Universidade do Minho
Escola de Engenharia
Departamento de Informática

Mohammad Reza Tabrizi

Semantic Segmentation of Medical Images with Deep Learning

Agosto 2023

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Mohammad Reza Tabrizi

Semantic Segmentation of Medical Images with Deep Learning

Dissertação de Mestrado em Engenharia Informática

Dissertação supervisionada por
Professor António Joaquim André Esteves

Agosto 2023

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my family for their unwavering support and patience throughout my master's degree journey in computer engineering. This journey took me to a new country and was filled with many challenges, but my family's love and encouragement kept me motivated. Their unwavering belief in me and my abilities inspired me to push through the toughest times and ultimately achieve my goal. I am forever grateful for their sacrifice and support.

I would also like to extend my sincere gratitude to my advisor, Professor António Joaquim André Esteves, for his guidance, mentorship, and valuable insights that were instrumental in shaping my academic and professional growth. His expertise, wisdom, and encouragement have been a constant source of inspiration throughout this journey.

Finally, I would like to acknowledge the professors at the University of Minho, who have made a tremendous impact on my academic growth through their lectures, guidance, and knowledge sharing. Their dedication to teaching and helping students has been truly inspiring, and I am grateful for the opportunity to learn from such knowledgeable individuals.

In conclusion, I would like to express my heartfelt gratitude to all of those who have supported me and made this journey possible. Their unwavering support and encouragement have been a constant source of strength, and I am proud to be a part of such a supportive community.

Thank you all very much.

ABSTRACT

The use of deep learning techniques in medical image analysis has been a subject of growing interest in recent years. One of the most important applications of these techniques is the detection and segmentation of tumors in histological images. This dissertation focused on investigating the use of deep learning models to segment tumors, with the aim of providing medical specialists with a tool that can help them make more precise diagnoses.

Tumor growth patterns are an important histological characteristic that can provide information about the aggressiveness and degree of malignancy of a tumor. Specifically, the epithelial-mesenchymal transition on the tumor front is a pattern that has been shown to confer high aggressiveness and a great capacity to invade tissues and cause metastases, leading to a poor prognosis regarding the evolution of the tumor. Therefore, detecting and segmenting tumors in histological images can be a critical step in the diagnosis and treatment of tumors.

The research process involved several steps, including preprocessing the images to prepare them for deep learning models. This step involved developing methods to enhance the quality of the images and make them suitable for training deep learning models. Two types of deep learning architectures, the U-Net and Tiramisu, were trained in a supervised way, and different types of loss functions were experimented with to measure their efficiency in controlling the training process. Additionally, different types of hyperparameters were tried, and the best value was chosen for each hyperparameter.

Finally, the effectiveness of the models was evaluated and compared both qualitatively and quantitatively based on their performance in image segmentation. The results obtained show that deep learning models surpassed the initially predicted values and reached a value above 94% based on the training data. for the Interception over the Union metric. This result demonstrates the potential of deep learning techniques to detect and segment tumors in histological images and reinforces the importance of continuing to investigate this topic. The best results of the present work were achieved with total loss, as explained on page 89.

Keywords: Medical image segmentation, brain tumor, deep learning, U-Net, Tiramisu, loss function.

RESUMO

A aplicação de técnicas de aprendizagem profunda na análise de imagens médicas tem sido alvo de um interesse crescente nos últimos anos. Uma das aplicações mais importantes destas técnicas é a deteção e segmentação de tumores em imagens histológicas. A presente dissertação fucou-se na investigação sobre a utilização de modelos de aprendizagem profunda para segmentar tumores, com o objetivo de fornecer aos especialistas médicos uma ferramenta que ajude a efetuar diagnósticos mais corretos.

Os padrões de crescimento tumoral são uma característica histológica importante, que pode fornecer informação sobre a agressividade e grau de malignidade dum tumor. Especificamente, a transição epitelial-mesenquimal na frente do tumor é um padrão que confere alta agressividade e grande capacidade de invadir tecidos e causar metástases, conduzindo a um mau prognóstico sobre a evolução do tumor. Portanto, a detecção e segmentação de tumores em imagens histológicas é um passo crítico no diagnóstico e tratamento dos tumores.

O trabalho desenvolvido decorreu em várias etapas, incluindo o pré-processamento das imagens para prepará-las para treinar os modelos de aprendizagem profunda. Essa etapa envolveu o desenvolvimento de métodos para melhorar a qualidade das imagens e torná-las adequadas para o treino de modelos de aprendizagem profunda. Dois tipos de modelo de aprendizagem profunda, U-Net e Tiramisu, foram treinados de forma supervisionada, e experimentaram-se diferentes tipos de função de perda para medir a sua eficácia no controlo do processo de treino. Adicionalmente, testaram-se diferentes tipos de hiperparâmetros e escolheu-se o melhor valor para cada hiperparâmetro a utilizar em futuras experiências.

Finalmente, a eficácia dos modelos foi avaliada e comparada tanto qualitativamente como quantitativamente com base no seu desempenho na segmentação de imagens. Os resultados obtidos mostram que os modelos de aprendizagem profunda ultrapassaram os valores inicialmente previstos e alcançaram um valor acima de 94% com base nos dados de treinamento para a métrica de Intercepção sobre União. Este resultado demonstra o potencial das técnicas de aprendizagem profunda para detetar e segmentar tumores em imagens histológicas e reforça a importância de continuar a investigar este tópico. Os melhores resultados deste trabalho foram obtidos com a função de perda total, como se mostra na página 89.

Palavras-chave: Segmentação de imagens médicas, tumor cerebral, aprendizagem profunda, U-Net, Tiramisu, função de perda.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

ACRONYMS

AI Artificial Intelligence. 1, 17, 31, 32, 47
ANN Artificial Neural Network. 1

BCE Binary Cross Entropy. 1, 71, 84

CNN Convolutional Neural Network. 1, 47, 52, 55, 57, 58, 60
CNS Central Nervous System. 1, 17
CT Computed Tomography. 1, 17, 23, 24, 26
CV Computer Vision. 1, 31

DCNN Deep Convolutional Neural Network. 1, 49, 55
DIP Digital Image Processing. 1, 23
DL Deep Learning. 1, 16–18, 29, 31, 32, 34–36, 42, 47, 65, 76, 77

EDA Exploratory Data Analysis. 1

FCN Fully Convolutional Network. 1, 43, 49, 57, 59
FDA Food and Drug Administration. 1, 24
FLAIR Fluid-Attenuated Inversion Recovery. 1, 62
FPN Feature Pyramid Network. 1, 52

GPS Global Positioning System. 1
GPU Graphics Processing Unit. 1, 35, 47, 49

IoU Intersection over Union. 1, 17, 72, 83

LGG Lower Grade Glioma. 1, 62
LiDAR Light Detection And Ranging. 1
LR learning rate. 1, 74, 76, 77

MAE Mean Absolute Error. 1, 39
MBE Mean Bias Error. 1, 39
MEI Mestrado em Engenharia Informática. 1
ML Machine Learning. 1, 16, 31, 32, 36, 39, 40, 64
MLP Multi-Layer Perceptron. 1, 48
MRI Magnetic Resonance Imaging. 1, 17, 23, 24, 27–29, 42, 60, 62, 102
MSE Mean Square Error. 1, 39

NDA	Non Disclosure Agreement. 1
NLL	Negative Log Likelihood. 1
RADAR	Radio Detection And Ranging. 1
ReLU	Rectified Linear Unit. 1, 32
RMS	Root Mean Squared. 1, 37
RMSE	Root Mean Square Error. 1, 39
SGD	Stochastic Gradient Descent. 1, 35–37, 75, 76
SPA	Single Page Application. 1
SVM	Support Vector Machine. 1, 40
TCGA	The Cancer Genome Atlas. 1, 62
TCIA	The Cancer Imaging Archive. 1, 62
TPU	Tensor Processing Unit. 1
UM	Universidade do Minho. 1

1 INTRODUCTION

According to the American Institute for Medical and Biological Engineering, the bioengineers work with cutting-edge technologies to address the grand challenges that define the human experience. They advance human health, develop better medicines, create tools for innovation and scientific discovery, and harness the power of biological processes to benefit our planet (AIMBE, Consulted in 2022). Biomedical engineering, or bioengineering, is the application of engineering principles to the fields of biology and healthcare. Bioengineers work with physicians, therapists, and researchers to develop systems, equipment, and devices to solve clinical problems. Biomedical engineers have developed a range of life-enhancing and life-saving technologies, including:

- Prosthetics, such as dentures and artificial limbs.
- Surgical devices and systems, such as robotic and laser surgery.
- Vital sign and blood chemistry monitoring systems.
- Implanted devices, such as insulin pumps, pacemakers, and artificial organs.
- Imaging techniques such as ultrasound, X-rays, particle beams, and magnetic resonance.
- Diagnostics, such as laboratory-on-a-chip and expert systems.
- Therapeutic equipment and devices, such as renal dialysis and transcutaneous electrical nerve stimulation (TENS).

The practice of biomedical engineering has a long history. One of the earliest examples is a prosthetic toe made of wood and leather found on a 3,000-year-old Egyptian mummy. Before that, even simple crutches and walking sticks were a form of engineering aid, and the first person to make a splint for a broken bone could be considered an early biomedical engineer (Lucas, 2014). In our health care society, there are many relationships that affect the level of care a patient receives. None is perhaps more important than the relationship between a patient and his doctor. As physicians strive to continuously improve the quality of their services, there is a growing need for more targeted and accurate patient data. Medical informatics provides this data. Medical informatics is the subdiscipline of health informatics that directly impacts the patient-doctor relationship. It focuses on information technology that enables effective data collection using technological tools to develop medical knowledge and facilitate medical care for patients. The goal of medical informatics is to ensure that patients have access to important medical information at the exact time and place where it is needed to make medical decisions. Medical informatics is also concerned with the management of medical data for research and teaching (Selig, 2020).

1.1 CONTEXT

Our lifestyles, global demographics, and needs as individuals are changing rapidly today, and more people than ever need healthcare. The cost of healthcare is also becoming more expensive, and with increasing demands and technological developments, major changes are underway in the healthcare value chain and business models, which will turn the healthcare system as we know it on its head.

It is time to stop seeing health and illness in static terms and to start seeing life as a dynamic process, where the maintenance of health takes place long before the onset of symptoms. Technological developments in data science have begun to impact healthcare, but have yet to realize their full potential. The digitization of healthcare over the past decade has led to a growing amount of data production. The amount of data produced is increasing exponentially, and the collection of health data from various sources is growing rapidly.

Collecting, managing, and storing all this data is very costly and may not be valuable if it is not transformed into insights that can be used by the healthcare ecosystem. For maximum optimization and usefulness, the data must be analyzed, interpreted, and utilized. Machine Learning (ML) and Deep Learning (DL) enable the generation of new knowledge from all collected data and can help reduce the cost and time burden of all healthcare systems by learning better predictions and diagnoses. These tools help to improve the entire clinical workflow, from the preventive and diagnostic phase to the prescriptive and restorative phase in healthcare management (Bohr and Memarzadeh, 2020).

Much of the cost reduction is due to the shift in the health model, from a reactive to a proactive approach that focuses on managing health rather than treating disease. This is expected to lead to fewer hospitalizations, fewer doctor visits, and fewer treatments. AI-based technology will play an important role in helping people stay healthy through continuous monitoring and coaching, and will ensure earlier diagnosis, tailored treatments, and more efficient follow-up care (Bresnick, 2017).

1.2 OBJECTIVES AND EXPECTED RESULTS

According to the American Cancer Society ACS, it was estimated that in 2021 and in the United States of America, 83,570 people were diagnosed with brain and other Central Nervous System (CNS) tumors, 24,530 malignant tumors, 59,040 nonmalignant tumors and 18,000 people will die from the disease. This data shows us the importance of this problem, but to detect brain tumors there exist various methods. The diagnosis of a brain tumor usually involves the following three steps.

- A neurological exam;
- Brain scans: (i) Computed Tomography (CT) or CT scan, (ii) Magnetic Resonance Imaging (MRI), or (iii) occasionally, an angiogram or X-ray;
- A biopsy, which is a tissue sample analysis.

The doctor will examine the patient to determine the existence of a brain tumor. The result of the examinations will determine:

- The type of brain tumor;
- Its severity, whether it is benign (non-aggressive) or malignant (aggressive).

One of the most important ways to detect tumors are images obtained by scans, but for different reasons some tumors are not detected. For example, due to low quality images, small tumor size, the tumor is in an early development stage, and finally, due to human diagnosing error. To solve this problem, there are several image segmentation methods to detect tumors, but this computer vision technique still has way to go.

Therefore, the present work will apply Artificial Intelligence (AI) to solve the problem of tumor identification, using DL models to segment the tumors present in the images obtained by scanning. The idea behind using DL models to perform semantic segmentation of images, and consequently to detect tumors, is to have a dynamic, effective, more economical and much faster method.

After reading the published literature and reviewing public projects, we expect to achieve a 60% to 65% Intersection over Union (IoU) score during model testing on the tumor segmentation task (Zheng and Guo, 18 November 2022). This will be the baseline score for comparing future implementations. We aim for a much better result after applying techniques such as hyperparameter optimization, training the models with several loss functions, and trying different model architectures.

To reach the main goal, there are other secondary objectives. First, it is necessary to search and select a suitable dataset to train and test the deep learning models.

We also intend to select and apply image preprocessing techniques. This task includes improving image quality using various computer vision techniques, such as histogram equalization.

Next, it is necessary to compare different deep neural network models and select a few for implementation, among fully convolutional neural networks, U-Net, SegNet, Tiramisu, and others.

The main outcome of the dissertation will be the implementation of the selected deep neural networks to perform the image segmentation task.

In order to obtain the best results in medical image segmentation, the optimization of the model hyperparameters is a critical step.

Different loss functions will be evaluated to find the alternative that drives the best segmentation results.

Finally, the performance of the models must be evaluated through testing. This includes the evaluation of segmentation quality, training time, model size, and inference time. The results of these evaluations will also be visualized to provide a clear understanding of the performance of each model.

1.3 DOCUMENT ORGANIZATION

In Chapter 2 we talk about images, including the types of images, the characteristics of digital images, and the processing of digital images in computer vision. Medical imaging, the evolution of medical imaging and types of medical images are also discussed. Finally, tumors are addressed, whose images will be the subject of study in this dissertation.

The third chapter addresses automatic image segmentation with deep learning models. We start by presenting neural networks, their optimization, the optimization algorithms available in DL frameworks, and some of the loss functions that are used in the optimization process. Then, some preprocessing techniques are summarized and the concept of feature maps, in the context of neural networks, is explained. Classical segmentation techniques and DL segmentation methods are presented here. To conclude, some of the neural network architectures developed specifically for the image segmentation task are detailed.

In chapter 4, we start by presenting the related work and the rest of the chapter details the methodology adopted during the development of deep learning models, to solve the image segmentation problem. The dataset used in this work and the applied preprocessing techniques are identified. Finally, the DL models that were selected for the segmentation task, U-Net and Tiramisu, are detailed, as well as some details regarding their implementation.

Chapter 5 contains the results obtained from the various experiments carried out with the U-Net and Tiramisu models. We start by identifying the models' evaluation metrics and hyperparameter optimization, with an emphasis on the learning rate. Afterward, all the evaluated loss functions are detailed: binary cross-entropy, focal, dice, Jaccard, Tversky, and functions that combine several of these losses. Finally, the results of the training and testing of the models are presented: the quality of segmentation (IOU), the training time, the size of the models, and the inference time.

Chapter 6 contains the main conclusions drawn from the dissertation and points out ideas to continue the developed work.

Finally, the appendix contains the code developed during the dissertation, as well as a compilation of the achieved results, and graphs of the architecture of the developed models.

2 MEDICAL IMAGES

In this chapter we will discuss the concepts of image, the different types of images, the characteristics of digital images, and digital image processing. Medical imaging, the evolution of medical imaging, and diverse types of medical images are also presented. Tumors, whose images will be the subject of study in this dissertation, are addressed.

2.1 COMPUTER VISION

Computer vision is the automated extraction of information from images. Information can be any feature from a 3D model, the camera position, the detection of an object, the details necessary for grouping and searching image content. In this dissertation we consider a wide definition of computer vision, to include things like image warping, image de-noising, and augmented reality. Sometimes computer vision tries to mimic human vision, sometimes uses a data and statistical approach, sometimes geometry is the key to solve problems. Practical computer vision contains a mixture of programming, modeling, and mathematics, and is sometimes difficult to grasp (Solem, 2012).

2.2 IMAGES

According to Aumont (1994), the image is always made up of deep structures related to the exercise of a language, as well as belonging to a symbolic organization (a culture, a society). The image is also a means of communication and representation of the world that has its place in all human societies. Belting (2007) says that an image is more than a product of perception, it manifests itself as the result of a personal or collective symbolization. Many of the events of the past, present, and future are known to us through images. Today, more than ever, the information we process, analyze, and synthesize at different levels, is received through images.

2.3 DIGITAL IMAGES

A digital image, or digital graphic, is a two- or multi-dimensional representation of a numerical matrix. Depending on the resolution of the image, it is static or dynamic, it can be a raster pattern (a bitmap represented by pixels) or a vector graphic (an image formed by the product of independent geometric objects like dots, lines, or polygons). Bitmap is the most commonly used image format in computing (Gómez, 2013).

An image can be defined as a two-dimensional function *f(X,Y)*, where *X* and *Y* define a spatial coordinate, and the value of *f* is called the intensity, or gray level, of the image at that coordinate. When *X*, *Y* and the amplitude values of *f* are all finite and discrete values, we are facing a digital image.

The digital image can be obtained from digital analog conversion devices, such as scanners and digital cameras (figure 1). It should be noted that it is absolutely possible to modify digital images, using filters to add or eliminate certain elements that are not available or, on the contrary, to remove those that are not desired, in the same way, the possibility to modify the size of an image and if necessary, even save it to a storage device, such as a CD or the computer's or hard drive.

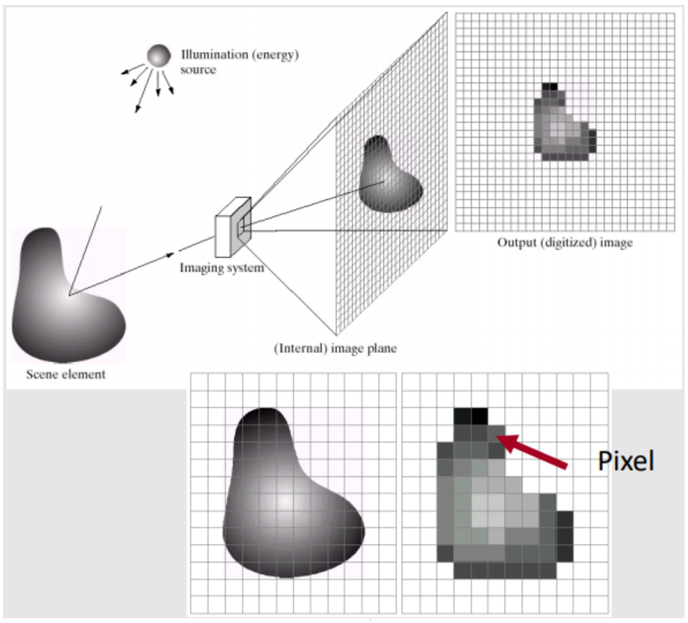


Figure 1: Capturing a digital image.

An **image histogram** is a type of histogram that acts as a graphical representation of the tonal frequency and distribution in a digital image (figure 2). Digital image processing is the set of techniques applied to digital images to improve their quality or facilitate the search for information. One of these techniques is **histogram equalization**. Histogram equalization is a nonlinear normalization transformation that extends the histogram area containing the highest frequent intensities and compresses the area containing the lowest frequent intensities (figure 3).

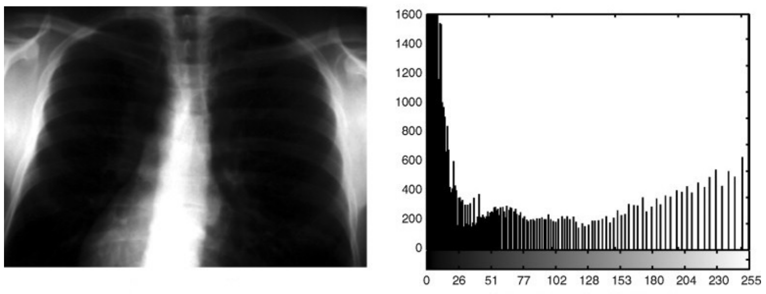


Figure 2: Histogram of a digital image.

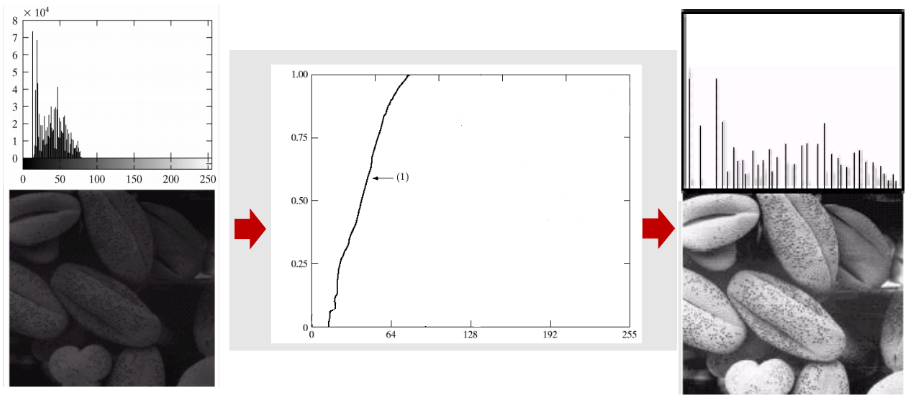


Figure 3: Histogram equalization.

Equalization of an image histogram can be expressed by the equation 1 (Sánchez, 2009).

$$S_k = T(r_k) = \sum_{j=1}^{k} p_r(r_j) = \sum_{j=1}^{k} \frac{n_j}{n} \tag{1}$$

where

- k is a value in the intensity range, for example, in the [0:255] interval.
- r_k is the input intensity.
- S_k is the output intensity, for example, a value in the range [0:1].
- n_j is the frequency of intensity j.
- n is the sum of all frequencies.
- T is the transformation applied to the pixel intensities.

2.4 DIGITAL IMAGE PROCESSING

Digital Image Processing (DIP) means the act, or discipline that studies it, of altering digital images through a digital computer and according to a certain algorithm. Unlike humans, who are limited to the visual band of the electromagnetic spectrum, machines can process virtually the entire electromagnetic spectrum, from gamma rays to radio waves. Machines can work with images generated by sources that humans are not used to, such as ultrasonic images, images from electron microscopy, and computer generated images.

There is no unanimous agreement on what topics digital image processing covers and what are its interrelationships with the close disciplines of computer vision and computer graphics. Since the 1960s, digital image processing has gradually become one of the most important areas of scientific research. However, some image processing algorithms require large processing power, and their development was limited to a few specialists and companies. But with the rapid development of computers, many people showed an enormous interest in image processing. The development of image processing is being further accelerated with the rapid advancement of technology. Computing and image acquisition systems are available with ever more storage capacity, and display devices with ever more spatial resolution and color depth.

In a broad sense, digital image processing involves 2D/3D image recognition, image analysis, image manipulation, image transmission, and other related topics. Examples of image manipulations are intensity transformation, filtering, frequency domain processing, restoration, compression, morphological processing, segmentation, describing the content, pattern and object recognition, and interpretation.

2.5 MEDICAL IMAGES

Imaging, or medical images, is used to reveal, diagnose, and examine diseases or to study the anatomy and functions of the body. Radiology, thermography, endoscopy, microscopy, and medical photography are techniques that allow images of the human body to be obtained for clinical or scientific purposes. Other procedures that allow us to obtain data that can be represented as maps or diagrams, such as electroencephalography, can also be included in imaging. Images play an important role in modern medicine. Modern imaging techniques, including X-rays, ultrasound, CT scans, and MRI, can show the internal structures of the human body in great detail.

Imaging is a range of tests used to create images of parts of the body. They can help identify possible health conditions before symptoms appear, diagnose the probable cause of existing symptoms, monitor health conditions that have been diagnosed, or follow the effects of ongoing treatments. Imaging is also called radiology, and imaging specialists are called radiologists.

There are many different types of imaging, such as X-rays, CT scans, MRI, and ultrasound. Each type uses a different technology to create images. The growing range of image types provides health professionals with many options to examine what is happening inside the patient's body. Radiology technicians, or imaging technologists, are health professionals who are trained to work with a specific type of image. For example, radiographers were trained to work with X-rays and sonographers were trained to work with ultrasound imaging.

2.6 The Evolution of Medical Imaging in Clinical Research

Medical imaging plays an instrumental role in the clinical development of new life science products. Although the medical imaging industry is in constant flux, due to increased investment in medical imaging companies and mergers and acquisitions, the adoption of novel imaging technologies to support clinical trials for the pharmaceutical, biotech, and medical device industries continues to increase. In fact, centralized imaging data is now used as a primary endpoint in many clinical research studies (Khaleel, 2017). The Food and Drug Administration Modernization Act (FDAMA) opened the door to the use of imaging modalities as a product development tool in clinical trials with medical device or pharmaceuticals, by allowing data generated through imaging modalities to be included in regulatory presentations (Food and Drug, 2017).

In March 2004, the Food and Drug Administration (FDA) launched the Critical Path Initiative (CPI), which launched the use of modern scientific and technological tools to predict the safety, efficacy, and manufacturing capacity of medical products, and to improve the accuracy of results during the investigation and testing phases (Food and Drug, 2017). In view of this, in August 2011 the FDA published the Manual of Process Standards and Parameters for Diagnostic Imaging Tests (for Drug Evaluation et al., 2020). This document describes the standards that sponsors can adopt to ensure that clinical imaging data is obtained in a way that:

- Complies with the protocol and test standards;
- Maintains the quality of the image data;
- Provides a verifiable record of the imaging process.

The support of this agency has increased the adoption of images in clinical trials and therefore has contributed to the growth of the market for basic laboratory imaging services. According to Markets and Markets, in 2016 the size of the global clinical imaging testing market was 773.4 million dollars annually, with a 6 to 8 % growth between 2016 and 2020 (Market and Market, 2016).

2.7 PARTICULARITIES OF MEDICAL IMAGES

"Sometimes a picture is worth more than a thousand words”. Diagnostic images are the set of studies that, through technology, collect and process images of the human body. The main role of imaging studies in medicine is to provide the information the doctor needs to make a diagnosis of the patient’s disease (Díaz, 2014). Next, we present a short overview of the most widely used medical image types.

2.7.1 *X-rays*

On November 8, 1895, the German physicist Wilhelm Conrad Roentgen discovered X-rays (figure 4). X-rays are a type of electromagnetic radiation, ionizing due to their short wavelength (0.01 to 10 nanometers) that has the ability to interact with the body tissues (Díaz, 2014).

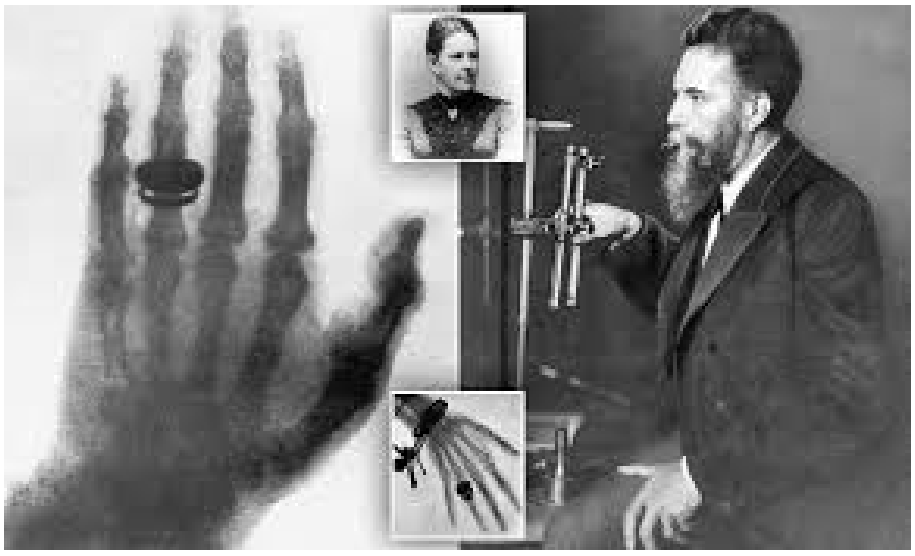


Figure 4: Daily Mail Roentgen’s first human X-ray of his wife’s hand (1895).

2.7.2 *Ultrasounds*

Ultrasonic signals applied to industrial and medical applications are rooted in nature. In 1779, the biologist Lazzaro Spallanzani discovered a class of waves associated with bat hunting activity. Ultrasounds are sound waves with a frequency higher than the upper audible limit of human hearing. Ultrasounds are used in many different fields, for example, to detect objects and to measure distances. Ultrasound imaging, or sonography, is often used in medicine (figure 5) (Rodríguez et al., 2007).

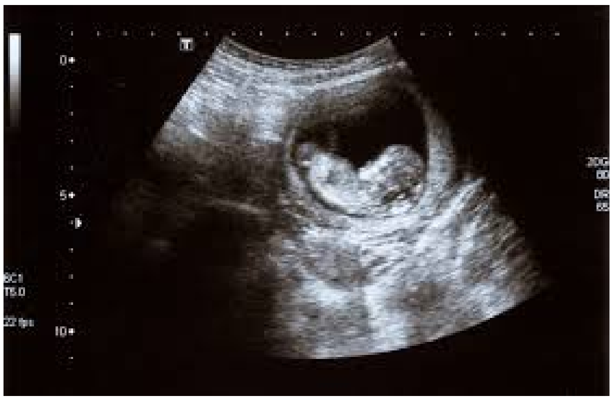


Figure 5: Ultrasound image.

2.7.3 *Computed Tomography*

Undoubtedly, CT is one of the radiological modalities that since its creation has evolved more and established more quickly in daily clinical practice. Current medicine does not tolerate diagnostic uncertainty, and doctors of any specialty aim to make a reliable diagnosis of the diseases of patients who come to the consultation. In most cases, CT allows for a reliable diagnosis. In 1973, Godfrey Hounsfield described CT, and six years later he received the Nobel Prize for Medicine. Through CT it is possible to study almost any organ of the body and its pathology in great detail (figure 6) (Bastarrika, 2007).

2.7.4 *Medical Photography and Microscopy*

Since medicine is a multidisciplinary subject, medical photography is also multidisciplinary. Medical photography can be used in diagnosis, communication between peers, and teaching. On the other hand, we can speak of portraying medical processes, patients, parts of the human body, or organs (figure 7). It is also possible to speak of medical photography, having as object of study the very actors, doctors, and other health technicians, as well as institutions.

From a very early age, photography was applied in the field of psychiatry. In 1852 the French neuron physiologist Guillaume Duchenne (1806-1875) began his studies by photographically documenting experiences that related physiognomic expression to the

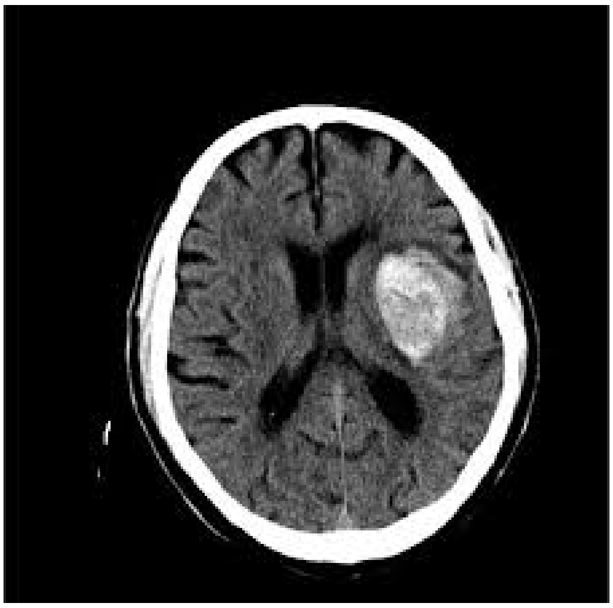


Figure 6: Computed tomography image.

stimulation of muscles made by Faradic currents. He published the result of these experiences illustrated with photographs in the monography *Mécanisme de la Physionomie Humaine*. This publication is considered one of the first scientific publications with photographic illustrations (Duchenne, 1862) (Peres, 2014).

Since the end of the twentieth century, microscopy has evolved, including new features that improve its practice. Among them, the virtual microscope highlights the synergy between disciplines such as pathology, histology, medical informatics, and image analysis. This technology has changed many paradigms in research, diagnosis, education, and medical training. Several decades ago, cameras were incorporated into microscopes to record images of sporadic experiments. Today, the use of digital cameras, coupled with microscopes, is a common situation.

2.7.5 *Magnetic Resonance Imaging*

MRI is one of the main medical diagnostic tools. MRI makes it possible to view and analyze a variety of characteristics of the organism, such as blood flow, blood distribution, and various physiological and metabolic functions. Think of it as a multipurpose image and analytical procedure that can be configured and optimized to answer a wide range of clinical questions for virtually all parts and systems of the body. It is a medical imaging process that uses a magnetic field and radio frequency (RF) signals to produce images of anatomical structures. MRI produces images that are clearly different from images produced by any

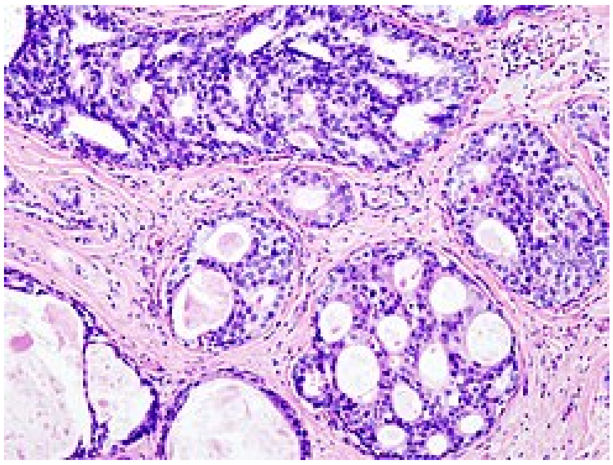


Figure 7: Growth of malignant cells in the breast tissue.

other imaging modality. A major difference is that the MRI process can obtain selective images of various parts of the body (Sprawls, 2000).

2.8 TUMORS AND PARTICULARITIES OF TUMORS

Tumor is an abnormal mass of tissue that forms when cells grow and divide more than they should or don't die when they should. Tumors can be benign (not cancerous) or malignant (cancerous). Benign tumors can grow very large but do not spread or invade nearby tissues or other parts of the body. Malignant tumors can spread or invade nearby tissues. They can also spread to other parts of the body through the blood and lymph systems. They are also called a neoplasm. Willis (1960) proposed a workable definition to distinguish true tumors from inflammatory and reparative proliferations, hyperplasias, and malformations with excessive tissue. According to this pathologist, a tumor is "an abnormal mass of tissue, the growth of which exceeds and is uncoordinated with that of the normal tissues, and persists in the same excessive manner after cessation of the stimuli which evoked the change". Every pathologist can think of exceptions, but these do not invalidate the general applicability of the definition. The commonly used and most useful classification of tumors is histogenetic, i.e., the tumors are named according to the tissues from which they arise and of which they consist. In most tumors, the neoplastic tissue consists of cells of a single type, and with experience, one can readily classify them. The types of histological differentiation found in tumors appear to be inherent in the parent tissues. Foulds (1940) concluded that most adult normal cells have a greater capacity for divergent differentiation than was previously supposed and that tumor cells are unlikely to acquire new capacities. The few types of tumors in which there is uncertainty about the precise tissue of origin require further histopathological research. Meanwhile, in these cases we must settle for non-committal identifying names (Murphy, 2007).

2.9 THE PROBLEM AND ITS DELIMITATION

The focus of the present work is the semantic segmentation of brain tumors in medical images. Brain tumors include the most threatening types of tumors. Glioma, the most common primary brain tumor, occurs due to the carcinogenesis of glial cells in the spinal cord and brain. Glioma is characterized by several histological and malignancy grades, and an average survival time of fewer than 14 months after diagnosis for patients with glioblastoma.

MRI is a popular noninvasive imaging technique that produces a large and diverse number of tissue contrasts and has been widely used by medical specialists to diagnose brain tumors. However, the manual segmentation and analysis of structural MRI images of brain tumors is an arduous and time-consuming task which, so far, can only be accomplished by professional neuroradiologists. Therefore, an automatic and robust brain tumor segmentation will have a significant impact on brain tumor diagnosis and treatment. Furthermore, it can also lead to a timely diagnosis and treatment of neurological disorders such as Alzheimer's disease (AD), schizophrenia, and dementia.

An automatic technique for lesion segmentation can help radiologists deliver key information about tumor volume, location, and shape, including enhancing tumor core regions and whole tumor regions, to make therapy progress more effective and meaningful. There are several differences between the tumor and its normal adjacent tissue, which hinder the effectiveness of segmentation in medical imaging analysis, e.g., size, bias (an undesirable artifact due to improper image acquisition), location, and shape. Several models that try to find accurate and efficient boundary curves of brain tumors in medical images have been implemented in the literature. In spite of tireless efforts of researchers, as a key challenge, accurate brain tumor segmentation still remains to be solved, due to various challenges such as location uncertainty, morphological uncertainty, low contrast imaging, annotation bias, and data imbalance.

Given the high time that a specialist needs to analyze a magnetic resonance image, the automatic segmentation of these images constitutes a good challenge. To develop the present work, a reliable data source will be selected, which includes magnetic resonance images and the respective masks, and which allows training neural networks to perform the semantic segmentation task.

The project will be developed in Python, using a few libraries of the Python ecosystem, namely TensorFlow, Keras and scikit-learn. Training and evaluation of DL models will be carried out mainly on the Google Collaboratory platform. The DL models that will be selected to solve the image segmentation problem are those that are considered the most capable of performing the medical image segmentation task, namely U-Net and Tiramisu. Throughout the work, different neural network architectures and different loss functions will be experimented, and the hyperparameters of the models will be fine-tuned, in order to select the best solution.

3

IMAGE SEGMENTATION WITH DEEP NEURONAL NETWORKS

How to process images with machine learning for object recognition and classification? The identification of objects in images has multiple applications: from something as prosaic as identifying cats or dogs in photographs to the detection of tumors in medical exams. This chapter will review in detail all aspects and techniques to perform automatic image segmentation.

3.1 DEEP LEARNING APPLIED TO MEDICAL IMAGES

With the rapid development of AI technology, using AI technology in clinical data mining has become a major trend in the medical industry. The use of advanced AI models for medical image analysis, one of the critical parts of clinical diagnosis and decision-making, has become an active research area in both industry and academia. Recent applications of DL in medical image analysis involve various computer vision-related tasks such as classification, detection, segmentation, and registration. Among them, classification, detection, and segmentation are the most fundamental and widely used tasks.

Although there exist a number of reviews on DL methods on medical image analysis, most of them emphasize either general deep learning techniques or specific clinical applications. Deep learning is a rapidly evolving research field, with numerous state-of-the-art works proposed every year. In this chapter, we review the latest developments in the field of medical image analysis. Applications of medical imaging to different types of disease are also presented, current problems are discussed, and possible solutions and future research directions are provided (Liu et al., 2021).

Deep learning is a class of machine learning algorithms characterized by the use of neural networks with several layers of artificial neurons capable of processing data, understanding human speech, and visually recognizing objects. DL provides state of the art solutions for Computer Vision (CV) problems, including classification, object detection, and image segmentation.

Medical image analysis is an invaluable tool in medicine as it is a critical component in diagnosis and treatment planning. Recent results of applying DL and ML models in the field of healthcare have surprised even the most experienced physicians, as they can help in early detection of diseases, allowing patients to have better treatments and even cure. The biggest current challenges in healthcare are:

- Identify and target diseases;
- Classify tumors according to their malignancy;
- Augment available datasets dedicated to medical imaging to enable effective training of increasingly capable ML and DL models;
- Train DL models on large datasets for disease prediction;
- Predict diseases with high accuracy.

3.2 NEURAL NETWORKS

Perceptron is the simplest unit that can be learned in a biologic or artificial neural network. In AI, the perceptron is a mathematical model inspired by the behavior of biologic neurons, which are cells of the nervous system responsible for transmitting information in the form of electrical signals. The perceptron architecture is illustrated in figure 8 and its operation can be described by the following steps:

- The perceptron receives a set of input signals (dendrites) and produces an output signal.
- The input signals are multiplied by the weights and combined to produce an activation signal.
- The activation signal also depends on an additional input, the bias, which shifts the activation level.
- The activation signal is passed through the activation function to yield the output signal. The activation function can be simply modeled by a step curve that enables or disables the activation signal to pass through.

In neural networks, **activation function** decides how the output of the weighted sum of the inputs is converted to the output signal of the neuron. This function is activated if the value of the neuron's activation signal is large enough to be passed to the next neuron. There are several activation functions, illustrated in figure 9, and the ones most commonly used in neural networks are the sigmoid, SoftMax, Rectified Linear Unit (ReLU), and hyperbolic tangent (`tanh`). Next, we summarize these functions.

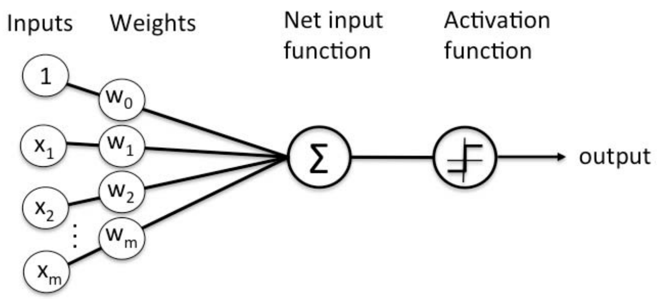


Figure 8: The perceptron architecture.

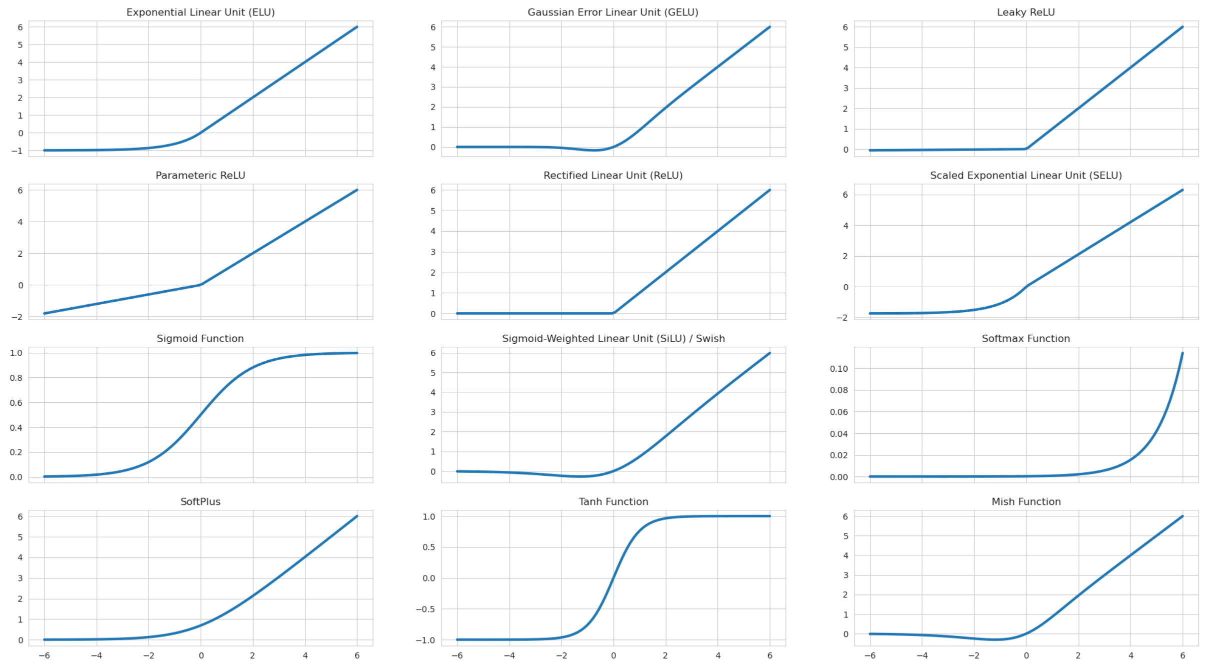


Figure 9: Activation functions.

- The linear activation function is modeled by a straight line (equation 2). Therefore, the output of the function will not be restricted to any range. The linear function does not help to learn the complexity of the data that neural networks usually receive.

$$y = w.x + b \tag{2}$$

- The sigmoid activation function is a real, differentiable, bounded function that is defined for all real input values and has a non-negative derivative at each point and exactly one inflection point (equation 3). The sigmoid function and the sigmoid curve refer to the same entity.

$$y = \frac{1}{1 + e^{-}(w.x + b)} \tag{3}$$

- The softmax function, also known as softargmax or normalized exponential function, is a generalization of the logistic function to multiple dimensions (equation 4). It is used in multinomial logistic regression, and it is often used as the last activation function on a neural network, to normalize the output to a probability distribution over predicted output classes, based on Luce's choice axiom.

$$softmax(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{4}$$

- The hyperbolic tangent activation function produces an output in the range [-1, 1] (equation 5). The hyperbolic tangent has the same advantages as sigmoid, and it solves one of its problems, being centered at zero. Its derivative also converges to zero and more quickly.

$$y = tanh(w.x + b) \tag{5}$$

- The rectified linear unit is the most widely used activation function in deep learning models. This function returns 0 if it receives a negative input and returns the input value if it is positive (equation 6).

$$y = max(0, w.x + b) \tag{6}$$

3.3 HOW AND WHEN TO ADOPT DEEP LEARNING MODELS

DL models have been applied in several fields with great results, including image classification, object detection, image segmentation, natural language translation, handwritten text recognition, digital assistants, playing games such as Go, chemical retrosynthesis, or classification of protein and DNA sequences. Naturally, DL is not the best choice to solve all problems. The factors that most influence the success of DL are:

- In most cases, data on a large scale is necessary to train DL models effectively. When we have little data, other models can give more consistent results than DL models, with less computational cost.
- The availability of computational resources, namely Graphics Processing Units (GPUs), is necessary to accelerate the training of very deep neural networks.
- Choosing the correct activation functions.
- Adopting a better method to initialize the weights can improve the model's capabilities and accelerate training.
- Try different optimization algorithms such as RMSprop, Stochastic Gradient Descent (SGD), or Adam, to find the one that leads to the best results.
- Apply methods to address the models' overfitting and underfitting.
- Whenever possible, adopt transfer learning, using pre-trained models on large datasets.

3.4 MODEL OPTIMIZATION

There are several algorithms that can be used to train/optimize neural networks, all based on gradient descent methods and the original idea of backpropagation (or chain rule). Over the past few years, several improvements have been made to the initial algorithm, resulting in new and more efficient algorithms, including RMSProp, Adam, and Adadelta. Optimization algorithms include a set of parameters that can be optimized like any other hyperparameter, among which **learning rate** α stands out:

- **Batch size** is the term used in machine learning to refer to the number of training examples used in an iteration. The batch size can assume one of three options:
  - In **batch mode** the batch size is equal to the size of the dataset, making an iteration and an epoch equivalent.
  - In **mini-batch mode** the batch size is greater than one, but less than the size of the dataset. Typically, it is a power of 2 by which the size of the data set is divisible.
  - In **stochastic mode** the batch size is one. Therefore, the gradient and neural network parameters are updated after forwarding each sample.

- **Stochastic gradient descent** and its variants are probably the most commonly used optimization algorithms in machine learning. It is possible to get one unbiased gradient estimation taking the mean gradient in a mini-batch of the data generation distribution. The parameters of the model, θ, are updated in each iteration of the training process with the rule defined by equation 7.

$$\theta_t = \theta_{t-1} - \alpha.\nabla_\theta L(\theta_t, X, y) \tag{7}$$

  Where θ denotes the parameters of the model, L is the loss function, and $\nabla_\theta L()$ is the gradient that gives us the direction that decreases the loss function L, and α is the learning rate.

- Although the optimization algorithm SGD is widely used in ML/DL, it is slow to converge to the minimum of the loss function. **Momentum**, or stochastic gradient descent with momentum, is an extension of the SGD algorithm thought to accelerate learning, especially when faced with a loss function with high curvature. The main idea behind momentum is to compute an exponentially weighted moving average of the gradients and use this average to update the model's weights, instead of using a single gradient. When past gradients are taken into account, the optimization trajectory becomes more smooth since the oscillations on the loss function are reduced.

  In each time step t of the training process, the exponentially weighted moving average of the gradients (v_t) is calculated by equation 8, and the parameters of the model (θ_t) are updated with the rule defined by equation 9. The variable v_t plays the role of a velocity and determines the direction and speed with which the parameters θ move on the parameter's space.

$$v_t = \beta.v_{t-1} + (1-\beta).\nabla_\theta L(\theta_t, X, y) \tag{8}$$

$$\theta_t = \theta_{t-1} - \alpha.v_t \tag{9}$$

  Where β is a hyperparameter that denotes the momentum constant, θ are the model parameters, L is the loss function, $\nabla_\theta L()$ is the loss function gradient L, and α is the learning rate.

  The name momentum derives from a physical analogy in which the negative gradient is a force that moves a particle through the parameter's space according to Newton's laws of motion. In the momentum algorithm, we assume a unitary mass, so the velocity v can also be thought of as the particle's momentum. The momentum constant β lies at the interval $[0, 1)$. The larger the momentum constant, the more past gradients are taken into account when computing the exponentially weighted moving average, resulting in smoother updates. However, choosing a value for β that is too close to 1 results in over-smoothing. A common value for β is 0.9.

- Instead of blindly updating the model parameters, using the direction given by the loss function gradient, the **Nesterov accelerated gradient** algorithm incorporates future information into the current update of the model parameters (Nesterov, 1983). We know that in step t we will use the momentum term $\beta.v_{t-1}$ to update the parameters θ_t. Thus, $\theta_t - \beta.v_{t-1}$ is an approximation of the next value of the parameters. We can now effectively look ahead and calculate the loss gradient, not with respect to the current parameters θ_t but with respect to the approximate future position of the parameters (equation 10). Again, the momentum constant β is defined with a value around 0.9.

$$v_t = \beta.v_{t-1} + (1-\beta).\nabla_\theta L(\theta_t - \beta.v_{t-1}, X, y) \tag{10}$$
$$\theta_t = \theta_{t-1} - \alpha.v_t \tag{11}$$

- **Adagrad** is an algorithm that optimizes the learning rate of SGD (Ruder, 2016). It does this by adapting the learning rate to the parameters, making larger updates to the less frequent parameters, and smaller updates to the more frequent ones. Thus, it is suitable for dealing with sparse data. Adagrad increases the robustness of SGD, and improves its performance in large neural networks. This optimizer avoids manually adjusting the learning rate. Adagrad applies a different learning rate to each parameter θ_i, at every time step t. The Adagrad update rule is given by equation 12.

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\alpha}{\sqrt{G_{t,ii} + \epsilon}}.\nabla_{\theta_t} L(\theta_{t,i}, X, y)) \tag{12}$$

  Where $G_t \in \mathbb{R}^{dxd}$ is a diagonal matrix where each element (i,i) is the sum of the squared gradients of L with respect to θ_i, with the sum calculated for all times up to t.

- **Adadelta** is an extension of Adagrad that aims to moderate its aggressive reduction in learning rate. Instead of accumulating all the previous squared gradients, Adadelta accumulates only the previous gradients that lie within a fixed window w. In practice, instead of storing w previous squared gradients, the algorithm stores a sum of all squared gradients that is recursively defined by a decaying average.

  The Root Mean Squared (RMS) error of parameter updates is given by equation 13.

$$RMS[\Delta\theta]_t = \sqrt{E[\Delta\theta^2]_t + \epsilon} \tag{13}$$

Since $RMS[\Delta\theta]_t$ is unknown, it is approximated by the RMS of the parameter updates until the previous time step. Changing the learning rate by $RMS[\Delta\theta]_{t-1}$, the Adadelta update rule is given by equations 14 and 15.

$$\Delta\theta_t = -\frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t}.g_t \tag{14}$$

$$\theta_{t+1} = \theta_t + \Delta\theta_t \tag{15}$$

To simplify the notation, the gradient of the loss function with respect to the parameters θ at the time step t, $\nabla_{\theta_t} L(\theta_t, X, y)$, was replaced by g_t.

With Adadelta, it is not necessary to define the default learning rate, since it was eliminated from the update rule.

- **Adam**, or Adaptive Moment Estimates, is another algorithm that computes a learning rate for each parameter θ_i. It computes an exponentially decaying average of past squared gradients (v_t in equation 17), like Adadelta and RMSprop do, but it also calculates an exponentially decaying average of past gradients (m_t in equation 16), similar to momentum.

$$m_t = \beta_1.m_{t-1} + (1-\beta_1).\nabla_{\theta_t} L(\theta_t, X, y) \tag{16}$$

$$v_t = \beta_2.v_{t-1} + (1-\beta_2).\nabla^2_{\theta_t} L(\theta_t, X, y) \tag{17}$$

Where m_t and v_t are estimates of the first and second moments of the gradients, respectively. Since both terms are biased towards zero, it is convenient to correct them using equations 18 and 19.

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t} \tag{18}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t} \tag{19}$$

With these two estimates we can update the weights using the rule expressed by equation 20. The recommended value for β_1 is 0.9, for β_2 is 0.999, and for ϵ is 10^{-8}.

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\hat{v}_t} + \epsilon}.\hat{m}_t \tag{20}$$

3.5 LOSS FUNCTIONS

Machine learning learns through a loss function. It is a method that calculates how well the algorithm models the data. If the predictions deviate too far from the expected outputs, the optimization algorithm adjusts the weights to minimize the error. Gradually, with the help of a loss function. There is no unique loss function suitable for all machine learning algorithms. Several factors influence the selection of a loss function for a specific problem, such as the type of ML problem to solve, the optimization algorithm being used, how easy it is to calculate the derivatives (gradients) and, to some extent, the percentage of outliers present in the dataset.

Broadly speaking, the loss functions can be classified into two main categories depending on the type of learning task we are addressing: loss functions for classification and loss functions for regression. In classification, we try to predict an output among a set of finite categorical values. On the other hand, in regression, the model predicts a continuous value.

Loss functions for regression:

- As the name suggests, Root Mean Square Error (RMSE) / Square Loss / L2 loss is defined as the mean of the square difference between the actual observations y_i and the predictions $\hat{y}_i$.

  $$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{21}$$

  RMSE is concerned only with the average magnitude of the error, regardless of its direction. However, due to the square, predictions that are far from the actual values are heavily penalized compared to less deviated predictions. Mean Square Error (MSE) also has interesting mathematical properties; for example, it is easier to calculate gradients.

- On the other hand, the Mean Absolute Error (MAE) / L1 loss is measured as the average of the absolute differences between the actual observations and the predictions. Like MSE, it calculates the magnitude of the error without considering its direction. Unlike MSE, calculating the gradient of MAE is more complex, and requires a method such as linear programming. On the other hand, MAE is more robust to outliers, as it does not square the errors.

- Compared to previous losses, Mean Bias Error (MBE) is much less common in the ML field. MBE is similar to MAE, with the only difference that MBE does not consider absolute values. Care must be taken, as positive and negative errors can cancel each other out. Although less accurate in practice, it can determine whether the model has a positive or negative bias.

$$MBE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i) \tag{22}$$

Loss functions for classification:

- **Hinge loss**: The score associated with the correct class must be greater than the sum of the scores of all incorrect classes by some margin of safety (usually one). Hinge loss is used on the maximum-margin rating, most notably on Support Vector Machines (SVMs). Although it is not smooth, it is a convex function that facilitates working with the common convex optimizers used in ML. The hinge loss is defined by equation 23, where y is the actual output (either 1 or -1), and $\hat{y}$ is the output of the classifier.

  $$l(\hat{y}) = max(0, 1 - y \cdot \hat{y}) \tag{23}$$

  The objective of optimizing an SVM classifier is to minimize the term w, which is a vector orthogonal to the hyperplane that separates our data (equation 24). The vector w is used to project the data points.

  $$\min_{w} \frac{1}{2}\sum_{i=1}^{n} w_i^2 \tag{24}$$

  The minimization objective expressed by equation 24 corresponds to the primal form of the hard margin SVM, which does not accept classification errors. In soft-margin SVM, the loss function combines the minimization objective with a loss function such as hinge loss (equation 25).

  $$\min_{w} \frac{1}{2}\sum_{i=1}^{n} w_i^2 + \sum_{j=1}^{m} max(0, 1 - \hat{y}_j \cdot y_j) \tag{25}$$

  The first term in equation 25 is a sum over the number of features (n), while the second term is a sum over the number of samples in the dataset (m). $\hat{y}$ is the output predicted by the model, and is calculated as the product of the weight parameter w and the input x: $\hat{y}_i = w^T x_j$.

- **Cross-entropy loss**, or **log loss**, measures the performance of a binary classifier whose output is a probability value between 0 and 1. The cross entropy loss increases as the predicted probability diverges from the actual class (equation 26). This loss heavily penalizes predictions that are confident but wrong.

$$CrossEntropyLoss(y, \hat{y}) = -(y_i.\log(\hat{y}_i) + (1 - y_i).\log(1 - \hat{y}_i)) \tag{26}$$

When the true output class is 1 ($y_i = 1$), the second term of the loss is zero. If the true output class is 0 ($y_i = 0$), the first term of the loss is zero.

3.6 DATA NORMALIZATION

Normalizing means compressing or expanding the variable values so that they fall within a defined range. Among the possible normalization methods, we summarize two of them next.

- In **variable scaler**, each input x_i is normalized to fit the $[0:1]$ interval, as specified by equation 27.

$$X_{normalized} = \frac{X - X_{min}}{X_{max} - X_{min}} \tag{27}$$

  The problem with this normalization is that it compresses the input data to fit a fixed $[0:1]$ interval. This means that when data has a few strong outliers, most of the data values will be compressed to a very narrow interval.

- **Standard scaler** is an alternative to variable scaling that applies another technique, known as the standard scaler. The mean μ of each input variable is subtracted from the variable's value and the resulting difference is divided by the variable standard deviation σ (equation 28).

$$X_{normalized} = \frac{X - \mu_x}{\sigma_x} \tag{28}$$

3.7 DATA PREPROCESSING

According to Techopedia, data preprocessing is a data mining technique that involves transforming raw data into an understandable format. Real-world data is often incomplete, inconsistent, and/or lacks certain behaviors or trends, and it is likely to contain errors. Data preprocessing is a proven method to solve these problems. Data preprocessing is one of the most important steps of data mining that deals with fetching, preparing, and transforming data from the dataset. Data preprocessing provides operations that can reorganize the data in a form that is more understandable, and that makes it easier to apply data mining techniques. There are mainly four data preprocessing methods: data cleaning, data integration, data transformation, and data reduction.

Data cleaning is the process of identifying and correcting, or removing, errors, inconsistencies and inaccuracies in data. This is done to ensure that data is accurate, consistent and useful for analysis and decision-making purposes. The goal of data cleaning is to improve the quality of the data and make it ready for use in various applications.

Data integration is the process of combining data from multiple sources into a single and unified view. This can include combining data from different databases, spreadsheets, or other systems into a single data repository, or mapping data from different sources to a common schema to enable cross-source analysis. The goal of data integration is to provide a single version of the truth that can be used for business intelligence and decision making.

Data transformation refers to the process of converting data from one format, or structure, to another in order to make it more suitable for analysis, modeling or further processing. This can involve mapping data from multiple sources into a common format, aggregating or summarizing data, or transforming data into a different format, or structure, to support specific use cases. The goal of data transformation is to prepare data for analysis and reporting, and to make it easier to work with in a variety of applications.

Data reduction refers to the process of reducing the size and complexity of a dataset, by removing or aggregating redundant or irrelevant information. This can be achieved through techniques such as data compression, feature selection, and dimensionality reduction. The goal of data reduction is to simplify the data and make it easier to analyze and process, while still retaining its important features and characteristics.

3.8 IMAGE SEGMENTATION

Image ranking, or classification, is a task that assigns a unique label to an input image. It was one of the first areas in which DL made an important contribution to medical image analysis. The difficulty of image classification results from image variation, which in turn can result from different lighting conditions, different scales, from object deformation, intra-class variation, and some objects being partially hidden. In the case of MRI of the brain, images usually do not suffer from most of the aforementioned problems. Problems such as occlusion, deformation, and background disorder do not occur in MRI, since the acquisition task is very controlled and the images are very similar. Variations can occur in (i) light conditions, due to different acquisition parameters, (ii) scale variations, due to a great difference in the age of the patients (a baby brain is smaller than an adult brain), or (iii) intraclass nuances (Fontainhas, 2018).

Image segmentation is usually defined as identifying the set of pixels that belong to a specific substructure, tissue, or other material of interest. It is the most popular and common task addressed in papers that apply DL to medical images. Some examples of DL models that were successfully applied to image segmentation are Fully Convolutional Networks (FCNs), VGG, SegNet, DeepLab, and U-NET (Fontainhas, 2018).

Image segmentation is a crucial task in computer vision that aims to divide an image into smaller, meaningful parts for easier analysis. The segmented components are typically objects or parts of objects, represented by groups of pixels or superpixels. By classifying pixels into larger segments, image segmentation reduces the complexity of image analysis by eliminating the need to analyze individual pixels. There are three levels of image analysis that benefit from image segmentation:

- **Classification** categorizes an entire image into a class, such as "people", "animal", "outdoor".
- **Object detection** aims to locate objects in an image, for example, by drawing a rectangle around each object.
- **Segmentation** identifies parts of the image and understand what object they belong to. Segmentation goes a step further in identifying content in images, than object detection and classification.

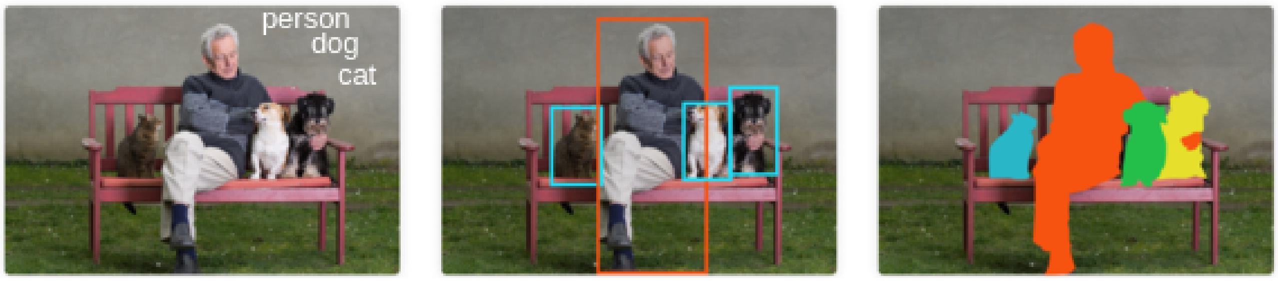


Figure 10: Classification, detection, and segmentation.

Image classification, image segmentation, object detection, and **instance segmentation** are all tasks in computer vision, but they differ in their goals and output.

- **Image classification** is the task of assigning a label to an entire image, indicating what object or class of objects is present in the image.
- **Image segmentation** involves dividing an image into segments or regions, each corresponding to a different object or part of an object.
- **Object detection** is the task of locating objects within an image and drawing a bounding box around each object. It combines both image classification and image segmentation.

- **Instance segmentation** is a more advanced version of object detection, where not only the bounding boxes of objects are predicted, but also the segmentation masks for each individual object within the image.

In summary, image classification focuses on categorizing an image into one or multiple classes, image segmentation separates an image into regions, object detection identifies objects and draws bounding boxes around them, and instance segmentation extends object detection by also generating segmentation masks for individual objects. Image segmentation can be divided into two main variants: semantic segmentation and instance segmentation. There is a third variant, called panoptic segmentation, that is not detailed here.

- **Semantic segmentation** classifies each pixel in an image into a class, among a predefined set of classes. The classes are semantically interpretable and correspond to real-world categories. For example, we can isolate all the pixels associated with cats and assign them a single color, for example, green. This is also known as dense classification because we predict the meaning of each pixel.

  In the example given in figure 11, pixels are classified into three classes: "person", “bike”, and "background". Usually, in an image with various entities, we want to know which pixel belongs to which entity. Semantic segmentation is different from object detection, as it does not predict any bounding boxes around the objects. In semantic segmentation, we do not isolate the different instances of the same class of objects. For example, there are multiple bikes in figure 11 and we assign the same label to all of them (Hack, 2019).

- In **instance segmentation** we want to identify each instance of every class of objects present in the image. In figure 12 there are three sheep. Semantic segmentation would classify all pixels belonging to the class "sheep" with the same label, while instance segmentation identifies each individual sheep and assigns a different label (blue, cyan, green in this case) to the pixels that belong to different instances of the class sheep.

Segmentation techniques can also be classified based on the domain in which they operate, namely the measurement space, spatial domain, and frequency domain. Segmentation techniques in the **measurement space** operate on the raw data values of the image, such as intensity, color, or texture. These techniques include **thresholding**, region growing, and clustering. Segmentation techniques in the **spatial domain** operate on the spatial arrangement of the image pixels, using information such as gradient, edge, or boundary information. These techniques include edge detection, region splitting, and region merging. Finally, segmentation techniques in the **frequency domain** operate on the frequency representation of the image, using the Fourier Transform or other frequency-domain representations. These techniques include wavelet analysis, singular value decomposition, and principal component

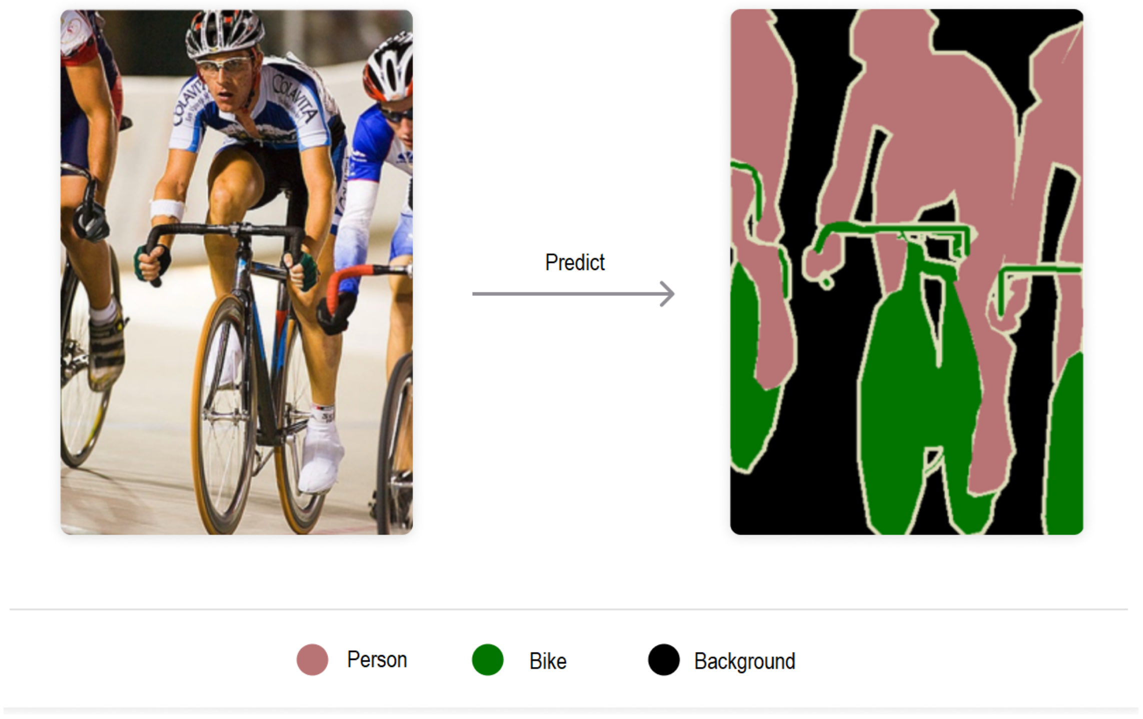


Figure 11: Semantic segmentation.

analysis. Each of these domains offers different advantages and disadvantages and different techniques may be better suited for different types of images or applications.

3.9 TRADITIONAL IMAGE SEGMENTATION METHODS

Naturally, before the emergence of deep learning techniques, image segmentation was already performed using more traditional techniques. The segmentation techniques that were commonly used in the past are less efficient than their deep learning counterparts, because they use rigid algorithms and require human intervention and experience. We next summarize a few of these traditional segmentation techniques.

- **Thresholding** splits an image into foreground and background. A specified threshold value separates the pixels into one of two levels to isolate objects. Thresholding converts a grayscale image to a binary one, or it distinguishes the lightest pixels from the darkest ones in a color image (figure 13).
- **K-means clustering** is an algorithm that identifies K groups in the data. The algorithm assigns each data point, or pixel, to one group. The group is selected according to the

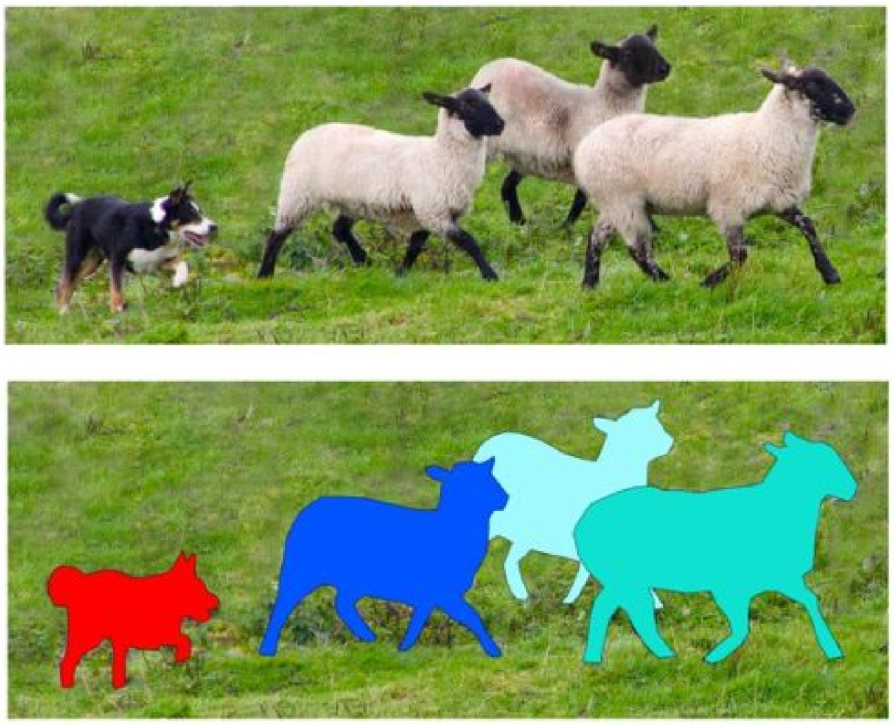


Figure 12: Instance segmentation.

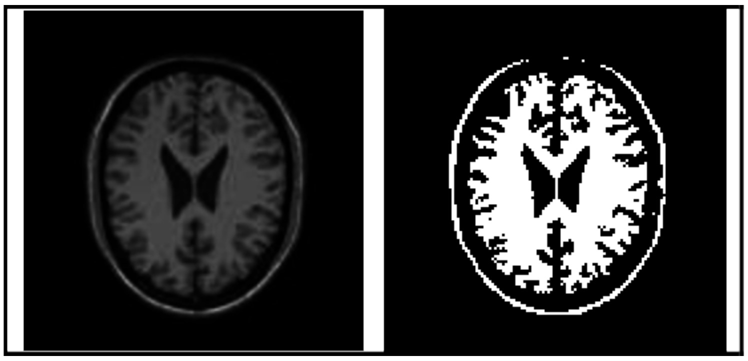


Figure 13: Thresholding segmentation.

highest similarity between the pixel and the members of every group. Instead of using predefined groups, groups are created dynamically (figure 14).

- **Histogram-based image segmentation** uses a histogram to group pixels. The histogram can represent the frequency of gray levels in an grayscale image. Simple images consist of an object and the background. The background has usually the same color and so there will be a large peak associated to the background color on the histogram. There will be smaller peak(s) associated with color(s) of the foreground object. This way, it is possible to separate background and object by using a threshold that lies between the stronger peaks of the histogram.

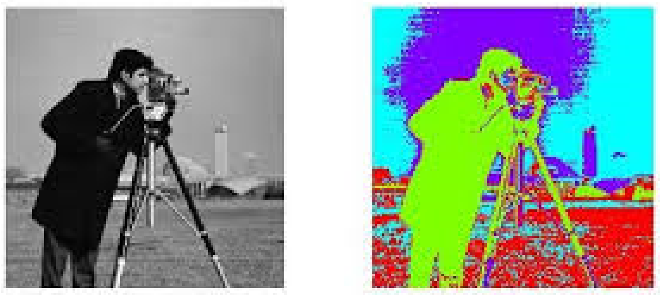


Figure 14: Segmentation with K-means clustering.

- **Edge detection** identifies abrupt changes, or discontinuities, in brightness in the image. The detected edges can be refined by connecting the discontinuity points with lines or curves. From these improved edges it is possible to delimit, i.e. segment, parts of the image.

3.10 IMAGE SEGMENTATION WITH DEEP LEARNING MODELS

Modern computer vision, based on AI and deep learning methods, has evolved dramatically in the past decade. Today, it is used in applications such as image classification, face recognition, object detection, human pose estimation, video analysis, video classification, image processing in robotics, and autonomous driving. Many computer vision tasks require intelligent segmentation of images in order to understand what is in the image and allow for an easier analysis of each part of the image. Current image segmentation techniques use DL models to understand, at an unimaginable level just a decade ago, what type of object each pixel in the image belongs to.

Deep learning can learn visual patterns in input to predict the classes of objects that make up an image. Since 2012, and until the arise of vision transformers, most deep learning models used in computer vision were based on Convolutional Neural Networks (CNNs). Some popular architectures are AlexNet, VGG, Inception, and ResNet. Deep learning models are typically trained on specialized hardware, namely graphics processing units (GPUs), to reduce the computation time. Next, we present an overview of DL architectures that have been applied in image segmentation (Minaee et al., 2020).

3.10.1 *Convolutional Neural Networks*

A convolutional neural network is a type of artificial neural network in which neurons have a delimited receptive field, analogously to what happens in neurons of the primary visual cortex of the human brain. This type of network is a variation of the Multi-Layer Perceptron (MLP). However, because they are applied to two-dimensional arrays, they are very effective in computer vision tasks, such as classification and segmentation of images.

Convolutional neural networks consist of multiple layers in which convolution filters, of one or more dimensions, are applied to the input of the layer. After each layer, a function is usually added to introduce a nonlinearity. The initial layers of the classification network carry out the feature extraction phase, usually accompanied by a reduction in the size of the feature maps. At the end of the network are commonly placed a few dense layers, to perform the final classification of the extracted features. The feature extraction phase is similar to the stimulating process in the cells of the visual cortex. This phase consists of alternating layers of convolution and feature size reduction (pooling). The output of a convolution neuron is calculated by the equation 29.

$$Y_{i,j} = g\left(b + \sum_{r=1}^{F}\sum_{s=1}^{F} K_{r,s} * X_{i-F/2+r,j-F/2+s}\right) \tag{29}$$

Where $Y_{i,j}$ is the output of the neuron (i, j), X is the input of the present layer (which is equal to the output of the previous layer), the summation carries out the convolution between a window of the input X (centered in the neuron i, j) and the filter K, b is the layer bias, and g is the nonlinear activation function. The size of the filters and the depth of the network determine its **receptive field**, which in simple terms may be defined as the parcel of the input image that contributes to the value of a given point on the output feature map.

The convolution operation has the effect of filtering the input "image" with a kernel that is trained together with the entire neural network. These kernels extract low-level features, such as vertical edges, horizontal edges, or rounded shapes.

After several convolutional layers, each one formed by a pair convolution-pooling, the model learns higher level features. The final dense layers, also called fully-connected layers, play the role of classifying the highest-level feature map in one class among the possible ones. Neurons in dense layers work in the same way as a multi-layer perceptron, and the output of each one is calculated as described by the equation 30.

$$y_j = g\left(b + \sum_i w_{i,j} \cdot x_i\right) \tag{30}$$

Where y_j is the output of neuron j, a value that is a weighted sum of all/many inputs x_i of the neuron, $w_{i,j}$ is the learned weight that multiplies the input x_i to produce the output of neuron j, b is the layer bias, and g is the non-linear activation function that depends on the type of classification task we are trying to solve.

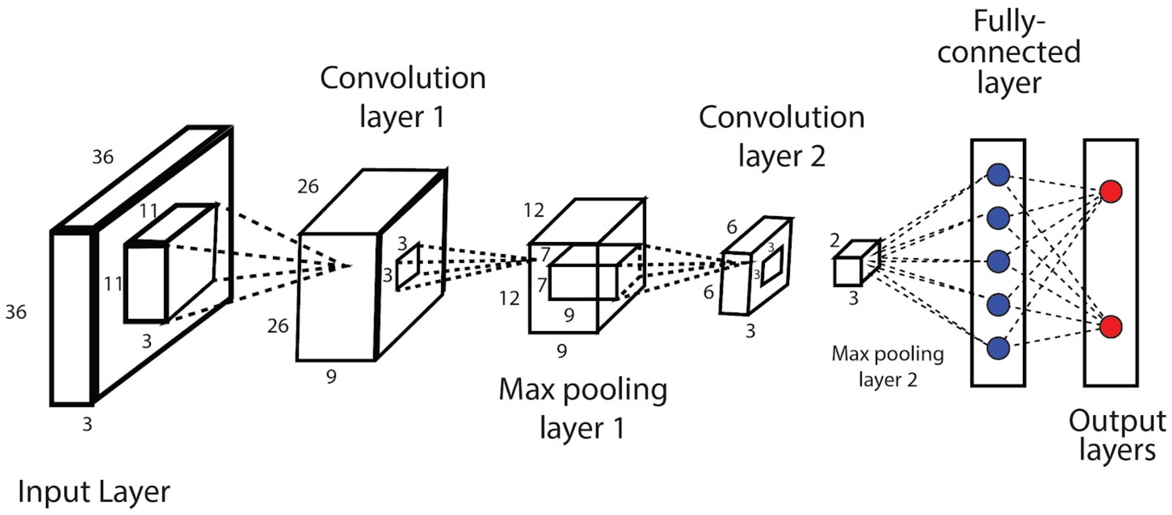


Figure 15: Convolutional neural tetwork.

3.10.2 *Fully Convolutional Networks*

Fully convolutional networks (FCNs) are used for computer vision tasks, such as semantic segmentation or super-resolution (Long et al., 2015b). One of its best properties is that they are applicable to entries of any size, for example, images of different sizes. However, when running these networks on a large-sized input, such as a high-resolution image or a video, they probably exhaust the available GPU memory. A fully convolutional network is a network composed only of convolutional layers (figure 16). In several image processing tasks, it is desired that input and output images have the same size. This can be achieved with FCNs using the appropriate padding to extend the images and recover the size that would be lost due to the edge effects associated with applying convolutions (Zuckerman, 2019). This type of architecture is useful in image processing tasks where it is important to maintain the size of the input and output images.

3.10.3 *DeepLab*

DeepLab is a state-of-the-art semantic segmentation model proposed by Google (Chen et al., 2017). The dense prediction is achieved by simply up-sampling the output of the last convolutional layer and computing a pixel-wise loss. DeepLab uses atrous convolutions on up-sampling. The repeated combination of max-pooling and striding, at consecutive layers in Deep Convolutional Neural Networks (DCNNs), significantly reduces the spatial resolution of the resulting feature maps. One solution to revert this situation is to use atrous convolutions layers to up-sample the resulting feature map. However, these layers implicate more memory and time to train the model. Atrous convolutions offer a simple yet powerful

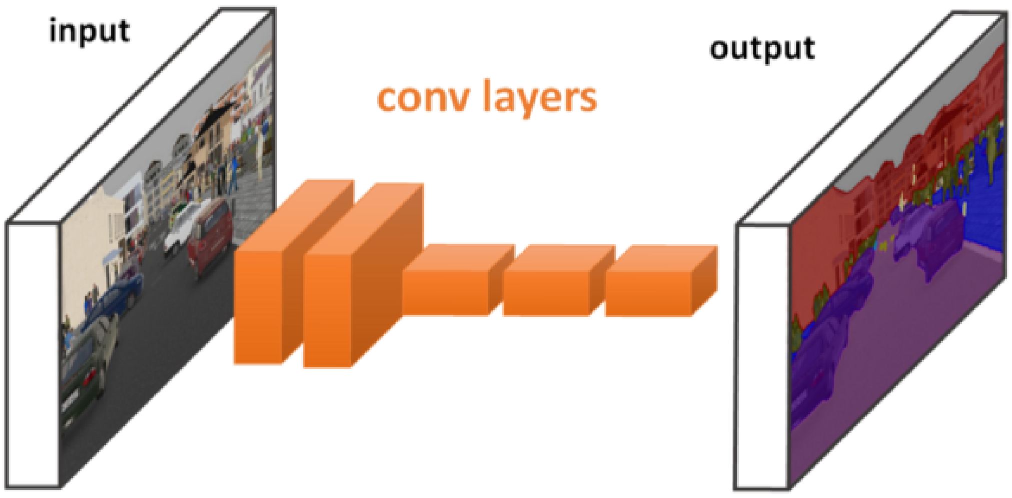


Figure 16: Fully Convolution Networks (FCNs).

alternative to regular convolutions. Atrous convolutions allow us to effectively enlarge the field of view of the filters without increasing the number of parameters or the training time (figure 17). An atrous convolution is equivalent to applying a regular convolution with a kernel that has holes ('trous' in French) between non-zero kernel values (Sahu, 2019). Mathematically, the output y_i of an atrous convolution between a one-dimensional signal x_i and a filter w of size K and stride rate r, is defined by equation 31.

$$y_i = \sum_{k=1}^{K} x_{i+r.k} * w_k \tag{31}$$

3.10.4 *SegNet Neural Network*

The SegNet neural network is a convolutional neural network developed for semantic pixel-level segmentation (Badrinarayanan et al., 2017). The architecture consists of an encoder, the corresponding decoder, and a softmax classification layer. The role of the decoder is to generate a low-resolution feature map suitable for pixel-level classification.

The encoder adopts the same topology as the 13 convolutional layers of the VGG16 model. Both encoder and decoder are organized in a sequence of five blocks, each with similar structure. Each decoder block uses the pooling indices, generated by the max-pooling layer of the corresponding encoder block, to perform a non-linear upsampling. In this way, it is not necessary to learn parameters for the upsampling layers. The upsampled feature maps have low resolution and so they are convolved with trainable filters to generate denser feature maps (figure 18).

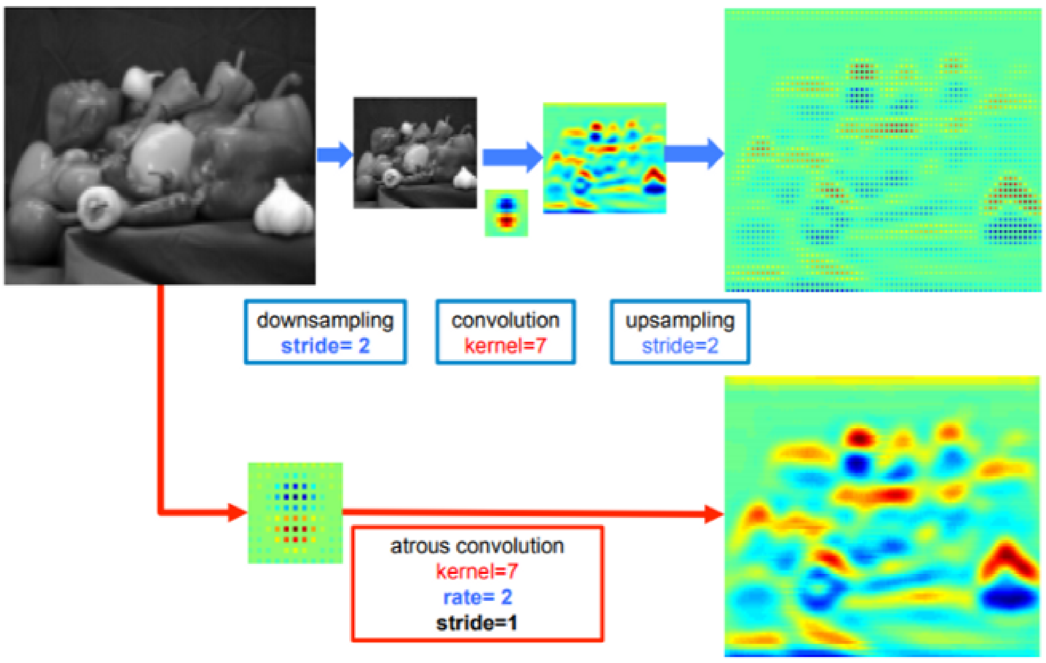


Figure 17: Regular convolution followed by up-sampling (top); atrous convolution generates a denser feature map (bottom) (Chen et al., 2017).

Each block of the encoder includes convolutional, batch normalization, ReLU activation, and max pooling layers. The decoder blocks include upsampling, convolutional, and batch normalization layers. The softmax classification layer outputs an image with K channels, which contain probabilities, and where K is the number of considered classes. For each pixel, the predicted class corresponds to the channel that contains the highest probability.

SegNet was primarily designed to carry out scene understanding; hence, it is efficient both in terms of memory and computational time during inference. It also has significantly fewer parameters than other segmentation models.

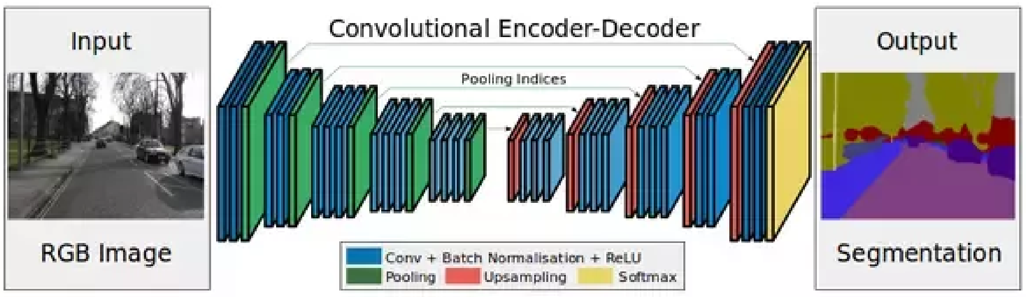


Figure 18: A deep convolution encoder-decoder architecture for image segmentation.

3.10.5 *Tiramisu*

A significant number of semantic segmentation models are based on CNNs and adopt an architecture composed of (i) a downsampling path (encoder) responsible for extracting low resolution feature maps, (ii) a upsampling path (decoder) that increases the feature maps resolution and, optionally, (iii) a post-processor to refine the predictions. Densely Connected Convolutional Networks, or simply DenseNets, achieved excellent results on image classification. The novelty introduced by DenseNets is connecting each layer directly to every other layer, following a feed-forward topology. Thus the model is more accurate and easier to train. The Tiramisu model extends DenseNets to address the semantic segmentation problem (Jégou et al., 2017). There are several advantages of using DenseNet over ResNet: (i) DenseNets are more efficient in terms of parameter usage, (ii) DenseNets perform deep supervision due to short paths to all feature maps in the architecture, and (iii) layers can reuse information from feature maps computed in preceding layers.

The Tiramisu architecture is built from dense blocks, transition down blocks, transition up blocks, convolutional layers, skip connections and concatenations (figure 19).

The **downsampling path** includes five replicas of the following topology: a dense block, a concatenation of the dense block output with the dense block input (direct connection), and a transition down block. As shown in right part of figure 19 a dense blocks is composed of four "layers" and feed-forward direct connections, similar to ResNet. Each "layer" consists on a batch normalization layer, a ReLU activation function, a convolution layer using a 3×3 kernel, and dropout. A transition down block includes a batch normalization layer, a ReLU activation function, a convolution layer using a 1×1 kernel, dropout, and a max pooling layer.

The **upsampling path** replicates the following topology five times: a transition up block, a concatenation of the transition up block output with the skip connection coming from the correct point on the downsampling path, and a dense block. A transition up block includes a transposed convolution with a 3×3 kernel and using stride 2. Between the end of the downsampling path and the beginning of the upsampling path there is an additional dense block.

3.10.6 *Feature Pyramid Network*

Feature Pyramid Network (FPN) is a type of deep learning architecture used in computer vision tasks, such as object detection and instance segmentation (Lin et al., 2016). It was designed to address the problem of preserving both fine and coarse features of an image, which is critical to accurately detect and segment objects. The main idea behind FPN is to construct a pyramid of features, where each layer of the pyramid corresponds to a different

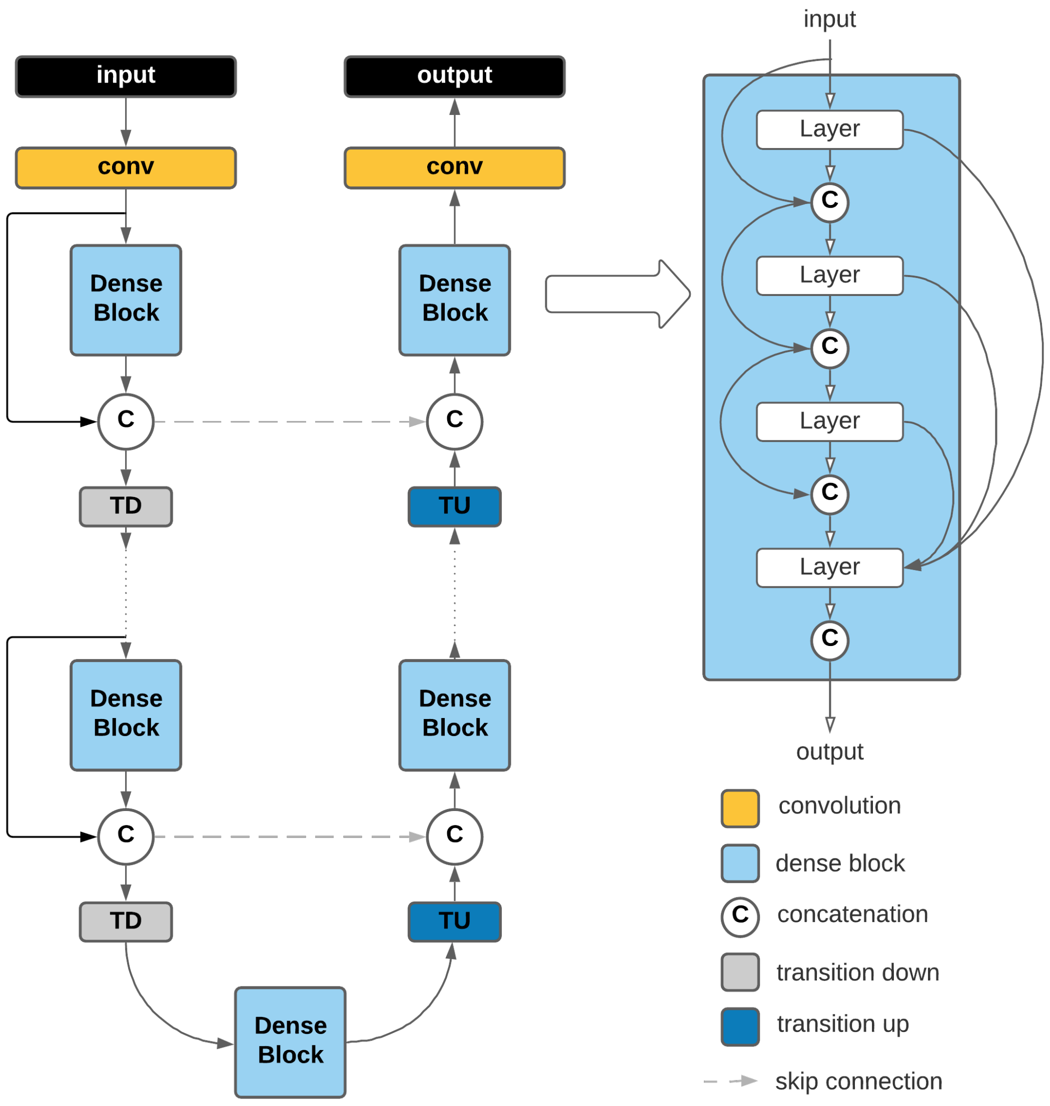


Figure 19: Tiramisu semantic segmentation model.

scale. The lower layers capture coarse information about the image, while the higher layers capture finer details. These features are combined and processed to produce accurate object detection and segmentation results.

3.10.7 *Mask R-CNN*

Mask R-CNN is a deep learning architecture thought for computer vision tasks, such as instance segmentation (He et al., 2017). It extends the popular two-stage object detection architecture, called Faster R-CNN, by adding a third branch that predicts the object mask, in parallel with the bounding box and class prediction. The Mask R-CNN architecture consists of a backbone network for feature extraction, a region proposal network for proposing candidate object regions, and two heads for classification and mask prediction. This architecture is able to perform instance segmentation, which involves both object detection and semantic segmentation of individual objects within an image.

4 PROPOSED METHODOLOGY

Today, we use technology daily, with computers, tablets, or mobile phones, and it is difficult to understand that only a few years ago this was not the reality.

4.1 RELATED WORK

In the late 1990s and early 2000s, researchers had little knowledge of the inner workings of CNNs and were considered black boxes. Complex and heavy architectures made training CNNs difficult. In early 2000, it was widely assumed that the backpropagation algorithm, used to train CNNs, was not effective in converging to the global minimum of the loss function. Therefore, CNNs were considered to be less effective feature extractors compared to traditional methods of feature engineering. However, with the technological advances that occurred at the level of datasets and libraries used to implement neural networks, the development of CNNs finally has the potential to evolve in a remarkable way. The significant improvement in performance achieved by CNNs is considered to have occurred between 2012 and 2019.

The learning capacity of deep CNNs is essentially due to the fact that they use several layers that can automatically extract representative features from the training data. Over the last decade, several innovative ideas have been introduced in CNNs, such as exploring different activation and loss functions, performing hyperparameter optimization, applying regularization techniques, and including architectural innovations in the network topology. However, it was the architectural innovations that brought the greatest gains to the performance of DCNNs. Some of these ideas include (i) exploring the spatial component and depth of the information, (ii) varying the depth and width of the layers, (iii) using topologies with multiple paths, (iv) using residual modules to reduce the problem of vanishing gradients in deep networks, or using the attention mechanism (Khan et al., 2020).

Several CNN architectures have been proposed in recent decades, such as LeNet, AlexNet, ZfNet, VGG, U-NET, GoogleNet, ResNet, Inception, Inception-ResNet, DenseNet, Wide ResNet, ResNeXt, PyramidNet or SE Net (figure 20).

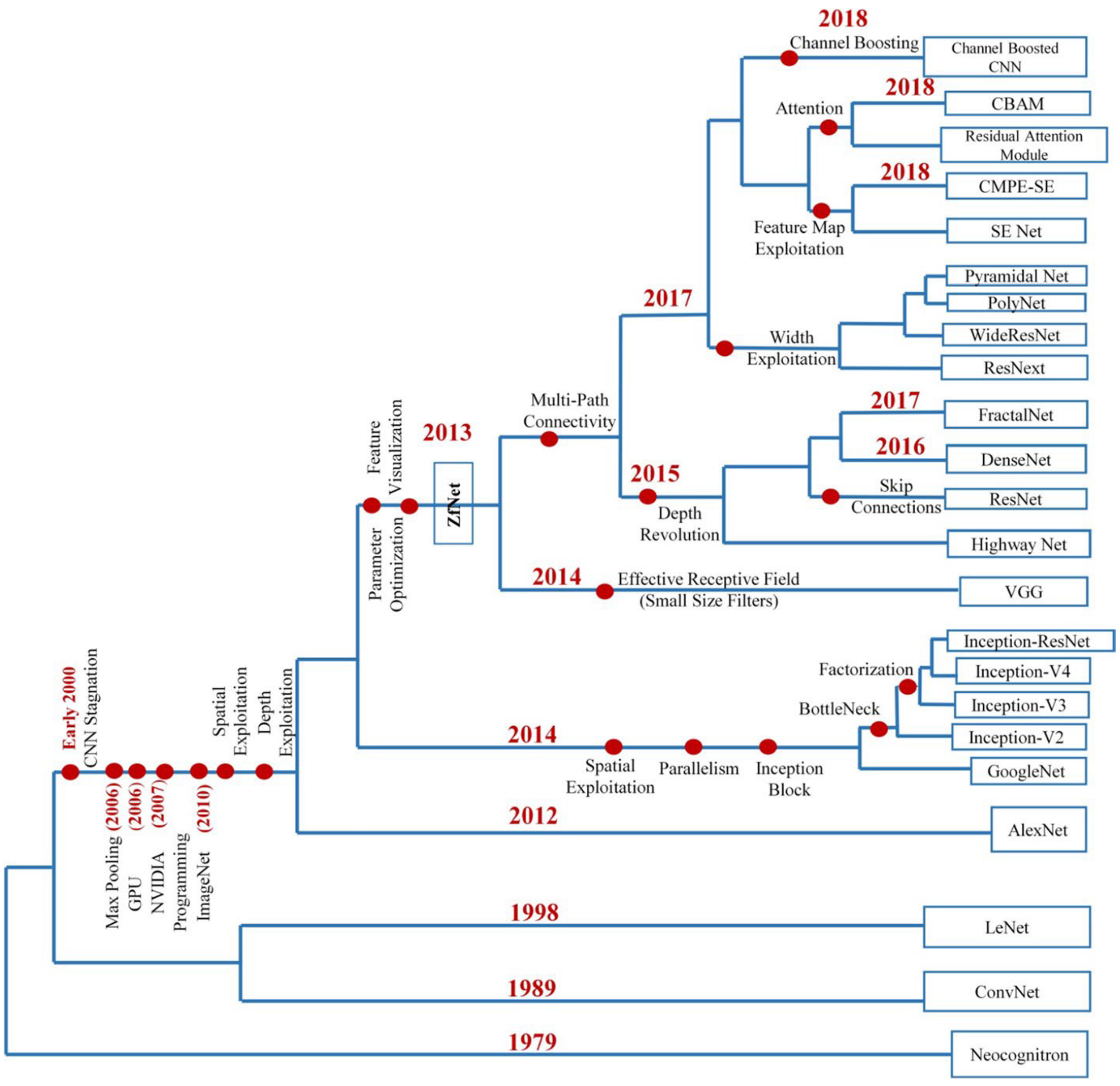


Figure 20: Milestones in CNN evolution.

The LeNet model was proposed by Yann LeCun in 1998 (LeCun et al., 1995). Its historical importance comes from being the first CNN to show top performance in the digit recognition task. It has the ability to classify the digits without being affected by small distortions, rotations, variations in position, and scale. LeNet is a feedforward neural network consisting of five pairs of convolution-pooling layers. Semantic segmentation makes predictions about the object category each pixel belongs to, providing a comprehensive description of the scene that includes the object category, its location, and shape information (figure 21). State-of-the-art semantic segmentation approaches are typically based on the fully convolutional networks (FCNs) structure (Long et al., 2015a). The adaptation of deep convolutional neural networks benefits from the rich object information scene categories and semantics learned from several sets of images (Zhang et al., 2018).

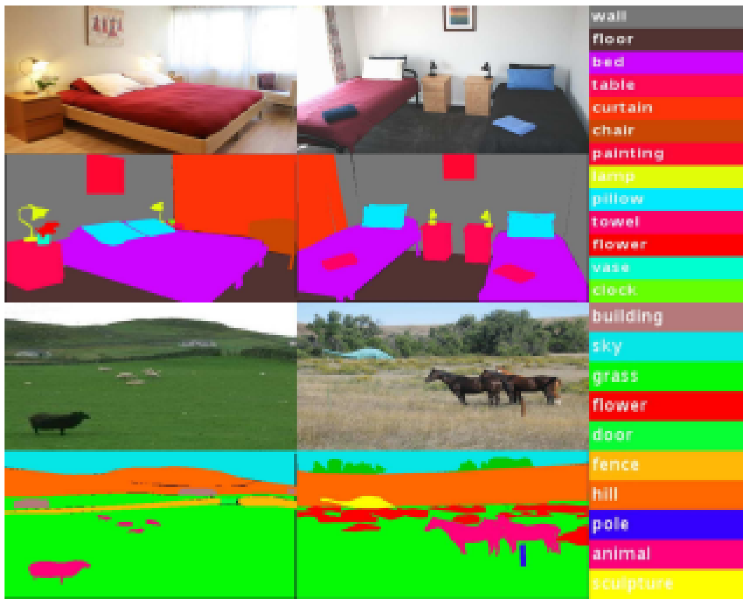


Figure 21: Labeling a scene at the pixel-level is a challenge for semantic segmentation models.

As CNN evolution progressed, Simonyan and Zisserman (2015) proposed a simple and efficient architecture called VGG. VGG is commonly used in two configurations with 16 or 19 layers. Compared to AlexNet and ZfNet, VGG has more layers and thus it has a higher representational capacity. VGG replaced (5×5) filters with more (3×3) filters and experimentally demonstrated that the use of smaller (3×3) filters can achieve the benefits of larger ones, such as (5×5) and (7×7). The use of smaller filters provides an additional advantage of lower computational effort due to the reduction in the number of parameters (Khan et al., 2020).

The GoogleNet model was the winner of the ILSVRC 2014 competition and is also known as Inception-V1. The main purpose of the authors was to increase accuracy with reduced computational cost (Szegedy et al., 2015). GoogleNet introduced the new concept of an initial block in the CNN, which incorporates multiscale convolution transformations using the idea of division, transformation, and merging. On GoogleNet, the conventional convolution layers are replaced by small blocks, similar to the idea of replacing each layer by a micro neural network as proposed in the Network in Network (NiN) architecture (Lin and and. S Yan, 2013). These blocks encapsulate parallel paths using filters with different sizes, (1×1), (3×3), and (5×5), to capture spatial information at different scales. Exploring the idea of division, transformation and fusion in GoogleNet helped to solve a problem related to learning variations in the same image that result from different resolutions or scales (Khan et al., 2020).

To increase the learning capacity of ResNet, Han et al. (2017) proposed the PyramidNet model. Instead of increasing the number of feature maps (or channels) very sharply when moving from one downsampling layer to the next, as occurs in ResNet, they opted for a more gradual increase in the number of channels, but involving as many layers as possible. This architect improved the generalization capability of the model. They also proposed a residual unit to improve the accuracy of classification. Due to a gradual increase in the depth of the feature maps, as we progress on the network, it was named a pyramidal network. In the pyramid lattice, the depth of the feature maps is regulated by the l-factor, which is calculated by equation 32 (Khan et al., 2020).

$$d_l = \begin{cases} 16 & , \text{if } l = 1 \\ [d_{l-1} + \frac{\lambda}{n}] & , \text{if } 2 \leq l \leq n+1 \end{cases} \tag{32}$$

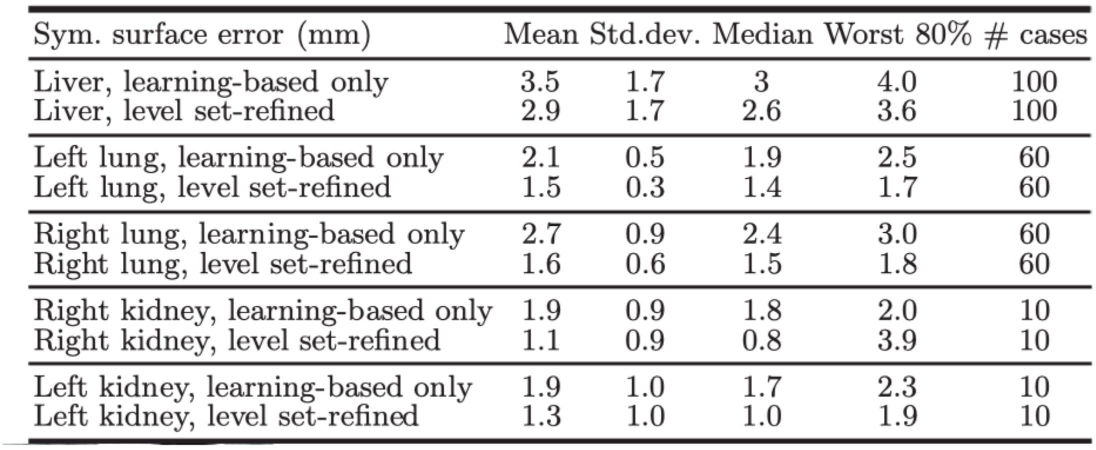
The authors Ronneberger et al. (2015) proposed a deep learning architecture, called U-Net, which is specifically designed for biomedical image segmentation, which is a critical task in medical imaging applications. The U-Net model consists of two parts: a contracting pathway to capture context and a symmetric expanding pathway to enable precise localization, along with skip connections that allow the network to use low-level features for accurate segmentation. They evaluated the U-Net architecture on two widely used medical imaging datasets, the ISBI cell tracking challenge and the EM segmentation challenge. The results demonstrated that the U-Net architecture outperformed other methods and achieved state-of-the-art performance on both datasets, highlighting its effectiveness for a wide range of biomedical image segmentation tasks, including segmenting cell structures and electron microscopy images. The work also showed that skip connections in the U-Net architecture were crucial for achieving accurate segmentation. Furthermore, the U-Net model was found to be easily adaptable to different imaging modalities and segmentation tasks, making it a valuable tool for medical imaging applications with small training datasets. Overall, the U-Net architecture presented in this study has significant potential for improving the accuracy and efficiency of biomedical image segmentation, which can have a positive impact on various medical applications.

The study by Long et al. (2012) proposes a new approach for semantic image segmentation using fully convolutional neural networks FCNs. The authors demonstrate that by replacing the fully connected layers in a traditional convolutional neural network with convolutional layers, the network can be trained to predict a segmentation map for each input image. The proposed FCNs method achieved state-of-the-art performance on the PASCAL VOC 2012 benchmark dataset, outperforming previous approaches that relied on hand-designed features and post-processing (table 1). The FCNs method also showed promising results on other datasets, including the SIFT Flow dataset and the NYUDv2 dataset. The authors further analyzed the FCNs approach and showed that it can be used for various semantic segmentation tasks, including object detection, image retrieval, and scene parsing. They also propose an end-to-end learning approach that can incorporate different sources of supervision, such as class labels and bounding boxes, to improve the accuracy of semantic segmentation. Overall, the study demonstrates the effectiveness of fully convolutional neural networks for semantic image segmentation and opens up possibilities for future research in this area. The output result of this study is a fully convolutional neural network architecture that can perform pixel-wise segmentation of images, which has applications in fields such as object detection, autonomous driving, and medical imaging.

	FCN-AlexNet	FCN-VGG16	FCN-GoogLeNet[4]
mean IU	39.8	**56.0**	42.5
forward time	50 ms	210 ms	59 ms
conv. layers	8	16	22
parameters	57M	134M	6M
rf size	355	404	907
max stride	32	32	32

Table 1: Results of different architecture evaluated by Long et al. (2012).

The work by Kohlberger et al. (2011a) presents a method for automatic segmentation of multiple organs in medical images. The proposed method combines a learning-based segmentation approach with level set optimization to achieve accurate and efficient segmentation. The authors explain the importance of accurate organ segmentation in medical imaging and the challenges associated with manual segmentation. They described the proposed method, which consists of two main stages: a learning-based segmentation and a level set optimization. In the first stage, the authors use a deep convolutional neural network to learn the appearance and shape of each organ. The CNN is trained on a large dataset of annotated medical images, and is able to accurately segment the organs in new images based on their appearance and context. In the second stage, the authors use a level set optimization to refine the initial segmentation obtained with the CNN. Level set optimization is a well-established method for segmentation that uses a mathematical model to evolve the segmentation boundary over time. The authors used a modified version of the level set method that incorporates shape priors learned by the CNN, as well as information from neighboring organs, to improve the accuracy of the segmentation (table 2). The authors evaluate their method on two different datasets of CT and MRI images, and compare it to several state-of-the-art segmentation methods. They show that their method achieves high accuracy and efficiency, and outperforms other methods in terms of both segmentation quality and computation time. In conclusion, the authors present a novel method for automatic multi-organ segmentation in medical images that combines a learning-based approach with a level set optimization. Their method achieves high accuracy and efficiency, and has the potential to improve clinical workflows and patient outcomes in a variety of medical applications.



Sym. surface error (mm)	Mean	Std.dev.	Median	Worst 80%	# cases
Liver, learning-based only	3.5	1.7	3	4.0	100
Liver, level set-refined	2.9	1.7	2.6	3.6	100
Left lung, learning-based only	2.1	0.5	1.9	2.5	60
Left lung, level set-refined	1.5	0.3	1.4	1.7	60
Right lung, learning-based only	2.7	0.9	2.4	3.0	60
Right lung, level set-refined	1.6	0.6	1.5	1.8	60
Right kidney, learning-based only	1.9	0.9	1.8	2.0	10
Right kidney, level set-refined	1.1	0.9	0.8	3.9	10
Left kidney, learning-based only	1.9	1.0	1.7	2.3	10
Left kidney, level set-refined	1.3	1.0	1.0	1.9	10

Table 2: Symmetric surface errors obtained with learning-based segmentation and after applying a level set-based refinement (Kohlberger et al., 2011a).

Sobhaninia et al. (2017) explored various perspectives of brain MRI imaging and applied distinct networks to segmentation. This work evaluated the impact of utilizing separate networks for MRI segmentation by comparing the outcomes to those achieved with a single network. Experimental assessments of the networks indicated that a Dice score of 0.73 was obtained using a single network, while a Dice score of 0.79 was obtained using multiple networks (table 3). These findings suggest that the use of multiple networks may improve the precision of brain tumor segmentation compared to using a single network. In general, this study highlights the importance of carefully selecting appropriate segmentation networks and considering multiple-view MRIs to achieve optimal segmentation results.

Method	**Data**	**Dice**
Single LinkNet for all angles	All angles	0.73
Separately trained Linknet networks for each angle	Coronal view	0.78
Separately trained Linknet networks for each angle	Sagittal view	0.79
Separately trained Linknet networks for each angle	Axial view	0.71

Table 3: Results of different approaches explored by Sobhaninia et al. (2017).

In the paper (Kohlberger et al., 2011b), the authors introduced a new segmentation method that combines learning-based and PDE optimization-based level set approaches to achieve accurate multi-organ segmentation in CT medical images. The proposed method provides several advantages, including an accurate organ boundary detection, with minimal overlap, and a high segmentation accuracy, with a 1.17-2.89mm average surface error. The authors also highlight the limitations of point cloud-based shape representations and the advantages of level set-based representations in encoding segment boundaries at a homogeneous resolution, which simplifies the detection and formulation of constraints to prevent overlaps. Furthermore, the proposed method preserves point-to-point correspondences estimated during learning-based segmentation and presents novel terms to impose region-specific geometric constraints between adjacent boundaries. The paper describes an energy-based minimization approach for multiple organs as a segmentation refinement stage and shows how it improves upon detection-based results for the liver, lungs, and kidneys. In addition, the authors also highlight its practical applications. The automatic multi-organ segmentation method has the potential to facilitate a variety of clinical workflows, including radiation therapy planning, diagnosis, and treatment monitoring. The key contributions include the combination of learning-based and level set approaches, using novel constraints to prevent overlaps between adjacent segment boundaries, reaching a high accuracy, and an efficient method to segment multiple organs. Their methodology significantly enhance the clinical utility of CT imaging by providing reliable and automated segmentation results, thereby enabling more effective diagnosis and treatment planning for various medical conditions. Overall, the paper represents a significant step forward in the field of medical image analysis and segmentation, with potential applications in a wide range of clinical scenarios.

4.2 PROPOSED METHODOLOGY

To solve the image segmentation problem two deep learning models will be used: U-Net and Tiramisu. The proposed methodology outlines the steps for training and evaluating these models on a dataset of images, and provides a plan for comparing their semantic segmentation performance, in terms of IoU and efficiency. The presentation of the methodology covers the following topics:

- Present the dataset that will be used during the project development.
- Preprocessing the dataset, including splitting it into training, validation, and evaluation subsets.
- Describe the models that were selected to perform medical image segmentation.
- Present the implementation of the selected models with the Keras library.

4.3 DATASET

The LGG Segmentation Dataset contains 3929 brain MRI scans, 3929 manual Fluid-Attenuated Inversion Recovery (FLAIR) abnormality segmentation masks, and occupies more than 10G bytes. The resolution of the images is 256×256 pixels. The dataset was obtained from The Cancer Imaging Archive (TCIA) and The Cancer Genome Atlas (TCGA), was recorded in 110 patient exams, 5 institutions, and with Lower Grade Gliomas (LGGs). A lower grade glioma is a type of tumor that develops in the brain and tends to grow slowly. The number of scans with tumor is 1373 and without tumor is 2556.

FLAIR is an advanced MRI sequence that reveals T2 tissues (fat and water), while suppressing cerebrospinal fluid, allowing the detection of superficial brain lesions. T2 MRI images are captured using radiofrequency pulse sequences with a timing that highlights fat and water within the body. The purpose of using FLAIR is to detect lesions that are usually missed by MRI (Bakshi et al., 2001).

The ground truth for training the segmentation models is the manual segmentation masks. The MRI images were obtained from the patients, but the ground truth masks were manually annotated by a medical student with experience in neuroradiology using drawing software. Later, these masks were validated and corrected by a radiologist. Figure 22 shows four samples of the dataset, where images are on the top and the masks are on the bottom.

4.4 DATA PREPROCESSING

Image preprocessing involves transforming raw images into an understandable and useful format. This step must be done before the images are applied in model training. It includes, but is not limited to, resizing, orienting, and applying color correction.

Image resizing is a manipulation applied to an image to create an image with different size but the same content. If our dataset has images with different sizes, we must resize

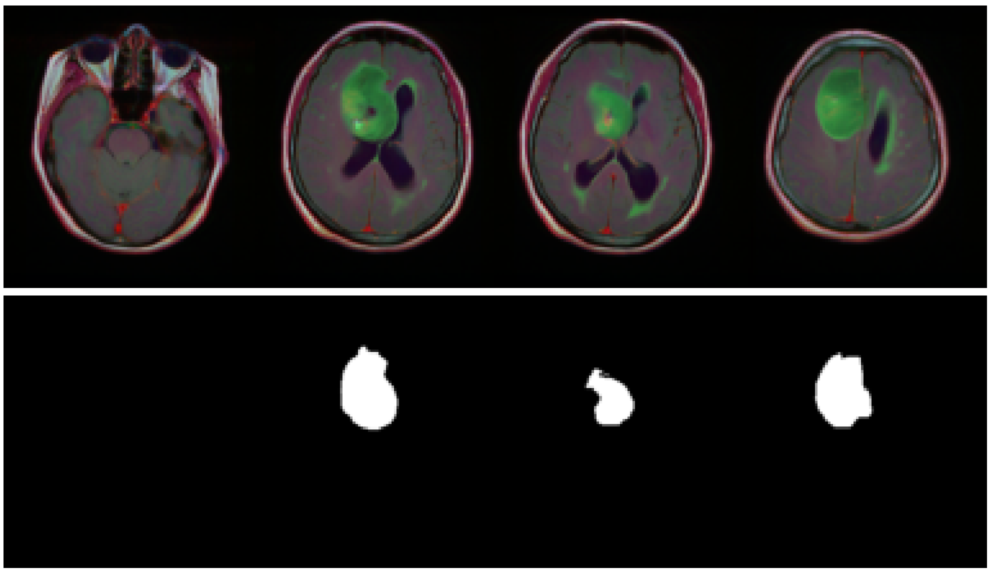


Figure 22: Samples of the LGG segmentation dataset.

them to get a uniform dataset. In our case, we resized all the images so that they have 128 × 128 pixels. In order to minimize computational cost, we changed the images to NPY format, which helps us in the training process. The fragment of Python code included in listing 4.1 illustrates the image resizing task.

NPY is a simple format for saving NumPy arrays to files, including full information about the arrays. NPY is the standard binary format in NumPy to store a single array on disk. Information such as the shape and and type of the array, which is necessary to reconstruct the array correctly even on another machine possibly with a different architecture, is also stored on the NPY file [1].

```
from PIL import Image
image_dataset = []
mask_dataset  = []
for i in tqdm(range(len(df))):
    img_path = train_files[i]
    image = cv2.imread(img_path,0)
    image = image.fromarray(image)
    image = image.resize((SIZE,SIZE))
    image_dataset.append(np.array(image))
    mask_path = mask_files[i]
    image = cv2.imread(msk_path,0)
    image = Image.fromarray(image)
    image = image.resize((SIZE,SIZE))
    mask_dataset.append(np.array(image))
```

Listing 4.1: Image resizing code.

1 https://numpy.org/devdocs/reference/generated/numpy.lib.format.html

When we read an image into memory, the pixels are usually encoded with an 8-bit integer, between 0 and 255, for each color channel. However, regression models, including neural networks, prefer an input that is encoded as floating point values within a smaller range. Often, we want the values to have mean 0 and standard deviation 1 as the standard normal distribution 28. This process is called **image normalization** and its implementation is illustrated in listing 4.2.

```
from tensorflow.keras.utils import normalize
image_dataset = np.expand_dims(normalize(np.array(image_dataset), axis=1),3)
mask_dataset = np.expand_dims((np.array(mask_dataset)),3)/255.
```

Listing 4.2: Image normalization code.

The fundamental goal of an ML model is to make accurate predictions on data not included in the training dataset. Before using a model to make predictions, we need to evaluate the predictive performance of the model. To assess the quality of predictions of an ML model when faced with data it has not seen, we can leave aside part of the dataset. The standard procedure for achieving this goal is to split the entire dataset into two or three parts: the training set, the validation set, and the test set (figure 23). In our case, we split the dataset into two parts, 80% for training and 20% for testing, selected randomly. The code included in listing 4.3 implements the splitting of the dataset.

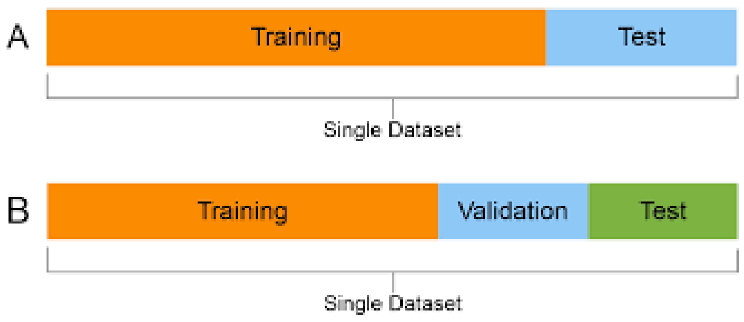


Figure 23: Splitting the dataset.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
    image_dataset,
    mask_dataset,
    test_size = 0.2,
    random_state = 0
    )
```

Listing 4.3: Code to split the dataset into training and test sets.

During the training phase, the DL models are fed a mini-batch of samples and respective masks at a time. Batches are randomly sampled from the dataset via python generators.

4.5 SELECTING MODELS FOR IMAGE SEGMENTATION

In this section, we present the details of the DL models that were selected to carry out semantic segmentation of the medical images, U-Net and Tiramisu network, describing some layers of the networks and aspects that are distinct in the selected architectures. We will also describe the training and inference processes.

Figure 24 gives a simplified view of the U-Net model during training. On the left, a (batch of) preprocessed image feeds the network. The image is forward propagated through the network layers and a mask is output (top image on the right). This mask is compared with the ground truth mask and by using a loss function an error is calculated. During the backpropagation step, we calculate the gradient of the loss function in relation to each parameter of the model, which are the weights and biases of the layers, and the error (or gradient) is used to adjust the respective parameter value.

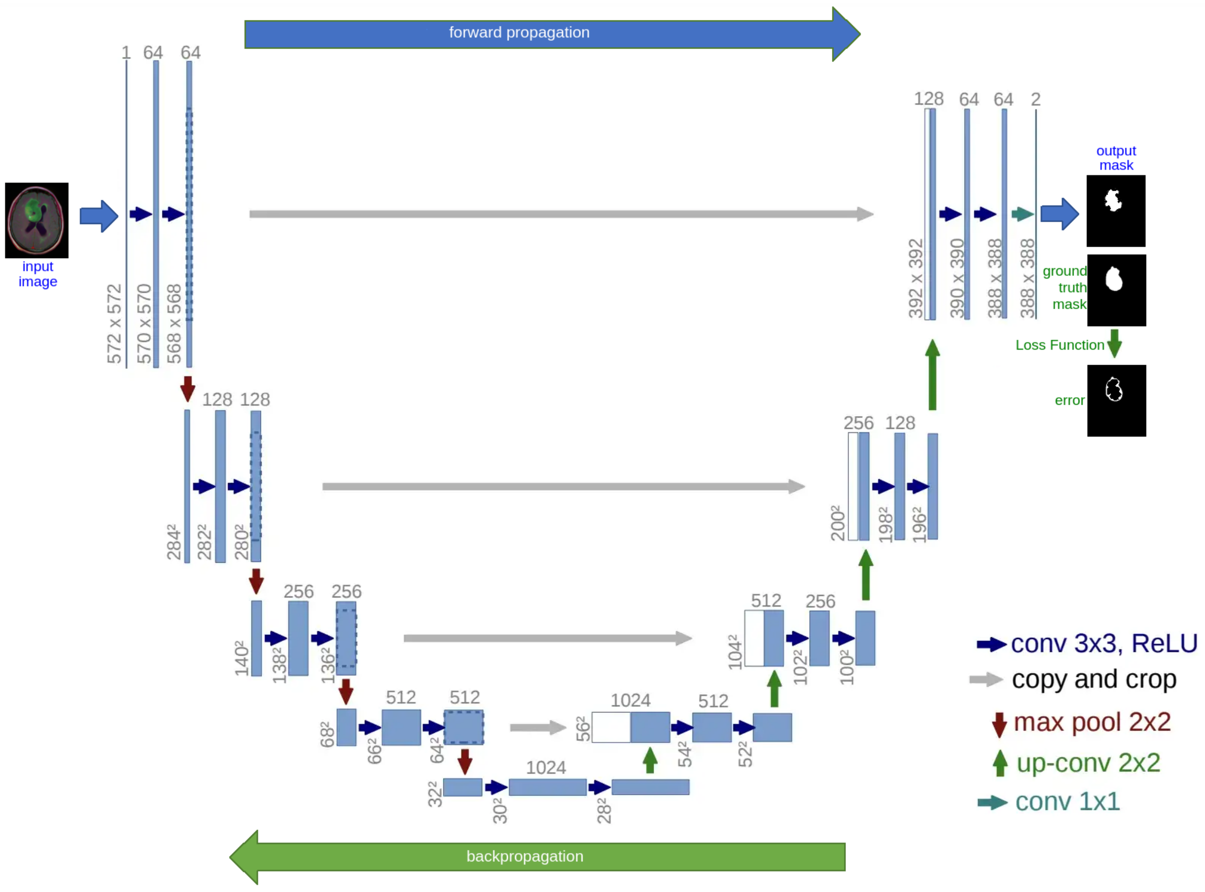


Figure 24: U-Net during training.

Figure 25 gives a simplified view of the trained U-Net performing inference. It begins with an input image from the test set feeding the network (on the left), the image is forward propagated through the network already trained, and the model outputs a segmentation mask that distinguishes tumor from non-tumor pixels (on the right).

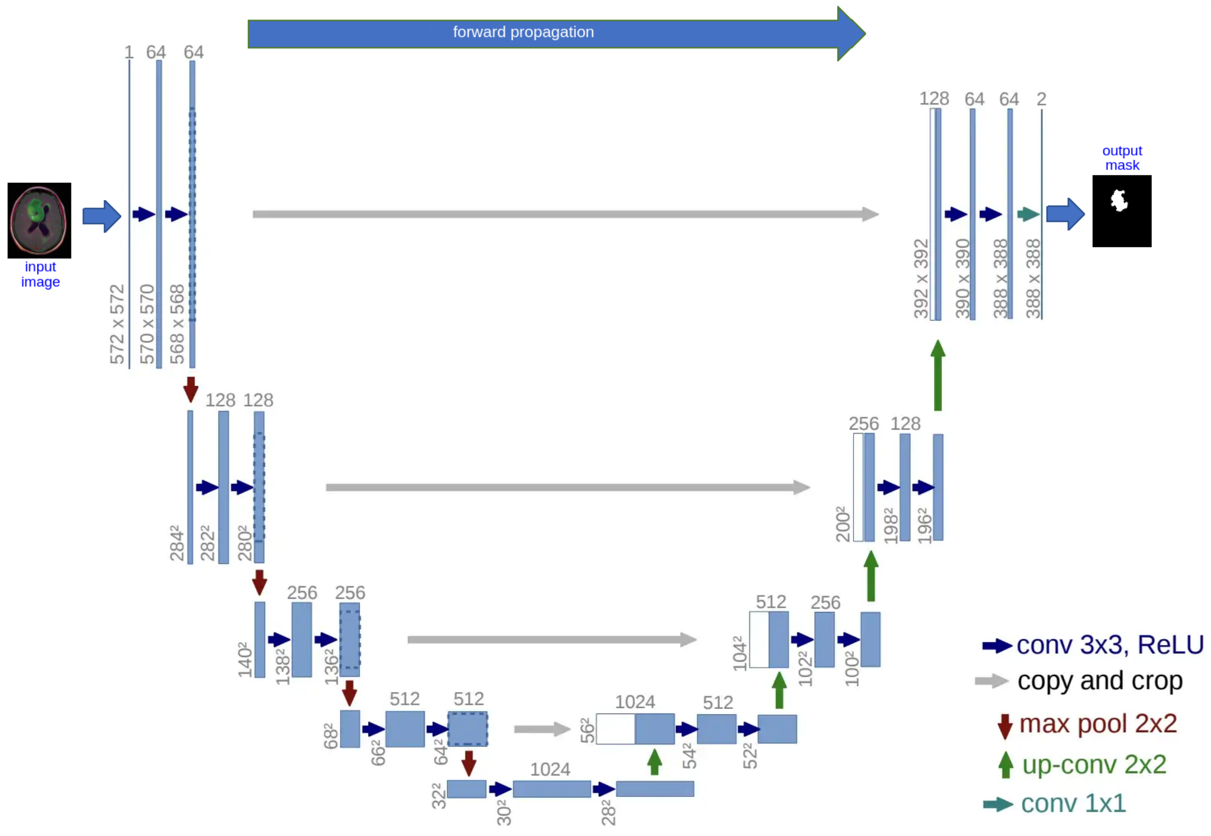


Figure 25: U-Net during inference.

4.5.1 *U-Net Model*

The U-Net architecture presented in figures 24 and 25 consists of a contracting path that captures context and a symmetric expanding path that enables precise localization. This simple encoder-decoder architecture has become very popular and has been adapted to deal with a variety of segmentation problems. In the standard U-Net model, the contracting path consists of a series of four blocks, each containing two convolution-ReLU layers and one max pooling layer. The expanding path includes four blocks, each containing a transposed convolution layer and two convolution-ReLU layers. Between the contracting and expanding paths there is a bottleneck block that has 2 convolution-ReLU layers. In addition, the network has a skip connection from each encoder block to the corespondent decoder block. These connections inject higher resolution information in later stages of the network.

As explained by Jordan (2018b), there are a number of more advanced blocks that can be stacked to build a U-Net network. Drozdzal et al. (2016) replaced the basic blocks, built with stacked convolutions, with residual blocks. The residual block introduces short skip connections, within the block, alongside the existing long skip connections, between the corresponding encoder and decoder blocks, found in the standard U-Net structure. They report that short skip connections allow for faster convergence when training and allow deeper models to be trained.

4.5.2 *Tiramisu Model*

By expanding the architecture of U-Net, Jégou et al. (2016) proposed using dense blocks, but still following a decoder-encoder network topology, arguing that the characteristics of DenseNets make them a very good fit for semantic segmentation, as they naturally induce skip connections and multiscale supervision. These dense blocks are useful because they carry low-level features from previous layers directly alongside higher-level features from later layers, allowing for highly efficient feature reuse. The proposed model is called Tiramisu and is depicted in figure 26.

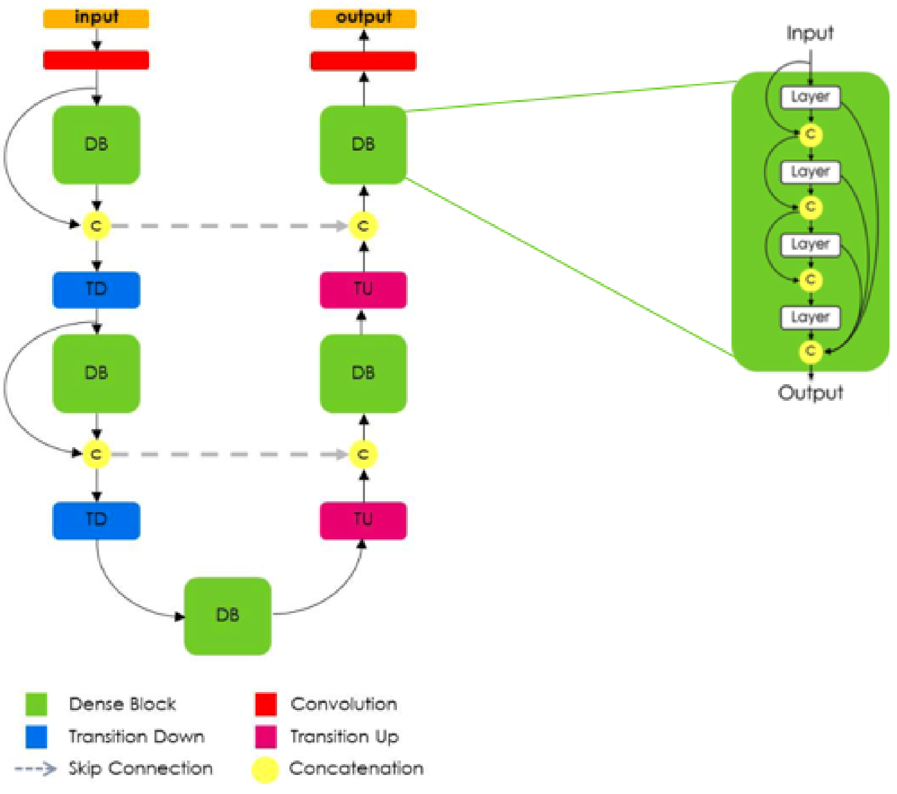
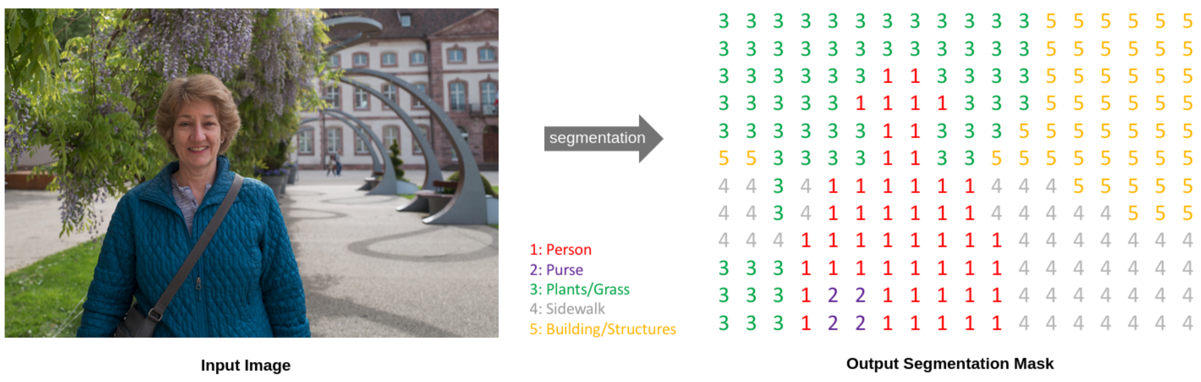


Figure 26: Tiramisu fully convolutional neural network.

A very important aspect of the Tiramisu architecture is the fact that the upsampling path does not have a skip connection between the input and output of a dense block. The authors note that because the upsampling path increases the feature maps spatial resolution, the linear growth in the number of features would be too memory demanding. Thus, only the output of each encoder dense block is passed directly to the correspondent decoder block.

In simple terms, the goal of segmentation is to take an RGB color image with size $H \times W \times 3$ or a grayscale image with size $H \times W \times 1$, and output a segmentation mask with size $H \times W \times 1$ and where each pixel is assigned a class label represented by an integer value. Figure 27 illustrates the segmentation of a scene in 5 classes. To simplify the visualization, the segmentation mask has a lower resolution than the input scene. In reality, the segmentation mask resolution should match the input image resolution. Instead of generating a single mask in which each pixel is assigned a value that identifies one of multiple classes, we can generate multiple binary masks, one for each class of our problem.



3	3	3	3	3	3	3	3	3	3	3	3	5	5	5	5	5	5
3	3	3	3	3	3	3	3	3	3	3	3	5	5	5	5	5	5
3	3	3	3	3	3	1	1	3	3	3	3	5	5	5	5	5	5
3	3	3	3	3	1	1	1	1	3	3	3	5	5	5	5	5	5
3	3	3	3	3	3	1	1	3	3	3	5	5	5	5	5	5	5
5	5	3	3	3	3	1	1	3	3	5	5	5	5	5	5	5	5
4	4	3	4	1	1	1	1	1	1	4	4	4	5	5	5	5	5
4	4	3	4	1	1	1	1	1	1	4	4	4	4	4	5	5	5
4	4	4	1	1	1	1	1	1	1	1	4	4	4	4	4	4	4
3	3	3	1	1	1	1	1	1	1	1	4	4	4	4	4	4	4
3	3	3	1	2	2	1	1	1	1	1	4	4	4	4	4	4	4
3	3	3	1	2	2	1	1	1	1	1	4	4	4	4	4	4	4

Figure 27: A simple example of semantic segmentation.

4.6 IMPLEMENTATION OF IMAGE SEGMENTATION MODELS

We present now a few implementation details about the model's implementation.

- The convolution layers are implemented with the `Conv2D` keras function. The input of the layer is convolved with a set of filters to produce a set of feature maps. The fundamental parameters of `Conv2D` are the size of the filters, the stride, the padding, and the activation functions. We chose to use all filters with size 3×3, the padding was set to `'same'` to maintain the size of the output equal to the input image size, and the selected activation function is ReLu.

- To reduce the computation cost and the number of parameters, when we increase the number of feature maps in successive convolution layers, the feature maps are often downsampled. A common implementation of downsampling in neural networks are the pooling layers. `MaxPooling2D` is a Keras layer that downsamples the input along its spatial dimensions, height and width, by taking the maximum value over a specified input window for each channel of the input. The window is repeatedly shifted by strides along each dimension. The main parameters of `MaxPooling2D` are the window size over which we calculate the maximum, the strides that specify how many positions the pooling window moves for in each pooling step, and the padding to apply to the input tensors of the layer.

- To reverse the effect of max pooling layers, the upsampling of feature maps is implemented with the `Conv2DTranspose` Keras layer. This layer implements a 2D transposed convolution, which is not a deconvolution. In transposed convolutions, the filters are used to learn meaningful decompression instead of compression as in regular convolutions. The transposed convolution operation is equivalent to the gradient calculation of a regular convolution, that is, it is equivalent to the backward pass of a regular convolution. A transposed convolution essentially computes the transposed matrix of a regular convolutional layer, swapping the effect of a regular convolution forward pass with its backward pass effect. The curious fact is that the weights of a transposed convolution can be learned (Dumoulin and Visin, 2018).

To conclude the presentation of the implementation of the U-Net and Tiramisu networks, we summarize the number of trainable parameters of both models in listings 4.4 and 4.5, respectively.

```
Total params: 1,941,105
Trainable params: 1,941,105
Non-trainable params: 0
```

Listing 4.4: Number of parameters in the U-Net model.

```
Total params: 13,824,657
Trainable params: 13,818,793
Non-trainable params: 5,864
```

Listing 4.5: Number of parameters in the Tiramisu model.

The architecture of U-Net reported by the Keras `Model.summary` method is detailed in listing 4.6. The main code that implements U-Net is included in appendix 7, on listings 7.8 and 7.9. Due to its long description, the summary of the Tiramisu model can be found in appendix 7, on listings 7.12 until 7.24. The main code that implements Tiramisu is included on listings 7.28 and 7.29.

```
Layer (type)                     Output Shape         Params      Connected to
input_9 (InputLayer)             [(None, 128, 128, 3) 0
lambda_8 (Lambda)                (None, 128, 128, 3)  0           input_9[0][0]
conv2d_152 (Conv2D)              (None, 128, 128, 16) 448         lambda_8[0][0]
dropout_72 (Dropout)             (None, 128, 128, 16) 0           conv2d_152[0][0]
conv2d_153 (Conv2D)              (None, 128, 128, 16) 2320        dropout_72[0][0]
max_pooling2d_32 (MaxPooling2D) (None, 64, 64, 16)   0           conv2d_153[0][0]
conv2d_154 (Conv2D)              (None, 64, 64, 32)   4640        max_pooling2d_32[0][0]
dropout_73 (Dropout)             (None, 64, 64, 32)   0           conv2d_154[0][0]
conv2d_155 (Conv2D)              (None, 64, 64, 32)   9248        dropout_73[0][0]
max_pooling2d_33 (MaxPooling2D) (None, 32, 32, 32)   0           conv2d_155[0][0]
conv2d_156 (Conv2D)              (None, 32, 32, 64)   18496       max_pooling2d_33[0][0]
dropout_74 (Dropout)             (None, 32, 32, 64)   0           conv2d_156[0][0]
conv2d_157 (Conv2D)              (None, 32, 32, 64)   36928       dropout_74[0][0]
max_pooling2d_34 (MaxPooling2D) (None, 16, 16, 64)   0           conv2d_157[0][0]
conv2d_158 (Conv2D)              (None, 16, 16, 128)  73856       max_pooling2d_34[0][0]
dropout_75 (Dropout)             (None, 16, 16, 128)  0           conv2d_158[0][0]
conv2d_159 (Conv2D)              (None, 16, 16, 128)  147584      dropout_75[0][0]
max_pooling2d_35 (MaxPooling2D) (None, 8, 8, 128)    0           conv2d_159[0][0]
conv2d_160 (Conv2D)              (None, 8, 8, 256)    295168      max_pooling2d_35[0][0]
dropout_76 (Dropout)             (None, 8, 8, 256)    0           conv2d_160[0][0]
conv2d_161 (Conv2D)              (None, 8, 8, 256)    590080      dropout_76[0][0]
conv2d_transpose_32 (Conv2DTr.) (None, 16, 16, 128)  131200      conv2d_161[0][0]
concatenate_32 (Concatenate)     (None, 16, 16, 256)  0           conv2d_transpose_32[0][0]
                                                                  conv2d_159[0][0]
conv2d_162 (Conv2D)              (None, 16, 16, 128)  295040      concatenate_32[0][0]
dropout_77 (Dropout)             (None, 16, 16, 128)  0           conv2d_162[0][0]
conv2d_163 (Conv2D)              (None, 16, 16, 128)  147584      dropout_77[0][0]
conv2d_transpose_33 (Conv2DTr.) (None, 32, 32, 64)   32832       conv2d_163[0][0]
concatenate_33 (Concatenate)     (None, 32, 32, 128)  0           conv2d_transpose_33[0][0]
                                                                  conv2d_157[0][0]
conv2d_164 (Conv2D)              (None, 32, 32, 64)   73792       concatenate_33[0][0]
dropout_78 (Dropout)             (None, 32, 32, 64)   0           conv2d_164[0][0]
conv2d_165 (Conv2D)              (None, 32, 32, 64)   36928       dropout_78[0][0]
conv2d_transpose_34 (Conv2DTr.) (None, 64, 64, 32)   8224        conv2d_165[0][0]
concatenate_34 (Concatenate)     (None, 64, 64, 64)   0           conv2d_transpose_34[0][0]
                                                                  conv2d_155[0][0]
conv2d_166 (Conv2D)              (None, 64, 64, 32)   18464       concatenate_34[0][0]
dropout_79 (Dropout)             (None, 64, 64, 32)   0           conv2d_166[0][0]
conv2d_167 (Conv2D)              (None, 64, 64, 32)   9248        dropout_79[0][0]
conv2d_transpose_35 (Conv2DTr.) (None, 128, 128, 16) 2064        conv2d_167[0][0]
concatenate_35 (Concatenate)     (None, 128, 128, 32) 0           conv2d_transpose_35[0][0]
                                                                  conv2d_153[0][0]
conv2d_168 (Conv2D)              (None, 128, 128, 16) 4624        concatenate_35[0][0]
dropout_80 (Dropout)             (None, 128, 128, 16) 0           conv2d_168[0][0]
conv2d_169 (Conv2D)              (None, 128, 128, 16) 2320        dropout_80[0][0]
conv2d_170 (Conv2D)              (None, 128, 128, 1)  17          conv2d_169[0][0]
```

Listing 4.6: Summary of the U-Net model.

5 RESULTS

After performing several experiments with U-Net and Tiramisu models, with different loss functions, this chapter reports the achieved results. We will show that U-Net allows us to get better results than Tiramisu. Several loss functions were tried with both models, including a combination of all losses, which we called total loss.

We begin the presentation of results with a summary of the experiments that were performed, including the model, the number of parameters of the model, the loss function, and the evaluation metric (table 4).

Model	Parameters	Loss Function	Metric
U-Net	1,941,105	Jaccard Binary Cross Entropy (BCE) BCE-dice Tversky Total	IoU
Tiramisu	13,824,657	Jaccard BCE BCE-dice Focal Tversky Binary focal	IoU

Table 4: Summary of the performed experiments.

In the field of statistics and artificial intelligence, it is common to work with loss functions to measure the error, or loss, of predictions. Since classification and regression are the two main machine learning tasks, there are different loss functions to evaluate the error in each of these tasks. Since segmentation is essentially a classification problem, we will use loss functions suitable for classification. The loss functions that were evaluated are Jaccard, Binary Cross-Entropy, Focal, Tversky, BCE-dice, and a combination of all the previous ones.

As described in section 3.4, there are several algorithms among which we can choose the one with which we are going to optimize the machine learning models. We selected the Adam optimizer. Adam calculates an exponentially decaying average of past squared gradients and an exponentially decaying average of past gradients, regulated by the hyperparameters β_1 and β_2.

5.1 EVALUATION METRIC

The IoU, also known as the Jaccard index or the Jaccard similarity coefficient, is one of the most widely used metrics in semantic segmentation. IoU is a very intuitive metric whose meaning is illustrated in figure 28. IoU is the area of interception between the predicted segmentation mask and the ground truth mask divided by the area of union between the predicted segmentation mask and the ground truth mask. The metric ranges from 0 to 1, or 0 to 100%, with 0 meaning no overlap and 1 meaning complete overlap between the segmentation masks.

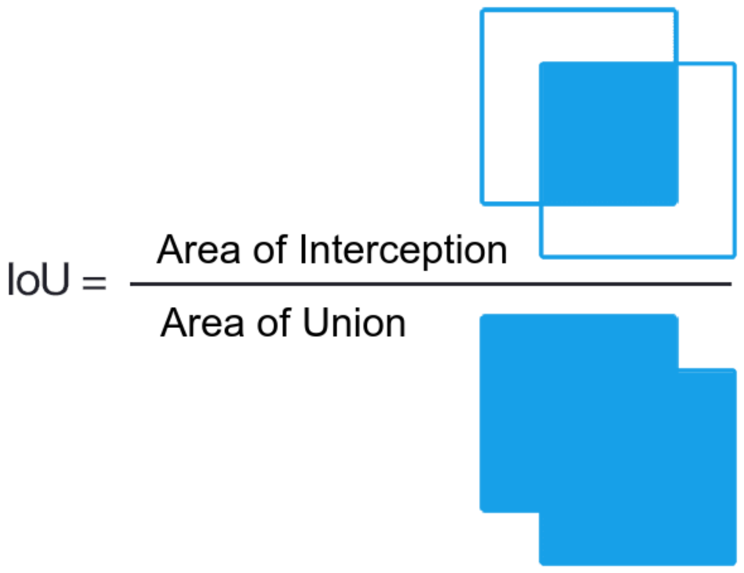


Figure 28: Meaning of the Intersection over Union metric.

In binary or multiclass image segmentation, the mean IoU of the prediction is calculated by taking the IoU of each class and averaging them. The IoU metric is better than the pixel accuracy to evaluate a segmentation model. For this reason, in the present work, the metric adopted to evaluate the performance of the models was IoU. IoU can be calculated with the method included in listing 5.1.

5.2 HYPERPARAMETER OPTIMIZATION

Hyperparameters are the parameters that allow us to configure several aspects of the models and of the training process itself. These parameters are not learned through the regular training process. They need to be optimized apart. In machine learning, hyperparameter optimization is the process of selecting the best combination of hyperparameters that deliver the best model performance. Several optimization techniques exist, such as Grid Search, Random Search, or Bayesian Search. There are also a few frameworks that implement these optimization techniques, for example Keras Tuner, Optuna, Hyperopt, Scikit-Optimize, Ray Tune, Talos, or BayesianOptimization. Finally, some integrated platforms support hyperparameter optimization, namely Weights and Biases, comet.ml, neptune.ai, or Polyaxon. Here we will give a brief overview of the explored solutions: Weights and Biases and Keras Tuner.

```
def iou(y_true, y_pred):
    y_true_f = K.flatten(y_true)
    y_pred_f = K.flatten(y_pred)
    smooth   = 0.00001
    intersection = K.sum(y_true_f * y_pred_f)
    union = K.sum(y_true_f)+K.sum(y_pred_f)-intersection
    iou_val = (intersection + smooth) / (union + smooth)
    return iou_val
```

Listing 5.1: Computing the IoU metric.

Weights and Biases

Weights and Biases (WandB) is a machine learning platform for developers to build better models faster. We can use WandB lightweight and interoperable tools to quickly track experiments, data versioning, manage models, evaluate model performance, reproduce experiments, visualize results, spot regressions, and collaborate. Weights and Biases sweeps facilitate the automation of hyperparameter optimization and to explore the space of possible solutions. The main benefits of WandB sweeps are the following:

1. Performing hyperparameter optimization only requires a few lines of code. We can launch a sweep across dozens of machines as easy as starting a sweep on our local machine.
2. All the optimization algorithms' code is open source.
3. Sweeps are completely customizable and configurable.

Keras Tuner

`KerasTuner` is an easy-to-use, scalable hyperparameter optimization framework that solves the pain points of hyperparameter search. One can easily configure the search space with a define-by-run syntax, then leverage one of the available search algorithms to find the best hyperparameter values for our models. `KerasTuner` comes with Bayesian Optimization, Hyper-band, and Random Search algorithms built-in, and it is also designed to be easily extended with new search algorithms. Listings 5.2 and 5.3 contains an example of code to search for the best learning rate value for training U-Net with `KerasTuner`.

5.3 LEARNING RATE

The learning rate (LR) is a hyperparameter that controls how much to change the weights and biases of the model, in response to the error calculated by the loss function, in each training iteration. The learning rate may be the most important hyperparameter when training deep neural networks (Brownlee, 2019). Therefore, it is vital to investigate the effect of the learning rate on model performance and to build an intuition about the effect of the learning rate dynamics on the model behavior. In this work, we evaluated the influence of the learning rate and its dynamics on the model performance. Based on experiments, we concluded that:

- Using too large learning rates results in unstable training and too small learning rates makes the training very slow.
- Selecting an optimization algorithm that uses momentum can accelerate training, and applying a learning rate schedule can help the optimization process to converge. By learning rate schedule, we mean varying the learning rate over time.
- Adaptive learning rates can accelerate training and alleviate the pressure of choosing the optimal learning rate and the optimal learning rate schedule.

```
import kerastuner as kt
from   tensorflow import keras
from   kerastuner.tuners import RandomSearch
from   kerastuner.engine.hypermodel import HyperModel
from   kerastuner.engine.hyperparameters import HyperParameter

def build_model(hp):
    model = UNet(128,128,1)
    model.compile(optimizer=keras.optimizers.Adam(
           hp.Choice('learning_rate',values=[1e-2, 1e-3, 1e-4])
       ),
       loss=[total_loss],metrics=[dsc])
    return model


```

Listing 5.2: Search the best learning rate for training U-Net with `KerasTuner` (part 1).

```
tuner = RandomSearch(build_model,kt.Objective("val_dsc",
direction="max"),max_trials=5,directory='test_dir')

tuner.search_space_summary()
'''
output: Search space summary
Default search space size: 1
learning_rate (Choice)
{'default': 0.01, 'conditions': [], 'values': [0.01, 0.001, 0.0001], 'ordered': True}
'''
tuner.search(X_train, y_train,epochs=5,validation_data=(X_test,y_test))
'''
Trial 4 Complete [00h 01m 35s]
val_dsc: 0.4713146984577179
Best val_dsc So Far: 0.7267881631851196
Total elapsed time: 00h 14m 00s
INFO:tensorflow:Oracle triggered exit
'''
tuner.results_summary()
'''
Results summary
Results in test_dir/untitled_project
Showing 10 best trials
Objective(name='val_dsc', direction='max')
Trial summary
Hyperparameters:
  learning_rate: 0.0001
Score: 0.7267881631851196
Trial summary
Hyperparameters:
  learning_rate: 0.001
  Score: 0.4713146984577179
Trial summary
Hyperparameters:
  learning_rate: 0.01
  Score: 0.018969120457776844
'''
```

Listing 5.3: Search the best learning rate for training U-Net with `KerasTuner` (part 2).

Deep learning neural networks are optimized with an algorithm based on stochastic gradient descent. SGD is an optimization algorithm that estimates the gradient of the error (or loss), for the current iteration, and then updates the weights and bias of the model using backpropagation of the gradients from the end to the beginning of the network. The amount in which the weights are updated is governed by the learning rate. The learning rate has a small positive value smaller than 1, often in the range 10^5 and 10^{-1}.

The learning rate controls how quickly the model is adapted to the problem. Smaller learning rates require more training epochs, given that smaller changes are made to the weights on each update, whereas larger learning rates result in rapid changes and require fewer training epochs. A learning rate that is too large can cause the model to converge too quickly to a suboptimal solution, while a learning rate that is too small can cause the process to get stuck.

The Keras DL library allows us to easily configure the learning rate for the different optimization algorithms based on stochastic gradient descent. Next, we present a few learning rate configurations allowed by Keras.

- Keras allows us to adjust the learning rate using a callback. The callback operates separately from the optimization algorithm, although it adjusts the learning rate used by the optimization algorithm. It is recommended to use the SGD algorithm when using a callback to adjust the learning rate. Callbacks are configured and instantiated, and then all are included in a list of callbacks passed as an argument to the `fit` method that trains the model.

  Keras provides the `ReduceLROnPlateau` method to adjust the learning rate when a plateau in model performance is detected, e.g., no change occurs for a given number of training epochs. This method can be used as a callback. `ReduceLROnPlateau` requires us to specify:

  - `monitor`, the metric to monitor during training;
  - `factor`, the value by which the learning rate will be multiplied when altered;
  - `patience`, which specifies the number of training epochs to wait before altering the learning rate.

  For example, we can monitor the validation loss, reduce the learning rate to $1/10$ of the actual value, and alter the LR when the validation loss does not improve for 100 epochs (table 5).

Model	`monitor`	`factor`	`patience`	`min_lr`
U-Net	Validation loss	0.1	100	0.0001

Table 5: Example of parameters used to reduce the learning rate with `ReduceLROnPlateau`.

Learning rate	**Train loss**	**Validation loss**	**Train IoU**	**Validation IoU**
fixed	1.134538	1.237166	0.936169	0.877554
adjustable	1.136153	1.227107	0.935371	0.884071

Table 6: Comparison between U-Net with total loss and fixed vs. adjustable LR after 300 epochs.

Table 6 shows the influence of the learning rate on the model performance. Adjusting LR with `ReduceLROnPlateau` results in a relatively small improvement in the validation

IoU. This trend can also be verified in figures 29 and 30, apart from a greater oscillation on the initial epochs, the final performance of U-Net is essentially the same.

The Keras DL library provides the following possibilities for adjusting the learning rate.

- Keras allow us to apply learning rate decay, or learning rate scheduling, to modify how the learning rate of the optimization process changes over time. The `learning rate schedules API` provides various methods to decay the LR: exponential, piecewise constant, polynomial, inverse time, cosine, and cosine with restarts. LR decay can be used with SGD, RMSprop, Adam, Adadelta, Adagrad, and other optimizers. Listing 5.4 gives an example of applying exponential learning rate decay.

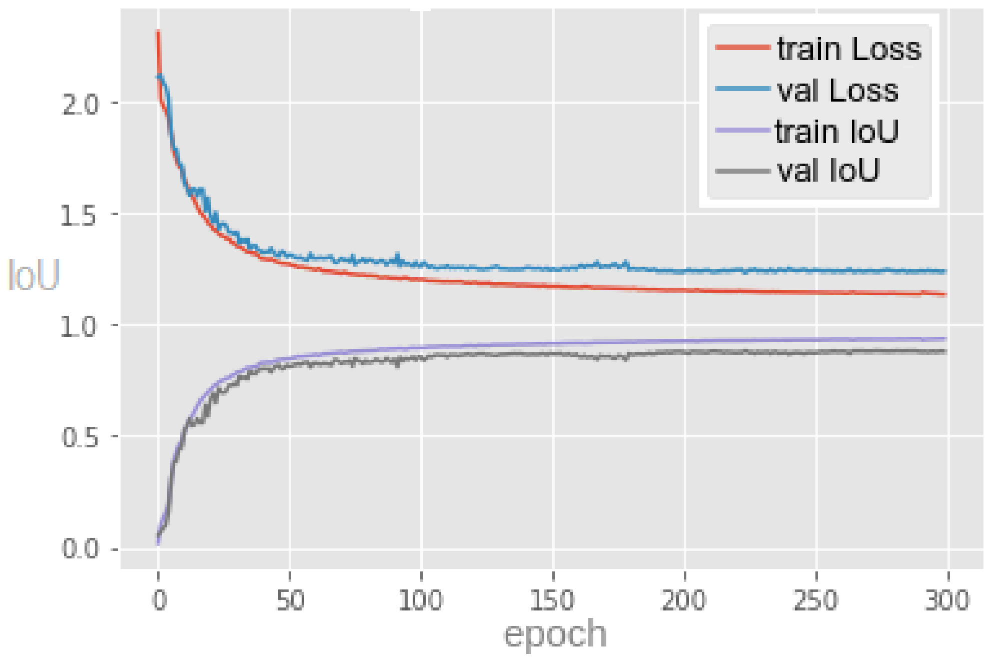


Figure 29: U-Net with total loss and fixed LR: loss value and IoU metric over 300 training epochs.

```
exp_lr_decay = keras.optimizers.schedules.ExponentialDecay(
    initial_learning_rate=1e-16,
    decay_steps=10000,
    decay_rate=0.9
    )
optimizer = keras.optimizers.Adam(learning_rate=exp_lr_decay)
```

Listing 5.4: Exponential learning rate decay in Keras.

- An adaptive control over the learning rate can also be applied by choosing an optimizer algorithm that implements it. Examples of these optimizers are `RMSprop`, `Adagrad`, or `Adam`.

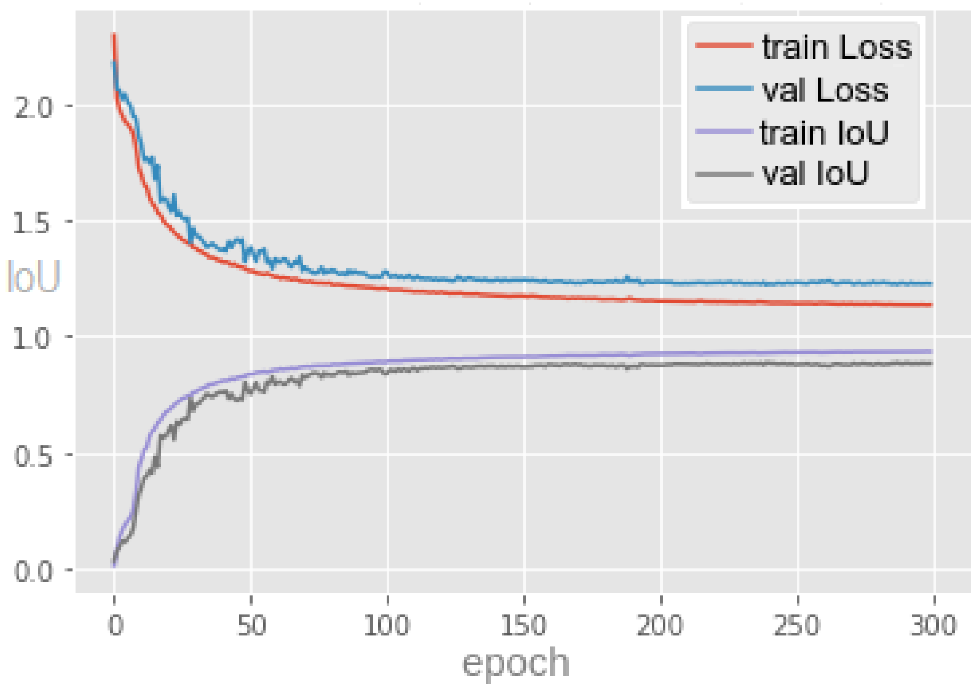


Figure 30: U-Net with total loss and adjustable LR: loss value and IoU metric over 300 training epochs.

5.4 LOSS FUNCTION

This section addresses the topic that was explored in greater depth in this dissertation, the evaluation of the impact of different loss functions on the performance of image segmentation models.

5.4.1 *Binary Cross Entropy Loss*

The binary cross-entropy function calculates the loss associated with one prediction $\hat{y}$, when the ground truth is y_i, by computing the equation 26. For a batch of samples, we can approximate their loss by an average of the individual losses. This is equivalent to the average result of the categorical cross-entropy loss function applied to many independent classification problems, each problem having only two possible classes with target probabilities y_i and $(1 - y_i)$. Binary cross entropy is available in Keras and, to use it, we just have to import the correct module, as shown in the listing 5.5.

```
from keras.losses import binary_crossentropy
```

Listing 5.5: Import the Keras module to use binary cross-entropy.

Table 7 and figure 31 show the loss and IoU metric associated with training the Tiramisu network with the binary cross-entropy loss function. We can observe that the model exhibits a large performance gap between training and validation. It seems that the model is too complex and does not generalize well when dealing with unseen data.

Table 8 and figure 32 show the loss and IoU metric associated with training the U-Net network with the binary cross-entropy loss function. U-Net has a slightly better performance than Tiramisu during training and shows a much higher generalization capability. Training stops before the planned 300 epochs, around 190 epochs, due to the `EarlyStop` callback.

Train loss	Validation loss	Train IoU	Validation IoU
0.004563	0.034805	0.875985	0.455383

Table 7: Tiramisu with Binary Cross-Entropy loss: loss value and IoU metric after 300 training epochs.

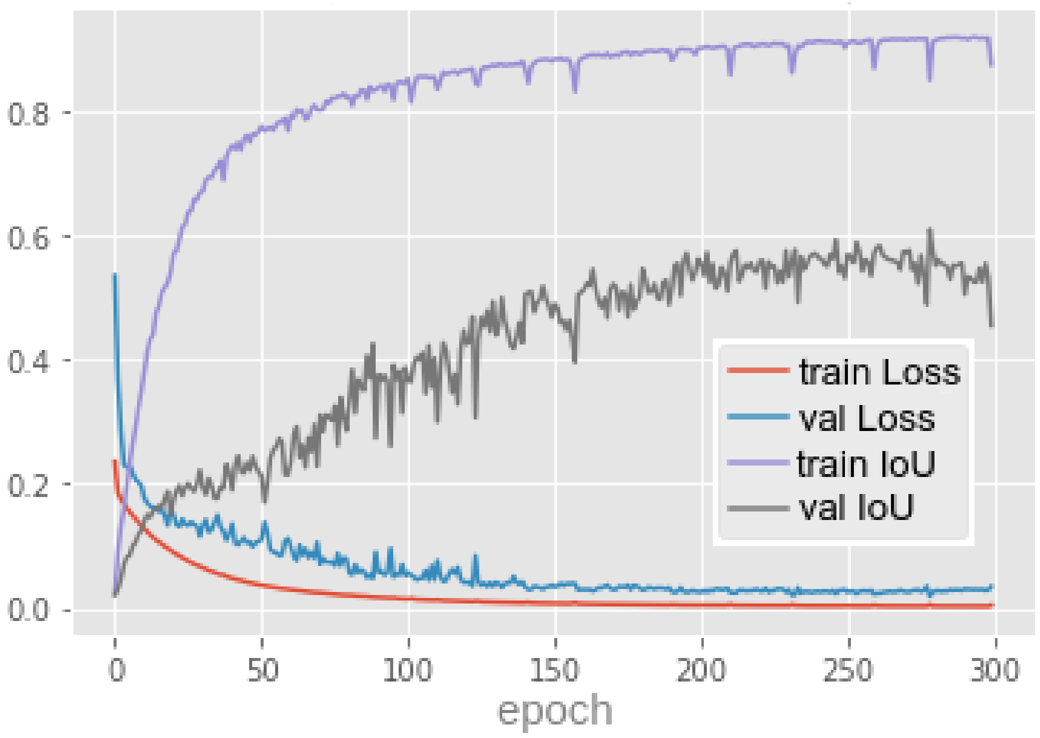


Figure 31: Tiramisu with BCE loss: loss value and IoU metric over 300 training epochs.

Train loss	Validation loss	Train IoU	Validation IoU
0.002910	0.010951	0.906896	0.839899

Table 8: U-Net with Binary Cross-Entropy loss: loss value and IoU metric after 300 epochs.

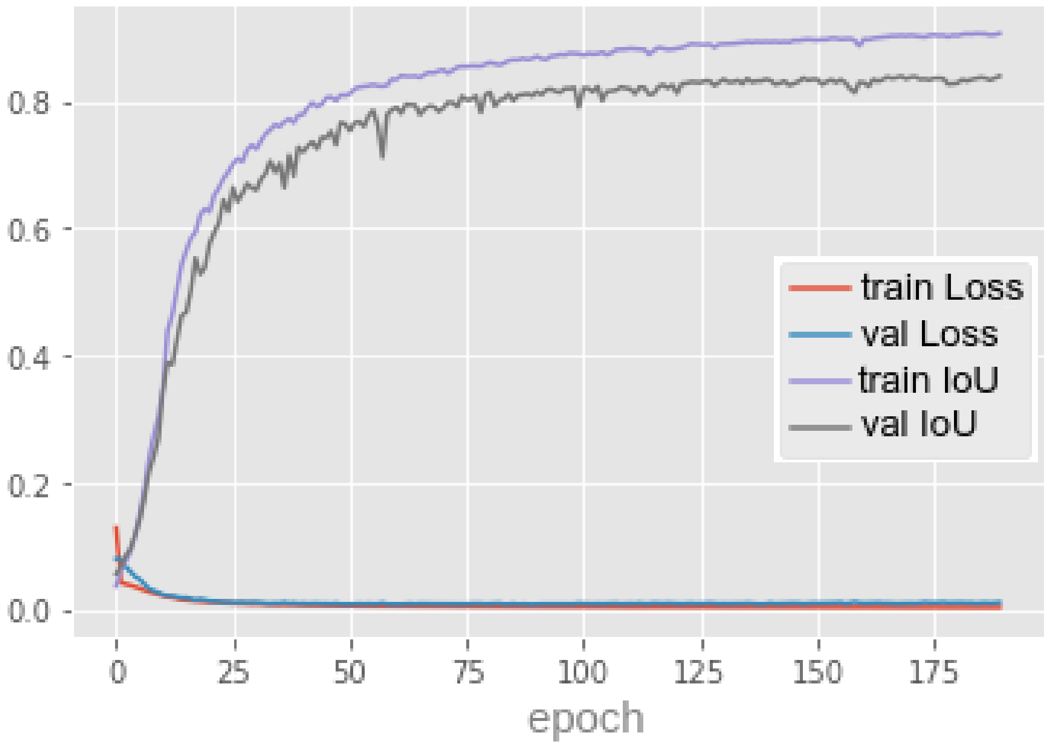


Figure 32: U-Net with BCE loss: loss value and IoU metric over 190 training epochs.

5.4.2 *Focal Loss*

The focal loss function generalizes the binary cross entropy by introducing a hyperparameter called **focusing parameter**, which allows hard-to-classify samples to be penalized more heavily than easy-to-classify samples (Lin et al., 2017). This loss can be considered as a loss function which down-weights the easily classified samples and gives much more importance to samples which are hard to classify. Focal loss solves the problem of class imbalance because samples from the majority class are usually easier to predict, whereas those from the minority class are harder to predict (khandelwal, 2020). This happens because samples from the majority class dominate the loss and, consequently, the backpropagation of the gradients. The focal loss can be seen as a cross-entropy loss modulated by the focusing parameter. The loss associated with the prediction $\hat{y}_i$, for the pixel i, is given by equation 33.

$$FocalLoss(\hat{y}_i) = -(1-\hat{y}_i)^{\gamma} log(\hat{y}_i) \tag{33}$$

Here $\gamma \geq 0$ is the focusing parameter and $\hat{y}_i$ is the prediction probability for the pixel i.

The image 33 shows how the focal loss behaves for different values of γ. When $\gamma = 0$, the focal loss is equivalent to the cross-entropy. Let us call the first part of the loss expression, $(1-\hat{y}_i)^{\gamma}$, the **modulating factor**. In the case of a misclassified sample, $\hat{y}_i$ is small and makes the modulating factor very close to 1. In this case, the loss function behaves as a Cross-Entropy loss. As the confidence of the model increases, $\hat{y}_i$ becomes close to 1 and the modulating factor approaches 0. In this case, the importance of well-classified samples in the loss is down-weighted. The focusing parameter γ will rescale the modulating factor, so that the easy-to-classify samples are down-weighted more than the hard-to-classify ones, reducing their impact on the loss function.

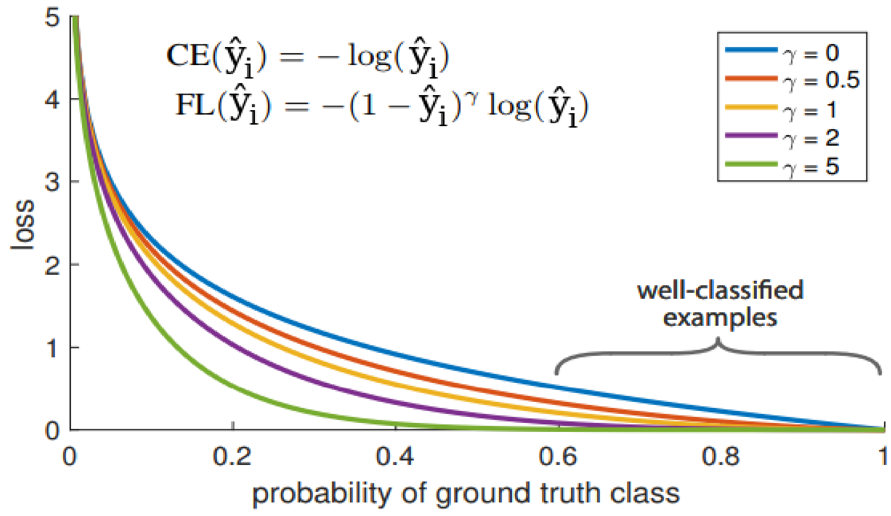


Figure 33: Behavior of the focal loss for different values of the focusing parameter γ.

Table 9 and figure 34 show the loss and IoU metric associated with U-Net training with focal loss. Training U-Net with BCE loss achieved better performance than training and with focal loss. The focal loss was expected to result in better performance in the segmentation task. Again, training stops before the planned 300 epochs due to the `EarlyStop` callback.

Train loss	**Validation loss**	**Train IoU**	**Validation IoU**
0.000847	0.003178	0.789232	0.726670

Table 9: U-Net with focal loss: loss value and IoU metric after 300 epochs.

5.4.3 *Dice Loss*

Dice is another popular loss function for the image segmentation task. It is based on the Dice coefficient, which is essentially a measure of the overlap between the prediction and the ground truth masks (figure 35). This measure ranges from 0 to 1, where a Dice coefficient of 1 denotes complete overlap. The Dice coefficient was originally developed for binary data and can be calculated using equation 34.

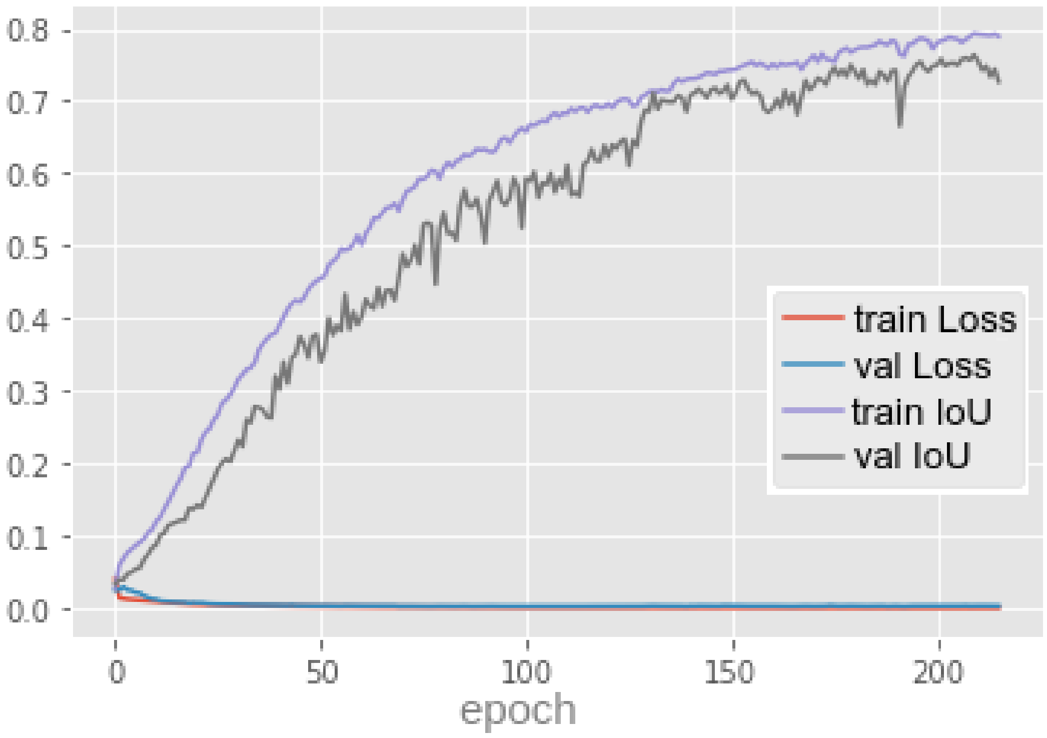


Figure 34: U-Net with focal loss: loss value and IoU metric over 215 epochs.

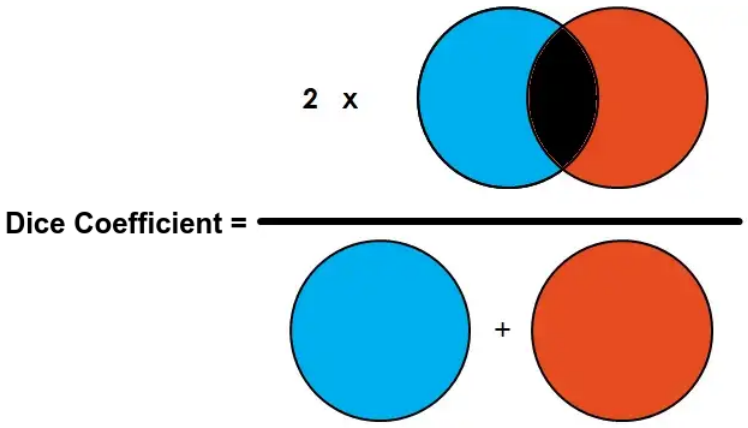


Figure 35: Meaning of the Dice coefficient.

$$Dice(Y, \hat{Y}) = \frac{2.|Y \cap \hat{Y}|}{|Y| + |\hat{Y}|} \tag{34}$$

where Y is the ground truth mask, $\hat{Y}$ is the predicted mask, $|Y \cap \hat{Y}|$ represents the common elements between the masks Y and $\hat{Y}$, $|Y|$ represents the sum of the elements in the mask Y, and $|\hat{Y}|$ is the sum of the elements in the mask $\hat{Y}$.

For the case of evaluating a Dice coefficient on predicted segmentation masks, we can approximate $|Y \cap \hat{Y}|$ by the element-wise multiplication between the predicted and ground truth masks (figure 36). The Dice coefficient can be calculated with the code included in listing 5.6.

$$|\mathbf{Y} \cap \hat{\mathbf{Y}}| = \underset{\text{ground truth}}{\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}} * \underset{\text{prediction}}{\begin{bmatrix} 0.01 & 0.03 & 0.02 & 0.02 \\ 0.05 & 0.12 & 0.09 & 0.07 \\ 0.89 & 0.85 & 0.88 & 0.91 \\ 0.99 & 0.97 & 0.95 & 0.97 \end{bmatrix}} \xrightarrow{\substack{\text{element-wise} \\ \text{multiply}}} \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0.89 & 0.85 & 0.88 & 0.91 \\ 0.99 & 0.97 & 0.95 & 0.97 \end{bmatrix} \xrightarrow{\text{sum}} 7.41$$

Figure 36: Example of calculating the $|Y \cap \hat{Y}|$ term present in the Dice coefficient.

Because our segmentation masks are binary, the product in the numerator has no contribution from the predictions whose ground truth is 0 . For the pixels that should be 1, the coefficient penalizes the low-confidence predictions, since they contribute with a small value when they should contribute with a value close to 1. A higher value for the product in the numerator leads to a better Dice coefficient. To quantify $|Y|$ and $|\hat{Y}|$, some researchers use the simple sum, whereas other researchers prefer the squared sum (Jordan, 2018a).

```python
def dsc(y_true, y_pred):
    smooth   = 0.00001
    y_true_f = K.flatten(y_true)
    y_pred_f = K.flatten(y_pred)
    intersection = K.sum(y_true_f * y_pred_f)
    score = ((2. * intersection + smooth) / (K.sum(y_true_f) + K.sum(y_pred_f) + smooth))
    return score
```

Listing 5.6: Computing the Dice coefficient.

The Dice loss is simply 1 minus the Dice coefficient, and as such its range of variation is also from 0 to 1 (equation 35).

$$DiceLoss(Y, \hat{Y}) = 1 - \frac{2.|Y \cap \hat{Y}|}{|Y| + |\hat{Y}|} \tag{35}$$

5.4.4 *Jaccard Loss*

The IoU, or Jaccard index, is similar to the Dice coefficient and is calculated by equation 36.

$$J(Y, \hat{Y}) = \frac{|Y \cap \hat{Y}|}{|Y + \hat{Y}|} = \frac{|Y \cap \hat{Y}|}{|Y| + |\hat{Y}| - |Y \cap \hat{Y}|} \tag{36}$$

The Jaccard loss is simply 1 minus the Jaccard index, and its range of variation is from 0 to 1 (equation 37). Using the `iou` method presented in listing 5.1 to calculate IoU, the Python implementation of the Jaccard loss is quite simple (listing 5.7).

$$JL(Y, \hat{Y}) = 1 - \frac{|Y \cap \hat{Y}|}{|Y| + |\hat{Y}| - |Y \cap \hat{Y}|} \tag{37}$$

Where Y is the ground truth mask and $\hat{Y}$ is the predicted mask.

```python
def jaccard_loss(y_true, y_pred):
    loss = 1 - iou(y_true, y_pred)
    return loss
```

Listing 5.7: Code to calculate the Jaccard loss.

Table 10 and figure 37 show the loss and IoU metric associated with the training of Tiramisu with the Jaccard loss. The large performance gap between training and validation, observed with the BCE loss, remains.

Table 11 and figure 38 present the loss and IoU metric for U-Net trained with the Jaccard loss function. In this case, U-Net exhibits the best performance among all configurations described until now.

Train loss	Validation loss	Train IoU	Validation IoU
0.102543	0.542375	0.933835	0.485366

Table 10: Tiramisu with Jaccard loss: loss value and IoU metric after 300 epochs.

Train loss	Validation loss	Train IoU	Validation IoU
0.063033	0.117658	0.936963	0.884401

Table 11: U-Net with Jaccard loss: loss value and IoU metric after 300 epochs.

5.4.5 *Binary Cross Entropy-Dice Loss*

The BCE-Dice loss function combines two terms, binary cross entropy and Dice (equation 38). The term *CrossEntropyLoss* is calculated with the equation 26 and term *DiceLoss* is calculated with the equation 35.

$$BceDiceLoss = CrossEntropyLoss + DiceLoss \tag{38}$$

Table 12 and figure 39 show the loss and IoU metric associated with the training of Tiramisu with the BCE-Dice loss. The validation IoU achieved by Tiramisu improved to 57.4%, when compared to previous Tiramisu configurations, but is still much lower than the U-Net results.

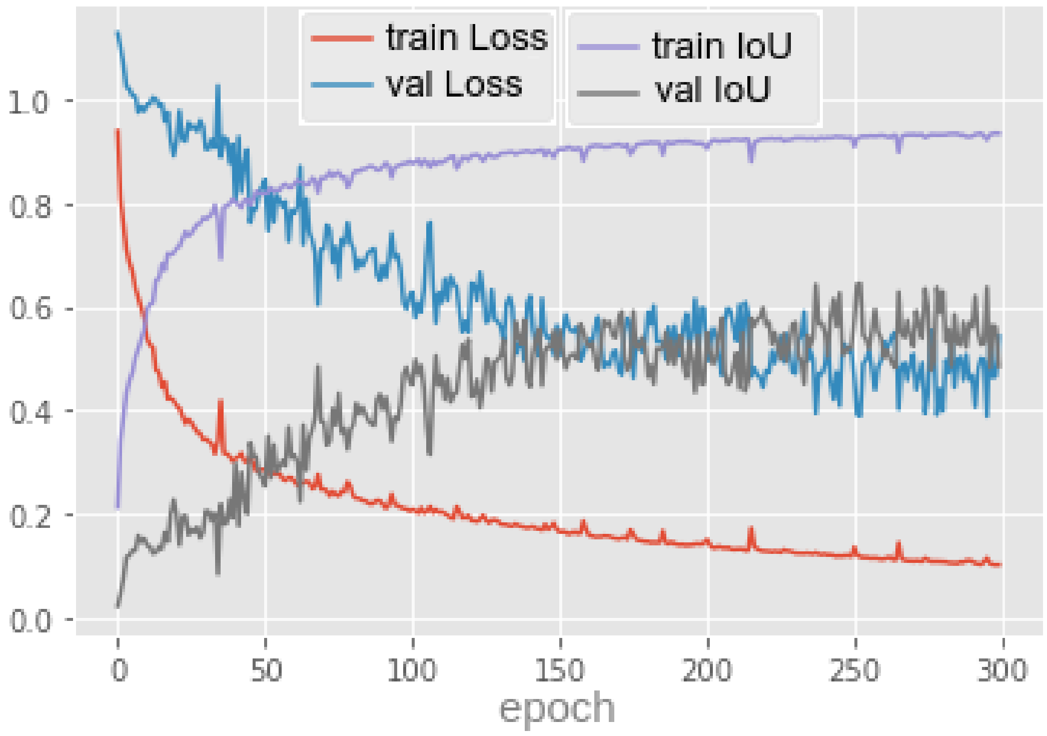


Figure 37: Tiramisu with Jaccard loss: loss value and IoU metric over 300 epochs.

Table 13 and figure 40 present the loss and IoU metric for U-Net trained with the BCE-Dice loss function. Although the training IoU is the highest among all configurations tested with U-Net, the validation IoU is 86.1%, which is 2% lower than the best result.

Train loss	Validation loss	Train IoU	Validation IoU
0.105560	0.514741	0.937979	0.574185

Table 12: Tiramisu with BCE-Dice loss: loss value and IoU metric after 300 epochs.

Train loss	Validation loss	Train IoU	Validation IoU
0.063428	0.139721	0.941786	0.861621

Table 13: U-Net with BCE-Dice loss: loss value and IoU metric after 300 epochs.

5.4.6 *Tversky Loss*

One of the limitations of the Dice loss function is that it equally weighs false positive and false negative predictions. Tversky loss was introduced by Salehi et al. (2017) and was thought to optimize segmentation models trained with unbalanced datasets. It is based on

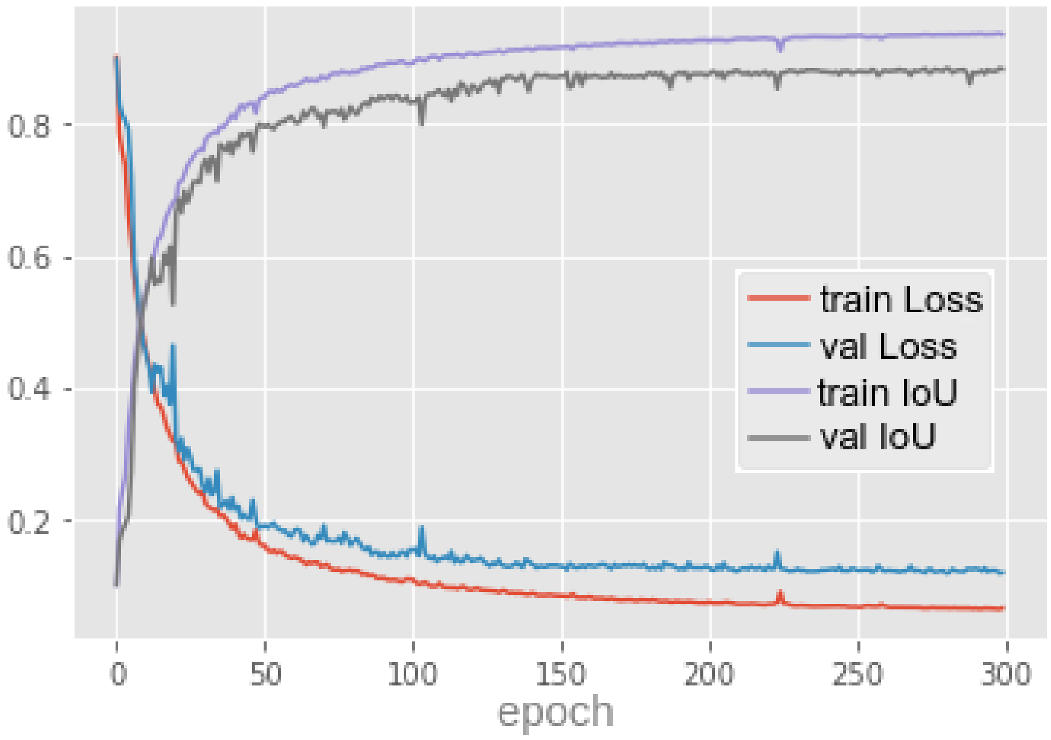


Figure 38: U-Net with Jaccard loss: loss value and IoU metric over 300 epochs.

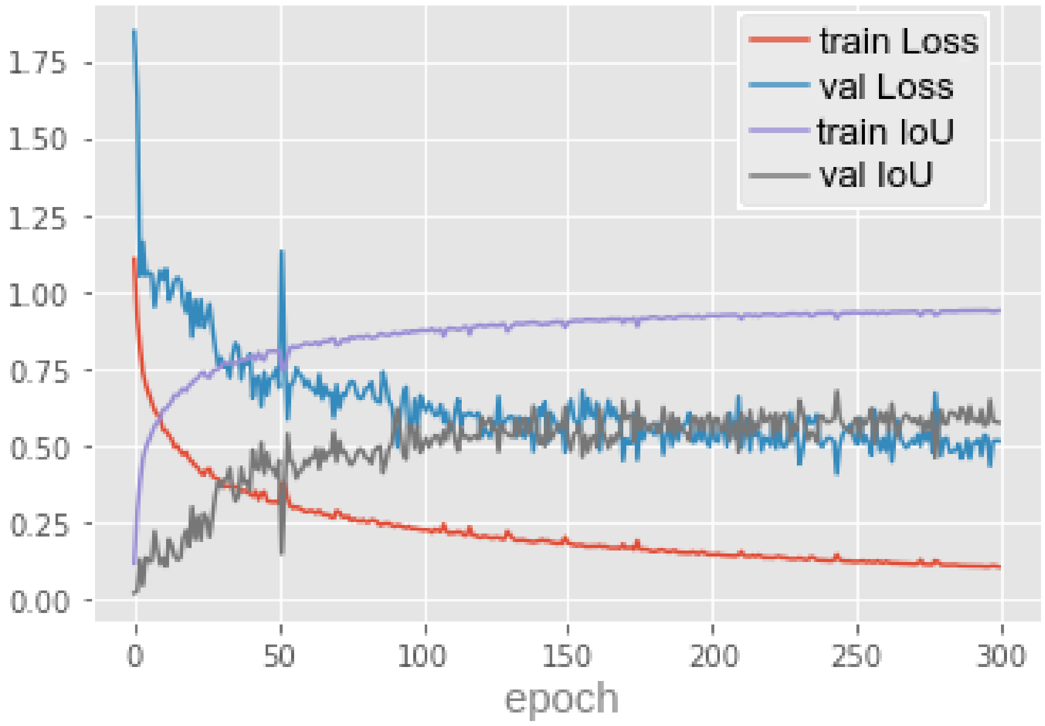


Figure 39: Tiramisu with BCE-Dice loss: loss value and IoU metric over 300 epochs.

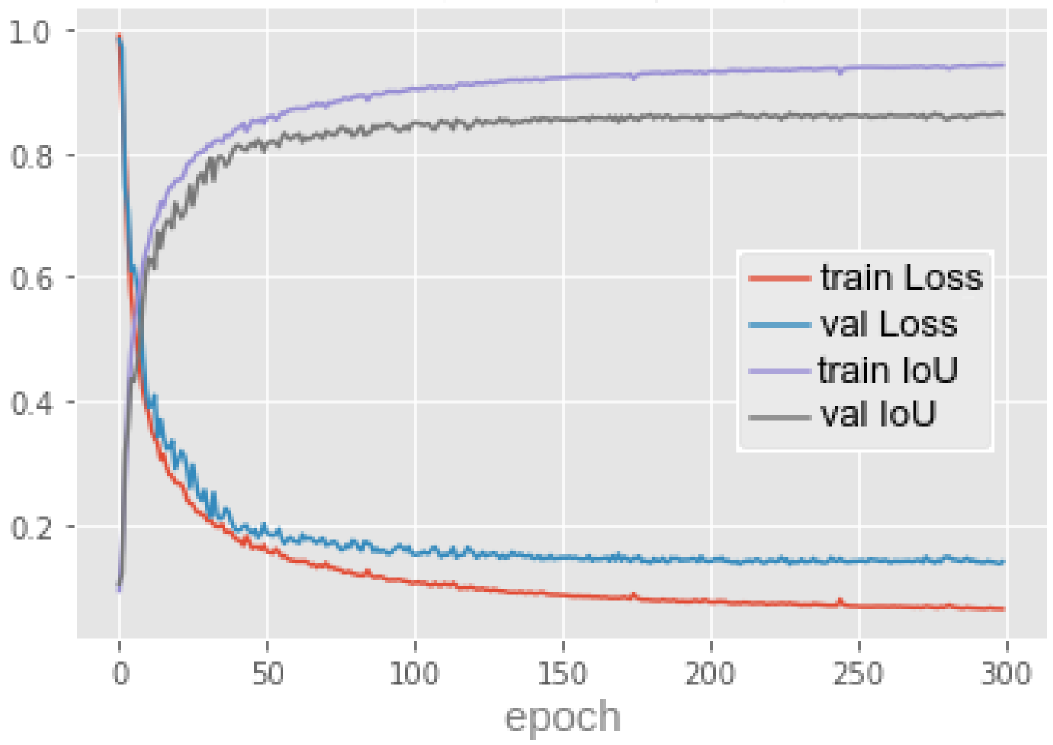


Figure 40: U-Net with BCE-Dice loss: loss value and IoU metric after 300 epochs.

the Tversky index and can be seen as a generalization of the Dice coefficient and the Jaccard index. The Tversky index is a number from 0 to 1, given by equation 39.

$$TverskyIndex(Y, \hat{Y}) = \frac{|Y \cap \hat{Y}|}{|Y \cap \hat{Y}| + \alpha|\hat{Y} \setminus Y| + \beta|Y \setminus \hat{Y}|} = \frac{TP}{TP + \alpha\ FP + \beta\ FN} \tag{39}$$

where

- Y is the ground truth mask.
- $\hat{Y}$ is the predicted mask.
- $|Y \cap \hat{Y}|$ is the set of elements common to Y and $\hat{Y}$, or the true positives (TP).
- $|Y \setminus \hat{Y}|$ is the relative complement of $\hat{Y}$ in Y, or the members of Y who are not members of $\hat{Y}$, or the false negatives (FN).
- $|\hat{Y} \setminus Y|$ is the relative complement of Y in $\hat{Y}$, or the members of $\hat{Y}$ who are not members of Y, or the false positives (FP).
- α is the parameter that controls the magnitude of the penalty applied to false positives (FP).

- β is the parameter that controls the magnitude of the penalty applied to false negatives (FN).

The parameters α and β are equal to or greater than 0. If $\alpha = \beta = 1$ the Tversky index becomes the Tanimoto coefficient, when $\alpha = \beta = 0.5$ the index is equal to the Dice coefficient, and if $\alpha = \beta = 1$ the index is equal to the Jaccard index. If we choose $\alpha > \beta$ we penalize false positives more than false negatives.

Since the Tversky index is a number between 0 and 1, we can define the Tversky loss as 1 minus the Tversky index (equation 40). The Python code included in the listing 5.8 implements the Tversky loss.

$$TverskyLoss(Y,\hat{Y}) = 1 - \frac{|Y \cap \hat{Y}|}{|Y \cap \hat{Y}| + \alpha|\hat{Y} \setminus Y| + \beta|Y \setminus \hat{Y}|} = 1 - \frac{TP}{TP + \alpha\ FP + \beta\ FN} \tag{40}$$

Table 14 and figure 41 present the loss and IoU metric for U-Net trained with the Tversky loss function. The validation IoU is 86.5% and is roughly the same as the U-Net trained with BCE-dice loss.

```python
def tversky(y_true, y_pred, alpha=0.7): # Tversky index
    y_true_pos = K.flatten(y_true)
    y_pred_pos = K.flatten(y_pred)
    true_pos   = K.sum(y_true_pos * y_pred_pos)
    false_neg  = K.sum(y_true_pos * (1-y_pred_pos))
    false_pos  = K.sum((1-y_true_pos)*y_pred_pos)
    Tindex = (true_pos + smooth)/(true_pos + alpha*false_neg + (1-alpha)*false_pos + smooth)
    return Tindex

def tversky_loss(y_true, y_pred, alpha=0.7):  # Tversky loss
    return 1 - tversky(y_true, y_pred, alpha)
```

Listing 5.8: Python code that implements the Tversky index and the Tversky loss function.

Train loss	Validation loss	Train IoU	Validation IoU
0.058718	0.117278	0.924808	0.865769

Table 14: U-Net with Tversky loss: loss value and IoU metric after 300 epochs.

5.4.7 *Focal Tversky Loss*

The focal Tversky loss (FTL) was proposed by Abraham and Khan (2019) to address the lower ability of the Dice loss to segment small-sized regions of interest. The focal Tversky loss is a generalization of the Tversky loss. It is defined by equation 41 and is parameterized by γ.

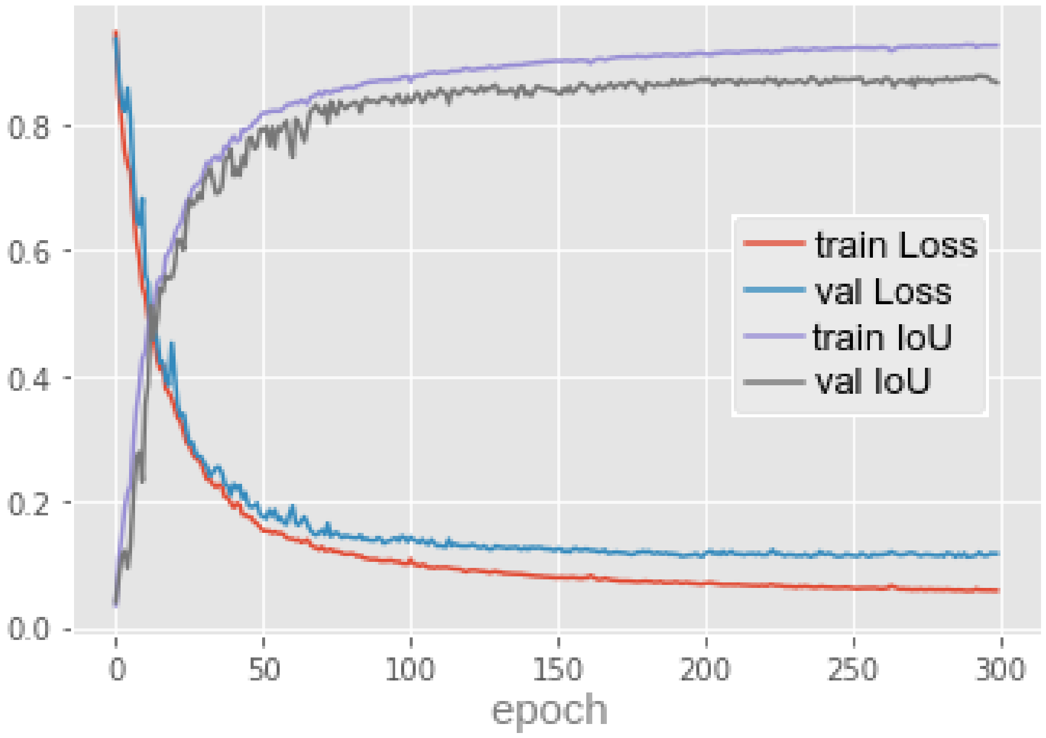


Figure 41: U-Net with Tversky loss: loss value and IoU metric over 300 epochs.

The authors suggested that γ must be in the range $[1 : 3]$, and γ favors the segmentation of regions of interest that are difficult to identify over the background that is easy to isolate.

$$FocalTverskyLoss(Y, \hat{Y}) = (1 - TverskyIndex(Y, \hat{Y}))^{\frac{1}{\gamma}} \tag{41}$$

The behavior of the focal Tversky loss as a function of the Tversky index, for different values of the parameter γ, is plotted in figure 42. In this figure, the linear blue line is the Tversky loss.

The nonlinear behavior of focal Tversky loss allows training to focus on detecting samples that are difficult to segment, i.e. samples with Tversky index < 0.5, by assigning a higher contribution to the loss when these samples are misclassified. On the other hand, the nonlinear behavior of focal Tversky loss reduces the contribution to the loss when a sample that is easy to segment is misclassified.

In cases where $\gamma < 1$, the loss gradient is higher for samples with a Tversky index > 0.5 (correctly classified samples), forcing the model to focus on correct segmentation of such samples. This behavior can be useful in later stages of the training process, as the model is encouraged to learn even though we are approaching convergence. However, this behavior will favor the correct detection of easy samples during the early stages of training, which can lead to weak learning (Vinod, 2020).

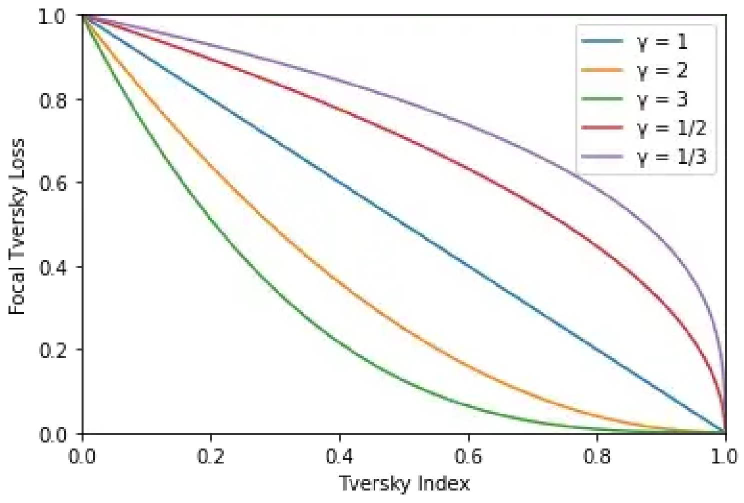


Figure 42: The focal Tversky loss behavior for different values of the parameter γ.

When dealing with an imbalanced dataset, the focal Tversky loss is useful when $\gamma > 1$. In this case, the loss gradient is higher for samples with a Tversky index < 0.5 (incorrectly classified samples). In this way, the loss encourages the model to focus on the correct classification of samples that are difficult to segment, especially small regions, which usually have a low Tversky index. Listing 5.9 contains the Python implementation of the focal Tversky loss function.

```python
def focal_tversky(y_true, y_pred, alpha=0.7, gamma=1.3333):  # Focal Tversky loss
    Tindex    = tversky(y_true, y_pred, alpha)
    inv_gamma = 1.0/gamma
    FTloss    = K.pow((1 - Tindex), inv_gamma)
    return FTloss
```

Listing 5.9: Python code that implements the focal Tversky loss function.

Table 15 and figure 43 show the loss and IoU metric associated with the training of Tiramisu with the focal Tversky. The validation IoU achieved by Tiramisu is only 45.3%.

Table 16 and figure 44 present the loss and IoU metric for U-Net trained with the focal Tversky loss function. The training and validation IoU are about the same as when using Tversky loss.

Train loss	Validation loss	Train IoU	Validation IoU
0.938525	0.944698	0.633185	0.453520

Table 15: Tiramisu with focal Tversky loss: loss value and IoU metric after 300 epochs.

Train loss	Validation loss	Train IoU	Validation IoU
0.121612	0.202741	0.922976	0.869215

Table 16: U-Net with focal Tversky loss: loss value and IoU metric after 300 epochs.

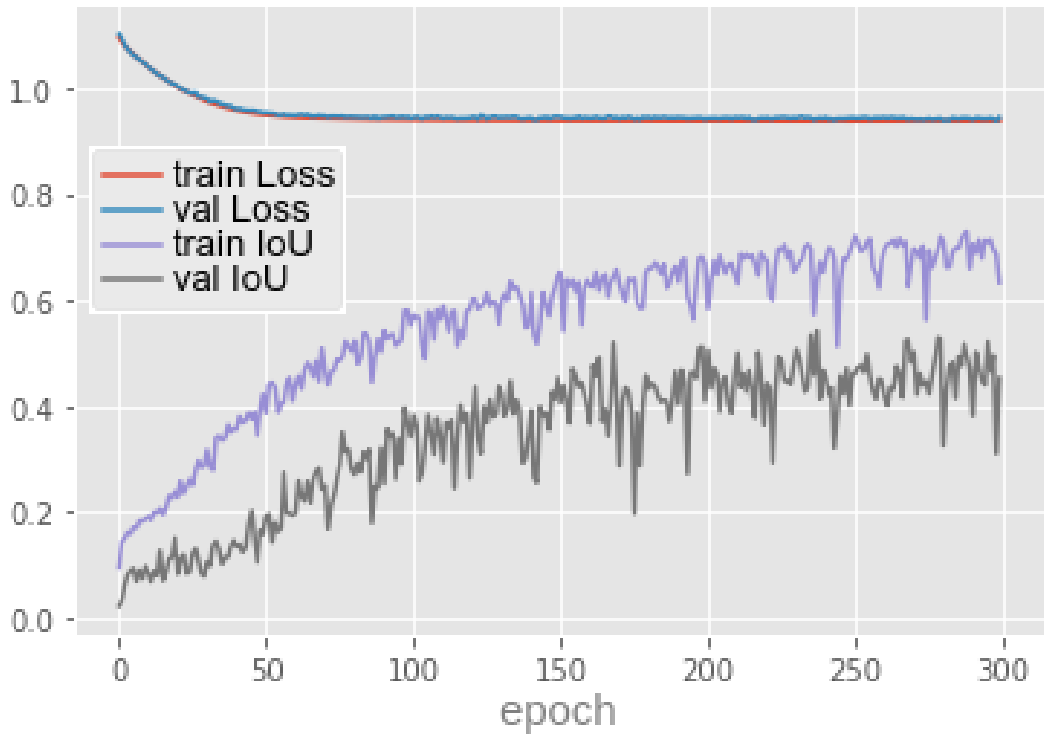


Figure 43: Tiramisu with focal Tversky loss: loss value and IoU metric after 300 epochs.

5.4.8 *Total Loss*

To conclude the evaluation of the loss functions, it was decided to combine most of the losses presented in a total loss function (equation 42).

$$\begin{aligned} TotalLoss = CrossEntropyLoss(Y, \hat{Y}) + FocalLoss(\hat{Y}) + DiceLoss(Y, \hat{Y}) + \\ TverskyLoss(Y, \hat{Y}) + FocalTverskyLoss(Y, \hat{Y}) \end{aligned} \tag{42}$$

Where Y is the ground truth mask and $\hat{Y}$ is the predicted mask.

Table 17 and figure 45 show the loss and IoU metric associated with the training of Tiramisu with the total loss. The validation IoU raised to 67.8%, by far the best result achieved by Tiramisu. Table 18 and figure 46 present the loss and IoU metric for U-Net trained with the total loss. The validation IoU is 87.7%, 0.7% less than U-Net trained with Jaccard loss.

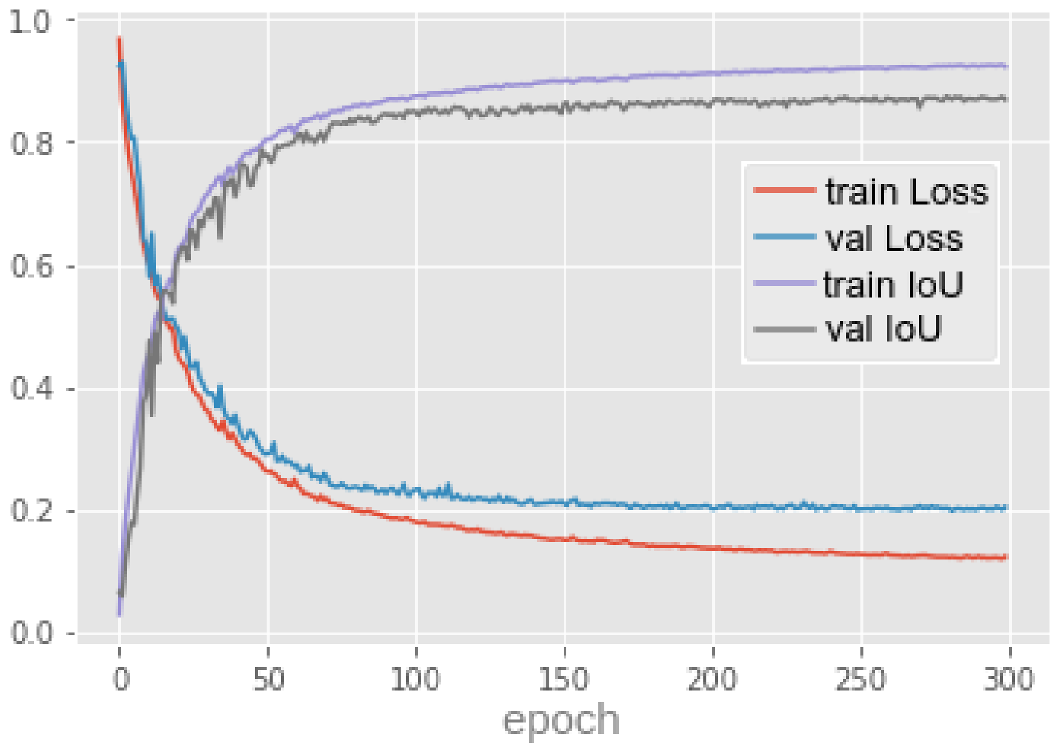


Figure 44: U-Net with focal Tversky loss: loss value and IoU metric over 300 epochs.

Train loss	Validation loss	Train IoU	Validation IoU
1.045244	1.339628	0.937572	0.677880

Table 17: Tiramisu with total loss: loss value and IoU metric after 300 epochs.

Train loss	Validation loss	Train IoU	Validation IoU
1.134538	1.237166	0.936169	0.877554

Table 18: U-Net with total loss: loss value and IoU metric after 300 epochs.

We summarize the results of all the experiments performed with the different loss functions in tables 19 and 20, for the Tiramisu and U-Net models, respectively.

Loss Function	Train loss	Validation loss	Train IoU	Validation IoU
Binary Cross Entropy	0.004563	0.034805	0.875985	0.455383
Jaccard	0.102543	0.542375	0.933835	0.485366
BCE-Dice	0.105560	0.514741	0.937979	0.574185
Focal Tversky	0.938525	0.944698	0.633185	0.453520
Total	1.045244	1.339628	0.937572	0.677880

Table 19: Summary of the training results for the Tiramisu model with different loss functions.

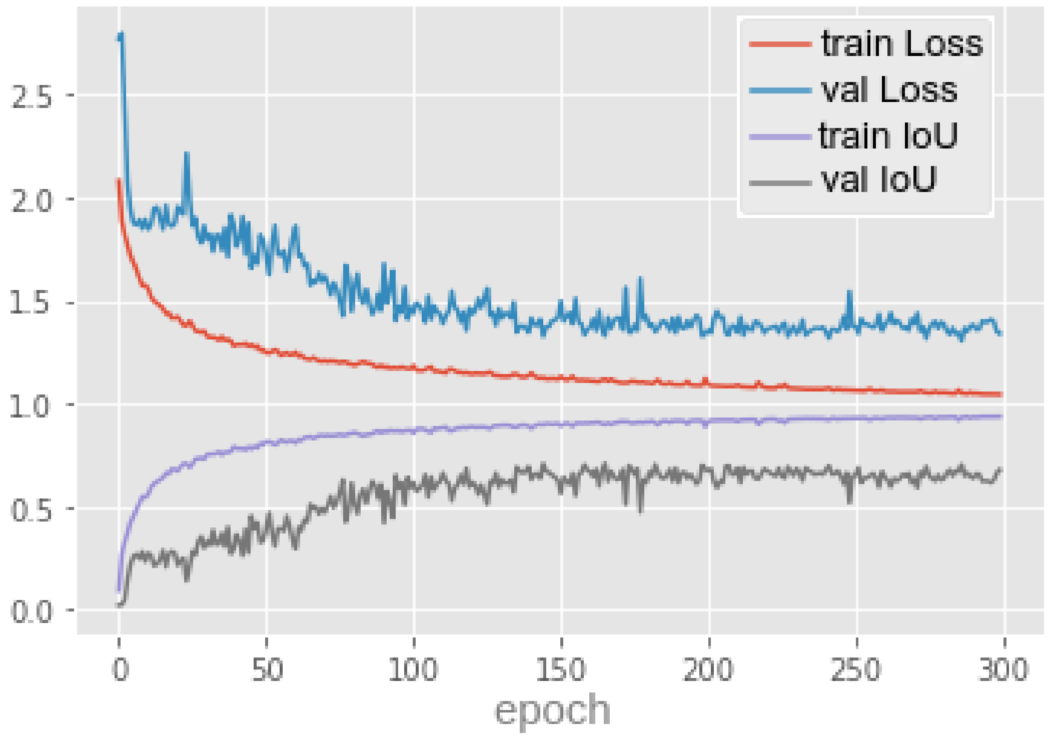


Figure 45: Tiramisu with total loss: loss value and IoU metric over 300 epochs.

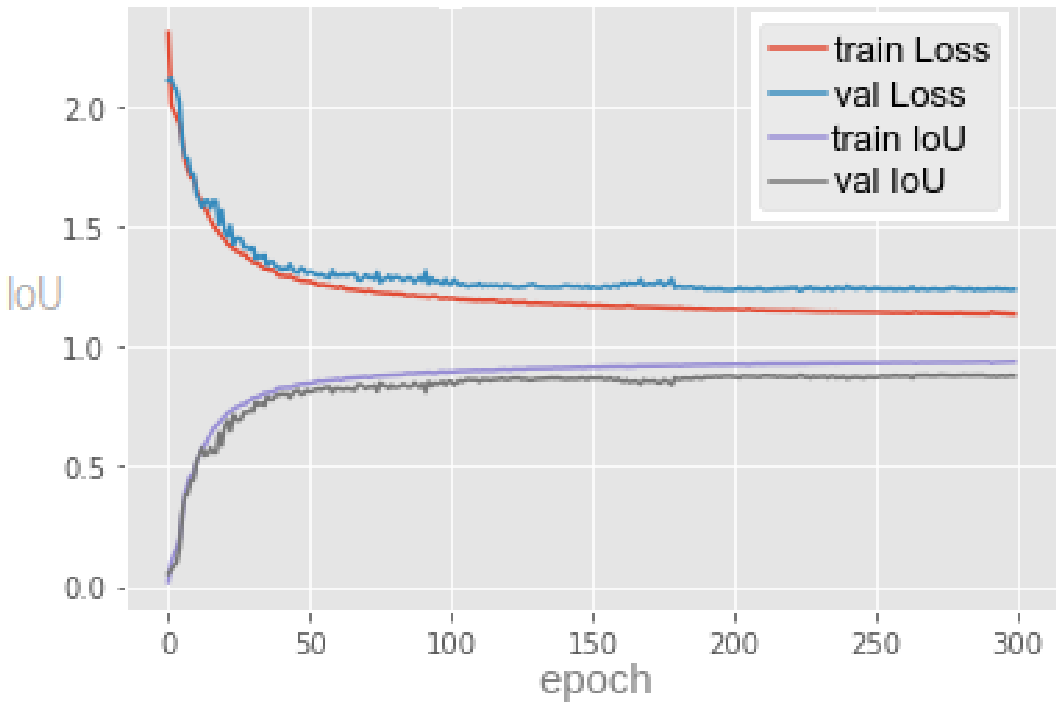


Figure 46: U-Net with total loss: loss value and IoU metric over 300 epochs.

Loss Function	**Train loss**	**Validation loss**	**Train IoU**	**Validation IoU**
Binary Cross Entropy	0.002910	0.010951	0.906896	0.839899
Focal	0.000847	0.003178	0.789232	0.726670
Jaccard	0.063033	0.117658	0.936963	0.884401
BCE-Dice	0.063428	0.139721	0.941786	0.861621
Tversky	0.058718	0.117278	0.924808	0.865769
Focal Tversky	0.121612	0.202741	0.922976	0.869215
Total	1.134538	1.237166	0.936169	0.877554

Table 20: Summary of the training results for the U-Net model with different loss functions.

5.5 MODEL TRAINING

In this section, additional information about model training is given. Training is the most computationally demanding task of the whole development process. To speed up training, it is convenient to run it on GPUs. Model training was run on the Google Colaboratory platform, with 25 GB of RAM and a freely available GPU. This is not high-end hardware at all, but it was enough to run model training.

In Python, the model optimization is done with the Keras `model.fit` method. To speed up the process of loading the training data to the Google Colaboratory platform, it is imported from a zip file. Table 21 contains the main hyperparameters of model training: batch size, number of epochs, optimizer, and the initial value of the learning rate (LR). As explained in the previous section, several loss functions were applied during training.

Model	Batch size	Epochs	Optimizer
U-Net	32	300	Adam, LR=0.0001, epsilon=1e-07
Tiramisu	32	300	Adam, LR=0.0001, epsilon=1e-07

Table 21: Model training hyperparameters LR initial value.

Figures 47 and 48 plot the accuracy metric (IoU) during the training of Tiramisu and U-Net, using different loss functions. We can confirm that total loss allows Tiramisu to obtain the best validation accuracy (67.8%). Excluding the BCE-focal loss, all loss functions allow U-Net to reach a similar accuracy, but training with the Jaccard loss resulted in the best validation accuracy (88.4%).

Besides analyzing the models' accuracy, another relevant aspect to consider is the time necessary to train the models. In Google Colaboratory platform, with the free plan, the time required for training each model is presented in table 22 and figure 49. Since Tiramisu is a model much larger than U-Net, its training is naturally much longer.

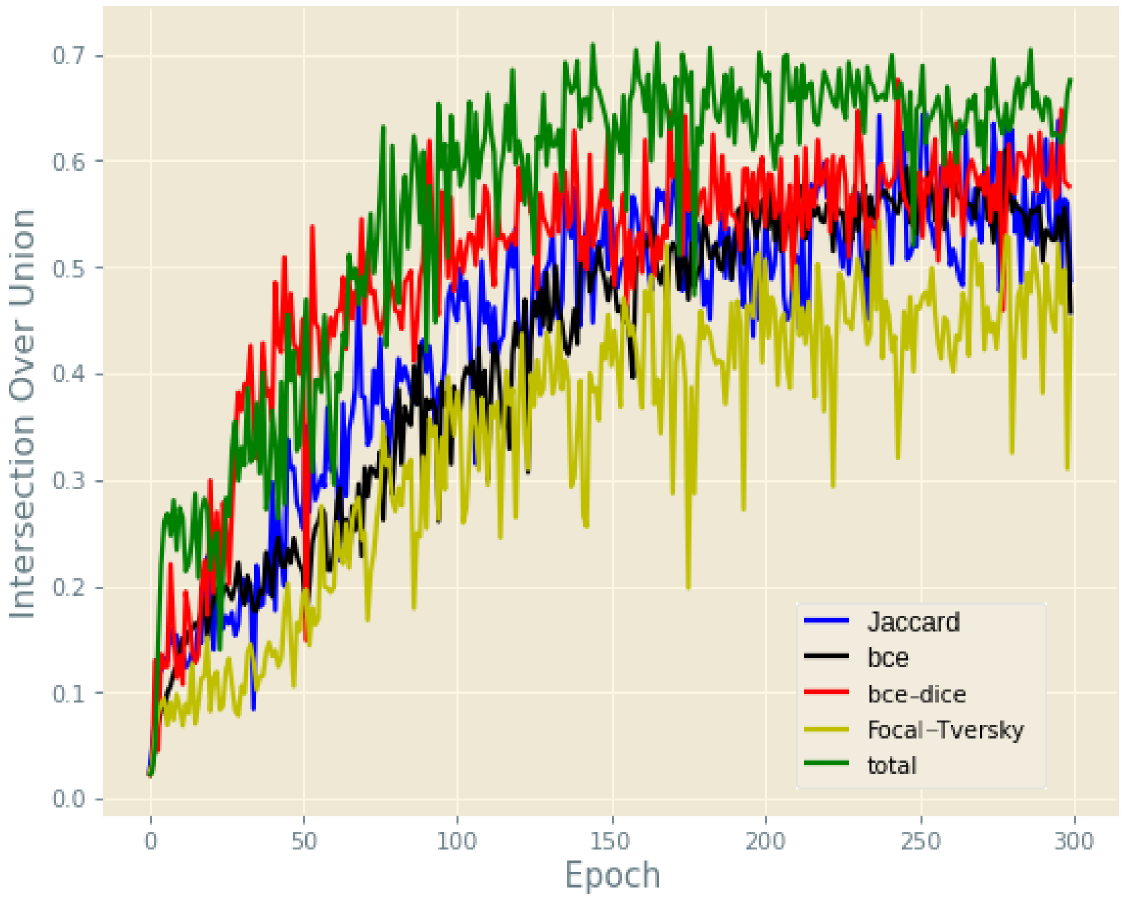


Figure 47: Training Tiramisu with different loss functions: IoU metric.

Model	**Loss function**	**Metric**	**Time** (hr:min)
Tiramisu	BCE	IoU	10:39
	BCE-Dice		4:50
	Focal Tversky		4:55
	Total		4:40
U-Net	BCE	IoU	0:43
	BCE-Dice		0:45
	Jaccard		0:43
	BCE-Focal		0:42
	Tversky		0:42
	Focal Tversky		1:39
	Total		1:40
	Total, reduce LR callback		1:40

Table 22: Time necessary to train the different models during 300 epochs.

Another aspect that was analyzed is the size of the models. The size occupied by the U-Net model on disk is given in table 23. Interestingly, training U-Net with Binary Cross Entropy Focal loss makes the model almost 3 times more spacy than the other losses.

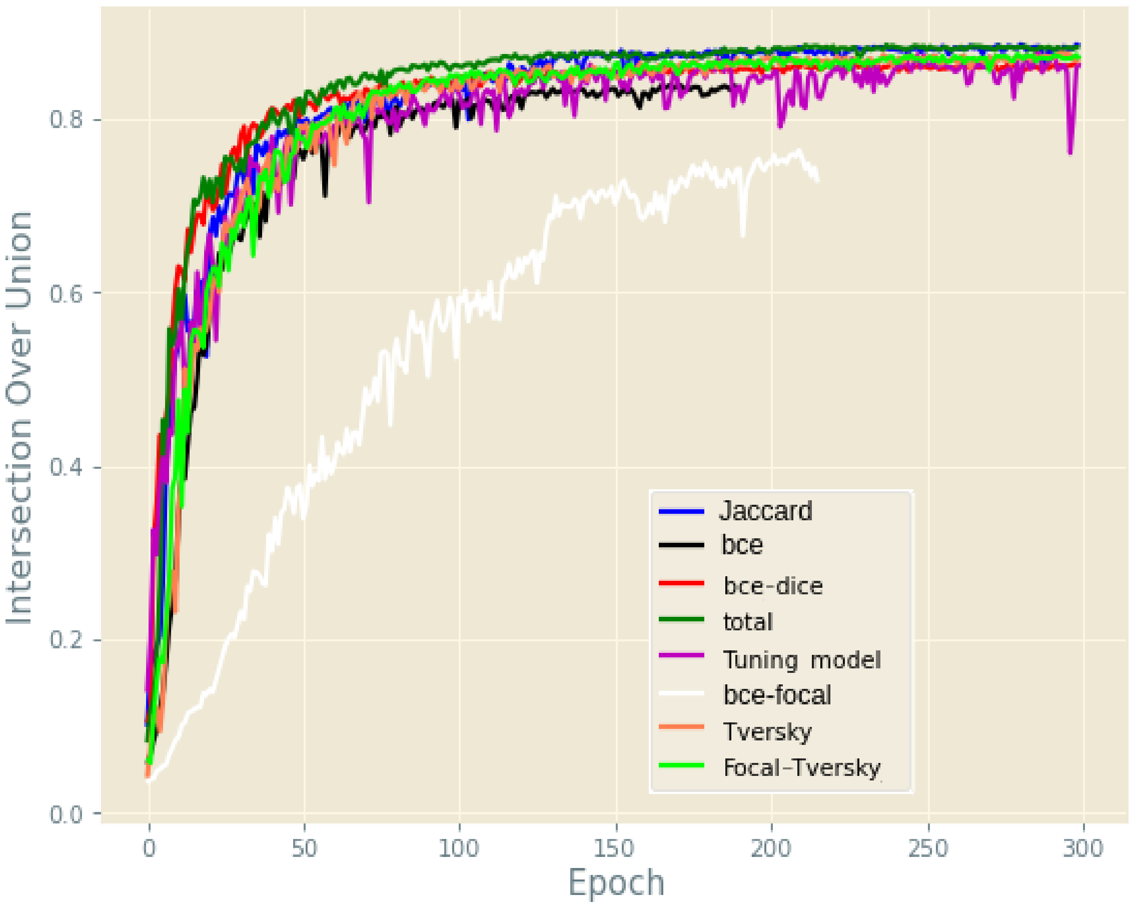


Figure 48: Training U-Net with different loss functions: IoU metric.

Model	**Loss function**	**Metric**	**Size** (B)
U-Net	BCE	IoU	52,237,793
	BCE-Dice		67,143,837
	Jaccard		22,428,841
	BCE-Focal		22,427,193
	Tversky		22,427,193
	Focal Tversky		22,427,353
	Total		22,428,093
	Total, reduce LR callback		22,427,353
	Jaccard+tuning model		37,334,114

Table 23: Size occupied by the U-Net model on disk.

5.6 MODEL TESTING

For a visual inspection of the quality of the segmentation, some samples were plotted during the test step. Thus, figure 50 shows 4 images from the test set, together with the respective

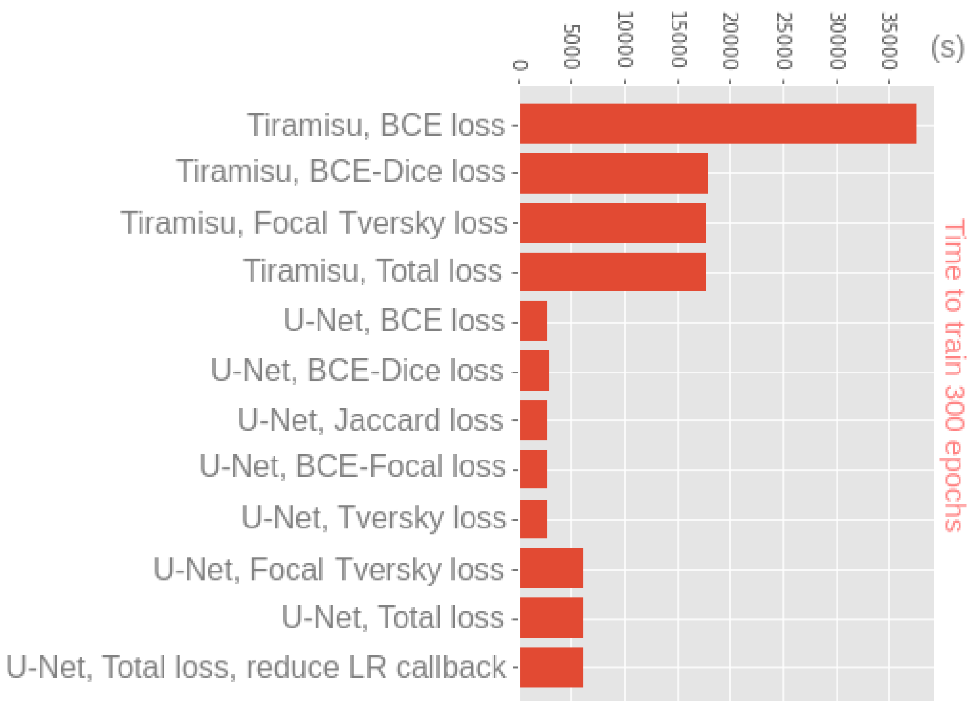


Figure 49: Time necessary to train the different models during 300 epochs.

ground truth mask and the predicted mask. The model used on the predictions is U-Net, trained with Dice, BCE-Dice, focal Tversky, and total losses.

Figure 51 makes it clear that the best IoU result during the test, among all configurations tried, is 88.65% and was obtained by U-Net trained with total loss and adjusting the learning rate with the `ReduceLROnPlateau` callback. The second best result is 88.59% and was achieved by U-Net trained with Jaccard loss.

Finally, the time that a model takes to make a prediction was analyzed, that is, the time taken to output a segmentation mask for an input image. To achieve this goal, the Python module `time` was used. The results are documented in table 25 and plotted in figure 52. The image inference time for U-Net is between 0.063 and 0.085 seconds. Tiramisu is much slower and takes between 0.24 and 0.26 seconds to make a prediction.

Model	**IoU**
Tiramisu, Jaccard loss	64.43
Tiramisu, BCE loss	61.04
Tiramisu, BCE-Dice loss	67.60
Tiramisu, Focal Tversky loss	54.33
Tiramisu, Total loss	71.01
U-Net, Jaccard loss	88.59
U-Net, BCE loss	83.99
U-Net, BCE-Dice loss	86.39
U-Net, BCE-Focal loss	76.37
U-Net, Tversky loss	87.62
U-Net, Focal Tversky loss	87.49
U-Net, Tuning model	87.51
U-Net, Total loss	88.15
U-Net, Total loss, reduce LR callback	88.65

Table 24: IoU metric achieved during the testing step by the U-Net and Tiramisu models.

Model	**Loss function**	**Metric**	**Time** (s)
Tiramisu	Jaccard	IoU	0.257766
	BCE		0.235551
	BCE-Dice		0.239501
	Focal Tversky		0.250234
	Total		0.250732
U-Net	BCE	IoU	0.070521
	BCE-Dice		0.069886
	Jaccard		0.085801
	BCE-Focal		0.081481
	Tversky		0.063535
	Focal Tversky		0.080072
	Total		0.066226
	Total, reduce LR callback		0.068562
	Jaccard+tuning model		0.083304

Table 25: Image inference time for the different models.

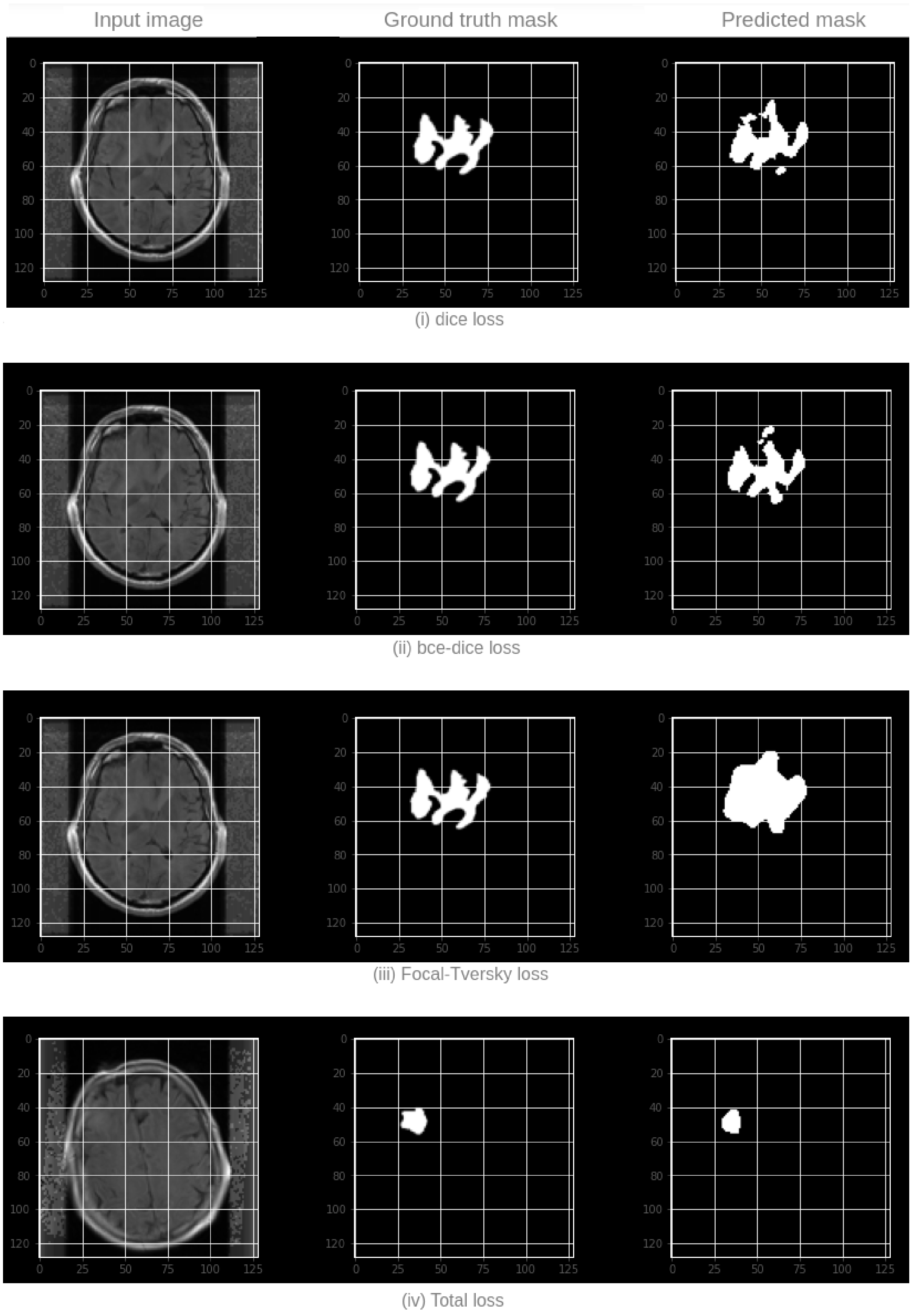


Figure 50: U-Net with different loss functions: four examples of prediction on test set.

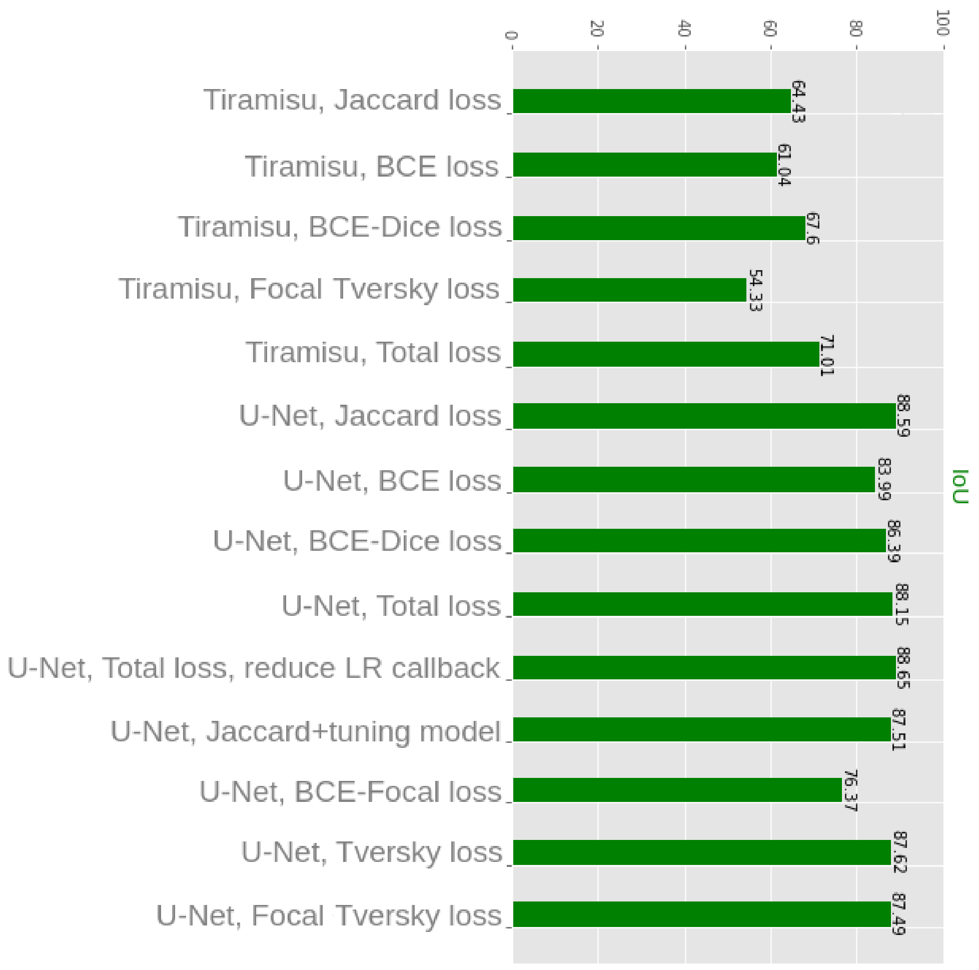


Figure 51: Results from the testing step for U-Net and Tiramisu models, trained with different loss functions.

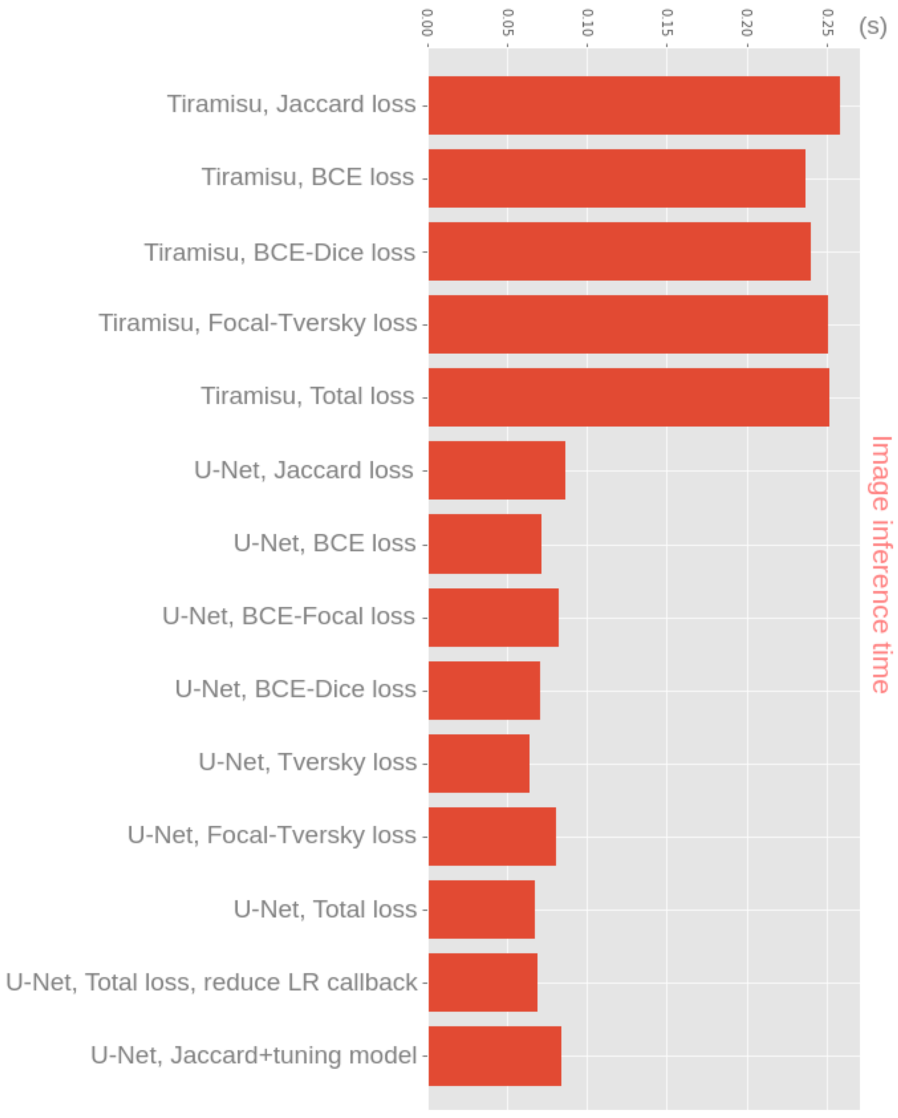


Figure 52: Image inference time for different models.

6

CONCLUSIONS AND FUTURE WORK

6.1 CONCLUSIONS

According to the initial objectives, the main contribution of this dissertation was the design and development of a predictive deep learning model that helps to detect tumors on the MRI images. This automatic help facilitates the tedious and error-prone task done by doctors and imaging specialists.

This kind of model also reduces the level of subjectivity that exists between different imaging specialists and clinicians when they have to analyze a certain image. Several models were developed to segment the aggressiveness pattern of tumors according to the cellular morphological architecture, both in the tumor front and inside it. An exhaustive analysis of segmentation techniques was conducted in the field of medical imaging, especially when applied to brain tumors, was conducted. Different methods were reviewed to improve the performance of tumor segmentation. The tumor pattern, the histology it acquires, its behavior, and how it affects the patient's health in terms of prognosis were analyzed in detail.

The available histological images require high computational costs for the process. Therefore, different methods were developed to allow building a base of MRI image data to be able to work on them and be compatible with the model's predictions. We generally do this process because in the future we will not have a problem with model training limitations and be able to train models with more complex architectures. Generally, U-Net and Tiramisu are models that have a lot of training parameters. So, we tried, since the beginning of the project, to reduce the computational cost at each stage.

With acquired knowledge about tumor aggressiveness patterns and with the supervision of experts in the field of tumor anatomy, each image was annotated with a tumor mask. These masks were used as the ground truth during the supervised training of deep learning models.

The selected data set was divided into training testing and validation data.

Two deep neural network architectures were selected to solve the tumor segmentation problem. The selected models, U-Net and Tiramisu, were trained in a supervised approach and evaluated. The interception over union (IoU) metric was selected to evaluate the models. An exhaustive study of the loss functions used in image segmentation problems was carried out, and most of them were implemented and tested with the two mentioned models.

A quantitative and qualitative analysis of the results obtained with each combination of model and loss function was performed. This allowed us to identify the trained model that gets the best performance on the segmentation task, the model that trains faster, makes inference faster, and occupies the least space on the disk. U-Net is the model that obtains the best result in all these aspects. The U-Net model, trained with total loss and adjusting the learning rate during training, got 88.65% IoU during the testing step, the best value among all trained models. After training the two models, U-Net and Tiramisu, we observed that the Tiramisu exhibits a lower IoU score, and apart from that, a particularity of the Tiramisu model is, that it has more training parameters, and when the number of training parameters increases, the computational cost also increases.

6.2 FUTURE WORK

There are several obstacles related to the segmentation of the image by computer vision that must be overcome to apply to an even wider range of areas than we have already obtained.

As explained before, this thesis tries to apply a novel method of Deep learning in the field of image segmentation, there is still a lot to do to improve the algorithm and the results and also to continue working to expand the scope of this project in the field of computer vision.

Overcome the obstacles related to semantic segmentation and may also take this algorithm and improve to apply to segmentation of the 3D images or videos in the future. on the other hand, using a more complete database with the supervision of experts in the field of medicine to increase the reliability of the model.

BIBLIOGRAPHY

American cancer society. https://www.cancer.org. Accessed on November 2022.

N. Abraham and N. Khan. A novel focal tversky loss function with improved attention u-net for lesion segmentation. In *IEEE International Symposium on Biomedical Imaging*, pages 683–687, 2019. URL https://arxiv.org/pdf/1810.07842.pdf.

American Institute For Medical & Biological Engineering AIMBE. Why biomedical engineering, Consulted in 2022. URL https://navigate.aimbe.org/?gclid=Cj0KCQjw8rT8BRCbARIsALWiOvS_z_u7XXbXFWNHaqVJN9Lp4vN1KcFR3leNA-9MfAyEu_mYxFFQjL4aAh4YEALw_wcB.

Jacques Aumont. *L'image*. France, Aubin Imprimeur, 1994.

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*, 39(12):2481 – 2495, 2017.

R. Bakshi, S. Ariyaratana, R. Benedict, and L. Jacobs. Fluid-attenuated inversion recovery magnetic resonance imaging detects cortical and juxtacortical multiple sclerosis lesions. *Archives of Neurology*, 58(5), 2001.

G. Bastarrika. Tomografía computarizada y práctica clínica. *Anales del Sistema Sanitario de Navarra*, 2007. URL http://scielo.isciii.es/scielo.php?script=sci_arttext&pid=S1137-66272007000300001.

Hans Belting. *Antropología de la Imagen*. KATS Editores, Argentina, 2007.

Adam Bohr and Kaveh Memarzadeh. *Artificial Intelligence in Healthcare*. Academic Press, Elsevier, 2020.

Jennifer Bresnick. Artificial intelligence in healthcare market to see 40% cagr surge, 2017. URL https://www.sciencedirect.com/science/article/pii/B9780128184387000010.

Jason Brownlee. *Machine Lerning Mastery*. 2019. URL https://machinelearningmastery.com/understand-the-dynamics-of-learning-rate-on-deep-learning-neural-networks/.

L. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. Yuille. Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2017. URL https://arxiv.org/pdf/1606.00915.pdf.

Michal Drozdzal, Eugene Vorontsov, Gabriel Chartrand, Samuel Kadoury, and Chris Pal. *The Importance of Skip Connections in Biomedical Image Segmentation*. 2016. URL https://arxiv.org/abs/1608.04117.

Guillaume Duchenne. Mécanisme de la physionomie humaine, 1862. URL https://gallica.bnf.fr/ark:/12148/bpt6k5699210s.texteImage.

V. Dumoulin and F. Visin. A guide to convolution arithmetic for deep learning, 2018. URL https://arxiv.org/pdf/1603.07285.pdf.

Raquel Raudales Díaz. Imágenes diagnósticas: Conceptos y generalidades. *Revista de la Facultad de Ciencias Médicas*, 2014. URL http://www.bvs.hn/RFCM/pdf/2014/pdf/RFCMVol11-1-2014-6.pdf.

Mariana Fontainhas. *Brain Semantic Segmentation: A Deep Learning approach in Human and Rat MRI studies*. 2018.

U.S. Food and Drug. U.s. food and drug, 2017. URL https://www.fda.gov/.

Center for Drug Evaluation, Research Center for Biologics Evaluation, and Research. U.s. food and drug, 2020. URL https://www.fda.gov/regulatory-information/search-fda-guidance-documents/clinical-trial-imaging-endpoint-process-standards-guidance-industry.

L. Foulds. The histological analysis of tumours: a critical review. *The American Journal of Cancer*, 38(1):1–24, 1940.

Marissa Gómez. Historia de la imagen digital, 2013. URL https://interartive.org/2017/04/historias-de-la-imagen-digital-marisa-gomez.

A Hack. *Semantic Segmentation*. 2019. URL https://medium.com/hackabit/semantic-segmentation-8f2900eff5c8.

D. Han, J. Kim, and J. Kim. Deep pyramidal residual networks. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, page 6307–6315, 2017. URL https://arxiv.org/pdf/1610.02915.pdf.

K. He, G. Gkioxari, P. Dollár, and R. Girshick. Mask r-cnn, 2017. URL https://arxiv.org/pdf/1703.06870.pdf.

Jeremy Jordan. An overview of semantic image segmentation, 2018a. URL https://www.jeremyjordan.me/semantic-segmentation/.

Jeremy Jordan. Common architectures in convolutional neural networks, 2018b. URL https://www.jeremyjordan.me/convnet-architectures/.

Simon Jégou, Michal Drozdzal, David Vazquez, Adriana Romero, and Yoshua Bengio. *The One Hundred Layers Tiramisu: Fully Convolutional DenseNets for Semantic Segmentation*. 2016. URL https://arxiv.org/abs/1611.09326.

Simon Jégou, Michal Drozdzal, David Vazquez, Adriana Romero, and Yoshua Bengio. *The One Hundred Layers Tiramisu: Fully Convolutional DenseNets for Semantic Segmentation*. 2017.

Samiya Luthfia Khaleel. The evolution of medical imaging in clinical research, 2017. URL https://www.clinicalleader.com/doc/the-evolution-of-medical-imaging-in-clinical-research-0001.

A. Khan, A. Sohail, U. Zahoora, and A. Qureshi. A survey of the recent architectures of deep convolutional neural networks. *Artificial Intelligence Review*, 53:5455–5516, 2020.

Sarthak khandelwal. Focal loss: An efficient way of handling class imbalance, 2020. URL https://medium.com/swlh/focal-loss-an-efficient-way-of-handling-class-imbalance-4855ae1db4cb.

Timo Kohlberger, Michal Sofka, Jingdan Zhang, Neil Birkbeck, Jens Wetzl, Jens Kaftan, Jerome Declerck, and Kevin Zhou. Automatic multi-organ segmentation using learning-based segmentation and level set optimization. In *Medical Image Computing and Computer-Assisted Intervention*, pages 28–36. Springer, 2011a. ISBN 978-3-642-23625-9. doi: 10.1007/978-3-642-23626-6_42. URL https://doi.org/10.1007/978-3-642-23626-6_42.

Timo Kohlberger, Brendt Wohlberg, Omid Masoud, Diana Mateus, Ender Konukoglu, Lily Ng, Martin Rajchl, Jeff Chuang, Guillaume Vaillant, Chi-Wing Fu, et al. Automatic multi-organ segmentation using learning-based segmentation and level set optimization. *Springer*, 6893, 2011b.

Y LeCun, LD Jackel, and L Bottou. *Learning algorithms for classification: a comparison on handwritten digit recognition. Neural Netw Stat Mech Perspect*. 1995.

M. Lin and Q. Chen and. S Yan. Network in network, 2013. URL https://doi.org/10.1109/asru.2015.7404828.

T. Lin, P. Dollár, R. Girshick, K. He an B. Hariharan, and S. Belongie. Feature pyramid networks for object detection, 2016. URL https://arxiv.org/pdf/1612.03144.pdf.

T. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár. *Focal Loss for Dense Object Detection*. 2017. URL https://arxiv.org/pdf/1708.02002.pdf.

Xiaoqing Liu, Kunlun Gao, Bo Liu, Chengwei Pan, Kongming Liang, Lifeng Yan, Jiechao Ma, Fujin He, Shu Zhang, and Siyuan Pan. *Advances in Deep Learning-Based Medical Image Analysis*. 2021. URL https://spj.sciencemag.org/journals/hds/2021/8786793/.

J. Long, E. Shelhamer, , and T. Darrell. Fully convolutional networks for semantic segmentation. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 3431—–3440, 2015a.

J. Long, E. Shelhamer, and T. Darrell. Fully convolutional networks for semantic segmentation. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015b. URL https://arxiv.org/pdf/1411.4038.pdf.

Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3431–3440, 2012.

J. Lucas. What is biomedical engineering?, 2014. URL https://www.livescience.com/48001-biomedical-engineering.html.

Market and Market. Clinical trial imaging market, 2016. URL https://www.marketsandmarkets.com/Market-Reports/clinical-trials-imaging-market-30446624.html.

S. Minaee, Y. Boykov, F. Porikli, A. Plaza, N. Kehtarnavaz, and D. Terzopoulos. Image segmentation using deep learning: A survey, 2020. URL https://arxiv.org/pdf/2001.05566.pdf.

Edwin D. Murphy. *Particularity of Tumors*, chapter 27. Dover Publications, 2007. URL http://www.informatics.jax.org/greenbook/chapters/chapter27.shtml.

Yurii Nesterov. A method for unconstrained convex minimization problem with the rate of convergence $o(1/k^2)$. *Doklady AN USSR*, 269:543–547, 1983.

Isabel Marília Peres. *Fotografia Médica*, chapter 5. Edições 70, 2014.

Martínez Rodríguez, Jairo Alejandro, and Vitola Oyaga. Fundamentos teórico-prácticos del ultrasonido. *Tecnura*, 2007. URL https://www.redalyc.org/pdf/2570/257021012001.pdf.

Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In Nassir Navab, Joachim Hornegger, William Wells, and Alejandro Frangi, editors, *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, volume 9351 of *Lecture Notes in Computer Science*. Springer, 2015. ISBN 978-3-319-24573-7. doi: 10.1007/978-3-319-24574-4_28. URL https://doi.org/10.1007/978-3-319-24574-4_28.

Sebastian Ruder. An overview of gradient descent optimization algorithms, 2016. URL https://arxiv.org/pdf/1609.04747.pdf.

Beeren Sahu. *The Evolution of Deeplab for Semantic Segmentation*. 2019. URL https://towardsdatascience.com/the-evolution-of-deeplab-for-semantic-segmentation-95082b025571.

S. Salehi, D. Erdogmus, and A. Gholipour. Tversky loss function for image segmentation using 3d fully convolutional deep networks. In *Machine Learning in Medical Imaging*, page 379–387, 2017. URL https://arxiv.org/abs/1706.05721.

Jay Selig. What is machine learning? a definition., 2020. URL https://www.expert.ai/blog/machine-learning-definition/#:~:text=Machine%20learning%20is%20an%20application,it%20to%20learn%20for%20themselves.

K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations (ICLR)*, 2015. URL https://arxiv.org/pdf/1409.1556.pdf.

Z. Sobhaninia, S. Rezaei, A. Noroozi, M. Ahmadi, H. Zarrabi, N. Karimi, A. Emami, and S. Samavi. Brain tumor segmentation using deep learning by type specific sorting of images, 2017. URL https://arxiv.org/pdf/1809.07786.pdf.

Jan Erik Solem. *Programming Computer Vision with Python*. O'Reilly Media, 2012.

Perry Sprawls. *Magnetic Resonance Imaging: Principles, Methods, and Techniques*. Medical Physics Publishing, Madison, Wisconsin, 2000. URL http://www.sprawls.org/resources/books/Sprawls%20Magnetic%20Resonance%20Imaging%20PMT%20%20.pdf.

C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, page 1–9, 2015.

Lydia Sánchez. Image, 2009. URL http://glossarium.bitrum.unileon.es/Home/imagen.

Robin Vinod. *Dealing with class imbalanced image datasets using the Focal Tversky Loss*. 2020. URL https://towardsdatascience.com/dealing-with-class-imbalanced-image-datasets-1cbd17de76b5.

Rupert Allan Willis. *Pathology of tumours*. Buttenvorth & Co., 1960.

Hang Zhang, Kristin Dana, Jianping Shi, Zhongyue Zhang, Xiaogang Wang, Ambrish Tyagi, and Amit Agrawal. Context encoding for semantic segmentation, 2018.

Ping Zheng and Wenbo Guo. Brain tumour segmentation based on an improved u-net, 18 November 2022. URL https://bmcmedimaging.biomedcentral.com/articles/10.1186/s12880-022-00931-1f.

Liad Pollak Zuckerman. *Efficient method for running Fully Convolutional Networks (FCNs)*. 2019. URL https://towardsdatascience.com/efficient-method-for-running-fully-convolutional-networks-fcns-3174dc6a692b.

7

DEVELOPED PYTHON CODE

This appendix contains most of the code developed during the present dissertation, as well as a compilation of results obtained and graphs of the architecture of the developed models.

```
import os
import random
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

plt.style.use("ggplot")
%matplotlib inline

import cv2
from tqdm import tqdm
from tqdm import tqdm_notebook, tnrange
from glob import glob
from itertools import chain
from skimage.io import imread, imshow, concatenate_images
from skimage.transform import resize
from skimage.morphology import label
from sklearn.model_selection import train_test_split

import tensorflow as tf
from skimage.color import rgb2gray
from tensorflow.keras import Input
from tensorflow.keras.models import Model, load_model, save_model
from tensorflow.keras.layers import Input, Activation, BatchNormalization,
     Dropout, Lambda, Conv2D, Conv2DTranspose, MaxPooling2D, concatenate
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint

from tensorflow.keras import backend as K
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint
```

Listing 7.1: Import the necessary libraries.

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from   sklearn.model_selection import train_test_split
from   keras.callbacks import EarlyStopping, ModelCheckpoint,ReduceLROnPlateau

X_all = np.load('bd_npy/x_BD_original_image.npy')
y_all = np.load('bd_npy/y_BD_original_image.npy')
#Data Visualization
def count_labels(y_all):
    how_many_0 = len(np.where(y_all==0)[0])
    how_many_1 = len(np.where(y_all==1)[0])

    print('#Tumor:',how_many_0)
    print('#Non Tumor:',how_many_1)

    return how_many_0,how_many_1
```

Listing 7.2: Python code for loading the dataset into memory.

```python
image_number = random.randint(0, len(X_train))
plt.figure(figsize=(12, 6))
plt.subplot(121)
plt.imshow(np.reshape(X_train[image_number], (128, 128)), cmap='gray')
plt.subplot(122)
plt.imshow(np.reshape(y_train[image_number], (128, 128)), cmap='gray')
plt.show()
```

Listing 7.3: Python code to visualize the effect of data normalization.

```python
print(X_train.shape)
print(y_train.shape)

(3143, 128, 128, 1)
(3143, 128, 128, 1
```

Listing 7.4: Print the dataset shape.

```python
rows,cols=3,3
fig=plt.figure(figsize=(10,10))
for i in range(1,rows*cols+1):
    fig.add_subplot(rows,cols,i)
    plt.imshow(image_dataset[i])
    plt.imshow(mask_dataset[i],alpha=0.4)
plt.show()
```

Listing 7.5: Python code to visualize a grid of 3x3 images and 3x3 masks from the dataset.

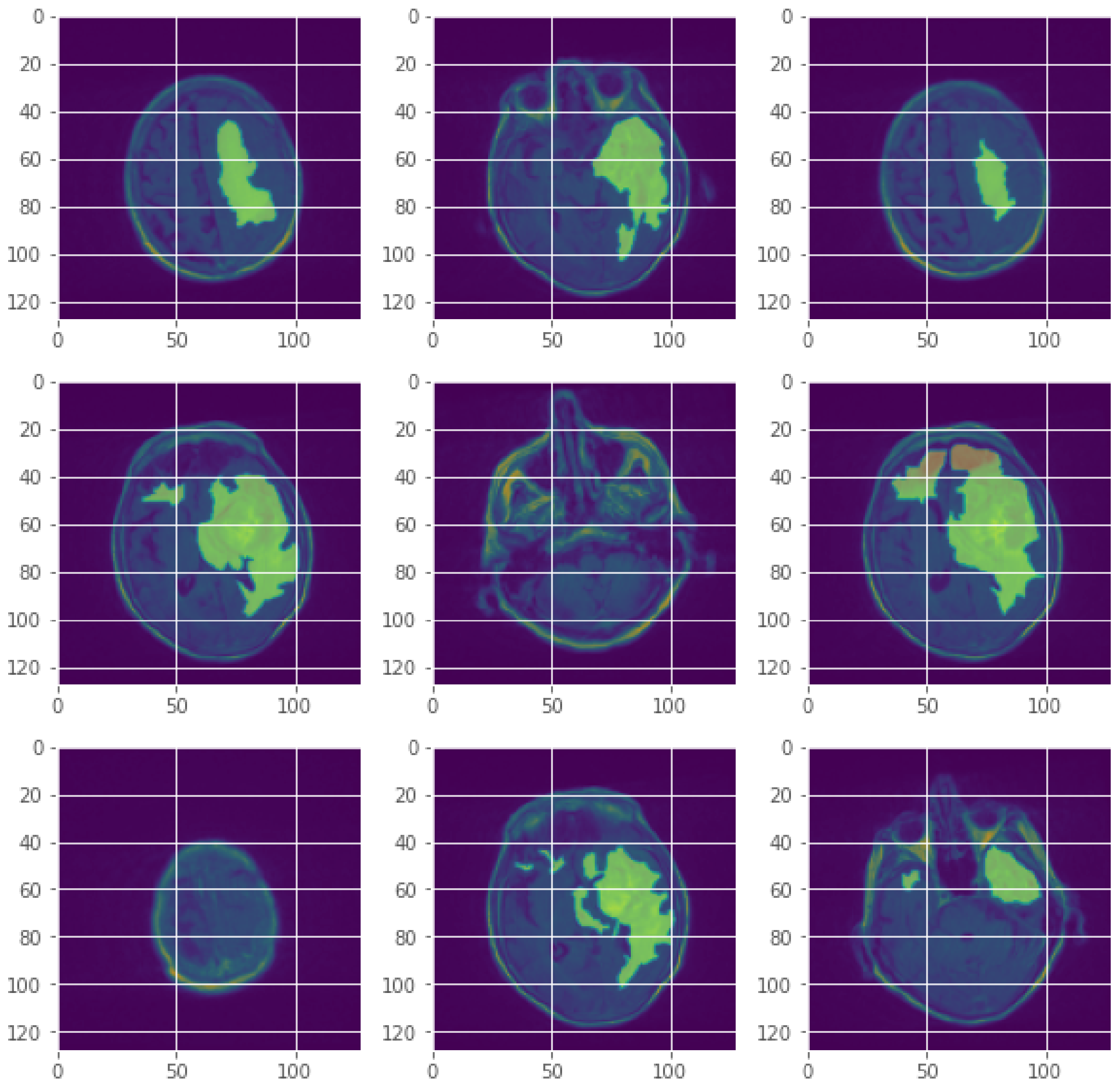


Figure 53: Visualization of 3x3 images from the dataset.

```python
from keras.losses import binary_crossentropy
import keras.backend as K
import tensorflow as tf

epsilon = 1e-5
smooth  = 0.00001

def lovasz_hinge_one(y_true, y_pred):
    logit = y_true
    truth = y_pred
    m = truth.detach()
    m = m*(margin[1]-margin[0])+margin[0]

    truth = truth.float()
    sign  = 2. * truth - 1.
    hinge = (m - logit * sign)
    hinge, permutation = torch.sort(hinge, dim=0, descending=True)
    hinge = F.relu(hinge)

    truth = truth[permutation.data]
    gradient = compute_lovasz_gradient(truth)

    loss = torch.dot(hinge, gradient)
    return loss

def dsc(y_true, y_pred):   # Dice's coefficient
    smooth =0.00001
    y_true_f = K.flatten(y_true)
    y_pred_f = K.flatten(y_pred)
    intersection = K.sum(y_true_f * y_pred_f)
    score = ((2. * intersection + smooth) / (K.sum(y_true_f) + K.sum(y_pred_f) + smooth))
    return score

def dice_loss(y_true, y_pred):
    loss = 1 - dsc(y_true, y_pred)
    return loss

def bce_dice_loss(y_true, y_pred):
    loss = binary_crossentropy(y_true, y_pred) + dice_loss(y_true, y_pred)
    return loss
```

Listing 7.6: Python code to implement loss functions (part 1).

```python

def confusion(y_true, y_pred):
    smooth=0.00001
    y_pred_pos = K.clip(y_pred, 0, 1)
    y_pred_neg = 1 - y_pred_pos
    y_pos = K.clip(y_true, 0, 1)
    y_neg = 1 - y_pos
    tp = K.sum(y_pos * y_pred_pos)
    fp = K.sum(y_neg * y_pred_pos)
    fn = K.sum(y_pos * y_pred_neg)
    prec = (tp + smooth)/(tp+fp+smooth)
    recall = (tp+smooth)/(tp+fn+smooth)
    return prec, recall

def tversky(y_true, y_pred):
    y_true_pos = K.flatten(y_true)
    y_pred_pos = K.flatten(y_pred)
    true_pos = K.sum(y_true_pos * y_pred_pos)
    false_neg = K.sum(y_true_pos * (1-y_pred_pos))
    false_pos = K.sum((1-y_true_pos)*y_pred_pos)
    alpha = 0.7
    return (true_pos + smooth)/(true_pos + alpha*false_neg + (1-alpha)*false_pos + smooth)

def tversky_loss(y_true, y_pred):
    return 1 - tversky(y_true,y_pred)

def focal_tversky(y_true,y_pred):
    pt_1 = tversky(y_true, y_pred)
    gamma = 0.75
    return K.pow((1-pt_1), gamma)

def lovasz(y_true, y_pred):
    beta=0.5
    y_true = tf.cast(y_true, tf.float32)
    y_pred = tf.math.sigmoid(y_pred)
     numerator = y_true * y_pred
    denominator = y_true * y_pred + beta * (1 - y_true) * y_pred +
                 (1 - beta) * y_true * (1 - y_pred)

    return 1 - tf.reduce_sum(numerator) / tf.reduce_sum(denominator)

def total_loss(y_true, y_pred):
    loss = binary_crossentropy(y_true, y_pred) + dice_loss(y_true, y_pred) +
           FocalTverskyLoss(y_true, y_pred, smooth=1e-6)+lovasz(y_true, y_pred)
    return loss
```

Listing 7.7: Python code to implement loss functions (part 2).

```python
def UNet(IMG_HEIGHT, IMG_WIDTH, IMG_CHANNELS):
    inputs = tf.keras.layers.Input((IMG_HEIGHT, IMG_WIDTH, IMG_CHANNELS))
    s = inputs
    # Contraction path
    c1 = tf.keras.layers.Conv2D(16, (3, 3), activation='relu', kernel_initializer='he_normal',
         padding='same')(s)
    c1 = tf.keras.layers.Dropout(0.1)(c1)
    c1 = tf.keras.layers.Conv2D(16, (3, 3), activation='relu', kernel_initializer='he_normal',
         padding='same')(c1)
    p1 = tf.keras.layers.MaxPooling2D((2, 2))(c1)

    c2 = tf.keras.layers.Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_normal',
         padding='same')(p1)
    c2 = tf.keras.layers.Dropout(0.1)(c2)
    c2 = tf.keras.layers.Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_normal',
         padding='same')(c2)
    p2 = tf.keras.layers.MaxPooling2D((2, 2))(c2)

    c3 = tf.keras.layers.Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_normal',
         padding='same')(p2)
    c3 = tf.keras.layers.Dropout(0.2)(c3)
    c3 = tf.keras.layers.Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_normal',
         padding='same')(c3)
    p3 = tf.keras.layers.MaxPooling2D((2, 2))(c3)

    c4 = tf.keras.layers.Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_normal',
         padding='same')(p3)
    c4 = tf.keras.layers.Dropout(0.2)(c4)
    c4 = tf.keras.layers.Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_normal',
         padding='same')(c4)
    p4 = tf.keras.layers.MaxPooling2D(pool_size=(2, 2))(c4)

    c5 = tf.keras.layers.Conv2D(256, (3, 3), activation='relu', kernel_initializer='he_normal',
         padding='same')(p4)
    c5 = tf.keras.layers.Dropout(0.3)(c5)
    c5 = tf.keras.layers.Conv2D(256, (3, 3), activation='relu', kernel_initializer='he_normal',
         padding='same')(c5)

    # Expansion path
    u6 = tf.keras.layers.Conv2DTranspose(128, (2, 2), strides=(2, 2), padding='same')(c5)
    u6 = tf.keras.layers.concatenate([u6, c4])
    c6 = tf.keras.layers.Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_normal',
         padding='same')(u6)
    c6 = tf.keras.layers.Dropout(0.2)(c6)
    c6 = tf.keras.layers.Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_normal',
         padding='same')(c6)
```

Listing 7.8: Python code that implements the U-Net model (part 1).

```
    u7 = tf.keras.layers.Conv2DTranspose(64, (2, 2), strides=(2, 2), padding='same')(c6)
    u7 = tf.keras.layers.concatenate([u7, c3])
    c7 = tf.keras.layers.Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_normal',
         padding='same')(u7)
    c7 = tf.keras.layers.Dropout(0.2)(c7)
    c7 = tf.keras.layers.Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_normal',
         padding='same')(c7)

    u8 = tf.keras.layers.Conv2DTranspose(32, (2, 2), strides=(2, 2), padding='same')(c7)
    u8 = tf.keras.layers.concatenate([u8, c2])
    c8 = tf.keras.layers.Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_normal',
         padding='same')(u8)
    c8 = tf.keras.layers.Dropout(0.1)(c8)
    c8 = tf.keras.layers.Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_normal',
         padding='same')(c8)

    u9 = tf.keras.layers.Conv2DTranspose(16, (2, 2), strides=(2, 2), padding='same')(c8)
    u9 = tf.keras.layers.concatenate([u9, c1], axis=3)
    c9 = tf.keras.layers.Conv2D(16, (3, 3), activation='relu', kernel_initializer='he_normal',
         padding='same')(u9)
    c9 = tf.keras.layers.Dropout(0.1)(c9)
    c9 = tf.keras.layers.Conv2D(16, (3, 3), activation='relu', kernel_initializer='he_normal',
         padding='same')(c9)

    outputs = tf.keras.layers.Conv2D(1, (1, 1), activation='sigmoid')(c9)

    model = tf.keras.Model(inputs=[inputs], outputs=[outputs])

    return model
```

Listing 7.9: Python code that implements the U-Net model (part 2).

```
Model: "Unet Model Jaccard"
Layer (type)                Output Shape         Params     Connected to
input_9 (InputLayer)        [(None, 128, 128, 3) 0
lambda_8 (Lambda)           (None, 128, 128, 3)  0           input_9[0][0]
conv2d_152 (Conv2D)         (None, 128, 128, 16) 448         lambda_8[0][0]
dropout_72 (Dropout)        (None, 128, 128, 16) 0           conv2d_152[0][0]
conv2d_153 (Conv2D)         (None, 128, 128, 16) 2320        dropout_72[0][0]
max_pooling2d_32 (MaxPooling2D) (None, 64, 64, 16)   0           conv2d_153[0][0]
conv2d_154 (Conv2D)         (None, 64, 64, 32)   4640        max_pooling2d_32[0][0]
dropout_73 (Dropout)        (None, 64, 64, 32)   0           conv2d_154[0][0]
conv2d_155 (Conv2D)         (None, 64, 64, 32)   9248        dropout_73[0][0]
max_pooling2d_33 (MaxPooling2D) (None, 32, 32, 32)   0           conv2d_155[0][0]
```

Listing 7.10: Summary of the U-Net model (part 1).

```
conv2d_156 (Conv2D)            (None, 32, 32, 64)   18496      max_pooling2d_33[0][0]
dropout_74 (Dropout)           (None, 32, 32, 64)   0          conv2d_156[0][0]
conv2d_157 (Conv2D)            (None, 32, 32, 64)   36928      dropout_74[0][0]
max_pooling2d_34 (MaxPooling2D) (None, 16, 16, 64)   0          conv2d_157[0][0]
conv2d_158 (Conv2D)            (None, 16, 16, 128)  73856      max_pooling2d_34[0][0]
dropout_75 (Dropout)           (None, 16, 16, 128)  0          conv2d_158[0][0]
conv2d_159 (Conv2D)            (None, 16, 16, 128)  147584     dropout_75[0][0]
max_pooling2d_35 (MaxPooling2D) (None, 8, 8, 128)    0          conv2d_159[0][0]
conv2d_160 (Conv2D)            (None, 8, 8, 256)    295168     max_pooling2d_35[0][0]
dropout_76 (Dropout)           (None, 8, 8, 256)    0          conv2d_160[0][0]
conv2d_161 (Conv2D)            (None, 8, 8, 256)    590080     dropout_76[0][0]
conv2d_transpose_32 (Conv2DTr.) (None, 16, 16, 128)  131200     conv2d_161[0][0]
concatenate_32 (Concatenate)    (None, 16, 16, 256)  0          conv2d_transpose_32[0][0]
                                                                conv2d_159[0][0]
conv2d_162 (Conv2D)            (None, 16, 16, 128)  295040     concatenate_32[0][0]
dropout_77 (Dropout)           (None, 16, 16, 128)  0          conv2d_162[0][0]
conv2d_163 (Conv2D)            (None, 16, 16, 128)  147584     dropout_77[0][0]
conv2d_transpose_33 (Conv2DTr.) (None, 32, 32, 64)   32832      conv2d_163[0][0]
concatenate_33 (Concatenate)    (None, 32, 32, 128)  0          conv2d_transpose_33[0][0]
                                                                conv2d_157[0][0]
conv2d_164 (Conv2D)            (None, 32, 32, 64)   73792      concatenate_33[0][0]
dropout_78 (Dropout)           (None, 32, 32, 64)   0          conv2d_164[0][0]
conv2d_165 (Conv2D)            (None, 32, 32, 64)   36928      dropout_78[0][0]
conv2d_transpose_34 (Conv2DTr.) (None, 64, 64, 32)   8224       conv2d_165[0][0]
concatenate_34 (Concatenate)    (None, 64, 64, 64)   0          conv2d_transpose_34[0][0]
                                                                conv2d_155[0][0]
conv2d_166 (Conv2D)            (None, 64, 64, 32)   18464      concatenate_34[0][0]
dropout_79 (Dropout)           (None, 64, 64, 32)   0          conv2d_166[0][0]
conv2d_167 (Conv2D)            (None, 64, 64, 32)   9248       dropout_79[0][0]
conv2d_transpose_35 (Conv2DTr.) (None, 128, 128, 16) 2064       conv2d_167[0][0]
concatenate_35 (Concatenate)    (None, 128, 128, 32) 0          conv2d_transpose_35[0][0]
                                                                conv2d_153[0][0]
conv2d_168 (Conv2D)            (None, 128, 128, 16) 4624       concatenate_35[0][0]
dropout_80 (Dropout)           (None, 128, 128, 16) 0          conv2d_168[0][0]
conv2d_169 (Conv2D)            (None, 128, 128, 16) 2320       dropout_80[0][0]
conv2d_170 (Conv2D)            (None, 128, 128, 1)  17         conv2d_169[0][0]
Total params:         1,941,105
Trainable params:     1,941,105
Non-trainable params: 0
```

Listing 7.11: Summary of the U-Net model (part 2).

```
Model: "model_Tiramisu"
Layer (type)                    Output Shape         Params      Connected to
input_12 (InputLayer)           [(None, 128, 128, 3) 0
conv2d_367 (Conv2D)             (None, 128, 128, 48) 1344        input_12[0][0]
batch_normalization_192 (BNorm) (None, 128, 128, 48) 512         conv2d_367[0][0]
activation_194 (Activation)     (None, 128, 128, 48) 0           batch_normalization_192[0][0]
conv2d_368 (Conv2D)             (None, 128, 128, 16) 6928        activation_194[0][0]
dropout_273 (Dropout)           (None, 128, 128, 16) 0           conv2d_368[0][0]
concatenate_228 (Concatenate)   (None, 128, 128, 64) 0           dropout_273[0][0]
                                                                 conv2d_367[0][0]
batch_normalization_193 (BNorm) (None, 128, 128, 64) 512         concatenate_228[0][0]
activation_195 (Activation)     (None, 128, 128, 64) 0           batch_normalization_193[0][0]
conv2d_369 (Conv2D)             (None, 128, 128, 16) 9232        activation_195[0][0]
dropout_274 (Dropout)           (None, 128, 128, 16) 0           conv2d_369[0][0]
concatenate_229 (Concatenate)   (None, 128, 128, 80) 0           dropout_274[0][0]
                                                                 concatenate_228[0][0]
batch_normalization_194 (BNorm) (None, 128, 128, 80) 512         concatenate_229[0][0]
activation_196 (Activation)     (None, 128, 128, 80) 0           batch_normalization_194[0][0]
conv2d_370 (Conv2D)             (None, 128, 128, 16) 11536       activation_196[0][0]
dropout_275 (Dropout)           (None, 128, 128, 16) 0           conv2d_370[0][0]
concatenate_230 (Concatenate)   (None, 128, 128, 96) 0           dropout_275[0][0]
                                                                 concatenate_229[0][0]
batch_normalization_195 (BNorm) (None, 128, 128, 96) 512         concatenate_230[0][0]
activation_197 (Activation)     (None, 128, 128, 96) 0           batch_normalization_195[0][0]
conv2d_371 (Conv2D)             (None, 128, 128, 16) 13840       activation_197[0][0]
dropout_276 (Dropout)           (None, 128, 128, 16) 0           conv2d_371[0][0]
concatenate_231 (Concatenate)   (None, 128, 128, 112 0           dropout_276[0][0]
                                                                 concatenate_230[0][0]
batch_normalization_196 (BNorm) (None, 128, 128, 112 512         concatenate_231[0][0]
activation_198 (Activation)     (None, 128, 128, 112 0           batch_normalization_196[0][0]
conv2d_372 (Conv2D)             (None, 128, 128, 112 12656       activation_198[0][0]
dropout_277 (Dropout)           (None, 128, 128, 112 0           conv2d_372[0][0]
max_pooling2d_46 (MaxPooling2D) (None, 64, 64, 112)  0           dropout_277[0][0]
batch_normalization_197 (BNorm) (None, 64, 64, 112)  256         max_pooling2d_46[0][0]
activation_199 (Activation)     (None, 64, 64, 112)  0           batch_normalization_197[0][0]
conv2d_373 (Conv2D)             (None, 64, 64, 16)   16144       activation_199[0][0]
dropout_278 (Dropout)           (None, 64, 64, 16)   0           conv2d_373[0][0]
concatenate_232 (Concatenate)   (None, 64, 64, 128)  0           dropout_278[0][0]
                                                                 max_pooling2d_46[0][0]
batch_normalization_198 (BNorm) (None, 64, 64, 128)  256         concatenate_232[0][0]
activation_200 (Activation)     (None, 64, 64, 128)  0           batch_normalization_198[0][0]
conv2d_374 (Conv2D)             (None, 64, 64, 16)   18448       activation_200[0][0]
dropout_279 (Dropout)           (None, 64, 64, 16)   0           conv2d_374[0][0]
concatenate_233 (Concatenate)   (None, 64, 64, 144)  0           dropout_279[0][0]
                                                                 concatenate_232[0][0]
```

Listing 7.12: Summary of the Tiramisu model (part 1).

```
batch_normalization_199 (BNorm) (None, 64, 64, 144)  256        concatenate_233[0][0]
activation_201 (Activation)     (None, 64, 64, 144)  0          batch_normalization_199[0][0]
conv2d_375 (Conv2D)             (None, 64, 64, 16)   20752      activation_201[0][0]
dropout_280 (Dropout)           (None, 64, 64, 16)   0          conv2d_375[0][0]
concatenate_234 (Concatenate)   (None, 64, 64, 160)  0          dropout_280[0][0]
                                                                concatenate_233[0][0]
batch_normalization_200 (BNorm) (None, 64, 64, 160)  256        concatenate_234[0][0]
activation_202 (Activation)     (None, 64, 64, 160)  0          batch_normalization_200[0][0]
conv2d_376 (Conv2D)             (None, 64, 64, 16)   23056      activation_202[0][0]
dropout_281 (Dropout)           (None, 64, 64, 16)   0          conv2d_376[0][0]
concatenate_235 (Concatenate)   (None, 64, 64, 176)  0          dropout_281[0][0]
                                                                concatenate_234[0][0]
batch_normalization_201 (BNorm) (None, 64, 64, 176)  256        concatenate_235[0][0]
activation_203 (Activation)     (None, 64, 64, 176)  0          batch_normalization_201[0][0]
conv2d_377 (Conv2D)             (None, 64, 64, 16)   25360      activation_203[0][0]
dropout_282 (Dropout)           (None, 64, 64, 16)   0          conv2d_377[0][0]
concatenate_236 (Concatenate)   (None, 64, 64, 192)  0          dropout_282[0][0]
                                                                concatenate_235[0][0]
batch_normalization_202 (BNorm) (None, 64, 64, 192)  256        concatenate_236[0][0]
activation_204 (Activation)     (None, 64, 64, 192)  0          batch_normalization_202[0][0]
conv2d_378 (Conv2D)             (None, 64, 64, 192)  37056      activation_204[0][0]
dropout_283 (Dropout)           (None, 64, 64, 192)  0          conv2d_378[0][0]
max_pooling2d_47 (MaxPooling2D) (None, 32, 32, 192)  0          dropout_283[0][0]
batch_normalization_203 (BNorm) (None, 32, 32, 192)  128        max_pooling2d_47[0][0]
activation_205 (Activation)     (None, 32, 32, 192)  0          batch_normalization_203[0][0]
conv2d_379 (Conv2D)             (None, 32, 32, 16)   27664      activation_205[0][0]
dropout_284 (Dropout)           (None, 32, 32, 16)   0          conv2d_379[0][0]
concatenate_237 (Concatenate)   (None, 32, 32, 208)  0          dropout_284[0][0]
                                                                max_pooling2d_47[0][0]
batch_normalization_204 (BNorm) (None, 32, 32, 208)  128        concatenate_237[0][0]
activation_206 (Activation)     (None, 32, 32, 208)  0          batch_normalization_204[0][0]
conv2d_380 (Conv2D)             (None, 32, 32, 16)   29968      activation_206[0][0]
dropout_285 (Dropout)           (None, 32, 32, 16)   0          conv2d_380[0][0]
concatenate_238 (Concatenate)   (None, 32, 32, 224)  0          dropout_285[0][0]
                                                                concatenate_237[0][0]
batch_normalization_205 (BNorm) (None, 32, 32, 224)  128        concatenate_238[0][0]
activation_207 (Activation)     (None, 32, 32, 224)  0          batch_normalization_205[0][0]
conv2d_381 (Conv2D)             (None, 32, 32, 16)   32272      activation_207[0][0]
dropout_286 (Dropout)           (None, 32, 32, 16)   0          conv2d_381[0][0]
concatenate_239 (Concatenate)   (None, 32, 32, 240)  0          dropout_286[0][0]
                                                                concatenate_238[0][0]
batch_normalization_206 (BNorm) (None, 32, 32, 240)  128        concatenate_239[0][0]
activation_208 (Activation)     (None, 32, 32, 240)  0          batch_normalization_206[0][0]
conv2d_382 (Conv2D)             (None, 32, 32, 16)   34576      activation_208[0][0]
dropout_287 (Dropout)           (None, 32, 32, 16)   0          conv2d_382[0][0]
```

Listing 7.13: Summary of the Tiramisu model (part 2).

```
concatenate_240 (Concatenate)   (None, 32, 32, 256)  0           dropout_287[0][0]
                                                                 concatenate_239[0][0]
batch_normalization_207 (BNorm) (None, 32, 32, 256)  128         concatenate_240[0][0]
activation_209 (Activation)     (None, 32, 32, 256)  0           batch_normalization_207[0][0]
conv2d_383 (Conv2D)             (None, 32, 32, 16)   36880       activation_209[0][0]
dropout_288 (Dropout)           (None, 32, 32, 16)   0           conv2d_383[0][0]
concatenate_241 (Concatenate)   (None, 32, 32, 272)  0           dropout_288[0][0]
                                                                 concatenate_240[0][0]
batch_normalization_208 (BNorm) (None, 32, 32, 272)  128         concatenate_241[0][0]
activation_210 (Activation)     (None, 32, 32, 272)  0           batch_normalization_208[0][0]
conv2d_384 (Conv2D)             (None, 32, 32, 16)   39184       activation_210[0][0]
dropout_289 (Dropout)           (None, 32, 32, 16)   0           conv2d_384[0][0]
concatenate_242 (Concatenate)   (None, 32, 32, 288)  0           dropout_289[0][0]
                                                                 concatenate_241[0][0]
batch_normalization_209 (BNorm) (None, 32, 32, 288)  128         concatenate_242[0][0]
activation_211 (Activation)     (None, 32, 32, 288)  0           batch_normalization_209[0][0]
conv2d_385 (Conv2D)             (None, 32, 32, 16)   41488       activation_211[0][0]
dropout_290 (Dropout)           (None, 32, 32, 16)   0           conv2d_385[0][0]
concatenate_243 (Concatenate)   (None, 32, 32, 304)  0           dropout_290[0][0]
                                                                 concatenate_242[0][0]
batch_normalization_210 (BNorm) (None, 32, 32, 304)  128         concatenate_243[0][0]
activation_212 (Activation)     (None, 32, 32, 304)  0           batch_normalization_210[0][0]
conv2d_386 (Conv2D)             (None, 32, 32, 304)  92720       activation_212[0][0]
dropout_291 (Dropout)           (None, 32, 32, 304)  0           conv2d_386[0][0]
max_pooling2d_48 (MaxPooling2D) (None, 16, 16, 304)  0           dropout_291[0][0]
batch_normalization_211 (BNorm) (None, 16, 16, 304)  64          max_pooling2d_48[0][0]
activation_213 (Activation)     (None, 16, 16, 304)  0           batch_normalization_211[0][0]
conv2d_387 (Conv2D)             (None, 16, 16, 16)   43792       activation_213[0][0]
dropout_292 (Dropout)           (None, 16, 16, 16)   0           conv2d_387[0][0]
concatenate_244 (Concatenate)   (None, 16, 16, 320)  0           dropout_292[0][0]
                                                                 max_pooling2d_48[0][0]
batch_normalization_212 (BNorm) (None, 16, 16, 320)  64          concatenate_244[0][0]
activation_214 (Activation)     (None, 16, 16, 320)  0           batch_normalization_212[0][0]
conv2d_388 (Conv2D)             (None, 16, 16, 16)   46096       activation_214[0][0]
dropout_293 (Dropout)           (None, 16, 16, 16)   0           conv2d_388[0][0]
concatenate_245 (Concatenate)   (None, 16, 16, 336)  0           dropout_293[0][0]
                                                                 concatenate_244[0][0]
batch_normalization_213 (BNorm) (None, 16, 16, 336)  64          concatenate_245[0][0]
activation_215 (Activation)     (None, 16, 16, 336)  0           batch_normalization_213[0][0]
conv2d_389 (Conv2D)             (None, 16, 16, 16)   48400       activation_215[0][0]
dropout_294 (Dropout)           (None, 16, 16, 16)   0           conv2d_389[0][0]
concatenate_246 (Concatenate)   (None, 16, 16, 352)  0           dropout_294[0][0]
                                                                 concatenate_245[0][0]
batch_normalization_214 (BNorm) (None, 16, 16, 352)  64          concatenate_246[0][0]
activation_216 (Activation)     (None, 16, 16, 352)  0           batch_normalization_214[0][0]
conv2d_390 (Conv2D)             (None, 16, 16, 16)   50704       activation_216[0][0]
```

Listing 7.14: Summary of the Tiramisu model (part 3).

```
dropout_295 (Dropout)           (None, 16, 16, 16)   0           conv2d_390[0][0]
concatenate_247 (Concatenate)   (None, 16, 16, 368)  0           dropout_295[0][0]
                                                                 concatenate_246[0][0]
batch_normalization_215 (BNorm) (None, 16, 16, 368)  64          concatenate_247[0][0]
activation_217 (Activation)     (None, 16, 16, 368)  0           batch_normalization_215[0][0]
conv2d_391 (Conv2D)             (None, 16, 16, 16)   53008       activation_217[0][0]
dropout_296 (Dropout)           (None, 16, 16, 16)   0           conv2d_391[0][0]
concatenate_248 (Concatenate)   (None, 16, 16, 384)  0           dropout_296[0][0]
                                                                 concatenate_247[0][0]
batch_normalization_216 (BNorm) (None, 16, 16, 384)  64          concatenate_248[0][0]
activation_218 (Activation)     (None, 16, 16, 384)  0           batch_normalization_216[0][0]
conv2d_392 (Conv2D)             (None, 16, 16, 16)   55312       activation_218[0][0]
dropout_297 (Dropout)           (None, 16, 16, 16)   0           conv2d_392[0][0]
concatenate_249 (Concatenate)   (None, 16, 16, 400)  0           dropout_297[0][0]
                                                                 concatenate_248[0][0]
batch_normalization_217 (BNorm) (None, 16, 16, 400)  64          concatenate_249[0][0]
activation_219 (Activation)     (None, 16, 16, 400)  0           batch_normalization_217[0][0]
conv2d_393 (Conv2D)             (None, 16, 16, 16)   57616       activation_219[0][0]
dropout_298 (Dropout)           (None, 16, 16, 16)   0           conv2d_393[0][0]
concatenate_250 (Concatenate)   (None, 16, 16, 416)  0           dropout_298[0][0]
                                                                 concatenate_249[0][0]
batch_normalization_218 (BNorm) (None, 16, 16, 416)  64          concatenate_250[0][0]
activation_220 (Activation)     (None, 16, 16, 416)  0           batch_normalization_218[0][0]
conv2d_394 (Conv2D)             (None, 16, 16, 16)   59920       activation_220[0][0]
dropout_299 (Dropout)           (None, 16, 16, 16)   0           conv2d_394[0][0]
concatenate_251 (Concatenate)   (None, 16, 16, 432)  0           dropout_299[0][0]
                                                                 concatenate_250[0][0]
batch_normalization_219 (BNorm) (None, 16, 16, 432)  64          concatenate_251[0][0]
activation_221 (Activation)     (None, 16, 16, 432)  0           batch_normalization_219[0][0]
conv2d_395 (Conv2D)             (None, 16, 16, 16)   62224       activation_221[0][0]
dropout_300 (Dropout)           (None, 16, 16, 16)   0           conv2d_395[0][0]
concatenate_252 (Concatenate)   (None, 16, 16, 448)  0           dropout_300[0][0]
                                                                 concatenate_251[0][0]
batch_normalization_220 (BNorm) (None, 16, 16, 448)  64          concatenate_252[0][0]
activation_222 (Activation)     (None, 16, 16, 448)  0           batch_normalization_220[0][0]
conv2d_396 (Conv2D)             (None, 16, 16, 16)   64528       activation_222[0][0]
dropout_301 (Dropout)           (None, 16, 16, 16)   0           conv2d_396[0][0]
concatenate_253 (Concatenate)   (None, 16, 16, 464)  0           dropout_301[0][0]
                                                                 concatenate_252[0][0]
batch_normalization_221 (BNorm) (None, 16, 16, 464)  64          concatenate_253[0][0]
activation_223 (Activation)     (None, 16, 16, 464)  0           batch_normalization_221[0][0]
conv2d_397 (Conv2D)             (None, 16, 16, 464)  215760      activation_223[0][0]
dropout_302 (Dropout)           (None, 16, 16, 464)  0           conv2d_397[0][0]
max_pooling2d_49 (MaxPooling2D) (None, 8, 8, 464)    0           dropout_302[0][0]
batch_normalization_222 (BNorm) (None, 8, 8, 464)    32          max_pooling2d_49[0][0]
activation_224 (Activation)     (None, 8, 8, 464)    0           batch_normalization_222[0][0]
```

Listing 7.15: Summary of the Tiramisu model (part 4).

```
conv2d_398 (Conv2D)             (None, 8, 8, 16)     66832       activation_224[0][0]
dropout_303 (Dropout)           (None, 8, 8, 16)     0           conv2d_398[0][0]
concatenate_254 (Concatenate)   (None, 8, 8, 480)    0           dropout_303[0][0]
                                                                 max_pooling2d_49[0][0]
batch_normalization_223 (BNorm) (None, 8, 8, 480)    32          concatenate_254[0][0]
activation_225 (Activation)     (None, 8, 8, 480)    0           batch_normalization_223[0][0]
conv2d_399 (Conv2D)             (None, 8, 8, 16)     69136       activation_225[0][0]
dropout_304 (Dropout)           (None, 8, 8, 16)     0           conv2d_399[0][0]
concatenate_255 (Concatenate)   (None, 8, 8, 496)    0           dropout_304[0][0]
                                                                 concatenate_254[0][0]
batch_normalization_224 (BNorm) (None, 8, 8, 496)    32          concatenate_255[0][0]
activation_226 (Activation)     (None, 8, 8, 496)    0           batch_normalization_224[0][0]
conv2d_400 (Conv2D)             (None, 8, 8, 16)     71440       activation_226[0][0]
dropout_305 (Dropout)           (None, 8, 8, 16)     0           conv2d_400[0][0]
concatenate_256 (Concatenate)   (None, 8, 8, 512)    0           dropout_305[0][0]
                                                                 concatenate_255[0][0]
batch_normalization_225 (BNorm) (None, 8, 8, 512)    32          concatenate_256[0][0]
activation_227 (Activation)     (None, 8, 8, 512)    0           batch_normalization_225[0][0]
conv2d_401 (Conv2D)             (None, 8, 8, 16)     73744       activation_227[0][0]
dropout_306 (Dropout)           (None, 8, 8, 16)     0           conv2d_401[0][0]
concatenate_257 (Concatenate)   (None, 8, 8, 528)    0           dropout_306[0][0]
                                                                 concatenate_256[0][0]
batch_normalization_226 (BNorm) (None, 8, 8, 528)    32          concatenate_257[0][0]
activation_228 (Activation)     (None, 8, 8, 528)    0           batch_normalization_226[0][0]
conv2d_402 (Conv2D)             (None, 8, 8, 16)     76048       activation_228[0][0]
dropout_307 (Dropout)           (None, 8, 8, 16)     0           conv2d_402[0][0]
concatenate_258 (Concatenate)   (None, 8, 8, 544)    0           dropout_307[0][0]
                                                                 concatenate_257[0][0]
batch_normalization_227 (BNorm) (None, 8, 8, 544)    32          concatenate_258[0][0]
activation_229 (Activation)     (None, 8, 8, 544)    0           batch_normalization_227[0][0]
conv2d_403 (Conv2D)             (None, 8, 8, 16)     78352       activation_229[0][0]
dropout_308 (Dropout)           (None, 8, 8, 16)     0           conv2d_403[0][0]
concatenate_259 (Concatenate)   (None, 8, 8, 560)    0           dropout_308[0][0]
                                                                 concatenate_258[0][0]
batch_normalization_228 (BNorm) (None, 8, 8, 560)    32          concatenate_259[0][0]
activation_230 (Activation)     (None, 8, 8, 560)    0           batch_normalization_228[0][0]
conv2d_404 (Conv2D)             (None, 8, 8, 16)     80656       activation_230[0][0]
dropout_309 (Dropout)           (None, 8, 8, 16)     0           conv2d_404[0][0]
concatenate_260 (Concatenate)   (None, 8, 8, 576)    0           dropout_309[0][0]
                                                                 concatenate_259[0][0]
batch_normalization_229 (BNorm) (None, 8, 8, 576)    32          concatenate_260[0][0]
activation_231 (Activation)     (None, 8, 8, 576)    0           batch_normalization_229[0][0]
conv2d_405 (Conv2D)             (None, 8, 8, 16)     82960       activation_231[0][0]
dropout_310 (Dropout)           (None, 8, 8, 16)     0           conv2d_405[0][0]
concatenate_261 (Concatenate)   (None, 8, 8, 592)    0           dropout_310[0][0]
                                                                 concatenate_260[0][0]
```

Listing 7.16: Summary of the Tiramisu model (part 5).

```
batch_normalization_230 (BNorm) (None, 8, 8, 592)    32          concatenate_261[0][0]
activation_232 (Activation)     (None, 8, 8, 592)    0           batch_normalization_230[0][0]
conv2d_406 (Conv2D)             (None, 8, 8, 16)     85264       activation_232[0][0]
dropout_311 (Dropout)           (None, 8, 8, 16)     0           conv2d_406[0][0]
concatenate_262 (Concatenate)   (None, 8, 8, 608)    0           dropout_311[0][0]
                                                                 concatenate_261[0][0]
batch_normalization_231 (BNorm) (None, 8, 8, 608)    32          concatenate_262[0][0]
activation_233 (Activation)     (None, 8, 8, 608)    0           batch_normalization_231[0][0]
conv2d_407 (Conv2D)             (None, 8, 8, 16)     87568       activation_233[0][0]
dropout_312 (Dropout)           (None, 8, 8, 16)     0           conv2d_407[0][0]
concatenate_263 (Concatenate)   (None, 8, 8, 624)    0           dropout_312[0][0]
                                                                 concatenate_262[0][0]
batch_normalization_232 (BNorm) (None, 8, 8, 624)    32          concatenate_263[0][0]
activation_234 (Activation)     (None, 8, 8, 624)    0           batch_normalization_232[0][0]
conv2d_408 (Conv2D)             (None, 8, 8, 16)     89872       activation_234[0][0]
dropout_313 (Dropout)           (None, 8, 8, 16)     0           conv2d_408[0][0]
concatenate_264 (Concatenate)   (None, 8, 8, 640)    0           dropout_313[0][0]
                                                                 concatenate_263[0][0]
batch_normalization_233 (BNorm) (None, 8, 8, 640)    32          concatenate_264[0][0]
activation_235 (Activation)     (None, 8, 8, 640)    0           batch_normalization_233[0][0]
conv2d_409 (Conv2D)             (None, 8, 8, 16)     92176       activation_235[0][0]
dropout_314 (Dropout)           (None, 8, 8, 16)     0           conv2d_409[0][0]
concatenate_265 (Concatenate)   (None, 8, 8, 656)    0           dropout_314[0][0]
                                                                 concatenate_264[0][0]
batch_normalization_234 (BNorm) (None, 8, 8, 656)    32          concatenate_265[0][0]
activation_236 (Activation)     (None, 8, 8, 656)    0           batch_normalization_234[0][0]
conv2d_410 (Conv2D)             (None, 8, 8, 656)    430992      activation_236[0][0]
dropout_315 (Dropout)           (None, 8, 8, 656)    0           conv2d_410[0][0]
max_pooling2d_50 (MaxPooling2D) (None, 4, 4, 656)    0           dropout_315[0][0]
batch_normalization_235 (BNorm) (None, 4, 4, 656)    16          max_pooling2d_50[0][0]
activation_237 (Activation)     (None, 4, 4, 656)    0           batch_normalization_235[0][0]
conv2d_411 (Conv2D)             (None, 4, 4, 16)     94480       activation_237[0][0]
dropout_316 (Dropout)           (None, 4, 4, 16)     0           conv2d_411[0][0]
concatenate_266 (Concatenate)   (None, 4, 4, 672)    0           dropout_316[0][0]
                                                                 max_pooling2d_50[0][0]
batch_normalization_236 (BNorm) (None, 4, 4, 672)    16          concatenate_266[0][0]
activation_238 (Activation)     (None, 4, 4, 672)    0           batch_normalization_236[0][0]
conv2d_412 (Conv2D)             (None, 4, 4, 16)     96784       activation_238[0][0]
dropout_317 (Dropout)           (None, 4, 4, 16)     0           conv2d_412[0][0]
concatenate_267 (Concatenate)   (None, 4, 4, 688)    0           dropout_317[0][0]
                                                                 concatenate_266[0][0]
batch_normalization_237 (BNorm) (None, 4, 4, 688)    16          concatenate_267[0][0]
activation_239 (Activation)     (None, 4, 4, 688)    0           batch_normalization_237[0][0]
conv2d_413 (Conv2D)             (None, 4, 4, 16)     99088       activation_239[0][0]
dropout_318 (Dropout)           (None, 4, 4, 16)     0           conv2d_413[0][0]
```

Listing 7.17: Summary of the Tiramisu model (part 6).

```
concatenate_268 (Concatenate)   (None, 4, 4, 704)    0           dropout_318[0][0]
                                                                 concatenate_267[0][0]
batch_normalization_238 (BNorm) (None, 4, 4, 704)    16          concatenate_268[0][0]
activation_240 (Activation)     (None, 4, 4, 704)    0           batch_normalization_238[0][0]
conv2d_414 (Conv2D)             (None, 4, 4, 16)     101392      activation_240[0][0]
dropout_319 (Dropout)           (None, 4, 4, 16)     0           conv2d_414[0][0]
concatenate_269 (Concatenate)   (None, 4, 4, 720)    0           dropout_319[0][0]
                                                                 concatenate_268[0][0]
batch_normalization_239 (BNorm) (None, 4, 4, 720)    16          concatenate_269[0][0]
activation_241 (Activation)     (None, 4, 4, 720)    0           batch_normalization_239[0][0]
conv2d_415 (Conv2D)             (None, 4, 4, 16)     103696      activation_241[0][0]
dropout_320 (Dropout)           (None, 4, 4, 16)     0           conv2d_415[0][0]
concatenate_270 (Concatenate)   (None, 4, 4, 736)    0           dropout_320[0][0]
                                                                 concatenate_269[0][0]
batch_normalization_240 (BNorm) (None, 4, 4, 736)    16          concatenate_270[0][0]
activation_242 (Activation)     (None, 4, 4, 736)    0           batch_normalization_240[0][0]
conv2d_416 (Conv2D)             (None, 4, 4, 16)     106000      activation_242[0][0]
dropout_321 (Dropout)           (None, 4, 4, 16)     0           conv2d_416[0][0]
concatenate_271 (Concatenate)   (None, 4, 4, 752)    0           dropout_321[0][0]
                                                                 concatenate_270[0][0]
batch_normalization_241 (BNorm) (None, 4, 4, 752)    16          concatenate_271[0][0]
activation_243 (Activation)     (None, 4, 4, 752)    0           batch_normalization_241[0][0]
conv2d_417 (Conv2D)             (None, 4, 4, 16)     108304      activation_243[0][0]
dropout_322 (Dropout)           (None, 4, 4, 16)     0           conv2d_417[0][0]
concatenate_272 (Concatenate)   (None, 4, 4, 768)    0           dropout_322[0][0]
                                                                 concatenate_271[0][0]
batch_normalization_242 (BNorm) (None, 4, 4, 768)    16          concatenate_272[0][0]
activation_244 (Activation)     (None, 4, 4, 768)    0           batch_normalization_242[0][0]
conv2d_418 (Conv2D)             (None, 4, 4, 16)     110608      activation_244[0][0]
dropout_323 (Dropout)           (None, 4, 4, 16)     0           conv2d_418[0][0]
concatenate_273 (Concatenate)   (None, 4, 4, 784)    0           dropout_323[0][0]
                                                                 concatenate_272[0][0]
batch_normalization_243 (BNorm) (None, 4, 4, 784)    16          concatenate_273[0][0]
activation_245 (Activation)     (None, 4, 4, 784)    0           batch_normalization_243[0][0]
conv2d_419 (Conv2D)             (None, 4, 4, 16)     112912      activation_245[0][0]
dropout_324 (Dropout)           (None, 4, 4, 16)     0           conv2d_419[0][0]
concatenate_274 (Concatenate)   (None, 4, 4, 800)    0           dropout_324[0][0]
                                                                 concatenate_273[0][0]
batch_normalization_244 (BNorm) (None, 4, 4, 800)    16          concatenate_274[0][0]
activation_246 (Activation)     (None, 4, 4, 800)    0           batch_normalization_244[0][0]
conv2d_420 (Conv2D)             (None, 4, 4, 16)     115216      activation_246[0][0]
dropout_325 (Dropout)           (None, 4, 4, 16)     0           conv2d_420[0][0]
concatenate_275 (Concatenate)   (None, 4, 4, 816)    0           dropout_325[0][0]
                                                                 concatenate_274[0][0]
batch_normalization_245 (BNorm) (None, 4, 4, 816)    16          concatenate_275[0][0]
activation_247 (Activation)     (None, 4, 4, 816)    0           batch_normalization_245[0][0]
```

Listing 7.18: Summary of the Tiramisu model (part 7).

```
conv2d_421 (Conv2D)             (None, 4, 4, 16)     117520      activation_247[0][0]
dropout_326 (Dropout)           (None, 4, 4, 16)     0           conv2d_421[0][0]
concatenate_276 (Concatenate)   (None, 4, 4, 832)    0           dropout_326[0][0]
                                                                 concatenate_275[0][0]
batch_normalization_246 (BNorm) (None, 4, 4, 832)    16          concatenate_276[0][0]
activation_248 (Activation)     (None, 4, 4, 832)    0           batch_normalization_246[0][0]
conv2d_422 (Conv2D)             (None, 4, 4, 16)     119824      activation_248[0][0]
dropout_327 (Dropout)           (None, 4, 4, 16)     0           conv2d_422[0][0]
concatenate_277 (Concatenate)   (None, 4, 4, 848)    0           dropout_327[0][0]
                                                                 concatenate_276[0][0]
batch_normalization_247 (BNorm) (None, 4, 4, 848)    16          concatenate_277[0][0]
activation_249 (Activation)     (None, 4, 4, 848)    0           batch_normalization_247[0][0]
conv2d_423 (Conv2D)             (None, 4, 4, 16)     122128      activation_249[0][0]
dropout_328 (Dropout)           (None, 4, 4, 16)     0           conv2d_423[0][0]
concatenate_278 (Concatenate)   (None, 4, 4, 864)    0           dropout_328[0][0]
                                                                 concatenate_277[0][0]
batch_normalization_248 (BNorm) (None, 4, 4, 864)    16          concatenate_278[0][0]
activation_250 (Activation)     (None, 4, 4, 864)    0           batch_normalization_248[0][0]
conv2d_424 (Conv2D)             (None, 4, 4, 16)     124432      activation_250[0][0]
dropout_329 (Dropout)           (None, 4, 4, 16)     0           conv2d_424[0][0]
concatenate_279 (Concatenate)   (None, 4, 4, 880)    0           dropout_329[0][0]
                                                                 concatenate_278[0][0]
batch_normalization_249 (BNorm) (None, 4, 4, 880)    16          concatenate_279[0][0]
activation_251 (Activation)     (None, 4, 4, 880)    0           batch_normalization_249[0][0]
conv2d_425 (Conv2D)             (None, 4, 4, 16)     126736      activation_251[0][0]
dropout_330 (Dropout)           (None, 4, 4, 16)     0           conv2d_425[0][0]
concatenate_280 (Concatenate)   (None, 4, 4, 896)    0           dropout_330[0][0]
                                                                 concatenate_279[0][0]
conv2d_transpose_46 (Conv2DTr.) (None, 8, 8, 240)    1935600     concatenate_280[0][0]
concatenate_281 (Concatenate)   (None, 8, 8, 896)    0           conv2d_transpose_46[0][0]
                                                                 concatenate_265[0][0]
batch_normalization_250 (BNorm) (None, 8, 8, 896)    32          concatenate_281[0][0]
activation_252 (Activation)     (None, 8, 8, 896)    0           batch_normalization_250[0][0]
conv2d_426 (Conv2D)             (None, 8, 8, 16)     129040      activation_252[0][0]
dropout_331 (Dropout)           (None, 8, 8, 16)     0           conv2d_426[0][0]
concatenate_282 (Concatenate)   (None, 8, 8, 912)    0           dropout_331[0][0]
                                                                 concatenate_281[0][0]
batch_normalization_251 (BNorm) (None, 8, 8, 912)    32          concatenate_282[0][0]
activation_253 (Activation)     (None, 8, 8, 912)    0           batch_normalization_251[0][0]
conv2d_427 (Conv2D)             (None, 8, 8, 16)     131344      activation_253[0][0]
dropout_332 (Dropout)           (None, 8, 8, 16)     0           conv2d_427[0][0]
concatenate_283 (Concatenate)   (None, 8, 8, 928)    0           dropout_332[0][0]
                                                                 concatenate_282[0][0]
batch_normalization_252 (BNorm) (None, 8, 8, 928)    32          concatenate_283[0][0]
activation_254 (Activation)     (None, 8, 8, 928)    0           batch_normalization_252[0][0]
conv2d_428 (Conv2D)             (None, 8, 8, 16)     133648      activation_254[0][0]
```

Listing 7.19: Summary of the Tiramisu model (part 8).

```
dropout_333 (Dropout)           (None, 8, 8, 16)     0           conv2d_428[0][0]
concatenate_284 (Concatenate)   (None, 8, 8, 944)    0           dropout_333[0][0]
                                                                 concatenate_283[0][0]
batch_normalization_253 (BNorm) (None, 8, 8, 944)    32          concatenate_284[0][0]
activation_255 (Activation)     (None, 8, 8, 944)    0           batch_normalization_253[0][0]
conv2d_429 (Conv2D)             (None, 8, 8, 16)     135952      activation_255[0][0]
dropout_334 (Dropout)           (None, 8, 8, 16)     0           conv2d_429[0][0]
concatenate_285 (Concatenate)   (None, 8, 8, 960)    0           dropout_334[0][0]
                                                                 concatenate_284[0][0]
batch_normalization_254 (BNorm) (None, 8, 8, 960)    32          concatenate_285[0][0]
activation_256 (Activation)     (None, 8, 8, 960)    0           batch_normalization_254[0][0]
conv2d_430 (Conv2D)             (None, 8, 8, 16)     138256      activation_256[0][0]
dropout_335 (Dropout)           (None, 8, 8, 16)     0           conv2d_430[0][0]
concatenate_286 (Concatenate)   (None, 8, 8, 976)    0           dropout_335[0][0]
                                                                 concatenate_285[0][0]
batch_normalization_255 (BNorm) (None, 8, 8, 976)    32          concatenate_286[0][0]
activation_257 (Activation)     (None, 8, 8, 976)    0           batch_normalization_255[0][0]
conv2d_431 (Conv2D)             (None, 8, 8, 16)     140560      activation_257[0][0]
dropout_336 (Dropout)           (None, 8, 8, 16)     0           conv2d_431[0][0]
concatenate_287 (Concatenate)   (None, 8, 8, 992)    0           dropout_336[0][0]
                                                                 concatenate_286[0][0]
batch_normalization_256 (BNorm) (None, 8, 8, 992)    32          concatenate_287[0][0]
activation_258 (Activation)     (None, 8, 8, 992)    0           batch_normalization_256[0][0]
conv2d_432 (Conv2D)             (None, 8, 8, 16)     142864      activation_258[0][0]
dropout_337 (Dropout)           (None, 8, 8, 16)     0           conv2d_432[0][0]
concatenate_288 (Concatenate)   (None, 8, 8, 1008)   0           dropout_337[0][0]
                                                                 concatenate_287[0][0]
batch_normalization_257 (BNorm) (None, 8, 8, 1008)   32          concatenate_288[0][0]
activation_259 (Activation)     (None, 8, 8, 1008)   0           batch_normalization_257[0][0]
conv2d_433 (Conv2D)             (None, 8, 8, 16)     145168      activation_259[0][0]
dropout_338 (Dropout)           (None, 8, 8, 16)     0           conv2d_433[0][0]
concatenate_289 (Concatenate)   (None, 8, 8, 1024)   0           dropout_338[0][0]
                                                                 concatenate_288[0][0]
batch_normalization_258 (BNorm) (None, 8, 8, 1024)   32          concatenate_289[0][0]
activation_260 (Activation)     (None, 8, 8, 1024)   0           batch_normalization_258[0][0]
conv2d_434 (Conv2D)             (None, 8, 8, 16)     147472      activation_260[0][0]
dropout_339 (Dropout)           (None, 8, 8, 16)     0           conv2d_434[0][0]
concatenate_290 (Concatenate)   (None, 8, 8, 1040)   0           dropout_339[0][0]
                                                                 concatenate_289[0][0]
batch_normalization_259 (BNorm) (None, 8, 8, 1040)   32          concatenate_290[0][0]
activation_261 (Activation)     (None, 8, 8, 1040)   0           batch_normalization_259[0][0]
conv2d_435 (Conv2D)             (None, 8, 8, 16)     149776      activation_261[0][0]
dropout_340 (Dropout)           (None, 8, 8, 16)     0           conv2d_435[0][0]
concatenate_291 (Concatenate)   (None, 8, 8, 1056)   0           dropout_340[0][0]
                                                                 concatenate_290[0][0]
batch_normalization_260 (BNorm) (None, 8, 8, 1056)   32          concatenate_291[0][0]
```

Listing 7.20: Summary of the Tiramisu model (part 9).

```
activation_262 (Activation)     (None, 8, 8, 1056)   0           batch_normalization_260[0][0]
conv2d_436 (Conv2D)             (None, 8, 8, 16)     152080      activation_262[0][0]
dropout_341 (Dropout)           (None, 8, 8, 16)     0           conv2d_436[0][0]
concatenate_292 (Concatenate)   (None, 8, 8, 1072)   0           dropout_341[0][0]
                                                                 concatenate_291[0][0]
batch_normalization_261 (BNorm) (None, 8, 8, 1072)   32          concatenate_292[0][0]
activation_263 (Activation)     (None, 8, 8, 1072)   0           batch_normalization_261[0][0]
conv2d_437 (Conv2D)             (None, 8, 8, 16)     154384      activation_263[0][0]
dropout_342 (Dropout)           (None, 8, 8, 16)     0           conv2d_437[0][0]
concatenate_293 (Concatenate)   (None, 8, 8, 1088)   0           dropout_342[0][0]
                                                                 concatenate_292[0][0]
conv2d_transpose_47 (Conv2DTr.) (None, 16, 16, 192)  1880256     concatenate_293[0][0]
concatenate_294 (Concatenate)   (None, 16, 16, 656)  0           conv2d_transpose_47[0][0]
                                                                 concatenate_253[0][0]
batch_normalization_262 (BNorm) (None, 16, 16, 656)  64          concatenate_294[0][0]
activation_264 (Activation)     (None, 16, 16, 656)  0           batch_normalization_262[0][0]
conv2d_438 (Conv2D)             (None, 16, 16, 16)   94480       activation_264[0][0]
dropout_343 (Dropout)           (None, 16, 16, 16)   0           conv2d_438[0][0]
concatenate_295 (Concatenate)   (None, 16, 16, 672)  0           dropout_343[0][0]
                                                                 concatenate_294[0][0]
batch_normalization_263 (BNorm) (None, 16, 16, 672)  64          concatenate_295[0][0]
activation_265 (Activation)     (None, 16, 16, 672)  0           batch_normalization_263[0][0]
conv2d_439 (Conv2D)             (None, 16, 16, 16)   96784       activation_265[0][0]
dropout_344 (Dropout)           (None, 16, 16, 16)   0           conv2d_439[0][0]
concatenate_296 (Concatenate)   (None, 16, 16, 688)  0           dropout_344[0][0]
                                                                 concatenate_295[0][0]
batch_normalization_264 (BNorm) (None, 16, 16, 688)  64          concatenate_296[0][0]
activation_266 (Activation)     (None, 16, 16, 688)  0           batch_normalization_264[0][0]
conv2d_440 (Conv2D)             (None, 16, 16, 16)   99088       activation_266[0][0]
dropout_345 (Dropout)           (None, 16, 16, 16)   0           conv2d_440[0][0]
concatenate_297 (Concatenate)   (None, 16, 16, 704)  0           dropout_345[0][0]
                                                                 concatenate_296[0][0]
batch_normalization_265 (BNorm) (None, 16, 16, 704)  64          concatenate_297[0][0]
activation_267 (Activation)     (None, 16, 16, 704)  0           batch_normalization_265[0][0]
conv2d_441 (Conv2D)             (None, 16, 16, 16)   101392      activation_267[0][0]
dropout_346 (Dropout)           (None, 16, 16, 16)   0           conv2d_441[0][0]
concatenate_298 (Concatenate)   (None, 16, 16, 720)  0           dropout_346[0][0]
                                                                 concatenate_297[0][0]
batch_normalization_266 (BNorm) (None, 16, 16, 720)  64          concatenate_298[0][0]
activation_268 (Activation)     (None, 16, 16, 720)  0           batch_normalization_266[0][0]
conv2d_442 (Conv2D)             (None, 16, 16, 16)   103696      activation_268[0][0]
dropout_347 (Dropout)           (None, 16, 16, 16)   0           conv2d_442[0][0]
concatenate_299 (Concatenate)   (None, 16, 16, 736)  0           dropout_347[0][0]
                                                                 concatenate_298[0][0]
batch_normalization_267 (BNorm) (None, 16, 16, 736)  64          concatenate_299[0][0]
activation_269 (Activation)     (None, 16, 16, 736)  0           batch_normalization_267[0][0]
```

Listing 7.21: Summary of the Tiramisu model (part 10).

```
conv2d_443 (Conv2D)             (None, 16, 16, 16)   106000      activation_269[0][0]
dropout_348 (Dropout)           (None, 16, 16, 16)   0           conv2d_443[0][0]
concatenate_300 (Concatenate)   (None, 16, 16, 752)  0           dropout_348[0][0]
                                                                 concatenate_299[0][0]
batch_normalization_268 (BNorm) (None, 16, 16, 752)  64          concatenate_300[0][0]
activation_270 (Activation)     (None, 16, 16, 752)  0           batch_normalization_268[0][0]
conv2d_444 (Conv2D)             (None, 16, 16, 16)   108304      activation_270[0][0]
dropout_349 (Dropout)           (None, 16, 16, 16)   0           conv2d_444[0][0]
concatenate_301 (Concatenate)   (None, 16, 16, 768)  0           dropout_349[0][0]
                                                                 concatenate_300[0][0]
batch_normalization_269 (BNorm) (None, 16, 16, 768)  64          concatenate_301[0][0]
activation_271 (Activation)     (None, 16, 16, 768)  0           batch_normalization_269[0][0]
conv2d_445 (Conv2D)             (None, 16, 16, 16)   110608      activation_271[0][0]
dropout_350 (Dropout)           (None, 16, 16, 16)   0           conv2d_445[0][0]
concatenate_302 (Concatenate)   (None, 16, 16, 784)  0           dropout_350[0][0]
                                                                 concatenate_301[0][0]
batch_normalization_270 (BNorm) (None, 16, 16, 784)  64          concatenate_302[0][0]
activation_272 (Activation)     (None, 16, 16, 784)  0           batch_normalization_270[0][0]
conv2d_446 (Conv2D)             (None, 16, 16, 16)   112912      activation_272[0][0]
dropout_351 (Dropout)           (None, 16, 16, 16)   0           conv2d_446[0][0]
concatenate_303 (Concatenate)   (None, 16, 16, 800)  0           dropout_351[0][0]
                                                                 concatenate_302[0][0]
batch_normalization_271 (BNorm) (None, 16, 16, 800)  64          concatenate_303[0][0]
activation_273 (Activation)     (None, 16, 16, 800)  0           batch_normalization_271[0][0]
conv2d_447 (Conv2D)             (None, 16, 16, 16)   115216      activation_273[0][0]
dropout_352 (Dropout)           (None, 16, 16, 16)   0           conv2d_447[0][0]
concatenate_304 (Concatenate)   (None, 16, 16, 816)  0           dropout_352[0][0]
                                                                 concatenate_303[0][0]
conv2d_transpose_48 (Conv2DTr.) (None, 32, 32, 160)  1175200     concatenate_304[0][0]
concatenate_305 (Concatenate)   (None, 32, 32, 464)  0           conv2d_transpose_48[0][0]
                                                                 concatenate_243[0][0]
batch_normalization_272 (BNorm) (None, 32, 32, 464)  128         concatenate_305[0][0]
activation_274 (Activation)     (None, 32, 32, 464)  0           batch_normalization_272[0][0]
conv2d_448 (Conv2D)             (None, 32, 32, 16)   66832       activation_274[0][0]
dropout_353 (Dropout)           (None, 32, 32, 16)   0           conv2d_448[0][0]
concatenate_306 (Concatenate)   (None, 32, 32, 480)  0           dropout_353[0][0]
                                                                 concatenate_305[0][0]
batch_normalization_273 (BNorm) (None, 32, 32, 480)  128         concatenate_306[0][0]
activation_275 (Activation)     (None, 32, 32, 480)  0           batch_normalization_273[0][0]
conv2d_449 (Conv2D)             (None, 32, 32, 16)   69136       activation_275[0][0]
dropout_354 (Dropout)           (None, 32, 32, 16)   0           conv2d_449[0][0]
concatenate_307 (Concatenate)   (None, 32, 32, 496)  0           dropout_354[0][0]
                                                                 concatenate_306[0][0]
batch_normalization_274 (BNorm) (None, 32, 32, 496)  128         concatenate_307[0][0]
activation_276 (Activation)     (None, 32, 32, 496)  0           batch_normalization_274[0][0]
conv2d_450 (Conv2D)             (None, 32, 32, 16)   71440       activation_276[0][0]
```

Listing 7.22: Summary of the Tiramisu model (part 11).

```
dropout_355 (Dropout)           (None, 32, 32, 16)   0           conv2d_450[0][0]
concatenate_308 (Concatenate)   (None, 32, 32, 512)  0           dropout_355[0][0]
                                                                 concatenate_307[0][0]
batch_normalization_275 (BNorm) (None, 32, 32, 512)  128         concatenate_308[0][0]
activation_277 (Activation)     (None, 32, 32, 512)  0           batch_normalization_275[0][0]
conv2d_451 (Conv2D)             (None, 32, 32, 16)   73744       activation_277[0][0]
dropout_356 (Dropout)           (None, 32, 32, 16)   0           conv2d_451[0][0]
concatenate_309 (Concatenate)   (None, 32, 32, 528)  0           dropout_356[0][0]
                                                                 concatenate_308[0][0]
batch_normalization_276 (BNorm) (None, 32, 32, 528)  128         concatenate_309[0][0]
activation_278 (Activation)     (None, 32, 32, 528)  0           batch_normalization_276[0][0]
conv2d_452 (Conv2D)             (None, 32, 32, 16)   76048       activation_278[0][0]
dropout_357 (Dropout)           (None, 32, 32, 16)   0           conv2d_452[0][0]
concatenate_310 (Concatenate)   (None, 32, 32, 544)  0           dropout_357[0][0]
                                                                 concatenate_309[0][0]
batch_normalization_277 (BNorm) (None, 32, 32, 544)  128         concatenate_310[0][0]
activation_279 (Activation)     (None, 32, 32, 544)  0           batch_normalization_277[0][0]
conv2d_453 (Conv2D)             (None, 32, 32, 16)   78352       activation_279[0][0]
dropout_358 (Dropout)           (None, 32, 32, 16)   0           conv2d_453[0][0]
concatenate_311 (Concatenate)   (None, 32, 32, 560)  0           dropout_358[0][0]
                                                                 concatenate_310[0][0]
batch_normalization_278 (BNorm) (None, 32, 32, 560)  128         concatenate_311[0][0]
activation_280 (Activation)     (None, 32, 32, 560)  0           batch_normalization_278[0][0]
conv2d_454 (Conv2D)             (None, 32, 32, 16)   80656       activation_280[0][0]
dropout_359 (Dropout)           (None, 32, 32, 16)   0           conv2d_454[0][0]
concatenate_312 (Concatenate)   (None, 32, 32, 576)  0           dropout_359[0][0]
                                                                 concatenate_311[0][0]
conv2d_transpose_49 (Conv2DTr.) (None, 64, 64, 112)  580720      concatenate_312[0][0]
concatenate_313 (Concatenate)   (None, 64, 64, 304)  0           conv2d_transpose_49[0][0]
                                                                 concatenate_236[0][0]
batch_normalization_279 (BNorm) (None, 64, 64, 304)  256         concatenate_313[0][0]
activation_281 (Activation)     (None, 64, 64, 304)  0           batch_normalization_279[0][0]
conv2d_455 (Conv2D)             (None, 64, 64, 16)   43792       activation_281[0][0]
dropout_360 (Dropout)           (None, 64, 64, 16)   0           conv2d_455[0][0]
concatenate_314 (Concatenate)   (None, 64, 64, 320)  0           dropout_360[0][0]
                                                                 concatenate_313[0][0]
batch_normalization_280 (BNorm) (None, 64, 64, 320)  256         concatenate_314[0][0]
activation_282 (Activation)     (None, 64, 64, 320)  0           batch_normalization_280[0][0]
conv2d_456 (Conv2D)             (None, 64, 64, 16)   46096       activation_282[0][0]
dropout_361 (Dropout)           (None, 64, 64, 16)   0           conv2d_456[0][0]
concatenate_315 (Concatenate)   (None, 64, 64, 336)  0           dropout_361[0][0]
                                                                 concatenate_314[0][0]
batch_normalization_281 (BNorm) (None, 64, 64, 336)  256         concatenate_315[0][0]
activation_283 (Activation)     (None, 64, 64, 336)  0           batch_normalization_281[0][0]
conv2d_457 (Conv2D)             (None, 64, 64, 16)   48400       activation_283[0][0]
dropout_362 (Dropout)           (None, 64, 64, 16)   0           conv2d_457[0][0]
```

Listing 7.23: Summary of the Tiramisu model (part 12).

```
concatenate_316 (Concatenate)   (None, 64, 64, 352)  0          dropout_362[0][0]
                                                                concatenate_315[0][0]
batch_normalization_282 (BNorm) (None, 64, 64, 352)  256        concatenate_316[0][0]
activation_284 (Activation)     (None, 64, 64, 352)  0          batch_normalization_282[0][0]
conv2d_458 (Conv2D)             (None, 64, 64, 16)   50704      activation_284[0][0]
dropout_363 (Dropout)           (None, 64, 64, 16)   0          conv2d_458[0][0]
concatenate_317 (Concatenate)   (None, 64, 64, 368)  0          dropout_363[0][0]
                                                                concatenate_316[0][0]
batch_normalization_283 (BNorm) (None, 64, 64, 368)  256        concatenate_317[0][0]
activation_285 (Activation)     (None, 64, 64, 368)  0          batch_normalization_283[0][0]
conv2d_459 (Conv2D)             (None, 64, 64, 16)   53008      activation_285[0][0]
dropout_364 (Dropout)           (None, 64, 64, 16)   0          conv2d_459[0][0]
concatenate_318 (Concatenate)   (None, 64, 64, 384)  0          dropout_364[0][0]
                                                                concatenate_317[0][0]
conv2d_transpose_50 (Conv2DTr.) (None, 128, 128, 80) 276560     concatenate_318[0][0]
concatenate_319 (Concatenate)   (None, 128, 128, 192 0          conv2d_transpose_50[0][0]
                                                                concatenate_231[0][0]
batch_normalization_284 (BNorm) (None, 128, 128, 192 512        concatenate_319[0][0]
activation_286 (Activation)     (None, 128, 128, 192 0          batch_normalization_284[0][0]
conv2d_460 (Conv2D)             (None, 128, 128, 16) 27664      activation_286[0][0]
dropout_365 (Dropout)           (None, 128, 128, 16) 0          conv2d_460[0][0]
concatenate_320 (Concatenate)   (None, 128, 128, 208 0          dropout_365[0][0]
                                                                concatenate_319[0][0]
batch_normalization_285 (BNorm) (None, 128, 128, 208 512        concatenate_320[0][0]
activation_287 (Activation)     (None, 128, 128, 208 0          batch_normalization_285[0][0]
conv2d_461 (Conv2D)             (None, 128, 128, 16) 29968      activation_287[0][0]
dropout_366 (Dropout)           (None, 128, 128, 16) 0          conv2d_461[0][0]
concatenate_321 (Concatenate)   (None, 128, 128, 224 0          dropout_366[0][0]
                                                                concatenate_320[0][0]
batch_normalization_286 (BNorm) (None, 128, 128, 224 512        concatenate_321[0][0]
activation_288 (Activation)     (None, 128, 128, 224 0          batch_normalization_286[0][0]
conv2d_462 (Conv2D)             (None, 128, 128, 16) 32272      activation_288[0][0]
dropout_367 (Dropout)           (None, 128, 128, 16) 0          conv2d_462[0][0]
concatenate_322 (Concatenate)   (None, 128, 128, 240 0          dropout_367[0][0]
                                                                concatenate_321[0][0]
batch_normalization_287 (BNorm) (None, 128, 128, 240 512        concatenate_322[0][0]
activation_289 (Activation)     (None, 128, 128, 240 0          batch_normalization_287[0][0]
conv2d_463 (Conv2D)             (None, 128, 128, 16) 34576      activation_289[0][0]
dropout_368 (Dropout)           (None, 128, 128, 16) 0          conv2d_463[0][0]
concatenate_323 (Concatenate)   (None, 128, 128, 256 0          dropout_368[0][0]
                                                                concatenate_322[0][0]
conv2d_464 (Conv2D)             (None, 128, 128, 1)  257        concatenate_323[0][0]
activation_290 (Activation)     (None, 128, 128, 1)  0          conv2d_464[0][0]
==============================================================================================
Total params: 13,824,657        Trainable params: 13,818,793    Non-trainable params: 5,864
```

Listing 7.24: Summary of the Tiramisu model (part 13).

```python
def fit_model(bat_siz,epoch,model_input):
  callbacks = [tf.keras.callbacks.EarlyStopping(patience=100, monitor='val_loss'),
               tf.keras.callbacks.TensorBoard(log_dir='logs')]
  model = model_input
  history = model.fit(
        X_train,
        y_train,
        batch_size = bat_siz,
        verbose=1,
        epochs=epoch,
        callbacks=callbacks,
        validation_data=(X_test, y_test),
        shuffle=False
        )
  return history
```

Listing 7.25: Python code that implements the model training and the callback functions.

```python
    Unet_total_loss_Metric_Iou = UNet(IMG_HEIGHT, IMG_WIDTH, IMG_CHANNELS)
    Unet_total_loss_Metric_Iou.summary()
    tf.keras.utils.plot_model(
        Unet_total_loss_Metric_Iou,
        "Unet_total_loss_Metric_Iou.png",
        show_shapes = True
        )
    Unet_total_loss_Metric_Iou.compile(optimizer=opt,loss = [total_loss], metrics=[dsc])
    checkpointer = tf.keras.callbacks.ModelCheckpoint(
        'Unet_total_loss_Metric_Iou.hdf5',
        verbose=1,
        save_best_only=True
        )
    history_Unet_total_loss_Metric_Iou = get_history(Unet_total_loss_Metric_Iou)
    Unet_total_loss_Metric_Iou.save('Unet_total_loss_Metric_Iou.hdf5')
    hist_df_history_Unet_total_loss_Metric_Iou =
        pd.DataFrame(history_Unet_total_loss_Metric_Iou.history)
    hist_df_history_Unet_total_loss_Metric_Iou.to_csv(
        'hist_df_history_Unet_total_loss_Metric_Iou',
        encoding='utf-8',
        index=False
        )
```

Listing 7.26: Python code for training U-Net with total loss.

```python
from matplotlib.pyplot import figure

figure(num=None, figsize=(10, 6), dpi=80, facecolor='w', edgecolor='k')

with plt.style.context('Solarize_Light2'):

    plt.plot(
        (history_Unet_dice_loss_Metric_Iou.history['val_dsc']),
        label='unet jaccard',
        color='b')
    plt.plot(
        (history_Unet_bce_dice_loss_Metric_Iou.history['val_dsc']),
        label='bce_dice_loss',
        color='k')
    plt.plot(
        (history_Unet_BinaryCrossentropy_loss_Metric_Iou.history['val_dsc']),
        label='BinaryCrossentropy_loss',
        color='r')
    plt.plot(
        (history_Unet_FocalTversky_loss_Metric_Iou.history['val_dsc']),
        label='FocalTverskyLoss',
        color='g')
    plt.plot(
        (history_UNet_lovasz_model.history['val_dsc']),
        label='LovaszLoss',
        color='w')
    plt.plot(
        (history_Unet_total_loss_Metric_Iou_opt2.history['val_dsc']),
        label='LovaszLoss',
        color='c')
    plt.plot(
        (history_Unet_total_loss_Metric_Iou.history['val_dsc']),
        label='total_loss',
        color='y')

    plt.title('Intersection over Union for different loss functions and U-Net')
    plt.xlabel('Number of Epochs', fontsize=14)
    plt.ylabel('Intersection over Union', fontsize=14)
    ax = plt.subplot(111)
    box = ax.get_position()
    ax.set_position([box.x0, box.y0, box.width * 0.8, box.height])

    ax.legend(loc='center left', bbox_to_anchor=(1, 0.5))
    #plt.legend(loc='best')

plt.show()
```

Listing 7.27: Python code to plot the results from training U-Net.

```python
from   keras.models import Model
from   keras.layers import Input, MaxPooling2D
from   keras.layers.convolutional import Conv2D, Conv2DTranspose
from   keras.layers.normalization import BatchNormalization
from   keras.layers.core import Activation, Dropout
from   keras.layers.merge import concatenate
from   keras import backend as K
from   keras.regularizers import l2
from   keras.utils import plot_model
import pydot
import graphviz

def denseBlock(t, nb_layers):
    for _ in range(nb_layers):
        tmp = t
        t = BatchNormalization(
            axis=1,
            gamma_regularizer=l2(0.0001),
            beta_regularizer=l2(0.0001)
            )(t)
        t = Activation('relu')(t)
        t = Conv2D(
            16,
            kernel_size=(3, 3),
            padding='same',
            kernel_initializer='he_uniform',
            data_format='channels_last'
            )(t)
        t = Dropout(0.2)(t)
        t = concatenate([t, tmp])
    return t

def transitionDown(t, nb_features):
    t = BatchNormalization(
        axis=1,
        gamma_regularizer=l2(0.0001),
        beta_regularizer=l2(0.0001)
        )(t)
    t = Activation('relu')(t)
    t = Conv2D(
        nb_features,
        kernel_size=(1, 1),
        padding='same',
        kernel_initializer='he_uniform',
        data_format='channels_last'
        )(t)
```

Listing 7.28: Python code that implements the Tiramisu model (part 1).

```
    t = Dropout(0.2)(t)
    t = MaxPooling2D(
        pool_size=(2, 2),
        strides=2,
        padding='same',
        data_format='channels_last'
        )(t)
    return t

def Tiramisu(layer_per_block, n_pool=3, growth_rate=16):
    input_layer = Input(shape=(128, 128, 1))
    t = Conv2D(48, kernel_size=(3, 3), strides=(1, 1), padding='same')(input_layer)

    # Dense block
    nb_features = 48
    skip_connections = []
    for i in range(n_pool):
        t = denseBlock(t, layer_per_block[i])
        skip_connections.append(t)
        nb_features += growth_rate * layer_per_block[i]
        t = transitionDown(t, nb_features)

    t = denseBlock(t, layer_per_block[n_pool]) # bottleneck

    skip_connections = skip_connections[::-1]  # subvert the array

    for i in range(n_pool):
        keep_nb_features = growth_rate * layer_per_block[n_pool + i]
        t = Conv2DTranspose(
            keep_nb_features, strides=2, kernel_size=(3, 3), padding='same',
            data_format='channels_last'
            )(t) # Transition up
        t = concatenate([t, skip_connections[i]])

        t = denseBlock(t, layer_per_block[n_pool+i+1])

    t = Conv2D(
        1, kernel_size=(1, 1), padding='same',
        kernel_initializer='he_uniform',
        data_format='channels_last'
        )(t)
    output_layer = Activation('sigmoid')(t)
    return Model(inputs=input_layer, outputs=output_layer)

layer_per_block = [2, 3, 5, 6, 5, 3,2]
```

Listing 7.29: Python code that implements the Tiramisu model (part 2).

```python
import tensorflow as tf
import math
import keras.models as models
from keras.callbacks import LearningRateScheduler
from keras.optimizers import RMSprop, Adam, SGD
from keras.callbacks import ModelCheckpoint
from keras.layers import Input, merge
from keras.regularizers import l2
from keras.models import Model
from keras import regularizers
from keras.models import Model
from keras.layers import Input, concatenate, Conv2D, MaxPooling2D, UpSampling2D, Cropping2D
from keras import backend as K
from keras import callbacks
from keras.layers.core import Layer, Dense, Dropout, Activation, Flatten, Reshape, Permute
from keras.layers.normalization import BatchNormalization
from keras.layers import Conv2DTranspose
```

Listing 7.30: Import the libraries necessary to implement the Tiramisu model.

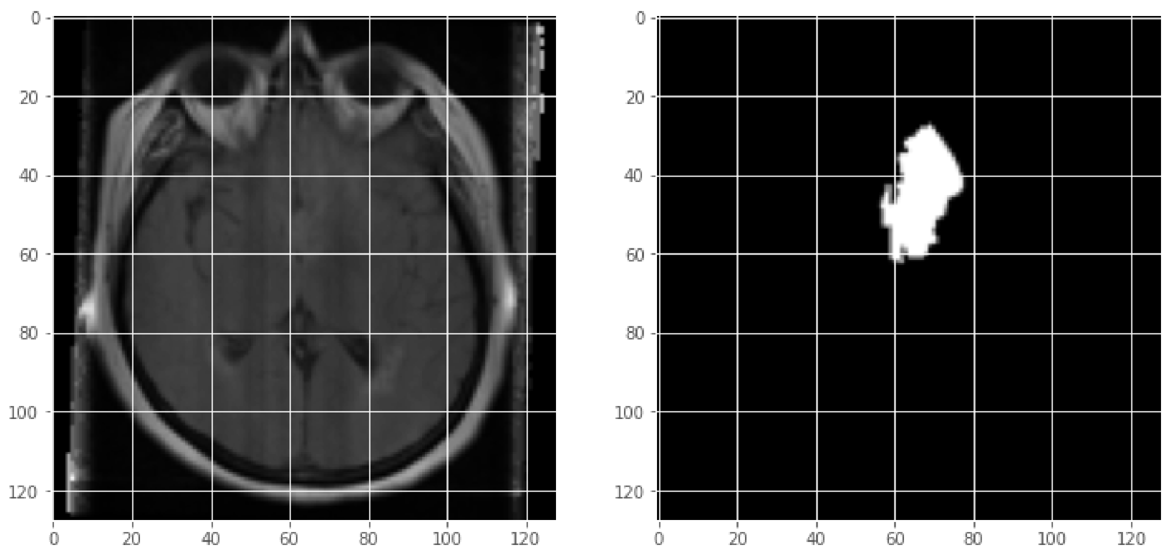


Figure 54: Random image from the dataset and the respective mask.

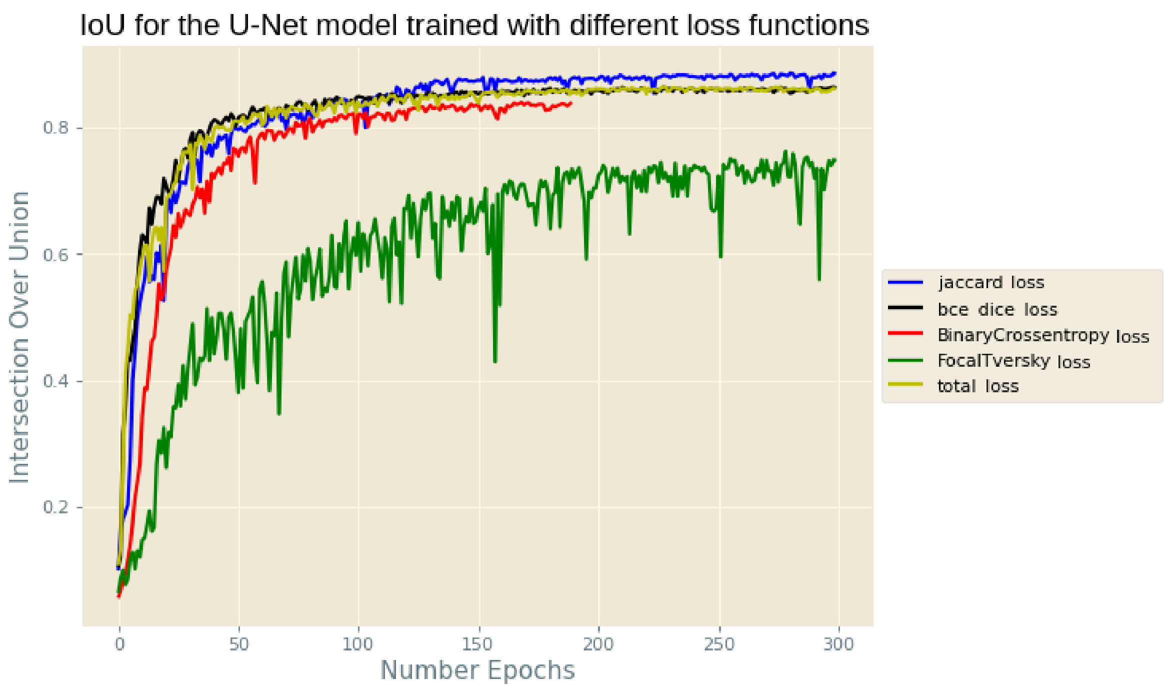


Figure 55: Plot of the IoU metric for the U-Net model trained with different loss functions.

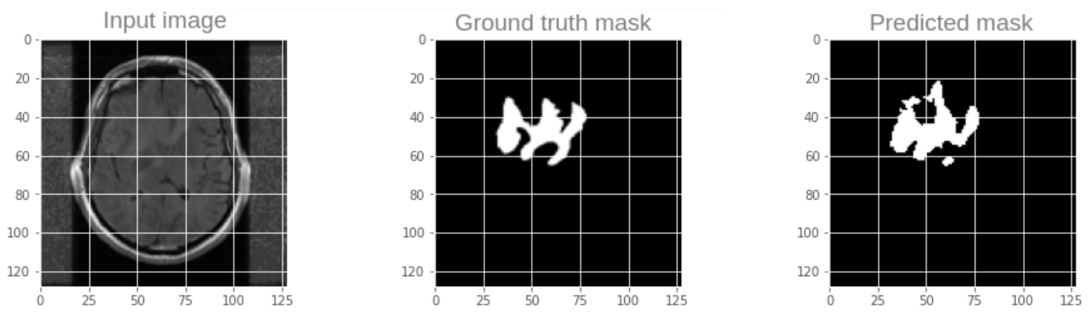


Figure 56: Input image 1, ground truth mask, and predicted mask by the U-Net model with Jaccard loss.

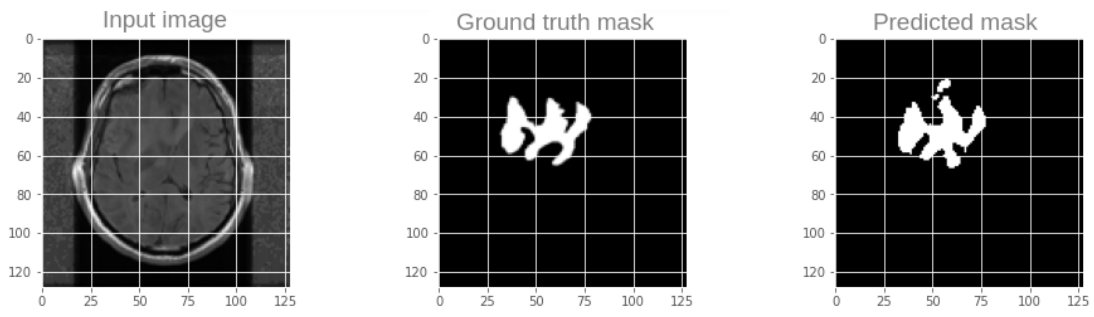


Figure 57: Input image 1, ground truth mask, and predicted mask by the U-Net model with BCE-Dice loss.

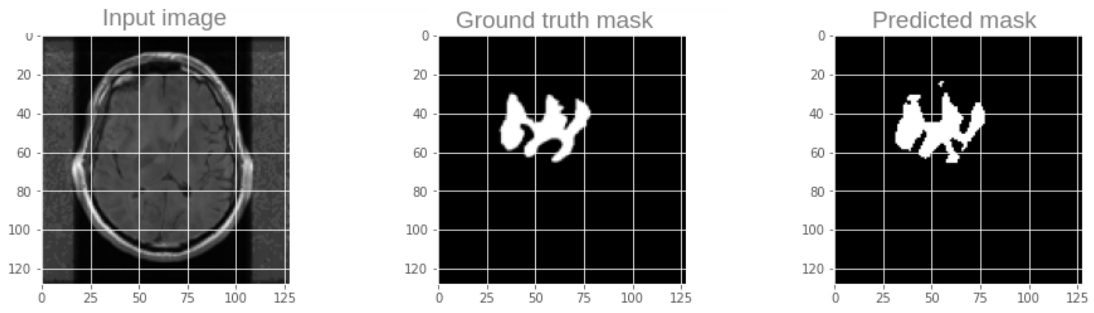


Figure 58: Input image 1, ground truth mask, and predicted mask by the U-Net model with Binary Cross Entropy loss.

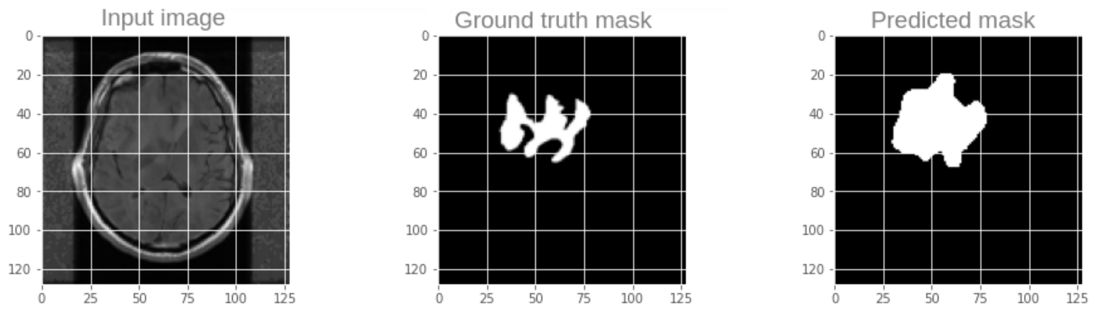


Figure 59: Input image 1, ground truth mask, and predicted mask by the U-Net model with focal Tversky loss.

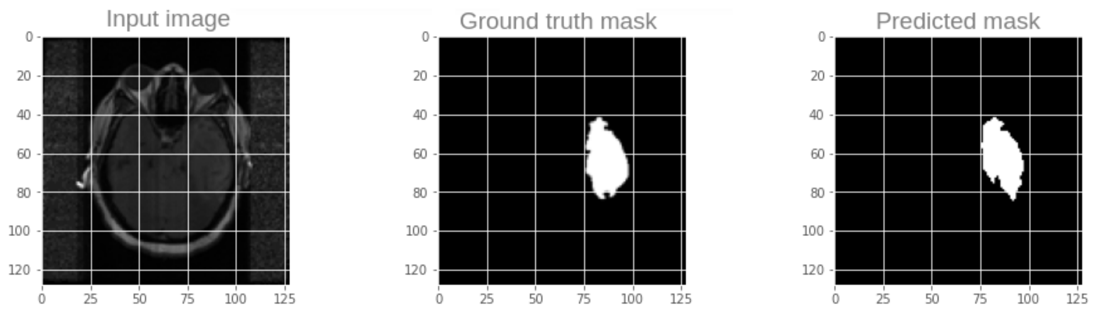


Figure 60: Input image 2, ground truth mask, and mask predicted by the Tiramisu model with BCE loss.

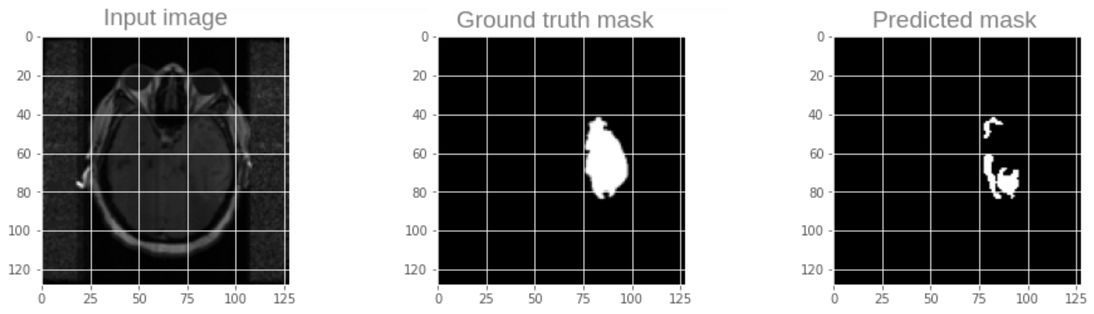


Figure 61: Input image 2, ground truth mask, and mask predicted by the Tiramisu model with Dice loss.

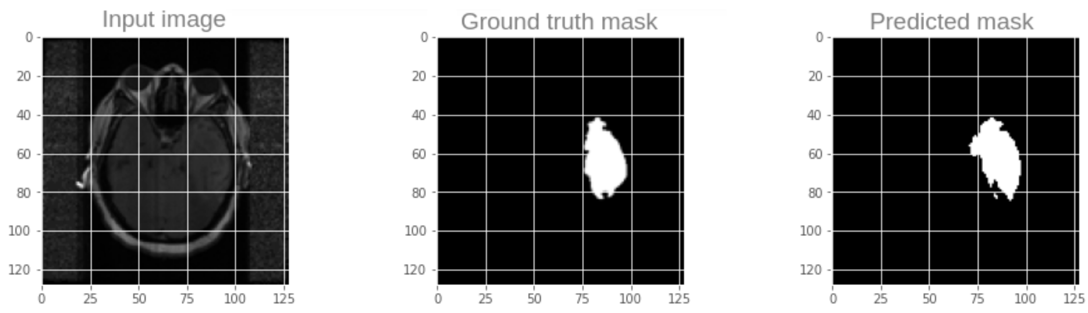


Figure 62: Input image 2, ground truth mask, and mask predicted by the Tiramisu model with BCE-Dice loss.

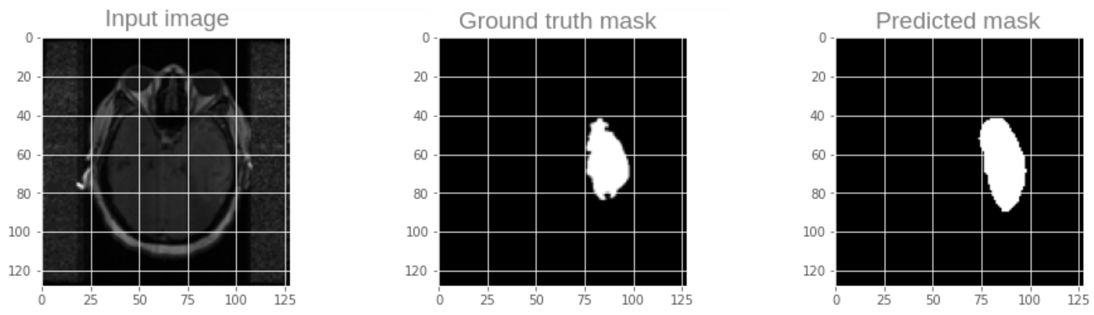


Figure 63: Input image 2, ground truth mask, and mask predicted by the Tiramisu model with focal Tversky loss.

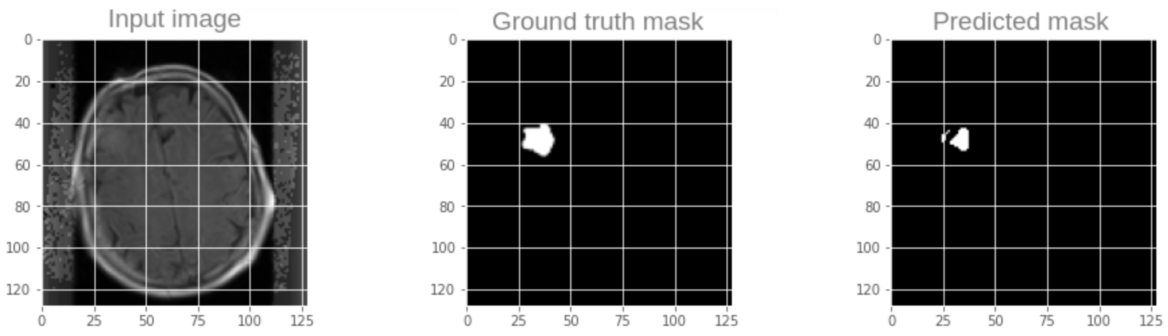


Figure 64: Input image 3, ground truth mask, and mask predicted by the U-Net model with total loss.

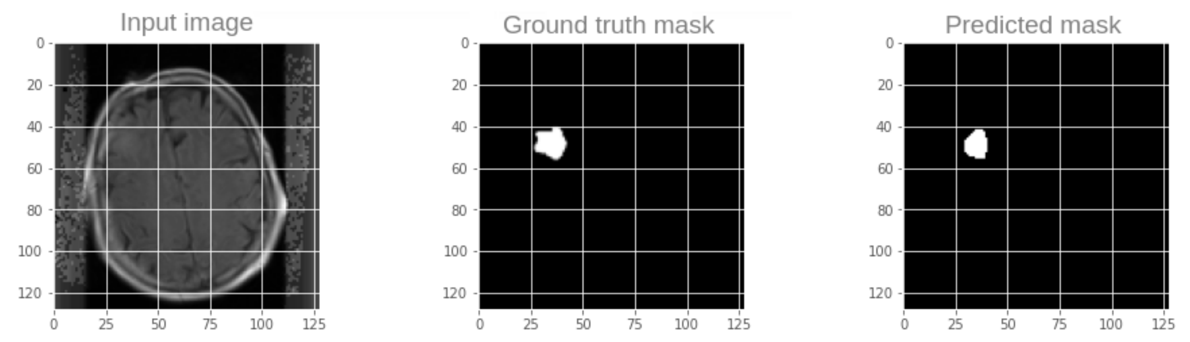


Figure 65: Input image 3, ground truth mask, and mask predicted by the U-Net model with Lovasz loss.

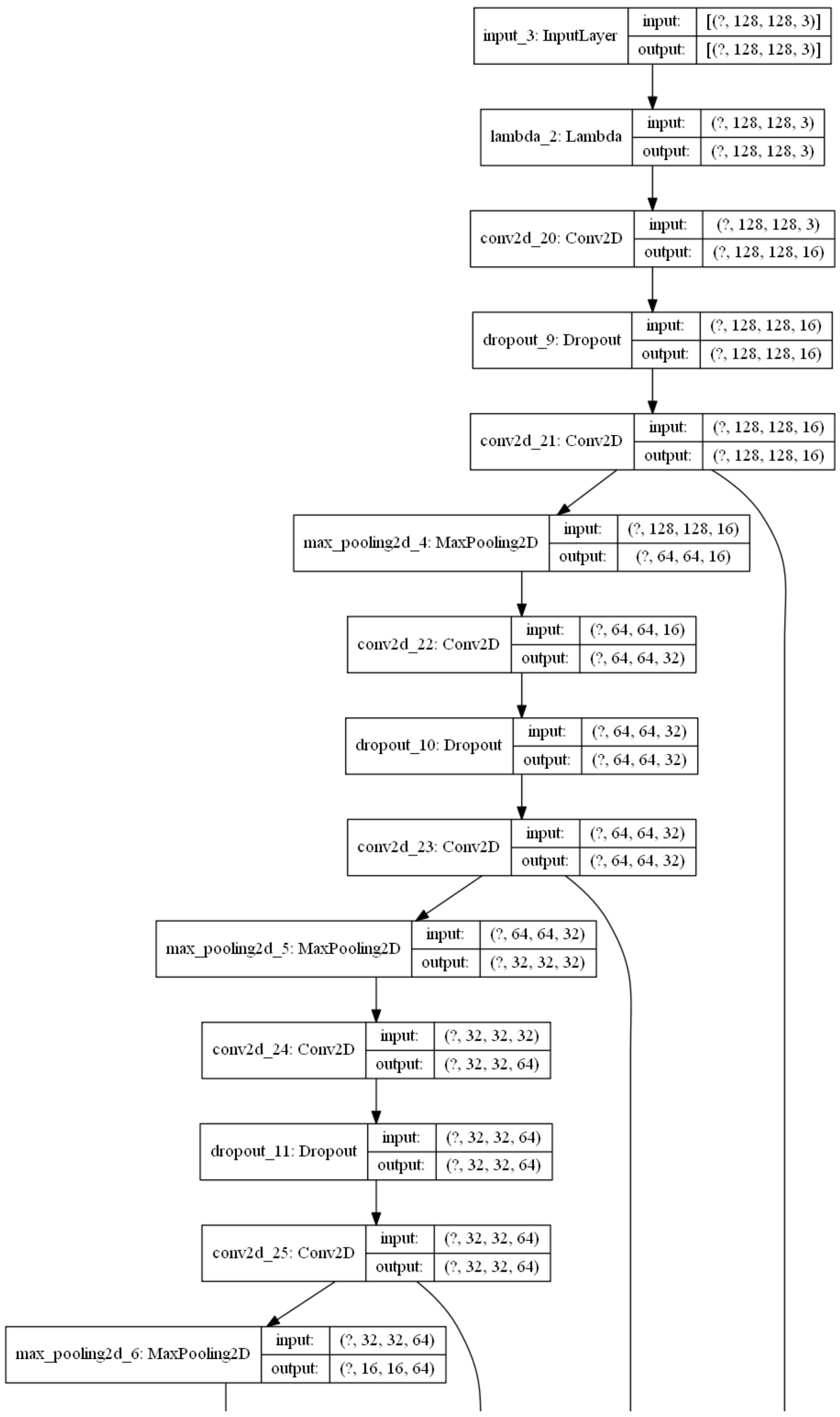


Figure 66: U-Net model architecture (part 1).

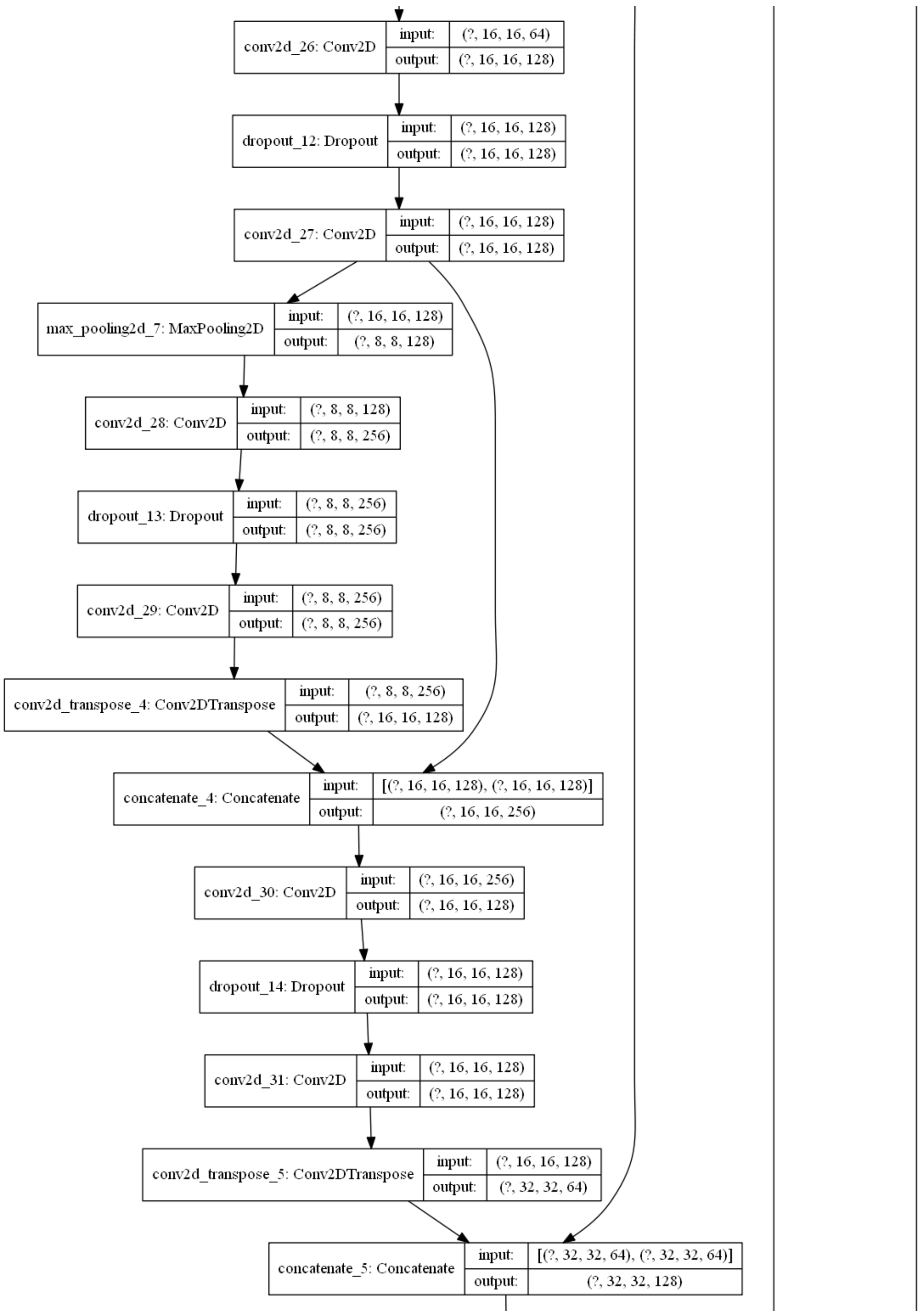


Figure 67: U-Net model architecture (part 2).

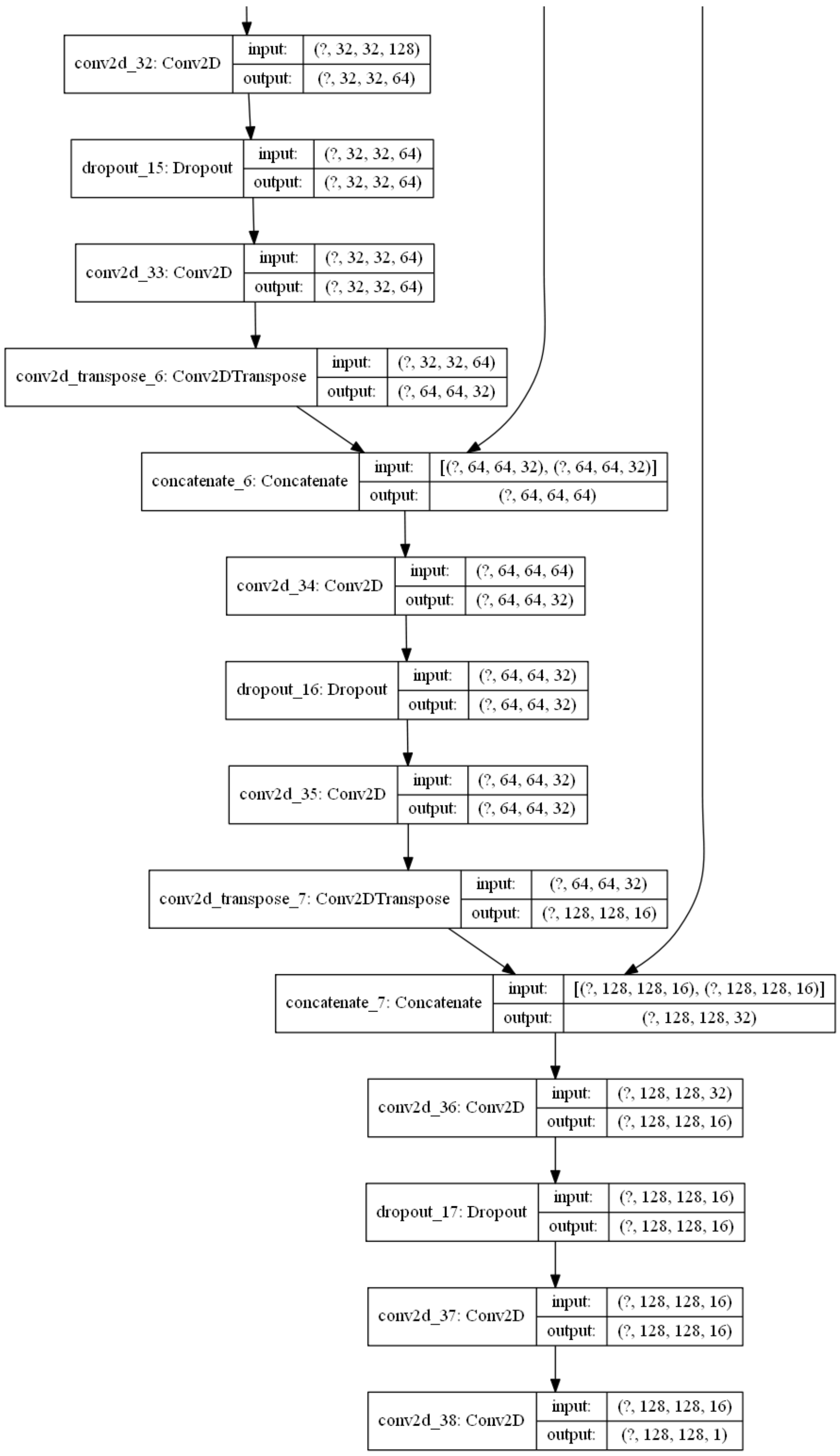


Figure 68: U-Net model architecture (part 3).

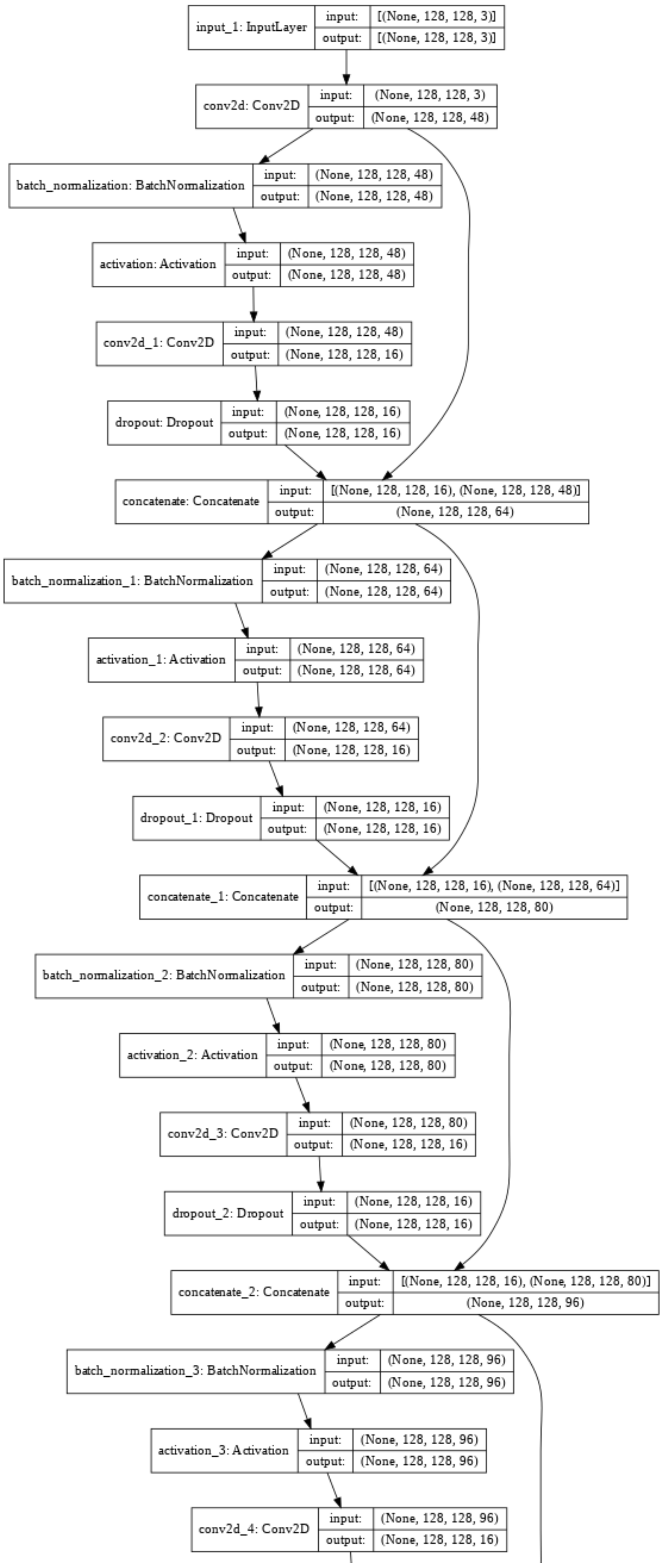

input_1: InputLayer
input: [(None, 128, 128, 3)]
output: [(None, 128, 128, 3)]
conv2d: Conv2D
input: (None, 128, 128, 3)
output: (None, 128, 128, 48)
batch_normalization: BatchNormalization
input: (None, 128, 128, 48)
output: (None, 128, 128, 48)
activation: Activation
input: (None, 128, 128, 48)
output: (None, 128, 128, 48)
conv2d_1: Conv2D
input: (None, 128, 128, 48)
output: (None, 128, 128, 16)
dropout: Dropout
input: (None, 128, 128, 16)
output: (None, 128, 128, 16)
concatenate: Concatenate
input: [(None, 128, 128, 16), (None, 128, 128, 48)]
output: (None, 128, 128, 64)
batch_normalization_1: BatchNormalization
input: (None, 128, 128, 64)
output: (None, 128, 128, 64)
activation_1: Activation
input: (None, 128, 128, 64)
output: (None, 128, 128, 64)
conv2d_2: Conv2D
input: (None, 128, 128, 64)
output: (None, 128, 128, 16)
dropout_1: Dropout
input: (None, 128, 128, 16)
output: (None, 128, 128, 16)
concatenate_1: Concatenate
input: [(None, 128, 128, 16), (None, 128, 128, 64)]
output: (None, 128, 128, 80)
batch_normalization_2: BatchNormalization
input: (None, 128, 128, 80)
output: (None, 128, 128, 80)
activation_2: Activation
input: (None, 128, 128, 80)
output: (None, 128, 128, 80)
conv2d_3: Conv2D
input: (None, 128, 128, 80)
output: (None, 128, 128, 16)
dropout_2: Dropout
input: (None, 128, 128, 16)
output: (None, 128, 128, 16)
concatenate_2: Concatenate
input: [(None, 128, 128, 16), (None, 128, 128, 80)]
output: (None, 128, 128, 96)
batch_normalization_3: BatchNormalization
input: (None, 128, 128, 96)
output: (None, 128, 128, 96)
activation_3: Activation
input: (None, 128, 128, 96)
output: (None, 128, 128, 96)
conv2d_4: Conv2D
input: (None, 128, 128, 96)
output: (None, 128, 128, 16)

Figure 69: Tiramisu model architecture (part 1).

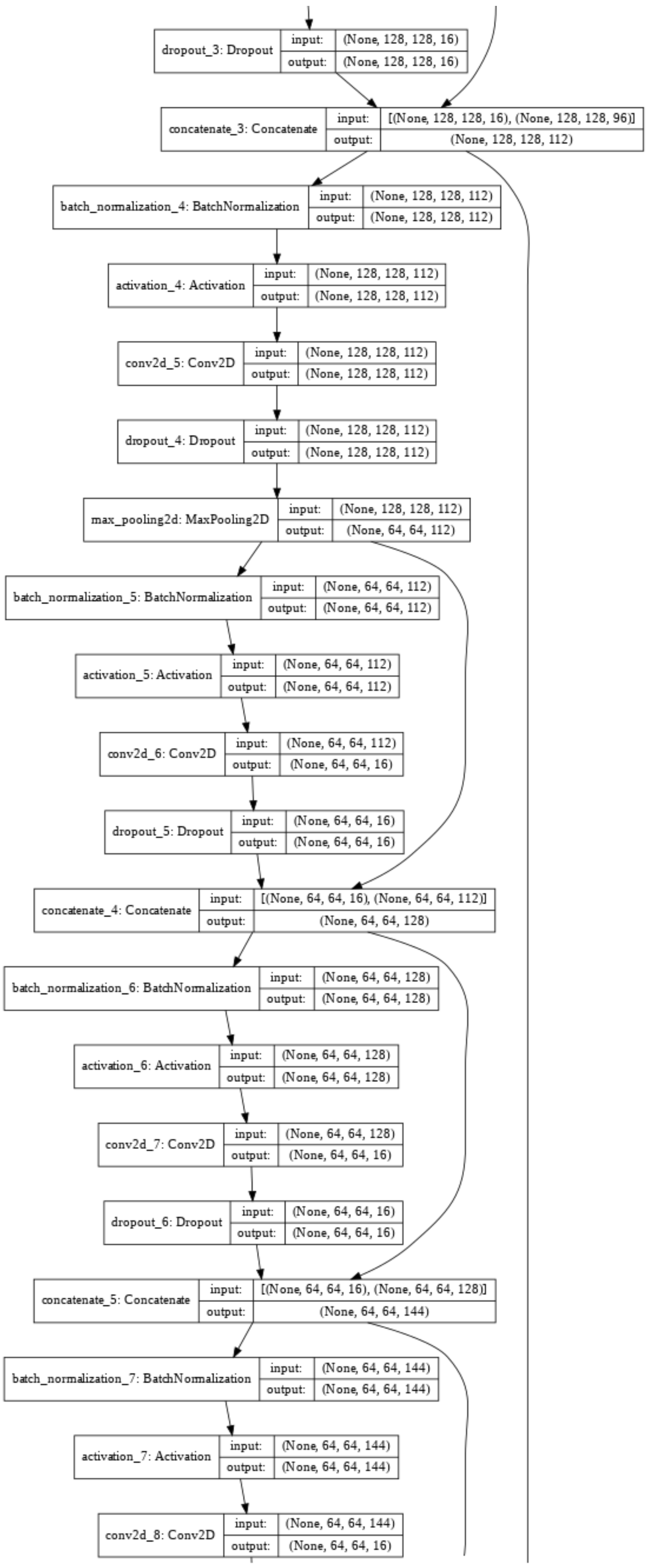

dropout_3: Dropout
input: (None, 128, 128, 16)
output: (None, 128, 128, 16)
concatenate_3: Concatenate
input: [(None, 128, 128, 16), (None, 128, 128, 96)]
output: (None, 128, 128, 112)
batch_normalization_4: BatchNormalization
input: (None, 128, 128, 112)
output: (None, 128, 128, 112)
activation_4: Activation
input: (None, 128, 128, 112)
output: (None, 128, 128, 112)
conv2d_5: Conv2D
input: (None, 128, 128, 112)
output: (None, 128, 128, 112)
dropout_4: Dropout
input: (None, 128, 128, 112)
output: (None, 128, 128, 112)
max_pooling2d: MaxPooling2D
input: (None, 128, 128, 112)
output: (None, 64, 64, 112)
batch_normalization_5: BatchNormalization
input: (None, 64, 64, 112)
output: (None, 64, 64, 112)
activation_5: Activation
input: (None, 64, 64, 112)
output: (None, 64, 64, 112)
conv2d_6: Conv2D
input: (None, 64, 64, 112)
output: (None, 64, 64, 16)
dropout_5: Dropout
input: (None, 64, 64, 16)
output: (None, 64, 64, 16)
concatenate_4: Concatenate
input: [(None, 64, 64, 16), (None, 64, 64, 112)]
output: (None, 64, 64, 128)
batch_normalization_6: BatchNormalization
input: (None, 64, 64, 128)
output: (None, 64, 64, 128)
activation_6: Activation
input: (None, 64, 64, 128)
output: (None, 64, 64, 128)
conv2d_7: Conv2D
input: (None, 64, 64, 128)
output: (None, 64, 64, 16)
dropout_6: Dropout
input: (None, 64, 64, 16)
output: (None, 64, 64, 16)
concatenate_5: Concatenate
input: [(None, 64, 64, 16), (None, 64, 64, 128)]
output: (None, 64, 64, 144)
batch_normalization_7: BatchNormalization
input: (None, 64, 64, 144)
output: (None, 64, 64, 144)
activation_7: Activation
input: (None, 64, 64, 144)
output: (None, 64, 64, 144)
conv2d_8: Conv2D
input: (None, 64, 64, 144)
output: (None, 64, 64, 16)

Figure 70: Tiramisu model architecture (part 2).

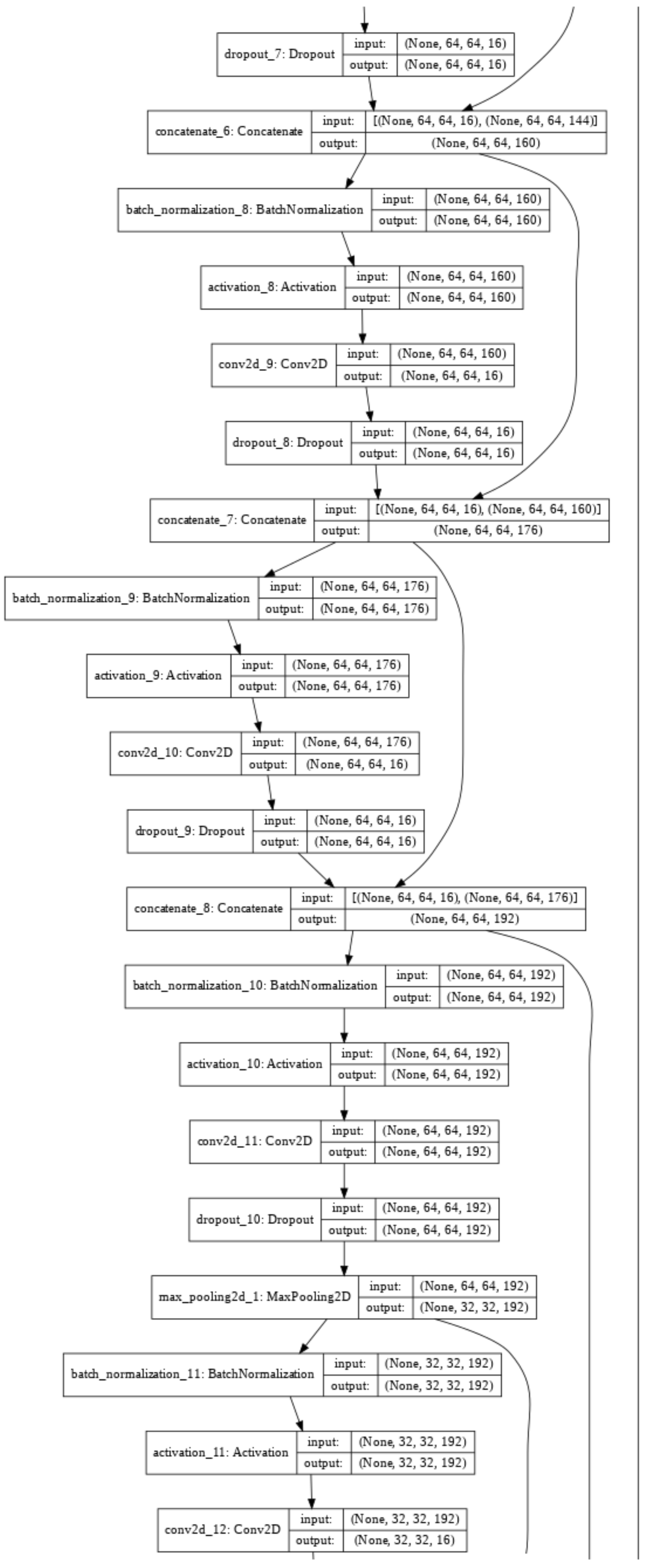

dropout_7: Dropout
input: (None, 64, 64, 16)
output: (None, 64, 64, 16)
concatenate_6: Concatenate
input: [(None, 64, 64, 16), (None, 64, 64, 144)]
output: (None, 64, 64, 160)
batch_normalization_8: BatchNormalization
input: (None, 64, 64, 160)
output: (None, 64, 64, 160)
activation_8: Activation
input: (None, 64, 64, 160)
output: (None, 64, 64, 160)
conv2d_9: Conv2D
input: (None, 64, 64, 160)
output: (None, 64, 64, 16)
dropout_8: Dropout
input: (None, 64, 64, 16)
output: (None, 64, 64, 16)
concatenate_7: Concatenate
input: [(None, 64, 64, 16), (None, 64, 64, 160)]
output: (None, 64, 64, 176)
batch_normalization_9: BatchNormalization
input: (None, 64, 64, 176)
output: (None, 64, 64, 176)
activation_9: Activation
input: (None, 64, 64, 176)
output: (None, 64, 64, 176)
conv2d_10: Conv2D
input: (None, 64, 64, 176)
output: (None, 64, 64, 16)
dropout_9: Dropout
input: (None, 64, 64, 16)
output: (None, 64, 64, 16)
concatenate_8: Concatenate
input: [(None, 64, 64, 16), (None, 64, 64, 176)]
output: (None, 64, 64, 192)
batch_normalization_10: BatchNormalization
input: (None, 64, 64, 192)
output: (None, 64, 64, 192)
activation_10: Activation
input: (None, 64, 64, 192)
output: (None, 64, 64, 192)
conv2d_11: Conv2D
input: (None, 64, 64, 192)
output: (None, 64, 64, 192)
dropout_10: Dropout
input: (None, 64, 64, 192)
output: (None, 64, 64, 192)
max_pooling2d_1: MaxPooling2D
input: (None, 64, 64, 192)
output: (None, 32, 32, 192)
batch_normalization_11: BatchNormalization
input: (None, 32, 32, 192)
output: (None, 32, 32, 192)
activation_11: Activation
input: (None, 32, 32, 192)
output: (None, 32, 32, 192)
conv2d_12: Conv2D
input: (None, 32, 32, 192)
output: (None, 32, 32, 16)

Figure 71: Tiramisu model architecture (part 3).

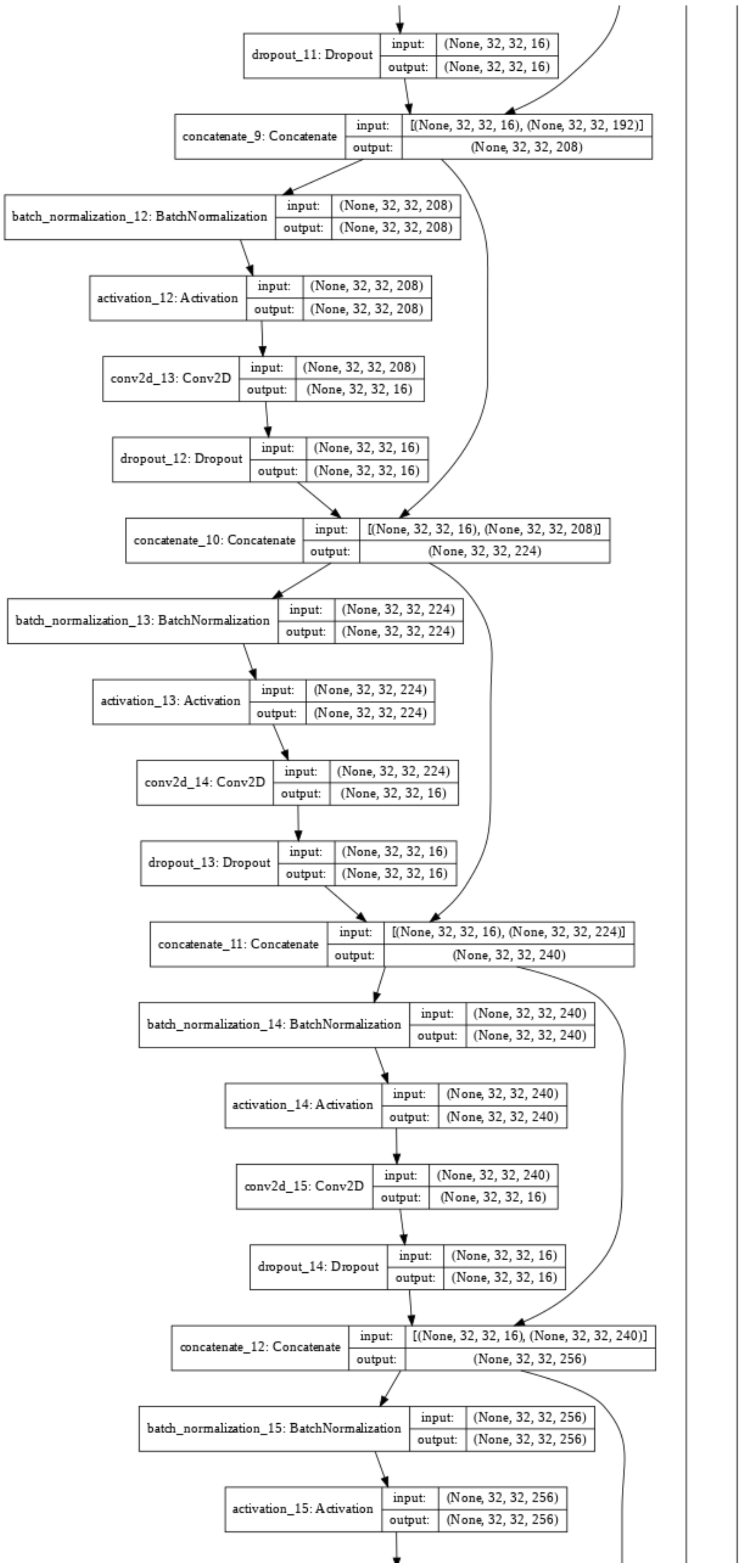

dropout_11: Dropout | input: (None, 32, 32, 16) | output: (None, 32, 32, 16)
concatenate_9: Concatenate | input: [(None, 32, 32, 16), (None, 32, 32, 192)] | output: (None, 32, 32, 208)
batch_normalization_12: BatchNormalization | input: (None, 32, 32, 208) | output: (None, 32, 32, 208)
activation_12: Activation | input: (None, 32, 32, 208) | output: (None, 32, 32, 208)
conv2d_13: Conv2D | input: (None, 32, 32, 208) | output: (None, 32, 32, 16)
dropout_12: Dropout | input: (None, 32, 32, 16) | output: (None, 32, 32, 16)
concatenate_10: Concatenate | input: [(None, 32, 32, 16), (None, 32, 32, 208)] | output: (None, 32, 32, 224)
batch_normalization_13: BatchNormalization | input: (None, 32, 32, 224) | output: (None, 32, 32, 224)
activation_13: Activation | input: (None, 32, 32, 224) | output: (None, 32, 32, 224)
conv2d_14: Conv2D | input: (None, 32, 32, 224) | output: (None, 32, 32, 16)
dropout_13: Dropout | input: (None, 32, 32, 16) | output: (None, 32, 32, 16)
concatenate_11: Concatenate | input: [(None, 32, 32, 16), (None, 32, 32, 224)] | output: (None, 32, 32, 240)
batch_normalization_14: BatchNormalization | input: (None, 32, 32, 240) | output: (None, 32, 32, 240)
activation_14: Activation | input: (None, 32, 32, 240) | output: (None, 32, 32, 240)
conv2d_15: Conv2D | input: (None, 32, 32, 240) | output: (None, 32, 32, 16)
dropout_14: Dropout | input: (None, 32, 32, 16) | output: (None, 32, 32, 16)
concatenate_12: Concatenate | input: [(None, 32, 32, 16), (None, 32, 32, 240)] | output: (None, 32, 32, 256)
batch_normalization_15: BatchNormalization | input: (None, 32, 32, 256) | output: (None, 32, 32, 256)
activation_15: Activation | input: (None, 32, 32, 256) | output: (None, 32, 32, 256)

Figure 72: Tiramisu model architecture (part 4).

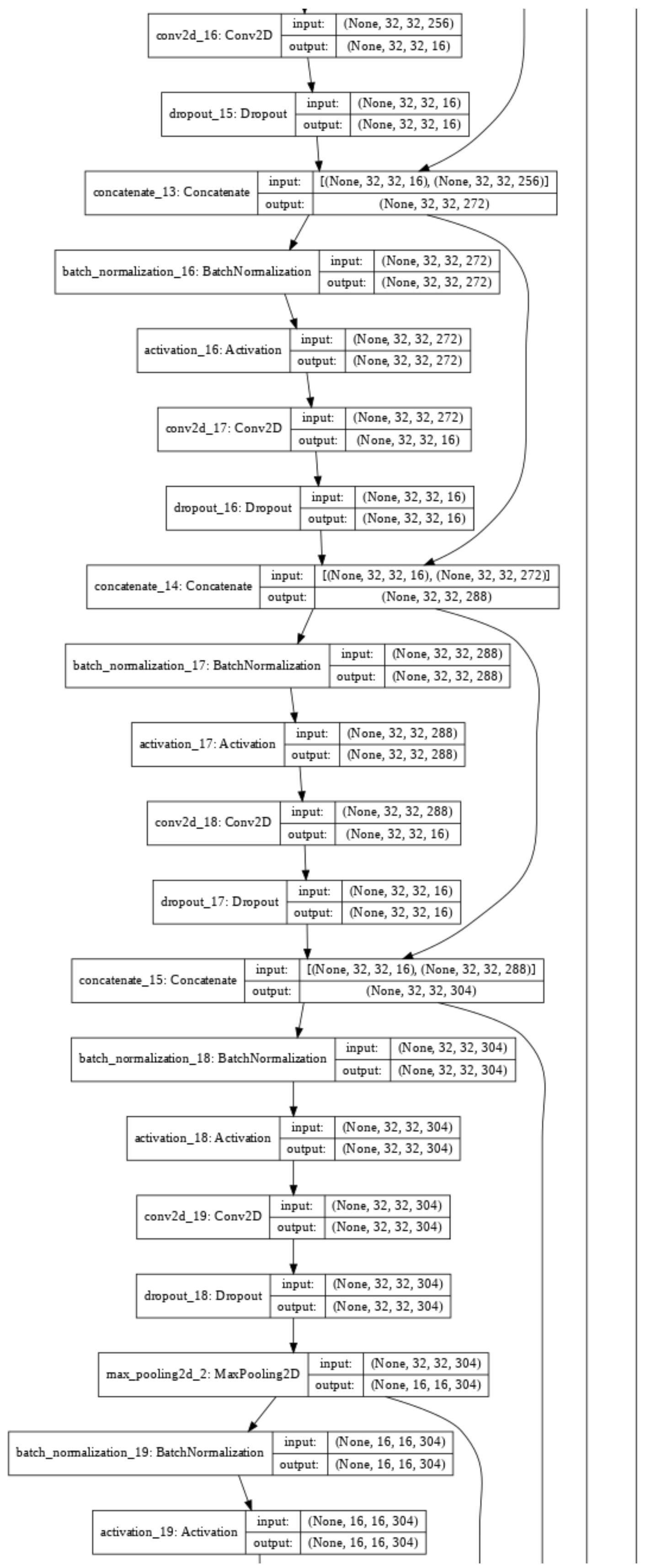

conv2d_16: Conv2D | input: (None, 32, 32, 256) | output: (None, 32, 32, 16)
dropout_15: Dropout | input: (None, 32, 32, 16) | output: (None, 32, 32, 16)
concatenate_13: Concatenate | input: [(None, 32, 32, 16), (None, 32, 32, 256)] | output: (None, 32, 32, 272)
batch_normalization_16: BatchNormalization | input: (None, 32, 32, 272) | output: (None, 32, 32, 272)
activation_16: Activation | input: (None, 32, 32, 272) | output: (None, 32, 32, 272)
conv2d_17: Conv2D | input: (None, 32, 32, 272) | output: (None, 32, 32, 16)
dropout_16: Dropout | input: (None, 32, 32, 16) | output: (None, 32, 32, 16)
concatenate_14: Concatenate | input: [(None, 32, 32, 16), (None, 32, 32, 272)] | output: (None, 32, 32, 288)
batch_normalization_17: BatchNormalization | input: (None, 32, 32, 288) | output: (None, 32, 32, 288)
activation_17: Activation | input: (None, 32, 32, 288) | output: (None, 32, 32, 288)
conv2d_18: Conv2D | input: (None, 32, 32, 288) | output: (None, 32, 32, 16)
dropout_17: Dropout | input: (None, 32, 32, 16) | output: (None, 32, 32, 16)
concatenate_15: Concatenate | input: [(None, 32, 32, 16), (None, 32, 32, 288)] | output: (None, 32, 32, 304)
batch_normalization_18: BatchNormalization | input: (None, 32, 32, 304) | output: (None, 32, 32, 304)
activation_18: Activation | input: (None, 32, 32, 304) | output: (None, 32, 32, 304)
conv2d_19: Conv2D | input: (None, 32, 32, 304) | output: (None, 32, 32, 304)
dropout_18: Dropout | input: (None, 32, 32, 304) | output: (None, 32, 32, 304)
max_pooling2d_2: MaxPooling2D | input: (None, 32, 32, 304) | output: (None, 16, 16, 304)
batch_normalization_19: BatchNormalization | input: (None, 16, 16, 304) | output: (None, 16, 16, 304)
activation_19: Activation | input: (None, 16, 16, 304) | output: (None, 16, 16, 304)

Figure 73: Tiramisu model architecture (part 5).

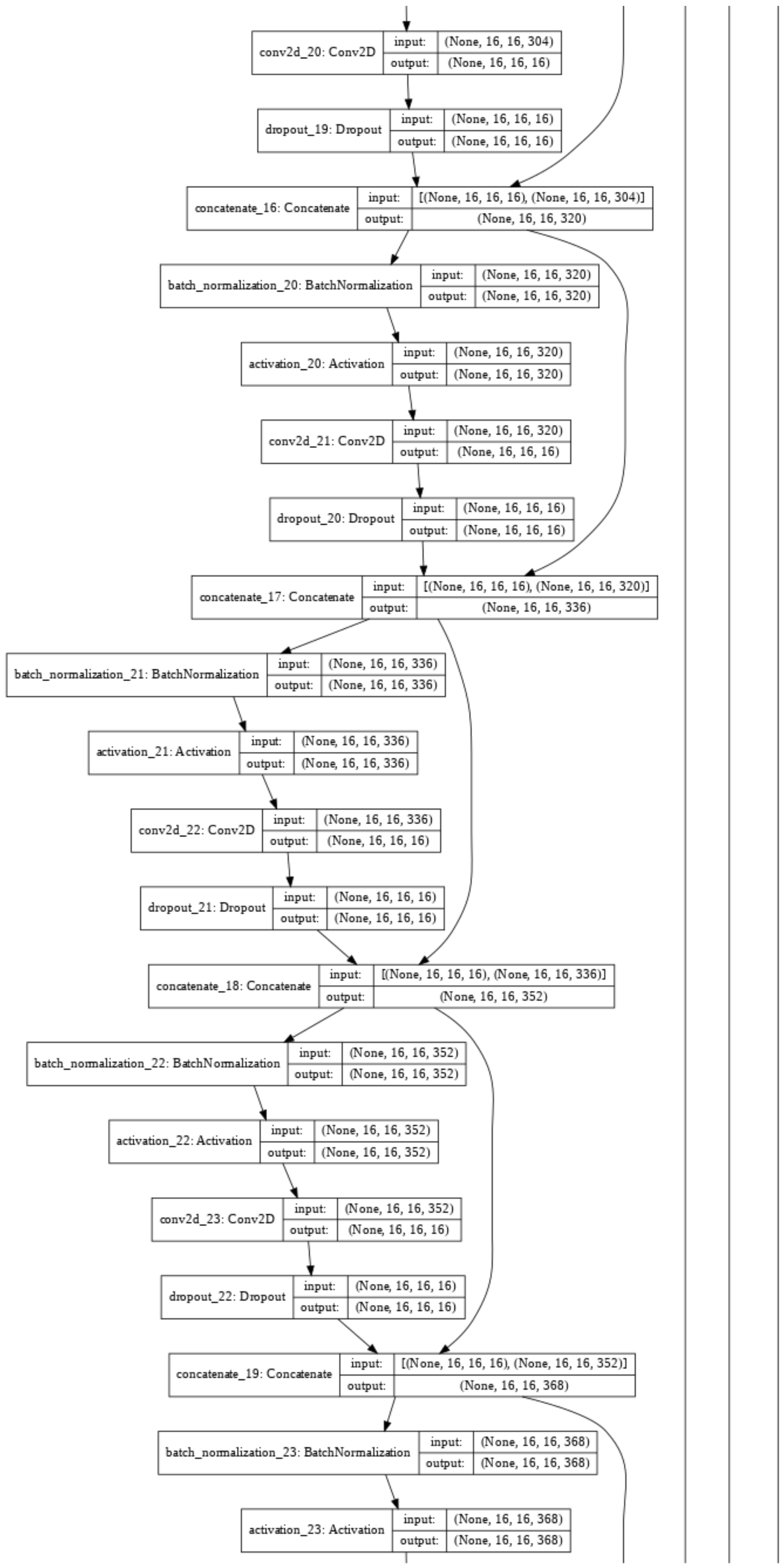

conv2d_20: Conv2D
input: (None, 16, 16, 304)
output: (None, 16, 16, 16)
dropout_19: Dropout
input: (None, 16, 16, 16)
output: (None, 16, 16, 16)
concatenate_16: Concatenate
input: [(None, 16, 16, 16), (None, 16, 16, 304)]
output: (None, 16, 16, 320)
batch_normalization_20: BatchNormalization
input: (None, 16, 16, 320)
output: (None, 16, 16, 320)
activation_20: Activation
input: (None, 16, 16, 320)
output: (None, 16, 16, 320)
conv2d_21: Conv2D
input: (None, 16, 16, 320)
output: (None, 16, 16, 16)
dropout_20: Dropout
input: (None, 16, 16, 16)
output: (None, 16, 16, 16)
concatenate_17: Concatenate
input: [(None, 16, 16, 16), (None, 16, 16, 320)]
output: (None, 16, 16, 336)
batch_normalization_21: BatchNormalization
input: (None, 16, 16, 336)
output: (None, 16, 16, 336)
activation_21: Activation
input: (None, 16, 16, 336)
output: (None, 16, 16, 336)
conv2d_22: Conv2D
input: (None, 16, 16, 336)
output: (None, 16, 16, 16)
dropout_21: Dropout
input: (None, 16, 16, 16)
output: (None, 16, 16, 16)
concatenate_18: Concatenate
input: [(None, 16, 16, 16), (None, 16, 16, 336)]
output: (None, 16, 16, 352)
batch_normalization_22: BatchNormalization
input: (None, 16, 16, 352)
output: (None, 16, 16, 352)
activation_22: Activation
input: (None, 16, 16, 352)
output: (None, 16, 16, 352)
conv2d_23: Conv2D
input: (None, 16, 16, 352)
output: (None, 16, 16, 16)
dropout_22: Dropout
input: (None, 16, 16, 16)
output: (None, 16, 16, 16)
concatenate_19: Concatenate
input: [(None, 16, 16, 16), (None, 16, 16, 352)]
output: (None, 16, 16, 368)
batch_normalization_23: BatchNormalization
input: (None, 16, 16, 368)
output: (None, 16, 16, 368)
activation_23: Activation
input: (None, 16, 16, 368)
output: (None, 16, 16, 368)

Figure 74: Tiramisu model architecture (part 6).

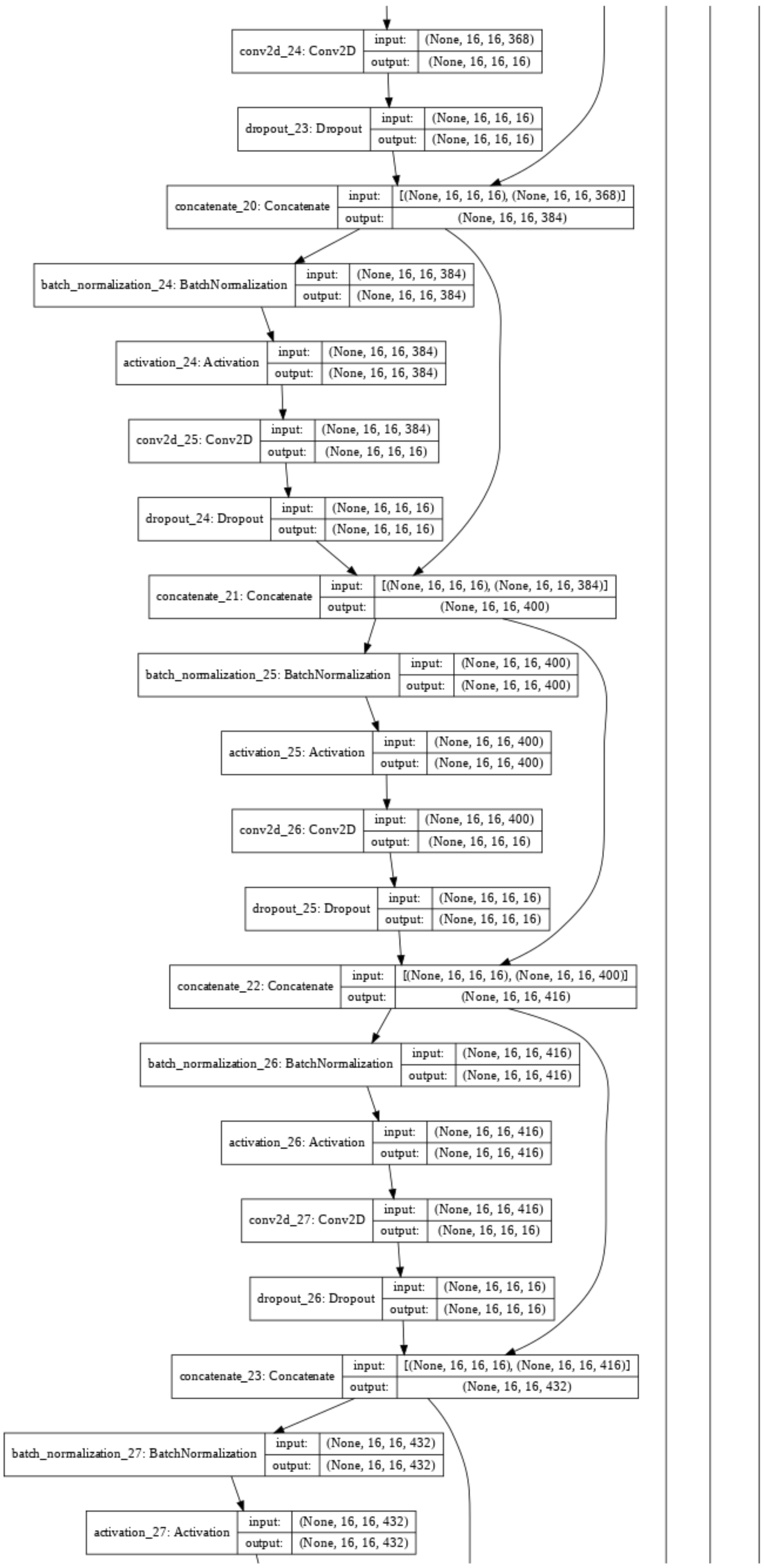

conv2d_24: Conv2D
input: (None, 16, 16, 368)
output: (None, 16, 16, 16)
dropout_23: Dropout
input: (None, 16, 16, 16)
output: (None, 16, 16, 16)
concatenate_20: Concatenate
input: [(None, 16, 16, 16), (None, 16, 16, 368)]
output: (None, 16, 16, 384)
batch_normalization_24: BatchNormalization
input: (None, 16, 16, 384)
output: (None, 16, 16, 384)
activation_24: Activation
input: (None, 16, 16, 384)
output: (None, 16, 16, 384)
conv2d_25: Conv2D
input: (None, 16, 16, 384)
output: (None, 16, 16, 16)
dropout_24: Dropout
input: (None, 16, 16, 16)
output: (None, 16, 16, 16)
concatenate_21: Concatenate
input: [(None, 16, 16, 16), (None, 16, 16, 384)]
output: (None, 16, 16, 400)
batch_normalization_25: BatchNormalization
input: (None, 16, 16, 400)
output: (None, 16, 16, 400)
activation_25: Activation
input: (None, 16, 16, 400)
output: (None, 16, 16, 400)
conv2d_26: Conv2D
input: (None, 16, 16, 400)
output: (None, 16, 16, 16)
dropout_25: Dropout
input: (None, 16, 16, 16)
output: (None, 16, 16, 16)
concatenate_22: Concatenate
input: [(None, 16, 16, 16), (None, 16, 16, 400)]
output: (None, 16, 16, 416)
batch_normalization_26: BatchNormalization
input: (None, 16, 16, 416)
output: (None, 16, 16, 416)
activation_26: Activation
input: (None, 16, 16, 416)
output: (None, 16, 16, 416)
conv2d_27: Conv2D
input: (None, 16, 16, 416)
output: (None, 16, 16, 16)
dropout_26: Dropout
input: (None, 16, 16, 16)
output: (None, 16, 16, 16)
concatenate_23: Concatenate
input: [(None, 16, 16, 16), (None, 16, 16, 416)]
output: (None, 16, 16, 432)
batch_normalization_27: BatchNormalization
input: (None, 16, 16, 432)
output: (None, 16, 16, 432)
activation_27: Activation
input: (None, 16, 16, 432)
output: (None, 16, 16, 432)

Figure 75: Tiramisu model architecture (part 7).

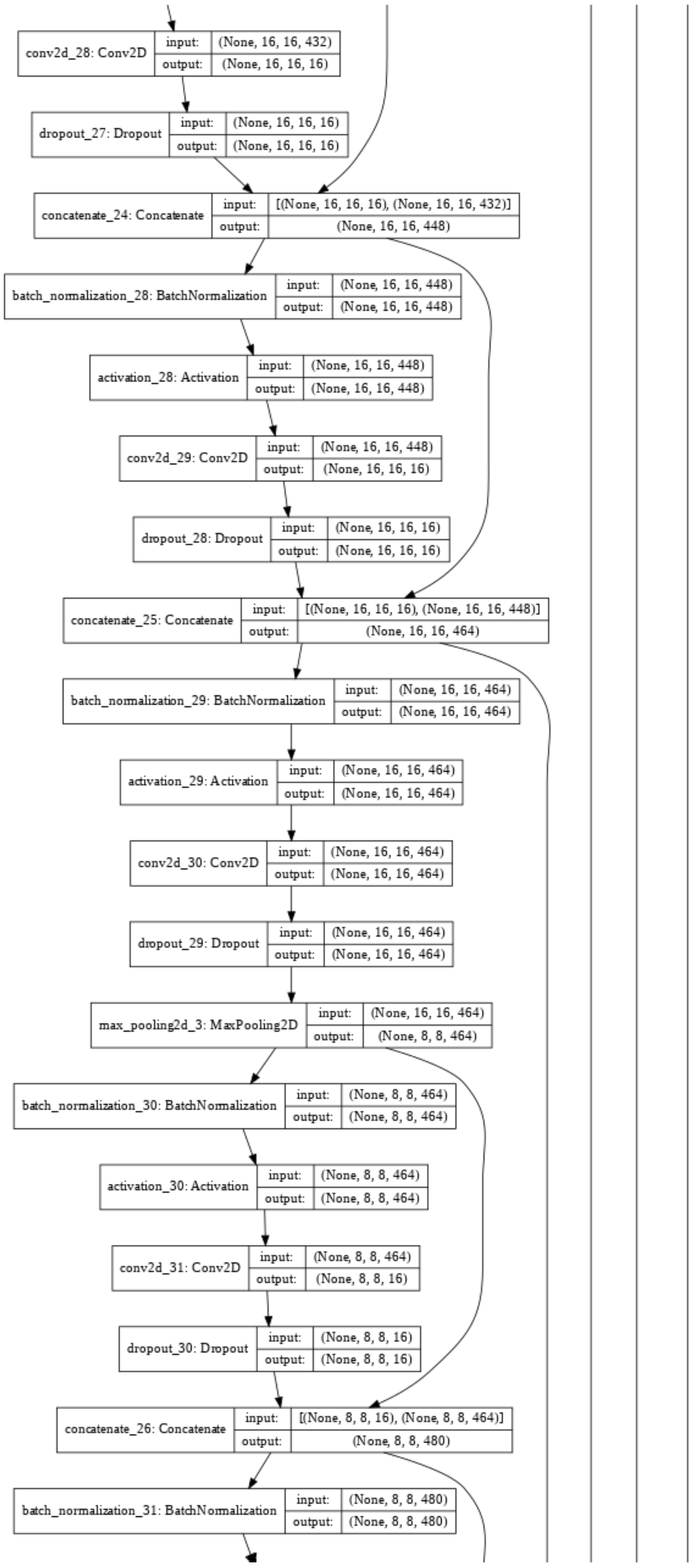

conv2d_28: Conv2D
input: (None, 16, 16, 432)
output: (None, 16, 16, 16)
dropout_27: Dropout
input: (None, 16, 16, 16)
output: (None, 16, 16, 16)
concatenate_24: Concatenate
input: [(None, 16, 16, 16), (None, 16, 16, 432)]
output: (None, 16, 16, 448)
batch_normalization_28: BatchNormalization
input: (None, 16, 16, 448)
output: (None, 16, 16, 448)
activation_28: Activation
input: (None, 16, 16, 448)
output: (None, 16, 16, 448)
conv2d_29: Conv2D
input: (None, 16, 16, 448)
output: (None, 16, 16, 16)
dropout_28: Dropout
input: (None, 16, 16, 16)
output: (None, 16, 16, 16)
concatenate_25: Concatenate
input: [(None, 16, 16, 16), (None, 16, 16, 448)]
output: (None, 16, 16, 464)
batch_normalization_29: BatchNormalization
input: (None, 16, 16, 464)
output: (None, 16, 16, 464)
activation_29: Activation
input: (None, 16, 16, 464)
output: (None, 16, 16, 464)
conv2d_30: Conv2D
input: (None, 16, 16, 464)
output: (None, 16, 16, 464)
dropout_29: Dropout
input: (None, 16, 16, 464)
output: (None, 16, 16, 464)
max_pooling2d_3: MaxPooling2D
input: (None, 16, 16, 464)
output: (None, 8, 8, 464)
batch_normalization_30: BatchNormalization
input: (None, 8, 8, 464)
output: (None, 8, 8, 464)
activation_30: Activation
input: (None, 8, 8, 464)
output: (None, 8, 8, 464)
conv2d_31: Conv2D
input: (None, 8, 8, 464)
output: (None, 8, 8, 16)
dropout_30: Dropout
input: (None, 8, 8, 16)
output: (None, 8, 8, 16)
concatenate_26: Concatenate
input: [(None, 8, 8, 16), (None, 8, 8, 464)]
output: (None, 8, 8, 480)
batch_normalization_31: BatchNormalization
input: (None, 8, 8, 480)
output: (None, 8, 8, 480)

Figure 76: Tiramisu model architecture (part 8).

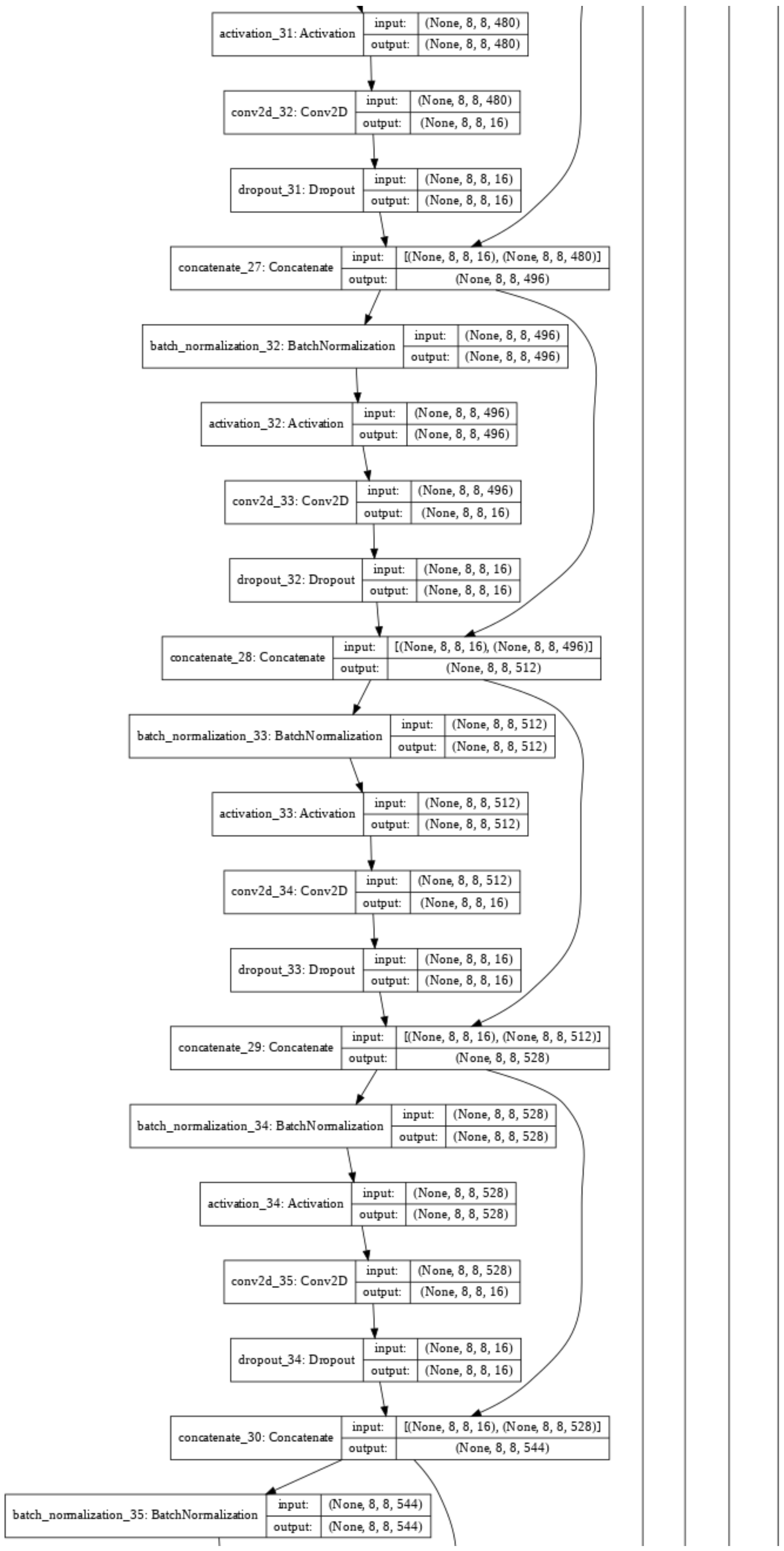

activation_31: Activation
input: (None, 8, 8, 480)
output: (None, 8, 8, 480)
conv2d_32: Conv2D
input: (None, 8, 8, 480)
output: (None, 8, 8, 16)
dropout_31: Dropout
input: (None, 8, 8, 16)
output: (None, 8, 8, 16)
concatenate_27: Concatenate
input: [(None, 8, 8, 16), (None, 8, 8, 480)]
output: (None, 8, 8, 496)
batch_normalization_32: BatchNormalization
input: (None, 8, 8, 496)
output: (None, 8, 8, 496)
activation_32: Activation
input: (None, 8, 8, 496)
output: (None, 8, 8, 496)
conv2d_33: Conv2D
input: (None, 8, 8, 496)
output: (None, 8, 8, 16)
dropout_32: Dropout
input: (None, 8, 8, 16)
output: (None, 8, 8, 16)
concatenate_28: Concatenate
input: [(None, 8, 8, 16), (None, 8, 8, 496)]
output: (None, 8, 8, 512)
batch_normalization_33: BatchNormalization
input: (None, 8, 8, 512)
output: (None, 8, 8, 512)
activation_33: Activation
input: (None, 8, 8, 512)
output: (None, 8, 8, 512)
conv2d_34: Conv2D
input: (None, 8, 8, 512)
output: (None, 8, 8, 16)
dropout_33: Dropout
input: (None, 8, 8, 16)
output: (None, 8, 8, 16)
concatenate_29: Concatenate
input: [(None, 8, 8, 16), (None, 8, 8, 512)]
output: (None, 8, 8, 528)
batch_normalization_34: BatchNormalization
input: (None, 8, 8, 528)
output: (None, 8, 8, 528)
activation_34: Activation
input: (None, 8, 8, 528)
output: (None, 8, 8, 528)
conv2d_35: Conv2D
input: (None, 8, 8, 528)
output: (None, 8, 8, 16)
dropout_34: Dropout
input: (None, 8, 8, 16)
output: (None, 8, 8, 16)
concatenate_30: Concatenate
input: [(None, 8, 8, 16), (None, 8, 8, 528)]
output: (None, 8, 8, 544)
batch_normalization_35: BatchNormalization
input: (None, 8, 8, 544)
output: (None, 8, 8, 544)

Figure 77: Tiramisu model architecture (part 9).

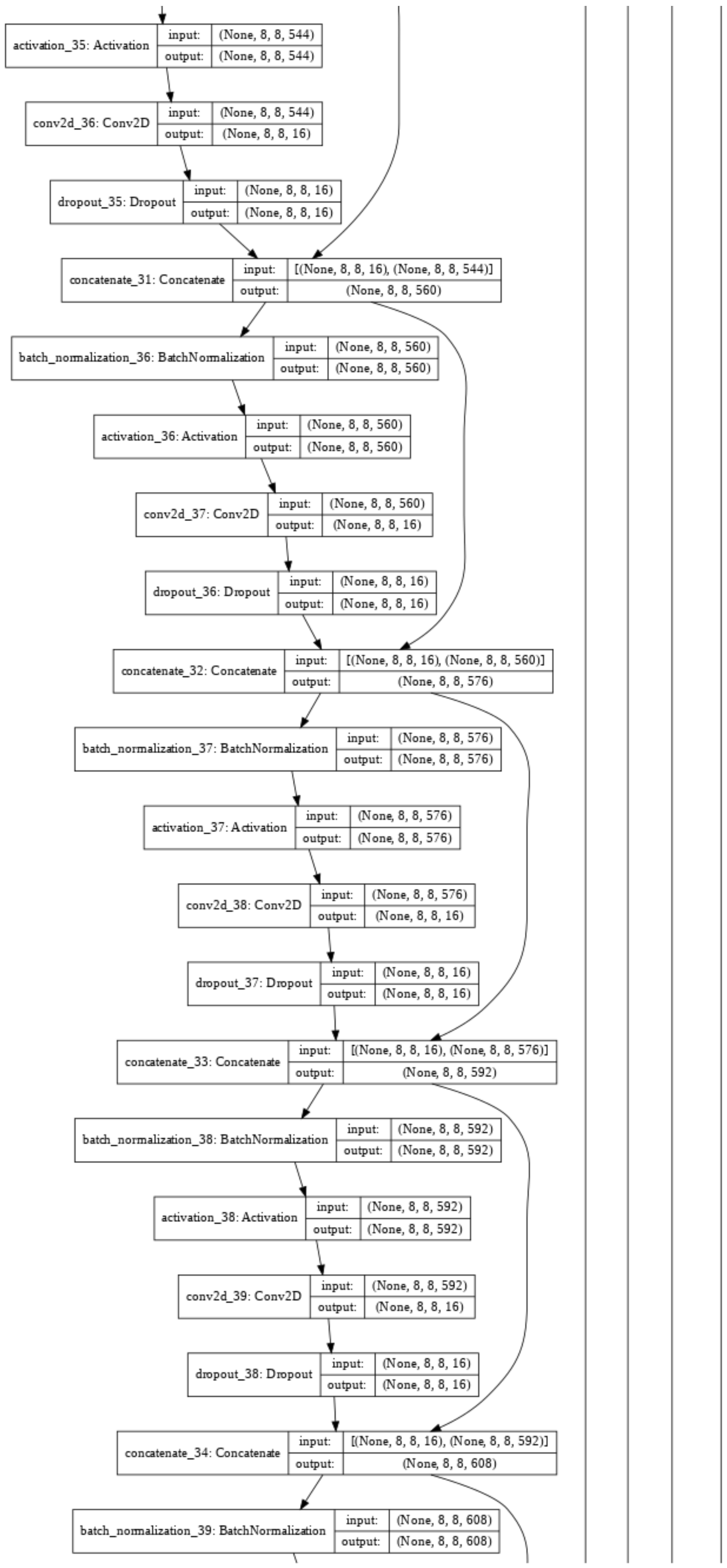

activation_35: Activation | input: (None, 8, 8, 544) | output: (None, 8, 8, 544)
conv2d_36: Conv2D | input: (None, 8, 8, 544) | output: (None, 8, 8, 16)
dropout_35: Dropout | input: (None, 8, 8, 16) | output: (None, 8, 8, 16)
concatenate_31: Concatenate | input: [(None, 8, 8, 16), (None, 8, 8, 544)] | output: (None, 8, 8, 560)
batch_normalization_36: BatchNormalization | input: (None, 8, 8, 560) | output: (None, 8, 8, 560)
activation_36: Activation | input: (None, 8, 8, 560) | output: (None, 8, 8, 560)
conv2d_37: Conv2D | input: (None, 8, 8, 560) | output: (None, 8, 8, 16)
dropout_36: Dropout | input: (None, 8, 8, 16) | output: (None, 8, 8, 16)
concatenate_32: Concatenate | input: [(None, 8, 8, 16), (None, 8, 8, 560)] | output: (None, 8, 8, 576)
batch_normalization_37: BatchNormalization | input: (None, 8, 8, 576) | output: (None, 8, 8, 576)
activation_37: Activation | input: (None, 8, 8, 576) | output: (None, 8, 8, 576)
conv2d_38: Conv2D | input: (None, 8, 8, 576) | output: (None, 8, 8, 16)
dropout_37: Dropout | input: (None, 8, 8, 16) | output: (None, 8, 8, 16)
concatenate_33: Concatenate | input: [(None, 8, 8, 16), (None, 8, 8, 576)] | output: (None, 8, 8, 592)
batch_normalization_38: BatchNormalization | input: (None, 8, 8, 592) | output: (None, 8, 8, 592)
activation_38: Activation | input: (None, 8, 8, 592) | output: (None, 8, 8, 592)
conv2d_39: Conv2D | input: (None, 8, 8, 592) | output: (None, 8, 8, 16)
dropout_38: Dropout | input: (None, 8, 8, 16) | output: (None, 8, 8, 16)
concatenate_34: Concatenate | input: [(None, 8, 8, 16), (None, 8, 8, 592)] | output: (None, 8, 8, 608)
batch_normalization_39: BatchNormalization | input: (None, 8, 8, 608) | output: (None, 8, 8, 608)

Figure 78: Tiramisu model architecture (part 10).

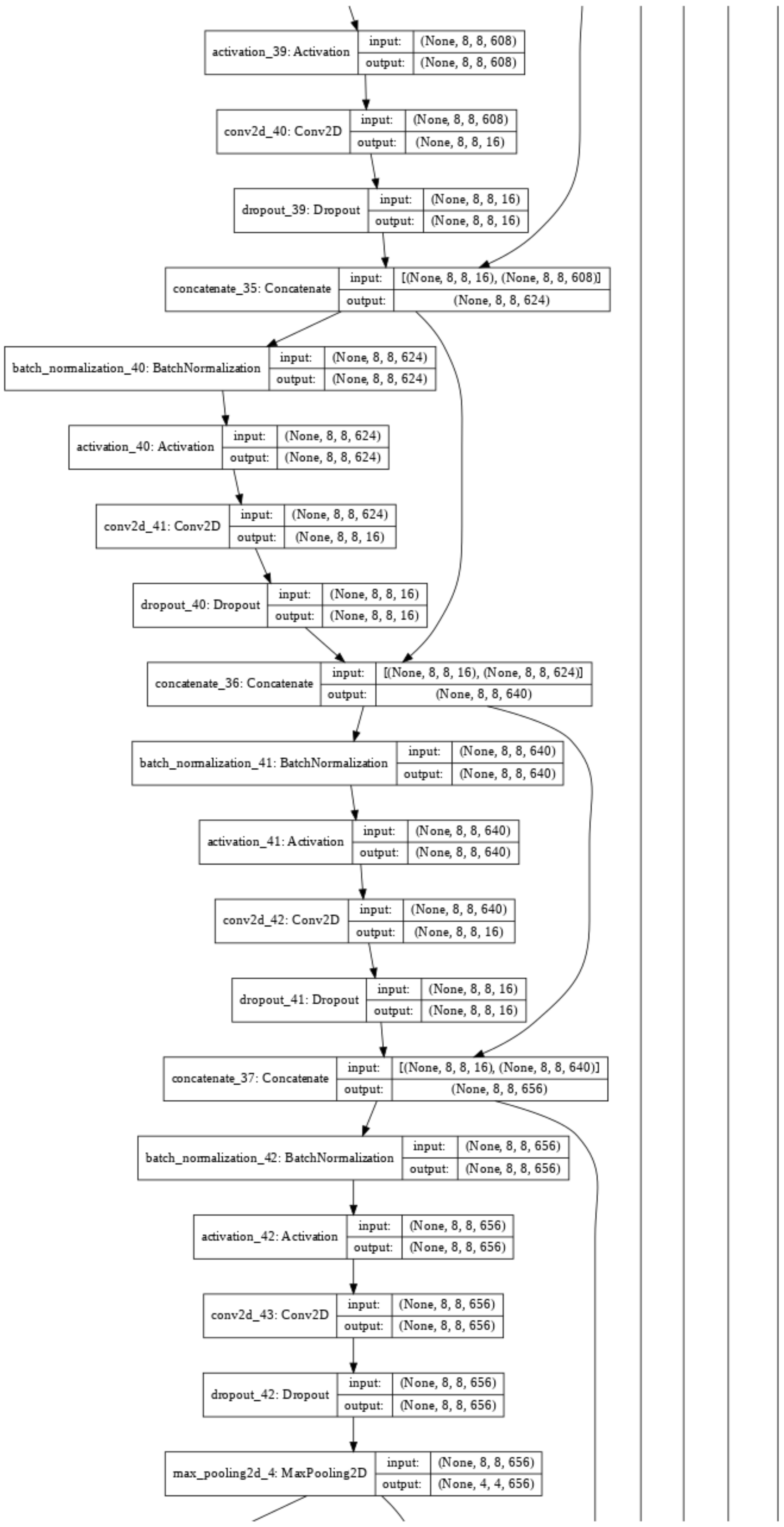
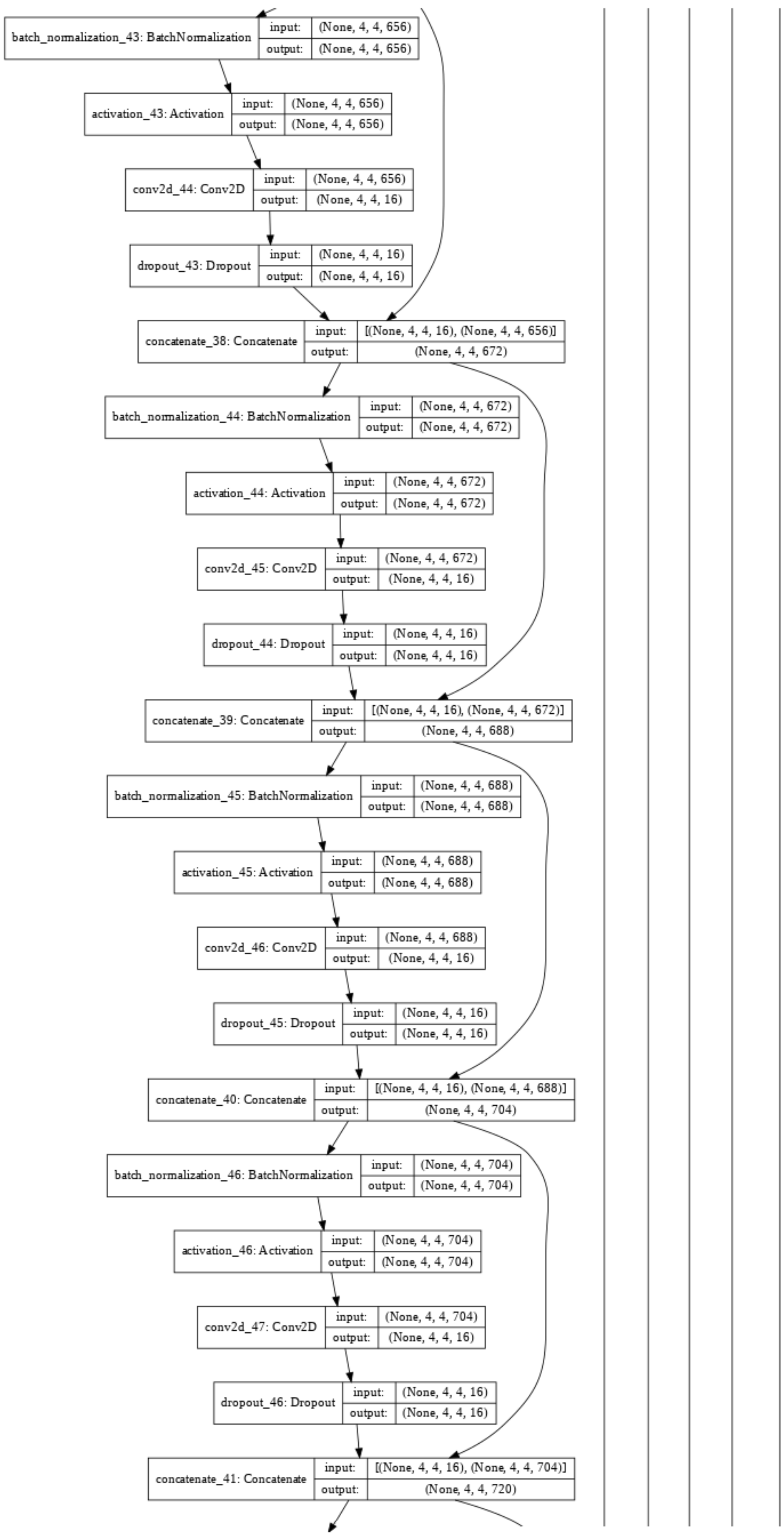

activation_39: Activation
input: (None, 8, 8, 608)
output: (None, 8, 8, 608)
conv2d_40: Conv2D
input: (None, 8, 8, 608)
output: (None, 8, 8, 16)
dropout_39: Dropout
input: (None, 8, 8, 16)
output: (None, 8, 8, 16)
concatenate_35: Concatenate
input: [(None, 8, 8, 16), (None, 8, 8, 608)]
output: (None, 8, 8, 624)
batch_normalization_40: BatchNormalization
input: (None, 8, 8, 624)
output: (None, 8, 8, 624)
activation_40: Activation
input: (None, 8, 8, 624)
output: (None, 8, 8, 624)
conv2d_41: Conv2D
input: (None, 8, 8, 624)
output: (None, 8, 8, 16)
dropout_40: Dropout
input: (None, 8, 8, 16)
output: (None, 8, 8, 16)
concatenate_36: Concatenate
input: [(None, 8, 8, 16), (None, 8, 8, 624)]
output: (None, 8, 8, 640)
batch_normalization_41: BatchNormalization
input: (None, 8, 8, 640)
output: (None, 8, 8, 640)
activation_41: Activation
input: (None, 8, 8, 640)
output: (None, 8, 8, 640)
conv2d_42: Conv2D
input: (None, 8, 8, 640)
output: (None, 8, 8, 16)
dropout_41: Dropout
input: (None, 8, 8, 16)
output: (None, 8, 8, 16)
concatenate_37: Concatenate
input: [(None, 8, 8, 16), (None, 8, 8, 640)]
output: (None, 8, 8, 656)
batch_normalization_42: BatchNormalization
input: (None, 8, 8, 656)
output: (None, 8, 8, 656)
activation_42: Activation
input: (None, 8, 8, 656)
output: (None, 8, 8, 656)
conv2d_43: Conv2D
input: (None, 8, 8, 656)
output: (None, 8, 8, 656)
dropout_42: Dropout
input: (None, 8, 8, 656)
output: (None, 8, 8, 656)
max_pooling2d_4: MaxPooling2D
input: (None, 8, 8, 656)
output: (None, 4, 4, 656)

Figure 79: Tiramisu model architecture (part 11).



batch_normalization_43: BatchNormalization | input: (None, 4, 4, 656) | output: (None, 4, 4, 656)
activation_43: Activation | input: (None, 4, 4, 656) | output: (None, 4, 4, 656)
conv2d_44: Conv2D | input: (None, 4, 4, 656) | output: (None, 4, 4, 16)
dropout_43: Dropout | input: (None, 4, 4, 16) | output: (None, 4, 4, 16)
concatenate_38: Concatenate | input: [(None, 4, 4, 16), (None, 4, 4, 656)] | output: (None, 4, 4, 672)
batch_normalization_44: BatchNormalization | input: (None, 4, 4, 672) | output: (None, 4, 4, 672)
activation_44: Activation | input: (None, 4, 4, 672) | output: (None, 4, 4, 672)
conv2d_45: Conv2D | input: (None, 4, 4, 672) | output: (None, 4, 4, 16)
dropout_44: Dropout | input: (None, 4, 4, 16) | output: (None, 4, 4, 16)
concatenate_39: Concatenate | input: [(None, 4, 4, 16), (None, 4, 4, 672)] | output: (None, 4, 4, 688)
batch_normalization_45: BatchNormalization | input: (None, 4, 4, 688) | output: (None, 4, 4, 688)
activation_45: Activation | input: (None, 4, 4, 688) | output: (None, 4, 4, 688)
conv2d_46: Conv2D | input: (None, 4, 4, 688) | output: (None, 4, 4, 16)
dropout_45: Dropout | input: (None, 4, 4, 16) | output: (None, 4, 4, 16)
concatenate_40: Concatenate | input: [(None, 4, 4, 16), (None, 4, 4, 688)] | output: (None, 4, 4, 704)
batch_normalization_46: BatchNormalization | input: (None, 4, 4, 704) | output: (None, 4, 4, 704)
activation_46: Activation | input: (None, 4, 4, 704) | output: (None, 4, 4, 704)
conv2d_47: Conv2D | input: (None, 4, 4, 704) | output: (None, 4, 4, 16)
dropout_46: Dropout | input: (None, 4, 4, 16) | output: (None, 4, 4, 16)
concatenate_41: Concatenate | input: [(None, 4, 4, 16), (None, 4, 4, 704)] | output: (None, 4, 4, 720)

Figure 80: Tiramisu model architecture (part 12).

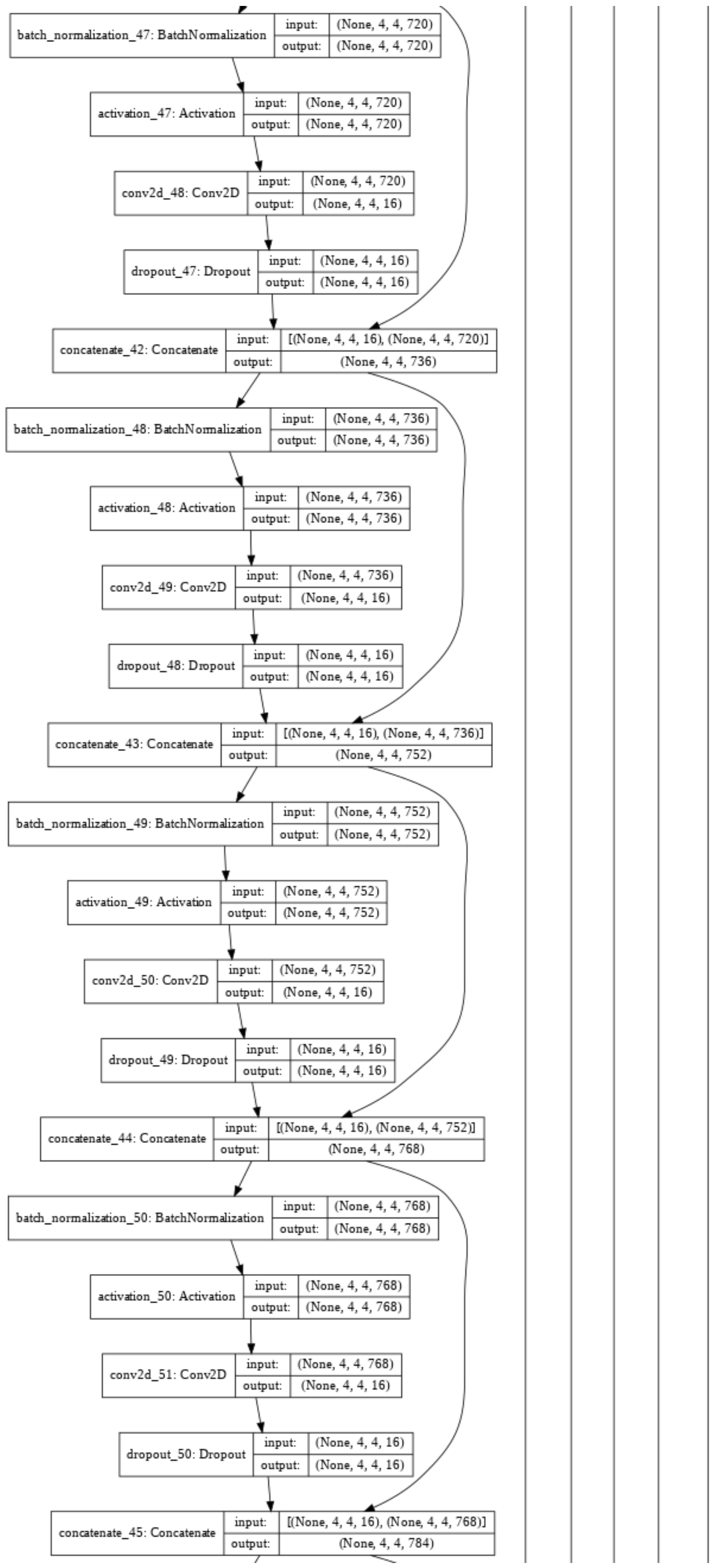

batch_normalization_47: BatchNormalization
input: (None, 4, 4, 720)
output: (None, 4, 4, 720)
activation_47: Activation
input: (None, 4, 4, 720)
output: (None, 4, 4, 720)
conv2d_48: Conv2D
input: (None, 4, 4, 720)
output: (None, 4, 4, 16)
dropout_47: Dropout
input: (None, 4, 4, 16)
output: (None, 4, 4, 16)
concatenate_42: Concatenate
input: [(None, 4, 4, 16), (None, 4, 4, 720)]
output: (None, 4, 4, 736)
batch_normalization_48: BatchNormalization
input: (None, 4, 4, 736)
output: (None, 4, 4, 736)
activation_48: Activation
input: (None, 4, 4, 736)
output: (None, 4, 4, 736)
conv2d_49: Conv2D
input: (None, 4, 4, 736)
output: (None, 4, 4, 16)
dropout_48: Dropout
input: (None, 4, 4, 16)
output: (None, 4, 4, 16)
concatenate_43: Concatenate
input: [(None, 4, 4, 16), (None, 4, 4, 736)]
output: (None, 4, 4, 752)
batch_normalization_49: BatchNormalization
input: (None, 4, 4, 752)
output: (None, 4, 4, 752)
activation_49: Activation
input: (None, 4, 4, 752)
output: (None, 4, 4, 752)
conv2d_50: Conv2D
input: (None, 4, 4, 752)
output: (None, 4, 4, 16)
dropout_49: Dropout
input: (None, 4, 4, 16)
output: (None, 4, 4, 16)
concatenate_44: Concatenate
input: [(None, 4, 4, 16), (None, 4, 4, 752)]
output: (None, 4, 4, 768)
batch_normalization_50: BatchNormalization
input: (None, 4, 4, 768)
output: (None, 4, 4, 768)
activation_50: Activation
input: (None, 4, 4, 768)
output: (None, 4, 4, 768)
conv2d_51: Conv2D
input: (None, 4, 4, 768)
output: (None, 4, 4, 16)
dropout_50: Dropout
input: (None, 4, 4, 16)
output: (None, 4, 4, 16)
concatenate_45: Concatenate
input: [(None, 4, 4, 16), (None, 4, 4, 768)]
output: (None, 4, 4, 784)

Figure 81: Tiramisu model architecture (part 13).

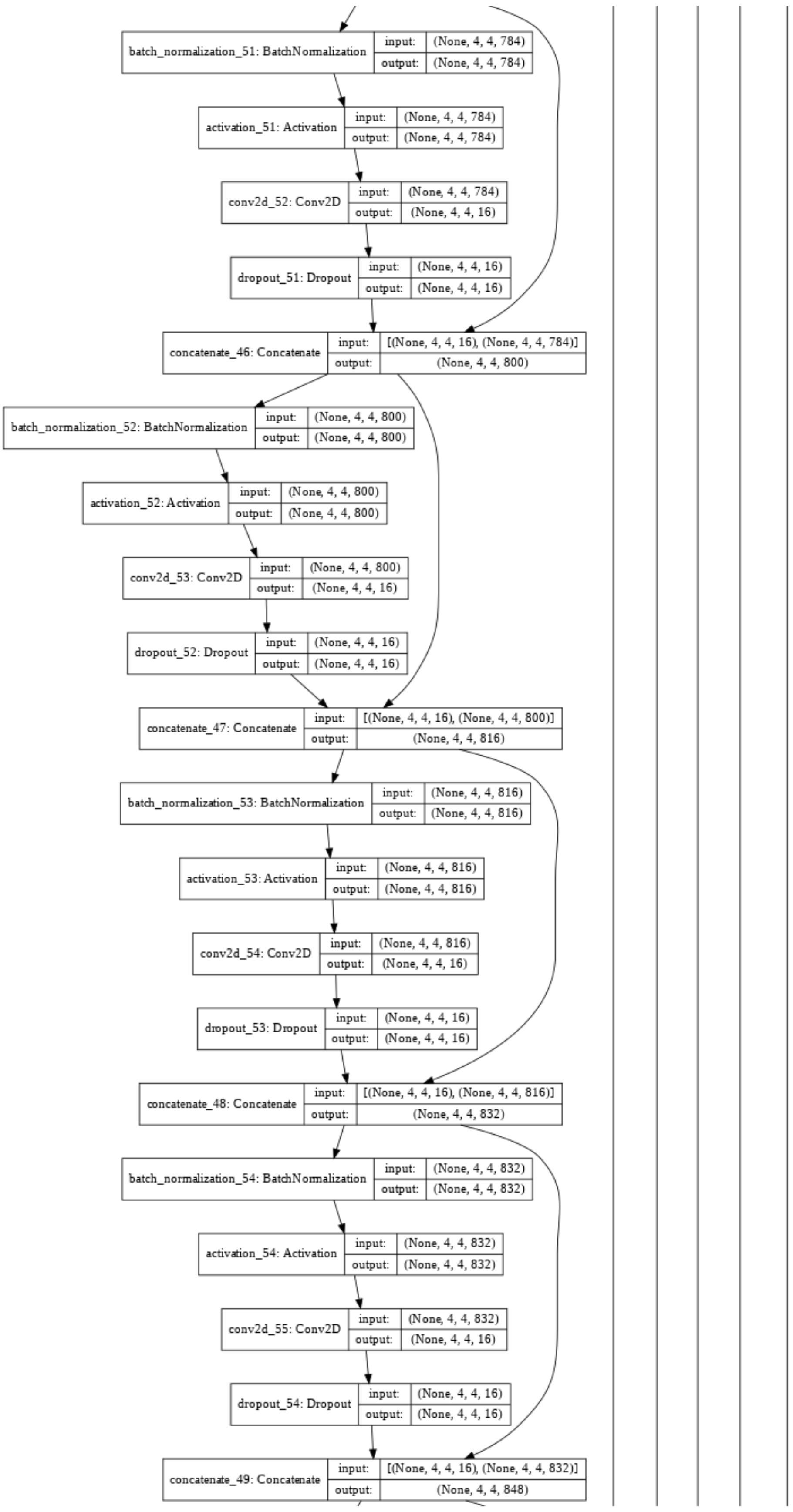

batch_normalization_51: BatchNormalization
input: (None, 4, 4, 784)
output: (None, 4, 4, 784)
activation_51: Activation
input: (None, 4, 4, 784)
output: (None, 4, 4, 784)
conv2d_52: Conv2D
input: (None, 4, 4, 784)
output: (None, 4, 4, 16)
dropout_51: Dropout
input: (None, 4, 4, 16)
output: (None, 4, 4, 16)
concatenate_46: Concatenate
input: [(None, 4, 4, 16), (None, 4, 4, 784)]
output: (None, 4, 4, 800)
batch_normalization_52: BatchNormalization
input: (None, 4, 4, 800)
output: (None, 4, 4, 800)
activation_52: Activation
input: (None, 4, 4, 800)
output: (None, 4, 4, 800)
conv2d_53: Conv2D
input: (None, 4, 4, 800)
output: (None, 4, 4, 16)
dropout_52: Dropout
input: (None, 4, 4, 16)
output: (None, 4, 4, 16)
concatenate_47: Concatenate
input: [(None, 4, 4, 16), (None, 4, 4, 800)]
output: (None, 4, 4, 816)
batch_normalization_53: BatchNormalization
input: (None, 4, 4, 816)
output: (None, 4, 4, 816)
activation_53: Activation
input: (None, 4, 4, 816)
output: (None, 4, 4, 816)
conv2d_54: Conv2D
input: (None, 4, 4, 816)
output: (None, 4, 4, 16)
dropout_53: Dropout
input: (None, 4, 4, 16)
output: (None, 4, 4, 16)
concatenate_48: Concatenate
input: [(None, 4, 4, 16), (None, 4, 4, 816)]
output: (None, 4, 4, 832)
batch_normalization_54: BatchNormalization
input: (None, 4, 4, 832)
output: (None, 4, 4, 832)
activation_54: Activation
input: (None, 4, 4, 832)
output: (None, 4, 4, 832)
conv2d_55: Conv2D
input: (None, 4, 4, 832)
output: (None, 4, 4, 16)
dropout_54: Dropout
input: (None, 4, 4, 16)
output: (None, 4, 4, 16)
concatenate_49: Concatenate
input: [(None, 4, 4, 16), (None, 4, 4, 832)]
output: (None, 4, 4, 848)

Figure 82: Tiramisu model architecture (part 14).

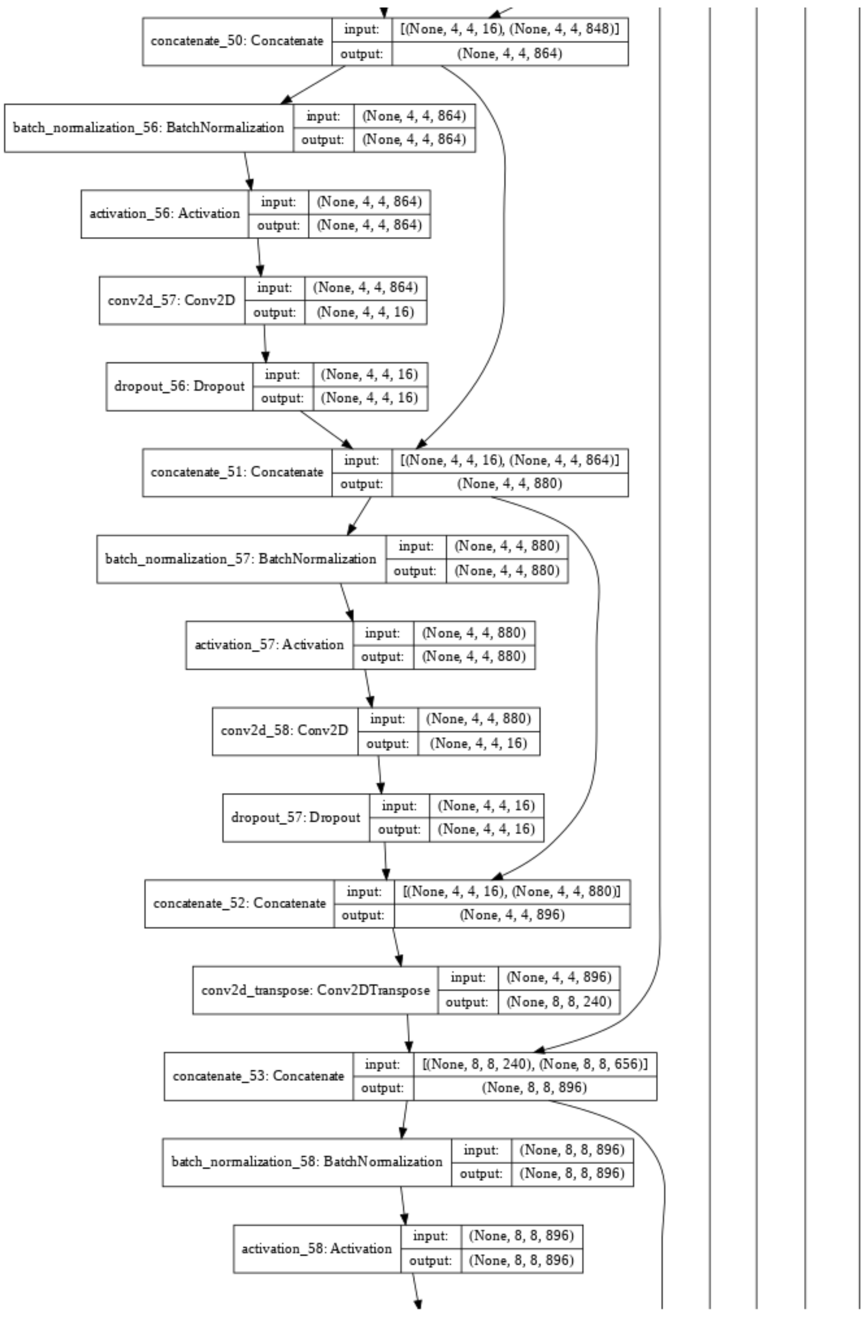

concatenate_50: Concatenate
input: [(None, 4, 4, 16), (None, 4, 4, 848)]
output: (None, 4, 4, 864)
batch_normalization_56: BatchNormalization
input: (None, 4, 4, 864)
output: (None, 4, 4, 864)
activation_56: Activation
input: (None, 4, 4, 864)
output: (None, 4, 4, 864)
conv2d_57: Conv2D
input: (None, 4, 4, 864)
output: (None, 4, 4, 16)
dropout_56: Dropout
input: (None, 4, 4, 16)
output: (None, 4, 4, 16)
concatenate_51: Concatenate
input: [(None, 4, 4, 16), (None, 4, 4, 864)]
output: (None, 4, 4, 880)
batch_normalization_57: BatchNormalization
input: (None, 4, 4, 880)
output: (None, 4, 4, 880)
activation_57: Activation
input: (None, 4, 4, 880)
output: (None, 4, 4, 880)
conv2d_58: Conv2D
input: (None, 4, 4, 880)
output: (None, 4, 4, 16)
dropout_57: Dropout
input: (None, 4, 4, 16)
output: (None, 4, 4, 16)
concatenate_52: Concatenate
input: [(None, 4, 4, 16), (None, 4, 4, 880)]
output: (None, 4, 4, 896)
conv2d_transpose: Conv2DTranspose
input: (None, 4, 4, 896)
output: (None, 8, 8, 240)
concatenate_53: Concatenate
input: [(None, 8, 8, 240), (None, 8, 8, 656)]
output: (None, 8, 8, 896)
batch_normalization_58: BatchNormalization
input: (None, 8, 8, 896)
output: (None, 8, 8, 896)
activation_58: Activation
input: (None, 8, 8, 896)
output: (None, 8, 8, 896)

Figure 83: Tiramisu model architecture (part 15).

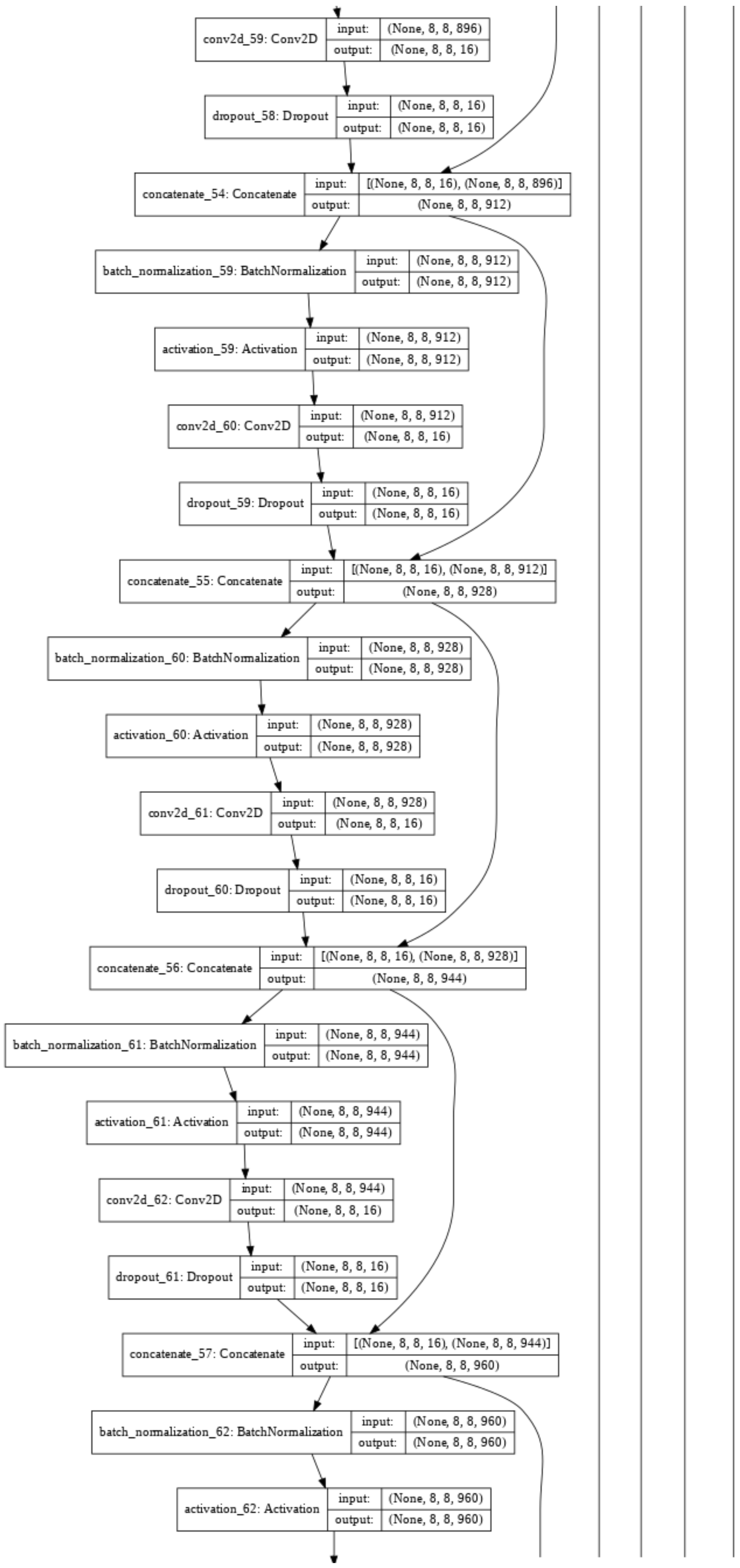

conv2d_59: Conv2D
input: (None, 8, 8, 896)
output: (None, 8, 8, 16)
dropout_58: Dropout
input: (None, 8, 8, 16)
output: (None, 8, 8, 16)
concatenate_54: Concatenate
input: [(None, 8, 8, 16), (None, 8, 8, 896)]
output: (None, 8, 8, 912)
batch_normalization_59: BatchNormalization
input: (None, 8, 8, 912)
output: (None, 8, 8, 912)
activation_59: Activation
input: (None, 8, 8, 912)
output: (None, 8, 8, 912)
conv2d_60: Conv2D
input: (None, 8, 8, 912)
output: (None, 8, 8, 16)
dropout_59: Dropout
input: (None, 8, 8, 16)
output: (None, 8, 8, 16)
concatenate_55: Concatenate
input: [(None, 8, 8, 16), (None, 8, 8, 912)]
output: (None, 8, 8, 928)
batch_normalization_60: BatchNormalization
input: (None, 8, 8, 928)
output: (None, 8, 8, 928)
activation_60: Activation
input: (None, 8, 8, 928)
output: (None, 8, 8, 928)
conv2d_61: Conv2D
input: (None, 8, 8, 928)
output: (None, 8, 8, 16)
dropout_60: Dropout
input: (None, 8, 8, 16)
output: (None, 8, 8, 16)
concatenate_56: Concatenate
input: [(None, 8, 8, 16), (None, 8, 8, 928)]
output: (None, 8, 8, 944)
batch_normalization_61: BatchNormalization
input: (None, 8, 8, 944)
output: (None, 8, 8, 944)
activation_61: Activation
input: (None, 8, 8, 944)
output: (None, 8, 8, 944)
conv2d_62: Conv2D
input: (None, 8, 8, 944)
output: (None, 8, 8, 16)
dropout_61: Dropout
input: (None, 8, 8, 16)
output: (None, 8, 8, 16)
concatenate_57: Concatenate
input: [(None, 8, 8, 16), (None, 8, 8, 944)]
output: (None, 8, 8, 960)
batch_normalization_62: BatchNormalization
input: (None, 8, 8, 960)
output: (None, 8, 8, 960)
activation_62: Activation
input: (None, 8, 8, 960)
output: (None, 8, 8, 960)

Figure 84: Tiramisu model architecture (part 16).

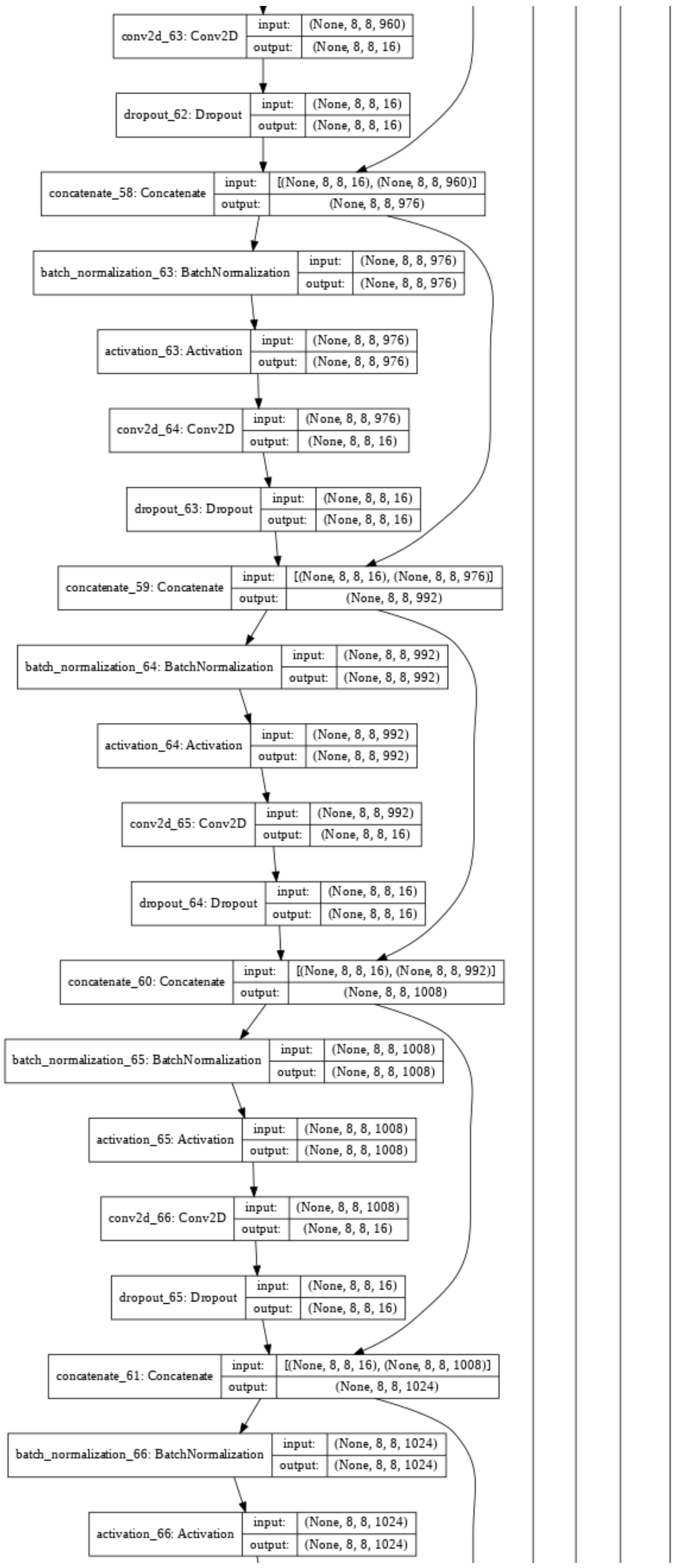

conv2d_63: Conv2D
input: (None, 8, 8, 960)
output: (None, 8, 8, 16)
dropout_62: Dropout
input: (None, 8, 8, 16)
output: (None, 8, 8, 16)
concatenate_58: Concatenate
input: [(None, 8, 8, 16), (None, 8, 8, 960)]
output: (None, 8, 8, 976)
batch_normalization_63: BatchNormalization
input: (None, 8, 8, 976)
output: (None, 8, 8, 976)
activation_63: Activation
input: (None, 8, 8, 976)
output: (None, 8, 8, 976)
conv2d_64: Conv2D
input: (None, 8, 8, 976)
output: (None, 8, 8, 16)
dropout_63: Dropout
input: (None, 8, 8, 16)
output: (None, 8, 8, 16)
concatenate_59: Concatenate
input: [(None, 8, 8, 16), (None, 8, 8, 976)]
output: (None, 8, 8, 992)
batch_normalization_64: BatchNormalization
input: (None, 8, 8, 992)
output: (None, 8, 8, 992)
activation_64: Activation
input: (None, 8, 8, 992)
output: (None, 8, 8, 992)
conv2d_65: Conv2D
input: (None, 8, 8, 992)
output: (None, 8, 8, 16)
dropout_64: Dropout
input: (None, 8, 8, 16)
output: (None, 8, 8, 16)
concatenate_60: Concatenate
input: [(None, 8, 8, 16), (None, 8, 8, 992)]
output: (None, 8, 8, 1008)
batch_normalization_65: BatchNormalization
input: (None, 8, 8, 1008)
output: (None, 8, 8, 1008)
activation_65: Activation
input: (None, 8, 8, 1008)
output: (None, 8, 8, 1008)
conv2d_66: Conv2D
input: (None, 8, 8, 1008)
output: (None, 8, 8, 16)
dropout_65: Dropout
input: (None, 8, 8, 16)
output: (None, 8, 8, 16)
concatenate_61: Concatenate
input: [(None, 8, 8, 16), (None, 8, 8, 1008)]
output: (None, 8, 8, 1024)
batch_normalization_66: BatchNormalization
input: (None, 8, 8, 1024)
output: (None, 8, 8, 1024)
activation_66: Activation
input: (None, 8, 8, 1024)
output: (None, 8, 8, 1024)

Figure 85: Tiramisu model architecture (part 17).

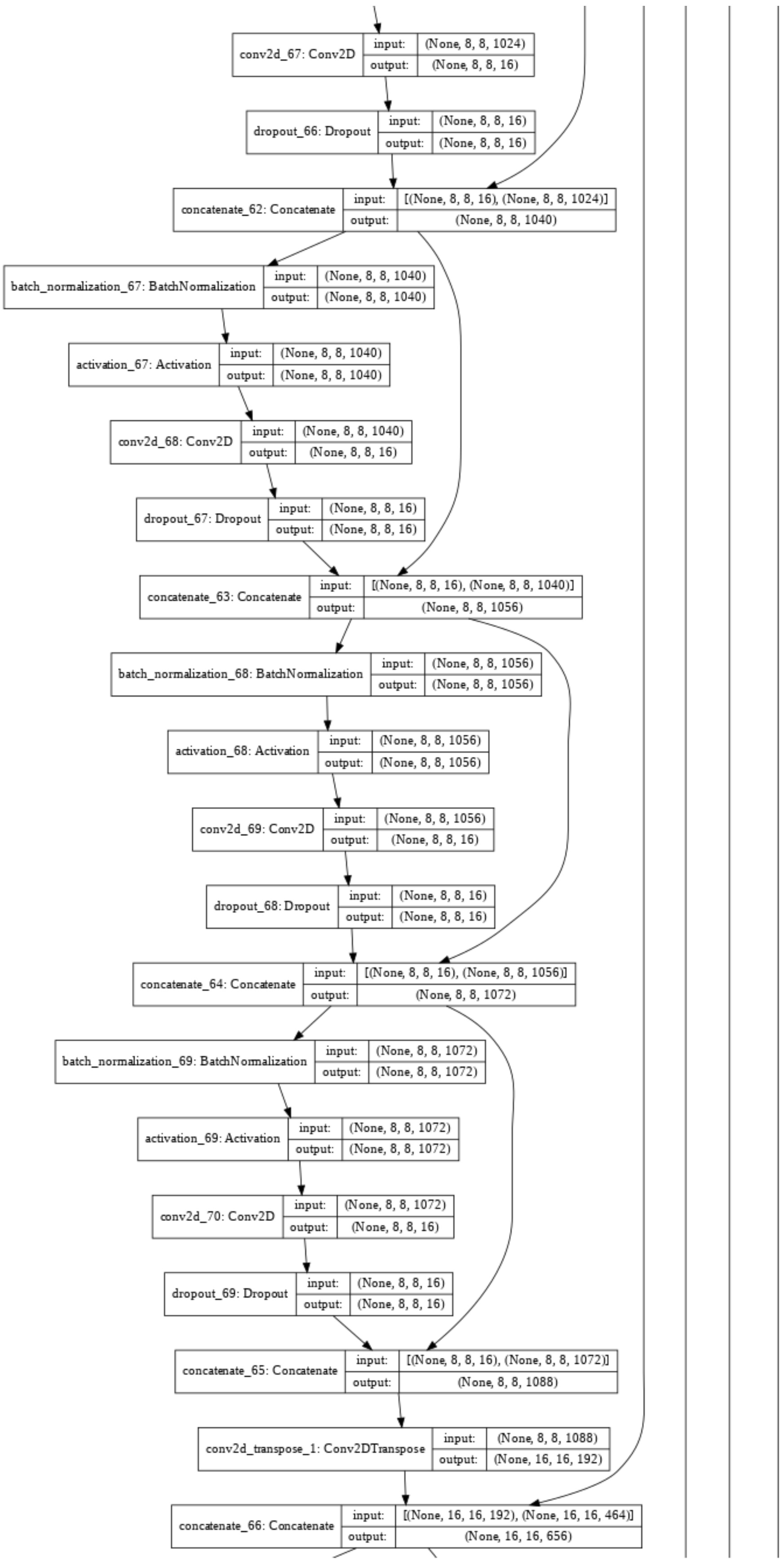

conv2d_67: Conv2D
input: (None, 8, 8, 1024)
output: (None, 8, 8, 16)
dropout_66: Dropout
input: (None, 8, 8, 16)
output: (None, 8, 8, 16)
concatenate_62: Concatenate
input: [(None, 8, 8, 16), (None, 8, 8, 1024)]
output: (None, 8, 8, 1040)
batch_normalization_67: BatchNormalization
input: (None, 8, 8, 1040)
output: (None, 8, 8, 1040)
activation_67: Activation
input: (None, 8, 8, 1040)
output: (None, 8, 8, 1040)
conv2d_68: Conv2D
input: (None, 8, 8, 1040)
output: (None, 8, 8, 16)
dropout_67: Dropout
input: (None, 8, 8, 16)
output: (None, 8, 8, 16)
concatenate_63: Concatenate
input: [(None, 8, 8, 16), (None, 8, 8, 1040)]
output: (None, 8, 8, 1056)
batch_normalization_68: BatchNormalization
input: (None, 8, 8, 1056)
output: (None, 8, 8, 1056)
activation_68: Activation
input: (None, 8, 8, 1056)
output: (None, 8, 8, 1056)
conv2d_69: Conv2D
input: (None, 8, 8, 1056)
output: (None, 8, 8, 16)
dropout_68: Dropout
input: (None, 8, 8, 16)
output: (None, 8, 8, 16)
concatenate_64: Concatenate
input: [(None, 8, 8, 16), (None, 8, 8, 1056)]
output: (None, 8, 8, 1072)
batch_normalization_69: BatchNormalization
input: (None, 8, 8, 1072)
output: (None, 8, 8, 1072)
activation_69: Activation
input: (None, 8, 8, 1072)
output: (None, 8, 8, 1072)
conv2d_70: Conv2D
input: (None, 8, 8, 1072)
output: (None, 8, 8, 16)
dropout_69: Dropout
input: (None, 8, 8, 16)
output: (None, 8, 8, 16)
concatenate_65: Concatenate
input: [(None, 8, 8, 16), (None, 8, 8, 1072)]
output: (None, 8, 8, 1088)
conv2d_transpose_1: Conv2DTranspose
input: (None, 8, 8, 1088)
output: (None, 16, 16, 192)
concatenate_66: Concatenate
input: [(None, 16, 16, 192), (None, 16, 16, 464)]
output: (None, 16, 16, 656)

Figure 86: Tiramisu model architecture (part 18).

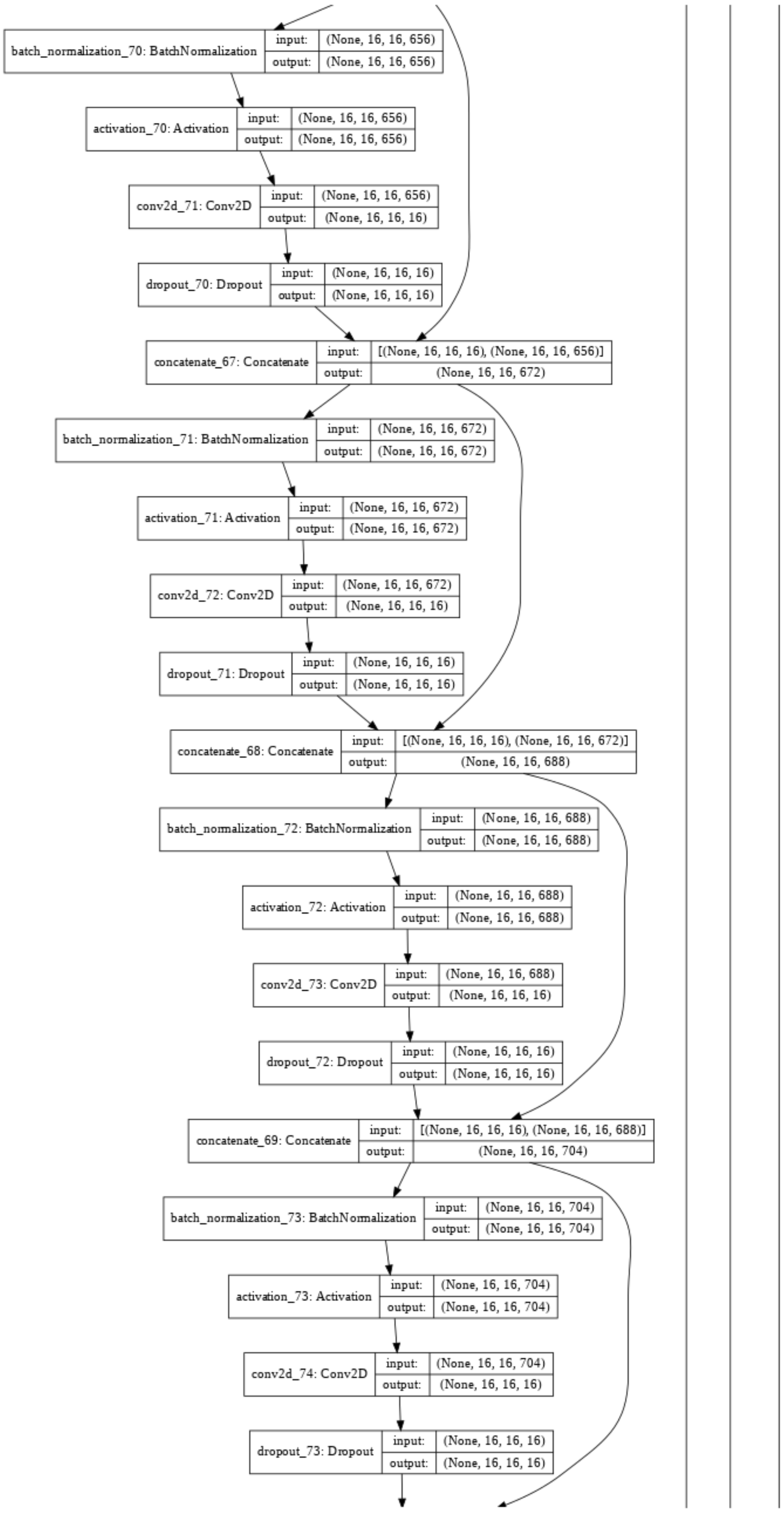

batch_normalization_70: BatchNormalization
input: (None, 16, 16, 656)
output: (None, 16, 16, 656)
activation_70: Activation
input: (None, 16, 16, 656)
output: (None, 16, 16, 656)
conv2d_71: Conv2D
input: (None, 16, 16, 656)
output: (None, 16, 16, 16)
dropout_70: Dropout
input: (None, 16, 16, 16)
output: (None, 16, 16, 16)
concatenate_67: Concatenate
input: [(None, 16, 16, 16), (None, 16, 16, 656)]
output: (None, 16, 16, 672)
batch_normalization_71: BatchNormalization
input: (None, 16, 16, 672)
output: (None, 16, 16, 672)
activation_71: Activation
input: (None, 16, 16, 672)
output: (None, 16, 16, 672)
conv2d_72: Conv2D
input: (None, 16, 16, 672)
output: (None, 16, 16, 16)
dropout_71: Dropout
input: (None, 16, 16, 16)
output: (None, 16, 16, 16)
concatenate_68: Concatenate
input: [(None, 16, 16, 16), (None, 16, 16, 672)]
output: (None, 16, 16, 688)
batch_normalization_72: BatchNormalization
input: (None, 16, 16, 688)
output: (None, 16, 16, 688)
activation_72: Activation
input: (None, 16, 16, 688)
output: (None, 16, 16, 688)
conv2d_73: Conv2D
input: (None, 16, 16, 688)
output: (None, 16, 16, 16)
dropout_72: Dropout
input: (None, 16, 16, 16)
output: (None, 16, 16, 16)
concatenate_69: Concatenate
input: [(None, 16, 16, 16), (None, 16, 16, 688)]
output: (None, 16, 16, 704)
batch_normalization_73: BatchNormalization
input: (None, 16, 16, 704)
output: (None, 16, 16, 704)
activation_73: Activation
input: (None, 16, 16, 704)
output: (None, 16, 16, 704)
conv2d_74: Conv2D
input: (None, 16, 16, 704)
output: (None, 16, 16, 16)
dropout_73: Dropout
input: (None, 16, 16, 16)
output: (None, 16, 16, 16)

Figure 87: Tiramisu model architecture (part 19).

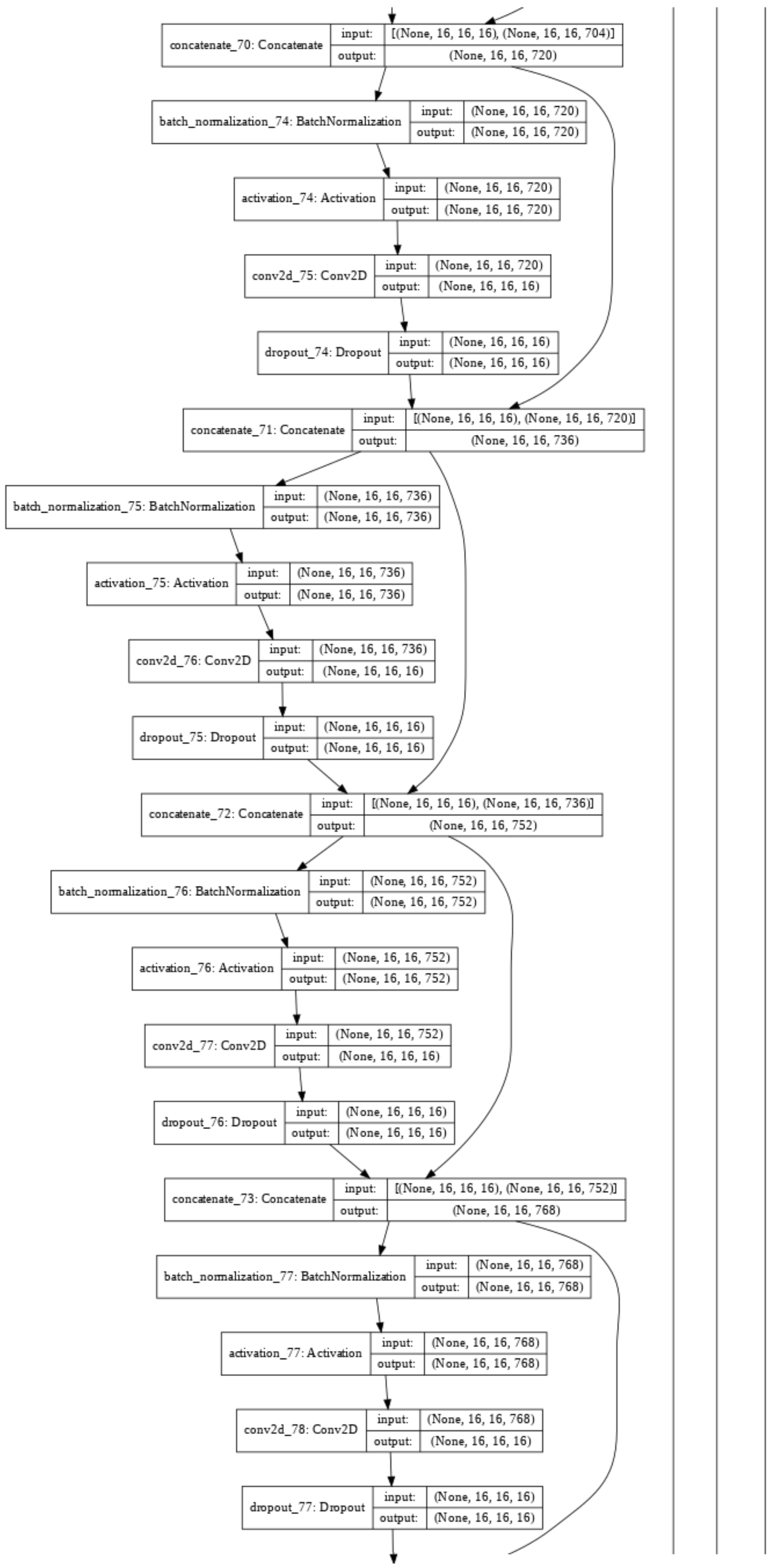

concatenate_70: Concatenate
input: [(None, 16, 16, 16), (None, 16, 16, 704)]
output: (None, 16, 16, 720)
batch_normalization_74: BatchNormalization
input: (None, 16, 16, 720)
output: (None, 16, 16, 720)
activation_74: Activation
input: (None, 16, 16, 720)
output: (None, 16, 16, 720)
conv2d_75: Conv2D
input: (None, 16, 16, 720)
output: (None, 16, 16, 16)
dropout_74: Dropout
input: (None, 16, 16, 16)
output: (None, 16, 16, 16)
concatenate_71: Concatenate
input: [(None, 16, 16, 16), (None, 16, 16, 720)]
output: (None, 16, 16, 736)
batch_normalization_75: BatchNormalization
input: (None, 16, 16, 736)
output: (None, 16, 16, 736)
activation_75: Activation
input: (None, 16, 16, 736)
output: (None, 16, 16, 736)
conv2d_76: Conv2D
input: (None, 16, 16, 736)
output: (None, 16, 16, 16)
dropout_75: Dropout
input: (None, 16, 16, 16)
output: (None, 16, 16, 16)
concatenate_72: Concatenate
input: [(None, 16, 16, 16), (None, 16, 16, 736)]
output: (None, 16, 16, 752)
batch_normalization_76: BatchNormalization
input: (None, 16, 16, 752)
output: (None, 16, 16, 752)
activation_76: Activation
input: (None, 16, 16, 752)
output: (None, 16, 16, 752)
conv2d_77: Conv2D
input: (None, 16, 16, 752)
output: (None, 16, 16, 16)
dropout_76: Dropout
input: (None, 16, 16, 16)
output: (None, 16, 16, 16)
concatenate_73: Concatenate
input: [(None, 16, 16, 16), (None, 16, 16, 752)]
output: (None, 16, 16, 768)
batch_normalization_77: BatchNormalization
input: (None, 16, 16, 768)
output: (None, 16, 16, 768)
activation_77: Activation
input: (None, 16, 16, 768)
output: (None, 16, 16, 768)
conv2d_78: Conv2D
input: (None, 16, 16, 768)
output: (None, 16, 16, 16)
dropout_77: Dropout
input: (None, 16, 16, 16)
output: (None, 16, 16, 16)

Figure 88: Tiramisu model architecture (part 20).

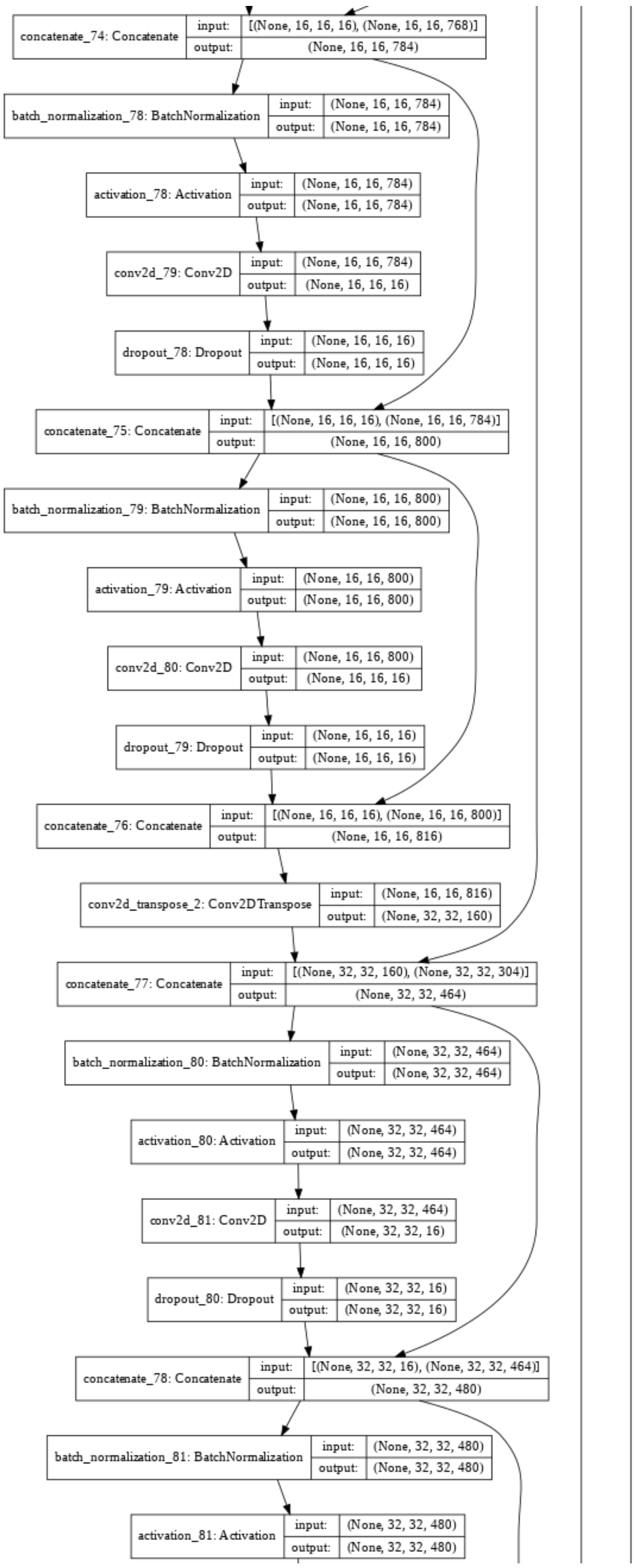

concatenate_74: Concatenate | input: [(None, 16, 16, 16), (None, 16, 16, 768)] | output: (None, 16, 16, 784)
batch_normalization_78: BatchNormalization | input: (None, 16, 16, 784) | output: (None, 16, 16, 784)
activation_78: Activation | input: (None, 16, 16, 784) | output: (None, 16, 16, 784)
conv2d_79: Conv2D | input: (None, 16, 16, 784) | output: (None, 16, 16, 16)
dropout_78: Dropout | input: (None, 16, 16, 16) | output: (None, 16, 16, 16)
concatenate_75: Concatenate | input: [(None, 16, 16, 16), (None, 16, 16, 784)] | output: (None, 16, 16, 800)
batch_normalization_79: BatchNormalization | input: (None, 16, 16, 800) | output: (None, 16, 16, 800)
activation_79: Activation | input: (None, 16, 16, 800) | output: (None, 16, 16, 800)
conv2d_80: Conv2D | input: (None, 16, 16, 800) | output: (None, 16, 16, 16)
dropout_79: Dropout | input: (None, 16, 16, 16) | output: (None, 16, 16, 16)
concatenate_76: Concatenate | input: [(None, 16, 16, 16), (None, 16, 16, 800)] | output: (None, 16, 16, 816)
conv2d_transpose_2: Conv2DTranspose | input: (None, 16, 16, 816) | output: (None, 32, 32, 160)
concatenate_77: Concatenate | input: [(None, 32, 32, 160), (None, 32, 32, 304)] | output: (None, 32, 32, 464)
batch_normalization_80: BatchNormalization | input: (None, 32, 32, 464) | output: (None, 32, 32, 464)
activation_80: Activation | input: (None, 32, 32, 464) | output: (None, 32, 32, 464)
conv2d_81: Conv2D | input: (None, 32, 32, 464) | output: (None, 32, 32, 16)
dropout_80: Dropout | input: (None, 32, 32, 16) | output: (None, 32, 32, 16)
concatenate_78: Concatenate | input: [(None, 32, 32, 16), (None, 32, 32, 464)] | output: (None, 32, 32, 480)
batch_normalization_81: BatchNormalization | input: (None, 32, 32, 480) | output: (None, 32, 32, 480)
activation_81: Activation | input: (None, 32, 32, 480) | output: (None, 32, 32, 480)

Figure 89: Tiramisu model architecture (part 21).

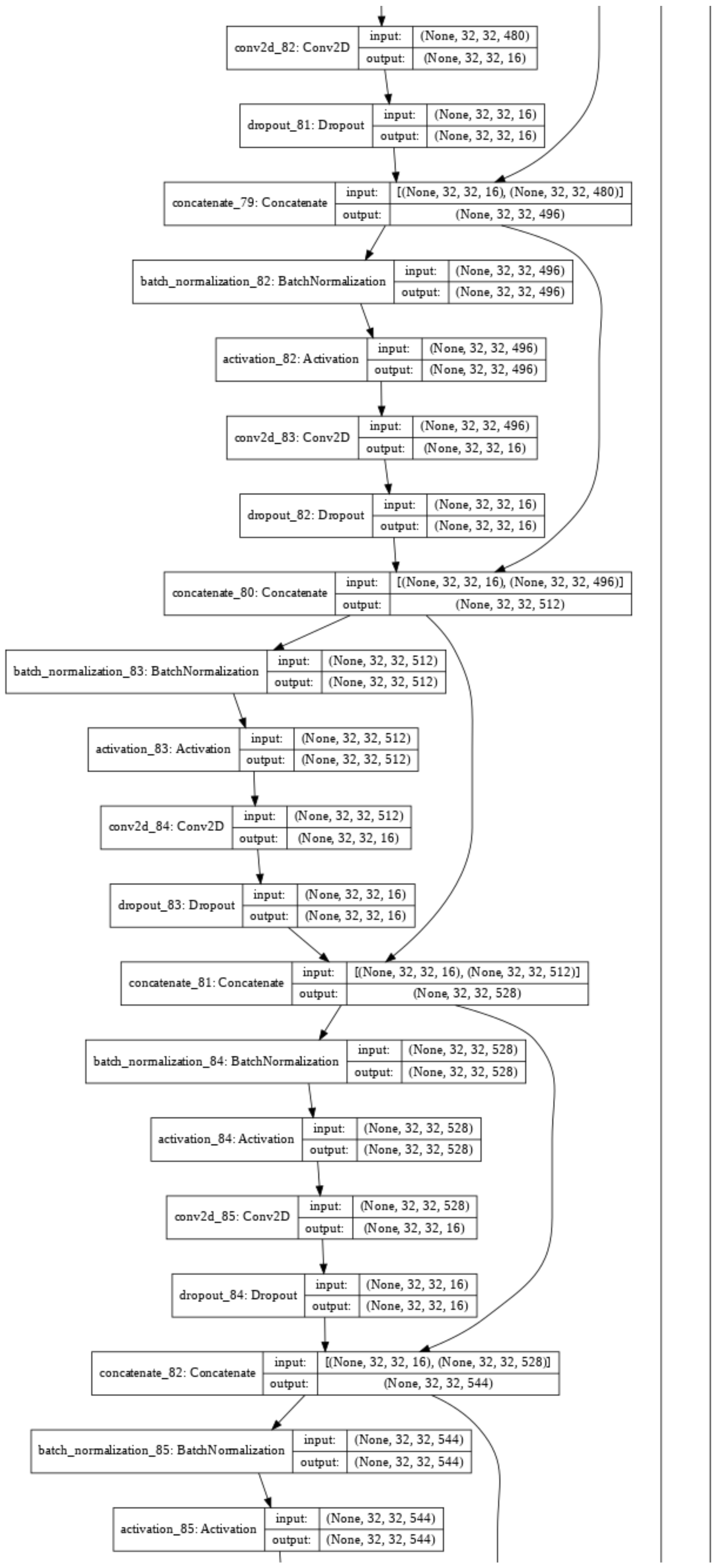

conv2d_82: Conv2D
input: (None, 32, 32, 480)
output: (None, 32, 32, 16)
dropout_81: Dropout
input: (None, 32, 32, 16)
output: (None, 32, 32, 16)
concatenate_79: Concatenate
input: [(None, 32, 32, 16), (None, 32, 32, 480)]
output: (None, 32, 32, 496)
batch_normalization_82: BatchNormalization
input: (None, 32, 32, 496)
output: (None, 32, 32, 496)
activation_82: Activation
input: (None, 32, 32, 496)
output: (None, 32, 32, 496)
conv2d_83: Conv2D
input: (None, 32, 32, 496)
output: (None, 32, 32, 16)
dropout_82: Dropout
input: (None, 32, 32, 16)
output: (None, 32, 32, 16)
concatenate_80: Concatenate
input: [(None, 32, 32, 16), (None, 32, 32, 496)]
output: (None, 32, 32, 512)
batch_normalization_83: BatchNormalization
input: (None, 32, 32, 512)
output: (None, 32, 32, 512)
activation_83: Activation
input: (None, 32, 32, 512)
output: (None, 32, 32, 512)
conv2d_84: Conv2D
input: (None, 32, 32, 512)
output: (None, 32, 32, 16)
dropout_83: Dropout
input: (None, 32, 32, 16)
output: (None, 32, 32, 16)
concatenate_81: Concatenate
input: [(None, 32, 32, 16), (None, 32, 32, 512)]
output: (None, 32, 32, 528)
batch_normalization_84: BatchNormalization
input: (None, 32, 32, 528)
output: (None, 32, 32, 528)
activation_84: Activation
input: (None, 32, 32, 528)
output: (None, 32, 32, 528)
conv2d_85: Conv2D
input: (None, 32, 32, 528)
output: (None, 32, 32, 16)
dropout_84: Dropout
input: (None, 32, 32, 16)
output: (None, 32, 32, 16)
concatenate_82: Concatenate
input: [(None, 32, 32, 16), (None, 32, 32, 528)]
output: (None, 32, 32, 544)
batch_normalization_85: BatchNormalization
input: (None, 32, 32, 544)
output: (None, 32, 32, 544)
activation_85: Activation
input: (None, 32, 32, 544)
output: (None, 32, 32, 544)

Figure 90: Tiramisu model architecture (part 22).

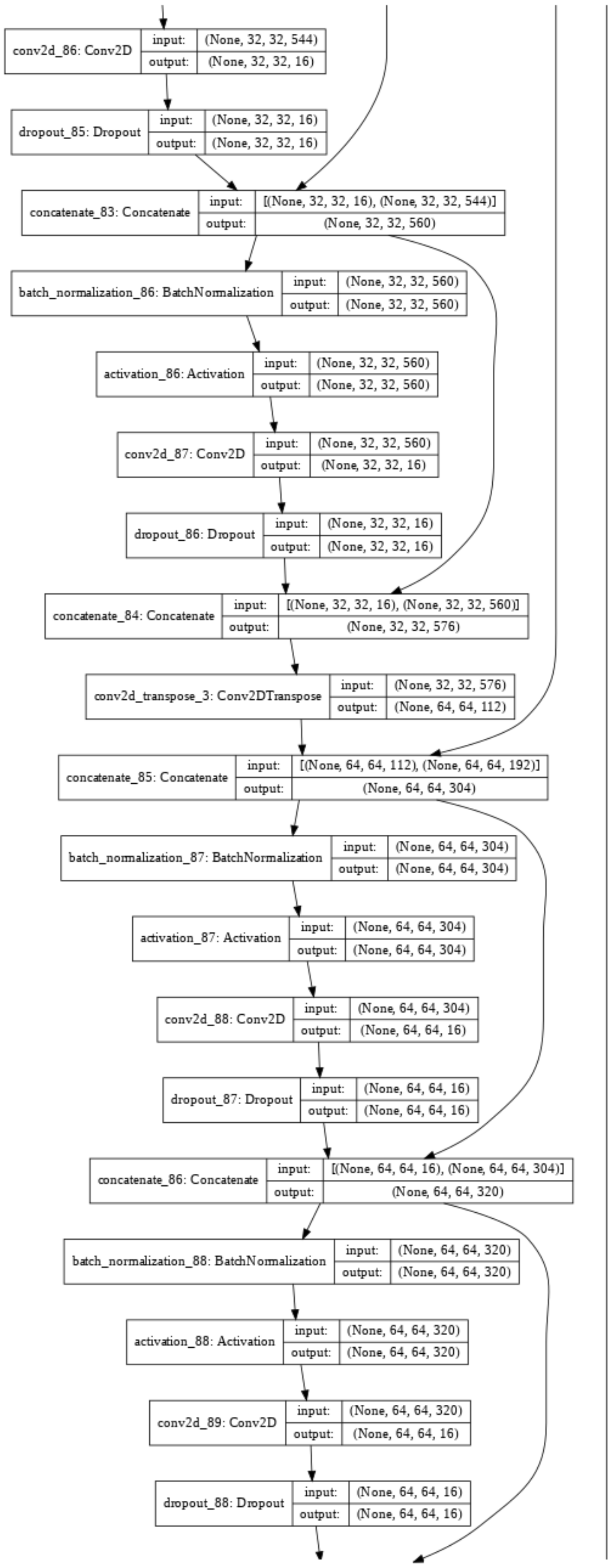

conv2d_86: Conv2D
input: (None, 32, 32, 544)
output: (None, 32, 32, 16)
dropout_85: Dropout
input: (None, 32, 32, 16)
output: (None, 32, 32, 16)
concatenate_83: Concatenate
input: [(None, 32, 32, 16), (None, 32, 32, 544)]
output: (None, 32, 32, 560)
batch_normalization_86: BatchNormalization
input: (None, 32, 32, 560)
output: (None, 32, 32, 560)
activation_86: Activation
input: (None, 32, 32, 560)
output: (None, 32, 32, 560)
conv2d_87: Conv2D
input: (None, 32, 32, 560)
output: (None, 32, 32, 16)
dropout_86: Dropout
input: (None, 32, 32, 16)
output: (None, 32, 32, 16)
concatenate_84: Concatenate
input: [(None, 32, 32, 16), (None, 32, 32, 560)]
output: (None, 32, 32, 576)
conv2d_transpose_3: Conv2DTranspose
input: (None, 32, 32, 576)
output: (None, 64, 64, 112)
concatenate_85: Concatenate
input: [(None, 64, 64, 112), (None, 64, 64, 192)]
output: (None, 64, 64, 304)
batch_normalization_87: BatchNormalization
input: (None, 64, 64, 304)
output: (None, 64, 64, 304)
activation_87: Activation
input: (None, 64, 64, 304)
output: (None, 64, 64, 304)
conv2d_88: Conv2D
input: (None, 64, 64, 304)
output: (None, 64, 64, 16)
dropout_87: Dropout
input: (None, 64, 64, 16)
output: (None, 64, 64, 16)
concatenate_86: Concatenate
input: [(None, 64, 64, 16), (None, 64, 64, 304)]
output: (None, 64, 64, 320)
batch_normalization_88: BatchNormalization
input: (None, 64, 64, 320)
output: (None, 64, 64, 320)
activation_88: Activation
input: (None, 64, 64, 320)
output: (None, 64, 64, 320)
conv2d_89: Conv2D
input: (None, 64, 64, 320)
output: (None, 64, 64, 16)
dropout_88: Dropout
input: (None, 64, 64, 16)
output: (None, 64, 64, 16)

Figure 91: Tiramisu model architecture (part 23).

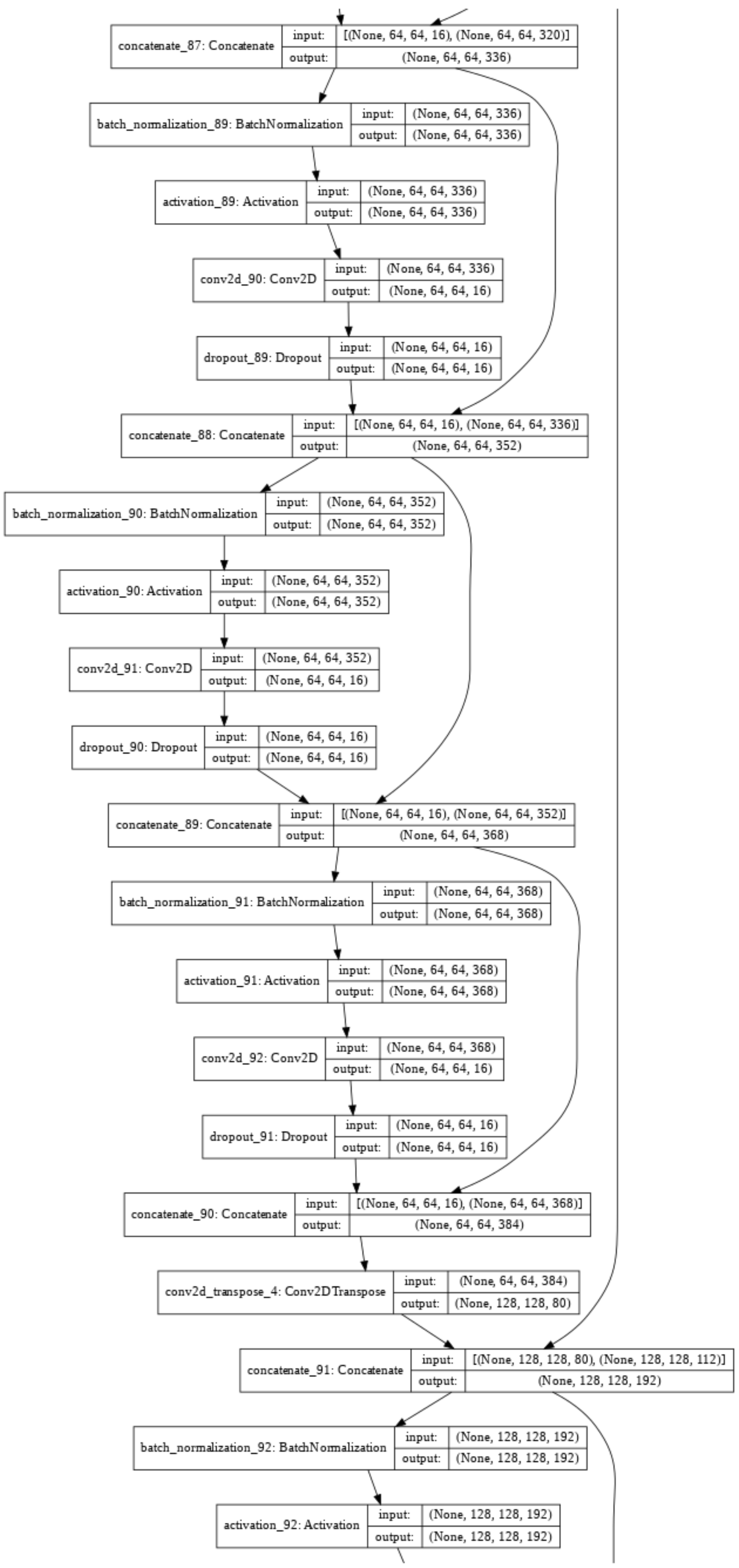

concatenate_87: Concatenate
input: [(None, 64, 64, 16), (None, 64, 64, 320)]
output: (None, 64, 64, 336)
batch_normalization_89: BatchNormalization
input: (None, 64, 64, 336)
output: (None, 64, 64, 336)
activation_89: Activation
input: (None, 64, 64, 336)
output: (None, 64, 64, 336)
conv2d_90: Conv2D
input: (None, 64, 64, 336)
output: (None, 64, 64, 16)
dropout_89: Dropout
input: (None, 64, 64, 16)
output: (None, 64, 64, 16)
concatenate_88: Concatenate
input: [(None, 64, 64, 16), (None, 64, 64, 336)]
output: (None, 64, 64, 352)
batch_normalization_90: BatchNormalization
input: (None, 64, 64, 352)
output: (None, 64, 64, 352)
activation_90: Activation
input: (None, 64, 64, 352)
output: (None, 64, 64, 352)
conv2d_91: Conv2D
input: (None, 64, 64, 352)
output: (None, 64, 64, 16)
dropout_90: Dropout
input: (None, 64, 64, 16)
output: (None, 64, 64, 16)
concatenate_89: Concatenate
input: [(None, 64, 64, 16), (None, 64, 64, 352)]
output: (None, 64, 64, 368)
batch_normalization_91: BatchNormalization
input: (None, 64, 64, 368)
output: (None, 64, 64, 368)
activation_91: Activation
input: (None, 64, 64, 368)
output: (None, 64, 64, 368)
conv2d_92: Conv2D
input: (None, 64, 64, 368)
output: (None, 64, 64, 16)
dropout_91: Dropout
input: (None, 64, 64, 16)
output: (None, 64, 64, 16)
concatenate_90: Concatenate
input: [(None, 64, 64, 16), (None, 64, 64, 368)]
output: (None, 64, 64, 384)
conv2d_transpose_4: Conv2DTranspose
input: (None, 64, 64, 384)
output: (None, 128, 128, 80)
concatenate_91: Concatenate
input: [(None, 128, 128, 80), (None, 128, 128, 112)]
output: (None, 128, 128, 192)
batch_normalization_92: BatchNormalization
input: (None, 128, 128, 192)
output: (None, 128, 128, 192)
activation_92: Activation
input: (None, 128, 128, 192)
output: (None, 128, 128, 192)

Figure 92: Tiramisu model architecture (part 24).

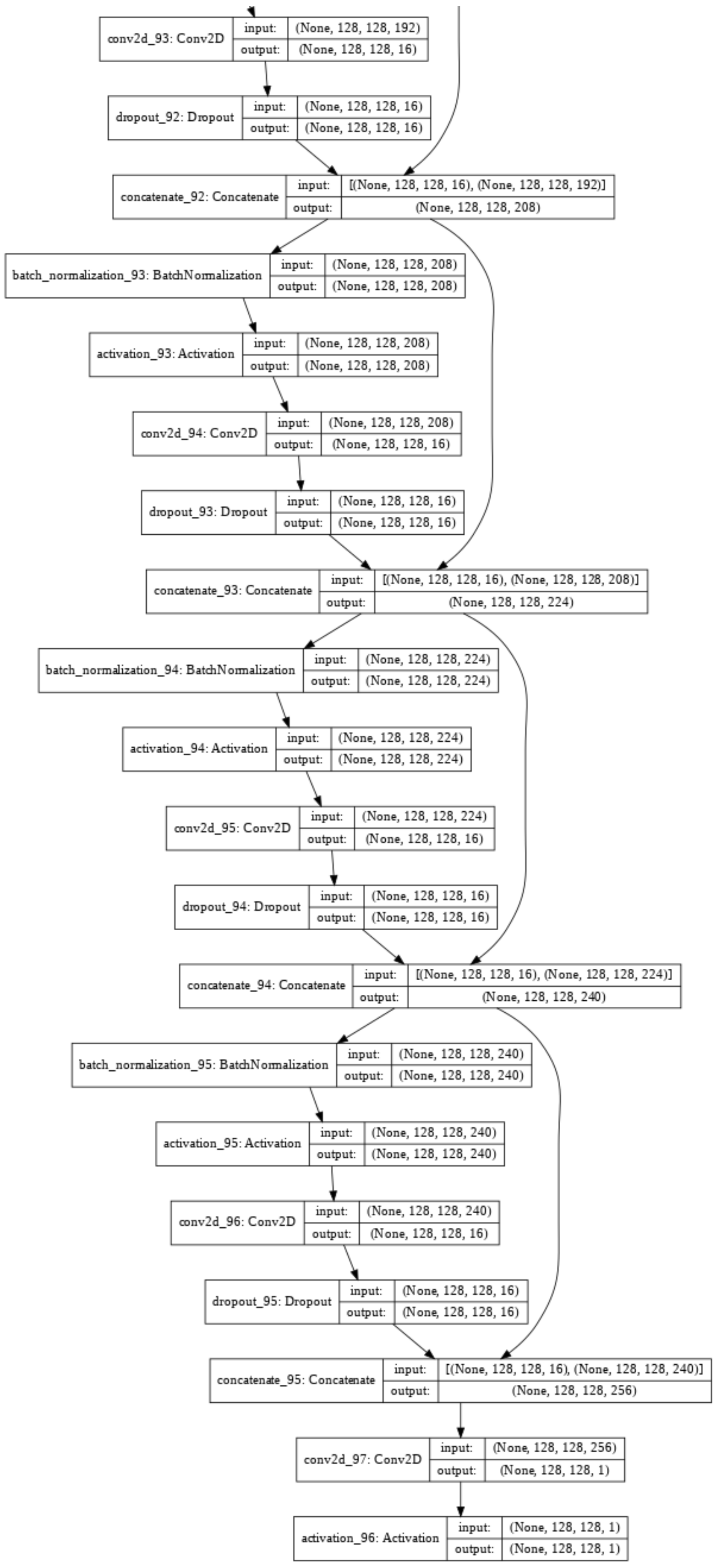

conv2d_93: Conv2D | input: (None, 128, 128, 192) | output: (None, 128, 128, 16)
dropout_92: Dropout | input: (None, 128, 128, 16) | output: (None, 128, 128, 16)
concatenate_92: Concatenate | input: [(None, 128, 128, 16), (None, 128, 128, 192)] | output: (None, 128, 128, 208)
batch_normalization_93: BatchNormalization | input: (None, 128, 128, 208) | output: (None, 128, 128, 208)
activation_93: Activation | input: (None, 128, 128, 208) | output: (None, 128, 128, 208)
conv2d_94: Conv2D | input: (None, 128, 128, 208) | output: (None, 128, 128, 16)
dropout_93: Dropout | input: (None, 128, 128, 16) | output: (None, 128, 128, 16)
concatenate_93: Concatenate | input: [(None, 128, 128, 16), (None, 128, 128, 208)] | output: (None, 128, 128, 224)
batch_normalization_94: BatchNormalization | input: (None, 128, 128, 224) | output: (None, 128, 128, 224)
activation_94: Activation | input: (None, 128, 128, 224) | output: (None, 128, 128, 224)
conv2d_95: Conv2D | input: (None, 128, 128, 224) | output: (None, 128, 128, 16)
dropout_94: Dropout | input: (None, 128, 128, 16) | output: (None, 128, 128, 16)
concatenate_94: Concatenate | input: [(None, 128, 128, 16), (None, 128, 128, 224)] | output: (None, 128, 128, 240)
batch_normalization_95: BatchNormalization | input: (None, 128, 128, 240) | output: (None, 128, 128, 240)
activation_95: Activation | input: (None, 128, 128, 240) | output: (None, 128, 128, 240)
conv2d_96: Conv2D | input: (None, 128, 128, 240) | output: (None, 128, 128, 16)
dropout_95: Dropout | input: (None, 128, 128, 16) | output: (None, 128, 128, 16)
concatenate_95: Concatenate | input: [(None, 128, 128, 16), (None, 128, 128, 240)] | output: (None, 128, 128, 256)
conv2d_97: Conv2D | input: (None, 128, 128, 256) | output: (None, 128, 128, 1)
activation_96: Activation | input: (None, 128, 128, 1) | output: (None, 128, 128, 1)

Figure 93: Tiramisu model architecture (part 25).