Norwegian University
of Life Sciences

Master's Thesis 2019 60 ECTS

Faculty of Science and Technology
Professor Cecilia Marie Futsæther

Deep learning for automatic delineation of tumours from PET/CT images

Yngve Mardal Moe
MSc Mathematical Physics and Computational Sciences

This page is intentionally left blank.

Acknowledgements

Foremost, I express my profound gratitude to my advisor, Prof. Cecilia Marie Futsæther, for the continuous and enthusiastic feedback during this project.

Moreover, I thank the scientists who participated in meetings regarding my project and came with valuable feedback: Ms Aurora Grøndahl, Dr Turid Torheim, Prof. Oliver Tomic, Prof. Ulf Geir Indahl, Prof. Kristian Liland and Prof. Eirik Malinen.

Further, I thank oncologist Dr Einar Dale and Prof. Eirik Malinen for access to the dataset, without which this research would not be possible. In addition, I thank Eik Idéverksted and Disruptive Engineering AS for allowing me to use their hardware, thus enabling much of the research presented in this thesis.

Next, I express thankfulness to all my friends who helped proofread this thesis and in particular Inês Neves for her thoughtful feedback.

I thank my girlfriend, Marie Roald, who supported me through long days of writing and assisted me to convert my hand-drawn scribbles into vector graphics.

Finally, I thank my parents for supporting me through this thesis and life in general.

Yngve Mardal Moe
Ås, February 28th 2019

Abstract

Purpose
The delineation of tumours and malignant lymph nodes in medical images is an essential part of radiotherapy. However, it is both time-consuming and prone to inter-observer variability. Automating this process is therefore beneficial as it will reduce the time effort of radiotherapy planning and the inter-observer variability. One method of automating delineation is by using neural networks. Deep learning experiments, however, requre tuning of a vast amount of parameters. Thus, a systematic methodology for conducting such experiments is vital to ensure reproducibility. This thesis will introduce the theory of deep learning and present the SciNets library, a framework for rapid model prototyping with guaranteed reproducibility. This framework was used to develop a model for automatic delineation of gross tumour volume and malignant lymph nodes in the head and neck region.

Methods
The SciNets library (available at https://github.com/yngvem/scinets/)is a Python library that creates and trains deep learning models parametrised by a series of JSON files containing model hyperparameters. Furthermore, an extensive visualisation suite is included to inspect the training process. This library was used to assess the applicability of neural networks for automatic tumour delineation. The dataset consisted of medical scans taken of 197 patients who recieved treatment at the Oslo University Hospital, The Radium Hospital. 18F-FDG-PET co-registered to contrast-enhanced CT scans (i.e. contrast-enhanced PET/CT scans) were available for all patients. The image dataset was split into a training set (142 patients), a validation set (15 patients) and a test set (40 patients), stratified by tumour stage. A vast parameter sweep was performed on this dataset.

All tested models were based on the U-Net architecture. Both the Cross Entropy and dice loss were tested, as well as the novel F_2 and F_4 loss introduced herein. Channel dropping and Hounsfield windowing were used for preprocessing, with varying window centres and widths. Both Adam and SGDR+momentum

were tested to optimise the loss. Furthermore, Improved ResNet layer types were tested against standard convolutional layers. Models were compared based on the average dice per image slice in the validation set. Only the highest performing models utilising only CT information, only PET information and both PET and CT information were used to delineate the test set. The sensitivity (sens), specificity (spec), positive predictive value (PPV) and dice score were computed for these models. Additional analysis was performed on the highest performing model utilising only CT information and the highest performing model utilising both PET and CT information. Ground truth and predicted delineations were visualised for a subset of the patients in the validation and test set for these models.

Results
The parameter sweep consisted of over 150 different parameter combinations and showed that using the newly introduced F_2 and F_4 loss provided a notable increase in performance compared to the Cross Entropy and dice loss. Furthermore, Hounsfield windowing yielded a systematic increase in performance; however, the choice of window centre and width did not yield any noticeable difference. There was no difference between the Adam optimiser and SGDR+momentum optimiser on either performance or training time. However, using a too low learning rate with the Adam optimiser resulted in poor performance on out of sample data (i.e. validation set). Models utilising ResNet layers experienced exploding gradients on the skip connections and did not converge. The highest performing PET/CT model (Dice: 0.66, Sens: 0.79, Spec: 0.99, PPV: 0.62) achieved higher overall performance compared to PET-only models (Dice: 0.64, Sens: 0.69, Spec: 0.99, PPV: 0.64) or CT-only models (Dice: 0.56, Sens: 0.58, Spec: 0.99, PPV: 0.62).

Conclusions
We have demonstrated that deep learning is a promising avenue for automatic delineation of regions of interest in medical images. The SciNets library was used to conduct a systematic and reproducible parameter sweep for automatic delineation of tumours and malignant lymph nodes in patients with head and neck cancer. This parameter sweep yielded a recommended set of hyperparameters for similar experiments as well as recommendations for further exploration.

The dice performance of both the PET/CT and CT-only model is similar to that expected between two radiologists. We can, however, not conclude that the automatically generated segmentation maps are of similar quality as to those generated by radiologists. The dice coefficient does not discern the severity of mistakes, only the percentage of overlap between the predicted delineation maps and the ground truth. Oncologists should, therefore, be consulted when assessing the quality of delineation masks in future experiments.

Contents

Appendices

List of Figures

All figures in this thesis are joint work by Yngve Mardal Moe and Marie Roald and are licensed under a Creative Commons Attributions only licence (CC BY 4.0).

List of Tables

List of Algorithms

Abbreviations

Abbreviation	Meaning
API	Application Programming Interface
BN	Batch Normalisation
Conv	Convolution
CNN	Convolutional Neural Network
CT	(X-Ray) Computerised Tomography
HDF(5)	Hierarchical Data Format (5)
HU	Hounsfield Unit
IO	Input/Output
JSON	JavaScript Object Notation (a standard data serialisation format)
PET	Positron Emmision Tomography
PPV	Positive Predive Value (i.e. precision)
ReLU	Rectified Linear Unit
ResNet	Residual Neural Network
RGB	Red, Green, Blue
SGD	Stochastic Gradient Descent
SGDR	Stochastic Gradient Descent with Warm Restarts
STD	Standard Deviation
SUV	Standardised Uptake Value

Mathematical notation

Mathematical symbol	Meaning
$\breve{f}_i$	The i-th layer of a neural network
$f(\boldsymbol{x};\mathcal{W}) = \breve{f}_n(\breve{f}_{n-1}(...\breve{f}_1(\boldsymbol{x};\mathcal{W}_1)...;\mathcal{W}_{n-1});\mathcal{W}_n)$	A neural network.
$f_i(\boldsymbol{x}) = \breve{f}_i(\breve{f}_{i-1}(...(\breve{f}_1(\boldsymbol{x};\mathcal{W}_1)...;\mathcal{W}_{i-1});\mathcal{W}_i)$	The output of the i-th layer of a neural network.
$\mathcal{W}_i$	The parameters of $\breve{f}_i$
$\mathcal{W} = \bigcup_i \mathcal{W}_i$	All parameters of a neural network
$\mathcal{X}_i$	A high dimensional sets
$\mathcal{X} = \{\boldsymbol{x}_i\}_i$	Input data
$\mathcal{Y} = \{\boldsymbol{y}_i\}_i$	Ideal output data
$\mathcal{T} = \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_i$	Training set
$\mathcal{B}_i \subset \mathcal{T}$	A batch, used for stochastic optimisation
$D^{(n)}[\boldsymbol{x}]$	A downsampling operation taking every n-th element of $\boldsymbol{x}$
$C_k[\boldsymbol{x}]$	The convolution of $\boldsymbol{x}$ with a kernel k
$\phi(x)$	An activation function
$\phi_{ReLU}(x) = max(0, x)$	The ReLU activation function
$\phi_{sigmoid}(x) = \frac{1}{1+exp(-x)}$	The sigmoidal activation function
$\phi_{softmax}(\boldsymbol{x}) = \left[\frac{exp(x_i)}{\sum_j exp(x_j)}\right]_i$	The softmax activation function
$BN(\boldsymbol{x})$	Batch normalisation of $\boldsymbol{x}$.
$J(f;\mathcal{T})$	A loss function parametrised by the training set $\mathcal{T}$.

Chapter 1

Introduction

1.1 Motivation

Over nine million people died of cancer in 2018 [1]. It is, therefore, integral to find effective and efficient treatments. Currently, more than half of the world's population lacks access to essential health care [2] and more than 85 countries reported having less than one doctor per 1000 inhabitants [3]. Meanwhile, the high costs associated to the currently available cancer treatment options make it inaccessible to a large proportion of those affected by it. Thus, reducing the time and cost of cancer treatment is essential to ensure that those suffering from cancer get treatment.

One conventional treatment for patients with cancer is radiotherapy [4]. In radiotherapy, the doctors use a linear accelerator that irradiates the cancer cells with X-Rays, with the aim of killing them in the process. Unfortunately, this irradiation also kills healthy tissue. Therefore, health professionals spend a large amount of time planning exactly where and how to irradiate the patients, in order to minimise the radiation dose given to healthy tissue and maximising the dose given to the tumour.

This precisely targeted irradiation requires a good understanding of the location of the tumour. Radiologists use medical imaging techniques, such as X-Ray computerised tomography (CT), positron emission tomography (PET) and magnetic resonance imaging (MRI), to view the structural information (e.g. tissue density) or the functional processes (e.g. glucose consumption) of the body [5]. This in-

formation is then used to discover where the tumour is, discern which stage it has reached, and plan how to irradiate the patient with X-Rays.

The tumour delineation process is, unfortunately, both time-consuming and subjective [6]. Finding methods to automate this process would therefore be highly beneficial, as it would reduce the time, cost and subjectivity of radiotherapy. However, tumour delineation is not a simple process as there is not necessarily a sharp boundary between healthy and cancerous tissue. Automating this process is therefore a challenge.

Tumour delineation can be viewed as a computer vision problem. Specifically, we can view it as an image segmentation problem. Using this perspective is beneficial, as the field of computer vision has moved forward rapidly since the popularisation of convolutional neural networks[1] in 2012/2013.

1.1.1 A brief introduction to deep learning

The invent of convolutional neural networks is often attributed to LeCun's seminal paper in 1989 [7]. There are several reasons why convolutional neural networks were not popularised until 2013. Amongst them were, undoubtedly, the computational power and the complicated pipeline necessary to train them. However, when Alex Krizhevsky, as the only competitor using convolutional neural networks, won the ImageNet competition in 2012 with a landslide [8], it became impossible to ignore the value of deep learning in its field, despite its cumbersome training pipeline. It is for this reason that Alex Krizhevsky is often credited for starting the "golden age" of deep learning.

During the last decade, deep learning has flourished, shown in Figure 1.1 by the rapid increase in publications after 2010. In 2010, Glorot and Bengio [9] showed that carefully initialised random weights could yield similar performance to that of networks pretrained in an unsupervised fashion. Glorot *et al.* [10] introduced the ReLU nonlinearity in 2011, thus removing the problem of vanishing gradients. In 2015, Ioffe and Szegedy [11] introduced batch normalisation, combatting the problem of exploding gradients. It has, in other words, become significantly easier to train these networks over the past decade. The main problems now is system resources and large datasets.

[1] Neural networks are a machine learning algorithm initially developed based on ideas from computational neuroscience. Much of its nomenclature is therefore derived from neuroscience, even though the neural networks do not resemble the current understanding of neuroscience.

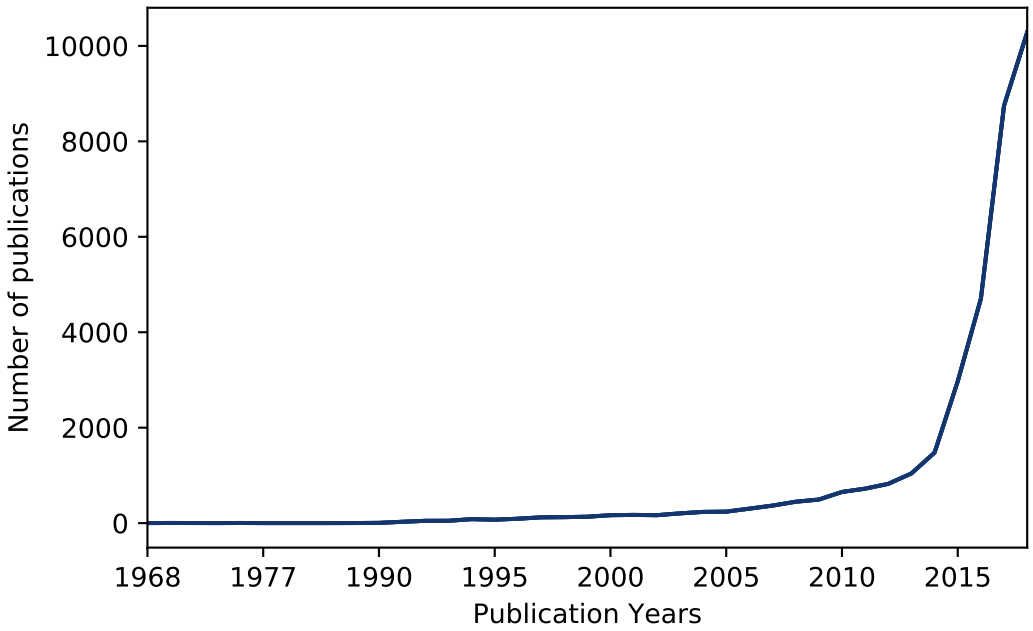


Figure 1.1: The number of articles matching the keyword "deep learning" on Web of Science plotted against publication year.

However, it is not only the theory of deep learning that has shown progress over the last decade – the engineering side has also progressed immensely [12]–[15]. This is essential as much compute power is required to train neural networks. Several tools have therefore been developed to utilise General Purpose Graphics Processing Unit (GPGPU) programming in this field, hence allowing neural networks to be trained on graphics cards. Using GPGPU programming is advantageous as graphics processing units (GPUs) are much more efficient at floating point arithmetic than central processing units (CPUs) [16].

Still, programming for GPUs is not a trivial task, as GPUs use a particular parallel processing paradigm called Single Instruction Multiple Threads (SIMT) [16]. Writing efficient programs for a SIMT processor requires, amongst other, extreme care for memory layout. Testing deep learning hypotheses on a GPU might, therefore, seem impossible. Luckily, companies such as Google [12], Facebook [13] and Microsoft [14] have spent countless resources to make high-level frameworks that enable deep learning algorithms to utilise GPUs.

However, the aforementioned frameworks have several weaknesses. Although they make GPU programming for deep learning much more straightforward, one would be stretched to say that they make it simple. Therefore, Chollet *et al.* [15] created Keras, a high-level deep learning framework that makes it easy to prototype using high-level components.

Unfortunately, Keras also suffers from the second problem that frameworks like TensorFlow[2], PyTorch[3] and CNTK[4] suffer from, namely that there are many ways to perform the same experiments, making automatic logging difficult. This weakness means that performing a vast parameter sweep requires us to put much thought and effort into how experiments are logged.

1.1.2 Automatic delineation of head and neck cancers

Head and neck cancer (HNC) is an umbrella term for cancers originating in the mouth, nose, throat, sinuses, larynx (voice box), or salivary glands [17]. However, patients with this type of tumour often have malignant lymph nodes as well. Hence, a radiologist must delineate several malignant regions of interest per patient. Also, the radiologist must delineate specific organs at risk that should recieve a radiation

[2]Made by Google.
[3]Made by Facebook.
[4]Made by Microsoft.

dose as small as possible.

Several imaging modalities are used when delineating tumours in HNC patients. Any combination of PET, MRI and CT can be used to detect the tumour. Furthermore, contrast agents can be used to make the tumour more visible in CT and MRI images [18].

Delineating HNC tumours is both time-consuming and subjective [6], [19]. As such, much effort has been made to automate this process. However, this area of research mainly focuses on automatic segmentation from PET images [20]. Furthermore, these studies often require the radiologist to manually find a small region within which the tumour or lymph nodes are contained [19], [20].

There are several downsides to this approach. The main downsides come from the imaging modality used. PET imaging is costly, as it requires the hospital to produce radioactive molecules [18]. Moreover, PET images contain little high-frequency information, making accurate estimates of the tumour border difficult. It is therefore advantageous to combine the PET images with high-frequency information from either CT images or MRI images [18].

In 'Automatic delineation of tumor volumes by co-segmentation of combined PET/MR data', Leibfarth *et al.* [21] introduced an algorithm to solve the problem of high-frequency information not being present in PET images by including MRI information (PET/MR). Their algorithm requires the radiologist to draw a square region of interest around the tumour before their algorithm automatically delineates it. Furthermore, in 'Globally Optimal Tumor Segmentation in PET-CT Images: A Graph-Based Co-segmentation Method', Han *et al.* [22] introduced a semiautomatic segmentation algorithm for co-registered PET/CT images. The algorithm requires a radiologist to mark where small parts of malignant tissue and healthy tissue were in each image slice. Using these "seeds", the algorithm will automatically label every pixel as either healthy or not healthy.

The algorithms above have two shortcomings. Firstly, they require the radiologist to find approximately where the tumours are in most image slices (interpolation can be used to reduce the delineation time). Secondly, they introduce interobserver variability (i.e. different radiologists might end up with different segmentation masks). To combat these problems, we will develop an algorithm that takes full body images as input and returns the segmentation masks as output. Thus, the interaction between the software and the radiologist is minimised, reducing both the time spent to delineate tumours and the interobserver variability.

The algorithms developed in this project are based on convolutional neural net-

works, as these have shown great success in the computer vision literature [23]–[28]. However, neither the use of deep learning for HNC tumour delineation nor fully automatic segmentation of HNC tumours using PET/CT are, to the author's knowledge, well-tested approaches. In 2009 Yu *et al.* [29] developed a decision tree based algorithm that uses local texture features to delineate HNC tumours automatically. However, this study is limited to ten patients, and their model testing/validation approach is not adequately documented.

There are, to the author's knowledge, only two published articles that use deep learning for automatic delineation of malignant tissue for HNC patients [30], [31]. In both studies, a delineation of the clinical tumour volume (CTV) is generated from CT images and already delineated gross tumour volumes (GTV). Both articles achieve outstanding performance, with Dice scores[5] in the range 0.70-0.85. However, neither article include a baseline performance by showing the Dice score between the GTV and CTV. Such a baseline is integral to their performance analysis as the GTV is entirely contained within the CTV. Additionally, the models depend on a radiologist spending the time to delineate the GTV. Finally, we note that [31] uses a simple two-layer stacked autoencoder, whereas [30] use a 3D U-Net architecture to achieve their results.

Work has, however, been done on automatic segmentation of organs at risk for HNC patients. Liang *et al.* [32] achieved exceptional accuracy on segmentation of organs at risk using a two-step deep learning approach. Firstly, a Faster R-CNN architecture [33] was used to propose bounding boxes for each organ. The contents of these boxes were then provided to a Fully Convolutional Network [28] that generated the final segmentation masks. Their algorithm used only CT images and achieved an average Dice of 0.69 to 0.94 for all organs.

Furthermore, deep learning approaches have shown state-of-the-art results in tumour delineation problems for other cancer types, such as brain tumours from MRI images [34] and nodules in lungs [35]. Deep learning is, in other words, a promising approach for segmenting both organs at risk and tumours.

1.2 Problem statement

This project aims to accomplish three separate, but connected, goals. The first goal is to introduce the theory of deep learning for image segmentation to the

[5] A measure of overlap, described on page 56.

reader. The reader should have a background in "standard" machine learning (e.g. having read most of *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition* [36]) and a strong background in linear algebra as well as a familiarity with multivariate calculus.

The second task is to create a framework for rapid prototyping of image segmentation algorithms using deep learning. It is paramount that reproducibility is ensured. A standardised method of performing experiments will, therefore, be developed. This method should be designed, such that automatic logging of experiment parameters and results is possible.

Finally, the developed framework will be tested by performing a vast parameter sweep for automatic segmentation of tumours and malignant lymph. The input to these algorithms will be PET/CT images of head and the neck cancers. This parameter sweep will then reveal the benefit of combining PET with CT images versus using only one of the imaging modalities.

1.3 Nomenclature and notation

1.3.1 Images

We will describe an image with n spatial dimensions as an n-dimensional image. Thus, an image with n spatial dimensions and c channels (e.g. $c = 2$ for PET/CT images) will be described as an n-dimensional image even though it is an $n + 1$ dimensional construct.

A m-by-n image with c colour channels, $\boldsymbol{x}$, will, in this text, be regarded as an element of $\mathbb{R}^{m \times n \times c}$. Similarly, an m-by-n-by-p image with c colour channels, $\boldsymbol{y}$, will be regarded an element of $\mathbb{R}^{m \times n \times p \times c}$. The elements $\boldsymbol{x}_{ij} \in \mathbb{R}^c$ and $\boldsymbol{y}_{ijk} \in \mathbb{R}^c$ are *pixels*[6] (short for picture element) of the images $\boldsymbol{x}$ and $\boldsymbol{y}$, respectively.

We will require certain operations on images, specifically, *downsampling operators* and *convolution operators*. Convolution is described in detail in Section 2.1.6. However, before that, we introduce some notation. Let $I * k$ be the convolution of an image I with the convolution kernel k. The convolution operator $*$ is bilinear,

[6]Some texts use the word *voxel* for elements of 3D images, we will not make that distinction here.

thus, we can define the linear operator

$$C_k[I] = I * k, \tag{1.1}$$

that is, C_k is the convolution operator with kernel k.

Furthermore, we define the downsampling operator $D^{(n)}[I]$ as the operator that takes every n-th pixels in each direction. Thus, for two dimensional images, $D^{(2)} : \mathbb{R}^{m\times n} \to \mathbb{R}^{\lfloor \frac{m}{2} \rfloor \times \lfloor \frac{n}{2} \rfloor}$, where $\lfloor \cdot \rfloor$ is the floor operator.

1.3.2 Linear analysis

A *linear operator* is a mapping $L : \mathcal{X} \to \mathcal{Y}$ such that $L[a\boldsymbol{x}+b\boldsymbol{y}] = aL[\boldsymbol{x}]+bL[\boldsymbol{y}]$ for any $\boldsymbol{x}, \boldsymbol{y} \in \mathcal{X}$ and $a, b \in \mathbb{R}$. Linear operators that map $\mathbb{R}^m$ to $\mathbb{R}^n$ are represented by $m \times n$ matrices. That is, for any linear mapping $L : \mathbb{R}^m \to \mathbb{R}^n$, there exists a matrix A such that $L[\boldsymbol{x}] = A\boldsymbol{x}$. All linear operators used in this text will be mappings between $\mathbb{R}^{m_1,\dots,m_s}$ and $\mathbb{R}^{n_1,\dots,n_s}$.

Furthermore, we must define the *adjoint*, or *dual*, of a linear operator. However, to define the adjoint, we must first define the *dot product*. Let $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^{m_1,\dots,m_s}$, the dot product, $\boldsymbol{x} \cdot \boldsymbol{y}$ is then defined as:

$$\boldsymbol{x} \cdot \boldsymbol{y} = \sum_i \dots \sum_p x_{i\dots p} y_{i\dots p}. \tag{1.2}$$

Using this, we can define the adjoint of a linear operator $L : \mathbb{R}^{m_1,\dots,m_s} \to \mathbb{R}^{n_1,\dots,n_s}$. The adjoint of this operator, $L^* : \mathbb{R}^{n_1,\dots,n_s} \to \mathbb{R}^{m_1,\dots,m_s}$, is the operator with the property that

$$L[\boldsymbol{x}] \cdot \boldsymbol{y} = \boldsymbol{x} \cdot L^*[\boldsymbol{y}], \tag{1.3}$$

for all $\boldsymbol{x} \in \mathbb{R}^{m_1,\dots,m_s}$ and $\boldsymbol{y} \in \mathbb{R}^{n_1,\dots,n_s}$. Notice that the linear operator maps from $\mathbb{R}^{m_1,\dots,m_s}$ to $\mathbb{R}^{n_1,\dots,n_s}$, whereas the adjoint operator maps from $\mathbb{R}^{n_1,\dots,n_s}$ to $\mathbb{R}^{m_1,\dots,m_s}$. The adjoint operator of a linear operator $L[\boldsymbol{x}] = A\boldsymbol{x}$ is given by $L * [\boldsymbol{y}] = A^T\boldsymbol{y}$. Thus, the adjoint operator is the generalisation of a matrix transpose.

Finally, we let $||\cdot||_2 : \mathbb{R}^{m_1,\dots,m_s} \to \mathbb{R}_+$, where $\mathbb{R}_+$ is the nonnegative real numbers, be the *Frobenius norm* of $\mathbb{R}^{m_1,\dots,m_s}$, that is,

$$||\boldsymbol{x}||_2 = \sqrt{\boldsymbol{x} \cdot \boldsymbol{x}} = \sqrt{\sum_i \dots \sum_p x^2_{i\dots p}}. \tag{1.4}$$

Chapter 2

Deep learning

2.1 Introduction to deep neural networks

2.1.1 The main components of deep learning

There are three necessary components of any deep learning system, or any supervised machine learning system for that matter. First, we need a goal; something to learn. This is represented by an unknown function, $f^\star$, which, in image segmentation, maps images to their ideal segmentation masks. Now, machine learning would generally not be necessary if we have direct access to this function. Therefore, we are interested in problems where we have many input-output pairs of the function $f^\star$. Thus, we have access to *training data* $\mathcal{T} = \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^n$, where $\boldsymbol{y}_i = f^\star(\boldsymbol{x}_i)$, and we want to recover the function $f^\star$. For segmentation problems, $\boldsymbol{x}_i$ is an image and $\boldsymbol{y}_i$ is the corresponding segmentation mask.

Note that it would be impossible to find the correct function $f^\star$ if we were to search among all possible functions, for this reason, we restrict the possible functions that we consider. The second component of a supervised machine learning system is, therefore, the class of functions $\mathcal{H}$ through which we search for a good approximator of $f^\star$. In deep learning, the class of functions we consider is often called the *architecture*. The neural network[1] approach is to specify $\mathcal{H}$ as a cascade

[1]Neural networks got their name because their structure is loosely inspired by structures in the mammalian brain [37]. We will, however, not introduce those similarities in this text, as it is not important for understanding how neural network based algorithms work.

of composed functions,

$$\mathcal{H} = \{f : \mathcal{X}_0 \to \mathcal{X}_n | f(\boldsymbol{x}; \mathcal{W}) = \breve{f}_n(\dots(\breve{f}_2(\breve{f}_1(x; \mathcal{W}_1); \mathcal{W}_2))\dots; \mathcal{W}_n), \mathcal{W}_i \in \mathbb{R}^{k_i}\} \tag{2.1}$$

where $\mathcal{X}_0$ is the input space of our function (e.g. the space of all PET/CT images) and $\mathcal{X}_n$ is the output space of our function (e.g. the space of all segmentation masks). The $\breve{f}_i$ functions are pre-specified, $\mathcal{W}_i$ is the collection of the parameters, or *weights*, of $\breve{f}_i$ and $\mathcal{W}$ is the union of all such weights [37]. Thus, the second component of a deep learning system is the set of functions we consider to approximate $f^\star$.

Finally, we need a way of measuring how well a proposed function $\tilde{f}$ approximates the true function $f^\star$. This is not possible, because we (as mentioned earlier) do not have access to $f^\star$. Therefore, we instead measure how well $\tilde{f}$ explains the relationship between the input data and output data. This is done through a loss (or cost) function, often denoted J [37] or L [36], [38]. Loss functions work by measuring how severe the mispredictions of our proposed function $\tilde{f}$ are. As a consequence of this, loss functions have the form

$$J[\tilde{f}; \mathcal{T}] = \sum_i j(\hat{\boldsymbol{y}}_i, \boldsymbol{y}_i), \tag{2.2}$$

where $\hat{\boldsymbol{y}}_i = \tilde{f}(\boldsymbol{x}_i)$ and $\mathcal{T} = \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^n$ is our training data. The interpretation of this equation is that $j(\tilde{f}(\boldsymbol{x}_i), \boldsymbol{y}_i)$ represents the cost we associate with $\tilde{f}(\boldsymbol{x}_i)$ being equal to $\hat{\boldsymbol{y}}_i$ instead of $\boldsymbol{y}_i$. The minimum of j is therefore obtained whenever $\hat{\boldsymbol{y}}_i = \boldsymbol{y}_i$. Thus, the goal of machine learning is to find the function $\tilde{f}$ that minimise the loss function $J[\tilde{f}; \mathcal{T}]$.

2.1.2 The terminology of deep learning.

Recall, that a neural network, f, is a function consisting of a cascade of composed functions;

$$f(\boldsymbol{x}; \mathcal{W}) = \breve{f}_n(\breve{f}_{n-1}(\dots(\breve{f}_1(\boldsymbol{x}; \mathcal{W}_1)); \mathcal{W}_{n-1}); \mathcal{W}_n). \tag{2.3}$$

We name each of the "subfunctions", $\breve{f}_i : \mathcal{X}_{i-1} \to \mathcal{X}_i$, *layers*; $\breve{f}_1$ is the first layer, $\breve{f}_2$ is the second layer and so on. Furthermore, we say that the network above has n layers and that the layer $\breve{f}_i : \mathcal{X}_{i-1} \to \mathcal{X}_i$ has $dim(\mathcal{X}_i)$ *neurons*. Next, we define the output of layer i as the function

$$f_i(\boldsymbol{x}; \cup_{j=1}^i \mathcal{W}_j) = \breve{f}_i(\breve{f}_{i-1}(...(\breve{f}_1(\boldsymbol{x}; \mathcal{W}_1); \mathcal{W}_{i-1}); \mathcal{W}_i). \tag{2.4}$$

This means that the i-th and l-th layer might be the equal, but their outputs might not be equal (because $\breve{f}_k = \breve{f}_l$ but $f_k \neq f_l$).

The layers, $\breve{f}_i$, are generally functions of the form

$$\breve{f}(\boldsymbol{x}; \mathcal{W}_i) = \phi_i(L_i(\boldsymbol{x}; \mathcal{W}_i)), \tag{2.5}$$

where L is a linear mapping from $\mathcal{X}_{i-i}$ to $\mathcal{X}_i$ and ϕ_i is a non-linear function that is applied element-wise on $L(\boldsymbol{x}; \mathcal{W}_i)$. The ϕ_i functions are generally called *activation functions* or *non-linearities*, and choosing the correct nonlinearity for the layers are an integral part to get good network performance [10]. In summary, a layer in a neural network, $\breve{f}(\boldsymbol{x})$, usually consists of a linear mapping $L_i[\boldsymbol{x}]$ and a one dimensional function, $\phi_i(x)$ that is applied to all elements of the output of $L_i[\boldsymbol{x}]$.

Finally, we discuss two different methods of designing neural networks – *wide networks* and *deep networks*. A wide network is a network with few layers and many neurons in each layer. The appealing property of such networks is that they can, if they are wide enough, approximate any function [38]. On the other hand, they are prone to *overfit*, or find a function that works well on the training data, but not new data points [37]. A deep network, on the other hand, is a network with many layers, but fewer neurons per layer than a wide network. This is the type of network most commonly used in computer vision today [8], [25], [39].

There are several beneficial property of deep networks. First and foremost, they are more interpretable. This is because the output of the first layer is generally low-level feature detectors, such as edge, corner and simple texture detectors. These low-level features are then combined to create more advanced feature detectors, such as eye or fur detectors in animal images, which might be the output of some intermediate layer. Then the final layer combines these again to, for example, create a dog detector [40]. The output of a deep network can, as we previously discussed, be approximated by a wide network. It has, however, been showed that the number of neurons needed for a wide network to approximate deep networks grow exponentially with respect to network depth (under reasonable assumptions) [41]. Summarising, we see that deep networks are more easily interpretable than wide networks and require fewer parameters than wide networks to approximate the same functions.

2.1.3 Loss functions

Loss functions are an integral part of deep learning, and the chosen loss function can have severe effects on model quality [42], [43]. We will here introduce two

popular loss functions used in machine learning. Loss functions aimed specifically at segmentation problems will be described in Section 2.3.2

The most well known loss function is probably the squared error loss function which is commonly used in regression problems [36], [38]. The definition of this loss is

$$J[\tilde{f};\mathcal{T}] = \sum_i (\tilde{f}(\boldsymbol{x}_i) - y_i)^2. \tag{2.6}$$

There are several reasons for the popularity of the squared error loss function. Firstly, it is shown to be the optimal loss function if our samples from $f^\star$ are influenced by normally distributed noise with constant variance [38]. Secondly, it has several nice mathematical properties (such as smoothness and convexity) which makes it easier to minimise than other loss functions. To illustrate this, we show an example from linear regression.

Example 2.1.1 (Least squared loss for linear regression)**.**
Let $\mathcal{T} = \{(x_i, y_i)\}_{i=1}^n$ be our training set and let $\mathcal{H}$ be the set of linear functions,

$$\mathcal{H} = \{f : \mathbb{R} \to \mathbb{R} | f(x) = ax + b\}. \tag{2.7}$$

Furthermore, we define the loss function

$$J[\tilde{f};\mathcal{T}] = \frac{1}{2n}\sum_{i=1}^n (\tilde{f}(x_i) - y_i)^2. \tag{2.8}$$

The goal now is to find the function $\tilde{f} \in \mathcal{H}$ that minimises this loss. In other words, we want to solve the equation

$$\arg\min_{\tilde{f}\in\mathcal{H}} J[\tilde{f};\mathcal{T}] = \arg\min_{\tilde{f}\in\mathcal{H}} \frac{1}{2n}\sum_{i=1}^n (\tilde{f}(x_i) - y_i)^2. \tag{2.9}$$

Observe that any function $\tilde{f} \in \mathcal{H}$ is parametrised by two real numbers, a and b. Using this, we can rewrite Equation (2.8),

$$J[\tilde{f};\mathcal{T}] = \frac{1}{2n}\sum_{i=1}^n (ax_i + b - y_i)^2 = J(a, b; \mathcal{T}). \tag{2.10}$$

Now, Equation (2.9) can be rewritten to be on the form

$$\arg\min_{\tilde{f}\in\mathcal{H}} J[f;\mathcal{T}] = \arg\min_{(a,b)\in\mathbb{R}^2} J(a,b;\mathcal{T}) = \arg\min_{(a,b)\in\mathbb{R}^2} \frac{1}{2n}\sum_{i=1}^{n}(ax_i+b-y_i)^2, \tag{2.11}$$

which is a smooth and convex optimisation problem. This problem is therefore equivalent to solving the equations

$$\frac{\partial J}{\partial a} = 0 \tag{2.12}$$

$$\frac{\partial J}{\partial b} = 0. \tag{2.13}$$

We compute the gradients and get

$$\frac{\partial J}{\partial a} = \frac{1}{n}\sum_{i=1}^{n}(ax_i+b-y_i)x_i = 0 \tag{2.14}$$

$$\frac{\partial J}{\partial b} = \frac{1}{n}\sum_{i=1}^{n}(ax_i+b-y_i) = 0, \tag{2.15}$$

which is a set of two linear equations and can therefore easily be solved analytically.

The above example illustrates that the minimum of the squared error loss function is simple to find. The example is, however, somewhat misleading as we only considered linear functions, which was an essential part of rephrasing the optimisation problem from a difficult one (over functions) to a simple one (a two dimensional convex problem). When we deal with deep learning problems we have thousands (if not millions) of parameters, which gives us a system of equations that is intractable to solve. This problem is aggravated by the fact that neural networks are non-convex and as a consequence, there might exist local minima and saddle points [37].

Another popular loss function is the *cross entropy* loss [38], which is often used in classification tasks [8], [39], [44] and segmentation tasks [24], [25]. The cross entropy loss measures the "similarity" of probability distributions [38] (for a thorough introduction to how this similarity is measured, see the book *Information Theory, Inference & Learning Algorithms* by MacKay [45]).

A consequence of the cross entropy measuring the similarity of probability distributions, is that any output vector $\hat{\boldsymbol{y}}_i = \tilde{f}(\boldsymbol{x}_i)$ must either sum to one or be a single

number between zero and one. The former is the case if there are more than two classes, and $\hat{y}_{i,j}$ specifies the probability of $\boldsymbol{x}_i$ being of class j. The latter is the case if there are only two classes, then $\hat{y}_i$ specifies the probability of one of the classes and $1 - \hat{y}_i$ specifies the probability of the other class.

Another requirement when using the cross entropy is that we have a probability distribution to approximate. We set that to be the probability distribution of the classes. This is done by noting that if data point i is of class j, then the probability $y_{i,j} = 1$ and zero otherwise. Using this, we get the following expression for the cross entropy [37]

$$J[\tilde{f}; \mathcal{T}] = -\frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{n} y_{i,j} log(\tilde{f}(\boldsymbol{x}_i)). \tag{2.16}$$

Equivalently, if there are only two classes, then y_i is one if the data point number i is of class one and zero otherwise. If this is the case, the cross entropy has the following expression [37]

$$J[\tilde{f}; \mathcal{T}] = -\frac{1}{n} \sum_{i=1}^{n} y_i log(\tilde{f}(\boldsymbol{x}_i)) + (1 - y_i) log(1 - \tilde{f}(\boldsymbol{x}_i)). \tag{2.17}$$

2.1.4 Activation functions

There are three main activation functions that we will consider in this text, sigmoidal functions [38], softmax functions [38] and the rectified linear unit (ReLU) nonlinearity [10]. We start by considering the sigmoidal activation function.

Sigmoidal activation function

First, we define the sigmoidal activation function, which is given in Definition 2.1.1.

Definition 2.1.1 (Sigmoidal activation function [38])**.** The sigmodal activation function, $\phi_{sigmoid}$ is given by

$$\phi_{sigmoid}(x) = \frac{1}{1 + exp(-x)}. \tag{2.18}$$

A plot of the sigmoidal activation function as well as its derivative can be seen in Figure 2.1.

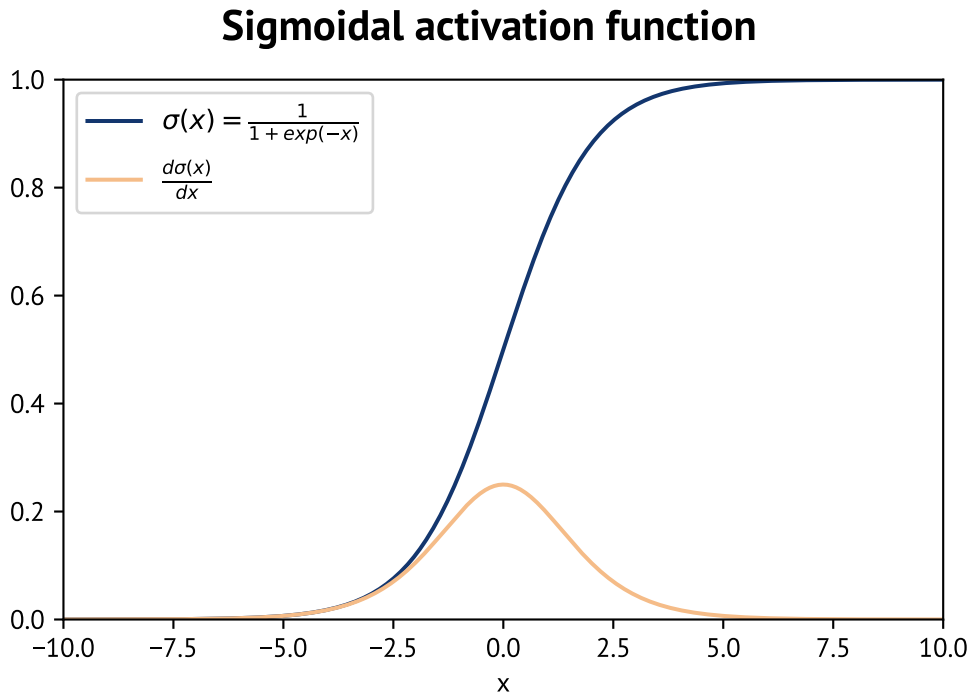


Figure 2.1: The sigmoidal activation function and its derivative.

The main issue with the sigmoidal activation function comes from its derivative. If the magnitude of the input to a sigmoidal function is sufficiently large, then the derivative will be close to zero. This is problematic as we use first order optimisation methods to minimise the loss. Therefore, the update of the network weights (i.e. parameters) are proportional to the magnitude of partial derivative of the loss function with respect to the weights. The magnitude of this gradient is (by the chain rule) proportional to the derivative of the activation. Therefore, if the derivative of the activation is negligible, then the weight updates will be so too. This problem is called the vanishing gradients problem [37].

There is, however, one redeeming quality of the sigmoidal activation function. Namely, that the output is a number between 0 and 1. Thus, it can be regarded as a probability and is often used as the activation function in the final layer of the networks used in binary classification problems.

Softmax activation function

Secondly, we define the softmax activation function, which is given in Definition 2.1.2.

Definition 2.1.2 (Softmax activation function [38])**.** The softmax activation function, $\phi_{softmax}$ is given by

$$\phi_{softmax}(\boldsymbol{x})_i = \frac{exp(x_i)}{\sum_j exp(x_j)}. \tag{2.19}$$

There is, as apparent above, a significant difference between the softmax function and other activation functions. Normally, activation functions take scalars as input, but the softmax function takes vectors as input. The reason for this is that the output vector of a softmax function sums to one. It can in other words be regarded as a probability mass function. As an effect of this interpretation, we see that it is the ideal activation function to use on the final layer in multi-class classification problems.

Rectified Linear Units

Finally, we define the Rectified Linear Unit (ReLU) activation function, which is defined in Definition 2.1.3.

Definition 2.1.3 (ReLU Activation function [10])**.** The ReLU ($\phi_{ReLU} : \mathbb{R} \to \mathbb{R}$) is given by

$$\phi_{ReLU}(x) = max(0, x). \tag{2.20}$$

There are several reasons for why this function sees much use. Firstly, it solves the problem of vanishing gradients; the derivative of ϕ_{ReLU} is zero for negative inputs and one for positive inputs. An illustration of this is given in Figure 2.2. As a consequence, it yields larger update steps and more efficient convergence. In addition, it has the benefit of being efficient to compute; it does not invovle any exponentials. For these reasons ReLU has become the nonlinearity most frequently used today [8], [39], [46], [47].

There are a plethora of other nonlinearities based on ReLUs that sometimes yields better resulsts. Examples of such nonlinearities are the ELU[48], SELU[49], maxout[50], CReLU[51] and many others. We will not introduce those as the increased

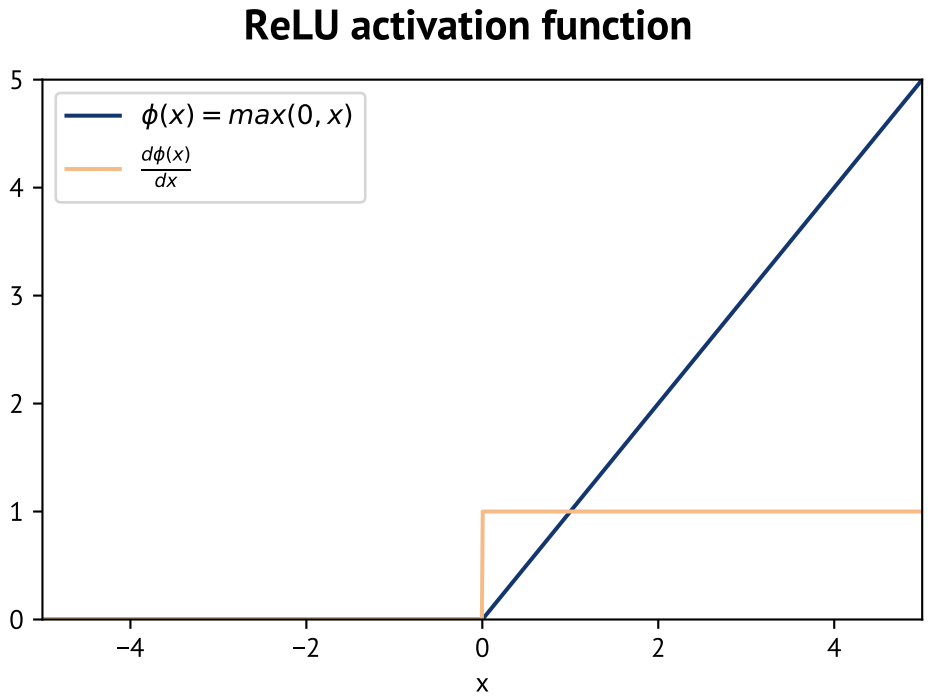


Figure 2.2: The ReLU activation function and its derivative.

performance is often outweighed by the fast computation time of ReLUs. It would also be infeasible to test more nonlinearities within the scope of this project.

2.1.5 Fully connected layers

Definition 2.1.4 (Fully connected layer)**.** Let $\breve{f}_{fc} : \mathbb{R}^m \to \mathbb{R}^n$ be a function on the form

$$\breve{f}_{fc}(\boldsymbol{x}) = \phi(A\boldsymbol{x}), \tag{2.21}$$

with A being a $(n \times m)$-matrix and ϕ an activation function. $\breve{f}_{fc}$ is then a *fully connected layer*.

One kind of layer that has seen much use are fully connected layers. In fully connected layers, the linear mapping L_i is a dense matrix. Thus, all possible linear mappings from the input-space to the output-space of the layer is learnable. At first glance, this might seem like a good idea, however, it has two main downsides; both based on the flexibility of the layer.

Notice how a single fully connected layer has $(n \times m)$ parameters, which results in

an immense amount of parameters for most layers. To illustrate this, consider a layer that takes as an input a 256-by-256 grayscale image and returns a 128-by-128 grayscale image. The total number of parameters for a single layer of this form is 1 073 741 824 and will require four gigabytes of RAM to store (using single precision floating point numbers). This high number of parameters is not only a concern for memory reasons, but it also leads to a high degree of overfitting. Training networks with fully connected layers for image processing is, therefore, not feasible.

The second problem with fully connected layers are a problem that is specific to image processing. For images to be used with fully connected layers, they need to be made into a vector. This is done by simply assigning each element in the vector to the value of a single pixel. As a result of this, the resulting vector, will change drastically by simply translating the contents of the image. This drastic change in the input will then create a drastic change in the output of the layer. This behaviour is unwanted, and a way to combat it is to use *convolutional layers* instead.

2.1.6 A brief interlude on convolutions

Before defining convolutional layers, we introduce definition of a discrete convolution.

Definition 2.1.5. Let A and B be rank N tensors with shape $(m_1, m_2, ..., m_N)$ and $(n_1, n_2, ..., n_N)$ respectively. We define the convolution of A and B as

$$[A * B]_{i_1,...,i_N} = \sum_{j_1=1}^{n_1} ... \sum_{j_N=1}^{n_N} A_{i_1-j_1,...,i_N-j_N} B_{j_1,...,j_N}. \tag{2.22}$$

Furthermore, we name B the convolution kernel, or simply, the kernel.

Generally, all the length of a convolution kernel is the same - e.g. 3×3 or 5×5. We will therefore say that a kernel has size k if it is of size $k \times \ldots \times k$.

There is one problem with the definition above; namely how to deal with the boundary. From the definition, we notice difficulties if i_k is too small as this will require terms with negative indices. Two separate steps are done to alleviate this problem.

Firstly, we require that each index of the convolution kernel has odd size (i.e. n_k is

odd). This ensures that $\frac{n_k \pm 1}{2}$ is a natural number. Following this, we can re-index the kernel. Let n_k be the length of the convolution kernel's (B) i-th dimension. The re-indexed convolution kernel, $\tilde{B}$, is then defined as follows.

$$\tilde{B}_{i_1,\ldots,i_N} = B_{i_1+\frac{n_1+1}{2},\ldots,i_N+\frac{n_1+1}{2},\ldots,i_N}, \tag{2.23}$$

where i_k ranges from $-\frac{n_k-1}{2}$ to $\frac{n_k-1}{2}$. Thus, Equation (2.22) becomes

$$[A * B]_{i_1,\ldots,i_N} = \sum_{j_1=-\frac{n_N-1}{2}}^{\frac{n_N-1}{2}} \ldots \sum_{j_N=-\frac{n_N-1}{2}}^{\frac{n_N-1}{2}} A_{i_1-j_1,\ldots,i_N-j_N} \tilde{B}_{j_1,\ldots,j_N}. \tag{2.24}$$

If we do this re-indexing the boundary trouble arises both if i_k is too small and if it is too large. More specifically, it arises if $\frac{n_k-1}{2} > i_k$ or if $i_k > m_k - \frac{n_k-1}{2}$.

The second step to alleviate the boundary problem can be performed in two ways. One way is to only compute the values of $A * B$ which are valid, shrinking each dimension of the output tensor by $n_k - 1$. Figure 2.3 shows how this is done with 1D convolutions.

Alternatively, we can "pad" the tensor A, expanding it in all directions by "creating" new tensor elements (with indices below 1 or above m_k) that are equal to zero. By doing this, the output of the convolution $(A * B)$ can have the same size as the input(A). An example is given in Figure 2.4.

We will generally use one-dimensional figures to illustrate the convolution concepts as that makes for illustrations that are easier to understand. However an illustration showing how the output of a two-dimensional convolution is given in Figure 2.5 as images are often represented as a stack of two-dimensional tensors.

Convolutions of images are an integral part of deep learning for image processing, and there are certain differences with how image convolutions are performed and how regular convolutions are performed. The reason for this is that n-dimensional images are $n+1$ dimensional constructs, where $(n+1)$-th dimension represent the image channel (e.g. the red, green and blue channels of an RGB image). This dimension is "ignored" when performing a convolution. If there are c different colour channels, then a convolution of an n dimensional image consists of performing c n-dimensional convolutions, one for each channel, and adding the result.

As an effect of how image convolutions are performed, the output has only one channel. To combat this, k different convolutions are often performed, one for each output channel. The convolution kernel for images are therefore an $n + 2$

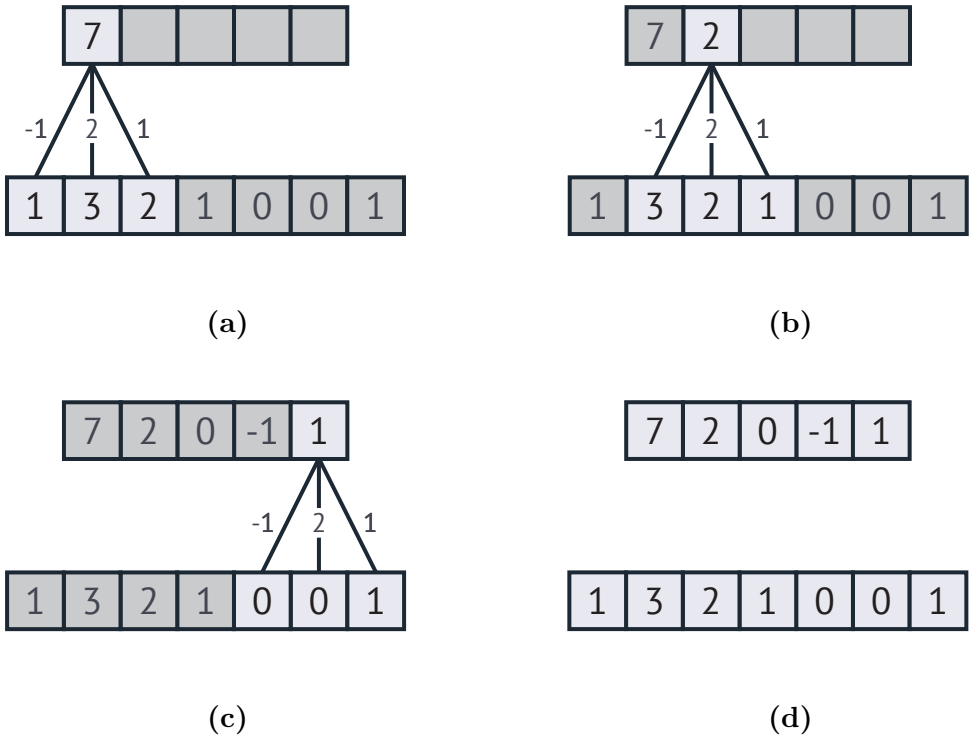


Figure 2.3: Illustration of 1D convolution with no padding. The bottom row of numbers contain the input vector, A, with A_1 being the leftmost element, the convolution kernel, B, is the vector $(1, 2, -1)$ and the top row is the output of the convolution. (a) shows the computation of the first element of $A * B$, (b) shows the computation of the second element of $A * B$, (c) shows the computation of the last element of $A * B$ and (d) shows the input and output vectors of the convolution. Notice that the size of output vector has shrunk with $k - 1$, where k is the kernel size.

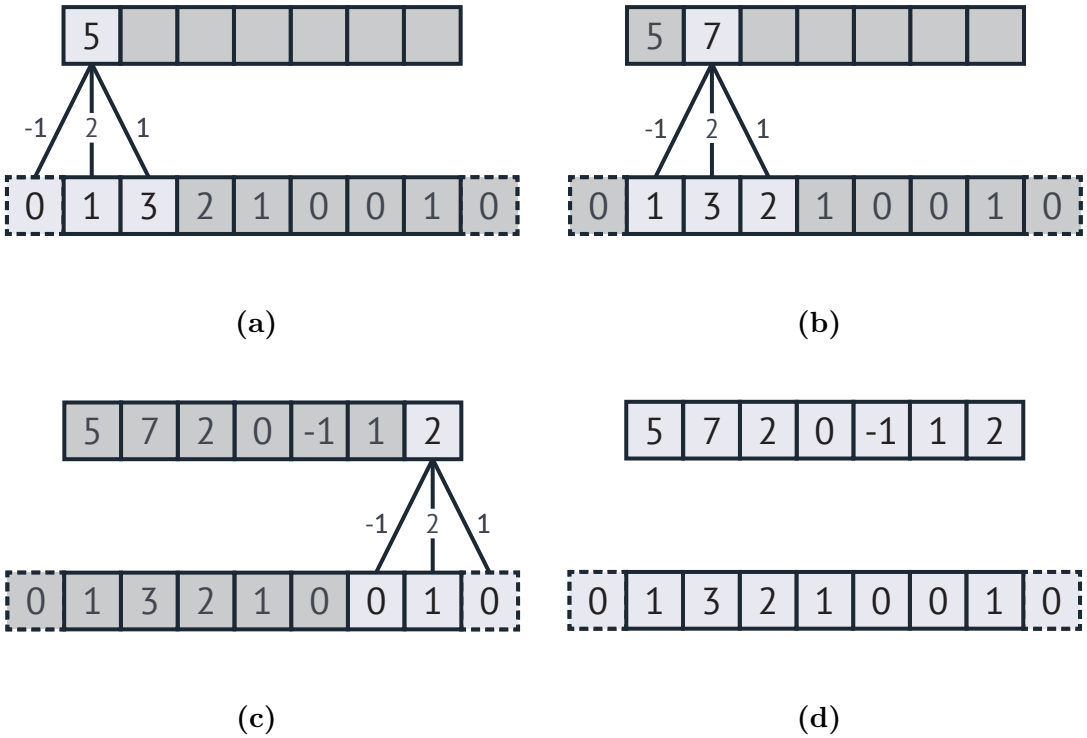


Figure 2.4: Illustration of 1D convolution with padding. The bottom row of numbers contain the input, A, vector with A_1 being the leftmost element, the convolution kernel, B, is the vector $(1, 2, -1)$ and the top row is the output of the convolution. (a) shows the computation of the first element of $A * B$, (b) shows the computation of the second element of $A * B$, (c) shows the computation of the last element of $A * B$ and (d) shows the input and output vectors of the convolution. Notice that the output is the same size as the input, as opposed to convolutions without padding where they shrink by $k - 1$, where k is the kernel size.

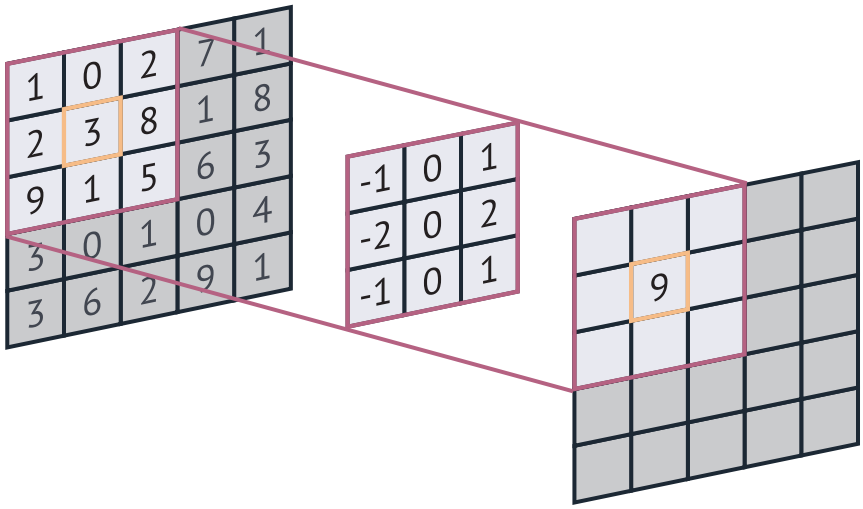


Figure 2.5: Illustration of 2D convolution. The kernel is displayed in the 3×3 square in the middle, and the value of the selected output pixel is computed as follows; $-1 \times 1 + (-2) \times 2 + (-1) \times 9 + 0 \times 0 + 0 \times 3 + 0 \times 1 + 1 \times 2 + 2 \times 8 + 1 \times 5 = 9$. A grayscale image can be represented as a two-dimensional tensor, and the above convolution can be regarded as an edge detector image represented this way [52].

dimensional construct, as c different n-dimensional convolutions are performed to create each of the k output channels.

The way image convolutions are performed means that it makes sense to perform a convolution with a kernel of size 1. This would, for regular convolutions, be equivalent to multiplying the convoluted tensor with a scalar. However, for images it is equivalent by multiplying each pixel with a matrix along the channel dimension. This can be used as a "channel mixing" operator to decrease or increase the number of channels in an image wilst the spatial information is unchanged.

2.1.7 Convolutional layers

Definition 2.1.6 (Convolutional layers)**.** Let $\breve{f}_{conv} : \mathcal{X}_i \to \mathcal{X}_{i+1}$ be a function on the form

$$\breve{f}_{conv}(\boldsymbol{x})_p = \phi(\boldsymbol{k_p} * \boldsymbol{x}), \tag{2.25}$$

where $\boldsymbol{k_p}$ is a tensor of the same order of x, ϕ an activation function and $A * B$ is the convolution of A and B. $\breve{f}_{conv}$ is then a *convolutional layer*.

The output from convolutional layers can be regarded as image feature extractors [8], [40]. Early layers represent low-level feature detectors such as edge and corner detectors. Later layers, on the other hand, represent high-level feature detectors such as snout and eye detectors (for natural image classification) [40]. This makes sense, as the input to the later layers is the output of the earlier layers. Thus, convolutional networks use presence and absence of low-level features to detect higher-level features.

One effect of using convolutions as linear transformations is that the layers become spatially invariant - if an image is shifted one pixel to the right, then the output of the convolutional layers are so too. This is a big contrast to the fully connected layers, where translation invariance can be difficult to learn.

An important question to regarding convolutional layers is what the kernel size should be. It is often set to be 3, after the VGG[2] architecture [46]. In 'Very Deep Convolutional Networks for Large-Scale Image Recognition', Simonyan and Zisserman showed that they could get excellent performance by using many 3×3 convolutional layers.

[2]VGG is short for the Visual Geometry Group in Oxford, the research group that discovered it.

This choice of kernel size might at first glance seem strange, as it limits the spatial size of the features the layers can detect. To show that this is not the case we first have to define the receptive field.

Definition 2.1.7 (Receptive field of a convolutional layer)**.** Let $\breve{f}_1, \breve{f}_2, \ldots, \breve{f}_l$ be convolutional layers with kernel sizes $k_1, k_2, \ldots, k_l$ respectively. The receptive field of f_l, the output of the l-th layer, is the region on the input image that affect one output pixel from the l-th layer.

From the above definition, we see that the receptive field of the first layer is k_1. Furthermore, we see that the receptive field of the l-th layer is given by

$$r_l = 1 + \sum_{i=1^l}(k_i - 1), \tag{2.26}$$

where r_l is the size of the receptive field for the l-th layer and k_i is the kernel size of the k-th layer.

Using the newly defined concept of receptive field, we can get an understanding of why using layers with a kernel size of 3 might be a good idea. Firstly, we notice that there are fewer parameters when using two convolutions of size three than when using one of size five (even though they have the same receptive field). This means that we can train deeper network without reducing the receptive field. These deeper networks will be "more" nonlinear than in shallow networks, which means that more complex models can be trained [41].

Another reason for why the VGG architecture performed so well is hypothesised to revolve around *effective receptive fields* [53]. In 'Understanding the effective receptive field in deep convolutional neural networks', Luo *et al.* demonstrate that, altough the theoretical receptive field might be large, the effective receptive field is not. The influence of pixels near the border of the receptive field is small compared to the centre pixels. For this reason, we say that the effective receptive field is smaller than the theoretical, as the border pixels barely affect the output of a layer.

Using this understanding, we can get some insight as to why stacking several convolutional layers with small kernels gives good results. When we use large kernels, we impose a "hard cutoff" where all pixels outside a box do not affect whether it activates or not, whereas all within have the same amount of influence. Both using a hard cutoff and a box seems arbitrary, and does therefore not make sense intuitively. These problems are overcome when stacking several convolutional layers with small kernels. The influence of pixels gradually decrease as the distance

Figure 2.6: Demonstration of how the effective receptive field is different from the theoretic receptive field. The left figure shows the relative influence of each pixel on the output of a composition of seven convolutions, each of size 3. The right figure shows the relative influence of each pixel on the output for a single convolution of size 15. The white line in the colour bar shows the influence of each pixel on the output of a single convolution of size 15.

from the kernel centre increases, and the shape of this effective perceptual field is more circular as can be seen from Figure 2.6.

It is necessary for the learned feature detectors to have a large receptive field. This is because we want to be able to recognise features that span most, if not all, of the image. As a consequence, we find one problem with using convolutions of size 3. Namely that the receptive field grows slowly. If we want a receptive field of more than 100 pixels, we need 50 layers, which leads to a high number of parameters to estimate. From this, we see that it is necessary to find methods of increasing the receptive field whilst not increasing the number of parameters significantly.

The most popular method of increasing the receptive field of a network is to use downsampling operations [8], [39], [46], [47] and one popular way of doing this is through *strided* convolutions [54]. Strided convolutions work by only computing every s-th output of a convolution, skipping the intermediate values. Figure 2.7 shows how stride work for one dimensional convolutions. The same concept can easily be generalised in several dimensions, skipping s_i values in the i-th dimension. Mathematically, this changes the way the convolution is computed as follows

$$[A * B]_{i_1,\dots,i_N} = \sum_{j_1=-\frac{n_1-1}{2s_1}}^{\frac{n_1-1}{2s_1}} \dots \sum_{j_N=-\frac{n_N-1}{2s_N}}^{\frac{n_N-1}{2s_N}} A_{s_1 i_1 - s_1 j_1,\dots,s_N i_N - s_N j_N} \tilde{B}_{j_1,\dots,j_N}, \tag{2.27}$$

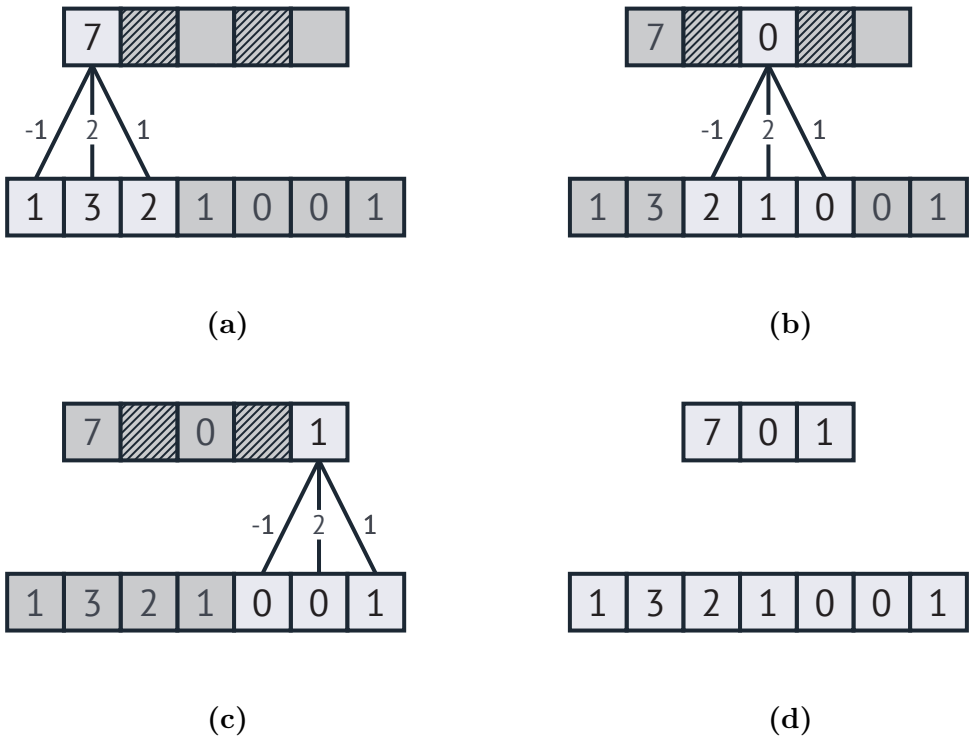


Figure 2.7: Illustration of 1D strided convolution. The bottom row of numbers contain the input vector with A_1 being the leftmost element, the stride is 2, the convolution kernel is the vector $(-1, 2, 1)$ and the top row is the output of the strided convolution.

where once again, B is the convolution kernel and A is the tensor we convolve with B. The stride of the i-th dimension is s_i, however, it is usually constant so $s_i = s_j$ for all i and j. The effect of strided convolutions (and other downsampling operators) is essentially to multiply the receptive field by s, the stride. As a result, such operators are popular to use in neural networks [8], [23], [39], [46].

There is, however, one effect of downsampling operations (also known as *pooling* operations), that may be unwanted. Namely that high frequency information is discarded [23]. It is of interest to find methods that increase the receptive field without discarding any information. One proposed way of doing this is through *dilated* or *atrous*[3] convolutions [56].

Dilated convolutions work in a similar fashion as strided convolutions, but instead

[3] Atrous comes from "algorithme à trous", or "hole argorithm" in English. Atrous convolutions were first proposed to compute fast wavelet transforms [55].

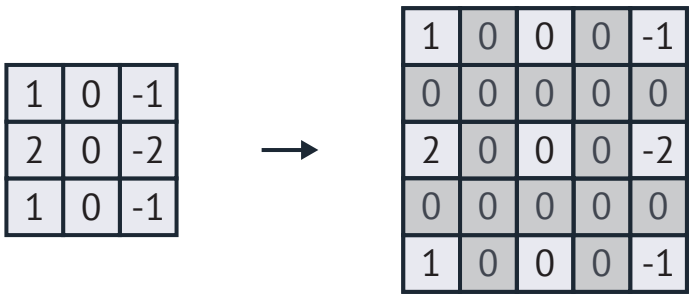


Figure 2.8: Illustration of a dilated convolution kernel. The left kernel is a normal 2D convolution kernel of size three, and the right is the equivalent kernel with a dilation rate, d, of two.

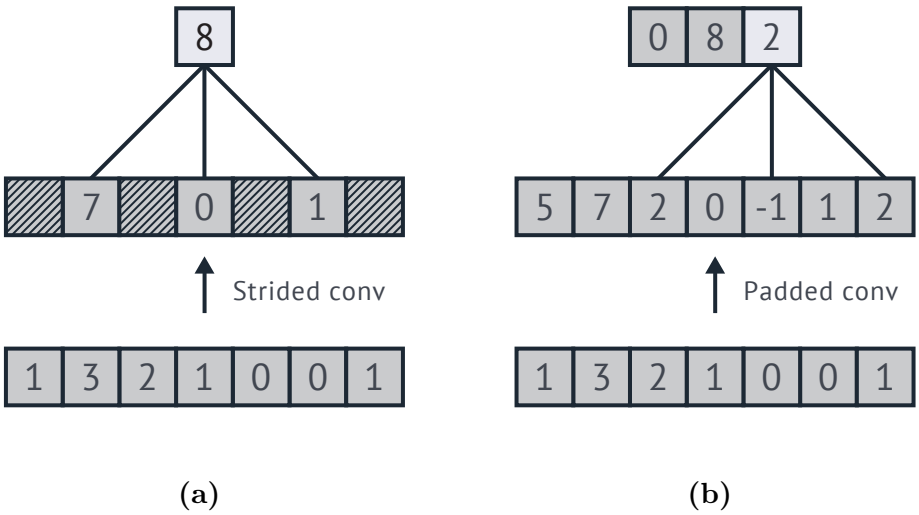
of skipping convolution computations, the filter size is increased without increasing the number of parameters. This is done by modifying the convolution definition as follows

$$[A * B]_{i_1,\ldots,i_N} = \sum_{j_1=-\frac{n_1-1}{2s_1}}^{\frac{n_1-1}{2}} \cdots \sum_{j_N=-\frac{n_N-1}{2}}^{\frac{n_N-1}{2}} A_{i_1-d_1 j_1,\ldots,i_N-d_N j_N} \tilde{B}_{j_1,\ldots,j_N}, \tag{2.28}$$

where B is the convolution kernel and A is the tensor we convolve with B. The dilation rate of the i-th dimension is d_i (which is usually the same for all dimensions). This is equivalent with increasing the size of the convolution kernel, filling the new values with zero, as demonstrated in Figure 2.8.

Notice how the definition of dilated convolutions and strided convolutions are very similar. Performing a strided convolution followed by a "standard" convolution is, in fact, equivalent to performing a normal convolution followed by a strided convolution. This is illustrated in Figure 2.9. Hence, we can increase the receptive field using dilated convolution in a similar fashion to how striding does, but without discarding information the same way. If layer number l has stride s, then the same effect on the receptive field can be attained by multiplying the dilation rate of all subsequent layers by s.

Finally, we note that fully connected layers can be implemented using convolutional layers. This is done by using a kernel that is of the same size as the image and no padding. The output of such a kernel will be an image of size one in all dimensions except for the channel dimension. The image is, in other words, equivalent to a vector. Thus, it has lately been popular to create fully convolutional networks



(a) **(b)**

Figure 2.9: In (a), we see the output of a strided convolution followed by a regular convolution. In (b), we see how the output of the operation demonstrated in (a) is a subset of the output of a normal convolution followed by a dilated convolution.

for classification, as such layers do not bound the input size of an image to the network [54].

2.1.8 Downsampling operations

We have already noted that downsampling operations, or *pooling* operations, are useful in convolutional networks. The strided convolution method described above is often used since it is easy to implement. There are, however, other methods that can be more efficient.

All pooling operators work similarly. Namely by having an n-by-n (for 2D) "window" that "slides" over the image similarly to how a convolution kernel with size n and stride n slides over the image. The difference is that the output of a pooling operator is not necessarily a weighted sum of the input as it is in strided convolutions. See Figure 2.10 for an illustration in 1D and Figure 2.11 for an illustration in 2D.

One of the, if not the, most used pooling operation is called *max pooling*. Max pooling works by computing the maximum pixel-value in the sliding window. This is the pooling operation used in Figure 2.10 and Figure 2.11. A natural interpretation

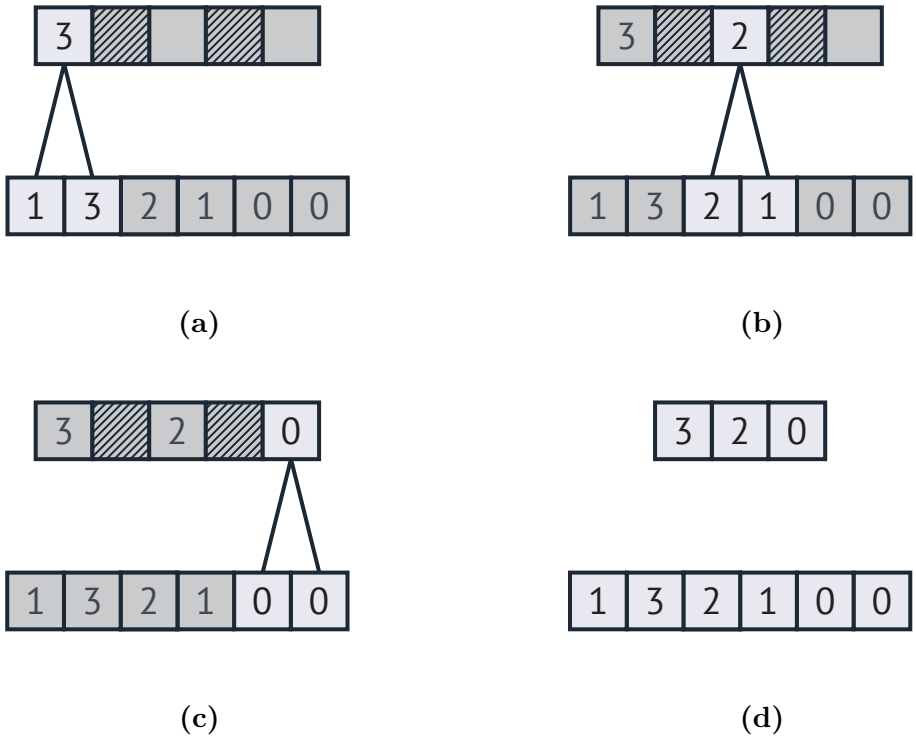


Figure 2.10: An illustration of a one-dimensional pooling operator. In this case, a max-pooling operator, where the output of each location is the maximum value in the input image. (a)-(c) shows the computation of the output of the max pooling operator, whereas (d) shows the input-output pair from the max-pooling operator.

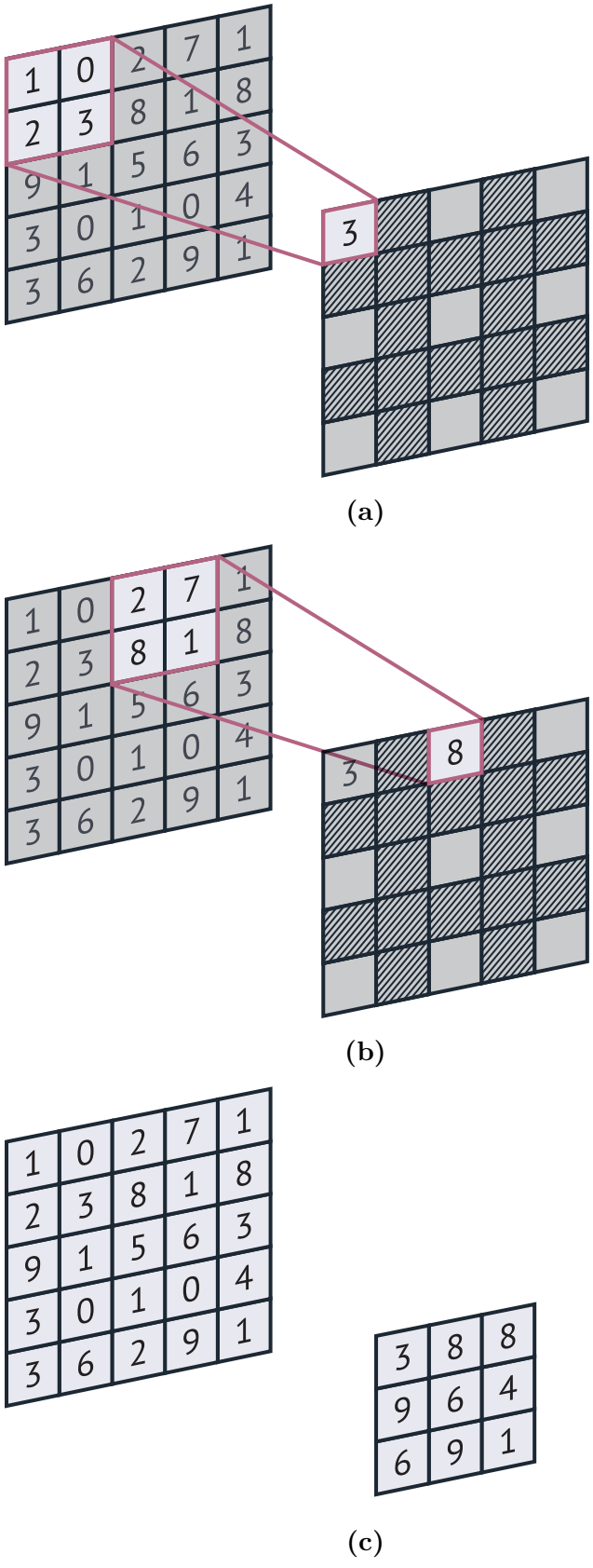


Figure 2.11: An illustration of a two-dimensional pooling operator. In this case, a max-pooling operator, where the output of each location is the maximum value in the input image. (a) and (b) shows the computation of the first and second pixel, where the left image is the input and the right is the output. (c) shows the completed pooling output.

of this pooling operation arises if we consider the feature-extractor interpretation of layers in neural networks. Each pixel value indicate the likelihood of a specific feature being present at the corresponding location in the image. Max pooling will then (roughly) return the likelihood of a specific feature being present in any of the pixels in the pooling window.

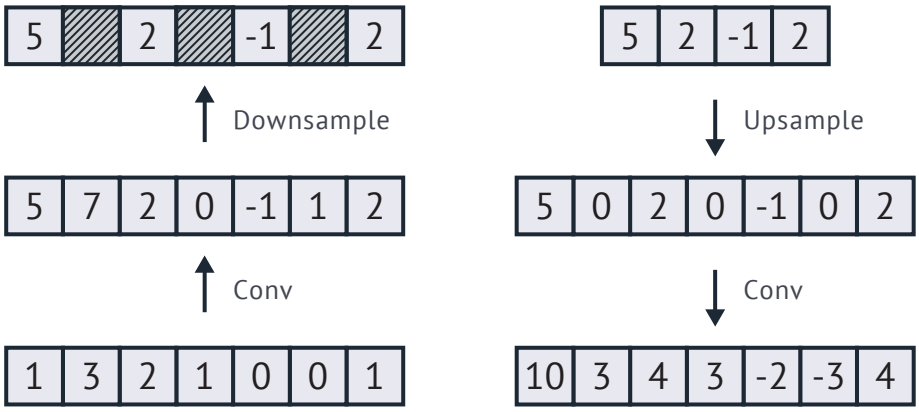
There exists a plethora of other pooling operations [37]. Those will, however, not be introduced here, as max pooling and strides are most often used in practice [23], [39], [46]. Max pooling often yield better results than strides [39], [46], [54], however, strided convolutions are often used instead as a result of their simplicity.

2.1.9 Upsampling operations

If we want to perform image segmentation or compression, then simply using downsampling operations are not sufficient. This is because the output of the network should have the same size as the input. One way of accomplishing this is to replace all pooling operations with dilated convolutions as described in Section 2.1.7 [25]–[27], [56]. This leads to a significant memory footprint and is therefore not always an option. As a solution to this memory problem, we might perform downsampling operations first, followed by upsampling operations later, yielding a smaller memory footprint on the layers between a downsampling and upsampling layer.

The most commonly used upsampling operation is the dual operation of strided convolutions and has many names. We will, in this text, use the name upconvolution (or upconv) [23], but it is often (misleadingly) refered as a deconvolution [57]. Other authors use transposed strided convolutions [58] or fractionally strided convolutions [58]. We will introduce two ways to think about the upconvolution operator. Firstly, it can be seen as the gradient of a strided convolution operator. Alternatively, it can be understood from a linear analysis viewpoint; any linear map $L : \mathcal{X} \to \mathcal{Y}$ has an adjoint operation $L^* : \mathcal{Y} \to \mathcal{X}$ (for matrices, this is the transpose). Thus, if $\mathcal{X}$ is larger than $\mathcal{Y}$, we have an upsampling operation. An illustration of how this upsampling operation works is given in Figure 2.12

Using upconvolutions for upsampling might, at first glance, seem like a logical choice. However, once we study what this operation does, we see that it is not necessarily the optimal choice. When we want to perform an upconvolution with a rate of two (that is the adjoint of a convolution with stride equal to two), we double the size of the input tensor in all directions before performing a standard convolution. This upsampling is done by filling the new tensor values with zeros



(a) **(b)**

Figure 2.12: (a) Shows how strided convolution can be regarded as a convolution operator followed by a downsampling operator. We start with the vector $[1, 3, 2, 1, 0, 0, 1]$, convolve it and downscale to get the vector $[5, 2, -1, 2]$. (b) Shows how the transpose of this convolution operator can be regarded as an upsampling operator. Here, we start with the vector $[5, 2, -1, 2]$, interleave it with zeros and convolve it to get the vector $[10, 3, 4, 3, -2, -3, 4]$.

[59], as showed in the following one-dimensional example.

$$(1, 2, 2, 1) \to (1, 0, 2, 0, 2, 0, 1). \tag{2.29}$$

It is clear that this might not be the best way to upsample an image.

We will now present an informal proof of why upconvolutions work this way. To do this, we consider how strided convolutions work. Consider a convolution operator, $C_k^{(2)}$ with kernel k and stride 2. That operator can be decomposed the following way

$$C_k^{(2)}[\boldsymbol{x}] = D^{(2)}[C_k[\boldsymbol{x}]], \tag{2.30}$$

where $\boldsymbol{x}$ is some tensor, C_k is the unstrided convolution operator with kernel k and $D^{(2)}$ is a downsampling operator which works by taking every second element of $\boldsymbol{x}$. The adjoint of this operator can be written in the following fashion

$$C_k^{(2)*}[\boldsymbol{x}] = C_k^*[D^{(2)*}[\boldsymbol{x}]]. \tag{2.31}$$

This is a basic linear algebra identity ($(AB)^* = B^*A^*$). adjoint of the convolution operator C_k is $C_{k'}$ – the convolution operator with kernel k', which is the mirror image of k along all axes. Thus, the only question remaining is what the adjoint of a downsampling operator is. For one-dimensional inputs, a downsampling operator with downsampling factor two can be expressed by the following matrix:

$$D^{(2)} = \begin{bmatrix} 1 & 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & 1 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \dots & 1 \end{bmatrix}, \tag{2.32}$$

whose adjoint (and therefore transpose) can be described with the following matrix

$$D^{(2)^T} = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix}. \tag{2.33}$$

Thus, using the adjoint of a downsampling operator is equivalent to upscaling the image and setting the new values equal to zero.

It has, as a consequence of how upconvolutions work, been proposed to perform upsampling through interpolation, resulting in the following operator [59].

$$C_{k'}[I^{(2)}[\boldsymbol{x}]],$$

where $C_{k'}$ is the adjoint convolution of C_k and $I^{(2)}$ is an interpolation operator that doubles the size of its input. By doing this, we avoid certain artefacts often seen in networks using upconvolution, namely *checkerboard artefacts* [59]. This is seldom done in practice as sufficient performance can be aquired by upconvolutions [23], [42]. However, if checkerboard artefacts are present, then linear upsampling followed by a convolution should be explored [59].

2.1.10 Batch Normalisation

There is one technique that is integral when training neural networks; batch normalisation [11]. The exact reason why batch normalisation works is still not fully understood [60], [61], but it is a tool seen in almost all modern neural network architectures [23]–[25], [39], [47].

Ioffe and Szegedy originally proposed batch normalization as a method of combatting *internal covariate shift* [11]. Internal covariate shift is the the name of the phenomena that the mean and standard deviations from layer outputs change over training iterations. Ioffe and Szegedy argues that this is a problem since classical machine learning algorithms perform better if the dataset has zero mean and unit variance, that is, it is *normalised*. For this reason this, Ioffe and Szegedy proposed to normalise the output of each layer before feeding it into the next, subtracting the batch mean and dividing by the batch standard deviation.

This batch mean and batch standard deviation is only used during training. During the prediction phase, a true estimate of the dataset mean and standard deviation is used. As a consequence of this, architectures developed after 2015 are often on the form

$$f(\boldsymbol{x}) = \breve{f}_n(BN(\breve{f}_{n-1}(BN(...(\breve{f}_1(BN(\boldsymbol{x})))...)), \tag{2.34}$$

where BN is the batch normalisation function, whose computation is explained in Algorithm 2.1.

Using batch normalisation makes it significantly easier to train neural networks. Before they were introduced, advanced pretraining techniques were used [62]. These techniques worked by first training the network to do one task that it can do unsupervised (e.g. recover the input, instead of generating segmentation masks),

Algorithm 2.1 Batch normalisation

procedure BATCHNORMALISATION($\boldsymbol{x}$, mode)
  if mode == 'train' **then**
    ▷ **Compute batch mean and std:**
      $\mu_{batch} \leftarrow mean(\boldsymbol{x})$
      $\sigma_{batch} \leftarrow std(\boldsymbol{x})$
    ▷ **Update prediction mean and std:**
      $\mu_{prediction} \leftarrow (1-\epsilon)\mu_{prediction} + \epsilon\mu_{batch}$
      $\sigma_{prediction} \leftarrow (1-\epsilon)\sigma_{prediction} + \epsilon\sigma_{batch}$
    return $\frac{\boldsymbol{x}-\mu_{batch}}{\sigma_{batch}}$
  else
    return $\frac{\boldsymbol{x}-\mu_{prediction}}{\sigma_{prediction}}$

before changing its goal to the task at hand [62]. However, this is no longer neccesary to achieve stable learning because we use batch normalisation [23], [39], [43].

One problem that batch normalisation solved was the problem of expoding gradients [11], [49]. The exploding gradients problem in deep neural networks comes from the derivative magnitudes growing as network depth increases. However, by introducing batch normalisation, we get scale-invariant layers, and as such, the exploding gradient problem is alleviated [11].

Furthermore, batch normalisation is known to reduce the effect of L_2 regularisation[4] [63]. The reason for this is that batch normalisation makes the outputs of layers invariant of the scale of the weights. Thus, we can achieve arbitrarily small 2-norm on our weights without changing the network output. However, adding L_2 regularisation does change the way the iterative optimisation algorithms approach the optimal weights [63].

It is important to note that recent work has been done to figure out why batch normalisation work [60], [61]. As a result of this work, its become clear that reduction of internal covariate shift is not the reason for the success of batch normalisation (it may in some cases increase it). The current hypothesis for the success of batch normalisation is that it changes the *loss-landscape* so that the loss function is smoother, and thus, easier to optimise with gradient based methods [60].

[4] L_2 regularisation will be introduced in Section 2.1.12 on page 40.

2.1.11 Residual networks and skip-connections

A key observation in the design of neural networks is that deep networks perform better than shallow [46]. Inspired by this, He *et al.* sought out to create very deep neural networks, with hundreds of layers. However, they noticed one problem with normal convolutional networks, and more importantly, the solution to this problem.

The output of each layer is a feature extractor, early layers detect simple features such as edges and corners, whereas later layers detect more complex features such as eyes and ears (in animal images). Our problem arises when an early layer (f_e) has learned a feature detector needed by a late layer (f_l). If that happens, the network needs to learn identity mappings in all layers between f_e and f_l. Unfortunately, the nonlinearities ϕ_i might make this difficult.

The solution to the problem above is to learn the residual map $\breve{f}_i^{(res)}$ instead of $\breve{f}_i$, given by

$$\breve{f}_i(\boldsymbol{x}) = \breve{f}_i^{(res)}(\boldsymbol{x}) + \boldsymbol{x}. \tag{2.35}$$

Thus, if the optimal function $\breve{f}_i^{(opt)}$ is the identity mapping, $\breve{f}_i^{(opt)}(\boldsymbol{x}) = \boldsymbol{x}$, then the optimal residual map is given by $\breve{f}_i^{(res)}(\boldsymbol{x}) = 0$. Figure 2.13 shows visual explanation of such layers and why adding the input to the output is often called a *skip connection*. He *et al.* named networks using such layers *residual neural networks*, or *resnet*.

There is, one obvious problem with the formula stated above. It will not work if the output dimension is changed after a convolution, either through strides, or through a change in the number of channels. Therefore, an approximate identity map is learned instead, which modifies Equation (2.35) in the following way

$$\breve{f}_i(\boldsymbol{x}) = \breve{f}_i^{(res)}(\boldsymbol{x}) + id(\boldsymbol{x}), \tag{2.36}$$

where id is some function that approximates the identity function. If the output dimension is the same as the input, this is the identity map. However, if $\breve{f}_i^{(res)}$ is a strided convolution, then $\breve{f}_i^{(res)}$ will be a downsampling operation. Equivalently, if $\breve{f}_i^{(res)}(\boldsymbol{x})$ has a different number of channels as $\boldsymbol{x}$, then id is a convolution with kernel size 1 and the same number of channels as $\breve{f}_i^{(res)}(\boldsymbol{x})$.

Residual networks became the solution to training very deep convolutional networks, and makes it possible to train networks with more than 100 layers, achieving state of the art performance [39]. In a later work, the He *et al.* explored different

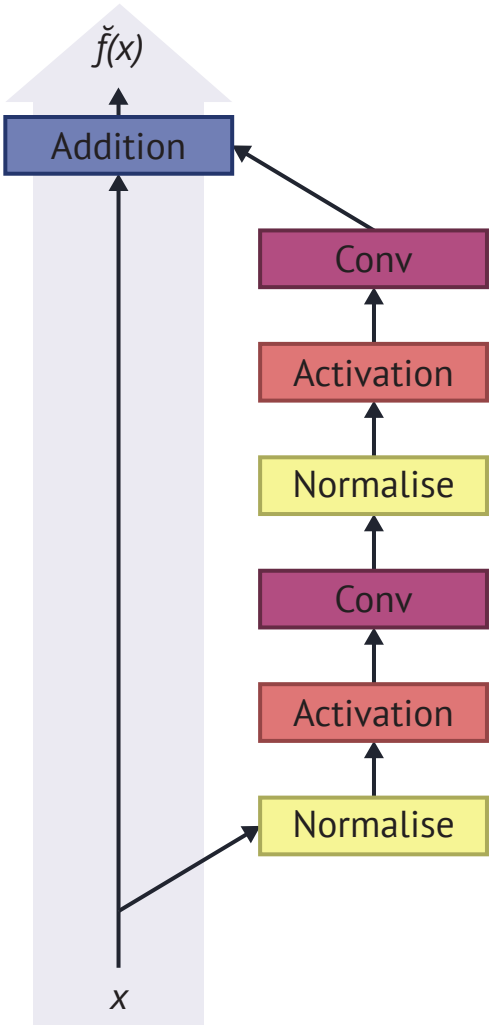


Figure 2.13: A graph showing the structure of a ResNet layer. Notice how the input "skips" past the convolution and nonlinearity, creating a "skip connection". Each square represents a mathematical operation and an arrow starting in a square represents the output of that mathematical operation. The tip of an arrow ending in a square means that the corresponding number is treated as an input to that mathematical operation. The arrow starting in x is the input to the layer and $\breve{f}(x)$ is the output of the layer.

methods of incorporating skip connections in neural networks and the best method they found is shown in Algorithm 2.2.

If we study the mappings learned by residual neural networks, we see that a large portion of the intermediate layers are identity maps (or approximate identity maps) [65]. For this reason, Veit *et al.* defined the *true depth* of a neural network as the number of layers that are not identity maps. By studying a 110 layer ResNet they found that the true depth is in fact between 10 and 34 layers [65].

Algorithm 2.2 A ResNet layer [64]

procedure $\breve{f}_{ResNet}(\boldsymbol{x}$, `mode`, `input_channels`, `output_channels`, `strides`$)$
  if `output_channels == input_channels and strides == 1` **then**
    ▷ **Compute "identity" map**
      $\boldsymbol{s} \leftarrow \boldsymbol{x}$
    ▷ **Compute residual part**
      $\boldsymbol{z} \leftarrow \phi(BN(\boldsymbol{x}))$
      $\boldsymbol{z} \leftarrow C_{k_1}(\boldsymbol{z})$
      $\boldsymbol{z} \leftarrow \phi(BN(\boldsymbol{z}))$
      $\boldsymbol{z} \leftarrow C_{k_2}(\boldsymbol{z})$
    return $\boldsymbol{s} + \boldsymbol{z}$
  else if `output_channels == input_channels and strides != 1` **then**
    ▷ **Compute "identity" map**
      $\boldsymbol{s} \leftarrow D^{(s)}(\boldsymbol{x})$
    ▷ **Compute residual part**
      $\boldsymbol{z} \leftarrow \phi(BN(\boldsymbol{x}))$
      $\boldsymbol{z} \leftarrow C^{(s)}_{k_1}(\boldsymbol{z})$
      $\boldsymbol{z} \leftarrow \phi(BN(\boldsymbol{z}))$
      $\boldsymbol{z} \leftarrow C_{k_2}(\boldsymbol{z})$
    return $\boldsymbol{s} + \boldsymbol{z}$
  else if `output_channels != input_channels` **then**
    ▷ **Compute "identity" map**
      $\boldsymbol{s} \leftarrow C^{(s)}_1(\boldsymbol{x})$
    ▷ **Compute residual part**
      $\boldsymbol{z} \leftarrow \phi(BN(\boldsymbol{x}))$
      $\boldsymbol{z} \leftarrow C^{(s)}_{k_1}(\boldsymbol{z})$
      $\boldsymbol{z} \leftarrow \phi(BN(\boldsymbol{z}))$
      $\boldsymbol{z} \leftarrow C_{k_2}(\boldsymbol{z})$
    return $\boldsymbol{s} + \boldsymbol{z}$

Note that we have some misuse of notation here, $C^{(s)}_1$ is a convolution with stride s and kernel size 1.

2.1.12 Regularisation

Regularisation is an important part of training neural networks. The idea behind it is to impose some information of how the trained network should behave. There are two main methods to accomplish this. The first is by changing the loss function to encourage the behaviour we want. The second method is by changing the training procedure altogether.

One popular method of changing the loss function to impose information is called *2-norm regularisation*, or L_2 *regularisation* [37]. The idea behind this regularisation method is that we want the network to be stable with respect to its input. Mathematically, this is equivalent to the magnitude of the gradient of the network being small.

Luckily, there is one simple way of encouraging a small network gradient,. This is done by modifying the loss function in the following fashion

$$\tilde{J} = J + \gamma ||\mathcal{W}||_2^2, \tag{2.37}$$

where J is the original loss function, $\tilde{J}$ is the modified loss function, γ is some parameter specifying the degree of regularisation and $||\mathcal{W}||_2^2$ is the sum of squared weights. The reason this works is that the magnitude of the derivative of the network is proportional to the magnitude of the weights. Thus, by encouraging low weight values, we encourage stability with respect to the input [38].

Another popular regularisation method is called *early stopping*. This idea behind this method is that by stopping the optimisation early, the network has less time to overfit to the training data. As a result of which, the network is better suited for out-of-sample data. Incidentally, it is possible to prove that early stopping is, in some form, equivalent to 2-norm regularisation for linear least squares problems [37], [66]. Consequently, this leads to 2-norm regularisation often not being used since it requires one to choose the regularisation parameter, γ, as opposed to with early stopping.

Finally, we discuss *dropout* regularisation [67]. The idea behind dropout regularisation is that ensemble methods, such as random forest, generally outperform non-ensemble methods, such as decision trees, in machine learning. However, training ensemble methods take much longer than their non-ensemble counterparts. Unfortunately, training an ensemble of neural networks is not feasible due to the amount of time it would take to train such ensembles. Srivastava *et al.*, therefore, found a method to train a neural network to have some of the same properties as ensemble methods do – and named it dropout.

The way dropout accomplishes this is by introducing a dropout layer to the network, which randomly sets parts of it input to zero. The probability of each input to the dropout layer being set to zero is denoted p and called the dropout rate. This is done by multiplying each input to the dropout layer elementwise with a random tensor whose elements has a probability of p for being equal to 0 and a probability of $1 - p$ for being equal to 1. Typically, p is set close to 1 (e.g. 0.9) [67]. To understand how this is done in pracitce, consider the following example.

Example 2.1.2. Let $\breve{f}_d$ be a dropout layer, with dropout rate $p = 0.6$. Let $f_{d-1}(\boldsymbol{x}) \in \mathbb{R}^m$ be the output of the layer before $\breve{f}_d$. The output of $\breve{f}_d$ can then be described by the following equation

$$f_d(\boldsymbol{x})_i = r_i f_{d-1}(\boldsymbol{x})_i, \tag{2.38}$$

where $i \in \mathbb{Z}_m$, $f_d(\boldsymbol{x})_i$ is the $i - th$ element of the output of our dropout layer and r_i is a random variable that takes the value 1 with a probability of 0.4 and the value 0 with a probability of 0.6.

Let $m = 5$ and $f_{d-1}(\boldsymbol{x}) = (1, 4, 3, 2, 2)$. The output of $\breve{f}_d$ can then be equal to

$$\breve{f}_d(\boldsymbol{x})_i = (0 \times 1, 0 \times 4, 1 \times 3, 1 \times 2, 0 \times 2) = (0, 0, 3, 2, 0). \tag{2.39}$$

The reason models trained with dropout share similarities with ensemble methods is that ensemble methods often use the method of *random projections*. That is, each model trained in an ensemble uses a subset of the available features. Thus, the different models need to find different ways to separate the input in the specified categories (for classification that is). Similarly, if we regard the outputs of each layer as feature detectors, then by setting the output to zero, we train the model to work with a subset of features.

Finally, we note that the dropout layer is removed when the model is predicting, as opposed to training. When that is done, the dropout layer is replaced with a layer that simply multiplies all the layer inputs by the dropout rate, p.

2.1.13 Optimisation

One of the most important parts of deep learning is how we optimise the loss functions. This is done using numerical optimisation algorithms. These algorithms work by choosing an initial guess for the initial weights, and then iteratively update them to improve performance. This optimisation procedure is called *training*.

The idea behind most numerical optimisation methods is as follows. Consider the problem

$$\arg\min_{\mathcal{W}\in\mathbb{R}^n} J(\mathcal{W}), \tag{2.40}$$

where $J : \mathbb{R}^n \to \mathbb{R}$ is some the loss that we want to minimise. We start with an initial "guess", $\mathcal{W}^{(0)}$ (often random) for what the the true minimiser of J is. We then compute some direction $\boldsymbol{d}^{(0)} \in \mathbb{R}^n$, and update our guess for $\mathcal{W}$ in the following fashion

$$\mathcal{W}^{(1)} = \mathcal{W}^{(0)} + \alpha^{(0)}\boldsymbol{d}^{(0)}, \tag{2.41}$$

where α_1 is some positive number that we call the *learning rate*. The same procedure is performed to find $\mathcal{W}^{(2)}$, which is done in the following fashion

$$\mathcal{W}^{(i+1)} = \mathcal{W}^{(i)} + \alpha^{(i)}\boldsymbol{d}^{(i)}. \tag{2.42}$$

Algorithm 2.3 shows a summary of how most optimisation algorithms work[5] [68].

Algorithm 2.3 Numerical Optimisation

1: **procedure** NUMERICALOPTIMISATION($J(\cdot), \mathcal{W}^{(0)}$)
2: $i \leftarrow 0$
3: **repeat**
4: Compute $\alpha^{(i)}$ ▷ Find step length
5: Compute $\boldsymbol{d}^{(i)}$ ▷ Find descent direction
6: $\mathcal{W}^{(i+1)} \leftarrow \mathcal{W}^{(i)} + \alpha^{(i)}\boldsymbol{d}^{(i)}$
7: $i \leftarrow i + 1$
8: **until** Convergence
9: **return** $\mathcal{W}^{(i)}$

There are key parts of Algorithm 2.3 that are left out, namely how to find $\boldsymbol{d}^{(i)}$ and $\alpha^{(i)}$. We start by discussing three methods of computing the direction, $\boldsymbol{d}$, before we introduce three methods of computing the learning rate, α.

Gradient descent

The easiest method to compute $\boldsymbol{d}^{(i)}$ is called *gradient descent*, or *steepest descent*. In this algorithm, the direction that will give the largest local decrease in loss, namely the direction of the negative gradient [68]. That is,

$$\boldsymbol{d}^{(i)} = -\alpha^{(i)}\nabla J(\mathcal{W}^{(i)}), \tag{2.43}$$

[5] In deep learning, we do not perform a line search, which is common in other optimisation algorithms [68]. We can therefore find the step length, α first before the descent direction, $\boldsymbol{d}$.

where $\boldsymbol{d}^{(i)}$ is the descent direction, and $\mathcal{W}^{(i)}$ are th parameters at time step i. Furthermore, J is the loss we want to minimise.

There are several downsides with gradient descent. One is the sheer computational power required to compute ∇J, as is demonstrated by the equation below.

$$\nabla J(\mathcal{W}^{(i)}) = \nabla J(\mathcal{W}^{(i)}; \mathcal{T}) = \nabla \sum_n j(\mathcal{W}^{(i)}; \boldsymbol{x}_n, \boldsymbol{y}_n), \tag{2.44}$$

where $\mathcal{T}$ is the training set, $\boldsymbol{x}_n$ and $\boldsymbol{y}_n$ is the n-th element of the training set and its corresponding output, respectively. Using the linearity of the differentiation operator, we get

$$\nabla J(\mathcal{W}^{(i)}) = \sum_n \nabla j(\mathcal{W}^{(i)}; \boldsymbol{x}_n, \boldsymbol{y}_n). \tag{2.45}$$

Thus, to compute the gradient of the loss function, we need to evaluate $j(\mathcal{W}^{(i)}; \boldsymbol{x}_n, \boldsymbol{y}_n)$ for all the elements in the training set. Unfortunately, this is often infeasible, as the training set is too large..

Luckily, there is a remedy to this problem, namely to replace the gradient ∇J with a random variable ∇J_{rand} with the following property

$$\mathbb{E}[\nabla J_{rand}] = \nabla J. \tag{2.46}$$

That is, the expected value of ∇J_{rand} is equal to the gradient of the loss. This algorithm is called *stochastic gradient descent*, or SGD. Generally, we choose ∇J_{rand} to be equal to

$$\sum_{\boldsymbol{x}_n, \boldsymbol{y}_n \in \mathcal{B}^{(i)}} \nabla j(W^{(i)}; \boldsymbol{x}_n, \boldsymbol{y}_n), \tag{2.47}$$

where $\mathcal{B}_i$ is a (small) random subset of the training set. Algorithm 2.4 shows the stochastic gradient descent algorithm.

The batches $\mathcal{B}_i$ are generally chosen as random subsets from $\mathcal{T}$ without replacement. Thus, if a datapoint has beed chosen from $\mathcal{T}$ in the first iteration, then that same datapoint will not be used in subsequent iterations. Once the whole dataset is iterated through, this procedure is restarted, and all elements from $\mathcal{T}$ is available. This gives rise to the term *epoch*. The number we train over is the number of times the whole dataset has been iterated through.

Another downside with gradient descent is that it does not converge particularly fast [44], [69]. One reason for this is demonstrated in Figure 2.14, namely that simply choosing the direction of steepest descent can lead to oscillating behaviour. There are several methods to combat this, and we will introduce two of them – gradient descent with momentum and Adam.

Algorithm 2.4 Stochastic gradient descent [38]

procedure SGD($J(\cdot), \mathcal{W}^{(0)}$)
  repeat
    Compute $\alpha^{(i)}$
    $\boldsymbol{d}^{(i)} \leftarrow -\nabla J_{rand}(W^{(i)})$
    $\mathcal{W}^{(i+1)} \leftarrow \mathcal{W}^{(i)} + \alpha^{(i)}\boldsymbol{d}^{(i)}$
    $i \leftarrow i + 1$
  until Convergence
  return $\mathcal{W}^{(i)}$

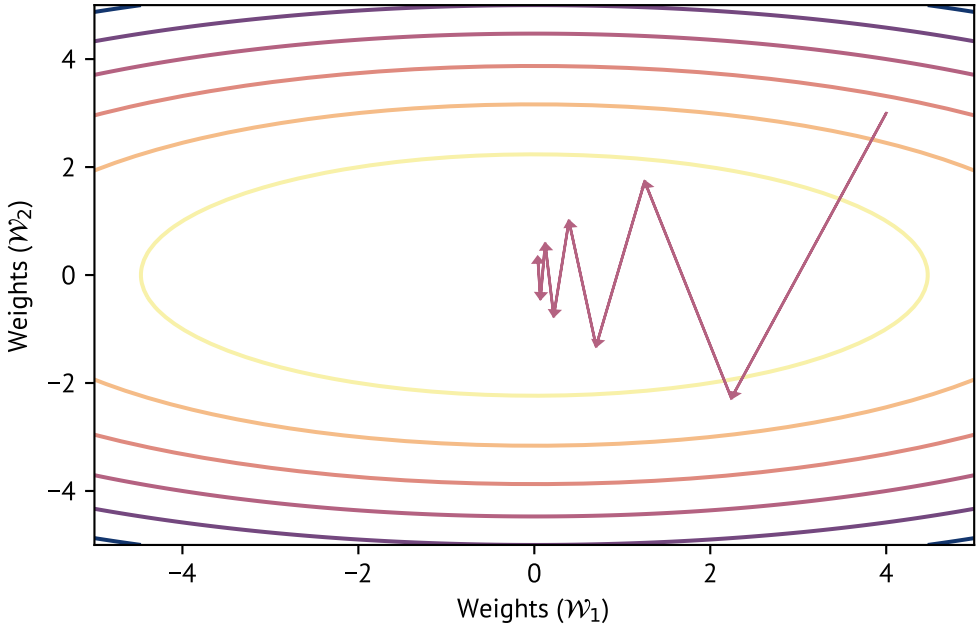


Figure 2.14: Demonstration of how choosing the steepest descent direction can lead to oscillations in $\mathcal{W}$ and thus reduce the convergence speed. The purple arrows correspond to the path taken by a gradient descent optimiser with a too large learning rate and the ellipses are the level curves of a quadratic.

Momentum gradient descent

One method of reducing the oscillation of gradient descent is by using *momentum*. In momentum stochastic gradient descent (momentum SGD), the new descent direction, $\boldsymbol{d}^{(i)}$ is a weighted sum of the previous descent direction, $\boldsymbol{d}^{(i-1)}$, and the gradient, $\nabla J(\mathcal{W}^{(i)})$. The full algorithm is summarised in Algorithm 2.5.

Algorithm 2.5 Momentum gradient descent [69]

procedure MOMENTUMSGD($J(\cdot), \mathcal{W}^{(0)}, \mu$)
  repeat
    Compute $\alpha^{(i)}$
    $\boldsymbol{d}^{(i)} \leftarrow -\alpha^{(i)} \nabla J_{rand}(W^{(i)}) + \mu \boldsymbol{d}^{(i-1)}$
    $\mathcal{W}^{(i+1)} \leftarrow \mathcal{W}^{(i)} + \boldsymbol{d}^{(i)}$
    $i \leftarrow i + 1$
  until Convergence
  return $\mathcal{W}^{(i)}$

There is one tuning parameter, μ, that needs to be set before this algorithm is ran. The purpose of μ is to control the previous iteration should be weighted, and a typical choice for it is 0.9 [70].

The intuition behind momentum gradient descent is similar to that of gradient descent. However, instead taking one step in the direction of steepest descent, we roll a heavy ball with down the landscape parametrised by the loss function. This ball will then achieve momentum in the direction that it is rolling, making it more difficult for it to accelerate in new directions. The $\mathcal{W}^{(i)}$-s is then the position sampled at timestep i after letting the "ball" roll. For an illustration of this, consider Figure 2.15.

Adam

Adam[6] is an algorithm designed by Kingma and Ba to improve momentum gradient descent. The idea is to modify the learning rate of each weight, $\mathcal{W}_n^{(i)}$. The reason for this is that not all parameters oscillate during training, and by giving only the parameters that do so a low learning rate, we might increase the convergence rate. The full algorithm is summarised in Algorithm 2.6.

[6] Adam is not an abbreviation, but its name is inspired by adaptive momentum.

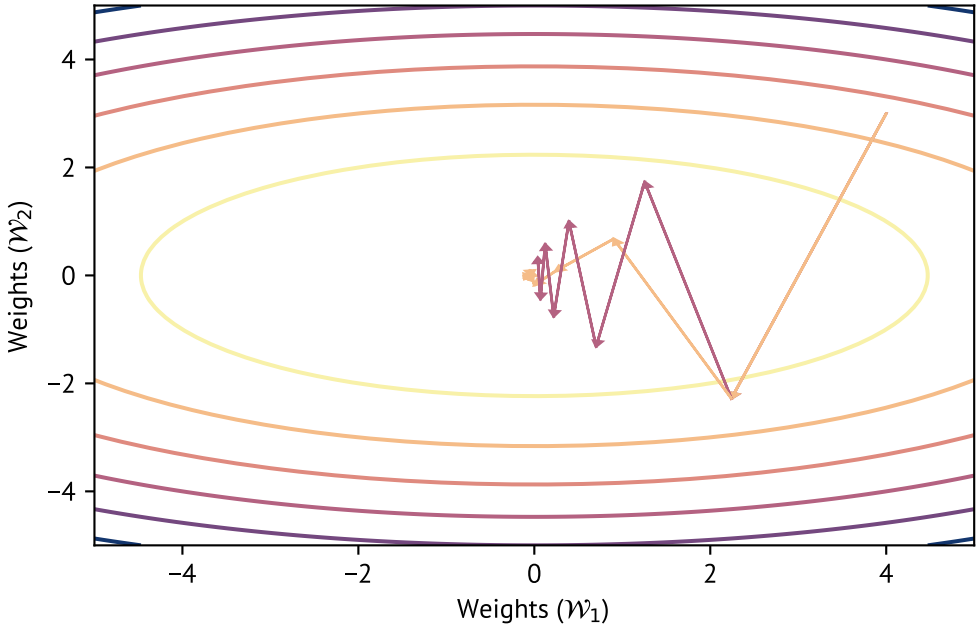


Figure 2.15: Illustration of momentum gradient descent. The purple arrows correspond to the path taken by a gradient descent optimiser and the orange arrows correspond to the path taken by a momentum optimiser. The ellipses are the level curves of a quadratic.

Algorithm 2.6 Adam [44]

procedure ADAM($\mathcal{W}^{(0)}, \beta_1 \in [0,1), \beta_2 \in [0,1)$)
  $\boldsymbol{m}^{(0)} \leftarrow 0$
  $\boldsymbol{v}^{(0)} \leftarrow 0$
  ϵ small
  repeat
    Compute $\alpha^{(i)}$
    ▷ **Compute gradient and moments**
      $\boldsymbol{m}^{(i+1)} \leftarrow (1-\beta_1)\nabla J_{rand}(\mathcal{W}^{(i)}) + \beta_1 \boldsymbol{m}^{(i)}$
      $\boldsymbol{v}^{(i+1)} \leftarrow (1-\beta_2)\left(\nabla J_{rand}(\mathcal{W}^{(i)})\right)^2 + \beta_2 \boldsymbol{v}^{(i)}$
    ▷ **Bias correction**
      $\hat{\boldsymbol{m}}^{(i+1)} = \frac{\boldsymbol{m}^{(i+1)}}{1-\beta_1^i}$
      $\hat{\boldsymbol{v}}^{(i+1)} = \frac{\boldsymbol{v}^{(i+1)}}{1-\beta_2^i}$
    ▷ **Weight Update**
      $\boldsymbol{d}_i \leftarrow -\alpha^{(i)} \frac{\hat{\boldsymbol{m}}^{(i+1)}}{\sqrt{\hat{\boldsymbol{v}}^{(i+1)}}+\epsilon}$
      $\mathcal{W}^{(i+1)} \leftarrow \mathcal{W}^{(i)} + \boldsymbol{d}^{(i)}$
    $i \leftarrow i+1$
  until Convergence
  return $\mathcal{W}^{(i)}$

All operations are element-wise

There is one key similarity between Adam and momentum gradient descent. Notice how the $\boldsymbol{m}$ term in Algorithm 2.6 is similar to the momentum update in Algorithm 2.5. If we set $\mu = \beta_1$ and $\alpha = (1 - \beta_1)$, then Line 4 in Algorithm 2.5 becomes the same as Line 8 in Algorithm 2.6. By doing this, we see that the differences lies on lines 8 through to 14 in Algorithm 2.6.

The main difference between Adam and momentum gradient descent is the $\boldsymbol{v}$ term. The idea of this term that if one parameter encountered very large updates the previous iterations, then we decrease its learning rate. We decrease its learning rate since large updates are often associated with numerical instabilities [44]. An illustration of this phenomena is seen in Figure 2.14. Equivalently, if a parameter has encountered very small updates the previous iterations, then it might be on a plateau [37], and using a large step sizes might therefore decrease the time spent on that plateau.

Next, there are two lines where a modified $\boldsymbol{m}$ and $\boldsymbol{v}$ is computed. The reason for this is that they are initialised as zero, which leads to the $\boldsymbol{m}$ and $\boldsymbol{v}$ vectors being biased towards zero. To fix this, Kingma and Ba derived a bias corrected $\boldsymbol{m}$ and $\boldsymbol{v}$, which are the $\hat{\boldsymbol{m}}$ and $\hat{\boldsymbol{v}}$ terms [44].

Finally, $\boldsymbol{d}$ is computed as $\frac{\hat{\boldsymbol{m}}^{(i+1)}}{\sqrt{\hat{\boldsymbol{v}}^{(i+1)}}+\epsilon}$. Thus, it is the momentum term scaled element-wise by the amount the corresponding parameter has changed the previous iterations. The ϵ term is to provide numerical stability, preventing divison by zero.

It is important to note that the idea of giving each parameter its own learning rate is not unique to Adam. Rather, Adam is an improvement upon the *AdaGrad* algorithm [71] and the *RMSProp* algorithm [72], two similar algorithms that we will not discuss he

A downside with Adam is that it involves choosing good hyperparameters, α, β_1, β_2 and ϵ. Luckily, Kingma and Ba provided recommendations for these parameters in [44], which you can see in Table 2.1 re.

There are both benefits and downsides to using the Adam optimiser. The main benefit is that it is very fast [44]. However, it is demonstrated that models trained with Adam often has worse generalisation properties than those trained with SGD and momentum SGD [73], [74]. This means that networks trained with Adam will often perform worse on data that wasn't used as part of the training procedure. In addition to this, Reddi *et al.* found a mistake in the original Adam article and showed an example where Adam converged to something other than the minimum of a function [75].

Table 2.1: Recommended values for the hyperparameters of Adam [44].

Parameter	Recommended value
α	0.001
β_1	0.9
β_2	0.999
ϵ	10^{-8}

Adam is still used, even though might converge to suboptimal points. The reason for this is that the time saved by training models with Adam allow for more fine tuning of the network architectures. However, the results in [73] demonstrate that training the final model with momentum SGD instead of Adam might improve performance on the validation data.

Learning rate scheduling

We have up til this point not focused on how to choose the learning rate, α. Often, this is just constant, such as in [23]. However, an improved rate of convergence is often seen when a learning rate schedule is used [74].

By a learning rate schedule, we mean that the learning rate is a function of the iteration number, i. We will introduce three methods of choosing the learning rate. The first is simply setting the learning rate to a constant (for Adam, 0.001 is recommended [44]). The second, is to use the following rule

$$\alpha^{(i)} = \alpha(0)\rho^{\xi}, \tag{2.48}$$

where ρ is the *decay rate* chosen beforehand, and ξ is the *epoch* number, given by

$$\xi(i) = \frac{\sum_j |\mathcal{B}|_j}{|\mathcal{T}|}, \tag{2.49}$$

that is, the number of iterations, times the batch size, divided by the size of the training set. If the batches drawn from the training set without replacement, this is the number of times the whole dataset has been used to compute ∇J_{rand}.

One popular modification of the above learning rate schedule is achieved by modifying the expression for ξ in the following fashion:

$$\xi(i) = \left\lfloor \frac{\sum_j |\mathcal{B}|_j}{|\mathcal{T}|} \right\rfloor, \tag{2.50}$$

Table 2.2: Two recommended hyper parameter settings for SGDR [74].

Parameter	Option 1	Option 2
α_{min}	0	0
α_{max}	0.05	0.05
$T_{restart}$ (initial)	100	10
T_{mult}	1	2

where $\lfloor \cdot \rfloor$ is the *floor* function. Thus, the above schedule is a discrete version of the linear decay schedule above where the learning rate is decreased by a factor of ρ every epoch.

The final method of computing the learning rate we will discuss is called *Stochastic Gradient Descent with Warm Restarts*, or *SGDR*. Other optimisation algorithms benefit from restarting the momentum and learning rate [74]. Therefore, Loshchilov and Hutter proposed to use a learning rate that gradually decays to zero before restarting [74]. Their proposed scheme is intended to be used with SGD (SGDR) and momentum SGD (SGDR+momentum) and uses the following learning rate schedule [74]

$$\alpha^{(i)} = \alpha_{min}^{(i)} + \frac{1}{2}(\alpha_{max} - \alpha_{min}) \left(1 + cos\left(\frac{T^{(i)}}{T_{restart}}\pi\right)\right). \tag{2.51}$$

α_{max} and α_{min} is the maximum and minimum learning rate, respectively, $T^{(i)}$ is the number of epochs since last restart and $T_{restart}$ is the number of iterations between each restart. Loshchilov and Hutter reccommend that $T_{restart}$ is multiplied by a factor of $T_{mult} > 1$ every restart. Thus, if we use momentum stochastic gradient descent with an SGDR learning rate scheme, we get Algorithm 2.7. A table of recommended hyperparameters is seen in Table 2.2 [74] and an illustration of how the SGDR learning rate schedule varies with training step is shown in Figure 2.16.

Initialisation

How we initialise neural networks, that is how we choose $\mathcal{W}^{(0)}$, can have substantial effects on the training result [37]. We will, therefore, introduce one popular initialisation algorithm – He initialisation [76].

Algorithm 2.7 Momentum SGDR [74].

procedure SGDR+MOMENTUM($\mathcal{W}^{(0)}$, μ, α_{min}, α_{max}, $T_{restart}$, T_{mult})
  $|\mathcal{B}| \leftarrow$ batch size
  $|\mathcal{T}| \leftarrow$ size of training set
  $T_{cur} \leftarrow 0$
  repeat
    ▷ **Compute** $\alpha^{(i)}$
      $T_{cur} \leftarrow T_{cur} \frac{|\mathcal{B}|}{|\mathcal{T}|}$
      if $T_{cur} \geq T_{restart}$ **then**
        $T_{cur} \leftarrow T_{cur} - T_{restart}$
        $T_{restart} \leftarrow T_{mult} T_{restart}$
      $\alpha^{(i)} \leftarrow \alpha_{min} + 0.5(\alpha_{max} - \alpha_{min}) \left(1 + cos\left(\frac{T_{cur}}{T_{restart}}\pi\right)\right)$
    ▷ **Momentum SGD update**
      $\boldsymbol{d}^{(i)} \leftarrow -\nabla J(W^{(i)}) + \mu d^{(i)}$
      $\mathcal{W}^{(i+1)} \leftarrow \mathcal{W}^{(i)} + \alpha^{(i)} \boldsymbol{d}^{(i)}$
  until Convergence
  return $\mathcal{W}^{(i)}$

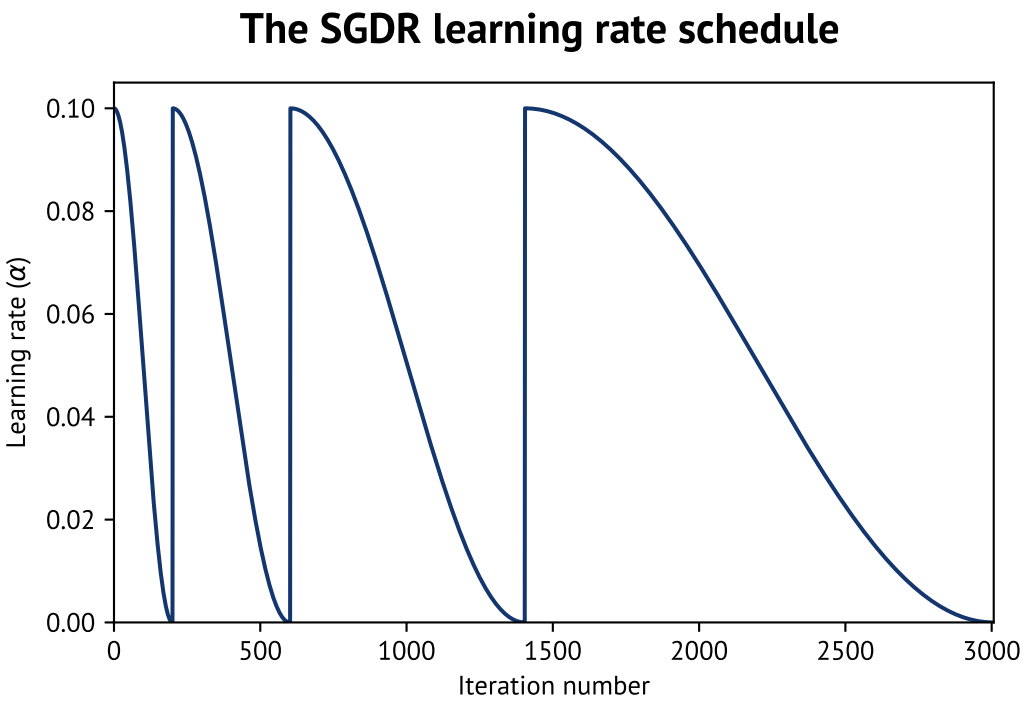


Figure 2.16: An illustration of the SGDR learning rate schedule. The parameters used here are: $\alpha_{min} = 0, \alpha_{max} = 0.1, T_{restart} = 2, T_{mult} = 2$.

He initialisation builds upon work by Glorot and Bengio, who discovered that choosing weights so that the variance of each layer's output should be equal to one. Thus, if we use ReLU nonlinearities, [76] showed that the weights should be chosen so that

$$var(\mathcal{W}_i) = \frac{2}{dim(\mathcal{X}_i)}, \tag{2.52}$$

where $dim(\mathcal{X}_i)$ is the dimensionality of the input space of $\breve{f}_i$.

2.2 Splitting the dataset

One problem with deep learning is that the models tend to overfit to the data [37]. Thus, the performance at the end of the training is not representative to how the model will perform on new data. To prevent this from being a problem, we divide the dataset in three parts. The first part is used to train the networks, and is often the largest. The other two are both used to test the models on out-of-sample data. One of these datasets is used to compare different models aptitude at handling such data and is called the *validation dataset*. Finally we need a dataset that is solely used to test the final model, this is called the *test dataset*.

2.3 Deep learning for image segmentation

The previous section covered general deep learning for image analysis, however, there are certain tools that are only used in segmentation algorithms. Those tools will be the focus of the following two sections

2.3.1 Performance metrics

There is one large problem with image segmentation as opposed to other image analysis algorithms – the difficulty of dealing with class imbalance. In classification problems, we classify each pixel in an image, and the object(s) we are interested in segmenting generally occupy a small are of the image. Thus, the (vast) majority of the pixels are background pixels.

The reason class imbalance problems are a particular problem in segmentation is that it cannot be resolved with the sampling algorithms [77]. These algorithms

work by subsampling, or oversampling the dataset to attain class imbalance. Unfortunately, this is not feasible for images, as whole images are fed into the network at once.

Thus, segmentation accuracy, that is, the fraction of correctly classified pixels is not an ideal measure of a networks performance. To illustrate this, consider an image where 9000 pixels are of the background class and 1000 pixels are of the infected tissue class. In this setting, an algorithm that classifies all pixels as background pixels will achieve an accuracy of 90% whilst completely failing at tackling the task at hand. As a result of this, we introduce several performance measures when we compare segmentation algorithms.

All performance measures we introduce are designed for binary classification problems. The background class is the *negative* class and the class of interest is the *positive* class. For segmentation problems with more than one class, the performance measures can be computed for each class separately. The class of interest is then used as the positive class and all other classes are combined into the negative class.

There are four terms that are integral when we design performance measures. *true positives*, *false positives*, *true negatives*, *false negatives*. See definitions 2.3.1 - 2.3.3 for a definition of those terms.

Definition 2.3.1 (True positives)**.** The number of true positives (TP) is the number of pixels belonging to the positive class that were correctly predicted as members of that class.

For a single image, TP is the total number of pixels belonging to the positive class that were correctly predicted as members of that class. For several images, this is summed up across all images.

Definition 2.3.2 (True negatives)**.** The number of true negatives (TN) is the number of pixels belonging to the negative class that were correctly predicted as members of that class.

For a single image, TN is the total number of pixels belonging to the negative class that were correctly predicted as members of that class. For several images, this is summed up across all images.

Definition 2.3.3 (False negatives)**.** The number of false negatives (FN) is the number of pixels belonging to the positive class that were incorrectly predicted as members of the negative class.

For a single image, FN is the total number of pixels belonging to the positive class that were incorrectly predicted as members of the negative class For several images, this is summed up across all images.

Definition 2.3.4 (False positives)**.** The number of false positives (FP) is the number of pixels belonging to the negatives class that were incorrectly predicted as members of the positive class.

For a single image, FP is the total number of pixels belonging to the Negative class that were incorrectly predicted as members of the Positive class For several images, this is summed up across all images.

Two very popular performance metrics are *sensitivity* and *specificity* [36]. These terms are defined in Definition 2.3.5 and Definition 2.3.6. Sensitivity measures the networks ability to correctly detect positive pixels, and specificity measures the networks ability to correctly detect negative pixels.

Definition 2.3.5 (Sensitivity)**.** The *sensitivity* is the *true positive rate* (TPR) of a network. That is, the proportion of the positives that were correctly identified. Mathematically, this is the same as

$$TPR = P(Predictedpositive|Positive) = \frac{TP}{TP + FN}, \tag{2.53}$$

where TP is the number of true positives and FN is the number of false negatives.

Another word used for sensitivity is *recall*.

Definition 2.3.6 (Specificity)**.** The *specificity* is the *true negative rate* (TNR) of a network. That is, the proportion of the negatives that were correctly identified. Mathematically, this is the same as

$$TNR = P(Predictednegative|Negative) = \frac{TN}{TN + FP}, \tag{2.54}$$

where TN is the number of true negatives and FP is the number of false positives.

Another word used for specificity is *selectivity*.

Using only the sensitivity or onlyt ehs specificity doesn't properly reflect model performance. However, the combination of the two carries much information about the networks efficiency. The reason for this, is that we can get a sensitivity of one if all pixels are predicted as positive pixels. Similarly, the specificity is one if all pixels are predicted as negative pixels. Thus, simply having a network with high sensitivity or specificity is not particularly enlightening.

Another performance metric that is important when reviewing tumour segmentation maps is the *precision*, or *positive predictive value* [78]. The definition for this metric is seen in Definition 2.3.7.

Definition 2.3.7 (Positive predictive value)**.** The *positive predictive value* (PPV), or *precision*, of a network is the probability that a positively predicted pixel in fact belong to the positive class. Mathematically, this is the same as

$$PPV = P(Positive|Predictedpositive) = \frac{TP}{TP + FP}, \tag{2.55}$$

where TP is the number of true positives and FP is the number of false positives.

The main downside with the PPV is that it is dependent on the dataset it is computed upon. However, if the class imbalance in our dataset is "typical", then the precision can be more descriptive than only the sensitivity and specificity. The reason for this is that the precision can be low, even if both the sensitivity and specificity is high so long as the class imbalance is severe enough. Thus, PPV gives a clear indication of how adapt the network is at detecting objects-of-interest in real-world images.

One metric that is particularly popular to assess image segmentation algorithms is the *Sørensen-Dice coefficient* [79], [80]. This particular metric has many names;

some of them are: *Dice similiarity coefficient* (DSC), *Dice score*, F*-score* and F_1*-score*. The Dice score is defined in Definition 2.3.8.

Definition 2.3.8 (Dice score)**.** The *Dice score* (DSC) is defined as the harmonic average of the precision (PPV) and the sensitivity (TPR). Mathematically, this is equivalent to

$$DSC = \frac{1}{\frac{\frac{1}{TPR}+\frac{1}{PPV}}{2}}, \tag{2.56}$$

that is, the reciprocal of the mean of the reciprocals of PPV and TPR. This can be rewritten as

$$DSC = \frac{2}{\frac{1}{TPR}+\frac{1}{PPV}} = \frac{2TP}{2TP+FN+FP}, \tag{2.57}$$

where TP is the number of true positives, FN is the number of false negatives and FP is the number of false positives.

From Definition 2.3.8, we see that one benefit of the Dice coefficient is that it doesn't involve the number of true negatives. This is beneficial since, if the object-of-interest is small compared to the background, then it is very easy to get a high number of true negatives.

The Dice coefficient is an average (specifically, the harmonic mean) of the sensitivity and precision. It is, therefore, easy to generalise it to a weighted average of sensitivity and precision. Doing this gives rise to the F_β score [81], which is defined in Definition 2.3.9. Thus, we have a method to weigh sensitivity more than precision and vice versa.

Definition 2.3.9 (F_β score)**.** The F_β score is defined as the weighted harmonic average of the precision (PPV) and the sensitivity (TPR), where the weight of the precision is β^2 and the sensitivity is 1. Mathematically, this is equivalent to

$$F_\beta = \frac{1}{\frac{\frac{1}{TPR} + \frac{\beta^2}{PPV}}{1+\beta^2}}, \tag{2.58}$$

which can be rewritten as

$$F_\beta = \frac{1+\beta^2}{\frac{\beta^2}{TPR} + \frac{1}{PPV}} = \frac{(1+\beta^2)TP}{(1+\beta^2)TP + \beta^2 FN + FP}, \tag{2.59}$$

where TP is the number of true positives, FN is the number of false negatives and FP is the number of false positives.

When we use the F_β score, the sensitivity has β times as much influence on the score as the PPV. Thus, if we value sensitivity more than the PPV, β should be high and vice versa. Explaining why the β value must be squared is outside the scope of this project, and the interested reader is recommended to read pages 133 and 134 of *Information Retrieval* by Rijsbergen [81].

2.3.2 Loss functions for image segmentation

One problem with the cross entropy loss is that it is sensitive to class imbalance [42]. As a result of this, Milletari *et al.* designed a loss function specific for segmentation that is based on the Dice coefficient. Their Dice loss is defined as follows

$$j_{DSC}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = 1 - \frac{2\sum_i [y_i \hat{y}_i]}{\sum_i [y_i^2] + \sum_i [\hat{y}_i^2]}, \tag{2.60}$$

where $\boldsymbol{y}$ is the true segmentation mask for an image and $\hat{\boldsymbol{y}}$ is the predicted segmentation mask for an image. If pixel i belongs to the positive class, then $y_i = 1$. Similarly, $y_i = 0$ if pixel i belongs to the negative class.

When Milletari *et al.* derived Equation (2.60), they stated, without a proof, that

$$DSC = \frac{2\sum_i [y_i \hat{y}_i]}{\sum_i [y_i^2] + \sum_i [\hat{y}_i^2]} \tag{2.61}$$

for binary vectors $\boldsymbol{y}$ and $\hat{\boldsymbol{y}}$ [42]. Furthermore proposed to use this identity in the case where $\hat{y}_i$ is the predicted probability of pixel i belonging to the positive

class. This relaxation is sensible, as demonstrated by the increase in performance Milletari *et al.* observed when using their Dice loss instead of a cross entropy loss [42].

To demonstrate why Equation (2.61) holds true, we prove Theorem 2.3.1. Then, we motivate the squares below thedenominator. After which, we introduce a general F_β loss that, to the authors knowledge, hasn't published in the current litterature.

Theorem 2.3.1. *Let $\boldsymbol{y}$ and $\hat{\boldsymbol{y}}$ be binary vectors representing segmentation maps. Then, the F_β score with the ground truth vector being $\boldsymbol{y}$ and prediction vector being $\hat{\boldsymbol{y}}$ is given by*

$$F_\beta = \frac{(1+\beta^2)\sum_i [y_i^p \hat{y}_i^q]}{\beta^2 \sum_i [y_i^r] + \sum_i [\hat{y}_i^s]}, \tag{2.62}$$

where $p, q, r, s \neq 0$.

To prove this, we need the following lemmas, which we give without proof.

Lemma 2.3.2. *Let $\boldsymbol{y}$ and $\hat{\boldsymbol{y}}$ be binary vectors representing segmentation maps. Then, the number of true positives, (TP) is given by*

$$\sum_i [y_i \hat{y}_i]. \tag{2.63}$$

Lemma 2.3.3. *Let $\boldsymbol{y}$ and $\hat{\boldsymbol{y}}$ be binary vectors representing segmentation maps. Then, the number of false positives, (FP) is given by*

$$\sum_i [(1-y_i)\hat{y}_i]. \tag{2.64}$$

Lemma 2.3.4. *Let $\boldsymbol{y}$ and $\hat{\boldsymbol{y}}$ be binary vectors representing segmentation maps. Then, the number of false negatives, (FN) is given by*

$$\sum_i [y_i(1-\hat{y}_i)]. \tag{2.65}$$

Proof of Theorem 2.3.1. We want to prove that

$$F_\beta = \frac{(1+\beta^2)\sum_i [y_i^p \hat{y}_i^q]}{\beta^2 \sum_i [y_i^r] + \sum_i [\hat{y}_i^s]} \tag{2.66}$$

for all $p, q, r, s \neq 0$.

To do this, we start with the definition of F_β;

$$F_\beta = \frac{(1+\beta^2)TP}{(1+\beta^2)TP + \beta^2 FN + FP}. \tag{2.67}$$

Inserting Lemmas 2.3.2 - 2.3.4, we get the following equation

$$F_\beta = \frac{(1+\beta^2)\sum_i[y_i\hat{y}_i]}{(1+\beta^2)\sum_i[y_i\hat{y}_i] + \beta^2\sum_i[y_i(1-\hat{y}_i)] + \sum_i[(1-y_i)\hat{y}_i]}, \tag{2.68}$$

we then divide the nominator and denominator by TP, which yields

$$F_\beta = \frac{(1+\beta^2)}{(1+\beta^2) + \beta^2\frac{\sum_i[y_i(1-\hat{y}_i)]}{\sum_i[y_i\hat{y}_i]} + \frac{\sum_i[(1-y_i)\hat{y}_i]}{\sum_i[y_i\hat{y}_i]}}. \tag{2.69}$$

Following that, we use the fact that $\sum_i[a_i(1-b_i)] = \sum_i[a_i] - \sum_i[a_ib_i]$ to get

$$F_\beta = \frac{(1+\beta^2)}{(1+\beta^2) + \beta^2\frac{\sum_i[y_i]-\sum_i[y_i\hat{y}_i]}{\sum_i[y_i\hat{y}_i]} + \frac{\sum_i[\hat{y}_i]-\sum_i[y_i\hat{y}_i]}{\sum_i[y_i\hat{y}_i]}}, \tag{2.70}$$

which through simple arithmetic becomes

$$F_\beta = \frac{(1+\beta^2)}{(1+\beta^2) + \beta^2\frac{\sum_i[y_i]}{\sum_i[y_i\hat{y}_i]} - \beta^2 + \frac{\sum_i[\hat{y}_i]}{\sum_i[y_i\hat{y}_i]} - 1}. \tag{2.71}$$

Rewriting this, we get

$$F_\beta = \frac{(1+\beta^2)}{\frac{\beta^2\sum_i[y_i]+\sum_i[\hat{y}_i]}{\sum_i[y_i\hat{y}_i]}}, \tag{2.72}$$

which we can rewrite as

$$F_\beta = \frac{(1+\beta^2)\sum_i[y_i\hat{y}_i]}{\beta^2\sum_i[y_i] + \sum_i[\hat{y}_i]}. \tag{2.73}$$

The only part missing for this equation to be the statement of the theorem are the nonzero powers p, q, r and s. To show that they can be placed on their respective places in the equation above, we note that $y_i, \hat{y}_i \in \{0, 1\}$. Thus,

$$y_i = y_i^k \qquad \forall k \neq 0 \tag{2.74}$$

and

$$\hat{y}_i = \hat{y}_i^k \qquad \forall k \neq 0. \tag{2.75}$$

Using this, we see that

$$F_\beta = \frac{(1+\beta^2)\sum_i[y_i^p\hat{y}_i^q]}{\beta^2\sum_i[y_i^r] + \sum_i[\hat{y}_i^s]} \tag{2.76}$$

for all $p, q, r, s \neq 0$, which is what we intended to prove. □

An obvious question now is how to choose the powers p, q, r and s. One immediate choice for these powers is $p = q = r = s = 1$, however, this is not what Milletari *et al.* chose. Instead they used $p = q = 1$ and $r = s = 2$. Since $\hat{y} \in [0, 1]$ and $y \in \{0, 1\}$, this leads to the denominator of the fraction becoming larger and uncertaintes in $\hat{\boldsymbol{y}}$ is, therefore, penalised less than if $r = s = 1$. We follow that decision and define a loss function that directly optimises the F_β score.

Definition 2.3.10 (F_β loss)**.** Let β be a positive real number. The F_β loss is then defined as

$$j_{F_\beta} = 1 - \frac{(1 + \beta^2) \sum_i [y_i \hat{y}_i]}{\beta^2 \sum_i [y_i^r] + \sum_i [\hat{y}_i^s]}. \tag{2.77}$$

The F_β loss (of which, the Dice loss is a special case) has one key benefit compared to the cross entropy loss. This stems from the fact that the cross entropy loss sums up the prediction error for each pixel in an image. The F_β loss, on the other hand, considers the quality of the whole segmentation map at once, and sums this up for each image. Intuitively, it therefore makes more sense to optimise the F_β score than the cross entropy.

There is also a downside with the F_β loss. Namely that it considers only two-class segmentation problems. This is problematic, as it can be useful to segment different organs with one algorithm. One way to solve this problem, is by using one-versus-all strategy. By doing this, we compute the loss for each class separately, using the class of interest as positive and all others as negative. This will give one loss per class, which we can average to get a single metric to optimise for.

2.3.3 Architectures for segmentation

One early method of designing convolutional networks for image segmentations was through an *encoder-decoder* architecture (also known as an hourglass architecture) [37], [57]. Encoding-decoder architectures are split in two parts, the *encoder network* and the *decoder network*.

In the encoder network, the input image is fed through convolutional layers and downsampling layers (e.g. two convolutional layers between each downsampling layer). This network generates a low dimensional "encoding" of the input image. The low dimensional encoding of the input image is then fed through a *decoder* network. This network alternates between a set amount of convolutional layers and upsampling layers (e.g. two convolutional layers between each upconvolution).

Thus, "mirroring" the encoder network. The output of the decoder network is the final segmentation mask.

There is one downside with the encoder-decoder architecture for image segmentation; high frequency information (e.g. information about borders) is discarded in the downsampling operations. As a consequence, the segmentation masks are often only rough estimates of the ground truth, missing the fine-detailed structure [82].

One way to combat the fact that low-frequency information is discarded in encoder-decoder network is through long-distance skip connections, which was introduced in the U-Net architecture [23]. By long-distance skip connections, we mean that the input to the first downsampling operator is concatenated to the output of the last upsampling operator. Similarly, the input to the second downsampling operator is concatenated to the output of the second to last upsampling operator, and so on. An illustration of this is given in Figure 2.17.

The U-Net architecture we introduce here differ slightly from the original architecture proposed in [23]. The original U-Net architecture used no padding for their convolutions, and had to crop the inputs to max pooling layers before concatenating them onto the outputs of upsampling operations. Moreover, we use upconvolutions of size 3×3 instead of 2×2. The final layer was a convolution of size 1, not one of size 3. This choice is not standard ([24], [57], [82]) and was therefore ommitted for simplicitly.

There are other architectures that are aimed at segmentation, such as Large Kernel Matters [24] Deeplab [25]–[27], CRF as RNN [83] and the Tiramisu net [84]. However, these models are somewhat more complicated to implement. Furthermore, the the performance gain reported in the papers these architectures were introduced in was not substantial $(2-5\%)$. Explaining how these architectures work is outside the scope of this project. The interested reader is, therefore, recommended to read their original papers where they are introduced [24]–[27].

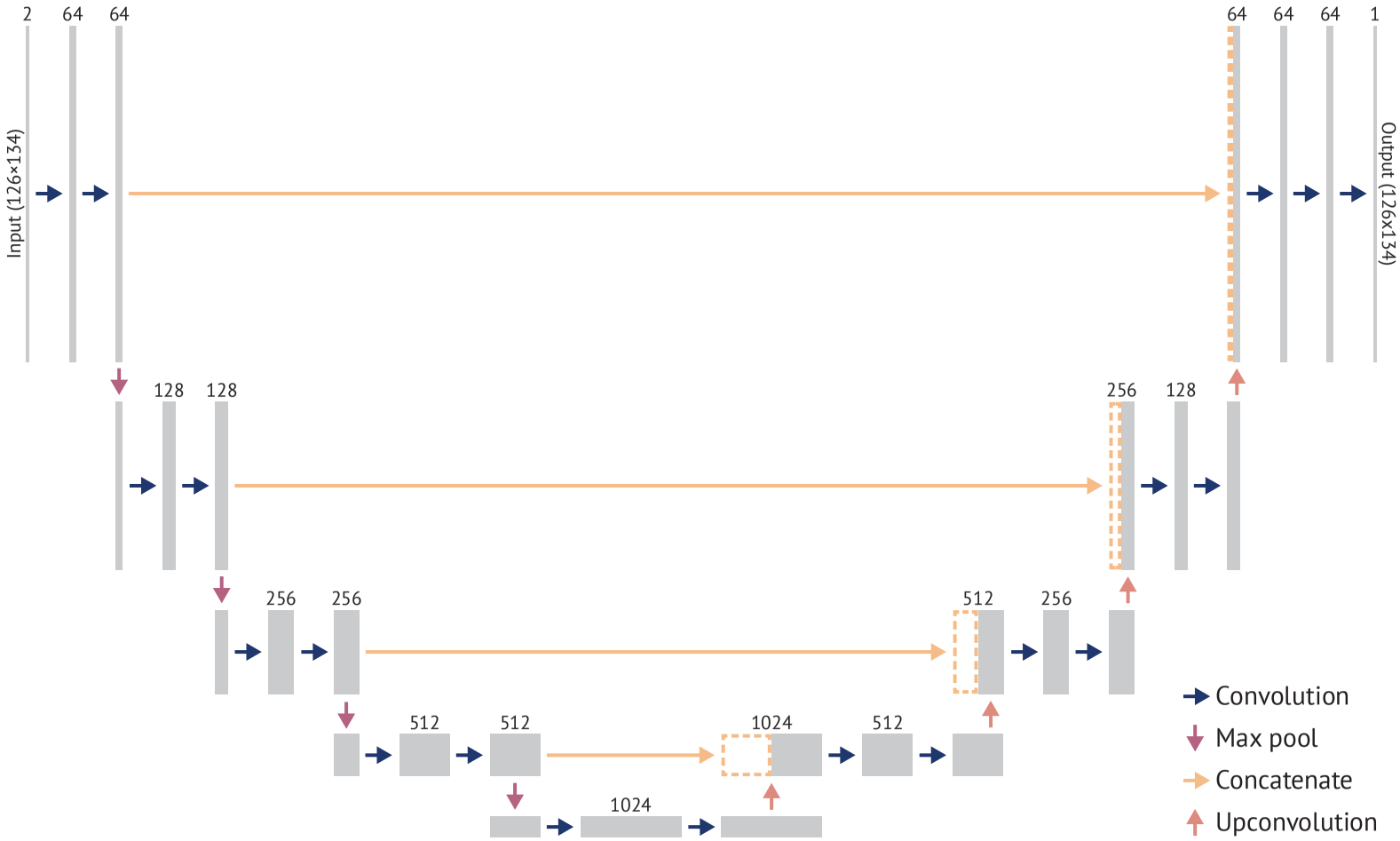


Figure 2.17: Illustration of the U-Net architecture [23]. The gray squares represent the input and output of layers, the height represents the spatial size and the length represents the number of channels. The blue arrows represent convolutional layers with kernel size 3, a ReLU nonlinearity and batch normalisation. The down-facing arrows represent a max pooling layer with pool size 2, the up-facing arrows represent a upconvolution with rate 2. The input to a max pooling layer is concatenated to the output of a upconvolution, which is represented as light yellow arrows and dashed yellow lines.

Chapter 3

Code

3.1 Code outline

User-friendliness was a big concern when creating the SciNets package provided by this project. As such, an object oriented interface is used and example scripts are supplied. The code follows the PEP-8 style-guide [85] (with the exception of having up to 88 characters per line) and are documented according to the Numpydoc specifications [86]. In addition, simple example scripts are supplied to easily get started.

There are two main benefits of using SciNets as compared with other low-threshold deep learning tools such as Keras [15]. SciNets is made for scientific purposes, which requires a focus on both reproducibility and the ability to efficiently test new layers and architectures. As a result, an extensive logging suite is provided, both for mid-training logging and result logging.

Furthermore, extendability was a concern when designing SciNets. For example, creating new layers are easily done by subclassing the `BaseLayer` and overloading its `_build_layer` function. Creating new architecture structures are also easily done by subclassing the `BaseModel` class and overloading its `build_model` function.

Finally, we note that the codebase uses the *TensorFlow* 1.12 framework [12], which is described below.

3.1.1 The TensorFlow framework

TensorFlow [12] is a deep learning framework in which all computations are performed in a two-step fashion. First, a *computation graph* is generated. Thereafter, this graph is used to compute the actual quantities we are interested in.

A computation graph is a Directed Acyclic Graph (DAG) in which every node represents a mathematical operation. Additionally there are certain "special" nodes that deal with the input/output (IO) stream of the graph. An edge starting in node i and ending in node j shows that the output of the operation that i represents is required as an input to the operation that j represents.

To see how we create a TensorFlow graph, consider the following example.

Example 3.1.1 (Simple TensorFlow code)**.**

```
1 import tensorflow as tf
2
3 A = tf.placeholder(tf.float32, shape=(2, 2))
4 x = tf.placeholder(tf.float32, shape=(2,))
5 y = tf.placeholder(tf.float32, shape=(2,))
6
7 z = x + y
8 w = tf.matmul(A, z)
```

In the above code several things happen. Firstly, we import the `tensorflow` module and name it `tf`. Afterwards, on lines 3-5 we create three placeholder nodes, `A`, `x` and `y`. A placeholder node is an IO node that represents data being sent into the computation graph.

After the `A`, `x` and `y` nodes are created, we create two additional nodes, `z` and `w`. The `z` node represents the sum of `x` and `y`, and the `w` node represents the matrix multiplication of `A` and `z`.

The computation graph created in this example can be seen in Figure 3.1.

It is important to note that none of the variables in the code above have any numerical values. As such, a computation graph does not have any value by itself.

We have now shown how to create a computation graph. Next we need to compute the values we are interested in. This is done by setting up a TensorFlow `Session` context and feeding in values using the specified IO mechanisms. Example 3.1.2

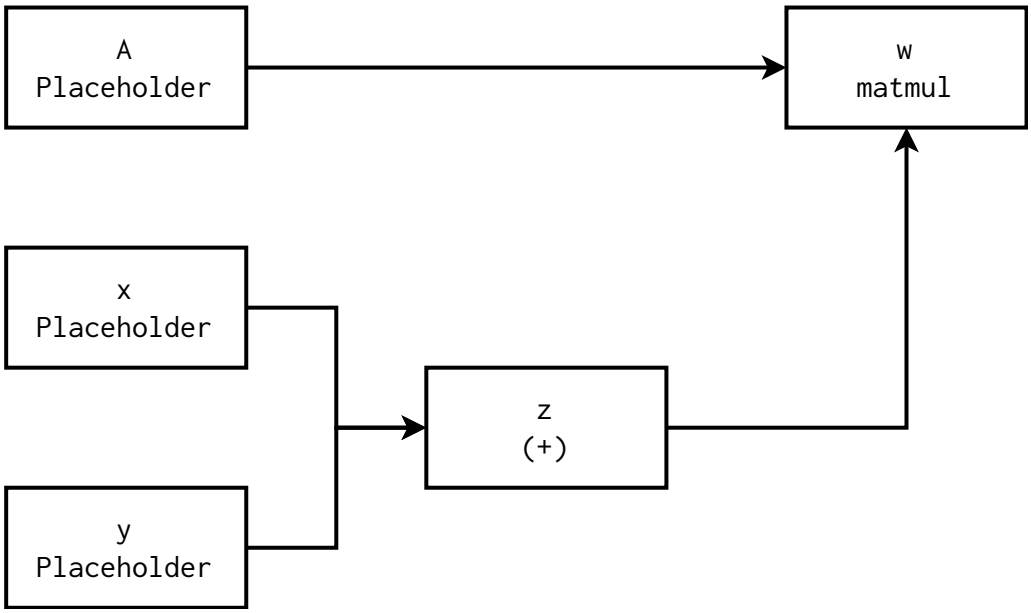


Figure 3.1: The computation graph created by the code in Example 3.1.1

demonstrates how we can use the code developed in Example 3.1.1 to compute the values of interest.

Example 3.1.2 (Performing computations with TensorFlow)**.**

```
import tensorflow as tf

A = tf.placeholder(tf.float32, shape=(2, 2))
x = tf.placeholder(tf.float32, shape=(2,))
y = tf.placeholder(tf.float32, shape=(2,))

z = x + y
w = tf.matmul(A, z)

with tf.Session() as sess:
    z1 = sess.run(z, feed_dict={x: [1, 2], y: [3, 4]}
    z2, w2 = sess.run((z, w), feed_dict={x: [1, 0], y: [0, 2],
                                         A: [[-1, 0], [0, 1]]}

print(f'The value of z1 is {z1}')
print(f'The value of z2 is {z2} and w2 is {w2}')
```

```
The value of z1 is [4, 6]
The value of z2 is [1, 2] and w2 is [-1 2]
```

In the above code, we see how the same computation graph can yield two different values, `z1` and `z2` depending on which values are fed into the graph. Furthermore, we see that we do not need to supply any values for `A` when we only want to compute the value of the `z` node, as `A` is not a parent of `z`.

Example 3.1.2 demonstrates that generating a computation graph is similar to defining functions. When we define a function, we create a "recipe" to compute something that varies with the function arguments. Similarly, when we create a computation graph, we generate a "recipe" for a variable that is dependent on the inputs of the graph.

There are several benefits gained by using a static computation graph the way TensorFlow does. By generating the computation graph before performing the computations, workload-planning is easier and optimisation procedures can be implemented [12].

Next, we note that there are several other special nodes in TensorFlow that we have not discussed. There are, for example, `Variables`, that have persistant values within one session, and special nodes that modify the value of a `Variable` [12].

The main benefit of using a computation graph, however, is that it makes autodifferentiation easier [12]. If all nodes in the graph have a method in which the partial derivatives with respect to the input is defined, then it is easy to compute the gradient of any node with respect to any parent node (using the chain rule). It is this benefit which makes TensorFlow suited for deep learning, as we only need to think about how to generate the model, not how to compute its gradient.

There are, however, also downsides with having a static computation graph. One particular downside is that the problem at hand needs to be thought of in a different way as compared to a "standard" programming approach. In general, we want reusable functions that create our neural networks, as such, we essentially create functions that create the functions we want to use. This added abstraction layer makes the development process more cumbersome at the same time as it complicates the debugging process.

There is one way to solve the afforementioned problem, namely dynamic computation graphs. Dynamic computation graphs are available in the latest version of TensorFlow and in the PyTorch framework [13]. Unfortunately, these tools did not have a mature documentation when this project was first envisioned, and were therefore not used.

3.1.2 The codebase

Before starting, we note that this thesis is written in British English. However, American English is the language used in the TensorFlow library which the presented code heavily relies on. As such, American English was chosen when writing the code to avoid inconsistent language within the code. There are, therefore, some inconsistensies in this chapter, especially with words ending with "-ise" in British English and "-ize" in American English. British English is used for all text written with a normal font, whereas American English is used for all text written in a monospaced font.

The codebase is available at https://github.com/yngvem/scinets/ and is structured into four Python modules: `data`, which is responsible for the dataset-pipline; `model`, which contains the layer and model classes; `trainer`, which provides a high-level training interface; and `utils` which contains general utility function and classes.

Apart from two classes (`Initializer` and `Optimizer`), all code presented herein was implemented as part of this project. Some classes (e.g. `Normalizer` and `Activation`) are small wrappers around TensorFlow functions, whereas other classes (e.g. `Model` and `Logger`) are not.

The `model` module

We start by discussing the `model` module and, in particular, the structure of the layer classes. All layer classes are created by subclassing the `BaseLayer` class, which contain several utility methods needed to create a layer. Additionally, all subclasses of the `BaseLayer` class are logged to a dictionary upon creation for reasons that will be apparent later.

Instead of highlighting every part of the `BaseLayer` class, we show how the `__init__` function is structured (see below). Afterwards, we demonstrate how to create a simple convolutional layer.

```
1 class BaseClass(ABC):
2     def __init__(
3         self,
4         x,
5         initializer=None,
6         regularizer=None,
7         activation=None,
```

```
8           normalizer=None,
9           is_training=None,
10          scope=None,
11          layer_params=None,
12          verbose=False,
13          *args,
14          **kwargs,
15      ):
16          if normalizer is not None and is_training is None:
17              raise ValueError(
18                  "You have to supply the `is_training` placeholder
                        for batch norm."
19              )
20          layer_params = layer_params if layer_params is not None else
                {}
21
22          self.input = x
23          self.is_training = is_training
24          self.scope = self._get_scope(scope)
25
26          self.initializer, self._init_str = self.
                _generate_initializer(initializer)
27          self.activation, self._act_str = self._generate_activation(
                activation)
28          self.regularizer, self._reg_str = self._generate_regularizer
                (regularizer)
29          self.normalizer, self._normalizer_str = self.
                _generate_normalizer(normalizer)
30
31          # Build layer
32          with tf.variable_scope(scope) as self.vscope:
33              self.output = self._build_layer(**layer_params)
34              self.params, self.reg_list = self._get_returns(self.
                    vscope)
35
36          if verbose:
37              self._print_info(layer_params)
```

The interesting part here lies in lines 26 through to 37. First, we create an initialiser instance, a regulariser instance, an activation function instance and a normaliser instance. We will focus on these parts later.

The key part of the `__init__` function is within the `tf.variable_scope` context[1]. Within this context, the correct TensorFlow nodes are set up by the `_build_layer` function. Afterwards, the trainable parameters and regularisation operators within

[1] A Python context is the block of code following a `with` statement.

the same variable scope are collected and stored in a dictionary and list, respectively.

Finally, if the verbosity level is nonzero (i.e. `verbose` $\neq 0$), then the `_print_info` function is called.

Let us now look at a way to implement a two-dimensional convolutional layer.

```
class Conv2D(BaseLayer):
    """A standard convolutional layer.
    """
    def _build_layer(
        self,
        out_size,
        k_size=3,
        use_bias=True,
        dilation_rate=1,
        strides=1,
        padding="SAME",
    ):
        out = tf.layers.conv2d(
            self.input,
            out_size,
            kernel_size=k_size,
            use_bias=use_bias,
            kernel_initializer=self.initializer,
            strides=strides,
            dilation_rate=dilation_rate,
            padding=padding,
            kernel_regularizer=self.regularizer,
        )
        out = self.activation(out)
        out = self.normalizer(out, training=self.is_training, name="
            BN")

        return out

    def _print_info(self, layer_params):
        print(
            "_________________Convolutional layer_________________\n",
            "Variable_scope: {}\n".format(self.vscope.name),
            "Kernel size: {}\n".format(layer_params.get("k_size", 3)
                ),
            "Output filters: {}\n".format(layer_params["out_size"]),
            "Strides: {}\n".format(layer_params.get("strides", 1)),
            "Dilation rate: {}\n".format(layer_params.get("
                dilation_rate", 1)),
            "Padding: {}\n".format(layer_params.get("padding", "SAME
                ")),
```

```
38              "Kernel initialisation: {}\n".format(self._init_str),
39              "Activation function: {}\n".format(self._act_str),
40              "Kernel regularisation: {}\n".format(self._reg_str),
41              "Number of regularizer loss: {}".format(len(self.
                    reg_list)),
42              "Use bias: {}\n".format(layer_params.get("use_bias",
                    True)),
43              "Normalization: {}\n".format(self._normalizer_str),
44              "Input shape: {}\n".format(self.input.get_shape().
                    as_list()),
45              "Output shape: {}".format(self.output.get_shape().
                    as_list()),
46          )
```

We see that the `_build_layer` function creates a TensorFlow `conv2d` layer, passes it through the activation function and normalises the output using the `self.normalizer` instance of the `BaseNormalizer` class. Generally, this will either represent an identity mapping or create the required TensorFlow nodes for a batch normalisation layer.

The normaliser, activation and regulariser classes have almost the same structure as the layer classes. There are three differences, firstly there is no `_print_info`. Secondly, the `_build_layer` is exchanged with a `build_normalizer`, `build_activation` and `_build_regularizer`, respectively. Finally, it is the `__call__` function that generates the TensorFlow nodes, not the `__init__` function.

The initialiser classes are Keras initialisers.

It is also important to note that the convolutional layer defined above is also stored in a `SubclassRegister` that is linked to the `BaseLayer` class. This allows us to define new layers without modifying the `scinets.model` module and without changing the structure of the configuration files. A thorough explanation of this is given in Section 3.1.2.

In general, all classes of the SciNets library are stored in `SubclassRegisters` with an associated *getter*. Thus, to create a regulariser instance, we can write

```
1 reg = models.get_regulariser('WeightDecay')()
```

Similarly, to create an optimiser instance we can write

```
1 optimizer = trainer.get_optimizer('ADAM')
```

Finally, we look at how the `BaseModel` class is structured. This is a complic-

Table 3.1: The inputs used to generate a `BaseModel` instance in SciNets

Argument	Description
`input_var`	TensorFlow node containing input.
`architecture`	List of dictionaries specifying the architecture.
`name`	Variable scope of the model.
`is_training`	TensorFlow placeholder specifying whether model is training or inferring new segmentation masks.
`true_out`	TensorFlow node containing the true output of the model.
`loss_function`	Dictionary specifying which loss function to use.
`verbose`	Boolean, whether or not to print out information in the terminal window

ated class, so we will not go in detail, but rather focus on the overarching design principles of the class.

In SciNets, a model is defined as the neural network and its loss function. This choice is made as the loss function heavily affects the functionality of a network. To illustrate why this is, we note that the only part that separates an object detection network from an image classification network is the choice of loss function and the training data [43]. As a consequence of which, SciNets models require several inputs and a complete list of these inputs is specified in Table 3.1 and an illustration of these inputs is shown in Figure 3.2.

When a SciNets model is created, the `_build_model` function is called. This function generates the neural network of the model. To do this, the architecture list is iterated through to assemble the network. Below is the `build_model` function of a normal feed-forward network.

```
def _build_model(self):
    """Assemble the network.
    """
    if self.verbose:
        print("\n" + 25 * "-" + "Assembling network" + 25 * "-")

    for layer in self.architecture:
        self._assemble_layer(layer, layer_input=self.out)

    if self.verbose:
        print(25 * "-" + "Finished assembling" + 25 * "-" + "\n")

```

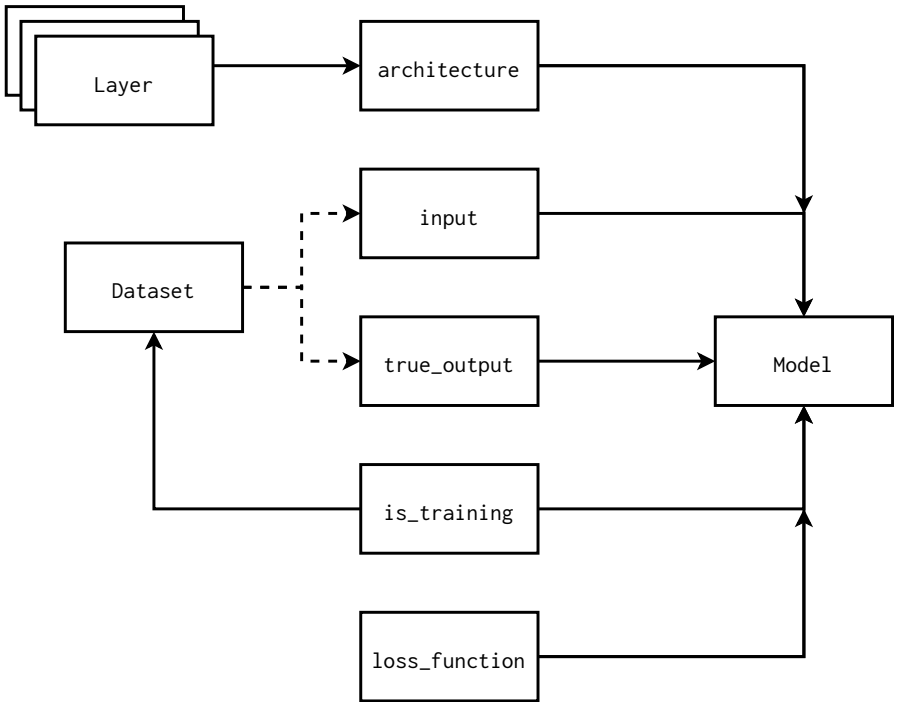


Figure 3.2: Flowchart showing the inputs and their dependencies to a SciNets model. The dashed arrows signifies that `Dataset` provides `input` and `true_output`, whereas the solid arrows signify that the starting node is an input of the ending node. The `Dataset` class will be described later.

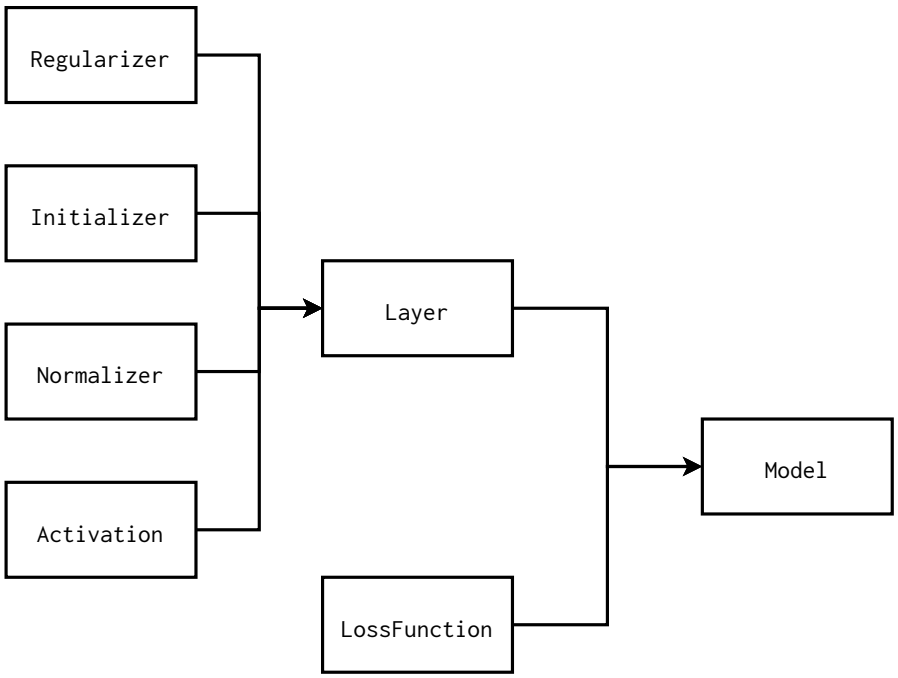


Figure 3.3: The class dependencies of the `Model` classes. The arrow that starts in the `LossFunction` node and ends in the `Model` node signals that an instance of the `Model` class contains an instance of the `LossFunction` class.

```
13      self.collect_regularizers()
```

Hence, the way a neural network is built is by calling the `_assemble_layer` function on all elements in the architecture list. Thus, to understand how a network is assembled, we need to inspect the `_assemble_layer` function.

```
1  def _assemble_layer(self, layer_dict, layer_input):
2      """Assemble the next layer.
3      """
4      layer_class = get_layer(layer_dict["layer"])
5      layer = layer_class(
6          layer_input,
7          is_training=self.is_training,
8          verbose=self.verbose,
9          **layer_dict
10     )
11
12     self.layers.append(layer)
13     self.out = layer.output
14
15     self.outs[layer_dict["scope"]] = self.out
16     self.reg_lists[layer_dict["scope"]] = layer.reg_list
17     for pname, param in layer.params.items():
18         self.params[layer_dict["scope"] + "/" + pname] = param
```

The code above does several things, let us therefore focus on the most important lines. Recall that all layers classes are added in a dictionary, the reason for this is so we can extract the class using a string. This is what line 4 does, it extracts the class with the name given by the `"layer"` key of the layer dictionary and stores this class in the `LayerClass` variable. Then, an instance of that class is created using the current network output as input to that layer. After which, the current network output is updated so it is equal to the output of the recently created layer.

The model building process can be summarised in Section 3.1.2.

In addition, the layer parameters are added to a dictionary to facilitate their logging. Similarly, the layer outputs are stored in a list. This is not only done for logging purposes, but also because the layer outputs are useful when interpreting a trained model. In addition, certain architectures require the outputs of earlier layers (e.g. U-Net [23]).

The TensorFlow nodes that correspond to the loss function is created after the neural network is created. The way loss functions are generated are in a similar fashion as to how regularisers are generated. Upon instance initialisation, the

Algorithm 3.1 How feed forward networks are generated in SciNets

```
1: procedure GenerateNeuralNetwork(Input, Architecture)
2:     The architecture is a list of layer specifications
3:     Output ← Input.
4:     for each layer_spec in Architecture do
5:         Layer = get_layer(layer_spec)
6:         Output ← Layer(Output)
7:     return Output
```

parameters of the loss function is set. Furthermore, the `__call__` method is overloaded to call the `_build_loss` function, which creates the TensorFlow nodes that represent the loss function. This function, and possibly the `__init__` function should be overloaded in loss function classes.

A flow-chart summarising the class dependencies in the `model` module is shown in Figure 3.3. Furthermore, an example where we create a SciNets model is given in Example 3.1.3

Example 3.1.3 (Creating a SciNets model.)**.**

```
import tensorflow as tf
import scinets.model


loss_function = ({"operator": "BinaryFBeta", "arguments": {"beta
    ": 2}},)
architecture = [
    {
        "layer": "Conv2D",
        "scope": "conv1",
        "layer_params": {"out_size": 8, "k_size": 3},
        "normalizer": {"operator": "BatchNormalization"},
        "activation": {"operator": "ReLU"},
        "initializer": {"operator": "he_normal"},
        "regularizer": {"operator": "WeightDecay", "arguments":
            {"amount": 1}},
    },
    {
        "layer": "Conv2D",
        "scope": "conv2",
        "layer_params": {"out_size": 8, "k_size": 3, "strides":
            4},
        "normalizer": {"operator": "BatchNormalization"},
        "activation": {"operator": "ReLU"},
        "initializer": {"operator": "he_normal"},
    },
    {
        "layer": "Conv2D",
        "scope": "conv3",
        "layer_params": {"out_size": 16, "k_size": 3},
        "normalizer": {"operator": "BatchNormalization"},
        "activation": {"operator": "ReLU"},
        "initializer": {"operator": "he_normal"},
    },
    {
        "layer": "LinearInterpolate",
        "scope": "linear_upsample",
        "layer_params": {"rate": 4},
    },
    {
        "layer": "Conv2D",
        "scope": "conv4",
        "layer_params": {"out_size": 32, "k_size": 3},
        "normalizer": {"operator": "BatchNormalization"},
        "activation": {"operator": "ReLU"},
        "initializer": {"operator": "he_normal"},
```

```
    },
    {
        "layer": "Conv2D",
        "scope": "conv5",
        "layer_params": {"out_size": 1, "k_size": 3},
        "normalizer": {"operator": "BatchNormalization"},
        "activation": {"operator": "Sigmoid"},
        "initializer": {"operator": "he_normal"},
    },
]
input = tf.placeholder(tf.float32, shape=(16, 256, 256, 2))
true_out = tf.placeholder(tf.float32, shape=(16, 256, 256, 1))
is_training = tf.placeholder(tf.bool, shape=(,))

model = scinets.model.NeuralNet(
    input=input,
    true_out=true_out,
    architecture=architecture,
    loss_function=loss_function,
    is_training=is_training,
)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())

    feed_dict = {input: INPUT_IMAGE, true_out: TRUE_MASK,
        is_training=False,}
    proposed_segmentation, loss = sess.run([model.out, model.
        loss], feed_dict=feed_dict)
```

Here we create a standard feed-forward neural network with no skip connections for image segmentation. The loss function is a F_2 style loss.

The `INPUT_IMAGE` is the PET/CT image we want to segment, and `TRUE_MASK` is the ground truth segmentation mask for that image. How we load these will be the focus of the next section.

Finally, we store the proposed segmentation mask and the loss in the associated variables.

The `data` module

The `data` module contains everything needed for data loading and preprocessing. It contains three base classes, one dataset class, one data reader class and one

preprocessing class. The full data loading pipeline is summarised in the flowcharts shown in Figure 3.4.

The `Reader` classes is responsible for loading data from disk, preprocessing it and feeding it to the TensorFlow graph. The way this is implemented is through a Python generator that iterates through the dataset in a random order and yields the preprocessed inputs and targets (the wanted outcome of the network). This generator is then fed into the TensorFlow `tf.nn.Dataset` class which stacks n outputs of the generator into a TensorFlow node [12].

The benefit of using the TensorFlow `tf.nn.Dataset` class are two-folds. Firstly, it offers a wrapper to Python generators, which allows the data to be lazily[2] loaded and preprocessed. Secondly, it allows for parallel *prefetching*. That is, loading and preprocessing a batch while the network is training with the current batch.

During the development phase of this module, it was found that prefetching a single batch yielded the same performance as keeping the entire dataset in memory. As a consequence, we gain optimal data throughput without sacrificing memory.

A key part of the data reader classes is the preprocessing of the input images and segmentation masks, which is accomplished by the `Preprocessing` classes. Every preprocessor has four methods, `__init__`, which sets the preprocessor parameters; `__call__`, which takes two numpy arrays, `image` and `target`, as input and returns a preprocessed version of them; `output_channels`, which returns the number of image channels after preprocessing; and `output_targets`, which returns the number of segmentation masks after preprocessing.

The usefulness of the `__init__` and `__call__` methods of the preprocessors is clear. They are the functions that set the preprocessor parameters and perform the preprocessing. The usefulness of `output_channels` and `output_targets`, on the other hand, is less clear. These functions have to be implemented since the TensorFlow IO nodes need to know the dimensions of the inputs. Thus, the results of these functions are fed into the TensorFlow `Dataset` constructor.

Finally, we introduce the main component of the `data` module, the scinets `tf.nn.Dataset` class. This class generates three instances of the `Reader` class, one for the training dataset, one for the validation dataset and one for the testing dataset. The TensorFlow IO nodes provided by the data readers are fed through conditional TensorFlow nodes to yield a single output node. These boolean TensorFlow nodes

[2] *Lazy* evaluation is a programming technique in which an iterable data structure generates its elements when they are requested. For more information, see the Python documentation for generators.

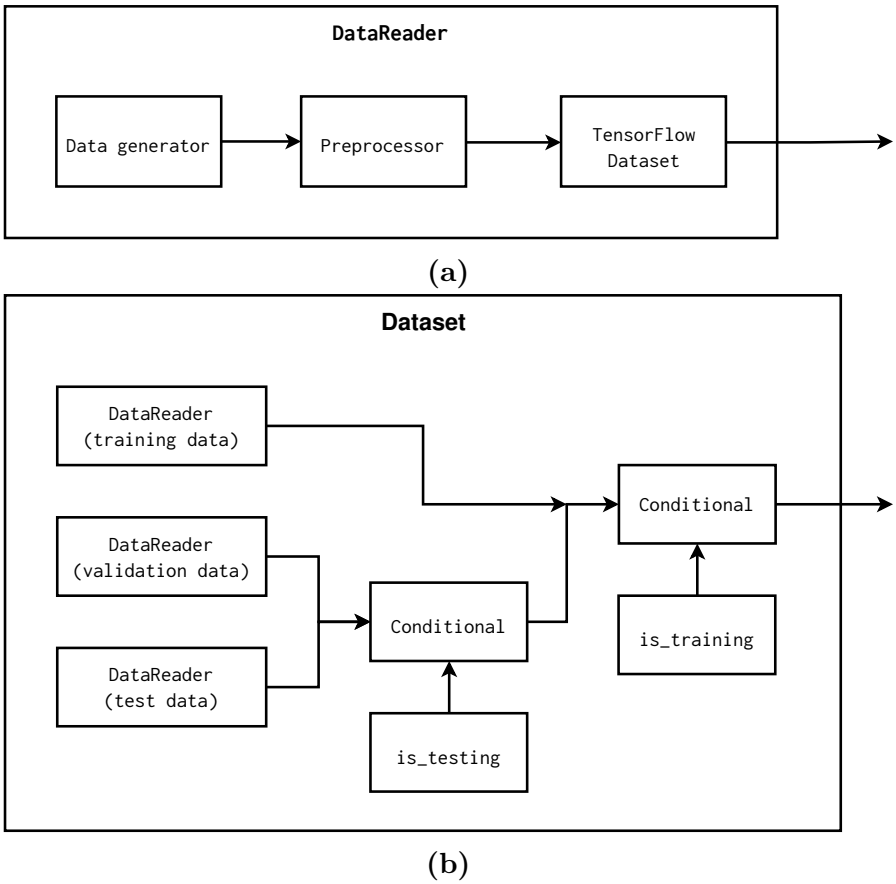


Figure 3.4: Flowcharts demonstrating how the data loading pipeline works. The arrows leaving the large boxes represent the contents of the TensorFlow node provided by the classes. (a) shows how the data reader class works; the data generator (lazily) loads single datapoints from the dataset in a random order. The dataset is then passed through the preprocessor as numpy arrays before being fed into the TensorFlow `Dataset`. (b) shows how the outputs of the data loader are passed through two conditional TensorFlow nodes, that depend on boolean TensorFlow placeholder nodes to discern which dataset to load data from. Thus, the single output node of the dataset class is the only node required as input to the neural network. Thus, the datapoints are only loaded from disk when they are required, which reduces the memory footprint.

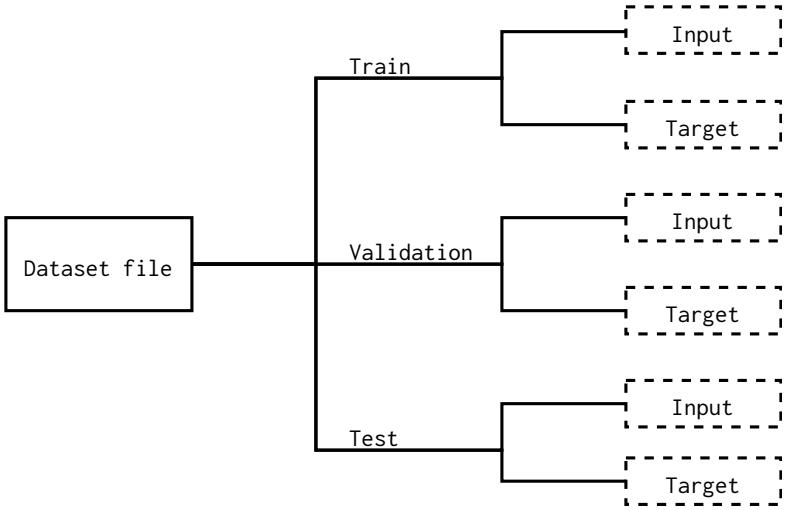


Figure 3.5: The structure of the dataset files. The dashed boxes are the HDF5 datasets, the forks with text overhead denotes HDF5 groups.

dependend on two TensorFlow `placeholder`s, `is_training` and `is_testing`, that specify which dataset to use.

There is one kind of data reader provided in SciNets; `HDFReader`. The `HDFReader` loads data on the HDF5 format, which is described in Section 3.2.1. For the dataset files to be supported by the `HDFReader`, it needs to be in a specific format. In particular, the inputs and targets of the model should be stored in separate HDF5 datasets[3] but in the same group. The first axis of the datasets represent the different datapoints (i.e. first element in the training dataset corresponds to the first element of the targets dataset and so forth).

Furthermore, a `HDFDataset` class is provided which wraps the `HDFReader`. For this class to work, the training, validation and testing sets must be in the same HDF5 file, but in separate groups. The dataset instance will then lazily load the datapoints from the HDF5 file as requested by the network. Figure 3.5 shows an illustration of how the HDF5 files should be structured for the `HDFDataset` class to work. Additionally, Example 3.1.4 shows how the `HDFDataset` class can be used with a SciNets model.

[3]In HDF5 files, a dataset is a multidimensional array stored to disk [87].

Example 3.1.4 (SciNets `HDFDataset` and model.)**.**

```
import tensorflow as tf
import scinets.model
import scinets.data


loss_function = ({"operator": "BinaryFBeta", "arguments": {"beta
    ": 2}},)
architecture = [
    ...
]

preprocessor = {
    "operator": "WindowingPreprocessor",
    "arguments": {
        "window_width": 100,
        "window_centers": 1070,
    }
}


is_training = tf.placeholder(tf.bool, shape=(,))
is_testing = tf.placeholder(tf.bool, shape=(,))

train_batch_size = 8
val_batch_size = 16
test_batch_size = 16

dataset = scinets.data.HDFDataset(
    data_path="/data/data_2d.h5"
    batch_size=[train_batch_size, val_batch_size,
       test_batch_size,]
    train_group="train"
    val_group="val"
    test_group="test",
    preprocessor=preprocessor,
    is_training=is_training,
    is_testing=is_testing
)
model = scinets.model.NeuralNet(
    input=input,
    true_out=true_out,
    architecture=architecture,
    loss_function=loss_function,
    is_training=is_training,
)

```

```
45 with tf.Session() as sess:
46     sess.run([tf.global_variables_initializer(), dataset.
           initializers])
47
48     feed_dict = {is_training=False, is_testing=False,}
49     proposed_segmentations, losses = sess.run([model.out, model.
           loss], feed_dict=feed_dict)
```

Here we create a `HDF5Dataset` using an HDF file with path "/data/data_2d.h5". Each training batch contain 8s samples, whereas the validation batch and testing batch contain 16 samples each.

The group with the training data is called "train", the group with the validation data is called "val" and the group with the test data is called "test".

For preprocessing, we use a `WindowingPreprocessor`, which simply reduces the dynamic range of the image.

Finally, we use this model as input to a feed forward network with the same architecture as in Example 3.1.3 and store the proposed segmentations and losses for 16 random validation images.

Also, note that the dataset class provides an initializers attribute, which must be run to activate the TensorFlow dataset nodes

The `trainer` module

The `trainer` module is also a relatively small module. It contains three kinds of classes, optimisers, learning rate schedulers and the network trainer. The optimisers provide an interface for updating the weights of the network, the learning rate schedulers enable a learning rate that varies with iteration number and the trainer class provides a high-level interface to train a SciNets model.

The learning rate schedulers generate the TensorFlow node representing the learning rate. This is done by providing the schedulers with a TensorFlow node that keeps track of the number of training steps that has been performed as well as the number of training steps per epoch. This allows us to have a learning rate that varies with the current iteration number. Thereafter, the `build_lr_scheduler` method is called, which creates and returns a TensorFlow node representing the current learning rate.

The optimisers are subclasses of the TensorFlow `tensorflow.train.Optimizer` class. See the TensorFlow documentation for more information about this class [88].

Finally, the `NetworkTrainer` class provide an easy method to train SciNets models. Upon initiation of this class, several inputs are provided, a list of which is seen in Table 3.2. After initialisation, we call the `train` method of the `NetworkTrainer` to train the model. The number of training steps to perform and an active TensorFlow session should be given as arguments to this function. Alternatively, it is possible to call the `train_step` method to perform a single training step at a time (which, amongst others is useful for debugging purposes). An example of the `NetworkTrainer` in action can be found in Example 3.1.5.

Finally, we note that the `train` and `train_step` methods have an optional argument, `additional_ops`. The input to this argument should be a list of additional TensorFlow operators to run during the training iterations. The output of these nodes are the return values of these methods. This option is added so we, for example, can perform logging during training.

Table 3.2: The inputs to the `scinets.trainer.NetworkTrainer` class.

Input argument	Description
`model`	The model that should be trained.
`steps_per_epoch`	The number of training steps per epoch.
`log_dir`	Directory to store checkpoints in
`train_op`	Dictionary parametrising the training operator.
`learning_rate_scheduler`	Dictionary parametrising the learning rate scheduler.
`max_checkpoints`	Maximum number of checkpoints to store.
`save_step`	The number of training steps to perform between each checkpoint.
`verbose`	The verbosity level.

The parametrising dictionaries is on the form {`"operator": name, "arguments": kwargs`}, where `name` is a string containing the name of the operator to use and `kwargs` is an optional keyword argument dictionary that is passed to the constructor.

Example 3.1.5 (Training a SciNets model)**.**

```
1 import tensorflow as tf
```

```
import scinets.model
import scinets.data
import scinets.trainer


loss_function = ({"operator": "BinaryFBeta", "arguments": {"beta
    ": 2}},)
architecture = [
    ...
]

preprocessor = {
    ...
}


is_training = tf.placeholder(tf.bool, shape=(,))
is_testing = tf.placeholder(tf.bool, shape=(,))

train_batch_size = 8
val_batch_size = 16
test_batch_size = 16

dataset = scinets.data.HDFDataset(
    ...
)
model = scinets.model.NeuralNet(
    ...
)

steps_per_epoch = len(dataset)//dataset.batch_size[0]
train_op = {"operator": "Adam"}
learning_rate_scheduler = {
    "operator": PolynomialDecay,
    "arguments": {"decay_steps": 10000, "power": 1,}
}

trainer = scinets.trainer.NetworkTrainer(
    model=model,
    log_dir="./logs",
    steps_per_epoch=steps_per_epoch,
    train_op=train_op,
    learning_rate_scheduler=learning_rate_scheduler,
    max_checkpoints=5,
    save_steps=2000,
    verbose=True,
)

```

```
49 with tf.Session() as sess:
50     sess.run([tf.global_variables_initializer(), dataset.
           initializers])
51
52     trainer.train(sess, num_steps=1000,)
53
54     feed_dict = {is_training=False, is_testing=False,}
55     proposed_segmentations, losses = sess.run([model.out, model.
           loss], feed_dict=feed_dict,)
```

The code above creates a neural network using a dataset in the same method as in Example 3.1.4. This time, however, we also create a network trainer, using the ADAM optimizer and a linear learning rate scheduler. The network is then trained for 10000 iterations, storing the network weights every 2000 iteration. Finally, the segmentation masks and losses for 16 random validation images are computed (with the trained model) and stored in the `proposed_segmentation` and `losses` variables.

The `utils` module

The `utils` module contains several utility classes separated over three submodules: the `evaluator` module, the `logger` module and the `experiment` module. The evaluator module contains two kinds of classes, evaluators and network testers. The former creates TensorFlow nodes that compute performance metrics of the network (such as accuracy, sensitivity and specificity). The network testers, on the other hand, are responsible for iterating through the whole dataset and computing the average performance metrics.

There are two kinds of evaluator supplied with SciNets, `ClassificationEvaluator` and `BinaryClassificationEvaluator`. These evaluators provide performance metrics specific for classification problems in general and binary classification problems. It is, however, not difficult to create an evaluator class, as its only property is having TensorFlow nodes as attributes. Thus, to create a new evaluator, one only needs to subclass the `BaseEvaluator` and add attributes for the quantities of interest.

In addition to the `evaluator` submodule, we have the `logger` submodule. In this module, we provide an easy-to-use interface to track the model performance during training.

To create a logger instance, it should be provided with an evaluator instance.

It will then create a TensorFlow node (or a collection of nodes in an iterable) responsible for creating "entries" in the training log and validation log, whose names are `train_log_op` and `val_log_op`, respectively.

In addition, the logger classes need to store log entries after they have been computed. There are two methods that do this, the `log` method and the `log_multiple` method. The former takes a single log entry as well as an integer that represents which iteration that entry corresponds to as well as a string that signifies whether it is a validation log or a training log, before adding that entry to its log. Similarly, the `log_multiple` method takes a list of entries and a list of iteration numbers and logs all of them in the correct log. Before these methods can be used, however, the `init_logging`, which does the final logger preparation (e.g. by storing the current session in an instance attribute) method must be called.

Thus, to log the performance during training, we first need to create an evaluator, then feed this evaluator into a logger. Thereafter, we start the TensorFlow session and initiate the loggers, using their `init_logging` method. Then, during training, we supply the `train_log_op`-operators so they are computed simultaneously as the variable update operator is called. Finally, we supply the results of the `train_log_op`-operators to the `log_multiple` method of the logger.

There are three loggers provided in the SciNets package; a `TensorboardLogger`, an `HDF5Logger` and a `SacredLogger`. The `TensorboardLogger` creates logs that are compatible with TensorBoard. TensorBoard is a dashboard created by Google to accompany the TensorFlow package [12]. It makes it easy to follow the training procedure as it is occuring, and can create a plethora of diagnostic graphs. Figures 3.6, 3.7, 3.8 and 3.9 show examples of different dashboards that can be created with TensorBoard.

Similarly, the `SacredLogger` creates logs that are compatible with the Sacred package [89]. Sacred is a general-purpose tool for logging of machine learning experiments and a particularly useful part of Sacred is its ability to connect with a database of already run experiments. Thus, by using the `SacredLogger`, we are able to collect a database of previously run experiments and compare their results. Keeping a database of the previously run experiments ensures reproducability. In addition, the SacredBoard library [89] allows us to host a webpage with a dashboard to easily interact with the database. Thus, by using the Sacred and SacredBoard libraries, we can compare the performance of the different models easily. Using the Sacred library involves some overhead, we therefore recommend to read the source code for the *run_sacred.py* command line interface program to understand how to use this logger.

Finally, we have the HDF5 logger. This logger simply creates a HDF5 file with the corresponding logs stored in the train group and validation group, respectively. Example 3.1.6 shows how to use the `TensorBoardLogger`.

Example 3.1.6 (Using the `TensorboardLogger` class)**.**

```

import tensorflow as tf
import scinets.model
import scinets.data
import scinets.trainer


loss_function = ({"operator": "BinaryFBeta", "arguments": {"beta
    ": 2}},)
architecture = [
    ...
]

preprocessor = {
    ...
}


is_training = tf.placeholder(tf.bool, shape=(,))
is_testing = tf.placeholder(tf.bool, shape=(,))

train_batch_size = 8
val_batch_size = 16
test_batch_size = 16

dataset = scinets.data.HDFDataset(
    ...
)
model = scinets.model.NeuralNet(
    ...
)

steps_per_epoch = len(dataset)/dataset.batch_size[0]
train_op = {"operator": "Adam"}
learning_rate_scheduler = {
    ...
}

trainer = scinets.trainer.NetworkTrainer(
    ...
)
evaluator = scinets.utils.Evaluator(model)
```

```
logger = scinets.utils.TensorboardLogger(
    evaluator,
    log_dir="./logs",
    log_dicts=[
        {
            "log_name": "Loss",
            "log_var": "loss",
            "log_type": "scalar",
        },
        {
            "log_name": "Dice",
            "log_var": "dice",
            "log_type": "scalar",
        },
        {
            "log_name": "PET",
            "log_var": "input",
            "log_type": "image",
            "log_kwargs": {"max_outputs": 1, "channel": 0}
        },
        {
            "log_name": "Mask",
            "log_var": "true_out",
            "log_type": "image",
            "log_kwargs": {"max_outputs":1}
        },
        {
            "log_name": "Probability_map",
            "log_var":"probabilities",
            "log_type": "image",
            "log_kwargs": {"max_outputs":1}
        },
    ]
)


with tf.Session() as sess:
    sess.run([tf.global_variables_initializer(), dataset.
        initializers])

    for i in range(10):
        train_summaries, it_nums = trainer.train(sess, num_steps
            =100, additional_ops=[logger.train_summary_op])
        logger.log_multiple(train_summaries, it_nums, log_type="
            train")
        val_summaries = sess.run(self.logger.val_summary_op)
        logger.log(val_summaries, it_nums[-1], log_type="val")

```

```
87      feed_dict = {is_training=False, is_testing=False,}
88      proposed_segmentations, losses = sess.run([model.out, model.
            loss], feed_dict=feed_dict,)
```

In this example, we see how to use the Tensorboard logger can be used to easily create a dashboard like the one shown in Figures 3.6 and 3.7. The `BinaryClassificationEvaluator` instance provides performance metrics for the neural network that can be logged by the `TensorboardLogger` instance. The logs are stored in the directory specified by the `log_dir` argument of the `TensorboardLogger`.

This example creates four logs, two line plots and two images. The `log_name` key of the a log dictionary represents the name of the TensorFlow node that computes the Tensorboard logs. The `log_var` key is which attribute of the `BinaryClassificationEvaluator` to log. Finally, the `log_type` and `log_kwargs` keys parametrise the log operator to create.

The last submodule in the `utils` module is the `experiment` module. This module contains the `NetworkExperiment` class, which wraps all classes mentioned above into an easy-to-use interface. The input to this class is a set of dictionaries that parametrise the dataset, model, trainer, evaluator and logger. An illustration of which dictionaries parametrise which part of a network experiment is shown in Figure 3.11 Thus, all parts in the training and evaluation pipeline are taken care of and best practices are enforced.

Several functionalities are available once `NetworkExperiment` is initiated[4]. First, and foremost, a `train` method is provided, which takes care of data loading, training and validation. As a consequence, one needs not worry about training or validating on the incorrect dataset.

Another method of interest is the `evaluate_model` method. This method computes the final evaluation metrics for all data points in the specified dataset and returns them as a dictionary (whose keys are evaluation metrics and values are a list containing the results for each data point).

An important use-case of the `evaluate_model` method is to find the best performing model amongst all checkpointed models. This is exactly what the `find_best_model` method does. It takes which evaluation metric to use as input, whether it is the mean or the median of it that should be used and which dataset to evalute on. If the test set is chosen, then the user will be warned twice before finding the best

[4]The format of the input dictionaries are described in Appendix A.

dataset. This is to prevent evaluating on the test set without being 100% certain that it is our goal.

Finally, we have the `store_outputs` method. This method computes the evaluation metrics and input-output pairs of all data points in the specified dataset and stores it in a HDF5 file, making post-training evaluation such as plotting histograms of performance metrics easy. For a full example that demonstrates how to use the `NetworkExperiment` class, see Example 3.1.7

Example 3.1.7 (Using the `NetworkExperiment` class).

```
from scinets.utils import NetworkExperiment
experiment_params = {
...
}
dataset_params = {
...
}
log_params = {
...
}
model_params = {
...
}
trainer_params = {
...
}

experiment = NetworkExperiment(
    experiment_params=experiment_params,
    model_params=model_params,
    dataset_params=dataset_params,
    trainer_params=trainer_params,
    log_params=log_params,
)

num_steps = 10000
experiment.train(num_steps)  # Train for 10 000 steps

best_it, result, result_std = experiment.find_best_model("val",
    "dice")
print(" Final score ")
print(
    f" Achieved a Dice of {result:.3f}, with a standard "
    f"deviation of {result_std:.3f}"
)
print(f" This result was achieved at iteration {best_it}")
```

This code will train a model according to the parameters. An overview of the parameter structure is given in Appendix A. After which, the network will iterate through all checkpointed models and compute the average Dice for all of them and print the best value. Example of a possible printout is given below:

```
Final score
Achieved a Dice of 0.567 with a standard deviation of 0.281
This result was achieved at iteration 4000
```

Figure 3.6: Screenshot of some automatic diagnostic line plots created by the `TensorboardLogger`.

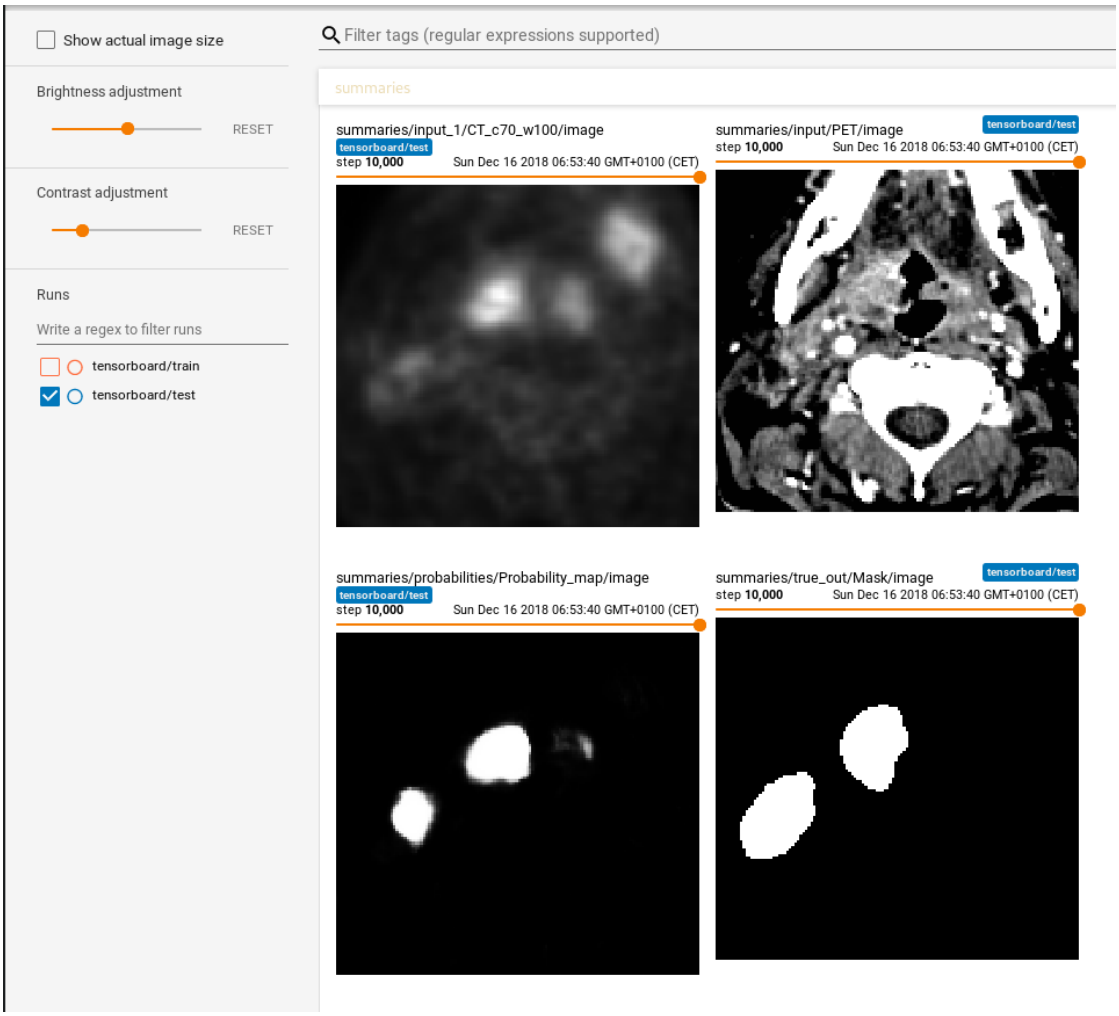


Figure 3.7: Screenshot of some automatic diagnostic image illustrations created by the `TensorboardLogger`. The images show the PET channel, CT channel, predicted mask and true mask.

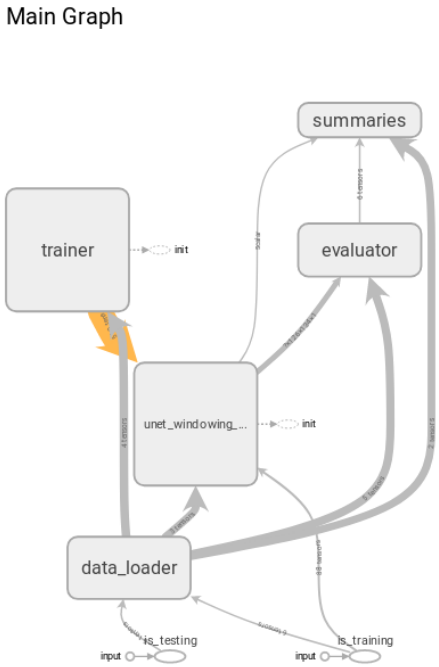


Figure 3.8: Screenshot of an automatically generated TensorFlow computation graph visualisation.

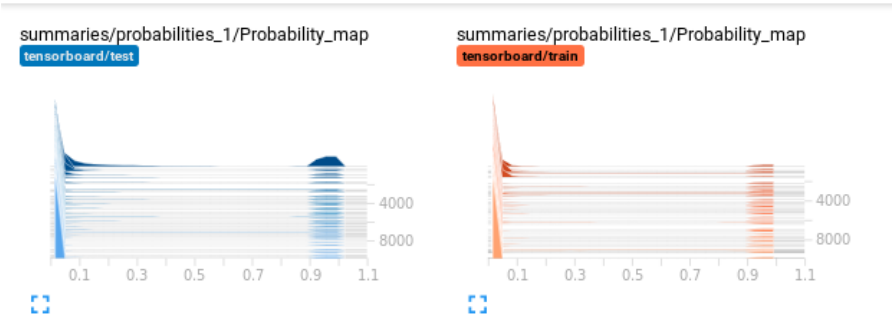


Figure 3.9: Screenshot of some automatic histograms created by the `TensorboardLogger`. These histograms show the distribution of predicted class labels for pixels.

	Id	Experiment name	Command	Start time	Last activity	Hostname	Result
+	163	unet_multiple_windows_basic_f2_adam	sacred_main	02:19:31 PM 12/12/2018	04:24:33 PM 12/12/2018	df90565ecf7f	0.6166774034500122
+	162	unet_no_pet_basic_f4_adam	sacred_main	01:32:55 PM 12/12/2018	02:45:35 PM 12/12/2018	bd8b4ffbda70	0.5235758423805237
+	161	unet_no_pet_basic_f2_adam	sacred_main	01:26:31 PM 12/12/2018	02:38:44 PM 12/12/2018	5fad69ae6f1c	0.5305169224739075
+	160	unet_multiple_windows_basic_cross_entropy_adam	sacred_main	12:24:27 PM 12/12/2018	02:27:31 PM 12/12/2018	84b21a1b42a6	0.5831844806671143
+	158	unet_multiple_windows_basic_f1_adam	sacred_main	12:14:51 PM 12/12/2018	02:19:28 PM 12/12/2018	df90565ecf7f	0.5841631293296814

(a)

	Id	Experiment name	Command	Start time	Last activity	Hostname	Result
-	163	unet_multiple_windows_basic_f2_adam	sacred_main	02:19:31 PM 12/12/2018	04:24:33 PM 12/12/2018	df90565ecf7f	0.6166774034500122

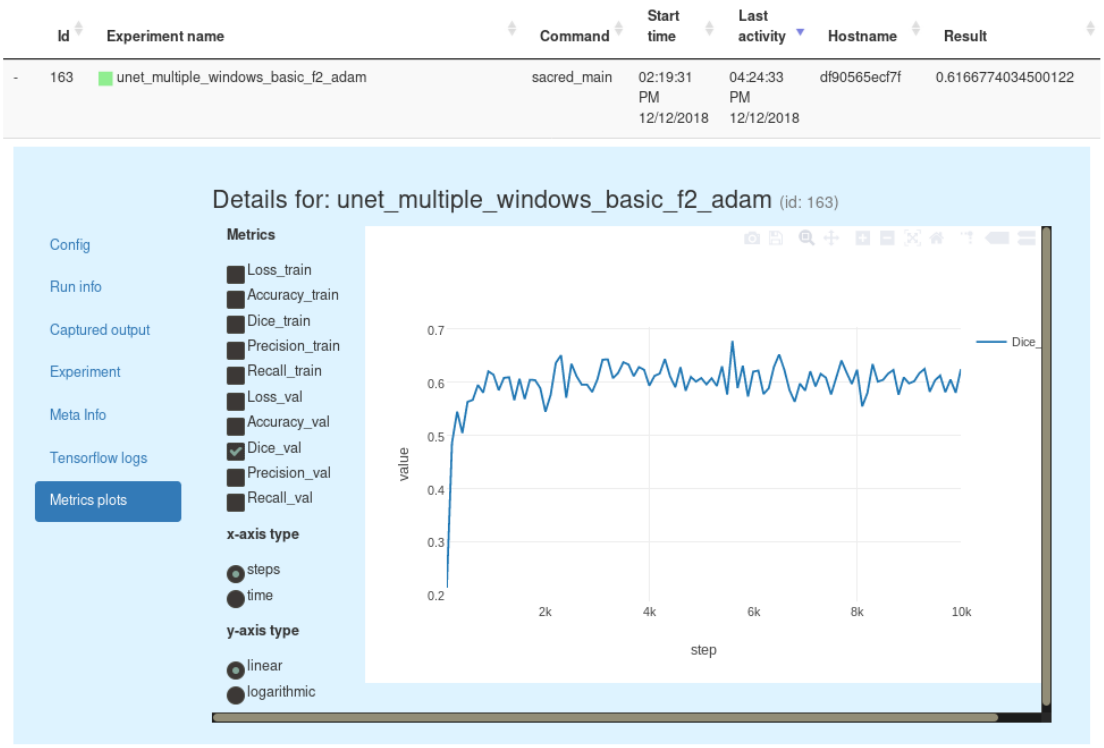
Details for: unet_multiple_windows_basic_f2_adam (id: 163)

Config
Run info
Captured output
Experiment
Meta Info
Tensorflow logs
Metrics plots

Run configuration

dataset_params		{...}
experiment_params		{...}
log_params		{...}
model_params		{...}
seed		350957780
trainer_params		{...}
train_op		{...}
arguments	{...}	
learning_rate		0.0001
operator	AdamOptimizer	

(b)



(c)

Figure 3.10: Three screenshots from the dashboard automatically created by the sacred logger. (a) shows the dashboard overview with the final evaluation metric (in this case Dice) in the results column, making it easier to compare models. (b) shows the view when one of the models are expanded, showing that all model parameters are logged with the experiment. (c) shows the logged performance metrics, which updates real-time during training.

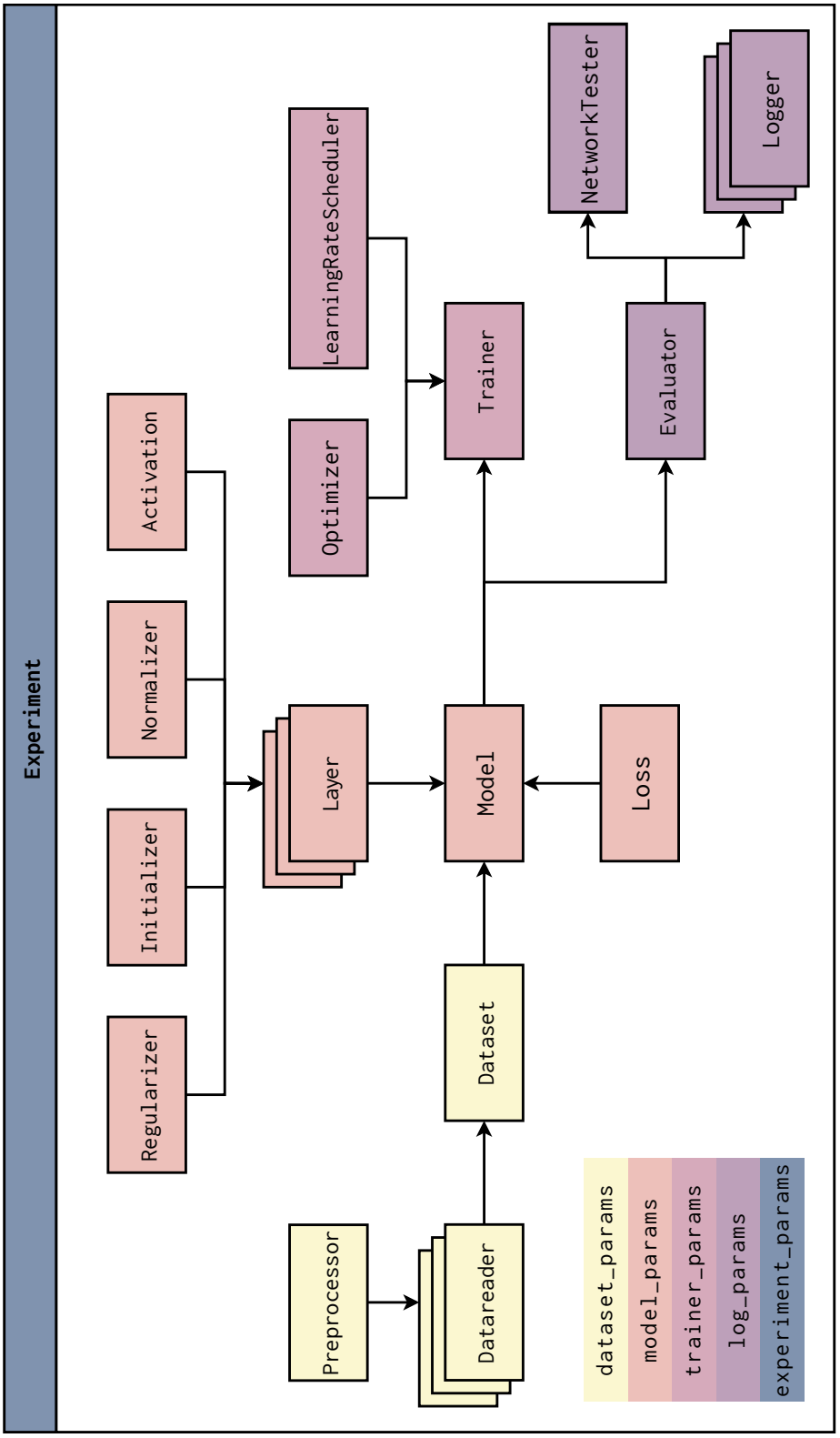


Figure 3.11: Flowchart illustrating the components of a `NetworkExperiment` instance. Each component is colour coded after the input-dictionary that parametrises it.

The subclass register

One key component of the SciNets library is the `SubclassRegister` utility class. This class is the backbone of how the different `get_` functions (e.g. `get_layer`) work, and is therefore integral to creating layers, regularisers, etc. from strings. To demonstrate how it works, consider Example 3.1.8

Example 3.1.8 (The subclass register)**.**

```
1  from scinets._backend_utils import SubclassRegister
2
3  car_register = SubclassRegister('car')
4
5  @car_register.link_base
6  class BaseCar:
7      def honk(self):
8          print('HONK')
9
10 class Sedan(BaseCar):
11     def __init__(self, num_seats):
12         self.num_seats = num_seats
13
14 class SUV(BaseCar):
15     def __init__(self, num_seats, trunk_size):
16         self.num_seats = num_seats
17         self.trunk_size = trunk_size
18
19
20 print(car_register.available_classes)
21 suv_instance = car_register["SUV"](4, 20)
22 print(type(first_sub_instance))
23 sedan_instance = car_register["sedan"](5)
24 print(type(sedan_instance))
```

```
('Sedan', 'SUV')
<class '__main__.SUV'>
------------------------------------------------------------
IndexError                Traceback (most recent call last)
...
IndexError: sedan is not a valid name for a car.
Available layers are (in decreasing similarity):
   * Sedan
   * SUV
```

In the code above, we demonstrate some of the components of a subclass register. Firstly, in line 3, we create a subclass register that should register

car classes (this name is only used for error messages). Then, we use the `link_base` class decorator when we create the base class for all cars. This decorator ensures that all subclasses of `BaseCar` are placed in the subclass register upon creation. Then, we create two car classes, `Sedan` and `SUV`.

Finally, we use the subclass register to create class instances. Firstly, we print the available classes, that is the name of all subclasses of `BaseCar`. Thereafter, we create a `SUV` instance, by indexing the subclass register for the class with name `"SUV"`. Finally, we try to create a `Sedan` instance, however, we forgot to capitalise the "S", which throws an index error. In the traceback of this exception, the available classes are listed, sorted by similarity with the query string.

The subclass register provides several essential parts of the SciNets package. Firstly, it makes it possible to specify which class to use in a string-format, whilst still making the library extendible. One option is to use the `getattr` builtin in Python. However, this would require us to change the SciNets library itself every time we want to add a new type of layer. By doing it this way, that is unnecessary, as will be demonstrated in Extending SciNets section on page 102. Furthermore, by listing the available subclasses in sorted order when a misspelling is used, we make it easy for the users to detect typos in their configuration files. The `SubclassRegister` class is, in other words an essential part of SciNets.

Command line interfaces

Several command line interfaces, or CLIs, are supplied with the SciNets library. We will now introduce two, the rest are presented in Appendix B. The first CLI, run_sacred, takes three arguments (database_credentials, experiment and num_steps) and two optional arguments (--eval and --name). An explanation of the arguments provided to this class is listed in Table 3.3.

The run_sacred program starts a connection with a MongoDB database in which the Sacred logs are stored before starting a SciNets experiment which is trained for num_steps training steps. When the SciNets experiment is finished training, all checkpoints are iterated through, finding which checkpoint yielded best performance on validation dataset with respect to the metric supplied with the --eval argument. This checkpoint is logged, and the evaluation metric is stored as the experiment result, making it easy to find in the Sacred dashboard. Example files for database credential files and experiment folders are supplied with the SciNets library.

Table 3.3: A short description of the arguments of the run_sacred CLI.

Argument	Description
database_credentials	Path to a YAML file contianing the database credentials.
experiment	Path to a folder containing the experiment files.
num_steps	The number of training steps to perform
--eval	The evaluation metric to use when finding the best checkpoint
--name	The name of the experiment (used in logs, etc.)

The experiment folder should contain the files "experiment_params.json", "dataset_params.json", "model_params.json", "trainer_params.json" and "log_params.json".

The second CLI provided by the SciNets library is the store_outputs program. This program takes two arguments (experiment, model_version and eval_metric) and three optional arguments (--storefile, --stepnum and --skip_summary). An explanation of these arguments are given in Table 3.4.

The store_outputs program creates an `NetworkExperiment` using the information in the experiment folder. The model_version is used to find the correct log/checkpoint folder. Thereafter, the program goes through all checkpoints and finds the optimal weights with respect to the specified evaluation metric. This step is skipped if the --stepnum argument is provided. Next, the program prints a summary of all evaluation metrics at the specified checkpoint in the terminal window, unless the --skip_summary argument is set to True. Finally, if the --storefile argument is provided, then all evaluation metrics for all images as well as all input-output pairs of the model are computed and stored in a HDF5 file.

The model_version argument is neccessary since the same experiment can be ran twice with the same name and the same log-folder. To prevent the logs from overwriting each other, the number of times the same experiment has been run previously is appended to the log-folder name (i.e. the first time an experiment is run, it gets the name ExperimentName00, the second time, it gets the name ExperimentName01 and so on).

Table 3.4: A short description of the arguments of the store_outputs CLI.

Argument	Description
experiment	Path to a folder containing the experiment files.
model_version	The "experiment ID".
eval_metric	The evaluation metric to use when finding best checkpoint.
--storefile	The path where the evaluation HDF5 file should be stored.
--stepnum	The best iteration number.
--skip_summary	True if the intermediate summary should be skipped.

The experiment folder should contain the files "experiment_params.json", "dataset_params.json", "model_params.json", "trainer_params.json" and "log_params.json".

The SciNets container

Installing the required version of TensorFlow is not a trivial task, as it requires the installation of old versions of CUDA and cuDNN. This problem is alleviated by the fact that SciNets is created using TensorFlow 1.X, not 2.X. TensorFlow 2.X includes a major rewrite of the TensorFlow API, and will, upon release, break the SciNets codebase[5]. Thus, to make SciNets easily accessible, we have provided a *Docker* image with SciNets installed.

Docker is a program that enables us to run virtual machines with little to no performance overhead [90]. To use it, we need to a *Dockerfile*, which is a "recipe" for creating a virtual machine. This image contains information about which operating system to use, which programs to install and what commands to run when the virtual machine is started. Thus, by creating a Docker image with the correct version of TensorFlow, CUDA and cuDNN, we ensure that the tools created in this project can be run on any computer. A Dockerfile with SciNets and all of its dependencies installed is available on the GitHub repository.

[5]The reason SciNets is built using TensorFlow 1.X is that 2.X is not released at the time of writing.

Extending SciNets

SciNets has, as mentioned earlier, been designed with the goal of extendability. To demonstrate this, we introduce two examples, one where we define a new layer and one where we define a new loss function.

Example 3.1.9 (Creating custom layers)**.**

```
import tensorflow as tf
import scinets.model
from scinets.model.layers import Conv2D

class Conv3D(Conv2D):
    def _build_layer(
        self,
        out_size,
        k_size=3,
        use_bias=True,
        dilation_rate=1,
        strides=1,
        padding="SAME",
    ):
        out = tf.layers.conv3d(
            self.input,
            out_size,
            kernel_size=k_size,
            use_bias=use_bias,
            kernel_initializer=self.initializer,
            strides=strides,
            dilation_rate=dilation_rate,
            padding=padding,
            kernel_regularizer=self.regularizer,
        )
        out = self.activation(out)
        out = self.normalizer(out, name="BN")

        return out


loss_function = ({"operator": "BinaryFBeta", "arguments": {"beta
    ": 2}},)
architecture = [
    {
        "layer": "Conv3D",
        "scope": "conv1",
        "layer_params": {"out_size": 8, "k_size": 3},
        "normalizer": {"operator": "BatchNormalization"},
```

```
39          "activation": {"operator": "ReLU"},
40          "initializer": {"operator": "he_normal"},
41          "regularizer": {"operator": "WeightDecay", "arguments":
                {"amount": 1}},
42      },
43      {
44          "layer": "Conv3D",
45          "scope": "conv2",
46          "layer_params": {"out_size": 8, "k_size": 3, "strides":
                4},
47          "normalizer": {"operator": "BatchNormalization"},
48          "activation": {"operator": "ReLU"},
49          "initializer": {"operator": "he_normal"},
50      },
51      {
52          "layer": "Conv3D",
53          "scope": "conv3",
54          "layer_params": {"out_size": 16, "k_size": 3},
55          "normalizer": {"operator": "BatchNormalization"},
56          "activation": {"operator": "ReLU"},
57          "initializer": {"operator": "he_normal"},
58      },
59      {
60          "layer": "LinearInterpolate",
61          "scope": "linear_upsample",
62          "layer_params": {"rate": 4},
63      },
64      {
65          "layer": "Conv3D",
66          "scope": "conv4",
67          "layer_params": {"out_size": 32, "k_size": 3},
68          "normalizer": {"operator": "BatchNormalization"},
69          "activation": {"operator": "ReLU"},
70          "initializer": {"operator": "he_normal"},
71      },
72      {
73          "layer": "Conv3D",
74          "scope": "conv5",
75          "layer_params": {"out_size": 1, "k_size": 3},
76          "normalizer": {"operator": "BatchNormalization"},
77          "activation": {"operator": "Sigmoid"},
78          "initializer": {"operator": "he_normal"},
79      },
80  ]
81  input = tf.placeholder(tf.float32, shape=(16, 256, 256, 256, 2))
82  true_out = tf.placeholder(tf.float32, shape=(16, 256, 256, 256,
        1))
83  is_training = tf.placeholder(tf.bool, shape=(,))
```

```
84
85 model = scinets.model.NeuralNet(
86     input=input,
87     true_out=true_out,
88     architecture=architecture,
89     loss_function=loss_function,
90     is_training=is_training,
91 )
92
93 with tf.Session() as sess:
94     sess.run([tf.global_variables_initializer(), dataset.
           initializers])
95
96     feed_dict = {input: INPUT_IMAGE, true_out: TRUE_MASK,
           is_training=False,}
97     proposed_segmentation, loss = sess.run([model.out, model.
           loss], feed_dict=feed_dict)
```

This code does the exact same as the code in Example 3.1.3, except with three dimensional convolutions instead of two dimensional ones. The lines of interest are 5-29 and every ”layer” line in the architecture definition.

Example 3.1.10 (Creating custom layers)**.**

```
1  import tensorflow as tf
2  import scinets.model
3  from scinets.model.loss import BaseLoss
4
5  class LpLoss(BaseLoss):
6      def __init__(self, p=2, name="loss_function"):
7          super().__init__(name=name)
8          self.p = p
9
10     def _buid_loss(self, prediction, target):
11         reduce_ax = range(1, len(prediction.get_shape().as_list
               ()))
12         p_err = tf.math.pow(prediction - target, self.p)
13         return tf.reduce_sum(p_err/p, ax=reduce_ax)
14
15 loss_function = ({"operator": "LpLoss", "arguments": {"p": 2}},)
16 architecture = [
17     ...
18 ]
19
20 preprocessor = {
21     ...
```

```
22 }
23
24
25 is_training = tf.placeholder(tf.bool, shape=(,))
26 is_testing = tf.placeholder(tf.bool, shape=(,))
27
28 train_batch_size = 8
29 val_batch_size = 16
30 test_batch_size = 16
31
32 dataset = scinets.data.HDFDataset(
33     ...
34 )
35 model = scinets.model.NeuralNet(
36     ...
37 )
38
39 steps_per_epoch = len(dataset)//dataset.batch_size[0]
40 train_op = {"operator": "Adam"}
41 learning_rate_scheduler = {
42     "operator": PolynomialDecay,
43     "arguments": {"decay_steps": 10000, "power": 1,}
44 }
45
46 trainer = scinets.trainer.NetworkTrainer(
47     model=model,
48     log_dir="./logs",
49     steps_per_epoch=steps_per_epoch,
50     train_op=train_op,
51     learning_rate_scheduler=learning_rate_scheduler,
52     max_checkpoints=5,
53     save_steps=2000,
54     verbose=True,
55 )
56
57 with tf.Session() as sess:
58     sess.run([tf.global_variables_initializer(), dataset.
        initializers])
59
60     trainer.train(sess, num_steps=1000,)
61
62     feed_dict = {is_training=False, is_testing=False,}
63     proposed_segmentations, losses = sess.run([model.out, model.
        loss], feed_dict=feed_dict,)
```

This code does the exact same as the code in Example 3.1.5, except with an L_2 loss instead of a F_2 loss. The lines of interest are Line 5 through to 15.

Both examples above demonstrate how easy it is to extend the SciNets library, without changing any of its components. This extendability is made possible by the `SubclassRegister`, which places the newly created classes in a dictionary. Thus, when the `get_layer` or `get_loss` function is called with the name of the new class, it is still found.

3.2 Organising the dataset

3.2.1 The HDF5 format

It is integral to keep the dataset organised when we tackle machine learning problems. We have chosen to use the HDF5 file format (or Hierarchical Data Format 5)[91], since it is recommended by NASA for storing large datasets [92]. A thorough introduction to the HDF5 data format and how to use it in Python is given in *Python and HDF5: Unlocking Scientific Data* by Collette [87]. We will, therefore, only give a very brief introduction to the file format here.

A key component of HDF5 files is *datasets*, which contain a data array (the data) and key-value pairs (the attributes). We will not discuss the key-value pairs in this section. The data array can be stored directly on disc, or in a compressed format. Furthermore, the data array can be stored in two ways, either in compressed chunks, or decompressed. To see how we can create a dataset using the `h5py` library, see Example 3.2.1.

One benefit of the HDF5 file format is that it allows for array slicing directly from disc. Say we have a multidimensional array, and we want every second element of the second axis. If this was a numpy array, we could simply write `data = arr[0, ::2]`. However, if we store this numpy array as a CSV file, we need to read the entire dataset from disc in order to find the correct bytes. With the HDF5 format, however, we can perform array slicing from disc, thus removing the time needed to loading everything. This is demonstrated in Example 3.2.1.

Example 3.2.1 (HDF5 datasets)**.**

```
1 import h5py
2 import numpy as np
3
4 random_data = np.random.randn(4, 2, 2).astype(np.float32)
5 with h5py.File("test_hdf5.h5", "w") as h5:
```

```
    h5.create_dataset("test_dataset", dtype=np.float32, shape=
        (4, 2, 2))
    h5["test_dataset"][...] = random_data

with h5py.File("test_hdf5.h5", "r") as h5:
    first_slice = h5["test_dataset"][0]
    other_slice = h5["test_dataset"][:, 0, ::2]
    all_data = h5["test_dataset"][...]

assert np.array_equal(random_data[0], first_slice)
assert np.array_equal(random_data[:, 0, ::2], other_slice)
assert np.array_equal(random_data, all_data)
```

Here we see that we can create a HDF5 file with filename "test_dataset.h5" using the `h5py.File("test_hdf5.h5", "w")` context. Furthermore, we can later read from this dataset using the `h5py.File("test_hdf5.h5", "r")` context.

Another important part of the HDF5 format is groups. A group is like a folder in the file system. These groups can contain two things, datasets and other groups and is the origin of the *hierarchical* in the name of the file format. To understand how groups can be used, consider Example 3.2.2

Example 3.2.2 (HDF5 groups)**.**

```
import h5py
import numpy as np

random_data1 = np.random.randn(4, 2, 2).astype(np.float32)
random_data2 = np.random.randn(2, 2, 8).astype(np.float16)
with h5py.File("test_hdf5.h5", "w") as h5:
    # Create groups
    group = h5.create_group("group")
    subgroup1 = group.create_group("subgroup1")
    subgroup2 = group.create_group("subgroup2")

    # Create datasets
    test_dataset1 = subgroup1.create_dataset("test_dataset",
        dtype=np.float32, shape=(4, 2, 2))
    test_dataset2 = subgroup2.create_dataset("test_dataset",
        dtype=np.float16, shape=(2, 2, 8))

    # Set the data
    test_dataset1[...] = random_data1
    test_dataset2[...] = random_data2

with h5py.File("test_hdf5.h5", "r") as h5:
```

```
    first_slice = h5["group/subgroup1/test_dataset"][0]
    other_slice = h5["group/subgroup2/test_dataset"][:, 0, ::2]

assert np.array_equal(random_data1[0], first_slice)
assert np.array_equal(random_data2[:, 0, ::2], other_slice)
```

Here we demonstrate how to crate HDF5 files with a folder layout in Python using `h5py`. The structure is as follows, in the file root, we have one group named "group". This group contains two subgroups, "subgroup1" and "subgroup2" which contain one dataset each.

We could also have several groups in the file root, have several datasets in the groups and have a combination of groups and subgroups in the same group.

Chapter 4

Experimental setup

4.1 The dataset

The dataset used in this project contains 3D PET/CT images and segmentation masks for 197 patients that underwent treatment at the Oslo University Hospital, the Radium Hospital. Affected lymph nodes and the gross tumour volume (GTV) was used as the segmentation goal. The union of the ground truth segmentation masks was used where multiple delineations existed. Finally, each image was cropped in 3D to reduce class imbalance. The final slices contained between 0.02% and 32% tumour/lymph-node pixels.

Three separate datasets are needed for deep learning; a training set, which we use to train our network; a validation set, which we use to compare models; and a test set, which we use to assess the quality of our final model. The datasets were stratified by tumour stage such that the same datasets could be used for radiomics research (e.g. by transfer learning). An overview of the dataset sizes are given in Table 4.2

Two classes were used for the dataset at hand, healthy tissue and affected tissue. A pixel was in the affected tissue class if it was delineated as either tumorous or lymph node.

Each dataset file used for the experiments in this project were structured as follows. There are three groups in the file root, “train” (142 patients), “test” (40 patients) and “val” (15 patients). Each of these groups contain four datasets, “images”, “masks”, “patient_id” and “slice_id”. An overview of the contents of these datasets

Table 4.1: Description of the dataset file structure

Dataset	Shape	Datatype	Contents
images	`[n_images, x, y, c]`	float32	The input images.
masks	`[n_images, x, y, m]`	float32	The segmentation masks.
patient_id	`[n_images]`	uint16	The patient ID number
slice_id	`[n_images]`	uint16	The current slice in the patient's 3D image.

Overview of the structure of the HDF5 files used in this project. `n_images` is the total number of images in the dataset, `x` is the number of pixels in the x direction, `y` is the number of pixels in the y direction, `c` is the number of input channels (for PET/CT this is 2) and `m` is the number of segmentation masks (for binary segmentation problems, this is a singleton axis).

Table 4.2: The number of patients in each of the datasets used to train the model.

Dataset	No. patients
train	142
val	15
test	40

are given in Table 4.1 and the exact dataset sizes are given in Table 4.2.

4.2 Model parameters

Architecture

Over 100 different experiments were run, all using a U-Net architecture (Section 2.3.3 on page 61) [23], with the same number of layers, convolutions per layer and skip connections as described in [23]. An overview of this architecture is given in Table 4.3. Furthermore, all hyperparameters are reported in Table 4.4. Certain parameters were discontinued during training when the performance was shown to be suboptimal.

Layer type

Two types of layers were examined, convolutional layers and improved ResNet layers. The convolutional layers used in this text consisted of a single kernel of size 3, followed by a ReLU nonlinearity and finally batch normalisation. The improved ResNet layers [64] are described in Section 2.1.11 on page 38.

Loss function

Four different loss functions were used, the cross entropy loss (Section 2.1.3 on page 13), the Dice loss [42] (Section 2.3.2 on page 58) and the F_β loss (Section 2.3.2 on page 60) with $\beta = 2$ and $\beta = 4$.

Optimiser

To train the network, the ADAM optimiser was used. After training all models with the Adam optimiser, the best performing model using only the CT channel and the best performing model using PET/CT were trained using the SGDR+Momentum algorithm. The hyperparameters chosen for the SGDR learning rate schedule are showed in Table 4.5.

Table 4.3: Overview of the architecture used in this project.

Name	Type	Input	No. output channels
Conv 1	Convolutional	Input image	64
Conv 2	Convolutional	Conv 1	64
MaxPool 1	Max Pooling	Conv 2	64
Conv 3	Convolutional	MaxPool 1	128
Conv 4	Convolutional	Conv 3	128
MaxPool 2	Max Pooling	Conv 4	128
Conv 5	Convolutional	MaxPool 2	256
Conv 6	Convolutional	Conv 5	256
MaxPool 3	Max Pooling	Conv 6	256
Conv 7	Convolutional	MaxPool 3	512
Conv 8	Convolutional	Conv 7	512
MaxPool 4	Max Pooling	Conv 8	512
Conv 9	Convolutional	MaxPool 4	1024
Conv 10	Convolutional	Conv 9	1024
UpConv 1	Upconvolutional	Conv10	512
Conv 11	Convolutional	UpConv 1, Conv 8	512
Conv 12	Convolutional	Conv 11	512
UpConv 2	Upconvolutional	Conv12	256
Conv 13	Convolutional	UpConv 2, Conv 6	256
Conv 14	Convolutional	Conv 13	256
UpConv 3	Upconvolutional	Conv14	128
Conv 15	Convolutional	UpConv 3, Conv 4	128
Conv 16	Convolutional	Conv 15	128
UpConv 4	Upconvolutional	Conv14	64
Conv 17	Convolutional	UpConv 4, Conv 2	64
Conv 18	Convolutional	Conv 17	64
Conv 19	Convolutional	Conv 18	1

The Convolutional and upconvolutional layers were either standard layers or Improved ResNet layers [64].
The inputs were concatenated for layers with multiple inputs.

Table 4.4: Overview of the hyperparameters used for the U-Net architecture.

Hyperparameter	Value(s)
Learning rate	[0.0001, 0.00001]
Optimiser	Adam
Nonlinearity	ReLU
Normalizer	Batch Normalisation
Initializer	Normally distributed He
Layer type	[Convolutional, Improved ResNet]
Loss	[Cross entropy, F_1, F_2, F_4]
Window centre	[60 HU, 70 HU]
Window width	[100 HU, 200 HU]
Batch size	16
Number of iterations	$10000 - 30000$
Iterations between checkpoints	2000

Table 4.5: Overview of the hyperparameters used for the SGDR+momentum optimiser.

Hyperparameter	Value(s)
α_{min}	0
α_{max}	0.05
$T_{restart}$	650 iterations ($\approx$ 10 epoch)
T_{mult}	2
μ	0.9
Batch size	16

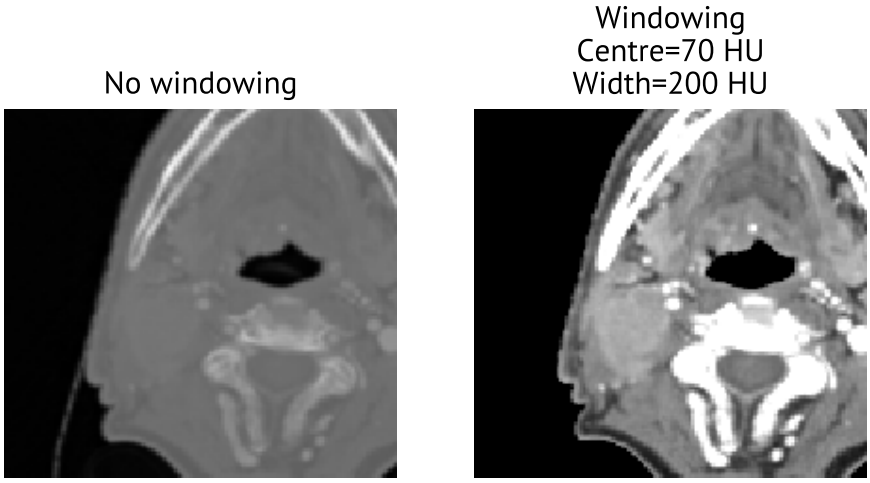


Figure 4.1: Illustration of Hounsfield windowing. The left image shows a CT slice with full dynamic range whereas the right image shows the same CT slice but with reduced dynamic range.

Preprocessing

There were three types of pre-processing performed on the training data. Removing the PET channel, removing the CT channel and reducing the dynamic range of the CT channel through thresholding (henceforth named Hounsfield windowing). All possible combinations of preprocessing were performed (i.e. PET/CT, PET/CT + windowing, CT, CT + windowing, PET). The focus of these experiments are CT-only and PET/CT models, since CT is standard procedure when performing a PET scan [5].

The Hounsfield windowing parameters were set after consulting with a radiologist. The window centres were set approximately equal to the average tumour value and median tumour value and the window size was set to encompass most of the soft tissue dynamic range. he windowing parameters are given in Table 4.4. Experiments were also run such that all windows were used simultaneously, fed in as different input channels. An illustration of Hounsfield windowing is given in Figure 4.1.

4.3 The training procedure

A server with five Nvidia GeForce 1080Ti, one Intel Xeon E5-2620 CPU and 64 GB RAM was used to train the models. Each model was trained using a single GPU, but up to five models were trained concurrently (dependent on the availability). Server restarts were common, as the company that owns it were doing experiments at the same time. This led to some experiments being repeated several times and others not being performed at all (in particular, the 60HU PET/CT experiments crashed at a time when they could not be repeated). It took five weeks to train all models. There was some downtime between experiments because of server restarts. The GPUs were, therefore, used approximately 60% of these three weeks. Running experiments in parallel reduced GPU time by approximately 40%.

For reproducability, all parameter files used in this project are located on the SciNets GitHub repository (https://github.com/yngvem/scinets/). Furthermore, a virtual server was rented to store the Sacred logs in an off-site MongoDB database, making it easy to compare runs performed on different computers.

4.4 Analysis of model performance

The average Dice score per slice was used to compare the different models.

Finally, when all models were trained, a network with the best hyperparameter-setup for each modality was trained with a dataset file that did not contain any validation-data or test-data. The Dice performance of these models on the validation set were then compared with the best over-all models to ensure that there was no dataset-contamination.

Chapter 5

Results

5.1 Hyperparameter effects on model performance

Let us start by exploring the effect of each hyperparameter on model performance. The average Dice per slice (henceforth called performance) of each trained model was stored in a database together with the hyperparameters of each run. Summary statistics of the performance distrubution for each hyperparameter were computed to assess which hyperparameter combination provided the highest performance (henceforth called best).

For example, all trained models were separated by which layer type they used. Then the summary statistics of the performances, such as mean and median performance, were computed for all models with ResNet layers and all models with convolutional layers. Thus, we have summary statistics that indicate whether convolutional layers or ResNet layers provided the highest performance.

The tables in this and the following section only contain the results from the experiments conducted with the Adam optimiser. The hyperparameters of the SGDR+momentum models were chosen based on these results. The results from the SGDR+momentum were therefore omitted here, to prevent skewing of the results.

To measure performance, all checkpointed weights were tested. The highest performance on the validation set was most commonly found at iteration 2000 and

4000 while it was seldomly found at iteration 6000 and 8000. An epoch was approximately 650 iterations. The choice of learning rate did not have a noticable impact on the optimal checkpoint. Typical loss and Dice curves are shown in Figure 5.1

5.1.1 Single hyperparameters

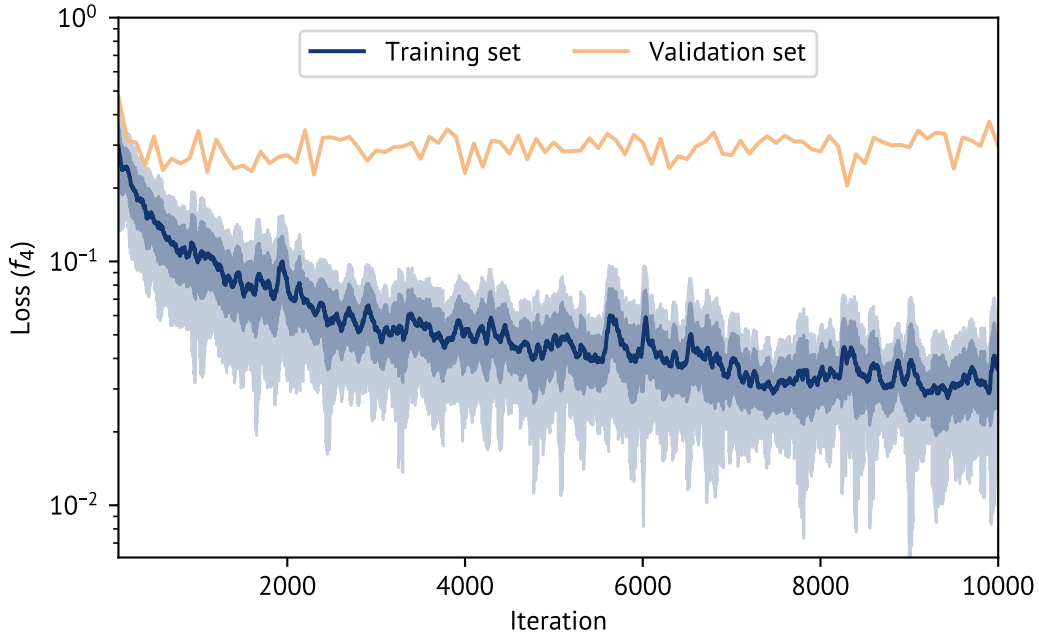
The training loss of networks with ResNet layers did not decrease (with the exception of two models with vastly different hyperparameters). This is illustrated in Table 5.1 which compares the efficiency of ResNet layers and convolutional layers on the validation set. During training, it became apparent that the partial derivatives with respect to the parameters of the skip-connections in the ResNet layers had significantly higher values than the partial derivatives with respect to the residual blocks. This indicated exploding gradients on the skip connections in ResNet layers. Therefore, all further analysis will disregard the results from models using ResNet layers.

In Table 5.2, the summary statistics of the effects of the "loss" hyperparameter on performance are shown. The cross entropy loss generally provided lower Dice values than the F_β loss for all tested values of β. Furthermore, the F_2 and F_4 loss had higher performance than the F_1 loss with respect to all performance summary statistics.

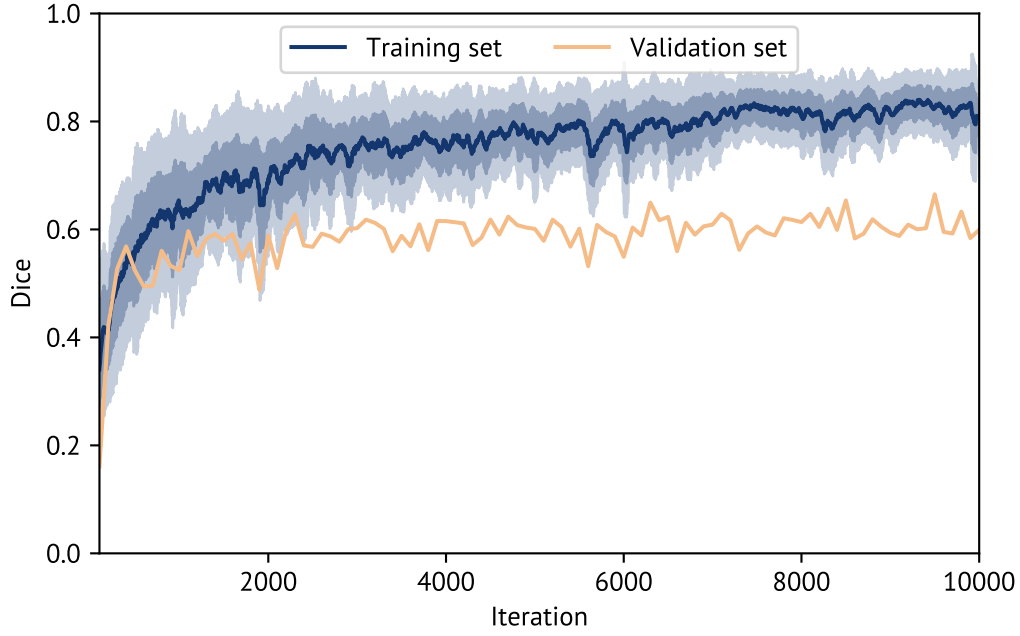
Table 5.3 demonstrates that the model performance differed greatly depending on the choice of "channels" hyperparameter. Models trained with PET/CT achieved the best result, followed by the PET-only models. The CT-only models had lower performance on all summary statistics of the average Dice.

The "learning rate" hyperparameter also has a notable effect on model performance, as demonstrated by Table 5.4. Specifically, it is clear that choosing a small learning rate led to convergence to a worse local minimum than with a higher learning rate.

Finally, Tables 5.5 to 5.7 display the results pertaining to CT windowing. Table 5.5 demonstrate that preprocessing CT images with Hounsfield windowing has a clear influence on model performance. The effect of the windowing parameters was, however, marginal, as is shown in Tables 5.6 and 5.7.



(a)



(b)

Figure 5.1: The loss curves on a logarithmic scale (a) and Dice curves (b) for a typical model. Note that the model has not converged, as the loss is still decreasing, however, the validation lines have plateaued. The dark blue lines show the loss/Dice evaluated on the training set (rolling average of performance over 51 iterations), the dark blue areas represent one standard deviation and the light blue areas represent two standard deviations. The orange lines show the loss/Dice performance on the validation set (evaluated every 100th iteration on 256 random slices).

Table 5.1: Dice results on the validation set for the "layer type" hyperparameter.

Layer type	Min	Max	STD	Mean	Median	No. experiments
Convolutional	0.439	0.620	0.043	0.549	0.548	168
ResNet	0.000	0.508	0.108	0.101	0.109	36

Table 5.2: Dice results on the validation set for the "loss" hyperparameter.

Loss	Min	Max	STD	Mean	Median	No. experiments
F_1	0.469	0.611	0.047	0.541	0.541	41
F_2	0.500	0.618	0.035	0.567	0.563	41
F_4	0.515	0.620	0.033	0.566	0.558	33
Cross entropy	0.439	0.604	0.042	0.533	0.514	33

Table 5.3: Dice results on the validation set for the "channels" hyperparameter.

Channels	Min	Max	STD	Mean	Median	No. experiments
CT	0.439	0.565	0.027	0.520	0.517	80
PET	0.557	0.594	0.012	0.577	0.580	14
PET/CT	0.545	0.620	0.018	0.593	0.595	54

Table 5.4: Dice results on the validation set for the "learning rate" hyperparameter.

Learning rate	Min	Max	STD	Mean	Median	No. experiments
0.00001	0.442	0.582	0.039	0.525	0.526	20
0.0001	0.439	0.620	0.042	0.552	0.555	148

Table 5.5: Dice results on the validation set for the "windowing" hyperparameter.

Windowing	Min	Max	STD	Mean	Median	No. experiments
False	0.439	0.595	0.046	0.545	0.561	48
True	0.469	0.620	0.042	0.550	0.546	120

Table 5.6: Dice results on the validation set for the "window centre" hyperparameter.

Window centre	Min	Max	STD	Mean	Median	No. experiments
60 HU	0.485	0.562	0.025	0.526	0.529	24
70 HU	0.493	0.565	0.021	0.527	0.528	32
Both	0.469	0.548	0.025	0.509	0.507	16

Only results from the CT-only models are shown here as the server crashed during the experiments with PET/CT input and window size 60. Including PET/CT performances where available would therefore yield interpretable results.

Table 5.7: Dice results on the validation set for the "window width" hyperparameter.

Window width	Min	Max	STD	Mean	Median	No. experiments
100 HU	0.488	0.617	0.041	0.552	0.547	44
200 HU	0.485	0.620	0.040	0.553	0.549	44
Both	0.469	0.617	0.045	0.544	0.542	32

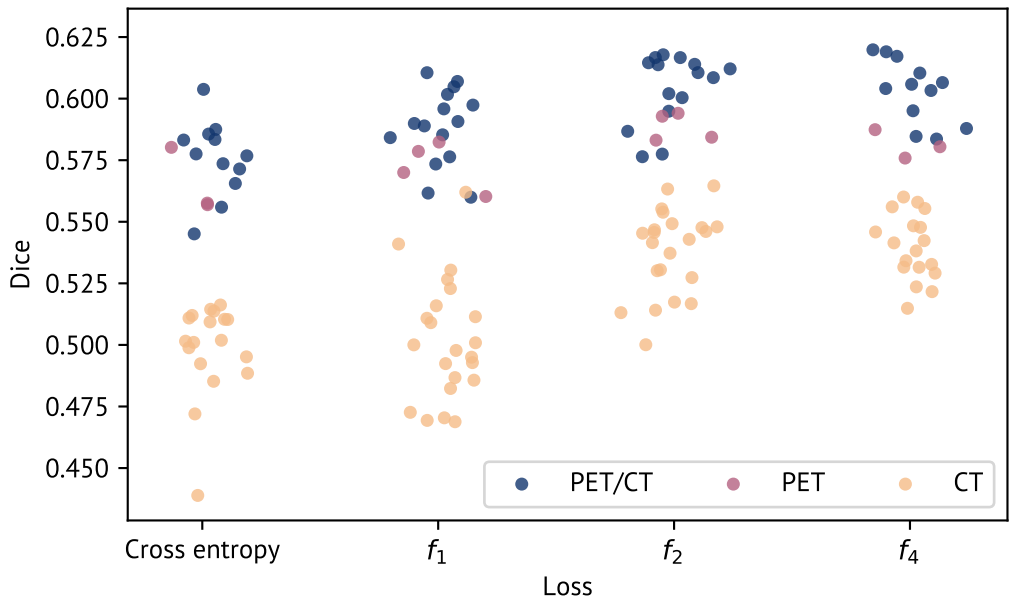


Figure 5.2: Jitter plot showing the performance of the different models on the validation set grouped by loss function and input channels.

5.1.2 Hyperparameter combinations

There are certain pairs of hyperparameters whose joint analysis are of interest. Pairwise tables and a colour-coded jitter plot are therefore provided, demonstrating model performance for certain hyperparameter combinations. It should be noted that the results obtained using ResNet and a low learning rate are omitted due to their poor performance.

Table 5.8 shows the Dice performance on the "loss" and "channels" hyperparameters, demonstrating that the F_2 and F_4 loss provided higher performance than the cross entropy and F_1 loss for all modalities. Furthermore, Table 5.8 demonstrates that the choice of loss function had a larger effect on the CT-only model than on the PET-only and PET/CT models. The same conclusions can be drawn from Figure 5.2, which shows the Dice distribution for these hyperparameters.

The effect of windowing was demonstrated in the previous section. However, how to choose correct windowing parameters was not obvious. Table 5.9 shows that windowing yields models with higher average Dice per slice for both CT-only and

Figure 5.3: Jitter plot showing performance on the validation set based on whether or not the CT-channel was processed using Hounsfield windowing.

PET/CT models. This is also shown in Figure 5.3. Table 5.10 demonstrates that the windowing parameters did not affect the model performance much.

Table 5.8: Dice results on the validation set for the "loss" and "channels" hyperparameters.

	Channels		
Loss	PET/CT	PET	CT
F_1	0.59/0.59 (15)	0.57/0.57 (4)	0.50/0.50 (22)
F_2	0.60/0.61 (15)	0.59/0.59 (4)	0.54/0.54 (22)
F_4	0.60/0.60 (12)	0.58/0.58 (3)	0.54/0.54 (18)
Cross entropy	0.58/0.58 (12)	0.56/0.56 (3)	0.50/0.50 (18)

The format is "mean/median (count)", where count is the number of experiments with this particular parameter configuration.

Table 5.9: Dice results on the validation set for the "windowing" and "channels" hyperparameters.

	Windowing	
Channels	True	False
CT	0.52/0.53 (66)	0.50/0.50 (14)
PET/CT	0.60/0.60 (40)	0.57/0.58 (14)

The format is "mean/median (count)", where count is the number of experiments with this particular parameter configuration.

Table 5.10: Dice results for the "window centre" and "window width" hyperparameters.

	Window centre		
Window width	60 HU	70 HU	Both
100 HU	0.52/0.52 (12)	0.53/0.53 (16)†	–/– (0)
200 HU	0.53/0.53 (12)	0.53/0.53 (16)†	–/– (0)
Both	–/– (0)	–/– (0)	0.51/0.51 (16)†

The format is "mean/median (count)", where count is the number of experiments with this particular parameter configuration.
† The PET/CT results are omitted here as the server crashed during the experiments with PET/CT input and a window centre of 60 HU. Including PET/CT results for these values would therefore not yield interpretable results.

5.1.3 The SGDR+momentum optimiser

The highest performing CT-only and PET/CT models were trained twice using the SGDR+momentum optimiser. The hyperparameters of these models are given in given in Table 5.11.

For CT-only, one of the runs with SGDR+momentum yielded an improvement on the validation performance compared to Adam, for PET/CT the results were unchanged. Consequently, the SGDR+momentum CT-only model was taken forward for further analysis and the Adam based models were taken forward for PET/CT and PET-only models. The average Dice performance of these trainings is given in Table 5.12.

Table 5.11: The hyperparameters of the models that achieved highest mean Dice on the validation dataset.

	Modality	
Hyperparameter	CT	PET/CT
Loss	F_2	F_4
Optimiser	Adam	Adam
Learning rate	0.0001	0.0001
Window centre	70 HU	70
Window width	100	200
Dice	0.56	0.62

Table 5.12: The results from the SGDR+momentum runs.

Modality	Run 1	Run 2
CT	0.58	0.54
PET/CT	0.62	0.61

5.2 The highest performing models

The hyperparameters of the highest performing models are given in Table 5.13. The summary statistics for the Dice distribution of these models are provided in Table 5.14. These models with these hyperparameters were trained with a dataset file that did not contain the test and validation data. When omitting these datasets from the training procedure, we achieved a mean and median Dice within 0.02 of the best overall models (evaluated on the actual validation data), hence implying no dataset contamination.

Additional performance metrics for the model with best average Dice score are provided in Table 5.15. This table shows that the sensitivity is highest for the PET/CT model and lowest for the CT-only model. Similarly, the PPV is highest for the PET-only model and lowest for the CT-only model. The specificity is high for all three models.

Furthermore, Table 5.15 shows that the sensitivity was higher than the PPV for both the PET-only model and the PET/CT model. On the CT-only model the mean PPV was slightly lower than the mean sensitivity, whereas the median PPV was slightly higher than the median sensitivity.

Table 5.16 demonstrates that there are trade-offs between the PET-only model and the CT-only model. There are four patients (patients 35, 87, 90 and 241) where the CT-only performs best, four where the PET-only model performs best (patients 29, 49, 229 and 246). The PET/CT model performs the best on the last seven patients. Furthermore, we note that the PET/CT model performs better than the PET-only model on both patients the CT-only model failed to delineate (patients 18 and 177).

The CT-only model performs better than the PET-only model and PET/CT model for both patients the CT-only model failed to delineate.

Table 5.13: The hyperparameters of the models that achieved highest mean Dice on the validation dataset.

	Modality		
Hyperparameter	CT	PET	PET/CT
Loss	F_2	F_4	F_4
Optimiser	SGDR+momentum	Adam	Adam
Learning rate	0.5	0.0001	0.0001
Window centre	70 HU	–	70
Window width	100	–	200
Dice	0.58	0.59	0.62

Table 5.14: Dice performance per slice in the validation set for the best models using each modality.

Modality	Mean	Median	STD	25% Quantile	75% quantile
CT	0.57	0.70	0.31	0.35	0.80
PET	0.59	0.66	0.25	0.47	0.79
PET/CT	0.62	0.71	0.26	0.47	0.82

Model specifications are given in Table 5.13

Table 5.15: Performance metrics for the best three models.

	Loss		
Metric	CT	PET	PET/CT
Sensitivity	0.60 (0.73) $\pm$ 0.34	0.63 (0.70) $\pm$ 0.30	0.72 (0.83) $\pm$ 0.31
Spesificity	0.99 (0.99) $\pm$ 0.01	0.99 (1.00) $\pm$ 0.01	0.99 (0.99) $\pm$ 0.01
PPV	0.61 (0.71) $\pm$ 0.33	0.66 (0.74) $\pm$ 0.29	0.62 (0.71) $\pm$ 0.29
Dice	0.57 (0.70) $\pm$ 0.31	0.59 (0.66) $\pm$ 0.25	0.62 (0.71) $\pm$ 0.26

The format is mean (median) $\pm$ standard deviation, computed on a slice-by-sliced basis.

Model specifications are given in Table 5.13

Table 5.16: Mean and median Dice for the best models using each modality, evaluated on the patients in the validation set.

Patient ID	CT	PET	PET/CT
29	0.71 (0.72)	0.75 (0.81)	0.67 (0.69)
35	0.62 (0.75)	0.48 (0.51)	0.62 (0.66)
38	0.14 (0.10)	0.38 (0.45)	0.39 (0.52)
49	0.66 (0.67)	0.71 (0.69)	0.69 (0.74)
70	0.49 (0.55)	0.47 (0.47)	0.53 (0.60)
87	0.75 (0.75)	0.74 (0.76)	0.74 (0.75)
90	0.54 (0.65)	0.45 (0.45)	0.37 (0.39)
98	0.73 (0.79)	0.65 (0.67)	0.77 (0.80)
163	0.80 (0.85)	0.80 (0.83)	0.83 (0.87)
170	0.65 (0.75)	0.68 (0.77)	0.70 (0.80)
177	0.00 (0.00)	0.05 (0.00)	0.08 (0.00)
213	0.71 (0.81)	0.74 (0.79)	0.75 (0.84)
229	0.40 (0.34)	0.58 (0.64)	0.57 (0.69)
241	0.70 (0.77)	0.62 (0.69)	0.69 (0.76)
246	0.37 (0.35)	0.57 (0.65)	0.46 (0.50)

The format is "mean (median)".
The model attained best performance for the patient in **bold**.
Model specifications are given in Table 5.13

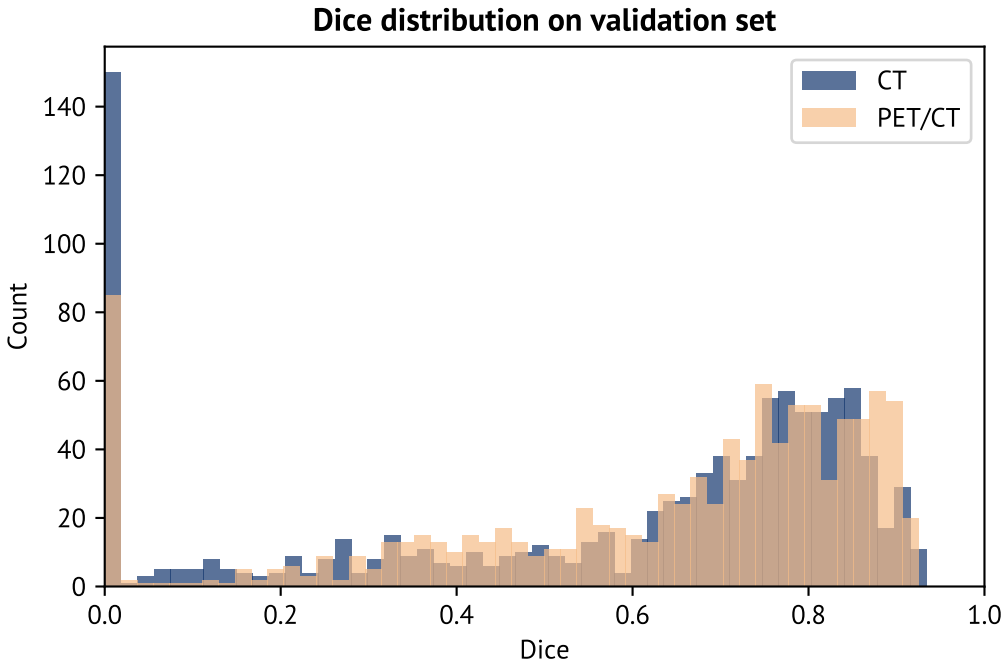
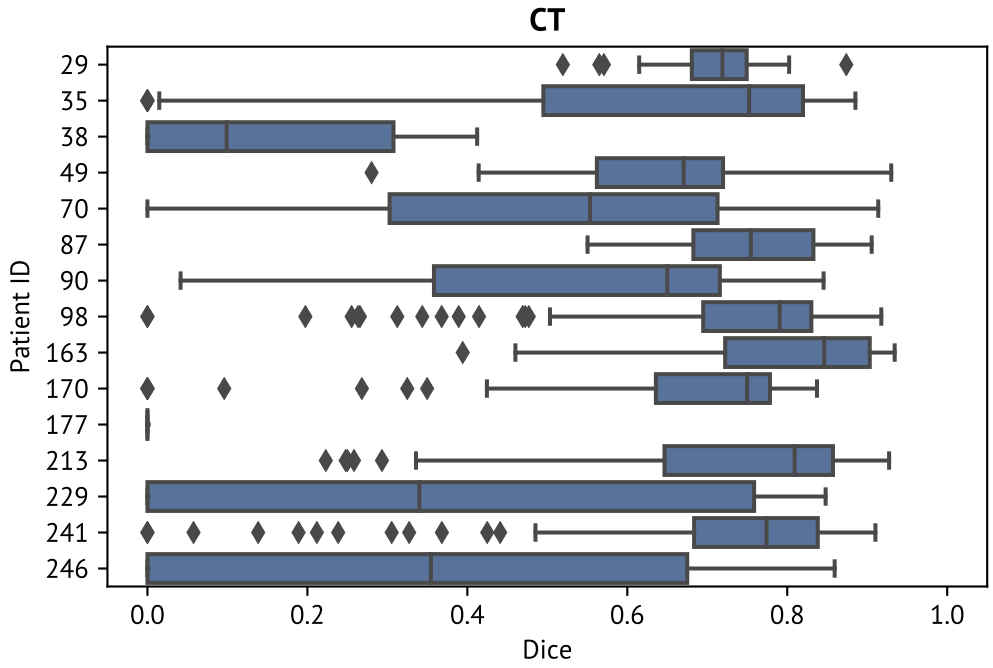


Figure 5.4: Histogram of the Dice per slice on the validation set for the best CT-only and PET/CT models. Model specifications are given in Table 5.13

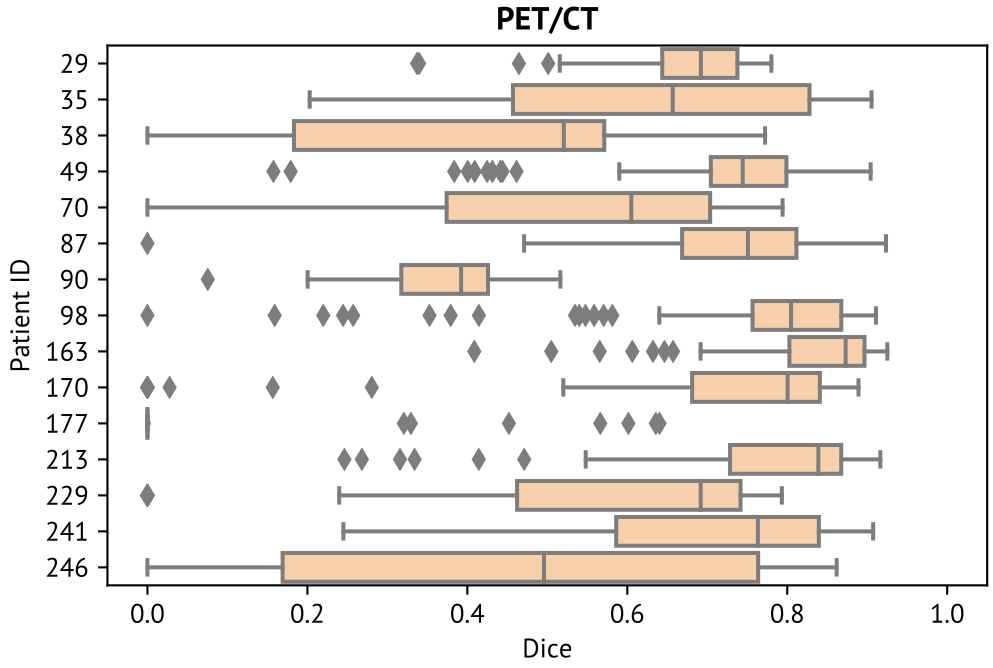
Analysis of the CT-only and PET/CT model

Figure 5.4 shows a histogram of the Dice per slice for the CT-only and PET/CT models that achieved highest average Dice on the validation set. It is apparent that the main difference between the CT-only model and the PET/CT model is the number of slices with close to no overlap between the predicted delineation and the oncologist's delineation. On images that are well segmented, the models are comparable.

Similarly, Figure 5.5 shows two boxplots that illustrate the Dice-performance for both models per patient in the validation set. There is one key differences between the PET/CT model and the CT-only model in these plots. Namely that the CT-only model has a larger number of slices where the network failed to delineate the affected tissue (i.e. Dice close to zero) than the PET/CT model. These failure slices occur in all patients. Furthermore, we see that the performance on some patients are slightly preferable on the CT-only model (e.g. patient 29).



(a)



(b)

Figure 5.5: Boxplots illustrating the Dice distribution for the best model (evaluated on the validation set) per slice for each patient. The line within each box represents the median, the box itself contains all data within the 25% quantile and the 75% quantile. The whiskers span the full dataset, excluding (automatically detected) outliers, which are marked with diamonds. (a) shows the results for the CT-only model and (b) shows the results for the PET/CT model. Model specifications are given in Table 5.13

5.3 Model performance on the test set

The models with the highest average Dice within each modality (reported in Table 5.13 on page 128) were run on the test set, and the summary statistics of the Dice performance are reported in Table 5.17. On average, the PET/CT model proveded the highest performance

Additionally, Table 5.18 shows that the sensitivity was higher than the PPV for both the PET-only model and the PET/CT model on the test data. For the CT-only model, on the other hand, the mean PPV was higher than the mean sensitivty (in contrast to the validation results).

Finally, we have the performance of the best models on each patient in Table 5.19. This table demonstrate that the PET/CT model had highest performance for 22 out of 40 patients, the PET-only model performed best on 12 out of 40 patients and the CT-only model performed the best on 6 out of 40 patients. If we consider the median performance instead of mean, we see that the CT-only model performs best on 8 patients, the PET-only model performs best on 10 patients and the PET/CT model performs best on 22 patients.

Table 5.17: Dice performance per slice in the test set for the best models using each modality.

Modality	Mean	Median	STD	25% Quantile	75% quantile
CT	0.56	0.65	0.29	0.38	0.79
PET	0.64	0.72	0.24	0.56	0.80
PET/CT	0.66	0.75	0.24	0.59	0.82

Model specifications are given in Table 5.13

Table 5.18: Performance metrics for the best three models on the test set.

	Modality		
Metric	CT	PET	PET/CT
Sensitivity	0.58 (0.64) ± 0.33	0.69 (0.76) ± 0.27	0.78 (0.89) ± 0.28
Spesificity	0.99 (1.00) ± 0.01	0.99 (0.99) ± 0.01	0.99 (0.99) ± 0.01
PPV	0.62 (0.71) ± 0.31	0.64 (0.71) ± 0.27	0.62 (0.68) ± 0.26
Dice	0.56 (0.65) ± 0.29	0.64 (0.72) ± 0.24	0.66 (0.75) ± 0.24

The format is mean (median) ± standard deviation, computed on a slice-by-sliced basis.
Model specifications are given in Table 5.13

Table 5.19: Mean and median Dice for the best models using each modality, evaluated on the patients in the test set.

Patient ID	CT	PET	PET/CT
5	0.74 (0.77)	0.70 (0.74)	0.74 (0.79)
8	0.37 (0.36)	0.60 (0.72)	0.65 (0.71)
13	0.45 (0.50)	0.59 (0.67)	0.58 (0.72)
16	0.31 (0.32)	0.32 (0.33)	0.47 (0.55)
18	0.48 (0.56)	0.70 (0.75)	0.69 (0.75)
21	0.51 (0.57)	0.66 (0.71)	0.63 (0.69)
36	0.74 (0.84)	0.64 (0.69)	0.74 (0.75)
44	0.41 (0.47)	0.73 (0.77)	0.68 (0.81)
52	0.65 (0.75)	0.74 (0.74)	0.76 (0.81)
55	0.64 (0.68)	0.67 (0.71)	0.75 (0.78)
60	0.64 (0.71)	0.63 (0.68)	0.65 (0.75)
61	0.56 (0.60)	0.57 (0.62)	0.61 (0.73)
67	0.57 (0.59)	0.67 (0.69)	0.58 (0.62)
73	0.67 (0.67)	0.65 (0.66)	0.62 (0.63)
74	0.54 (0.55)	0.69 (0.73)	0.65 (0.71)
77	0.77 (0.84)	0.79 (0.81)	0.76 (0.78)
82	0.12 (0.00)	0.59 (0.60)	0.54 (0.70)
91	0.68 (0.79)	0.63 (0.71)	0.69 (0.78)
93	0.60 (0.60)	0.41 (0.37)	0.56 (0.58)
99	0.60 (0.68)	0.71 (0.80)	0.74 (0.86)
110	0.12 (0.16)	0.00 (0.00)	0.19 (0.20)
116	0.53 (0.77)	0.45 (0.60)	0.51 (0.59)
120	0.80 (0.82)	0.75 (0.77)	0.80 (0.82)
130	0.59 (0.69)	0.58 (0.62)	0.73 (0.82)
140	0.76 (0.79)	0.72 (0.76)	0.67 (0.69)
148	0.70 (0.72)	0.69 (0.71)	0.73 (0.77)
153	0.28 (0.33)	0.65 (0.72)	0.58 (0.67)
154	0.55 (0.60)	0.56 (0.71)	0.60 (0.72)

The format is “mean (median)”.
Model specifications are given in Table 5.13

Patient ID	CT	PET	PET/CT
162	0.55 (0.67)	0.62 (0.76)	0.63 (0.77)
164	0.44 (0.45)	0.44 (0.40)	0.47 (0.54)
169	0.68 (0.76)	0.62 (0.71)	0.72 (0.77)
184	0.57 (0.58)	0.81 (0.87)	0.82 (0.86)
191	0.61 (0.69)	0.59 (0.70)	0.67 (0.78)
194	0.68 (0.72)	0.66 (0.67)	0.71 (0.74)
209	0.69 (0.73)	0.80 (0.84)	0.77 (0.79)
217	0.57 (0.65)	0.74 (0.81)	0.69 (0.75)
223	0.54 (0.60)	0.58 (0.66)	0.50 (0.53)
233	0.59 (0.65)	0.72 (0.78)	0.72 (0.81)
242	0.64 (0.73)	0.63 (0.76)	0.70 (0.79)
249	0.67 (0.76)	0.60 (0.69)	0.67 (0.74)

The format is “mean (median)”.
Model specifications are given in Table 5.13

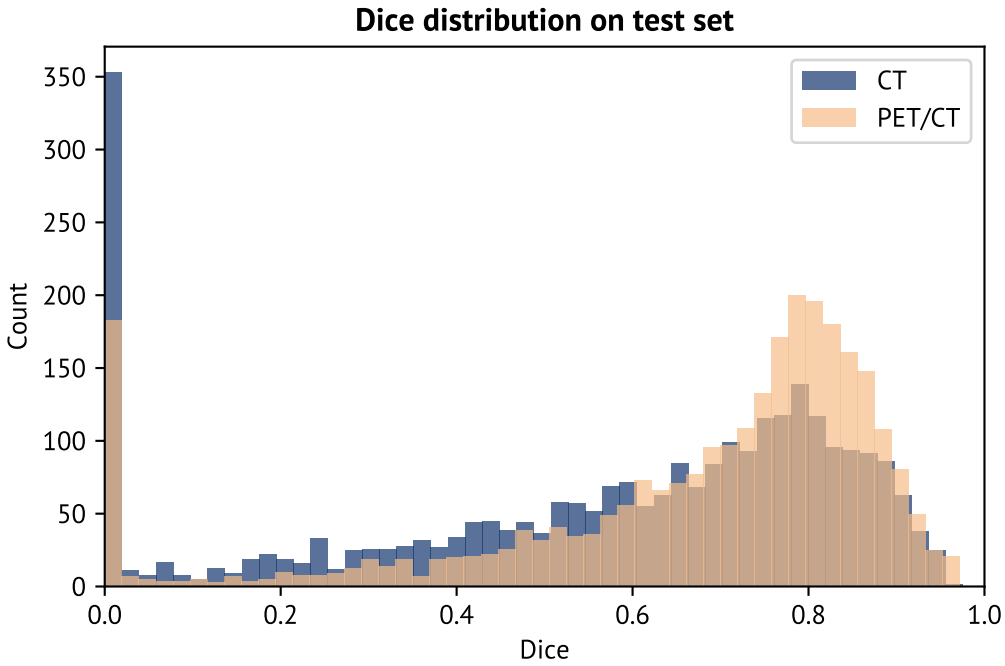


Figure 5.6: Histogram of the Dice per slice on the validation set for the best CT-only and PET/CT models. Model specifications are given in Table 5.13

5.3.1 Analysis of the CT-only and PET/CT model

Similarly to Figure 5.4, Figure 5.6 shows the Dice-per-slice distribution of the CT-only model and the PET/CT model. The Dice distribution of the test set bares strong resemblance to the distribution of the validation set, albeit with an important difference; the number of failure slices was less than half on the PET/CT model as compared with the CT-only model. Furthermore, the tail of the CT-only model's Dice distribution is more prominent than the tail of the PET/CT model's Dice distribution.

Figures 5.7 and 5.8 show boxplots of the Dice-distribution for the CT-only model and the PET/CT model, respectively. These also demonstrate that the PET/CT model was preferable to the CT-only model. The median Dice was consistently higher for the PET/CT model and the spread was consistently lower.

This page is intentionally left blank.

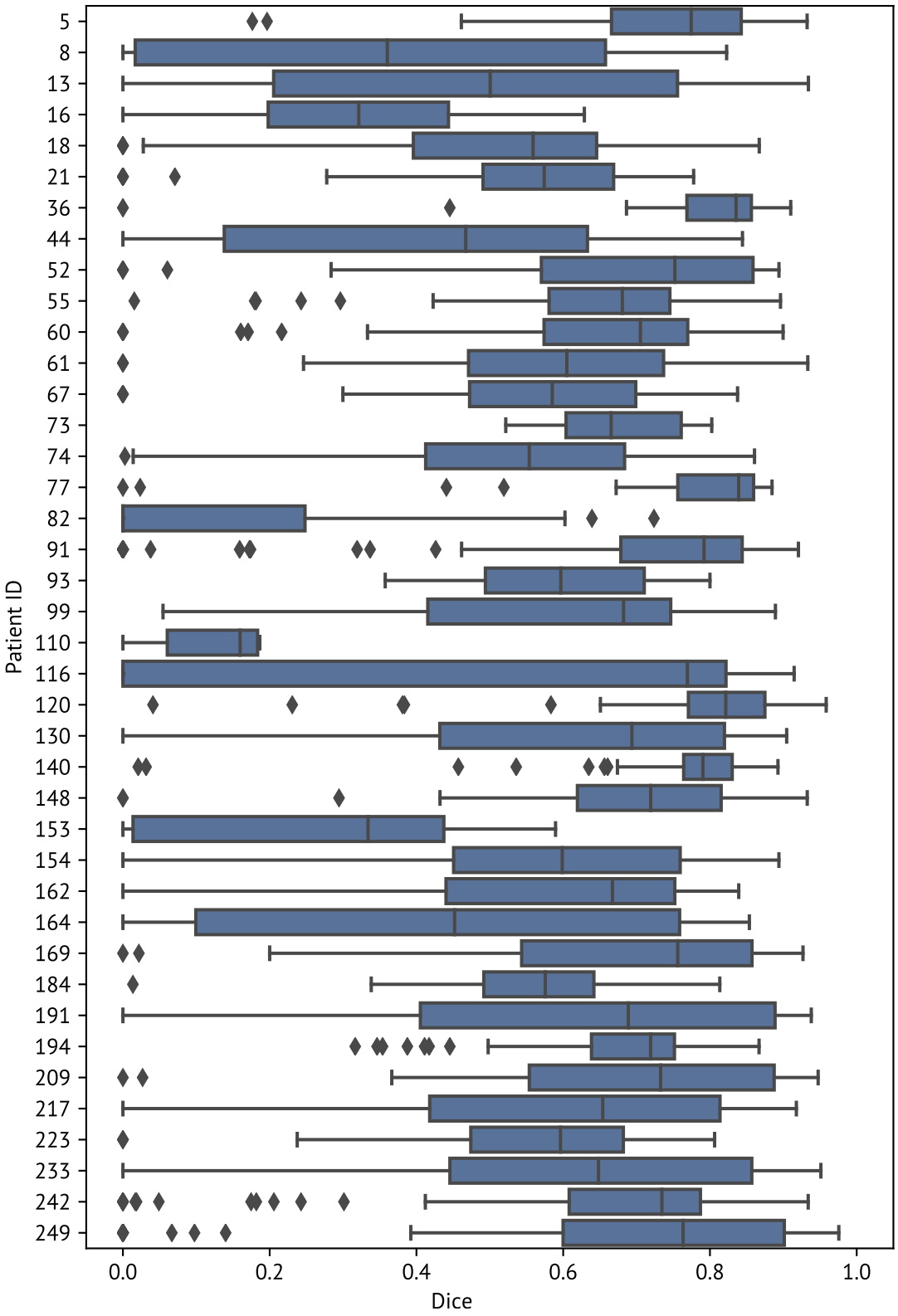


Figure 5.7: Boxplot illustrating the Dice distribution for the highest performing CT-only model (evaluated on the validation set) per slice for each patient on the test set. The line within each box represents the median, the box itself contains all data within the 25% quantile and the 75% quantile. The whiskers span the full dataset, excluding (automatically detected) outliers, which are marked with diamonds. Model specifications are given in Table 5.13

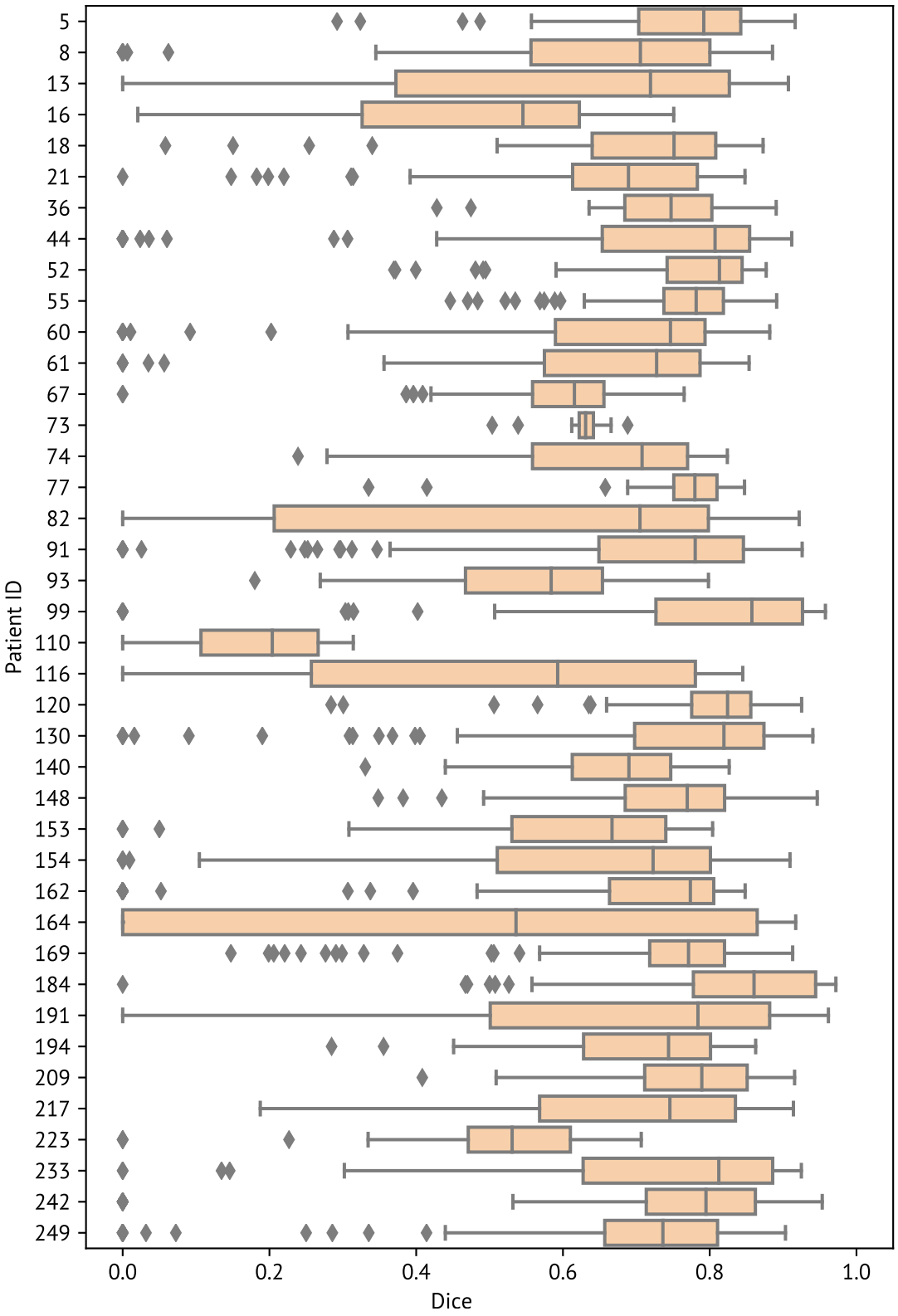


Figure 5.8: Boxplot illustrating the Dice distribution per patient for the highest performing PET/CT model (evaluated on the validation set) per slice for each patient on the test set. The line within each box represents the median, the box itself contains all data within the 25% quantile and the 75% quantile. The whiskers span the full dataset, excluding (automatically detected) outliers, which are marked with diamonds. Model specifications are given in Table 5.13

5.4 Visualisation of the segmentation masks

To assess the quality of the predicted segmentation masks, twelve linearly spaced slices were visualised for each patient in the validation and test set. A selection of patients are shown in this section. The input to the network are shown as images and the outline of the predicted and ground truth segmentation masks are given. Note that the dynamic range of the CT-only model and PET/CT model are different as the optimal window size for the CT-only model was not the same as that of the PET/CT model.

PET images were normalised so the maximal standardised uptake value amongst all patients were set as the maximum signal on all visualisation. A gamma transform with $\gamma = \frac{1}{10}$ was chosen for the PET images to enhance changes at small standardised uptake values. Note that the PET signal is not easily discernable when printed in grayscale.

Predicted delineations in the validation set

Both the PET/CT model and the CT-only model performed poorly on patient 177. There was no overlap between the predicted segmentation and the ground truth for the CT-only model (Figure 5.9). The PET/CT model, on the other hand, managed to find some of the affected tissue in early slices (Figure 5.10).

Note that in this patient, the tumour and the lymph nodes are difficult to discern in the images (at least for lay people). Some of the lymph nodes are discernable on the CT slices (on slice 64-72 and on slice 79 in Figure 5.9). However, they are not as easily visible as on several other patients. Furthermore, the PET signal was not enhanced in the tumour of lymph nodes (Figure 5.10).

The ground truth and predicted segmentation mask of patient 229 are shown in Figures 5.11 and 5.12. The PET/CT model performed somewhat poorly, with an average Dice of 0.57, and the CT-only model had an average Dice of only 0.40. Again, we see that both models struggled to delineate the affected tissue on the tongue. Additionally, there are some lymph nodes on slice 39 to 71 that the models struggled to delineate. One main difference between this patient and patient 177, is that this patient had a visible PET signal, which made it easier for the models to detect the regions of interest.

Figure 5.13 and Figure 5.14 shows the predicted segmentation masks for patient

98. In this case both models yielded high-quality segmentation masks with an average Dice of over 0.70. Furthermore, both models successfully segmented the lymph node on the early slices on all but the first two slices.

For patient 98, both the CT-only model and the PET/CT model delineated regions that did not appear in the original segmentation masks. Some of these regions (e.g. PET/CT, slice 99 in Figure 5.14) are clearly not lymph nodes, whereas others (e.g. CT-only slice 63 in Figure 5.13) may be. Finally, we note that the CT-only model successfully delineated the tumour on several slices with severe beam hardening artefacts (e.g. slice 124-136 Figure 5.13).

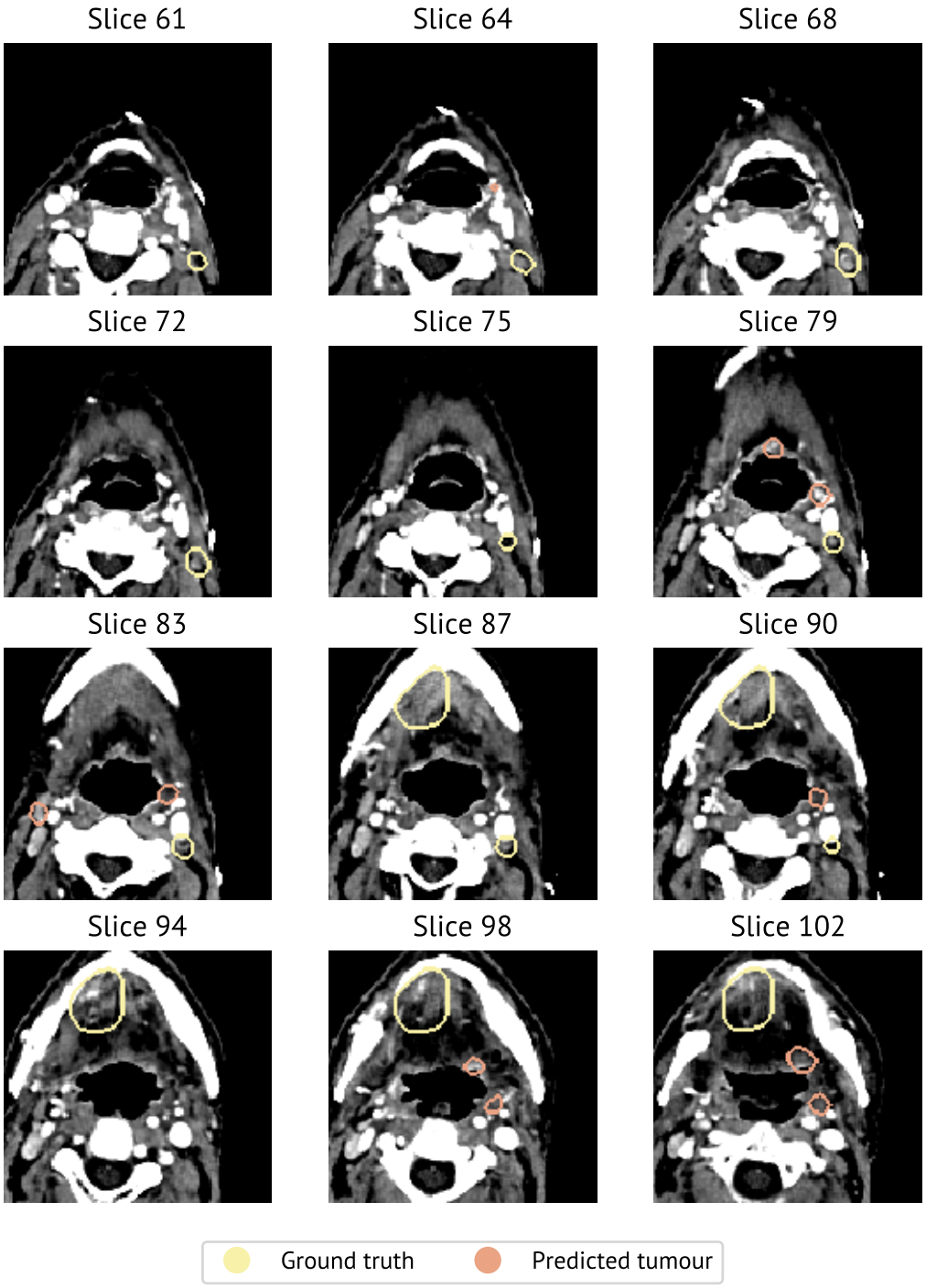


Figure 5.9: Slices showing the segmentation masks predicted by the CT-only model (Table 5.13) for patient 177. The ground truth is also given. The average Dice for this patient was 0.0 and the standard deviation was 0.0.

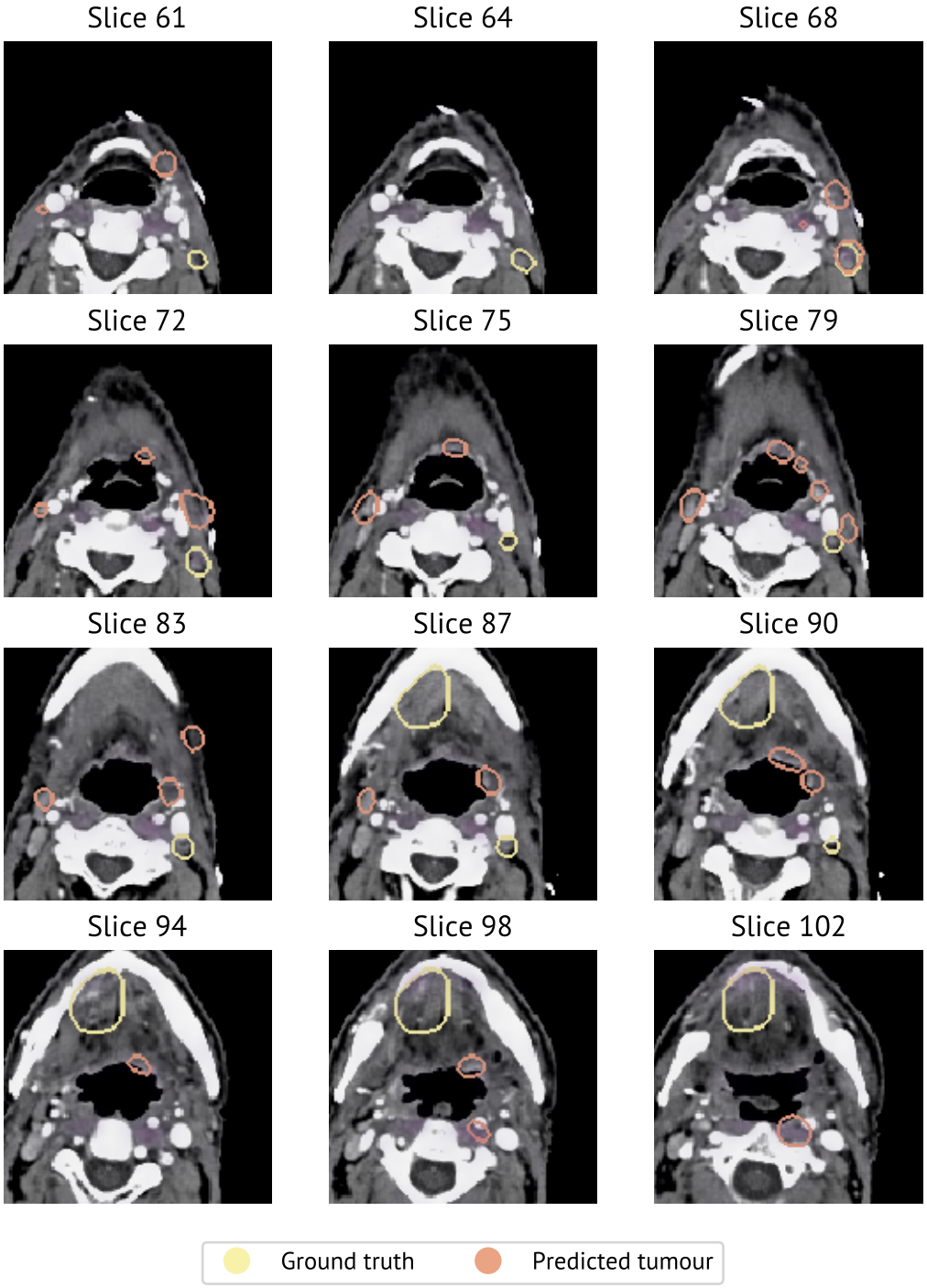


Figure 5.10: Slices showing the segmentation masks predicted by the PET/CT model (Table 5.13) for patient 177. The ground truth is also given. The average Dice for this patient was 0.08 and the standard deviation was 0.20. The PET-channel is gamma transformed with $\gamma = \frac{1}{10}$

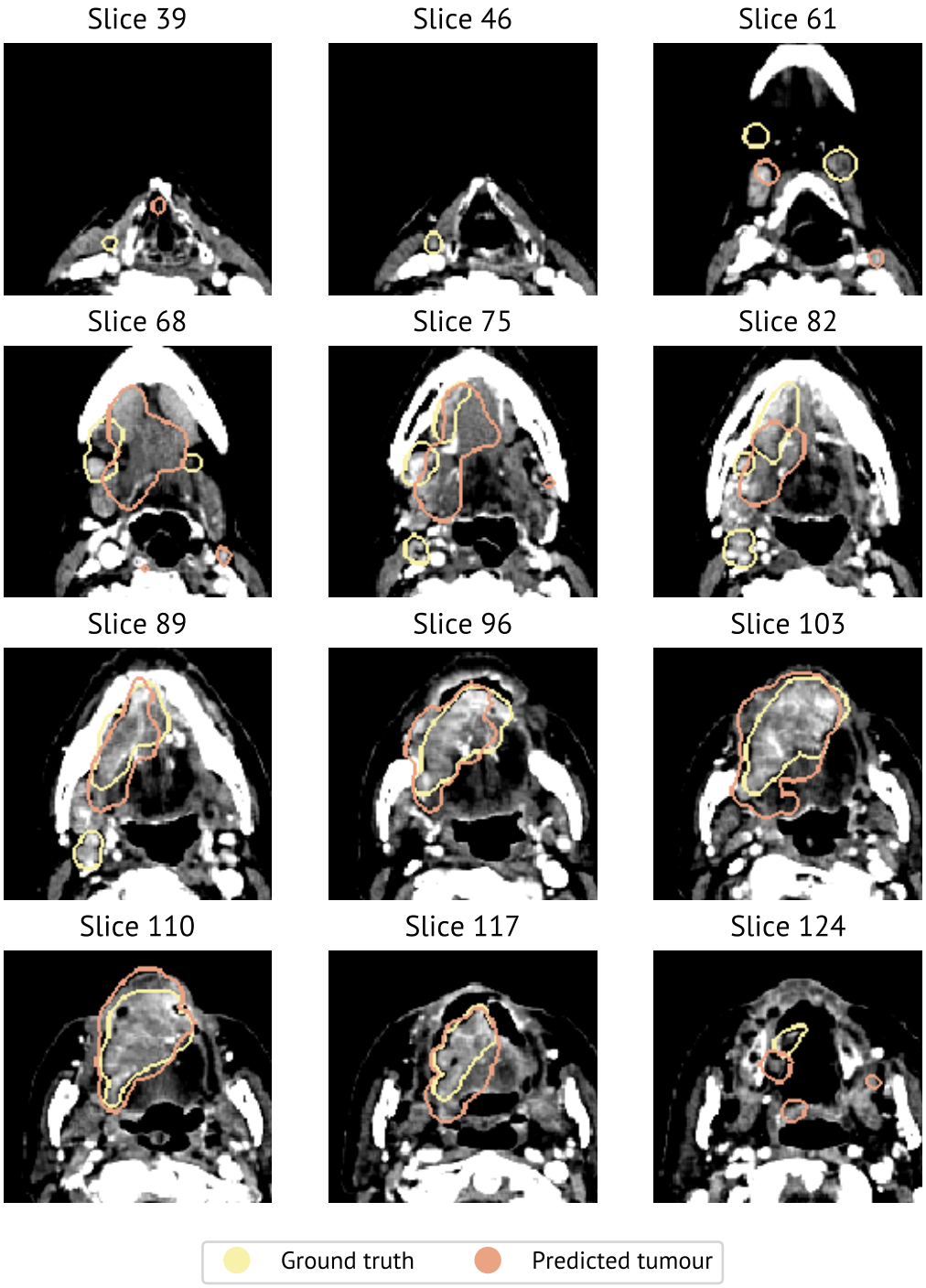


Figure 5.11: Slices showing the segmentation masks predicted by the CT-only model (Table 5.13) for patient 229. The ground truth is also given. The average Dice for this patient was 0.40 and the standard deviation was 0.33.

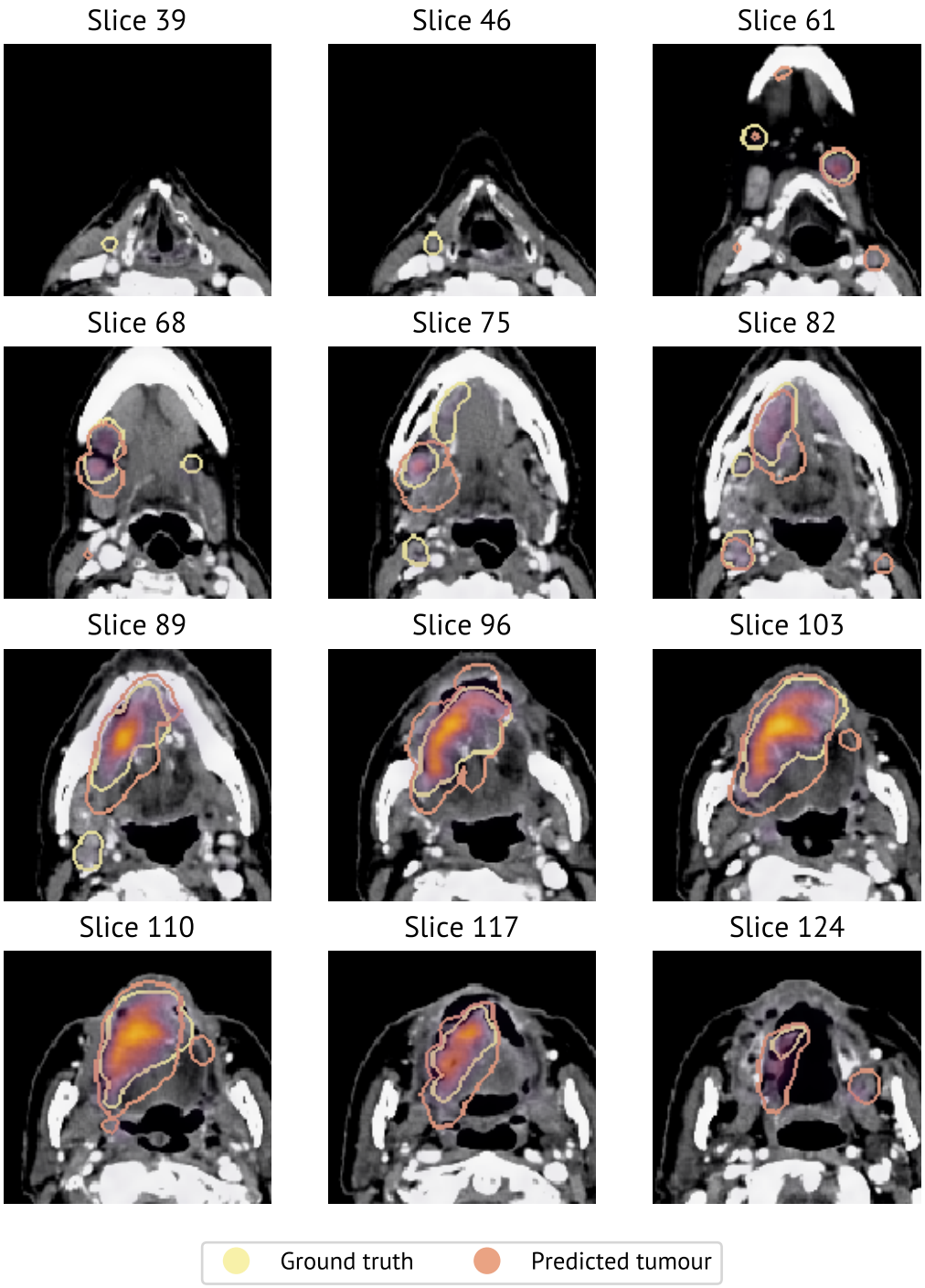


Figure 5.12: Slices showing the segmentation masks predicted by the PET/CT model (Table 5.13) for patient 229. The ground truth is also given. The average Dice for this patient was 0.57 and the standard deviation was 0.25. The PET-channel is gamma transformed with $\gamma = \frac{1}{10}$

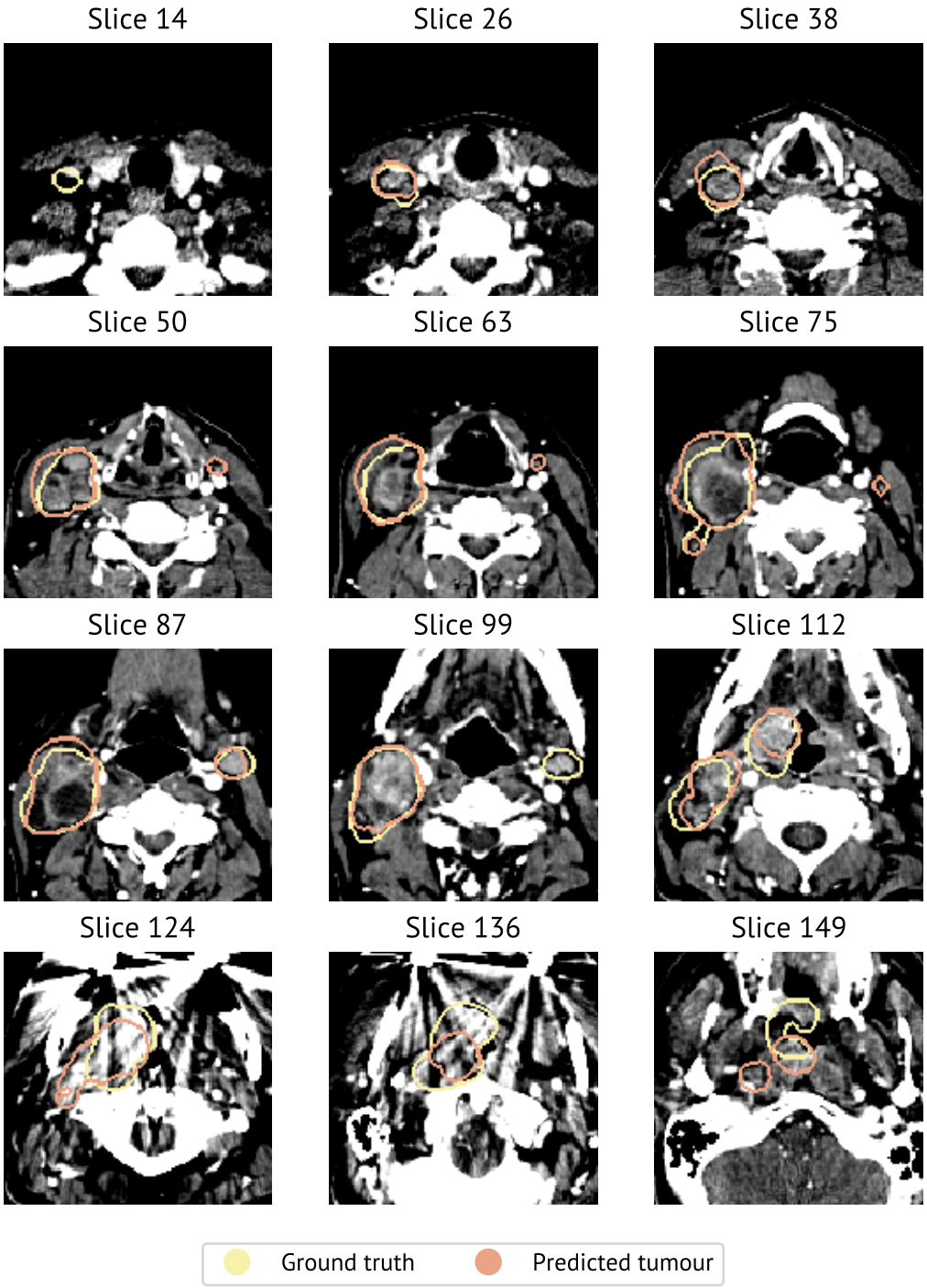


Figure 5.13: Slices showing the segmentation masks predicted by the CT-only model (Table 5.13) for patient 98. The ground truth is also given. The average Dice for this patient was 0.73 and the standard deviation was 0.17.

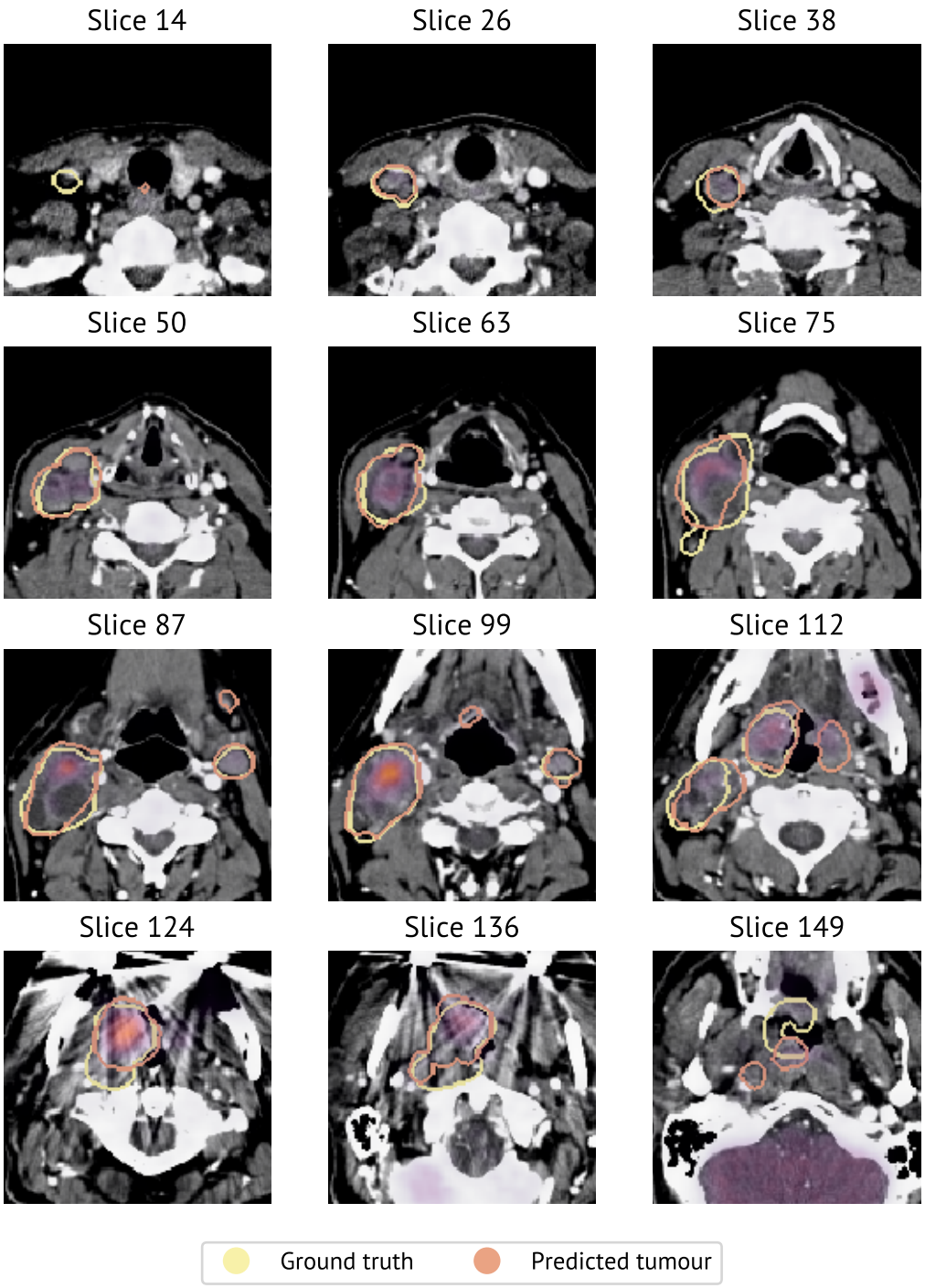


Figure 5.14: Slices showing the segmentation masks predicted by the PET/CT model (Table 5.13) for patient 98. The ground truth is also given. The average Dice for this patient was 0.77 and the standard deviation was 0.16. The PET-channel is gamma transformed with $\gamma = \frac{1}{10}$

This page is intentionally left blank.

Predicted delineations on the test set

The segmentation masks of several patients in the test set were also visualised. Firstly, we consider the segmentation masks for patient 5, shown in Figures 5.15 and 5.16. Again, both the PET/CT model and the CT-only model performed well on all slices, including those with considerable beam hardening artefacts.

Figures 5.17 and 5.18 show the segmentation masks of patient 110. Both the PET/CT and CT-only models delineate a region on the right hand side of the patient where contrast agent has accumulated. Consequently, the Dice score is close to zero. This large region is not part of the original region of interest, thereby reducing the performance severely. Furthermore, both models delineate a small region of tissue on the left hand side of the patient not delineated in the ground truth.

Both models achieved highest performance for patient 120, shown in Figures 5.19 and 5.20. There are some falsely discovered small lymph nodes in these segmentation masks (slice 59 in Figure 5.19 and slice 32 in Figure 5.20).

Finally, the predicted segmentation masks of patient 249 are shown in Figures 5.21 and 5.22. The performance for the PET/CT model and CT-only model were the same for this patient, with the CT-only model performing well over average and the PET/CT model performing slightly above average.

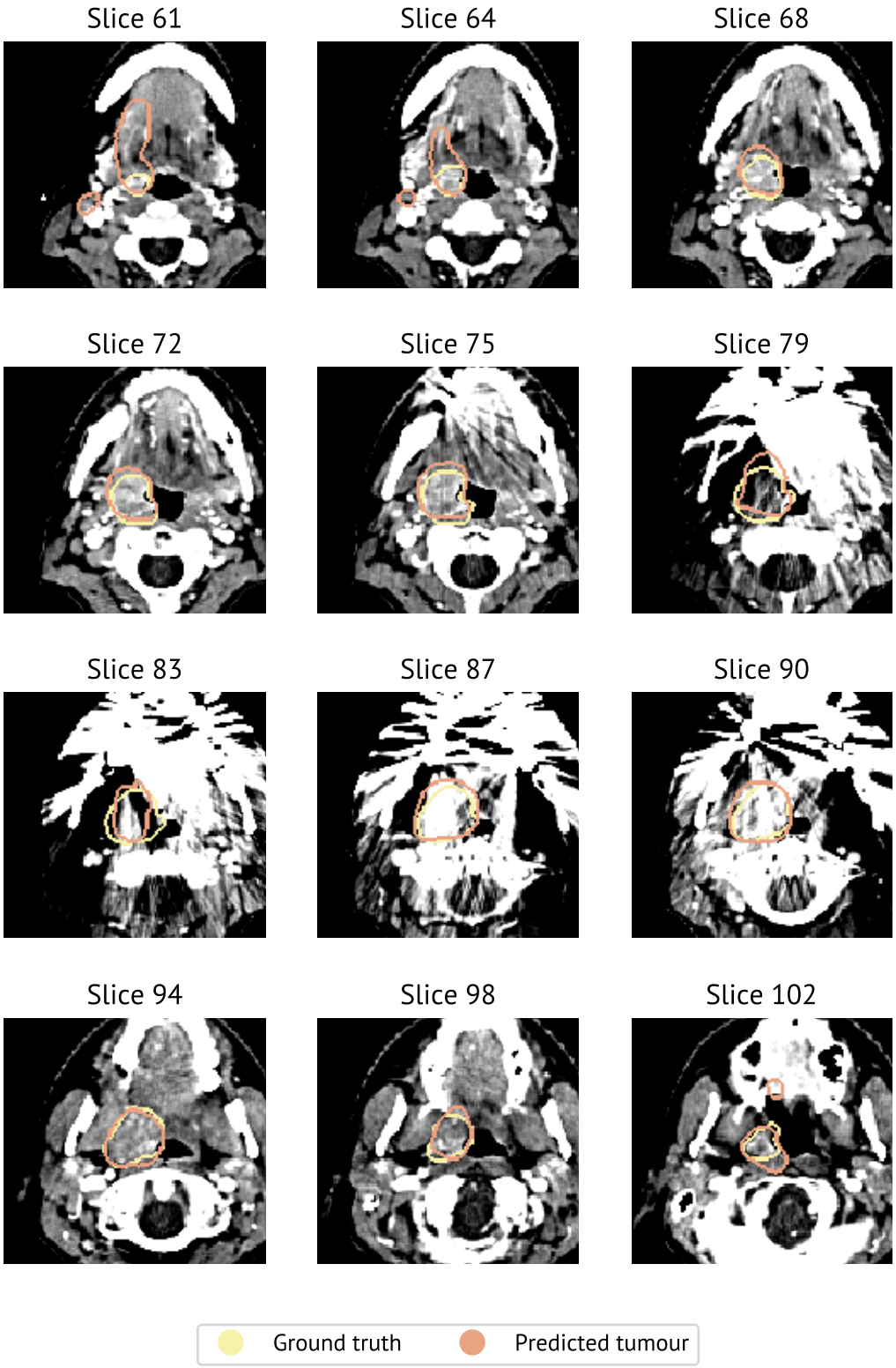


Figure 5.15: Slices showing the segmentation masks predicted by the CT-only model (Table 5.13) for patient 5. The ground truth is also given. The average Dice for this patient was 0.74 and the standard deviation was 0.17.

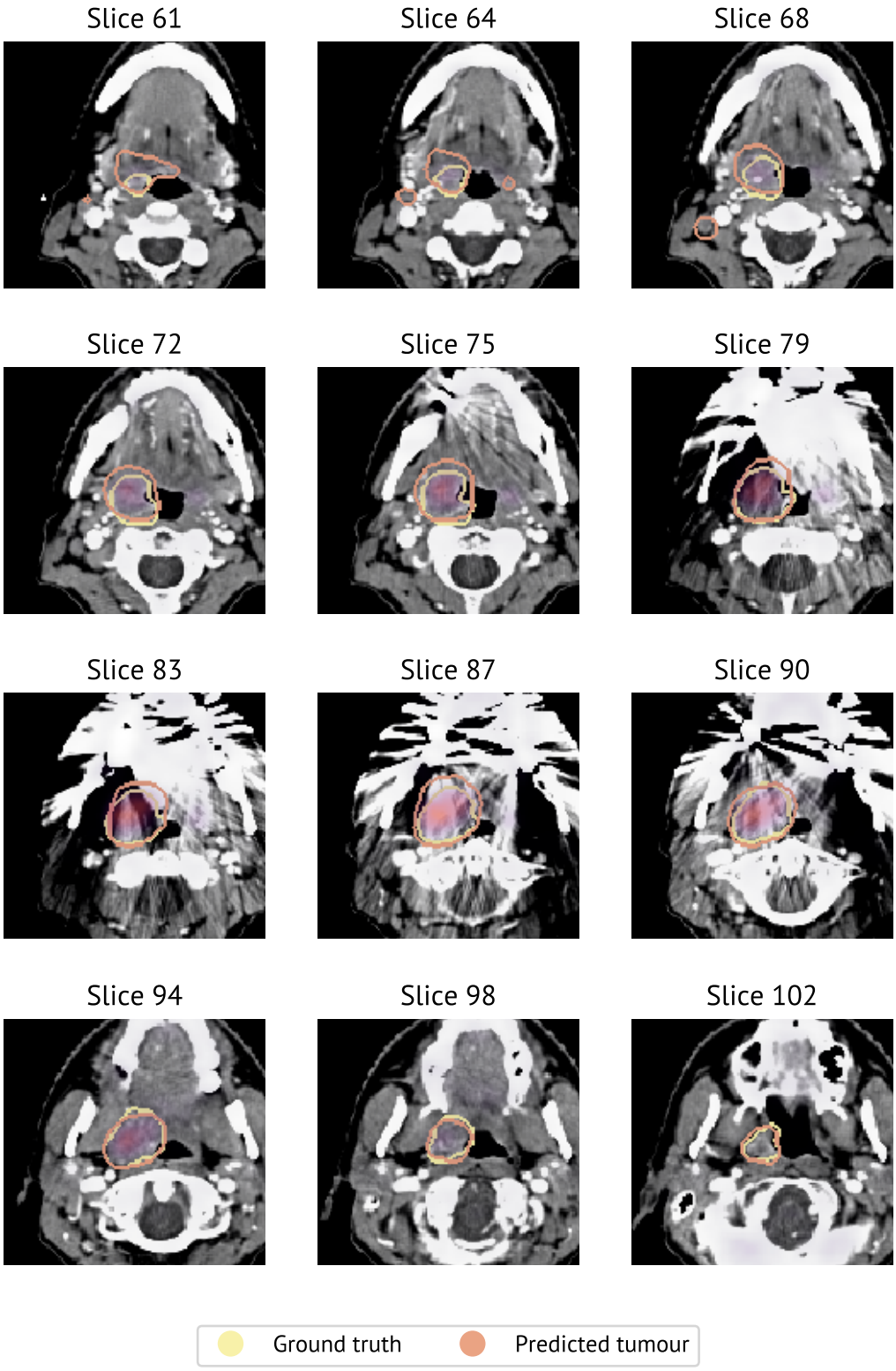


Figure 5.16: Slices showing the segmentation masks predicted by the PET/CT model (Table 5.13) for patient 5. The ground truth is also given. The average Dice for this patient was 0.74 and the standard deviation was 0.15. The PET-channel is gamma transformed with $\gamma = \frac{1}{10}$

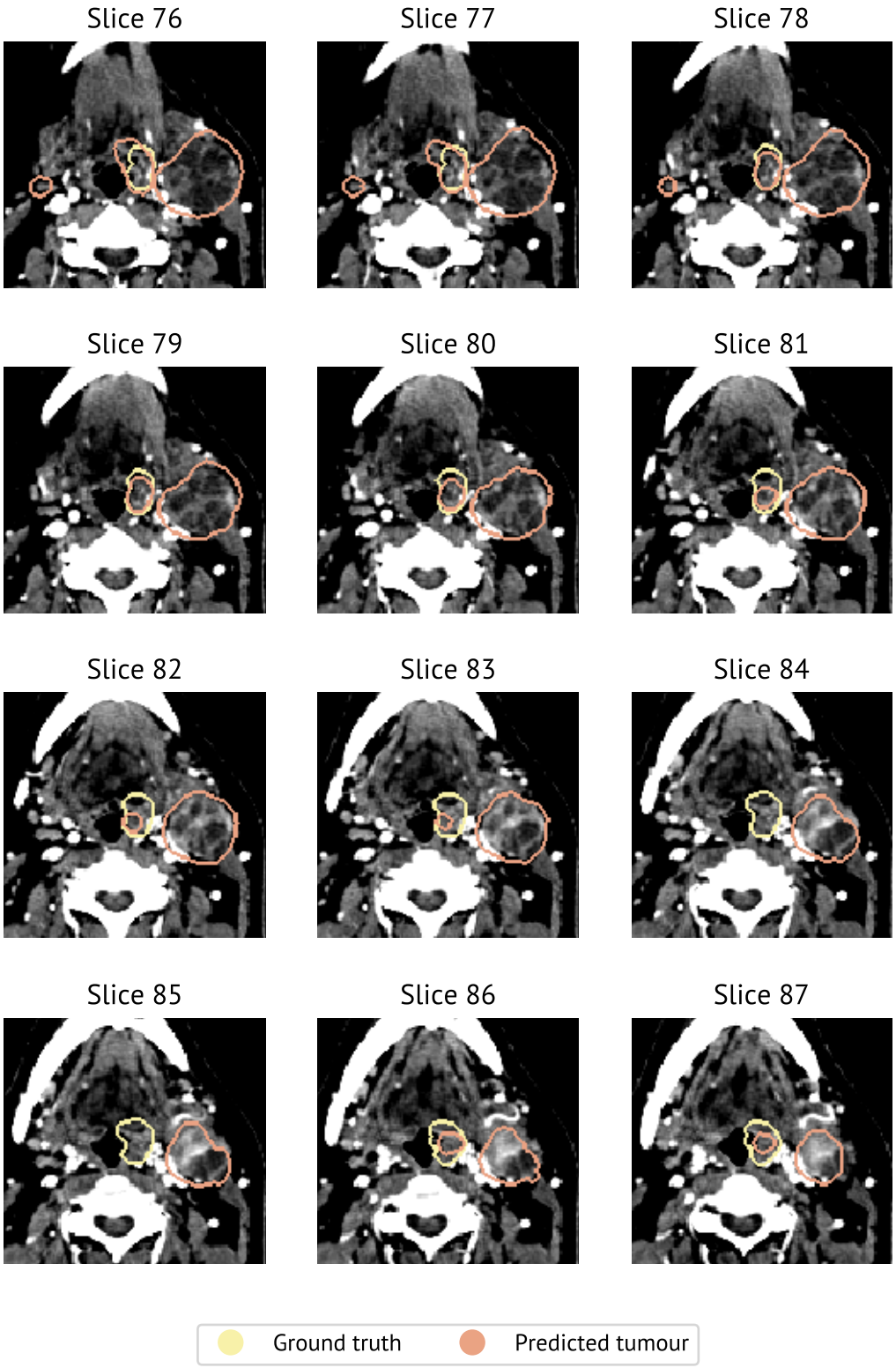


Figure 5.17: Slices showing the segmentation masks predicted by the CT-only model (Table 5.13) for patient 110. The ground truth is also given. The average Dice for this patient was 0.12 and the standard deviation was 0.072.

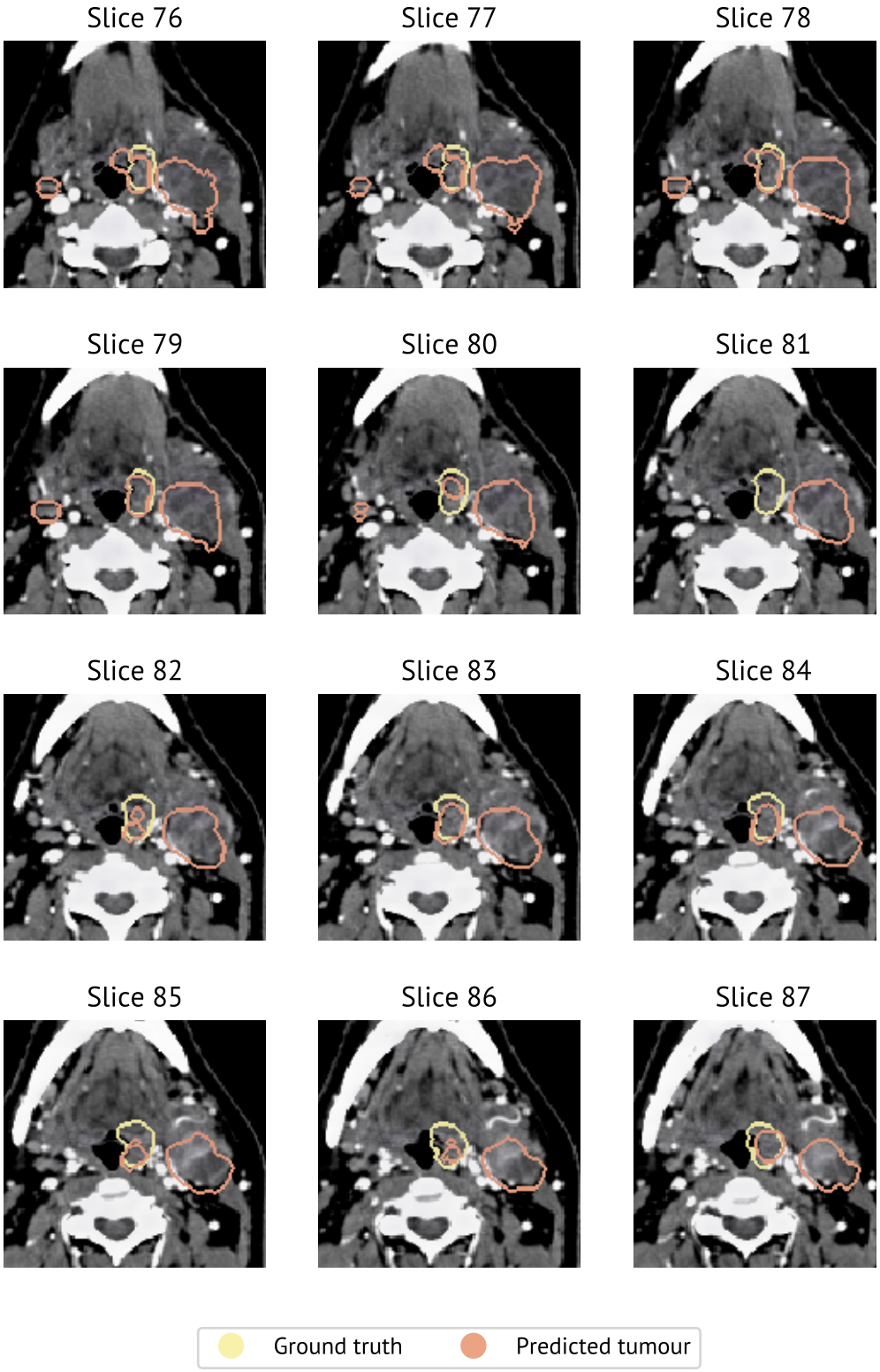


Figure 5.18: Slices showing the segmentation masks predicted by the PET/CT model (Table 5.13) for patient 110. The ground truth is also given. The average Dice for this patient was 0.19 and the standard deviation was 0.094. The PET-channel is gamma transformed with $\gamma = \frac{1}{10}$

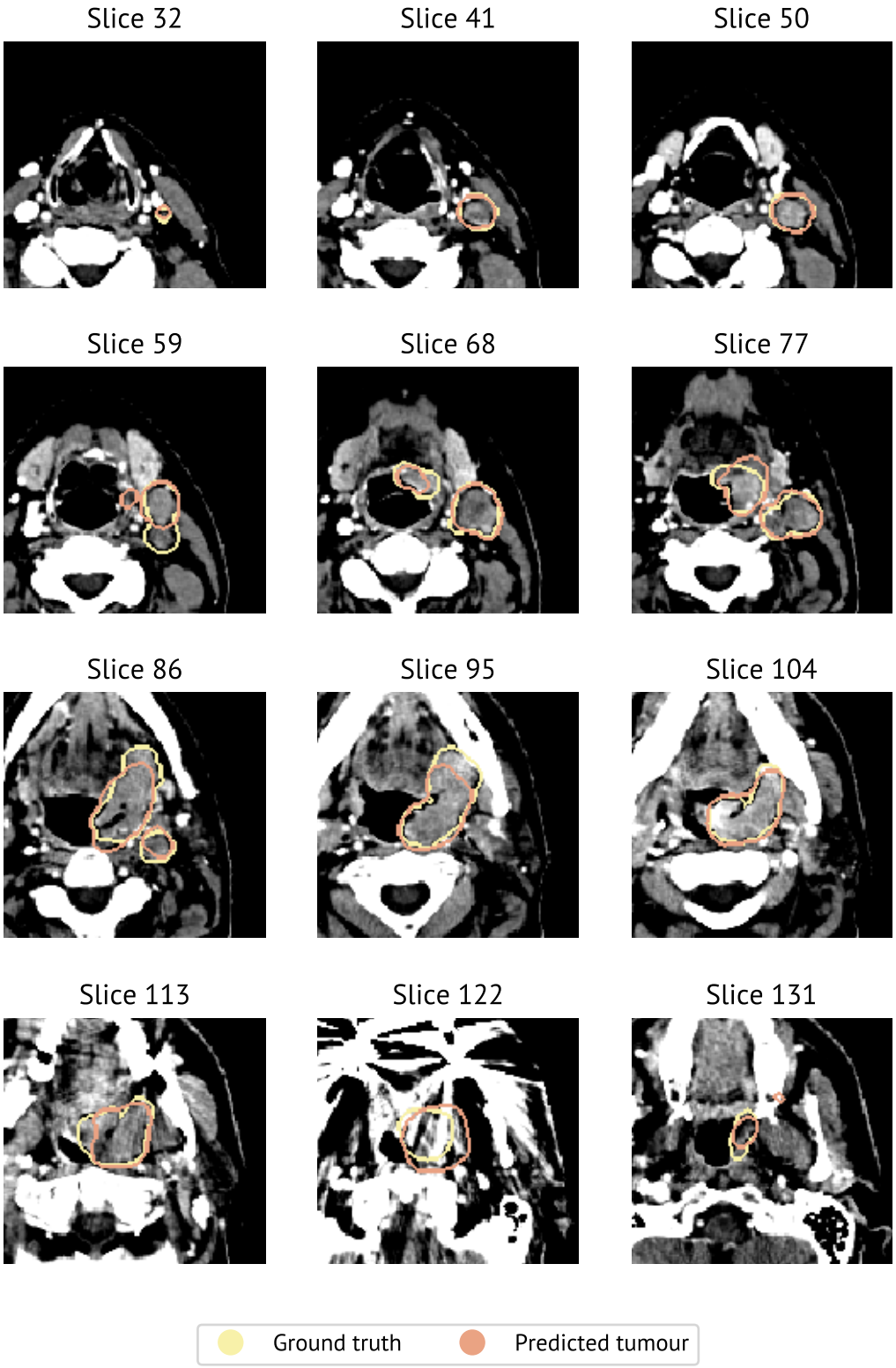


Figure 5.19: Slices showing the segmentation masks predicted by the CT-only model (Table 5.13) for patient 120. The ground truth is also given. The average Dice for this patient was 0.80 and the standard deviation was 0.14.

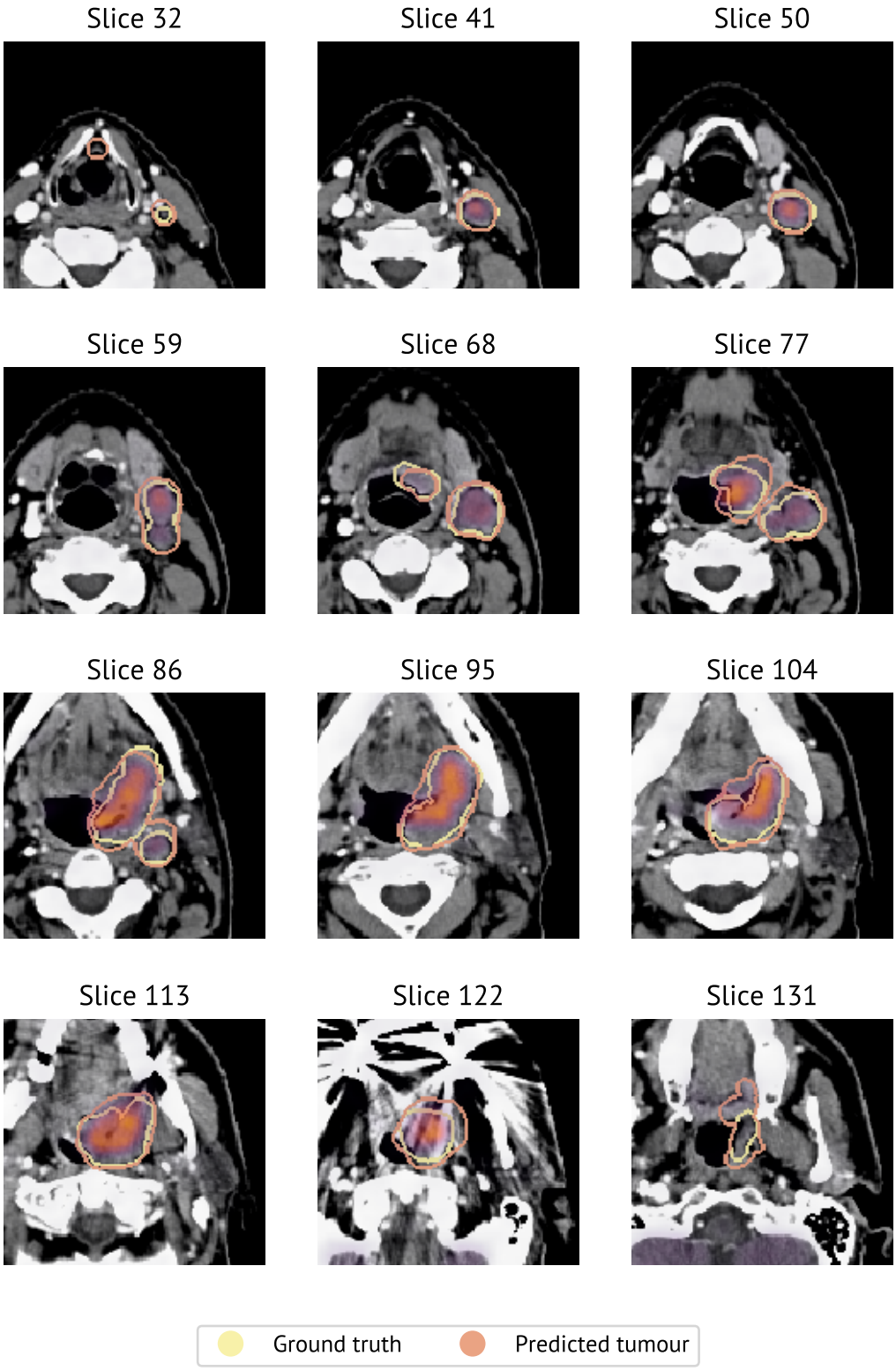


Figure 5.20: Slices showing the segmentation masks predicted by the PET/CT model (Table 5.13) for patient 120. The ground truth is also given. The average Dice for this patient was 0.80 and the standard deviation was 0.10. The PET-channel is gamma transformed with $\gamma = \frac{1}{10}$

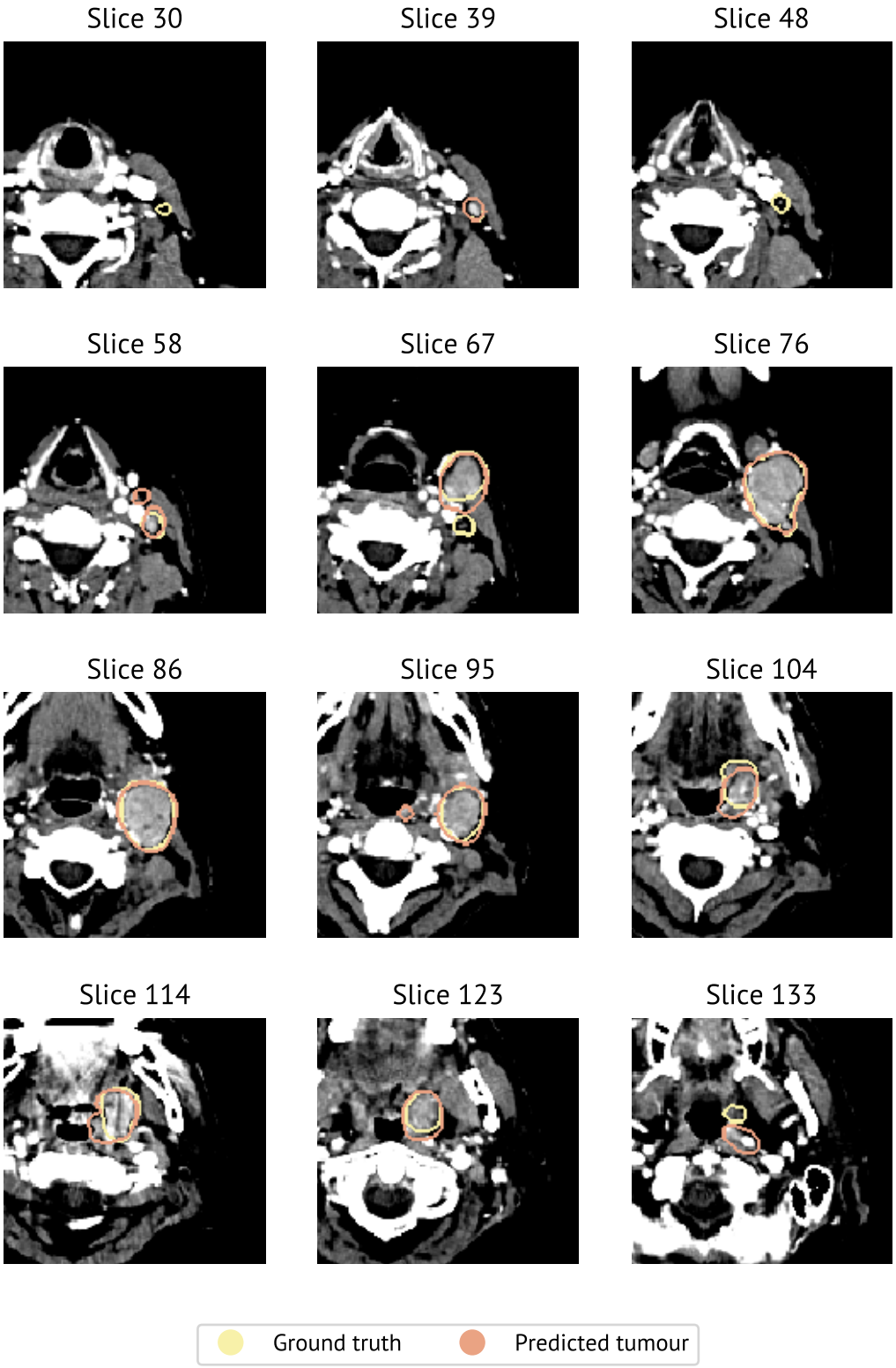


Figure 5.21: Slices showing the segmentation masks predicted by the CT-only model (Table 5.13) for patient 249. The ground truth is also given. The average Dice for this patient was 0.67 and the standard deviation was 0.30.

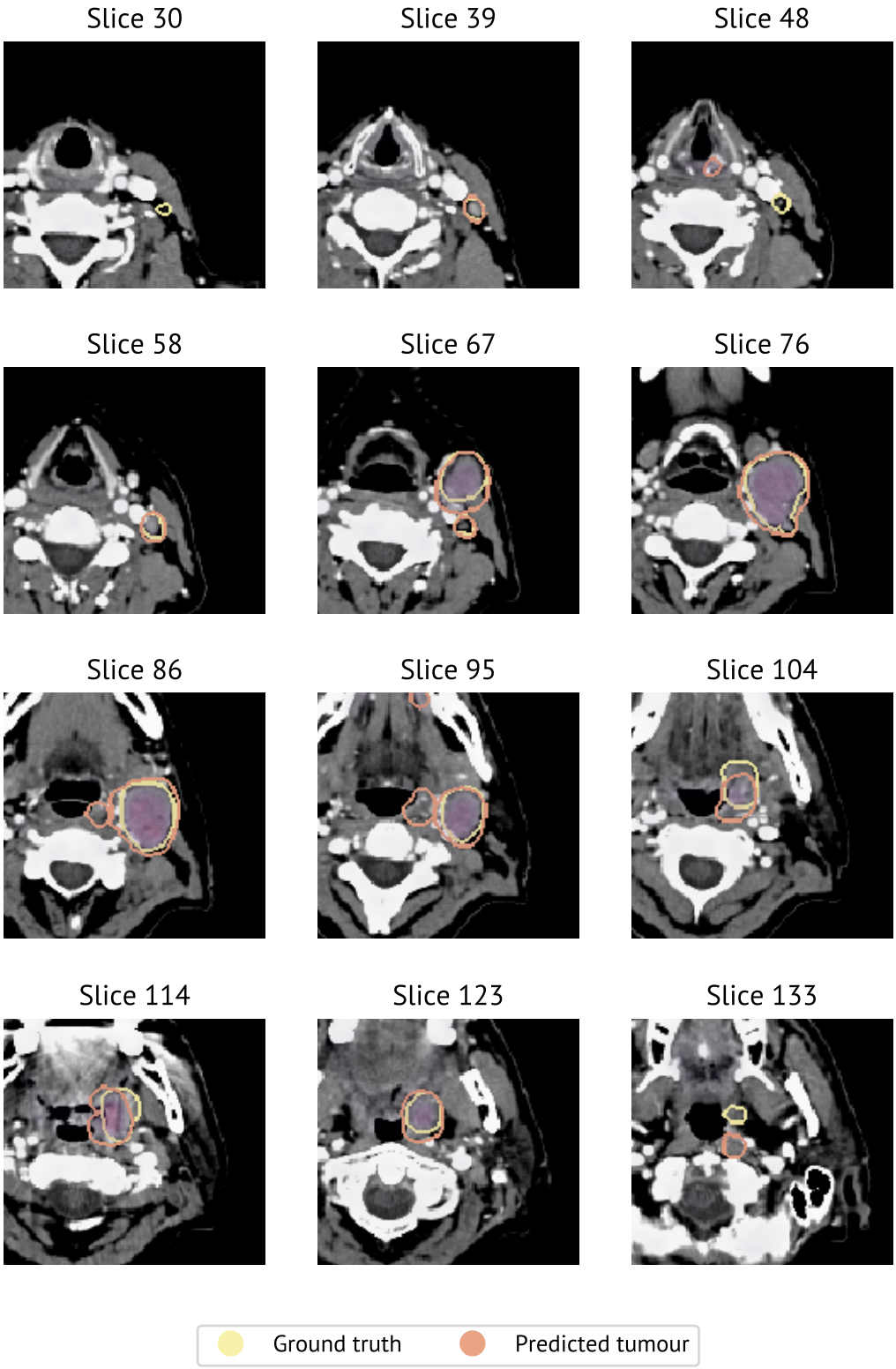


Figure 5.22: Slices showing the segmentation masks predicted by the PET/CT model (Table 5.13) for patient 249. The ground truth is also given. The average Dice for this patient was 0.67 and the standard deviation was 0.24. The PET-channel is gamma transformed with $\gamma = \frac{1}{10}$

Chapter 6

Discussion

6.1 Model hyperparameters

We start by discussing the effect of hyperparameters on the neural network performance[1]. It is clear, from the tables in Sections 5.1.1 and 5.1.2, that the choice of hyperparameters influenced the model performance.

The initial plan for this analysis was to use ANOVA+post-hoc tests to compare the different modalities. However, when studying the distribution of data points, it became apparent that the results were not normally distributed and thus, the assumptions of ANOVA were not fullfilled. ANOVA tests were therefore not conducted.

6.1.1 Assessment of the loss functions

It is apparent that choosing the correct loss function is a key part of maximising model performance. Table 5.2 demonstrates that using the cross entropy loss instead of the F_2 or F_4 loss led to an expected drop in Dice performance by 0.03 units.

The difference between the F_1 (Dice) loss versus the cross entropy loss, however, was notably low. In 'V-net: Fully convolutional neural networks for volumetric medical image segmentation', Milletari *et al.* [42] demonstrate that the Dice loss

[1]Recall from the previous chapter, that performance meant Dice per slice.

leads to consistently higher Dice than the cross entropy loss. In our analysis, this was not the case. Figure 5.2 on page 122 might support this claim, as it demonstrates that using a Dice loss can result in higher performance than the cross entropy loss. However, the effect is so small that it can be random, especially when we consider that more experiments were run with the F_1 loss than the cross entropy loss (Table 5.8 on page 124).

The effect of using the F_2 or F_4 loss as opposed to the F_1 or cross entropy loss was greater. The F_2 and F_4 loss yielded an expected increase in Dice performance of $0.02-0.04$ for all modalities (most for CT-only, Table 5.8). Additionally, Table 5.2 shows that the F_2 and F_4 loss were better than the F_1 and cross entropy loss on summary statistics of the performances. Thus, the generalised F_β loss introduced in this thesis is indeed preferable to the cross entropy loss and Dice loss.

To understand the difference between the Dice loss and the F_β loss for higher values of β, we need to look at the definition of the F_β metric (Definition 2.3.9 on page 57). The F_β metric is analogous to the Dice score, except that it weighs sensitivity β times more than the precision. Thus, the increase in Dice performance observed when optimising for higher values of β signifies that the generalisation gap for sensitivity is larger than that for precision (i.e. the performance decrease between the train-set and the validation/test set for sensitivity is larger than that for precision).

Furthermore, choice of loss function had a larger effect on the performances of the CT-only models than on the performances of the PET/CT and PET-only models. This indicates that the generalisation gap in sensitivity is larger for CT-only models than for PET/CT and PET-only models. Testing this hypothesis is unfortunately outside the scope of this project, and further experiments should be conducted to test this hypothesis.

Finally, we propose a method for choosing the correct β value when training neural networks. First, train a model using the Dice loss, then inspect the generalisation gap for the sensitivity and PPV. If the generalisation gap is larger for sensitivity, increase the β value, if the generalisation gap is larger for PPV, decrease the β. Algorithm 6.1 demonstrates this algorithm in practice.

There are two weaknesses with the proposed algorithm. The first is how actively the validation set is being used, which most likely will lead to overfitting on the training set. We therefore recommend using only parts of the validation set for this procedure.

Furthermore, training a network with a specific value of β is not deterministic

Algorithm 6.1 Optimal β search for the F_β loss.

procedure OPTIMALβSEARCH($\mathcal{T}$, ϵ)
    ▷ **Initiate β search**
        $\beta_0 \leftarrow 1$
        $\beta_1 \leftarrow 1$
        Train model with F_{β_0} loss.
        Compute validation PPV and Sensitivity, store in
        PPV_0 and Sensitivity_0.
        $\text{PPV}_1 \leftarrow \text{PPV}_0$
        $\text{Sensitivity}_1 \leftarrow \text{Sensitivity}_0$
        while $\text{PPV}_0 < \text{Sensitivity}_0$ **do**
            $\beta_0 \leftarrow 0.5\beta_0$
        while $\text{PPV}_1 > \text{Sensitivity}_1$ **do**
            $\beta_1 \leftarrow 2\beta_1$
    ▷ **Perform β search**
        while $\beta_1 - \beta_0 > \epsilon$ **do**
            $\beta^\star \leftarrow 0.5\beta_0 + 0.5\beta_1$
            $\text{PPV}^\star \leftarrow$ PPV on validation set
            $\text{Sensitivity}^\star \leftarrow$ Sensitivity on validation set
            if $\text{PPV}^\star > \text{Sensitivity}^\star$ **then**
                $\beta_1 \leftarrow \beta^\star$
            else
                $\beta_2 \leftarrow \beta^\star$
    return $\beta^\star$

(because of random batches, initialisation, etc). Thus, the computed sensitivity and PPV might not be representative. However, training the model several times is often computationally infeasible. The non-determinicity of neural network training is therefore a potential weakness of the proposed algorithm for finding β.

The proposed method of finding β for the F_β loss is, in other words, not without its weaknesses and it should therefore be tested in practice. Unfortunately, testing this algorithm was outside the scope of this project and should be done in future works.

6.1.2 Layer type selection

It is clear, from Table 5.1 on page 120, that choosing ResNet layer types did not work for U-Net architectures. The reason for this, however, is less clear.

One possible reason for why ResNet layers yielded suboptimal results is the exploding gradient problem (see page 35). Batch normalisation will often alleviate this problem [11]. However, the 1×1 convolutions performed in the skip-connection whenever the number of channels was reduced were not normalised. Hence, they might lead to an exploding gradient problem. The fact that the partial derivatives with respect to the skip-connections were significantly higher than those with respect to the residual connections support this hypothesis.

However, the only way to test the exploding gradient hypothesis is to run experiments with normalised skip connections. Luckily, this is easily done with the SciNets package, as it is only necessary to overload the `generate_skip_connection` method of the `ResNet` layer class. Thus, such experiments are cheap with respect to the number of man-hours required and should be done in later analyses.

6.1.3 Optimiser selection

The hyperparameter sweep was performed using an Adam optimiser, as it is known to be a very fast optimiser [44]. The highest performing models were then trained with the SGDR+momentum algorithm. However, the SGDR+momentum runs converged after approximately the same number of epochs as the Adam runs. Using Adam for exploration first was, in other words, not neccessary.

Unfortunately, only four SGDR+momentum models were trained with two differ-

ent hyperparameter combinations, in order to reduce the number of experiments conducted. If both optimisers were tested for the entire dataset, the total training time would be approximately two months. It was, in other words, not possible to compare Adam to SGDR+momentum with as many hyperparameters as were tested in this project.

We do, however, still recommend that future tumour delineation experiments are run using the SGDR+momentum. The main reason for this is the SGDR+momentum optimiser's theoretical advantages over Adam [73], [74]. Furthermore, Table 5.12 on page 126 demonstrates that the SGDR+momentum optimiser achieved the same or better results at the same speed as the Adam optimiser on the validation set for both hyperparameter configurations tested.

An observant reader will notice that the learning rate used for the Adam experiments is either a factor 10 or a factor 100 lower than the recommended learning rate [44]. Such small learning rates were chosen because preliminary experiments showed that the networks trained poorly with a higher learning rate. Experiments with a learning rate of 10^{-5} converged to an optimum with poorer generalisation capabilities (Table 5.4 on page 120). Thus, a more in-depth parameter sweep of the learning rate might yield even better generalisation capabilities.

On the other hand, training with SGDR+momentum yielded similar generalisation performance as training with Adam. Hence, the generalisation achieved with Adam and a learning rate of 10^{-4} might be close to the optimum. Consequently, we cannot conclude whether the experiments would improve with a higher learning rate, but testing a lower one is most likely not necessary.

Increasing the batch size has the effect of increasing the stability when training. Increasing the batch size might, therefore, improve the quality of the models and reduce the time taken to train them. The batch size used in this project was chosen dependent on the available VRAM, increasing it was, therefore, not possible while training on a single GPU.

Furthermore, gradually increasing the batch size while training has been demonstrated to be preferable compared to reducing the learning rate [93]. Increasing the batch size this way leads to high exploration of the parameter space in early iterations and more accurate optimisation later.

6.1.4 Assessment of preprocessing parameters

Hounsfield windowing yielded an expected performance improvement of 0.02 for both CT-only and PET/CT models. Additionally, both the highest and lowest performing models with windowing achieved a Dice score approximately 0.03 higher than the highest and lowest performing models without preprocessing, respectively. Thus, it is clear that using Hounsfield windowing leads to improved performance.

Hounsfield windowing can be expressed as a single-layer neural network with ReLU nonlinearities. Thus, the models trained without windowing should be able to learn to perform the windowing. The fact that they did not learn this suggests that there was not enough data at hand.

A logical next step would be to learn the optimal windowing parameters automatically. To demonstrate that this is possible, we will derive a layer that does this using only the ReLU function and additions. Thereafter, we will derive the gradient of this layer.

Let w represent the window width and c represent the window centre. Performing Houndsfield windowing ($HW(\boldsymbol{x})$) on an image ($\boldsymbol{x}$) is then equivalent to the following operation:

$$HW(\boldsymbol{x}) = \min\left(c + \frac{w}{2}, \max\left(c - \frac{w}{2}, \boldsymbol{x}\right)\right). \tag{6.1}$$

We can then use the fact that $\max(a, b) = max(0, b - a) + a$, the definition of the ReLU nonlinearity ($\breve{f}_{ReLU}(\boldsymbol{x}) = \max(0, \boldsymbol{x})$) and the fact that $\min(a, b) = -\max(-a, -b)$ to get

$$HW(\boldsymbol{x}) = c + \frac{w}{2} - \breve{f}_{ReLU}\left(w - \breve{f}_{ReLU}\left(\boldsymbol{x} + \frac{w}{2} - c\right)\right), \tag{6.2}$$

which is differentiable with respect to w and c almost everywhere.

To compute the gradient of HW, we start by noting that the derivative of $\breve{f}_{ReLU}(x)$ is given by the following equation,

$$\frac{d\breve{f}_{ReLU}(x)}{dx} = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x < 0 \end{cases}. \tag{6.3}$$

Thus, the partial derivative of HW with respect to c is given by

$$\frac{\partial HW(x)}{\partial c} = \begin{cases} 1 & \text{if } c - \frac{w}{2} > x \text{ or } x > c + \frac{w}{2} \\ 0 & \text{if } c - \frac{w}{2} < x < c + \frac{w}{2} \end{cases}. \tag{6.4}$$

Similarly, the partial derivative of HW with respect to w is given by

$$\frac{\partial HW(x)}{\partial w} = \begin{cases} -0.5 & \text{if } c - \frac{w}{2} > x \\ 0 & \text{if } c - \frac{w}{2} < x < c + \frac{w}{2} \\ 0.5 & \text{if } c + \frac{w}{2} < x \end{cases}. \tag{6.5}$$

Thus, the windowing parameters are learnable.

6.1.5 Architecture selection

The U-Net architecture [23] was chosen because of its simplicity. It was relatively easy to implement, reducing the development phase of the project. As such, other architectures might yield better results.

One example of another architecture aimed at segmentation is the Large Kernel Matters architecture (LKM) [24]. It works similarly to U-Net [23], using long-range skip connections. However, LKM includes global, low-level information in these skip connections through the use of *Global Convolutional Networks* (GCNs). A GCN is essentially a layer consisting of a constrained convolution with kernel size 15 and no nonlinearity (for more information, see [24]). This architecture was not tested to reduce development time.

Another architecture that can be used for segmentation purposes is the DeepLab family [25]–[27]. In these architectures, the downsampling operations are replaced with dilated convolutions, thus, reducing the amount of discarded information. This architecture was considered. However, the memory footprint increases considerably by not using downsampling. Therefore, training with deeplab architectures was deemed infeasible in the scope of this project.

Finally, we have the V-net architecture [42]. This architecture is essentially the same as U-Net, but with 3D convolutions instead of 2D convolutions. Unfortunately, this leads to a large memory footprint, making training on GPU impossible. As a consequence, training models with 3D convolution would take too long time for the hyperparameter sweep to be feasible.

There are several ways to reduce the memory footprint of the architectures mentioned above. One method is by downscaling the input. However, by doing this, we discard valuable information. Another method is to divide each image into several pieces, and use each piece as an image instead. Unfortunately, this would increase the development phase excessively. Finally, we can reduce the batch sizes and

use batch renormalisation [94] to alleviate the instability of batch normalisations with small batches. Unfortunately, this involves two additional hyperparameters to tune. These methods should be tested in further work.

6.1.6 Hyperparameter recommendations

The experiments performed in this project resulted in some recommendations choosing for choosing hyperparameters for segmenting tumours and lymph nodes in PET/CT images. Firstly, improved ResNet layers are not recommended, as this layer type yielded exploding gradients.

Moreover, the generalised F_β loss should be used, and values for β between two and four yielded good results. Alternatively, β can be chosen by inspecting the generalisation gap for PPV and sensitivity, as described in Section 6.1.1 on page 160.

To optimise the loss, the SGDR+momentum algorithm should be used. If this algorithm is either too slow or does not result in acceptable performance, then the Adam algorithm can be tested with a learning rate of 10^{-4}.

Finally, we recommend using Hounsfield windowing. In this project, it was found that using the mean or median Hounsfield unit of the tumour as window centre and a window width of $100HU - 200HU$ leads to adequate performance. However, other windowing parameters should be tested.

6.1.7 Further work in hyperparameter exploration

There are three venues for further exploration of hyperparameters. The automatic learning of windowing parameters should be tested. Furthermore, adding batch normalisation on the ResNet layers should be considered to test whether or not ResNet layers are applicable with U-Net architectures. And lastly, the algorithm for finding correct β value for the F_β loss should be tested.

Furthermore, no regularisation (except for early stopping) was used. Dropout regularisation is known to have beneficial effects [67] and should therefore be tested. L_2 regularisation was not considered as it loses its regularising effect when used in tandem with batch normalisation [63].

The preprocessing pipeline was also not explored in detail. Currently, the only form

of preprocessing was windowing. However, several other forms of preprocessing operators should be tested. Specifically, data augmentation methods, such as random rotations and mirroring of the images, should be tested as they have a proven positive effect on similar tasks [35].

Additionally, all 3D information was discarded in the experiments run as part of this project. Using architectures with three dimensional convolutions as opposed to two dimensional convolutions is an obvious way to remedy this. As such, the V-net architecture [42] should be tested, potentially using batch renormalisation instead of batch normalisation to reduce the memory footprint.

Finally, no post-processing was performed. Earlier work has demonstrated that post-processing with conditional random fields (CRFs) is beneficial for segmentation with neural networks [25]. Additionally, the CRF models would allow 3D information to be encoded in the segmentation masks without using 3D convolutions. One particularly efficient CRF model is the dense CRF model [95], which is commonly used to improve the output of neural networks [24], [25].

One benefit with CRF post-processing is that it only requires probability maps and input images [96], [97]. As such, it can, and should, be tested on the outputs of the best models developed as part of this project.

We have already discussed the potential of small architectural changes (e.g. using the Large Kernel Matters architecture [24]). However, there are some larger architectural changes that might yield further improvements on model performance. Firstly, the model presented herein use a one-step process for segmentation. One potential improvement would be to train an additional network whose goal is to predict whether or not a slice will be segmented well by the network. Thus, the prediction process would consist of a two-step process. Firstly, one model would predict whether or not the segmentation network is able to segment the affected tissue sufficiently and then, if this is the case, the image would be delineated.

Finally, we note that using neural networks that has been trained to perform other tasks can yield good result for medical imaging tasks [98]. In particular, models that has been trained to perform segmentation of natural images, can be modified to segment tumours. Modifying previously trained networks to perform new tasks is called *transfer learning*. By choosing the degree of modification based on the amount of data available, Tajbakhsh *et al.* [98] achieved better performance than with random initialisation of the network weights. Transfer learning should also be tested in further work.

6.2 Analysis of the top performing models

6.2.1 Comparison based on model input

The best CT-only model performed on par with the best PET-only model on the validation data (Table 5.14 on page 128). On the test data (Table 5.17 on page 132), however, the CT-only model had a considerable drop in performance (especially median performance). Thus, the hyperparameter sweep likely ended up with a model that performed well on the validation set by mere chance. Studying the performance of the CT-only model on the validation set will therefore likely end with too optimistic conclusions.

Recall that neither the PET/CT model, nor the PET-only model had a noticable drop in the PPV and sensitivity between validation data (Table 5.15 on page 128) and the test data (Table 5.18 on page 133). There was, on the other hand, a drop in sensitivity of 0.04 units for the CT-only model. Thus, the PET/CT and PET-only model performances on the validation set were representative for the test set, whereas the CT-only performance was not.

Furthermore, upon inspection, we see that the main difference between the performance of the CT-only models and the models using PET-information was the sensitivity (Tables 5.15 and 5.18). The PPV was the same for the CT-only model and PET/CT model. There are at least two possible reasons for these results.

Firstly, the tumours are more difficult to locate using only CT information, which is the reason PET/CT often is used [5]. Thus, this might account for the drop in sensitivity.

The other reason stems from the loss functions used; the CT-only model used the F_2 loss, whereas the PET-incorporating models used the F_4 loss. Thus, the decrease in sensitivity is probably also influenced by this choice. Further experiments should therefore be conducted to fully understand if the choice of loss function affects the generalisation gap in sensitivity. One interesting experiment would be to check if the sensitivity increases and the PPV decreases for the CT-only models trained with an F_4 loss.

The exact reason for why the sensitivity of the CT-only model is lower than that of the PET/CT and PET-only models is, in other words, unknown. It might either be because the CT-only model was trained with a loss function that weighted sensitivity less, or that it is more difficult to obtain a high sensitivity with only

CT images.

Table 5.16 (page 129) and Table 5.19 (page 135) show the mean and median performance per patient for each modality on the validation set and test set. We see that the PET/CT generally scored better than CT-only and PET-only. The PET/CT model was, however, not always the best model. There were several patients in which either the CT-only model, the PET-only model or both had higher performance than the PET/CT model. On average, however, the PET/CT model gave the highest performance scores.

6.2.2 Assessment of model behaviour

The focus of this section will be on the PET/CT and CT-only models, as CT is a standard procedure when performing PET scans. The performance of the PET-only model is, therefore, only interesting to understand if the inclusion of CT information provides improvements upon the segmentation masks of the PET-only models, which it has been shown to do.

A main difference between the CT-only model and the PET/CT model is the consistency. The CT-only model's median segmentation quality is not far from the PET/CT model's median segmentation quality (wrt. Dice). However, the CT-only model has a much higher probability of producing poor segmentation masks. This is particularly well demonstrated by the boxplots in Figures 5.5, 5.7 and 5.8 on pages 131, 138 and 139, respectively, and the histograms in Figures 5.4 and 5.6 on pages 130 and 136.

Upon studing the performance metrics in Table 5.18 on page 133, we see that there is a large drop in PPV between the PET/CT model and the CT-only model. This means that the CT-only model has considerably more false positives than the PET/CT model. Furthermore, the sensitivity of the CT-only model is smaller than that of the PET/CT model. Thus, the CT-only model has more false positives and fewer true positives.

Both the PET/CT and the CT-only model sometimes missed affected tissue altogether. This is problematic as the ground truth delineation masks do not differ between lymph nodes and the gross tumour volume (GTV). Thus, we have no data on how commonly the models missed the GTV completely. One method to test how often the models missed the GTV completely would be to segment the lymph nodes and tumour separately using a multi-class approach. However, doing this multi-class approach might reduce performance as it is difficult for lay-people to

discern GTV from PET-positive lymph nodes.

It was not solely the CT-only model that had a large portion of false positives, but also the PET/CT model. These false positives were regularly shared between the two models (e.g. patient 229 in the validation set and patient 249 in the test set), signifying that the models picked up on the same erroneous cues.

Some of the falsely delineated areas contained buildup of CT contrast agent. An example of this is patient 110, where the buildup of contrast agent on the right side resembles a lymph node to the untrained eye. It is important to know why the radiologist did not delineate this region when assessing the model performance.

The falsely delineated areas with contrast agent buildup demonstrate one weakness with automatic delineation from PET/CT images. Particularly, that the radiologists use more information than what is apparent in the images. This became apparent after consulting with oncologist Dr Dale at The Norwegian Radium Hospital who described how radiologists delineate tumours and lymph nodes in the head and the neck. Other sources of information that the radiologists use include endoscopy images as well as physical examinations. Thus, we cannot expect a fully automatic tumour delineation system without input from radiologists.

We should also inspect the slices in which the model successfully delineated the tumour. Specifically, we should pay attention to how the CT-only model successfully delineated the GTV in the slices with strong beam hardening artefacts. The success in these slices indicates that the model might delineate the tumour based on biases in the dataset instead of information in the images. The reason is that the tumour is in no form visible to the (untrained) human eye. Understanding the biases in the dataset and how to prevent that the model learns them are integral when developing an automatic tumour delineation system.

One bias that the model might have picked detected stems from the cropping of the images. The images were cropped such that the centre of mass of the segmentation masks were close to the image centre. Thus, if the tumour centre of mass is on the right hand side of the patient, the patient might be shifted to the left. The placement of the tumour might, in other words, be influenced by the placement of the patient in the image.

It might, at first, seem like the position of the patient in an image should not influence the output of a convolutional network, as the convolution operator is translationally invariant. There are two reasons why this is not the case; boundary effects and maxpooling layers. Specifically, finite convolutions are not spatially invariant unless we impose cyclic boundary conditions. These boundary conditions

are not used here, implying that the convolution operator is not translationally invariant. Additionally, strided maxpooling layers are not translationally invariant. Thus, there might be a way for the network to learn if the patient is shifted to the left or right of the image.

One way to assess the above hypothesis is to artificially crop the images so the tumour is located on the same side of the image as the patient is shifted. Thus, if the network still delineates the tumour correctly on slices with severe beam hardening artefacts (such as patient 5 in the test set), then the network did indeed learn how to delineate the tumour in such slices.

There are two weaknessess of deep learning. Firstly, neural networks are overconfident [99]. As a consequence, the network reports a 100% probability of healthy tissue being cancerous. We can, in other words, not trust the confidence of the segmentation masks. This is problematic, as we do not know which segmentation masks likely represent a tumour or not.

Bayesian deep learning is one method to combat the overconfidence problem [99] and has successfully been used for stroke lesion segmentation [100]. There are at least two reasons why Bayesian deep learning is not more popular. The computational cost of training such networks are high [101] and the theoretic background required for these algorithms is more difficult [101].

Another method to combat the overconfidence problem is to train a network to predict the quality of each connected component in the predicted segmentation mask. One way to acquire training data for this is to compute the Dice of each predicted connected component and all true connected components that overlap with them. Thus, each connected component would get a score between 0 and 1 which a classification network could be trained to predict. Such a classification network would take the predicted mask and the PET/CT image (or CT image for CT-only models) as input and return the quality score of that segmentation mask.

Another weakness of deep learning is demonstrated by the proposed delineation of patient 110. For this patient, both the PET/CT model and the CT-only model predict a small region of affected tissue on the left side of the patient on the early slices. However, with little to no local change in the image, the network stopped to delineate that region. Thus, the network either adapted to global cues, or local cues that were imperceptible for the untrained eye. Understanding what makes a network delineate different regions is important, however, it is difficult with neural networks working as black box models.

A visualisation suite should, therefore, be developed to understand what parts of

the image are important when detecting the different tumours/lymph nodes. There are, to the author's knowledge, no work on explicitly visualising pixel importance in segmentation tasks. However, guided backpropagation [54] is a classification network visualisation algorithm that easily can be generalised for the segmentation case.

Guided backpropagation is an algorithm for visualisation of classification networks using the ReLU nonlinearity [54]. The idea is to visualise the parts of the input that were most important when classifying the image. This is visualised using an image the same size as the input whose pixel values indicate the influence of the corresponding pixel to the network output.

One naive way to accomplish this is to differentiate the network output with respect to the input image. However, this method has one weakness, namely that the gradient of a neural network with respect to the input image is very noisy. Therefore, Springenberg *et al.* [54] introduced guided backpropagation, which modifies the way the derivative of ReLU nonlinearities are computed to yield more informative saliency maps.

There are several ways that guided backpropagation can be generalised for the segmentation case. Either, each pixel can get its own saliency map. However, this would only be useful through an interactive tool where the user picks a single pixel, and the saliency map for that pixel is computed. The other, possibly better, way of generalising guided backpropagation to segmentation maps is to compute the mean saliency map for connected regions of delineated tissue.

6.2.3 Evaluation of model performance

Before assessing the performance of the networks, we must discuss the ground truth data. Recall that the data was collected over several years and was delineated by both radiologists and nuclear physisists. In the cases where several segmentation masks were available, the union of them was used.

Furthermore, one goal of this thesis was to assess whether or not deep learning is a promising approach for head and neck cancer delineation. As such, enlarged lymph nodes and the GTV were considered to be the same class to make the classification task easier. This was thought to be easier since it is difficult for lay people to discern between GTV and lymph nodes. However, whether or not merging GTV and lymph nodes into one class improves model performance is not known. Furthermore, it makes comparison to other methods more difficult.

Another facet to consider is that two different radiologists might create vastly different delineations of the same tumour. This is known as interobserver variability. In a recent study [6], three radiologists, each with vast experience in head and neck cancers, were presented with PET/CT scans from ten head and neck cancer patients. The radiologists were first asked to delineate the GTV and organs at risk based solely on the contrast agent CT scan. Thereafter, they were presented with the PET signal as well and were tasked with delineating the GTV once more. The average Dice on the GTV between the radiologists delineation was 0.57 ± 0.12 for contrast agent CT and 0.69 ± 0.08 for PET/CT [6].

Recall that the highest performing PET/CT model achieved an average Dice of 0.66 ± 0.24, whereas the highest performing CT-only model achieved an average Dice of 0.56 ± 0.29. Thus, it might seem like the highest performing models achieve radiologist level performance. However, that conclusion cannot be drawn simply by examining the performance metrics. There are several reasons for this. Firstly, our ground truth data is the union of several delineations, not the delineation from a single radiologist. Thus, the comparison is not fully valid. Furthermore, the errors made by the models described in this text may be more severe than those made by radiologists. The only way to truthfully conclude about the model performance is by consulting with radiologists, which, unfortunately, was outside the scope of this project.

The fact that the model achieved Dice performance close to that between two radiologists must be acknowledged. However, with such high performance, we must be certain that the model performance is generalisable. To ensure no contamination of the test and training data, several measures were taken. Firstly, the dataset was thoroughly investigated to make sure that no slices from patients in the validation or test set were available in the training set. Furthermore, to ensure that the SciNets package did not train the models using the validation or test set, additional models were trained with the best hyperparameter combinations. However, these models were trained with a dataset file where the validation and test set were replaced with the training set. After fitting the models, these models were tested with the correct validation set, which showed similar performance to the highest performing models. Finally, Figure 5.1 on page 119 demonstrates that the validation loss stops decreasing after approximately 2000 iterations, whereas the training loss did not. If the models trained on validation data, this would not be observed. We can, therefore, conclude that the model did not train on testing and validation data.

It is difficult to compare the performance of the deep learning models explored here to that of other automatic head and neck tumour delineation models. The

reason is that there is, to the author's knowledge, no other work taking such a broad approach to HNC tumour delineation.

In 'Automated Radiation Targeting in Head-and-Neck Cancer Using Region-Based Texture Analysis of PET and CT Images', Yu *et al.* [29] do not report any local cropping around the ROIs. We, therefore, assume their tumour prediction model works similarly to ours, by delineating full images. By using their reported sensitivity and specificity, we compute the expected Dice of the their model on our dataset to be approximately 0.85. Thus, their performance was not only better than the performance presented herein, it was vastly better than that between radiologists [6]. However, the study only included 10 patients and the testing and validation procedure was not well described. Their algorithm does, in other words, not have the same generalisation guarantees as the models presented in this text.

Two other relevant studies described prediction of CT images and delineated clinical tumour volumes (CTV) [30], [31]. In these studies, CT images and delineated GTVs were used to predict the clinical tumour volume (CTV). Both papers report Dice performances in the range 0.7 to 0.85. Their results were, in other words, much stronger than those presented herein. However, neither of the studies reduce interobserver variability as well as the models presented herein because both require the GTV delineation to predict the CTV.

Finally, we have the work by Han *et al.* [22]. In this study the authors develop a graph cut based Markov Random Fields model [97] to create semiautomatic segmentation maps from PET/CT images. They achieved stellar results, with a Dice score in the range of 0.8 to 0.9. The algorithm presented in [22] requires that a radiologist marks parts of the tumour and parts of the healthy tissue (called "seeds"). A thorough comparison of their result to ours is therefore difficult because semiautomatic systems do not yield unique segmentation maps for each image.

6.3 Evaluation of the SciNets library

In this thesis, we introduced the SciNets library, which allowed us to systematically run a large parameter sweep for image segmentation using a U-Net architecture. The straightforward API allowed us to shift focus away from the implementation details of the neural networks when running experiments.

Another benefit of the SciNets library is that the same parameter files can be

used for experiments with vastly different data. For this reason, HDF5 files for segmentation of organs at risk from PET/CT scans as well as HDF5 files for segmentation of rectal cancers from MRI images of approximately 200 patients, were prepared. How, neither of these experiments were run due to time limitations.

Several weaknesses of the SciNets library became apparent during and after the training. Firstly, the `DataReader` class will allways shuffle the dataset, which made it difficult to assess patient-to-patient performance. It was, however, possible as the `store_outputs` method stores the input-output pairs of the model as well as the the index of each such pair.

The ideal way to fix the above problem is to implement a context manager that prevents dataset shuffling. Then, the TensorFlow session be should generated within that context. Below is a demonstration of how the API of such a solution should be.

```
dataset = scinets.data.HDFDataset(
    data_path="/datasets/val_split_2d.h5"
    batch_size=[train_batch_size, val_batch_size, test_batch_size,]
    train_group="train"
    val_group="val"
    test_group="test",
    preprocessor=preprocessor,
    is_training=is_training,
    is_testing=is_testing
)

with dataset.no_shuffle():  # This context is not implemented in SciNets
    with tf.Session() as sess:
        # Do something
```

Furthermore, the way the parameter logging was performed made it cumbersome to compare models using more than one performance metric (in our case, the mean Dice per slice). There is, unfortunately, no easy method to integrate multiple final performance metrics in SacredBoard.

There are, however, two solutions to this problem. One solution is to develop a similar tool as Sacred and SacredBoard. However, this is no small feat. Therefore, we recommend using `Comet.ml` for tracking instead. `Comet.ml` is a commercial service that provides a tool similar to SacredBoard, but without having to rent a server and set up a database. Furthermore, it offers the use of several evaluation metrics, instead of just one. Additionally, `comet.ml` offers an easier API for experiment logging.

The main downside of Comet.ml is that it is not open source, but rather the product of a company. Thus, if the company goes bankrupt, or the pricing model changes, then the database of experiments might be lost. Luckily, the paid plans are currently free for academic use.

Comet.ml was not implemented as part of SciNets for three reasons. Firstly, it was launched in April 2018, four months after the development of SciNets had started, and it was, as of July 2018, not completely stable. Furthermore, as a new, commercial product with unknown lifetime, it was deemed an unstable option. Finally, the downsides of Sacred and SacredBoard were not known before the final analysis of the results was conducted.

The generation of the final evaluation plots and tables was unnecessarily cumbersome and automating this is clearly beneficial. SciNets should, therefore, be extended to automatically generate a wide range of tables and plots chosen by the user. These plots and tables could then be generated for each model. If a Comet.ml logger is implemented, these plots and tables could be uploaded to the Comet.ml project corresponding to the experiments.

There is one caveat to keep in mind when automating the generation of final evaluation tables and figures, namely, how to deal with the test set. Performances on the test set should not be generated automatically for all models, as the comparison of models should not be performed on the basis of the test set. Thus, the best way to implement this is to have a class that generates the evaluation tables and results for a given dataset, and call this function with the validation set automatically after each model is trained.

The analysis of the experiments also revealed the need for good visualisation tools. Adding an interactive toolkit is outside the scope of SciNets. However, extending the final final evaluation pipeline to generate guided backpropagation outputs is within the scope. Thus, for a subset of the patients, guided backpropagation visualisation can be created for separately for all connected components in the proposed segmentation map.

Another shortcoming of the library was that creating of experiment parameter files was time consuming. This was mainly an effect of the way the `TensorboardLogger`, particularly how the Tensorboard image logger, was implemented was implemented. Problems with the image logger arose when different windowing and channel settings were used, thereby changing the number of input channels and requiring different loggers. This meant that the dataset parameters and the logger parameters were coupled, so simply creating log files with all possible parameter

combinations was not possible.

One way to combat the problem with the image loggers is to create a new log type. The new logger would not take a parameter that represents which input channel to visualise, but rather make one Tensorboard image log per input channel. By implementing such a logger, it would be trivial to generate all combinations of parameter configurations automatically.

The experiment phase revealed certain problems that caused poor GPU utilisation (20%) and had to be corrected. The first problem was that bicubic interpolation was used after upconvolutions to ensure that the output of the upconvolutional layer and the input to the corresponding maxpooling layer were of the same size. This is necessary because a max pooling followed by an upconvolution will have dimensions that differ by one if the input to the max pooling layer is odd. After profiling the code, it became apparent that TensorFlow's bicubic interpolation layer was unable to run on a GPU. By using bilinear interpolation layers instead, the GPU utilisation increased markedly.

The other performance bottleneck came from how the datasets were loaded from disc. Two separate methods were tested to reduce the time taken to load; keeping the entire dataset in RAM and fetching the next batch while computing the current batch gradient. There were no noticeable performance gains by keeping the whole dataset in RAM. Prefetching is therefore recommended.

Chapter 7

Conclusion

In this thesis, we have introduced the SciNets library for deep learning. The library is aimed at image segmentation and classification and can take any stack of images as inputs (e.g. RGB images, PET/CT images or MRI images). Further, SciNets enables rapid model prototyping whilst ensuring reproducible results. SciNets is implemented using the TensorFlow framework by Google [12] and can amongst others, automatically generate TensorBoard [12] and SacredBoard [89] logs for visualisation. By using the SciNets library, we successfully performed a vast parameter sweep with over 160 different neural networks for ROI delineation in HNC patients.

Furthermore, we introduced the novel F_β loss – a generalisation of the Dice loss introduced in [42]. The F_β loss allows for different weighting of sensitivty and PPV in the loss by varying the β value. The β value represents the weighing factor of the sensitvity. If $\beta = 1$, then the sensitivity and PPV is weighted the same and the loss function is equivalent to the Dice loss. Similarly, if $\beta = 2$, then the the sensitivity is weighted twice as much as the PPV and if $\beta = 0.5$, then the sensitivity is weighted half that of the PPV.

The parameter sweep resulted in a set of recommended hyperparameters for similar segmentation tasks (presented on page 166). Notably, we demonstrated that using the newly introduced F_2 and F_4 loss gave an expected improved validation performance compared to the Dice loss and cross entropy loss for all parameter combinations.

During the delineation experiments, we demonstrated that deep learning is applicable to the segmentation of tumours and malignant lymph nodes tissue in PET/CT

images of HNC patients. The PET/CT model achieved highest performance with an average Dice of 0.66 ± 0.24, sensitivity of 0.79 ± 0.28, Specificity of 0.99 ± 0.01 and PPV of 0.62 ± 0.26. The PET-only model achieved second best overall with a Dice of 0.64 ± 0.24, a sensitivity of 0.69 ± 0.27, a specificity of 0.99 ± 0.01 and a PPV of 0.64 ± 0.27. Finally, the CT-only models achieved lowest overall performance with a Dice of 0.56 ± 0.29, a sensitivity of 0.58 ± 0.33 a specificity of 0.99 ± 0.01 and a PPV of 0.62 ± 0.31.

Thus, we demonstrated that deep learning models can acquire radiologist-level Dice performance on delineation of affected tissue in PET/CT images. However, the Dice performance is not a true measure of segmentation quality as some errors are more severe than others. Future work should, therefore, consult with radiologists to get a true measure of model performance.

Bibliography

[1] World Heath Organization, *All cancers fact sheet*, http://gco.iarc.fr/today/data/factsheets/cancers/39-All-cancers-fact-sheet.pdf, Downloaded 2019-02-14, 2018.

[2] World Health Organization, *Tracking universal health coverage: 2017 global monitoring report*, https://www.who.int/healthinfo/universal_health_coverage/report/2017/en/, Downloaded 2019-02-20, 2018.

[3] ——, *Density of physicians (total number per 1000 population, latest available year)*, https://www.who.int/gho/health_workforce/physicians_density/en/, Downloaded 2019-02-20, 2018.

[4] J. J. Caudell, J. F. Torres-Roca, R. J. Gillies, H. Enderling, S. Kim, A. Rishi, E. G. Moros and L. B. Harrison, 'The future of personalised radiotherapy for head and neck cancer', *The Lancet Oncology*, vol. 18, no. 5, e266–e273, 2017, ISSN: 1470-2045. DOI: https://doi.org/10.1016/S1470-2045(17)30252-8. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1470204517302528.

[5] R. L. Wahl and H. N. Wagner, *Principles and practice of PET and PET/CT*. Lippincott Williams & Wilkins, 2009.

[6] S. Gudi, S. Ghosh-Laskar, J. P. Agarwal, S. Chaudhari, V. Rangarajan, S. N. Paul, R. Upreti, V. Murthy, A. Budrukkar and T. Gupta, 'Inter-observer variability in the delineation of gross tumour volume and specified organs-at-risk during imrt for head and neck cancers and the impact of fdg-pet/ct on such variability at the primary site', *Journal of Medical Imaging and Radiation Sciences*, vol. 48, no. 2, pp. 184–192, 2017, ISSN: 1939-8654. DOI: https://doi.org/10.1016/j.jmir.2016.11.003. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1939865416301679.

[7] Y. LeCun, L. Bottou, Y. Bengio and P. Haffner, 'Gradient-based learning applied to document recognition', *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[8] A. Krizhevsky, I. Sutskever and G. E. Hinton, 'Imagenet classification with deep convolutional neural networks', in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou and K. Q. Weinberger, Eds., Curran Associates, Inc., 2012, pp. 1097–1105. [Online]. Available: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[9] X. Glorot and Y. Bengio, 'Understanding the difficulty of training deep feedforward neural networks', in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, 2010, pp. 249–256.

[10] X. Glorot, A. Bordes and Y. Bengio, 'Deep sparse rectifier neural networks', in *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, 2011, pp. 315–323. [Online]. Available: http://proceedings.mlr.press/v15/glorot11a/glorot11a.pdf.

[11] S. Ioffe and C. Szegedy, 'Batch normalization: Accelerating deep network training by reducing internal covariate shift', in *International Conference on Machine Learning*, 2015, pp. 448–456. [Online]. Available: https://arxiv.org/pdf/1502.03167.pdf.

[12] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu and Xiaoqiang Zheng, *TensorFlow: Large-scale machine learning on heterogeneous systems*, Downloaded 2019-02-10, 2015. [Online]. Available: https://www.tensorflow.org/.

[13] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga and A. Lerer, 'Automatic differentiation in pytorch', in *Conference on Neural Information Processing System*, 2017.

[14] F. Seide and A. Agarwal, 'Cntk: Microsoft's open-source deep-learning toolkit', in *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16, San Francisco, California, USA: ACM, 2016, pp. 2135–2135, ISBN: 978-1-4503-4232-

2. DOI: 10.1145/2939672.2945397. [Online]. Available: http://doi.acm.org/10.1145/2939672.2945397.

[15] F. Chollet *et al.*, *Keras*, https://keras.io, Downloaded 2019-02-10, 2015.

[16] Nvidia, *Cuda c programming guide*, https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, Downloaded 2019-02-20, 2018.

[17] National Cancer Institute, *Head and neck cancers*, https://www.cancer.gov/types/head-and-neck/head-neck-fact-sheet, Downloaded 2019-02-14, 2018.

[18] N. B. Smith and A. Webb, *Introduction to medical imaging: physics, engineering and clinical applications*. Cambridge university press, 2010.

[19] M. Hatt, B. Laurent, A. Ouahabi, H. Fayad, S. Tan, L. Li, W. Lu, V. Jaouen, C. Tauber, J. Czakon, F. Drapejkowski, W. Dyrka, S. Camarasu-Pop, F. Cervenansky, P. Girard, T. Glatard, M. Kain, Y. Yao, C. Barillot, A. Kirov and D. Visvikis, 'The first miccai challenge on pet tumor segmentation', *Medical Image Analysis*, vol. 44, pp. 177–195, 2018, ISSN: 1361-8415. DOI: https://doi.org/10.1016/j.media.2017.12.007. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1361841517301895.

[20] M. Hatt, J. A. Lee, C. R. Schmidtlein, I. E. Naqa, C. Caldwell, E. De Bernardi, W. Lu, S. Das, X. Geets, V. Gregoire, R. Jeraj, M. P. MacManus, O. R. Mawlawi, U. Nestle, A. B. Pugachev, H. Schöder, T. Shepherd, E. Spezi, D. Visvikis, H. Zaidi and A. S. Kirov, 'Classification and evaluation strategies of auto-segmentation approaches for pet: Report of aapm task group no. 211', *Medical Physics*, vol. 44, no. 6, e1–e42, 2017. DOI: 10.1002/mp.12124. eprint: https://aapm.onlinelibrary.wiley.com/doi/pdf/10.1002/mp.12124. [Online]. Available: https://aapm.onlinelibrary.wiley.com/doi/abs/10.1002/mp.12124.

[21] S. Leibfarth, F. Eckert, S. Welz, C. Siegel, H. Schmidt, N. Schwenzer, D. Zips and D. Thorwarth, 'Automatic delineation of tumor volumes by co-segmentation of combined pet/mr data', *Physics in Medicine & Biology*, vol. 60, no. 14, p. 5399, 2015.

[22] D. Han, J. Bayouth, Q. Song, A. Taurani, M. Sonka, J. Buatti and X. Wu, 'Globally optimal tumor segmentation in pet-ct images: A graph-based co-segmentation method', in *Information Processing in Medical Imaging*, G. Székely and H. K. Hahn, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 245–256, ISBN: 978-3-642-22092-0.

[23] O. Ronneberger, P. Fischer and T. Brox, 'U-net: Convolutional networks for biomedical image segmentation', in *International Conference on Medical image computing and computer-assisted intervention*, Springer, 2015, pp. 234–241. [Online]. Available: http://lmb.informatik.uni-freiburg.de/.

[24] C. Peng, X. Zhang, G. Yu, G. Luo and J. Sun, 'Large kernel matters - improve semantic segmentation by global convolutional network', *CoRR*, vol. abs/1703.02719, 2017. arXiv: 1703.02719. [Online]. Available: http://arxiv.org/abs/1703.02719.

[25] L. Chen, G. Papandreou, I. Kokkinos, K. Murphy and A. L. Yuille, 'Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs', *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 40, no. 4, pp. 834–848, Apr. 2018, ISSN: 0162-8828. DOI: 10.1109/TPAMI.2017.2699184.

[26] L.-C. Chen, G. Papandreou, I. Kokkinos, K. Murphy and A. L. Yuille, 'Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs', *IEEE transactions on pattern analysis and machine intelligence*, vol. 40, no. 4, pp. 834–848, 2018.

[27] L.-C. Chen, G. Papandreou, F. Schroff and H. Adam, 'Rethinking atrous convolution for semantic image segmentation', 2017.

[28] J. Long, E. Shelhamer and T. Darrell, 'Fully convolutional networks for semantic segmentation', in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 3431–3440.

[29] H. Yu, C. Caldwell, K. Mah, I. Poon, J. Balogh, R. MacKenzie, N. Khaouam and R. Tirona, 'Automated radiation targeting in head-and-neck cancer using region-based texture analysis of pet and ct images', *International Journal of Radiation Oncology*Biology*Physics*, vol. 75, no. 2, pp. 618–625, 2009, ISSN: 0360-3016. DOI: https://doi.org/10.1016/j.ijrobp.2009.04.043. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S036030160900618X.

[30] C. E. Cardenas, B. M. Anderson, M. Aristophanous, J. Yang, D. J. Rhee, R. E. McCarroll, A. S. R. Mohamed, M. Kamal, B. A. Elgohari, H. M. Elhalawani, C. D. Fuller, A. Rao, A. S. Garden and L. E. Court, 'Auto-delineation of oropharyngeal clinical target volumes using 3d convolutional neural networks', *Physics in Medicine & Biology*, vol. 63, no. 21, p. 215 026, Nov. 2018. DOI: 10.1088/1361-6560/aae8a9. [Online]. Available: https://doi.org/10.1088%5C%2F1361-6560%5C%2Faae8a9.

[31] C. E. Cardenas, R. E. McCarroll, L. E. Court, B. A. Elgohari, H. Elhalawani, C. D. Fuller, M. J. Kamal, M. A. Meheissen, A. S. Mohamed, A. Rao, B. Williams, A. Wong, J. Yang and M. Aristophanous, 'Deep learning algorithm for auto-delineation of high-risk oropharyngeal clinical target volumes with built-in dice similarity coefficient parameter optimization function', *International Journal of Radiation Oncology*Biology*Physics*, vol. 101, no. 2, pp. 468–478, 2018, ISSN: 0360-3016. DOI: https://doi.org/10.1016/j.ijrobp.2018.01.114. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0360301618302451.

[32] S. Liang, F. Tang, X. Huang, K. Yang, T. Zhong, R. Hu, S. Liu, X. Yuan and Y. Zhang, 'Deep-learning-based detection and segmentation of organs at risk in nasopharyngeal carcinoma computed tomographic images for radiotherapy planning', *European Radiology*, Oct. 2018, ISSN: 1432-1084. DOI: 10.1007/s00330-018-5748-9. [Online]. Available: https://doi.org/10.1007/s00330-018-5748-9.

[33] S. Ren, K. He, R. Girshick and J. Sun, 'Faster r-cnn: Towards real-time object detection with region proposal networks', in *Advances in neural information processing systems*, 2015, pp. 91–99.

[34] S. Pereira, A. Pinto, V. Alves and C. A. Silva, 'Brain tumor segmentation using convolutional neural networks in mri images', *IEEE Transactions on Medical Imaging*, vol. 35, no. 5, pp. 1240–1251, May 2016, ISSN: 0278-0062.

[35] D. Hammack, 'Forecasting lung cancer diagnoses with deep learning', Tech. Rep., 2017. [Online]. Available: https://github.com/dhammack/DSB2017/blob/master/dsb_2017_daniel_hammack.pdf.

[36] T. Hastie, R. Tibshirani and J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition*, ser. Springer Series in Statistics. Springer New York, 2009, ISBN: 9780387848587. [Online]. Available: https://books.google.no/books?id=tVIjmNS3Ob8C.

[37] I. Goodfellow, Y. Bengio and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[38] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006, ISBN: 0387310738.

[39] K. He, X. Zhang, S. Ren and J. Sun, 'Deep residual learning for image recognition', in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2016, pp. 770–778. DOI: 10.1109/CVPR.2016.90.

[40] C. Olah, A. Mordvintsev and L. Schubert, 'Feature visualization', *Distill*, 2017, https://distill.pub/2017/feature-visualization. DOI: 10.23915/distill.00007.

[41] B. Poole, S. Lahiri, M. Raghu, J. Sohl-Dickstein and S. Ganguli, 'Exponential expressivity in deep neural networks through transient chaos', in *Advances in Neural Information Processing Systems 29*, D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon and R. Garnett, Eds., Curran Associates, Inc., 2016, pp. 3360–3368. [Online]. Available: http://papers.nips.cc/paper/6322-exponential-expressivity-in-deep-neural-networks-through-transient-chaos.pdf.

[42] F. Milletari, N. Navab and S.-A. Ahmadi, 'V-net: Fully convolutional neural networks for volumetric medical image segmentation', in *3D Vision (3DV), 2016 Fourth International Conference on*, IEEE, 2016, pp. 565–571. [Online]. Available: https://arxiv.org/pdf/1606.04797.pdf.

[43] J. Redmon, S. Divvala, R. Girshick and A. Farhadi, 'You only look once: Unified, real-time object detection', in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 779–788. [Online]. Available: http://pjreddie.com/yolo/.

[44] D. P. Kingma and J. L. Ba, 'ADAM: A Method for Stochastic Optimization', in *International Conference on Learning Representations*, 2014. [Online]. Available: https://arxiv.org/pdf/1412.6980.pdf.

[45] D. J. C. MacKay, *Information Theory, Inference & Learning Algorithms*. New York, NY, USA: Cambridge University Press, 2002, ISBN: 0521642981.

[46] K. Simonyan and A. Zisserman, 'Very deep convolutional networks for large-scale image recognition', in *International Conference on Learning Representations*, 2015.

[47] G. Huang, Z. Liu and K. Q. Weinberger, 'Densely connected convolutional networks', *CoRR*, vol. abs/1608.06993, 2016. arXiv: 1608.06993. [Online]. Available: http://arxiv.org/abs/1608.06993.

[48] D.-A. Clevert, T. Unterthiner and S. Hochreiter, 'Fast and accurate deep network learning by exponential linear units (elus)', in *International Conference on Learning Representations*, 2015.

[49] G. Klambauer, T. Unterthiner, A. Mayr and S. Hochreiter, 'Self-normalizing neural networks', in *Advances in Neural Information Processing Systems*, 2017, pp. 971–980.

[50] I. J. Goodfellow, D. Warde-Farley, M. Mirza, A. Courville and Y. Bengio, 'Maxout networks', in *Proceedings of the 30th International Conference on International Conference on Machine Learning-Volume 28*, JMLR. org, 2013, pp. III–1319.

[51] W. Shang, K. Sohn, D. Almeida and H. Lee, 'Understanding and improving convolutional neural networks via concatenated rectified linear units', in *International Conference on Machine Learning*, 2016, pp. 2217–2225.

[52] R. Gonzalez and R. Woods, *Digital Image Processing*. Pearson, 2017, ISBN: 9780133356724.

[53] W. Luo, Y. Li, R. Urtasun and R. Zemel, 'Understanding the effective receptive field in deep convolutional neural networks', in *Advances in neural information processing systems*, 2016, pp. 4898–4906.

[54] J. T. Springenberg, A. Dosovitskiy, T. Brox and M. A. Riedmiller, 'Striving for simplicity: The all convolutional net', *CoRR*, vol. abs/1412.6806, 2014. arXiv: 1412.6806. [Online]. Available: http://arxiv.org/abs/1412.6806.

[55] M. Holschneider, R. Kronland-Martinet, J. Morlet and P. Tchamitchian, 'A real-time algorithm for signal analysis with the help of the wavelet transform', in *Wavelets*, Springer, 1990, pp. 286–297.

[56] F. Yu and V. Koltun, 'Multi-scale context aggregation by dilated convolutions', *International Conference on Learning Representations*, vol. abs/1511.07122, 2015. arXiv: 1511.07122. [Online]. Available: http://arxiv.org/abs/1511.07122.

[57] H. Noh, S. Hong and B. Han, 'Learning deconvolution network for semantic segmentation', in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1520–1528.

[58] V. Dumoulin and F. Visin, 'A guide to convolution arithmetic for deep learning', 2016.

[59] A. Odena, V. Dumoulin and C. Olah, 'Deconvolution and checkerboard artifacts', *Distill*, 2016. DOI: 10.23915/distill.00003. [Online]. Available: http://distill.pub/2016/deconv-checkerboard.

[60] S. Santurkar, D. Tsipras, A. Ilyas and A. Madry, 'How does batch normalization help optimization?(no, it is not about internal covariate shift)', 2018.

[61] J. Kohler, H. Daneshmand, A. Lucchi, M. Zhou, K. Neymeyr and T. Hofmann, 'Exponential convergence rates for Batch Normalization: The power of length-direction decoupling in non-convex optimization', Oct. 2018.

[62] Y. Bengio, P. Lamblin, D. Popovici and H. Larochelle, 'Greedy layer-wise training of deep networks', in *Advances in neural information processing systems*, 2007, pp. 153–160.

[63] T. van Laarhoven, 'L2 regularization versus batch and weight normalization', in *Conference on Neural Information Processing Systems*, 2017.

[64] K. He, X. Zhang, S. Ren and J. Sun, 'Identity mappings in deep residual networks', in *European conference on computer vision*, Springer, 2016, pp. 630–645.

[65] A. Veit, M. J. Wilber and S. Belongie, 'Residual networks behave like ensembles of relatively shallow networks', in *Advances in Neural Information Processing Systems*, 2016, pp. 550–558.

[66] L. Rosasco and P. Tomaso, *Machine learning: A regularization approach*, Lecture notes, May 2017.

[67] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever and R. Salakhutdinov, 'Dropout: A simple way to prevent neural networks from overfitting', vol. 15, 2014, pp. 1929–1958. [Online]. Available: http://jmlr.org/papers/v15/srivastava14a.html.

[68] S. Wright and J. Nocedal, 'Numerical optimization', *Springer Science*, vol. 35, no. 67-68, p. 7, 1999.

[69] B. Polyak, 'Some methods of speeding up the convergence of iteration methods', *USSR Computational Mathematics and Mathematical Physics*, vol. 4, no. 5, pp. 1–17, 1964, ISSN: 0041-5553. DOI: https://doi.org/10.1016/0041-5553(64)90137-5. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0041555364901375.

[70] S. Ruder, 'An overview of gradient descent optimization algorithms', 2016.

[71] J. Duchi, E. Hazan and Y. Singer, 'Adaptive subgradient methods for online learning and stochastic optimization', *Journal of Machine Learning Research*, vol. 12, no. Jul, pp. 2121–2159, 2011.

[72] T. Tieleman and G. Hinton, *Lecture 6.5–RmsProp: Divide the gradient by a running average of its recent magnitude*, COURSERA: Neural Networks for Machine Learning, 2012.

[73] A. C. Wilson, R. Roelofs, M. Stern, N. Srebro and B. Recht, 'The marginal value of adaptive gradient methods in machine learning', in *Advances in Neural Information Processing Systems*, 2017, pp. 4148–4158.

[74] I. Loshchilov and F. Hutter, 'Sgdr: Stochastic gradient descent with warm restarts', in *International Conference on Learning Representations*, 2016.

[75] S. J. Reddi, S. Kale and S. Kumar, 'On the convergence of adam and beyond', in *International Conference on Learning Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=ryQu7f-RZ.

[76] K. He, X. Zhang, S. Ren and J. Sun, 'Delving deep into rectifiers: Surpassing human-level performance on imagenet classification', in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1026–1034.

[77] S. Raschka and V. Mirjalili, *Python machine learning*. Packt Publishing Ltd, 2017.

[78] M. Hatt, J. A. Lee, C. R. Schmidtlein, I. E. Naqa, C. Caldwell, E. De Bernardi, W. Lu, S. Das, X. Geets, V. Gregoire, R. Jeraj, M. P. MacManus, O. R. Mawlawi, U. Nestle, A. B. Pugachev, H. Schöder, T. Shepherd, E. Spezi, D. Visvikis, H. Zaidi and A. S. Kirov, 'Classification and evaluation strategies of auto-segmentation approaches for pet: Report of aapm task group no. 211', *Medical Physics*, vol. 44, no. 6, e1–e42, 2017. DOI: 10.1002/mp.12124. eprint: https://aapm.onlinelibrary.wiley.com/doi/pdf/10.1002/mp.12124. [Online]. Available: https://aapm.onlinelibrary.wiley.com/doi/abs/10.1002/mp.12124.

[79] L. R. Dice, 'Measures of the amount of ecologic association between species', *Ecology*, vol. 26, no. 3, pp. 297–302, 1945, ISSN: 00129658, 19399170. [Online]. Available: http://www.jstor.org/stable/1932409.

[80] T. Sørensen, 'A method of establishing groups of equal amplitude in plant sociology based on similarity of species and its application to analyses of the vegetation on danish commons', *Biol. Skr.*, vol. 5, pp. 1–34, 1948.

[81] C. J. V. Rijsbergen, *Information Retrieval*, 2nd. Newton, MA, USA: Butterworth-Heinemann, 1979, ISBN: 0408709294.

[82] V. Badrinarayanan, A. Kendall and R. Cipolla, 'Segnet: A deep convolutional encoder-decoder architecture for image segmentation', *IEEE transactions on pattern analysis and machine intelligence*, vol. 39, no. 12, pp. 2481–2495, 2017.

[83] S. Zheng, S. Jayasumana, B. Romera-Paredes, V. Vineet, Z. Su, D. Du, C. Huang and P. H. Torr, 'Conditional random fields as recurrent neural networks', in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1529–1537.

[84] S. Jégou, M. Drozdzal, D. Vazquez, A. Romero and Y. Bengio, 'The one hundred layers tiramisu: Fully convolutional densenets for semantic segmentation', in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2017, pp. 11–19.

[85] G. van Rossum, B. Warsaw and N. Coghlan, *PEP 8 – Style Guide for Python Code*, https://www.python.org/dev/peps/pep-0008/, Downloaded 2019-02-10, 2001.

[86] numpydoc maintainers, *numpydoc*, https://numpydoc.readthedocs.io/en/latest/, Downloaded 2019-02-10, 2017.

[87] A. Collette, *Python and HDF5: Unlocking Scientific Data.* " O'Reilly Media, Inc.", 2013.

[88] TensorFlow, *Tensorflow api r1.12*, https://www.tensorflow.org/versions/r1.12/api_docs/python/tf, Downloaded 2019-02-10, 2018.

[89] K. Greff, A. Klein, M. Chovanec, F. Hutter and J. Schmidhuber, 'The Sacred Infrastructure for Computational Research', in *Proceedings of the 16th Python in Science Conference*, K. Huff, D. Lippa, D. Niederhut and M. Pacer, Eds., 2017, pp. 49–56. DOI: 10.25080/shinma-7f4c6e7-008.

[90] Docker Inc, *Docker documentation*, https://docs.docker.com/, Downloaded 2019-02-10, 2018.

[91] The HDF Group, *Hdf home - the hdf group*, https://www.hdfgroup.org/, Downloaded 2019-02-10, 2018.

[92] NASA, *Hdf5 data model, file format and library – hdf5 1.6*, https://earthdata.nasa.gov/user-resources/standards-and-references/hdf5, Downloaded 2019-02-10, 2018.

[93] S. L. Smith, P.-J. Kindermans and Q. V. Le, 'Don't decay the learning rate, increase the batch size', in *International Conference on Learning Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=B1Yy1BxCZ.

[94] S. Ioffe, 'Batch renormalization: Towards reducing minibatch dependence in batch-normalized models', in *Advances in Neural Information Processing Systems*, 2017, pp. 1945–1953.

[95] P. Krähenbühl and V. Koltun, 'Efficient inference in fully connected crfs with gaussian edge potentials', in *Advances in neural information processing systems*, 2011, pp. 109–117.

[96] D. Koller, N. Friedman and F. Bach, *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

[97] Y. Boykov, O. Veksler and R. Zabih, 'Fast approximate energy minimization via graph cuts', in *Proceedings of the Seventh IEEE International Conference on Computer Vision*, IEEE, vol. 1, 1999, pp. 377–384.

[98] N. Tajbakhsh, J. Y. Shin, S. R. Gurudu, R. T. Hurst, C. B. Kendall, M. B. Gotway and J. Liang, 'Convolutional neural networks for medical image analysis: Full training or fine tuning?', *IEEE Transactions on Medical Imaging*, vol. 35, no. 5, pp. 1299–1312, May 2016, ISSN: 0278-0062. DOI: 10.1109/TMI.2016.2535302.

[99] C. Blundell, J. Cornebise, K. Kavukcuoglu and D. Wierstra, 'Weight uncertainty in neural network', in *International Conference on Machine Learning*, 2015, pp. 1613–1622.

[100] Y. Kwon, J.-H. Won, B. J. Kim and M. C. Paik, 'Uncertainty quantification using bayesian neural networks in classification: Application to ischemic stroke lesion segmentation', in *Medical Imaging with Deep Learning*, 2018.

[101] Y. Gal and Z. Ghahramani, 'Dropout as a bayesian approximation: Representing model uncertainty in deep learning', in *International Conference on Machine Learning*, 2016, pp. 1050–1059.

Appendix A

SciNets experiment structure

The `NetworkExperiment` class has a general structure for its input. In particular, the input is a set of dictionaries with a clear structure. We will now provide an example of each such dictionary and explain some of the implementation details of SciNets. We start with the dataset parameters.

Dataset parameters

```
dataset_params = {
    "operator": "HDFDataset",
    "arguments": {
        "data_path": "/dataset/val_split_2d.h5",
        "batch_size": [16, 128, 128],
        "val_group": "val",
        "preprocessor": {
            "operator": "PreprocessingPipeline",
            "arguments": {
                "preprocessor_dicts": [
                {
                    "operator": "ChannelRemoverPreprocessor",
                    "arguments": {"channel": 1}
                },
                {
                    "operator": "HoundsfieldWindowingPreprocessor",
                    "arguments": {
                        "window_width": 100,
                        "window_center": 70
                    }
                }
```

```
                ]
            }
        }
    }
}
```

Above we see that the `dataset_params` dictionary has a clear pattern. It contains two keys, `"operator"` and `"arguments"`. The `"operator"` key is used to extract a class from the `dataset_register`. A dataset instance of this class is generated, providing the `"arguments"` dictionary as keyword arguments through dictionary unpacking[1]. Thus, the code above creates a dataset the following way.

```
dataset = HDFDataset(
    data_path="/dataset/val_split_2d.h5",
    batch_size=[16, 128, 128],
    val_group="val",
    preprocessor={
        "operator": "PreprocessingPipeline",
        "arguments": {
            "preprocessor_dicts": [
            {
                "operator": "ChannelRemoverPreprocessor",
                "arguments": {"channel": 1}
            },
            {
                "operator": "HoundsfieldWindowingPreprocessor",
                "arguments": {
                    "window_width": 100,
                    "window_center": 70
                }
            }
            ]
        }
    }
)
```

This pattern is repeated when the preprocessor instance is created. Thus the `"operator"`, `"arguments"` pattern is a standard pattern in SciNets. The value of an `"operator"` key is provided to some subclass register to get a class. An instance of this class is then generated using the `"arguments"` dictionary as keyword arguments through dictionary unpacking.

Thus, to see the possible key-value combinations in the `"arguments"` dictionaries, we must look at the implementation of the class given by the `"operator"` key.

[1] See PEP 448 for information about this.

Log parameters

```
log_params = {
    "val_log_frequency": 100,
    "evaluator": {"operator": "BinaryClassificationEvaluator"},
    "loggers": [
        {
            "operator": "TensorboardLogger",
            "arguments": {
                "log_dicts": [
                    {
                        "log_name": "Loss",
                        "log_var": "loss",
                        "log_type": "scalar"
                    },
                    {
                        "log_name": "Accuracy",
                        "log_var": "accuracy",
                        "log_type": "scalar"
                    },
                    {
                        "log_name": "Dice",
                        "log_var": "dice",
                        "log_type": "scalar"
                    },
                    {
                        "log_name": "Probability_map",
                        "log_var":"probabilities",
                        "log_type": "image",
                        "log_kwargs": {"max_outputs":1}
                    },
                    {
                        "log_name": "Mask",
                        "log_var": "true_out",
                        "log_type": "image",
                        "log_kwargs": {"max_outputs":1}
                    },
                    {
                        "log_name": "CT_c70_w200",
                        "log_var": "input",
                        "log_type": "image",
                        "log_kwargs": {"max_outputs": 1, "channel":
                            0}
                    },
                    {
                        "log_name": "Probability_map",
                        "log_var":"probabilities",
                        "log_type": "histogram"
                    }
```

```
                ]
            }
        },
        {
            "operator": "SacredLogger",
            "arguments": {
                "log_dicts": [
                    {
                        "log_name": "Loss",
                        "log_var": "loss"
                    },
                    {
                        "log_name": "Accuracy",
                        "log_var": "accuracy"
                    },
                    {
                        "log_name": "Dice",
                        "log_var": "dice"
                    }
                ]
            }
        },
        {
            "operator": "HDF5Logger",
            "arguments": {
                "log_dicts": [
                    {
                        "log_name": "Loss",
                        "log_var": "loss"
                    },
                    {
                        "log_name": "Accuracy",
                        "log_var": "accuracy"
                    },
                    {
                        "log_name": "Dice",
                        "log_var": "dice"
                    }
                ]
            }
        }
    ],
    "network_tester": {
        "metrics": ["dice", "true_positives", "true_negatives", "
            sensitivity", "precision"]
    }
}
```

Here, we see a similar pattern to that of the dataset parameters. However, a single SciNets experiment might have several loggers. Thus, the loggers are parametrised in the `"loggers"` list. Furthermore, there are some parameters that are set in the `NetworkExperiment` class, these are provided as additional key-value pairs. The `"val_log_frequency"` key specifies how often logs should be computed on the validation dataset, the `"evaluator"` key contains a dictionary that parametrises the network evaluator and the `"network_tester"` key specifies which parameters to compute for the validation dataset after training.

Model parameters

```
{
    "operator": "UNet",
    "arguments": {
        "loss_function": {
            "operator": "BinaryFBeta",
            "arguments": {"beta": 2}
        },
        "skip_connections": [
            ["conv2", "upconv4"],
            ["conv4", "upconv3"],
            ["conv6", "upconv2"],
            ["conv8", "upconv1"]
        ],
        "architecture": [
            {
                "layer": "Conv2D",
                "scope": "conv1",
                "layer_params": {
                    "out_size": 64,
                    "k_size": 3
                },
                "normalizer": {
                    "operator": "BatchNormalization"
                },
                "activation": {
                    "operator": "ReLU"
                },
                "initializer": {
                    "operator": "he_normal"
                }
            },
            {
                "layer": "Conv2D",
                "scope": "conv2",
                "layer_params": {
```

```
                    "out_size": 64,
                    "k_size": 3
                },
                "normalizer": {
                    "operator": "BatchNormalization"
                },
                "activation": {
                    "operator": "ReLU"
                },
                "initializer": {
                    "operator": "he_normal"
                }
            },
            {
                "layer": "MaxPool",
                "scope": "max_pool1",
                "layer_params": {
                    "pool_size": 2
                }
            },
            ...
            {
                "layer": "Upconv2D",
                "scope": "upconv4",
                "layer_params": {
                    "out_size": 64,
                    "k_size": 3
                },
                "normalizer": {
                    "operator": "BatchNormalization"
                },
                "activation": {
                    "operator": "ReLU"
                },
                "initializer": {
                    "operator": "he_normal"
                }
            },
            {
                "layer": "Conv2D",
                "scope": "conv17",
                "layer_params": {
                    "out_size": 64,
                    "k_size": 3
                },
                "normalizer": {
                    "operator": "BatchNormalization"
                },
                "activation": {
```

```
                    "operator": "ReLU"
                },
                "initializer": {
                    "operator": "he_normal"
                }
            },
            {
                "layer": "Conv2D",
                "scope": "conv18",
                "layer_params": {
                    "out_size": 64,
                    "k_size": 3
                },
                "normalizer": {
                    "operator": "BatchNormalization"
                },
                "activation": {
                    "operator": "ReLU"
                },
                "initializer": {
                    "operator": "he_normal"
                }
            },
            {
                "layer": "Conv2D",
                "scope": "conv19",
                "layer_params": {
                    "out_size": 1,
                    "k_size": 3
                },
                "initializer": {
                    "operator": "he_normal"
                },
                "activation": {
                    "operator": "Sigmoid"
                }
            }
        ]
    }
}
```

Again, we see the `"operator"`–`"arguments"` dictionary structure. We now briefly describe the structure of the U-Net class. We see that the `UNet` class takes three keyword arguments, the `loss_function`, the `skip_connections` and the `architecture`. How a normal feed-forward network is generated is described in 3.1.

Trainer parameters

```
trainer_params = {
    "save_step": 2000,
    "max_checkpoints": 20,
    "train_op": {
        "operator": "MomentumOptimizer",
        "arguments": {
            "momentum": 0.9
        }
    },
    "learning_rate_scheduler": {
        "operator": "CosineDecayRestarts",
        "arguments": {
            "learning_rate": 0.05,
            "first_decay_steps": 650,
            "t_mul": 10,
            "m_mul": 1,
            "alpha": 0.01
        }
    }
}
```

There is only one `NetworkTrainer` class, and the generation of a network trainer does, therefore, not include "operator" and "arguments" keys. Rather, the whole `trainer_params` dictionary is provided as keyword arguments when initiating the `NetworkTrainer` instance.

Experiment parameters

```
experiment_params = {
    "continue_old": False
    "log_dir": "/myhdd/logs"
    "name": "unet_only_ct_windowing_c70_w200_basic_f2_sgdr"
    "verbose": True
}
```

The `experiment_params` dictionary contains only four key-value pairs. `"continue_old"` specifies whether or not training should start at the latest checkpoint. `"log_dir"` specifies which directory to store the checkpoints and log files. `"name"` specifies the name of the experiment and `"verbose"` is `True` if additional printouts should be shown in the terminal window. We recommend setting the verbosity to `True`.

Appendix B

The CLI programmes in SciNets

Several CLI programmes are provided as part of SciNets. Here, we will highlight "run_sacred" and "run_experiment". These CLI programmes perform the same task, namely running an experiment based on a set of JSON files. The difference is that "run_sacred" creates a connection with a sacred database and stores the sacred logs there. Thus, the "run_sacred" CLI programme must be ran if a sacred logger is present in the `log_params` dictionary, whereas "run_experiment" should be ran if it is not.

The structure of the parameter dictionaries is explicitly designed to easily be stored in JSON files. This is, in fact, the sole reason for the `SubclassRegister` , which allows instances of classes to be generated from a string with the class name. Thus, to run an experiment, each of the above dictionaries are stored in JSON file with the following names: "dataset_params.json", "log_params.json", "model_params.json", "trainer_params.json" and "experiment_params.json".

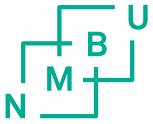
Furthermore, a CLI programme for creating a plethora of such experiments is provided; "create_experiments". This CLI takes amonst others a path as input. The folder at the provided path should contain four subfolders: "dataset_params", "log_params", "model_params" and "trainer_params". Each of these folders should contain at least one JSON file. The "create_experiments" programme will iterate through all possible combinations of the parameter files in the afforementioned folders and create an experiment folder for each of them. The "experiment_params.json" file is autogenerated by the other inputs to the "create_experiments" programme.

Finally, we have the "store_outputs" programme. This programme will find the

highest performing model using a specified performance metric and store input-output pairs as well as performance metrics per slice for either the validation set (default) or test set (must be specified).

All CLI programmes are documented using the `argparse` builtin library in Python with a description of each mandatory and optional input argument.

Thank you.



Norges miljø- og biovitenskapelige universitet
Noregs miljø- og biovitskapelege universitet
Norwegian University of Life Sciences

Postboks 5003
NO-1432 Ås
Norway