الجمهورية الجزائرية الديمقراطية الشعبية
République Algérienne Démocratique et Populaire
وزارة التعليم العالي والبحث العلمي
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Nº Réf :……

Centre Universitaire
Abd Elhafid Boussouf Mila

Institut des Sciences et Technologie
Département de Mathématiques et Informatique

Mémoire préparé en vue de l'obtention du diplôme de Master

En : Informatique

Spécialité: Sciences et Technologies de l'Information et de la Communication (STIC)

Image segmentation using deep learning

Préparé par : BOUBRIM Nadjib
SELLAI Mohammed Raid

Soutenue devant le jury

Encadré par..…….. Dr. Aissa Boulmerka
Président …….. Mme. Meriem Talai
Examinateur …… Mme. Souheila Khalfi

Année Universitaire : 2020/2021

Abstract

الهدف من هذه المذكرة هو الحصول على فهم شامل لمجال التجزئة الدلالي للصورة بالإضافة إلى التعمق في مجال التعلم العميق وكيف يمكننا استخدام أنواع الشبكات المختلفة، وخاصة الشبكات العصبية التلافيفية، وبشكل أكثر تحديداً بنية U-Net **ومتغيراتها** Attention U-Net **و** Attention Residual U-Net **في مجال الصور الطبية باستخدام صور المجهر الإلكتروني. توضح نتائج الاختبارات وتقييم البنى المذكورة بوضوح مزايا استخدام مثل هذه الشبكات، فضلاً عن فعاليتها في حل مشاكل التجزئة الدلالية للصور.**

الكلمات الرئيسية ـ التجزئة الدلالية، التعلم العميق، الشبكات العصبية التلافيفية، U-Net **،** Attention U-Net **،** Attention Residual U-Net **، الصور المجهرية الإلكترونية.**

The goal of this dissertation is to gain a thorough understanding of the field of image semantic segmentation as well as a deep dive into the domain of deep learning and how we can use various networks, particularly convolutional neural networks and, more specifically, the U-Net architecture and its variants Attention U-Net and Attention Residual U-Net, in the medical imaging field with electron microscopy images. The tests and evaluation of the mentioned architectures demonstrate the advantages of using such networks and their effectiveness in solving image semantic segmentation problems.

Keywords — Semantic segmentation, deep learning, convolutional neural networks, U-Net, Attention U-Net, Attention Residual U-Net, electron microscopy images.

Acknowledgement

First and foremost, I thank Allah who gave me the strength and the will power to continue studying and pursuing my dream despite of all the struggles I've had in the past.

I am very grateful to Dr.Aissa Boulmerka, whose expertise, understanding, generous guidance and support made it possible for me to work on a domain that was of great interest to me.

I would also like to thank my dear parents, for all their sacrifices, their support, and their prayers for me. I won't forget my brothers and sisters also for being there for me.

A big thank you to all my professors and my colleagues whom i've had with the best time at the University of Mila, and especially the one and only Sellai Mohammed Raid, the best friend a person could ever ask for.

- M. Boubrim Nadjib

First, I want to thank Allah, the Almighty, the Most Generous, and the Most Merciful, for all of the blessings I've received during my studies and in finishing my thesis.

I would also like to give my warmest thanks to my supervisor Dr.Aissa Boulmerka who made this work possible. His guidance and advice carried me through all the stages of writing my project. I will always appreciate being one of your students.

I would like to give special thanks to my father Abdellah who did not fail in my upbringing, education and getting me to him now. I would like to extend a special thanks also to the soul of my mother in her grave, which did not leave me even for a moment, and I feel encouraged by her. I would also like to thank my brothers Dhirgham, Farouk, and Racim for supporting me in all the times I have been through. I can never forget my two grandmothers, to whom I extend my most heartfelt thanks. I also want to express my thanks and appreciation to my family, particularly my uncles and aunts.

I want to express my sincere gratitude to my partner, colleague, and brother Boubrim Nadjib, who shared my exhaustion and insomnia in research and persistence.

Final thanks to those who helped me to stand and battle for this, whether they were close or distant from my relatives, teachers or employees at the University.

- M. Sellai Mohammed Raid

Contents

List of Figures

List of Tables

List of Abbreviations

DL	Deep Learning
NN	Neural Network
ANN	Artificial Neural Network
CNN	Convolutional Neural Network
DNN	Deep Neural Network
RNN	Recurrent Neural Network
LSTM	Long Short-Term Memory
ED	Encoder-Decoder
AE	Auto-Encoder
GAN	Generative Adversarial Network
GUI	Graphical User Interface
IDE	Integrated Development Environment
CPU	Central Processing Unit
GPU	Graphics processing Unit
TPU	Tensor Processing Unit
MLP	Multi-Layer Perceptron
Tanh	Hyperbolic Tangent
ReLU	Rectified Linear Unit
BN	Batch Normalization
EMA	Exponential Moving Average
IoU	Intersection-Over-Union
MSE	Mean Square Error
MAE	Mean Absolute Error
MBE	Mean Bias Error
FL	Focal Loss
BGD	Batch Gradient Descent
SGD	Stochastic Gradient Descent

MGD	Mini-batch Gradient Descent
RMSprop	Root Mean Squared Propagation
Adagrad	Adaptive Gradient Algorithm
LR	Learning Rate
FC	Fully Connected
ResNet	Residual Neural Network
NLP	Natural Language Processing
SDAE	Stacked Denoising Auto-Encoder
VAE	Variational Auto-Encoder
JIT	Just-In-Time
FTRL	Follow The Regularized Leader
API	Application Programming Interface
ALUS	Arithmetic Logic Units
DRAM	Dynamic Random Access Memory
ASICs	Application-Specific Integrated Circuits
GLS	Gray Level Segmentation
CRF	Conditional Random Field
ILSVRC	ImageNet Large Scale Visual Recognition Challenge
FCN	Fully Convolutional Network
ASPP	Atrous Spatial Pyramid Pooling
PSPNet	Pyramid Scene Parsing Network
EM	Electron Microscopy
ISBI	International Symposium on Biomedical Imaging
ssTEM	serial section Transmission Electron Microscopy
hdf5	Hierarchical Data Format version 5
npy	NumPy array file

General introduction

Since the early days of computer vision, image segmentation has been a fundamental problem. Many visual understanding systems rely on image segmentation and object detection to make sense of images (or video frames), including medical image analysis (e.g., tumor boundary extraction and tissue volume measurement), autonomous vehicles (e.g., navigable surface and pedestrian detection), and video surveillance, to name a few.

On one hand, semantic image segmentation (classifying pixels with semantic labels) and instance segmentation (partitioning of specific objects) are two ways to approach image segmentation (panoptic segmentation). Object categories (e.g., human, automobile, tree, sky) are applied at the pixel level, making semantic segmentation a more difficult task than whole-image classification, which assigns a single label to the entire image. Each object of interest in an image can be detected and separated using instance segmentation, extending the semantic segmentation process (e.g., individual people). From early methods such as histogram-based methods, region growth, k-means clustering, and watershed methods to more complex algorithms like active contours, graph cuts, and Markov random fields.

On the other hand, DL models have produced a new generation of image segmentation models that outperform the performance of the older methods, typically obtaining the highest accuracy rates on standard benchmarks. In the field, this has led to a paradigm change.

Throughout the contents of this dissertation, we have explained in great detail several approaches and strategies that allow the implementation of image semantic segmentation. The content of the present dissertation can be described as follows:

We begin the first chapter by introducing several terms and concepts that are significant and related to deep learning (DL). Then we describe some basic DL architectures such as CNN, RNN, LSTM, ED and AE Models, and GANs. Following that, we provide an overview of some prominent DL frameworks, such as TensorFlow, PyTorch, and Keras. Finally, we provide and discuss some DL-related hardware, which includes CPUs, GPUs, and TPUs.

In the second chapter, we begin by offering a broad understanding of image semantic segmentation by defining and comparing it to other computer vision tasks, providing similar approaches and techniques to semantic segmentation, and explaining the link between this last and DL. This is followed by a look at some common CNN designs, including LeNet-5, AlexNet, VGG-16, ResNet, GoogleNet, and Mo-

bileNet, where we identify the most significant features and components that make them up using graphics and tables. Finally, we discuss the DL-Based image segmentation methods, including explanations and specifics, so we explore Fully convolutional networks, SegNet, DeepLab, Pyramid scene parsing network, and U-Net and its variants (Attention U-net, Attention Residual U-net).

In the third and final chapter, we present an application to semantic segmentation of electron microscope images. We begin by defining the problem we are trying to solve. Then, we describe the dataset we will be using and how it will be augmented. We next go to the network design details employed to solve the problem and how they were trained. After that, we conclude by describing our experiments, tests, and results. The obtained results are reported in tables and evolution curves of metrics of each model. In addition, the predictions of each trained model are shown in a collection of images. Finally, we analyze each model and make a comparison between their performances.

Chapter 1

Deep Learning

1.1 Introduction

The structure of the human brain influenced the fundamental paradigm for DL in an attempt to build structures that learn similarly to humans. As a result, several foundational terms of DL can be traced back to neurology.

In this chapter, we will cover the topic of DL, starting by giving some general concepts including ANNs, perceptrons, MLPs, activation functions, cost functions, gradient descent, LR, and finally, forward propagation and backpropagation. Then, we describe DNN architectures used widely by the computer vision community, including CNN, RNN, and LSTM. After that, we go through the popular frameworks used in DL: TensorFlow, PyTorch, and Keras. Finally, we introduce the hardware used by DL techniques such as CPUs, GPUs, and TPUs.

This chapter will make terms, concepts and approaches clearer using solid definitions, details and examples.

1.2 General concepts

1.2.1 Definitions

DL is a branch of machine learning that deals with ANNs, which are algorithms inspired by the biological structure and function of the brain. It enables computational models constructed of numerous processing layers to learn data representations with varying levels of abstraction.

These methods have significantly advanced the state-of-the-art in speech recognition, visual object recognition, object detection, and a variety of other fields such as drug discovery and genomics [1].

DL reveals complicated structures in large data sets by using the backpropagation algorithm to demonstrate how a machine's internal parameters that are used to compute the representation in each layer from the representation in the previous layer should be adjusted.

One of the critical distinctions between machine learning and DL models is feature extraction; in machine learning, feature extraction is performed by humans, whereas DL models figure it out on their own.

1.2.2 Artificial neural networks

ANNs are influenced by the way biological neural systems process information, such as the brain [2].

The data processing system is made up of numerous highly interconnected processing elements called **neurons** that collaborate to fix targeted problems. ANNs, like human beings, learn by example and its learning process, just like learning in biological systems, involves changes to the synaptic connections that exist between neurons.

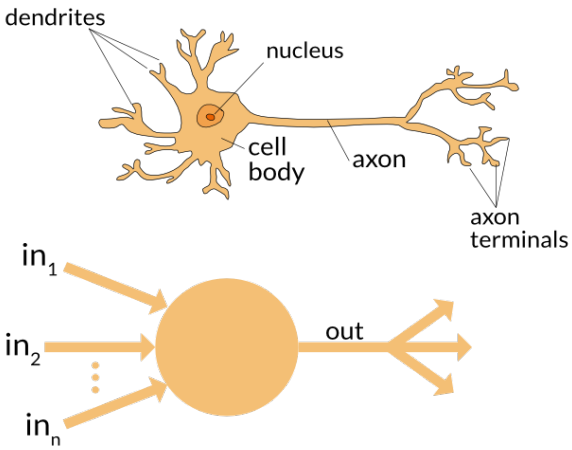


Figure 1.1: Illustration showing the similarities between biological neuron and artificial neuron

1.2.3 Perceptrons

A single-layer neural network is referred to as a **perceptron**. They are made up of four major components: input values, weights and bias, net sum, and an activation function [3].

The procedure begins by multiplying all the input values by their weights. The weighted sum is then computed by adding all the multiplied values together.

The weighted sum would then be applied to the activation function, yielding the output of the perceptron. The activation function is crucial in ensuring that the output is mapped between required values such as (0,1) or (-1,1).

It is critical to understand that the weight of an input indicates the strength of a node. Likewise, the bias value of an input allows you to shift the activation function curve up or down.

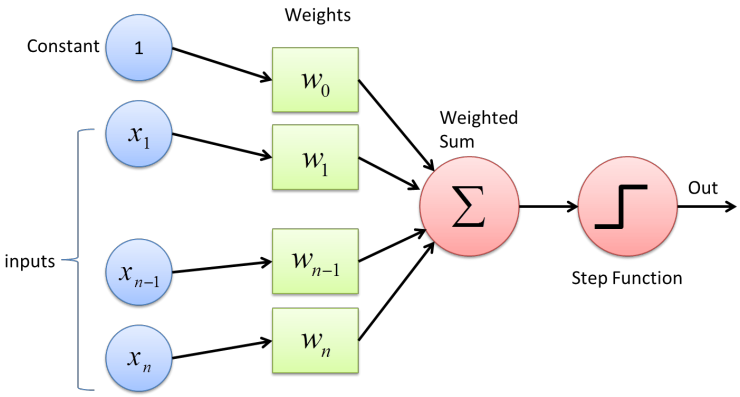


Figure 1.2: Illustration of a Perceptron

1.2.4 Multi-Layer Perceptrons

Highly complex tasks would be impossible for a single neuron to perform. As a result, we use neuron stacks to produce the ideal outputs. The most basic MLP would consist of an input layer, a hidden layer, and an output layer [3].

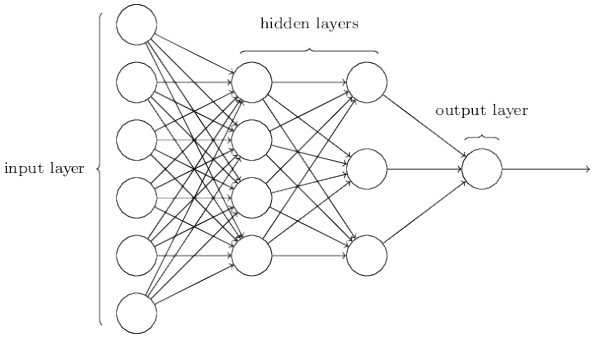


Figure 1.3: Architecture of a Multi-Layer Perceptron containing 2 hidden layers

Each layer contains multiple neurons, and each layer's neurons are all connected to the neurons in the next layer; these networks are also referred to as **FC networks**

MLPs are frequently used for classification, particularly when classes are exclusive, as in the classification of digit pictures (in classes from 0 to 9). After using an activation function, the output layer returns the probability of belonging to each of the classes.

1.2.5 Activation functions

An activation function is a critical part of an ANN. It determines whether a neuron is activated, and it defines the output of the input, or set of inputs of that node [4].

A neural network that lacks an activation function is a linear regression model; the weights and biases would perform a linear transformation without the activation feature. Among the most well-known activation functions we find **Sigmoid** function, **Tanh**, **ReLU**, **Softmax**, etc.

1.2.5.1 Sigmoid

The Sigmoid or Logistic function is one of the most commonly used activation functions [5].

It is defined as follows:

$$Sigmoid(x) = \frac{1}{1 + e^{-}x}$$

The sigmoid transformation produces a continuous range of values between 0 and 1.

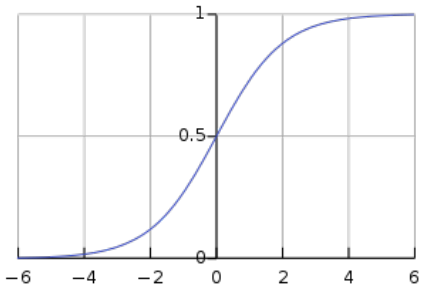


Figure 1.4: Sigmoid activation function

1.2.5.2 Tanh

The Hyperbolic Tan can also be called as symmetric sigmoid is, in fact, a scaled sigmoid function. Keep in mind that the slope for tanh is stronger than for sigmoid (derivatives are steeper).

It is defined as follows:

$$tanh(x) = 2sigmoid(2x) - 1$$

which is:

$$f(x) = tanh(x) = \frac{2}{1 + e^{-}2x} - 1$$

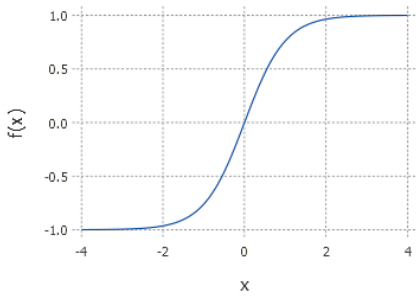


Figure 1.5: Hyperbolic Tangent activation function

1.2.5.3 ReLU

ReLU is another popular function, and it's preferred over sigmoid in recent networks.
it is simply defined as follows:

$$f(x) = max(x, 0)$$

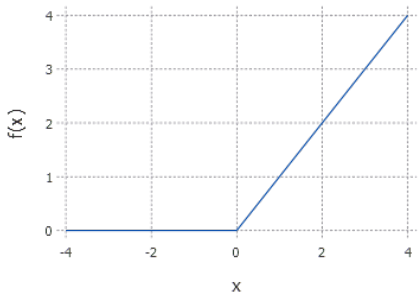


Figure 1.6: Rectified Linear Units activation function

The main advantage of using ReLU is that its derivative value is constant for all inputs greater than 0. The constant derivative value allows the network to train more quickly [6].

1.2.5.4 Softmax

The softmax function is a form of sigmoid function that is particularly useful when dealing with multi-class classification problems, It can be defined as the sum of several sigmoid functions [7].

$$S(x_i) = \frac{e_i^x}{\sum_{j=1}^{n} e_j^x}$$

The sigmoid function would be appropriate if we had a binary output; however, if we have a multiclass classification problem, softmax makes it extremely simple to assign values to each class that can be easily interpreted as probabilities.

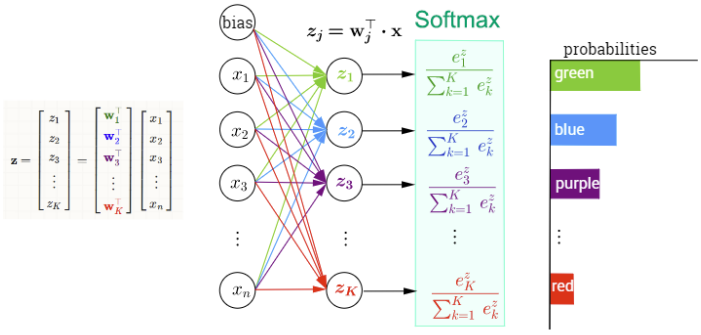


Figure 1.7: Multi-class classification with NN and softmax function

1.2.6 Batch normalization

As an algorithmic technique, batch-normalization speeds up and improves the stability of DNNs [8]. After BN alters the signal at each hidden layer, it looks like this:

$$(1) \quad \mu = \frac{1}{n}\sum_i Z^{(i)} \qquad (2) \quad \sigma = \frac{1}{n}\sum_i (Z^{(i)} - \mu)$$

$$(3) \quad Z^{(i)}_{norm} = \frac{(Z^{(i)} - \mu)}{\sqrt{\sigma^2 - \varepsilon}} \qquad (4) \quad \breve{Z} = \gamma * Z^{(i)}_{norm} + \beta$$

Using (1) and (2), the BN layer first calculates the mean μ and standard deviation σ of the activation values throughout the batch (2). The activation vector $Z^{(i)}_{norm}$ is then normalized with (3). As a result, the output of each neuron follows a conventional normal distribution across the batch (For numerical stability, ε is utilized as a constant).

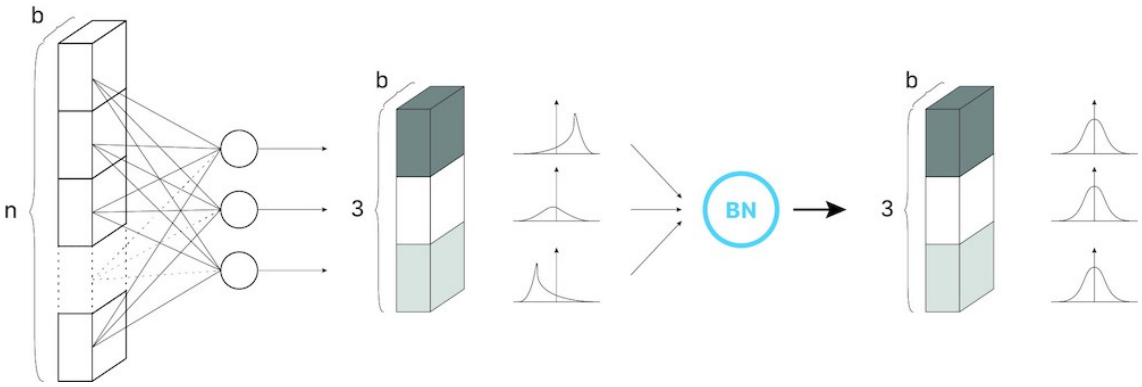


Figure 1.8: Batch normalization first step. Example of a 3-neurons hidden layer, with a batch of size b. Each neuron follows a standard normal distribution from [9].

By applying a linear transformation with two trainable parameters γ and β, it calculates the layer's output $Z^{(i)}_{norm}$ at the end (4). This phase allows the model to select the best distribution for each hidden layer by modifying two parameters: γ allows to alter the standard deviation; β permits to alter the bias by moving the curve to the right or left.

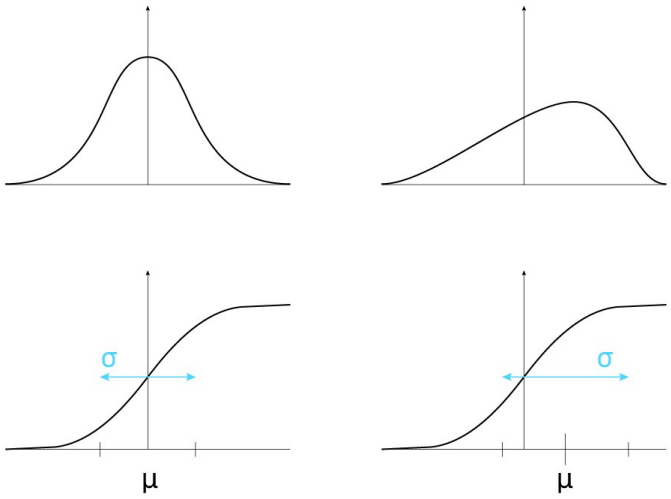


Figure 1.9: Benefits of γ and β parameters. Modifying the distribution (on the top) allows us to use different regimes of the nonlinear functions (on the bottom) from [9]

The network calculates the mean μ and standard deviation σ for the current batch at each iteration. When γ and β are ready, they are trained using gradient descent and an EMA to provide more weight to recent iterations.

1.2.7 Performance metrics

After creating a machine learning model capable of forming classifications, the next step is to calculate its predictive capability. We will need to divide our data into a training set and a validation set to calculate these metrics.

1.2.7.1 Intersection-Over-Union

Simple loss functions are commonly used to train DNNs (e.g., softmax loss). These loss functions are suited for conventional classification tasks where overall classification accuracy is the criterion. The two classes (foreground and background) are highly unbalanced when it comes to image segmentation. Any image segmentation technique's performance is often measured using the IoU approach [10]. Mean IoU is a typical semantic image segmentation assessment metric that computes the IoU for each semantic class before averaging over all classes. IoU is defined as follows:

$$IOU = \frac{True\ Positive}{(True\ Positive + False\ Positive + False\ Negative)}.$$

The predictions are accumulated in a confusion matrix, weighted by a variable, and the metric is then calculated.

1.2.7.2 Accuracy

One of the most important parameters for assessing machine learning models is accuracy. Casually, accuracy refers to the percentage of correct predictions made by our model. The following is the formal definition of accuracy:

$$Accuracy = \frac{Number\ of\ correct\ predictions}{Total\ number\ of\ predictions}$$

The following formula can be used to calculate accuracy if we had positive and negative numbers in a binary classification:

$$Accuracy = \frac{True\ Positives + True\ Negatives}{True\ Positives + True\ Negatives + False\ Positives + False\ Negatives}$$

1.2.7.3 Precision

Precision is a metric that measures how many correct positive predictions have been made. As a result, precision estimates the accuracy of the minority class. The percentage of properly predicted positive instances divided by the total number of positive examples anticipated is used to compute it.

$$Precision = \frac{True\ Positives}{False\ Positives + True\ Positives}$$

1.2.7.4 Recall

The recall is a metric that measures how many correct positive predictions were produced out of all possible positive predictions. Unlike precision, which only considers the accurate positive predictions out of all positive predictions, recall considers the positive predictions that were missed. In this approach, recall offers some indication of the positive class's coverage.

$$Recall = \frac{True\ Positives}{True\ Positives + False\ Negatives}$$

1.2.7.5 F-Measure (F1 Score)

Because classification accuracy is a single metric used to describe model performance, it is extensively utilized. F-measure is a technique for combining precision and recall into a single metric.

Neither precision nor recall can give the complete picture on their own. We might have high precision but poor recall, or vice versa. With the F-measure, it may convey both worries with a single score.

Once precision and recall for a binary or multiclass classification problem have been determined, the two scores may be combined to calculate the F-Measure. This is how the conventional F measure is calculated:

$$F1\,Score = 2 * \frac{Precision * Recall}{Precision + Recall}$$

1.2.7.6 Jaccard similarity coefficient

The Jaccard similarity coefficient is a simple, intuitive formula that can be used for various applications, including image segmentation and other activities. Image segmentation quality is evaluated using this metric, which measures the similarity between the ground truth and segmentation results. The Jaccard similarity coefficient is defined as: let S and G denote the segmentation result and ground truth, respectively.

$$E = \frac{A(G \cap S)}{A(G \cup S)}$$

Where $A(x)$ is the operation of counting quantity. The numerator in the equation refers to the number of matching pixels or true positives. The total number of matching and mismatched pixels is counted in the denominator [11].

1.2.8 Loss functions

When creating a neural network, the network tries to predict the output as similar to the existing value as possible. The loss or cost function also called the error function, is used to assess the network's accuracy. When the network makes mistakes, the cost or loss function attempts to penalize it.

While running the network, our goal is to improve prediction accuracy and reduce error, thereby minimizing the loss function. The most optimized output is the one with the lowest cost or loss function value.

The learning process is centered on reducing costs. Depending on the type of learning task, loss functions can be divided into two major categories — Classification and regression losses [12].

1.2.8.1 Regression losses

- Mean Square Error/Quadratic Loss/L2 Loss

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

- Mean Absolute Error/L1 Loss

$$MAE = \frac{\sum_{i=1}^{n}|y_i - \hat{y}_i|}{n}$$

- Mean Bias Error

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

1.2.8.2 Classification losses

- Hinge Loss/Multi class SVM Loss

$$SVMLOSS = \sum_{j \neq y_i} max(0, s_j - s_y i + 1)$$

- Cross Entropy Loss/Negative Log Likelihood

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

- Focal Loss

$$FocalLoss = FL(p_t) = -(1 - p_t)^{\gamma} log(p_t)$$

FL function for binary classification is a function that generalizes binary cross-entropy by introducing a hyperparameter called the focusing parameter that allows hard-to-classify examples to be penalized more heavily relative to easy-to-classify examples [13].

1.2.9 Hyperparameters

DL models have several hyperparameters, and determining the ideal configuration for these parameters is not easy. Setting the hyperparameters necessitates experience and a lot of observation and experimentation. Configuring hyperparameters, such as gradient descent, LR, epochs, steps per epoch, and batch size, is not straightforward. They function as knobs that can be adjusted throughout the model's training. We must discover the optimal value of these hyperparameters for the model to produce the best results.

1.2.9.1 Gradient descent

Gradient descent is among the most popular optimization algorithms, and it has always been the most common way to optimize neural networks [14].

Gradient descent has three variants that differ in how much data is used to compute the gradient of the objective function.

- Batch gradient descent

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta)$$

- Stochastic gradient descent

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i)}; y^{(i)})$$

- Mini-batch gradient descent

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)})$$

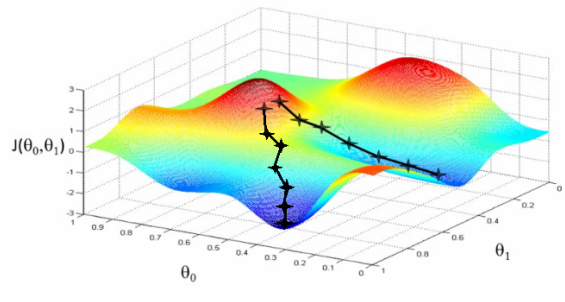
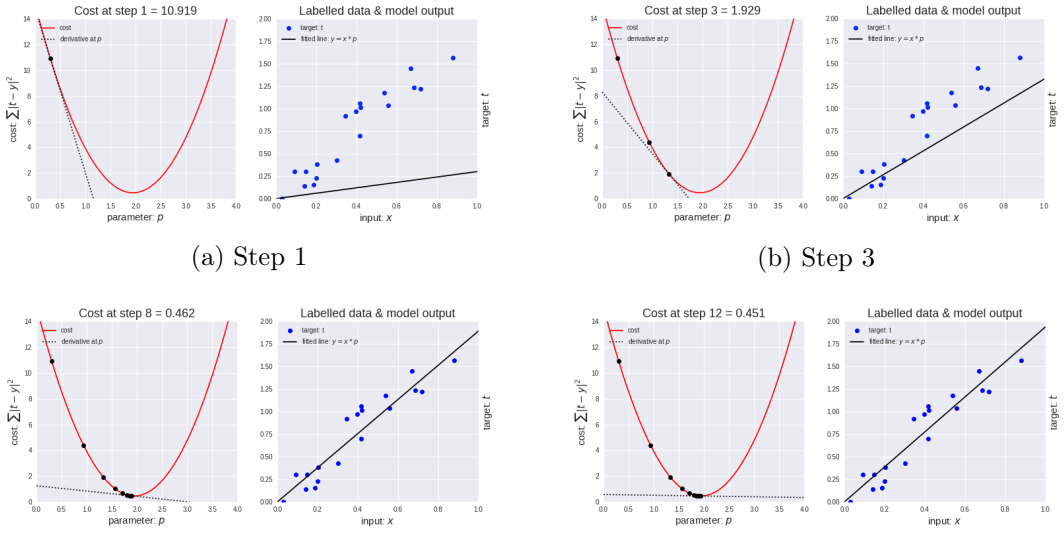


Figure 1.10: 3D Gradient Descent

However, vanilla mini-batch gradient descent does not guarantee good convergence and presents a few challenges, As a result, some algorithms are widely used by the DL community to address those challenges such as Momentum [15], Nesterov accelerated gradient [16], Adagrad [17], RMSprop, Adam [18], AdaMax [18], Nadam [19], etc.

Because the method is iterative, we must obtain the results numerous times to obtain an ideal outcome. The gradient descent's iterative quality aids an under-fitted graph in achieving the best possible fit to the data.



(a) Step 1

(b) Step 3

(c) Step 8

(d) Step 12

Figure 1.11: Gradient descent optimization from [20]

The LR is a much-needed parameter in gradient descent. As seen in the figures above (red curve), the steps are larger at first, indicating a higher LR, and as the point decreases, the LR decreases due to the smaller step size. In addition, the loss function is dropping (which is a good sign).

1.2.9.2 Learning rate

LR is an important **hyperparameter** to optimize for effective DNN training. Even with a constant LR as a starting point, selecting a good constant value for training a DNN is difficult [21].

Dynamic LRs entail multistep tuning of LR values at various stages of the training process and provide high accuracy and rapid convergence. The process of tuning this hyperparameter is a delicate balancing act between underfitting and overfitting. When a model is unable to minimize error for either the test or training set, this is known as **underfitting**. The underlying complexity of the data distributions is too complicated for an underfitting model to fit. On the other hand, when a model is so powerful that it fits the training set too well, **overfitting** occurs, and the generalization error rises. Overfitting can also occur if the LR is too low. Large LRs assist to keep training consistent, but if they are too high, the training will diverge.

In simple words, the LR is the rate at which we descend towards the cost function minima. We should choose the LR carefully because it should not be so high that the optimal solution is missed, nor should it be so low that the network takes forever to converge.

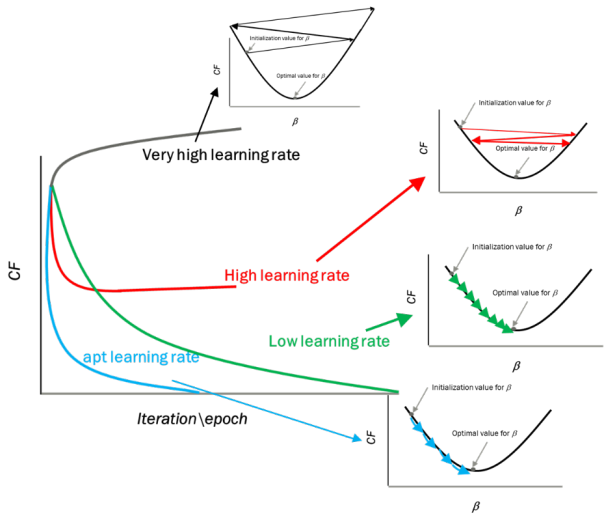


Figure 1.12: Learning Rate expletive illustration

1.2.9.3 Batch size

The batch size is a hyperparameter that defines the number of samples to work through before updating the internal model parameters. Think of a batch as a for-loop iterating over one or more samples and making predictions. When the batch is over, the error value is calculated based on comparing the predictions and the expected output. From this error, the update algorithm is used to improve the model. A training dataset can be divided into one or more batches.

1.2.9.4 Epochs

It is called an epoch when an entire dataset is only processed through the neural network once. We break the epoch into numerous smaller batches since one epoch is too large to provide the computer all at once. Because one epoch is insufficient for updating the weights, we employ numerous epochs. As the number of epochs grows, the weights in the neural network are modified more often, and the curve shifts from underfitting to the optimum to overfitting, as seen in (Figure 1.13). There is no set number of epochs. However, we can assume that the number of epochs is proportional to the diversity of the data.

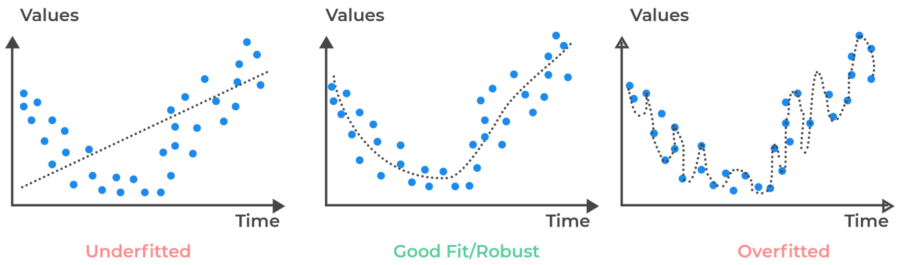


Figure 1.13: Underfitting, Optimum and Overfitting

1.2.9.5 Steps per epoch

The number of times the training loop in the learning algorithm will run to update the parameters in the model is known as step per epoch. It will process a block of data, which is essentially a batch, at each loop iteration. The gradient descent technique is commonly used in this loop. Because this will use all the data points, one batch size worth at a time, the steps per epoch are traditionally calculated as train length divided by the batch size. In the case of augmented data, we multiply the previous operation by 2 or 3, and so on. However, if the training has been going on for too long, we'll just keep to the old method.

1.2.10 Forward propagation and backpropagation

1.2.10.1 Forward propagation

The input flow via the hidden layers to the output layers is referred to as forward propagation. It is the movement of information in only one direction. The layer input will provide information to hidden layers, generating output that keeps moving in the same direction without going backward.

1.2.10.2 Backpropagation

Backpropagation of error is a technique used to train feed-forward neural networks. An iterative procedure that adjusts network weight parameters based on the gradient of an error measure is a specific implementation of backpropagation. The procedure is carried out by calculating an error value for each output unit and then propagating the error values through the network [22].

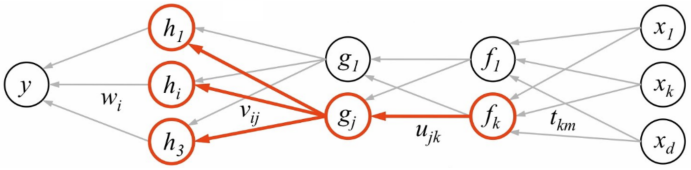


Figure 1.14: Backpropagation expletive illustration

In simple terms, when we define a neural network, we specify arbitrary weights and biases for each node. We can calculate the network's error after receiving the output for a single iteration. This error, along with the gradient of the cost function, is then fed back into the network to update the network's weights. These weights are then updated to reduce errors in subsequent iterations, so backpropagation is updating weights using the gradient of the cost function.

1.2.11 Data augmentation

When there are just a few training instances available, data augmentation is needed to teach the network the required invariance, and resilience [23]. It is the process of applying a sequence of deformations to a set of labeled training data to generate more diverse and extra training data. Geometric transformations (flipping and rotation, clipping and scaling), color space transformations (alteration of RGB channel intensities), kernel filters, mixing images, random erasing, adversarial training,

neural style transfer, noise injection, and meta-learning schemes are just a few of the data augmentation techniques that have been proposed [24][25]. The most fundamental premise of data augmentation is that the deformations used should not modify the labels' semantic meaning [24].

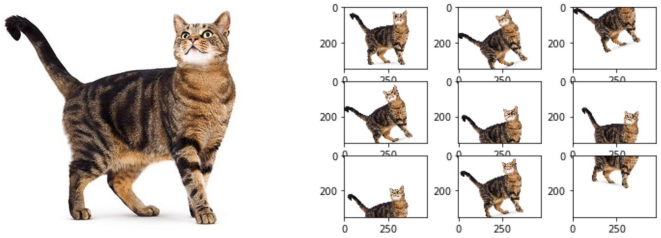


Figure 1.15: Representation of data augmentation

1.2.12 Global Average Pooling

Global Average Pooling replaces the traditional FC layers in CNN. The aim is to construct one feature map for each matched category of the classification task in the final Multilayer Perceptron layer. Rather than constructing FC layers on top of the feature maps, we take the average of each feature map and feed the resultant vector directly into the softmax layer. Global average pooling has an advantage over FC layers in that it is more natural to the convolution structure by enforcing correspondences between feature maps and categories. As a result, the feature maps may be thought of as category confidence maps. Another benefit of global average pooling is that there are no parameters to tune. Thus overfitting is prevented at this layer [26].

1.3 Basic deep learning architectures

1.3.1 Convolutional neural networks

CNNs are the obvious choice for image recognition tasks because of the capability of multi-layer networks trained with gradient descent to learn complex, high-dimensional, nonlinear mappings from extensive collections of examples. Unlike the traditional models of pattern recognition that have problems dealing with large images, which may have several hundred variables (pixels) [1].

There are four important elements (layers) when stacked together they make a CNN architecture, which are **convolutional layers**, **nonlinear layers**, **pooling layers** and **FC layers**.

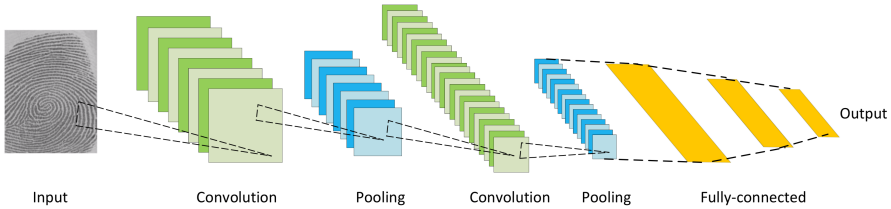


Figure 1.16: Architecture of a simple CNN from [27]

A convolutional layer is where the extraction of various features from the input images happens using a filter (kernel) that makes a dot product with the input image; the resulting output is a feature map that is fed to a nonlinear layer, which applies an activation function (ReLU, Softmax, etc) to feature maps, thus enabling the network to model nonlinear function.

Usually, a convolutional layer is followed by a pooling layer that uses statistical information (such as mean, max) to decrease the size of the feature maps by replacing small neighborhoods.

Lastly, an FC layer is where all of the previous outputs of the previous layers get flattened and fed to some mathematical functions to get the last result see (Figure 1.16).

Over the years, many successful CNNs architectures helped make smaller numbers of parameters than FC neural networks because all of the receptive fields in a layer share parameters(weights). Some of the widely known architectures include AlexNet [21], VGGNet [22], ResNet [28], and GoogLeNet [29].

1.3.2 Recurrent neural networks and the LSTM

For sequential data, RNNs are one of the powerful models for data. RNNs can estimate the next value of the input data based on the old information (memory) according to a special mechanism.

RNNs are used in many fields such as speech recognition, machine translation, music composition, gamma learning, stock prediction, self-driving cars, to name a few.

The RNN consists of 3 layers, the input and output are the first and last layers in this order, and the middleware is the layer that contains a complex form of neural networks. A cyclic form characterizes this layer; in each cycle, the network makes a prediction until it ends at the desired output see (Figure 1.17).

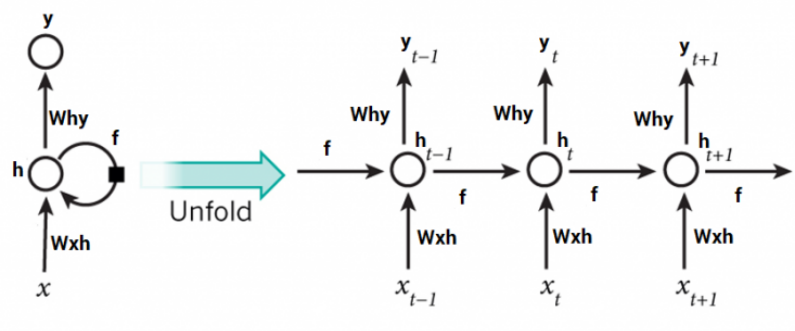


Figure 1.17: Architecture of a simple RNN

Since many years, developers have developed several types of RNN such as Elman networks [30], Jordan networks [31], and Echo State Networks [32]. Moreover, in the last few years, neural networks based on LSTM cells have become the most performing neural network.

The problem with RNNs is that they have a very short-term memory, and in many real-world applications, they will have terrible results with long sequences and often suffer from problems such as vanishing/exploding gradients. LSTM was the key to avoid these problems, with an appropriate gradient-based learning algorithm[33].

The LSTM architecture, see (Figure 1.18). includes three gates (input gate, output gate, forget gate) enforcing constant error flow through internal states of special units called memory cells that store values for arbitrary time intervals.

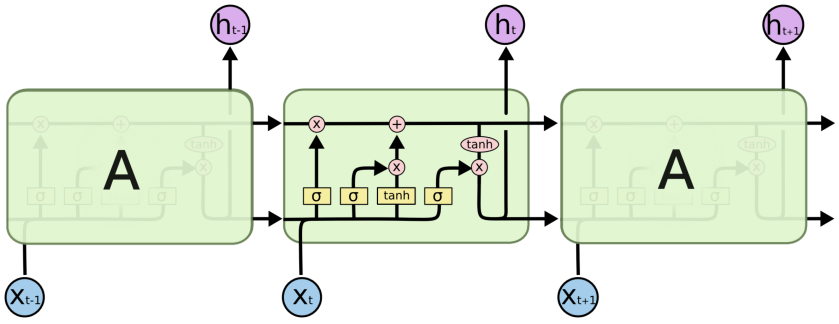


Figure 1.18: Architecture of LSTM from [34]

1.3.3 Encoder-Decoder and Auto-Encoder Models

ED models are widely used, and they were proven to be very effective for dealing with translation problems as well as for sequence models in NLP [35].

An ED model is architectured with two networks in mind, an encoder network (might be of any type like CNN, RNN) and a corresponding decoder network; the encoder takes a sequence as the input and compresses it into a fixed-length numeric vector (feature), the decoder then predicts the output from that vector see (Figure 1.19).

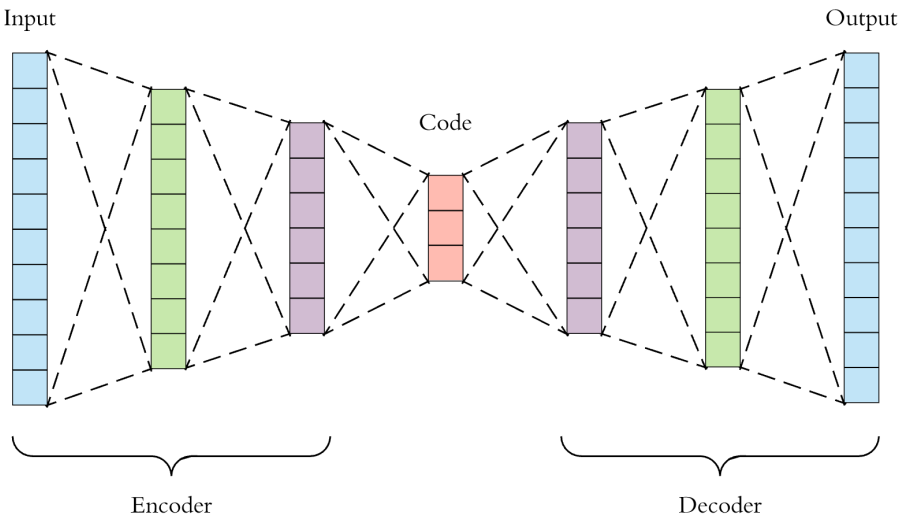


Figure 1.19: Architecture of an Encoder-Decoder from [36]

An AE is a particular variant of the ED models with a distinguished change. The number of neurons is the same in the input and the output (called reconstructed input). Therefore we can expect that the input and the output to be the same sequence of data.

There are many well-known AE models such as the SDAE [37], and the VAE [38], etc.

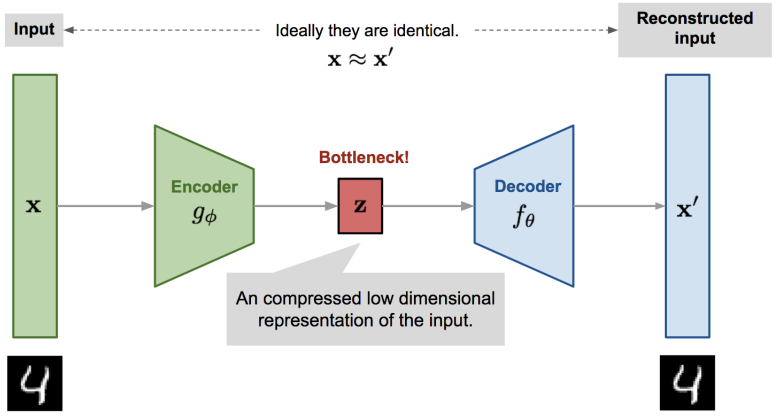


Figure 1.20: Architecture of Auto-Encoder from [39]

1.3.4 Generative Adversarial Networks

One of the most recent models of DL are GANs, they are used heavily in image manipulation and generation, and they can also be deployed in tasks like understanding risk, recovery in health care and even in pharmacology, etc.

They consist of two crucial components a generator network **G** and a discriminator network **D**. The mechanism behind it is by feeding a random noise vector $\boldsymbol{z}$ to **G** so it can use it to generate fake samples that are fed to **D**. In meanwhile, real images are fed to **D** and its job is to try and distinguish real data from fake data. If discriminator **D** fails in its job in distinguishing real data from fake data, then

we get a well-trained generator model **G**, which has learned the distribution of real data [40].

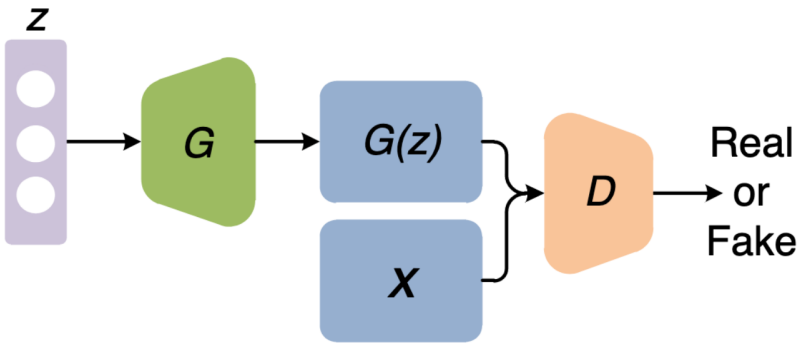


Figure 1.21: Architecture of Generative Adversarial Networks

1.4 Deep learning frameworks

1.4.1 TensorFlow

TensorFlow is a free and open-source tool for building and deploying machine learning models. It incorporates all of the standard algorithms and patterns required for machine learning, concentrating on the scenario rather than learning all of the fundamental math and logic. It is targeted at anyone from hobbyists to technical developers to artificial intelligence experts pushing the envelope. It also allows models to be deployed to the internet, cloud, smartphone, and embedded systems [41].

Figure 1.22: Tensorflow logo

TensorFlow has the following features:

- Its essence is somewhat similar to **NumPy**, but with GPU support;
- It also supports distributed computing (across multiple devices and servers); It has a kind of JIT compiler that helps it to customize computations for speed and memory usage: it extracts the computation graph from a Python function, optimizes it (for example, by pruning unused nodes), and then runs it efficiently (e.g., by automatically running independent operations in parallel);
- Computation graphs can be exported to a portable format, allowing to train a TensorFlow model in one environment (for example, Python on Linux) and then run it in another (for example, Java on an Android device). It includes auto diff as well as some excellent optimizers like **RMSProp**, **Nadam**, and **FTRL**, allowing you to effectively reduce a variety of loss functions.

- TensorFlow has a lot of functionality built on top of these key features: the most critical is **tf.keras**, but it also has data loading and preprocessing ops (**tf.data**, **tf.io**, and so on), gif processing ops (**tf.image**), signal processing ops (**tf.signal**), and etc [42].

1.4.2 PyTorch

PyTorch is an end-to-end machine learning platform with a user-friendly front-end, distributed training, and ecosystem of tools and libraries that enable quick, modular exploration and productive development. PyTorch has a rich ecosystem of software and libraries for expanding it and promoting growth in fields ranging from machine vision to reinforcement learning, thanks to an engaged group of researchers and developers [43].

Figure 1.23: Pytorch logo

PyTorch is well-supported on major cloud platforms, allowing for frictionless deployment and scaling through pre-configured images, wide GPU training, and the ability to execute models in a manufacturing environment, among other features.

1.4.3 Keras

Keras is a human-centric API, not a machine-centric one. Keras adheres to best practices for minimizing cognitive burden, such as providing reliable and easy APIs, reducing the number of user activities needed for typical use cases, and providing transparent and actionable error messages. It comes with a lot of documents and developer manuals [44].

Figure 1.24: Keras logo

Keras provides the opportunity to do new experiments, test more hypotheses, and do better than the competition. Keras is an industrial-strength architecture built on top of TensorFlow 2.0 that can scale to massive clusters of GPUs or an entire TPU pod. It is not only feasible; it is also easy.

Keras is a crucial component of the TensorFlow 2.0 ecosystem, and it covers every aspect of the machine learning process, from data management to hyperparameter training to deployment solutions.

Keras is used by CERN, NASA, the National Institutes of Health, and several other research institutions worldwide. it has the low-level stability to execute any research hypothesis while still providing high-level convenience functionality to shorten experimentation intervals.

Keras is the DL solution of choice for many university courses due to its ease of use and emphasis on user experience. It is generally viewed as one of the most effective methods for studying DL.

1.5 Hardware used in deep learning

1.5.1 Central processing units

A CPU is a computing machine that includes several ALUs, a Control Unit to control certain ALUs, a Cache Memory, and a DRAM. Since CPUs are more powerful, computers can perform any task with precision and versatility. This versatility comes from the CPU's memory power, which can exceed 1TB of RAM, allowing it to fetch memory packages in the RAM faster and with lower latency CPUs, unlike GPUs, do not yet meet the stringent standards of DL. CPUs, on the other hand, assist GPUs by feeding them enough data, and reading/writing files from/to RAM/HDD during preparation [45].

Many CPUs, including AMD Ryzen 9 3900X, Intel Core i9-9900K, AMD Ryzen Threadripper 3990X, AMD Ryzen 5 2600, are currently considered the most appropriate when it comes to training DL models.

1.5.2 Graphics processing units

GPUs are graphics processors that create polygon-based computer graphics. GPUs have acquired huge computing powers in recent years, because of the complexity and desire for realism in recent video games and graphic engines. NVIDIA is the market leader, with processors with thousands of cores built to compute at nearly 100% performance. It turns out that these processors are also capable of performing neural network computations and matrix multiplications [46].

GPUs are currently the standard for training DL systems, whether they are CNNs (CNN) or RNNs (RNN). In only a few milliseconds, they will practice on massive batches of images, such as 128 or 256 images.

However, they absorb 250 W and need a full PC with an additional 150 W of power to run. A greater GPU system can use up to 400 watts of power.

ZOTAC GeForce GTX 1070, ASUS ROG Strix Radeon RX 570, Gigabyte GeForce GT 710, and Sapphire Radeon Pulse RX 580 are among the top-rated GPUs used for training DL neural networks.

1.5.3 Tensor processing units

From Google-designed ASICs we find TPUs that accelerate machine learning caseloads. TPUs was designed from the ground up with Google's extensive machine learning expertise and leadership. Linear algebra computing, which is extensively used in machine learning applications, is accelerated by cloud TPU tools.

When training big, complex neural network models, TPUs reduce the time-to-accuracy. They will converge models that historically took weeks to learn on other hardware platforms in hours. TPUs in the cloud are tailored to individual workloads. You may want to use GPUs or CPUs on Compute in some cases. In certain cases, you might want to run the machine learning workloads on Compute Engine instances using GPUs or CPUs. In general, you should choose the right hardware for your workload [47].

1.6 Conclusion

This chapter discussed the most common DL concepts and delved deeper into their architectures, the main frameworks used by the community, and the hardware required to achieve the best results.

In the following chapter, we will discuss DL applications in computer vision, specifically semantic image segmentation.

Chapter 2

Semantic Segmentation

2.1 Introduction

In computer vision, semantic segmentation is considered a challenging task. DL techniques have immensely enhanced the performance of semantic segmentation in recent years. Several innovative methods have been proposed, making segmentation algorithms more efficient and precise, and various new applications have widely used them.

In this chapter, we will start by giving some general concepts of semantic segmentation, starting by defining the term itself, noting its aims, old methods, and techniques, and comparing it to other everyday computer vision tasks, and we will finish by covering the impact of DL in semantic segmentation, as well as the link between the two. After that, we will go over some of the most typical CNN architectures by giving information about its production and content. We will target LeNet-5, AlexNet, VGG-16, ResNet-50, GoogleNet and MobileNet. Most importantly, we will go through the state-of-the-art methods in DL-based semantic image segmentation. These methods include FCN, SegNet, U-net, DeepLab, and PSPNet.

This chapter will attempt to detail the semantic segmentation of images and the use of DL in this field.

2.2 General concepts of semantic segmentation

2.2.1 Definition

Semantic segmentation plays a vital role in computer vision. It is the process of categorizing each pixel as belonging to a specific label. This categorization is part of scene comprehension, or better interpreting the image's overall context. It does not differ between instances of the same item. For example, if an image contains two women, semantic segmentation assigns the same label to all the pixels for both women see (figure 2.1).

Many new applications have emerged from this field, such as handwriting recognition, medical imaging, autonomous driving, industrial robots, indoor navigation, virtual or augmented reality systems, portrait mode in new smartphones, and even social media filters and virtual make-up, which required the use of semantic segmentation.

Figure 2.1: An example of semantic segmentation

2.2.2 Comparison with other computer vision tasks

Semantic segmentation differentiates itself from other familiar computer vision tasks like image classification, object detection, and instance segmentation.

The goal of image classification is to appoint one or more category labels to an entire image. In other words, an image classification algorithm tries to tell us which objects are present in a given image.

Object detection takes things a step further. It must understand what objects take place in an image and where they are in the image scene.

In contrast to these two tasks, semantic segmentation has relatively high requirements because it aims to accurately divide each object region from the background region while defining the targeted object's boundaries. It is far more complicated

than the other two tasks because it requires fully bridging the semantic gap between low-level features and high-level semantics [48].

In order to advance the evolution of semantic segmentation, instance segmentation assigns various labels to distinguish instances of objects related to the same class. As a result, instance segmentation can be characterized as a technique for simultaneously handling the problems of object detection, and semantic segmentation [49].

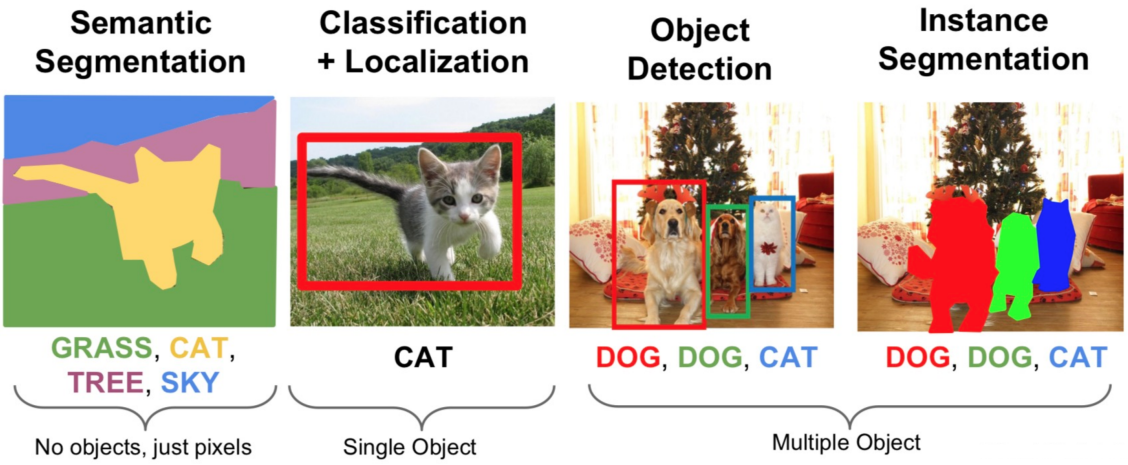


Figure 2.2: An example of various computer vision tasks

2.2.3 Methods and Techniques

Before the introduction of DL, image segmentation challenges were solved using conventional machine learning approaches such as GLS [50], CRFs [51] or using popular algorithms like support vector machines, random forest, and K-means clustering. However, as with most image-related research problems, DL outperformed earlier solutions and has become state-of-the-art when dealing with semantic segmentation.

2.2.4 Deep learning for semantic segmentation

The significance of scene understanding as a core computer vision problem is illustrated by the fact that a growing number of applications benefit from inferring knowledge from visual images. Various conventional computer vision and machine learning techniques have been used in the past to tackle this problem.

Despite their prominence, the DL revolution has reversed conditions so that many computer vision problems, including semantic segmentation, are now being solved using deep architectures, typically CNNs, which outperform other approaches in terms of accuracy and efficiency.

2.3 Common convolutional neural network architectures

Certain deep networks have made such substantial contributions to the domain that they are now broadly accepted standards. This is truly the case for LeNet-5, AlexNet, VGG-16, ResNet-50, and GoogleNet (InceptionV1). Because of their significance, they are now used as a basic building block in several segmentation architectures. As a result, we will devote this section to going over them, with the help of the legend below see (figure 2.3).

We will use the following mathematical equation to calculate the output of convolutional layers:

$$\left[\frac{n+2p-f}{s}+1\right] * \left[\frac{n+2p-f}{s}+1\right]$$

Where we consider a $n * n$ image as an input, a $f * f$ filter, a padding p, and a stride s.

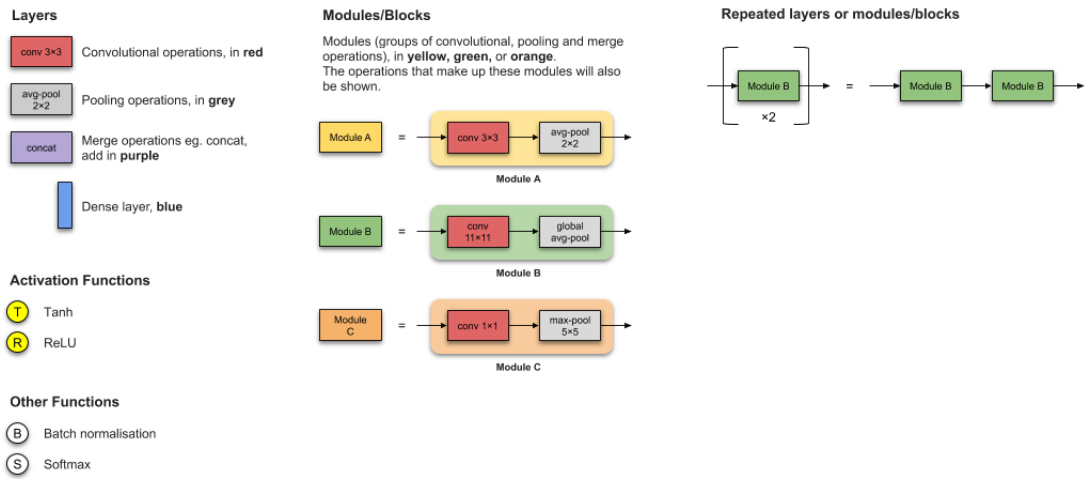


Figure 2.3: Legend used for the various architectures from [39]

2.3.1 LeNet-5

LeNet-5 is a pioneering 7-level convolutional network developed by LeCun et al. in 1998 [52] to recognize handwritten numbers on checks scanned in 32x32 pixel grayscale input images used by several banks. It is among the most fundamental architectural designs. It has 61K parameters and two convolutional, and three FC layers.

This architecture has become the industry standard: convolutions with activation functions are stacked, layers are pooled, and the network is finished with one or more completely linked layers. It has the architecture represented in the following Table, see (Table 2.1).

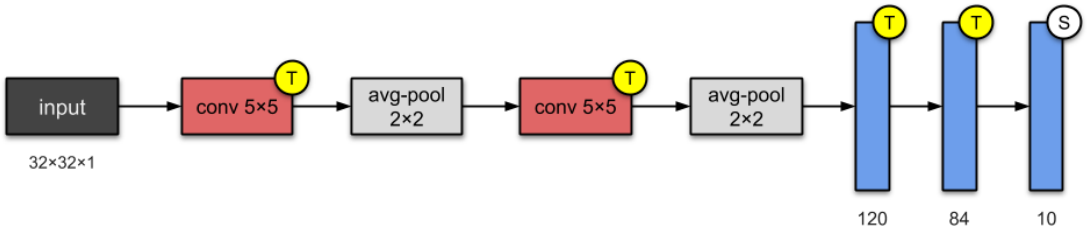


Figure 2.4: LeNet architecture from [39]

Layer		Feature Map	Size	Kernel Size	Stride	Activation	Parameters
Input	Image (Grayscale)	1	32x32	-	-	-	-
1	Convolution	6	28x28	5x5	1	tanh	156
2	Average Pooling	6	14x14	2x2	2	tanh	0
3	Convolution	16	10x10	5x5	1	tanh	2416
4	Average Pooling	16	5x5	2x2	2	tanh	0
5	FC	120	1x1	5x5	1	tanh	48120
6	FC	-	84	-	-	tanh	10164
Output	FC	-	10	-	-	softmax	850
Total number of parameters							61,706

Table 2.1: LeNet structural details

2.3.2 AlexNet

AlexNet beat all previous competitors in 2012 [53]. The network's architecture was quite similar to LeNet, but it was deeper, with 62M parameters and more filters per layer and layered convolutional layers.

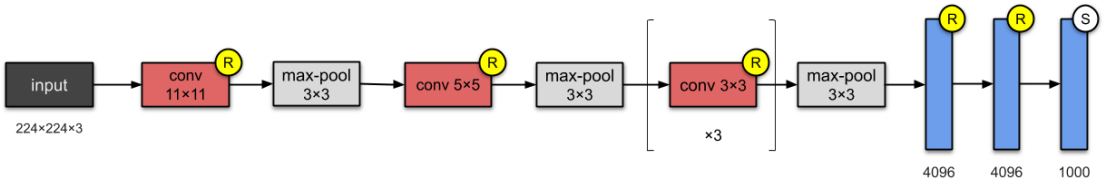


Figure 2.5: AlexNet architecture from [39]

Convolutions, max pooling, dropout, data augmentation, ReLU activations, and SGD with momentum (which is a technique that aids in the acceleration of gradient vectors in the proper directions, resulting in quicker convergence) were among the features. After each convolutional and FC layer, it added ReLU activations. See the difference in the layers and parameters in Table (2.2).

Layer		Feature Map	Size	Kernel Size	Stride	Activation	Parameters
Input	Image (RGB)	1	227x227x3	-	-	-	-
1	Convolution	96	55x55x96	11x11	4	relu	34944
	Max Pooling	96	27x27x96	3x3	2	relu	0
2	Convolution	256	27x27x256	5x5	1	relu	614656
	Max Pooling	256	13x13x256	3x3	2	relu	0
3	Convolution	384	13x13x384	3x3	1	relu	885120
4	Convolution	384	13x13x384	3x3	1	relu	1327488
5	Convolution	256	13x13x256	3x3	1	relu	884992
	Max Pooling	256	6x6x256	3x3	2	relu	0
6	FC	-	4096	-	-	relu	37752832
7	FC	-	4096	-	-	relu	16781312
Output	FC	-	1000	-	-	softmax	4097000
Total number of parameters							62,378,344

Table 2.2: AlexNet structural details

2.3.3 VGG-16

Simonyan and Zisserman created VGGNet [22], the runner-up in the ILSVRC 2014 competition. VGGNet comprises 13 convolutional and three FC layers, all of which build on the ReLU heritage established by AlexNet. While prior AlexNet derivatives focused on reduced window sizes and strides in the first convolutional layer, VGG addresses another critical element of CNNs, which is depth.

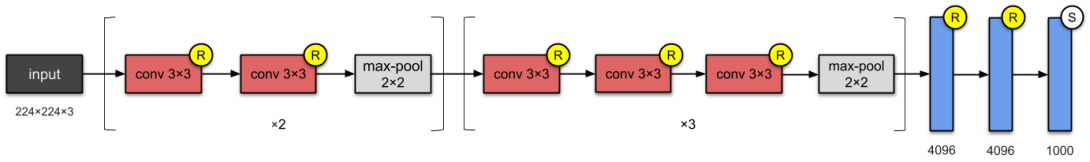


Figure 2.6: VGG-16 architecture from [39]

The distinction between VGG and AlexNet is that it has numerous characteristics that set it apart from other competing models:

- Rather than using big receptive fields like AlexNet (11x11 with a stride of 4), VGG employs extremely small receptive fields (3x3 with a stride of 1). The decision function is much more discriminative now that there are three ReLU units instead of simply one. There are also fewer parameters (27 channels as opposed to AlexNet's 49 channels).
- VGG employs 1x1 convolutional layers to increase the nonlinearity of the decision function without modifying the receptive fields.
- Because of the small dimensions of the convolution filters, VGG can have a large number of weight layers.

Layer		Feature Map	Size	Kernel Size	Stride	Activation	Parameters
Input	Image (RGB)	1	224x224x3	-	-	-	-
1	2xConvolution	64	224x224x64	3x3	1	relu	38720
	Max Pooling	64	112x122x64	2x2	2	relu	0
3	2xConvolution	128	112x122x128	3x3	1	relu	221440
	Max Pooling	128	56x56x128	2x2	2	relu	0
5	3xConvolution	256	56x56x256	3x3	1	relu	1475328
	Max Pooling	256	56x56x256	2x2	2	relu	0
7	3xConvolution	512	28x28x512	3x3	1	relu	5899776
	Max Pooling	512	14x14x512	2x2	2	relu	0
10	3xConvolution	512	14x14x512	3x3	1	relu	7079424
	Max Pooling	512	7x7x512	2x2	2	relu	0
14	FC	-	4096	-	-	relu	102764544
15	FC	-	4096	-	-	relu	16781312
Output	FC	-	1000	-	-	softmax	4097000
Total number of parameters							134,264,641

Table 2.3: VGG-16 structural details using padding = 1

2.3.4 ResNet

During the ILSVRC 2015, Kaiming He et al. proposed the so-called ResNet [28], a novel design with "skip connections" and substantial BN. They are also known as gated units or gated recurrent units, and they look a lot like recent successful RNN components.

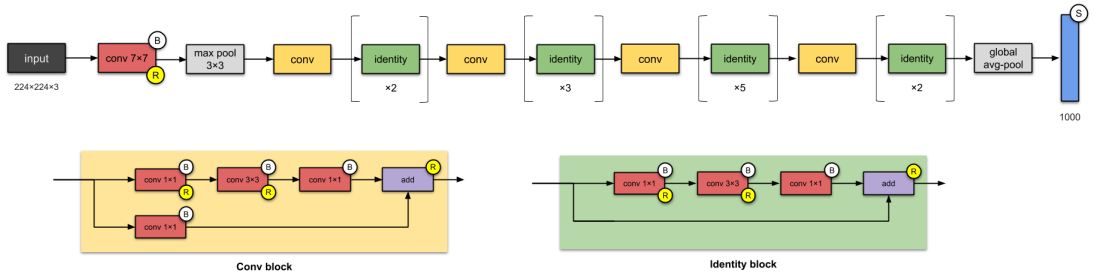


Figure 2.7: ResNet-50 architecture from [39]

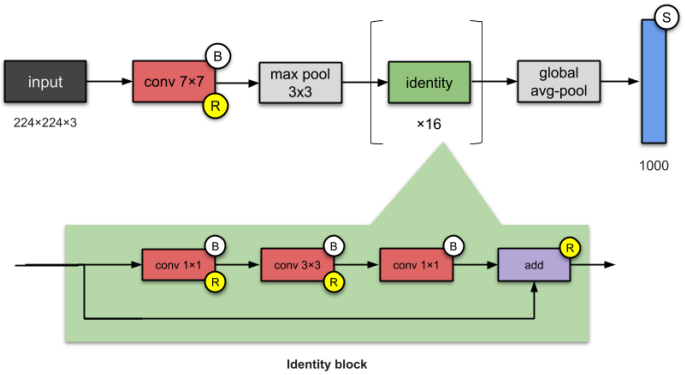


Figure 2.8: ResNet identity block from [39]

Many convolutional architectures include a skip connection module as a standard module [54]. We can use it to provide an alternate path for the gradient (with

backpropagation). Skip connections in deep architectures, as the name implies, skip some layers in the neural network and send the result of one layer as the input to the subsequent layers (instead of only the next one).

They trained a neural network with 152 layers without sacrificing the model's generalization capacity while maintaining a lower complexity than VGGNet using this methodology. The basic building blocks for ResNets are the convolutions and identity blocks. It has approximately a total of 26 million parameters.

Layer	Input size	Output size	Filter	Parameters
Convolution1	224x224x3	112x122x64	7x7x64, stride = 2	9472
3xConvolution2 (Convolution Block + 2x(Identity block))	112x122x64	56x56x64	3x3 Max pooling, stride = 2	0
			1x1x64 3x3x64 1x1x256	214800
4xConvolution3 1x(Convolution Block) + 3x(Identity block)	56x56x64	28x28x128	1x1x128 3x3x128 1x1x512	1216000
6xConvolution4 1x(Convolution Block) + 5x(Identity block)	28x28x128	14x14x256	1x1x256 3x3x256 1x1x1024	7088128
3xConvolution5 1x(Convolution Block) + 2x(Identity block)	14x14x256	7x7x512	1x1x512 3x3x512 1x1x2048	14953472
FC	7x7x512	1x1x1000	Average pooling, 1000-d FC, Softmax	2049000
Total number of parameters				25,636,712

Table 2.4: ResNet structural details

2.3.5 GoogleNet

The Inception architecture is used to create GoogLeNet, a sort of CNN. It took first place in the classification challenge at ILSVLC-2014 [29]. It employs Inception modules, whose concept is to employ all of the operations simultaneously. It runs many kernels of varying sizes in parallel on the same input map, concatenating their results into a single output. Having multiple-size filters function on the same level. The neural network would effectively become "wider" rather than "deeper". The authors of this architecture intended for the inception module to represent this idea.

The network was built with computational efficiency and practicality in mind, allowing inference to be performed on individual devices with minimal processing capabilities, particularly those with a small memory footprint. When only layers with parameters are counted, the network has 22 layers (or 27 layers if we also count pooling).

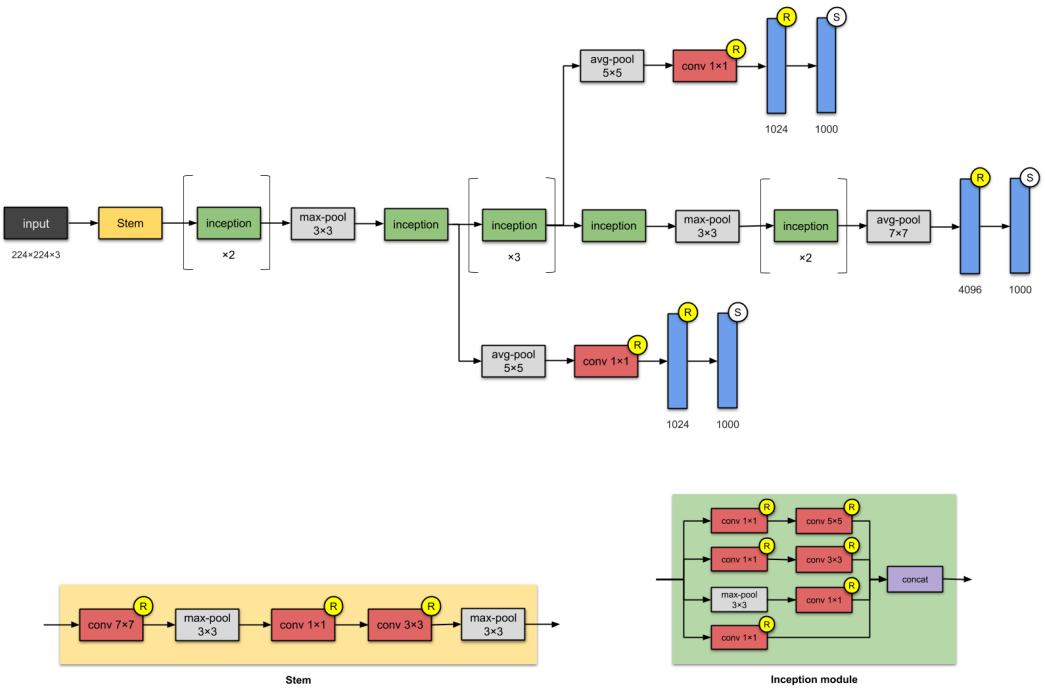


Figure 2.9: GoogleNet architecture (Inception V1) from [39]

The network is constructed using around 100 layers (independent building elements). This number, however, is based on the machine learning infrastructure system in use. Unlike VGGNet and AlexNet, the network structure is substantially different. In the center of the network, it has a 1×1 Convolution. Also, global average pooling is employed at the network's conclusion rather than employing completely linked layers. GoogLeNet has around 6.8 million parameters (without auxiliaries layers) 9 times fewer than AlexNet and 20 times fewer than its competitor VGG-16.

Type	Patch size / Stride	Output size	Depth	#1x1	#3x3 Reduce	#3x3	#5x5 Reduce	#5x5	Pool Proj	Parameters
Convolution	7x7/2	112x112x64	1	-	-	-	-	-	-	2.7K
Max Pooling	3x3/2	56x56x64	0	-	-	-	-	-	-	-
Convolution	3x3/1	56x56x192	2	-	64	192	-	-	-	112K
Max Pooling	3x3/2	28x28x192	0	-	-	-	-	-	-	-
Inception (3a)	-	28x28x256	2	64	96	128	16	32	32	159K
Inception (3b)	-	28x28x480	2	128	128	192	32	96	64	380K
Max Pooling	3x3/2	14x14x480	0	-	-	-	-	-	-	-
Inception (4a)	-	14x14x512	2	192	96	208	16	48	64	364K
Inception (4b)	-	14x14x512	2	160	112	224	24	64	64	437K
Inception (4c)	-	14x14x512	2	128	128	256	24	64	64	463K
Inception (4d)	-	14x14x528	2	112	144	288	32	64	64	580K
Inception (4e)	-	14x14x832	2	256	160	320	32	128	128	840K
Max Pooling	3x3/2	7x7x832	0	-	-	-	-	-	-	-
Inception (5a)	-	7x7x832	2	256	160	320	32	128	128	1072K
Inception (5b)	-	7x7x1024	2	384	192	384	48	128	128	1388K
Average Pooling	7x7/1	1x1x1024	0	-	-	-	-	-	-	-
Dropout (40%)	-	1x1x1024	0	-	-	-	-	-	-	-
Linear	-	1x1x1000	1	-	-	-	-	-	-	1000K
Softmax	-	1x1x1000	0	-	-	-	-	-	-	-
Total number of parameters										6,8M

Table 2.5: GoogleNet structural details

2.3.6 MobileNet

The MobileNet model is TensorFlow's first mobile computer vision model, and it was developed for usage in mobile applications. This method uses depth-wise separable convolutions. Compared to a network with simple convolutions of the same depth in the nets, it vastly decreases the number of parameters. As a consequence, lightweight DNNs are created. Two procedures are used to create a depthwise separable convolution:

- Depthwise convolution: it's is the channel-wise $D_k * D_k$ spatial convolution. It is also a map of a single convolution applied to each input channel. As a result, the number of output channels equals the number of input channels. The cost of computing it is: $D_f^2 * M * D_k^2$.
- pointwise convolution: A convolution with a kernel size of 1x1 simply mixes the depthwise convolution's characteristics. The cost of computing it is: X

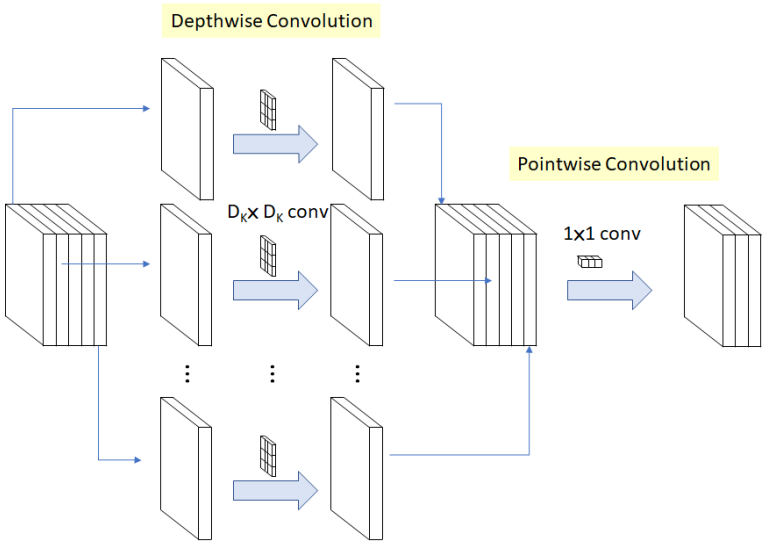


Figure 2.10: MobileNet architecture from [55]

MobileNet is a CNN class that was open-sourced by Google [56], and it provides us with an ideal starting point for training our ultra-small and ultra-fast classifiers. the architecture of this class will be represented in the following table:

Layer	Input size	Stride	Filter	Parameters
Convolution1	224x224x3	2	3x3x3x32	992
Convolution_dw1	112x112x32	1	3x3x32 dw	416
Convolution_pw1	112x112x32	1	1x1x32x64	2304
Convolution_dw2	112x112x32	2	3x3x64 dw	832
Convolution_pw2	56x56x64	1	1x1x46x128	8704
Convolution_dw3	56x56x128	1	3x3x128 dw	1664
Convolution_pw3	56x56x128	1	1x1x128x128	16896
Convolution_dw4	56x56x128	2	3x3x128 dw	1664
Convolution_pw4	56x56x128	1	1x1x128x256	33792
Convolution_dw5	56x56x256	1	3x3x256 dw	3328
Convolution_pw5	56x56x256	1	1x1x256x256	66560
Convolution_dw6	56x56x256	2	3x3x256 dw	3328
Convolution_pw6	14x14x256	1	1x1x256x512	133120
5xConvolution_dw7	14x14x512	1	3x3x512 dw	33280
5xConvolution_pw7	14x14x512	1	1x1x512x512	1320960
Convolution_dw8	14x14x512	2	3x3x512 dw	6656
Convolution_pw8	7x7x512	1	1x1x512x1024	528384
Convolution_dw9	7x7x1024	2	3x3x1024 dw	13312
Convolution_pw10	7x7x1024	1	1x1x1024x1024	1052672
Average pooling	7x7x1024	1	7x7x1024	0
FC	7x7x1024	1	1024x1000	1025000
Softmax	1x1x1000	1	-	0
Total number of parameters				4,253,864

Table 2.6: MobileNet structural details

2.4 Deep learning based semantic segmentation methods

The consistent excellence of DL methods in numerous high-level computer vision tasks – mainly supervised approaches such as CNNs for image classification or object detection – prompted researchers to examine the capabilities of these networks for problems such as semantic segmentation. Over a hundred DL-based segmentation methods proposed until 2021 have had a significant impact on the domain, but we will be highlighting just some of them in this section.

2.4.1 Fully convolutional networks

One of the first DL approaches for semantic image segmentation was Fully Convolutional Networks (FCN) [57], and it inspired some of the most successful state-of-the-art DL techniques.

It contains only convolutional layers, allowing it to take an image of any size and generate a segmentation map of the same size. The model converts all FC layers to convolutional layers and outputs spatial maps rather than classification scores.

Name	Type	Input Size	Output Size	Kernel Size	Stride
data	data	3 ×500 ×500	3 ×500 ×500	-	-
2xconv1	convolution	3 ×500 ×500	64 ×500 ×500	3	-
pool1	max pooling	64 ×500 ×500	64 ×250 ×250	2	2
2xconv2	convolution	128 ×250 ×250	128 ×250 ×250	3	-
pool2	max pooling	128 ×250 ×250	128 ×125 ×125	2	2
3xconv3	convolution	256 ×125 ×125	256 ×125 ×125	3	-
pool3	max pooling	256 ×125 ×125	256 ×63 ×63	2	2
3xconv4	convolution	512 ×63 ×63	512 ×63 ×63	3	-
pool4	max pooling	512 ×63 ×63	512 ×32 ×32	2	2
3xconv5	convolution	512 ×32 ×32	512 ×32 ×32	3	
pool5	max pooling	512 ×32 ×32	512 ×16 ×16	2	2
fc6	convolution	512 ×16 ×16	4096 ×10 ×10	7	-
drop6	dropout (rate 0.5)	4096 ×10 ×10	4096 ×10 ×10	-	-
fc7	convolution	4096 ×10 ×10	4096 ×10 ×10	1	
drop7	dropout (rate 0.5)	4096 ×10 ×10	4096 ×10 ×10	-	-
score	convolution	4096 ×10 ×10	21 ×10 ×10	1	-
score2	deconvolution	21 ×10 ×10	21 ×22 ×22	4	2
score-pool4	convolution	512 ×32 ×32	21 ×32 ×32	1	-
score-pool4c	crop	21 ×32 ×32	21 ×22 ×22	-	-
score-fuse	eltwise	21 ×22 ×22	21 ×22 ×22	-	-
bigscore	deconvolution	21 ×22 ×22	21 ×368 ×368	32	16
upscore	crop	21 ×368 ×368	21 ×500 ×500	-	-
output	softmax	21 ×500 ×500	21 ×500 ×500	-	-

Table 2.7: Proposed architecture network based on FCN from [58]

This research is a pivotal point in image segmentation because it demonstrates that deep networks can be trained for semantic segmentation on variable-sized images from start to finish.

Despite its popularity and effectiveness, the conventional FCN model has some limitations: it is not fast enough for real-time inference, it does not efficiently account for global context information, and it is not easily transferable to 3D images [27].

2.4.2 SegNet

SegNet is another promising work proposed by V. Badrinarayanan et al. [59], a convolutional encoder-decoder architecture for image segmentation, see (figure 2.11).

Its encoder part is topologically identical to the VGG-16 [22] in its 13 convolutional layers. Moreover, the decoder network enhances the location information first with max-pooling indices generated by the corresponding encoder, followed by a pixel-wise classification layer.

The decoder up-samples its lower resolution input feature vector, removing the necessity for up-sampling learning. The up-sampled maps are then convolved with trainable filters to generate dense feature maps.

SegNet also has a significantly lower number of trainable parameters.

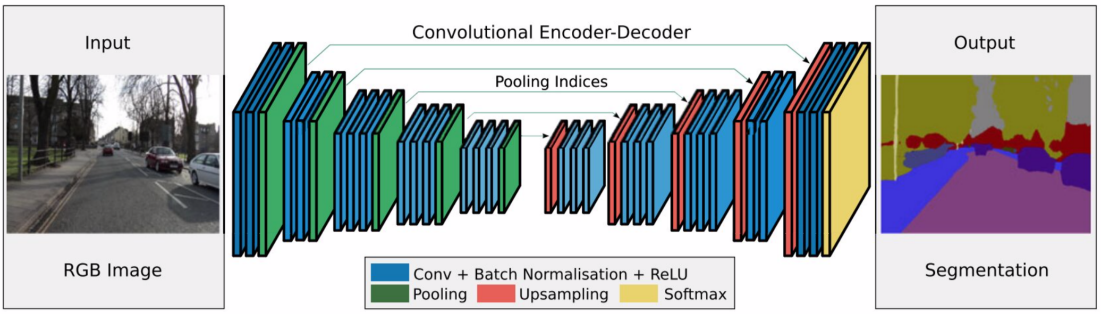


Figure 2.11: SegNet architecture from [59]

2.4.3 DeepLab

DeepLabv1[60] and DeepLabv2[61] were created by Chen et al., which are two of the most widely used image segmentation methods. The latter has three distinguishing characteristics. The first is the use of dilated convolution to deal with the network's declining resolution (caused by max-pooling and striding). The second method is ASPP, which utilizes Atrous convolution with various dilation to capture object multiscale information. The third improvement is enhanced object boundary localization using a combination of deep CNNs and probabilistic graphical models.

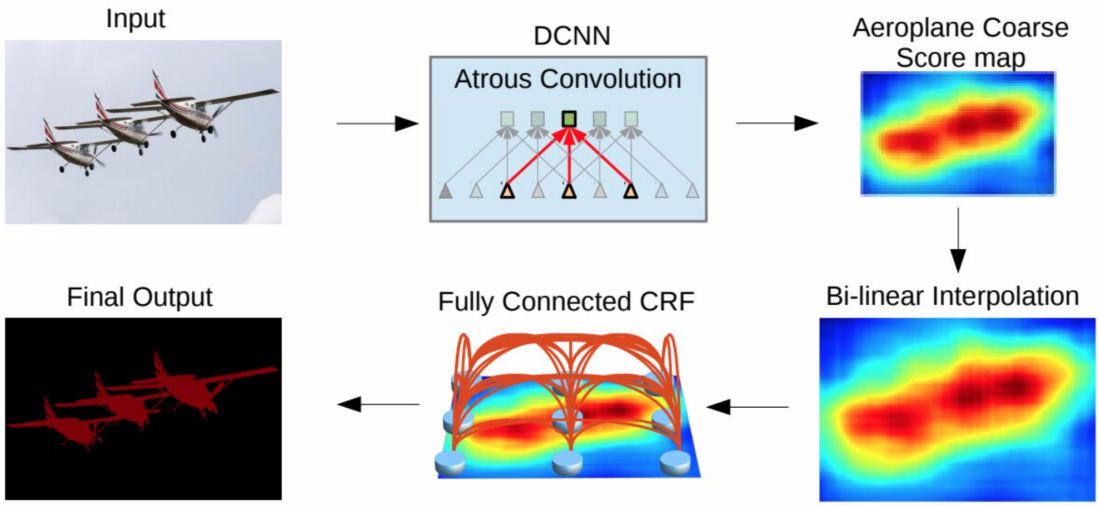


Figure 2.12: The DeepLab model

The researchers then released DeepLabv3 [62], which includes cascaded and parallel dilated convolutions modules. The ASPP is where the parallel convolution modules are gathered; it includes a 1×1 convolution and BN. To construct the final output with logits for each pixel, all outputs are concatenated and processed by another 1×1 convolution.

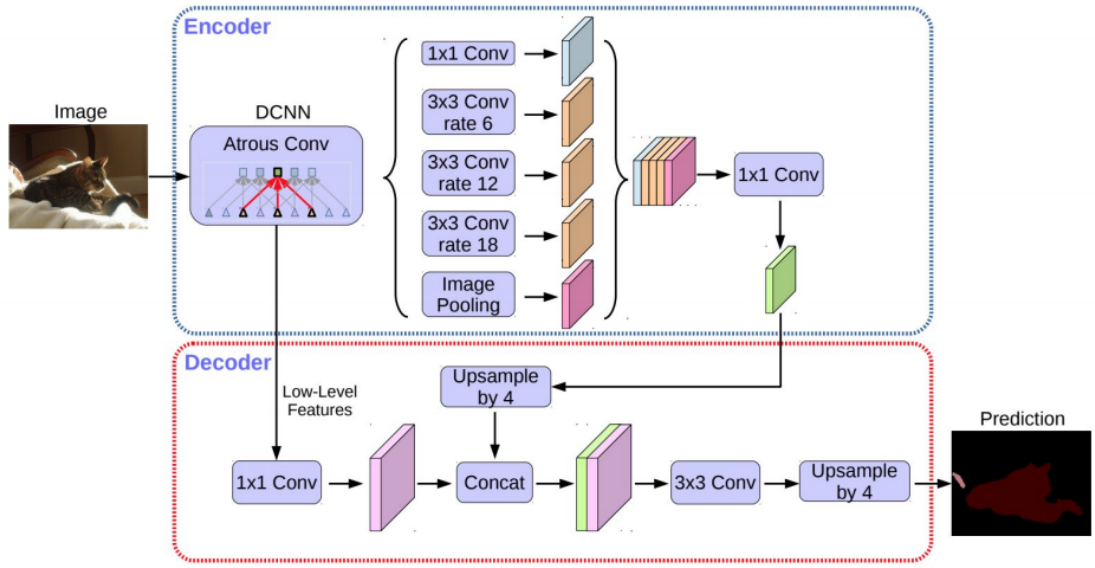


Figure 2.13: The DeepLabv3+ model from [63]

Deeplabv3+ was released in 2018 [63], and it uses an encoder-decoder architecture see (figure 2.13) with atrous separable convolution, which is made up of a depthwise (dw) convolution (spatial convolution for each channel of the input) and a pointwise convolution (1×1 convolution with the depthwise convolution as input). As an encoder, they employed the DeepLabv3 framework. The most relevant model uses a modified Xception [64] backbone with additional layers, dilated depth-wise separable convolutions rather than max pooling, and BN instead of max pooling.

2.4.4 Pyramid scene parsing network

Zhao et al.[65] developed PSPNet, which is based on DeepLab, and it is a multiscale network that attempts to learn the global context representation of a scene.

It makes use of the pyramid pooling module to aggregate image global context data with an auxiliary loss. PSPNet can be applied to VGG and ResNet-based network structures because DeepLab provides two versions of the model adapted from VGG-16 and ResNet-101, respectively.

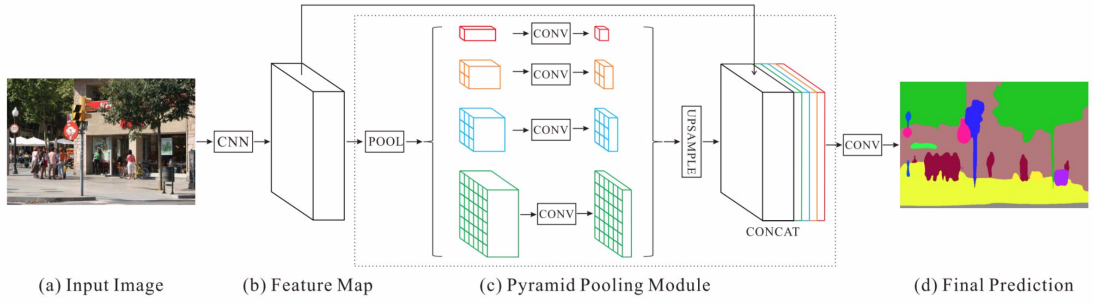


Figure 2.14: PSPNet architecture from [23]

A ResNet and a dilated network are used as feature extractors to derive different patterns from an input image. These feature vectors are then forwarded into a pyramid pooling module, which differentiates patterns of various dimensions.

They are pooled at four scales, each relative to a pyramid level, and reduced by a 1x1 convolutional layer. The pyramid levels outputs are up-sampled and summed with the initial feature maps to grasp local and global context information. Lastly, pixel-wise predictions are generated using a convolutional layer.

2.4.5 U-Net and its variants

2.4.5.1 U-net

U-net, which was initially developed for medical/biomedical image segmentation, is also based on FCN and is inspired by ED architecture [23].

The network and training method relies on data augmentation to learn more effectively from the available labeled data. The U-Net architecture, see (figure 2.15), is made up of a contracting path for context capture and a symmetric expanding path for accurate localization.

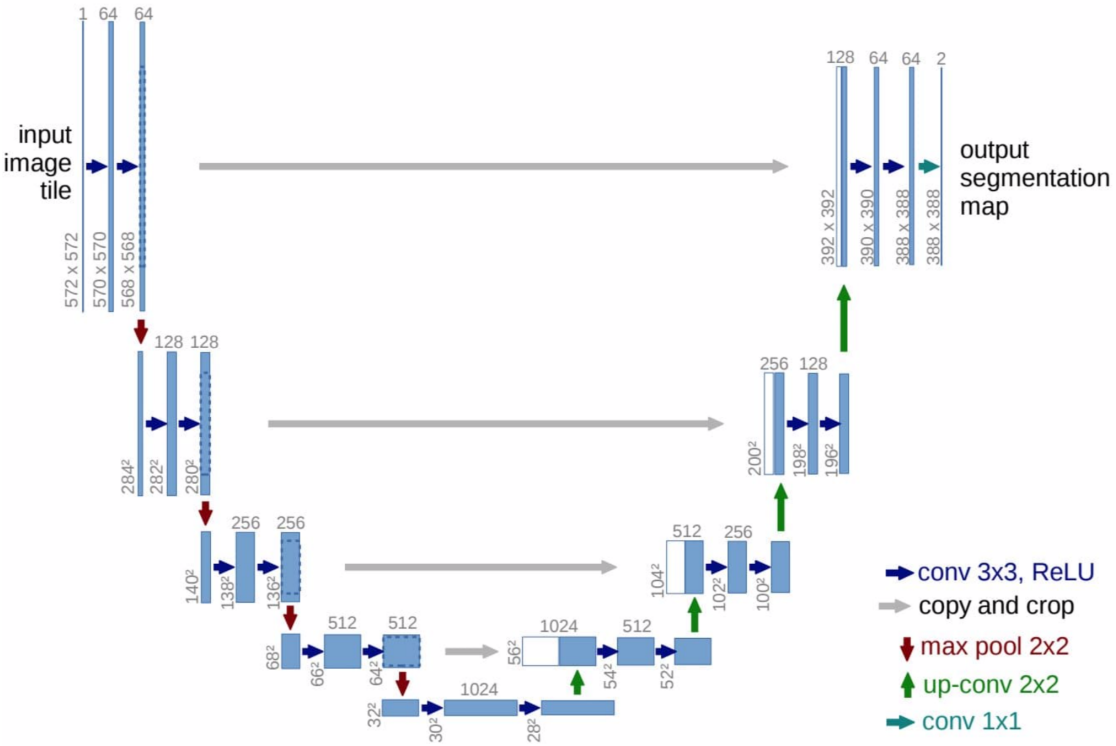


Figure 2.15: U-Net architecture

The down-sampling phase employs an FCN-like architecture to extract features using 3x3 convolutions. Deconvolution is used in the up-sampling phase to reduce the number of feature maps while increasing their dimensions.

Feature maps from the network's down-sampling part are copied to the network's up-sampling part to prevent losing pattern information. Finally, a 1x1 convolution

processes the feature maps to produce a segmentation map that classifies each pixel in the input image.

2.4.5.2 Attention U-Net

The Attention U-Net is a model based on the U-Net architecture see (figure 2.15) and some attention gate blocks see (figure 2.16). In the context of image segmentation, attention is a technique for highlighting only the activations that are relevant during training. This technique aims to save computational resources by reducing the number of activations that are not relevant, giving the network more generalization power. In other words, the network may pay "attention" to certain areas of the image.

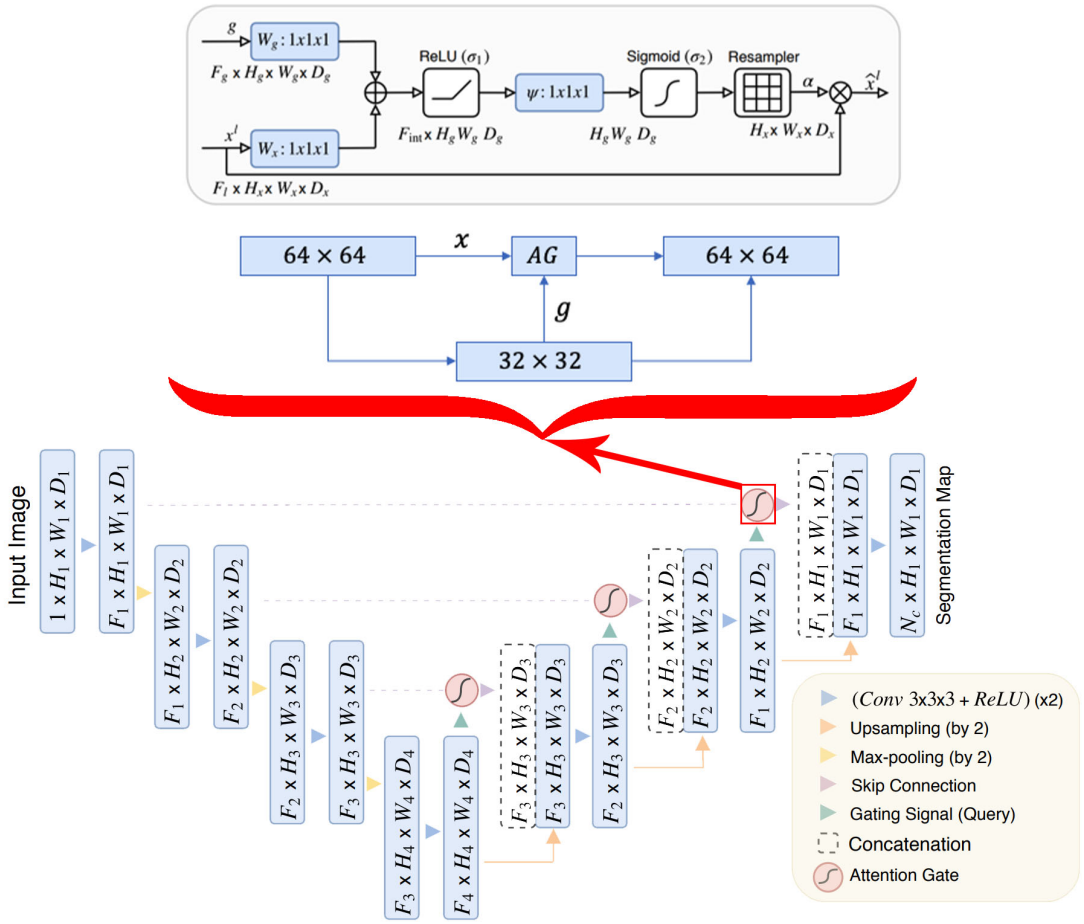


Figure 2.16: Attention U-Net architecture from [66]

The attention gate requires two inputs, x, and g, which are vectors. The vector g is obtained from the network's next lowest tier. Given that it arrives from further into the network, the vector has lower dimensions and better feature representation. Vector x would be 64x64x64 (filters x height x width) in the example (figure 2.16), while vector g would be 32x32x32. Vector x undergoes a stridden convolution, resulting in dimensions of 64x32x32, while vector g undergoes a 1x1 convolution, resulting in 64x32x32. Each of the vectors is added element-by-element. As a result of this process, aligned weights grow in size while unaligned weights reduce

in size. The resulting vector is processed with a ReLU activation layer and a 1x1 convolution, reducing the dimensions to 1x32x32. This vector is scaled between [0,1] by a sigmoid layer, which produces the attention coefficients (weights), with coefficients closer to 1 indicating significant features. Trilinear interpolation is used to up-sample the attention coefficients to the original dimensions (64x64) of the x vector. The original x vector is multiplied element-by-element with the attention coefficients, scaling the vector according to relevance. This is then passed along normally in the skip connection.

2.4.5.3 Attention Residual U-Net

Attention Residual U-Net is another model based on a previous architecture, specifically, Attention U-Net. A regular convolutional block has inputs going into the convolution block, and it may have one or more convolutional operations followed by a ReLU activation and the option to have BN as a part of it. After those blocks, we will have the outputs, and that is the basic model. The residual convolutional blocks are what we can consider a typical convolutional block. However, then the outputs are summed between the convolution block output and also the original input itself, meaning the part in the middle (figure 2.17) when the training is happening, we are training those blocks for the residual part hence the name residual CNNs.

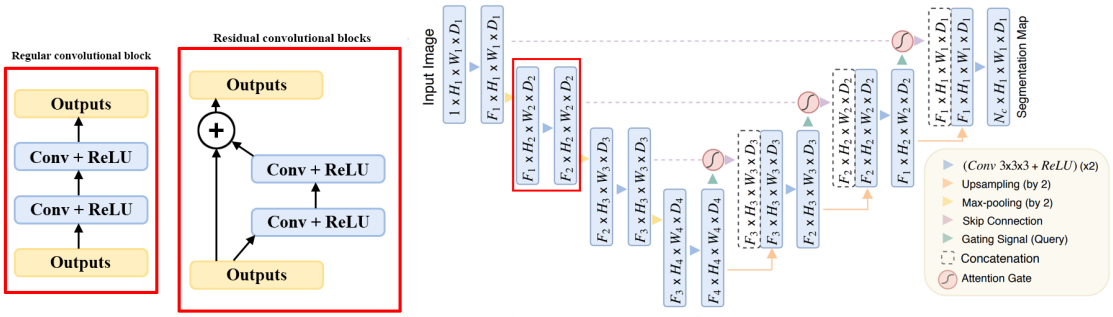


Figure 2.17: Attention Residual U-Net architecture

2.5 Conclusion

In this chapter, we discussed a set of principles related to semantic segmentation. We defined and connected it to DL and also compared it to other tasks. After that, we went over various prominent CNN architectures before introducing a set of DL-based image segmentation methods.

In the following chapter, we will deal with a real-world application for semantic segmentation using U-Net architecture and both its variants we have mentioned earlier, Attention U-Net and Attention Residual U-Net, and we will be experimenting with a medical dataset and taking notes about the different results. We will see the training process, the metrics used, how to incorporate augmented data, and the result of our model.

Chapter 3

Application to semantic segmentation of electron microscopy images

3.1 Introduction

To discover more about what semantic segmentation can offer as a DL method, we combined some methods and models discussed in the previous chapter with other techniques to produce realistic results that allow us to evaluate the technology and compare the methods used.

In this chapter, we start by defining the problem selected. Then we give more details about the dataset used. After that, we explain how we use data augmentation to have more data available for the training and test phases. Subsequently, we talk about the model building blocks, where we explain the construction of the network architectures of the models used and how to train it. Next, we discuss the experiments, tests, and results, starting by explaining the method of conducting the experiments and the environment used, then we show the test results, and then compare those results. Finally, we evaluate our results using the standard IoU metric, suitable for semantic segmentation problems.

This chapter will enable us to see the exclusive results, making it easier to study the effectiveness of the follow-up methods used in semantic segmentation.

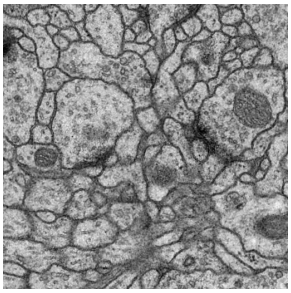
3.2 Problem definition

The assembly of cells forms the so-called membrane, which is found in various organs of the body of living organisms. Each cell of the membrane is made up of essential components that allow it to take a specific shape and perform its vital functions. We will identify a component, which is the cytoplasm, using microscopic images of cross-sections in an organic membrane.

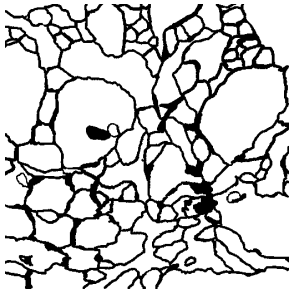
The fundamental issue we are going to address in our experiment and especially in the field of neuroanatomy is how to automate the segmentation of neuronal structures displayed in stacks of EM images, see (Figure 3.1) which represent actual images in the real-world, containing some noise and small image alignment errors. This is required in order to map 3D brain anatomy and connectivity efficiently. To segment biological neuron membranes, we employ a well-known network architecture which is U-Net, and we use the ISBI challenge dataset [67].

3.3 Dataset

The dataset consists of 30 sections from a serial section Transmission ssTEM data set of the ventral nerve cord of a Drosophila first instar larva. The microcube has a 4x4x50 nm/pixel resolution and measures approximately 2 x 2 x 1.5 microns. The associated binary labels (ground truth) are provided in an in-out fashion, i.e., white for segmented object pixels and black for the remaining pixels (which correspond mostly to membranes) [67].



(a) Input image



(b) Corresponding label

Figure 3.1: Example of membrane image and its corresponding segmentation

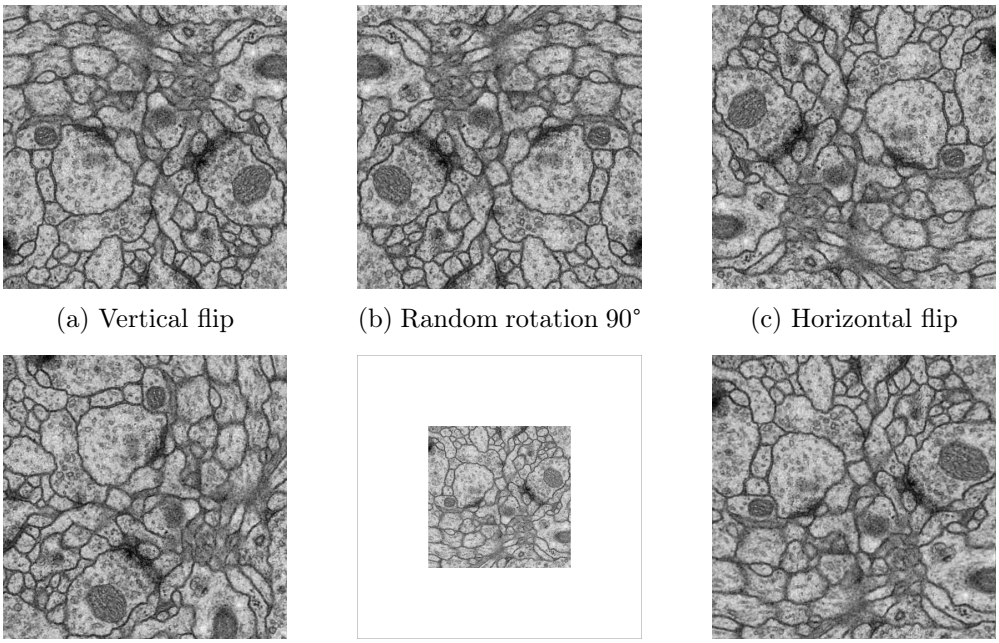
3.4 Data augmentation

In our dataset, we have 30 microscopical images and their masks, and we used a Python library for fast and flexible image augmentations called Albumentations [68]. We have used six spatial-level transforms that, along with modifying the input image, will modify extra targets like masks simultaneously. Those transformers are:

- vertical/horizontal flip that flips the input vertically/horizontally around the x-axis.
- random rotation that randomly rotates the input by 90 degrees zero or more times.

- transposes the input by swapping rows and columns.
- resizes the input to the given height and width.
- and lastly but not least grid distortion, which makes weird distortions to an input image.

An important parameter that controls these transformers is the probability of applying the transform, and it is 0.5 by default.



(a) Vertical flip (b) Random rotation 90° (c) Horizontal flip

(d) Transpose (e) Resize (f) Grid distortion

Figure 3.2: Deformations used in data augmentation for image in figure 3.1 (a)

3.5 Model building blocks

3.5.1 Network architectures

We had shown the network architecture in an earlier figure see (Figure 2.15) when we discussed the U-Net architecture. It is made up of a contracting path (on the left) and an expansive path (on the right). The network's contracting path follows the standard architecture of a CNN. It comprises two 3x3 convolutions (unpadded convolutions) repeatedly applied, each followed by batch normalization, a ReLU, and a 2x2 max pooling layer with a stride of 2 for downsampling. We quadruple the number of feature channels with each downsampling step. Every step in the expansive path starts with an upsampling of the feature map, then a 2x2 convolution (up-convolution) to half the number of feature channels, a concatenation with the equally cropped feature map from the contracting path, and two 3x3 convolutions, each followed by batch normalization, a ReLU. Because every convolution loses border pixels, cropping is required. A 1x1 convolution is employed in the final layer to convert each 64-component feature vector to the desired number of classes. There are 23 convolutional layers in total in this network. The overall number of

parameters used in this architecture was 31,402,501.

The other two models used in our application were the variants of U-Net which are Attention U-Net (Figure 2.16) and Attention Residual U-Net (Figure 2.17). The first model is perfectly similar to U-net except for the addition of the attention block as mentioned in the earlier definition. The overall number of parameters used in this one was 37,334,665. For the second model, we have completely changed the convolutions with residual convolution blocks beside an attention block, and the overall number of parameters used in this architecture was exactly 39,090,377.

Because unpadded convolution is employed, the output size is less than the input size. The overlap-tile approach is utilized instead of downsizing before the network and upsampling after the network. As a result, the entire image is predicted piece by piece, as shown in the figure below see (Figure 3.3). The blue area in the image is used to predict the yellow area. Mirroring is used to extend the image at the image boundary.

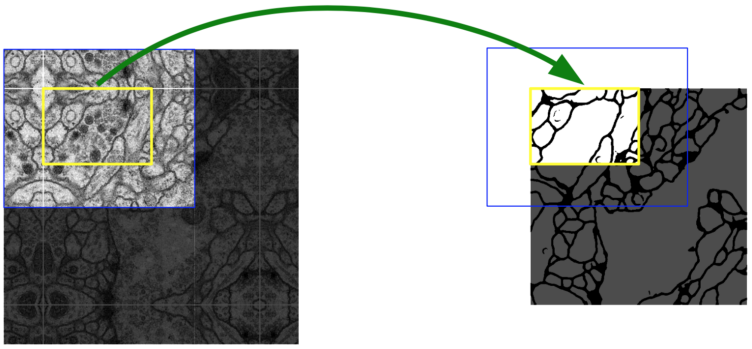


Figure 3.3: The overlap-tile approach for seamless segmentation of arbitrary large images from [23]

3.5.2 Training

To train our models, we use **90%** of all of the training stack's available slices, which is 540 images of the membrane with **256x256** resolution, and **10%** we use it for the validation process. Our models were compiled with Adam optimizer, and we use binary focal loss function since there are only two classes; white for the pixels of segmented objects and black for the rest of pixels (which correspond mostly to membranes). We use a batch size of **8**, with **50** epochs (which we can control as we like) and the default number of steps per epoch which is 68 calculated by dividing the number of training images by the batch size ($540/8 = 67.5 \approx 68$), and it was trained using a learning rate of 10^{-2}.

We save the models as a **hdf5** file, and the history of the training containing our metrics (Accuracy, Loss, and Jaccard coefficient) as a **npy** file after the completion of the training, so we can use them any time we want without re-training the model from scratch. We also compute a standard performance measure, which is the mean IoU. We trained the models on a Google Colab GPU (NVIDIA ® Tesla ® K80), and we made a table to note the execution time for all the models in two cases (per epoch and the overall time):

	1 epoch	50 epochs
U-Net	84s	1h 11min 27s
Attention U-Net	94s	1h 19min 30s
Attention Residual U-Net	119s	1h 39min 00s

Table 3.1: Results of execution time for the three models

3.6 Experiments, tests and results

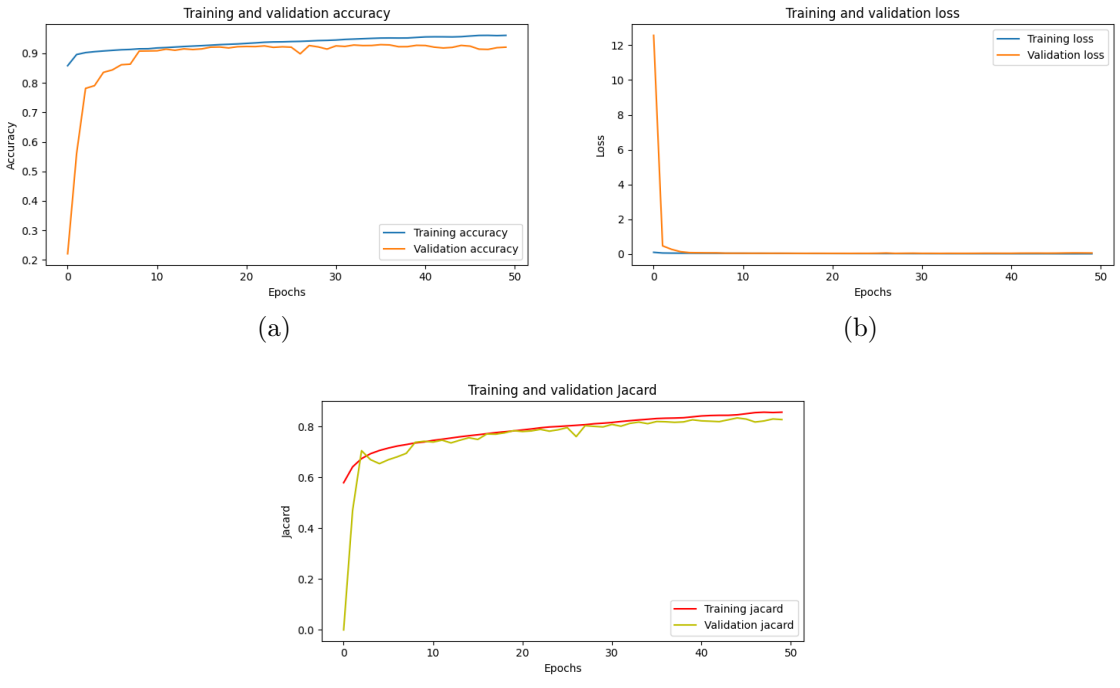
3.6.1 Experiments

In the process of training our three models, we have fixed the training hyperparameters so that we can put all the models in the same training environment and evaluate their performance metrics such as Accuracy, Loss, and Jaccard coefficient on both the training set and the validation (or test) set in the following table:

	loss	val_loss	accuracy	val_accuracy	jacard_coef	val_jacard_coef
U-Net	0.0239	0.0641	0.9603	0.9161	0.8540	0.8182
Attention U-Net	0.0296	**0.0480**	0.9502	**0.9267**	0.8325	0.8148
Attention Residual U-Net	**0.0228**	0.0636	**0.9620**	0.9227	**0.8592**	**0.8342**

Table 3.2: Results of the performance metrics for each model

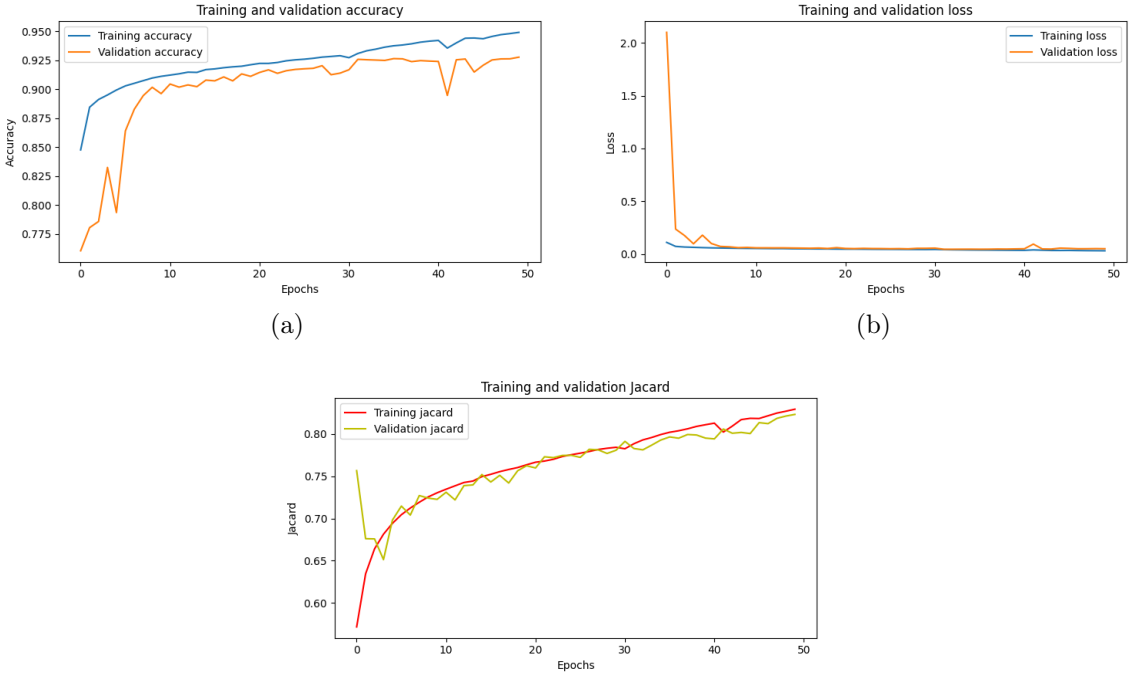
In (Figures 3.4, 3.5 and, 3.6) we display the evolution of the values of the performance metrics in every epoch of the training and validation process in our models.



(a) (b)

(c)

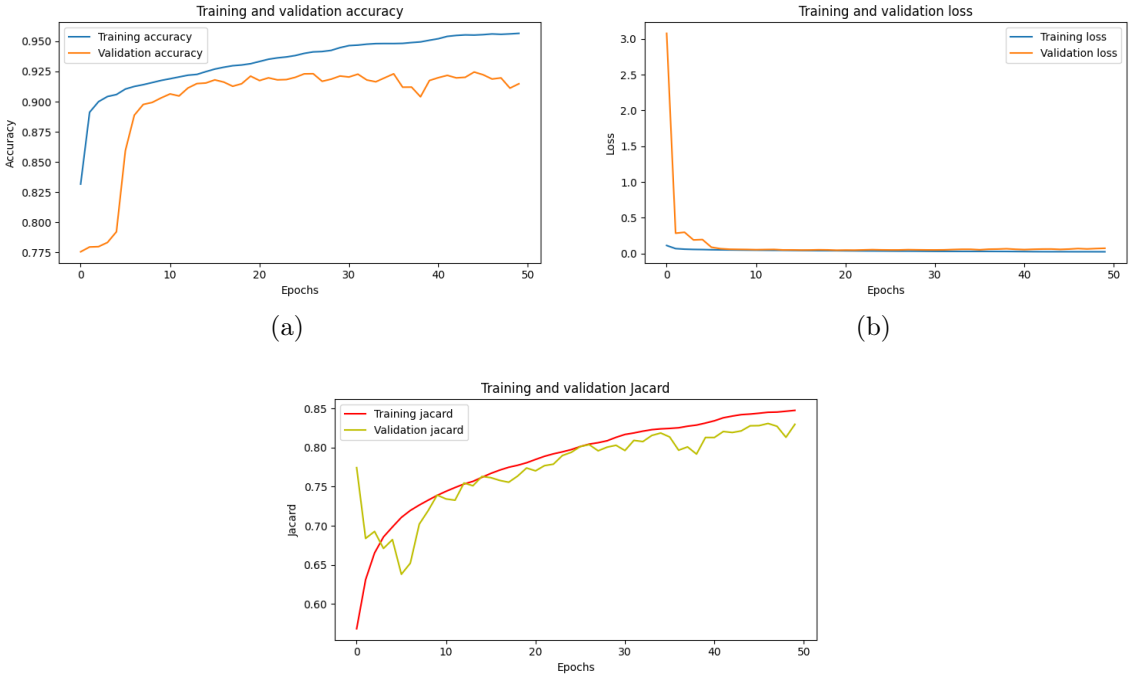
Figure 3.4: Evolution of the values of the performance metric in U-Net



(a) (b)

(c)

Figure 3.5: Evolution of the values of the performance metric in Attention U-Net



(a) (b)

(c)

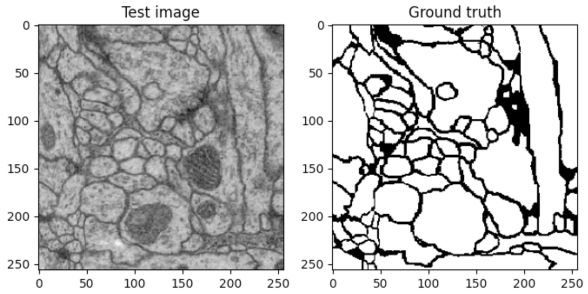
Figure 3.6: Evolution of the values of the performance metric in Attention Residual U-Net

When we look at the Loss, the three models almost overlap with the slightest change in Attention Residual U-Net at the end, making the best value at 0.0228 of Loss for the training set, but when it comes to the validation set, Attention U-Net had the best value at 0.0480. For Accuracy, Attention Residual U-Net had the best

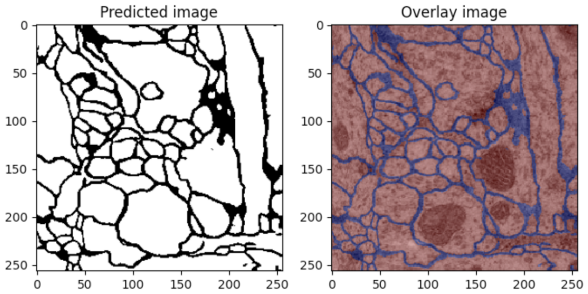
value at the end at 0.9620 followed by U-Net at 0.9603, and Attention U-Net at 0.9502 for the training set. For the validation set, Attention U-Net and Attention Residual U-Net were almost identical with a slight difference. For the Jaccard coefficient, Attention Residual U-Net had the best value in both the training and the validation set, but it was a minor advantage over the other two models. We cannot conclude that a model is better than the others because the results were too close, but some slight changes would help uncover more details of the segmentation when we do the testing, and we might see them in the results.

3.6.2 Tests and results

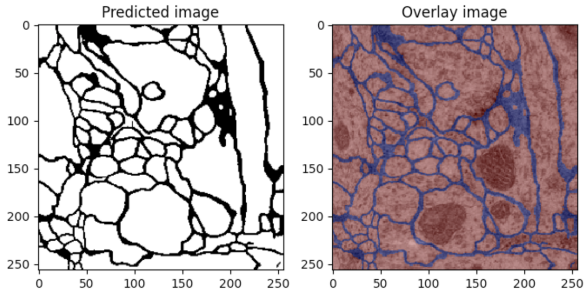
In the next step, we will compute and predict labels of three given test images and their ground truth using the three models, the results will be shown in the following figures:



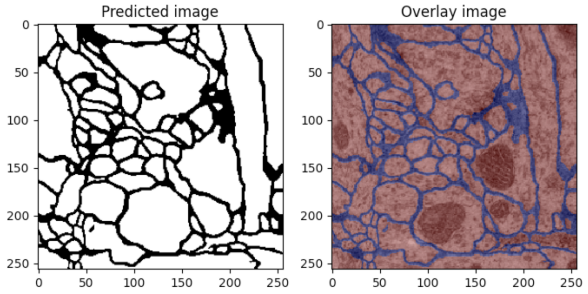
(a) Test image & Ground truth



(b) U-Net prediction

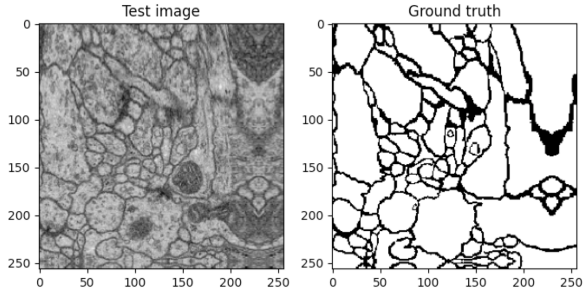


(c) Attention U-Net prediction

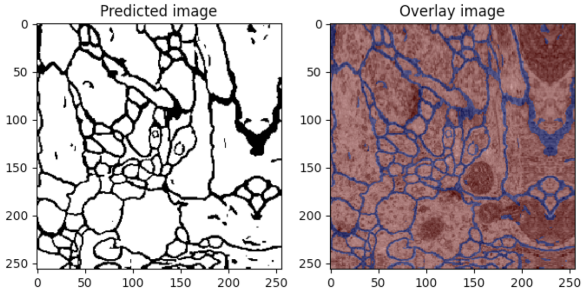


(d) Attention Residual U-Net prediction

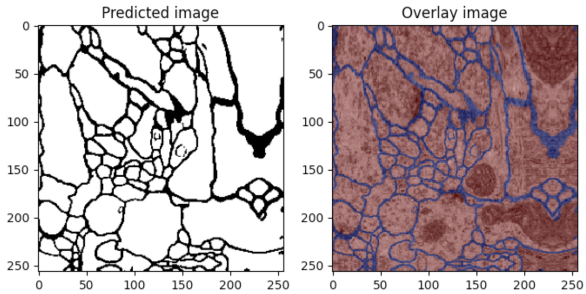
Figure 3.7: Results of prediction on test image 1



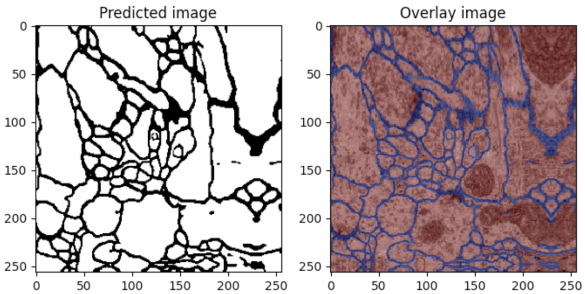
(a) Test image & Ground truth



(b) U-Net prediction

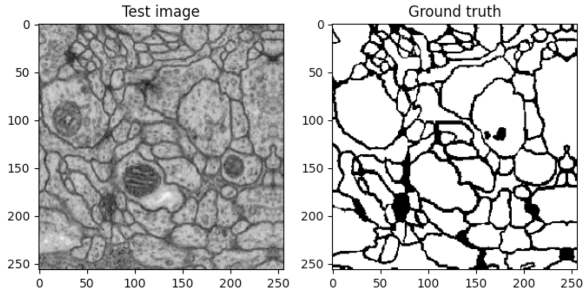


(c) Attention U-Net prediction

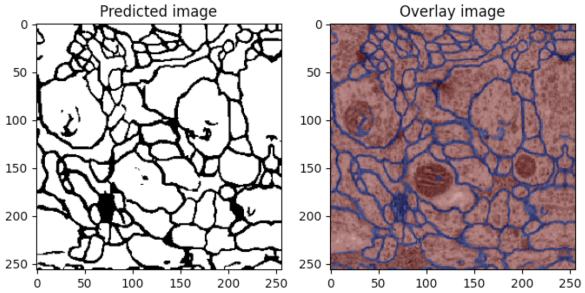


(d) Attention Residual U-Net prediction

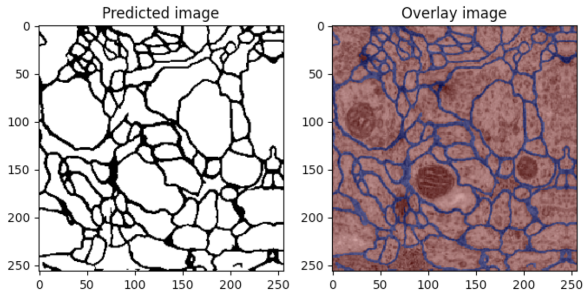
Figure 3.8: Results of prediction on test image 2



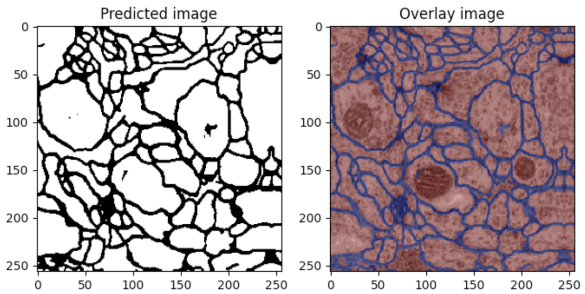
(a) Test image & Ground truth



(b) U-Net prediction



(c) Attention U-Net prediction



(d) Attention Residual U-Net prediction

Figure 3.9: Results of prediction on test image 3

In (Figures 3.7, 3.8 and 3.9) we give an illustration of the segmentations generated using the three networks. We notice in (Figure 3.7) some minor differences between

the results, especially between the U-Net (b) and Attention Residual U-Net (d) predictions, with the better result using Attention U-Net (c). In the next (Figure 3.8), we notice similar results with the earlier ones with clearly better performance from the Attention U-Net (c) like always, followed by Attention Residual U-Net (d) then U-Net (b). The last (Figure 3.9) Attention U-Net (c) continues with better results than both of the other models but with a minor advantage over them.

3.6.3 Evaluation

In this section, we are going to evaluate our three models using the IoU of an individual image and the mean IoU of all test images:

Mean IoU	**Image 1**	**Image 2**	**Image 3**	**All images**
U-Net	0.823767	0.80377704	0.758395	0.79322557
Attention U-Net	**0.8376042**	**0.838861**	**0.7944911**	**0.8146624**
Attention Residual U-Net	0.8273399	0.8107518	0.7664076	0.79647577

Table 3.3: Results of Mean IoU for test images

As shown in (Table 3.3), we utilize three examples of test images and then all images to get the mean IoU of every model. The table shows that the Attention U-Net model outperforms other models in terms of the IoU of each of the three images and in terms of the mean IoU of all images. However, there is a difference in superiority between U-Net and Attention Residual U-Net. This latter is better in two images, although U-Net is superior in the mean IoU.

3.7 Software and tools

This section will provide the definitions of the languages, software, and tools we have used to develop our application.

3.7.1 Python programming language

Python is a dynamically semantic high-level programming language that is interpreted and object-oriented. Combined with dynamic type and dynamic binding, its high-level built-in data structures make it ideal for Faster Development and for usage as a scripting or glue language to bring existing components together. Python's straightforward, easy-to-learn syntax prioritizes readability, lowering the cost of program maintenance. Python has support for modules and packages, which promotes program modularity and code reuse. Python's interpreter and substantial standard library are freely accessible in source or binary form for all major platforms. It can be open to public distribution [69].

Figure 3.10: Python logo

3.7.2 PyCharm IDE

PyCharm is a specialized python IDE that offers a wide range of necessary python developer tools that are deeply intertwined to offer a pleasant environment for productive python, data science development, and the web [70].

Figure 3.11: PyCharm logo

3.7.3 Google Colaboratory

Google research's Collaboratory, or **Colab** for short, is a product. Colab is a web-based Python editor that allows anybody to create and run arbitrary Python code. It is notably helpful for machine learning, data analysis, and teaching. Colab is a hosted Jupyter notebook service that does not require any setup and offers free access to computational resources, including GPUs, for free [71].

Figure 3.12: Google Colaboratory logo

3.7.4 PySimpleGUI

PySimpleGUI is a python library that allows Python programmers of all abilities to construct GUIs. PySimpleGUI defines the GUI window using a **layout** that consists of widgets (Elements). Using the layout, one of the four supporting frameworks creates a window that may be displayed and interacted with. Frameworks that are supported include Tkinter, Qt, WxPython, and Remi. These packages are commonly referred to as "wrappers" [72].

Figure 3.13: PysimpleGUI logo

3.7.5 NumPy

NumPy is a python package that is essential for numerical computation. It offers a multidimensional array object, derivative objects (including masked arrays and matrices), and a variety of routines for quick array operations such as mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation, and more [73].

Figure 3.14: NumPy logo

3.7.6 Matplotlib

Matplotlib is a python 2D plotting framework that generates publication-quality figures in a range of hard copy and interactive formats across several platforms. It is compatible with python scripts, the python and IPython shells, the Jupyter notebook, web application servers, and four graphical user interface toolkits. Among the visualizations that can be created using matplotlib are bar graph and pie chart, box plot and histogram plots, scatter plot, as well as figures cite77.

Figure 3.15: Matplotlib logo

3.7.7 Pandas

Pandas is a popular open-source python library for data science, data analysis, and machine learning activities. It is based on the NumPy library, which supports multidimensional arrays. Pandas, as one of the most popular data-wrangling packages, integrates well with many other data science modules within the python ecosystem and is typically included in every python distribution [74].

Figure 3.16: Pandas logo

3.7.8 Scikit-learn

Scikit-learn is an open-source python machine learning library. It is regarded as a straightforward and effective technique for analyzing predictive data. It is based on the NumPy, SciPy, and matplotlib libraries. This library can be used in a variety of

uses, such as classification, regression, clustering, dimensionality reduction, model selection, and preprocessing [75].

Figure 3.17: Scikit-learn logo

3.8 Application's implementation

Our experiments, tests, and evaluations were done using a well-structured GUI application which we will describe in detail in this section.

3.8.1 Main window

The **main** window is the first interface that users will have to face, which allows us to launch every task in the application. It is composed of several buttons on the left and a layout to choose and display dataset images on the right see (figure 3.18).

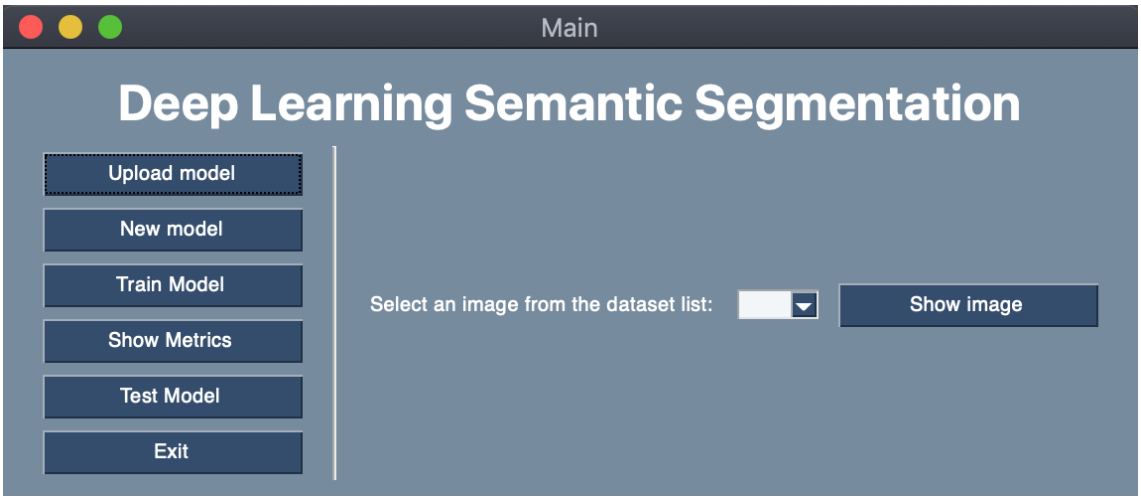


Figure 3.18: Main window interface

The images in the dataset can be seen by selecting an image number from the combo box and clicking on the button **show image**, see (figure 3.19).

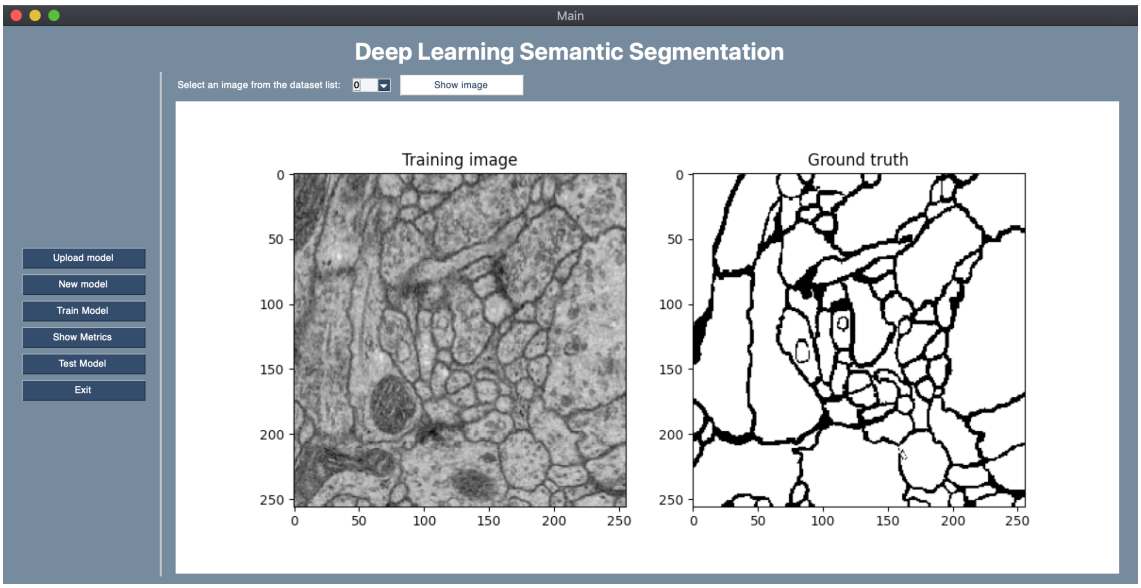


Figure 3.19: Main window interface with images

- The **Upload model** button opens up a window that allows us to upload a pre-trained model.
- The **New model** button opens up a window that allows us to choose a new model and build it.
- The **Train model** button opens up a window that allows us to train the chosen model.
- The **Show metrics** button opens up a window that allows us to check the different metrics used.
- The **Test model** button opens up a window that allows us to test our model on a test dataset.
- The **Exit** button is used to terminate the program.

3.8.2 Upload model window

The **Upload model** interface allows us to upload a pre-trained model from our device and use it in our application, see (figure 3.20).

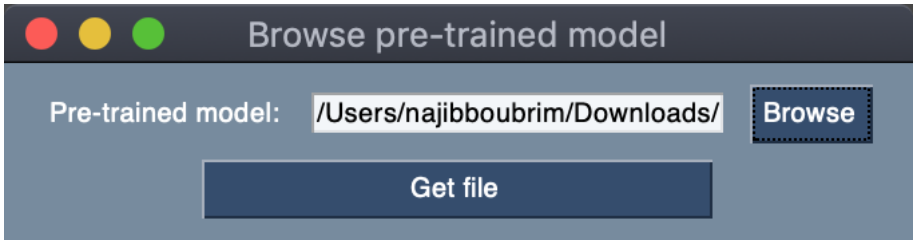


Figure 3.20: Browse model interface

The text field used to enter the file path. The **Browse** button allows us to search for a file in our device using its path, see (figure 3.21).

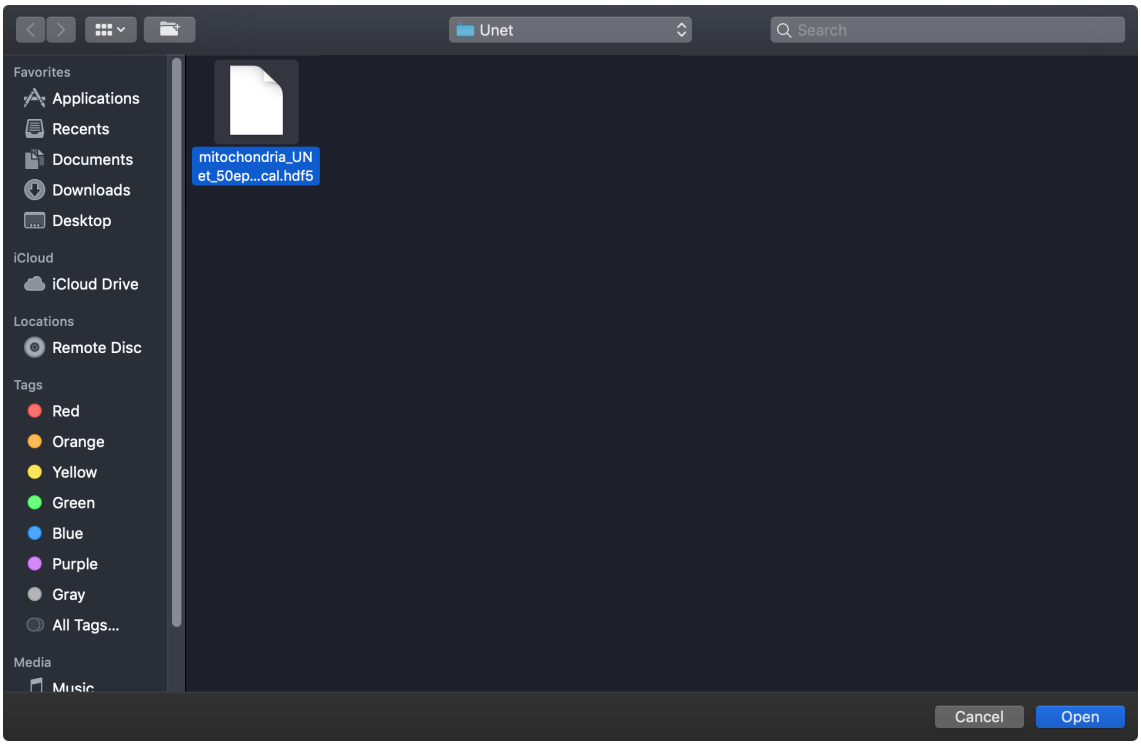


Figure 3.21: File searching interface

The **Get file** button allows us to collect the file path from the input field and save it in a program variable.

3.8.3 New model window

The **New Model** interface allows us to choose a model from the three models we have and build it, see (figure 3.22).

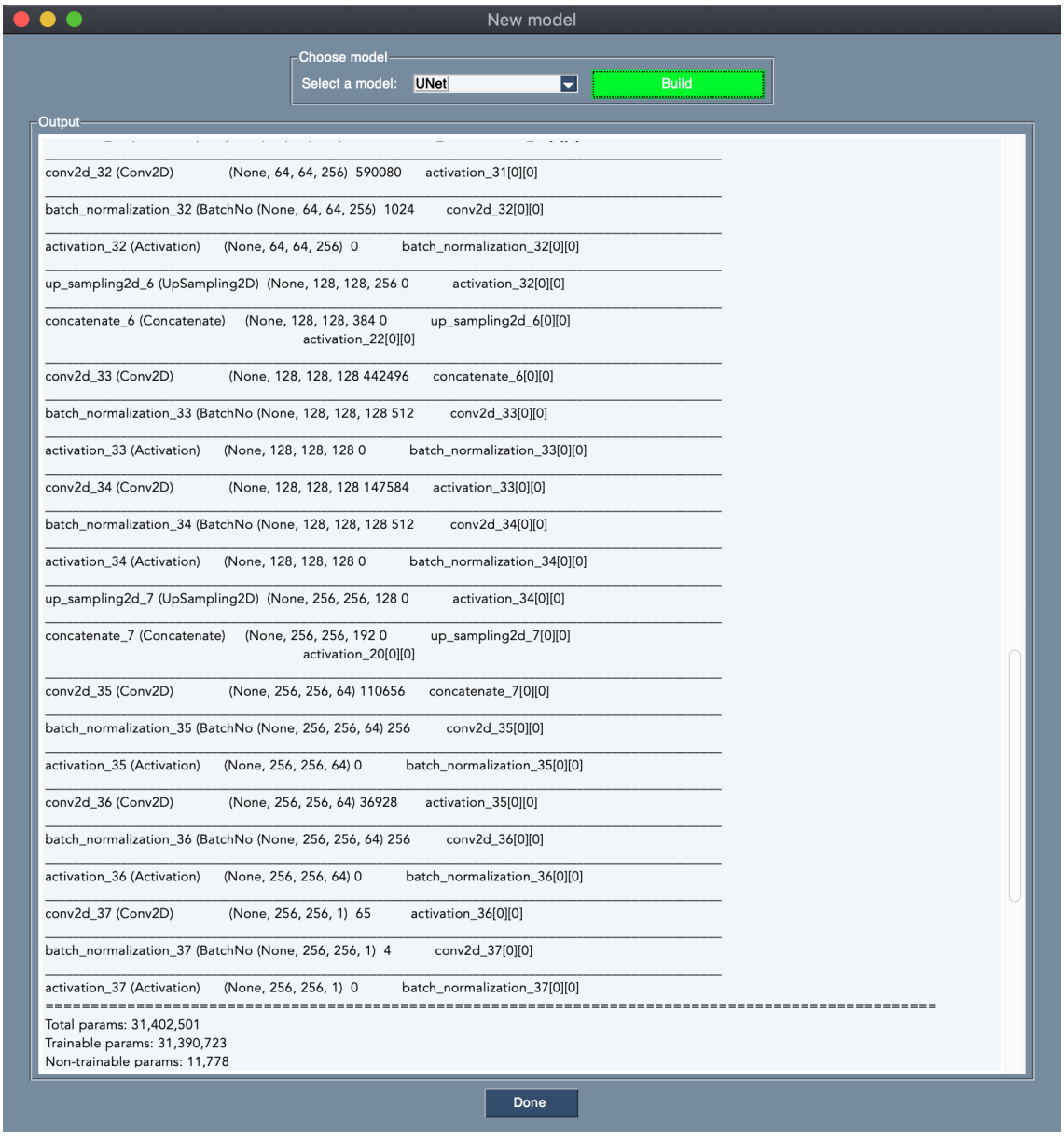


Figure 3.22: New model interface

The combo box contains models that can be built into this program. The **Build** button allows us to build the chosen model. The output frame is used to show the summary of the chosen model after it has been built (compiled).

3.8.4 Training window

The **Training** interface is used to start the training of the chosen model, see (figure 3.23).

Figure 3.23: Training interface

The text field is used to specify the number of epochs that are going to be used to train the model. The **Start training** button is used to start training. The output frame allows us to follow the steps and epochs of the training and to monitor everything from time of execution to the different metrics used. The **Done** button is used when the training is finished to close the window.

3.8.5 Metrics window

The **Metrics** interface contains a list of buttons and a field to show different images. The **Accuracy** button is used to show the image that contains the training accuracy, and validation accuracy curves see (figure 3.24).

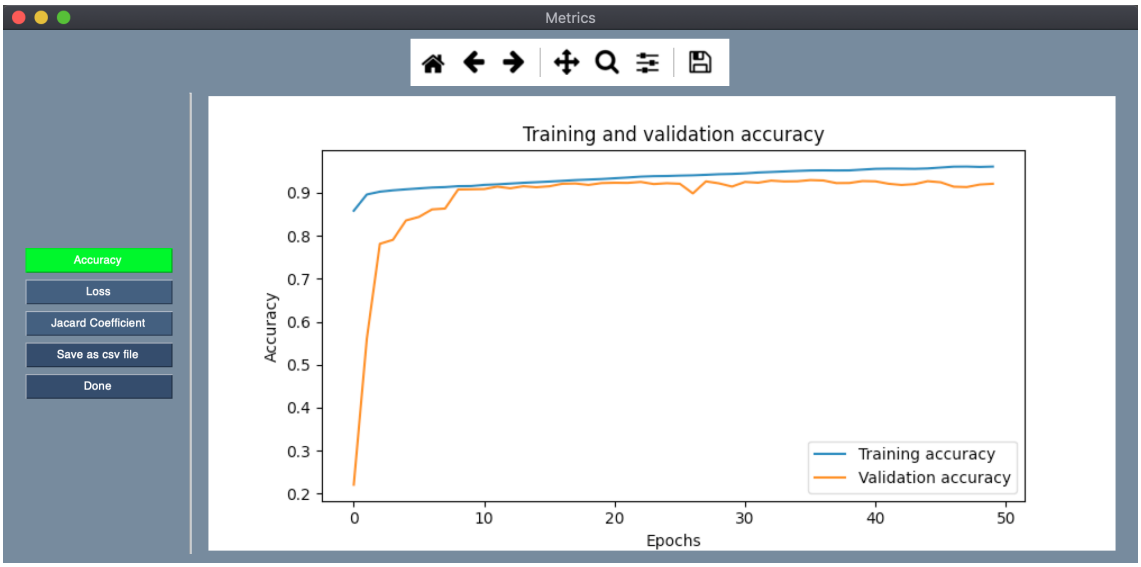


Figure 3.24: Accuracy interface

The **Loss** button is used to show the image that contains the training loss and validation loss curves see (figure 3.25).

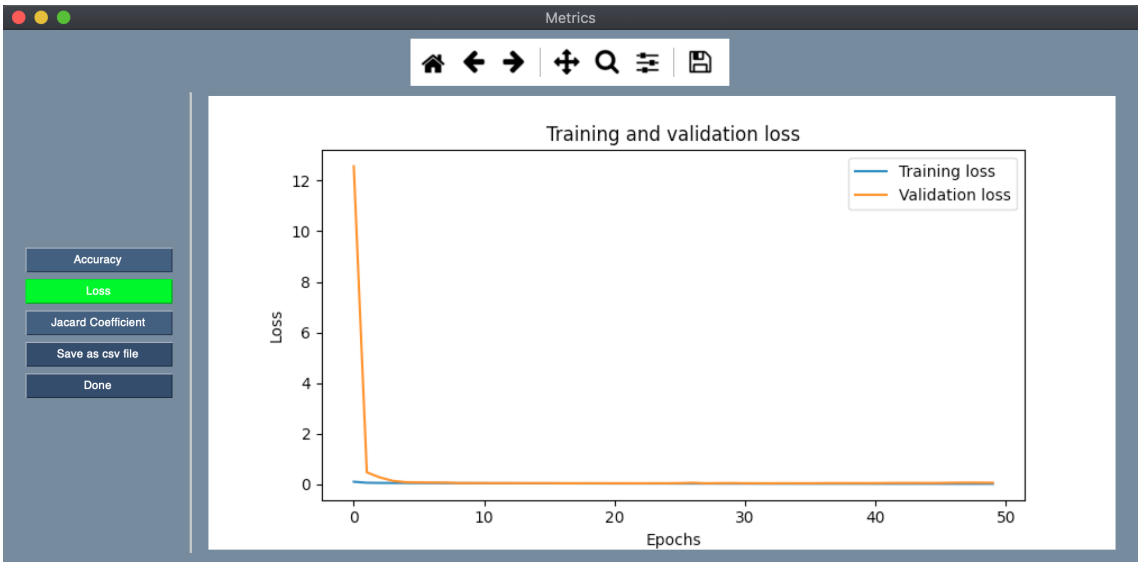


Figure 3.25: Loss interface

The **Jacard Coefficient** button is used to show the image that contains the training Jaccard and validation Jaccard curves see (figure 3.26).

Figure 3.26: Jacard Coefficient interface

The **Save as csv file** button allows us to save the output data from the earlier metrics which they were generated during the training phase in a csv file. The **Done** button is used to close the window.

3.8.6 Test window

The **Test** interface contains several buttons on the right and two different fields to show test images and the prediction, see (figure 3.27, and figure 3.28).

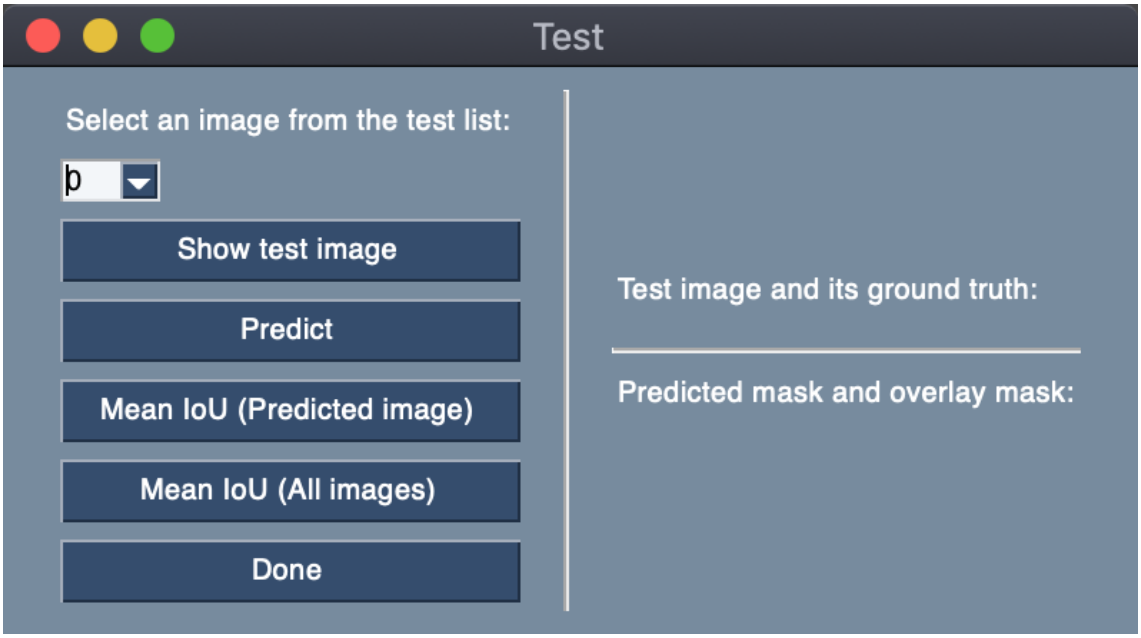


Figure 3.27: Test interface

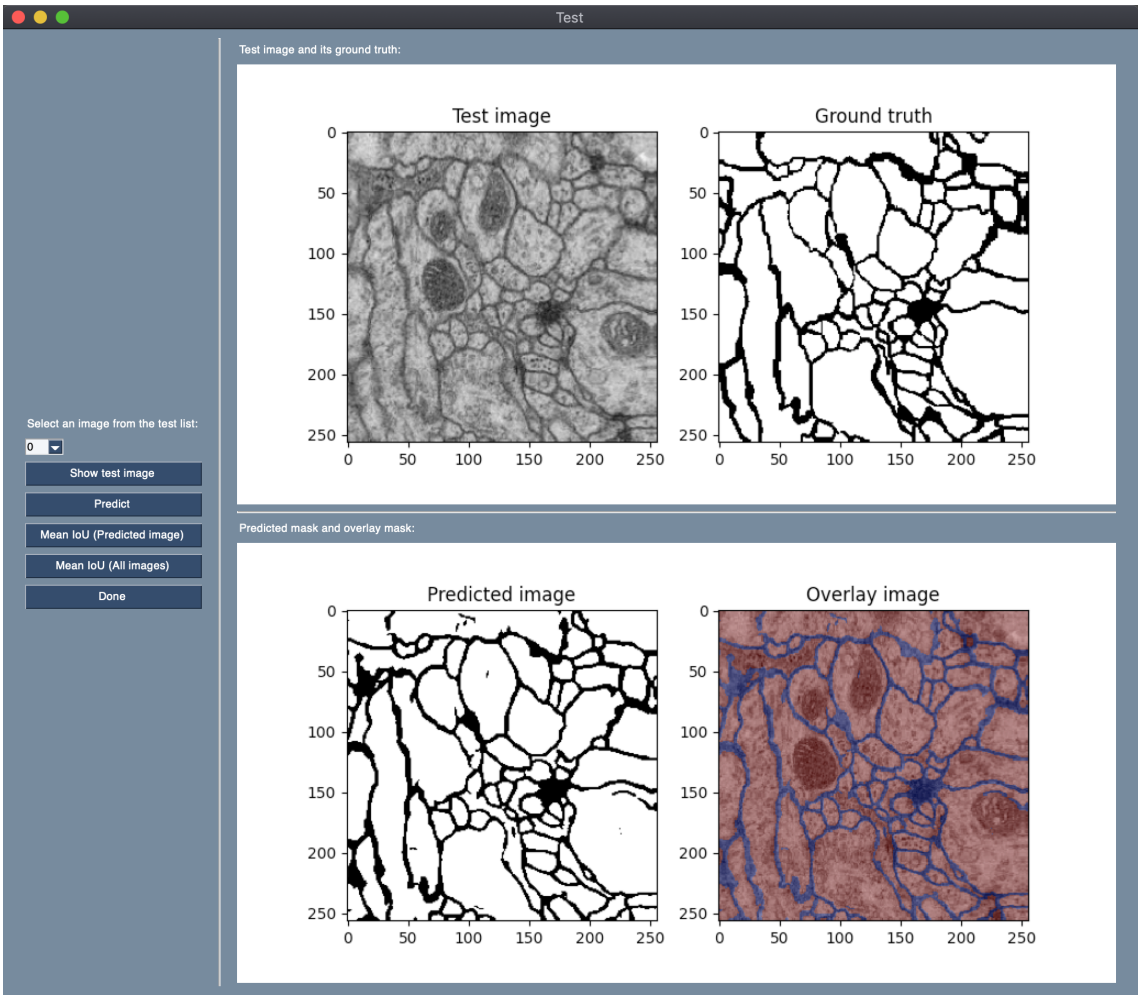


Figure 3.28: Test interface with results

The combo box is used to select the number of the image to be displayed. The **Show test image** button is used to show the image that contains the selected image from the test dataset and its ground truth in the upper image field. The **Predict** button allows us to start the prediction process using the selected image and show the result with an overlay image also in the bottom image field. The **Mean IoU (Predicted image)** button is used to show the mean IoU value of the predicted image in a popup window. The **Mean IoU (All images)** button is used to show the mean IoU values of every image available in the test dataset. The **Done** button is used to close the window.

3.8.6.1 Mean IoU (one image)

The **Mean IoU (Predicted image)** interface is used to show the IoU of the predicted image, see (figure 3.29).

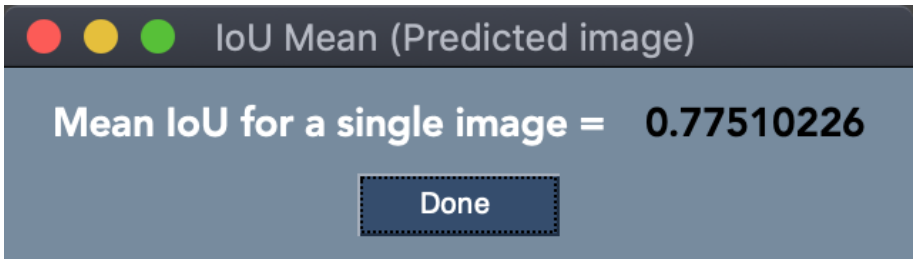


Figure 3.29: Mean IoU (Predicted image) interface

The text field is used to show the mean IoU value. The **Done** button is used to close the window.

3.8.6.2 Mean IoU (all images)

The **Mean IoU (All image)** interface is used to show the mean IoU values of every image available in the test dataset and the overall result see (figure 3.30). The **Done** button is used to close the window.

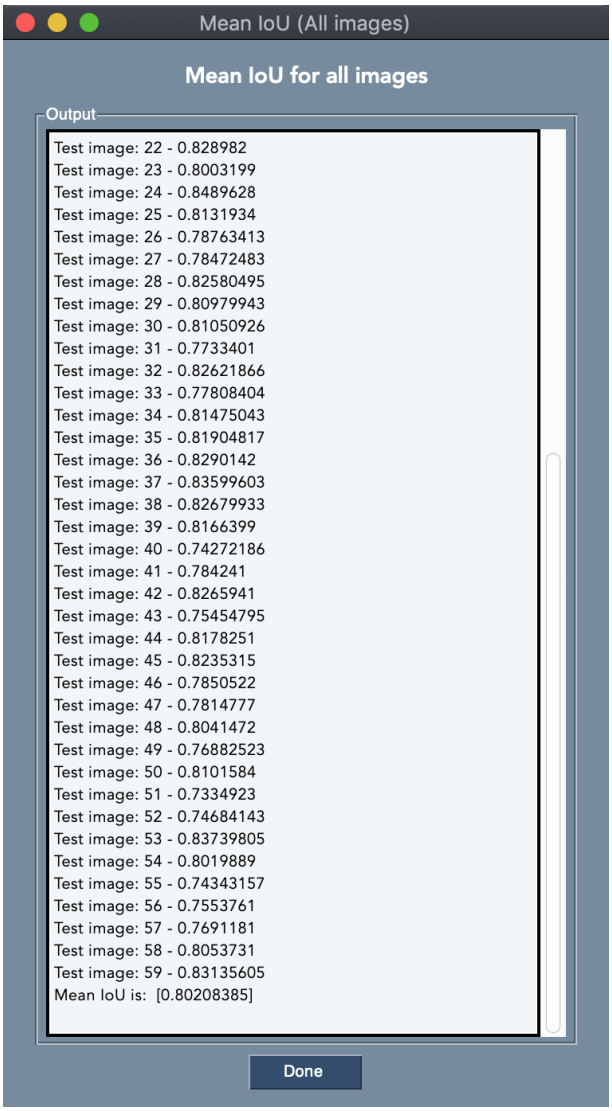


Figure 3.30: Mean IoU (all images) interface

3.9 Conclusion

In this chapter, we have applied the methods and techniques we have learned in the second chapter about U-Net and its variants to the semantic segmentation of EM images. We have gone through the steps to build the models and tune the hyperparameters, then we have trained all of them using the available data after augmentation because of the lack of a big dataset. Next, we presented the experiments, tests, and results we got and evaluated everything using the proper metrics. Finally, we defined the software, tools, and language used to make our application, in addition to the full explanation of all the graphical user interfaces we have made. In general, our chosen models produced excellent results in semantic image segmentation after they were trained for several hours, and there was a slight difference in the results obtained. However, Attention U-Net outperformed U-Net and Attention Residual U-Net overall.

General conclusion

This dissertation was interested in deep learning in general and neural networks used for semantic segmentation in particular. Our primary objective was to create several networks (referred to as models) that would function best with an image segmentation problem in the field of microscopical imagery.

The dissertation was divided into three different chapters for structure. The first chapter dealt with definitions and general concepts about deep learning, the second was a deep dive into the theoretical notions for semantic segmentation and deep learning approaches, and the third was the experimental part where we have had our tests and results.

The first chapter was simply a compilation of fundamental concepts and primary information necessary to speed up this domain. Then the basic deep learning architectures, where we got to talk about them and how we could utilize them in different scenarios. We have also mentioned the most used deep learning frameworks and the advantages and inconveniences for each one. Lastly, we went over the best hardware used for the process of deep learning.

In the second chapter, we got into the essential part of our dissertation, which is the semantic segmentation, where we spoke about its general concepts from the definition, comparison with other computer vision tasks to the methods and techniques used in that field, especially using deep learning methods. Next, we have gone through some of the most known CNN architectures. We have finished talking about the deep-learning-based image segmentation methods, and finally, the most important one that we used in our application is U-Net and its variants.

The last chapter was about displaying the results of our application tests, which combined three models U-Net, Attention U-Net, and Attention Residual U-Net, in one place. We have chosen a medical problem related to the segmentation of electron microscopy images. We have augmented our dataset to get more images to work with, and we have built all three models and trained them all with the same hyperparameters to compare them. The results proved that our models were very efficient and adaptable in image segmentation problems, the results were close in terms of performance metrics, but the Attention U-Net model showed a slight superiority in some areas.

Without question, deep learning has considerably aided image segmentation, but there are still numerous difficulties ahead. Future works will improve the current models and add new elements to them to make them work more efficiently. We may

use more difficult datasets with several objects and overlapping objects or utilize 3D image datasets. We can use newer models like transformer such as TransUNET [76], or Swin-UNET [77] that may be used as strong encoders for medical image segmentation challenges, with the addition of U-Net to improve finer details, and they can outperform other approaches in a variety of medical applications, such as multi-organ segmentation and cardiac segmentation.

Bibliography

[1] LeCun Y., Bengio Y., and G. Hinton. "Deep learning". In: *Nature* vol 521 (2015), pp. 436–444. DOI: https://doi.org/10.1038/nature14539.

[2] Grosan C. and Abraham A. "Artificial Neural Networks". In: *Intelligent Systems: A Modern Approach* vol 17 (2011). Ed. by Springer Berlin Heidelberg, pp. 281–323. DOI: https://doi.org/10.1007/978-3-642-21004-4_12.

[3] S. I. Gallant. "Perceptron-based learning algorithms". In: *IEEE Transactions on Neural Networks* vol 1 (1990). Ed. by Springer Berlin Heidelberg, pp. 179–191. DOI: https://doi:10.1109/72.80230.

[4] M. Goyal et al. "Activation Functions". In: *Deep Learning: Algorithms and Applications, W. Pedrycz and S.-M. Chen* (2020). Ed. by Cham: Springer International Publishing, pp. 1–30. DOI: https://doi:10.1007/978-3-030-31760-7_1.

[5] N. C. Steele, C. R. Reeves, and E. I. Gaura. "Activation Functions". In: *Artificial Neural Nets and Genetic Algorithms* (2001). Ed. by Kůrková V. et al., pp. 27–30. DOI: https://doi.org/10.1007/978-3-7091-6230-9_5.

[6] V. Nair and G.E. Hinton. *Rectified linear units improve restricted boltzmann machines*. ICML'10. Omnipress, Haifa, Israel: Proceedings of the 27th International Conference on International Conference on Machine Learning, 2010, pp. 807–814. ISBN: 978-1-60558-907-7.

[7] C.M. Bishop. *Pattern Recognition and Machine Learning*. Springer, Berlin: Information Science and Statistics, 2006. ISBN: 0387310738.

[8] Sergey Ioffe and Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. arXiv: 1502.03167 [cs.LG].

[9] Johann Huber. *Batch normalization in 3 levels of understanding*. Ed. by Medium - Towards data science. Nov. 2020. URL: https://towardsdatascience.com/batch-normalization-in-3-levels-of-understanding-14c2da90a338.

[10] Md Atiqur Rahman and Yang Wang. "Optimizing Intersection-Over-Union in Deep Neural Networks for Image Segmentation". In: *Advances in Visual Computing*. Ed. by George Bebis et al. Cham: Springer International Publishing, 2016, pp. 234–244. DOI: 10.1007/978-3-319-50835-1_22.

[11] Ran Shi, King Ngi Ngan, and Songnan Li. "Jaccard index compensation for object segmentation evaluation". In: *2014 IEEE International Conference on Image Processing (ICIP)*. 2014, pp. 4457–4461. DOI: 10.1109/ICIP.2014.7025904.

[12] Ravindra Parmar. *Common Loss functions in machine learning*. Ed. by Towards Data Science. Sept. 2018. URL: https://towardsdatascience.com/common-loss-functions-in-machine-learning-46af0ffc4d23.

[13] Tsung-Yi Lin et al. *Focal Loss for Dense Object Detection*. 2018. arXiv: 1708.02002 [cs.CV].

[14] Sebastian Ruder. "An overview of gradient descent optimization algorithms". In: *CoRR* abs/1609.04747 (2016). arXiv: 1609.04747. URL: http://arxiv.org/abs/1609.04747.

[15] Ning Qian. "On the momentum term in gradient descent learning algorithms". In: *Neural networks :the official journal of the International Neural Network Society* (1999), pp. 145–151. DOI: https://doi.org/10.1016/S0893-6080(98)00116-6.

[16] Yurii Nesterov. "A method for unconstrained convex minimization problem with the rate of convergence $o(1/k^2)$". In: *Doklady AN USSR* 269 (1983), pp. 543–547. URL: https://ci.nii.ac.jp/naid/20001173129/en/.

[17] John Duchi, Elad Hazan, and Yoram Singer. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization". In: *Journal of Machine Learning Research* 12.61 (2011), pp. 2121–2159. URL: http://jmlr.org/papers/v12/duchi11a.html.

[18] Diederik P. Kingma and Jimmy Lei Ba. "Adam: a Method for Stochastic Optimization". In: *International Conference on Learning Representations* (2017), pp. 1–13. arXiv: 1412.6980 [cs.LG]. URL: https://arxiv.org/abs/1412.6980.

[19] Timothy Dozat. "Incorporating Nesterov Momentum into Adam". In: *ICLR Workshop* (2016).

[20] Peter Roelants. *Machine Learning 101*. Ed. by Medium - Onfido Tech. May 2017. URL: https://medium.com/onfido-tech/machine-learning-101-be2e0a86c96a.

[21] Yanzhao Wu et al. *Demystifying Learning Rate Policies for High Accuracy Training of Deep Neural Networks*. 2019. arXiv: 1908.06477 [cs.LG].

[22] Karen Simonyan and Andrew Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. 2015. arXiv: 1409.1556 [cs.CV].

[23] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. *U-Net: Convolutional Networks for Biomedical Image Segmentation*. 2015. arXiv: 1505.04597 [cs.CV].

[24] T. Li et al. "Random-Drop Data Augmentation of Deep Convolutional Neural Network for Mineral Prospectivity Mapping". In: *Natural Resources Research, vol. 30, no. 1* (Feb. 2021), pp. 27–38. DOI: 10.1007/s11053-020-09742-z.

[25] C. Shorten and T. M. Khoshgoftaar. "A survey on Image Data Augmentation for Deep Learning". In: *Journal of Big Data, vol. 6, no. 1* (July 2019), p. 60. DOI: 10.1186/s40537-019-0197-0.

[26] Min Lin, Qiang Chen, and Shuicheng Yan. *Network In Network*. 2014. arXiv: 1312.4400 [cs.NE].

[27] Shervin Minaee et al. *Image Segmentation Using Deep Learning: A Survey*. 2020. arXiv: 2001.05566 [cs.CV].

[28] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV].

[29] C. Szegedy et al. "Going deeper with convolutions". In: *Proceedings of the IEEE conference on computer vision and pattern recognition* (2015), pp. 1–9.

[30] J. L. Elman. "Finding structure in time". In: *Cognitive science, vol. 14, no. 2* (1990), pp. 179–211.

[31] M. I. Jordan. "Serial order : A parallel distributed processing approach". In: *Advances in psychology, vol. 121* (1997), pp. 471–495.

[32] H. Jaeger and H. Haas. "arnessing nonlinearity : Predicting chaotic systems and saving energy in wireless communication". In: *Science, vol. 304, no. 5667* (2004), pp. 78–80.

[33] S. Hochreiter and J. Schmidhuber. "Long short-term memory". In: *Neural computation, vol. 9, no. 8* (1997), pp. 1735–1780.

[34] Andre Ye. *Long Short-Term Memory Networks Are Dying: What's Replacing It?* Ed. by Medium - Towards data science. Sept. 2020. URL: https://towardsdatascience.com/long-short-term-memory-networks-are-dying-whats-replacing-it-5ff3a99399fe.

[35] B. van Merrienboer K. Cho et al. "Learning phrase representations using RNN encoder-decoder for statistical machine translation". In: *Proc. of EMNLP'14* (2014), pp. 1724–1734.

[36] Arden Dertat. *Applied Deep Learning - Part 3: Autoencoders*. Ed. by Medium - Towards data science. Oct. 2017. URL: https://towardsdatascience.com/applied-deep-learning-part-3-autoencoders-1c083af4d798.

[37] P. Vincent et al. "Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion". In: *Journal of machine learning research, vol. 11, no. Dec* (2010), pp. 3371–3408.

[38] D. P. Kingma and M. Welling. "Auto-encoding variational bayes". In: *arXiv preprint arXiv:1312.6114* (2013).

[39] Renu Khandelwal. *Anomaly Detection using Autoencoders*. Ed. by Medium - Towards data science. Jan. 2021. URL: https://towardsdatascience.com/anomaly-detection-using-autoencoders-5b032178a1ea.

[40] Zhaoqing Pan et al. "Recent Progress on Generative Adversarial Networks (GANs): A Survey". In: *IEEE Access* 7 (2019), pp. 36322–36333. DOI: 10.1109/ACCESS.2019.2905015.

[41] Laurence Moroney. *AI and Machine Learning for Coders*. O'Reilly Media, Inc., Oct. 2020. ISBN: 9781492078197.

[42] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 2nd Edition*. O'Reilly Media, Inc., Sept. 2019. ISBN: 9781492032649.

[43] Pytorch by Facebook. Sept. 2016. URL: https://pytorch.org/.

[44] Keras. Mar. 2015. URL: https://keras.io/.

[45] Yasmeen Khaled and Amr Kayid. "Performance of CPUs/GPUs for Deep Learning workloads". In: *Artificial General Intelligence* (May 2018). DOI: 10.13140/RG.2.2.22603.54563.

[46] Eugenio Culurciello. *Hardware for Deep Learning*. Ed. by Towards Data Science. Mar. 2017. URL: https://towardsdatascience.com/hardware-for-deep-learning-8d9b03df41a.

[47] Google Cloud. *Cloud Tensor Processing Units (TPUs)*. May 2016. URL: https://cloud.google.com/tpu/docs/tpus.

[48] S. Hao, Y. Zhou, and Y. Guo. "A Brief Survey on Semantic Segmentation with Deep Learning, vol 206". In: *Neurocomputing* (2020), pp. 302–321. DOI: https://doi.org/10.1016/j.neucom.2019.11.118.

[49] Abdul Mueed Hafiz and Ghulam Mohiuddin Bhat. "A survey on instance segmentation: state of the art". In: *International Journal of Multimedia Information Retrieval* 9.3 (July 2020), pp. 171–189. ISSN: 2192-662X. DOI: 10.1007/s13735-020-00195-x. URL: http://dx.doi.org/10.1007/s13735-020-00195-x.

[50] Sourav Samantaa et al. *Multilevel Threshold Based Gray Scale Image Segmentation using Cuckoo Search*. 2013. arXiv: 1307.0277 [cs.CV].

[51] Anurag Arnab et al. "Conditional Random Fields Meet Deep Neural Networks for Semantic Segmentation: Combining Probabilistic Graphical Models with Deep Learning for Structured Prediction". In: *IEEE Signal Processing Magazine* 35.1 (2018), pp. 37–52. DOI: 10.1109/MSP.2017.2762355.

[52] Y. Lecun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE, vol. 86, no. 11* (1998), pp. 2278–2324. DOI: 10.1109/5.726791.

[53] A. Krizhevsky, I. Sutskever, and G. E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *in Advances in Neural Information Processing Systems, vol. 25* (2012). URL: https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[54] Dongxian Wu et al. *Skip Connections Matter: On the Transferability of Adversarial Examples Generated with ResNets*. 2020. arXiv: 2002.05990 [cs.LG].

[55] Sik-Ho Tsang. *Review: MobileNetV1 — Depthwise Separable Convolution (Light Weight Model)*. Ed. by Medium - Towards data science. Oct. 2018. URL: https://towardsdatascience.com/review-mobilenetv1-depthwise-separable-convolution-light-weight-model-a382df364b69.

[56] Andrew G. Howard et al. *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*. 2017. arXiv: 1704.04861 [cs.CV].

[57] Jonathan Long, Evan Shelhamer, and Trevor Darrell. *Fully Convolutional Networks for Semantic Segmentation*. 2015. arXiv: 1411.4038 [cs.CV].

[58] Kim K et al. "A Deep Learning-Based Automatic Mosquito Sensing and Control System for Urban Mosquito Habitats". In: *Sensors* 2785.12 (2019). DOI: https://doi.org/10.3390/s19122785.

[59] V. Badrinarayanan, A. Kendall, and R. Cipolla. "SegNet: A deep convolutional encoder-decoder architecture for image segmentation". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 39, no. 12* (2017), pp. 2481–2495.

[60] Liang-Chieh Chen et al. *Semantic Image Segmentation with Deep Convolutional Nets and Fully Connected CRFs*. 2016. arXiv: 1412.7062 [cs.CV].

[61] Liang-Chieh Chen et al. *DeepLab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs*. 2017. arXiv: 1606.00915 [cs.CV].

[62] Liang-Chieh Chen et al. *Rethinking Atrous Convolution for Semantic Image Segmentation*. 2017. arXiv: 1706.05587 [cs.CV].

[63] Liang-Chieh Chen et al. *Encoder-Decoder with Atrous Separable Convolution for Semantic Image Segmentation*. 2018. arXiv: 1802.02611 [cs.CV].

[64] François Chollet. *Xception: Deep Learning with Depthwise Separable Convolutions*. 2017. arXiv: 1610.02357 [cs.CV].

[65] Hengshuang Zhao et al. "Pyramid Scene Parsing Network". In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2017), pp. 6230–6239. DOI: 10.1109/CVPR.2017.660.

[66] Ozan Oktay et al. *Attention U-Net: Learning Where to Look for the Pancreas*. 2018. arXiv: 1804.03999 [cs.CV].

[67] IEEE International Symposium on Biomedical Imaging (ISBI). Mar. 2012. URL: http://brainiac2.mit.edu/isbi_challenge/.

[68] Alexander Buslaev et al. "Albumentations: Fast and Flexible Image Augmentations". In: *Information* 11.2 (2020). ISSN: 2078-2489. DOI: 10.3390/info11020125. URL: https://www.mdpi.com/2078-2489/11/2/125.

[69] Python. *What is Python? Executive Summary*. URL: https://www.python.org/doc/essays/blurb/.

[70] Jetbrains. *PyCharm documentation (Quick start guide)*. URL: https://www.jetbrains.com/help/pycharm/quick-start-guide.html.

[71] Google Research. *Colaboratory - Frequently Asked Questions*. URL: https://research.google.com/colaboratory/faq.html.

[72] PysimpleGUI. *Python GUIs for Humans*. URL: https://pypi.org/project/PySimpleGUI/.

[73] NumPy. *What is NumPy?* URL: https://numpy.org/doc/stable/user/whatisnumpy.html.

[74] Matplotlib. *Pandas Documentation*. URL: https://pandas.pydata.org/docs/.

[75] Scikit-learn. *Scikit-learn - Machine Learning in Python*. URL: https://scikit-learn.org/stable/.

[76] Jieneng Chen et al. *TransUNet: Transformers Make Strong Encoders for Medical Image Segmentation*. 2021. arXiv: 2102.04306 [cs.CV].

[77] Hu Cao et al. *Swin-Unet: Unet-like Pure Transformer for Medical Image Segmentation*. 2021. arXiv: 2105.05537 [eess.IV].