Applying Deep Learning to Medical Images

Ricardo Jorge da Silva Diniz

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisors: Prof. Dr. Arlindo Manuel Limede de Oliveira
Prof. Dr. Mário Alexandre Teles de Figueiredo

Examination Committee

Chairperson: Prof. Dr. João Fernando Cardoso Silva Sequeira

Supervisor: Prof. Dr. Arlindo Manuel Limede de Oliveira

Members of the Committee: Prof. Dr. Alexandre José Malheiro Bernardino

July 2019

Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

I thank my parents for their unconditional love and continued support, your encouragement and sometimes harsh but much needed words were what made all of this possible. To my extended family, particularly, my brother João and my uncle Jorge, I offer my deepest appreciations, your thoughts and actions helped me retain focus during my lowest moments.

I would like to extend my thanks to my supervisor Professor Arlindo Oliveira and co-supervisor Professor Mário Figueiredo for the opportunity to tackle this challenging topic and peer into their vast pool of knowledge despite their busy schedules.

Finally, to my friends and colleagues with whom I've spent these last couple of years studying, working and living with, in and outside of Técnico. Special thanks are due to a very special friend and companion, Teresa Albino, to my housemate and one of my greatest friends, Nuno Oliveira, to my housemate and oldest friend, Francisco Ferreira, and to another one of my greatest friends, João Ponte.

Abstract

Deep convolutional networks have recently been embraced by the academic community as a competitive solution for visual recognition tasks. Among these networks, the fully convolutional neural networks have been gaining traction as they drop the traditional fully-connected layers of CNNs in favor of more convolutional layers.

The original fully convolutional network, using layer skipping, was capable of achieving great results when provided enough samples. This architecture was extended into the U-Net which outperforms the FCNN, while being both faster and less computationally cumbersome than it. Both architectures are designed to work with 2D input images. However most medical images, such as ultrasounds and MRIs, are 3D. Built upon the underlying principles beyond the U-Net and the FCNN, the V-Net was created. It is a volumetric FCNN which introduces a new objective function, discards pooling layers in favor of more convolutional layers and performs residual propagation. V-Nets have achieved a good performance across all visual recognition tasks, being comparable to the state-of-the-art solutions while requiring a fraction of the processing time.

In this thesis several variants of U-Net and V-Net are implemented to, firstly, attest to their good performance on visual segmentation tasks of medical data, and, secondly, to assess how the objective function, kernel's receptive fields, residual propagation, activation functions and optimization method impact the model's performance. A secondary objective of this thesis is to bridge the gap between theoretical knowledge and practical implementations by analyzing Google's Tensorflow API, which was designed specifically for distributed computing based machine learning.

Keywords

FCNN; U-Net; V-Net; Deep Learning; Image Segmentation; Tensorflow.

Resumo

Redes convolucionais profundas foram recentemente adotadas pela comunidade científica como uma solução competitiva para tarefas de reconhecimento visual. Entre estas redes, as FCNN têm ganho popularidade por obterem bons resultados ao abandonarem as camadas finais totalmente conectadas das CNN clássicas.

A FCNN original, com utilização do salto entre camadas, foi capaz de atingir bons resultados para *datasets* grandes. Esta arquitetura inspirou a U-Net, que apresenta um melhor desempenho sendo, simultaneamente, mais rápida e computacionalmente mais leve. Estas estão projetadas para trabalhar com imagens 2D. No entanto a maioria das imagens médicas, tais como ultrassons e ressonâncias magnéticas, são 3D. Surge então a V-Net, que consiste numa rede neuronal totalmente convolucional desenhada para operar com imagens 3D, a qual introduz uma nova função objetivo, remove camadas de filtragem por agregação e efetua propagação residual. V-Nets apresentam bons resultados em diversas tarefas de reconhecimento visual, exigindo uma fração do tempo de processamento para atingir os mesmos resultados que os seus competidores.

Neste trabalho são implementadas variantes de U-Net e V-Net para comprovar o bom desempenho destas arquiteturas em tarefas de segmentação visual de imagens do foro médico e para avaliar a forma através da qual a função objetivo, os parâmetros das camadas convolucionais, a propagação de resíduos, as funções de ativação e o método de otimização afetam o desempenho.
Um objetivo secundário do presente trabalho é o de estabelecer uma relação entre a base teórica e uma implementação através da análise da API Tensorflow da Google, desenhada para aprendizagem automática, com ênfase em computação distribuída.

Palavras Chave

FCNN; U-Net; V-Net; Aprendizagem Profunda; Segmentação de Imagem; Tensorflow.

Contents

List of Figures

List of Tables

Acronyms

CNN Convolutional Neural Network
FCNN Fully Convolutional Neural Network
2D Two Dimensional
3D Three Dimensional
API Application Programming Interface
ILSVRC ImageNet Large-Scale Visual Recognition Challenge
MLP Multilayer Perceptron
MLE Maximum Likelihood Estimation
BGD Batch Gradient Descent
SGD Stochastic Gradient Descent
MNIST Modified National Institute of Standards and Technology
IoU Intersection over Union
ReLu Rectified Linear Unit
PReLu Parametrized Rectified Linear Unit
MRI Magnetic Resonance Imaging
B-spline Basis Spline
RLE Run-length Encoding
RGB Red Green Blue
GPU Graphics Processing Unit
CPU Central Processing Unit

Chapter 1

Introduction

This introductory chapter aims to give an overview of the thesis. It includes the context and motivation under which it was developed, its objectives and its general structure.

1.1 Motivation

Convolutional Neural Networks (CNNs) have existed for a long time [8]. However until recently the available processing power, limited size of available datasets, hardware limitations such as memory storage and the shallowness of the proposed architectures worked against their good performance. In 2012 AlexNet [1] was introduced to the scientific community and quickly became the first widespread deep convolutional net. It was based on Gradient-based learning applied to document recognition [2] and won that year's ImageNet Large-Scale Visual Recognition Challenge (ILSVRC). From then onwards the focus on CNNs brought many and great advancements to their architectures by tweaking activation and objective functions, adding more layers or reformulating how layers interact with each other, creating new layers, and other innovations [3], [4] and [5].

The Fully Convolutional Neural Network (FCNN) architecture [6] has proven itself to be competitive for image segmentation at a relatively low memory and computational power footprint compared to alternatives. In the medical and biomedical domains, the V-Net [7] and its predecessor the U-Net [8], both types of FCNNs, have been widely accepted as fitting solutions to several visual recognition tasks due to their ability to achieve good results from small datasets in a timely manner. By expanding on them and understanding the improvements one offers over the other, as well as how to implement them in modern application programming interface (API) we can attest to their worth and, simultaneously, consider possible improvements.

1.2 Objectives

This thesis aims to:

- Compare different V-Net configurations with variations of a well reputed predecessor, U-Net, to attest to the improvements of the former versus the latter and provide an explanation of these improvements;
- Provide valuable insights on how to implement such models and their extended pipeline in Python with recourse to the Tensorflow API.

The first objective can be narrowed down to attesting to the relatively good performance of modern FCNN implementations towards medical image segmentation, therefore corroborating their respective authors' work.

The Tensorflow API was designed with machine learning at its core for distributed computations. Through a distributed computation setting it is possible to train very deep networks on massive augmented datasets in a short period of time. As such bridging the gap between the theoretical knowledge underlying the networks in study and the tools that allow for their implementation in such a setting is also considered to be of utmost importance.

1.3 Structure

This thesis' structure revolves around five major chapters, Chapter 1 deals with the introduction and general motivation beyond the work done as well as the thesis' objectives, Chapters 2 and 3 cover theoretical and practical aspects, Chapter 4 is centered around the results obtained and Chapter 5 provides a summary of the thesis, some insightful conclusions and future work proposal.

Chapter 2 consists of the background theory required to understand what deep learning is, particularly, convolutional neural networks, what is a deep learning algorithm and how is it formulated, with emphasis on supervised learning, and how to quantify and measure learning and the processes it entails.

Chapter 3 bridges the gap between theory and practice by covering the full pipeline of a deep learning algorithm, in the supervised learning setting, from preprocessing to postprocessing and all steps in between. It also provides an overlook on the FCNN and the Tensorflow API.

Chapter 4 cover the U-Net and V-Net, as originally proposed by their author's, followed by the variants that were implemented to tackle each challenge. The results obtained are briefly commented upon.

Finally, Chapter 5 provides closure to the work done by providing a summary of the thesis, followed by the most insightful conclusions to be drawn from the results obtained. At its very end, a brief discussion on future work to be done is provided as well.

Chapter 2

Background

2.1 Introduction

This chapter aims to provide the scientific and technical background necessary to understand the work developed in Chapter 4. It covers some relevant aspects of machine learning, since the full analysis and etymology of machine learning and deep learning are considered to be outside the scope of this dissertation. Section 2.2 provides a brief introduction to machine and deep learning as a whole, Section 2.3 exposes some of the technical aspects and machine learning concepts with a general focus on deep learning and Section 2.4 deals with optimization and regularization with emphasis on how these affect the learning process.

2.2 Deep Learning

This section is mostly based on chapters 1, 5 and 9 of the reference deep learning material [9].

2.2.1 Overview

Machine learning is, roughly put, a form of applied statistics allied to computing power to estimate functions of increased complexity without emphasizing the definition of confidence intervals for such functions. Deep learning is a branch of machine learning in which complex tasks can be learned through successive stacking of simpler tasks in a layer format. While the former has been a recurrent topic within the scientific community as a whole since the mid twentieth century, albeit with different nomenclatures and approaches, the latter appeared in the late twentieth century and vanished due to the limited success and demanding computational requirements, but has since resurfaced in the early twenty-first century and has been growing in popularity amongst the scientific community due to the success in tackling speech and object recognition with improved generalization compared to traditional machine learning methods.

Deep learning algorithms usually consist of a combination of a dataset, an objective function, an optimization method and a model. Barring the dataset, most of the previous topics are still actively being researched within the scientific community. Typically, deep learning deals with tasks that fall under the domain of supervised learning in which there is a training set with well-established output targets that is fed to the network to teach it and a test set used to evaluate the performance of the model under a specific metric.

Much of the statistical and technical background of machine learning is common to deep learning. Therefore it is critical to cover some of the basic principles of the former to understand the

foundations of the latter. Due to constraints on scope and the vast amount of information and technical aspects of deep learning as a whole, most of the topics below tie in with feedforward deep networks.

A feedforward deep network, also referred to as a multilayer perceptron (MLP), is a mathematical function mapping a set of input values to another set of output values. This function is obtained through the composition of several simpler functions, with each distinct application resulting in a new representation of that function's input. MLP's are the standard of deep learning applications as very few models stray from this conventional structure.

2.2.2 Convolutional Neural Networks

Convolutional neural networks are a specific type of neural network specialized in processing data that has a known grid-like topology. This type of model is the cornerstone of the work presented in the following chapters. Such data can be of the one-dimensional type, such as time-series with a regular span between samples, two-dimensional such as a grid of pixels when processing images and onwards for third-dimensional and higher data.

Generally, a convolution is a linear operation on two real valued argument functions, but in neural network convention the first argument, or input, is usually a multidimensional array of data and the second argument, the kernel, typically is a multidimensional array of parameters known as tensors whose values will be changed through the learning process. The output of such an operation is often described as a feature map. A convolution with a two-dimensional image I for input and a two-dimensional kernel K can be written in the following form

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) \qquad (2.1)$$

which using the commutative property of the convolution operation can be rewritten to

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n) \qquad (2.2)$$

The former expression can be very computationally expensive due to the massive amount of variation of the values involved and the latter, or flipped version, is considerably more desirable computational effort wise. Due to the fact that this flipping process brings little to no advantages towards the successful performance of a deep learning model, most libraries available implement a cross-correlation function, under the guise of convolution, which is given by

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n K(i+m, j+n)I(m,n) \qquad (2.3)$$

which functionally works as a convolution without the flipping.

The main concepts that help consolidate the perceived usefulness of convolutional neural networks are sparse interactions, parameter sharing and equivariant representations.

While traditional neural network layers use matrix multiplication between themselves and a parameter matrix with another parameter establishing the interaction between each input unit and each output unit, convolutional neural networks have sparse interactions due to the kernels' reduced size when compared with the input, which effectively means not all input units interact with all output units as showcased in Figure 2.1.

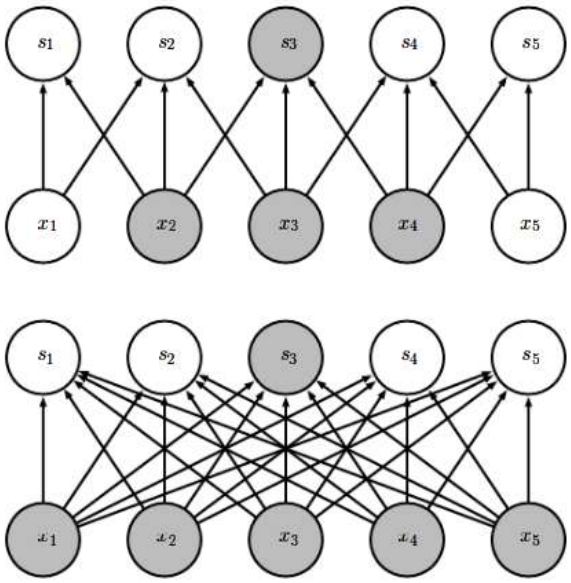


Figure 2.1: An output unit, s3, is highlighted as well as the input units, x, that interact with it. (Top) Sparse connectivity between units of adjacent layers. (Bottom) Full connectivity between units of adjacent layers [9].

In traditional neural networks each element of the weights' matrix is used only once when computing the output of a unit, by multiplying it by an element of the input unit. In a convolutional neural network each member of the kernel is used at every position of the input, barring border cases where only part of the kernel overlaps with the input. Thus the same weight is reused by several inputs. This is known as parameter sharing and can significantly reduce the number of parameters of the network as a whole when compared with traditional networks.

An equivariant function is one in which changes to the input reflect themselves in the same manner on the output or, in other words, $f(x)$ is equivariant to g if

$$f(g(x)) = g(f(x)) \qquad (2.4)$$

Performing a convolution with a translation function g applied to a given input, that is, one that shifts the input, results in the same output as taking the original unshifted version, applying the

convolution and then shifting this output. In the convolutional neural network paradigm this property allied with parameter sharing leads to each convolutional layer being insensitive to translation. As an example, this equivariant representation property extends to an whole image since knowledge of certain features such as edges can be propagated to other points of the image. Some applications such as facial detection, for instance, do not benefit from total parameter sharing and require special attention and constraints to effectively learn the intended outputs without relying on the equivariant representation property.

Further details on this type of neural network and its properties are provided in the following chapters and sections of this thesis where the relevant corresponding background is covered, with particular incidence in Chapter 3 where the practical implementation of the convolutional neural network is covered.

2.3 Technical Background

2.3.1 Building the Algorithm

A machine learning task consists of how a machine learning algorithm should process a collection of features that were collected from some object or event and are quantitatively defined, as seen on chapter 5 of the reference deep learning material [9]. There are many tasks a machine learning algorithm can tackle such as classification, regression, anomaly detection and denoising, for example. This thesis delves exclusively into binary object classification tasks, which entail having the algorithm specify the class to which each individual feature, in this case pixel or voxel, belongs to. These effectively function as segmentation tasks.

There are several ways to approach this task, but the one chosen consists of outputting a probability distribution over both classes and setting a probability threshold above which that class is selected for the specific pixel or voxel. To vaguely illustrate this process, consider a vector $x \in \mathbb{R}$, where each entry x is a feature, in this example a pixel of an image whose values range from 0 to 255. The solution to this task entails producing a function f which outputs a probability distribution over classes, and then outputting a $y = P(f(x \geq 0{,}5))$, where each y is a binary digit which represents whether x belongs to a class or not.

To attain completion of a task a machine learning algorithm requires feedback from a quantitative measure of performance. This measure must be designed to fit the task at hand and corresponds to one of the major factors that influence the success of an algorithm, as a great algorithm with poor performance measure will lead to learning the wrong output. Simple measures of performance, often used for classification tasks are the proportion of examples to which the model produces the correct output, accuracy, and the proportion of examples to which the model produces the incorrect output, error rate. Empirical testing, supported by theoretical background, has led the scientific community to define more complex measures of performance which are designed to aptly fit the task at hand, often demanding extensive knowledge of the data generating source and domain.

Machine learning algorithms are loosely categorized by the way they perform the learning process, that is how they extract pattern information from each dataset point, although these categories are not rigid and some belong to several categories. Unsupervised learning algorithms usually intend to learn the probability distribution that generated the dataset or try to categorize it by drawing their own insights, or patterns, from it. Supervised learning algorithms use a dataset of labelled features by, typically, performing their task multiple times and comparing the output with the labels or targets provided. This means that, typically, an unsupervised algorithm will learn probability distributions, $p(x)$, by finding patterns on an observed dataset, x, while supervised learning algorithms observe x, compare their output with the labels of x, y, and attempt to learn their task by, usually, predicting y from x in the form of computing a $p(x \mid y)$.

Finally, there are some machine learning algorithms which do not experience a fixed dataset and instead interact with an environment. These algorithms are designated by reinforcement learning algorithms as they learn from a feedback loop between the learning system and its experiences. The work covered in Chapter 4 and throughout this thesis consists solely of supervised machine learning algorithms.

To be relevant a machine learning algorithm must be able to perform well on previously unseen data, that is, must be able to generalize well. This generalization is often assessed through two particular parameters, training error and testing error.

The training error is present in virtually any supervised learning model and at its core is, in essence, an optimization problem. It is measurable through a custom metric, for example, autoencoders attempt to copy their input, as such the differences between the original and the copy or more complex derivatives can be used as a measurement for training error. It is of utmost importance to have well defined metrics so these are typically formulated from that particular data domain's own theoretical and empirical knowledge pool allied to statistics as well as information theory.

Training error goes hand in hand with the test error, also denominated generalization error, to help define how much the model generalizes. Test error is defined as the expected value of future errors and is typically measured during the evaluation stage and ideally with inputs that differ from those of the training set. There is a limit to how much a model can learn new patterns or generalize and this is generally denominated as capacity.

Supported on the concepts previously covered, particularly capacity, the metric which is universally used to assess how well a model is modelling the actual data, fitting, can now be defined. Model fitting is defined as the model's ability to learn the underlying patterns to a specific problem. In a sense it can be seen as an attempt to choose the type of function, from the limited set expressed by its capacity, that, with some parameter tweaking, most aptly fits the data distribution from which the dataset is generated. To exemplify consider Figure 2.2 that showcases an attempt to fit a polynomial function to raw data. Supose that same algorithm is able to generate from first to ninth order polynomial functions, then it can be seen that it is able to learn the parameters of that function in such a way it appears to be modelling the data correctly, that is, has an adequate capacity so it selects an appropriate ninth order polynomial fit to the data.

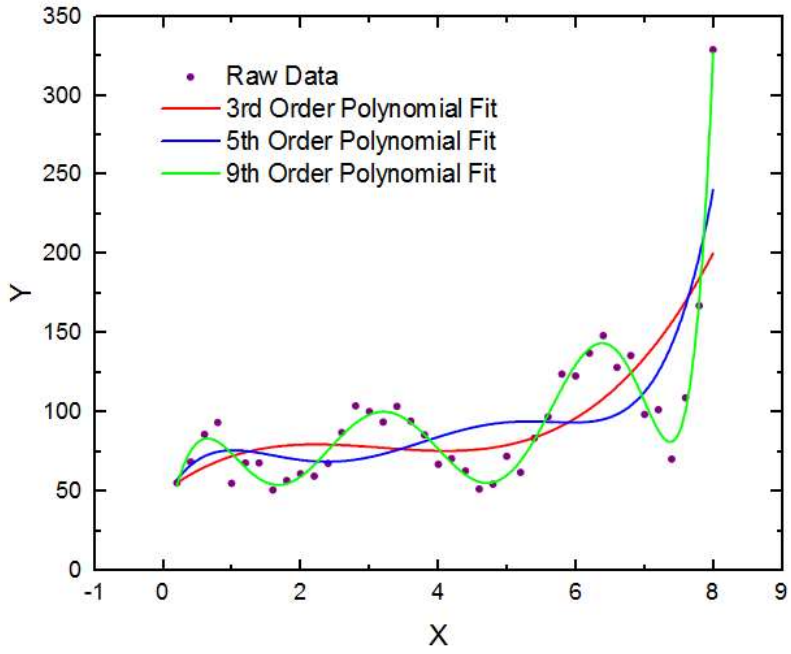


Figure 2.2: Polynomial fit of the raw data with recourse to 3rd, 5th and 9th order polynomials [Matlab].

An under fitted model is one that fails to accurately follow the behavior of the input data, often through lack of non-linearity, lack of training data, training with corrupted data or even training with samples that share too many of the same patterns. It is measured by the training error, relative to an error metric, in such a way that a model with relatively low training error is considered to fit it well and a model with relatively high training error underfits that data. In the example of Figure 2.2, if the model's capacity only allowed for up to the fifth order polynomial functions than the model would underfit the data it was trying to model, no matter how well the parameter optimization process went.

Overfitting, on the other hand, is one of the most discussed topics on the machine learning community. It occurs when a model performs well on the training set of samples, that is achieves a relatively low training error on known data, but displays a bad performance when fed new, previously unseen data, even if this unseen data respects the basis of sharing a similar data generating source. Returning to the example in Figure 2.2, and assuming that the raw data showcased is but one of several samples then a situation such as that shown in Figure 2.3 might occur. Essentially the apparent adequate fit of the underlying data turns out to be an overfit as the algorithm does not properly adapt to new data fed to it or, in other words, cannot learn the parameter values necessary to generalize and be well performant for most data samples, not just those it has already seen.

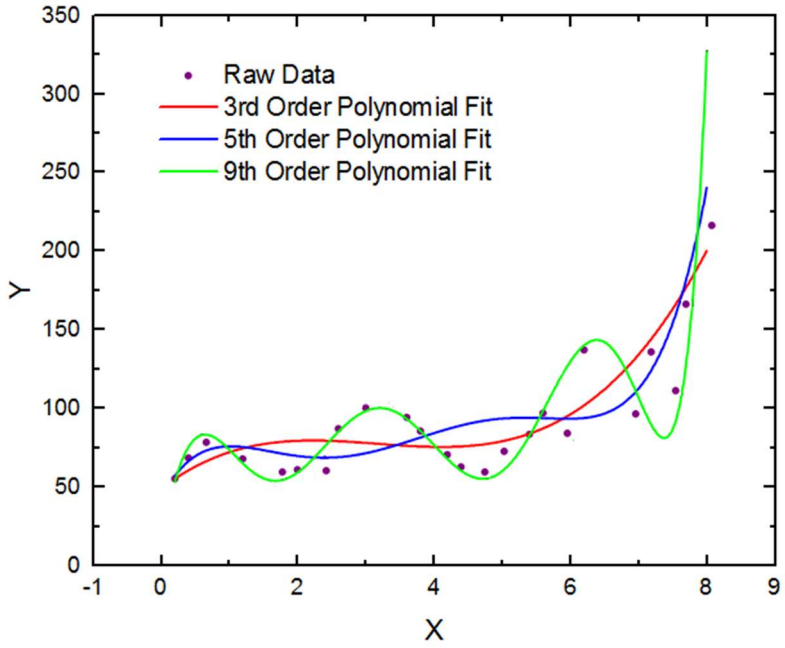


Figure 2.3: Overfitting a data distribution by a 9th order polynomial function and underfitting it by 3rd and 5th order polynomials [Matlab].

An objectively good model therefore requires a good capacity, in the sense that it can learn functions that fit the underlying data well, and it must be capable of learning from new data samples by tweaking the parameters of these functions to suit the general underlying data. This implies a tradeoff between training and evaluation which is often done through means of regularization, covered in Section 2.3.4.

2.3.2 Tuning the Model

Hyperparameters are parameters which are not adapted by the learning algorithm and are those used to control the algorithm's behavior, although in some cases the hyperparameters can be learned and optimized by another algorithm. Usual hyperparameters are those parameters which are too difficult to optimize on the training process and the parameters which pertain to the model's capacity, since due to their nature they will opt for the highest capacity possible, eventually leading to overfitting. A polynomial fit, for example, during the training stage will always opt for a higher-degree polynomial and no penalty over a lower-degree polynomial with penalties since capacity optimization will inherently be a part of its behavior.

A relatively common method used to train hyperparameters is splitting the training set of data and using only part of it to train the model. The unused set or validation set can then be used to define hyperparameters to, hopefully, help improve the generalization of the model and, consequently, the algorithm itself. At the cost of extra computational effort, but often with great results, cross validation can be performed instead. It entails doing multiple training set and validation set splits with differing sets

and then repeating the process of training the hyperparameters followed by a run through specific criteria to decide the weight of each hyperparameter optimization.

Learning rate is one such parameter, and its importance lies in the fact that it acts as a control cap to how much the weights of a network update at every iteration. Initial values, particularly whether it should be a global hyperparameter or there should be a learning rate per parameter and how it should adapt throughout the learning process, are a hot topic of research with plenty of conflicting views within the deep learning community. Typical, but outdated approaches, are to decrease the learning rate monotonically [1], [3], [13] or leaving it at a fixed value [6]. Modern adaptive learning rates, in which criteria are established to define whether the learning rate should decrease, increase or remain the same and how these changes should reflect themselves, for a particular iteration, have been empirically proven to improve results for the same model when compared with the traditional technique [10], [11] [12].

Estimators for parameters, variance and bias are some of the statistical tools that have a relatively great impact on a model's performance. Assuming parameter θ can be point estimated by $\hat{\boldsymbol{\theta}}$ and the set $(x^{(1)},...,x^{(m)})$ is a set of m independent and identically distributed data points, then a point estimator is any function of the data

$$\hat{\boldsymbol{\theta}}_{\boldsymbol{m}} = g(x^{(1)},...,x^{(m)}), \qquad (2.5)$$

which provides a very wide range of values $\hat{\boldsymbol{\theta}}_{\boldsymbol{m}}$ and therefore a wide range of estimator functions. Through the frequentist approach of statistics there is the assumption that there is a fixed unknown θ and $\hat{\boldsymbol{\theta}}$ is processed as a random variable. Function estimation is very similar but the function estimator acts as point estimator in function space. Bias of an estimator is defined as

$$E\left[\hat{\boldsymbol{\theta}}_{\boldsymbol{m}}\right] - \boldsymbol{\theta}, \qquad (2.6)$$

where $E\left[\hat{\boldsymbol{\theta}}_{\boldsymbol{m}}\right]$ is the expectation over the data and θ is the supposed true underlying value used to define the data generating distribution. Bias is typically detrimental to the performance of an algorithm although it tends to be relatively easy to counterbalance on the types of tasks which are dealt in this thesis, provided that there is sufficient knowledge of the data generating source and the domain itself. In some cases, bias can be act as a form of regularization which might improve the performance of a model.

Variance of an estimator is defined as an expectation of how the estimator varies as a function of the data samples. In this context the square root of the variance is denominated standard error instead of the statistical standard deviation nomenclature. The variance is a source of error which should be quantified to then improve the machine learning algorithm as a whole.

Unfortunately, neither the square root of the sample variance nor the square root of the unbiased estimator of the variance, the most commonly used approaches, provide an unbiased estimate of the standard deviation, which tends to be underestimated.

Bias and variance measure the expected deviation from the true value of the function or parameter and the deviation from the expected estimator value that any sampling of the data is likely to cause, respectively, which are two distinct sources of error for an estimator. In practical applications model estimators often cannot have both low variance and low bias, therefore cross-validation, the mean-squared error or other criteria are used to choose an estimator that establishes a decent tradeoff between these. There is a link between bias and variance and the model's capacity, underfitting and overfitting such that an increase in capacity tends to lead to an increase in variance and a decrease in bias and vice-versa. This relationship is loosely illustrated in Figure 2.4.

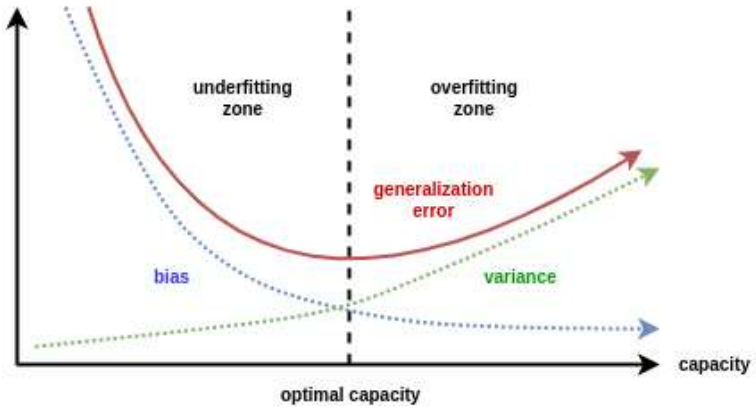


Figure 2.4: Loose relationship between generalization error, capacity, bias and variance [9].

To avoid having to manually select potential estimators and then analyze their variance and bias, some methods have been devised to facilitate this search. Maximum Likelihood Estimation is the most popular of such alternative methods to choose a proper point estimator for a specific model.

Consider the same set of m examples $(x^{(1)},...,x^{(m)})$ defined above, generated by the distribution $p_{model}(\boldsymbol{x};\boldsymbol{\theta})$, and $p_{model}(\boldsymbol{x};\boldsymbol{\theta})$ the parametric family of probability distributions indexed by θ. The maximum likelihood estimator of θ is given by

$$\boldsymbol{\theta}_{ML} = \arg\max_{\theta} \prod_{i=1}^{m} p_{model}(\boldsymbol{x}^{(i)};\boldsymbol{\theta}) \qquad (2.7)$$

In practice the logarithm of the MLE is used instead since it preserves the functionality of the MLE but is less prone to underflow and leads to multiplication operations becoming sum operations, which are easier to compute. Taking into the account the expression obtained in (2.7) by applying the logarithm and dividing by m we obtain

$$\boldsymbol{\theta}_{ML} = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{m} \log p_{\boldsymbol{model}}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}) \qquad (2.8)$$

MLE can thus be interpreted as an attempt to make the model distribution match the observed distribution, which in turn provides the most information about the true underlying data generation distribution. The former can then, hopefully, get better estimators as a result of MLE.

In the supervised learning context pertaining to this thesis, a generalized version of MLE to estimate a conditional probability in order to predict a y given x by means of $\boldsymbol{\theta}$ is often used. Assuming that $\boldsymbol{X}$ represents all inputs and $\boldsymbol{Y}$ all the observed targets then the conditional maximum likelihood estimator is

$$\boldsymbol{\theta}_{ML} = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{m} P(\boldsymbol{Y}|\boldsymbol{X}; \boldsymbol{\theta}) \qquad (2.9)$$

which will degenerate to

$$\boldsymbol{\theta}_{ML} = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{m} \log P(\boldsymbol{y}^{(i)} \mid \boldsymbol{x}^{(i)}; \boldsymbol{\theta}) \qquad (2.10)$$

in such a case where the samples are independent and identically distributed.

The main appeal of MLE is that it can be shown to be the best estimator asymptotically in terms of rate of convergence, as the number of examples increases. It also benefits from consistency should the underlying true data distribution actually belong to the estimators model family and be depicted exclusively by a single estimator.

Unlike the frequentist statistics approach covered up to this point, the Bayesian statistics approach considers all possible $\boldsymbol{\theta}$ values when making a prediction instead of using the single $\boldsymbol{\theta}$ value chosen to do all predictions. It uses probability to discern degrees of certainty in states of knowledge, in such a way as to observe the dataset directly and use the true unknown $\boldsymbol{\theta}$ as a random variable instead.

The process begins by establishing a prior probability distribution for $\boldsymbol{\theta}$ with the highest entropy possible, thus often a Gaussian or Uniform distribution. Then by observing the dataset itself and combining it with the prior through Bayes' rule we obtain a measure of the impact of the true observations on the prior. The prior suffers from a gradual entropy loss as it focuses on specific, and likely, values of the parameters.

Bayesian estimation differs from MLE in two major ways, the former uses the full distribution over $\boldsymbol{\theta}$ to make predictions while the latter makes prediction from a single point estimate of $\boldsymbol{\theta}$ and the Bayesian estimator's prior distribution usually influences the process by shifting probability mass density towards regions of the parameter space which were given preference beforehand. Bayesian estimation

tends to generalize better for scenarios of limited training data, while also avoiding overfitting, but requires a significant computational effort increase with regards to MLE which can be drastically costly for large training sets. Bayesian estimates also return a covariance matrix with information on how likely each model parameter value is.

2.4 Learning Techniques

2.4.1 Optimization

This section is mainly inspired by chapters 4 and 5 of the deep learning reference material [9] and two papers relevant to the subject at hands [12] [14].

Optimization tasks most often refer to an attempt to minimize or maximize a specific function by tweaking their parameters. Throughout this dissertation we aim exclusively to minimize said functions, as such they're referred to as cost or loss functions. Given real x and y numbers and supposing $y = f(x)$, the derivative of $f(x)$, $f'(x)$, provides insights on how to scale a small change of the input to obtain a change in the output. Assuming a small enough ε we can state that

$$f(x+\varepsilon) \approx f(x) + \varepsilon f^{'}(x) \qquad (2.11)$$

This, in turn, supports the notion that by slowly moving x towards the opposite sign of the derivative we can reduce $f(x)$. This process is otherwise known as gradient descent. When dealing with multiple inputs partial derivatives are used.

Some of the major hurdles of gradient descent techniques lie with the distinction between stationary points, $f'(x) = 0$, since we cannot infer on which direction to move. To find the global minima of the function we intend to optimize it becomes necessary to avoid too small steps when at local minima or saddle points in near flat regions.

The implementation of the standard gradient descent also carries some challenges with it, the most significant one for practical applications being how cumbersome it is, as equation (2.11) implies processing the entire dataset, also known as batch of data. Implementations of said equation are therefore designated as batch gradient descent (BGD) since the full dataset must be processed before updating the weights of the network and thus are mostly unused in cutting edge solutions. In practice we often choose a good enough local minimum due to computation or complexity restrictions.

Despite the aforementioned disadvantage BGD also has some desirable properties such as well understood convergence conditions which, if met, will eventually lead to convergence to local or global minima, a plethora of convergence acceleration techniques defined over the span of decades and simpler theoretical analysis of weights actualization and convergence rates. Major disadvantages

of virtually all competitor gradient descent techniques mostly stem from the fact that they do not provide any assurance on successful convergence, on when convergence occurs or even if the network will converge to a good solution should it converge at all.

Given a training set of data with m samples and assuming that the loss function used by an algorithm is given by the sum over the m samples of some per-sample cost function, as is most often the case, it can be shown that the batch gradient descent of the negative conditional log-likelihood of the training data results in a computational effort proportional to the number of samples, as previously mentioned. This can become prohibitively costly to compute for larger training sets which is where stochastic gradient descent (SGD) proves to be a valuable technique. It is the single most used algorithm in the current context of deep learning and many other applications.

SGD handles the gradient as an expectation which can in turn be approximately estimated through a batch of examples, or minibatch, with size m', drawn uniformly from the training set. As such we can fit an infinitely large training set by using only updates with dozens or hundreds of examples with the gradient estimation formed as

$$g = \frac{1}{m'}\nabla_\theta \sum_{i=0}^{m'} L(x^{(i)}, y^{(i)}, \theta) \qquad (2.12)$$

where L, the per-example loss, is given by $L(x, y, \theta) = -\log p(y \mid x; \theta)$. The SGD algorithm then follows $\theta \leftarrow \theta - \varepsilon g$ where ε is the learning rate. In practice enhanced versions of the SGD are mainly used.

There are several advantages associated with SGD such as, and most importantly, the fact that for successful convergence scenarios it is several orders of magnitude faster than BGD. SGD can also perform with datasets updated during model training, often results in better solutions since drawing a random minibatch from the dataset every iteration and using it to compute the gradient descent introduces regularization, which may lead to models with greater capacity, and, if fine-tuned, can be successfully used to track and model changes to the data generator, as it lacks the averaging or other method used to factor the whole dataset in BGD. SGD has also been shown to perform almost as well as BGD in terms of successful convergence for some specific conditions.

SGD also carries with it several disadvantages, the most relevant one being the lack of effective methodology for defining the size of the mini-batches of data and how that reflects itself on the learning rate. This is intuitive as a minibatch with a single sample is most likely not very representative of the remaining dataset due to noise or other factors and might lead to steep fluctuations of the objective function, which in turn might lead to an overshoot of local and global minima. On the other hand, minibatch with several samples can be biased towards certain sample types or remove the generalization boost through noise masking. An illustrative example of the progression of these techniques towards an arbitrary global minimum of a convex surface, the red dot, can be seen in Figure 2.5.

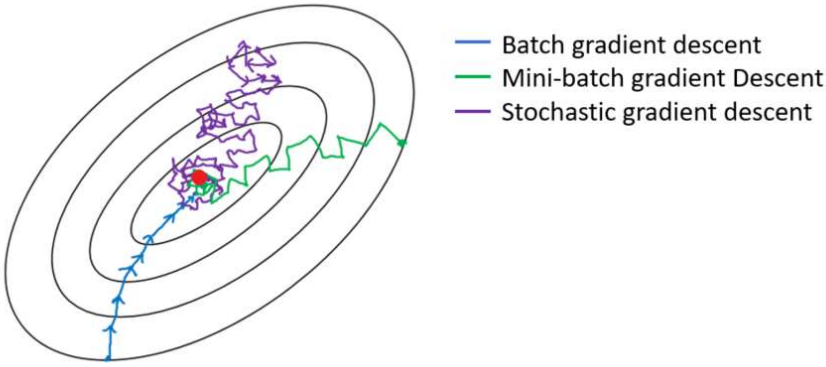


Figure 2.5: Progression of BGD, mini-BGD and SGD towards a global minimum of a convex surface [9].

The work elaborated in Chapter 4 also uses an enhanced version of SGD, as Tensorflow API allows the use of their own implemented enhanced stochastic gradient descent algorithms for optimization with relative ease, being the de-facto algorithm variants for forward and backward propagation.

2.4.2 Regularization

This section is mainly inspired by chapters 5 and 7 of the deep learning reference material [9], and two relevant papers on this subject [15] [16].

As seen in Section 2.2.1 there are several tradeoffs implied when there's an attempt to build an algorithm, and its underlying model, so that it can perform well on a specific type of task. In practical applications we do not expect, for instance, that a model with a trained binary classification algorithm generalizes well for all binary classification tasks, since it will most likely perform poorly compared to specialized counterparts, who are trained to handle a specific type of data from a specific data generating source.

The decision of where to set the thresholds for too little generalization and excessive generalization for a specific model lies with the one designing the model's algorithm and depends on the perceived value of the aforementioned tradeoffs. In this section some of the most relevant tools available to achieve this balance are covered. Due to the scope of this thesis there's a general focus on the application of these tools towards feedforward deep nets.

Regularization is the act of purposely introducing loss of information on any particular step of sample processing in order to achieve a more general and thus better model according to the specified metrics. Regularization comes in many forms and can be done in training, evaluation or testing and works in line with preprocessing and postprocessing to enhance the model's performance. Both the former and the latter are themselves almost entirely forms of regularization on their own. It stands, along with optimization, as one of the most discussed and heated topics in the data science community, since

most of the empirically successful regularization forms still lack proper academic formalism and are only applicable on a case by case basis. Some of the most common methods of regularization are exposed below.

Parameter norm penalties, for instance, consider a parameter norm penalty Ω($\boldsymbol{\theta}$) and add it to the objective, or cost function, $\tilde{J}$. In this scenario the regularized function, $\tilde{J}$, is

$$\tilde{J}(\boldsymbol{\theta};\boldsymbol{X},\boldsymbol{y}) = J(\boldsymbol{\theta};\boldsymbol{X},\boldsymbol{y}) + \alpha\Omega(\boldsymbol{\theta}), \qquad (2.13)$$

where α is a parameter set beforehand whose function is to weight the relative contribution of Ω. The term Ω is denoted regularizer and we can tune α to zero for no regularization or

$$\tilde{J}(\boldsymbol{\theta};\boldsymbol{X},\boldsymbol{y}) = J(\boldsymbol{\theta};\boldsymbol{X},\boldsymbol{y}), \qquad (2.14)$$

or give it a value to get it to have a more relevant impact in the outcome, increasing the training error for that specific sample but, hopefully, decreasing the test error overall. Two of the most used parameter norm penalties are the L2 and L1 regularizers.

L2 regularization or weight decay adds the term $\Omega(\boldsymbol{\theta}) = \frac{1}{2}\|\boldsymbol{w}\|_2^2$ the objective function and drives values closer to the origin while L1 regularization with focus on the origin adds the term $\Omega(\boldsymbol{\theta}) = \|\boldsymbol{w}\|_1$ to the objective function and uses the parameter set beforehand α to control the impact of the sum of absolute values of the individual parameters $\boldsymbol{w}$. A more detailed example to clarify the behavior of L2 and L1 regularization can be found in the source material [15].

Norm penalties as constrained optimization entails using the standard parameter norm penalties added to the objective function to formulate a generalized Lagrange function constrained by some condition. As a simple example, let's consider J defined above and assume the target is to have a constraint that's less than some real constant k. In this scenario a generalized Lagrange function could be:

$$\mathcal{L}(\boldsymbol{\theta},\alpha;\boldsymbol{X},\boldsymbol{y}) = J(\boldsymbol{\theta};\boldsymbol{X},\boldsymbol{y}) + \alpha(\Omega(\boldsymbol{\theta}) - k), \qquad (2.15)$$

to which the solution is

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta},\alpha) = \arg\min_{\boldsymbol{\theta}} \qquad (2.16)$$

The solution depends on changing both $\boldsymbol{\theta}$ and α, so α is fixed to assess the problem as a function of $\boldsymbol{\theta}$ such that it becomes

$$\boldsymbol{\theta} = \arg\min_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}, \alpha) = \arg\min_{\boldsymbol{\theta}} J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \alpha(\Omega(\boldsymbol{\theta}) - k) \qquad (2.17)$$

It can be seen that this scenario degenerates into the previously covered regularized training problem of minimizing $\tilde{J}$. Taking this into account it is possible to infer that the constraint acts on the values of weights between the neurons of the network and, in a sense, α plays the part of widening or narrowing the constraint region such that the smaller the α the larger the constraint region. For example, if the penalty term is an L2 or L1 regularization term and the model works with third-dimensional tensors then the values will be constrained to a surface such as that presented in Figure 2.6.

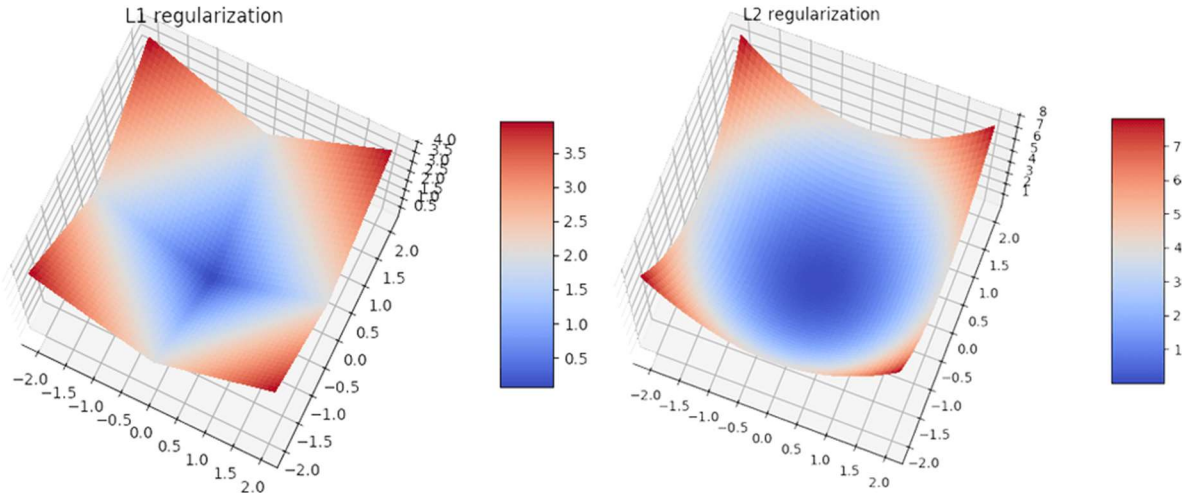


Figure 2.6: Illustrative 3D surfaces to which parameter values are constrained to. (left) L1 regularization surface for the L1 penalty term. (right) L2 regularization surface for the L2 penalty term [27].

Usually this technique of using penalties as a constraint is only used in very well-defined tasks, since it might require a significant computational effort to compute k and the corresponding value for α or penalties might be stuck in local minima for small theta values effectively neglecting some of the units in the network therefore limiting the amount of functions the network can learn. Instead reprojection and explicit constraints techniques are used due to requiring less computational effort, limiting the values of network weights to a general maximum, which corresponds to the limits of the constraining region, instead of forcing them to go towards the origin and they may impose stability on the optimization procedure.

Under constrained problems can also benefit from regularization and for some it is a critical factor for success. This kind of regularization implies knowing beforehand the nature and details of the data or at least having some insights on its composition. It tends to be used to solve underdetermined

problems, for example, those in the linear model domain such as adding an offset to a singular matrix because no variance was detected in some feature so that it becomes inversible and the problem gets a closed form solution.

Parameter tying and sharing unlike the previously mentioned forms of regularization which are done in regard to a fixed region or point is more abstract and aims to establish dependencies between parameters that previous knowledge of the domain allows us to suppose exist whether through constraints or penalties.

Parameter sharing is particularly relevant for the type of model and domain combination this thesis covers, convolutional neural networks and computer vision, respectively. It ties directly with the lower number of unique model parameters on convolutional neural networks, the equivariant representations property of the network and it is a major factor that accounts for the significantly increased network sizes used without a matching increase in training samples.

Augmentation is one of the most important forms of regularization for the classification tasks tackled in this thesis and object recognition in general. It is the process through which datasets are virtually augmented by fake samples either by generating samples similar to those the data generating source does or applying transformations to copies of the dataset's samples so that the model can be trained to become insensitive to translations, cropping, rotations, color, blur, artefacts, scaling, noise and many others. An example of artificial samples generated from an original sample, Figure 2.7, by applying different techniques can be seen in Figure 2.8.

Figure 2.7: Two-dimensional image of a Quokka with its corresponding localization masks [28].

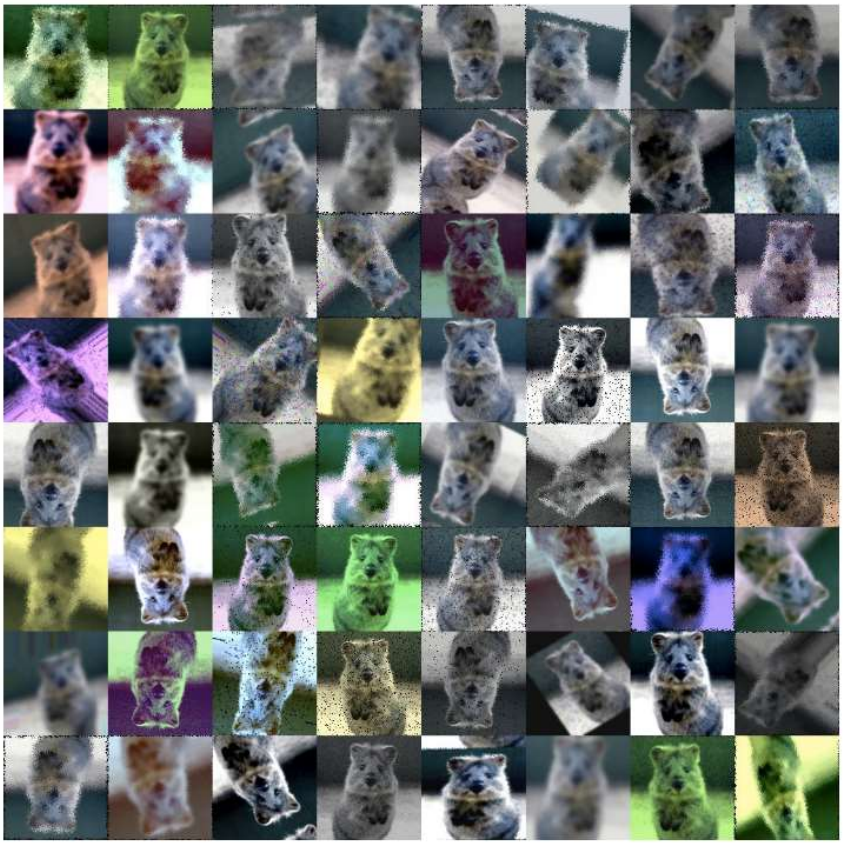


Figure 2.8: Augmented samples generated from Figure 2.7 [28].

When performing dataset augmentations, one must consider the type of task at hand so that augmentation does not introduce more testing error instead of improving the generalization ability of the model's algorithm. One such example is the rotated 5 digit from the MNIST dataset in Figure 2.9, which can be classified as a 6, a 9, a 4 and, of course, a 5. There are also some constraints on the sort of operations one can do over copies of the dataset, for example, out-of-plane rotations cannot be handled easily like the example in Figure 2.10 which shows a face rotating out-of-plane.

Figure 2.9: Rotations applied to a 5 digit from the MNIST dataset [9].

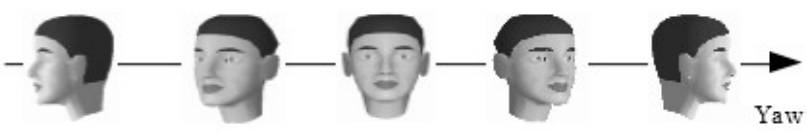


Figure 2.10: Out-of-plane rotation of a face along an arbitrary Yaw axis [30].

Noise robustness implies injecting noise into the model at some stage of it to make it more resistant to noise disturbance on the dataset or during the learning of the task and has been partially covered in the dataset augmentation portion of this section. The introduced methodology of injecting noise into copies of samples from the dataset covers but a small amount of noise robustness techniques.

Noise can be added to model parameters or a network's weights so that it becomes more resilient to such perturbations and manages to perform well by reducing the impact of ambiguous decision thresholds or pushing the parameter values it sets to values that, when perturbed, result in minor weight adjustments instead of introducing a significant change.

Injecting noise into a deep learning model's hidden layers themselves can lead to massive improvements. There are several ways to go about this but the most relevant one for the context of this work is dropout, which entails generating a subnetwork by dropping a connection so that it has no impact on that training step, theoretically improving the detection of other patterns which might have been disregarded up to that point.

Dropout also carries with it the risk of ruining an entire step of the learning process if enough units are dropped so as to erase a path from the input to the output units. The possibilities of dropout are illustrated in Figure 2.11 by the simple network with two input units, x_1 and x_2, two hidden units, h_1 and h_2, and one output unit, y, and the corresponding ensemble of subnetworks generated by dropping different subsets of units.

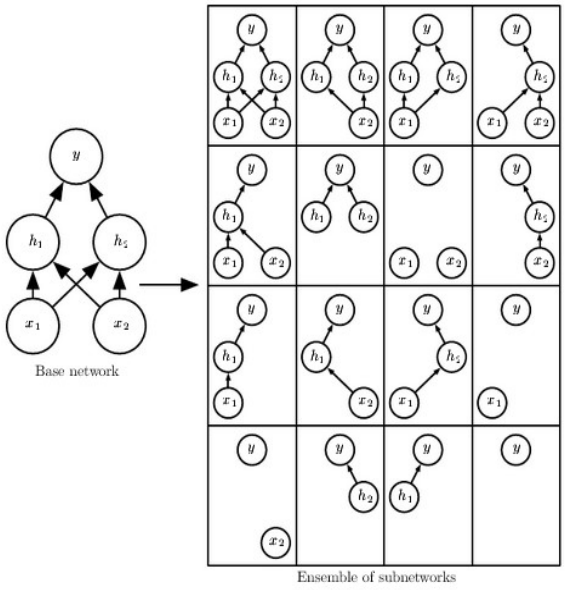


Figure 2.11: Subnetworks generated from a base network by dropping one or more subsets of units [9].

Adversarial training [17] is also a form of regularization by noise. The main difference lies in the fact that adversarial training requires selective noise injection so as to distort the model's interpretation of specific inputs by, for example, misclassifying an object. This ability to distort the original correct output stems from the excessive linearity of the neural network itself and can result in significant noise robustness if correctly used during the training and evaluation process. In Figure 2.12 this effect is

illustrated by distorting the original correct labelling of the image as a temple, with ninety-seven percent probability, to the mislabeling of a temple as an ostrich with ninety-eight percent probability by adding some well-conceived perturbations.

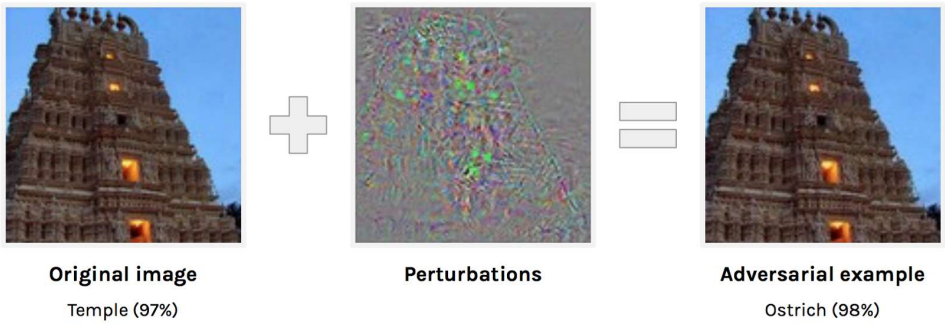


Figure 2.12: Adversarial example which leads to mislabeling [26].

Semi-supervised learning is a technique where both unlabeled, $P(x)$, and labeled samples, $P(x,y)$, are used to estimate $P(x|y)$ or predict y from x. In the deep learning community, it usually refers to learning a representation so that samples from the same class have similar representations.

Sparse representations is a technique which involves applying penalties on the model's parameters or the unit's activation functions. L1 penalties tend to pull the values of parameters towards the origin, a process designated as sparse parametrization. Representational sparsity differs from this due to the fact that the model's representation of the underlying data disregards or greatly decreases the importance of certain values. To illustrate this difference Figure 2.13 showcases a sparsely parametrized linear regression model while Figure 2.14 shows a linear regression with a sparse representation, $\boldsymbol{h}$, of the data $\boldsymbol{x}$.

$$\begin{bmatrix} 18 \\ 5 \\ 15 \\ -9 \\ -3 \end{bmatrix} = \begin{bmatrix} 4 & 0 & 0 & -2 & 0 & 0 \\ 0 & 0 & -1 & 0 & 3 & 0 \\ 0 & 5 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & -1 & 0 & -4 \\ 1 & 0 & 0 & 0 & -5 & 0 \end{bmatrix} \begin{bmatrix} 2 \\ 3 \\ -2 \\ -5 \\ 1 \\ 4 \end{bmatrix}$$

$$\boldsymbol{y} \in \mathbb{R}^m \qquad \boldsymbol{A} \in \mathbb{R}^{m \times n} \qquad \boldsymbol{x} \in \mathbb{R}^n$$

Figure 2.13: Sparsely parametrized linear regression model in matrix form [9].

$$\begin{bmatrix} -14 \\ 1 \\ 19 \\ 2 \\ 23 \end{bmatrix} = \begin{bmatrix} 3 & -1 & 2 & -5 & 4 & 1 \\ 4 & 2 & -3 & -1 & 1 & 3 \\ -1 & 5 & 4 & 2 & -3 & -2 \\ 3 & 1 & 2 & -3 & 0 & -3 \\ -5 & 4 & -2 & 2 & -5 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ 2 \\ 0 \\ 0 \\ -3 \\ 0 \end{bmatrix}$$

$$\boldsymbol{y} \in \mathbb{R}^m \qquad \boldsymbol{B} \in \mathbb{R}^{m \times n} \qquad \boldsymbol{h} \in \mathbb{R}^n$$

Figure 2.14: Linear regression with representation $\boldsymbol{h}$ of data $\boldsymbol{x}$ in matrix form [9].

Multitask learning is a way to improve generalization by pooling samples outputted by several tasks. Should the tasks share some statistical similarity among themselves then sharing parts of the model between tasks acts like training with relatively different samples in the sense that the model becomes constrained towards parameter values that generalize well. The main challenge lies in how to segment parameters into two categories, task-specific parameters, which cannot be shared among tasks, and generic parameters which do benefit from the pooled data.

Early stop is a type of regularization applied to models with adequate capacity to model the underlying data but prone to overfit. When you plot the model's training and validation set error over time or over successive passes of the same training set, epochs, some models will reach a point where the training error steadily decreases but the validation error stops improving and at some point will start increasing. Early stop is therefore used to avoid overfitting which, assuming the validation set accurately depicts the model's behavior with the test set, will lead to a model that generalizes well for its task.

Bagging or bootstrap aggregation is a form of regularization where several different models are used to solve the same tasks and then vote on its output. The outcome of said tasks might therefore benefit from the fact that different models will, most likely, not err in the same portions of the test set and improve because of it. This sort of regularization performs, on average, as well as any of the members of the ensemble although having to choose relevant models to compete for the solutions and the considerable increase in computational effort that comes from the additional models are steep disadvantages. Outside of regularization this method is used in boosting a model, where new neural networks or layers are added to the original effectively boosting the model's capacity.

There are many other forms of regularization but the ones covered in this section are within, and to some extent exceed, the scope of the work done in Chapter 4.

Chapter 3

Deep Learning Pipeline

3.1 Introduction

This chapter aims to add to the scientific and technical background covered in Chapter 2 the knowledge on how to build a deep learning pipeline and the practical implementation tools available in Tensorflow API for this purpose. It also delves into an explanation of fully-convolutional neural networks, which is relevant for the work done in Chapter 4. Section 3.2 delves into the major blocks of a deep learning algorithm, preprocessing, training, evaluation, testing and postprocessing and Section 3.3 provides a brief introduction to the aforementioned API and some of the operations available that allow for the construction of a deep learning pipeline in a practical setting. Finally, Section 3.4 covers the FCNN topic.

3.2 Deep Learning

As previously mentioned in section 2.2 a feedforward deep network, also referred to as a multilayer perceptron (MLP), is a mathematical function mapping a set of input values to another of output values. This function is obtained through the composition of several simpler functions, with each distinct application resulting in a new representation of that function's input. A convolutional neural network is in turn a particular type of feedforward neural network which specializes in processing data that has a known grid-like topology.

As Chapter 2 focused particularly on foundational concepts for machine learning and deep learning as a whole, Chapter 3 focuses on the way these theoretical concepts and definitions tie with a practical application, or in other words, the building blocks of a deep learning algorithm. As such in this section an overview of the practical pipeline of an algorithm designed to tackle the binary classification task is provided. A portion of this section draws inspiration from chapter 11 of Deep Learning Book [9].

3.2.1 Data Preprocessing

Preprocessing data is usually the first step of the pipeline of a neural network. It entails the loading, reading or otherwise mapping, as well as eventual sorting and potential augmentation of the original data. In the first part of this process some important yet often overlooked decisions are made pertaining feature engineering.

Feature engineering is the process of designing the inputs of the deep learning model. It includes decisions such as whether to preserve the original dimensions of the data or alter its shape and whether to compress or modify the original data or attempt to store as much of the original untouched. A major portion of preprocessing consists of feature engineering, but it also entails decisions

on whether to use all of the data for training or split it into a validation and training sets, for supervised learning, and the corresponding proportions of each set. Each of these, individually, might not be that relevant to some models but often translate in faster training, improved convergence and even increased generality of the model.

Within the feature engineering domain, the first of the aforementioned decisions entails, for example, the number of channels of an image to keep. Three are regularly used, often translate the luminance and both chrominances or RGB levels, but in some cases, such as that of medical images, the use of more channels may be desirable. The user might also choose to use extra channels to store masks with relevant visual information, often the targets of the examples available for training. It also potentially includes cropping or resizing of the originals and how to perform such actions in order to preserve as much information with relevance to the task at hand as possible.

The comprehension beyond the term compression as used in this context can have a plurality of interpretations ranging from computer vision and natural language processing, where the user decides whether to store the data in its raw format or encode it with a lossy algorithm, so that the modelling stage doesn't include so much irrelevant information, to time series analysis where hash encoding, for example, is done for features that highly correlate and don't add much to the model individually and can be better used as more relevant, albeit general, group of placeholder features.

Modifications of the original data usually translate to, for instance, contrast normalization, scaling of individual pixel values and border erosion between adjacent objects for images, but might also include dropping samples all together due to their incorrect labelling and corruption, such as missing or highly unlikely values. Outside the computer vision branch this step often entails replacing or dropping missing values or categorizing and encoding features on a dataset.

After the previously mentioned steps it is necessary to define whether the model will have an evaluation stage which, empirically, leads to improved scores for virtually all sorts of metrics and applications across the board or not. It is also necessary to define whether there should be a split of data between training and validation sets or they should share the same data pool and, finally, what should the method on which the aforementioned sets will be used to evaluate be. Data augmentation, (Section 2.4.2) is a very versatile tool to expand a model's capacity. It consists of applying functions to copies of the original dataset to artificially augment, hence the name, the size of it. In the field of computer vision there have been several major leaps regarding this technique which proved to empirically lead to an improvement across all metrics used [1], [4], [7], [8]. It is however a potential source of deterioration of a model's performance if used incorrectly, as it can easily lead to overfitting on one end or "unlearning" of useful and desired patterns on the other. Some of the major techniques used are randomly overlapping an image and its mask with one or several others, introducing pseudo randomized or purposely engineered instances, the adversarial regularization mentioned in section 2.4.2, noise to copies of existing samples, inverting an image's colors, assembling new samples from random crops of others and their masks and translating, although not for CNNs, or rotating existing samples.

For the work developed in the context of this thesis only some of these techniques are used for this step, namely, cropping, artificial sample enlargement by mirroring the original or padding, border

erosion between adjacent objects and irregular borders, normalization of pixel values and binary conversion applied to the label masks.

Another aspect that requires consideration is how the data should be stored. Storing the whole dataset, or batch, in-memory is not feasible for large, or memory heavy datasets, but iterating through them and, for each sample, applying the necessary regularization and augmentation on the fly also consumes computational resources.

In practice tradeoffs are usually made such as initially storing the labels of the samples, performing all possible and desirable augmentations over the label samples and then iterating through the dataset's images while applying whatever regularization was applied to the corresponding labels. Augmentation, for example, can complicate the process since some forms of it, like generating adversarial examples or general noise addition, might require accessing several samples simultaneously. A popular tradeoff is randomly storing in-memory a set of randomly or purposely chosen samples, a minibatch, and then repeating the process until the whole dataset has been fed to the model.

This thesis is limited to in-memory storage which severely limits the potential size of the input samples. It should be noted that the underlying code is done with distributed computations in mind, and this in-memory storage was thought out as to allow all machines to access a common cache.

3.2.2 Training and Evaluation

Training and evaluation cycles are typically the deep learning pipeline's bottleneck, regardless of whether computations are distributed or non-distributed. It is at this stage that the model effectively learns how to tackle the task at hand by finding and modelling the patterns it is being taught through a combination of operations stacked by layers, the initialization of each of the units' weights and biases or per layer initializations, regularization such as batch normalization or dropout, the intended objective function and the performance metrics in play as well as the number of epochs (Section 2.4.2) used.

To create a model for a neural network algorithm it is first necessary to define its input and output. In the computer vision domain, inputs typically consist of ordered vectors that contain each individual pixel or voxel value together with channel vectors and a vector of the target masks. In this thesis the inputs consist of tensors with the aforementioned pixel or voxel information in a matrix-like format, the corresponding channels and the labelled binary masks of the targets.

The model output has its own particularities since it is an intermediate step of the whole algorithm. Model outputs in this domain usually consist of logits which, unlike their mathematical designation, that states that a logit is a logistic function, are typically a vector of non-normalized predictions, raw predictions, done by the model. The raw prediction method is then processed through a function, usually softmax for multi-class classification tasks or sigmoid cross-entropy with logits for binary classification tasks. In this thesis' context the output of the model is a tensor that contains the probability distribution over each individual pixel of how likely it is to belong to the target class. This output is obtained via use of the sigmoid function which is related to the logistic function in the sense that sigmoid is a specific form of the logistic function. The model's output logits are therefore converted to a real number between zero and one.

After establishing the network's inputs and outputs the next step is to define the objective or loss function which the model will try to optimize and the process through which this optimization will be monitored and assessed. The loss function is the objective function which the model intends to minimize as mentioned in Section 2.4.1. In this work the chosen main loss function is the sigmoid cross-entropy loss, which acts as a measure of model performance by comparing predicted probabilities with the correct labels. Recently, a loss function from the 1940s, specifically designed for the binary classification task, has found a new use in the computer vision domain, either with volumetric or two-dimensional images and has gained some traction, the dice coefficient loss [7]. It is a measure designed to assess the similarity between samples in the following manner, consider two masks, X and Y, their cardinalities, $|X|$ and $|Y|$, and the intersection of these masks, the dice coefficient, using the Sorensen formulation, can be computed by

$$Dcoef = \frac{2|X \cap Y|}{|X| + |Y|} \qquad (3.18)$$

Alternatively, taking the correctly classified pixels that belong to the class of interest, true positives, and those that don't, true negatives and the incorrectly classified pixels that were considered to belong to the class of interest, false positives, and those that weren't but should have been, false negatives, into account, then the dice coefficient can also be formulated in the following manner

$$Dcoef = \frac{2TP}{2TP + FP + FN} \qquad (3.19)$$

In Chapter 4 some model variants also use a specific formulation of the Dice coefficient to assess the loss of the model in conjunction with sigmoid cross-entropy loss. Only one of the variants uses Dice as its only loss function.

The optimization method beyond the optimization of the objective function, mentioned in Section 2.4.1, is a variant of the gradient descent, namely the adaptive moment estimation optimizer, Adam, as designated by its creators [18], which in turn is itself a variant of mini batch stochastic gradient descent. To best understand Adam some context of its predecessors must be provided, [12] for a detailed overview. As such brief introductions to SGD with momentum, AdaGrad and Adadelta are provided below.

SGD on its own tends to oscillate in areas where surfaces curve much more in a specific dimension than the others, a situation typically found around local minima, which, at best, introduces a great delay in convergence or, at worst, leads to no convergence at all due the desirable decreasing learning rate mentioned in Section 2.4.1. Momentum [19], is a method which consists of adding a fraction, scaled by $\lambda \in \mathbb{R}$ with $0 < \lambda < 1$, of the update vector of the preceding step to the update vector of the current step to accelerate convergence in a given direction. So, if for the update of mini-batch gradient descent, with n examples, training examples x and labels y, we have

$$\theta = \theta - \eta \cdot \nabla_{\theta} J(\theta; x^{(i:i+n)}; y^{(i:i+n)}), \qquad (3.20)$$

where $\nabla_{\theta} J(\theta)$ is the gradient of the objective function parametrized by θ, η is the learning rate and the set of training examples and labels, $x^{(i:i+n)}$ and $y^{(i:i+n)}$, respectively, than the update for mini-batch gradient descent with momentum is given by

$$v_t = \lambda v_{t-1} - \eta \cdot \nabla_{\theta} J(\theta), \qquad (3.21)$$

where λ is the momentum term and where v_t and v_{t-1} are the current update vector and the update vector of the previous time step, respectively. The momentum term has some additional drawbacks since it tends to follow the "slope", making abrupt changes difficult even when these are necessary for successful global minima convergence.

Adagrad, [20], is a stochastic gradient descent optimization variant which associates different learning rates to different parameters based on their feature frequency leading to smaller updates for frequent features and larger updates for less frequent ones, where mini-batch gradient descent and the variant with momentum update parameters using a single learning rate. For each parameter, θ_i, at every time step t, Adagrad's per-parameter update, regarding the actual implementation, is given by

$$\theta_{t+1,i} = \theta_{t,i} - \eta \cdot \nabla_{\theta_t} J(\theta_{t,i}), \qquad (3.22)$$

with $g_{t,i} = \nabla_{\theta_t} J(\theta_{t,i})$, as the partial derivative of the objective function in function of the parameter θ_i at time step t. The update rule is therefore given by,

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \varepsilon}} \cdot g_{t,i}, \qquad (3.23)$$

where $G_t \in \mathbb{R}^{d \times d}$ is a diagonal matrix where each element, $_{ii}$, is the sum of the squares of the gradients with respect to θ_i up to time step t and $\varepsilon \in \mathbb{R}$ is an offset term to avoid division by zero. By performing an element-wise matrix-vector multiplication between $G_t \in g_t$ we obtain the vectorized form of (3.22), which is then the full Adagrad optimizer expression. Adagrad's main drawback lies in the accumulation of squared gradients in the denominator as it keeps growing during training eventually leading to a very small learning rate, effectively stagnating the learning process.

Adadelta, [11], is an extension of Adagrad. The former intends to solve the issue of the vanishing learning rate of the latter by replacing the accumulation of all the past squared gradients with a window

with the accumulation of only a portion of the past squared gradients. Practice wise instead of a fixed window, a decaying average of all past squared gradients is used, mitigating the impact of older squared gradients compared to new ones. Consider a running average given by

$$E[g^2]_t = \lambda E[g^2]_{t-1} + (1-\lambda) g_t^2, \qquad (3.24)$$

where $\lambda \in \mathbb{R}$ and $0 < \lambda < 1$ is a parameter which defines the importance of the previous squared gradients. To accommodate these changes, we refactor the previously presented SGD update in terms of the parameter update vector, $\Delta\theta_t$, as

$$\begin{aligned} \Delta\theta_t &= -\eta \cdot g_{t,i} \\ \theta_{t+1} &= \theta_t + \Delta\theta_t, \end{aligned} \qquad (3.25)$$

and, by replacing $G_{t,ii}$ with the decaying average over past squared gradients, $E[g^2]_t$, we obtain the corresponding parameter update vector

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \varepsilon}} g_t. \qquad (3.26)$$

With some more manipulation since the denominator is the root mean squared error criterion of the gradient and with an approximation up to the previous time step's gradient, the practical Adadelta implementations bypass the need to define a learning rate at all and is given by

$$\begin{aligned} \Delta\theta_t &= -\frac{\sqrt{E[\Delta\theta^2]_{t-1} + \varepsilon}}{\sqrt{E[g^2]_t + \varepsilon}} g_t. \\ \theta_{t+1} &= \theta_t + \Delta\theta_t \end{aligned} \qquad (3.27)$$

Adam was designed to enhance Adadelta and other similar algorithms [12]. It does so by, like momentum, storing an exponentially decaying average of the past gradients and, like Adadelta, storing an exponentially decaying average of past squared gradients. These can be interpreted as estimates of the second moment, v_t and the first moment, m_t, of the gradients. These vectors are usually initialized as vectors of zeros, which, for small decay rates, β_1 and β_2, and the initial time steps introduces a bias towards zero value. To counteract this effect the authors proposed first-moment and second-moment bias corrected estimates. The general form of v_t and m_t is therefore

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \qquad (3.28)$$

which in turn yields the Adam update rule

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{v_t} + \varepsilon} m_t \qquad (3.29)$$

Regarding the error metrics used throughout this work we have the mean intersection over union, IoU, which is the main metric of interest due to being the one proposed by the nuclei detection challenge as the target metric and due to its appropriate assessment of correctly predicted masks compared to the ground-truths covered in Chapter 4, but the accuracy, recall and precision metrics are also used in tandem with the mean IoU. Intersection over union is a metric that, at its core, entails comparing the intersection of two samples over their union. Consider a predictions mask B1 and the original mask B2, standard intersection over union computation can proceed as illustrated in Figure 3.15.

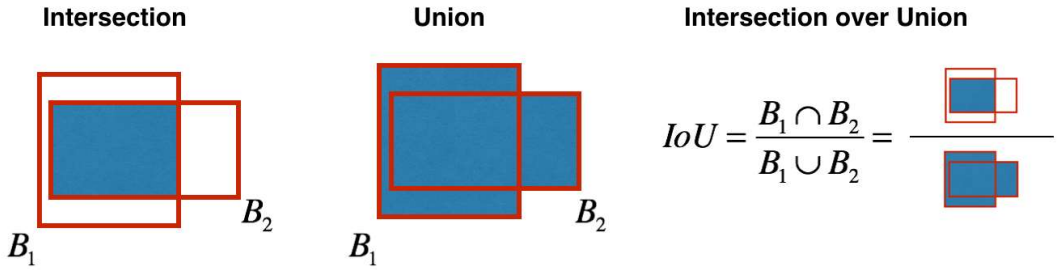


Figure 3.15: Visual understanding of intersection over union for two-dimensional inputs [31].

Accuracy generally measures the number of correct predictions over the total predictions made, but on binary classification tasks it can also be measured through a combination of true positives and negatives and false positives and negatives in the following manner

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN} \qquad (3.30)$$

Accuracy is not as relevant to the task at hand since it will take into account the correct classification of all pixels which could, asymptotically, lead to a very high accuracy for the classification of say a 512 by 512 pixels image with 10 pixels of the class of interest as having no pixels belonging to this aforementioned class of interest. This example would lead to an approximately 99.996185% accurate prediction.

Recall measures the portion of the pixels correctly classified as belonging to the class of interest over all the pixels that belong to that class or, using the true positives and false negatives convention,

$$precision = \frac{TP}{TP + FP} \quad (3.31)$$

Recall on its own also has drawbacks, since labelling every pixel as one that belongs to the class of interest will lead to a perfect recall.

Precision measures the portion of the pixels correctly classified as belonging to the class of interest over all the pixels classified as belonging to that class or, using the true positives and false positives convention,

$$recall = \frac{TP}{TP + FN} \quad (3.32)$$

Like recall, precision on its own has some major drawbacks since perfect precision can be obtained by correctly classifying a single pixel as belonging to the class of interest and no others as long as no pixel is incorrectly classified as belonging to that class. In practice precision and recall work together to help assess a model's performance. With the previous topics all defined, it becomes necessary to formulate the architecture of the neural network, namely its number of layers, the number of units, or neurons, in each layer, the functions pertaining to the units of each particular layer and the way information is passed to adjacent layers and, particularly, to which units. The specific architectures of the networks in use are detailed on Chapter 4 so the remainder of this section will focus on activation and initialization as well as the evaluation process.

Activation functions are used to map the output of a unit to a specific range of values and to introduce non-linearity to a model. Throughout this thesis only the original and a variant of the rectified linear unit, ReLu, are used as activation functions. A ReLu [21] is a real function which returns zero for negative values, $f(x)=0$, $x<0$, and that same value, $f(x)=x$, if the output is zero or a positive value, $x\geq 0$, as shown in Figure 3.16.

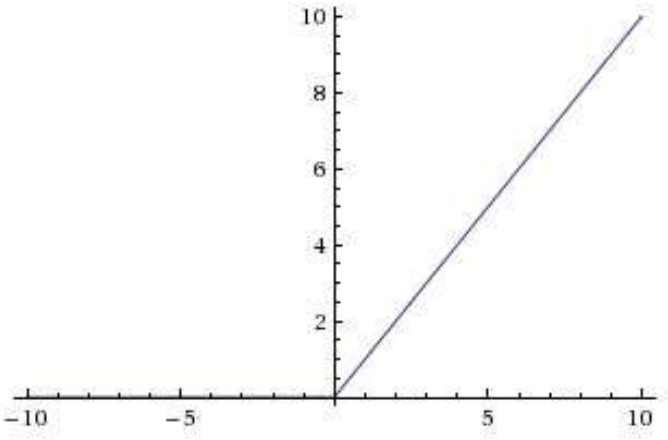


Figure 3.16: ReLu visual representation [Matlab].

The variant used in this thesis for $x \geq 0$ behaves the same way as the base ReLu function but for x<0 the output is instead given by $f(x) = mx$, where 0< m < 1 and m stands for the slope. These inherent properties are what inspired the designation of this variant, leaky-ReLu, one of the most used activation functions in the neural network domain, as seen in Figure 3.17. This is due to its effectiveness in avoiding the vanishing gradient function by assuring the unit only outputs zero for zero inputs.

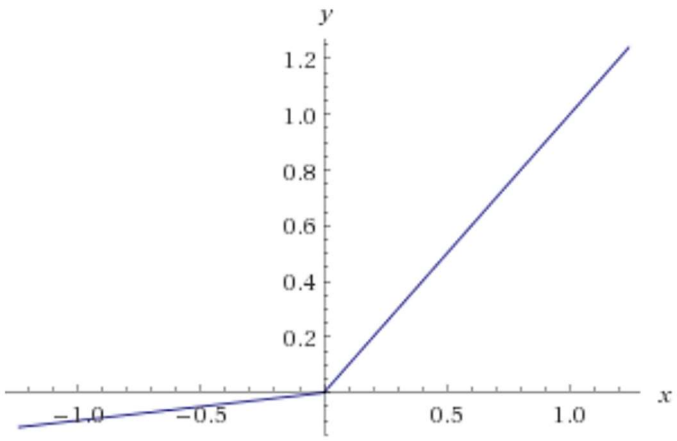


Figure 3.17: Leaky-ReLu visual representation for $m = 0,1$ [Matlab].

The next step before engaging the training cycle is to initialize the weights between connections of the network and the per unit or per layer biases. Towards this end there are several proposed strategies.

For weights initialization two of them are particularly relevant to this thesis' context, the Xavier initialization family, proposed by Xavier Glorot and Yoshua Bengio [22], and He initialization, proposed by Kaiming He, Xiangyu Zhang, Shaoqing Ren and Jian Sun [23]. Both these techniques rely on attempting to set the layer outputs' variance to be the same as the inputs, but while the former was developed with the logistic sigmoid activation function in mind the latter is a more generalized version of it, with ReLu activation at its core.

Bias initialization relies either on knowledge of the data generating source, as mentioned in Section 2.3.2, or on techniques tried and tested empirically but lacking proper theoretic foundations. As such batch normalization is typically used to bypass the need to define a specific bias at all. Typical initialization values when there is no prior knowledge involved are zero, one or negative one, each with their pros and cons in the community's views.

Training thus proceeds by iterating through the dataset's samples, steps, often processing the same sample multiple times, epochs, in-order to extract all relevant features from it. This process is also a recurrent source of generalization error since unbalanced datasets, that is those that have many samples with similar features which are themselves only a subset of the features of interest as a whole, can influence the weights attributed to each unit, in such a way that the network becomes biased towards these recurrent patterns.

After a set amount of training steps, set thresholds or other user defined criteria, evaluation of the current algorithm's performance takes place by processing the validation set, without the label information. Then these labels are used to check the metrics defined and compute the validation set loss. Early stop regularization, explained in Section 2.4.2, usually kicks in when one or more error metrics reach an error threshold defined by the designer of the deep learning algorithm. In the work developed for this thesis the early stop regularization technique is also used to keep the model from overfitting. It is done, manually, by checking the model's progression on every evaluation step, assessing the error metrics the current loss, and then choosing a checkpoint that strikes a good balance between these, with emphasis on loss and mean IoU.

3.2.3 Testing and Postprocessing

Testing a deep learning algorithm's performance entails the same steps defined to evaluate a model, as described in Section 3.2.2, barring the inclusion of the labels' information. It is essentially the final step of a model and, for all intents and purposes, works as a prediction of the desired output, which is typically considered to be the final step of a deep learning algorithm. To draw conclusions from these predictions these outputs are usually presented to domain specialists.

Model outputs do not necessarily coincide with the final stage of a deep learning algorithm, as these outputs can often be refined based on pre-existing knowledge of the target features, might be a part of an ensemble of different models to be factored on a final output and, most importantly, might be outputs pertaining to the task of learning something related but not necessarily equal to the task at hand. The act of further working over a model's outputs is collectively known as postprocessing.

When an ensemble of models is used to tackle a task, there might be dependence between competing models, either by parameter sharing or by having models learn its own parameters from those of another model, or the competing models can be fully independent. In either scenario some criteria, such as a weighted average, must be taken into account to factor each model's output contributions.

Postprocessing based on domain knowledge often implies removing or correcting outliers from the model's output or simply disregarding certain predictions. In a computer vision binary classification task this often means disregarding pixels from outside known regions of interest or filling in adjacent pixels with logical true values. The latter is usually done for situations such as that of the work done in this thesis, where objects in an image are known not to have internal gaps.

Another frequent postprocessing scenario lies with training a model on transformed data and then reverting those transformations after the model outputs its final results. A similar scenario occurs for models that train with the original data but different error metrics or, most frequently, generalized metrics that prove not to be sufficiently tailored to the task at hand. In either case the output must then be further worked upon to match the task we intend to solve.

Finally, postprocessing is most commonly used to encode a model's outputs so as to be processed by another algorithm or assessed, such as run length encoding a binary output mask to

reduce the memory footprint of the findings by mapping pixels that are determined to belong to the target class.

3.3 Fully Convolutional Neural Networks

Now that the basic steps beyond the deep learning pipeline have been covered it is possible to discuss the implementation of Convolutional Neural Networks in their practical sense, particularly a type of CNN which holds interesting properties to Chapter 4's task. As such a brief explanation of the practical side of a CNN is provided followed by an analysis of this particular type of CNN architecture, the FCNN.

3.3.1 CNN Implementation

As mentioned in Section 2.2 a convolutional neural network is a network specialized in processing data that has a known grid-like topology and scalable to large sizes and depths. This type of network has played a major role throughout deep learning history and represents a landmark on successfully applying insights of the human brain to machine learning applications.

Convolutional neural networks are still a very competitive choice for two-dimensional image topology, [3], [4] and [5], and the solutions based on CNNs keep being updated with major breakthroughs occurring sometimes months apart. The focus of this section, as well as the work done in Chapter 4, lies in this particular type of input. A CNN model architecture typically consists of several stacked convolutional and pooling layers followed by a fully connected layer.

In a convolutional layer the parameters are a set of learnable filters which are spatially compact, meaning the height and width pixel-wise is very small compared to the input sample such as 5 by 5 pixel for the filters for 126 by 126 pixel input images as an example, and extend throughout the full depth of the input image, for example 3 for the color channels in RGB images. For each convolutional layer, during the forward pass of the model, each filter, or kernel, convolves across the width and height of the input computing the dot product, effectively functioning as a two-dimensional activation map with the responses of each particular filter at every spatial position, as exemplified in Figure 3.18 where the first entry of a feature map is computed for an input and a kernel solely composed of 0 and 1 entries. Unlike the provided example both the kernel and the input are volumetric, but the depth is preserved throughout the network. The size of these kernels determines the receptive field, which is the number of connections between a neuron and its input which translates to the sparse connectivity property, discussed in Sections 2.2.2 and 2.4.2.

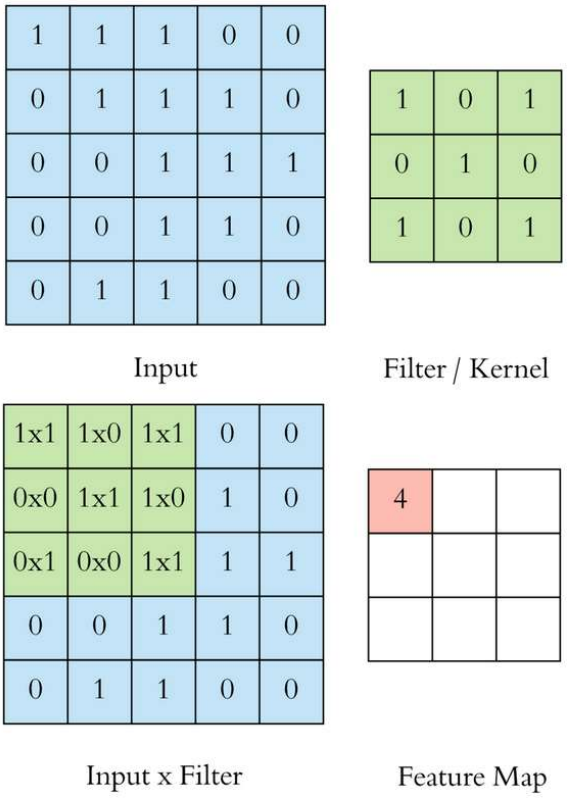


Figure 3.18: Representation of the first entry of a feature map resulting from a convolution between the input and the kernel [29].

The output neurons are determined by a combination of stride, depth and padding. Stride is an hyperparameter which determines how much the kernel window slides between convolutions, typically either 2, to spatially reduce the volume of the output, or 1, to preserve the dimensions.

Depth is another hyperparameter of the output volume which states the number of kernels intended. Here a tradeoff is required, as a lack of kernels might lead to the loss of patterns of interest but an excessive number of kernels will increase the computational effort disproportionally compared to the network's overall improvement or even introduce a tendency towards certain features due to the sheer amount of kernels detecting them.

Finally, zero-padding is an optional hyperparameter which entails controlling the output's spatial dimensions by introducing zeroes, theoretically allowing an increase in spatial dimensions without affecting the activation of the kernels, which is particularly useful to ensure the borders of an input sample are not lost due to dimensions mismatch between the kernels and the input image. Padding does not necessarily have to be zero-padding, but other types of padding require significantly more preparation and case study, making them a tailor fit solution at best.

Generally, these hyperparameters relate to each other and the input and output values like

$$O = \frac{(W - F + 2P)}{S + 1} \qquad (3.33)$$

where O is the spatial output volume size, W is the spatial input volume size, F the receptive field of the neuron, S the stride and P the amount of zero padding. This way it is possible to set fixed hyperparameters and the desired O and compute the amount of padding or the receptive field required.

In this thesis's networks a single depth slice's neurons share the same weight vector so during the forward pass of the network, the convolutional layer effectively computes a convolution per depth slice between the neuron's weights with the input volume, which is why these depth slice weights are designated kernels.

A pooling layer is a layer where a pooling function is gradually applied to the whole output of the preceding layer in order to replace output values with a summary statistic of their neighboring output values, that is, reduce the spatial size of each depth slice while retaining the depth dimension of the input. Although there are several pooling methods the most used is max pooling, albeit for empirical reasons not mathematical ones, which entails replacing a rectangular area of the input values by the maximum value found in that area. In essence pooling introduces invariance to small translations of the input and reduces the number of parameters of the network. The typical max pooling layer uses 2 by 2 filters with stride 2, as higher spatial dimensions tend to remove too much information, and the same depth dimension as the input as previously mentioned. Figure 3.19(a) showcases potential drawbacks of max pooling and average pooling for different inputs, supporting the argument of its source [24] which is to use mixed pooling and Figure 3.19(b) showcases the corresponding pooling operation with 2 by 2 filters with stride 2.

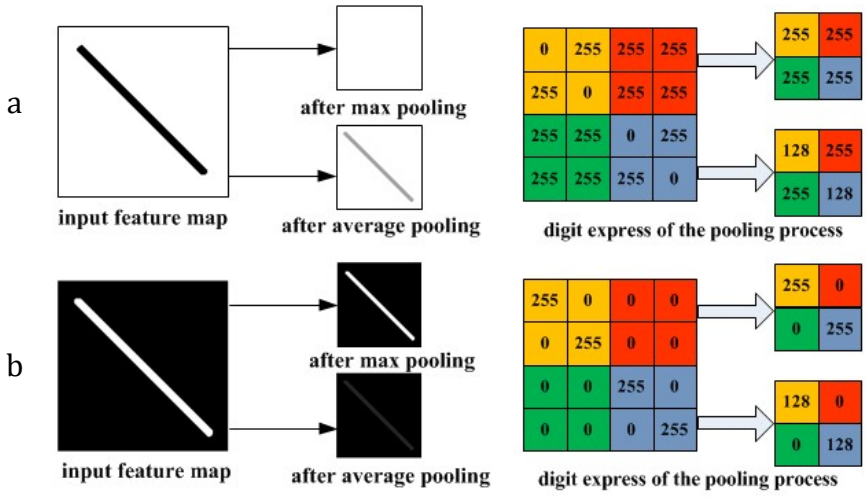


Figure 3.19: Representation of the average and max pooling operations. (a) Visual representation. (b) Feature maps drawn from convolving the inputs using a 2 by 2 kernel with stride 2 [24].

The final layer in a typical convolutional neural network is the fully connected layer which encompasses all the parameters learned by the layer before it due its connection to each activation of the previous layer. This layer is used as a way to combine the different high-level features extracted by the convolutional layers.

3.3.2 FCNN Implementation

A fully-convolutional neural network [6], differs from the typical CNN due to having no fully connected layer at its end, instead relying solely on convolutional, pooling and activation functions. The authors of the original paper proposed an architecture that retains the traditional initial layers of a CNN but replace the fully connected output layer with a deconvolutional layer, [3]. A deconvolutional layer is essentially an up sampling layer that can be obtained through a convolutional layer with a fractional stride for its convolution kernels. This can be done by taking the desired upscaling factor, f, and computing the stride of the convolution as $\frac{1}{f}$, assuming f is an integer.

The first of the authors' proposed architectures, Figure 3.20, is the FCN-32s, which, in order, consists of two sequential sets of two convolutional layers followed by a pooling layer, three sequential sets of three convolutional layers followed by a pooling layer and two convolutional layers, the former corresponding to 32 pixel strides, followed by a deconvolution layer to turn predictions to pixel values.

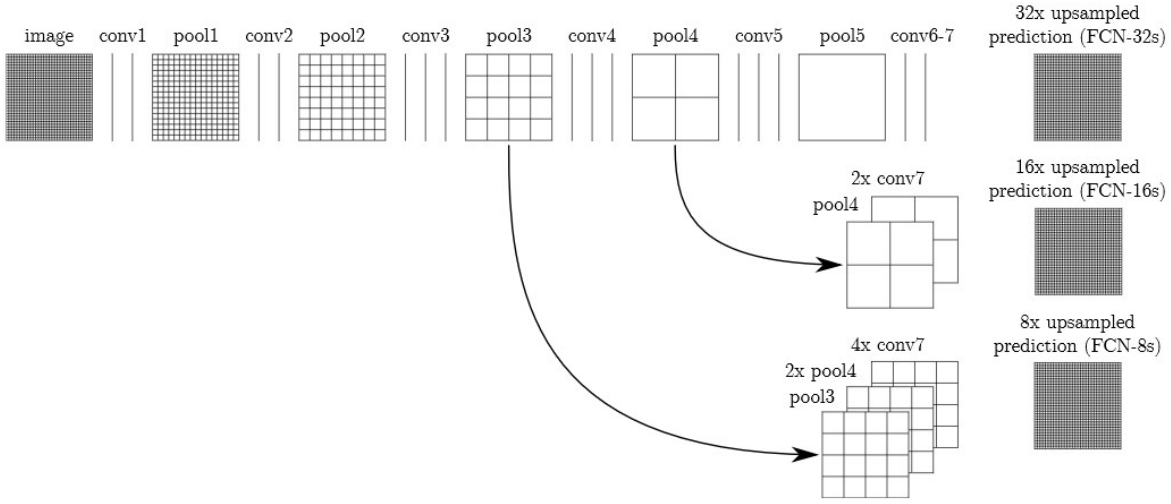


Figure 3.20: A visual representation of FCN-32s, FCN-16s and FCN-8s [6].

This network's outputs are coarse so the authors propose working with skips, that is using the output of pooling layer 4, pool4, filtered through a 1 by 1 convolution with 16-pixel strides, in conjunction with the output of convolutional layer 7 up sampled twice to obtain stride 16 predictions, which are in turn up sampled twice back to the image. This architecture is the FCN-16s, also in Figure 3.20, and it is learned from end-to-end and initialized by the parameters of FCN-32s. By emulating this logic even further the authors propose the final network stacked, FCN-8s, which allows even finer detail. The side-by-side comparison of these networks can be seen in Figure 3.21 which exposes the ground-truth and the predictions of each of these networks.

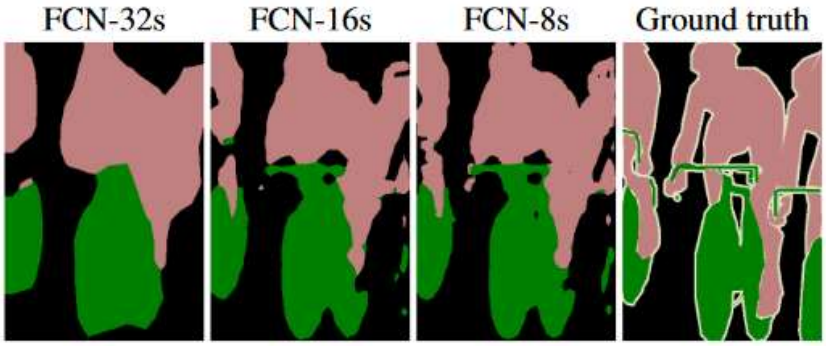


Figure 3.21: Predictions mask of FCN-32s, FCN-16s and FCN-8s side-by-side with the ground-truth mask [6].

Essentially the fully convolutional neural network [6] entails a compression path where feature maps extract patterns that contain spatial awareness due the way the convolution is processed. This path is then followed by a short decompression path that gathers that information in a final predictions mask. The authors went as far as adapting popular CNN architectures to their FCNN counterpart to discuss the smaller memory footprint and computational effort required by the latter in regard to the former.

3.4 Tensorflow API

One of the objectives of this thesis consists of breaching the gap between theoretical and practical knowledge on the aforementioned topics and those that follow, this entails providing a possible implementation to the theoretical knowledge exposed. Towards this end it is considered necessary to approach the chosen pythonian application programming interface in further detail, exposing some of its inner workings.

Tensorflow is a Python based open source library for high performance numerical computations with a strong emphasis on distributed computing, where a computational task is split into multiple smaller ones and deployed to several virtual or physical machines in parallel. It is largely used for machine learning and deep learning purposes and provides a very flexible API which can be used hand in hand with several other widely used Python libraries such as sklearn and pandas.

Tensorflow has two different approaches to computation. The main one is graph computation, detailed in Section 3.4.1, and all operations need to conform to this convention. The other is eager execution which, unlike graph computation, allows for immediate evaluation of operations, usually available for intermediate methods.

3.4.1 Computational Graph

Tensorflow uses computational graphs for data and operation representation. There are several conventions regarding how to structure a computational graph. In this section each node of the graph indicates a variable, which may be a tensor, scalar, matrix or other types of variable. Operations are functions of one or more variables that output a single variable, they are atomic in nature so more complex operations must be decomposed into several simpler operations. If a variable is obtained by applying an operation on another variable, then their respective nodes are connected by a directed edge starting from the latter and ending at the former. To illustrate this convention, consider the example of a backpropagation graph of a multilayer perceptron with one hidden layer detailed in chapter 6 of the reference material on deep learning [9] presented in Figure 3.22.

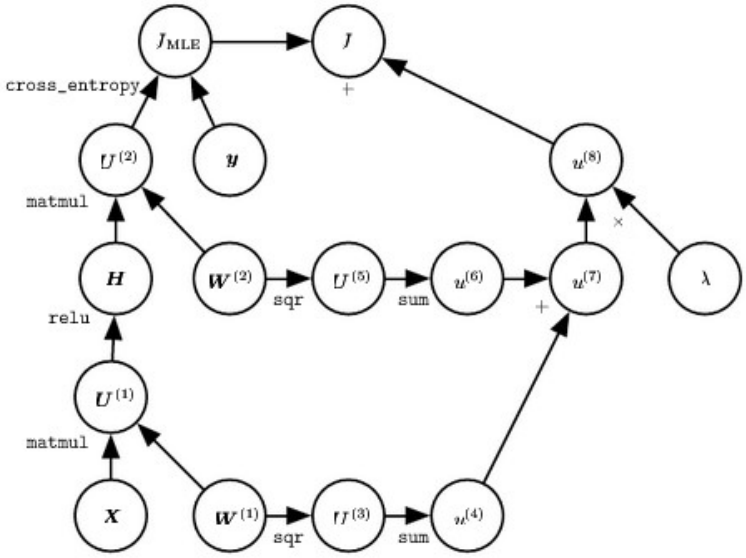


Figure 3.22: Computational graph for the backpropagation portion of a MLP with a single hidden layer [9].

Tensorflow's convention uses operations as the nodes and introduces significantly more complexity to graph logic due to the sheer size of the graph since it must encompass the whole algorithm. To compute the graph, internally tf.Graph, or a specific operation tf.Operation, tensor tf.Tensor or variable tf.Variable, the user needs to instance a tf.Session object to create a session, and, within that session, call tf.Session.run with a list of the operations to be made, so that Tensorflow can compute the subgraph corresponding to this set of operations. Tensorflow allows several graphs to exist simultaneously and does not impose a constraint on when to effectively run parts or the whole of each of these graphs.

Tensorflow provides several incorporated high-level APIs for the construction of these computational graphs meant to simplify the process of making a such a graph while at the same time limiting user agency. It also offers more low-level operations which are almost totally customizable but introduce significant complexity to this process.

Finally, Tensorflow has the option to store the computational graph and variable values between runs which facilitates the training of machine learning algorithms and allows for visualization of the graph itself and the progression of the variable values, such as parameters, metrics and loss through Tensorflow's own visualization tool, Tensorboard. To take advantage of the graphical component it is useful to label subgraphs in the graph and to that end it is necessary to use tf.variable_scope which allows the user to input the label associated with that portion of the computational graph.

3.4.2 Data Processing

The tf.data API provides several tools towards preprocessing data. It essentially provides two major abstractions to Tensorflow. The tf.data.Dataset represents a sequence of elements in which each element contains tensors, usually used to create a source, that is, constructing a dataset from tensors by applying the Dataset.from_tensor_slices operation. The tf.data.Iterator provides a way to extract elements from a dataset by, for example, associating an object to a particular Dataset object through a one-shot iterator and then using the Iterator.get_next operation to yield the next element of that dataset when executed.

Four iterators are available in tf.data API, however due to limitations in compatibility with the Estimator API only the most rudimentary one, the one-shot iterator, is used in Chapter 4's deep learning algorithms. One-shot iterators support iterating through a dataset once and require no explicit initialization, therefore being generated by the calling the make_one_shot_iterator operation on a dataset.

Another major functionality added by this API is the Dataset.map operation which facilitates the preprocessing of data by applying functions over all tensors of a dataset and then returning a new dataset composed of the transformed tensors. Dataset.map works with external python libraries' functions by invoking tf.py_func operation instead of a typical operation.

Operations provided by tf.data are not limited to preprocessing and can be found throughout the whole pipeline. Due to the scope of this work only a brief introduction to this API is presented.

3.4.3 Estimator API

The estimator API's tf.estimator instances are used mainly for training, evaluation and testing a model. Its main advantages are the ability to standardize operations that can be run both locally or distributed, the way its operations facilitate the implementation of complex models while retaining code readability and functionality, integration with automatic computational graph building, the ability to save computational graphs without specific user interaction and the structured way its operations allow for model construction separately from preprocessing. Some of the most relevant objects and operations from this API, used in the context of this thesis, are covered below.

Regarding basic setup there is the tf.estimator.ModeKeys object, which allows the user to split the training, TRAIN, validation, EVAL, and prediction, PREDICT, sets beforehand and effortlessly. It

works with tf.estimator.EstimatorSpec, which defines the full model to be run by an estimator. It uses mode keys to select which arguments to use for each portion of the model, TRAIN requires loss and train_op arguments, EVAL requires only the loss and PREDICT requires the predictions argument, although several other arguments are useful such as eval_metric_ops, which contains the metrics used to assess the model during validation.

The tf.estimator.RunConfig is used to define additional configurations to an estimator such as how and when to store checkpoints, which tasks to assign to which devices on a cluster for distributed computations, how many workers to assign to each task and other related arguments. It is related to the tf.estimator.Estimator, an object that wraps a model defined by a model function, which in turn is built through the composition of the outputs of EstimatorSpec and other auxiliary methods, a work directory and the RunConfig, to be used for training, testing and evaluation. To actually trigger the process of training and evaluating the trained estimator we use tf.estimator.train_and_evaluate.

Chapter 4

Practical Work and Results

4.1 Introduction

As stated in Chapter 1 this thesis objectives are twofold. The first objective is to attest to the performance of convolutional neural networks as efficient in terms of memory and computation while also a competitive solution to power a deep learning algorithm designed to tackle binary classification tasks. And the second goal is to provide an objective implementation with recourse to the Tensorflow API which connects several theoretical concepts to the empirical domain.

With the foundational machine learning and deep learning concepts defined in Chapter 2 and the connection between these concepts and practical implementations established in Chapter 3 the groundwork is set to advance towards Chapter 4 which covers the work done. Section 4.2 deals with the exposure of the two network architectures implemented, the U-Net and the V-Net, as proposed by their authors and the changes made to these for this thesis' work and Section 4.3 covers the application of these networks, or variants of them, to a challenge which consists on the segmentation of nuclei in two-dimensional images, Data Science Bowl 2018, and a similar challenge task that entails segmenting specific nerve structures in two-dimensional images, Ultrasound Nerve Segmentation.

4.2 U-Net and V-Net

In this section a comprehensive approach to the U-Net and V-Net model architectures is provided, including the specifics of their implementation, also taking into account their respective author's insights. The model architectures are based on the FCCN presented in Section 3.3.2.

Convolutional neural networks are typically used to tackle classification tasks, which in the scenario of two-dimensional image data input relevant to this thesis translates to outputting only a class or set of classes with the probabilities of each sample belonging to particular classes. In image processing for visual recognition tasks, particularly in the medical and biomedical field, localization is also required, that is, every single pixel or voxel should be classified.

Chapter 2 and Chapter 3 introduced the main task the deep learning algorithms implemented on this chapter intend to tackle as the binary classification task, but the general framework is to solve this task for every single pixel or voxel input and, from the multiple classifications, become able to implicitly segment desirable events in the input images, such as nuclei or nerve structures.

4.2.1 U-Net

The U-Net architecture was first introduced by Olaf Ronneberger, Philip Fischer and Thomas Brox in 2015 [8]. The author's objectives were twofold, retain the advantages of the FCNN while reducing its computational strain and build a model that could quickly learn specific patterns from a limited two-dimensional dataset, ranging from less than one hundred labelled samples to less than a thousand labelled samples, even taking augmentation into account. The network's architecture consists of a contracting path to capture context and a symmetric expanding path.

The U-Net shares the increased resolution of the output and localization properties of the FCNN and further enhances them through several changes to the original. Two of the major changes are the symmetrical expanding path that enables precise localization, and the large amount of feature channels that allow for context propagation to higher resolution layers.

The architecture's initial layers consist of a contracting path with 3 successive sets, steps of the contracting path between pooling layers, of 3x3, unpadded, convolution layers followed by a ReLu activation function and then sent to max pooling layer with operations with 2x2 pixels and stride 2, which were covered in Section 3.3.1. Each down sampling of the input effectively doubles the number of feature channels.

The symmetrical expanding path of the architecture corresponds to the latter layers of the network and begins with an up sampling operation paired with 2x2 convolutions to halve the number of feature channels, followed by a concatenation with the cropped feature map of the corresponding step of the contracting path, which in turn is followed by two successive 3x3 convolutions and ReLu activation pairs.

This layer layout is then repeated until the expanding path's depth reaches the contracting path's depth, in this case 3 times. Through the concatenation the U-Net, like the FCNN, effectively uses skipping to propagate context and ensure features are not lost between the contracting and expanding paths. In the output layer a 1x1 convolution is used to map each of the 64-component feature vector to the desired number of classes accordingly.

The output segmentation map has a different dimension from the input so, to be compared to the former, the latter has to be cropped first since all 23 convolutions are done without padding which leads to the loss of border pixels. The network imposes two-dimensional inputs with matching heights and widths pixel-wise to ensure a seamless tiling on the output segmentation map due to the 2x2 pooling operations in the contracting path. The full network schema for the lowest resolution of 32x32 pixels can be seen in Figure 4.23.

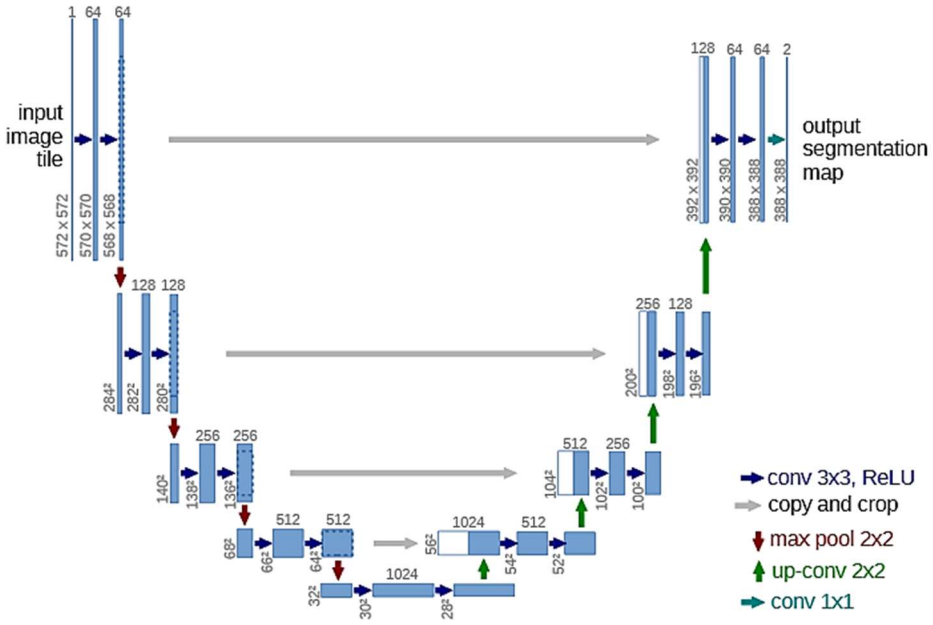


Figure 4.23: Graphical representation of the original U-Net architecture for 32 by 32 pixels lowest resolution [8].

In this thesis all convolutions are padded with zeros when there is a spatial dimension mismatch between inputs and kernels so that the initial input's spatial dimensions are preserved throughout the network even at the cost of some unintended regularization and increased computational effort.

The loss function used is the pixel-wise softmax (Section 3.2.2) over the final feature map in conjunction with the cross-entropy loss function. In this thesis' implementations, instead of the pixel-wise softmax, the sigmoid cross-entropy loss is used, sometimes in conjunction with a formulation of the dice coefficient. The rationale beyond this choice lies with the dataset disparity between the paper's data and the data available for both the challenges at hand which are, number-wise, about one order of magnitude greater than the that of the paper, which in turn might lead to an unbalanced dataset regarding features of interest. This is severely penalized by the dice coefficient when compared with the cross-entropy loss which, hopefully, mitigates the overall impact on the learning process. Another fact that impacted this choice is that the paper's data has multi-class objects for ground-truth mask while the challenges have binary ground-truth masks.

The authors propose pre-computing each ground-truth weights' map to facilitate class balancing and the distinction between bordering objects, in their case cells. Each weight map is computed as:

$$w(x) = w_c(x) + w_0 \cdot \exp\left(-\frac{\left(d_1(x) + d_2(x)\right)^2}{2\sigma^2}\right) \qquad (4.34)$$

where $w_c(x)$ is the weight map to balance class frequencies, d_1 the distance to the border of the nearest cell and d_2 the distance to the border of the second nearest cell. Through empirical testing the authors reached the values $w_0 = 10$ and sigma approximately equal to 5 pixels as the optimal values.

This thesis' models only benefit from the border distinction and not the multi-class balancing. The separation border is computed using morphological operations. In this work's context, binary erosion is used, that is, border pixels are discarded in their entirety if the there are separate foreground objects closeby.

To initialize the network's weights in such a way as to ensure that the feature maps have almost unit variance, the authors use He initialization. In this thesis, variance scaling is also used to initialize the weights of the network but there are instances of Xavier initialization as well. Both initializations are discussed in Section 3.3.2.

The U-Net uses a mini-batch SGD, defined in Section 2.4.1, with momentum variant with high momentum, 0.99, to optimize the network while the work presented in Section 4.3 uses the Adam optimizer, described in Section 3.2.2 along with SGD with momentum.

According to the authors, data augmentation using elastic deformations was one of the key factors that accounted for the achievement of a well-performant algorithm. This statement is widely accepted in the deep learning community since it is corroborated by empirical data.

The implementation of this network was made in Caffe API so there is no direct translation to the Tensorflow API, but the corresponding functions are very accessible, particularly due to the Tensorflow library entries in Tensorflow.nn, which contains the layers in-use on this thesis' models, such as the two-dimensional convolutional layer, transposed convolutional layer, also known as deconvolutional layer with the deconvolution operation covered in Section 3.3.2 at its core and many other layers.

4.2.2 V-Net

The V-Net was first introduced to the academic community in 2016 by Fausto Milletari, Nassir Navab and Seyed-Ahmad Ahmadi [7]. Its authors intended to leverage the CNN's ability to, autonomously, learn a hierarchical representation of input data so as to process third-dimensional images to segment prostate MRI volumes.

The existing drawbacks of potential alternative solutions, such as lack of context learning, high computational effort requirements and the known failures to successfully perform segmentation on ultrasound volumetric images led the authors to turn to the FCNN which at that point in time had achieved great results with a low memory and computational effort footprints but only for two-dimensional input images, as seen in Section 3.2.2 and Section 4.2.1, which implies slicing volumetric images towards that end when required.

Since it was inspired by the U-Net the V-Net it shares many structural similarities with the former such as the symmetrical contracting and expanding paths, henceforth named the compression and decompression paths, to follow the author's terminology which comes from signal processing. Also shared is the skipping property where the compression path outputs, at each stage and before down sampling, are propagated to the input of the corresponding decompression path stage. This process is also known as feature forwarding.

In terms of architecture the V-Net's main innovations regarding its predecessor and asides from the ability to process third-dimensional inputs are the residual network, the replacement of pooling layers by convolutional layers with stride greater than one, the introduction of a dice loss layer based on the dice coefficient to avoid convergence to local minima which are suboptimal at best and terrible at worst, and the use of parametric ReLu activation functions to introduce non-linearity.

The residual network entails learning a residual function, first proposed in a relevant paper [5]. The authors approach to learning this residual function is to take the input of a stage and add it to the output of that stage before down sampling, for the compression path, or up sampling, for the decompression path.

Replacing a pooling layer by a convolutional layer implies emulating the behavior of spatially reducing the size of the input of the pooling layer. The pooling layer displays a poor behavior regarding computational effort and it introduces complex issues to the study of forward and backwards propagation, due to the increase of difficulty on computing the gradient, which leans the optimization algorithm towards a black box approach while transparency is preferred, among other drawbacks discussed in the reference material [13]. A convolutional layer with 2x2x2 kernels and stride 2 can emulate, to an extent, the behavior of a pooling layer, but it requires less computational effort and provides more transparency to the computation of the gradient during forward and backwards propagation.

In the work presented in Section 4.3 this convolutional layer for down sampling remains the same but for two-dimensional inputs so the kernels have 2X2 pixels of receptive field and stride 2.

The dice loss layer uses a formulation of the dice coefficient, otherwise discussed in Section 3.2.2, as a basis towards a loss function. This formulation, following the same terminology stipulated in the aforementioned section, is

$$Dcoef = \frac{2|X \cap Y|}{|X| + |Y|} \qquad (4.35)$$

which for sums running over N voxels of the predicted binary segmentation volume $p_i \in P$ and the ground-truth binary volume $g_i \in G$ can be rewritten as

$$Dcoef = \frac{2\sum_{i}^{N} p_i g_i}{\sum_{i}^{N} p_i^2 + \sum_{i}^{N} g_i^2} \qquad (4.36)$$

The authors, and many other researchers on the community such as Fidon, Li, Garcia-Peraza-Herrera, Ekanayake, Kitchen, Ourselin and Vercauteren [25], have empirically tested and verified the improvements brought by using this objective function, in the context of this sort of task, instead of a

typical multinomial logistic loss with sample re-weighting. As an additional criterion in favor of this loss the need to attribute weights to foreground objects is eliminated.

In the work done in Section 4.3 most models use a standalone sigmoid cross-entropy loss function, some of them use a mix of sigmoid cross entropy loss with dice loss, as formulated in this section, instead of the formulation presented in Section 3.2.2. Others only use the dice loss.

Finally, the parametric ReLu activation function, PReLu, is identical to the leaky-ReLu activation function covered in Section 3.2.2, but instead of using the hyperparameter alpha to define the slope for negative inputs, alpha is itself treated as a parameter of the network which is subject to optimization. The work done does not reflect this however as, instead, exclusively ReLu or leaky-ReLu activation functions are used.

Regarding the specific architecture proposed and used by the authors, the initial layers consist of a contracting path with stages, which are sets of successive layers between down sampling layers, ranging from one convolutional layer to three at each stage. The first stage has a single 5x5x5 convolutional layer followed by the down sampling layer, which then outputs 16 feature maps and halves the spatial resolution of the input, followed by a PReLu activation function. The second stage consists of two 5x5x5 convolutional layers followed by the down sampling, which again doubles the number of feature maps while halving their spatial resolution and then the output is sent to a PReLu activation function. The fourth and fifth stages have three 5x5x5 convolutional layers and follows the same procedure as the stages above regarding outputs. The sixth and final stage also has three convolutional layers with the same spatial dimensions as the stages before it but is followed by an up sampling layers which halves the number of feature maps while doubling their spatial resolution, which makes this stage the tipping point between the compression path and the decompression path.

The symmetrical decompressing path of the architecture corresponds to the latter layers of the network and begins with the output of stage six's up sampling operation followed by a PRelu pair and then, in a sense, reverses the operation done throughout each of the corresponding compression path stages by interweaving feature maps, eventually culminating in the final output layer. As mentioned previously, the V-Net retains the skipping seen in the U-Net and the FCNN but also propagates the residual function's between each stage of input and output as well.

In the output layer a 1x1x1 convolution is used to preserve the original input's volume followed by the voxel-wise softmax operation to convert the input values to probabilities of a voxel belonging to the background or the foreground. The full schema of the network's architecture for 128 by 128 by 64 input volumetric images can be seen in Figure 4.24.

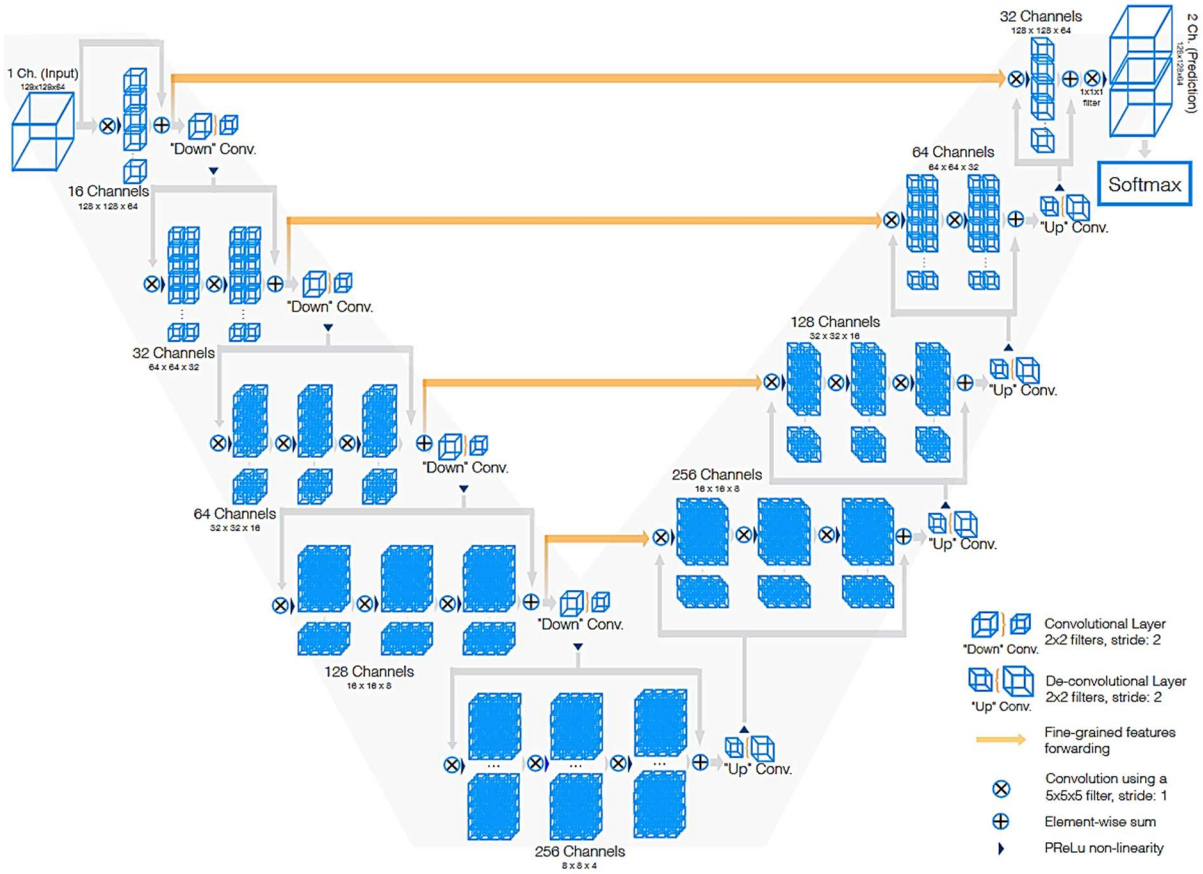


Figure 4.24: Graphical representation of the original V-Net architecture for 8 by 8 by 4 voxels lowest resolution [7].

The V-Net uses mini-batch SGD, defined in Section 2.4.1, with a momentum variant with high momentum, 0.99, as well as a 0.0001 initial learning rate, which decreases by one order of magnitude at every 25 000 iterations to optimize the network, while the work presented in Section 4.3 uses the Adam optimizer, described in Section 3.2.2 along with SGD with momentum.

Due to the very limited, labelled, volumetric datasets, augmentation proved crucial for the model's performance. The authors opted by performing augmentations after loading the batch of samples to reduce the memory footprint and used B-spline interpolation to generate deformed versions of the training images.

Like the U-Net, the implementation of this network was made in Caffe API so there is no direct translation to the Tensorflow API, but the corresponding Tensorflow library also provide third-dimensional convolutional layers, deconvolutional layers as well as many others suited for third-dimensional inputs, including all those available for two-dimensional data.

4.3 The Challenges

Having covered the original U-Net and V-Net architectures, as conceived by their respective authors, we can now advance to the practical variants and the challenges they are applied to. As

mentioned throughout Section 4.2 several changes were made to the original networks to either adapt them to the context at hand or due to time, equipment and even API knowledge constraints.

Both challenges were over before the completion of the algorithms' pipeline. Therefore, instead of testing the developed architectures on the competition's actual setting a portion of the labelled data was set aside and used for evaluation, which limits the viability of the results obtained.

4.3.1 Nuclei Segmentation Challenge

Most of the human body's 30 trillion cells contain a nucleus full of DNA. For that reason identifying cell's nuclei is one of the first steps in most of the efforts to research diseases and their cures, as it facilitates tracking down their corresponding cell, which in turn must be analyzed to understand the biological reactions to new pharmaceutical drugs. On average each new drug developed takes about 10 years to reach the market, but by improving nuclei detection this time can be considerably shortened.

Cells and their nuclei can be vastly different from each other, so this challenge favors models that generalize well rather than specialized models. The provided labelled dataset reflects this as it has samples acquired under a variety of conditions, from different cell types to different modalities.

This competition uses a modified version of intersection over union as score metric, the average precision at different intersection over union thresholds. Their definition of precision, at each threshold value t, is non-canonical and given by

$$precision(t) = \frac{TP(t)}{TP(t) + FP(t) + FN(t)} \qquad (4.37)$$

To score their metric a sweep over a range of IoU thresholds is made, and each point an average precision value is calculated.

As previously mentioned, several changes were made to the original U-Net and V-Net architectures ranging from the activation and loss functions used to the number of feature maps, depth and the spatial resolution and receptive field of the kernels of the convolutional layers. As such the common aspects regarding hyperparameters, preprocessing, training, evaluation and postprocessing are covered first.

The preprocessing portion of the deep learning algorithm pipeline consists of storing the labelled samples and their assembled ground-truths in-memory and then applying augmentation. The original dataset has several ground-truth masks per sample so in order to merge these into a single mask, all ground-truths are normalized and merged. From there each merged ground-truth mask is assigned a weights map based on the same criteria as the original U-Net, Section 4.2.1, and a morphological binary erosion transformation filter is used to clearly delimit each cell's border from its neighbors. An example of a 256x256 pixels sample, its eroded ground-truth mask and the morphological filter used, from right to left respectively, can be seen in Figure 4.25. The colored label represents the intensity of each pixel, yellow being the maximum and dark purple the minimum.

All models split the original dataset of 670 samples into about 70% training set samples and 30% validation set samples and the available training set for each model contains the same samples. All the inputs are also required to have the same spatial dimensions for height and width, as such the samples and corresponding ground-truths of the original dataset that do not meet this requirement are mirrored and cropped to have either 256x256 pixels or 512x512 pixels as spatial dimensions.

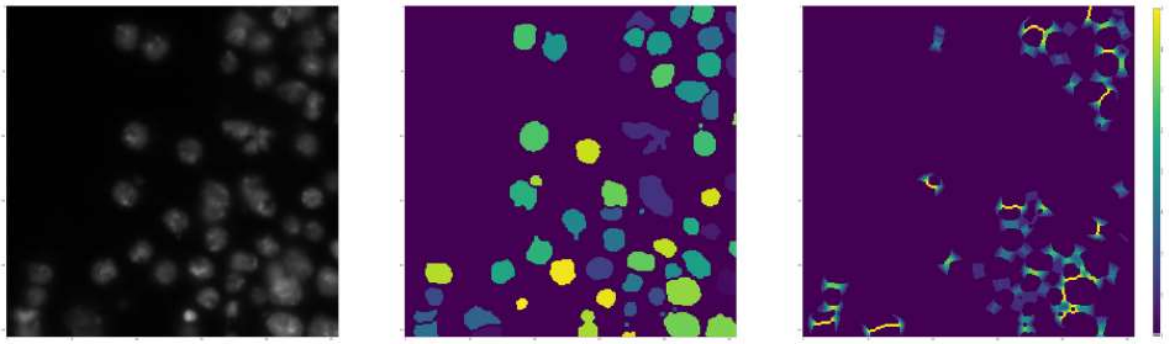


Figure 4.25: Original 256 pixels of width by 256 pixels of height image and its corresponding eroded ground-truth and morphological binary filter.

Samples that have the same width and height result in augmented samples that either preserve the original dimension, in which case no augmentation is done, are fully mirrored horizontally, vertically and diagonally to match 512x512 pixels, as seen in Figure 4.26 which is generated from Figure 4.25 sample, or are fully mirrored and then cropped to match the desired spatial dimensions.

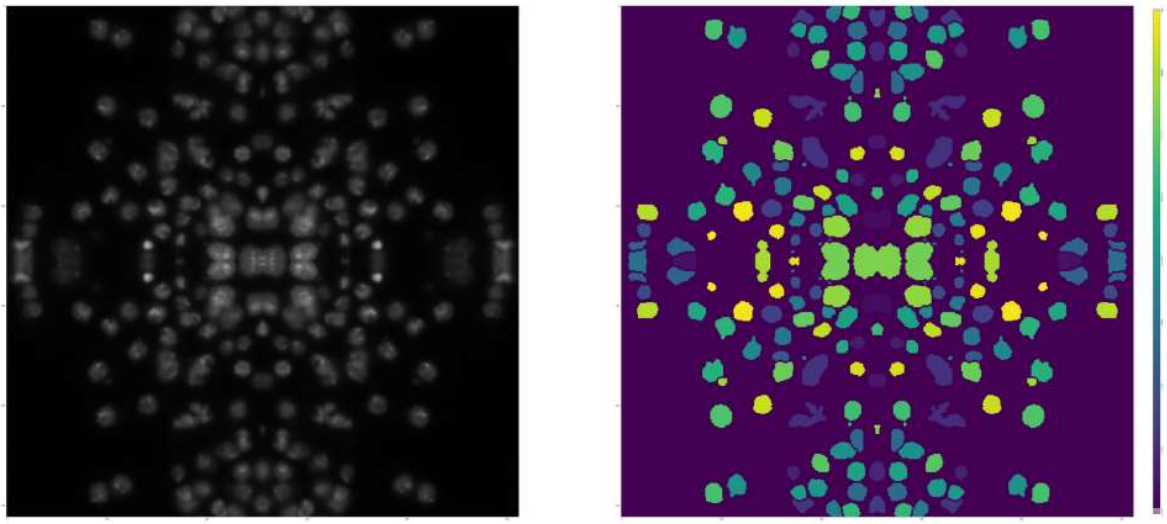


Figure 4.26: Symmetrically mirrored augmentation on the sample of Figure 4.25 to expand it to 512x512 pixels.

Samples that do not have the same spatial dimensions along their axis, such as that of Figure 4.27 where the original sample exceeds 256 pixel width but retains 256 pixel height, can also be augmented through the mirroring and cropping process where they are mirrored in one or more ways and then cropped to match the desired spatial dimensions which, in this case, leads to the four augmented samples, displayed in Figure 4.28, each with 256x256 pixels.

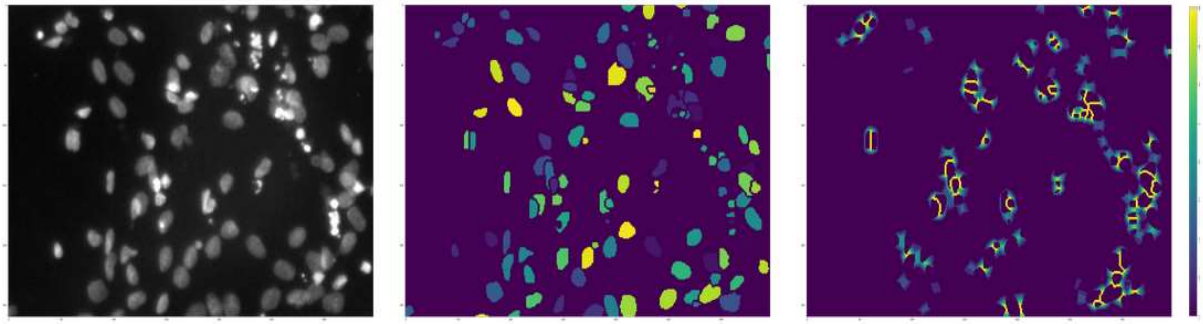


Figure 4.27: Sample with spatial dimensions disparity and its corresponding eroded ground-truth and morphological binary filter.

The samples in Figures 4.28(a) and 4.28(b) are horizontally symmetrical crops of the original sample while Figures 4.28(c) and 4.28(d) are vertically mirrored cropped samples of the original while also horizontally symmetrical to each other.

Augmented samples are not subjected to further augmentation themselves and during the training process they are randomly selected so that all models share the same preprocessing stage and differ only in their training stage, without diverging significantly from each other sample-wise as the number of samples processed increases. Post-augmentation, the number of 256x256 samples ranges in the thousands and the total amount of samples is increased almost tenfold.

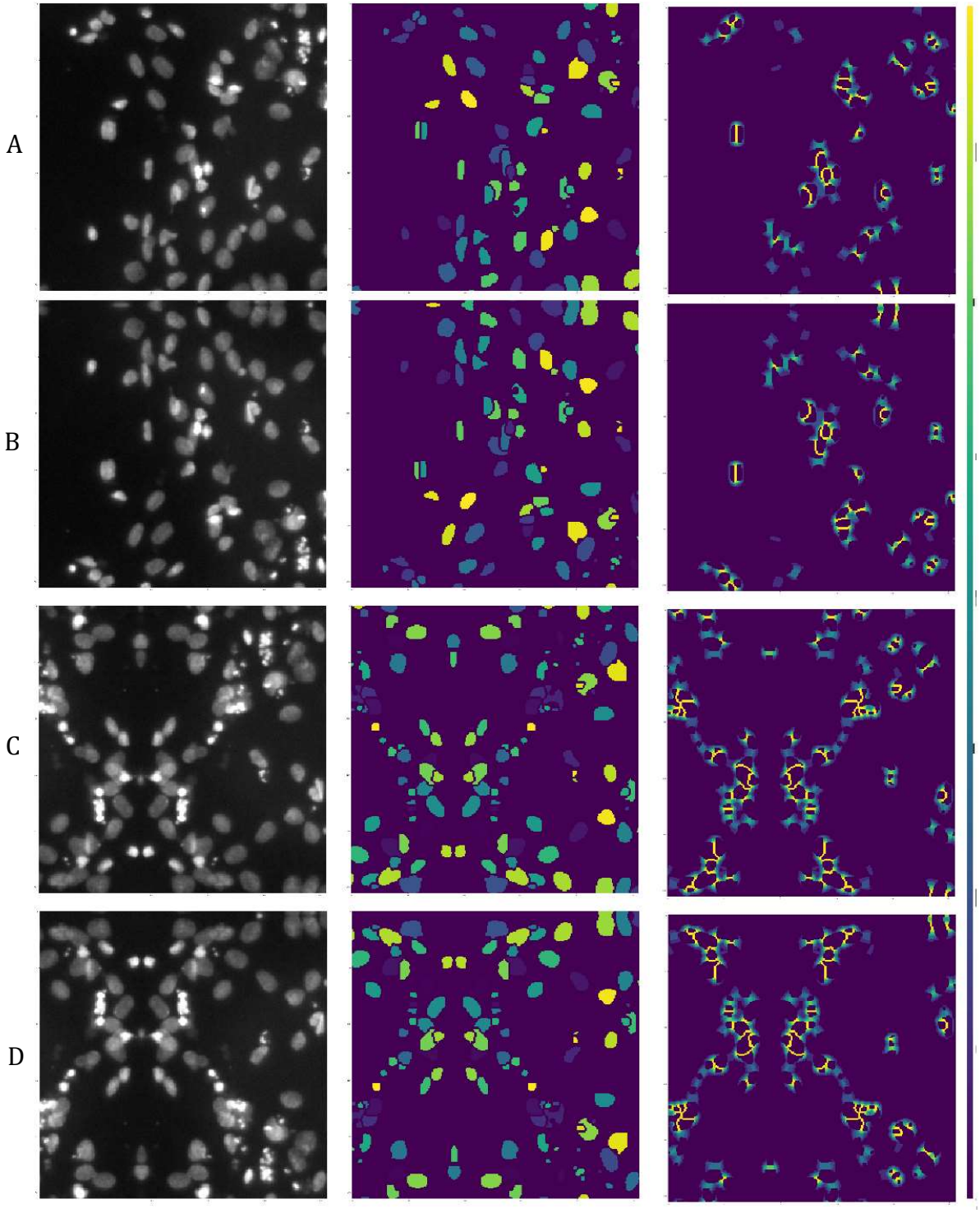


Figure 4.28: Augmented 256 by 256 pixel samples, their ground-truths and morphological binary filters.

Post-processing entails generating a predictions mask from the probabilities of each individual pixel belonging to either the foreground or the background by setting a threshold and applying binary dilation to try and mitigate the binary erosion applied. All models consider probabilities which are above the 0.5 threshold as valid foreground pixels. To illustrate this process the sample from Figure 4.28(a),

the corresponding predictions mask outputted by one of the models and the final binary predictions mask, from left to right respectively, are shown in Figure 4.29.

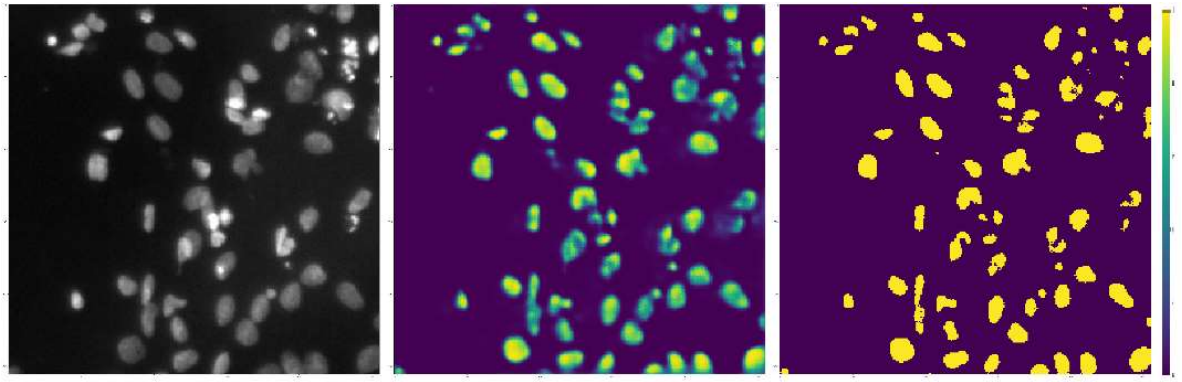


Figure 4.29: Augmented sample from Figure 4.28(a), its probabilistic predictions mask and the corresponding binary predictions mask.

The final part of the post-processing process requires reversing the augmentation process by iterating through each augmented sample's binary predictions mask, reversing the transformations applied and merging all masks into the final predictions mask. The challenge required outputs to be submitted in the RLE target encoding format which is basically a two-column dataset, where the first column contains rows of sample identifier strings and the second the corresponding localization of each pixel in the foreground for that sample in string format.

Having covered the preprocessing and postprocessing stages of the pipeline which are shared among the various models, other common aspects, such as hyperparameters are now covered. All models use the Adam variant of SGD, covered in Section 3.2.2, with an initial learning rate of 1×10^{-4} as the optimizer for forward and backwards propagation. Each training sample is processed 5 times before advancing to the next sample, that is advancing to the next step. Early stop is used, for reasons detailed in Section 2.4.2, so the networks do not exceed 3500 training steps.

As previously mentioned (Section 4.2), each down sampling layer, either max pooling for the U-Net variants or the convolution with stride 2 for the V-Net variants, doubles the number of feature maps while halving their spatial resolution. For these models the number of feature maps ranges from the initial 8 up to 128. Likewise, for each up sampling layer, implemented using the deconvolution operation mentioned in Section 3.3.2, the number of feature maps is halved while the spatial resolution doubles, from 128 feature maps down to 8.

Finally, the outputs are all submitted to a 1 by 1 convolutional layer with stride 1 which outputs the logits. These are then processed by the sigmoid function to output a probability distribution over each pixel of that particular pixel belonging to the foreground.

With the common aspects of the implementations handled we now move on to description of each variant. In total 8 models are used, two are U-Net variants, one is a U-Net/V-Net hybrid, and five are V-Net variants. The characteristics of each model can be seen on Table 4.1.

The first U-Net variant, U3CLR, closely resembles the original implementation covered in Section 4.2.1. Architecture-wise there is one main difference from the original, which is the padding

used in all convolutions, discussed in Section 3.3.1. Unlike the original this implementation uses the sigmoid cross-entropy loss function as the objective function to be optimized, since the foreground object segmentation is treated as a pixel-by-pixel binary classification task. To showcase the performance of this network a couple of its predictions are compared with the expected ground-truths from the validation set in Figure 4.30.

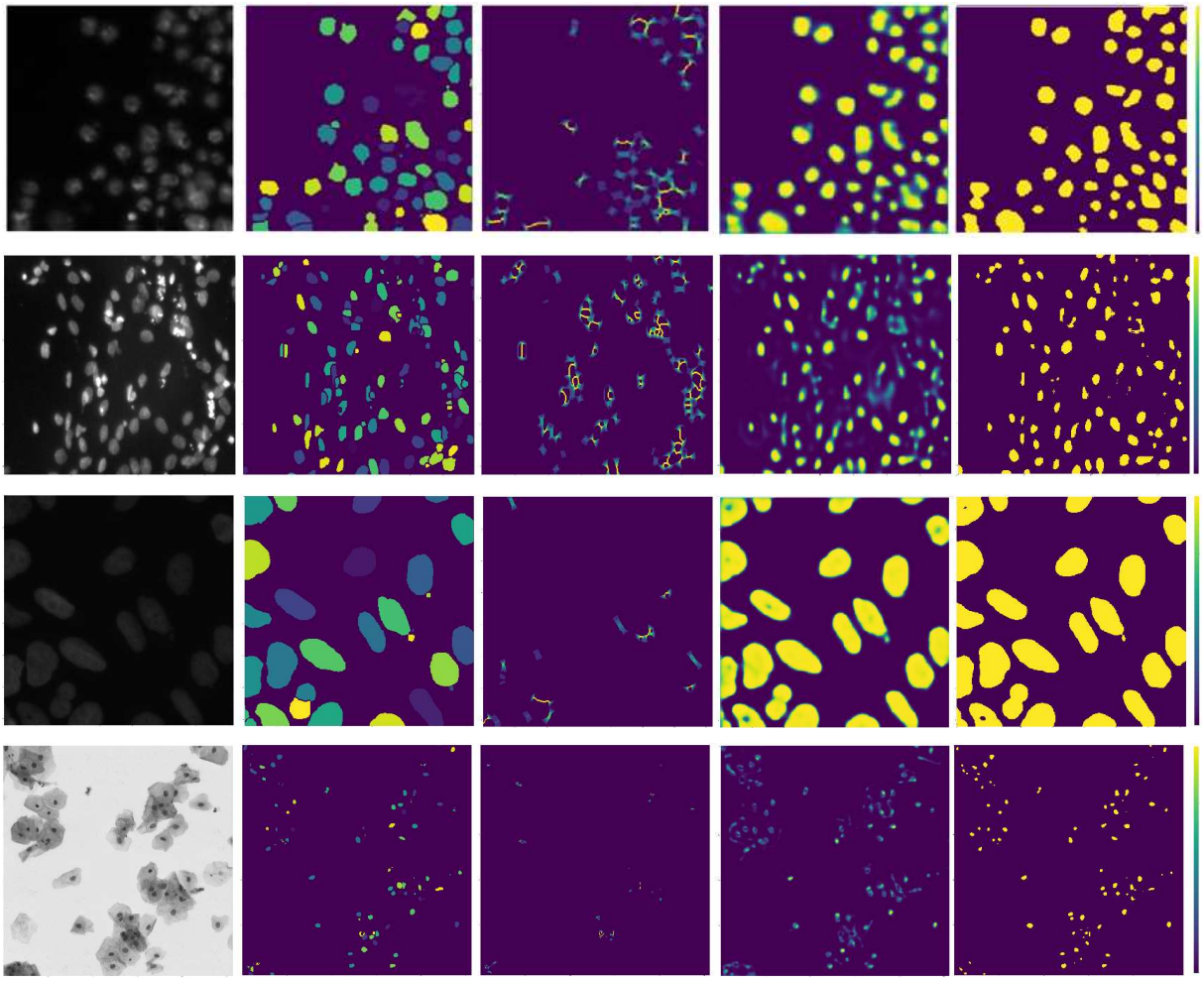


Figure 4.30: The input images, their corresponding ground-truth and binary erosion filter (left) side-by-side with the probabilistic prediction mask and binary prediction mask (right) outputted by the first U-Net model.

The second U-Net variant, U5HLLR, introduces some enhancements to the original and the first variant. In terms of the architecture itself, the convolutional layers have their receptive fields expanded to 5 by 5 pixels and, instead of relying on ReLu activation functions after each convolutional layer, this variant uses the leaky-ReLu activation function. It uses He initialization instead of Xavier initialization. In terms of the loss function, this variant uses a hybrid loss function, the sigmoid cross-entropy loss from the first variant conjoined with the soft-dice loss discussed in Section 4.2.2. To showcase the performance of this network a couple its predictions are compared with the expected ground-truths from the validation set in Figure 4.31.

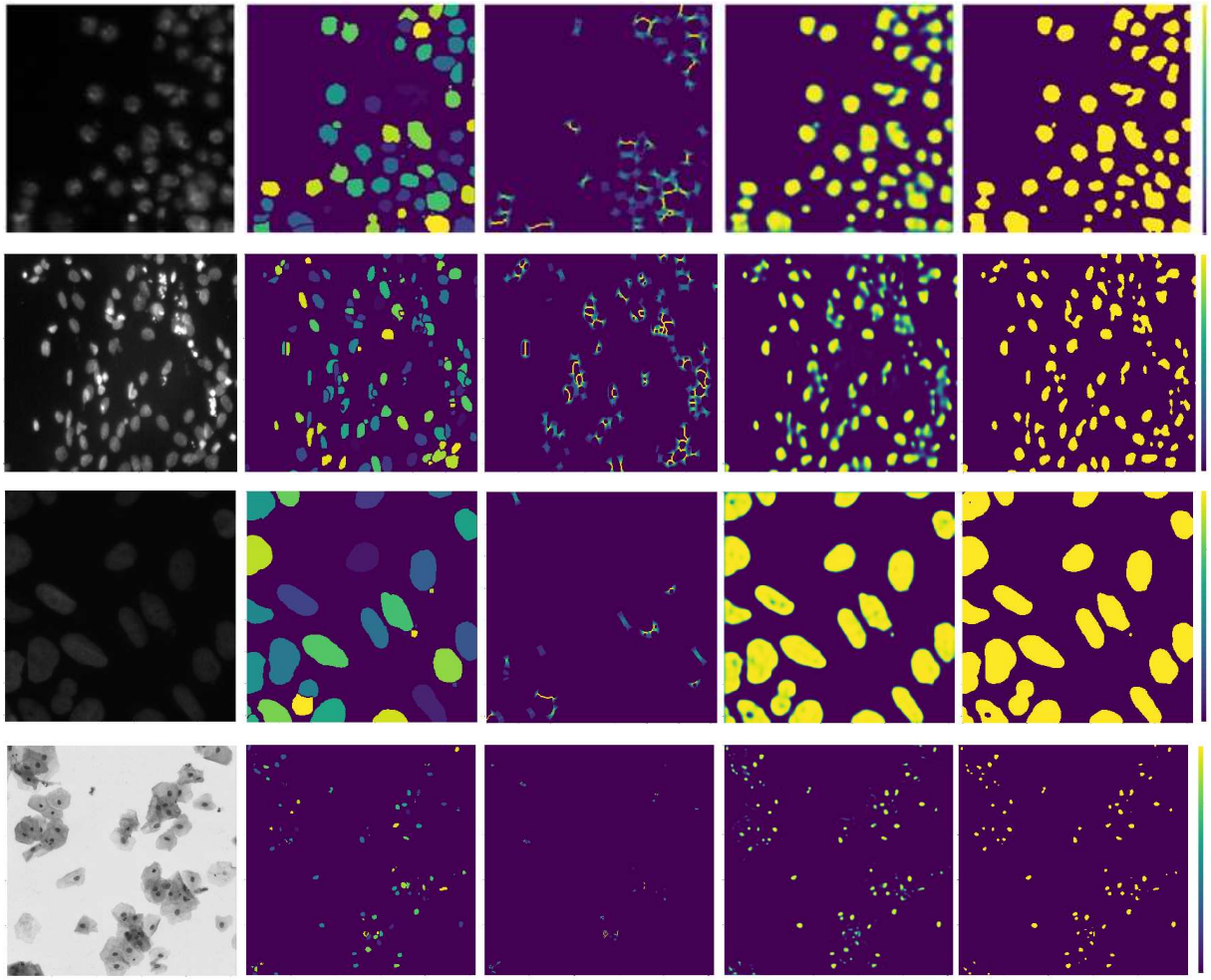
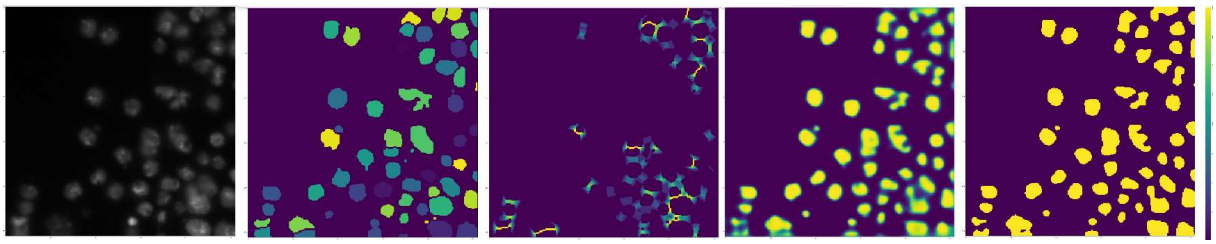


Figure 4.31: The input images, their corresponding ground-truth and binary erosion filter (left) side-by-side with the probabilistic prediction mask and binary prediction mask (right) outputted by the second U-Net model.

The hybrid U-Net/V-Net variant, HUV, is essentially a U-Net implementation with the standard max pooling layers replaced by convolutional layers with 2 by 2 pixel-wide receptive fields and stride 2 for their kernels, covered in Section 3.3.1. It uses 5 by 5 receptive fields for its standard convolutional layers but also ReLu as activation function and the sigmoid cross-entropy loss function as the objective function to minimize. To become a full-fledged two-dimensional V-Net implementation this variant lacks the residual propagation network covered in Section 4.2.2. To showcase the performance of this network a couple its predictions are compared with the expected ground-truths from the validation set in Figure 4.32.



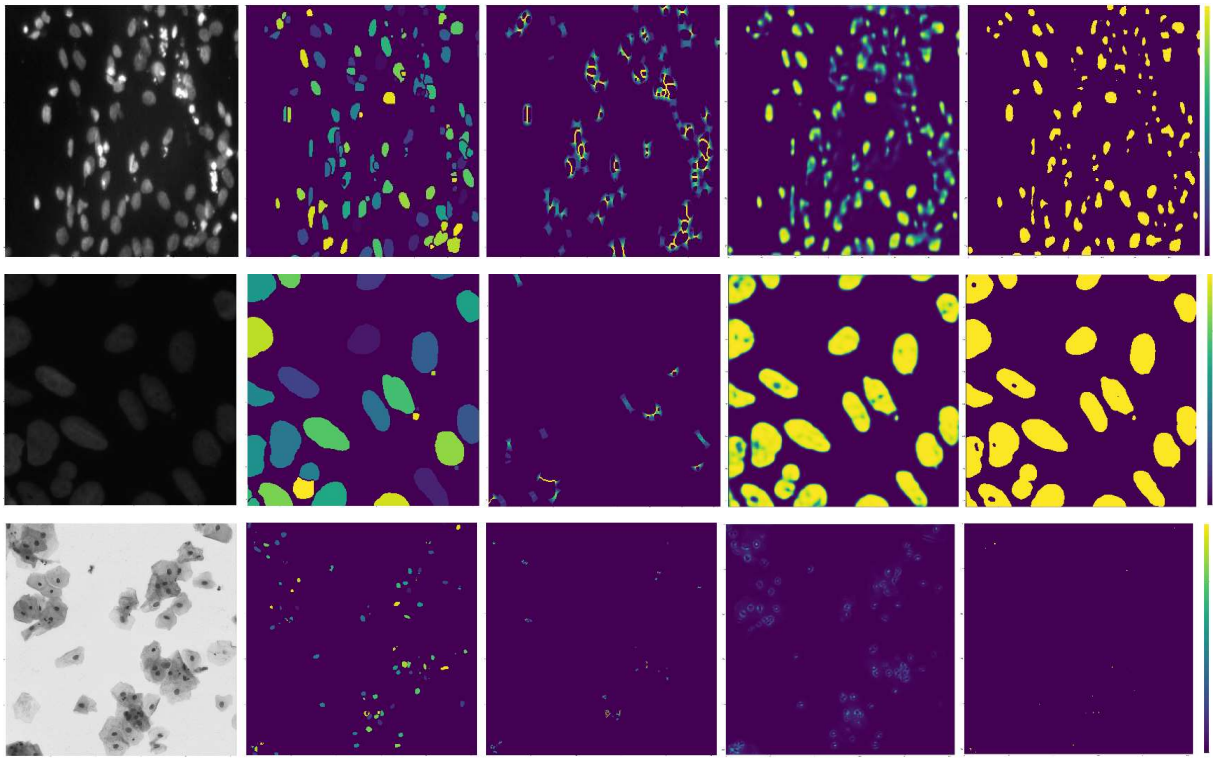
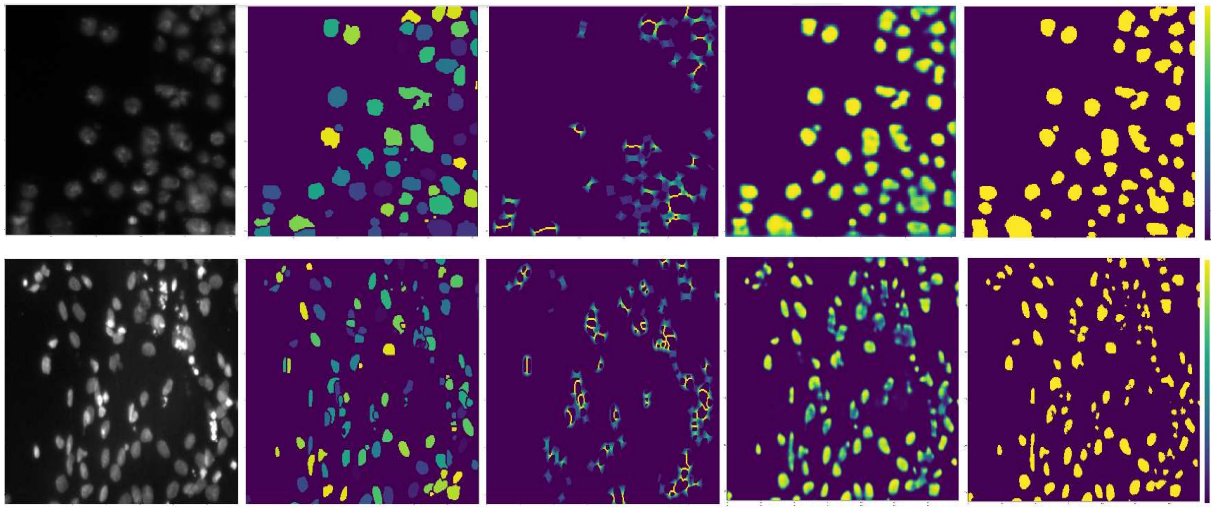


Figure 4.32: The input images, their corresponding ground-truth and binary erosion filter (left) side-by-side with the probabilistic prediction mask and binary prediction mask (right) outputted by the hybrid U-Net/V-Net model.

The first V-Net variant, VCR, differs the most from the implementation done by the original authors, aside from having one spatial dimension less like all other variants. In terms of the architecture of the network itself, the major differences this variant has are the ReLu activation functions used after every convolutional layer, while the original uses a PReLu activation function after each down sampling and up sampling layer and the initialization of weights through the use of Xavier initialization, covered in Section 3.2.2. Regarding the loss function this variant uses the sigmoid cross-entropy loss while the authors propose the use of dice coefficient loss. To showcase the performance of this network a couple of its predictions are compared with the expected ground-truths from the validation set in Figure 4.33.



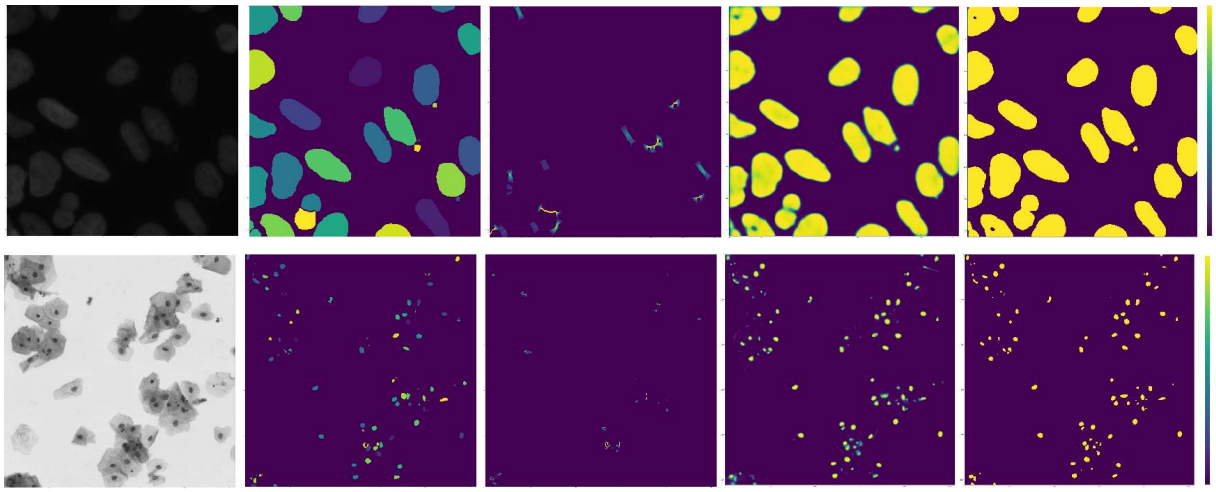
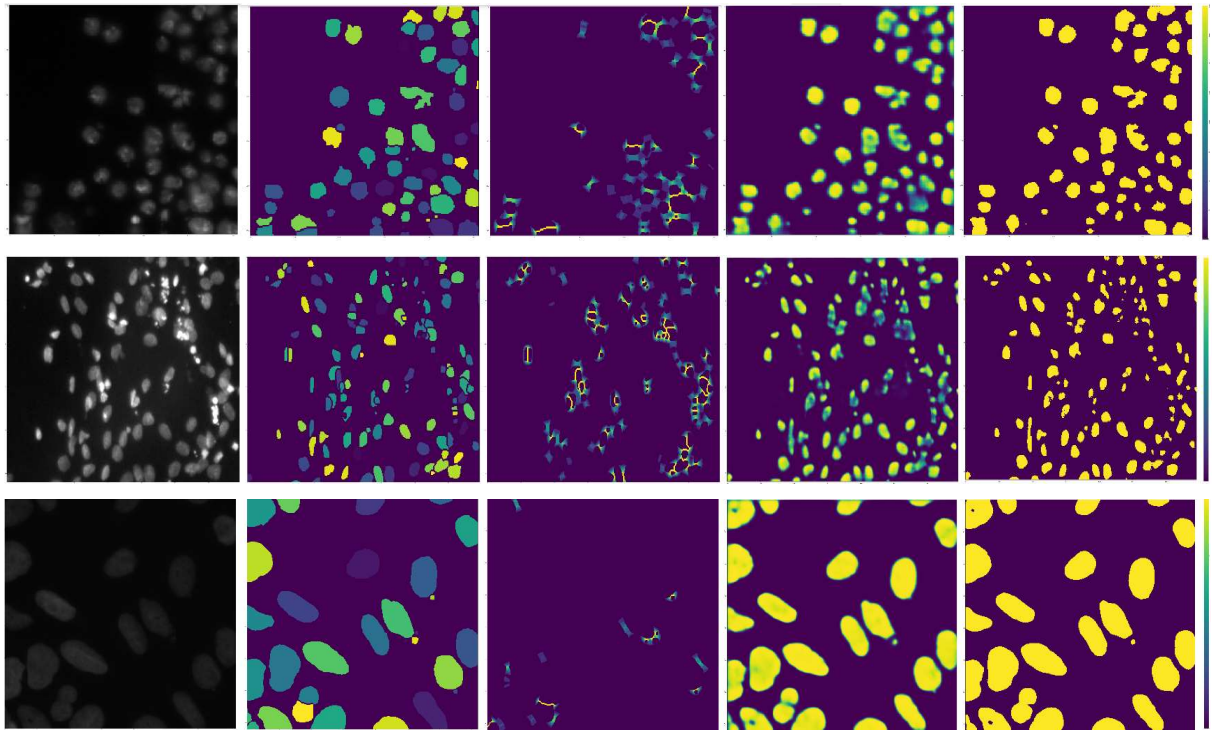


Figure 4.33: The input images, their corresponding ground-truth and binary erosion filter (left) side-by-side with the probabilistic prediction mask and binary prediction mask (right) outputted by the first V-Net model.

The second V-Net variant, VHLR, is similar to the first, barring the loss function, which is a combination of the sigmoid cross-entropy loss with the dice coefficient loss proposed by the authors of the original V-Net. To showcase the performance of this network a couple of its predictions are compared with the expected ground-truths from the validation set in Figure 4.34.



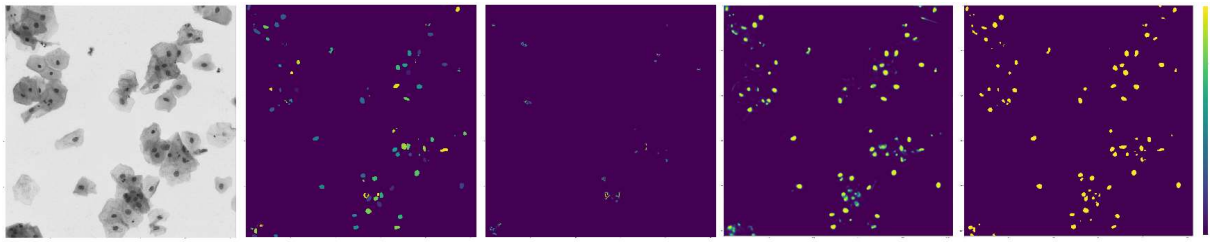


Figure 4.34: The input images, their corresponding ground-truth and binary erosion filter (left) side-by-side with the probabilistic prediction mask and binary prediction mask (right) outputted by the second V-Net model.

The third V-Net variant, VCLR, works upon the first by replacing the ReLu activation functions with leaky-ReLu activations which, as seen in Section 3.2.2, closely resemble the PReLu activation functions used by the authors of the original V-Net. To showcase the performance of this network a couple of its predictions are compared with the expected ground-truths from the validation set in Figure 4.35.

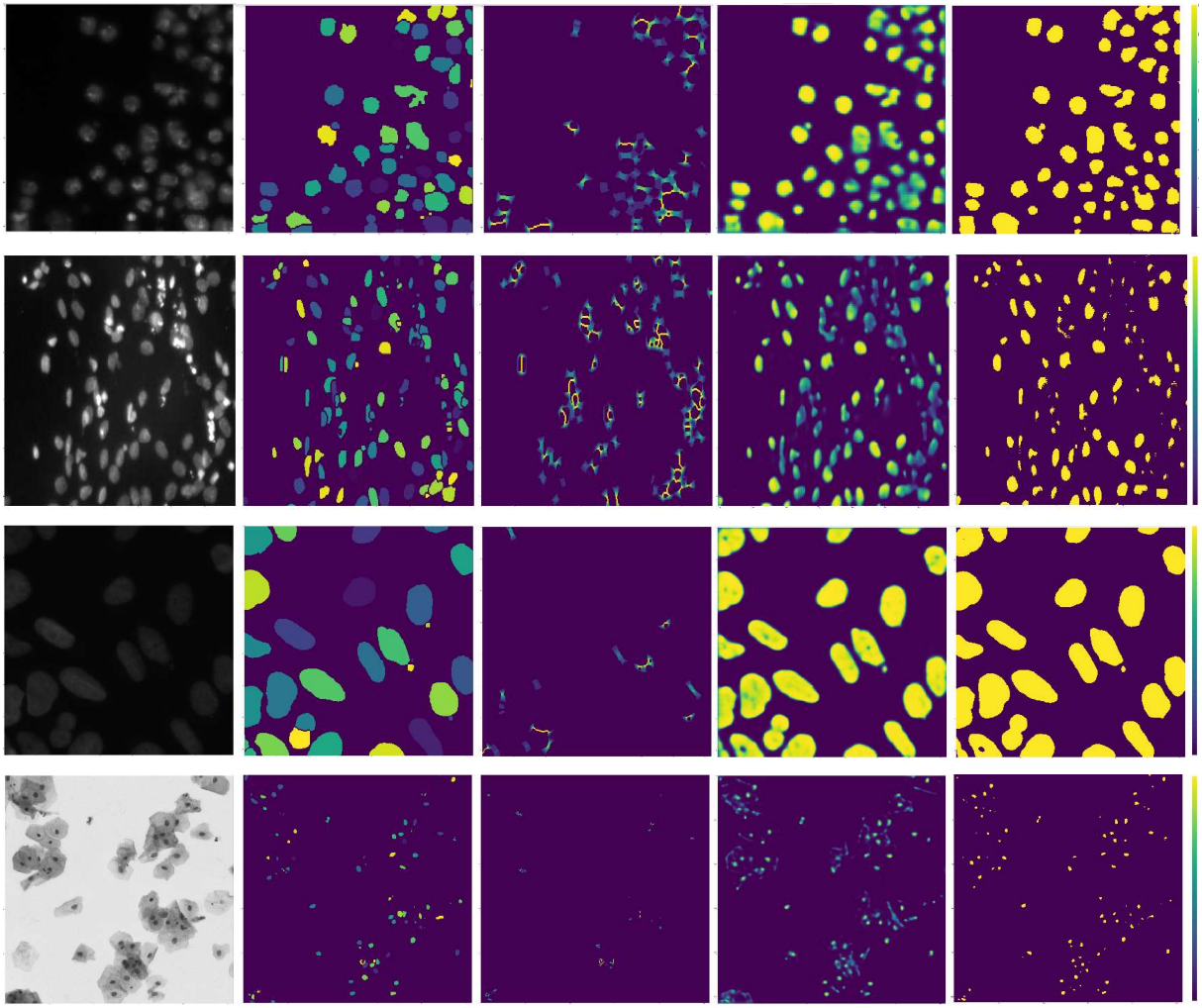


Figure 4.35: The input images, their corresponding ground-truth and binary erosion filter (left) side-by-side with the probabilistic prediction mask and binary prediction mask (right) outputted by the third V-Net model.

The fourth V-Net variant, VHLLRHe, resembles the second variant but instead of ReLu activation functions and Xavier initialization it uses leaky-ReLu activation functions and He initialization. To showcase the performance of this network a couple of its predictions are compared with the expected ground-truths from the validation set in Figure 4.36.

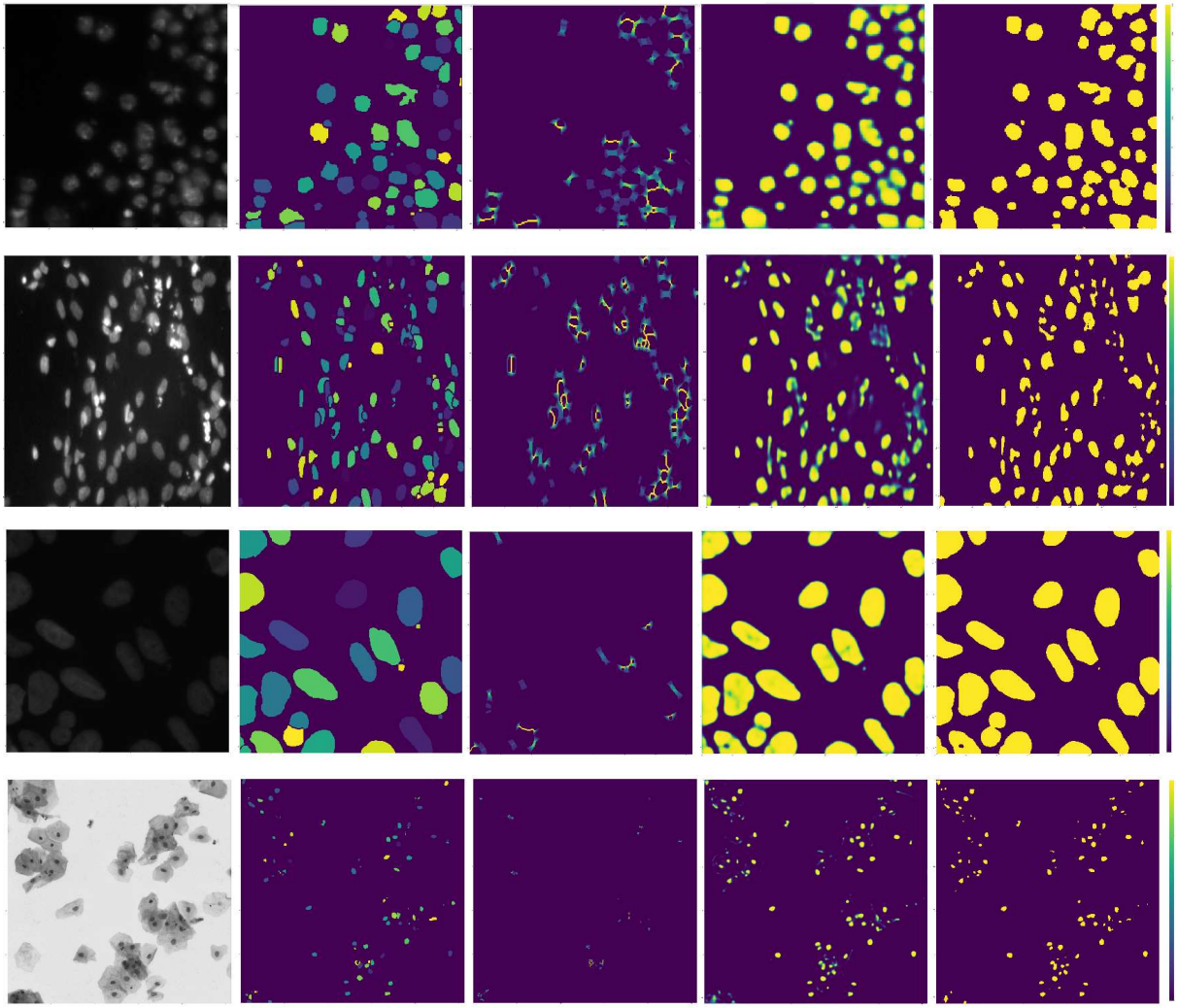


Figure 4.36: The input images, their corresponding ground-truth and binary erosion filter (left) side-by-side with the probabilistic prediction mask and binary prediction mask (right) outputted by the fourth V-Net model.

The fifth V-Net variant, VDLRHe, differs from the previous one due to the use of the dice coefficient loss as the sole objective function. To showcase the performance of this network a couple of its predictions are compared with the expected ground-truths from the validation set in Figure 4.37.

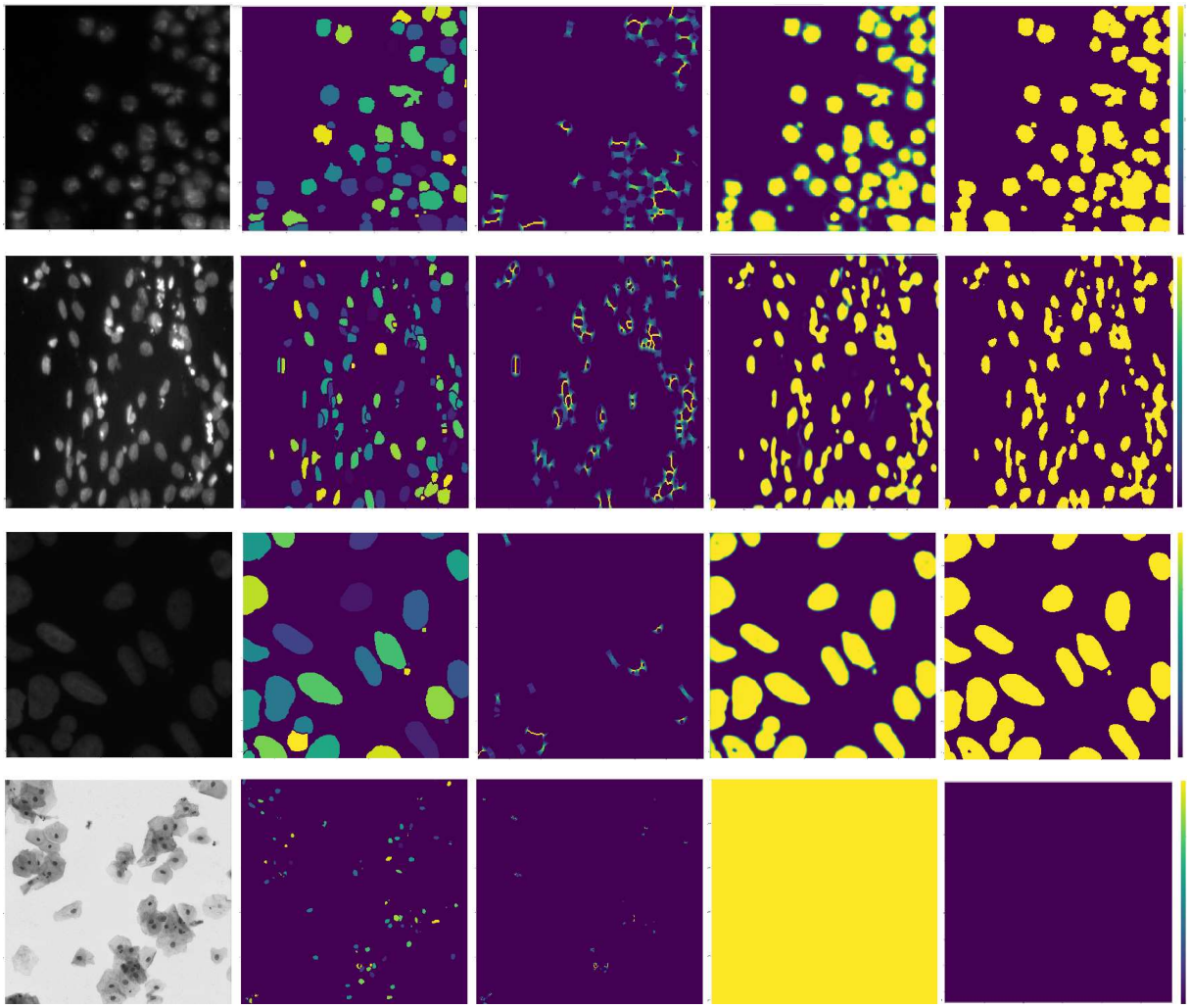


Figure 4.37: The input images, their corresponding ground-truth and binary erosion filter (left) side-by-side with the probabilistic prediction mask and binary prediction mask (right) outputted by the fifth V-Net model.

By analyzing the results exposed in Table 4.2, rounded to the fourth decimal place, conclusions can be drawn for the overall model performances. All variants of the U-Net and V-Net perform rather well regarding the metric that most closely resembles the one used to assess the model's performance in the actual challenge, the intersection over union. The U-Net variants, due to lacking the residual function propagation and using pooling layers, train much slower than the V-Net variants and tend to perform slightly worse under similar conditions.

Table 4.1: Variant characteristics.

Configurations	Loss	Activation	Initialization
U3CLR	Cross-entropy	ReLu	Xavier
U5HLLR	Cross-entropy + Dice	Leaky-ReLu	He
HUV	Cross-entropy	ReLu	Xavier
VCR	Cross-entropy	ReLu	Xavier
VHLR	Cross-entropy + Dice	Leaky-ReLu	Xavier
VCLR	Cross-entropy	Leaky-ReLu	Xavier
VHLLRHe	Cross-entropy + Dice	Leaky-ReLu	He
VDLRHe	Dice	Leaky-ReLu	He

Table 4.2: U-Net and V-Net model performance.

Configurations	Loss	Mean IOU	Accuracy [%]	Precision [%]	Recall [%]
U3CLR	0,0745	0,8932	97,6663	93,5002	86,1195
U5HLLR	0,2392	0,9054	97,8816	91,0258	90,9301
HUV	0,0747	0,8864	97,5772	96,5799	82,2877
VCR	0,0753	0,8929	97,7037	95,9587	83,9880
VHLR	0,2084	0,9161	98,1818	94,6954	89,5369
VCLR	0,0675	0,9067	97,9704	94,3029	88,0402
VHLLRHe	0,2114	0,9173	98,1960	94,0450	90,3635
VDLRHe	0,2870	0,5255	76,1173	32,4727	95,7107

Overall the use of a hybrid objective function seems to be the decisive differentiating factor, as it leads to mean intersection over union improvements in the order a fraction of a percent, while changing

the activation function from ReLu to leaky-ReLu or using Xavier or He initialization only influences this metric by up to an order of magnitude below this.

It is also noteworthy that, despite the poor performance, those models which rely solely on the Dice objective function tend to achieve a much better recall at the cost of precision, while the models that lack residual propagation overfit much faster than those who incorporate it, particularly when coupled with a lack of pooling layers.

Figures 4.38 consists of full, simplified, diagram of the V-Net implementation of the VHLLRHe model. Figures 4.39 and 4.40 contain the stepwise evolution of the error metrics for evaluation and training, respectively.

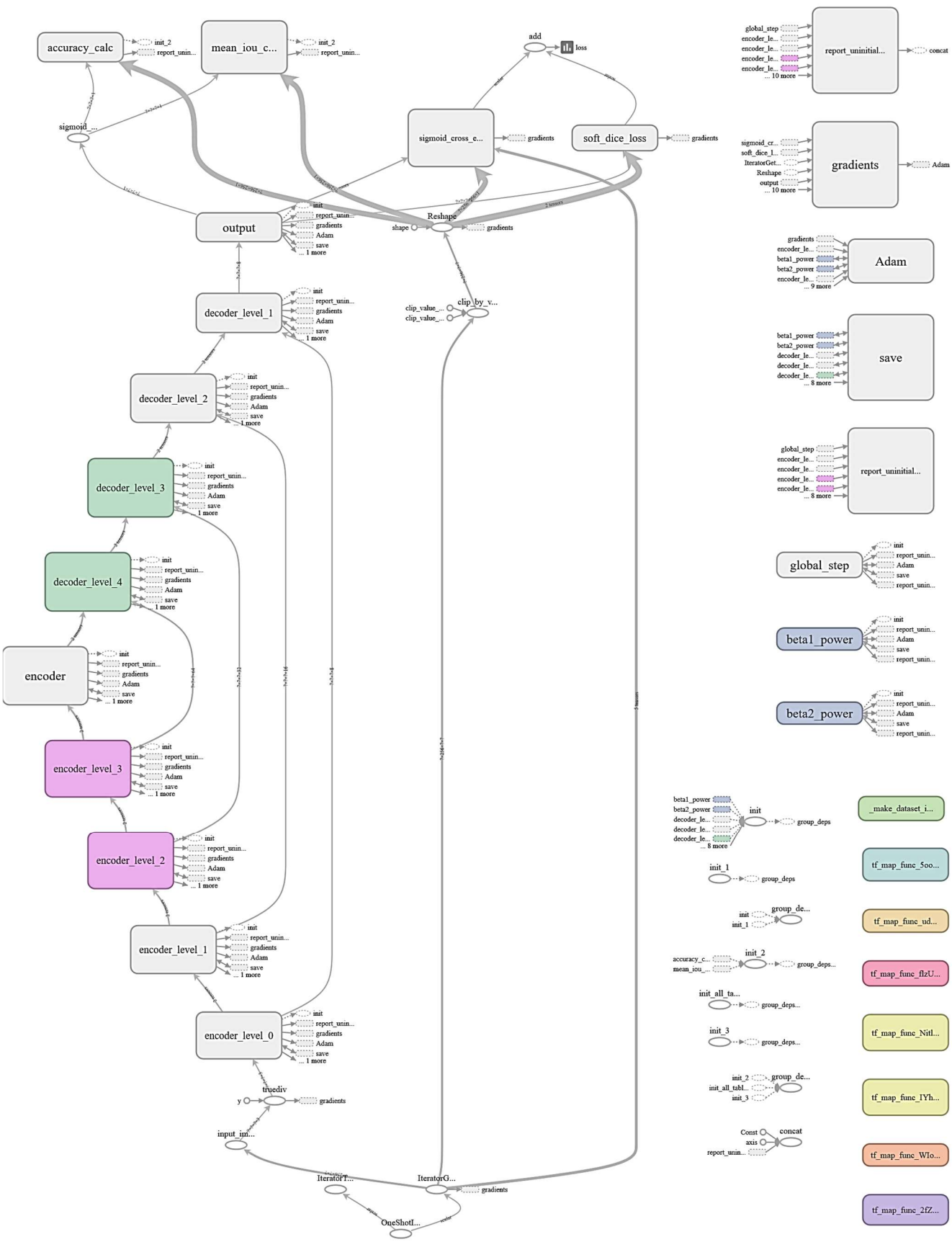


Figure 4.38: Diagram of the V-Net implementation of VHLLRHe model.

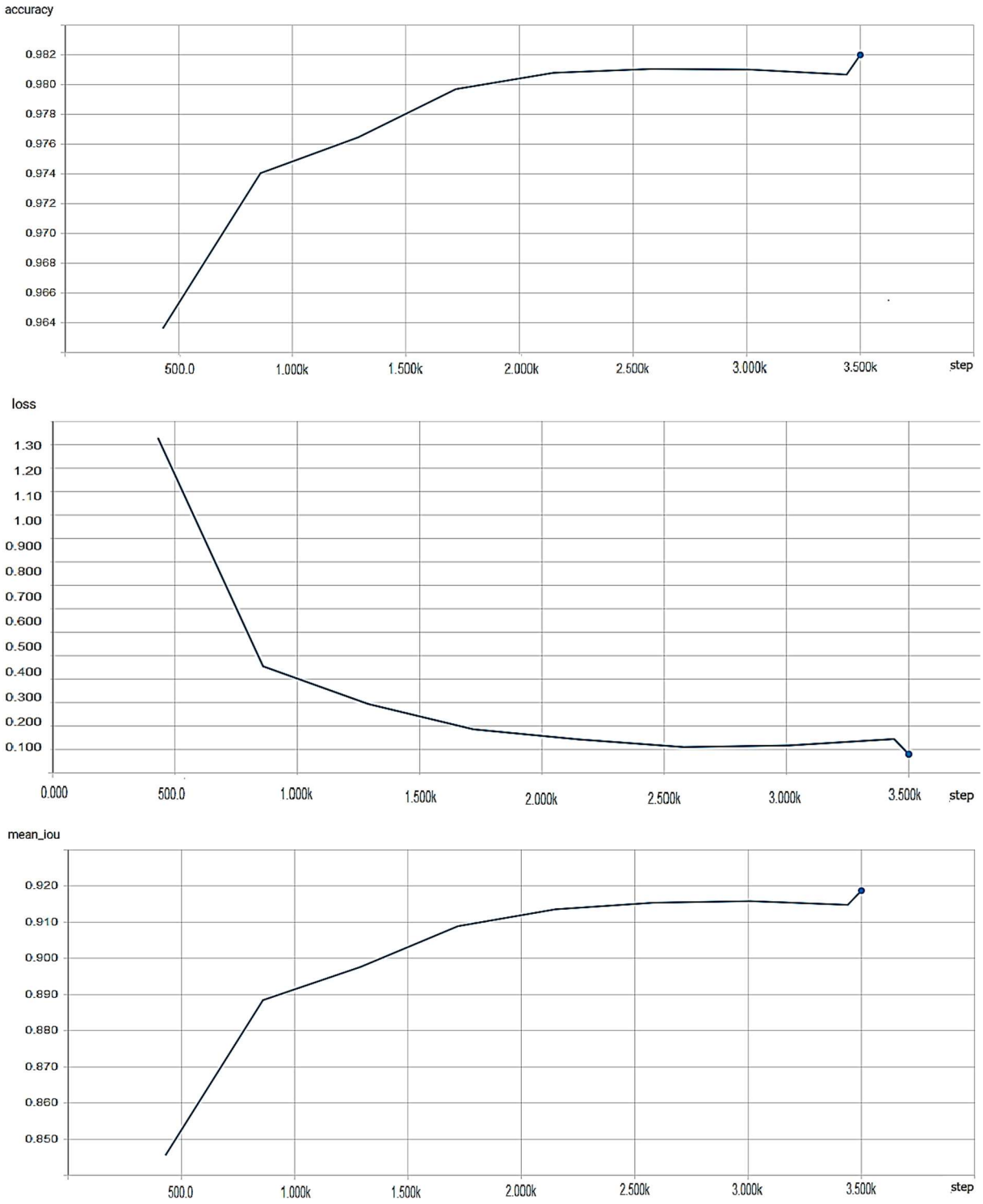


Figure 4.39: Error metric progression for VHLLRHe model's evaluation stage.

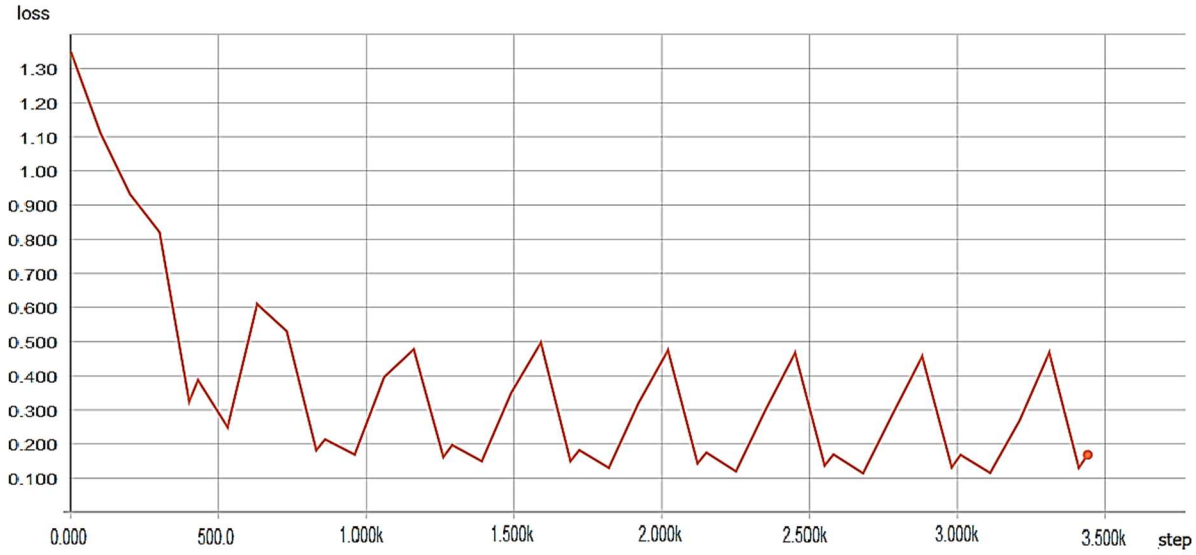


Figure 4.40: Loss metric progression for VHLLRHe model's training stage.

4.3.2 Nerve Segmentation Challenge

The main motivation behind accurately segmenting nerve lesions, in this challenge's context, is to improve patient's pain management through the use of indwelling catheters that block or mitigate the pain at its source, speeding up patient recovery and avoiding the risks associated with narcotics. Toward that end, the competition's sponsor proposed the nerve lesion segmentation challenge, or Ultrasound Nerve Segmentation, which entails identifying nerve structures of the neck area through labelled ultrasounds.

The precise task is to segment a collection of nerves called Brachial Plexus through the processing of two-dimensional images that correspond to slices of a volumetric ultrasound accompanied by their corresponding binary ground-truth masks. In total there are 5635 slices corresponding to 47 ultrasounds.

From the comparisons made in Section 4.3.1 the model which struck the best performance to computational effort ratio was chosen to tackle this challenge. Originally the intention was to scale it to accept third-dimensional input data by combining the provided two-dimensional labelled samples and assembling them back into 47 volumetric ultrasound images with their corresponding ground-truths, but due to time constrictions and the availability of hardware resources to allocate towards this task this idea was abandoned.

Since the challenge's metric of evaluation is the Sorensen-Dice coefficient loss leaving the model's implementation as is, is considered a good practice since it is already designed to train on a soft-dice coefficient loss. Implementations relying solely on the soft-dice coefficient loss are also present to assess their performance.

The soft-dice coefficient loss used in conjunction with the sigmoid cross-entropy loss function can be a boon to the model's overall performance and both the residual propagation network and the lack of pooling layers provide a great reduction in the convergence time and underlying computational effort the choice of what configuration to use is narrowed down to VHLLRHe and

VNet_5by5_HybridLoss_Relu. Due to achieving the best performance at 3500 training steps for the Data Science Bowl 2018 challenge variant VHLLRHe is chosen, its pipeline is fully preserved as well, data augmentation included.

A two-dimensional ultrasound sample with the relevant nerve structure can be seen in Figure 41 alongside its ground-truth mask with the demarcated foreground object.

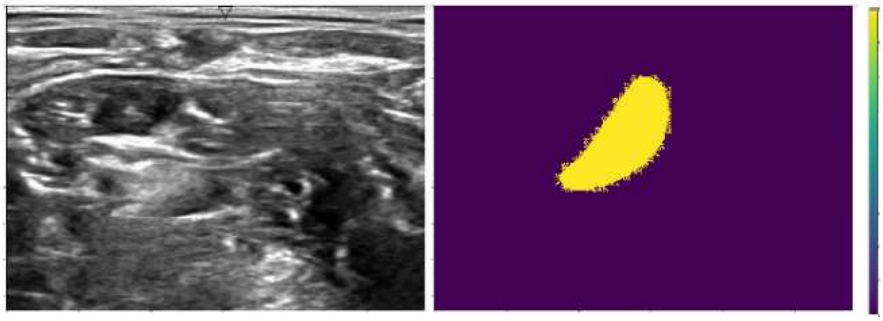


Figure 4.41: Ultrasound input sample and its denoted ground-truth mask.

As in Section 4.2.1 some samples have no pixels belonging to the foreground, like the example seen on Figure 4.42. Unlike the Data Science Bowl 2018 dataset however, the Ultrasound Nerve Segmentation one has far less samples and a greater disparity between the number of foreground pixels and background ones, with the majority belonging to the latter class of course. The dice coefficient is, fortunately, the best tool in-use to ensure proper penalties are issued to the training process to counteract this imbalance, as discussed in Sections 3.2.2 and 4.1.2.

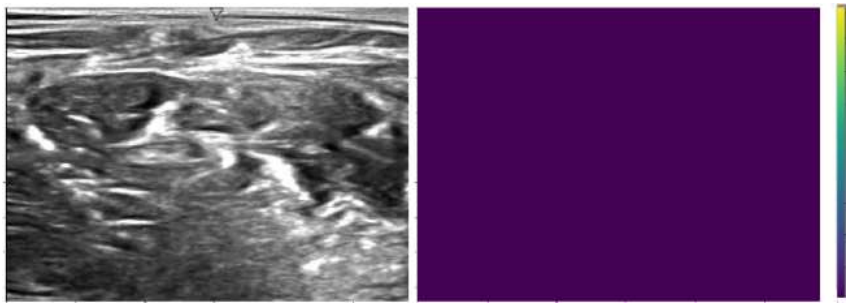


Figure 4.42: Ultrasound input sample, its empty ground-truth mask.

Since there are far less foreground pixels, the model itself also trains much faster with the caveat of learning much slower. As an attempt to improve the learning process, instead of training the model from Xavier or He initialized weights an alternative solution is proposed. It entails porting the weight values learned by the original VHLLRHe implementation in Section 4.3.1 and using them as initializers for this network's weights. For the models which solely employ the soft-dice loss, due to poor results from their predecessor, different learning rates were used instead.

We first compare the results obtained by the model which ports the architecture of VHLLRHe from Section 4.3.1, but not the weights from its graph, No_Prior, with the model which does start its training process from the weights of its predecessor, From_Prior, close to or at steps 20000, 25000 and 30000. Figure 4.43 showcases the outputs of No_Prior by presenting a couple of its predictions side-by-side with the expected ground-truths from the validation set.

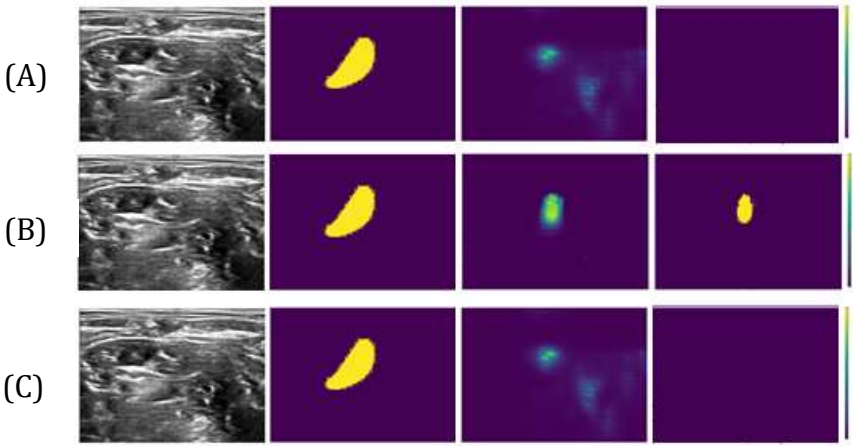


Figure 4.43: An input image, its ground-truth (left) side-by-side with the probabilistic prediction mask and binary prediction mask (right) outputted by the VHLLRHe model with imported weights from Section 4.3.1 at (a) roughly step 20000; (b) roughly step 25000; (c) roughly step 30000.

By analyzing the outputs and the error metrics presented in Table 4.3, rounded to the fourth decimal place, and the progression of evaluation metrics in Figure 4.44 it can be seen that without further spatial context the performance of this model is consistently poor, and it is unable to learn the task at hand. In Figures 4.45 the outputs of No_Prior are displayed side-by-side with the original input sample at or close to steps 20000, 25000 and 30000, respectively.

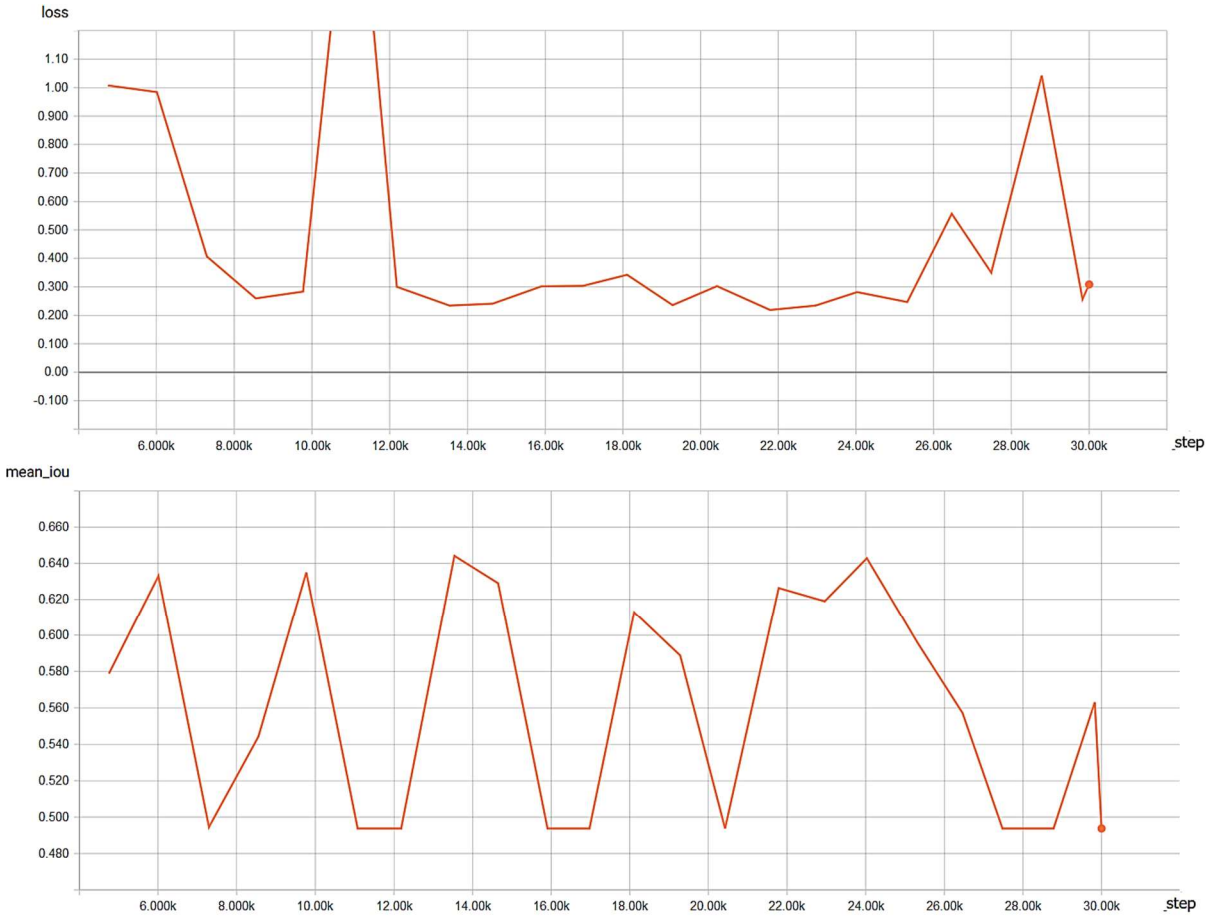


Figure 4.44: Error metric progression for VHLLRHe model's evaluation stage with imported weights.

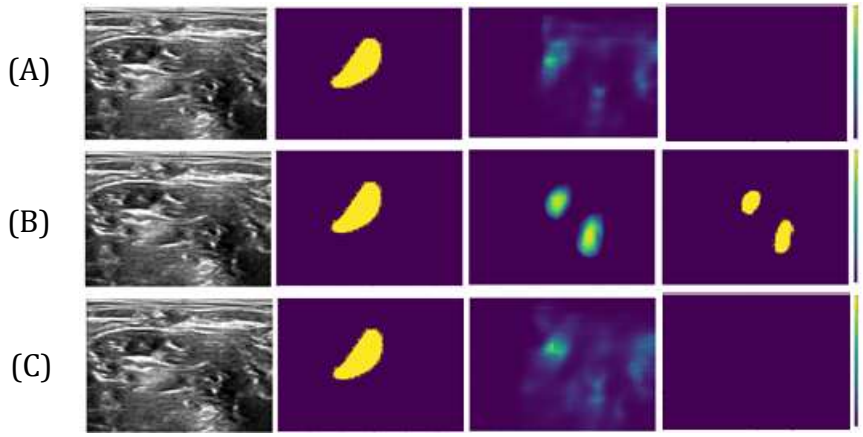


Figure 4.45: An input image and its ground-truth (left) side-by-side with the probabilistic prediction mask and binary prediction mask (right) outputted by the VHLLRHe model without imported weights from Section 4.3.1 at (a) roughly step 20000; (b) roughly step 25000; (c) roughly step 30000.

Although the results are barely an improvement when compared with those of the From_Prior model, largely due to the same issues discussed above, its results, from Table 4.3, are marginally better which in turn supports the notion that the introduced prior constraint is too straining for the model's learning process and hinders it instead of acting as a boon. Figure 4.46 shows the progression of the main error metrics, loss and mean IoU, for the evaluation stage.

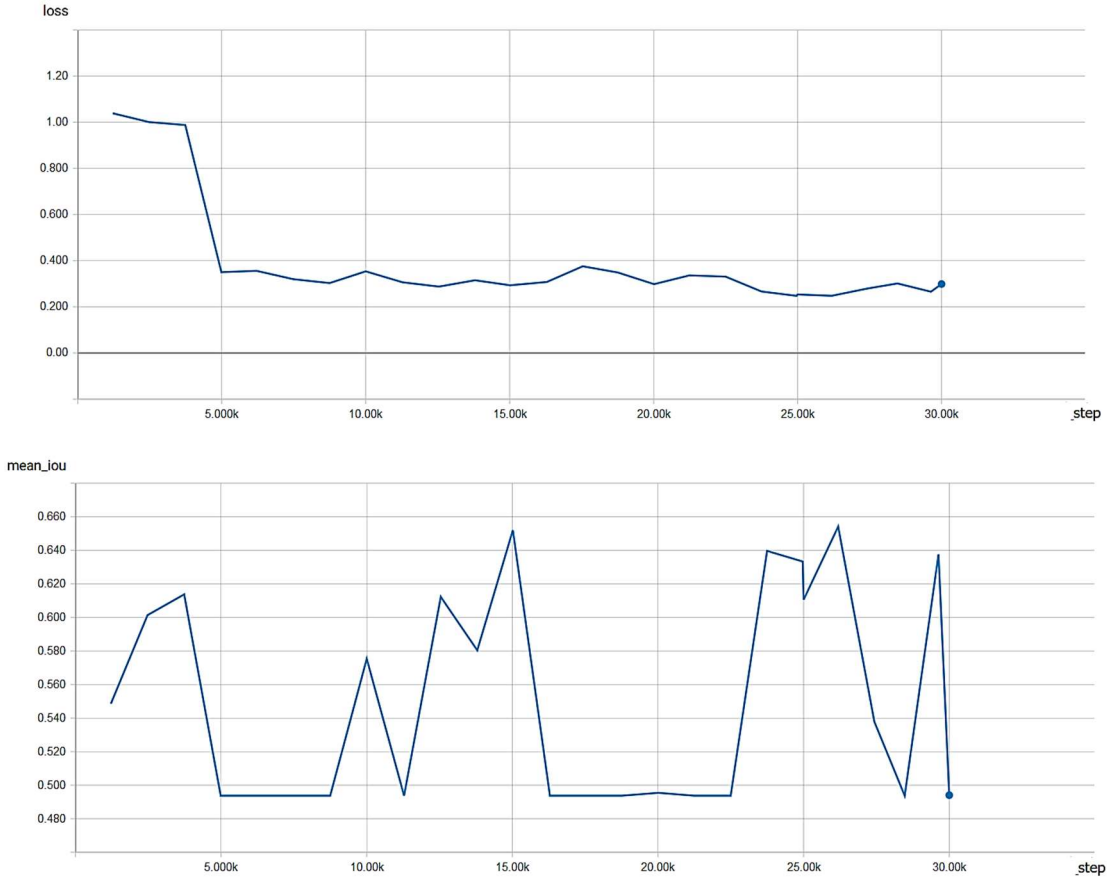


Figure 4.46: Error metric progression for VHLLRHe model's evaluation stage without imported weights.

Table 4.3: V-Net model performance.

Configurations	Loss	Mean IOU	Accuracy [%]	Precision [%]	Recall [%]	Steps
From_Prior	0,3024	0,4937	98,7450	0	0	20423
No_Prior	0,2983	0,4954	98,7487	89,0099	0,0336	20015
From_Prior	0,2432	0,6038	98,8343	57,8895	26,0979	25321
No_Prior	0,2537	0,6105	98,8211	55,9475	28,5110	25000
From_Prior	0,3084	0,4937	98,7450	0	0	30000
No_Prior	0,2989	0,4940	98,7458	94,3506	0,0066	30000
Dice_Lr3	0,1386	0,4937	98,7450	0	0	25000
Dice_Lr4	0,1386	0,4937	98,7450	0	0	25000
Dice_Lr5	0,9042	0,5825	90,4250	21,7931	72,7321	25000

As mentioned above the following set of models use the same conditions as the models already presented with the caveat of exchanging the objective function to the soft-dice loss as a standalone. Due to the poor all-round performance of the models, some variation of the learning rate was introduced, namely, 1×10^{-3} for the Dice_Lr3 model, 1×10^{-4} for the Dice_Lr4 model and 1×10^{-5} for the Dice_Lr5 model. An example of the results obtained at step 25000 can be seen in Figure 4.47.

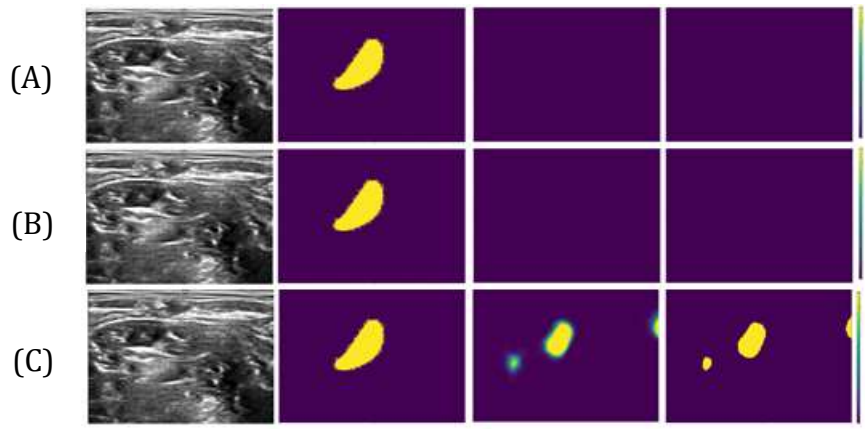
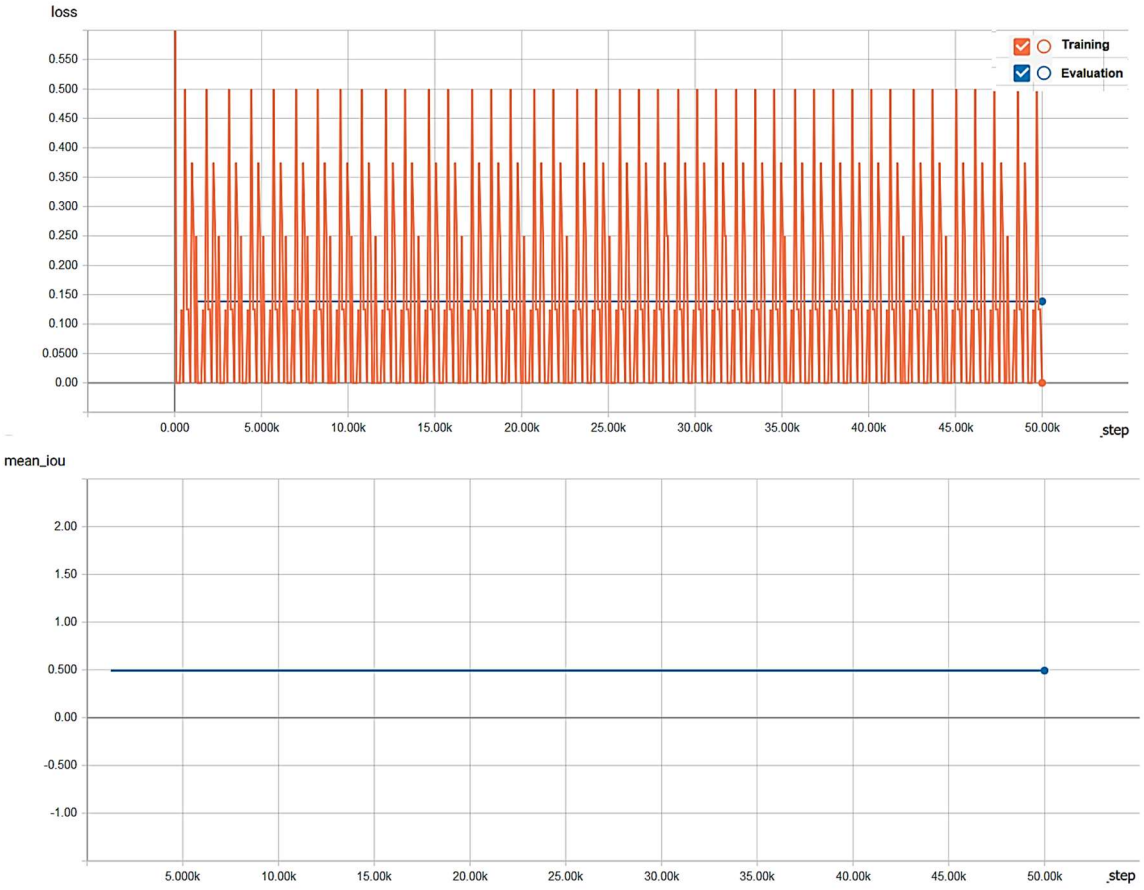


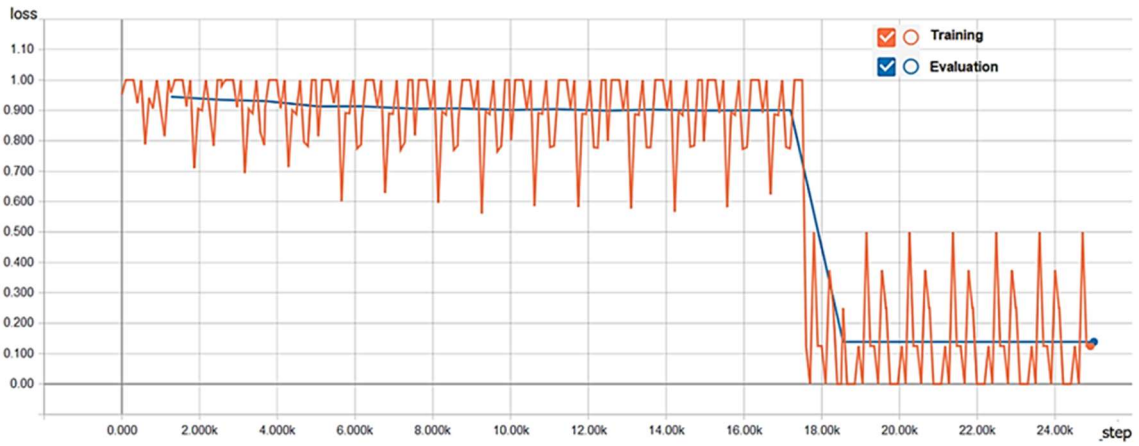
Figure 4.47: An input image and its corresponding ground-truth (left) side-by-side with the probabilistic prediction mask and binary prediction mask (right) outputted by the Dice_Lr models at step 25000. (a) Dice_Lr3; (b) Dice_Lr4; (c) Dice_Lr5.

Finally, the progression of loss and mean IoU metrics assessed at training and evaluation for all three models is displayed below on Figure 4.48, for 50000 steps in the case of Dice_Lr3 and 30000 steps for both Dice_Lr4 and Dice_Lr5.

(A)



(B)



(C)

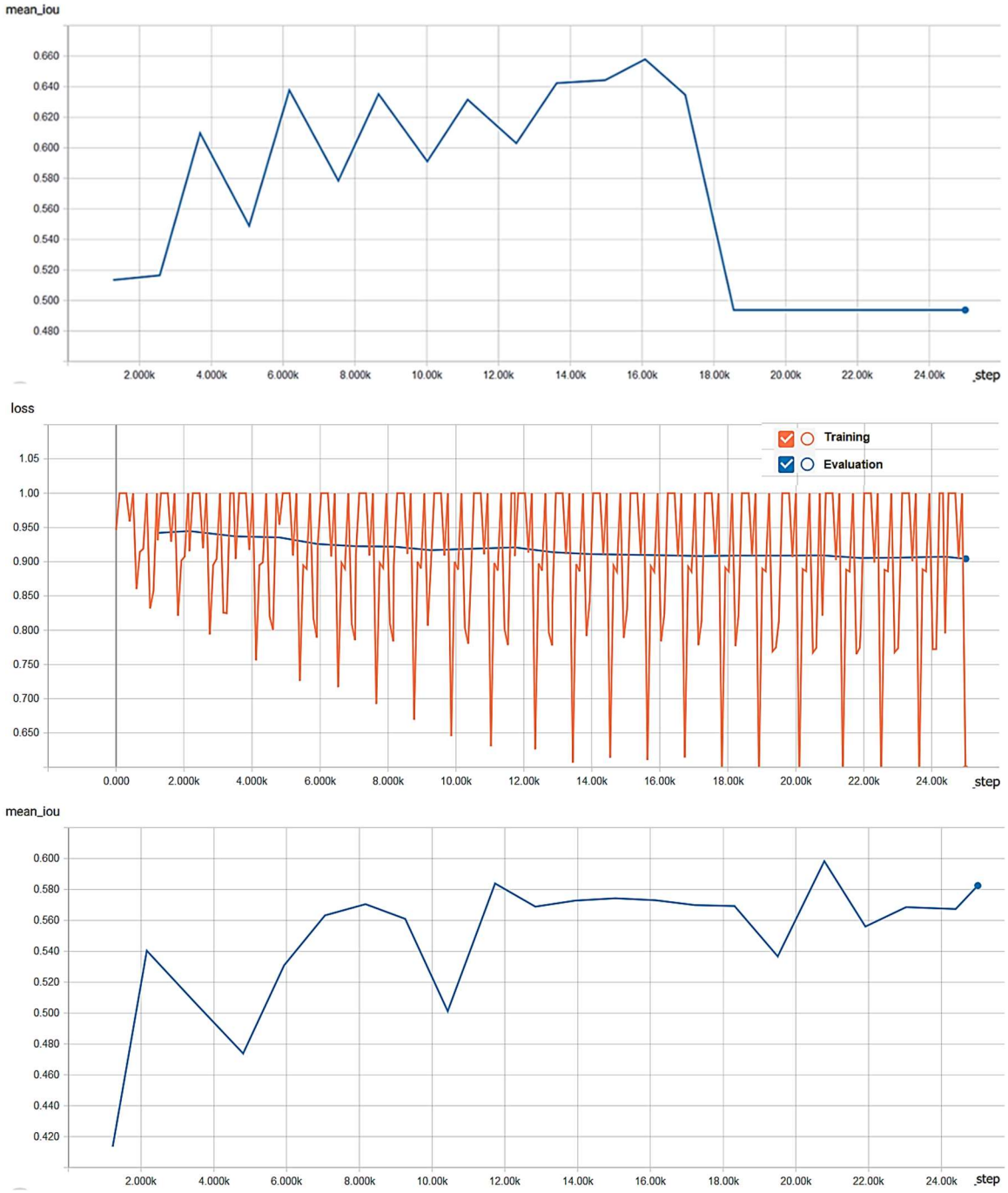


Figure 4.48: Error metric progression for the Dice_Lr models. (a) Dice_Lr3 with 50000 steps; (b) Dice_Lr4 with 25000 steps; (c) Dice_Lr5 with 25000 steps.

From these results and associated error metrics it is possible to infer that the learning rate value used as standard is excessive for this underlaying data. The Dice_Lr3 model quickly converged to a local minimum and got stuck there, which impaired its learning ability early on and throughout the whole 50000 steps. When compared with the Data Science Bowl 2018 challenge these models perform much worse, with the best mean IoU score of about 0.61 at 25000 steps for a hybrid loss model. To improve this performance scored, the spatial context provided by the use of third-dimensional samples as input as well as learning rate variations are the suggested starting point.

Chapter 5

Conclusions and Future Work

5.1 Summary

This thesis' objectives were essentially two-fold. The first was to attest to the reliability of two state-of-the-art FCNN architectures, the U-Net and the V-Net, in the binary classification task of two-dimensional medical visual data, effectively segmenting volumes of interest as belonging to the foreground of predictions masks, pixel-by-pixel, while analyzing the characteristics of each architecture and testing implementations of their variants. The second objective entailed bridging the gap between theoretical knowledge and practical implementation on the Tensorflow API, while taking into account the whole pipeline necessary to create a well performant deep learning algorithm, suitable to being run on a distributed computing setting.

To achieve these objectives, Chapter 2 first presented some of the more theoretical aspects of machine learning, particularly oriented towards deep learning, such as the task, estimators, supervised learning, optimization and regularization. Chapter 3 extended the concepts presented in Chapter 2 by providing a connection with the practical pipeline of a deep learning model and introducing the original FCNN, as well as the Tensorflow API toolset.

In the first portion of Chapter 4, the actual models, U-Net and V-Net, which inspired those used in this thesis were covered, with emphasis on their respective architectures. For the second portion of Chapter 4 the actual implementations were presented.

All implemented models were trained using the same pipeline, baring the portions which were under study. All variants of U-Net and V-Net used simple cropping and symmetrical mirroring-based augmentations, a binary erosion morphological filter, zero-padded convolutions, pixel normalization, up sampling layers based on deconvolution, initial weight masks for the ground-truth masks and the Adam optimizer with an initial learning rate of 1×10^{-4}. From the results displayed, after 3500 training steps in Section 4.2.1 and the several step milestones in 4.2.2, both for multiple runs, a number of conclusions can be drawn.

Dropping the pooling layer in favor of a down sampling convolutional layer drastically speeds up the convergence of the models while barely affecting the results themselves as, given enough time, the U-Net variants still score about the same when compared with the V-Net variants.

The residual function learning speeds up convergence times and appears to slightly improve the results obtained compared to the U-Net variants, corroborating the findings of the original authors of the V-Net. It also seems to limit overfitting, as through successive iterations of the different models it became noticeable that the disparity between the training and test error remained the same after stabilization for those models that used both residual propagation and layer skipping, while those that relied solely on skipping had a noticeable overfitting at 5000 training steps, although it was not

showcased on Section 4.2.1, due to disparity in training time between some models and the time constraints that did not allow for running the final version of all models up to 5000 steps a second time.

It is apparent that combining two different, differentiable, loss functions such as the binary cross-entropy loss and the dice coefficient loss can prove beneficial to a model. The reason lies with the objective improvement of the model's performance in terms of metric scores under the same conditions of those that used only one loss function, presumably due to the more diverse penalties.

Binary sigmoid cross-entropy loss penalizes any incorrect prediction of a pixel value the same way. For example, given a pixel that belongs to the foreground, if the probability of that pixel belonging to the foreground is below 0.5, the threshold defined, than the penalty will be the same as scoring it 0.4 or 0 probability and, likewise, if that probability is above or equal to 0.5 no penalty will be imposed. The dice-coefficient loss measures the similarity of each predicted mask with the corresponding ground-truth and heavily penalizes any diverging pixels, ideally until all probabilities are either 1 or 0 for each pixel in the predictions mask. In practice the use of the dice coefficient on its own proved to be lacking for two-dimensional inputs, perhaps due to a much slower convergence to a minimum while, on the other hand, binary cross-entropy loss on its own tended to converge faster but got stuck on the same solutions, local minima, after a number of steps.

The use of ReLu or leaky-ReLu activation functions as well as He or Xavier initialization did not appear to have a great impact performance-wise. This might be due to ill-conditioning of the models, the lack of enough training steps, or other factors. The activation functions' usage also differed from that of the original [7], which translates to more non-linearity after each convolution, while the original introduces said non-linearity after down sampling and up sampling.

The binary erosion morphological filter proved to be effective for the cell segmentation challenge, where several foreground objects tended to be present and bordering each other. In this challenge it consistently over eroded the borders of the objects of interest and that reflected itself on the final predictions mask, even after postprocessing dilation.

The basic data augmentation performed on the pipeline used for all models already amounts to a great score improvement compared to early versions of the implementations, which were not displayed on Chapter 4. Further and more complex augmentation should be made, such as that seen on Figure 2.8, particularly to improve generalization of the nuclei segmentation challenge which consistently scores worse for a particular type of cells whose number of representative samples is lacking compared with the others, and to improve the scores of the nerve segmentation challenge by simulating more neck region nerve structure examples under different conditions and dilations.

The learning rate used as standard, 1×10^{-4}, generally allowed for competitive performance metrics in cell nuclei segmentation, except for those models which rely solely on the soft-dice coefficient loss, as shown in Section 4.3.1. As for Section 4.3.2, the results overall were poor and a learning rate decrease of an order of magnitude for the soft-dice as sole objective function models led to significant improvements albeit at a much slower rate of progression. On the other hand, an increase of an order of magnitude in the learning rate led to a similar result as the model converged to a local minimum and kept going back to it, which effectively disabled the learning process of the model entirely.

Finally, it is important to mention that the lack of criteria to compare the different loss functions directly, the lack of more combinations of activation, initialization and loss functions as well as the limited training and validation sets used, which remained the same across all implementations, undermine the significance of the conclusions drawn from obtained results.

5.2 Future Work

First and foremost, to tackle the challenge of 4.2.2 the third-dimensional V-Net should be implemented, even if it is just the scaled version of the solution used, and a study similar to the one made in 4.2.1 should be made for the spatially scaled versions of each of the implemented variants. Besides that, it is of direct relevance to implement the U-Net and V-Net as they are proposed in their respective origin papers, down to the optimizers, to establish a comparison baseline for variation assessment.

Preprocessing was made in a rather crude manner, so a more sophisticated approach is necessary to ensure that further score improvements are attained. Some of these are, instead of storing the whole original dataset in memory, samples should be processed and augmented “on-the-fly”, the binary erosion morphological filter should be replaced by a less abrasive watershed morphological filter and data augmentation should be made much more comprehensive including, potentially, instances of noise addition, rotation, dilation, expansion, hue saturation and more.

In terms of hyperparameters, other variants should explore alternatives for optimizers, different initial learning rates, increased network depths, increased numbers of convolutional layers on each stage and the use of more feature maps.

Training and validation should be extended to include cross-validation, the loss functions used should be even more diversified, as stated before a broader range of learning rates needs to be experimented with and the early-stop criterion should be automated, based on the preferred error metrics for each challenge. It is also important to test the models on new data from a similar data generating source to attest to their generalization ability. Cross-validation in particular is required to provide statistical validation to the results obtained.

Finally, all model’s initial implementations were first tested on Google Cloud Machine Learning platform, until all trial credits were exhausted. From then onwards training was made, first, on a laptop with a Nvidia GeForce GTX860M GPU with two dedicated gigabits of memory, until it overloaded and broke down, and then on a laptop with a Nvidia GeForce GTX 1070 with eight dedicated gigabits of memory, which allowed relatively slow, albeit functional, training. Bearing this in mind it is also relevant to take full advantage of the distributed computing abilities provided by the Tensorflow API to train the model for many more steps, until the order of the tens of thousands of iterations when necessary, with larger resolutions per sample as well.

5.3 Bibliography

[1] - Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems* (pp. 1097-1105).

[2] - LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, *86*(11), 2278-2324.

[3] - Zeiler, M. D., & Fergus, R. (2014, September). Visualizing and understanding convolutional networks. In *European conference on computer vision* (pp. 818-833). Springer, Cham.

[4] - Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

[5] - He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 770-778).

[6] - Long, J., Shelhamer, E., & Darrell, T. (2015). Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 3431-3440).

[7] - Milletari, F., Navab, N., & Ahmadi, S. A. (2016, October). V-net: Fully convolutional neural networks for volumetric medical image segmentation. In *3D Vision (3DV), 2016 Fourth International Conference on* (pp. 565-571). IEEE.

[8] - Ronneberger, O., Fischer, P., & Brox, T. (2015, October). U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention* (pp. 234-241). Springer, Cham.

[9] - Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. MIT press.

[10] - Smith, L. N. (2017, March). Cyclical learning rates for training neural networks. In *Applications of Computer Vision (WACV), 2017 IEEE Winter Conference on* (pp. 464-472). IEEE.

[11] - Zeiler, M. D. (2012). ADADELTA: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*.

[12] - Ruder, S. (2016). An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*.

[13] - Springenberg, J. T., Dosovitskiy, A., Brox, T., & Riedmiller, M. (2014). Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806*.

[14] - LeCun, Y. A., Bottou, L., Orr, G. B., & Müller, K. R. (2012). Efficient backprop. In *Neural networks: Tricks of the trade* (pp. 9-48). Springer, Berlin, Heidelberg.

[15] - Ng, A. Y. (2004, July). Feature selection, L1 vs. L2 regularization, and rotational invariance. In *Proceedings of the twenty-first international conference on Machine learning* (p. 78). ACM.

[16] - Zhang, C., Bengio, S., Hardt, M., Recht, B., & Vinyals, O. (2016). Understanding deep learning requires rethinking generalization. *arXiv preprint arXiv:1611.03530*.

[17] - Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., ... & Bengio, Y. (2014). Generative adversarial nets. In *Advances in neural information processing systems* (pp. 2672-2680).

[18] - Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

[19] - Sutton, R. S. (1986). Two problems with backpropagation and other steepest-descent learning procedures for networks. *Proc. 8th Annual Conf. Cognitive Science Society.*

[20] - Duchi, J., Hazan, E., & Singer, Y. (2011). Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *Journal of Machine Learning Research*, 12, 2121–2159.

[21] - Richard HR Hahnloser, Rahul Sarpeshkar, Misha A Mahowald, Rodney J Douglas, and H Sebastian Seung. Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit.Nature, 405(6789):947, 2000.

[22] - Glorot, X., & Bengio, Y. (2010, March). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics* (pp. 249-256).

[23] - He, K., Zhang, X., Ren, S., & Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision* (pp. 1026-1034).

[24] - Yu, D., Wang, H., Chen, P., & Wei, Z. (2014, October). Mixed pooling for convolutional neural networks. In *International Conference on Rough Sets and Knowledge Technology* (pp. 364-375). Springer, Cham.

[25] - Fidon, L., Li, W., Garcia-Peraza-Herrera, L. C., Ekanayake, J., Kitchen, N., Ourselin, S., & Vercauteren, T. (2017, September). Generalised wasserstein dice score for imbalanced multi-class segmentation using holistic convolutional networks. In *International MICCAI Brainlesion Workshop* (pp. 64-76). Springer, Cham.

[26] - Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., ... & Bengio, Y. (2014). Generative adversarial nets. In *Advances in neural information processing systems* (pp. 2672-2680).

[27] - Valverde J., (2018). Difference between L1 and L2 regularization, implementation and visualization in Tensorflow.

[28] - Alexander, J., (2017). Image augmentation for machine learning experiments. https://github.com/aleju/imgaug [Retrieval date May 2019]

[29] - Dertat, A., (2017). Applied Deep Learning – Part 4: Convolution Neural Networks. https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2 [Retrieval date May 2019]

[30] - He, Z., Ding, X., Fang, C., & Wang, Y. (2014, March). Research on the face pattern space division in images based on their different views. In *Imaging and Multimedia Analytics in a Web and Mobile World 2014* (Vol. 9027, p. 90270M). International Society for Optics and Photonics.

[31] - Berhane, F., (2017). Autonomous driving Car detection. https://datascience-enthusiast.com/DL/Autonomous_driving_Car_detection.html [Retrieval date May 2019]