compare the performance of multiple architectures belonging to the class of CNNs and FCNs on a single semantic segmentation task. This may provide further insight into the superiority of FCNs over CNNs when performing semantic segmentation. Recent FCNs can be characterized by two main design aspects: the number of pooling steps in the architecture, and the presence or absence of skip connections. In existing literature, these hyperparameters are typically provided without a detailed analysis of their effects. As such, we will also investigate these design aspects and quantify their contribution towards the overall performance of the network in various scenarios. Determining the possible impact of the main design aspects of FCN architectures would provide a better understanding on how to construct new architectures for future applications.

The unique capability of FCNs to produce entire segmentation maps also allows the exploration of earlier approaches that were previously restricted to individual layers. That is, unsupervised pre-training through input reconstruction, which is typically only used in regular and convolutional autoencoder layers. FCNs provide the opportunity to explore end-to-end unsupervised pre-

training which, if effective, could lead to improved performance especially in application domains where the availability of labeled data is generally limited (semi-supervised learning). Bio-image data sets are of particular interest, as the process of obtaining labeled data is time consuming and expensive, with the annotations performed by an expert in the respective field. Unsupervised pre-training for FCNs would reduce the need for fully annotated bio-image data sets, thus reducing the cost and time spent to annotate the data. We propose an approach to apply unsupervised pre-training to FCNs, and investigate its effectiveness on both fully annotated bio-image data and artificially created partially labeled bio-image data.

1.2 Objectives

The objectives of this study are:

- To compare the performance of FCNs against a conventional CNN for the task of semantic segmentation.
- To identify and analyze the main aspects of the FCN architecture and their contribution to the performance of the model.
- To apply unsupervised pre-training to FCNs, and consider its effectiveness.
- To investigate the effectiveness of using partially labeled bio-image data to train FCNs against using the fully annotated data.

1.3 Contributions

The work presented in this thesis made the following contributions:

- We showed that FCNs are capable of outperforming both the current processing pipeline and a CNN on the task of segmenting *Caenorhabditis elegans* worms in optical microscopy images. This work was presented as a peer-reviewed paper at the 2016 International Joint Conference on Neural Networks (Wiehman and De Villiers, 2016).
- We found that increasing the number of feature map resolutions in the FCN architecture (by adding additional pooling steps) leads to improved performance, given the inclusion of the necessary skip connections.
- We found that model performance starts to saturate as the number of feature map resolutions increased, but hypothesize that each application domain may have its own saturation point.

- We showed that each skip connection makes a specific contribution towards the overall performance of the network, and that some skip connections are more important than others depending on the application domain.
- We proposed a novel augmentation to FCN architectures that allow them to undergo end-to-end unsupervised pre-training, and showed that our approach provides a significant reduction in the standard deviation of the performance of trained models. This work was presented as a peer-reviewed paper at the 2016 conference of the Pattern Recognition Association of South Africa (Wiehman *et al.*, 2016).
- We showed that FCNs are capable of reaching near state-of-the-art performance on the nerve cell membrane data set when presented with only a small fraction of the original fully annotated ground truth.

1.4 Thesis Layout

This thesis is structured as follows: Chapter 2 provides the essential background information on artificial neural networks that is required for the understanding of the rest of this thesis. Chapter 3 introduces the two bio-image data sets that were used throughout this work. Chapter 4 presents a comparative study between a conventional CNN and three FCNs. Chapter 5 presents an analysis of the two main design aspects of FCN architectures and their contribution towards the overall model performance. Chapter 6 proposes a novel augmentation to FCNs to facilitate the use of end-to-end unsupervised learning and investigates its effectiveness. Chapter 7 investigates a more conventional semi-supervised setting by using partially labeled data to train the FCNs. Finally, Chapter 8 provides a summary of the work done in this thesis and suggests possible future work.

Chapter 2

Background

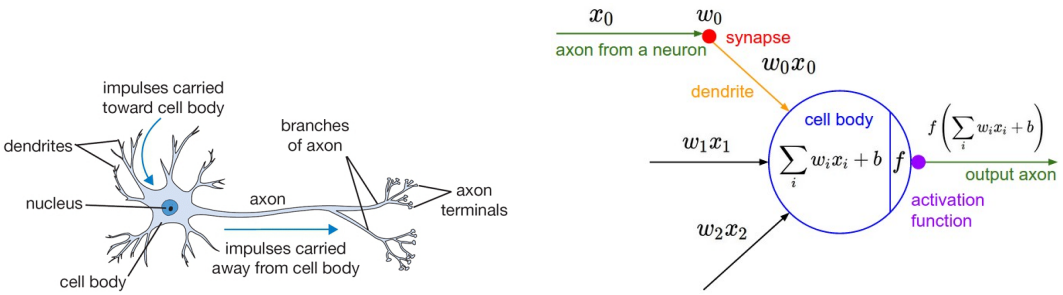
This chapter provides essential background information on artificial neural networks required for the rest of this thesis. Section 2.1 starts by defining artificial neurons and activation functions and some of the common types of neural network layers that can be found in the literature. Section 2.2 then provides an overview of how neural networks are typically trained. Section 2.3 presents four learning optimization techniques which are typically used to accelerate training. Section 2.4 describes how the neural networks used in this thesis were implemented. Lastly, Section 2.5 provides an overview of the two key types of architectures used in this work: convolutional neural networks and fully convolutional networks.

2.1 Artificial Neural Networks

Artificial neural networks are biologically inspired systems meant to simulate the computational approach of the human brain. These artificial networks typically consist of thousands of interconnected neurons (or nodes), often arranged as multiple layers stacked on top of each other. These neurons are the basic computational units of the network, allowing the network to perform a wide variety of complex tasks in various application areas. An overview of some successful applications of neural networks can be found in Kalogirou (2001).

2.1.1 The Artificial Neuron

An artificial neuron is a simplified representation of a biological neuron, in that it can accept and combine multiple inputs, perform some defined operation on the combined input and then produce an output. An illustration of a biological neuron, compared to its mathematical abstraction, is given in Figure 2.1. An input to the neuron, x_i (an axon from another neuron), is multiplied by an associated connection weight, w_i (synapse strength), and combined with all



(a) A biological neuron (b) Mathematical model

Figure 2.1: An illustration of a biological neuron compared to the mathematical model. Images extracted from Karpathy (2017*a*).

other neural inputs to form the neuron's pre-activation:

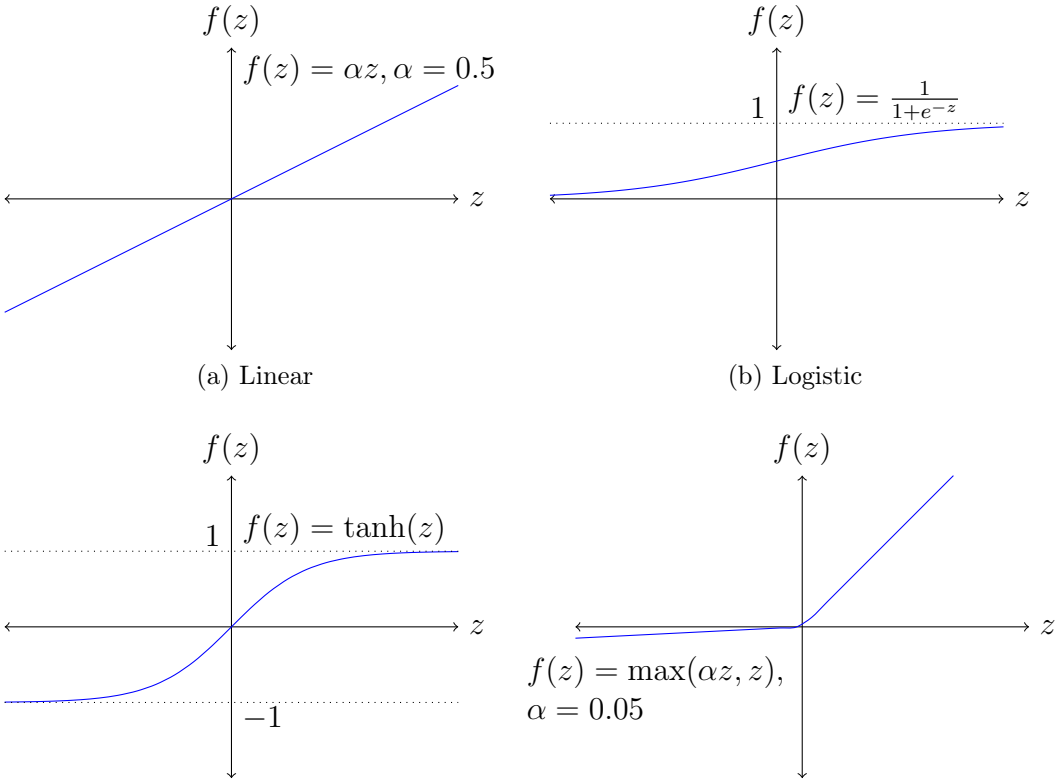
$$z = b + \sum_i w_i x_i, \tag{2.1}$$

where b represents the bias of the neuron. This bias term can also be interpreted as the spiking threshold of the neuron from a biological point of view. The neuron can then perform some predefined operation f (referred to as the activation function) on the pre-activation z, resulting in the output (or activation) $o = f(z)$.

It is general practice to choose an activation function which is non-linear, monotonically increasing and continuously differentiable (either by definition or by implementation) — some common examples are discussed in Section 2.1.2. Although linear neurons (or networks) are easier to understand and analyze, most real-world problems are non-linear and can only be solved when using non-linear networks (Zhang *et al.*, 1998; Chen and Billings, 1992). The basic problem is that a linear network collapses to a single layer via matrix multiplication. So, a multilayer linear network is no more expressive than a single layer linear network. The monotonic property of the activation function allows easier convergence during training towards an optimal solution. The differentiability of the activation function is required for training, which will be discussed in more detail in Section 2.2.

2.1.2 Activation Functions

In principle, any function can be used as the activation function of a neuron, as there are in general no restrictions placed on the type of function. There are a few commonly used functions that will be discussed in this section, some of which are depicted in Figure 2.2.



(a) Linear (b) Logistic

(c) Hyperbolic tangent (d) Leaky rectified linear unit

Figure 2.2: A visual representation of a few activation functions. The rectified linear unit and parametric rectified linear unit are not shown in this figure as they have a similar shape to the leaky rectified linear unit, with different values of α.

Linear

A linear activation function, as mentioned previously, is not particularly useful in neural networks when applied to non-linear problems. It is, however, necessary to define a linear activation function, since it will be used in the output layer of a network in Chapter 6. A linear activation function is defined as

$$f(z) = \alpha z, \tag{2.2}$$

with α being some constant. An example is shown in Figure 2.2a with $\alpha = 0.5$.

Logistic

The logistic function is a classic example of a sigmoid function. It is a real-valued function which maps the interval $(-\infty, \infty)$ onto the bounded range $(0, 1)$ (Figure 2.2b). It is defined by

$$f(z) = \frac{1}{1 + e^{-z}}, \tag{2.3}$$

and is typically used for the output layer in binary classification problems. They can be used in other areas of the network; however, care should be taken for deep networks, as the function starts to flatten when the value of z tends towards positive or negative infinity. This flattening is called the saturation of the function, and results in very small gradients which ultimately slow down learning (Glorot and Bengio, 2010). A common problem that can occur when using saturating functions, especially in deep networks, is the vanishing gradient problem (see Section 2.2.3 for more on the backpropagation algorithm and the vanishing/exploding gradient problems).

Hyperbolic Tangent

The hyperbolic tangent function is another example of a sigmoid function. It differs from the logistic function both in definition and in range, in that it maps the interval $(-\infty, \infty)$ onto the bounded range $(-1, 1)$ (Figure 2.2c), and is defined as

$$f(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}. \tag{2.4}$$

The hyperbolic tangent function is recommended over the normal logistic function, as it provides much stronger gradients when z is close to zero. It also has rotational symmetry about the origin, which has been shown to result in faster convergence than the non-symmetric logistic function when working with normalized input data (LeCun *et al.*, 1998). It should be noted that the hyperbolic tangent function can still become saturated (Glorot and Bengio, 2010; LeCun *et al.*, 1998).

Rectified Linear Units

The rectified linear unit (ReLU) activation function is one of the more popular functions to be used in recent years, since they allow networks to train substantially faster than the logistic and hyperbolic tangent functions (Krizhevsky *et al.*, 2012). This is due to the function having a piecewise constant gradient[1], either 0 or 1, depending on the value of z. The key characteristic of ReLUs is that they are unbounded on the positive domain, which means they cannot saturate and slow down training. The ReLU activation function is given by

$$f(z) = \begin{cases} z, & \text{if } z > 0 \\ 0, & \text{if } z \leq 0 \end{cases}, \tag{2.5}$$

a special case of the Leaky ReLU function given in Figure 2.2d with $\alpha = 0$ (also see below). The ReLU activation function does have a notable weakness in that it produces a zero gradient during training when $z < 0$. Any neuron

[1] Although the function has no mathematically defined gradient at $z = 0$, implementations of the function and variations of it typically include a self-defined gradient at $z = 0$.

that produces a zero gradient implies that the neuron is 'dead' with respect to training on that particular input. When considering a full neural network with multiple input examples, a neuron is only considered to be truly dead when all of the input examples lead to a pre-activation of $z < 0$. Considering the calculation of z, such a situation can only occur when either the weights are initialized incorrectly or a poor learning rate was chosen (large gradients cause large weight fluctuations that subsequently can push z too far into the negative).

Leaky Rectified Linear Units

A leaky rectified linear unit activation function is a variation of normal ReLUs that attempts to circumvent the zero gradient problem of conventional ReLUs. Instead of the function being zero for all $z < 0$, the pre-activation is multiplied with a small constant factor. It is defined as the function

$$f(z) = \begin{cases} z, & \text{if } z > 0 \\ \alpha z, & \text{if } z \leq 0 \end{cases}, \tag{2.6}$$

where α is a small positive constant. This choice of activation function implies that there is a small, but non-zero, gradient for negative pre-activations $z < 0$. Note that α can either be defined for the entire layer (see Section 2.1.3 for more on neural network layers), or for each neuron/feature map in the layer.

Parametric Rectified Linear Units

A parametric rectified linear unit (PReLU) generalizes the leaky ReLU concept, in that the small positive constant α now becomes a trainable parameter. They were introduced by He *et al.* (2015) as a generalization of regular ReLU, with α being trained simultaneously alongside the other parameters of the neural network. The PReLU activation function is given by

$$f(\alpha, z) = \begin{cases} z, & \text{if } z > 0 \\ \alpha z, & \text{if } z \leq 0 \end{cases}, \tag{2.7}$$

where α is now a trainable parameter instead of a constant factor. PReLUs are the activation function employed throughout this work, as they have led to improved performance on various image-based tasks (He *et al.*, 2015; Xu *et al.*, 2015). It was also shown to improve the performance of the U-net architecture, introduced in Chapter 5.

Softmax

The final activation function that will be discussed is the softmax function. The softmax function is typically used in the output layer, but unlike the logistic

function, it is not restricted to binary classification problems. To define the softmax function, consider an output layer containing N neurons (layers will be discussed in Section 2.1.3), for an N-class classification problem. The softmax function is then defined as

$$f(\mathbf{z})_n = \frac{e^{z_n}}{\sum_{k=1}^{N} e^{z_k}}, \quad \text{for } n = 1, \ldots, N, \tag{2.8}$$

where $f(\mathbf{z})_n$ is the activation of neuron n based on the N neural pre-activations given by $\mathbf{z}$. The output $f(\mathbf{z}) = \mathbf{o}$ produced by the layer now contains real values in the range $(0, 1)$ with $\sum_{k=1}^{N} o_k = 1$. Each entry, o_n, can be interpreted as the probability that the given input belongs to class n.

2.1.3 Layers

The concept of layers was briefly touched upon when discussing the softmax activation function. There are two main variations of layers: fully connected layers and convolutional layers. This section will first briefly discuss fully connected layers, before moving on to convolutional layers which are central to the work performed in this thesis.

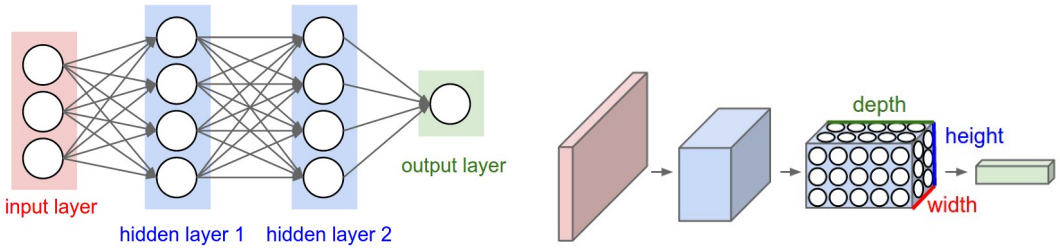
2.1.3.1 Fully Connected Layers

Fully Connected Layer (General Case) A fully connected layer (depicted in Figure 2.3a) can contain an arbitrary number of neurons, each connected to all of the M inputs to the layer (either samples from a data set or the output of neurons in the preceding layer). As such, each neuron in the layer receives the same input $\mathbf{x} = (x_0, \ldots, x_m, \ldots, x_M)$, but with their own weights $\mathbf{w}_n = (w_{0,n}, \ldots, w_{m,n}, \ldots, w_{M,n})$ and bias b_n. The weights of the neurons in the layer can be collected into a single $N \times M$ weight matrix $\mathbf{W}$ and an N-dimensional bias vector $\mathbf{b}$. Equation 2.1 can then be rewritten as an N-dimensional vector containing the pre-activations of all the neurons in the layer, defined as

$$\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b}. \tag{2.9}$$

Lastly, the output of the layer is then defined as the N-dimensional vector $\mathbf{o} = (o_0, \ldots, o_n, \ldots, o_N)$ with $o_n = f(z_n)$ (or $o_n = f(\mathbf{z})_n$ if f is the softmax function). In summary, a fully connected layer is essentially any layer where every neuron in the layer is connected to all of the layer inputs.

Softmax Layer The term 'softmax layer' is typically used to refer to the output layer of a neural network, specifically a fully connected layer which utilizes the softmax activation function over the neurons in the layer. The number of neurons in this layer depends on the number of classes present in the data set to which the network is applied.



(a) Fully connected layers (b) Convolutional layer

Figure 2.3: A representation of the spatial arrangement of the neurons in a fully connected layer compared to the neurons in a convolutional layer. Images extracted from Karpathy (2017*b*).

Autoencoder Layer The last variation of fully connected layers that will be discussed is the *autoencoder layer*. An autoencoder layer differs from regular layers, in that it is trained in two stages. When combined into a network architecture consisting of mostly autoencoder layers (called stacked autoencoders), each layer is first trained individually, before the full network is trained. This approach is often called greedy layer-wise unsupervised pre-training followed by supervised fine-tuning (Erhan *et al.*, 2010).

The difference between supervised and unsupervised learning, at least for neural networks, is that supervised learning requires labeled data that can be used to train the network through normal gradient descent (see Section 2.2 for more on how to train a neural network). Unsupervised learning does not use target labels; in this case, autoencoding essentially requires each layer in the network to be able to reconstruct the corresponding layer input, by having the layer learn a useful representation of its input. Hence, this unsupervised learning task can be referred to as input reconstruction, while the supervised learning task is commonly regression or classification (Erhan *et al.*, 2010).

A typical autoencoder is essentially a regular fully connected layer, with an additional trainable weight matrix $\mathbf{W}'$ and bias vector $\mathbf{b}'$. These additional trainable parameters are then used to transform the output of the layer, $\mathbf{o}$ (recall the output of a layer is defined as $\mathbf{o} = f(\mathbf{W}\mathbf{x}+\mathbf{b})$), into a reconstructed approximation of its input, $\mathbf{x}'$, given by

$$\mathbf{x}' = f(\mathbf{W}'\mathbf{o} + \mathbf{b}'). \tag{2.10}$$

The process of getting from the inputs $\mathbf{x}$ to the output $\mathbf{o}$ is called encoding, while the reconstruction phase in Equation 2.10 is referred to as decoding. A more in-depth discussion of autoencoders and stacked autoencoders can be found in Vincent *et al.* (2010). The task of input reconstruction can also be taken one step further, where the autoencoder layer is required to reconstruct the original input from a corrupted version of the input. Such layers are often referred to as *denoising autoencoders* (Vincent *et al.*, 2010; Erhan *et al.*, 2010).

2.1.3.2 Convolutional Layers

Convolutional Layers (General Case) Convolutional layers are similar to regular fully connected layers, in that they also have a trainable weight matrix and bias and perform the dot product to obtain the pre-activations followed by an activation function. The difference is that the connectivity of a convolutional layer to its input is restricted in such a way that it is especially useful for working with images. The material presented in this section was adapted from Karpathy (2017*b*).

The neurons in fully connected layers are each connected to every input to the layer, which makes it useful to get a global representation of the input space. This property does not work that well with images, however, since each pixel in the image counts as one or more input sources (depending on the number of color channels), with typical images containing thousands to millions of pixels. Any reasonable fully connected layer with this number of input sources will result in an impractical number of connection weights that need to be trained (and stored).

Convolutional layers circumvent this problem in that (a) a large portion of the weights are tied together and (b) by restricting each neuron to get information from a small localized region in the input rather than the entire image. One way to demonstrate how the neurons are arranged in a convolutional layer and the associated weight restrictions, is to use a three-dimensional representation, with a width, height and depth (Figure 2.3b). Depth in this instance, can also be referred to as the number of *feature maps*, with width and height being the size of each feature map.

A feature map is essentially a 2D arrangement of neurons, with the restriction that all of the neurons in the same feature map share one set of weights and a bias. This weight sharing between the neurons in a feature map is what is referred to as the weights being tied together, in that each neuron uses an identical copy of the weights and any change resulting from learning through any one neuron is reflected in all of the neurons in the feature map. The weight matrix associated with a given feature map can also be interpreted as a *filter*, which is slid over the input space to determine the activations of the neurons corresponding to the location in the input space. This sliding of the filters over the input space is mathematically equivalent to the convolution operation.

To better understand convolution, consider the following two finite, discrete, 1D functions f and g, where $supp(g) = \{-M, -M+1, \ldots, M-1, M\}$, then the convolution of these two functions is defined as

$$(f * g)(n) = \sum_{m=-M}^{M} f(n-m)g(m), \tag{2.11}$$

where n is the index into f where the convolution is performed. Consider the example in Figure 2.4, with f as an array of 6 numbers, and g an array of 3

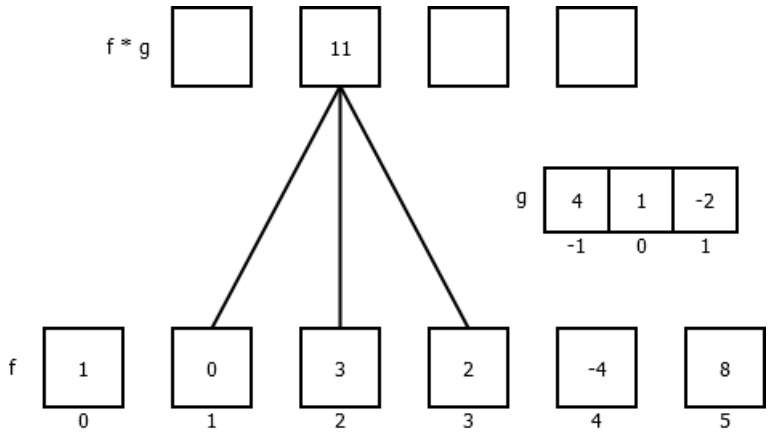


Figure 2.4: An example of the convolution of two finite, discrete, 1D functions f and g. The numbers beneath the boxes indicate the positions, n for function f and m for function g. The numbers inside the boxes are the actual values of the respective functions at those positions. The lines indicate which values of f are used to calculate the corresponding value in $f * g$. Note that in this top-down representation, the filter g needs to be flipped before calculating $f * g$.

numbers. To perform the convolution at $n = 2$,

$$\begin{aligned}(f * g)(2) &= \sum_{m=-1}^{1} f(2-m)g(m) \\ &= f(3)g(-1) + f(2)g(0) + f(1)g(1) \\ &= (2)(4) + (3)(1) + (0)(-2) \\ &= 11.\end{aligned}$$

Notice that the values from the function f change depending on where the convolution is performed, while the values from the function g always remain the same. Equation 2.11 can also be generalized to higher dimensions, for instance a convolution of two 3D functions

$$(f * g)(x, y, z) = \sum_{i=-I}^{I} \sum_{j=-J}^{J} \sum_{k=-K}^{K} f(x-i, y-j, z-k)g(i, j, k). \tag{2.12}$$

In order to bring the mathematical definition of a convolution back into the context of a convolutional layer, consider the function f to be the input to the layer (an image or the output of another convolutional layer) and the function g to be the weight matrix associated with an individual feature map. The pre-activation of a neuron at position i, j on feature map n can then be defined as

$$z_n(i, j) = (x * W_n)(i, j) + b_n, \tag{2.13}$$

where x and W_n are now 3-dimensional matrices corresponding to the width, height and depth of the input and filter shape, respectively.

For computational efficiency and following the notation in Theano (a neural network framework for Python), the shape of the weight matrix for a convolutional layer is defined to be (N, c, h, w), with N being the total number of feature maps in the layer, c the number of input channels (for example 1 for a gray scale image and 3 for an RGB image), h the height of the filter (number of rows) and w the width of the filter (number of columns). Note that c, the number of input channels, can also refer to the number of feature maps in the preceding convolutional layer that acts as input to the current layer. Also, the filters can take on any rectangular shape (h and w); however, square filters are the typical choice.

To illustrate the shape of the weight matrix and the correspondence between the neurons and the input, consider the example in Figure 2.5, which depicts a convolutional layer on the right with an RGB image of size 32×32 on the left as its input. The convolutional layer has five feature maps, indicated by the number of circles shown, with each circle representing a single neuron in each feature map that corresponds to the shown location in the input. Supposing these maps use square filters with a side length of 7, the resulting shape of the weight matrix for the convolutional layer in Figure 2.5 will then be $(5, 3, 7, 7)$. It should be noted that this figure does not show the filters used for each neuron, but only indicates the correspondence between the neurons and the location in the input space where the filters are applied.

The same example can be used to demonstrate how the weight restrictions in a convolutional layer affect the number of trainable weights compared to a fully connected layer with an equal number of neurons. As such, consider the input of size 32×32 with 3 color channels and a convolutional layer with a weight matrix of shape $(5, 3, 7, 7)$ and one bias per feature map (5 total). The convolutional layer contains $5 \times 3 \times 7 \times 7 + 5 = 740$ trainable parameters. Now consider a fully connected layer with an equal number of neurons as in the convolutional layer. To do this, we first need to determine the shape of the feature maps of the convolutional layer. Suppose the convolutional layer performs 'valid' convolutions (see Figure 2.6b and rest of this section), the number of neurons in the feature maps would be $5 \times 26 \times 26 = 3380$ neurons. A fully connected layer with this number of neurons would amount to more than 10 million trainable parameters ($32 \times 32 \times 3 \times 3380$).

Notice from the shape of the weight matrix for a convolutional layer that the number of trainable weights is not dependent on the resolution of the input, but instead only the number of feature maps in the layer, the number of input channels and the shape of the filters. Note that the shape of the filters refers to the height and width of the filters, while the shape of the weight matrix refers to the full four dimensional shape. Since the number of trainable parameters in a convolutional layer is not dependent on the size of the input, this means that the same example convolutional layer can be applied to an arbitrarily

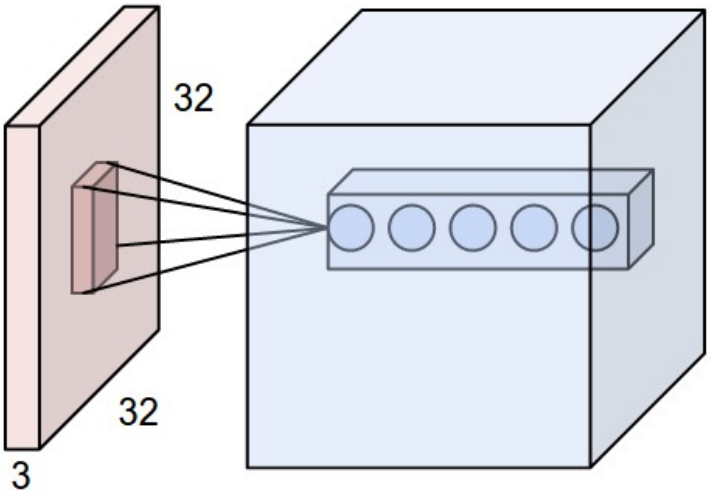
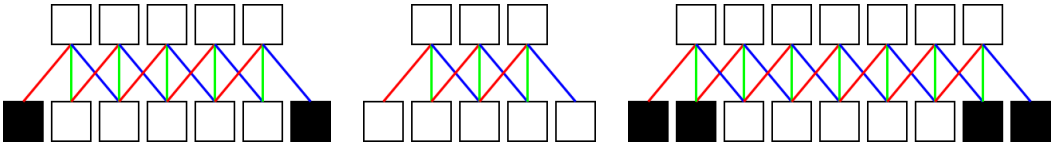


Figure 2.5: An illustration of a convolutional layer (right) with five feature maps, indicated by the number of circles, with an RGB image as input (left). Suppose these feature maps use square filters with a side length of 7, then the resulting shape of the weight matrix would be $(5, 3, 7, 7)$. Note that the image depicts the correspondence between the single neurons in each feature map and the location in the image where the filters are applied. Image extracted from Karpathy (2017*b*).



(a) 'same' (b) 'valid' (c) 'full'

Figure 2.6: The difference between various input padding configurations. The bottom layer of squares represents the input, with white squares the original input and black squares the border padding. The top layer of squares represents the resulting output after convolution using a filter of size 3 represented by the red, green and blue lines.

sized RGB image.

The final aspect needing discussion about regular convolutional layers is the hyperparameters that control them. Apart from choosing appropriately sized filters, there are three additional hyperparameters that determine the number of neurons in the layer. The first was briefly touched upon, which is the number of feature maps. It is important to note that the number of feature maps in the layer equals the number of filters that the layer will use.

The second hyperparameter is the input padding. Input padding is a technique that involves augmenting the resolution of the input to the layer with artificial values around the border. We follow the fairly standard convention of

using zeroes (this is the default in Theano), although other forms of padding, such as mirroring at the borders, are also sometimes used. There are three main configurations of input padding that can be found in neural network frameworks, namely: 'same', 'valid' and 'full':

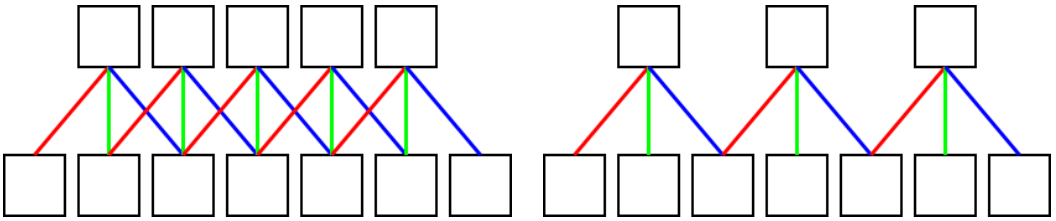
Consider the size of a square filter to be an odd number h. Then the 'same' configuration involves padding the input with a border of width $\frac{h-1}{2}$. When h is even, opposite borders will have different widths, one with width $\frac{h}{2}$ and the other with width $\frac{h}{2} - 1$. This choice of padding, in conjunction with a filter stride of 1 (the third hyperparameter discussed later in this section), will ensure that the size of the feature maps will be the same as the size of the original input. For example, Figure 2.6a shows an input of size 5 (bottom, white squares) being padded with a border of width 1 (black squares). The padded input is convolved with a filter of size 3, with the red, green and blue lines representing the different weights in the filter, and lines of identical color indicating that the weights are tied. This results in an output of size 5 (top), the same size as the original input.

The 'valid' configuration does not involve any padding at all, and results in feature maps that are slightly smaller than the original input (Figure 2.6b and Figure 2.7).

The 'full' configuration involves padding the input with a border of width $h - 1$, which increases the size of the resulting feature maps compared to the original input (Figure 2.6c). The 'full' configuration is typically not useful in practice, but used as an alternative to the 'same' configuration should the 'same' configuration not be available in the neural network framework. In such an occasion, the output after performing a 'full' convolution (convolution using the 'full' configuration) can simply be indexed to extract the output corresponding to the 'same' configuration.

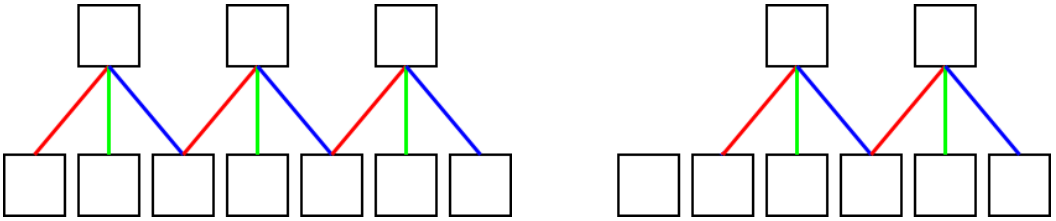
The final hyperparameter is the filter stride. The filter stride refers to the size of the step taken between two adjacent convolutions. Consider the simplified example of 'valid' convolutions in Figure 2.7, where both instances use an input of size 7 (the bottom layer of squares) and a filter of size 3. Figure 2.7a uses a filter stride of 1, as indicated by how the filter is moved to the left/right by one input square for each successive output neuron. Figure 2.7b on the other hand, uses a filter stride of 2, as indicated by how the weights of the filter are applied to every second input square for each successive output neuron. Figure 2.7b also shows the consequence of using a filter stride larger than 1: it significantly reduces the size of the output (beyond what is typical during 'valid' convolutions), and effectively downsamples the input (similar to a pooling layer which is discussed later in this section). Note that the filter stride in higher-dimensional inputs is typically the same in each dimension.

One important factor when considering a filter stride larger than 1 is whether the combination of filter size and stride fits with the dimensionality of the input. In other words, can the filters be slid across the input such that all of the input is used as part of a convolution. For instance, consider the



(a) Filter stride of 1 (b) Filter stride of 2

Figure 2.7: A depiction of how the filter stride hyperparameter affects the number of convolutions performed, and with respect to convolutional layers, the size of individual feature maps. Both instances use an input of size 7 (bottom layer of squares) and a filter of size 3 (red, green and blue line), with lines of identical color indicating the weights are tied. Figure 2.7a uses a filter stride of 1, indicated by the red/green/blue lines being moved one input square to the left/right for each successive output neuron. Figure 2.7b uses a filter stride of 2, indicated by the red/green/blue lines being moved two input squares to the left/right for each successive output neuron.



(a) A fitting example (b) A non-fitting example

Figure 2.8: An illustration of two scenarios that can occur when considering a filter stride larger than 1. Both instances use a filter of size 3 (red, green and blue line), with lines of identical color indicating the weights are tied, and a filter stride of 2. Figure 2.8a receives an input of size 7, which allow three convolutions to be performed. In this instance, the choice of filter size and stride fits the dimensionality of the input, as all the input squares form part of a convolution. Figure 2.8b receives an input of size 6, which only allows two convolutions to be performed. In this instance, the choice of filter size and stride does not fit the dimensionality of the input, as there is one input square that is not used.

convolutional layer in Figure 2.8, which uses a filter size of 3 and a filter stride of 2. In Figure 2.8a, the layer receives an input of size 7, which allows three convolutions to be performed whereby all of the input is used. In Figure 2.8b, the layer receives an input of size 6, which only allows two convolutions to be performed, resulting in one input square that is not used. Note that this scenario can also occur in pooling layers (discussed later in this section). Although a filter stride of 2 in a convolutional layer would downsample its input similar to a pooling layer, the stride does change where the convolutional filters are applied. This means that a different set of filters would be learned in a convolutional layer with a filter stride of 2 than in a convolutional layer with a filter stride of 1 and an attached pooling layer.

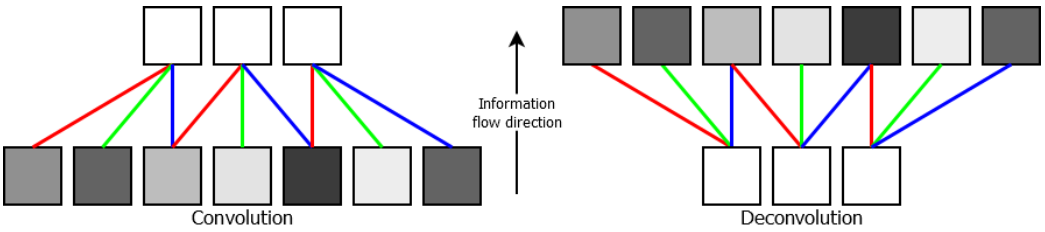
ConvSoftmax Layer The convolutional softmax layer, so named to be distinguished from a regular fully connected softmax layer, is a regular convolutional layer which utilizes the softmax activation function. It is defined slightly differently from the regular softmax function (see Equation 2.8), in that it is no longer calculated over all of the neurons in the layer, but over all the corresponding neurons in the feature maps: Consider an $X \times Y \times N$ arrangement of neural activations ($\mathbf{Z}$) in a convolutional layer, where $X \times Y$ is the size of each feature map and N is the number of feature maps corresponding to an N-class classification problem. The resulting output at position i, j for class n is then defined as

$$f(\mathbf{Z})_n^{i,j} = \frac{e^{z_n^{i,j}}}{\sum_{k=1}^{N} e^{z_k^{i,j}}}, \tag{2.14}$$

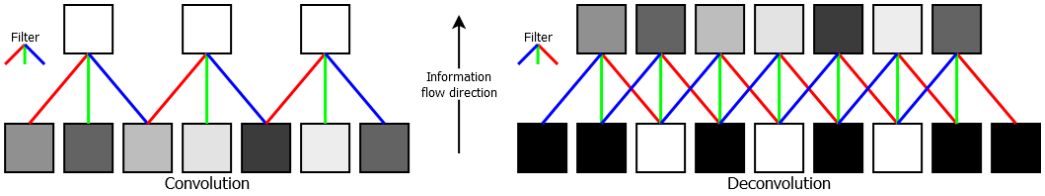
for $1 \leq i \leq X$, $1 \leq j \leq Y$ and $1 \leq n \leq N$. Each feature map can then be interpreted as a heat map (referred to as a segmentation map), indicating the affinity of the corresponding regions in the input to a specific class. Note that it is not each feature map that represents a probability distribution, but instead the values for each location i, j across all of the feature maps.

Deconvolutional Layers There are two directions in which values can be propagated through a neural network layer, the forward pass and the backward pass: Consider a convolutional layer, then the forward pass operation involves propagating the values of the layer inputs towards the output of the layer — this is typically used in both the training phase and during deployment in order to produce an output for input data (prediction). The backward pass operation involves propagating values from the output of the layer towards the layer inputs — this is typically used to propagate the gradients back through the network (see Section 2.2.3 on backpropagation).

First, it is important to note that the reverse operation of convolution, is in itself also convolution, just with spatially-flipped filters. A spatially-flipped filter is simply the original filter flipped along each spatial dimension (height and width). For instance, consider the 3×3 filter given below, and first flipping



(a) Direct deconvolution



(b) Indirect deconvolution

Figure 2.9: A 1D representation of the forward pass operation in a convolutional layer compared to the forward pass operation in a deconvolutional layer, with both layers utilizing a filter stride of 2. Figure 2.9a illustrates the direct route in performing deconvolution, utilizing the backward pass function which is built into the neural network framework. Should such a function not exist, Figure 2.9b illustrates how to manually perform deconvolution by convolving the padded input with spatially-flipped filters. It is important to note that the same colored connections represent the same weight (spatially), but not the same value (i.e. it does not represent numerical equivalence).

the filter horizontally, followed by flipping the filter vertically.

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \xrightarrow[\text{flip}]{\text{Horizontal}} \begin{bmatrix} 3 & 2 & 1 \\ 6 & 5 & 4 \\ 9 & 8 & 7 \end{bmatrix} \xrightarrow[\text{flip}]{\text{Vertical}} \begin{bmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{bmatrix}$$

The resulting filter is then what is referred to as a spatially-flipped version of the original filter.

Before illustrating the concept of performing deconvolution (also referred to as transposed convolution) using the regular convolution operation, it is important to note that the same colored connections in Figure 2.9 represent the same weight (spatially), but not the same numerical value. For instance, in Figure 2.9b, during the convolution operation, the leftmost input square is connected to the leftmost feature map square via the red connection. Notice in the same figure that during the deconvolution operation, the leftmost (white) feature map square (bottom row) is connected to the leftmost reconstructed input square (top layer) also via the red connection.

With this in mind, Figure 2.9b illustrates the concept of performing deconvolution as regular convolutions using spatially-flipped filters. In this instance, the filter stride does not control how the filters should be moved over the input, but instead how the input should be padded to get the same effect as in

Figure 2.9a. For example, a filter stride of 2 means the input squares must be separated by one padded square (in general, a filter stride of s means the input squares must be separated by $s-1$ padded squares). In addition to the padding between input squares, the input must also be padded using the 'full' configuration. Regular convolution, using spatially-flipped filters, can then be performed to produce an output.

The backward pass operation of a convolutional layer was originally only employed to backpropagate the errors during training until the recent advent of deconvolutional layers in Long *et al.* (2015). The deconvolutional layers in Long *et al.* (2015) were used as a means to upscale the output of the prediction layer to the size of the original input of the network. Although Long *et al.* (2015) did not specify a selection rule for the size of the filters, the models they created showed square filters of size $2s$ for a filter stride of s.

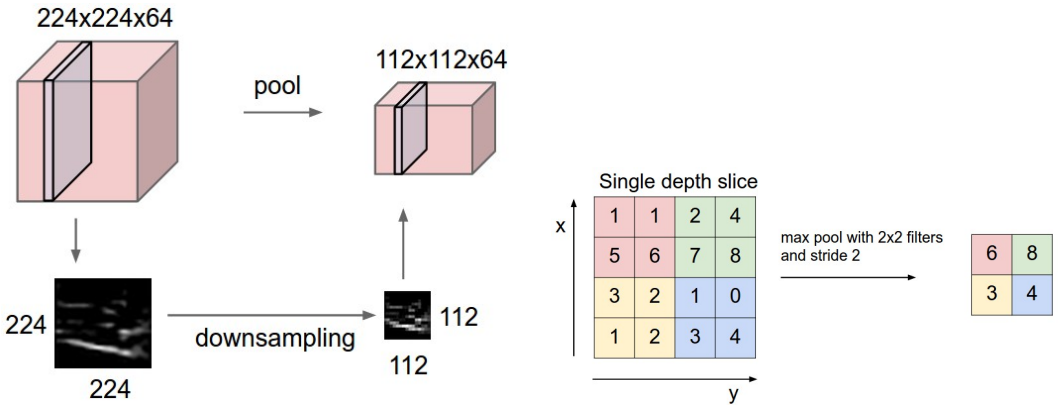
In summary, a deconvolutional layer performs the backward pass operation of regular convolutional layers as the forward pass operation. This means that, where a regular convolutional layer, with a filter stride of 2, would downsample the input it receives (reducing feature map size), a deconvolutional layer with the same stride will do the opposite, i.e. it will upsample the received input to yield larger feature maps.

Convolutional Autoencoder Layers A convolutional autoencoder is similar to a regular autoencoder (see Section 2.1.3.1), in that it also has two sets of filters and biases, one for encoding and one for decoding. They were first introduced in Masci *et al.* (2011), where the regular convolution operation was performed for encoding and reverse convolution used for decoding (the term deconvolution was not yet employed within the scope of neural networks at this time — it was first introduced in Long *et al.* (2015)).

Convolutional autoencoders are much more dependent on strict regularization rules (see Section 2.2.4), since the easiest solution to input reconstruction for convolutional layers is simply to use an identity filter. As such, convolutional autoencoders typically include a pooling operation (see below) before the reconstruction phase, creating a bottleneck and encouraging a more meaningful encoding of input feature maps.[2] Similar to regular autoencoders, a denoising component can also be added to the input reconstruction task (Masci *et al.*, 2011). Also like regular autoencoders, they can be stacked to create an architecture equivalent to a CNN.

Pooling Layers The final layer type we discuss is pooling layers. Although they do not perform a conventional convolutional operation like the other layers in this section, they are typically only used in conjunction with convolutional layers, hence their inclusion in this section. A pooling layer, sometimes called

[2]In regular autoencoders, this bottleneck is created by simply having fewer neurons in the layer than the number of inputs to the layer.



(a) Spatial representation (b) Numerical representation

Figure 2.10: A spatial and numerical representation of the operation performed by a max-pooling layer. Images extracted from Karpathy (2017*b*).

a downsampling layer, is used to reduce the size of its input to an extent depending on the hyperparameters that were chosen for the layer.

Similar to a convolutional layer, the pooling layer also has a filter, but not a bias, and differs from regular convolutional layers in that the filter does not contain trainable weights (either fixed weights or no weights at all). Instead, a function is applied over the input values within the scope of the filter, as it is slid over the layer input with a given filter stride. A popular choice of function for pooling layers is the max operation; however, other functions can also be used such as an averaging function (which is a fixed convolution).

Consider the example pooling layer in Figure 2.10 that has 2×2 filters and a stride of 2, the most commonly used hyperparameters in practice. Given the input to the pooling layer with shape $X \times Y \times N$ (this can either be an image or the output of a convolutional layer), a pooling layer with this choice of hyperparameters will reduce the size of the layer input by 75%, i.e. from $X \times Y \times N$ to $\frac{X}{2} \times \frac{Y}{2} \times N$.[3] Similar downsampling can be achieved by using a convolutional layer with a filter stride of 2, in which case the filters of the layer are trainable.

2.2 Neural Network Training

Given the above background on neurons, activation functions and the structured collections of neurons known as layers, we now have the machinery to construct a neural network. A typical neural network can be built by stacking multiple layers on top of each other, where the output of one layer becomes the input to the next. This creates a chain of computations, starting from the

[3]This assumes that both X and Y are even, as illustrated in Figure 2.10. If this is not the case, the choice of filter stride could cause some layer inputs to not be used.

input of the network and ending at the output of the last layer in the network. However, in order to understand how one trains the weights in such a network, a few additional concepts are necessary.

First, Section 2.2.1 defines a loss function and presents a few that are commonly used. Section 2.2.2 then motivates how the gradient of the loss function with respect to each weight can be utilized to train the network, after which Section 2.2.3 demonstrates how to calculate these gradients. Finally, Section 2.2.4 illustrates a few regularization techniques which can be used to mitigate overfitting.

2.2.1 Loss Function

The loss function, also known as the objective function, plays an important role for the training of any neural network. The loss function can also be considered to be a metric evaluating how well the network performs its task, with lower values corresponding to better performance on the task. There are three loss functions that were used in this work, which will be discussed in this section.

Mean Squared Error

The mean squared error (MSE) calculates the squared difference between the output of the network and the target labels, and averages over the number of labels. Consider presenting M training examples to a neural network with an output layer containing a single neuron. For each training example, which is associated with the target label y_i, the neural network produces the output o_i. The mean squared error is then defined as

$$L = \frac{1}{M}\sum_{i=1}^{M}(y_i - o_i)^2. \tag{2.15}$$

The MSE formed a part of the loss function for the neural network in Chapter 6, and was used for the autoencoder stage during training. As such, Equation 2.15 needs to be generalized to the 2D case to evaluate the difference between an input image and a reconstructed image. Consider the output $\mathbf{o}$ produced by a convolutional layer with a single feature map of size $X \times Y$, and the corresponding region in the original grayscale input of the same size, $\mathbf{I}$. The equation for the MSE then becomes

$$L = \frac{1}{XY}\sum_{i=1}^{X}\sum_{j=1}^{Y}(I_{i,j} - o_{i,j})^2, \tag{2.16}$$

where each pixel in $\mathbf{I}$ is used as the target label.

Cross-Entropy Error

The cross-entropy loss function is specifically used for probability target labels $0 \leq y_i \leq 1$. As such, it is typically used as a loss function for neural networks which use either a logistic (binary classification) or a softmax activation function on the output layer, both of which produce an output $0 < o_i < 1$.

Given a training example with its corresponding N-dimensional vector of probabilities $\mathbf{y}$, and the output probabilities $\mathbf{o}$ produced by a softmax layer, the cross-entropy loss for a single example is then given by

$$L = -\sum_{i=1}^{N} y_i \log(o_i). \tag{2.17}$$

This can be extended to any number of examples, summing over each and averaging over the total number of examples. Note that the `log` function requires that $o_i > 0$, which is satisfied by both the logistic and the softmax functions.

As with the MSE, the cross-entropy loss function can be used in the 2D case, on segmentation maps produced by the ConvSoftmax layer. In this instance, not only do the number of images count towards the total number of examples, but also every labeled pixel in each image. Note that this does not require all the pixels in any given image to be labeled; partially labeled images can be used as well, where unlabeled pixels are simply excluded from the loss function. This variation of the cross-entropy loss is mainly used for the networks in Chapter 4.

Boosted Cross-Entropy Error

The boosted cross-entropy error is a variation of regular cross-entropy that was introduced by Huang *et al.* (2014). It was inspired by the notion that people learn more from difficult problems in which they are more likely to make mistakes, and is defined by adding a weighting factor $(1-o_i)^\alpha$ to Equation 2.17, where α can be used to control the emphasis on difficult examples through the gradients of the resulting loss function (more on the gradients and their use to train neural networks in Section 2.2.2 and Section 2.2.3). A difficult example in this instance is simply any example that the network predicts incorrectly, i.e. $y_i = 1, o_i \ll 1$. The boosted cross-entropy loss function is given by

$$L_\alpha = -\sum_{i=1}^{N} y_i (1 - o_i)^\alpha \log(o_i). \tag{2.18}$$

More specifically, the weighting factor $(1 - o_i)^\alpha$ places more emphasis on incorrectly predicted examples, and less on correctly predicted examples. As mentioned previously, this is done through the partial derivatives of the loss function with respect to the pre-activations $\mathbf{z}$ of the neurons in the output

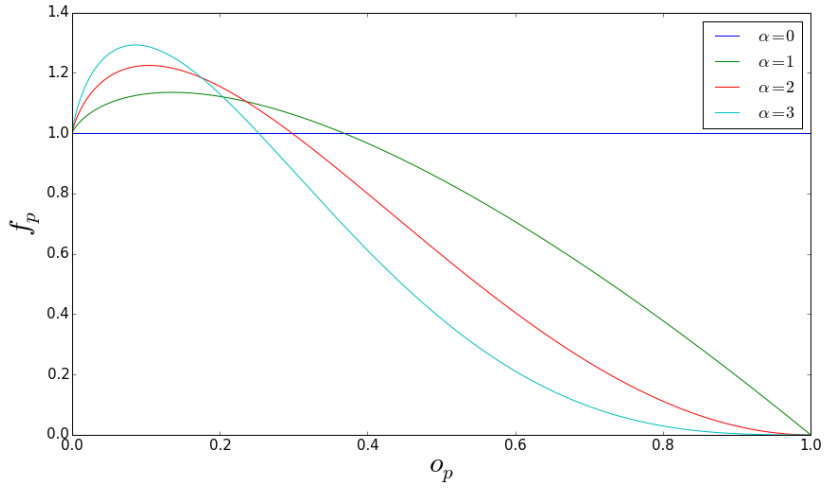


Figure 2.11: The scaling factor f_p for various values of α.

layer (assuming an output layer with softmax activation), which can be shown to be

$$\frac{\partial L_\alpha}{\partial z_i} = -y_i f_i + o_i \sum_{k=1}^{N} y_k f_k, \tag{2.19}$$

where $f_i = (1-o_i)^{\alpha-1}(1-o_i-\alpha o_i \log(o_i))$. The gradient $\frac{\partial L_\alpha}{\partial z_i}$ can be simplified further, with the assumption that $\mathbf{y}$ is a one-hot vector (a vector containing one 1 for the target class and the rest 0's) and that the 1 is at position p, to

$$\frac{\partial L_\alpha}{\partial z_i} = f_p(o_i - y_i), \tag{2.20}$$

where f_p is now the scaling factor for the original partial derivative of Equation 2.17, which can be shown to be $\frac{\partial L}{\partial z_i} = o_i - y_i$. Figure 2.11 shows the curve of f_p with respect to o_p, for various values of α. Notice in Figure 2.11 that the value of f_p is much larger when o_p is close to zero (which indicates an incorrect prediction) than when o_p is close to one.

The boosted cross-entropy loss function was considered for the work done in this thesis on the nerve cell membrane data set (Chapter 3.2), in part because it has shown improvement over neural networks trained with regular cross-entropy (Huang *et al.*, 2014), but also because the remaining errors that are made by approaches on this data set are likely attributable to either difficult to detect membranes or infrequent occurrences of a particular shape of membrane.

2.2.2 Gradient Descent

There are a number of approaches that can be used to train neural networks, each with their own advantages and disadvantages. Some of these include

particle swarm optimization (Meissner *et al.*, 2006), genetic algorithms (Leung *et al.*, 2003) and gradient descent (Rumelhart *et al.*, 1985; LeCun *et al.*, 1998). This thesis focuses on gradient descent as it is the most popular approach.

The idea of gradient descent relies on the fact that the loss function, viewed as a function of all the parameters of the neural network, specifies a high dimensional error surface. Assuming that the parameters of the neural network are randomly initialized (and not from a pre-trained network), training will start at a random location on this error surface. At any point on the error surface, the gradient can be computed with respect to the parameters of the network, by calculating the partial derivatives of the loss function with respect to the individual parameters.[4] The gradient (vector of partial derivatives) obtained indicates the direction in which the parameter vector should be changed to maximize the loss function. Since the goal of supervised learning is to minimize the loss function, the parameter values should be changed in the opposite direction of the gradient. The repeated application of this rule with suitable step sizes should eventually lead to a local minimum of the loss function. This step-wise minimization using the gradients is referred to as gradient descent (Rumelhart *et al.*, 1985; LeCun *et al.*, 1998).

There are various ways in which the gradients can be approximated, including batch gradient descent, stochastic gradient descent and mini-batch gradient descent (LeCun *et al.*, 1998).

Batch gradient descent involves evaluating the neural network for all the examples in the training set, calculating and collecting the gradient vectors in the process. When a complete pass through the entire training set has been performed, the average gradient is then used to update the network parameters.

Stochastic gradient descent differs from batch gradient descent, in that a single (random) example is chosen from the training set, for which the gradient can be calculated (considered as an estimation of the true gradient) and the parameters updated accordingly (note no averaging).

Lastly, mini-batch gradient descent represents a compromise between batch gradient descent and stochastic gradient descent, in that the gradients are averaged over a (random) subset of the training data.

To demonstrate the weight updates mathematically, consider the loss function L and an example layer weight $w_k(t)$ at training step t. The gradient in the direction of w_k is given by the partial derivative of the loss function with respect to the weight $\frac{\partial L}{\partial w_k}(I, \mathbf{y}, \mathbf{o}, \Theta)$, evaluated for the current input example I and its associated target label vector $\mathbf{y}$, the output of the network $\mathbf{o}$ and all the parameters of the network Θ. Should either batch or mini-batch gradient descent be used, $\frac{\partial L}{\partial w_k}(I, \mathbf{y}, \mathbf{o}, \Theta)$ is evaluated for each training example in the batch/mini-batch and averaged over the number of examples in the current training batch/mini-batch. Henceforth, the partial derivative will be referred

[4] This is the reason why activation functions should be differentiable, as otherwise gradient descent cannot be used.

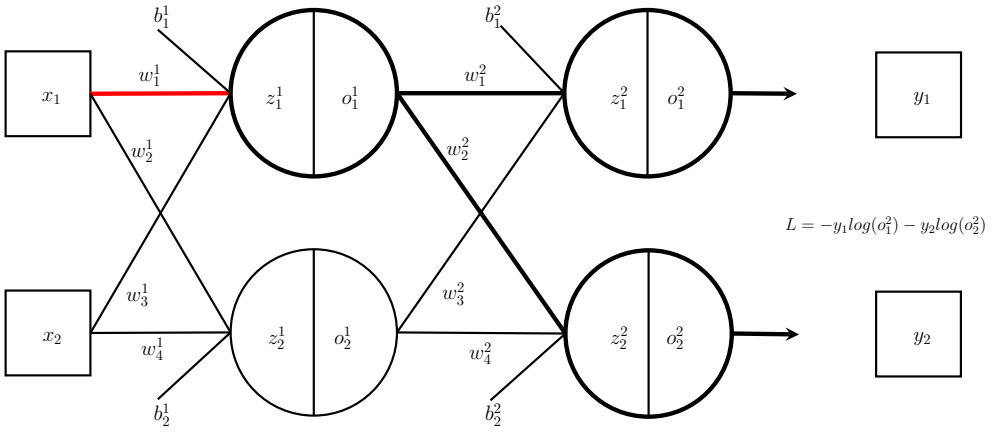


Figure 2.12: A neural network example. The network has two inputs, two fully connected layers each containing two neurons, and produces two outputs. The network uses the standard cross-entropy loss function L as indicated in the figure.

to as $\frac{\partial L}{\partial w_k}$ for readability and represents either the 'true' gradient (stochastic) or the average gradient (batch/mini-batch) in the direction of w_k. The change in the weight value is then given by

$$w_k(t+1) = w_k(t) - \ell \frac{\partial L}{\partial w_k}, \tag{2.21}$$

where ℓ refers to the learning rate, which determines the size of the step that should be taken. The learning rate is one of the more important hyperparameters of the training procedure and will be discussed in more detail in Section 2.3.

Now that we understand how to train the parameters of a neural network, we need to address the question of how to obtain the needed partial derivatives. This can be done by evaluating the loss function for the current training examples and propagating the errors back through the neural network, calculating the partial derivatives in each step. The process of efficiently calculating the partial derivatives for each parameter in the network is called backpropagation.

2.2.3 Backpropagation

The derivation of the backpropagation algorithm is quite extensive, and providing the full derivation in this section is beyond the scope of this thesis. Instead, this section will illustrate the underlying concept of backpropagation using an example. For a fully detailed derivation of the backpropagation algorithm, refer to Rumelhart *et al.* (1986).

Consider the neural network in Figure 2.12. The network takes two inputs, x_1 and x_2, and produces two outputs corresponding to the target labels, y_1 and y_2. The architecture of the network consists of two fully connected layers, with

each layer containing two neurons. The parameters belonging to the first layer are denoted using the superscript 1 and the second layer using the superscript 2. Information flows from the inputs, x_1 and x_2, through layers 1 and 2, to the loss function. The goal of this example is to calculate the partial derivative of the loss function L with respect to w_1^1 (the red connection in Figure 2.12), through the pathways indicated by the bold lines.

The first step is to realize that the partial derivatives are calculated from the output of the network to the input, which means we first calculate the partial derivatives with respect to the parameters of layer 2, before moving on to the parameters of layer 1. As such, consider the partial derivatives of L with respect to the pre-activations of both neurons in layer 2,

$$\frac{\partial L}{\partial z_1^2} \quad \text{and} \quad \frac{\partial L}{\partial z_2^2}. \tag{2.22}$$

For this, the chain rule can be applied:

$$\frac{\partial L}{\partial z_1^2} = \frac{\partial L}{\partial o_1^2}\frac{\partial o_1^2}{\partial f_1^2}\frac{\partial f_1^2}{\partial z_1^2} \tag{2.23}$$

and

$$\frac{\partial L}{\partial z_2^2} = \frac{\partial L}{\partial o_2^2}\frac{\partial o_2^2}{\partial f_2^2}\frac{\partial f_2^2}{\partial z_2^2}. \tag{2.24}$$

With $\frac{\partial L}{\partial z_1^2}$ and $\frac{\partial L}{\partial z_2^2}$ now defined, it is possible to calculate the partial derivatives of L with respect to both the parameters of layer 2 and the parameters of layer 1. For instance, let us calculate the partial derivative of L with respect to the weights w_1^2 and w_2^2, then the partial derivatives are

$$\frac{\partial L}{\partial w_1^2} = \frac{\partial L}{\partial z_1^2}\frac{\partial z_1^2}{\partial w_1^2} \quad \text{and} \quad \frac{\partial L}{\partial w_2^2} = \frac{\partial L}{\partial z_2^2}\frac{\partial z_2^2}{\partial w_2^2}. \tag{2.25}$$

Similarly, we can use $\frac{\partial L}{\partial z_1^2}$ and $\frac{\partial L}{\partial z_2^2}$ to calculate the partial derivative of L with respect to the output of neuron 1 in layer 1. This is where it starts to become complicated, as there are multiple paths through the network that can be followed to get from the loss function to the output of this neuron. The partial derivative of L with respect to o_1^1 is then given by

$$\frac{\partial L}{\partial o_1^1} = \frac{\partial L}{\partial z_1^2}\frac{\partial z_1^2}{\partial o_1^1} + \frac{\partial L}{\partial z_2^2}\frac{\partial z_2^2}{\partial o_1^1}. \tag{2.26}$$

Now, the final step is to calculate the partial derivative of L with respect to w_1^1 (the red connection weight). As such, the partial derivative is given by

$$\frac{\partial L}{\partial w_1^1} = \frac{\partial L}{\partial o_1^1}\frac{\partial o_1^1}{\partial f_1^1}\frac{\partial f_1^1}{\partial z_1^1}\frac{\partial z_1^1}{\partial w_1^1}. \tag{2.27}$$

Notice from Equation 2.26 that the deeper the parameter is in the network (from the output) and with additional neurons in each layer, the number of summations and factors in the partial derivatives will increase accordingly. Also notice that some of the factors, such as Equation 2.23 and Equation 2.24, were reused in the calculation of the partial derivative with respect to multiple network parameters. This re-usability of factors opens up the possibility of a dynamic programming solution. A similar process can be followed to obtain the partial derivative for each of the remaining parameters of the network. Given the multiplicative nature of the partial derivative calculation, care should be taken against multiple small-valued factors, causing the gradients to approach zero (vanishing gradient problem), or multiple large-valued factors, causing the gradients to approach infinity (exploding gradient problem) (Hochreiter *et al.*, 2001).

This process of calculating the partial derivatives can be applied to networks with any number of neurons and layers, provided that the functions used in the model (activation functions and loss function) are differentiable. Thankfully, it is no longer required to calculate these partial derivatives by hand, as modern frameworks such as Theano (Bergstra *et al.*, 2010; Bastien *et al.*, 2012) (which is used in this thesis, see Section 2.4) provide the functionality of calculating the gradients automatically.

2.2.4 Regularization

We next discuss techniques used to tackle overfitting, a common problem encountered when training neural networks. Overfitting occurs when a neural network memorizes the properties specific to the training data, rather than generalizing from it (Hawkins, 2004; Srivastava *et al.*, 2014). This can often be detected by comparing the performance of the network on the training data to a portion of data that it has not seen before, which is referred to as the validation set.

A clear indication of overfitting is shown in Figure 2.13, where the validation error starts to increase after a certain number of training cycles, while the training error continues to improve. This example network is no longer generalizing well on the validation set after the point labeled 'Early stopping'. Should the data set not have a dedicated validation set, cross-validation can also be used to detect overfitting (Kohavi, 1995). An n-fold cross-validation involves dividing the training set into n equally sized, non-overlapping subsets, where 1 subset is used as the validation set while training on the remaining $n-1$ subsets. Each subset is in turn used as a validation set while training one model, resulting in a total of n differently trained models. The final performance of the model can then be estimated by averaging the errors of the trained models on their corresponding validation sets. For instance, consider a 5-fold cross-entropy, where 5 versions of the same model were trained and evaluated on different folds of training and validation sets. If model 1 evalu-

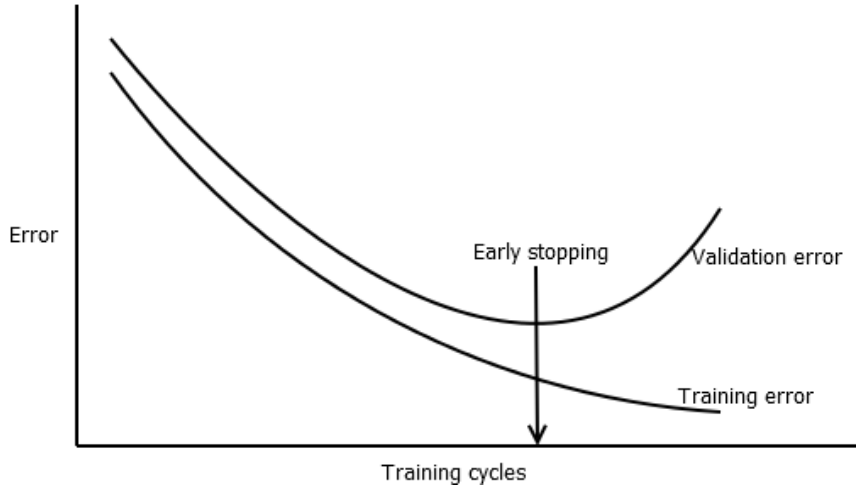


Figure 2.13: The idealized error curves of a neural network evaluated on a training set and a validation set, over the duration of training. The neural network clearly seems to be overfitting on the training data, as the training error decreases while the validation error increases.

ated on fold 1 yields error e_1, model 2 on fold 2 yields e_2, up to model 5 on fold 5 yielding e_5, then the final performance of the model is estimated to be $\frac{1}{5}\sum_{i=1}^{5} e_i$.

There are a number of regularization techniques that can be used to reduce the amount of overfitting of a network, including early stopping, data enrichment, L_1 and L_2 regularization, Dropout and DropConnect. A brief overview of these techniques is given in the following paragraphs.

Early Stopping Early stopping is a technique used to retain the optimal network parameters based on the validation error (Prechelt, 2012). It does not reduce overfitting, but rather stops the training procedure of the neural network before overfitting can occur. Figure 2.13 provides an illustration of where early stopping should be applied to retrieve the 'best' generalized model according to the validation error.

Data Enrichment The only regularization technique that was used in this work is data enrichment. It was applied to the nerve cell membrane data set (Section 3.2) since the training set only contained a small number of images. The idea behind data enrichment is to artificially expand the number of training examples by applying reasonable image transformation techniques, such as mirroring and rotations. This increases the variation of data that the network can learn from, and ultimately allows the network to respond better to similar variations present in the test set.

L_1 and L_2 Regularization L_1 and L_2 regularization are similar in that they both modify the loss function. Both regularization techniques penalize the network for having large connection weights, with L_2 regularization being the more common choice (Nielsen, 2015).

L_1 regularization corresponds to assuming a Laplacian prior on the weights (excluding the biases), driving most of the parameters of the network to zero, enforcing sparse connectivity in the network and inherently performing feature selection (Figueiredo, 2003; Ng, 2004). The full loss function with L_1 regularization[5] is given by

$$L_{reg} = L + \lambda \sum_{k=1}^{M} |w_k| = L + \lambda \|\mathbf{w}\|_1, \tag{2.28}$$

where L is the original loss function, $\mathbf{w}$ is an M-dimensional collection of all the connection weights in the network (not including bias) and λ is the hyperparameter controlling the importance of the L_1 cost compared to L.

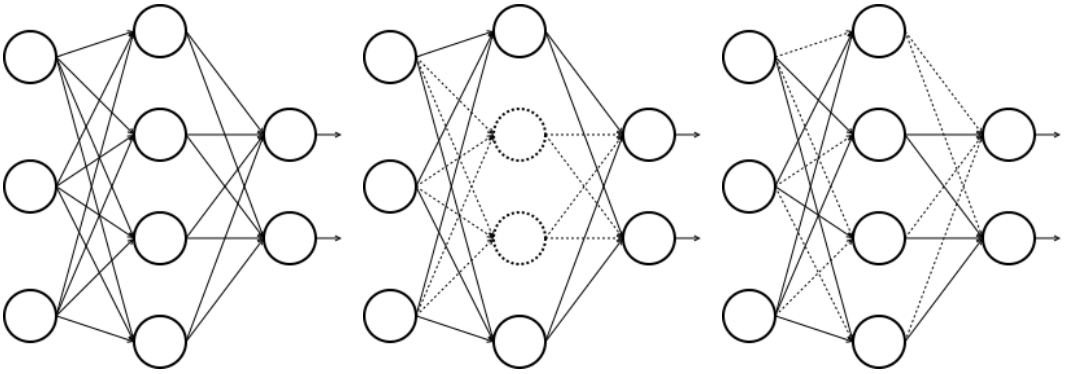
L_2 regularization, also referred to as weight decay, corresponds to assuming a Gaussian prior on the weights (excluding the biases), driving the parameters of the network to be close, but not equal, to zero (Figueiredo, 2003). The full loss function with L_2 regularization is given by

$$L_{reg} = L + \lambda \sum_{k=1}^{M} w_k^2 = L + \lambda \|\mathbf{w}\|_2^2. \tag{2.29}$$

Dropout Dropout is a technique applied separately to a single layer in which a random portion of the neurons in the layer is temporarily removed during each training iteration. It involves specifying a fraction (or percentage) that acts as the drop chance of any given neuron in the layer. For instance, a 10% dropout applied to a fully connected layer means that each neuron in the layer has a 10% chance of being dropped during each training iteration. Note that dropout (of equal or different percentage) can be applied to any of layers (including input) in any combination, except for the output layer of the neural network.

Consider the full network diagram in Figure 2.14a. For a single training iteration (one stochastic example, one mini-batch or one batch of examples), 50% dropout was applied to the center layer, resulting in the network shown in Figure 2.14b. This network, now with only two active neurons in the center layer, has to learn a more meaningful representation of its input, such that the output layer is still able to make accurate predictions without relying on the dropped neurons. For each training iteration, a different random set of neurons are dropped.

[5]Similar to ReLUs (and variations thereof), the L_1 loss is not differentiable at $w_k = 0$, but neural network frameworks typically include a self-defined derivative of zero at $w_k = 0$.



(a) Fully connected network (b) Network with Dropout (c) Network with DropConnect

Figure 2.14: A small fully connected network to which either Dropout (Figure 2.14b) or DropConnect (Figure 2.14c) has been applied. Dotted lines indicate weights/neurons that were temporarily dropped during training.

Dropout is a technique that forces neurons to learn more robust features, since they cannot rely on the presence of the other neurons in the layer. Essentially, dropout averages the performance of multiple neural networks sharing weights within a single network. More detailed discussions on dropout can be found in Nielsen (2015), Hinton *et al.* (2012*b*) and Srivastava *et al.* (2014).

DropConnect DropConnect is a generalization of Dropout, where instead of removing a random subset of neurons in a fully connected layer, random connections to the neurons are removed, basically creating a sparsely connected layer (Wan *et al.*, 2013). DropConnect was only applied to fully connected layers in Wan *et al.* (2013) and it was shown to often outperform models using regular Dropout. Figure 2.14 shows the difference in the resulting configurations between Dropout and DropConnect when applied to fully connected layers.

With one or more of these regularization techniques, it is possible to reduce the amount of overfitting of a network to the training data. Next, we look at the learning rate hyperparameter and techniques on how to optimize gradient descent and speed up learning.

2.3 Optimizations and the Learning Rate

The learning rate ℓ, a small positive constant, is one of the more important hyperparameters of the network, since it has direct influence on the success of training. A poorly chosen learning rate can either prevent the network from converging to a local minimum (when the learning rate is too large) or make the network converge extremely slowly (when the learning rate is too

small). Choosing the correct learning rate often requires experimentation, testing multiple learning rates to find the best.

As such, this section will focus on common techniques that can be used to optimize the gradient descent algorithm. Some of these techniques achieve this by (dynamically) modifying the learning rate, thus making the choice of learning rate a bit easier. Note that in the equations for the different techniques, all of the additional parameters introduced by the techniques need to be calculated for the current training step t, in order to determine the weights used in the next training step $t+1$.

Momentum The first technique we discuss is momentum, a common gradient descent optimization technique. The concept of momentum is similar to that of momentum in physics, in that an external force (negative gradients) exerted on an object (parameters) increases its velocity (momentum) in the direction of the force. Recall the definition for the weight update, Equation 2.21, given by

$$w_k(t+1) = w_k(t) - \ell \frac{\partial L}{\partial w_k}.$$

By using momentum, the update rule now becomes a pair of equations, given by

$$v_k(t) = \mu v_k(t-1) - \ell \frac{\partial L}{\partial w_k}, \tag{2.30}$$

$$w_k(t+1) = w_k(t) + v_k(t), \tag{2.31}$$

where v_k is the velocity parameter for weight w_k and μ is an additional hyperparameter that controls the rate at which the parameter velocity decays (corresponding to the coefficient of friction) (Nielsen, 2015).

RMSProp RMSProp (or Root Mean Square Propagation) is one of the first learning rate acceleration techniques employed during this work. It is considered to be a generalization of RProp (short for Resilient Propagation) — a learning rate acceleration technique developed for batch gradient descent only (Riedmiller and Braun, 1993) — to enable its use with mini-batch gradient descent. RMSProp was presented in Hinton *et al.* (2012*a*), and involves keeping an exponentially weighted moving average of the magnitudes of recent partial derivatives to normalize the current partial derivative. The regular weight update rule, Equation 2.21, then becomes a pair of equations, given by

$$E_k(t) = \gamma E_k(t-1) + (1-\gamma)\left(\frac{\partial L}{\partial w_k}\right)^2, \tag{2.32}$$

$$w_k(t+1) = w_k(t) - \frac{\ell}{\sqrt{E_k(t)+\epsilon}} \frac{\partial L}{\partial w_k}, \tag{2.33}$$

where γ is an additional hyperparameter — the smoothing parameter for the moving average — and ϵ is a small fixed positive constant for numerical stability.

RMSProp was used during initial experimentation for the work done in Chapter 4, but was later replaced with the more sophisticated ADADELTA (Zeiler, 2012), an extension of ADAGRAD (Duchi *et al.*, 2011). Experimentally, RMSProp significantly reduced the amount of time required to train the neural networks; however, it did prove to be extremely unstable near the end of training. Investigation into the instability of RMSProp suggested that $E_k(t)$ for some parameters become very small near the end of training, since the parameter values start to stabilize (small partial derivatives). An $E_k(t) < 1$ would emphasize the corresponding parameter in the direction of the partial derivative, thereby destabilizing the value of the parameter (potentially causing it to explode). A attempted, but later unsuccessful, solution was to replace ℓ with a decaying learning rate, reducing the learning rate by a constant factor each time the training cost increases.[6]

ADAGRAD The ADAGRAD acceleration technique is also a parameter-specific adaptive learning rate (Duchi *et al.*, 2011). It speeds up learning for slow-learning parameters, and regulates fast-learning parameters. This is done by accumulating all the partial derivatives for a parameter since the start of training. As such, the regular weight update rule, Equation 2.21, is replaced by a pair of equations, given by

$$G_k(t) = G_k(t-1) + \left(\frac{\partial L}{\partial w_k}\right)^2, \tag{2.34}$$

$$w_k(t+1) = w_k(t) - \frac{\ell}{\sqrt{G_k(t)+\epsilon}}\frac{\partial L}{\partial w_k}. \tag{2.35}$$

In practice, ADAGRAD has been shown to provide a significant boost in training speed; however, due to the constant accumulation of partial derivatives in G_k, the denominator in Equation 2.35 can quickly drive the learning rate to zero, essentially stopping learning prematurely (Zeiler, 2012).

ADADELTA The ADADELTA acceleration technique is an extension of ADAGRAD which tries to prevent its rapid convergence to zero learning rates (Zeiler, 2012). This adaptive learning rate was used in all experiments reported in this work, as it has proved to provide similar speed-ups to RMSProp without the corresponding instability at the end of training. ADADELTA also does not require a global learning rate that needs to be manually adjusted. Similar

[6] A decreasing training cost suggests an appropriately sized step in the direction of the partial derivatives. A increasing training cost suggests that the step size is too large, and needs to be scaled down.

to RMSProp, ADADELTA keeps an exponentially weighted moving average of past partial derivative magnitudes, but it also keeps an exponentially weighted moving average of the magnitude of recent parameter updates. The regular weight update rule, Equation 2.21, is then replaced by four equations, given by[7]

$$E_k^g(t) = \gamma E_k^g(t-1) + (1-\gamma)\left(\frac{\partial L}{\partial w_k}\right)^2, \tag{2.36}$$

$$E_k^w(t) = \gamma E_k^w(t-1) + (1-\gamma)(\Delta w_k(t))^2, \tag{2.37}$$

$$\Delta w_k(t) = -\sqrt{\frac{E_k^w(t-1)+\epsilon}{E_k^g(t)+\epsilon}}\frac{\partial L}{\partial w_k}, \tag{2.38}$$

$$w_k(t+1) = w_k(t) + \Delta w_k(t). \tag{2.39}$$

2.4 Neural Network Implementation

The neural network architectures used in this thesis were implemented using Theano (Bergstra *et al.*, 2010; Bastien *et al.*, 2012) in Python. The networks were trained on a desktop workstation containing an Intel Core i7-4790 3.6GHz CPU, 16GB of main memory and an NVIDIA GeForce GTX980 Ti graphics processor with 6GB of memory.

2.5 Neural Network Architectures

There are a large variety of neural network architectures discussed in the literature. Given the building blocks of neural networks, the various layers discussed in Section 2.1.3 and others, the architecture of the network is only limited by the imagination, subject to technological limitations such as computer memory, the availability of one or more GPUs and the memory available on the GPUs. As such, in earlier years (1980s - 1990s), much smaller stacked fully connected layers and autoencoders with at most a couple of thousand trainable parameters were popular. However, as technology improved, the neural networks grew in size. To date, one of the largest neural networks that has been trained successfully was reported by Digital Reasoning Systems in 2015 (Trask *et al.*, 2015), containing a staggering 160 billion trainable parameters. The specific architectures of interest for this study are standard convolutional neural networks (CNNs) and the recently introduced fully convolutional networks (FCNs).

[7]Note that ϵ must occur in both the numerator and the denominator in Equation 2.38. Both E_k^g and E_k^w are initialized to zero ($E_k^g(0) = 0$ and $E_k^w(0) = 0$), thus ϵ in the numerator ensures a non-zero $\Delta w_k(1)$ allowing learning to start. If E_k^w is initialized to be non-zero, then ϵ can be dropped in the numerator.

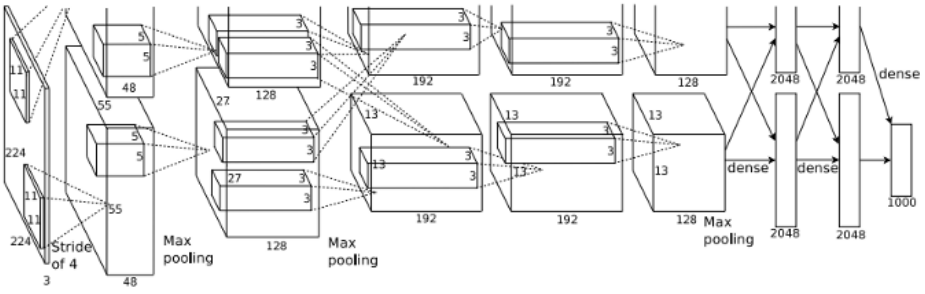


Figure 2.15: The architectural diagram of AlexNet from Krizhevsky *et al.* (2012).

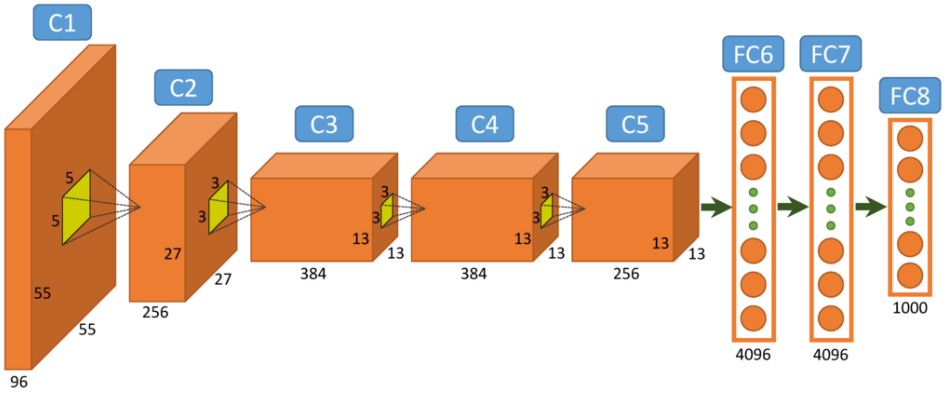


Figure 2.16: A simplified representation of AlexNet. Image extracted from Collet (2017).

2.5.1 Convolutional Neural Networks

Standard CNNs, whether for image classification or semantic segmentation, typically consist of a number of convolutional and max-pooling layers, followed by fully connected layers and ending with a softmax output layer. In this work (see Chapter 4), a standard CNN (containing about 600,000 parameters) is used as the basis for comparison with multiple FCN architectures.

One well-known CNN, AlexNet (Krizhevsky *et al.*, 2012), contains about 60 million trainable parameters and was used on the ImageNet 1000-class problem. The full published architecture of AlexNet is illustrated in Figure 2.15. Although it looks complicated, the architecture was designed to allow training across multiple GPUs, and can ultimately be seen as having five convolutional layers, three pooling layers, two fully connected layers and a softmax output layer. A simplified representation of AlexNet which shows this is given in Figure 2.16. Note that the three pooling layers are not illustrated in Figure 2.16, between `C1` and `C2`, `C2` and `C3`, and `C5` and `FC6`.

2.5.2 Fully Convolutional Neural Networks

FCNs form the basis of the work done in this thesis. They were first introduced in Long *et al.* (2015), where they took well-known CNNs, such as AlexNet (Figure 2.15), and converted them to what are now known as FCNs. They performed this conversion using the basic principle that any fully connected layer can also be interpreted as a convolutional layer.

To illustrate this, consider the fully connected layer, `FC6`, in Figure 2.16. It has an input of size $256 \times 6 \times 6$ (the pooling layer reduces the size from $256 \times 13 \times 13$ to $256 \times 6 \times 6$) and consists of 4096 neurons. In order to convert `FC6` to a convolutional layer, consider creating a new convolutional layer named `C6` having an equal number of feature maps as there are neurons in `FC6`. The resulting convolutional layer will then have 4096 feature maps. Next, the filters of `C6` are created to be the same shape as the input to `FC6` (i.e. 6×6 filters with 256 channels) and to use 'valid' padding. The resulting shape of the weight matrix for `C6` will then be $(4096, 256, 6, 6)$. All that is left, is to use the same activation function for `C6` that was used for `FC6`. With this, we can now replace `FC6` with the equivalent convolutional layer `C6`.

The second fully connected layer, `FC7`, can now be converted to a convolutional layer following the same process. `FC7` now receives the output of `C6` as input, which is of size $4096 \times 1 \times 1$. The equivalent convolutional layer to `FC7`, named `C7`, would then have 1×1 filters with 4096 channels and 4096 feature maps with a size of 1×1. Lastly, the softmax output layer `FC8` corresponding to the 1000-class classification task can be replaced by a ConvSoftmax layer. The ConvSoftmax layer would receive an input of size $4096 \times 1 \times 1$, use 1×1 filters with 4096 channels and contains 1000 feature maps. With this, the resulting FCN is equivalent to the original CNN with an input size of 224×224, without the corresponding restriction of a fixed input size. Note that the equivalence between the resulting FCN and the original CNN, also mean that the two architectures contain the same number of parameters.

In Long *et al.* (2015), their converted CNN (or FCN) was applied to the 21-class segmentation task of the PASCAL VOC challenge (Everingham *et al.*, 2011)). This meant that the number of feature maps in the ConvSoftmax layer had to be reduced to 21 (from 1000). They used their FCN on 500×500 input images to generate output maps of size 10×10 (the feature maps of each layer changes accordingly to the size of the input). This 10×10 output produced by the FCN had to be upsampled back to the resolution of the original input to the network if segmentation is to be performed. The output of the ConvSoftmax layer can be upsampled by attaching a deconvolutional layer to its output, with the deconvolutional layer having 21 feature maps using 64×64 filters and a filter stride of 32 . Recall that a filter stride of 2 downsamples the dimensions of the input by a factor of 2 in the case of a convolutional layer. Conversely, a filter stride of 2 upsamples the dimensions of the input by a factor of 2 in the case of a deconvolutional layer. Note that the upsampled output would

not be the same size as the original input, as there are instances where the border pixels are lost through pooling. For instance, if we ignore the number of feature maps for now, `C2` is downsampled through pooling from 27×27 to 13×13. Upsampling the 13×13 feature maps back to the original resolution would result to a size of 26×26, which differs from the original size of the feature maps of `C2`.

There are four points in the network where the resolution of the original input is downsampled, the first of which is by a factor of 4 as a result of the filter stride of 4 in `C1`. Next, each max-pooling layer (total of 3) adds a further downsampling factor of 2, which leads to a downsampling factor of $4 \times 2 \times 2 \times 2 = 32$, corresponding to the filter stride used in the deconvolutional layer. Note that after upsampling, the output of the network is no longer a probability distribution at each pixel location over the feature maps, which was corrected in Long *et al.* (2015) by removing the softmax activation function from the ConvSoftmax layer (it now uses a linear activation function) and applying the softmax function in the deconvolutional layer.

After testing and observing the performance of their converted CNN, it was found that the predictions made by these convolutionalized networks were too coarse and not sufficient for segmentation. The networks were able to localize the objects to a specific region in the input, but could not provide a detailed segmentation of the object. Long *et al.* (2015) concluded that a single deconvolutional layer attached to the output of the network was unable to provide the finer details required for segmentation, and addressed this issue by creating three new architectures, each consisting of multiple convolutional, deconvolutional and max-pooling layers.

A combination of the architectures which is based on the code provided by Long *et al.* (2015) is depicted in Figure 2.17. They created three networks, FCN-32s, FCN-16s and FCN-8s, with each being an extension of the network that comes before it.

The first network, FCN-32s, follows the backbone of the architecture (from the `input` down to `fc7`), followed by a linear convolutional layer with 1×1 filters (blue vertical line) and upsampled by a deconvolutional layer with 64×64 filters and a stride of 32. The softmax activation function is then applied over the feature maps produced by the deconvolutional layer, resulting in the output labeled FCN-32s (orange vertical line on the right).

The second network, FCN-16s, attaches a linear convolutional layer with 1×1 filters to the pooling layer `Pool 4`. Also, the linear convolutional layer in the FCN-32s path is upsampled using a deconvolutional layer with 4×4 filters and a stride of 2. The output of the linear convolutional layer attached to `Pool 4` is then cropped and summed with the output of the deconvolutional layer, before being further upsampled by a deconvolutional layer with 32×32 filters and a stride of 16. Similar to FCN-32s, the softmax activation function is then applied over the resulting feature maps, giving the output labeled FCN-16s (orange vertical line in the middle).

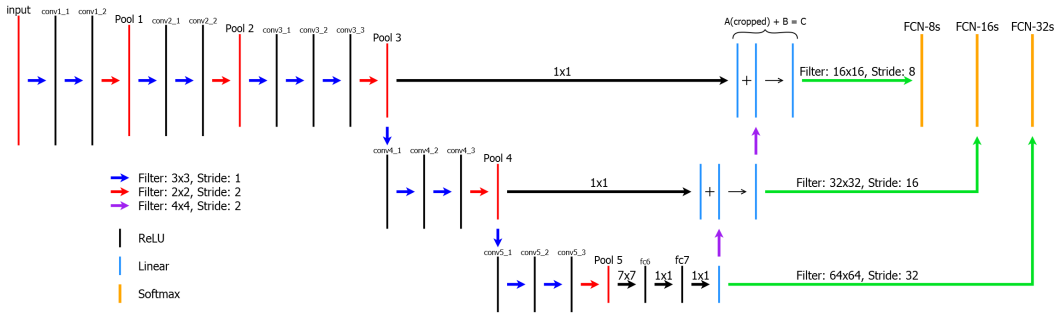


Figure 2.17: An architectural diagram designed from the specifications given in Long *et al.* (2015) for the three models they created: FCN-8s, FCN-16s and FCN-32s. Vertical lines represent the various layers of the network, with the color of the line representing a specific activation function indicated in the legend. Red lines do not have an activation function, as they represent either the output of a pooling layer or the original input. Blue and black arrows represent convolutions; red arrows represent max-pooling operations; green and purple arrows represent deconvolutions. The size and stride of the filters for each layer is indicated in the diagram and unless specified otherwise, the filter stride defaults to 1.

A similar procedure as in FCN-16s was then applied to the pooling layer `Pool 3`, resulting in the model FCN-8s (orange vertical line on the left). Similar to the convolutionalization process of AlexNet, it should be noted that each consecutive pooling layer increases the total downsampling factor by a factor of 2, hence the downsampling factors for `Pool 1` to `Pool 5`, are 2, 4, 8, 16 and 32, respectively. This downsampling factor corresponds to the filter stride of the deconvolutional layer, which enables it to upsample the output of the respective pooling layer back to the scale of the input image. Their resulting model, FCN-8s, achieved state-of-the-art performance on multiple semantic segmentation data sets, including PASCAL VOC 2011 (Everingham *et al.*, 2011), NYUDv2 (Silberman *et al.*, 2012) and SIFT Flow (Liu *et al.*, 2009).

In summary, any neural network consisting only of the layers discussed in Section 2.1.3.2 can be classified as an FCN. All of the decisions made in the output layer of an FCN are based on localized regions in the input, with the size of these regions being dependent on the architecture of the network (see Chapter 4). Any neural network containing layers from both Sections 2.1.3.1 and Section 2.1.3.2 are referred to as a CNN, which bases its decisions on the entire input. There are one CNN and four FCNs used in this thesis, presented in Chapter 4 and Chapter 5.

2.6 Conclusion

This chapter provided an overview of the field of neural networks, focusing on aspects most pertinent to the work done in this thesis. Artificial neurons were discussed and how they relate to biological neurons. The various activation functions that can be used were then discussed, followed by a brief overview of commonly used layers. Particular attention was paid to convolutional layers, as they form the basis of the work done in this thesis. The commonly used loss functions were then presented, as well as an overview of how neural networks are trained using gradient descent and backpropagation. Multiple approaches were presented that can be used to reduce overfitting and improve network generalization. An overview of learning techniques was then presented, giving examples of adaptive learning rates. Lastly, architectures of interest to this work, namely convolutional neural networks and fully convolutional neural networks, were discussed. The next chapter discusses the two main data sets that were used in this work.

Chapter 3

Data Sets

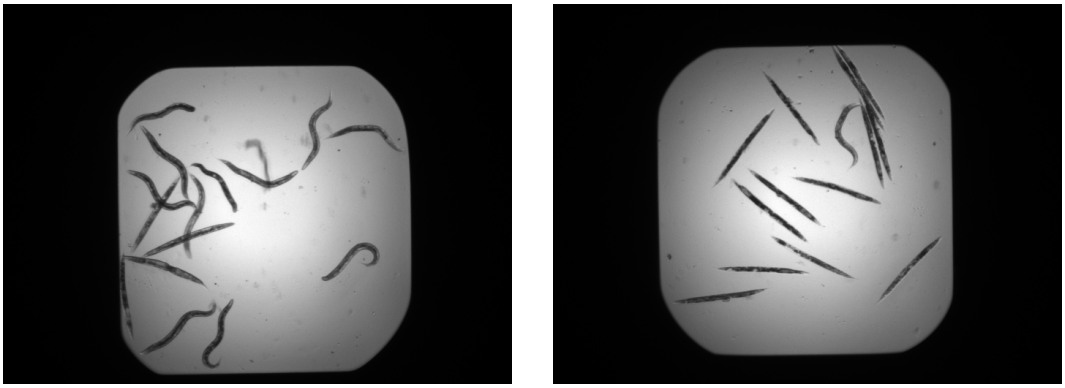
The previous chapter presented the necessary background information that is required for the understanding of the work done in this thesis. The chapter discussed the basic building blocks of neural networks: neurons, activation functions and commonly used layers. It also discussed the process of training a neural network using gradient descent, backpropagation and various optimizations that can be used.

This chapter will present the data sets that were used in this work. Two data sets were chosen, specifically in the domain of bio-images, with the task of performing semantic segmentation. Bio-image data sets for segmentation typically consist of at most a couple of hundred fully labeled images, as the labeling process is typically expensive and time-consuming since it needs to be performed by an expert (Kraus *et al.*, 2016). The small size of these data sets could present the challenge of overfitting for neural networks, since less data could be easier to memorize (Hawkins, 2004). This was circumvented in this work through the use of extensive data enrichment techniques and cross-validation.

The data sets were chosen based on whether fully annotated ground truth information was available. The first, the *Caenorhabditis elegans* (*C. elegans*) live/dead assay data set, is presented in Section 3.1, and the second, the nerve cell membrane data set, is presented in Section 3.2. Each section includes a detailed description of the respective data set, the metrics that were used to measure performance on the data set, and the pre-processing techniques that were applied to enable the use of the data with the implemented system.

3.1 *C. elegans* Live/Dead Assay

C. elegans is a species of small transparent worm roughly 1 millimeter in length, which is used as a model organism in the study of various biological processes (Kaletta and Hengartner, 2006). Their fast reproductive rates and short lifespans allow large-scale assays to be performed using microtiter plates



(a) Predominantly live image (b) Predominantly dead image

Figure 3.1: An example of a predominantly live and a predominantly dead image.

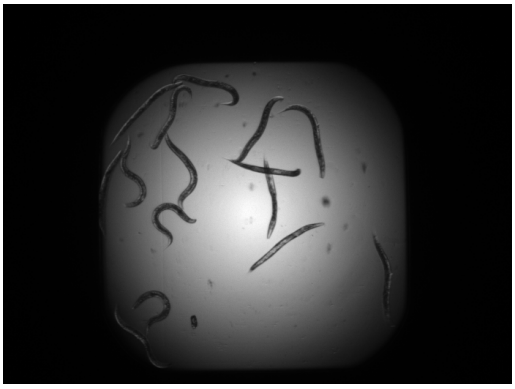
that contain a set number of wells (depending on the size of the plate), with each well containing a liquid culture of worms. Automated high-throughput screening is then performed through a microscope taking static images of each well, creating the need to automate the analysis of the resulting images through image analysis algorithms (Wählby *et al.*, 2012; Kaletta and Hengartner, 2006).

The *C. elegans* live/dead assay data set, version 1, is available for download from the Broad Bioimage Benchmark Collection (BBBC) (Ljosa *et al.*, 2012). The data was collected as part of the study performed by Moy *et al.* (2009) using worms infected with *Enterococcus faecalis*, a gut bacteria that can be lethal to the worms. They used a 384-well microtiter plate (24 columns, 16 rows) which contained liquid cultures with approximately 15 infected adult worms per well. The wells in columns 1 through 12 were treated with an antibiotic, which killed the bacteria and prevented the death of the worms in the corresponding wells. The remaining wells in columns 13 though 24 were mock-treated with DMSO, which has no effect on the bacteria thus permitting the death of most of the worms in the corresponding wells. Lastly, all the wells were also stained with SYTOX, which is a fluorescent marker that highlights dead cells.

The images in this data set correspond to 100 wells of the 384-well plate used in Moy *et al.* (2009), with the task of localizing each worm in an image and determining whether the worm is alive or dead. Some of the wells do not have a representative image in the data set, for unspecified reasons. The data set contains 97 16-bit TIFF files of bright-field microscopy images. Images corresponding to columns 1 through 12 are labeled as *predominantly live* images (total of 52 images), since most of the worms in these images should be alive. Similarly, images corresponding to columns 13 through 24 are labeled as *predominantly dead* images (total of 45 images) which contain mostly dead worms.

(a) High illumination

(b) SYTOX image



(c) Low illumination

(d) SYTOX image

Figure 3.2: The difference between a high illumination image (Figure 3.2a) and a low illumination image (Figure 3.2c). The corresponding SYTOX images highlight the dead worms in each image. Figure 3.2c illustrates how it may be more difficult to localize and classify worms along the borders in a low illumination setting.

This data set contains two foreground classes and one background class, with the foreground classes being: alive worms and dead worms. Each foreground class has its own set of visual characteristics, which segmentation methods could potentially learn to recognize. The live worms are smooth in texture and can take on any shape, and thus usually exhibit some curvature. On the other hand, dead worms have a rod-like shape and are uneven in texture. Examples of a predominantly live and a predominantly dead image are given in Figure 3.1.

Also included in the data set are 98 16-bit TIFF files of SYTOX-stained fluorescence microscopy images. Each of these images correspond to a single bright-field image and can be used as an indication of whether or not any

(a) Bright-field image (b) Binary segmentation (c) Single worm mask

Figure 3.3: An example of a binary segmentation mask and a particular single worm segmentation mask for a corresponding bright-field image.

given worm is alive or dead.[1] Both of these image sets have a fairly simplistic background; however, they do suffer from uneven illumination. This can, in some cases, make it difficult to observe the class of a worm. To visualize the difference in illumination levels, two images are presented in Figure 3.2, together with their corresponding SYTOX images.

There are two sets of ground truth provided in the data set. The first set contains 100 binary segmentation masks that show the location of all the worms in each bright-field image. This is one of the main ground truth labelings that will be used in this work. The problem of using these binary masks is that they do not distinguish between live worms and dead worms, since a single bright-field image can contain both — this is discussed further in Section 3.1.2. The second set contains 1407 single worm binary segmentation masks, where each mask shows the location of a single worm in a single bright-field image. An example bright-field image with the corresponding binary segmentation mask and a single worm segmentation mask is given in Figure 3.3. It should be noted that similar to the binary segmentation masks, the single worm segmentation masks also do not distinguish between the live and dead worms.

3.1.1 Metrics

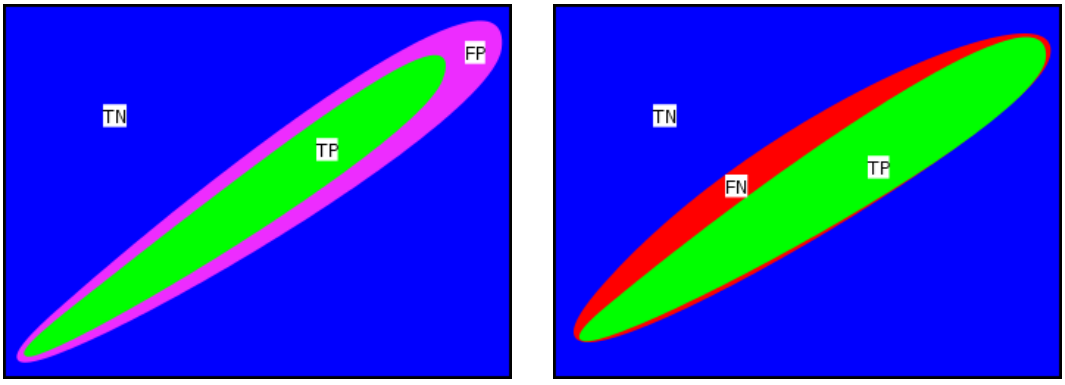
Wählby *et al.* (2012) reported four binary classification metrics: accuracy, precision, recall and F-factor[2]. A binary classification problem is any classification problem that has only two possible classes, a positive and a negative class (Powers, 2011; Fawcett, 2006). Given such a data set, a classifier can be used to predict the class of each example in the set. As such, the classifier can either correctly predict the class of an example (True Positive (TP) or True Negative (TN)), or it can make a mistake (False Positive (FP) or False Negative (FN)). The number of occurrences of each case can then be summarized

[1]Note that the 97 bright-field images each have a corresponding SYTOX image, leaving one extra SYTOX image without a bright-field image.

[2]F-factor in this instance refers to the F_1 score.

Table 3.1: Confusion matrix for a binary classification problem.

		Predicted Class	
		Positive	Negative
Actual Class	Positive	True Positive (TP)	False Negative (FN)
	Negative	False Positive (FP)	True Negative (TN)



(a) Overrepresentation (b) Underrepresentation

Figure 3.4: An illustration of over- and underrepresentation within the scope of this work. Overrepresentation refers to the foreground object being represented as larger than it actually is. Underrepresentation refers to the foreground object being represented as smaller than it actually is.

in a confusion matrix (sometimes referred to as a contingency table), as shown in Table 3.1.

Before discussing the four metrics and how to calculate them, it is important to first define the following two segmentation concepts: overrepresentation and underrepresentation. Consider Figure 3.4, which illustrates these concepts of over- and underrepresentation as per the following definitions:

> **Overrepresentation** is when too many pixels are predicted as belonging to a foreground object. In other words, the foreground object is represented as being larger than it actually is. This occurs when background pixels close to the border of the foreground object are labeled as the object itself (generating false positives — see Figure 3.4a).
>
> **Underrepresentation** is when too few pixels are predicted as belonging to a foreground object. In other words, the foreground object is represented as being smaller than it actually is. This occurs when the foreground pixels close to the border of the foreground object are labeled as background pixels (generating false negatives — see Figure 3.4b).

With these concepts in mind, it is now possible to discuss the four metrics used for this task.

Accuracy

The accuracy of a classifier is a measure of how often it correctly predicts the class of an example in the data set, and can be calculated as

$$Accuracy = \frac{TP + TN}{TP + FN + FP + TN}. \tag{3.1}$$

Precision

The precision of a classifier is a measure of how often a positive class prediction is a correct prediction, and can be calculated as

$$Precision = \frac{TP}{TP + FP}. \tag{3.2}$$

With regards to segmentation, where the positive class is the foreground object (assuming no overlap with other foreground objects), precision can also be interpreted as a measure of overrepresentation. This can be seen in Figure 3.4a, in that as the region labeled FP becomes smaller (closer to the region labeled TP), the precision becomes closer to 1.

Recall

The recall of a classifier is a measure of how often the actual positive class examples are predicted correctly by the classifier, and can be calculated as

$$Recall = \frac{TP}{TP + FN}. \tag{3.3}$$

With regards to segmentation, where the positive class is the foreground object, the recall can also be interpreted as a measure of underrepresentation. This can be seen in Figure 3.4b, in that as the region labeled TP becomes larger within the region FN (consequently the area of FN becomes smaller), the recall becomes closer to 1.

F-measure

Figure 3.4 also illustrates the complication of using precision and recall, in that a high precision is not necessarily accompanied by a high recall (Figure 3.4b). Similarly, a high recall is not necessarily accompanied by a high precision (Figure 3.4a). This is where the F-measure, also known as the F-score or F_1 score, becomes useful.

The F_1 score, a specific instance of the F_β score, is defined as

$$F_{\beta=1} = \frac{(1 + \beta^2) \cdot Precision \cdot Recall}{\beta^2 \cdot Precision + Recall} \tag{3.4}$$

$$= \frac{2 \cdot Precision \cdot Recall}{Precision + Recall}. \tag{3.5}$$

The F_β score can be interpreted as a weighted harmonic mean between the precision and recall, with $\beta = 1$ weighting the precision and recall equally.

3.1.1.1 Binary Metrics on Non-Binary Problems

The task for this data set is considered to be a multi-class classification problem (3 classes), which makes it difficult to utilize binary classification metrics to evaluate performance. As such, the task was transformed into two binary classification problems. The first considered the segmentation of the worms, which provided an indication of how well the applied approach segmented each worm from the background (worm versus background). The second was with regards to whether the segmented worms were alive or dead, which provided an indication as to how well the applied approach distinguished between living and dead worms (live versus dead).

All four metrics were utilized in both binary classification problems, and the distinction between the metrics used for each problem is made by referring to the metrics as the *segmentation metrics* and the *classification metrics*, respectively. Wählby *et al.* (2012) also reported an additional segmentation metric which utilized an F-measure threshold of 0.8. This additional metric can be interpreted as a measure of segmentation correctness (and is referred to as such), in that the segmentation of a worm is considered to be correct/complete if it has a segmentation F-measure above this threshold. It should be noted that the results of these metrics are dependent on which class is considered the positive class. To facilitate comparison with Wählby *et al.* (2012), the worm pixels (either alive or dead) were considered as the positive class for the segmentation metrics, with the background pixels as the negative class. Similarly, the live worm pixels were considered as the positive class for the classification metrics, with the dead worm pixels as the negative class.

The metrics were also evaluated at two levels, the *pixel level* and the *worm level*. For the pixel level, the natural output level produced by neural networks, the metrics were simply evaluated over all of the pixels in the test images. For the worm level, the level at which the results were reported in Wählby *et al.* (2012), the metrics were evaluated over the pixels for each individual worm, followed by averaging over the total number of worms in the test images. Section 3.1.2.2 provides a more detailed discussion on how to calculate the worm-level metrics from the pixel-level output produced by neural networks.

3.1.2 Data Preparation

There were three concerns that had to be addressed to use this data set with the neural networks employed in this work. The first concern was addressing the inaccuracies introduced by the binary segmentation masks with regards to the live or dead status of each worm. The second concern was converting the pixel-level output produced by the neural networks to the worm level in order

to make a meaningful comparison with the results reported in Wählby *et al.* (2012). The third concern was processing the images and ground truth to be used for training.

3.1.2.1 Ground Truth Relabeling

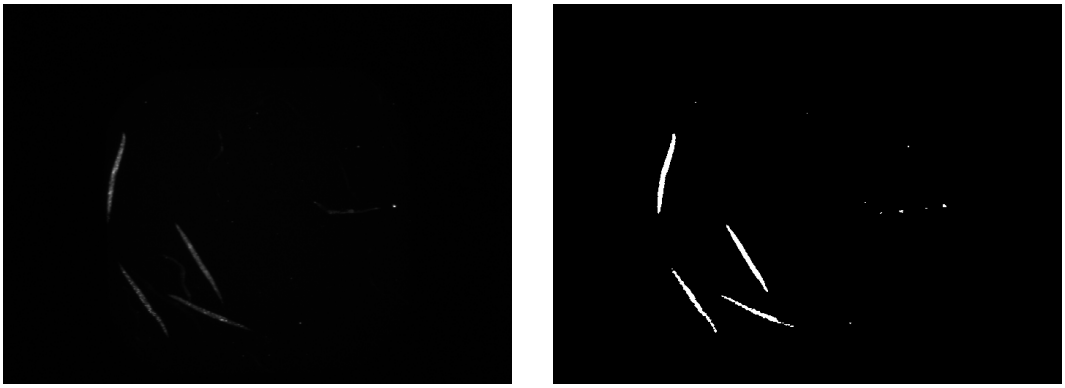
The first complication that had to be addressed was the inability to differentiate between living and dead worms when using the binary segmentation masks. As mentioned earlier, the binary segmentation masks do not distinguish between living worms and dead worms. Instead, they only show the location of the worms in each image and the live or dead status of each worm is derived from the predominantly live or dead label of the image containing the worm. For instance, a predominantly live image could contain dead worms; however, since the label for all of the worms are derived from the label of the image, this can lead to incorrect labeling during training and testing. Because this could potentially cause inaccuracies when training and evaluating the neural networks used in this work, a new set of ground truth images was created to better reflect the actual class of each worm. This was achieved by using the SYTOX images and the single worm segmentation masks.

Like the bright-field images, the SYTOX images (Figure 3.2) also suffered from uneven illumination. As such, the SYTOX images had to be manually processed by applying image-specific thresholds in order to detect the highlighted worms. The single worm segmentation masks that correspond to the highlighted worms in the SYTOX images were then relabeled as belonging to the dead class. The remaining single worm segmentation maps were then relabeled as belonging to the live class.

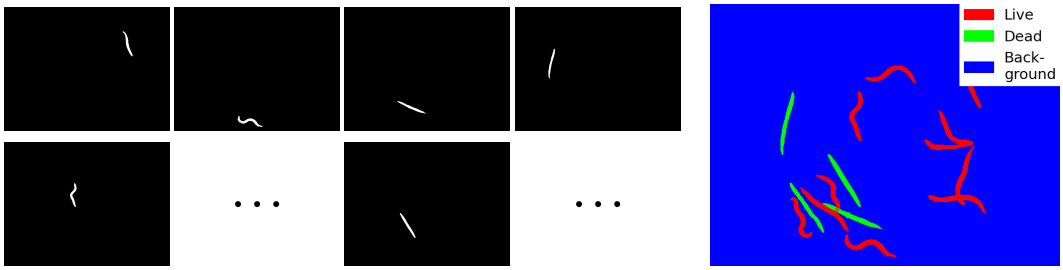
Once each worm was assigned a class, the (now labeled) single worm masks were merged to create a three channel image with the same resolution as the binary masks, with each channel representing a single class: the red channel was assigned to the live class, the green channel to the dead class and the blue channel to the background class. A value of 254 in a respective channel indicates that the pixel at that location belongs to the corresponding class. There are also some cases where two or more worms would overlap. If the overlapping worms belong to different classes, the pixels in the overlapping region were then given the value 127 in both the live and the dead channel, indicating equal probability of either being part of a living worm or a dead worm so as to not bias the classifier to any one class. These images will henceforth be referred to as the *relabeled segmentation masks*. This process is illustrated in Figure 3.5.

3.1.2.2 Metric Level Conversion

The metrics used to evaluate performance on this data set were calculated at two levels: pixel-level and worm-level.



(a) SYTOX image (b) Thresholded SYTOX

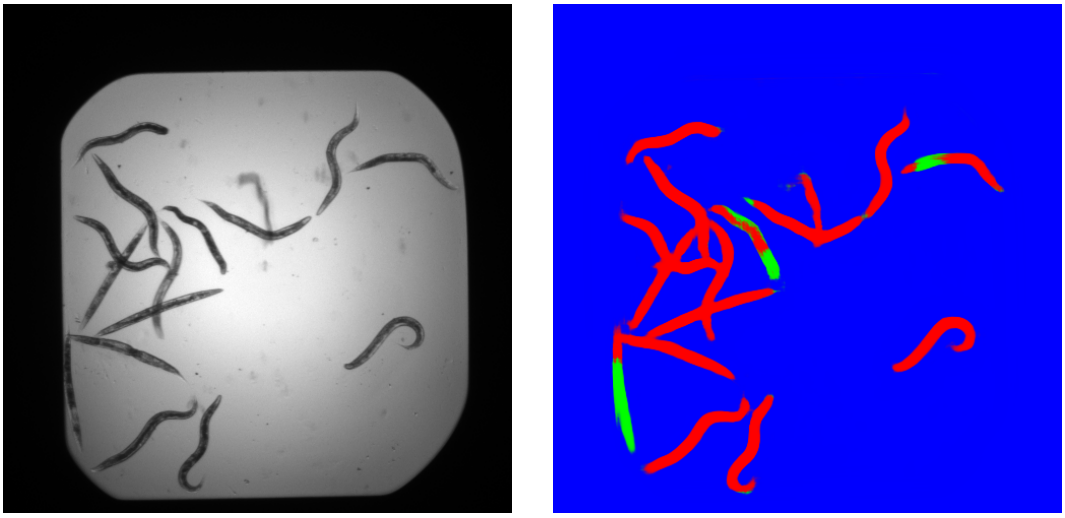


(c) Live worm masks (d) Dead worm masks (e) Relabeled RGB mask

Figure 3.5: An illustration of how the relabeled ground truth was created. An image-specific threshold was applied to the SYTOX images (Figure 3.5a) to highlight the dead worms (Figure 3.5b). The single worm segmentations that corresponded to the highlighted worms were labeled as dead (Figure 3.5d) and the remaining single worm segmentations as live (Figure 3.5c). The now labeled single worm segmentations were then merged to form an RGB image (Figure 3.5e), with red corresponding to the live class, green to the dead class and blue to the background.

Pixel-level metrics The pixel-level metrics consider each pixel as an independently labeled example and can be calculated by simply collecting all of the pixel predictions in the test images into the confusion matrix (Table 3.1). The neural networks employed in this work produce an output for each pixel in an image, which makes it relatively simple to calculate the pixel-level metrics. The results reported in Wählby *et al.* (2012), however, were calculated with respect to each individual worm. This difference in metric level requires careful consideration when comparing the neural network results to these in Wählby *et al.* (2012).

Worm-level metrics The worm-level metrics differ from the pixel-level metrics in that they are calculated based on the segmentation and classification of each individual worm, averaged over the total number of worms in the test



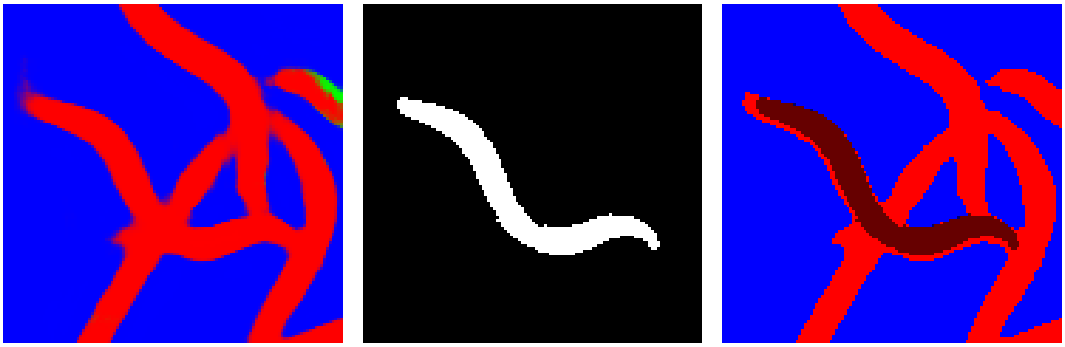
(a) Network input (b) Network output

Figure 3.6: The output produced by a neural network for the given input image.

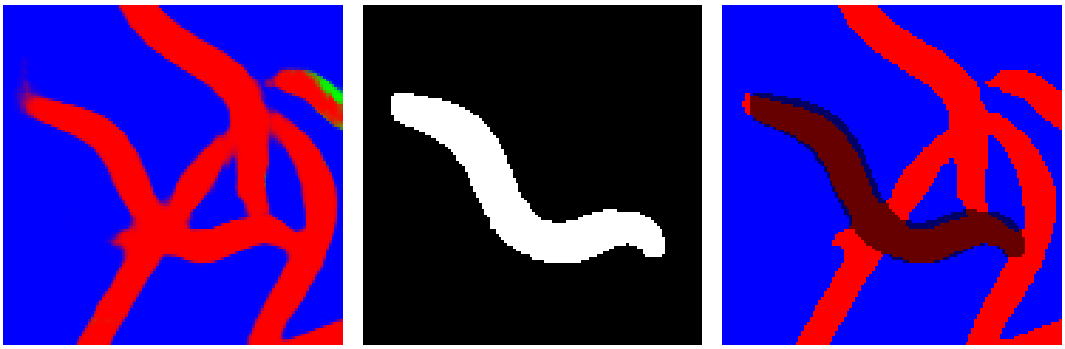
images. According to the processing pipeline outlined in Wählby *et al.* (2012), the worms were first segmented from the background and separated if there was any overlap between multiple worms. The segmentation metrics were then calculated based on how well each individual worm was segmented, followed by a classifier predicting the live or dead status of the individual worms (the ground truth used to train and/or evaluate this classifier was not specified nor provided). The classification metrics were then calculated based on the status prediction of the entire worm, not the pixels belonging to the worm.

One approach to calculate the worm-level metrics from the neural network output, is to utilize the single worm segmentation masks as a means to extract the pixel-level outputs corresponding to any particular worm. Consider the pixel-level output produced by a neural network for the particular input image in Figure 3.6. For any particular worm in the image, the pixel outputs that do not intersect with the corresponding single worm segmentation mask are temporarily considered to be background pixels. The remaining pixel outputs (which do intersect with the mask) can then be used to calculate the worm-level metrics for that particular worm. This process is repeated for every available single worm segmentation mask and the resulting worm-level metrics are averaged over the total number of worms considered.

A complication that arises by following this approach is that the single worm segmentation masks will not include possible overrepresentation produced by the networks, since any misclassified pixels outside of the mask will be discarded. For example, should the original single worm segmentation mask be used to evaluate model performance for the worm in Figure 3.7a, a misleading precision of 100% will be obtained. Although this does not necessarily



(a) Network output (zoomed) (b) Original single worm mask (c) Extracted region

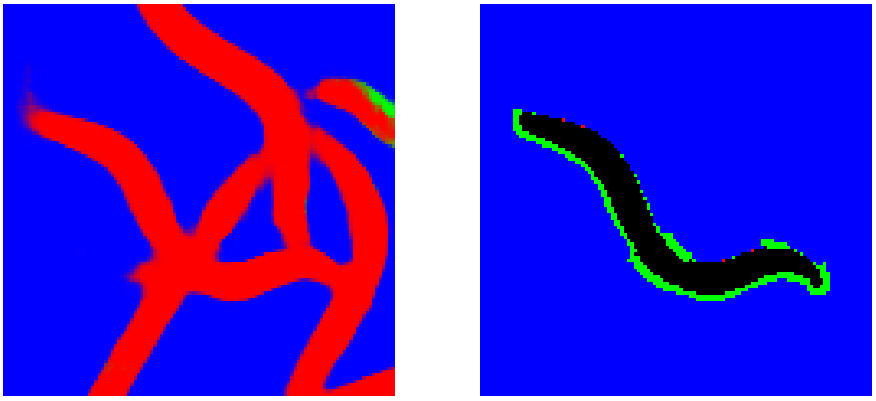


(d) Network output (zoomed) (e) Dilated single worm mask (f) Extracted region

Figure 3.7: A comparative illustration of the extracted region when utilizing either an original single worm segmentation mask (Figure 3.7b) or a dilated single worm segmentation mask (Figure 3.7e). The dark region in Figures 3.7c and 3.7f shows the region that will be extracted as potentially being part of the worm under consideration, while the rest is considered to be background. The dilated mask is able to capture most of the oversegmentation produced by the network, which results in more accurate metric scores.

seem like a problem, consider a neural network output labeling all pixels as worm pixels (and thus no background pixels). Using the original single worm segmentation masks on this output would also result in a precision of 100%. Simply put, the original single worm segmentation masks may not include false positive pixels around the worms, thus providing an inaccurate measure of precision. To overcome this complication, the single worm segmentation masks can be dilated to include extra pixels around the border of the worm. This then presents the question: by how much should the segmentation masks be dilated to detect overrepresentation?

To decide this, the single worm segmentation masks were processed using the morphological dilation operation (Szeliski, 2010). The single worm segmentation mask was dilated with a 3×3 structural element (henceforth referred to as a 3×3 dilation) and applied to the network output of model



(a) Network output (zoomed) (b) Extracted worm segmentation

Figure 3.8: The resulting segmentation when the dilated mask is used. The colors in Figure 3.8b correspond to the entries in the confusion matrix (Table 3.1), where black corresponds to True Positive, green to False Positive (overrepresentation), red to False Negative (underrepresentation) and blue to True Negative.

A from Section 4.1, which visibly caused the most overrepresentation. This process was repeated a number of times, with each repeat using an increased number of 3×3 dilations on the single worm segmentation mask. It was determined by visual inspection that two 3×3 dilations, or equivalently, one 5×5 dilation, managed to detect most of the overrepresented worm pixels. Unfortunately, having to depend on a dilated mask to extract the pixels belonging to an individual worm, also meant that the pixels of overlapping worms would be included as overrepresentation. This can be seen in Figure 3.8 where the overlapping pixels belonging to neighboring worms are included as false positives. It is important to note that the choice of dilation and the resulting metrics are sensitive to the overall level of over-/underrepresentation of the models considered.

Consider the single worm segmentation mask in Figure 3.7b, which results in the mask in Figure 3.7e when dilated using a 5×5 structural element. Using this dilated mask to extract the corresponding worm then results in the segmentation shown in Figure 3.8b, which was recolored to reflect the entries in the confusion matrix (Table 3.1).

Figure 3.7 also provides a comparative illustration between using the original mask and the dilated mask in the case of overlapping worms. Specifically, Figure 3.7f illustrates the compromise that needs to be made between including as many false positive pixels around the particular worm versus including true positive pixels of neighboring worms as false positives for the worm under consideration. Consequently, the results generated from using the dilated mask will only be an estimation of the true performance of the neural networks at the worm level. Using the same example as with the original single worm seg-

mentation mask (Figure 3.7), using the dilated mask results in a more realistic precision of 74.5%.

Once the ability to extract the relevant pixels for any specific worm using the corresponding dilated single worm segmentation mask has been established, it becomes possible to calculate the worm-level metrics for the neural networks. The worm-level segmentation metrics can be calculated directly from the extracted pixels and are averaged over the total number of worms in the test set. The worm-level classification metrics, on the other hand, are calculated based on the class assigned to each worm as a whole (i.e. not the pixels). The class of the worm can be obtained by assigning hard labels to the extracted worm pixels using a threshold of 0.5 on the probabilistic output from the neural network. The majority rule can then be applied using the hard labels to predict whether the worm is alive or dead. Note that there could be other approaches to predict the class of the whole worm.

Both the pixel-level metrics and the worm-level metrics will be reported for work done using this data set (see Chapter 4). This is necessary to be able to compare both with Wählby *et al.* (2012) at the worm level, as well as any other future segmentation approaches that produce results at the pixel level. The following four sets of metric results will be reported:

- mean pixel-level segmentation;
- mean pixel-level classification;
- mean worm-level segmentation; and
- mean worm-level classification.

3.1.2.3 Preprocessing and Training

Having processed the ground truth labeling, the images themselves required only normalization and cropping before use. First, the original 696×520 16-bit grayscale images were shifted and scaled such that the pixel values fall within the range $[0, 1]$. The images were also cropped to 501×501 in size, taking care to manually ensure that the illuminated region containing the worms was roughly centered. This choice of cropping ensured that only the excess dark background pixels were discarded and allowed all the images to be stored within a single tensor.

To evaluate generalization and ensure that the models are not overfitting on the training data, performance is often measured on a separate set of validation or test images. This data set, however, does not have a dedicated validation or test set. As such, 5-fold cross-validation was employed as per the discussion in Section 2.2.4. With 97 images in the set, two randomly chosen images, one from each predominant class, were omitted from training and testing to allow equally sized subsets. The remaining 95 images were randomly divided into

5 subsets, each containing 19 images, resulting in 76 training images and 19 testing images for each fold. These training/test splits were identical for all experiments.

The neural network models used in Chapter 4 were trained on this data set using mini-batch gradient descent. For the full set of 76 training images, approximately 200 mini-batches were generated and iterated through each epoch. Each mini-batch was randomly generated as it was required, ensuring that no two mini-batches were exactly the same.

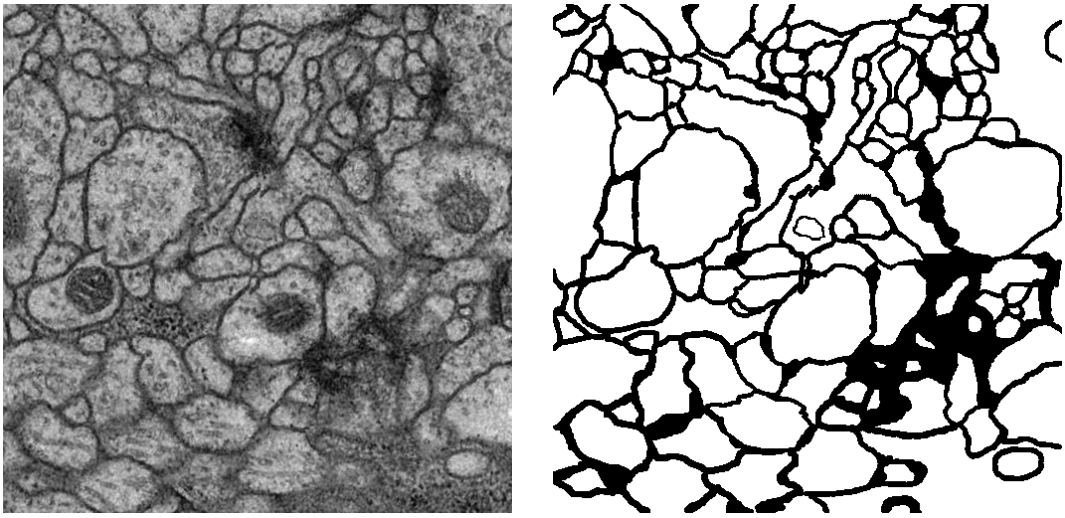
A mini-batch contained 256 image patches, with the size of patch being dependent on the model being trained (see Chapter 4). Each patch was randomly extracted from the training images, based on the class of the center pixel. First, the class requirement of the center pixel was determined, where each of the three classes had equal probability of being chosen. A random training image was then selected (provided it contained the selected class) and a patch was extracted around a random center pixel corresponding to the selected class. This sampling strategy helped to ensure that each mini-batch had a balanced class distribution.

The randomness introduced by this sampling strategy ensured that each iteration was different from the next, and greatly reduced the chances of the neural networks overfitting to the training data. It is important to note that the same seed was used for the random number generator that controlled the extraction of patches, which ensured that the training data remained consistent for each training session. Lastly, the 19 test images underwent a similar sampling strategy, with the model performance being evaluated every 5 epochs on approximately 250 mini-batches, each containing 256 image patches. This was necessary to validate that the models were not overfitting to the training sets.

3.2 Nerve Cell Membrane

The second data set used in this work was from the Electron Microscopy (EM) challenge which formed part of the International Symposium on Biomedical Imaging (ISBI) 2012 (Cardona *et al.*, 2010, 2012). This data set can be downloaded from the challenge website and will henceforth be referred to as the *nerve cell membrane data set*.

The data set consists of two sets of 30 serial section transmission electron microscopy images of size 512×512, which show portions of the ventral nerve cord from a *Drosophila* larva. Each image depicts a number of cells separated by membranes; the task is to segment the image by labeling the pixels as being either part of a cell or part of a membrane. One set of 30 images is the dedicated training set, for which a set of fully annotated binary ground truth labels are provided — for an example, see Figure 3.9.



(a) Training image (b) Ground truth

Figure 3.9: An example training image and corresponding label image from the ISBI 2012 EM segmentation challenge. In Figure 3.9b, black pixels correspond to membranes while white pixels correspond to cells.

The second set of 30 images is reserved for official testing by the organizers. The images in the test set can be used for unsupervised learning; however, no ground truth is provided to prevent supervised learning on the test set. In order to get an official score on the challenge ranking system, participants are required to submit the output generated by their segmentation approach using the test set as input.

This data set presents a number of challenges that the segmentation approach needs to overcome. Some of these challenges that are observable in the images include:

- visible noise;
- large variations in membrane width (distance between two adjacent cells); and
- large variations in membrane visibility.

3.2.1 Rand Score Thin Metric

The challenge originally reported three different metrics: the Rand error, warping error and pixel error. Evaluation of these metrics have led to the conclusion that they were not sufficiently robust to variations in membrane width (Arganda-Carreras *et al.*, 2015). These metrics were then replaced with the

Rand score thin and *information score thin* metrics[3], which better matched the organizers' qualitative judgment of segmentation quality. The challenge leaderboard is ranked by the Rand score thin metric, as Arganda-Carreras *et al.* (2015) argued that it was the more robust of the two (Arganda-Carreras *et al.*, 2015). As such, only the Rand score thin metric will be discussed and reported in this work.

Consider a single predicted segmentation map. The Rand score thin metric was chosen by the challenge organizers as a means to evaluate how well any given segmentation approach segmented each individual cell by predicting the borders between the cells. As such, the width of a border is not considered to be as important as the fact that a border exists, which meant the Rand score thin metric had to be robust to variations in border width. This robustness is attributed to the application of the watershed transform algorithm from Soille and Vincent (1990) to the predicted segmentation map, which thins the borders and groups the pixels into differently labeled regions. Similarly, connected component labeling is applied to the ground truth segmentation map, which also yields differently labeled regions corresponding to each cell in the input image. Unlike the thinned border pixels in the predicted segmentation map, the border pixels in the ground truth segmentation map do not contribute in the calculation of the Rand score thin metric.

A contingency table A is then created showing the frequency distribution of the region labels for the pixels in the predicted map versus the region labels for the pixels in the ground truth. Each entry A_{ij} represents the number of pixels that are labeled as being part of region i in the predicted map, but should be labeled as being part of region j according to the ground truth. From this contingency table, one can then calculate the joint probability p_{ij} that a randomly chosen pixel belongs in region i in the predicted map and region j in the ground truth map, given by

$$p_{ij} = \frac{A_{ij}}{N}, \tag{3.6}$$

where N is the total number of cell pixels in the ground truth map (normalization constant).

The probability that a randomly chosen pixel belongs to region i in the predicted map, is given by the marginal probability distribution $s_i = \sum_j p_{ij}$. Similarly, the probability that a randomly chosen pixel belongs to region j in the ground truth map, is given by $t_j = \sum_i p_{ij}$. Lastly, the probability that two randomly chosen pixels belong to the same region in the predicted map and the same region in the ground truth is given by $\sum_{ij} p_{ij}^2$.

There are two types of errors that can occur when segmenting the nerve cells: either multiple cells are merged together (caused by gaps in the predicted

[3]Abbreviated forms of *maximal foreground-restricted Rand score after thinning* and *maximal foreground-restricted information theoretic score after thinning*, respectively.

borders) or a single cell is split into multiple cells (caused by incorrectly placed borders). As such, the Rand score thin metric uses two components, a merge score and a split score.

The merge score, $V_{\text{merge}}^{\text{Rand}}$, is defined as the probability that two randomly chosen pixels belong to the same region in the ground truth map, given that they belong in the same region in the predicted map. As such, the merge score is defined as

$$V_{\text{merge}}^{\text{Rand}} = \frac{\sum_{ij} p_{ij}^2}{\sum_i s_i^2}. \tag{3.7}$$

Similarly, the split score, $V_{\text{split}}^{\text{Rand}}$, is defined as the probability that two randomly chosen pixels belong to the same region in the predicted map, given that they belong in the same region in the ground truth map. As such, the split score is given by

$$V_{\text{split}}^{\text{Rand}} = \frac{\sum_{ij} p_{ij}^2}{\sum_j t_j^2}. \tag{3.8}$$

The resulting Rand score thin metric, V_{α}^{Rand}, is then defined as the weighted harmonic mean between the merge and split scores, given by

$$V_{\alpha}^{\text{Rand}} = \frac{\sum_{ij} p_{ij}^2}{\alpha \sum_i s_i^2 + (1-\alpha) \sum_j t_j^2}, \tag{3.9}$$

where α was fixed as 0.5 in the calculation.

With this, it is possible to calculate the Rand score thin metric for a single predicted segmentation map. Note that the challenge organizers also allow the submission of probabilistic output, in which case the predicted segmentation map first needs to be generated by applying a threshold to the probabilistic output. Since this raises the question of which threshold to use, the organizers decided to use multiple thresholds, from 0.0 to 1.0 in steps of 0.1, where any pixel probability larger than the threshold is considered to be a cell pixel and a membrane pixel otherwise. The Rand score thin metric is then calculated at each threshold using the ground truth segmentation map and the resulting predicted segmentation map. When the metric has been calculated at all of the thresholds, the maximum Rand score thin result over the 11 thresholds is then reported as the performance of the segmentation approach.

3.2.2 Data Preparation

The data was subdivided depending on the situation it was used in. When an official evaluation was required in order to compare with other work, the neural networks were trained using all 30 available training images. For all other experiments, the training set was randomly divided into two sets of 15

images each, with one set used to train the networks and the other to test their performance by calculating the Rand score thin metric using a Beanshell script (Arganda-Carreras, 2016) which was provided by the organizers of the ISBI challenge.

In either case, the number of images in this data set is low. Thus, a different sampling approach from the *C. elegans* data set was used. Similar to other work done on this data set (Ronneberger *et al.*, 2015), one large input patch per mini-batch was favored over multiple smaller input patches, which resulted in a single input patch per mini-batch with a size of 476×476 and an output size of 292×292.[4] For each epoch, 30 randomly generated mini-batches were iterated through and used for training.

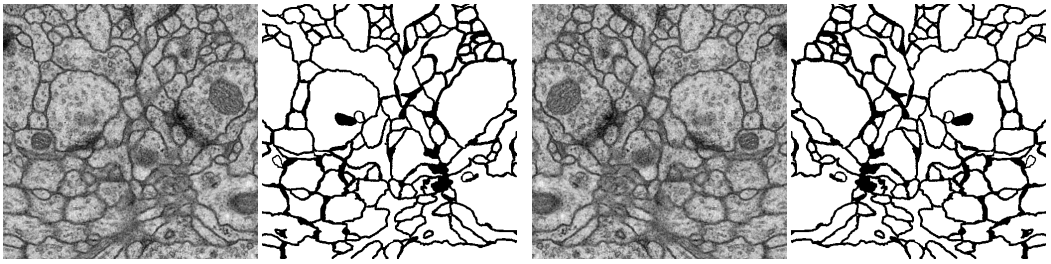
The generation of these mini-batches included sampling a random patch from the training set, followed by a random combination of image transformations. These transformations were possible horizontal mirroring, rotations[5] by multiples of 10°, and elastic deformations, with the goal of creating biologically plausible data to enrich the data set with more training examples.

First, a random image-label pair is sampled from the training set (Figure 3.10a), to which horizontal mirroring is applied in 50% of the cases (see Figure 3.10b). Following this, a random angle is sampled from the range $[0, 360)$ at multiples of 10°, which is then used to rotate the input and the label images (see Figure 3.10c).

The final transformation, elastic deformation, involves the translation of the image pixels according to a grid of displacement vectors. This transformation was also used in Ronneberger *et al.* (2015), and is achieved by first sampling the entries of two 3×3 matrices from a Gaussian distribution with a mean of zero and a standard deviation of 10 pixels. The first matrix represents the displacement of the pixels in the x-direction and the other for the y-direction. These 3×3 matrices are then resized to the dimension of the data, 512×512, with the corner entries of the 3×3 matrices corresponding to the corner entries of the resulting 512×512 matrices. Cubic interpolation is then used to fill in the missing entries in the larger matrices (the reader is referred to Szeliski (2010) for a discussion on geometric transformations with interpolation). The resulting 512×512 matrices are then used to remap the x and y positions of each pixel in the input and label images. For example, consider that the 512×512 displacement matrices indicate that a certain pixel $p_{x,y}$ should be moved -3 positions in the x-direction and 5 positions in the y-direction. In the deformed image, the original pixel $p_{x,y}$ would now be $p_{x-3,y+5}$. Note that the displacement of a pixel is not necessarily measured in integers,

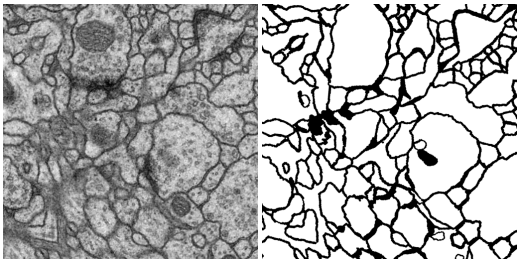
[4]Note that the gradients are averaged over the 292×292 label patch, hence the use of mini-batches over stochastic gradient descent. One input patch contains a large number of training examples.

[5]It was decided that selecting the angle of rotation at fixed multiples of 10° is sufficient to enrich the data.

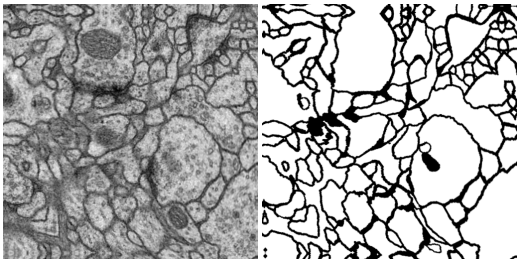


(a) Original image-label pair

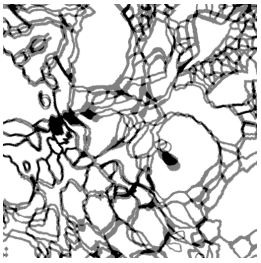
(b) Mirrored image-label pair



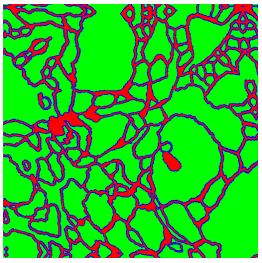
(c) Rotated image-label pair (310°)



(d) Deformed image-label pair

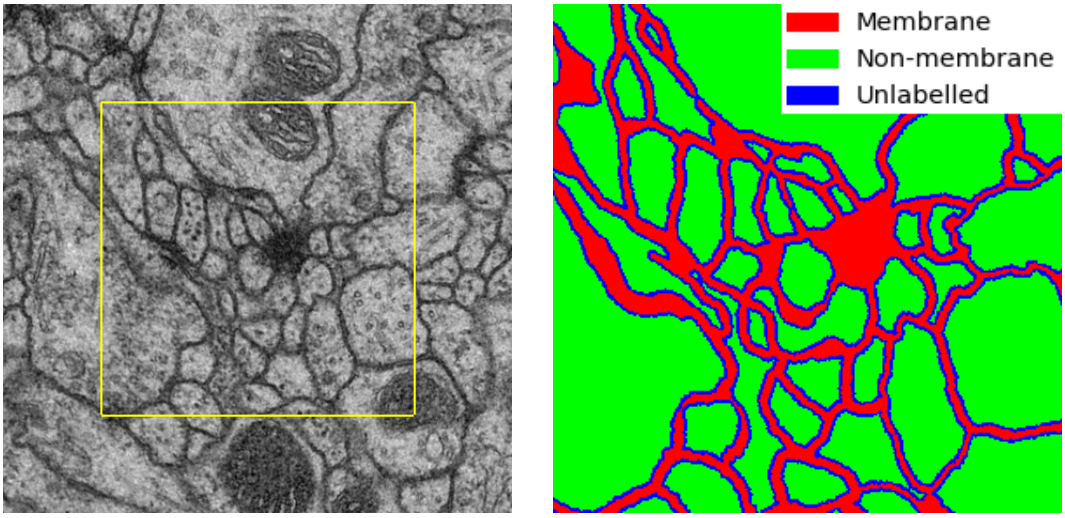


(e) Label comparison



(f) Filtered label

Figure 3.10: Resulting figures from the various enrichment techniques. Starting from the original image-label pair (Figure 3.10a), horizontal mirroring can be applied (Figure 3.10b). Then the images are rotated (Figure 3.10c), followed by an elastic deformation (Figure 3.10d). Figure 3.10e illustrates the difference between the label figures before and after elastic deformation. Finally, Figure 3.10f shows an RGB representation of the label vector after the label pixels were filtered, where red corresponds to membranes, green to cells and blue to unlabeled pixels (see Figure 3.11b for an enlarged example).



(a) Input patch (b) Label patch

Figure 3.11: An example network input patch (Figure 3.11a) where the region in the yellow square indicates the output region of the neural network, and corresponds to the pixel labels in Figure 3.11b, where green indicates cells, red the membranes and blue the unlabeled pixels.

in which case linear interpolation is used to calculate the appropriate pixel value.

Once these pixel displacements are applied to both the input image and the corresponding label image, the result is a randomly transformed image-label pair (Figure 3.10d). To better illustrate the operation performed by the elastic deformations, the label image before deformation was superimposed on the label image after deformation (Figure 3.10e).

Each transformation was accompanied by the use of linear interpolation on both the input and the label images. The original label image provided pixel labels that are assumed to be completely accurate. The interpolation that formed part of the transformations, however, meant that some of the labels in the transformed label image are no longer accurate. This necessitated a filter to remove the pixel labels that are considered to be inaccurate after transformation. As such, linear interpolation was preferred over nearest-neighbor interpolation, as it facilitated a means to filter the transformed pixels based on their values.

The label vector for each individual pixel is two-dimensional, with the first dimension representing the membrane class and the second, the cell class. To generate these label vectors, the transformed label image is divided into three regions and for each pixel value c at location (i, j), the corresponding label

vector $v(i,j)$ is defined as

$$v(i,j) = \begin{cases} (1,0), & \text{if } 0 \leq c(i,j) < 10 \\ (0,0), & \text{if } 10 \leq c(i,j) \leq 245 \\ (0,1), & \text{if } 245 < c(i,j) \leq 255 \end{cases} . \tag{3.10}$$

Here, $v(i,j) = (1,0)$ indicates that the pixel belongs to the 'membrane' class and $v(i,j) = (0,1)$ to the 'cell' class. Any pixel where $v(i,j) = (0,0)$ indicates that the pixel label was rejected as being too inaccurate, hence considering the pixel as being 'unlabeled'. It should be noted that pixels with a $(0,0)$ label vector have zero contribution to the cross entropy loss function (Equation 2.17). An illustration of this label vector is given in Figure 3.10f.

With this sampling strategy, it is now possible to randomly generate the mini-batches on demand. The randomness in the transformations ensure that any specific transformed image is extremely unlikely to reoccur during training, thus significantly reducing the risk of overfitting. An example input patch and an illustration of the corresponding label patch is given in Figure 3.11. The yellow square in Figure 3.11a indicates the output region of the network, which corresponds to the pixel labels in Figure 3.11b, where green indicates cells, red the membranes and blue unlabeled pixels. Lastly, similar to the sampling strategy for the *C. elegans* data set, the seed for the random number generators that controlled the transformations and sampling was kept constant over different training sessions.

3.3 Conclusion

This chapter introduced the two data sets that were used in this work. The first data set was the *C. elegans* live/dead assay data set. An overview was provided on how the data was collected, followed by a description of all the images and ground truth labels included in the data set. The metrics used to quantify segmentation performance were then discussed and some concerns were raised that had to be addressed before the data set could be used. First, new ground truth labels had to be created that distinguished between living and dead worms within a single image, followed by a technique to estimate the worm-level performance of the neural networks. Lastly, the remaining pre-processing of the input images and the sampling strategy of the mini-batches were discussed.

The second data set that was introduced was the nerve cell membrane data set. First, the images and ground truth labels provided in the data set were discussed, followed by an overview on how to calculate the primary metric — the Rand score thin metric — which was used to evaluate segmentation performance. Lastly, a full discussion on the data enrichment techniques that were used was provided, together with the random sampling strategy used to generate the mini-batches.

The next chapter builds a variety of neural network architectures and compares their performance using the *C. elegans* live/dead assay data set.

Chapter 4

Fully Convolutional Networks

The previous chapter introduced two bio-image data sets that are used in this work. The first data set, the *C. elegans* live/dead assay data set, contains roughly 100 microscopy images showing multiple worms in each microtiter plate well. The task of the *C. elegans* data set is to simultaneously segment the worms from the background and classify them as either alive or dead.

This chapter aims to build an understanding of how a variety of fully convolutional network (FCN) architectures compare to a standard per-pixel convolutional neural network (CNN) approach when performing semantic segmentation. A standard CNN typically produces a segmentation map one pixel at a time, which can be time-consuming, especially for large images. The elegance of FCNs lies in their ability to produce a full segmentation map in one forward pass, essentially enabling them to process entire images at a time. The architectures used in this chapter will be applied to the *C. elegans* live/dead assay data set introduced in Section 3.1.

Section 4.1 introduces four neural network architectures (1 standard per-pixel CNN and 3 FCNs) which were used to compare performance, both with respect to previous work (Wählby *et al.*, 2012), as well as with respect to architecture complexity. Section 4.2 presents the results obtained for all experiments, followed by a detailed discussion of the results in Section 4.3.

4.1 Architectures

Four neural network architectures are used in this chapter. This section will present the architectures we considered, followed by a short discussion on how these architectures were identified. The first architecture, model A, is a standard CNN used as a baseline for comparison with the three FCNs: model B, model C and model D. The three FCN models grow in complexity by adding pooling and corresponding deconvolution steps, starting with model B having no pooling, followed by model C with one pooling step, and finally model D with three pooling steps. All of the layers for the four models use PReLUs as

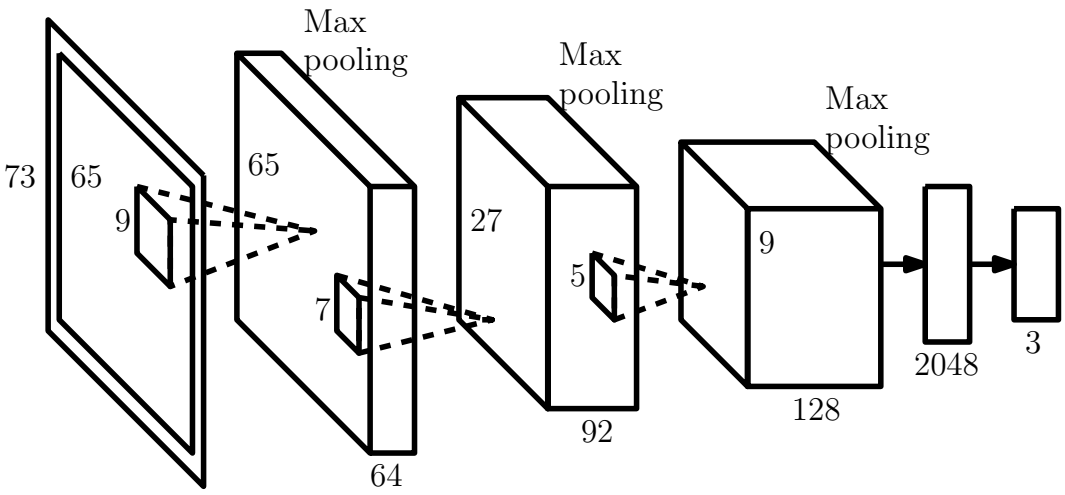


Figure 4.1: Model A (≈ 4.5 million parameters). The 65×65 input patch was zero-padded to 73×73. All convolutional kernel strides are 1. All max-pooling kernels are 2×2 with a stride of 2. The model contains three convolutional layers, three max-pooling layers, one fully connected layer and one softmax output layer. This model produces a single pixel prediction for a given input patch (see Figure 4.2a).

activation function, unless specified otherwise.

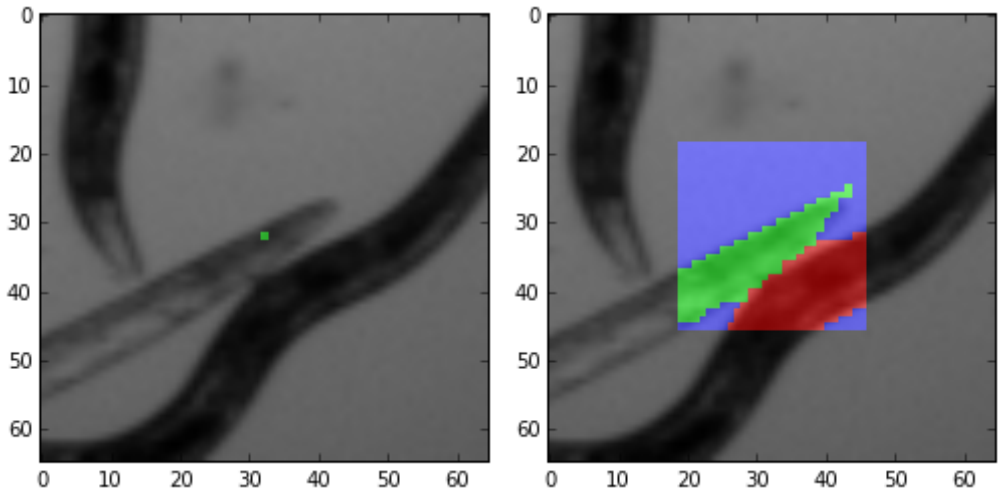
Model A

The first model, referred to as model A, is a standard CNN based on the model used in Ciresan *et al.* (2012). The model contains three consecutive convolutional layers, each followed by a max-pooling layer. The final max-pooling layer is followed by a fully connected layer and then finally a softmax output layer. Figure 4.1 depicts the architecture of model A.

The model accepts an input patch of size 65×65, which is then zero padded by half the kernel size of the first layer to allow predictions on near-border pixels. It then produces an output prediction for a single pixel, typically for the center pixel of the input patch (see Figure 4.2a). In order to generate a complete segmentation map for any given image, the model has to be evaluated for each pixel using the corresponding input patch. This model was used to provide a performance baseline to compare the FCNs against.

Model B

The first FCN model, model B, is considered the simplest of the three FCNs as it contains no downsampling (max-pooling layers) or upsampling (deconvolutional layers) steps. Furthermore, it is also the shallowest architecture of the three FCNs, consisting only of three convolutional layers and a ConvSoftmax layer. A full depiction of model B is provided in Figure 4.3.



(a) Single pixel label (b) Patch of pixel labels

Figure 4.2: Single pixel label (Figure 4.2a) versus a patch of pixel labels (Figure 4.2b). A standard convolutional neural network will receive the label of the center pixel (represented by the green colored pixel in Figure 4.2a) as target for the corresponding input patch. An FCN on the other hand will receive a patch of pixel labels (represented by the red, green and blue colored pixels in Figure 4.2b) as target for the corresponding input patch. Red, green and blue correspond to the live, dead and background classes, respectively.

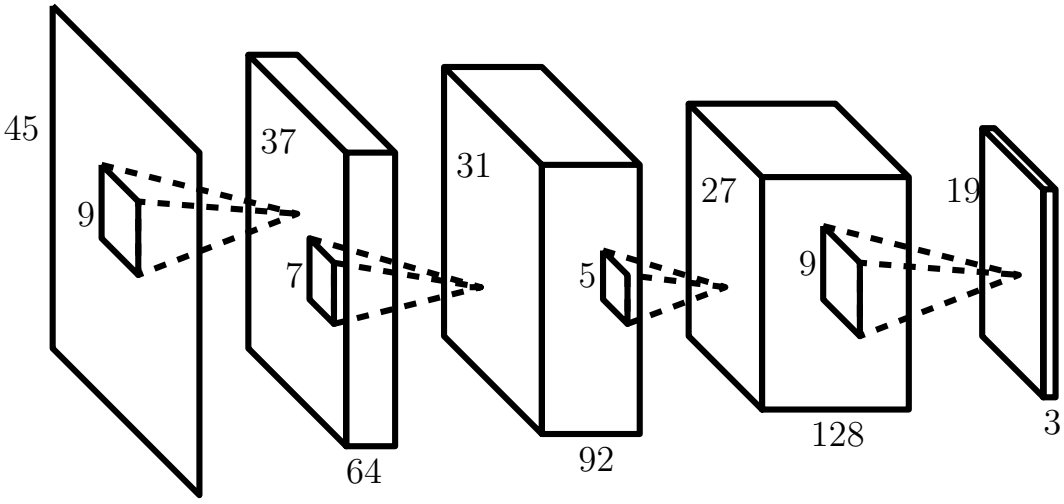


Figure 4.3: Model B ($\approx$ 600000 parameters). All convolutional kernel strides, including the ConvSoftmax output layer, are 1. The model contains three regular convolutional layers and one ConvSoftmax output layer. This model produces a patch of pixel predictions for a given input patch (see Figure 4.2b).

In the case of FCNs, it is important to distinguish between the contextual window of the network and the input patch. The term *contextual window* refers to the region in the input patch that the network considers in order to produce an output prediction for a single pixel. For regular convolutional neural networks, such as model A, the contextual window is equivalent to the input patch of the network, since the entire input patch is considered to produce a single pixel prediction. In FCNs, however, this is not the case. Notice in Figure 4.3 that the ConvSoftmax layer uses a filter size of 9×9 instead of the 1×1 filters of models C and D presented later in this section. This choice of filter size was necessary in order to increase the size of the contextual window of the network, which is otherwise typically achieved through the use of downsampling steps (the max-pooling layers in models C and D). The resulting contextual window of model B was much smaller than that of model A, with a size of 27×27.

Another concern that emerged from training on a patch of pixel labels for any given input patch is the class imbalance problem (Japkowicz and Stephen, 2002). The class imbalance problem is a well known problem in machine learning where a trainable classifier could become biased towards the majority class in a highly imbalanced data set. This problem was mitigated for the data sets in Chapter 3 through the use of a balanced sampling strategy based on the class of the center pixel of an input patch. Training on a patch of pixel labels, however, rather than a single pixel label could potentially reintroduce this problem for FCNs, since the class distribution within the label patch is not controlled.

The potential occurrence of the class imbalance problem was investigated by observing how the size of the input patch affects the performance of the network. In addition to the input patch size of 45×45 shown in Figure 4.3, one smaller patch size and one larger patch size were also explored. With the background being the majority class in the *C. elegans* data set, the larger the input patch (and corresponding label patch), the more background pixels will be included relative to the number of worm pixels. This should increase the risk of the model becoming biased towards the background class as the input patch size becomes larger.

The smaller input patch was chosen to correspond to the contextual window of model B, hence an input patch size of 27×27. This choice of input patch size resulted in the network producing an output for a single pixel (an output size of 1×1), effectively training model B as a standard convolutional neural network. This allows a direct comparison between training on single pixel labels versus training on a patch of pixel labels for a given input patch (Figure 4.2).

The larger input patch was chosen in such a way that most of the available GPU memory was used given the current training strategy. This resulted in a patch size of 65×65. Of the three input patch sizes, it is expected that this choice of patch size should yield a trained model with the most bias towards the background class. The results for this investigation of the class

imbalance problem is presented in Section 4.2 and discussed in more detail in Section 4.3.3.

Model C

Expanding on model B, a number of layers were added to the architecture to make it deeper. First, a max-pooling layer was inserted after the first convolutional layer. An alternative approach of downsampling through the use of larger filter strides in the convolutional layer was also tested; however, it did not provide a notable performance difference compared to using a max-pooling layer.

Second, the downsampling performed by the max-pooling layer effectively reduces the resolution of the information traveling through the neural network. In order to produce a segmentation map of the same resolution as the input image, this reduction in resolution needs to be undone through the use of an upsampling operation. The architecture of model C thus includes a trainable upsampling operation, a deconvolutional layer, inserted before the ConvSoftmax layer.

Lastly, the architecture of model C also includes two additional convolutional layers, making the architecture a total of eight layers deep. Consequently, model C is structured as follows: First a convolutional layer, followed by a max-pooling layer. After the max-pooling layer come four consecutive convolutional layers, then a deconvolutional layer, and finally a ConvSoftmax layer. Note that the filter size of the ConvSoftmax layer was reduced to 1×1, as the max-pooling layer and the increase in network depth provided an adequately sized contextual window. A full depiction of the architecture for model C is provided in Figure 4.4.

Model D

The architecture for the final model was determined based on its potential contextual window with each additional pooling layer. It was determined that a network architecture with three max-pooling layers would have a contextual window of similar size to that of an average fully grown worm. Like with model C, each pooling layer was accompanied by a deconvolutional layer later in the network.

Considering that the texture of the worm partially contributes towards the overall predicted class, it was concerning that performing three downsampling operations could discard critical information. To remedy this, a similar approach as in FCN-8s and FCN-16s of Long *et al.* (2015) was used by creating multiple pathways for information to travel through the network. Consider a basic neural network containing four convolutional layers: `D1`, `D2`, `D3` and `D4`, connected in this order. The main path through this network would be from `D1` to `D2`, `D2` to `D3` and `D3` to `D4`. An alternative pathway can be created by

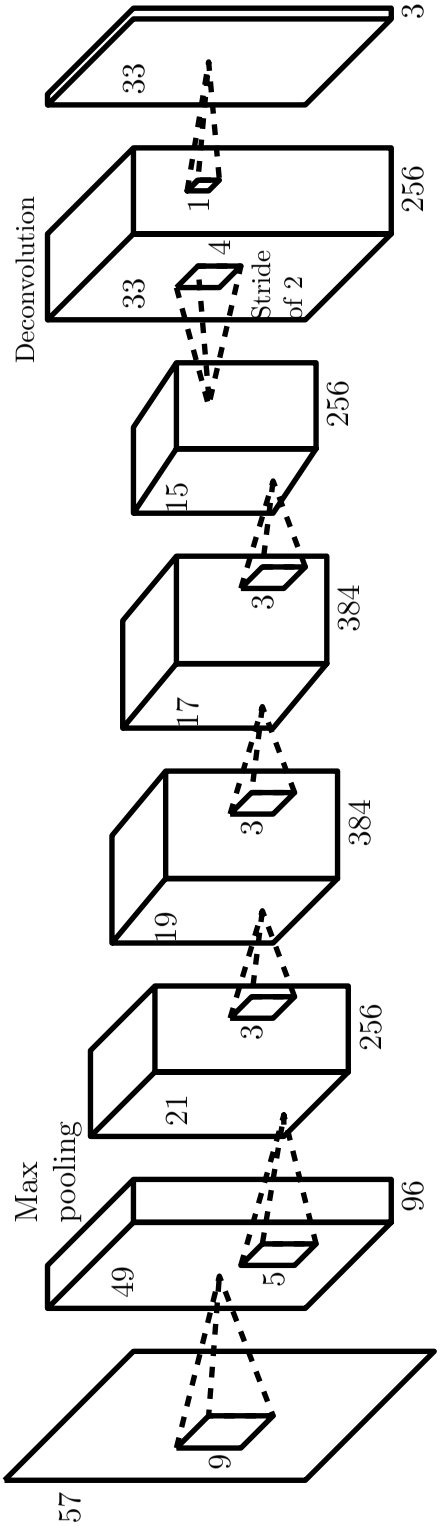


Figure 4.4: Model C (≈ 4.5 million parameters). All convolutional kernels including the ConvSoftmax kernel have a stride of 1. The single max-pooling layer has a 2×2 kernel with a stride of 2. This model produces a patch of pixel predictions for a given input patch (see Figure 4.2b).

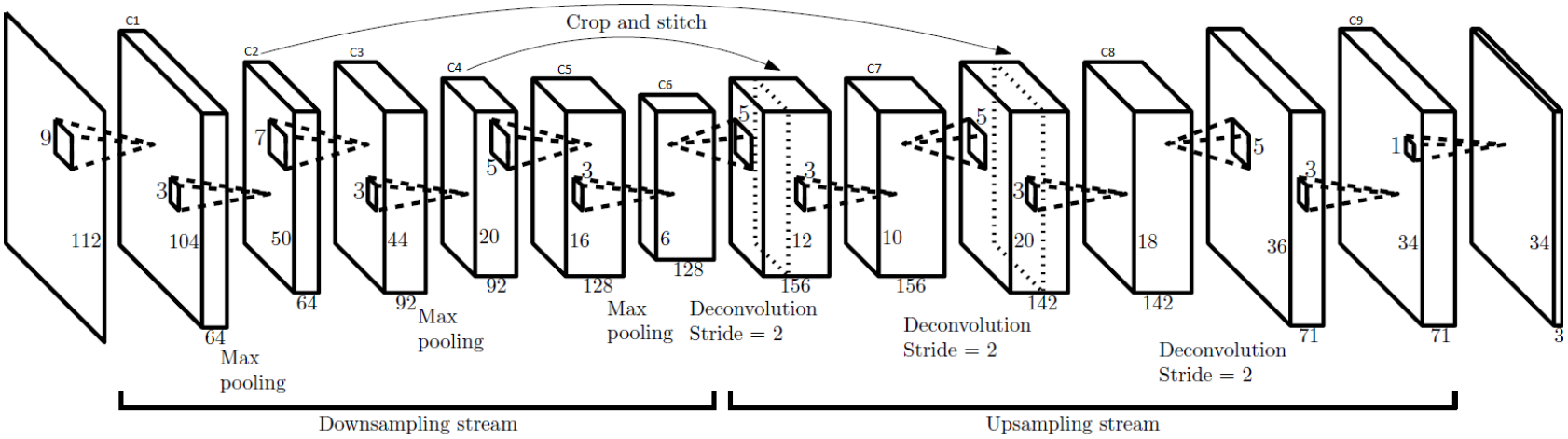


Figure 4.5: Model D ($\approx$ 2 million parameters). All convolutional kernels including the ConvSoftmax kernel have a stride of 1. The three max-pooling layers have 2×2 kernels with a stride of 2. Convolutional layer `C7` has a skip connection from layer `C4`. Similarly, convolutional layer `C8` has a skip connection from layer `C2`. Skip connections are implemented using the crop-and-stitch technique (see Figure 4.6). This model produces a patch of pixel predictions for a given input patch (see Figure 4.2b).

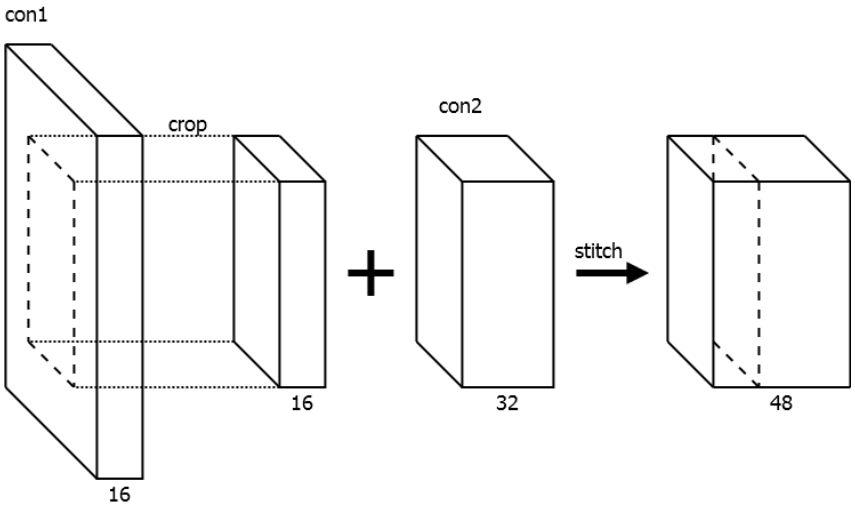


Figure 4.6: An illustration of the crop-and-stitch technique. The larger feature maps of `con1` are first cropped to the size of the feature maps of `con2`. Now that the feature maps are all the same size, they are stitched together to create one larger collection of feature maps.

having a layer, for instance `D1`, be connected to later layers not directly after it in the main pathway, for instance `D4`. The connection between `D1` and `D4` skips the intermediate layers, `D2` and `D3`, and is referred to as a *skip connection*. These skip connections were added to the architecture of model D to act as bridges for relevant information to bypass the deeper levels of the network. The concept of skip connections have been around for years — often referred to as skip-layer connections, shortcut connections or residual connections — such as in Intrator and Intrator (2001), where a skip connection is used from the input of the network directly to the output layer.

The skip connections were implemented using what is referred to as the *crop-and-stitch* technique, which involves the feature maps of two or more convolutional layers. First, the larger feature maps are 'cropped' to the size of the smallest feature map involved in the technique. After cropping, all of the feature maps are 'stitched' together, creating a larger collection of feature maps that can be used as input to another layer.

For instance, consider the two convolutional layers in Figure 4.6, `con1` and `con2`, with `con1` having 16 feature maps of size 30×30 and `con2` having 32 feature maps of size 18×18. First, the feature maps of `con1` are cropped to the size of the feature maps of `con2`, namely 18×18. Now that the feature maps of both layers are the same size, the feature maps of both layers are stitched together to create a collection of 48 feature maps $(16 + 32)$ of size 18×18.

The full architecture of model D is depicted in Figure 4.5. Model D can be divided into two streams, a downsampling stream and an upsampling stream. The downsampling stream contains six convolutional layers, with a max-pooling layer after layers `C1`, `C3` and `C5`. The upsampling stream has

three deconvolutional layers, each followed by a convolutional layer, and finally a ConvSoftmax layer as output.

Each deconvolutional layer in this instance halves the number of feature maps of the layer that comes before it. The crop-and-stitch technique is applied using the 92 feature maps of the convolutional layer `C4` and the 64 feature maps of the first deconvolutional layer, creating a combined input of 156 feature maps to layer `C7`. Similarly, the crop-and-stitch technique is applied using layer `C2` and the second deconvolutional layer to create a combined input of 142 feature maps to layer `C8`. Unlike the previous models, the square size of the input patches used to train model D was required to be some multiple of 8. This choice of edge length avoided the possibility of extensive information loss along the borders when using three downsampling steps.

Choice of Models

This section will briefly outline how the models identified above were developed and selected for use in this study. Initially, a number of basic CNNs were designed and tested as part of the debugging phase. Following the completion of the debugging phase, model A was then created based on the model presented in Ciresan *et al.* (2012). The architecture of model A contained one less convolutional and pooling layer (three instead of four), as four pooling steps seemed too many for a starting model on the *C. elegans* data set.

At this stage, we started investigating FCNs with model B being the first FCN architecture that was created. Model B was then used as an architectural basis in three small scale experiments. The first experiment was briefly discussed when presenting model B, namely training the model using three input patch sizes. The second experiment involved inserting a pooling layer into the architecture of model B, with the goal of comparing various upsampling techniques. The following three techniques were considered:

- Resizing the output of the network with interpolation (a fixed upsampling technique as post-processing);
- Inserting a deconvolutional layer before the output layer (a one-step trainable upsampling technique);
- Resizing the feature maps of the second-to-last convolutional layer by duplicating the values in each spatial dimension, followed by a 2×2 convolution (a two-step trainable upsampling technique).

Of these three techniques, the deconvolutional layer was found to provide the best performance. The final experiment involved comparing two downsampling techniques: a max-pooling layer and a convolutional layer with a filter stride of 2. Both techniques showed fairly similar performance, and as such, we opted for the max-pooling layer as it is the simpler option of the two and contains no trainable parameters.

The architecture of model C then followed from the experiments performed using model B, in that the architecture of model B was deepened by inserting a max-pooling layer, a deconvolutional layer and two additional convolutional layers. The results of model C seemed to suggest that making the architecture deeper with more pooling and deconvolutional layers was beneficial for performance. As such, we decided to create an FCN architecture with the same number of pooling layers as in model A, with each pooling layer accompanied by a corresponding deconvolutional layer later in the network. At this stage, the architecture did not contain any skip connections, which raised the concern of whether or not the model would be able to generate high level segmentations after three pooling steps. After much consideration, taking into account findings from previous work such as Long *et al.* (2015) (as discussed in Section 2.5.2), two sets of skip connections were incorporated to obtain model D.

4.2 Experiments and Results

This section will present the results obtained from experimentation with the described architectures. It follows a similar structure as in Section 4.1, where all experiments performed with a certain model are described under the corresponding model heading. Some models were utilized in additional experiments used to illustrate certain concepts when working with neural networks.

The experiments were subdivided into major experiments and minor experiments. The major experiments involved utilizing the 5-fold cross validation strategy set out in Section 3.1.2.3 to train and evaluate each model. The results of the major experiments are summarized in Table 4.1, and are presented as the *mean* $\pm$ *standard deviation* (over the 5 folds). These results were used to compare performance against Wählby *et al.* (2012) in Section 4.3.

The minor experiments were mainly used to illustrate certain concepts such as the importance of balanced data and how it affects the performance of the neural network. The minor experiments only utilized the first fold of the 5-fold cross validation strategy set out in Section 3.1.2.3. The results for the minor experiments involving model B are presented in Table 4.2.

The connection weights of each model were randomly initialized using the normalized initialization strategy presented in Glorot and Bengio (2010). The models were trained for 200 epochs utilizing mini-batch gradient descent as described in Section 2.2.2 and the standard cross-entropy loss function. Each epoch iterated over approximately 200 mini-batches, with each mini-batch containing 256 randomly selected image patches. These hyperparameters were determined based on machine memory availability and the change in training and test performance over time. In particular, we observed in preliminary tests that 200 epochs was generally more than sufficient to ensure that at the end of training, all of the models showed very little change in their training and

Table 4.1: The results for the major experiments utilizing 5-fold cross validation. The results are divided into four sets: pixel-level and worm-level segmentation (Seg.) and classification (Class.). Included in the worm-level segmentation set is the segmentation correctness (Seg. Corr.) measure from Wählby *et al.* (2012). The results for `Model A (orig)` were generated using the original binary segmentation masks, provided for completeness only (hence the use of italics). Results are provided in the form *mean* ± *standard deviation*, calculated over the 5 folds. Results in bold indicate the largest mean result over the 4 models for each metric.

	Pixel Level						
		Model A (orig)	Model A	Model B	Model C	Model D	Wählby *et al.* (2012)
Seg.	Accuracy	*0.9806±0.0012*	0.9820±0.0004	0.9902±0.0005	**0.9904±0.0008**	0.9898±0.0011	-
	Precision	*0.7955±0.0113*	0.8096±0.0038	0.9328±0.0122	0.9340±0.0101	**0.9362±0.0051**	-
	Recall	*0.9842±0.0044*	**0.9820±0.0049**	0.9201±0.0088	0.9296±0.0095	0.9425±0.0104	-
	F-Measure	*0.8798±0.0058*	0.8875±0.0039	0.9263±0.0046	0.9316±0.0064	**0.9393±0.0069**	-
Class.	Accuracy	*0.8844±0.0176*	0.8619±0.0071	0.8338±0.0067	0.8437±0.0077	**0.8666±0.0272**	-
	Precision	*0.8920±0.0420*	0.8582±0.0342	0.8265±0.0210	0.8341±0.0277	**0.8647±0.0305**	-
	Recall	*0.9016±0.0233*	0.8569±0.0445	0.8312±0.0207	0.8482±0.0241	**0.8696±0.0503**	-
	F-Measure	*0.8962±0.0253*	0.8559±0.0097	0.8284±0.0101	0.8400±0.0092	**0.8668±0.0185**	-
	Worm Level						
		Model A (orig)	Model A	Model B	Model C	Model D	Wählby *et al.* (2012)
Seg.	Accuracy	*0.9989±0.0000*	0.9990±0.0000	**0.9994±0.0000**	**0.9994±0.0000**	**0.9994±0.0001**	-
	Precision	*0.7937±0.0060*	0.8154±0.0035	**0.9285±0.0120**	0.9277±0.0097	0.9236±0.0051	-
	Recall	*0.9670±0.0081*	**0.9794±0.0065**	0.9190±0.0088	0.9281±0.0105	0.9297±0.0145	-
	F-Measure	*0.8698±0.0034*	0.8879±0.0044	0.9213±0.0048	**0.9254±0.0079**	0.9235±0.0111	-
	Seg. Corr.	*0.9409±0.0153*	0.9588±0.0150	0.9686±0.0067	**0.9730±0.0119**	0.9662±0.0219	0.9400
Class.	Accuracy	*0.8970±0.0169*	0.8872±0.0071	0.9015±0.0182	0.8984±0.0237	0.8894±0.0277	**0.9700**
	Precision	*0.9142±0.0427*	0.8795±0.0352	0.8798±0.0189	**0.8830±0.0193**	0.8706±0.0269	0.8300
	Recall	*0.8990±0.0336*	0.8806±0.0489	**0.9147±0.0236**	0.9025±0.0327	0.8973±0.0404	-
	F-Measure	*0.9057±0.0242*	0.8781±0.0089	**0.8966±0.0124**	0.8914±0.0159	0.8828±0.0212	-

test performance. Lastly, the adaptive learning rate technique ADADELTA (see Section 2.3) was used to speed up learning and to eliminate the need for setting the global learning rate hyperparameter.

Model A

Model A was used in two major experiments using the 5-fold cross validation strategy set out in Section 3.1.2.3. The first experiment used the binary segmentation masks as training target — the results of which are summarized in Table 4.1 as `Model A (orig)`. Note that the output produced by model A detected both living and dead worms in a single image, which contradicts the uniform worm class suggested by the binary segmentation masks. This misclassification of the worms in a single image raised the following two scenarios when considering the true labels of the worms:

1. The model classifies the worm correctly, but it is considered incorrect according to the uniform class of the binary segmentation mask.

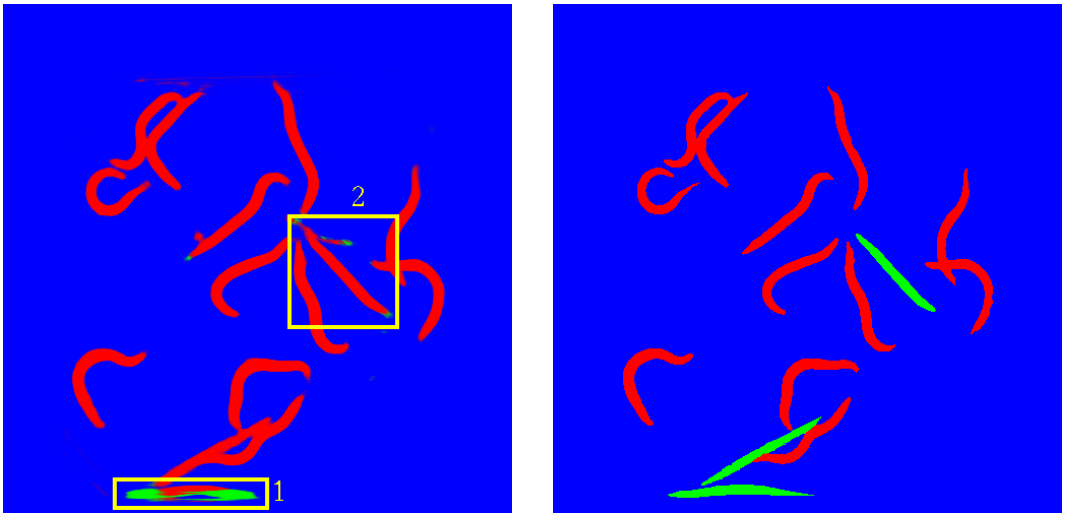
2. The model classifies the worm incorrectly, but it is considered to be correct according to the uniform class of the binary segmentation mask.

Consider the example output in Figure 4.7, produced by model A from the first experiment. An example of the first possibility is represented by worm 1, where a truly dead worm was classified as such, but it is seen as incorrect as it occurs in a predominantly live image. Similarly, an example of the second possibility is represented by worm 2, where a truly dead worm was classified as being alive in a predominantly live image. This phenomenon is what motivated the need to generate the relabeled ground truth, as set out in Section 3.1.2.

The second experiment involving model A used the relabeled segmentation masks as training target — these results are summarized in Table 4.1 as `Model A`. The results from this experiment were used as a baseline to compare the FCNs against. The model achieved comparable performance to the first experiment with regards to segmentation, but performed worse on classification. Of all the other models, model A achieved the highest segmentation recall for both the pixel level and the worm level, at 98.2% and 97.94% respectively. Model A also achieved the worst segmentation precision on both levels, at 80.96% and 81.54%.

Model B

Model B was used in both a minor and a major experiment. The minor experiment investigated the class imbalance problem by observing how the size of the input patch affects the performance of the model. As mentioned in Section 4.1, three patch sizes were considered: 27×27, 45×45 and 65×65. These experiments were performed using only the first fold, hence the metric results given in Table 4.2 not showing standard deviations.



(a) Model A output (b) Relabeled ground truth

Figure 4.7: The output produced by model A when trained using the binary segmentation masks for a predominantly live image. The output shows the two possible errors that can occur in comparison with the relabeled ground truth. The color scheme follows the same convention as in Figure 3.5e, where red represents the live class, green the dead class and blue the background. Worm 1 is an example where the worm as a whole would be classified as dead, which is correct according to the relabeled ground truth but not according to the binary segmentation masks. Worm 2 is an example where the worm as a whole would be classified as live, which is incorrect according to the relabeled ground truth but considered correct according to the binary segmentation masks.

It is immediately clear from Table 4.2 that the model with an input patch size of 45×45 performed the best overall. Starting with the smallest input patch size, the 27×27 case achieved comparable metric values to the 45×45 case, with the larger differences being a much lower segmentation precision and higher segmentation recall. It is worth noting that the 27×27 case behaves similarly to model A — a standard convolutional neural network with a much larger contextual window — in that model A also exhibits low segmentation precision and high segmentation recall.

The larger input patch size, the 65×65 case, yielded much worse segmentation and classification metric values overall. This choice of input patch size, however, did achieve the best segmentation precision for both the pixel level and the worm level of the three sizes. An important observation is that where the 27×27 case achieved a low segmentation precision and high segmentation recall, the 65×65 case achieved the opposite, a high segmentation precision and low segmentation recall.

The major experiment thus used the 45×45 input patch size version of

Table 4.2: The results for the minor experiment investigating the effect of input patch size on the performance of model B. The metric result in bold indicates the best performing case for that metric. The results are divided into four sets: pixel-level segmentation (Seg.) and classification (Class.), and worm-level segmentation and classification. Included in the segmentation results is the segmentation correctness (Seg. Corr.) measure from Wählby *et al.* (2012).

		Pixel Level			Worm Level		
		27 × 27	45 × 45	65 × 65	27 × 27	45 × 45	65 × 65
Seg.	Accuracy	0.9811	**0.9881**	0.9846	0.9989	**0.9993**	0.9991
	Precision	0.7703	0.8819	**0.9526**	0.7969	0.8894	**0.9486**
	Recall	**0.9800**	0.9433	0.8225	**0.9790**	0.9404	0.8175
	F-Measure	0.8625	**0.9116**	0.8828	0.8759	**0.9103**	0.8694
	Seg. Corr.	-	-	-	0.9541	**0.9618**	0.8206
Class.	Accuracy	0.8379	**0.8477**	0.7324	0.9045	**0.9236**	0.8206
	Precision	0.8028	**0.8316**	0.7078	0.8899	**0.9245**	0.7837
	Recall	**0.8247**	0.8146	0.6906	0.8818	**0.8909**	0.7909
	F-Measure	0.8136	**0.8230**	0.6991	0.8858	**0.9074**	0.7873

model B with the 5-fold cross validation strategy set out in Section 3.1.2.3. Model B was trained using the relabeled segmentation masks as training targets, with the results of the experiment summarized in Table 4.1 as `Model B`. The model showed mostly improved segmentation metric results over model A, achieving the best worm-level segmentation accuracy (tied with model C and D) and precision with a mean of 99.94% and 92.85%, respectively. It did, however, have a lower segmentation recall than model A.

Model B also improved on all of the worm classification metrics over model A, achieving the best worm classification recall and F-measure with means of 91.47% and 89.66%, respectively. It also achieved the best worm classification accuracy of the four models with a mean of 90.15%. Lastly, model B showed the worst performance on the pixel-level classification metrics compared to the other three models.

Model C

Building on the results of model B, the effect of increasing the number of layers in the architecture and adding a single downsampling and upsampling step was explored. Model C was used in a major experiment using the 5-fold cross validation strategy set out in Section 3.1.2.3. The results of the experiment are summarized in Table 4.1 as `Model C`. The model showed improved pixel classification results and slightly worse worm classification results compared to those of model B, with the exception being the worm classification precision where model C scored the best with a mean of 88.30%.

Model C also showed comparable segmentation results to model B, with

slight improvements on all of the pixel segmentation metrics and most of the worm segmentation metrics. The model achieved the best pixel-level segmentation accuracy with a mean of 99.04%, as well as the best worm-level segmentation F-measure with a mean of 92.54%. Lastly, model C was also able to achieve the best performance on the segmentation correctness metric introduced in Wählby *et al.* (2012) with a mean of 97.30%.

Model D

Since model C showed positive results from increasing the depth of the network and adding a downsampling step to increase the contextual window of the network, model D expanded on model C by adding two more downsampling layers. Model D was used in a major experiment using the 5-fold cross validation strategy set out in Section 3.1.2.3. The results of the experiment are summarized in Table 4.1 as `Model D`. The model showed a notable improvement on all of the pixel-level classification metrics, showing the highest mean metric values among the four models trained on the relabeled segmentation masks.

The model also improved on most of the pixel-level segmentation metrics, achieving the highest precision and F-measure with means of 93.62% and 93.93%, respectively. Model D also shows the best pixel-level segmentation recall of the FCN models with a mean of 94.25%. Although model D managed to produce the best mean metric values for most of the pixel-level metrics, it was still outperformed on the worm-level by both model B and model C for both segmentation and classification.

4.3 Discussion

This section will provide a detailed discussion on all of the results that were obtained during experimentation with the four models. Since the goal of this chapter was to improve on the ad-hoc image processing pipeline presented in Wählby *et al.* (2012), an overview of the results they reported is given in Section 4.3.1. This section also investigates which of their results are the most appropriate to use as a benchmark. Section 4.3.2 then provides a comparison between the best performing models in this chapter and the results reported in Wählby *et al.* (2012). Finally, Section 4.3.3 discusses a number of observations that were made based on the behavior of the various models.

4.3.1 Previous Work

Only one set of results was found on the *C. elegans* live/dead assay data set at the time of this work, namely the results reported by Wählby *et al.* (2012). They presented an ad-hoc image processing pipeline that can be reduced to

two critical processing steps: segmentation of each worm from the background, followed by the live or dead classification of the individual worms. Provided that the *C. elegans* live/dead assay data set is the only data used for this study, only results that were generated on this data set would be deemed appropriate for use as a benchmark.

Wählby *et al.* (2012) reported three metric results: the segmentation correctness measure as discussed in Section 3.1.1, the worm-level classification accuracy and the worm-level classification precision. Recall from Section 3.1.1 that the segmentation correctness measure is calculated by using a threshold of 0.8 on the worm-level segmentation F-measure of each worm. Wählby *et al.* (2012) calculated their segmentation correctness measure based on the segmentation of the worms in this data set, achieving a segmentation correctness of 94%.

The other metric results reported by Wählby *et al.* (2012) are a worm-level classification accuracy of 97% and a worm-level classification precision of 83%. Unfortunately, these metric results were calculated using data that was not made available to the public. The authors claim to have verified the feasibility of their classifier by predicting the class of the worms in this data set, but did not report any results. Instead, they performed their own high-throughput screening experiment (real-world situation) and tested their classifier on the newly obtained images, achieving the reported accuracy and precision.

The problem with using the binary classification metrics, is that the values obtained for the metrics are only meaningful if they were calculated over a balanced test set. For instance, consider a data set containing 100 positive examples and 900 negative examples. Applying a classifier on this data set that predicts each example as negative will yield an accuracy of 90%, giving the impression that the classifier performs very well. This is not the case when the same classifier is tested on a balanced test set, as it will only achieve an accuracy of 50%. Consider the same skewed data set — 100 positive examples and 900 negative examples — and a classifier that predicts 90% of both the positive and the negative examples correctly. Applying this classifier to this skewed data set will yield a precision of 50% ($TP = 90, FP = 90$), giving the impression that the classifier has a poor precision. This is not the case when the same classifier is tested on a balanced test set, as it will achieve a better precision of 90%. Since no information was provided by Wählby *et al.* (2012) on the class distribution within their private data, the reliability of their classification accuracy and precision comes into question.

The two metric values that Wählby *et al.* (2012) provided were used to approximate the ratio between the positive and negative classes. Based on the ratios obtained, it seems that even with perfect classification recall, the test

data they used was highly skewed towards the negative class.[1] This means that the classification accuracy they reported is likely skewed by the imbalance in the test data, making their classifier appear better. The classification precision on the other hand might be skewed differently, in that with the majority class being negative, this could lead to a larger number of false positives and consequently a much lower precision (as in the example above).

This situation simply highlights the importance of providing enough information on the data that were used for testing, as well as the importance of reporting enough results to provide an indisputable indication of classifier performance. Providing enough information is especially important in situations where classification results are reported using non-public data, which is the case since the ground truth that was used to train and/or evaluate the classifier in Wählby *et al.* (2012) was not specified nor provided.

4.3.2 Result Comparison

Based on the investigation into the results reported in Wählby *et al.* (2012), only the segmentation correctness measure was thus considered to be appropriate to use as a comparative benchmark. Their model's reported classification accuracy and precision are still included in Table 4.1, but only for completeness. Both the standard convolutional neural network (model A) and the three FCNs (models B, C and D) that were tested in this work were able to outperform the segmentation correctness measure of 94% reported by Wählby *et al.* (2012). Model C achieved the best segmentation correctness measure with a mean of 97.30%.

With regards to the worm classification accuracy of 97% and worm classification precision of 83% reported by Wählby *et al.* (2012), the highest worm-level classification accuracy was achieved by model B with a mean performance of 90.15%. On the other hand, all four models were able to outperform the 83% worm classification precision, with model C achieving the best mean performance of 88.30%. The disparity observed in the classification accuracy between model B and the classifier used by Wählby *et al.* (2012) is most likely attributed to a difference in the class distribution of the test data that were used, as discussed in the previous section. A similar argument can be made for the disparity observed in the classification precision.

4.3.3 Observations

There are a number of observations that can be made from the results obtained for each model. Recall from the beginning of this chapter, that FCNs are considered to be an elegant solution to performing semantic segmentation,

[1] A recall of 100% predicts a ratio of approximately 1 positive example for every 6 negative examples given the accuracy and precision reported. Our results suggest that our test data contains approximately 9 positive examples for every 10 negative examples.

given their ability to generate a full segmentation map in one forward pass. This is compared to CNNs, which produce a segmentation map one pixel at a time, thus requiring multiple forward passes. To quantify the difference in processing time between a CNN and an FCN, both model A and model C was tasked with generating a segmentation map for a single validation image. The CNN required about 63 seconds to generate a full segmentation map, while the FCN was able to generate the same size output in about 1 second (98% reduction in processing time). This result supports the elegance of FCNs to perform semantic segmentation, and their superiority over CNNs on this type of task. The remainder of the observations can be grouped together under four themes: the task complexity, the class imbalance problem, uneven illumination and the contextual window.

Task Complexity

The first observation that can be made is the notable performance difference of model A when trained on the binary segmentation masks compared to training on the relabeled segmentation masks (`Model A (orig)` versus `Model A`, see Section 4.2 and Table 4.1). The two versions of model A achieve comparable segmentation performance; however, the version of model A trained on the binary segmentation masks performed better on the classification metrics. Since the two versions of the model started off being identical before training, the difference in performance has to be attributed to changing the training target.

One hypothesis for this difference is that the binary segmentation masks convey a much simpler classification task than the relabeled segmentation masks. The binary segmentation masks do not distinguish between living and dead worms within a single image. The neural network could exploit this information by predicting the class of a worm based on the class of the neighboring worms that are visible within the input patch. This is not possible with the relabeled segmentation masks, as the class of the worm under consideration is less dependent on the class of neighboring worms.

The difference in performance could also be attributed to the fact that the two versions of model A were evaluated using different ground truth. Since `Model A (orig)` was trained on the binary segmentation masks, the resulting model was also evaluated using the binary segmentation masks. Similarly, the version of model A that was trained on the relabeled segmentation masks were also evaluated using the relabeled segmentation masks.

Class Imbalance Problem

An interesting observation from Table 4.1 is that both variations of model A show an exceptionally high pixel segmentation recall and a low precision. A similar observation can be made in Table 4.2 for the 27×27 input size version

of model B. The opposite behavior can be observed in Table 4.2 for the 65×65 input size version of model B, which shows an exceptionally high pixel segmentation precision and a low recall.

At first, when only observing the results from model A, it seemed likely that a model trained on a single pixel label for a given input patch had no notion of where the borders between classes are. This was thought to be the reason why FCNs are better for segmentation, in that the borders between classes can be captured in the patch of pixel labels that are used for training. Although there does not seem to be evidence to rule out this hypothesis, the additional results observed in Table 4.2 of model B does suggest another possibility.

It would appear that the class imbalance problem seems to manifest slightly differently in semantic segmentation problems. Recall from Section 3.1.1 that a low precision is indicative of overrepresentation and a low recall of underrepresentation. Take note that a worm that is substantially overrepresented all around its border will have very little to no underrepresentation, hence a low precision and a high recall. Similarly, a worm that is substantially underrepresented all around its border will have very little to no overrepresentation, hence a low recall and a high precision.

Notice that both model A and the 27×27 input size version of model B were trained on single pixel labels per input patch, using a balanced sampling strategy (see Section 3.1.2.3). This meant that each class had an equal probability of being selected as a training example, which could have driven the models to become slightly more likely to predict the minority class (live or dead) than its frequency in the data. This bias could make the models more likely to predict one of the worm classes for the regions surrounding the worms, manifesting as overrepresentation and resulting in a low pixel segmentation precision and a high recall.

By increasing the size of the patch of pixel labels, the class distribution captured within the patch becomes closer to the true pixel-level class distribution of the data set. This also means that there might exist an ideal patch size beyond which the class distribution becomes more in favor of the majority class (background). The 65×65 input size version of model B seems to be an example of this, where the patch of pixel labels simply includes too many background pixels, causing the model to become biased by predicting the background class too often. This bias could make the model more likely to predict the background class on and around the borders of the worms, manifesting as underrepresentation and resulting in a low recall and a high precision.

Interestingly enough, the architecture of the network also seems to play a role in how the class imbalance problem affects the behavior of the network. This can be observed in the performance of both models C and D in Table 4.1, both of which have a similarly sized output to that of the 65×65 version of model B. The difference is that the 65×65 version of model B showed a low pixel segmentation recall and high precision, while models C and D showed a balanced distribution between their pixel segmentation precision and

recall. This result could suggest that the added complexity in the architectures of models C and D allow the models to better manage the class imbalance problem.

Uneven Illumination

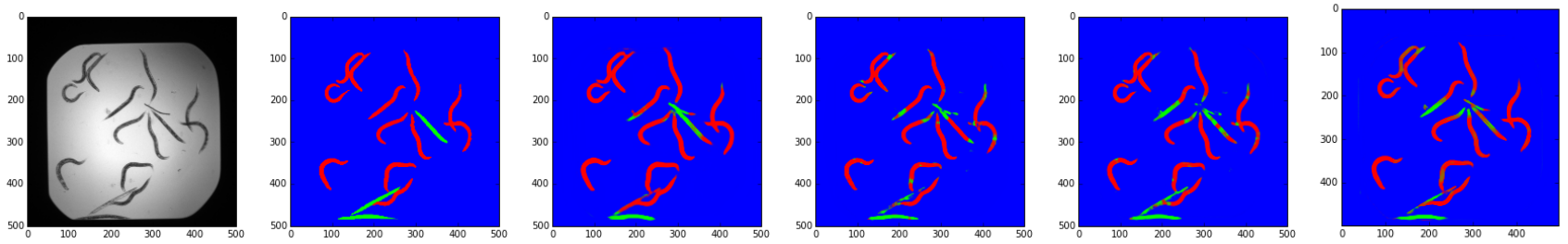
One observation that is clear from the results in Table 4.1, is that there does not seem to be a notable effect when changing the FCN model architecture for the worm-level metrics. The worm-level performance of the models was expected to improve as the models increased in depth and in contextual window size. This was not the case, as all of the FCN models achieved similar performance on the worm-level metrics. One hypothesis for this behavior is that some of the worms in the data set are simply more difficult to classify than others, which could suggest two scenarios: Either a more complex approach is needed to classify the more difficult worms, or the worms are difficult to classify due to the quality of the images. Since changing the complexity of the FCN architectures did not seem to make much difference in the worm-level metrics, the second scenario seems more likely.

The models were then used to generate segmentation maps for the image examples in the test set of the first fold (19 images). These segmentation maps were then inspected for any indication as to why there is little difference in performance for the various models. It was found that the illumination level of the images had an influence on the segmentation and classification performance of the models.

An example of a brightly illuminated input image and the corresponding segmentation maps produced by each model is given in Figure 4.8. Similarly, an example of a poorly illuminated input image and the corresponding segmentation maps produced by each model is given in Figure 4.9. Notice how each model struggles in a similar way to detect the worms along the borders in Figure 4.9. All of the models also seem to struggle to predict the class of the worms in Figure 4.9 due to their much darker appearance.

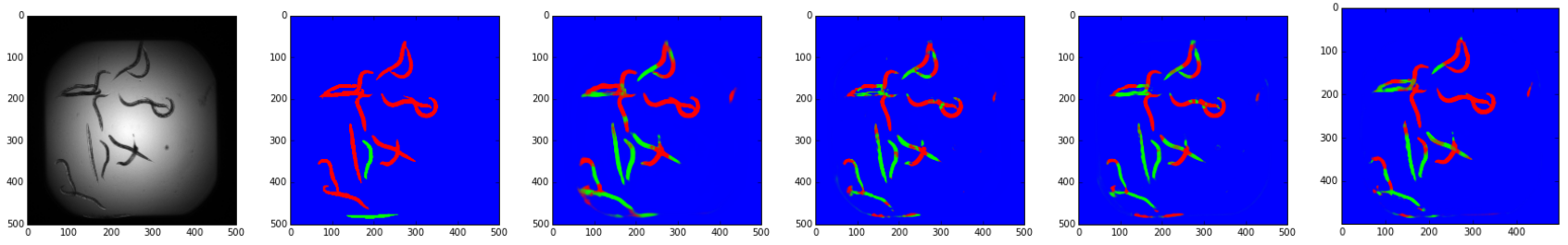
Based on this observation, we first hypothesized that the lower illumination levels change the texture of the worms, thereby making it more difficult to distinguish between the live and dead classes. However, despite close inspection of the input images, we were not convinced that such textural information exists, given the low resolution of the images. The comments of Wählby *et al.* (2012) and the behavior of our networks, on the other hand, seem to support the notion that the images could contain texture information. As such, any further discussion in this work related to texture information in the images are under the assumption that such information does exist.

Under this assumption, the poor illumination seems to make it difficult for the models to detect the texture of the worms (due to their darker appearance) in order to make a live or dead prediction. There were also instances of poorly illuminated images where the boundaries between the worms and the



(a) Input image (b) Ground truth (c) Model A (d) Model B (e) Model C (f) Model D

Figure 4.8: A segmentation example showing the output probabilities generated by each model for the given input image (Figure 4.8a) and its corresponding ground truth label (Figure 4.8b). Figures 4.8c, 4.8d, 4.8e and 4.8f show the segmented output produced by models A, B, C and D respectively. Red, green and blue correspond to live, dead and background.



(a) Input image (b) Ground truth (c) Model A (d) Model B (e) Model C (f) Model D

Figure 4.9: An example input (Figure 4.9a) where low illumination levels hampered classification by the models. The ground truth is given in Figure 4.9b. Figures 4.9c, 4.9d, 4.9e and 4.9f show the segmented output produced by models A, B, C and D respectively. Red, green and blue correspond to live, dead and background.

background were obscured. This suggests that the performance of the models may improve if the illumination levels of the images are corrected, or the models are made invariant to differences in illumination. Further investigation into the illumination levels of the images is listed as possible future work (see Section 8.2).

Contextual Window

Another observation that can be made is how model B seems to perform better than model A with regards to the pixel-level segmentation metrics (except recall), but worse on the pixel-level classification metrics. This suggests that model B is better at detecting worm pixels, but worse at classifying these pixels as alive or dead. One hypothesis is that model B performs worse on the pixel classification metrics because of its relatively small contextual window size compared to the other models. An adult worm would roughly fit into a 100×100 patch (possibly larger), which is much larger than the 27×27 contextual window of model B. This meant that model B cannot use the overall shape information about the worm in order to make a prediction, since only a small part of the worm is visible at a time.

The final observation that can be made is with regards to the pixel-level classification metrics of the three FCNs in Table 4.1. Notice how the mean performance for the pixel classification metrics increase as the number of pooling layers in the model architecture increases. Consider also that model A had the same number of pooling layers as model D, albeit in different places in the architecture, and performed only slightly worse than model D.

It would seem that the differences in pixel classification performance between the four models can be partially attributed to the differing contextual windows of the networks. For instance, arranging models A to D in ascending order of contextual window size yields: model B (27×27), model C (30×30), model A (63×63) and lastly, model D (72×72). This arrangement also corresponds to their mean pixel classification metrics.

One hypothesis for this behavior is that the larger the network's contextual window is, the more information the model gets with which to formulate a prediction for the corresponding pixels. For instance, the small contextual windows of model B and model C might have only allowed these networks to observe the texture of the worms, with perhaps only a minimal indication of the shape of the worms. The larger contextual windows of models A and D on the other hand, may have allowed the models to observe both the texture as well as important shape information of the worms.

4.4 Conclusion

Various neural network architectures, one standard convolutional neural network and three FCNs of varying depth, were tested on the *C. elegans* live/dead assay data set. It was noted that the results reported in Wählby *et al.* (2012) might have been skewed because of a possible imbalanced test set. All of the tested models showed considerable improvement on two of the three key metrics over the ad-hoc image processing pipeline in Wählby *et al.* (2012). Although the networks did not perform as well with regards to the worm classification accuracy, it did manage to outperform the pipeline in both the worm segmentation correctness measure and the worm classification precision.

The binary segmentation masks might have allowed model A to base its decision for one worm on the potential class of a neighboring worm in the same input patch. The class imbalance problem was found to manifest slightly differently in semantic segmentation tasks, where the model can either become biased towards the minority class, causing overrepresentation, or the model can become biased towards the majority class, causing underrepresentation. The severity of the class imbalance problem seems to depend on both the architecture of the network and the class distribution in the label patches.

Although the architecture of the models changed considerably, all of the models showed comparable performance in most of the metrics. This observation led to the hypothesis that the poor illumination levels in some of the images could obscure key morphological information, such as the texture of the worms or the border between worm and background. Correcting the poor illumination levels or making the models invariant to the illumination level in the images might be the key to further improve model performance. It was also shown that all of the FCN approaches performed better than the standard convolutional neural network with regards to segmentation. Lastly, the contextual window of the network was found to play a key role in enabling the network to achieve better pixel-level classification performance.

While the work described in this chapter was being conducted, another group published an architecture similar to, but deeper than, model D. They applied their model, named U-net, to the nerve cell membrane data set (Section 3.2) Ronneberger *et al.* (2015). Given the similarities to model D and the increased complexity introduced by additional pooling, convolutional and deconvolutional layers, the U-net model along with one of the data sets it was applied to, was chosen to form the basis for the rest of this work. The next chapter will use the U-net architecture to determine how the depth of the network and the presence or absence of skip connections affect the overall performance of the model.

Chapter 5

U-net

The previous chapter introduced a standard CNN and three FCNs of increasing depth. The four models were applied to the *C. elegans* live/dead assay data set and their performance was compared. All of the models outperformed the ad-hoc image processing pipeline of Wählby *et al.* (2012) on the appropriate metrics. The three FCN architectures were also found to perform better with respect to the segmentation metrics compared to the CNN.

This chapter is a continuation of the previous chapter, centered on the recently developed U-net architecture (Ronneberger *et al.*, 2015). The U-net architecture follows a similar structure to that of model D used in Chapter 4. Both architectures consist of a downsampling pathway containing multiple pooling layers, followed by an upsampling pathway containing the same number of upsampling layers. The two pathways in the architecture are also connected by means of skip connections between various layers in the pathways.

The major architectural difference between model D and U-net is that U-net is much deeper, having almost double the number of layers of model D including an additional pooling and upsampling layer. The goal of this chapter is to investigate the role of the two main design aspects that characterize the architecture of FCNs — the use of pooling layers and the presence or absence of skip connections at various locations in the architecture. The U-net architecture was chosen as the focus of this study, as it contains considerably more layers than model D.

Another factor that contributed to the choice of using the U-net architecture, is that it is a proven architecture capable of reaching state-of-the-art performance on three different bio-image segmentation data sets (focusing on cell segmentation). As such, an important consideration was to utilize one of the data sets the architecture was originally developed for. Out of the three data sets, the nerve cell membrane data set (Section 3.2) was chosen as it was the most convenient to obtain.

This chapter is structured as follows: Section 5.1 introduces the U-net architecture as it was used in Ronneberger *et al.* (2015), and the augmentations made that were thought to improve the model performance even further. Sec-

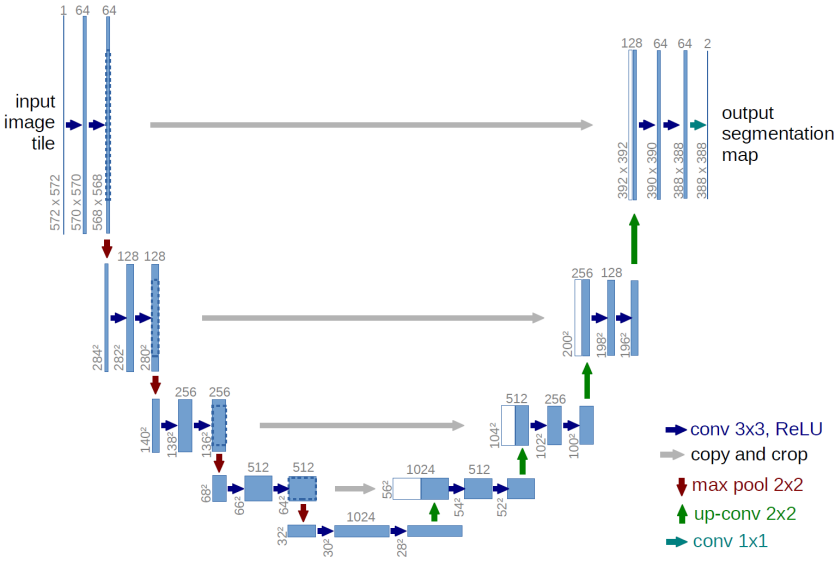


Figure 5.1: The original U-net architecture from Ronneberger *et al.* (2015) ($\approx$ 31 million parameters).

tion 5.2 investigates the two design aspects by building and training smaller networks, and measuring how performance changes. Lastly, Section 5.3 follows an alternative approach to that of Section 5.2, whereby the connection weights of different layers in the architecture are zeroed after training.

5.1 U-net architecture

This section describes the original U-net architecture from Ronneberger *et al.* (2015), depicted in Figure 5.1, which consists of a downsampling pathway followed by an upsampling pathway. Each corresponding level in the two pathways is connected by a skip connection, with each skip connection acting as a channel over which higher resolution information (which might otherwise be lost during downsampling) can be transferred.

All of the regular convolutional layers perform valid convolutions using 3×3 filters and have ReLU activation functions. The downsampling path contains multiple max-pooling layers, each performing a 2×2 pooling operation with a stride of 2. The upsampling pathway contains the same number of upsampling layers as in the downsampling path. Each upsampling layer performs a fixed upsampling of the feature maps by copying the values at each location in the feature maps to a corresponding 2×2 region. For instance, consider a single

2×2 feature map and its upsampled version:

$$\begin{bmatrix} 2 & 5 \\ 8 & 4 \end{bmatrix} \rightarrow \begin{bmatrix} 2 & 2 & 5 & 5 \\ 2 & 2 & 5 & 5 \\ 8 & 8 & 4 & 4 \\ 8 & 8 & 4 & 4 \end{bmatrix}. \tag{5.1}$$

The upsampled version of the feature maps are then convolved with a 2×2 filter in order to halve the number of feature maps. The output of the network is then produced by a ConvSoftmax layer utilizing a 1×1 filter. Finally, 50% dropout was applied on the incoming and outgoing layers of the deepest level.

The resulting model also utilized a custom pixel weighting function designed specifically to be used for the three data sets U-net was applied to. This custom pixel weighting function used the distances to the borders of cells as a way to emphasize the regions that separate neighboring cells. The pixels that are close to these regions are weighted as being more important in the cross-entropy loss function than the pixels inside of the cells.

The official version of U-net reported state-of-the-art performance with a Rand score thin metric (Section 3.2.1) value of 97.27%. It is worth noting that the U-net architecture was re-implemented in our system, based on the architectural information that was made available. Unfortunately, the details required to implement the weighting function were not specified, and as a result it was omitted from our implementation.

Our implementation was trained using ADADELTA (instead of momentum used in the original work) to reduce the time required for training and followed the sampling strategy set out in Section 3.2.2. Training was performed using the standard cross-entropy loss function for a training duration of 200 epochs. The resulting model yielded a Rand score thin metric value of 95.94% as the best out of three submissions to the test server.

5.1.1 Augmentations

A number of changes were made to the original architecture to simplify and generalize it. Firstly, the dropout operations on the incoming and outgoing layers of the deepest level were removed as they were deemed unnecessary: The use of dropout was motivated by Ronneberger *et al.* (2015) as a form of data augmentation, which was used in conjunction with a finite amount of data enrichment involving similar techniques as described in Section 3.2.2. Initial experimentation using our implementation showed that dropout had no effect on the performance of the network, which can likely be attributed to our extensive data enrichment through the use of on-demand randomized transformations of training data.

Secondly, the upsampling layers were replaced with deconvolutional layers. The deconvolutional layers had a filter size of 5×5 with a stride of 2. This allowed the upsampling and halving of the number of feature maps to be done

in a single, trainable operation. The final augmentation to the model involved replacing the ReLU activation function of the convolutional layers with its more general form, PReLU (He *et al.*, 2015).

Based on how close the top results on the challenge leader board are to a perfect segmentation and observing the output produced by the original U-net architecture, it was speculated that the remaining errors made by the models could be because of membranes that are difficult to detect. As a result, the augmented model was trained for 200 epochs using the boosted cross-entropy loss function (Section 2.2.1) with $\alpha = 1$, placing more emphasis on the difficult examples. With these augmentations, the model achieved a Rand score thin metric value of 97.08% as the best out of three submissions to the test server.

5.2 Architecture Analysis

Before continuing this section, it is necessary to explain our usage of a few terms. The *level* in a network is used to refer to any one of the different feature map resolutions that occur within the network, meaning that each additional pooling layer creates a new level in the network. In addition, the *depth* of the network refers to the total number of levels in the architecture, not the number of layers. For instance, the U-net architecture is considered to be five levels in depth, with each level of the architecture being separated by a pooling layer and a deconvolutional layer. This is important to note, as the networks used in this chapter are referred to by the number of levels in their architecture.

The results of Chapter 4 indicated that increasing the depth of the network (by adding more levels) had a positive effect on performance. The U-net architecture is relatively complex, consisting of 27 layers and there are few indications as to why certain design choices were made to achieve good performance. As such, a set of experiments were devised to better understand the effects of various facets of the U-net architecture, focusing on the number of levels in the architecture as well as the role of the various skip connections.

5.2.1 Network Depth

The first important design aspect of the U-net architecture that needs exploration is the network depth. For the first set of experiments, five network architectures of increasing depth were created. In order to avoid simply adding an arbitrary number of layers to make the networks deeper, a recursive approach was devised to simplify the process. The recursive approach involves inserting a predefined sub-network (Figure 5.2) into the middle of the central four convolutional layers of an existing network.

All of the architecture figures in the rest of this chapter will follow the same convention as in Figure 5.2. Each arrow depicts a pooling, convolutional

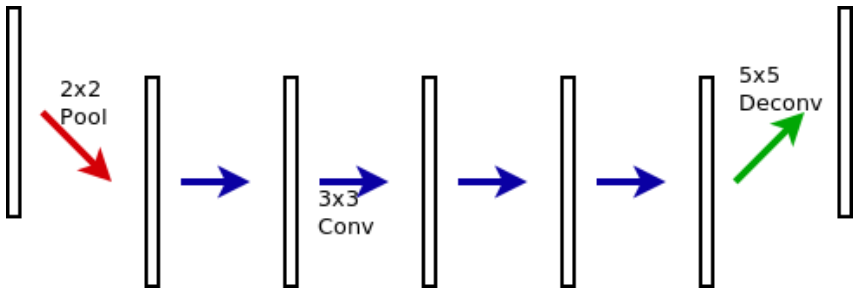


Figure 5.2: A representation of the sub-network used to deepen an existing network by one level. Each arrow depicts a pooling, convolutional or deconvolutional operation being performed, with arrows of similar color representing the same operation.

or deconvolutional operation being performed, with arrows of the same color representing the same operation. The blocks on either side of the arrows represent the feature maps of the previous and current layers that act as the input and output of the operation represented by the arrow. It is important to note that this sub-network does not include skip connections.

The first step is to start with a basic Level 1 architecture similar to that of model B (Figure 4.3), which corresponds to the first level in the U-net architecture. The Level 1 network (Figure 5.4a) consists of four convolutional layers, each with 3×3 filters, and a ConvSoftmax layer with a 1×1 filter as output. This architecture is then progressively increased in depth by disconnecting the two halves of the network in the middle of the central four convolutional layers. The sub-network (Figure 5.2) can then be inserted between the two halves of the network by redirecting information to pass from the first half to the sub-network and from the sub-network to the second half. An illustration of the deepening process is given in Figure 5.3.

Starting from a basic Level 1 model, the recursive deepening approach was continuously applied up to a Level 5 model, which corresponds to the depth of the U-net architecture. This resulted in five different architectures, referred to as Level 1 through 5, corresponding to the number of levels in the respective architecture. A full depiction of each architecture is given in Figure 5.4. Note that adding skip connections as the network is made deeper is *not* part of the process.

These experiments were not submitted for official evaluation, and as such utilized the 15 training - 15 testing split approach set out in Section 3.2.2. The networks were trained for 200 epochs using ADADELTA and the boosted cross entropy loss function. The Rand score thin metric results for each network is summarized in Table 5.1 in the form *mean* $\pm$ *standard deviation*, where applicable. The number in parentheses next to each result indicates the number of times the experiment was repeated with a different initial seed.

Initially, the results in Table 5.1 seem to support the observations made in Chapter 4, where performance improves as the network increases in depth

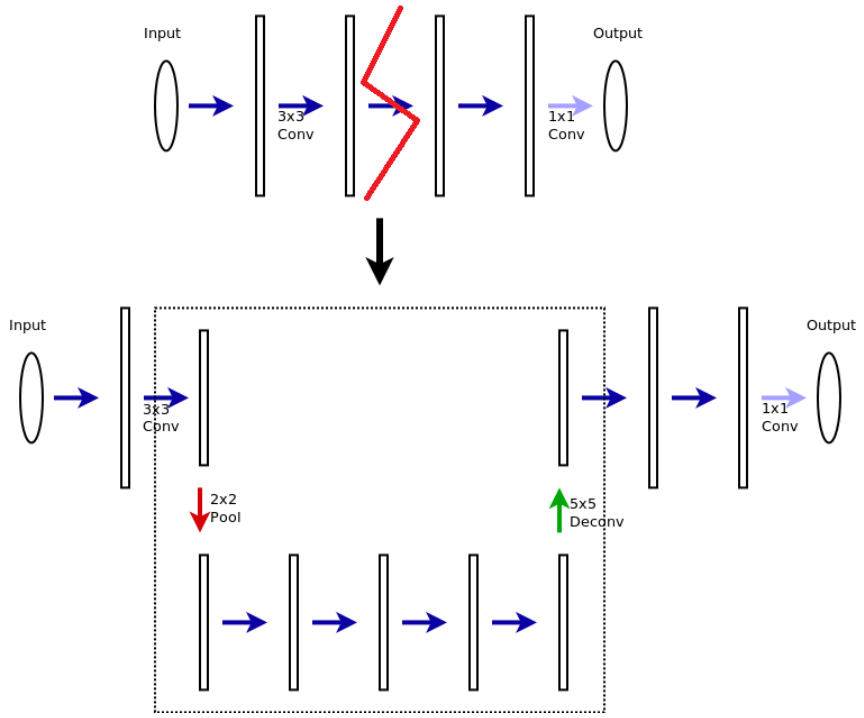
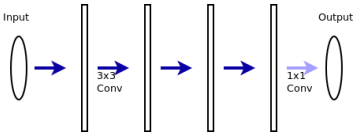


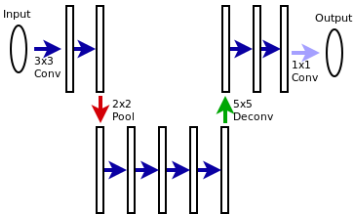
Figure 5.3: A diagram illustrating the recursive deepening approach. First, the top network (in this case a Level 1 network) has to be disconnected in the middle of the central four convolutional layers, essentially splitting the network in half. This is done by removing the incoming connections of the third convolutional layer. The sub-network can then inserted between the two halves of the network. The output of the last layer in the first half of the top network is then connected as input to the pooling layer in the sub-network. Similarly, the output of the deconvolutional layer in the sub-network is then connected as input to the first layer in the second half of the top network. The resulting network is then one level deeper than before — in this case creating a Level 2 network.

Table 5.1: The Rand score thin results obtained for the five architectures. The number in parentheses indicates the number of repetitions that were performed.

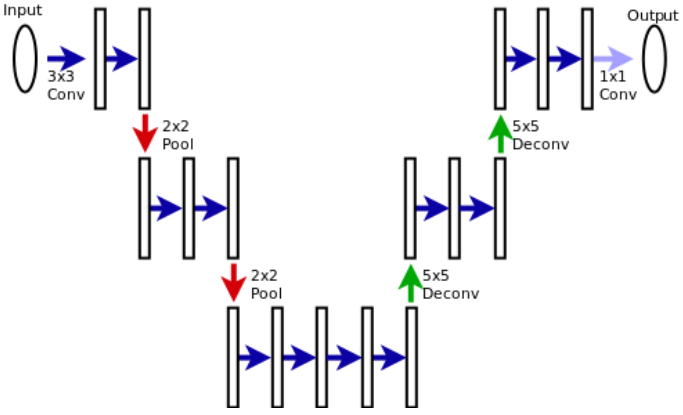
Model:	Level 1	Level 2	Level 3	Level 4	Level 5
Rand score thin:	83.34 ± 0.59 (3)	94.80 ± 0.53 (3)	94.95 ± 0.95 (3)	7.63 (1)	7.63 (1)



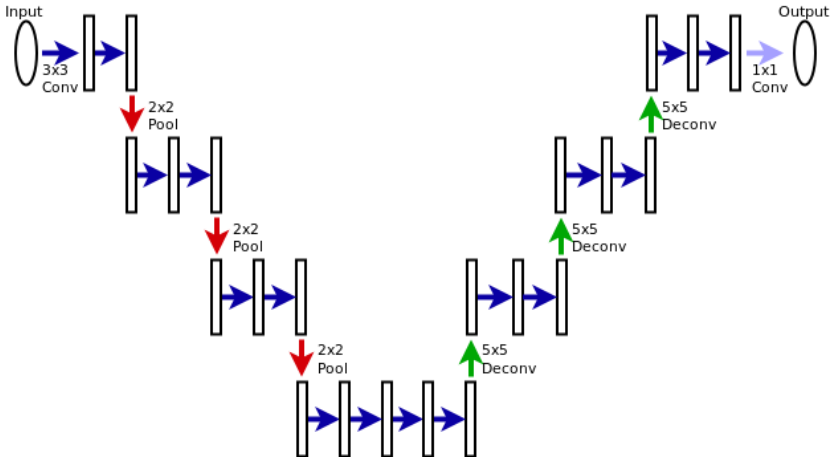
(a) Level 1 model



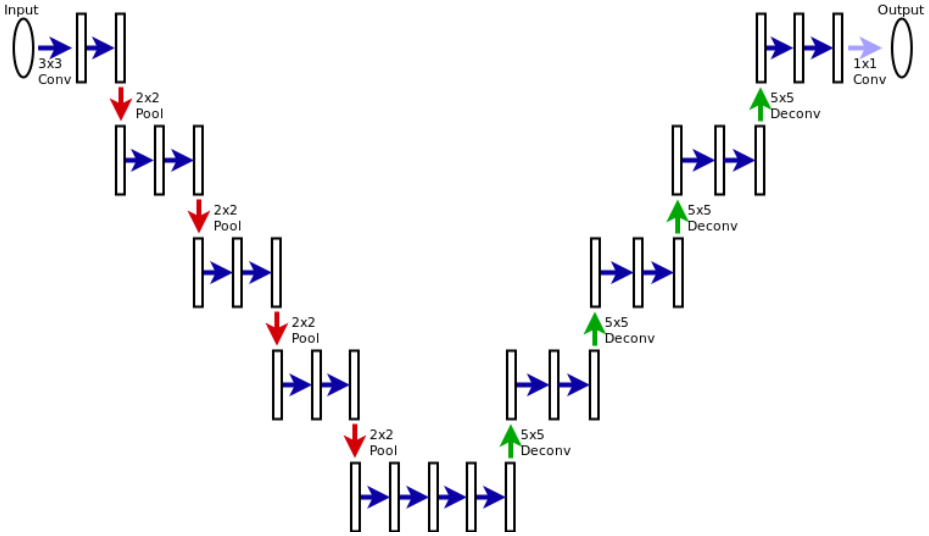
(b) Level 2 model



(c) Level 3 model



(d) Level 4 model



(e) Level 5 model

Figure 5.4: All of the resulting models that are used in Section 5.2.1. Arrows of the same color represent the same operation being performed.

(83.34% → 94.80% → 94.95%). Unfortunately, this pattern does not continue to hold when the model is deeper than three levels: Both the Level 4 and Level 5 models obtained a Rand score thin metric value of 7.63%, which was empirically found to be the lowest possible value. Visualizing the output produced by these models showed that they were not able to learn anything meaningful, as the models simply produced static noise. This result suggests that there might be some domain-specific limitation as to how deep a network can become (without skip connections) before being unable to perform the segmentation task.

In Chapter 4, the use of skip connections in model D was motivated as a means to prevent excessive loss of information through the pooling layers. The skip connections allowed higher resolution information to transfer between the two pathways in the network, preserving the information required for a meaningful segmentation. Whether or not this loss of information did occur, remained unclear, as the experiments in Chapter 4 were not aimed at determining whether or not this was the case.

These experiments, on the other hand, do suggest that too much information is lost through pooling. It is clear that for the first three levels, the upsampling pathway in the network seems to be able to reproduce most of the high resolution information that is required for segmentation. Once the network becomes deeper than three levels, however, this is no longer the case. This result suggests that there is not enough information retained in the Level 4 and 5 models for the upsampling pathway to produce a high resolution segmentation. Considering that the membranes in the data set are relatively thin compared to the size of the cells, but still crucial for good segmentation, it is possible that the position of the membranes are lost after three or more pooling layers. It is clear from these results that skip connections are required, especially for deep networks.

5.2.2 Skip Connections

The preceding section showed that making an existing architecture deeper by adding more levels can lead to an improvement in performance; however, this observation does not hold for all network depths. These results suggest that there might be some domain-specific limitation on the network depth, beyond which the network appears to retain too little information to produce sufficiently high resolution segmentations. It is hypothesized that by adding skip connections to these five networks, the degradation of network performance due to losing too much information, especially in the deeper networks, might be mitigated.

Experimentation with the skip connections will be divided into two parts: The first part will observe how the different networks change in performance when *all* of the skip connections are enabled. The second part will observe how the performance of the networks change when only *some* of the skip con-

Table 5.2: The Rand score thin metric results obtained for the five architectures where all of the skip connections were enabled. The number in parentheses indicates the number of repetitions that were performed. The results in italics were reused from previous tables.

Model:	Level 1	Level 2	Level 3	Level 4	Level 5
Rand score thin:	*83.34 ± 0.59 (3)*	94.32 ± 0.82 (3)	96.76 ± 0.58 (4)	97.41 ± 0.19 (4)	97.51 ± 0.35 (4)

nections are enabled. For illustrative purposes, the full network architectures with labeled skip connections are given in Figure 5.5.

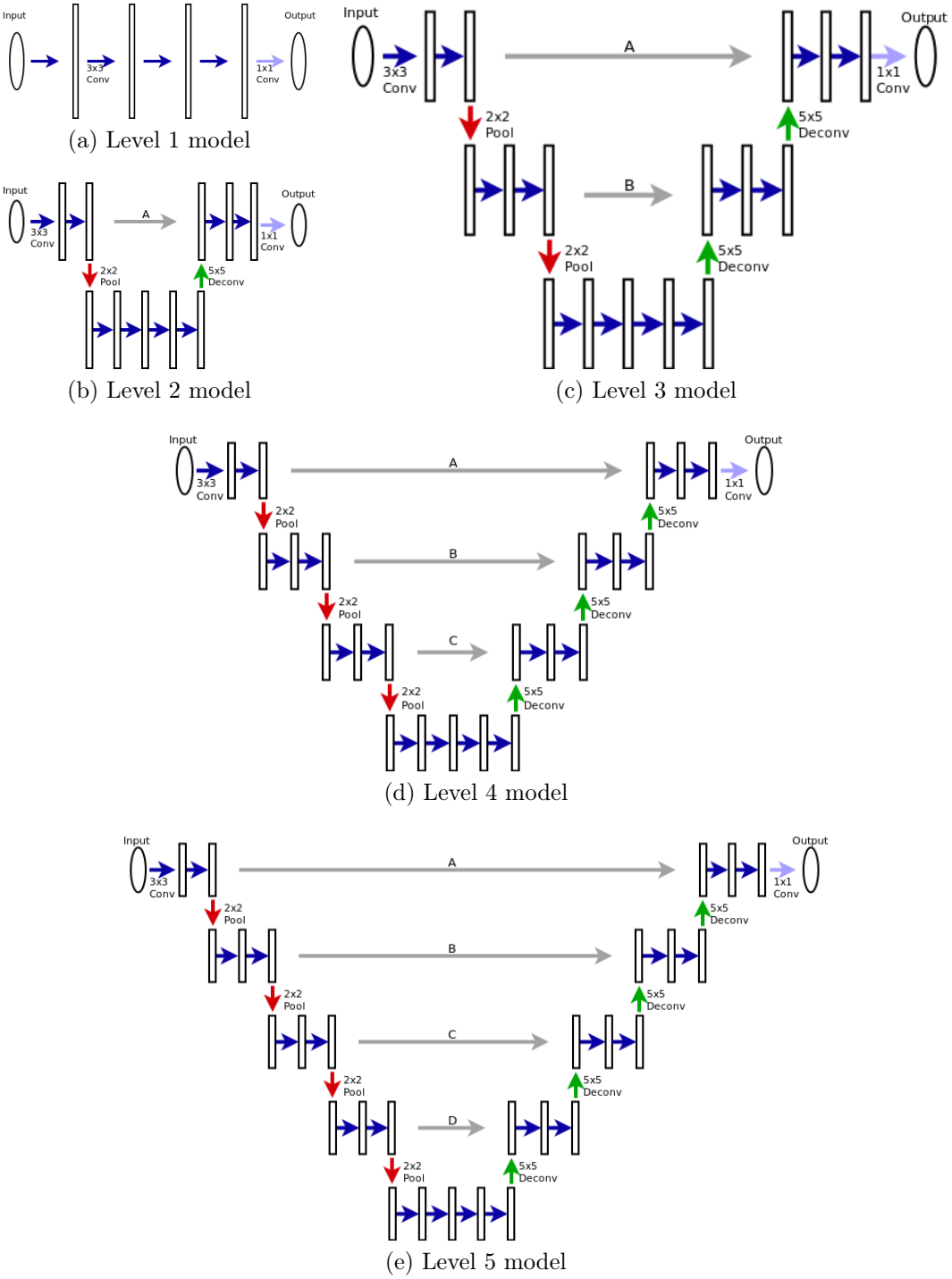
Part 1: Full Skip Connection Configuration

The networks can be considered as having two extreme configurations in terms of the skip connections, where either *all* of the skip connections are enabled or *none* of them are. The experiments performed in the Section 5.2.1, which investigated how the depth of the network affected its performance, also provided some insight into one of the extremes where none of the skip connections were enabled. We now consider the other extreme of having all of the skip connections enabled.

The full configuration of the five networks where all of the skip connections are shown and labeled is presented in Figure 5.5. All of the networks were trained using the same training procedure as before. The Rand score thin metric result for each network is summarized in Table 5.2 in the form *mean* ± *standard deviation*, where the number in parentheses shows the number of repetitions. The result for the Level 1 model (reused from Table 5.1) is only shown for completeness, but because it does not contain any skip connections, it will not be discussed further.

All of the models, apart from the Level 2 model, showed a notable improvement in performance when all of the skip connections were enabled. This result, together with the results in Section 5.2.1, strongly suggest that there is indeed information required for segmentation lost through the pooling layers. The presence of the skip connections in the architecture of the models seem to preserve this required information to be used later in the network, thus preventing the degradation of the network performance and improving the network results. One counterexample could be the Level 2 model, as performance decreased with the addition of its skip connection. However, this difference in the mean performance is not large enough to be significant.

Another observation that can be made from the results in Table 5.2, is that it appears that performance starts to saturate as the depth of the network increases. This suggests that a Level 5 network (such as U-net), or potentially a well-trained Level 4 network, is sufficient for the task of this data set, and that making the network even deeper may yield little to no benefit with respect to performance. Similarly, other data sets may have their own optimal network



(a) Level 1 model

(b) Level 2 model

(c) Level 3 model

(d) Level 4 model

(e) Level 5 model

Figure 5.5: An illustration showing all of the possible skip connections for each of the five models. Each skip connection is also labeled, which will be used to refer to the specific skip connection in the experiments in Section 5.2.2 and Section 5.3. Arrows of the same color represent the same operation being performed. The arrows for the skip connections are representative of the crop-and-stitch technique.

depths after which performance would start to saturate. This observation could also relate to the contextual window results of Chapter 4, in that the contextual window of a Level 5 network is sufficiently large to view some of the largest cells in the data set.

Part 2: Partial Skip Connection Configuration

In light of the results so far, it is clear that skip connections are beneficial for performance. This still leaves the question of whether all of the skip connections are required, or if similar performance can be achieved with a specific combination of skip connections. As such, the different combinations of active skip connections were tested in the Level 3 to 5 models. The results presented in this section will include the results of Table 5.1 and Table 5.2 that were discussed in the previous sections for completeness.

The complete results for all of the models with various combinations of enabled skip connections are summarized in Table 5.3. The labels `A`, `B`, `C` and `D` refers to the skip connections of the corresponding model in Figure 5.5. Each row in Table 5.3 should be read as the model architecture (without skip connections) plus the corresponding enabled skip connections (indicated by the black squares). A white square indicates that the corresponding skip connection was omitted from the model architecture. Lastly, the results are given in the form *mean* $\pm$ *standard deviation*, where the number in parentheses indicates the number of repetitions.

The Level 1 and Level 2 models were not part of the experiments performed in this section, and thus will not be discussed further. The results for these two models are included in Table 5.3 for comparison only. The Level 3 model seems to perform as well when either skip connection is active as it does when both are active, with a Rand score thin metric value of approximately 96%. This result suggests that only one of the two skip connections is truly needed, as opposed to having both active.

A similar observation can be made for the Level 4 model: In all instances of the Level 4 model where all but one of the skip connections were enabled, the network managed to achieve roughly the same performance of approximately 97% to 98%. The performance of the Level 4 network starts to vary when only one skip connection is enabled. Looking at the results for the three cases where only one skip connection was enabled, it would appear that the deeper the enabled skip connection is in the architecture, the better the model is able to perform. This result suggests that the performance of the Level 4 model is highly dependent on the availability of skip connection C, without which performance drops unless both A and B are otherwise enabled.

The Level 5 model is more complicated to investigate, since there are a large number of possible combinations of enabled skip connections. To start with, when all of the skip connections are enabled except for one, the deeper the disabled skip connection is in the network, the greater the drop in per-

Table 5.3: The Rand score thin metric results for all of the models with different combinations of skip connections enabled. The labels `A`, `B`, `C` and `D` refers to the skip connections of the corresponding model in Figure 5.5. A dash indicates that the corresponding model does not contain that specific skip connection. Each row should be read as the model architecture (without skip connections) plus the enabled skip connections (black squares). A white square indicates that the corresponding skip connection was omitted from the model architecture. Results in italics were reused from previous tables.

Model	A	B	C	D	Rand score thin
Level 1	-	-	-	-	*83.34 ± 0.59 (3)*
Level 2 +	□	-	-	-	*94.80 ± 0.53 (3)*
Level 2 +	■	-	-	-	*94.32 ± 0.82 (3)*
Level 3 +	□	□	-	-	*94.95 ± 0.95 (3)*
Level 3 +	■	□	-	-	96.45 ± 0.19 (3)
Level 3 +	□	■	-	-	96.10 ± 0.60 (3)
Level 3 +	■	■	-	-	*96.76 ± 0.58 (4)*
Level 4 +	□	□	□	-	*7.63 (1)*
Level 4 +	■	□	□	-	88.53 ± 7.72 (3)
Level 4 +	□	■	□	-	96.38 ± 0.59 (3)
Level 4 +	□	□	■	-	97.56 ± 0.23 (3)
Level 4 +	■	■	□	-	97.04 ± 0.54 (3)
Level 4 +	■	□	■	-	97.94 ± 0.07 (3)
Level 4 +	□	■	■	-	97.45 ± 0.51 (3)
Level 4 +	■	■	■	-	*97.41 ± 0.19 (4)*
Level 5 +	□	□	□	□	*7.63 (1)*
Level 5 +	■	□	□	□	84.63 ± 0.55 (3)
Level 5 +	□	■	□	□	94.35 ± 0.71 (3)
Level 5 +	□	□	■	□	92.80 ± 0.95 (3)
Level 5 +	□	□	□	■	7.63 (1)
Level 5 +	■	■	□	□	93.30 ± 0.33 (3)
Level 5 +	■	□	■	□	95.99 ± 0.46 (3)
Level 5 +	■	□	□	■	88.94 ± 7.13 (3)
Level 5 +	□	■	■	□	95.26 ± 1.76 (3)
Level 5 +	□	■	□	■	95.88 ± 0.56 (3)
Level 5 +	□	□	■	■	97.50 ± 0.32 (3)
Level 5 +	■	■	■	□	95.12 ± 0.37 (3)
Level 5 +	■	■	□	■	96.36 ± 1.11 (3)
Level 5 +	■	□	■	■	96.56 ± 0.91 (3)
Level 5 +	□	■	■	■	97.54 ± 0.18 (3)
Level 5 +	■	■	■	■	*97.51 ± 0.35 (4)*

formance is. It is interesting to note that for both the Level 3 and Level 4 models, disabling a single skip connection did not significantly influence the performance, which does not seem to be the case for the Level 5 model. One possibility is that the replications of the Level 5 model were simply more sensitive to random initialization, as indicated by the relatively large standard deviations observed for some of these cases.

Another interesting observation is that any combination of enabled skip connections that do not include either B or C, results in a significant drop in performance. This suggests that the most important information required for a good segmentation might be captured by either skip connection B or C, whichever one is enabled. Notice also that no combination of enabled skip connections that do not include skip connection D is able to achieve a Rand score thin metric value of more than 96%. This suggests that the information captured by skip connection D cannot be captured by any other available connection, making skip connection D critical for good performance.

Lastly, similar behavior to that of the Level 4 model is observed when skip connections A, B and C are all disabled. Without any of these skip connections, no matter the status of skip connection D, the Level 5 model does not appear to retain enough information to produce a sufficiently high resolution segmentation. This result supports the hypothesis presented in Section 5.2.1 of a domain-specific limitation on the depth of a network without skip connections, which in the case of the nerve cell membrane data set appears to be three levels.

5.3 Pathway Disabling

An alternative approach to investigate the role that each design choice plays towards the overall performance, is by taking a trained model, zeroing the weights of the specific layer that corresponds to the design choice that is under investigation and observing how the performance and predictions of the model changes. This process can also be considered as applying 100% drop-connect on the connections of a specific layer during testing. A similar approach as in the previous section will be followed, first by investigating the depth of the network and then different combinations of skip connections.

5.3.1 Level Disabling

To investigate how each level contributes to the overall performance, a fully trained U-net model was used instead of a Level 5 model. Ten instances of the full U-net model were trained, using different random initializations. For each trained instance, the weights of the various deconvolutional layers were zeroed, essentially disconnecting the lower levels of the network from contributing to the produced output.

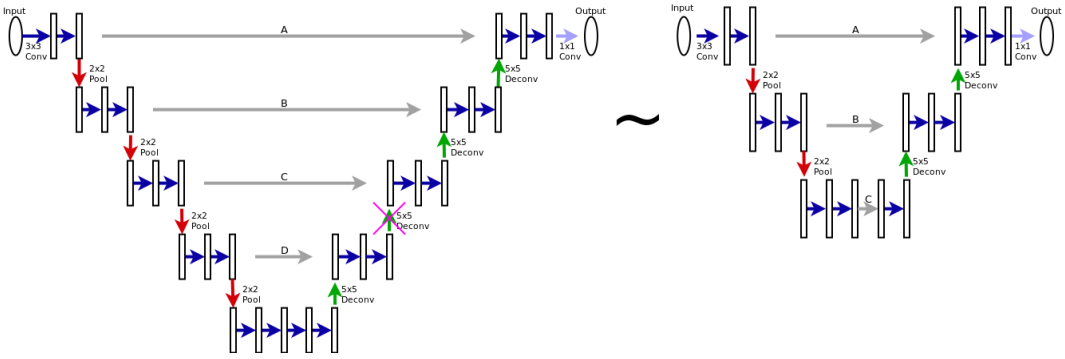


Figure 5.6: An illustration of how the architecture for a Level 5 model becomes equivalent to a Level 3 model when the weights of the second deconvolutional layer are zeroed (indicated by the magenta cross). Skip connection C then essentially becomes a regular connection between the downsampling and upsampling pathways, as in the original Level 3 model.

Table 5.4: The Rand score thin metric results obtained after removing the various levels calculated over the ten repetitions.

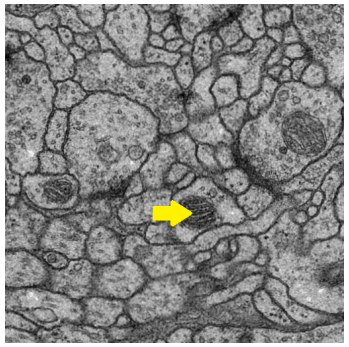
Equivalent model:	Level 1	Level 2	Level 3	Level 4	Level 5
Rand score thin:	19.96 ± 19.33	89.16 ± 1.13	96.66 ± 0.53	97.46 ± 0.44	97.43 ± 0.67

For instance, consider the Level 5 model in Figure 5.6 with all skip connections enabled, where the weights of the second deconvolutional layer in the upsampling pathway are zeroed (indicated by the magenta cross). When those weights are zeroed, the model loses the ability to utilize the bottom two levels, making the resulting model equivalent to a Level 3 model with all skip connections enabled. As indicated in the figure, skip connection C then becomes a regular connection between two layers.

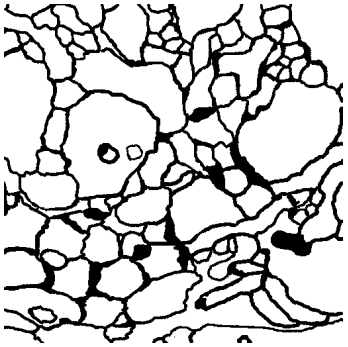
The results of these experiments are given in Table 5.4. It is clear from the results obtained that all of the levels except for level 5 contribute to the overall performance of the network. With each additional level that the network is allowed to use to produce an output, the performance of the network increases quickly until finally saturating at level 4.

From these results, as well as the results in Section 5.2.2, it is unclear why the U-net architecture uses five levels, since a Level 4 model is already capable of achieving similar performance. Since the U-net architecture was applied on three different segmentation data sets, it is possible that the fifth level might have been required on one of the other data sets.

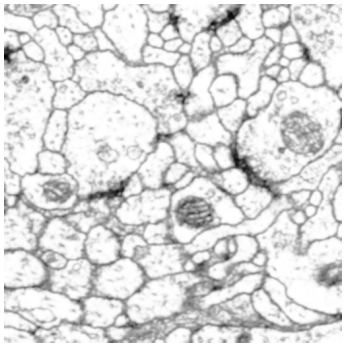
This investigation was continued by visually inspecting the output produced by the network after the connection weights of the various deconvolutional layers had been zeroed. An example input image and its corresponding ground truth is provided in Figure 5.7, along with the output produced by the model after zeroing the weights of the various deconvolutional layers.



(a) Input image

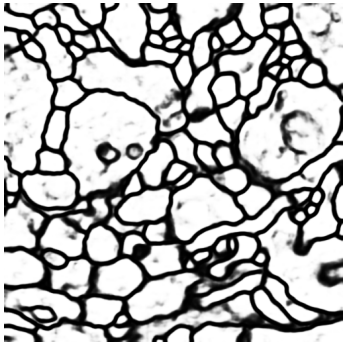


(b) Ground truth

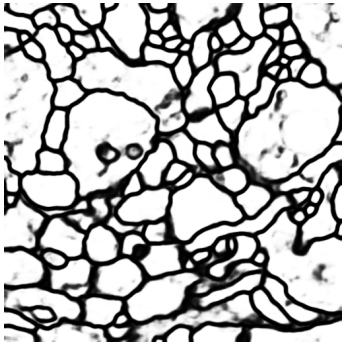


(c) Equiv. Level 1 model

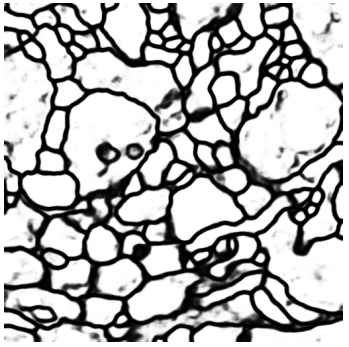
(d) Equiv. Level 2 model



(e) Equiv. Level 3 model



(f) Equiv. Level 4 model



(g) Equiv. Level 5 model

Figure 5.7: The output produced by a single trained U-net model where the connection weights of the various deconvolutional layers of the network were zeroed, creating a model equivalent in architecture to the various Level models in Section 5.2. These output visualizations indicate to some degree what each level in the U-net architecture tries to achieve. One of the difficulties of this task is to not include cellular organelles, such as the cell nucleus (indicated by the arrow in Figure 5.7a), as part of the produced segmentation.

These output figures provide some indication as to what role each level in the U-net architecture plays in the overall functioning of the network. As shown in Figure 5.7c, it would appear that the first level learns some form of prior that resembles the input figure, which could be helpful when combined with the deeper levels to produce a high resolution segmentation. With the addition of the second level (Figure 5.7d), the model already seems certain where most of the membranes occur within the image; however, there is still a lot of noise present within the cells.

Looking at the output in Figure 5.7d and comparing it to the deeper level outputs, it would appear that the deeper levels are more oriented towards filling in the gaps in the segmentation and removing the noise. The third level (Figure 5.7e) seems to remove most of the noise in the cells, and in some cases also to strengthen the membranes that are still uncertain from the second level. Finally, levels 4 and 5 (Figures 5.7f and Figure 5.7g) then seem to remove larger objects from the cell segmentation, such as the cell nuclei (indicated by the arrow in Figure 5.7a).

The quantitative results in Table 5.4 did not show any difference in performance between the fourth and fifth level. The qualitative results in Figure 5.7 seem to concur with the quantitative results, as it appears that both level 4 and level 5 are performing a similar task, perhaps even strengthening each other. These output visualizations seem to suggest that each level in the fully trained U-net architecture performs their own distinct function, which when combined produces a high resolution segmentation with state-of-the-art performance.

5.3.2 Skip Connection Disabling

Next, the contribution of the various skip connections are investigated, by taking the different sets of trained networks (Level 2 to 5) in which all the skip connections are enabled and zeroing various combinations of the skip connections before evaluation. The complete results for all of the models with various combinations of skip connections zeroed are summarized in Table 5.5, which is formatted similarly to Table 5.3 — the only difference is that the white squares in Table 5.5 now indicates that the corresponding skip connection was zeroed during the evaluation of the model, instead of omitted from the architecture during training.

In most instances, it is immediately evident that removing skip connections from a trained model will greatly reduce the model performance. There are some interesting observations that can still be made, such as the importance of certain skip connections that seem to be dependent on the random initialization of the model.

For instance, consider the Level 4 model. In all of the combinations where skip connection A was zeroed, with or without any other skip connection, a notably larger standard deviation can be observed compared to the other combinations. The large standard deviation suggests that the removal of skip

Table 5.5: The Rand score thin metric results for the all of the models with different combinations of skip connections zeroed. The labels `A`, `B`, `C` and `D` refers to the skip connections of the corresponding model in Figure 5.5. A dash indicates that the corresponding model does not contain that specific skip connection. Each row should be read as the model architecture (without skip connections) plus the enabled skip connections (black squares). A white square indicates that the corresponding skip connection was zeroed during the evaluation of the model. Results in italics were reused from previous tables.

Model	A	B	C	D	Rand Score
Level 1	-	-	-	-	*83.34 ± 0.59 (3)*
Level 2 +	□	-	-	-	77.78 ± 3.58 (3)
Level 2 +	■	-	-	-	*94.32 ± 0.82 (3)*
Level 3 +	□	□	-	-	92.14 ± 1.22 (4)
Level 3 +	■	□	-	-	95.48 ± 0.91 (4)
Level 3 +	□	■	-	-	94.78 ± 0.34 (4)
Level 3 +	■	■	-	-	*96.76 ± 0.58 (4)*
Level 4 +	□	□	□	-	57.65 ± 20.63 (4)
Level 4 +	■	□	□	-	92.58 ± 0.81 (4)
Level 4 +	□	■	□	-	79.12 ± 6.78 (4)
Level 4 +	□	□	■	-	38.91 ± 18.16 (4)
Level 4 +	■	■	□	-	96.65 ± 0.54 (4)
Level 4 +	■	□	■	-	94.32 ± 0.63 (4)
Level 4 +	□	■	■	-	72.84 ± 14.54 (4)
Level 4 +	■	■	■	-	*97.41 ± 0.19 (4)*
Level 5 +	□	□	□	□	27.62 ± 12.05 (4)
Level 5 +	■	□	□	□	72.51 ± 8.06 (4)
Level 5 +	□	■	□	□	82.59 ± 19.26 (4)
Level 5 +	□	□	■	□	70.37 ± 8.72 (4)
Level 5 +	□	□	□	■	16.03 ± 15.42 (4)
Level 5 +	■	■	□	□	88.51 ± 7.65 (4)
Level 5 +	■	□	■	□	90.99 ± 3.07 (4)
Level 5 +	■	□	□	■	48.96 ± 33.66 (4)
Level 5 +	□	■	■	□	95.19 ± 1.07 (4)
Level 5 +	□	■	□	■	89.24 ± 4.53 (4)
Level 5 +	□	□	■	■	58.45 ± 21.07 (4)
Level 5 +	■	■	■	□	97.15 ± 0.09 (4)
Level 5 +	■	■	□	■	93.12 ± 2.86 (4)
Level 5 +	■	□	■	■	94.39 ± 1.73 (4)
Level 5 +	□	■	■	■	95.81 ± 0.75 (4)
Level 5 +	■	■	■	■	*97.51 ± 0.35 (4)*

connection A appears to affect the performance of some of the Level 4 models more than others. Since the repetitions of each model were the same in every way apart from the initialization, this result suggests that the importance of skip connection A varies depending on the initialization of the model.

A similar observation can be made for the Level 5 model and skip connections B and C: Any instance of the Level 5 model where neither of skip connections B or C are zeroed results in a notably smaller standard deviation. It is important to realize that all of the models were trained with all of the skip connections enabled, which yielded relatively small standard deviations (less than 0.6% in most models). Any large standard deviation observed when zeroing a skip connection is then likely linked to the initialization of the model. These observations suggest that the initialization of the model may impact which skip connections are (heavily) used when multiple options are available.

In the previous set of experiments, where various combinations of skip connections were omitted from the architecture being trained (Section 5.2.2), it was observed that certain skip connections can be omitted from the architecture without losing performance. The most important observation that can be made from the results of zeroing the different skip connections is likely the following: Even though a model might not need a particular skip connection, it is likely to learn to utilize that skip connection if it is provided. There is not one instance in Table 5.5 where zeroing a skip connection had no effect on the performance of the network.

5.4 Conclusion

This chapter investigated how the two main design aspects that characterize the U-net architecture contributed to its overall performance. The investigation started by first creating five different models of varying depth with no skip connections. It was found that the performance of the network improves as the depth of the network increases. However, the results suggest that without any skip connections, there exists some domain-specific limitation on the depth of the network after which it will not retain enough fine-level information to produce a decent quality segmentation.

Skip connections were then added to the models, after which a consistent increase in performance was observed as the depth of the network increased. It was found that there appears to be a saturation point after which increasing the depth of the network will have little to no effect on performance. It is possible that this saturation point is specific to the domain the network is applied to, depending on the size of the objects that need to be segmented.

The different configurations of enabling the various skip connections were then investigated to determine the necessity of each skip connection. It appears to be most beneficial to have all of the skip connections active for this data set; however, similar to the depth of the network, it could be dependent on the

data set and the task that needs to be performed. It seems possible, in some instances, to train a model without the top-most skip connections (either A or B) without a significant loss of performance.

The investigation then proceeded to explore how a trained model utilized the various layers in its architecture that correspond to the two design choices, by zeroing the connection weights of the layer under consideration. Some interesting observations were made when zeroing the different levels of the U-net architecture, in particular, we saw that the deepest level does not seem to contribute much towards the overall performance of the network. A visualization of the output produced by the model after zeroing the different levels, seems to suggest that each level in the full architecture performs a specific function, which only when combined results in a high resolution segmentation with state-of-the-art performance.

The last set of experiments involved zeroing different combinations of the skip connections in the five Level models. It was found that the level of contribution of a certain skip connection could depend on the initialization of the model. Given that only the initialization of a particular model is changed during the repetitions, it would appear that the initialization of the model together with the training procedure determines to what degree a particular skip connection is utilized. Lastly, it was noted that even though a model might not require a particular skip connection to obtain good performance, the model will nevertheless utilize that skip connection if it is provided.

The next chapter will investigate the possibility of using unsupervised learning in the form of input reconstruction to pre-train the augmented U-net architecture presented in this chapter.

Chapter 6

Unsupervised Learning

In the previous chapter, we investigated how the depth of the network and the various skip connections contribute towards the overall performance of the U-net architecture. We showed that each additional level in the architecture resulted in improved performance, provided the availability of all the skip connections. The best performance was achieved when all of the skip connections were enabled; however, the shallower skip connections were found to be of less importance than the deeper skip connections. Lastly, it was found that each application domain could have its own optimal network depth, and that increasing the depth of the network beyond this point would have no effect on performance.

This chapter aims to expand on the augmented U-net architecture from the previous chapter (Section 5.1.1) by modifying the architecture to accommodate unsupervised learning. Unsupervised learning, as used in convolutional autoencoder layers (see Section 2.1.3.2), involves each layer producing a reconstruction of its input, before being fine tuned by supervised learning. One approach would be to replace each convolutional layer in the U-net architecture with a convolutional autoencoder layer. However, since convolutional autoencoder layers undergo unsupervised learning one layer at a time, an architecture of this size would simply be impractical to train. Instead, this chapter will explore end-to-end unsupervised learning, exploiting the ability of FCNs to produce an output of the same scale as its input.

First, Section 6.1 motivates the use of unsupervised learning to pre-train the U-net model, followed by a detailed description of the methodology to accommodate unsupervised learning in the U-net architecture (and FCNs in general) in Section 6.2. Lastly, Section 6.3 analyzes the performance of the models using various significance tests and provides a discussion on the outcome of these tests, followed by suggestions on further improvements that can be made to the approach.

6.1 Motivation

Unsupervised learning can be used as a pre-training technique and has been shown to have a regularization effect on multiple machine learning approaches (Erhan *et al.*, 2010). The particular form of unsupervised learning that is considered is known as input reconstruction, and is typically used for pre-training in regular and convolutional autoencoder layers. One approach to using unsupervised learning to pre-train the U-net architecture, would be to replace each convolutional layer with the corresponding autoencoder layer. The drawback of this approach lies in the pre-training procedure, in that each autoencoder layer needs to be trained consecutively, which will be particularly time-consuming considering that the U-net architecture contains 18 regular convolutional layers.

It is fairly easy to realize that the time required for the pre-training of stacked autoencoder layers does not scale well as the architecture increases in depth. As such, a more efficient approach is needed to allow FCNs to benefit from unsupervised learning without severely affecting the time required for training. The U-net architecture (and FCNs in general) present a unique opportunity to apply unsupervised pre-training due to their ability to produce a full segmentation map rather than a single pixel label for a given input patch. This can be done by requiring the full architecture to reconstruct the input of the network, rather than having each layer reconstruct its own input. At the time of writing, the closest that unsupervised pre-training in the form of input reconstruction has come to application in FCN architectures, is by using stacked convolutional autoencoders (Masci *et al.*, 2011).

Employing unsupervised pre-training followed by supervised fine-tuning can also be considered as semi-supervised learning. Conventional semi-supervised learning as it is used in neural networks often involves a data set of which only a small portion is labeled (Erhan *et al.*, 2010; Hong *et al.*, 2015; Kingma *et al.*, 2014; Rasmus *et al.*, 2015). There are a number of approaches that make use of semi-supervised learning, all of which have shown an improved performance over their purely supervised counterparts. In Hong *et al.* (2015), a decoupled neural network was proposed that consists of two separate networks, one for classification and one for segmentation, connected by bridging layers. This approach represents a special instance of semi-supervised learning: one where the data set consists of weakly labeled data with a small portion of strongly labeled data.[1] In Kingma *et al.* (2014), semi-supervised learning was used with deep generative models, showing state-of-the-art performance on the MNIST data set. In Rasmus *et al.* (2015), unsupervised Ladder networks (Valpola, 2015) were extended by adding a supervised learning component. Their resulting model reached state-of-the-art performance on both

[1]Weakly labeled data typically comprises labels for complete images or bounding boxes around regions, while strongly labeled data refers to pixel-level segmentation maps.

MNIST and CIFAR-10. The positive results reported for these approaches further motivates the use of unsupervised learning to potentially improve the performance of the U-net architecture and FCNs in general.

6.2 Methodology

The approach set out in this section rests on the ability of FCNs to output segmentation maps corresponding to (portions of) the original input. This section will be divided into three parts: the first contains the augmentations made to the U-net architecture and the accompanying changes to the loss function (Section 6.2.1). Section 6.2.2 then describes the experiments that were performed, followed by an overview of the hypothesis tests that were used to analyze the results in Section 6.2.3.

6.2.1 Augmentations

Consider a regular convolutional autoencoder layer: it accepts an input of arbitrary size and is tasked with reconstructing that input to the best of its ability according to some loss function (typically mean squared error). Depending on the configuration of the hyperparameters for the layer, the reconstructed output can correspond either to the full input or a portion of it. Now, consider replacing the single autoencoder layer with a full FCN architecture that can produce the same reconstructed output for a given input image. This will allow the FCN architecture to undergo end-to-end autoencoding by having the *full model* reproduce the network input.

To accommodate the use of unsupervised learning on the augmented U-net architecture, we further augment the network by adding an additional output layer parallel to the ConvSoftmax layer — see the boxed region in Figure 6.1. Similar to the ConvSoftmax layer, the additional output layer (henceforth referred to as the *reconstruction layer*) performed a 1×1 convolution and used a linear activation function. The addition of the reconstruction layer to the architecture allows unsupervised learning to be performed in an end-to-end fashion, instead of the greedy layer-wise approach employed in autoencoder layers.

Having the ConvSoftmax layer and the reconstruction layer in parallel also allowed supervised and unsupervised training to be performed in a single training session. This was achieved by using an extra control parameter in the cost function to switch smoothly between these two modes of training. As such, the resulting cost function for the two parallel layers is given by

$$L = \beta(t)L_S + (1 - \beta(t))L_R, \tag{6.1}$$

where L_S is the ConvSoftmax loss (standard cross-entropy loss averaged over all pixels), L_R is the reconstruction loss (standard per-pixel mean squared

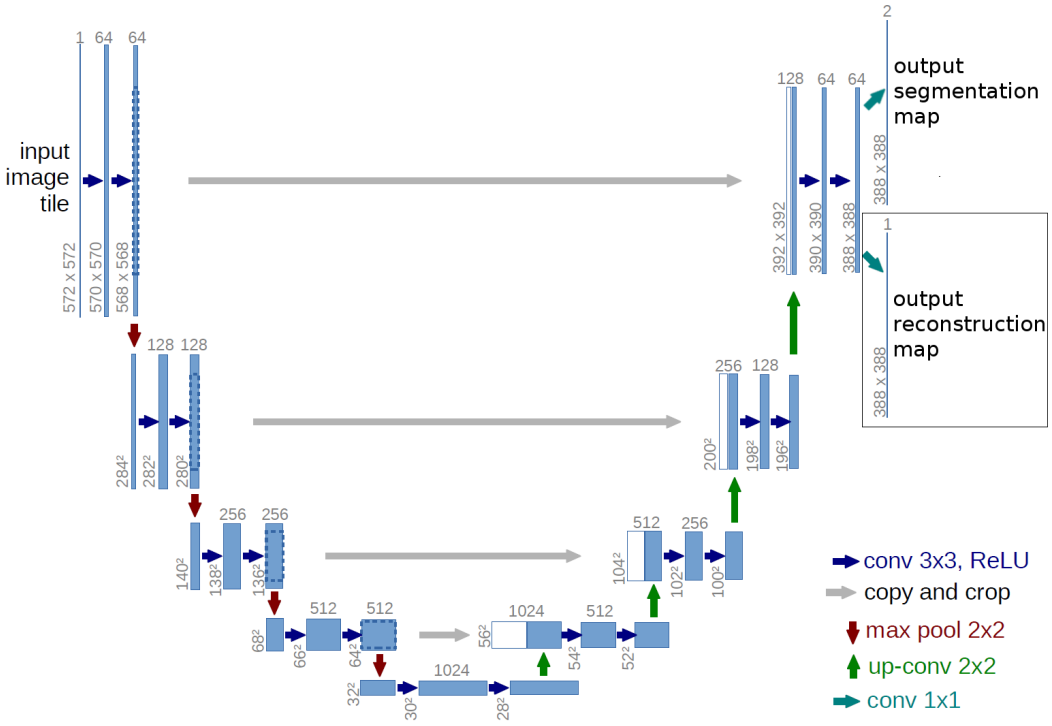


Figure 6.1: The U-net architecture from Ronneberger *et al.* (2015). The box on the right hand side indicates an additional reconstruction layer not present in the original network, but added in this study.

error) and $0 \leq \beta(t) \leq 1$ encodes the trade-off between these two loss functions. L_S could also have been the boosted cross-entropy loss function, but it was decided to keep the two parts of L as simplistic as possible for the initial testing of the proposed approach.

6.2.2 Experiments

A total of 20 experiments were performed, 10 for the purely supervised case and 10 for the pre-trained case, with the only difference between experiments being the initialization of the model parameters. The model in each experiment was trained for a total of 200 epochs using the adaptive learning rate ADADELTA and followed the 15 training - 15 testing split sampling strategy set out in Section 3.2.2. Unlike traditional unsupervised pre-training approaches, our approach only employed the labeled data that was provided for regular training. Furthermore, the exact same labeled training data was used both with and without unsupervised pre-training to facilitate comparing the two scenarios. Ten random numbers were generated which were used as seeds in both cases, hence the only difference between the two was the additional cost L_R from the reconstruction layer and the use of $\beta(t)$ to switch between two tasks.

In our experiments, we set $\beta(t)$ to the shifted sigmoid

$$\beta(t) = \frac{1}{1 + \exp(K - t)}, \tag{6.2}$$

where t is the current epoch number and K is a parameter which can roughly be seen as the epoch number at which the transition should occur. Initial experimentation showed that $K = 50$ appears to be sufficient to ensure pre-training convergence. This choice of $\beta(t)$ ensured a smooth transition to focus primarily on unsupervised learning at the start of training and supervised learning at the end of training. In the purely supervised case, $\beta(t)$ was simply set to one for all epochs.

6.2.3 Analysis

The models were evaluated over the 15 test images. The results are summarized in Table 6.1 and were then used in a battery of hypothesis tests. The results for each case are samples from some underlying distribution with the sample mean and sample standard deviation shown in Table 6.1. The two distributions are compared to each other using statistical analyses based on their means and variance (squared standard deviation). A total of six hypothesis tests were performed, testing for both equality of means and equality of variance of the two distributions at the 5% significant level.

The t-test was used to test for the equality of means: The null hypothesis states that the means of the two distributions are equal, and the alternative hypothesis states that they are not. It is important to note that the t-test assumes that the underlying distributions are both normal, which is unlikely given the close proximity of the results to the upper bound. There are two variations of the t-test that can be used, the Student's t-test (Student, 1908) and Welch's t-test (Welch, 1947), with the difference being that the Student's t-test also assumes equal variance. The assumption of normality motivated the inclusion of the Mann-Whitney U-test (Mann and Whitney, 1947), which does not make this assumption. Note that the null hypothesis for the Mann-Whitney U-test states that the medians (not means) of the two distributions are equal, which also makes it more robust.

The F-test (Shafer and Zhang, 2012), Levene's test (Levene, 1960), Bartlett's test (Snedecor and Cochran, 1989) and the Brown-Forsythe test (Brown and Forsythe, 1974) were used to test for the equality of variances: In all cases, the null hypothesis is that the variances of the two distributions are equal, while the alternative hypothesis is that they are not. Similar to the t-test, the F-test and Bartlett's test both assume that the underlying distributions are normal, with the F-test being more sensitive to non-normality. The F-test is typically used when comparing the variance of two populations, while Bartlett's test can be used for any number of populations. Since only two populations are

Table 6.1: Rand score thin metric results for all 20 experiments.

No.	Supervised Model	Pre-trained Model
1	96.17	97.53
2	97.86	97.69
3	97.12	97.96
4	96.90	97.00
5	97.88	97.25
6	97.67	97.55
7	98.39	97.76
8	98.05	98.07
9	97.46	97.50
10	96.83	97.42
Mean $\pm$ SD	97.433 ± 0.673	97.573 ± 0.317

being compared in this work, both the F-test and Bartlett's test will be reported. Levene's test and the Brown-Forsythe test, on the other hand, do not assume normality, and can both be used for any number of populations. The calculation of the Brown-Forsythe test is similar to Levene's test, but it uses the median of the population whereas Levene's test uses the mean, making the Brown-Forsythe test more robust.

6.3 Results and Discussion

The Rand score thin metric result for each model in the two cases, as well as the resulting sample means and sample standard deviations are shown in Table 6.1. The distributions of the results are also illustrated in Figure 6.3 by means of boxplots, providing a visualization of the difference between the two approaches. The resulting p-values obtained from the various hypothesis tests are shown in Table 6.2. Lastly, a qualitative comparison between the output produced by each approach for three example input images is provided in Figure 6.2.

A qualitative analysis of the output produced by the two approaches was performed in order to identify any resulting effects from pre-training the models. Arganda-Carreras *et al.* (2015) identifies 4 types of errors that the Rand score thin metric is sensitive to, namely the splitting of cells, the merging of cells and the complete addition or removal of cells. Even with this information, it is still challenging to qualitatively distinguish which approach performs better. There are some differences that are immediately apparent between the two, as pointed out by the green boxes in Figure 6.2, but in no way is it indicative of which approach is better, which leaves only a quantitative comparison.

As clearly shown in Figure 6.3, the metric results in Table 6.1 indicate that the pre-trained model performed slightly better on average and had a tighter

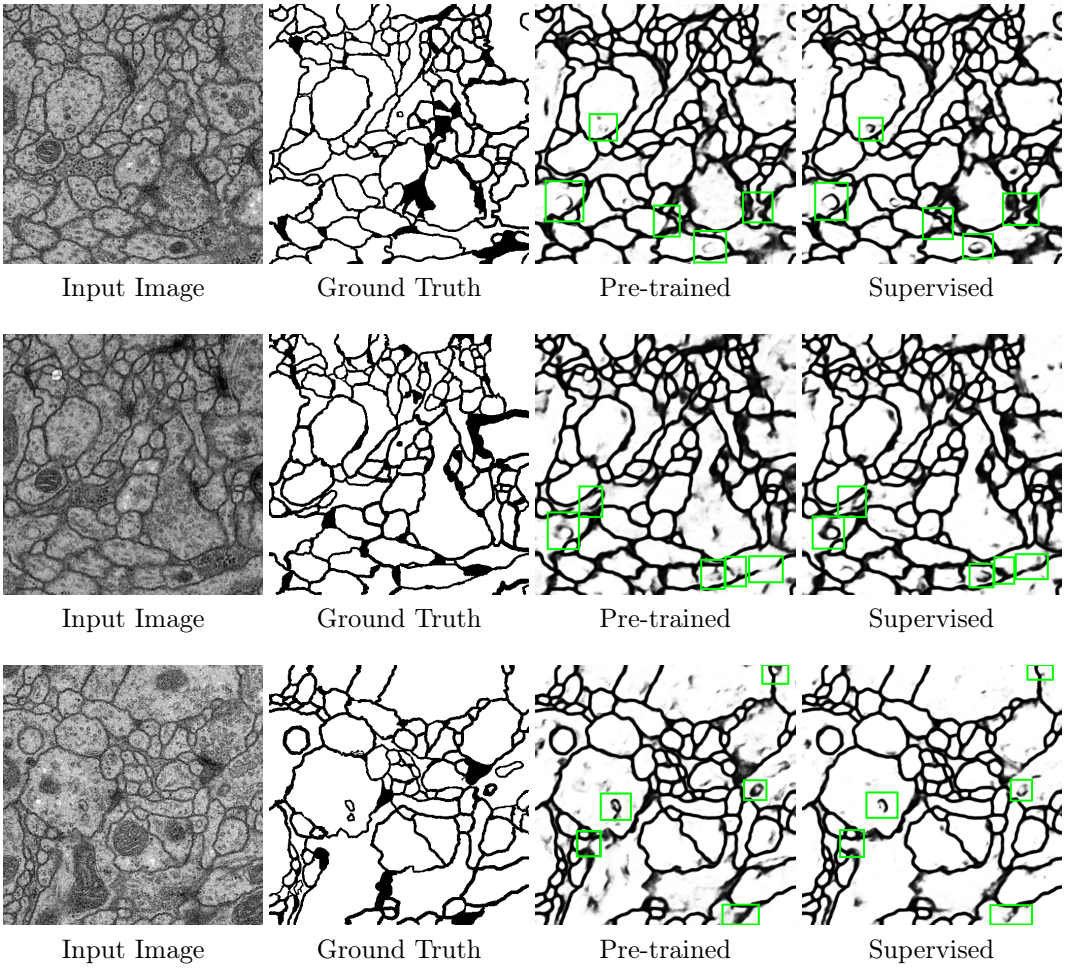


Figure 6.2: Three examples from the test set showing the segmentation output of the pre-trained model compared to the purely supervised model. Highlighted in the green boxes were some of the most apparent differences that could potentially have an influence on the Rand score thin metric value.

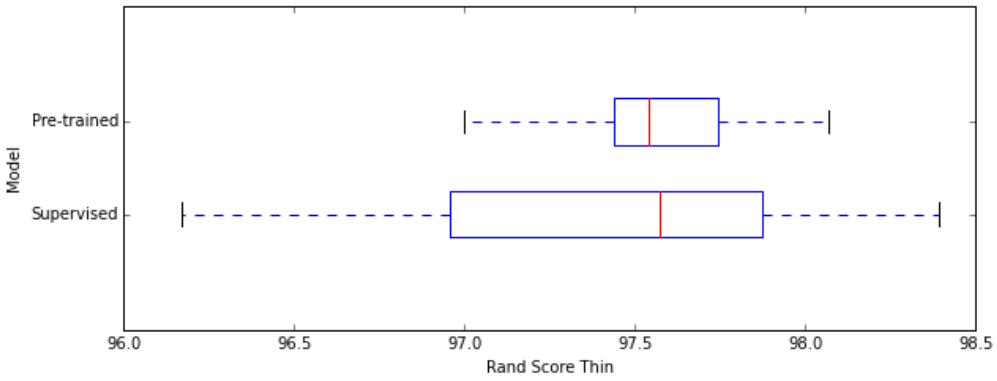


Figure 6.3: Boxplot of the distributions for the purely supervised and pre-trained models in Table 6.1 ($n = 10$ in each case). Higher values are better.

Table 6.2: The p-values for the various hypothesis tests.

Mean/Median Tests			
Student's t-test	Welch's t-test	Mann-Whitney U-test	
0.55936	0.56283	0.73373	
Variance Tests			
F-test	Levene's Test	Bartlett's Test	Brown-Forsythe Test
0.00943	0.02872	0.03553	0.04246

(lower variance) output distribution. Both variations of the t-test (Student's t-test and Welch's t-test) and the Mann-Whitney U-test failed to reject the null hypothesis of equal mean scores for the two approaches, however. This suggests that our experiments were not sufficient to detect any possible underlying difference in the average performance of the models.

Previous work suggests that models trained in a true semi-supervised setting (with few labeled data), can show improved performance over its purely supervised counterpart (Erhan *et al.*, 2010; Hong *et al.*, 2015; Kingma *et al.*, 2014; Rasmus *et al.*, 2015). However, our failure to obtain such an improvement still makes sense in this setting, due to the fact that all of the training data was labeled. The additional unlabeled data used in unsupervised training is thought to provide the classifier with some prior information on the expected distribution of the input. This allows the classifier to focus more effectively on relevant portions of the input space when the labeled training data does not adequately represent the input distribution. In our setting, since no additional unlabeled data is provided, our classifier could not benefit from this.

The remaining statistical tests, the F-test, Levene's test, Bartlett's test and the Brown-Forsythe test, evaluated the null hypothesis of equal variances of the metric under both approaches. All these tests rejected the null hypothesis at a 5% significance level, suggesting that it is improbable that the difference observed in the variance for the two approaches was by chance, despite the small sample size. Given that the only difference between the experiments for each case was their initialization, the reduction in the variance for the pre-trained model suggests that unsupervised pre-training via the reconstruction loss made the model more robust to random initialization. This result aligns well with the findings of Erhan *et al.* (2010), who make a case (for much smaller networks) that unsupervised pre-training acts as a regularizer which adds robustness against random initialization and as such, reduces the variance in the model performance.

The process of converting a purely supervised FCN to one capable of undergoing unsupervised pre-training in the form of input reconstruction appears to be fairly straightforward. This process is dependent on the FCN producing an output of the same scale as the original input, that is the FCN is used to produce a semantic segmentation map of the input. The particular architecture of

U-net, specifically the presence of the skip connections, did pose an interesting challenge. The skip connections could potentially act as short-cuts during unsupervised training, leading to little or no benefit for the deeper levels. Upon further investigation using the same analysis approach as in Section 5.3, this was indeed the case.

By zeroing the weights of the individual skip connections on the various levels of the architecture and measuring the difference in performance, it was found that the reconstruction after pre-training was entirely dependent on the top-most skip connection. This suggests that more work is required in augmenting an FCN with skip connections to allow unsupervised pre-training that is beneficial to the entire network, not just a small portion of it. One such approach would be to have multiple reconstruction layers, one on each level in the architecture, with the objective of reconstructing the input for the respective level it is attached to. Alternatively, convolutional autoencoders have been shown to require strict regularization rules in order to prevent the trivial solution from being learned. The lack of benefit from end-to-end autoencoding in FCNs suggests that similar regularization rules might be required to learn more meaningful encodings in the various layers. For example, one could somehow encourage the network to use the deeper pathways by penalizing it differently for using the skip connections at various depths.

6.4 Conclusion

This chapter investigated unsupervised learning in the form of input reconstruction that is typically used in autoencoder layers. We proposed a novel augmentation for FCNs which allowed end-to-end unsupervised learning of this form to be used as pre-training step. Analysis suggested that performing unsupervised pre-training provides a statistically significant reduction in the variance of the model performance compared to a purely supervised FCN. This reduction in variance further supports the generalizer hypothesis of Erhan *et al.* (2010), which suggests that unsupervised pre-training adds robustness to the model against random initialization, reducing the model variance accordingly.

The work done in this chapter also indicated that further development of the approach is required. We observed that the skip connections in the U-net architecture allowed unsupervised learning to bypass the deeper levels of the network, suggesting that a more robust approach is needed to reap the full benefits of unsupervised learning. The approach might, like convolutional autoencoder layers, require stricter regularization rules during unsupervised training to ensure that a more meaningful encoding is learned in each layer of the architecture. Although the proposed end-to-end autoencoding approach is still incomplete, we showed that it does hold promise as an unsupervised learning technique.

The next chapter will further investigate our approach of unsupervised pre-training by applying both the supervised model and the pre-trained model in a more conventional semi-supervised setting.

Chapter 7

Partial Labeling

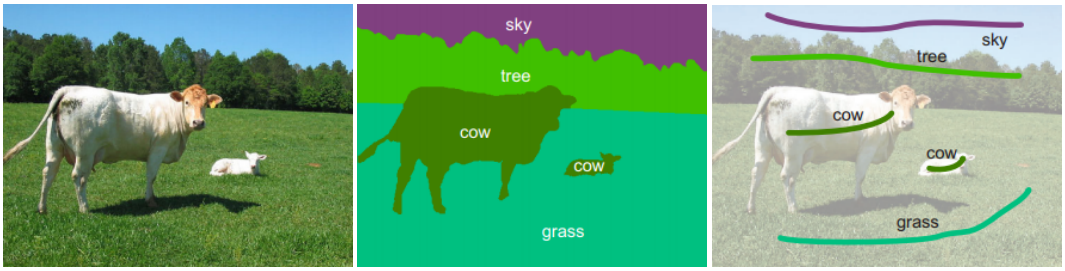
In the previous chapter, we proposed a novel augmentation to FCNs to facilitate end-to-end unsupervised pre-training. It was shown that unsupervised pre-training made the models more robust to random initialization; however, it was also found that the majority of the architecture did not benefit from pre-training. The proposed approach holds promise as an end-to-end unsupervised learning technique, but further investigation is required to fine-tune the approach.

This chapter instead focuses on creating a more conventional semi-supervised setting to investigate whether our pre-training approach shows similar improvements as in previous semi-supervised work under such conditions. Given that the nerve cell membrane data set is fully annotated, we can create a semi-supervised setting by removing large portions of the available pixel labels. As such, this chapter presents a practical partial labeling approach for use with this data set, and aims to explore how the augmented U-net architecture (Section 5.1.1) performs under various levels of sparsity of pixel labels. Furthermore, the U-net architecture will be trained on the resulting data both with and without the added reconstruction layer to compare the two approaches.

Section 7.1 starts by presenting a practical approach to partially label the images in the nerve cell membrane data set. Section 7.2 then provides a detailed description of the alterations that had to be made to the experimental setup to accommodate the partial pixel labels. Lastly, Section 7.3 presents the results that were obtained and the observations that were made.

7.1 Partial Labels

While fully annotated ground truth refers to images where most if not every individual pixel is given a semantic label, partial labeling refers to situations where only a (typically small) fraction of the pixels in the image are given semantic labels. There is no absolute partial labeling approach, but there are some techniques that have been shown to work in practice. Note that partial



(a) Input Image (b) Fully Annotated Mask (c) Scribble Annotation (Partial)

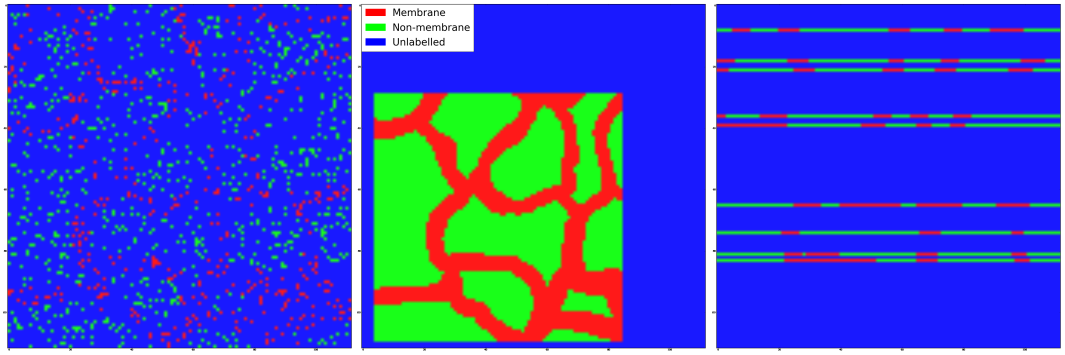
Figure 7.1: An example depicting a general image with its corresponding fully annotated ground truth and scribble annotation. The images were extracted from Lin *et al.* (2016).

labeling should not be confused with weak labeling. Recall from Section 6.1 that weakly labeled data does not involve individual pixel labels, but only indicates the object of interest in the image or the general location of it in the case of bounding boxes.

One of the most widely used partial labeling techniques is referred to as the *scribble* technique (Delong *et al.*, 2011; Diebold *et al.*, 2015; Lin *et al.*, 2016). The scribble technique involves the annotator drawing a curve inside of each object of interest. All of the pixels that fall on the curve for a particular object are then annotated as belonging to the corresponding object class. All of the remaining pixels in the image that do not occur on a curve are then considered to be unlabeled. An example of the scribble technique from Lin *et al.* (2016) is given in Figure 7.1.

Although the nerve cell membrane data set does not contain partially labeled data, the availability of fully annotated ground truth enables the rapid exploration of multiple configurations of partial labeling. Given a fully annotated image, an automated solution can be created to extract the labels of random pixels according to some pre-defined selection rule. By changing the seed of the random number generator, a different collection of partial labels can be obtained. The selection rule in this instance simply specifies which pixels should be extracted as part of the partial labeling, while non-selected pixels are discarded.

The ideal selection rule for our experiments would be one that allows a progressive increase in the number of pixel labels. A number of selection rules were considered, the first of which involved simply extracting one or more random pixels from the ground truth (see Figure 7.2a). The number of selected pixel labels can easily be controlled by changing the probability of a pixel being selected. However, using a selection probability does introduce some inconsistencies in the number of partial labels, especially when the probability is small. This is not ideal for our purposes, as we want to compare different



(a) Random Pixels (b) Random Regions (c) Random Lines

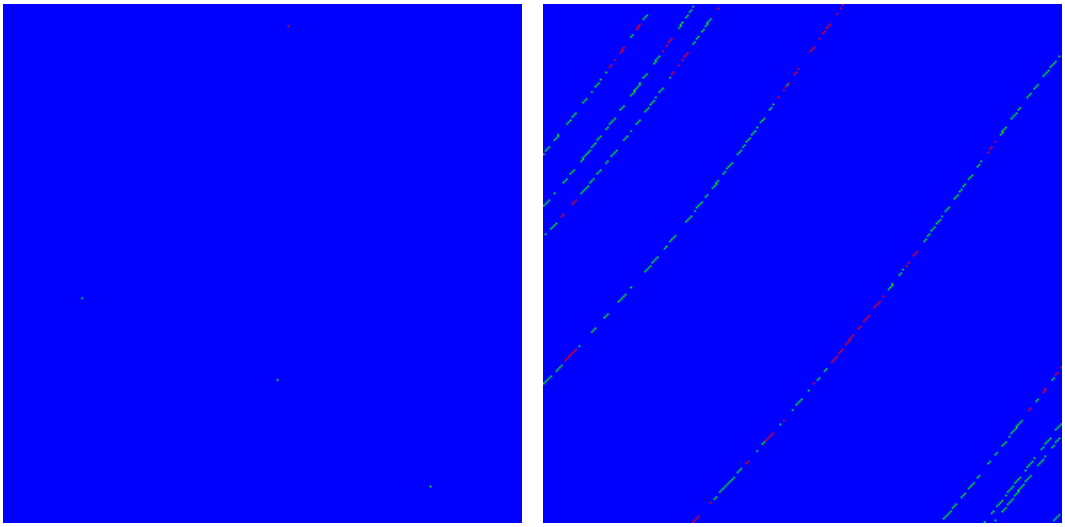
Figure 7.2: A selection of programming solutions to extract random pixel labels from the fully annotated ground truth. Figure 7.2a shows an example where single pixel labels were selected at random with a probability of 10%. Figure 7.2b shows an example of a specified region (in this case an 81×81 square) around a randomly selected center pixel. Lastly, Figure 7.2c shows an example of the lines approach. Membrane pixels are represented in red, cell pixels in green and unlabeled pixels in blue (see legend in Figure 7.2b).

selections of partial labeling, all with an equal total number of pixel labels.

The second selection rule that was considered was to extract random regions of pixel labels rather than single pixel labels. In this instance, the region can have an arbitrary shape and size, and any number of regions can be extracted from the ground truth. For example, consider Figure 7.2b where a single 81×81 square of pixel labels is extracted as the partial label for the respective input image. This approach is similar to the scribble technique, in that each curve in the scribble annotation can also be considered as a random region of pixel labels. Unfortunately, this approach is also inconsistent in the number of partial labels, in that controlling the number of selected pixel labels is more difficult (depends on the size and shape of the region, the number of regions and overlapping regions).

The final selection rule that was considered, was to select random horizontal lines across the ground truth, creating a horizontal line mask (see Figure 7.2c). This meant that a single horizontal line over a 512×512 image would be 512 pixels long and 1 pixel wide. The lines are sampled without replacement, thus producing a consistent number of selected pixel labels for different selections of n lines — for instance, 10 random lines would always yield 5120 pixel labels no matter which lines were selected. This approach can also be thought of as a particular instance of the scribble annotation technique, only with straight, one pixel wide horizontal lines as curves.

We chose the random horizontal lines approach for our experiments, as it is easy to reproduce and is relatively straightforward to scale to different numbers of pixel labels, while still being analogous to the real-world scribble annotation



(a) Original Thresholds (b) New Thresholds

Figure 7.3: An illustration showing the difference between the original thresholds in the calculation of the label vector compared to the new thresholds. Notice that there are only a few pixel labels visible in Figure 7.3a that persisted through all of the transformation calculations. Using the more lenient thresholds (Figure 7.3b), more pixel labels are retained with highly uncertain pixels still being rejected. As before (Figure 7.2b), red illustrates membrane pixels, green the cell pixels and blue the unlabeled pixels.

technique.[1] Each set of experiments was then obtained from the previous set by increasing the number of labeled lines by approximately a factor of either 5 or 2. Starting with one random line per training image, this resulted in the following cases that were investigated: 1 line, 5 lines, 10 lines, 51 lines and 102 lines. The next case in the sequence would be 512 lines, which is equivalent to using the fully annotated ground truth as in Chapter 6.

Using lines that are only one pixel in width in conjunction with the random transformations described in Section 3.2.2 results in many pixels on the selected lines being discarded in the calculation of the label vector (Equation 3.10). This occurrence can be seen in Figure 7.3a, which shows that only a few pixel labels are retained with the original thresholds. In order to prevent this loss of labeled pixels while still maintaining the transformation strategy, Equation 3.10 was modified to be more lenient in treating the uncertainty of

[1]Note that such labelings can be obtained efficiently by having an annotator indicate the class of the first pixel on each line, as well as the points on the lines where the pixel class changes.

pixel labels. As such, the calculation of the label vector was changed to

$$v(i,j) = \begin{cases} (1,0), & \text{if } 0 \leq c(i,j) < 50 \\ (0,0), & \text{if } 50 \leq c(i,j) \leq 205 \\ (0,1), & \text{if } 205 < c(i,j) \leq 255 \end{cases} . \tag{7.1}$$

This choice of thresholds retained a fair number of transformed pixel labels while still rejecting transformed pixel labels that are considered to be too uncertain (see Figure 7.3b). Note that using this choice of thresholds on the fully annotated data would simply reduce the size of the uncertain regions between the cell and the membrane classes, and would likely not effect the performance of the model.

7.2 Experimental Setup

There are a number of experimental variables that had to be set in order to perform comparable experiments. It was determined that the random lines approach described in the previous section would be the most convenient to allow a natural progression in increasing the number of labeled pixels. However, since the lines across the ground truth are selected at random, the training effectiveness of a neural network using these lines becomes dependent on the choice of lines. Section 7.2.1 describes how the lines were generated for each case, while a description of the experiments that were performed follows in Section 7.2.2.

7.2.1 Line Generation

The experiments were initially performed using a fixed selection of lines. That is, all of the experiments for a particular case used the exact same partial labels. It was later realized that multiple experiments using the same partial labels would only describe the average performance of the neural network on that particular selection of lines. As such, it is possible that a different selection of lines could yield better or worse performing models compared to the selection considered.

For example, consider the 1 line case and the task of correctly segmenting a cellular object (such as the cell nucleus in Figure 5.7a) as belonging to the cell class. A bad choice of lines would be where none of the lines intersect the cellular object, thus leaving the model with no guidance as to how it should segment it. The resulting model would likely perform worse than a model trained on lines that did intersect the cellular object. Conversely, there could also be an optimal choice of lines that could yield the best performing model for a particular n-line case.

Consequently, the selection of lines were randomly generated for each experiment using the same seed that was used to initialize the weights of the

neural network. As it is not known which choice of lines would be the best choice to use, generating a new set of lines for each experiment should provide a better indication of the overall average model performance. As such, the average model performance obtained for each n-line case can be interpreted as the expected model performance for any particular choice of lines within the respective case.

7.2.2 Experiments

As mentioned in the previous section, five cases of partial labeling were investigated using the lines approach, with each case increasing the number of lines per training image. The five cases that were explored are 1 line, 5 lines, 10 lines, 51 lines and 102 lines. For each case, two sets of experiments were performed, with one set using the augmented U-net architecture without the reconstruction layer (referred to as the *supervised model*) and the other using the augmented U-net architecture with the reconstruction layer (referred to as the *pre-trained model*). Each set consisted of 30 repetitions of the respective model, with each repetition using a different initialization seed. Note that the initialization seeds were kept consistent across all sets of experiments.

The two versions of the model were trained using the adaptive learning rate ADADELTA and followed the 15 training - 15 testing split sampling strategy (see Section 3.2.2). Similar to the experiments in Chapter 5, the standard cross-entropy loss function was replaced with the boosted cross-entropy loss, putting the focus on correctly predicting as many of the available labels as possible. In Chapter 6, both the supervised model and the pre-trained model were trained for a total of 200 epochs. This meant that the pre-trained model effectively underwent 50 epochs fewer of supervised learning, since the first 50 epochs were used for unsupervised learning. The reduced supervised training time of the pre-trained model was not considered to be a problem in Chapter 6, as the fully annotated ground truth allowed the models to converge considerably earlier than the end of training. The additional 50 epochs of supervised training that the supervised model received could make a difference in the convergence of the models when considering partial labels. As such, the training duration of the pre-trained model was increased by 50 epochs, to ensure that both versions of the model received an equal amount of supervised training time. Thus, the supervised model was trained for a total of 200 epochs, while the pre-trained model was trained for 250 epochs using the same $\beta(t)$ with $K = 50$ as in Chapter 6.

By increasing the training duration of the pre-trained model, this also meant that the model would see an additional 50 epochs worth of training data that the supervised model did not. Since the goal of experimenting with both versions of the model is two compare the two under different conditions of partial labels, it would be best if the two versions used the exact same training data. The labeled training data of the first 50 epochs is only partially

utilized in the pre-trained model, as it mainly focuses on unsupervised learning as a result of the transition function $\beta(t)$. This was rectified by duplicating the first 50 epochs of training data, such that it could be used for both pre-training and for supervised learning. This could also be thought of as shifting the labeled training data used for supervised learning to epochs 50 to 250, thereby ensuring that both versions of the model use the same labeled data for supervised learning, and allowing the pre-trained model to undergo pre-training on the first 50 epochs of training data. In short, for the full duration of 250 epochs, the original training data for epochs 1–50 was used for pre-training, followed by the original training data for epochs 1–200 which was used for supervised learning.

During the initial experiments, it was also observed that the some of the models experienced a spike in their validation pixel error near the end of training (see Figure 7.6). If the peak of this spike occurred on the final epoch of training, which is used for the Rand score thin metric evaluation, the model would show a worse performance compared to when the evaluation is performed at neighboring epochs. For instance, the model in Figure 7.6 was part of the 5 line case and yielded a Rand score thin metric value of 62.52 when evaluated at epoch 250. Evaluating the same model 5 epochs sooner (epoch 245) before the spike yielded a Rand score thin metric value of 95.08. This suggested that there might be some instability in the training data when using partial labels.

In order to avoid this instability affecting the results, it was decided to rather evaluate each model at five different epochs around the final epoch. The supervised model was evaluated at epochs 190, 195, 200, 205 and 210. The pre-trained model was evaluated at epochs 240, 245, 250, 255 and 260. Using a similar approach as early stopping, the best performance over the five evaluation points was then used as the final performance of the model. Unfortunately, it is not possible to use regular early stopping based on the validation pixel error of the model: Although there appears to be a highly negative correlation between the validation pixel error and the Rand score thin metric, the relationship between the two is not perfect (see Section 7.3.2).

7.3 Results and Discussion

The results reported in this section will be for the final experiments that were performed. That is, the mean and standard deviation over the 30 models in each set, calculated using the maximum performance over the five evaluation epochs for each model. The results for each case are given in Table 7.1 in the form *mean* $\pm$ *standard deviation*, and are also plotted in Figure 7.4 for a visual illustration. Note that the same 30 initialization seeds were used for each set of experiments, which means that the difference between two cases observed in the results can be attributed to the difference in the (amount of) labeled

Table 7.1: The means and standard deviations calculated over 30 repetitions of the respective model version for each n-line case. The results from Chapter 6 where the full ground truth (equivalently 512 lines) was used is included for comparison. Note that the methodology of Chapter 6 differed from that of this chapter.

Lines	Supervised Model	Pre-trained Model
1	92.211 ± 1.928	91.478 ± 2.038
5	95.808 ± 1.143	95.290 ± 1.019
10	96.572 ± 0.690	96.185 ± 0.674
51	97.227 ± 0.519	97.111 ± 0.498
102	97.316 ± 0.345	97.261 ± 0.358
512 (Chapter 6)	97.433 ± 0.673	97.573 ± 0.317

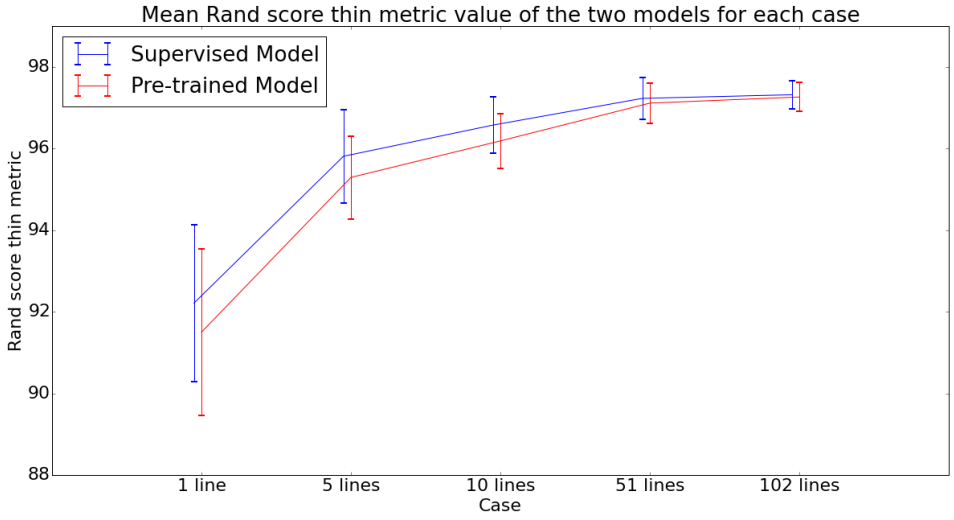


Figure 7.4: A visual representation of the results in Table 7.1.

data. The results of the final experiments will be discussed in Section 7.3.1, followed by the findings based on initial experimentation in Section 7.3.2.

7.3.1 Final Experiments

This section will discuss the observations that can be made from the results in Table 7.1. The first observation that is immediately clear is that the mean performance of the networks improve as the number of lines increases. This conforms with the notion that increasing the amount of labeled data presented to a model (in a meaningful way), typically improves the performance of the model — provided that the model has the necessary complexity to perform the task in the first place (Banko and Brill, 2001). We also observe that the standard deviation decreases as the number of lines increase. Recall that both the

initialization of the models and the choice of lines change for each repetition. The reduction in the standard deviation is expected when considering only the change in the model initialization, as providing a model with more training data typically leads to improved and more stable model performance. With the additional change in the choice of the lines, the reduction in the standard deviation could also suggest that by increasing the number of lines, the models become more robust to the specific choice of lines.

The work done in Chapter 6 showed a significant reduction in the variance of the models when using unsupervised pre-training. It was expected based on other semi-supervised approaches, that the pre-trained model would have shown improved performance compared to the supervised model. However, this was not the case, as the two versions of the model seemed to show similar performance given the small differences observed between the means and standard deviations. The similar performance between the two versions of the model suggest that the current approach of performing unsupervised pre-training does not provide any benefit to the model. Since these results conflict with those of Chapter 6, one explanation could be that the changes made to the experimental approach affected the results. Alternatively, the results observed in Table 7.1 could also support one of the conclusions made in Chapter 6, which is that the current unsupervised pre-training approach is still incomplete and that more work is required to improve the approach.

The choice of the transition function $\beta(t)$ could also have affected the performance of the pre-trained model. Considering that the transition function is a shifted sigmoid, it should never reach a value of $\beta(t) = 1$. This means that it might be possible that some of the reconstruction cost leaks through to the gradient calculations, thereby preventing learning from focusing on the supervised task. Although the leaked reconstruction cost would be small, it might still have some unforeseen effect on the performance of the model. The transition function was then investigated by considering two alternative functions. The first function was used to determine whether the reconstruction cost leaked through to the gradient calculations during supervised learning, and is given by

$$\beta(t) = \begin{cases} \frac{1}{1+\exp(K-t)}, & \text{if } t \leq 100 \\ 1, & \text{if } t > 100 \end{cases}. \tag{7.2}$$

This choice of transition function showed no difference in the final performance of the models, indicating that the reconstruction cost had no effect on the performance of the models when focusing on supervised learning. The second function was then used to investigate the smooth switching between unsupervised and supervised learning by using a shifted Heaviside step function, given by

$$\beta(t) = \begin{cases} 0, & \text{if } t < 50 \\ 1, & \text{if } t \geq 50 \end{cases}. \tag{7.3}$$

Table 7.2: The results from experimenting with a different choice of transition function, applied to a single experiment from each case.

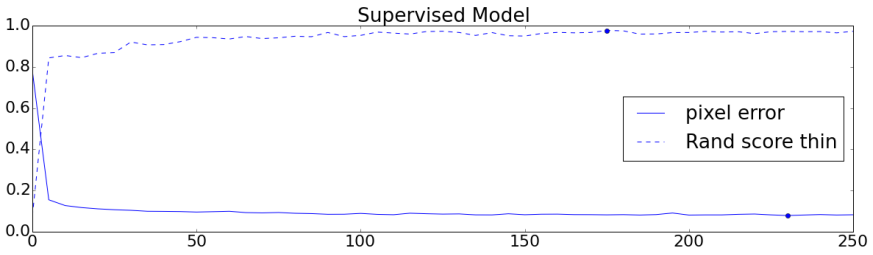
$\beta(t)$	1 Line	5 Lines	10 Lines	51 Lines	102 Lines
Shifted sigmoid	89.23	94.54	95.99	97.23	96.94
Shifted Heaviside	91.44	95.16	95.50	97.23	97.94

This choice of transition function was applied to a single experiment from each case, with the results summarized in Table 7.2. Using the shifted Heaviside step function instead of the shifted sigmoid function was found to improved model performance in three out of the five experiments. These results show that the choice of transition function could have a positive or a negative effect on the performance of the pre-trained models. To determine whether the shifted Heaviside step function is the better choice for any of the n-line cases, all 30 of the original n-line experiments need to be repeated using this choice of transition function.

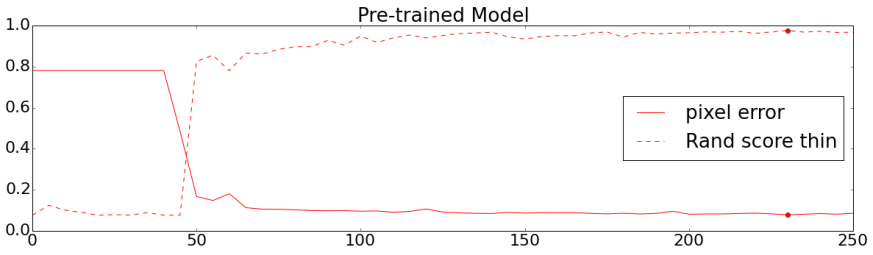
Another interesting observation is how well the models performed even with the reduced amount of labeled data. Consider for example the 1 line case, which only contained about 0.2% of the total number of pixel labels provided in the fully annotated ground truth. Both models managed to achieve comparable performance to a number of approaches on the public leader board of the data set (Shaar, 2012), with all of these approaches using the full ground truth. To add some perspective to this result, consider that using a fixed threshold to generate segmentations from the original images only achieved a Rand score thin metric value of 72.45. This suggests that only a few data labels are required to achieve near state-of-the-art performance, with more labels leading to even better performance. Note that this result might be specific to this application domain, and that applying these experiments on other domains may not show similar results.

7.3.2 Initial Experiments

It is also worth discussing some of the results of the initial exploratory experiments that were performed prior to those reported in Table 7.4, as these experiments also provided some key insights into the current training approach. One observation was that while there us a clear negative correlation between the validation pixel error and the Rand score thin metric value, the relationship between the two is not perfect. This can be seen in Figure 7.5, where both a supervised model and a pre-trained model were trained for 250 epochs using the fully annotated ground truth. The dot on each line represents the maximum Rand score thin metric value and minimum pixel error for the respective lines. It is clear that a model showing a reduction in the pixel error through training would generally also show improved performance on the Rand score



(a) Supervised model



(b) Pre-trained model

Figure 7.5: An illustration of how the Rand score thin metric value and validation pixel error changes over the entire training duration for both the supervised and the pre-trained model. The maximum Rand score thin metric value and minimum pixel error value is indicated by the dot on the respective line for each model. Both models were trained using the fully annotated ground truth for a duration of 250 epochs.

thin metric. Since the relationship is not perfect, however, the model with the best pixel error might not achieve the best Rand score thin metric value. This is indicated in Figure 7.5a where the maximum Rand score thin metric value and the minimum pixel error occur at different epochs during training. Consider another example, where a pre-trained model was trained for 250 epochs using the 5 lines partial labeling case. The version of the model that showed the best validation pixel error only achieved a Rand score thin metric value of 93.62, while the end-of-training version of the model achieved a better metric value of 95.71. These results suggested that the validation pixel error is a poor indicator for finding the best-performing model with respect to the Rand score thin metric.

The second observation made from the initial experiments was that there appears to be certain points during training where the model shows a spike in the validation pixel error (see Figure 7.6). This became particularly problematic when the peak of the spike occurred during the final epoch of training, as the resulting model yielded a much lower Rand score thin metric value. A likely scenario that could cause the sudden drop in the performance of the model is the use of a bad mini-batch of training examples in conjunction with ADADELTA: Firstly, it is important to realize that by keeping a moving aver-

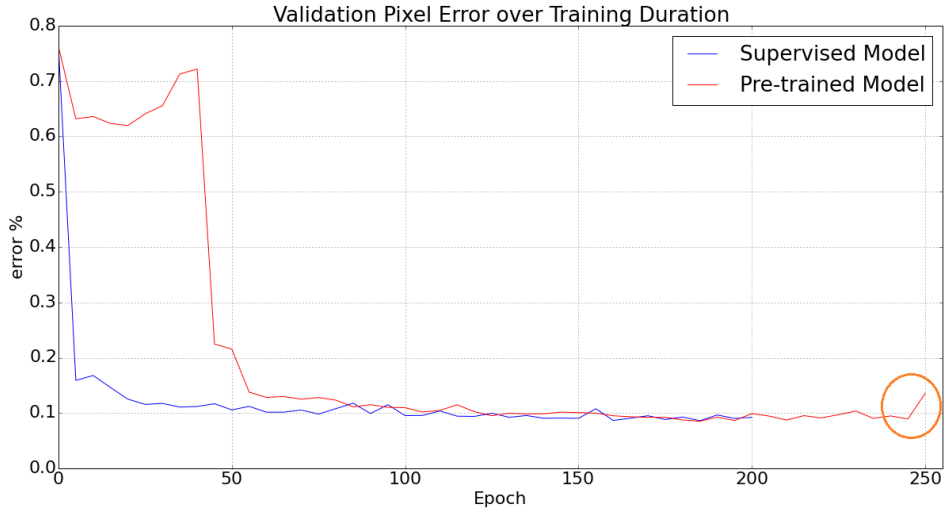


Figure 7.6: The validation error of both a supervised model and a pre-trained model over the duration of training, measured every 5 epochs. Notice the large spike in the validation error near the end of training of the pre-trained model (highlighted by the orange circle), which is considered to be irregular behaviour when compared against the validation error for the epochs before the spike.

age of both the gradients and the parameter updates, ADADELTA is able to react much faster based on the gradients it receives. This allows ADADELTA to greatly reduce the amount of time required for training, but also means that ADADELTA is more sensitive to a sudden change in the gradients as a result of new training data. It might be possible that a bad mini-batch of training data could have a significantly greater effect on the performance of the model when using ADADELTA instead of a slower learning approach such as momentum. Secondly, within the context of the random lines partial labeling approach, there are a number of ways one might obtain a bad mini-batch of training examples. One way would be a bad selection of lines such as lines with high class imbalance, or a poor choice of transformations. This suggested that the current sampling approach might not be the best choice to use with partial labels when considering a single evaluation point.

7.4 Conclusion

This chapter focused on creating a more conventional semi-supervised setting by removing large portions of the pixel labels from the nerve cell membrane data set. This was achieved by using the 'random lines' approach, where a set number of horizontal lines were chosen per training image to act as partial labels. Five cases of differing number of lines were explored, starting from 1 line per training image up to 102 lines per training image. Both the supervised

model and the pre-trained model were trained on each case, and the results were compared.

As expected, it was found that increasing the number of lines per training image resulted in an increase in the mean and a decrease in the standard deviation in the test scores of both models. Unfortunately, no significant difference was observed between the performance of the supervised model and the pre-trained model, which suggests that the current unsupervised learning approach still requires further investigation to establish its usefulness. It was also found that the choice of transition function could affect the final performance of the pre-trained model, and that further investigation is required to find an optimal choice of $\beta(t)$. Initial experimentation showed that there appears to be a negative correlation between the validation pixel error and the Rand score thin metric value, but the relationship is not perfect. This meant that the model with the best validation pixel error might not yield the best Rand score thin metric value. In some of the initial experiments, a spike in the validation pixel error was observed, which suggested that a more robust sampling strategy might be required when working with partially labeled data. Despite these results, we showed that for this application domain, it is possible to achieve competitive performance with only a few of the original pixel labels.

The next chapter will conclude this work by providing a summary of our findings and discussing possible avenues to explore in future work.

Chapter 8

Conclusion

This chapter will conclude this work by providing a detailed summary of the investigations that were performed and the conclusions that were drawn (Section 8.1). Section 8.2 presents the possible avenues that can be explored for future work, followed by a short summary of the contributions made by this work in Section 8.3.

8.1 Investigation Results

This thesis employed two bio-image data sets that were used as training data for a variety of FCN architectures. The first data set, the *C. elegans* live/dead assay data set, was used to train a conventional CNN and three FCNs with varying depth. All of the models showed an improvement on the appropriate metrics over previous work, with the FCNs showing better performance than the CNN.

The investigation of these architectures also led to a number of interesting observations. First, the class imbalance problem was found to manifest slightly differently in semantic segmentation tasks. In regular classification tasks, the class imbalance problem causes a classifier to become biased to predicting the majority class, typically causing the minority examples to be misclassified. In semantic segmentation tasks, the bias caused by the class imbalance affects the representation of the foreground objects in the segmentation, thus causing over-/underrepresentation as per the definitions in Section 3.1.1. Secondly, the poor or uneven illumination in some of the *C. elegans* data set images was found to obscure some key morphological information, such as the border between worm and background and the texture of the worms. Lastly, the contextual window of the network was found to play a key role in improving the pixel-level classification performance, with larger windows resulting in better performance. This investigation was presented as a peer-reviewed paper at the 2016 International Joint Conference on Neural Networks (Wiehman and De Villiers, 2016).

At the time of this investigation, we found an established FCN architecture named U-net which had been applied to the second data set, the nerve cell membrane data set. The U-net architecture was chosen as the base architecture for the remainder of the investigations in this thesis.

The U-net architecture was separated into the two main design aspects that is characteristic of FCNs: the various levels of the architectures (the depth of the network or equivalently the number of feature map resolutions — each separated by a pooling layer) and the presence or absence of various skip connections. The depth of the network was investigated separately, before gradually adding the skip connections to the architecture. It was found that improved performance can be achieved by increasing the depth of the network (without skip connections), up to a depth limit of three levels. Increasing the depth of the network beyond this limit caused the network to not retain enough information to produce a meaningful segmentation. We conjectured that this depth limit could be different between application domains, so further testing is required on other domains. Adding skip connections further improved the performance of the networks, with performance starting to saturate after four levels. This saturation point was hypothesized to also be dependent on the application domain. This investigation then proceeded by zeroing the connection weights of various layers that correspond to either of the two design choices after training. This procedure effectively removed the respective layer from the architecture when producing segmentation maps, providing some indication as to how much that particular layer contributes towards the overall functioning of the network. Visualizing the output produced by the model after zeroing the various layers associated with the depth of the network suggested that each level in the architecture performs a specific function. Furthermore, the deepest level of the U-net architecture was found to make an insignificant contribution towards the performance of the network. Lastly, the level of contribution of each skip connection towards the overall performance of the network appears to fluctuate depending on the initialization of the model.

The investigation then shifted towards applying the concept of unsupervised learning through autoencoding to the entire FCN architecture. We proposed a novel augmentation to the FCN architecture which allowed end-to-end unsupervised learning. Unsupervised learning was then used as a pre-training step, which was found to provide a statistically significant reduction in the variance of the model performance. This investigation was presented as a peer-reviewed paper at the 2016 conference of the Pattern Recognition Association of South Africa (Wiehman *et al.*, 2016).

In the literature, using unsupervised learning as a pre-training step is often more beneficial in a semi-supervised setting (little labeled data and a large amount of unlabeled data). A more conventional semi-supervised setting was thus created by removing large portions of the pixel labels in the nerve cell membrane data set. Both the supervised model and the pre-trained model were tested on this partially labeled data, but no significant difference was found

in the means and standard deviations of the model performances. A rather surprising result was that as little as 0.2% of the original labeled data allowed the models to achieve competitive performance on the nerve cell membrane data set. This means that partially labeled data is nearly as effective a training resource as the fully labeled data in this application domain, thus potentially saving both time and financial resources annotating a data set.

8.2 Future Work

There are a number of avenues that can be explored on the subject of investigating FCNs. This section will discuss some possibilities for future work, both to improve on the work done in this thesis, as well as to further expand our understanding of FCNs.

First and foremost, we can increase the number of data sets used in our investigations. Each set of results in this thesis were generated using only one of the two data sets. As such, the results we obtained could either be specific to the data set that was used, or it could be behavior common to multiple data sets. Repeating our experiments on more than one data set would allow us to better establish the generality of our results.

Following on the work in Chapter 4, the poor/uneven illumination levels in the images of the *C. elegans* data set can be corrected. Image processing techniques can be used to adjust the illumination levels in each image, making the difficult-to-distinguish worms more visible for the neural networks. The different architectures can then be retested on the corrected data to determine whether results could be improved further. Alternatively, the training data can be adjusted to train the neural networks to become invariant to differences in illumination levels.

Also following on the work in Chapter 4 would be to implement a worm extraction approach that does not depend on the single worm segmentation masks. The worm-level results of the neural networks on the *C. elegans* data set were approximated by using the single worm segmentation masks to extract the segmentation of each worm. The same single worm segmentation mask was then used to evaluate the quality of the segmentation of the worm. Implementing an approach to extract the pixel segmentation of each individual worm without relying on the provided single worm segmentation masks could yield more accurate results. Furthermore, such an extraction approach would make neural networks viable for application when single worm segmentation masks are not available.

We can improve the accuracy of the means and standard deviations for the experiments in Chapter 5, which investigated the two main design aspects that characterize FCN architectures. The small number of repetitions (either 3 or 4) performed were only enough to provide an initial idea of the performance contribution of each layer. However, more repetitions would provide a better

estimation of the means and standard deviations, which can either strengthen current conclusions or provide entirely new insights.

Following on the work in Chapter 6, finding a more sophisticated approach to perform end-to-end unsupervised learning can be investigated. The unsupervised learning approach used in Chapter 6 showed the viability of the approach as well as the flaws in the approach that need to be corrected. The current approach is only beneficial to the first level in the architecture, as the top-most skip connection allows the reconstruction task to bypass the deeper levels of the architecture. One solution would be to have multiple reconstruction output layers, one on each level, with the task of reconstructing the input for the respective level. Another approach to consider would be to prevent the reconstruction task from using the skip connection on the corresponding level, that is, forcing the reconstruction task to pass through at least one max-pooling layer. A different activation function can be used for the reconstruction layer, which can also be accompanied by a different reconstruction cost function. A denoising component can be added to the reconstruction task by requiring the network to reconstruct the original images from corrupted inputs — this would be analogous to the use of denoising autoencoders (Vincent *et al.*, 2010). Alternative choices for the transition function $\beta(t)$ such as the Heaviside step function can also be explored as it was found that the choice of transition function could affect the final performance of the models.

Based on the work in Chapter 7, we found that a more stable sampling technique is required for training. By randomly generating new training data every epoch, the neural networks would rarely see the same training data twice during a full training session. This significantly reduced the risk of the networks overfitting on the training data, but it was found that it could also affect the final performance of the network. Each mini-batch of training data has a chance of containing a poor representation of the input space, which in conjunction with ADADELTA could cause an anomaly during training. The risk of a poor training batch appears to increase as the amount of partially labeled data decreases. One solution would be to generate a large amount of new training data beforehand and train on the full expanded data set each epoch. This approach would significantly increase the duration required for training, due to the expanded data set, but the training data will be consistent every epoch and the model performance should show a stable improvement as training progresses.

We can also explore the use of alternative learning rate approaches. The benefit of ADADELTA is that it does not require a global learning rate to be set, thus removing one hyperparameter that needs to be fine-tuned. Some of the experiments in this thesis have shown that ADADELTA could cause unfavorable reactions to a sudden change in the gradients as a result of one or more bad mini-batches. A common choice of optimizing gradient descent is to use momentum, which may greatly increase the duration of training but it is generally less sensitive to sudden changes in the gradient than ADADELTA.

The final avenue of future work that can be explored is to optimize the Rand score thin metric directly. For the FCNs used in this thesis, optimizing the Rand score thin metric directly is not possible. This does not mean, however, that there does not exist an approach to optimize the Rand score thin metric. This can be achieved by using maximin affinity learning (Briggman *et al.*, 2009), a particular approach that is used to minimize the Rand index directly using conventional CNNs. This approach can be extended to FCNs by making the FCNs generate a weighted affinity graph instead of regular pixel labels. Creating an affinity FCN was briefly touched upon during the investigations performed in this thesis, but more time would be required to fully unlock the potential of such an approach.

8.3 Summary

This thesis investigated FCNs for the purpose of semantic segmentation on two bio-image data sets. We showed that FCNs are capable of outperforming both the current ad-hoc image processing pipeline and a conventional CNN on the task of segmenting and classifying *C. elegans* worms (Wiehman and De Villiers, 2016). This result provides further evidence that FCNs are better than CNNs at performing semantic segmentation.

We showed that increasing the number of feature map resolutions in the FCN architecture by adding additional pooling steps leads to improved performance, assuming the inclusion of the necessary skip connections. Furthermore, we found that the performance of the models started to saturate as the depth of the network increased, and conjectured that the saturation point could be dependent on the application domain. We showed that each skip connection appears to have a specific contribution towards the functioning of the network, and that some skip connections might be more important than others. These results provide some insight into the functionality of the different parts of the FCN architecture, which could ease the design process of new FCNs for different application domains.

We proposed a novel augmentation to FCN architectures that allow them to undergo end-to-end unsupervised pre-training, and showed a significant reduction in the variance of the performance of trained models (Wiehman *et al.*, 2016). Lastly, we showed that FCNs are capable of reaching competitive performance with as little as 0.2% of the original pixel labels. These results provide a foundation for further research into a more sophisticated end-to-end unsupervised learning approach. The results also suggest that partially labeled data can be nearly as effective as fully labeled data in training FCNs, which could save some of the time and expenses of annotating training data in other domains.

List of References

Arganda-Carreras, I. (2016). Segmentation evaluation after border thinning - script. Available at: `http://imagej.net/Segmentation_evaluation_after_border_thinning_-_Script`

Arganda-Carreras, I., Turaga, S.C., Berger, D.R., Cireşan, D., Giusti, A., Gambardella, L.M., Schmidhuber, J., Laptev, D., Dwivedi, S., Buhmann, J.M., Liu, T., Seyedhosseini, M., Tasdizen, T., Kamentsky, L., Burget, R., Uher, V., Tan, X., Sun, C., Pham, T.D., Bas, E., Uzunbas, M.G., Cardona, A., Schindelin, J. and Seung, H.S. (2015). Crowdsourcing the creation of image segmentation algorithms for connectomics. *Frontiers in Neuroanatomy*, vol. 9, p. 142.
Available at: `http://journal.frontiersin.org/article/10.3389/fnana.2015.00142`

Banko, M. and Brill, E. (2001). Scaling to very very large corpora for natural language disambiguation. In: *Proceedings of the 39th Annual Meeting on Association for Computational Linguistics*, pp. 26–33. Association for Computational Linguistics.

Bastien, F., Lamblin, P., Pascanu, R., Bergstra, J., Goodfellow, I.J., Bergeron, A., Bouchard, N. and Bengio, Y. (2012). Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop.

Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., Turian, J., Warde-Farley, D. and Bengio, Y. (2010). Theano: a CPU and GPU math expression compiler. In: *Proceedings of the Python for Scientific Computing Conference (SciPy)*. Oral Presentation.

Briggman, K., Denk, W., Seung, S., Helmstaedter, M.N. and Turaga, S.C. (2009). Maximin affinity learning of image segmentation. In: *Advances in Neural Information Processing Systems*, pp. 1865–1873.

Brown, M.B. and Forsythe, A.B. (1974). Robust tests for the equality of variances. *Journal of the American Statistical Association*, vol. 69, no. 346, pp. 364–367.

Cardona, A., Saalfeld, S., Preibisch, S., Schmid, B., Cheng, A., Pulokas, J., Tomancak, P. and Hartenstein, V. (2010). An integrated micro- and macroarchitectural analysis of the *Drosophila* brain by computer-assisted serial section electron microscopy. *PLOS Biology*, vol. 8, no. 10, pp. 1–17.
Available at: `http://dx.doi.org/10.1371%2Fjournal.pbio.1000502`

Cardona, A., Saalfeld, S., Schindelin, J., Arganda-Carreras, I., Preibisch, S., Longair, M., Tomancak, P., Hartenstein, V. and Douglas, R.J. (2012). TrakEM2 software for neural circuit reconstruction. *PLOS ONE*, vol. 7, no. 6, pp. 1–8. Available at: `http://dx.doi.org/10.1371%2Fjournal.pone.0038011`

Chen, S. and Billings, S.A. (1992). Neural networks for nonlinear dynamic system modelling and identification. *International Journal of Control*, vol. 56, no. 2, pp. 319–346. Available at: `http://dx.doi.org/10.1080/00207179208934317`

Ciresan, D., Giusti, A., Gambardella, L.M. and Schmidhuber, J. (2012). Deep neural networks segment neuronal membranes in Electron Microscopy images. In: *2012 Advances in Neural Information Processing Systems*, pp. 2843–2851. Available at: `http://papers.nips.cc/paper/4741-deep-neural-networks-segment-neuronal-membranes-in-electron-microscopy-images.pdf`

Collet, S. (2017). Object detection part 1. `https://www.saagie.com/blog/object-detection-part1`. Accessed: 15/09/2017.

Delong, A., Gorelick, L., Schmidt, F., Veksler, O. and Boykov, Y. (2011). Interactive segmentation with super-labels. In: *Energy Minimization Methods in Computer Vision and Pattern Recognition*, pp. 147–162. Springer.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K. and Fei-Fei, L. (2009). ImageNet: A large-scale hierarchical image database. In: *Proceedings of the 2009 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 248–255.

Diebold, J., Demmel, N., Hazırbaş, C., Moeller, M. and Cremers, D. (2015). Interactive multi-label segmentation of RGB-D images. In: *2015 International Conference on Scale Space and Variational Methods in Computer Vision*, pp. 294–306. Springer.

Duchi, J., Hazan, E. and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, vol. 12, pp. 2121–2159.

Erhan, D., Bengio, Y., Courville, A., Manzagol, P.-A., Vincent, P. and Bengio, S. (2010). Why does unsupervised pre-training help deep learning? *Journal of Machine Learning Research*, vol. 11, pp. 625–660.

Everingham, M., Eslami, S.M.A., Van Gool, L., Williams, C.K.I., Winn, J. and Zisserman, A. (2015). The PASCAL Visual Object Classes challenge: A retrospective. *International Journal of Computer Vision*, vol. 111, no. 1, pp. 98–136.

Everingham, M., Van Gool, L., Williams, C.K.I., Winn, J. and Zisserman, A. (2011). The PASCAL Visual Object Classes challenge 2011 (VOC2011) results. Available at: `http://www.pascal-network.org/challenges/VOC/voc2011/workshop/index.html`

Fawcett, T. (2006). An introduction to ROC analysis. *Pattern Recognition Letters*, vol. 27, no. 8, pp. 861–874.

Figueiredo, M.A. (2003). Adaptive sparseness for supervised learning. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 25, no. 9, pp. 1150–1159.

Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In: *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics*, vol. 9, pp. 249–256.

Hawkins, D.M. (2004). The problem of overfitting. *Journal of Chemical Information and Computer Sciences*, vol. 44, no. 1, pp. 1–12.
Available at: `http://dx.doi.org/10.1021/ci0342472`

He, K., Zhang, X., Ren, S. and Sun, J. (2015 December). Delving deep into rectifiers: surpassing human-level performance on ImageNet classification. In: *2015 IEEE International Conference on Computer Vision (ICCV)*.

Hinton, G., Srivastava, N. and Swersky, K. (2012*a*). Lecture 6: Overview of mini-batch gradient descent.
Available at: `https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`

Hinton, G.E., Srivastava, N., Krizhevsky, A., Sutskever, I. and Salakhutdinov, R.R. (2012*b*). Improving neural networks by preventing co-adaptation of feature detectors.
Available at: `https://arxiv.org/pdf/1207.0580.pdf`

Hochreiter, S., Bengio, Y., Frasconi, P. and Schmidhuber, J. (2001). Gradient flow in recurrent nets: the difficulty of learning long-term dependencies. In: *A Field Guide to Dynamical Recurrent Neural Networks*. IEEE Press.

Hong, S., Noh, H. and Han, B. (2015). Decoupled deep neural network for semi-supervised semantic segmentation. In: *2015 Advances in Neural Information Processing Systems*, pp. 1495–1503.

Huang, Z., Li, J., Weng, C. and Lee, C.-H. (2014). Beyond cross-entropy: Towards better frame-level objective functions for deep neural network training in automatic speech recognition. In: *Proceedings of the 15th Annual Conference of the International Speech Communication Association*.
Available at: `https://www.microsoft.com/en-us/research/publication/beyond-cross-entropy-towards-better-frame-level-objective-functions-for-deep-neural-network-training-in-automatic-speech-recognition/`

Intrator, O. and Intrator, N. (2001). Interpreting neural-network results: a simulation study. *Computational Statistics & Data Analysis*, vol. 37, no. 3, pp. 373–393.

Japkowicz, N. and Stephen, S. (2002). The class imbalance problem: A systematic study. *Intelligent Data Analysis*, vol. 6, no. 5, pp. 429–449.

Kaletta, T. and Hengartner, M.O. (2006). Finding function in novel targets: *C. elegans* as a model organism. *Nature Reviews Drug Discovery*, vol. 5, no. 5, pp. 387–399.

Kalogirou, S.A. (2001). Artificial neural networks in renewable energy systems applications: a review. *Renewable and Sustainable Energy Reviews*, vol. 5, no. 4, pp. 373 – 401.
Available at: `http://www.sciencedirect.com/science/article/pii/S1364032101000065`

Karpathy, A. (2017*a*). CS231n convolutional neural networks for visual recognition course notes. `http://cs231n.github.io/neural-networks-1/`. Accessed: 02/08/2017.

Karpathy, A. (2017*b*). CS231n convolutional neural networks for visual recognition course notes. `http://cs231n.github.io/convolutional-networks/`. Accessed: 02/08/2017.

Khan, A.M., Raza, S.-E.-A., Khan, M. and Rajpoot, N.M. (2014). Cell phenotyping in multi-tag fluorescent bioimages. *Neurocomputing*, vol. 134, no. 1, pp. 254 – 261.
Available at: `http://www.sciencedirect.com/science/article/pii/S0925231214000988`

Kingma, D.P., Mohamed, S., Rezende, D.J. and Welling, M. (2014). Semi-supervised learning with deep generative models. In: *2014 Advances in Neural Information Processing Systems*, pp. 3581–3589.
Available at: `http://papers.nips.cc/paper/5352-semi-supervised-learning-with-deep-generative-models.pdf`

Kohavi, R. (1995). A study of cross-validation and bootstrap for accuracy estimation and model selection. In: *1995 International Joint Conference on Artificial Intelligence (IJCAI)*, vol. 14, pp. 1137–1145. Stanford, CA.

Kraus, O.Z., Ba, J.L. and Frey, B.J. (2016). Classifying and segmenting microscopy images with deep multiple instance learning. *Bioinformatics*, vol. 32, no. 12, pp. i52–i59.
Available at: `http://dx.doi.org/10.1093/bioinformatics/btw252`

Krizhevsky, A., Sutskever, I. and Hinton, G.E. (2012). ImageNet classification with deep convolutional neural networks. In: *2012 Advances in Neural Information Processing Systems*, pp. 1097–1105.

LeCun, Y.A., Bottou, L., Orr, G.B. and Müller, K.-R. (1998). Efficient backprop. In: *Neural networks: Tricks of the Trade*, vol. 1524, pp. 9–50. Springer.

Leung, F.H.F., Lam, H.K., Ling, S.H. and Tam, P.K.S. (2003). Tuning of the structure and parameters of a neural network using an improved genetic algorithm. *IEEE Transactions on Neural Networks*, vol. 14, no. 1, pp. 79–88.

Levene, H. (1960). Robust tests for equality of variances. *Contributions to probability and statistics: Essays in honor of Harold Hotelling*, vol. 2, pp. 278–292.

Lin, D., Dai, J., Jia, J., He, K. and Sun, J. (2016). Scribblesup: Scribble-supervised convolutional networks for semantic segmentation. In: *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 3159–3167.

Liu, C., Yuen, J. and Torralba, A. (2009). Nonparametric scene parsing: Label transfer via dense scene alignment. In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1972–1979. IEEE.

Ljosa, V., Sokolnicki, K.L. and Carpenter, A.E. (2012). Annotated high-throughput microscopy image sets for validation. *Nature Methods*, vol. 9, no. 7, p. 637.

Long, J., Shelhamer, E. and Darrell, T. (2015). Fully convolutional networks for semantic segmentation. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 3431–3440.

Mann, H.B. and Whitney, D.R. (1947). On a test of whether one of two random variables is stochastically larger than the other. *The Annals of Mathematical Statistics*, vol. 18, no. 1, pp. 50–60.
Available at: `http://dx.doi.org/10.1214/aoms/1177730491`

Masci, J., Meier, U., Cireşan, D. and Schmidhuber, J. (2011). Stacked convolutional auto-encoders for hierarchical feature extraction. In: *2011 International Conference on Artificial Neural Networks*, pp. 52–59. Springer.

Meissner, M., Schmuker, M. and Schneider, G. (2006). Optimized particle swarm optimization (OPSO) and its application to artificial neural network training. *BMC Bioinformatics*, vol. 7, no. 1, p. 125.
Available at: `https://doi.org/10.1186/1471-2105-7-125`

Moy, T.I., Conery, A.L., Larkins-Ford, J., Wu, G., Mazitschek, R., Casadei, G., Lewis, K., Carpenter, A.E. and Ausubel, F.M. (2009). High-throughput screen for novel antimicrobials using a whole animal infection model. *ACS Chemical Biology*, vol. 4, no. 7, pp. 527–533.
Available at: `http://dx.doi.org/10.1021/cb900084v`

Ng, A.Y. (2004). Feature selection, L1 vs. L2 regularization, and rotational invariance. In: *Proceedings of the 21st International Conference on Machine learning*, p. 78. ACM.

Nielsen, M.A. (2015). *Neural Networks and Deep Learning*. Determination Press.
Available at: `http://neuralnetworksanddeeplearning.com/`

Peng, H. (2008). Bioimage informatics: a new area of engineering biology. *Bioinformatics*, vol. 24, no. 17, pp. 1827–1836.
Available at: `http://bioinformatics.oxfordjournals.org/content/24/17/1827`

Pinheiro, P.H.O. and Collobert, R. (2014). Recurrent convolutional neural networks for scene parsing. In: *Proceedings of the 31st International Conference on Machine Learning*, pp. 82–90.
Available at: `http://jmlr.org/proceedings/papers/v32/pinheiro14.pdf`

Powers, D.M. (2011). Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation. *Journal of Machine Learning Technologies*, vol. 2, no. 1, pp. 37–63.

Prechelt, L. (2012). Early stopping - but when? In: *Neural Networks: Tricks of the Trade*, vol. 7700, 2nd edn, pp. 53–67. Springer.

Rasmus, A., Berglund, M., Honkala, M., Valpola, H. and Raiko, T. (2015). Semi-supervised learning with Ladder networks. In: *2015 Advances in Neural Information Processing Systems*.

Riedmiller, M. and Braun, H. (1993). A direct adaptive method for faster backpropagation learning: the RPROP algorithm. In: *1993 IEEE International Conference on Neural Networks*, vol. 1, pp. 586–591.

Ronneberger, O., Fischer, P. and Brox, T. (2015). U-net: convolutional networks for biomedical image segmentation. In: *Proceedings of the 2015 International Conference on Medical Image Computing and Computer-Assisted Intervention*, pp. 234–241.

Rumelhart, D.E., Hinton, G.E. and Williams, R.J. (1985). Learning internal representations by error propagation. Tech. Rep., DTIC Document.

Rumelhart, D.E., Hinton, G.E. and Williams, R.J. (1986). Learning representations by back-propagating errors. *Nature*, vol. 323, pp. 533–536.

Shaar, N. (2012). ISBI 2012 challenge. Accessed: 30/09/2017.
Available at: `http://brainiac2.mit.edu/isbi_challenge/leaders-board-new`

Shafer, D. and Zhang, Z. (2012). *Introductory Statistics*. Saylor Academy.
Available at: `https://open.umn.edu/opentextbooks/BookDetail.aspx?bookId=135`

Silberman, N., Hoiem, D., Kohli, P. and Fergus, R. (2012). Indoor segmentation and support inference from RGB-D images. In: *2012 European Conference on Computer Vision*, pp. 746–760. Springer.

Snedecor, G.W. and Cochran, W.G. (1989). *Statistical methods*. 8th edn. Iowa State University Press.

Socher, R., Lin, C.C., Manning, C. and Ng, A.Y. (2011). Parsing natural scenes and natural language with recursive neural networks. In: *Proceedings of the 28th International Conference on Machine Learning (ICML)*, pp. 129–136.

Soille, P. and Vincent, L.M. (1990). Determining watersheds in digital pictures via flooding simulations. vol. 1360.
Available at: `http://dx.doi.org/10.1117/12.24211`

Srivastava, N., Hinton, G.E., Krizhevsky, A., Sutskever, I. and Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958.

Student (1908). The probable error of a mean. *Biometrika*, vol. 6, no. 1, pp. 1–25.
Available at: `http://www.jstor.org/stable/2331554`

Szeliski, R. (2010). *Computer Vision: Algorithms and Applications*.
Available at: `http://szeliski.org/Book/`

Trask, A., Gilmore, D. and Russell, M. (2015). Modeling order in neural word embeddings at scale. In: *Proceedings of the 32nd International Conference on Machine Learning*, vol. 37.

Valpola, H. (2015). From neural PCA to deep unsupervised learning. In: *Advances in Independent Component Analysis and Learning Machines*, pp. 143–171. Academic Press.

Vincent, P., Larochelle, H., Lajoie, I., Bengio, Y. and Manzagol, P.-A. (2010). Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *Journal of Machine Learning Research*, vol. 11, pp. 3371–3408.

Wählby, C., Kamentsky, L., Liu, Z.H., Riklin-Raviv, T., Conery, A.L., O'Rourke, E.J., Sokolnicki, K.L., Visvikis, O., Ljosa, V., Irazoqui, J.E., Golland, P., Ruvkun, G., Ausubel, F.M. and Carpenter, A.E. (2012). An image analysis toolbox for high-throughput *C. elegans* assays. *Nature Methods*, vol. 9, no. 7, pp. 714–716.

Wan, L., Zeiler, M., Zhang, S., Cun, Y.L. and Fergus, R. (2013). Regularization of neural networks using dropconnect. In: *Proceedings of the 30th International Conference on Machine Learning*, pp. 1058–1066.

Welch, B.L. (1947). The generalization of 'Student's' problem when several different population variances are involved. *Biometrika*, vol. 34, no. 1, pp. 28–35.
Available at: `http://www.jstor.org/stable/2332510`

Wiehman, S. and De Villiers, H. (2016). Semantic segmentation of bioimages using convolutional neural networks. In: *2016 International Joint Conference on Neural Networks (IJCNN)*, pp. 624–631. IEEE.

Wiehman, S., Kroon, S. and De Villiers, H. (2016). Unsupervised pre-training for fully convolutional neural networks. In: *2016 Pattern Recognition Association of South Africa and Robotics and Mechatronics International Conference (PRASA-RobMech)*, pp. 1–6. IEEE.

Xu, B., Wang, N., Chen, T. and Li, M. (2015). Empirical evaluation of rectified activations in convolutional network. *CoRR*, vol. abs/1505.00853.
Available at: `http://arxiv.org/abs/1505.00853`

Zeiler, M.D. (2012). ADADELTA: An Adaptive Learning Rate Method. *CoRR*, vol. abs/1212.5701.
Available at: `http://arxiv.org/abs/1212.5701`

Zhang, G., Patuwo, B.E. and Hu, M.Y. (1998). Forecasting with artificial neural networks: The state of the art. *International Journal of Forecasting*, vol. 14, no. 1, pp. 35–62.