Automatic detection of Ionospheric Alfvén Resonances using U-net

Paolo Marangio

August 23, 2019

MSc in High Performance Computing with Data Science

The University of Edinburgh

Year of Presentation: 2019

Abstract

The occurrence of Ionospheric Alfvén Resonances (IARs) constitutes a fascinating but relatively unstudied phenomenon that is associated with vibrations of Earth's magnetic field lines as a result of their excitement by electric fields generated by lighting strikes. Using a cutting-edge set of equipment and data analysis method, the British Geological Survey (BGS) has collected over 6 years worth of data about this phenomenon. However, not only is this dataset difficult to analyse due to the presence of confounding datapoints, the employed data analysis method includes an image processing step that requires the tuning of several parameters and allows IARs signal detection with a low signal to noise ratio. In this dissertation project, the creation of a predictive model for the detection of IARs signal using the fully convolutional neural network U-net is proposed. U-net was able to achieve a low training loss without overfitting to a curated dataset of the IARs pheonomenon. The resulting model generated by the trained U-net was able to detect IARs signal while recognizing considerably less noise than the current data analysis method. To our knowledge, this constitutes the first account of the application of a neural network for the task of pattern recognition of unstructured image data about a global electromagnetic resonance phenomenon. BGS can now leverage this tool in order to analyse their vast dataset in a fully automated way.

Acknowledgements

I would like to thank my dissertation supervisor Dr. Rosa Filgueira and the industrial collaborators Dr. Ciaran Beggan and Dr. Vyron Christodoulou for their full engagement to the project, their support and guidance, and for all the knowledge they have shared from their respective domains of expertise during our collaboration. A special thanks also goes to the British Geological Survey, for letting me work on the data they have collected about the intriguing geomagnetic phenomenon of the Ionospheric Alfvén Resonances.

I am forever grateful to my family for their invaluable and unconditional support to my work and efforts throughout a challenging and stimulating academic year while being enrolled in the MSc in High Performance Computing with Data Science at the Edinburgh Parallel Computing Centre.

Contents

List of Figures

List of Tables

List of Abbreviations

Abbreviations

AI Artificial Intelligence

ANN Artificial Neural Network

BGS British Geological Survey

CNN Convolutional Neural Network

CPU Central Processing Unit

CV Computer Vision

DL Deep Learning

FCN Fully Convolutional Network

FFT Fast Fourier Transform

HPC High Performance Computing

IAR Ionospheric Alfvén Resonances

IoU Intersection over Union

ISBI International Symposium on Biomedical Imaging

LOOCV Leave-one-out cross-validation

ML Machine Learning

PR Pattern Recognition

ReLu Rectified Linear Unit

SR Schumann Resonances

TPU Tensor Processing Unit

Chapter 1

Introduction

The aim of this manuscript is to give an overview of the scope and the current state of the MSc dissertation project named *Automatic detection of Ionospheric Alfvén Resonances using U-net*, following the work conducted between May and August 2019. The industrial partner for the project is the British Geological Survey (BGS), a partly publicly funded body formed in 1835 that aims to advance the geoscientific knowledge of the United Kingdom by means of surveying, monitoring and research. The report is laid out as follows: Chapter 2 will introduce the background of the geophysical problem being investigated and the associated data analysis method; Chapter 3 will delineate a literature review of Deep Learning and its application in the field of Computer Vision while considering its relevance to the project; Chapter 4 will describe the project goals that motivated this piece of work along with the proposed methodology that was employed while carrying out the project with respect to Machine Learning (ML) and High Performance Computing (HPC) type of activities, respectively; Chapters 5 and 6 will present the results of the conducted experiments that fall into ML or HPC, respectively; Chapter 7 will be a discussion of the limitations of the approach used, strengths and weaknesses of the developed data analysis tool and ideas for potential, further work; finally, Chapter 8 will conclude the manuscript by summarizing the key findings and considering the outreach of the project.

Chapter 2

Project Background

In this Chapter, the geophysical phenomenon of the Ionospheric Alfvén Resonances (IARs) is first introduced. This phenomenon falls into the category of global electromagnetic resonances, which have been formally researched for the first time in 1952 by Winfried Otto Schumann (see Besser [10] for a review of Schumann's life, his work and the historical development of the main ideas leading to the formulation of the hypothesis of Earth-ionosphere cavity electromagnetic oscillations). As with any scientific endeavour, advanced data collection and analysis methods are crucial towards fully appreciating the causes of such natural phenomena. Geomagnetic research is definitely important to BGS. Their vast data and deep expertise about such phenomena can be used in order to develop a further scientific understanding of both the interior of the planet and space [11]. In turn, this information can be put into use for navigation products, magnetic charts and for servicing oil operators. In the remainder part of the chapter, the data on the IARs phenomenon collected over the past 7 years and the current data analysis method employed by BGS are reviewed.

2.1 The phenomenon of Ionospheric Alfvén Resonances

The Earth, with its 40,000 km in circumference and conductive material present in its core, creates a magnetic field that stretches for thousands of kilometres outwards into space. This magnetic field produces a magnetic 'bubble' known as the magnetosphere (Fig. 2.1) that shields the Earth from a broad range of radiation sources, including particle radiation emitted from the sun.

Induction coil magnetometers are one instrument that can be used in order to measure very small and rapid changes in the Earth's magnetic field. In particular, it is possible to observe very weak yet repeating patterns when the Earth's magnetic field is examined in the range between 1 and 50 times per second (Hz). Several lines of proof suggest that at least a portion of these patterns can be attributed to Earth's magnetic field line changes induced by lightning strikes in near-equator thunderstorms [62, 72].

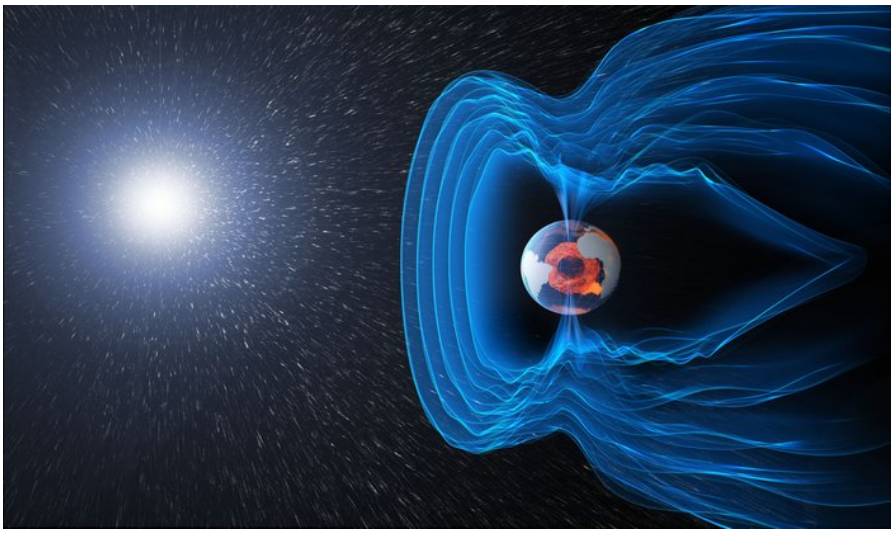

Figure 2.1: **Earth's magnetosphere**. The Earth is surrounded by a large 'bubble' called the magnetosphere (shown by the blue lines extending from the Earth into space), which is part of a dynamic system that protects the planet from solar, planetary, and interstellar conditions (shown by white particles flowing towards the blue lines). Fig. taken from [21].

In September 2012, BGS installed two induction coil magnetometers at the Eskdalemuir Observatory (Fig. 2.2), which is situated in a region of open moorland surrounded by conifer forests in the Southern Uplands of Scotland. Its remote location and the absence of mobile phone transmitters or antennae for other types of signal make this an electromagnetically quiet region of the UK. Since then, fluctuations in the Earth's magnetic field in the range between 0.1 and 50 Hz have been recorded by these devices. This range includes the extremely low frequency band of the electromagnetic spectrum. This band encompasses global magnetospheric phenomena such as the Schumann resonances (SRs) and ionospheric Alfvén resonances (IARs). SRs arise due to the partial reflectance of the EM energy (i.e. radio waves) that is emitted during lightning strikes by an atmospheric layer called ionosphere. Because of the chemical composition of this atmospheric layer, this reflectance effect is particularly effective during the day, and due to the cavity-like shape of this layer, it results in the generation of a repeating signal with a wavelength approximately divisible by Earth's circumference [5]. In a similar fashion as a resonating bell or the rippling effect caused by throwing a stone into a pond, the lightning strikes essentially produces radio waves that are powerful enough to 'echo' several times around the world. In particular, SRs constitute a set of fixed band harmonics formed at frequencies around $f_n = c/2\pi\alpha \cdot \sqrt{n(n+1)}$ (where n represents the harmonic, c denotes the speed of light, and a is the radius of the Earth) that have originated from a localized region of Earth where generally multiple lighting strikes have occurred [5].

IARs constitute a largely unstudied set of resonances in the frequency range between 0.5 and 10 Hz that are predominantly recognizable during the night and exhibit seasonal variation, as they are most common in winter [62, 67]. A simplistic account of IARs is that they represent vibrations of Earth's magnetic field lines (Fig. 2.3) which are excited

Figure 2.2: **Stages of high frequency induction coil installation at Eskdalamuir Observatory**. Left picture shows an induction coil before installation. A single induction coil consists of two coils of insulated copper wire wound around an iron core. Top right picture shows an induction coil during installation. The induction coil is placed on a levelled bed of gravel and linked to a break-out box and digitiser. Bottom right picture shows an induction coil after installation. The connected induction coil is positioned underneath some wooden covering, whose purpose is to protect the induction coil from environmental conditions (e.g. rain, wind and snow). Data are passed through a digitiser and delivered via the internet to BGS's office in Edinburgh on an hourly basis. Fig. taken from [11].

by electric fields from lightning strikes. These vibrations on the magnetic field lines leak through the ionosphere, then pass up from the lower ionosphere (≈100 km) (shown as blue dotted circle around the Earth in Fig. 2.3) to a second reflective layer (≈1000 km) (shown as magenta dotted lines) where they bounce back down and are then detected on the surface of the Earth. Unlike SRs, which are essentially made up of radio waves, IARs actually constitute Alfvén waves, namely magnetohydrodynamic waves that propagate along magnetic field lines [2]. Due to the excitation by electric fields from lighting strikes, the geomagnetic field lines (shown as blue arrowed lines) vibrate at a number of different frequencies that are harmonics of each other, leading to the repeating IARs signal (shown as small, repeating orange lines within the geomagnetic field lines) detected on Earth (Fig. 2.3). IARs have a baseline frequency that is dependent on the Alfvén speed, which is calculated as $V_A = B/\sqrt{\mu_0 \rho}$ (where B is the magnetic field strength, ρ is the plasma mass density and μ_0 is the permeability of the vacuum along the magnetic field line) [58].

IARs are an intriguing geophysical phenomenon and as such have been closely researched and modelled in order to fully understand their occurrence. Most importantly, several lines of evidence have shown that in combination with ionosonde data (i.e. data generated from a special radar used for the examination of the ionosphere), some parameters that can be extracted from IARs data (such as the frequency interval between resonances) can be used to determine the effective density of the ionosphere [28, 67]. In turn, any information that can be collected about this

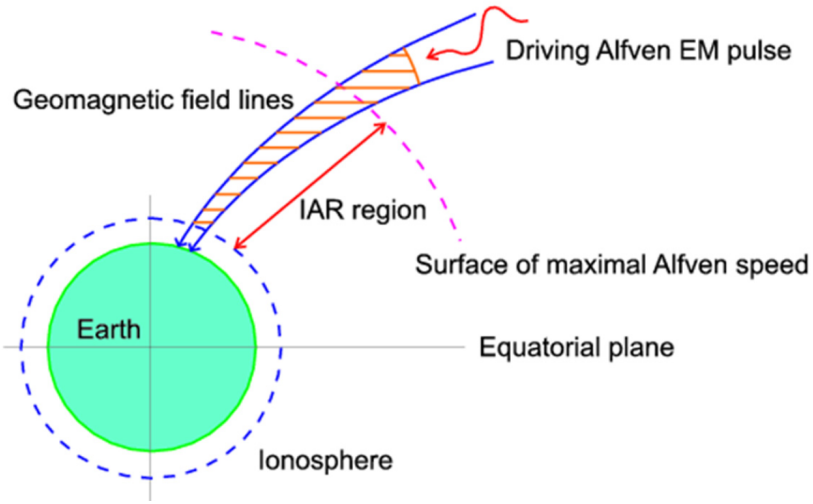

Figure 2.3: **Diagram of the Ionospheric Alfvén Resonanaces (IARs) phenomenon**. IARs occur between the lower bound of the ionosphere (marked by blue dotted lines) at an altitude of 100 km and an additional layer (marked by magenta dotted lines) ≈1000 km away from the Earth, which is the registered location of the peak in velocity at which the Alfvén waves travel away or towards the Earth. Fig. taken from [58].

difficult-to-access atmospheric region could be used by geoscience research centers like BGS to reliably monitor worldwide lightning storms and associated ionospheric events, and, more generally, to enable more accurate modeling of GPS signal travel through the ionosphere.

2.2 Current data analysis method and dataset available

The induction coil magnetometers at Eskadalemuir Observatory have been running with only a few interruptions since their installation in 2012. The extraction of useful parameters from the raw data generated from these instruments is not only time-consuming, but it also requires a high level of manual intervention. BGS has developed a method for the automatic detection of IARs-associated signal based on a mixture of signal and image processing techniques coded in Matlab. The code for the data analysis method has been made available by BGS at the beginning of the dissertation project and was executed in order to understand its parts. In this Section, the method used to extract the IARs signal and associated parameters presented in Beggan [4] is described in detail. The entire data analysis method can be broadly divided into two functional steps: image generation and detection.

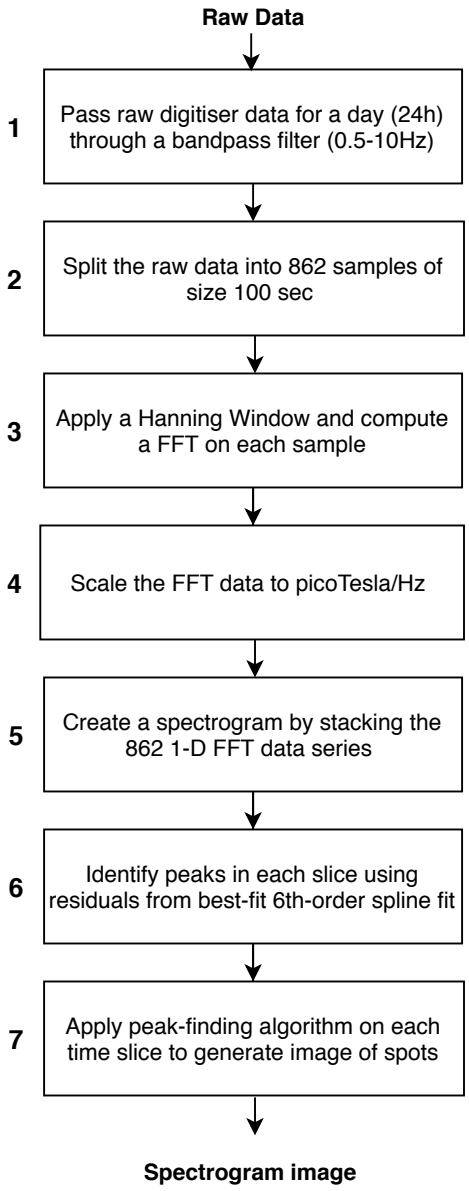

Figure 2.4: **Breakdown of the steps associated with the Image Generation part of the pipeline**. This picture is a diagrammatic summary of the relevant information published in [4].

2.2.1 Image generation part

The image generation part (Fig. 2.4) of the data analysis method takes in raw digital data generated from induction coil magnetometers and generates a spectrogram displaying the pattern of geomagnetic signal (quantified in picoTesla per square root of Hertz) over the frequency spectrum as it varies with time.

The two horizontal induction coil magnetometers installed at the Eskdalemuir Observatory are orientated such that they are able to detect north-south and east-west magnetic field lines vibrations, respectively. They are both connected to a 24 bit Guralp

digitiser, which converts the output signal from the coils, namely analogue voltage, into digitiser count.

These data are then bandpass filtered between 0.5 and 10 Hz using a five-pole Butterworth filter in the time domain (step 1 in Fig. 2.4 and panel a in Fig. 2.5) in order to remove longer period variations.

Given than there are 86400 seconds in a day, the entire dataset for a given day can be split into 864 slices of 100 seconds (step 2 in Fig. 2.4). Since the induction coils sample the magnetic field 100 times per second (i.e. 100 Hz), each slice consists of 100 samples for 100 seconds and a whole day worth of data therefore consists of a total of 8,640,000 samples. The choice of 100 seconds for the size of the slice is somewhat arbitrary and it is a good trade-off between total number of samples for a whole day and time resolution of each sample.

Subsequently, a Hanning window is applied to the data in order to gradually decrease the edges to 0 before a Fast Fourier Transform (FFT) is computed (step 3 in Fig. 2.4) using the Welch method. The FFT converts the signal from the time domain to a representation in the frequency domain. Unavoidably, 100 seconds are lost at the start and the end of the day due to the way in which the Welch method is implemented. This is the reason why the final spectrogram generated by the image generation part consists of 862 samples rather than 864.

The FFT data are then scaled to a unit that is consistent with the International System of Units by taking into account the scaling and calibration factors of the digitiser and induction coil frequency response (step 4 in Fig. 2.4 and panel b in Fig. 2.5). The digitiser essentially converts an analogue voltage to a digital number and by identifying the number of counts that are between 0 and 1 Volt, the data can be converted from digitiser units to count per volt. On the other hand, induction coils are calibrated in Volts per nanoTesla but in the frequency domain, with a response around 50 nanoTesla per microVolt between 0.1 and 50 Hz. Hence, after the FFT is performed (and the data are now in the frequency domain), the data in units of volts/Hz can be converted to nanoTesla/Hz.

A spectrogram can be then created by putting together the outputs of the 862 one-dimensional FFT traces into a two-dimensional matrix (step 5 in Fig. 2.4). Any peaks present in each of these traces can be identified by first using the residuals from a best-fit sixth-order spline fit to remove background trend (step 6 in Fig. 2.4), followed by a peak-finding algorithm (step 7 in Fig. 2.4 and panel c in Fig. 2.5). The position of the peaks from each time slice is recorded in a separate matrix that is now treated as an image. Given that the FFT used has a length of 4096 points for each 100 Hz sample, the frequency resolution for every point is 1/40.96 Hz. Assuming that the frequency region of interest to be displayed is between 0.5 and 10.5 Hz (giving a total of 10 Hz) and considering that the frequency resolution can be approximated to 41 points per Hz, only the required 410 points (i.e. 41 points per Hz $\times$ 10 Hz) are extracted from the FFT. This gives a final image size of 862 rows by 410 columns (panel a in Fig. 2.7).

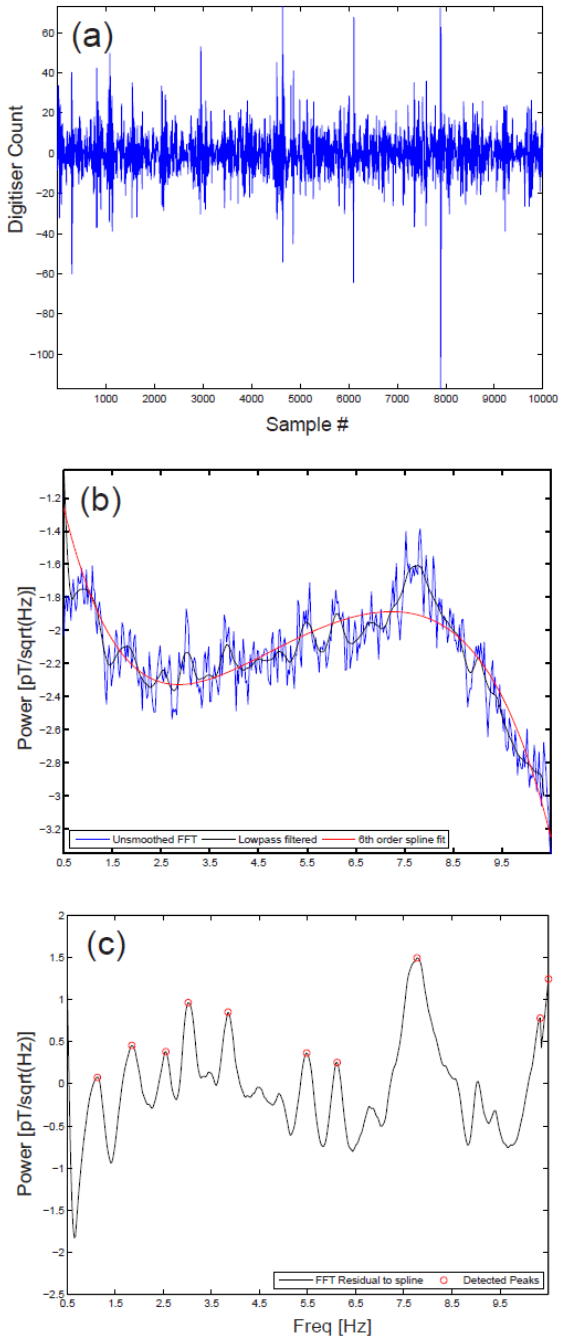

Figure 2.5: **Detailed representation of some of the steps from image generation part of the data analysis method**. (**a**) 100 seconds of bandpass-filtered input data; (**b**) FFT performed on a sample of Hanning-windowed data shown in blue, FFT data smoothed with lowpass filter shown in black and sixth-order spline line fitted to the data shown in red; (**c**) residual from spline background line fitted to the data shown in black, with main peaks identified with peak-finding algorithm shown as red circles. Fig. taken from [4].

2.2.2 Image detection part

The image detection part (Fig. 2.6) of the data analysis method computes the position of the IARs-associated signal from the spectrogram image generated from the image generation part. In particular, this part of the data analysis method is attempting to link together continuous peaks across FFT time slices and avoid sporadic peaks as much as possible.

First, the spots are dilated by an extrusion of 15 pixels in the vertical direction (step 1 in Fig. 2.6) and then eroded by a line reduction of 3 pixels in the horizontal direction (step 2 in Fig. 2.6). These two threshold values are chosen based on experimentation.

The connection of any two pixels that are touching each other after applying steps 1 and 2 is dilated across both dimensions (step 3 in Fig. 2.6). These regions can be dilated further to give a more refined position of different regions of IAR signal (step 4 in Fig. 2.6). The combined effect of steps 1-4 from Fig. 2.6 is shown in panel b in Fig. 2.7.

The image generated from step 4 in Fig. 2.6 (panel b in Fig. 2.7) is passed as argument to Matlab's built-in function *regionprops* (step 5 in Fig. 2.6). This function computes the area of any region and draws a bounding box that contains the region. If the area of the detected region is smaller than 80 pixels, then it is ignored (step 6a in Fig. 2.6). If the area is larger than or equal to 80 pixels, then the weighted mean of the pixels within the region according to the bounding box is computed (step 6b in Fig. 2.6 and panel c in Fig. 2.7). Again, these two threshold values applied for condition checking are chosen based on experimentation. Step 6a in Fig. 2.6 essentially identifies the position of strongest pixels in each row of the original 'image of spots' generated by the image generation part of the data analysis method (panel a in Fig. 2.7). These positions can be mapped as continuous lines on the original spectrogram image (step 7 in Fig. 2.6 and panel d in Fig. 2.7). This is also why in Fig. 2.6 the input original spectrogram image has an arrow pointing to step 5: this image is passed as a reference to the *regionprops* function, such that individual FFTs values are used directly in subsequent steps.

Panel d in Fig. 2.7 displays a colour scheme that essentially indicates the amount of picoTesla per square root of Hz for every one of the 410 points (from 0.5 to 10 Hz) for every FFT performed on each of the 862 time slices. Hence, this constitutes a complete representation of the detected IARs signal, as the continuous lines artificially generated in the previous steps are now complemented by values of the magnetic field strength.

2.2.3 Performance of current data analysis method

A single picture depicting the frequency measurements associated with a given calendar day can be subjected to the entire data analysis method by operating a single Matlab script (assuming Matlab's built-in *Image* and *Signal Processing* toolboxes were previously installed and mounted). This procedure is typically conducted by BGS in parallel in order to boost performance by using the in-house Linux blade servers that

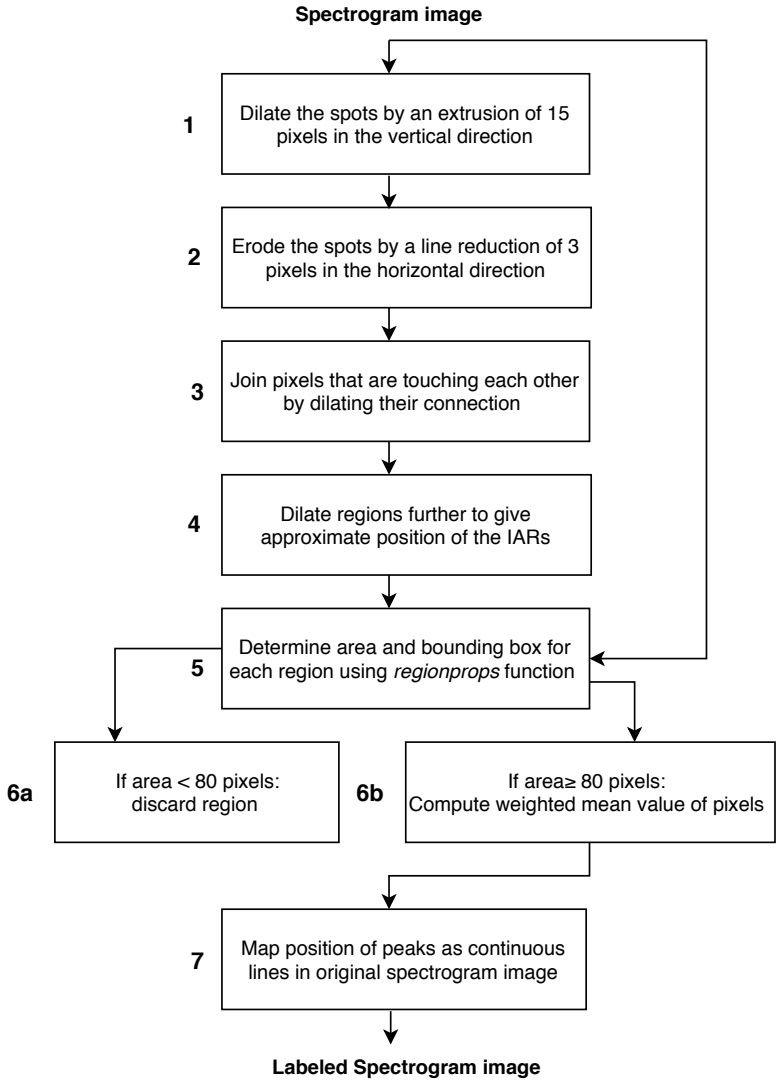

Figure 2.6: **Breakdown of the steps associated with the image detection part of the data analysis method**. This picture is a diagrammatic summary of the relevant information published in [4].

enable virtual clusters with up to 32 CPUs cores to be created. In particular, parallelism can be applied either at the level of processing of a single day worth of data (i.e. each CPU core processes a different day), or at the level of individual FFTs required to convert the time-series of magnetic field measurements into the spectrogram outputted at the end of the image generation part of the data analysis method (i.e. each CPU core performs a single FFT). Because a single picture requires about 20 seconds for it to be processed (including writing to disk), handling an entire year worth of data requires between 8 and 200 minutes, depending on the number of CPU cores used.

Beggan [4] originally employed the entire data analysis method on data associated with the 525 days between September 2012 and February 2014. Only 152 days out of the total were adequately labeled (assessed by visual inspection) using the proposed data analysis method. A spectrogram is defined as adequately labeled if some meaningful

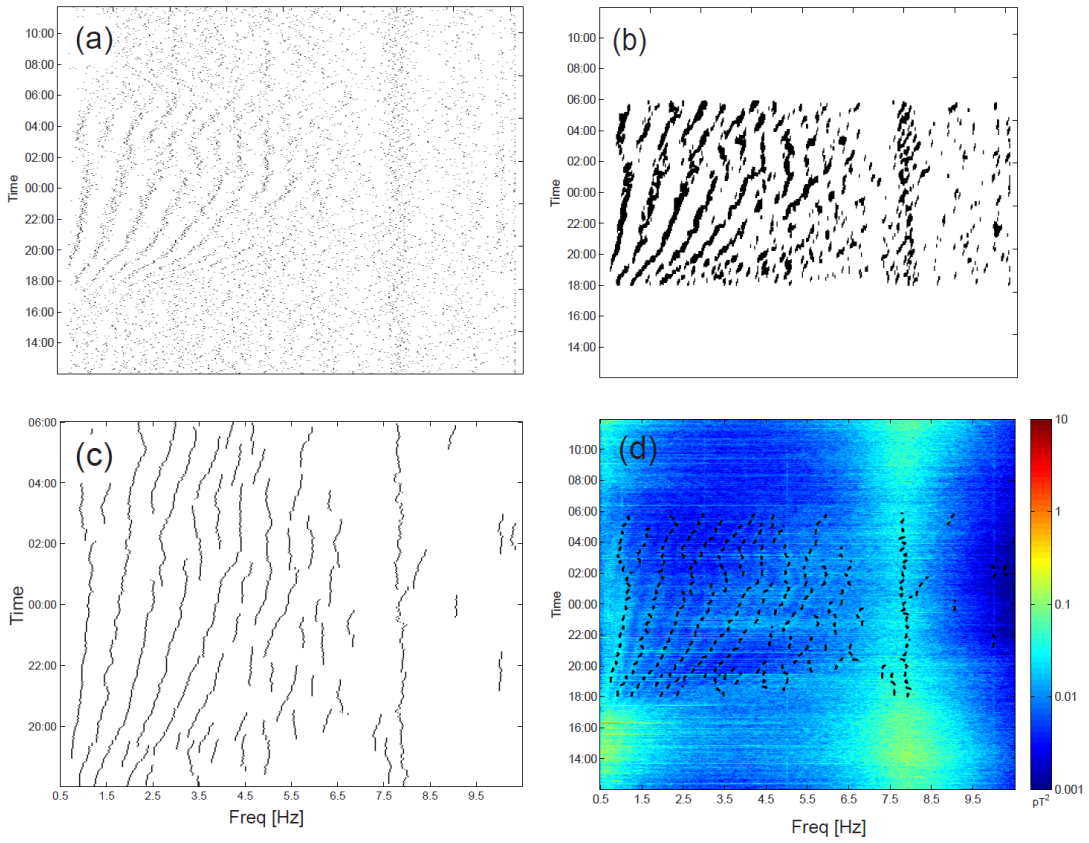

Figure 2.7: **Detailed representation of some of the steps from image detection part of the data analysis method**. (**a**). 'Image of spots' generated after identifying peaks in the smoothed FFT data; (**b**). Application of dilation, erosion and bridging of 'spots' from panel **a**; (**c**). Computation of weighted mean position of IARs signal; (**d**). Superposition of locations of IARs signal peak with original spectrogram image. Any signal outside 18:00 and 06:00 is ignored in panels **b**, **c**, and **d**. Fig. taken from [4].

IARs pattern can be seen in the spectrogram images following labeling with the data analysis method (region between 18:00 and 10:00, and between 0.5 and 5.5 Hz in panel b in Fig. 2.8). This outcome evidently demonstrates both the weaknesses of the method and the difficulties present in the dataset. As it can be seen from Fig. 2.8, not only is the method prone to detecting noise and therefore identify 'false positives' in some cases (panel c in Fig. 2.8), but the dataset itself also contains several days worth of data contaminated by elevated geomagnetic activity, local lightning storms or man-made interference (panel d in Fig. 2.8). It therefore appears that the current method cannot clearly distinguish between real signal (resulting in properly labeled spectrograms) and irrelevant or artificial signal (leading to poorly labeled spectrograms).

It is also worth noting that the initial dataset has now quadrupled, as it currently spans the 2313 calendar days between September 2nd 2012 and January 1st 2019. Although the method exhibits some strengths and is still reasonably robust to noise (tuning of threshold parameters is still required), a complete and accurate identification of IARs-associated signal does not seem to be feasible at present. The aim of the project

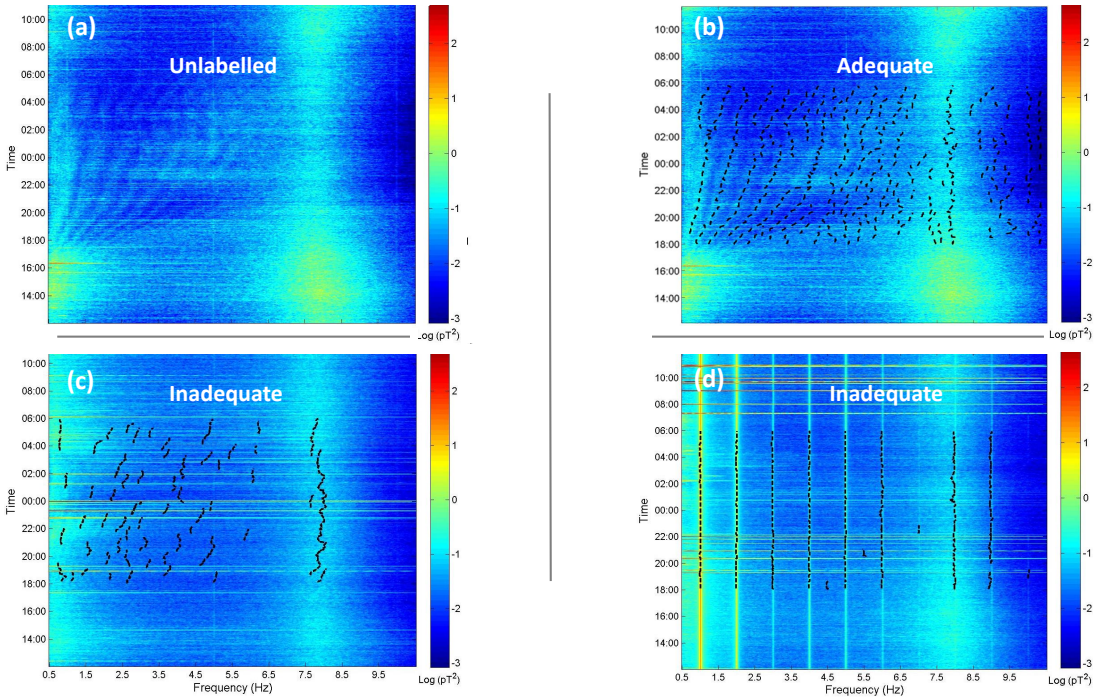

Figure 2.8: **Performance of current data analysis method**. (**a**). Unlabeled spectrogram image showing IARs-associated signal as alternating darker and brighter blue patterns occurring between 18:00 and 06:00. (**b**). Adequate spectrogram labelling instance, as label (black dotted lines) co-locates with IARs-associated signal (see image **a**), as well as with noise at frequencies greater than ≈7.5 Hz. (**c**). Inadequate example of spectrogram labelling, because a label has been assigned to seemingly random positions in the image where there is no IARs-associated signal. (**d**). Inadequate example of spectrogram labelling, because even if most of the vertical lines have been labeled, the vertical lines themselves are associated with man-made interference (e.g. electric fence) and therefore do not represent IARs-associated signal. Figs. have been kindly provided by Dr. Beggan.

is therefore to improve upon the current detection of IARs-associated signal using ML in the interest of offering a viable solution to BGS's data analysis problem.

Chapter 3

Literature Review

As a branch of engineering, pattern recognition (PR) is concerned with "*the automatic discovery of regularities in data through the use of computer algorithms and with the use of these regularities to take actions such as classifying the data into different categories*" [12]. The current data analysis method developed by BGS constitutes an example of the application of PR for the analysis of IARs data. In particular, as clearly detailed in Section 2.2, the data analysis method includes both hand-crafted rules and heuristics (e.g. thresholds for peak-finding algorithm, size of dilation and severity of erosion). An alternative approach to PR is Machine Learning (ML), which is concerned with the automation of learning using mathematics and statistics. It can be argued that using a ML algorithm it could be possible to replace the image detection part of the current data analysis method with a tool that is capable of robustly identifying the 'fringe' patterns in spectrograms while obviating the need of setting any thresholds. In this chapter, the field and practice of ML is introduced, while placing particular emphasis on the ML method known as Deep Learning.

3.1 Machine Learning and Deep Learning: a historical introduction

Arthur Samuel, a pioneer in artificial intelligence (AI) who worked at IBM and Stanford, defined machine learning as follows:

"*The field of study that gives computers the ability to learn without being explicitly programmed.*" [66]

Samuel created an ingenious piece of software that simulated a checkers game and was able to tune its strategy by associating different arrangements of the board with the probability of winning and losing. It is argued that this idea of searching for patterns and reinforcing the successful ones is still central to the current practice of ML [66].

Deep learning (DL) is a branch of ML that revolves around a type of algorithms inspired

Milestone/contribution	Contributor, year
MCP model, regarded as the ancestor of the Artificial Neural Network	McCulloch & Pitts, 1943
Hebbian learning rule	Hebb, 1949
First perceptron	Rosenblatt, 1958
Backpropagation	Werbos, 1974
Neocognitron, regarded as the ancestor of the Convolutional Neural Network	Fukushima, 1980
Boltzmann Machine	Ackley, Hinton & Sejnowski, 1985
Restricted Boltzmann Machine (initially known as Harmonium)	Smolensky, 1986
Recurrent Neural Network	Jordan, 1986
Autoencoders	Rumelhart, Hinton & Williams, 1986 Ballard, 1987
LeNet, starting the era of Convolutional Neural Networks	LeCun, 1990
LSTM	Hochreiter & Schmidhuber, 1997
Deep Belief Network, ushering the "age of deep learning"	Hinton, 2006
Deep Boltzmann Machine	Salakhutdinov & Hinton, 2009
AlexNet, starting the age of CNN used for ImageNet classification	Krizhevsky, Sutskever, & Hinton, 2012

Figure 3.1: **History of DL**. This table lists important milestones in the history of neural networks and Machine Learning, leading up to the era of Deep Learning. Fig. taken from [81].

by the structure and function of the brain called artificial neural networks (ANNs) [13]. The field of DL has recently undergone a major transition fueled by the enormous increase in computational power and, as a result, the earlier types of ANNs used for DL from the 1980s, loosely defined as ANNs with more than two layers [66], have evolved into networks with a large number of parameters that fall into one of the following categories: unsupervised pretrained networks, convolutional neural networks, recurrent neural networks and recursive neural networks.

Some of the characteristics that distinguish the modern categories of ANNs are: more neurons than previous networks; more complex ways of connecting layers and neurons; explosion in the amount of computing power available for performing training; and automatic feature extraction [66]. Moreover, among the most prominent factors that contributed to the huge boost of deep learning is the appearance of large, high-quality, publicly available labelled datasets, such as MNIST [46], ImageNet [19] and COCO [50]. One additional factor that played a prominent role was the empowerment of parallel GPU computing, which enabled the transition from CPU-based to GPU-based training and, as a result, allowed for significant acceleration in the training of neural networks (Fig. 3.1).

3.2 Deep Learning: a mathematical introduction

3.2.1 Neurons, layers and vectorisation

DL is concerned with the formulation of computational models that include multiple, successive processing layers, which allows to represent data using an abstraction hierarchy. DL and in particular neural networks are inspired by how the brain processes complex information [57, 48].

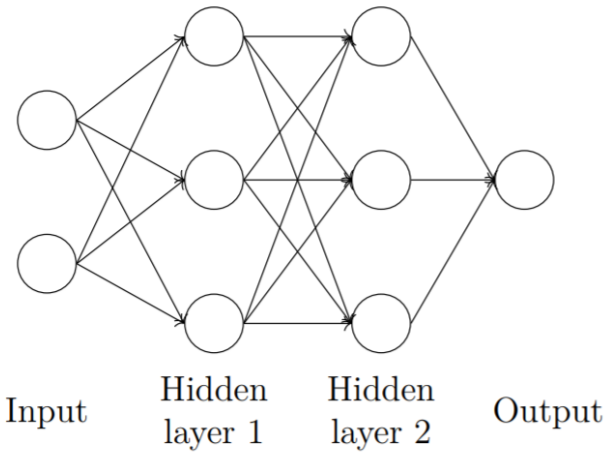

Figure 3.2: **Structure of a standard neural network**. The architecture for a typical neural network, with an input layer followed by two fully connected, hidden layers, and a single output, is shown.

Neural networks can be simply understood as an input, one or more hidden layers, and an output layer (Fig. 3.2). Typically the input is not regarded as a layer per se as it does not perform any computations. A layer is made up of neurons and different neurons from different layers are connected together such that information can flow between such layers. In the neural network shown in Fig. 3.2, layers are fully connected to each other. It is worth noting that modern types of ANNs exhibit variations of this pattern that make them more efficient and/or suitable to the tasks they are designed for.

The function of a single neuron is to receive a vector x as input, which represents the values of the features associated with a single training example, and compute a single value $\hat{y}$ (Fig. 3.3). This function can be broken down into further steps that are performed at each iteration of the neural network algorithm: calculating a weighted sum of the values of the vector x using a (row) vector of weights w; adding a bias b to it; subjecting this weighted average to a non-linear activation function g.

The notation for the mathematical operation carried out by a single neuron can be generalized to the entire layer (Fig. 3.4) by replacing x, which is the input to a single neuron, with a, namely the activations of all the neurons in a previous layer. It is acceptable to do this because in a network made up of fully-connected layers like the one shown in Fig. 3.2, a neuron in a given layer is fully connected to all the neurons from the previous layer. This notation allows the mathematical operations carried out by all the neurons in a given layer to be represented as a system of equations (Fig. 3.5).

Moreover, the mathematical expression shown in Fig. 3.5 can be conveniently converted into a matrix equation by stacking all the row vectors of the weights and the biases belonging to each neuron from a given layer into matrix W and column vector b, respectively (Fig. 3.6). By performing this vectorisation, it is possible to carry out the calculation of the activations for all the neurons in a layer at once.

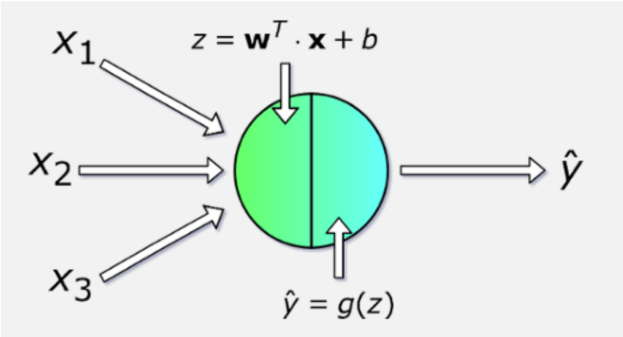

Figure 3.3: **Activation of a single neuron**. The function of a single neuron is to receive a (column) vector x of values as input, which represents the values of the features associated with a single training example, and compute a single value $\hat{y}$. x subscript 1 to 3 indicate different values for the different features. w superscript T indicates the transposed (row) vector of weights used to calculate the weighted sum of the values of x. b is a single value for the bias of the neuron and is added to the weighted sum in order to calculate z. g is a mathematical function that takes the computed value of z and outputs $\hat{y}$. Fig. taken from [74].

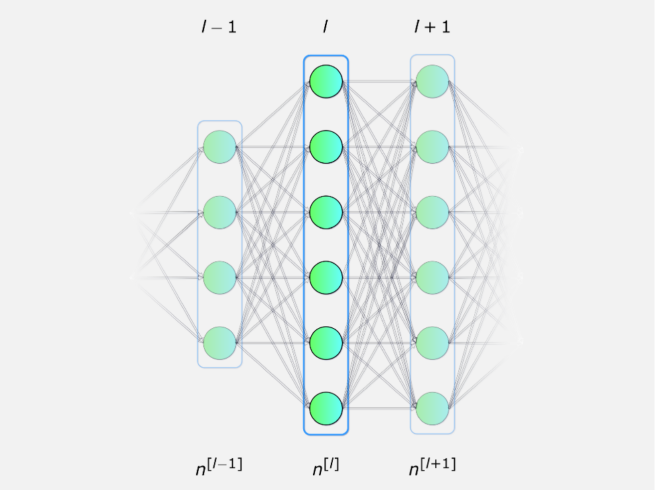

Figure 3.4: **Neurons grouped into multiple, successive, hidden layers**. Neurons are grouped into layers (blue box), which are indexed by l. N superscript l indicates the number of neurons within a given layer. Fig. taken from [74].

3.2.2 Activation function

The activation function is a key element of neural networks, as it allows to typically introduce non-linearity within different layers of the network (see Nwankpa et al. [64] for a review of the existing activation functions used in DL applications). In turn, this confers greater flexibility and creation of complex data-processing behaviours by the network. The activation function is also know to affect speed of learning. The two most commonly used functions are the Rectified Linear Unit (ReLu) for hidden layers [59], and sigmoid for the output layer [27].

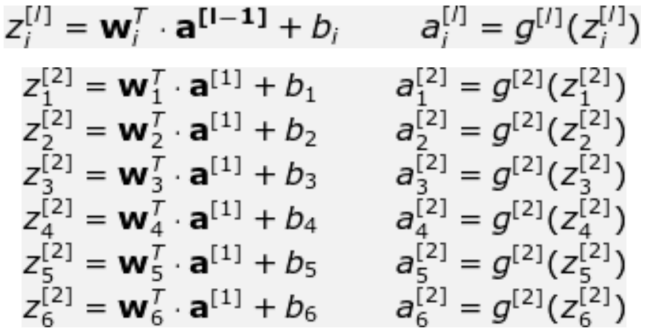

Figure 3.5: **General notation for activation of a single layer**. The notation for the activation of a given, single layer is given by equations on the top. Each weight vector and bias value associated with each neuron in a given layer is indexed with *i* subscript. For the sake of clarity, the individual equations for the activation of the 6 neurons from the second hidden layer from Fig. 3.4 are shown below the the general equations for the activation of a layer. *z* superscript *l* and subscript *i* computes the weighted sum of inputs from the previous layer *l-1* (i.e. first layer) for a given neuron *i* within a given layer *l* and then adds the bias **b** associated with that neuron. *a* superscript *l* and subscript *i* (equations on bottom right) calculates the activation for a given neuron *i* within a given layer *l* after applying the activation function *g* to the corresponding weighted sum *z* calculated for the neuron (equations on bottom left). Fig. taken from [74].

3.2.3 Loss function

The loss function is another key element of neural networks that signals how far the neural network is from an ideal solution (see Janocha and Czarnecki [33] for a review of different loss functions and how they affect DL models). Neural network models learn a mapping from inputs to outputs from examples and the choice of loss function must match the framing of the specific predictive modeling problem, such as classification or regression. The most common loss functions for these types of tasks are the mean squared error (also known as L_2 loss) and binary cross-entropy (also known as log loss), respectively. The choice of loss function for this project is reviewed in more detail in Section 4.2.3.

3.2.4 Learning process

After having expressed the relationships between all the layers of a neural network as matrix-vector multiplications and having defined a loss function, the goal of DL can be understood as tuning the values of the W and b parameters for each layer such that the loss function is minimized. This optimization task is achieved using a method called stochastic gradient descent (SGD) [41], whereby at each iteration the values of the loss function partial derivatives with respect to each of the parameters of the neural network are calculated.

Backpropagation is an algorithm that allows to compute the negative gradient of the loss function, which in turn determines the required changes in the weights and biases as to most efficiently minimize the loss function [71].

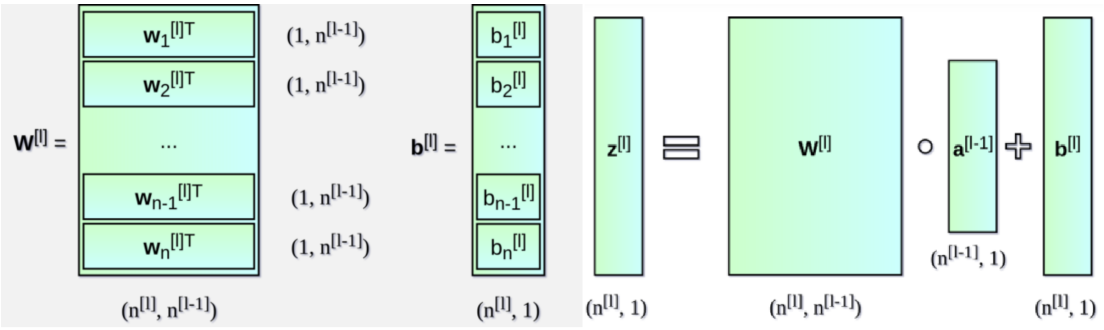

Figure 3.6: **vectorisation of the calculation of activation for each layer**. The calculation of the activation for each layer can be sped up by using vectorisation. (**Left**). The horizontal, transposed vectors of weights *w* (each of size 1, *n* superscript *l-1*) are stacked together to build matrix *W* (of size *n* superscript *l*, *n* superscript *l-1*). Similarly, the individual biases of each neuron in the layer are stacked together, thereby creating column vector *b* (of size *n* superscript *l*, 1). (**Right**). The activations of the individual neurons from the previous layer are stacked together, thereby creating column vector *a* (of size *n* superscript *l-1*, 1). Now, the calculations for all the neurons of the layer can be performed at once as a single matrix-vector multiplication between the weight matrix *W* and the activation column vector *a* from the previous layer followed by addition of bias column vector *b*. Fig. taken from [74].

3.3 Computer Vision and Image Segmentation

Computer vision (CV) is an interdisciplinary scientific field that encompasses how computers can be used to generate a high-level understanding from digital images or videos. Put in an engineering perspective, it aims to automate the functions that the human visual system can perform already [44].

Since its early days, CV was found to have a strong overlap with AI. In particular, certain area of AI were and still are concerned with autonomous decision-making for robotic systems. As such, CV systems could be used to help provide a detailed, high-level understanding of the environment and the robot itself such that the robot would be capable of navigating through it.

DL has recently been a driving force behind the huge progress made while tackling a variety of CV problems, such as object detection, motion tracking, action recognition, human pose estimation and semantic segmentation [81]. The results on these applications have been so promising that the whole field of CV is shifting entirely towards being DL-based, while relinquishing the need of pipelines of specialized, hand-crafted methods.

In particular, semantic segmentation is an advanced form of CV task that aims to both classify and localize all the pixels in a given image or video as to form clusters of pixels that have semantic meaning (Fig. 3.7). A parallel can be clearly drawn between the task of semantic segmentation within the context of CV and the application of PR for the analysis of IARs data. The objective of the image detection part of the data analysis method developed by BGS is to form clusters of pixels that have similar properties (i.e. pixels should be connected over the time domain, with little shift in the frequency domain) in the spectrogram image and label such clusters as IARs signal.

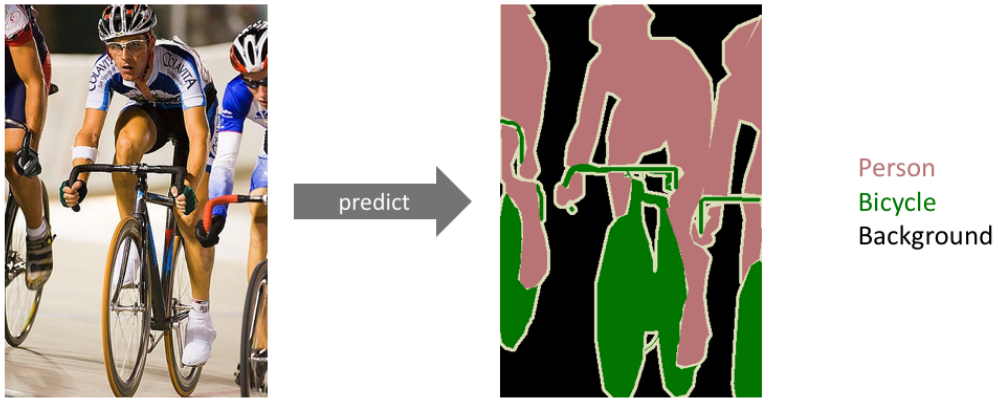

Figure 3.7: **Semantic Image Segmentation**. The goal of semantic image segmentation is to predict a class labels for each pixel in an image. Fig. taken from [44].

Some of the early approaches for semantic segmentation of images and photographs relied on techniques such as conditional random field (CRF) and Support Vector Machine (SVM). CRF can be used to model the relationship between nearby pixels probabilistically [73]. For example, nearby pixels or pixels with similar colour would be more likely to have the same label. Using superpixels as the basic unit of class segmentation constitutes an additional improvement as it allows for the local redundancy in the data to be captured [23]. SVM is a very popular and well-studied ML algorithm that has been shown to perform well in a variety of CV tasks, including segmentation [83, 82, 52, 43]. The main drawback of SVM is that extensive care must be taken in selecting the image features to be used for performing pixel-level classification [60].

With the advent of ANNs and, in particular, convolutional neural networks (CNNs), all the approaches described above have been superseded. This is evidenced by the fact that between the period 2011-2017 the number of published articles with the topic of ANNs has increased by 260% compared to the period 2000-2010, while the number of articles with the topic of SVM has only increased by 35% [34]. The unique capabilities and features of CNNs that are behind their recent popularity is described in the next Section.

3.4 Convolutional neural networks: fully connected vs. fully convolutional networks

CNNs are a specialized type of neural network designed for working with two-dimensional image data and are typically organized into a series of interconnected layers with specific functions [47] (Fig. 3.8). This hierarchical, connectivity pattern

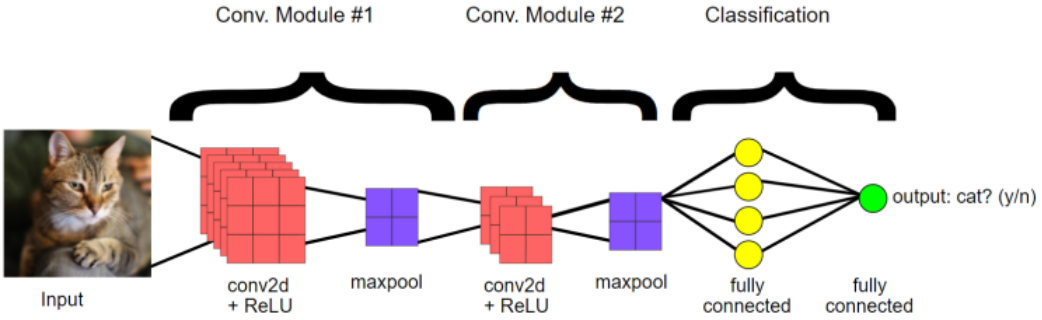

Figure 3.8: **Diagram of a typical CNN used for whole-image classification**. The CNN shown here contains two convolutional modules (each made up of a convolution followed by ReLu activation and max pooling) for feature extraction and two fully connected layers for classification. Other CNNs may contain a larger or smaller number of convolutional modules. Fig. taken from [26].

between layers of neurons with different functions is inspired by the organization of the different types of cells in the animal visual cortex [31, 22]. Central to the convolutional neural network is the convolutional layer that gives the network its name (Fig. 3.9). This layer performs an operation called convolution, which is a linear operation that involves the dot product between an array of input data with a two-dimensional array of weights, called a filter or a kernel [14]. In addition to convolutional layers, a CNN comprises two more types of neural layers, namely pooling layers and fully connected layers.

The function of the pooling layer is to reduce the dimensionality of the input volume (i.e. width and height) to the next convolutional layer without affecting the depth dimension of the volume (Fig. 3.10) [81]. This operation is also known as subsampling or downsampling. Although the reduction in size results in a loss of information, such loss comes with two benefits: lowering the computational cost, by reducing the number of parameters to be learnt; preventing overfitting to the training data, by creating an abstracted form of the representation of the input from the previous layer. Average pooling and max pooling are the most commonly used strategies for pooling, whereby a maximum or average filter is applied to usually non-overlapping subregions of the initial representation, respectively (Fig. 3.10).

As shown in Fig. 3.2, in a fully connected (FC) layer, neurons have connections to the activation of all the neurons in the previous layer. This kind of layers is key for enacting the high-level reasoning in neural networks, by converting 2D feature maps into a 1D feature vector that can be used for classification [81]. It is worth noting that while convolutional and pooling layers are necessary building blocks to any CNN, FC layers are not ubiquitous in neural networks, and for obvious reasons. In the context of CV tasks, the presence of a FC layer enables the network to essentially perform whole-image classification based on the features extracted by the previous layers. However, this constitutes a relatively coarse type of inference, in the sense that classifying the content of an image and performing bounding box object detection is not as fine and detailed as making a classification at every pixel. Moreover, another

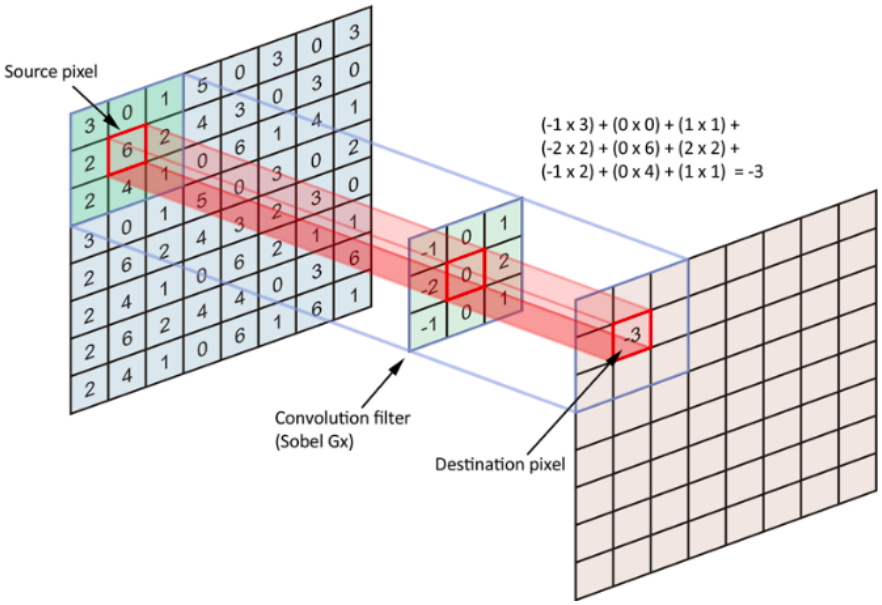

Figure 3.9: **2D convolution**. A convolution extracts 2D tiles from the input feature map (here only 1 channel is shown for simplicity) and applies filters to them in order to compute new features and produce an output feature map. This output feature map may have a different size and depth than the input feature map. Convolutions are defined by 4 parameters: size of the tiles that are extracted (typically 3x3 or 5x5 pixels); the depth of the output feature map, which corresponds to the number of filters that are applied; the size of the stride, which is the amount by which the filter shifts over the input image; and the size of the padding, which is the amount of zeros to be added around the input image in order to preserve image size. During a convolution, the convolution filters (i.e. matrices that have the same size as the tile size) are slid over the input feature map horizontally and vertically, one pixel at a time (if stride=1), extracting each corresponding tile. For each filter-tile pair, the CNN performs element-wise multiplication between the filter matrix and the tile matrix, and then sums all the elements of the resulting matrix in order to get a single value. During training, the CNN is able to tune the values for the filter matrices that enable it to extract meaningful features (e.g. textures, edges, shapes) from the input feature map. In the picture shown here, 3x3 convolution is not padded, which leads to a reduction of 2 pixels for both height and width in the output feature map. Fig. taken from [18].

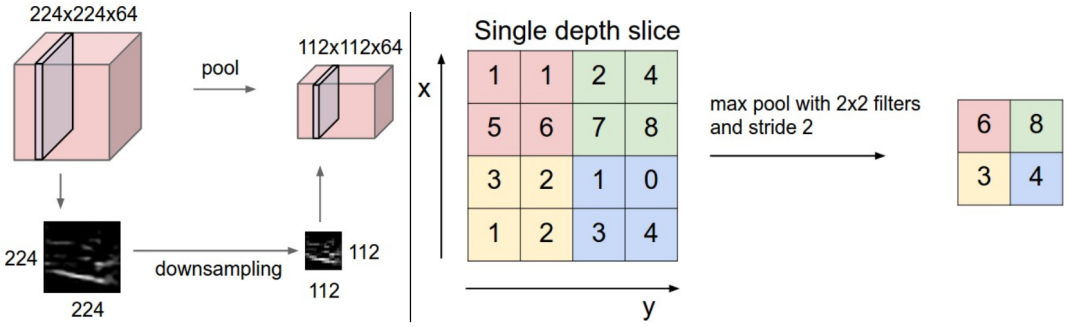

Figure 3.10: **Pooling Operation**. In the left picture, an input volume of size 224x224x64 is pooled with filter size 2 and stride 2 into output volume of size 112x112x64. Pooling preserves the input volume depth. In the right picture, the max pooling operation with filter size 2 and stride 2 is shown. Namely, the maximum value over the 4 numbers in the non-overlapping, 2x2 tile in the input image is taken. Fig. taken from [37].

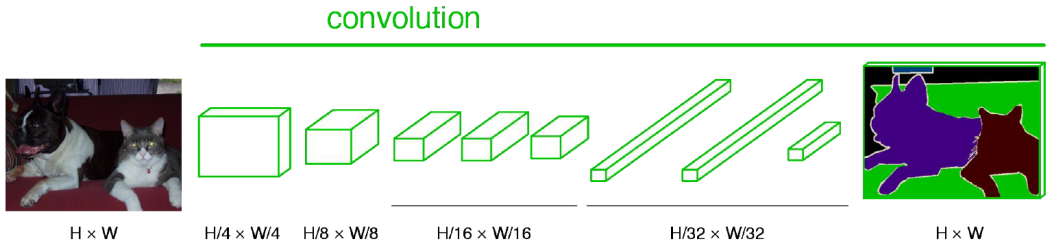

Figure 3.11: **Architecture of the Fully Convolution Neural Network proposed by Long et al. [53]**. A fully convolutional neural network is made up of blocks of convolutions, followed by pooling and activation functions that introduce nonlinearity. The successive application of these blocks causes the size of the input volume (here only 1 channel is represented) to be progressively halved. The upsampling step is a key step that maps the feature representations back onto the original pixels positions. Combined with the introduction of a pixel-wise loss, upsampling allows to generate an output of the same (or smaller, depending if convolutions contain no padding) size as the input that contains predictions at the pixel level. Fig. taken from [53] .

limitation of CNNs that include FC layers is that such networks can only accept input images of fixed size. This is because FC layers have matrices and vectors for the weights, biases and outputs of size that is fixed based on the input image size. Finally, it can be argued that the great success of traditional CNNs, especially on the large DL datasets mentioned above, is to be credited in part to the very large size of the available training sets and of the networks themselves, in terms of number of layers and parameters of such networks.

In a landmark paper by Long et al. [53], the end-to-end training of a fully convolutional network (FCN) in order to perform pixel-level classification was first described (Fig. 3.11). In particular, the key modifications to the structure of typical CNNs in order to achieve this task are: the replacement of FC layer by convolutional layers, based on the assumption that FC layers can be viewed as convolutions with kernels that cover their entire input regions; the introduction of in-network upsampling with up-convolutional layers that allow to assign the semantic meaning back to the pixels that generated the semantic information in the earlier, downsampling layers; the introduction of a pixel-wise loss that is computed over the spatial dimensions of the final layer; the introduction of skip connections that combine fine layers (generated from upsampling) with coarse layers (generated from downsampling) and let the model make local predictions that respect global structure.

FCNs clearly constitute the state of the art method for semantic segmentation, as they enable to reliably classify pixels into different clusters with semantic meaning. Most importantly, the fact that these networks work directly at the pixel level while still being able of making predictions that respect global structure makes them suitable for the type of task that the data analysis method developed by BGS is trying to carry out. Although grouping individual pixels into meaningful clusters is crucial, keeping the correct proportions between the different IARs 'fringes' is equally important, as the distance between fringes is used in order to calculate the value of parameters of interest to BGS, such as the frequency interval between fringes.

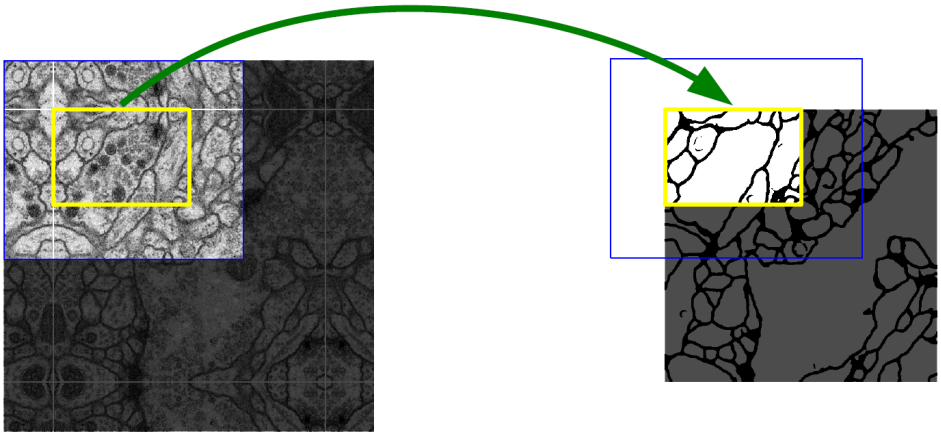

Figure 3.12: **Overlap-tile strategy for processing images with the U-net**. Left picture shows input image to the network, while right picture shows segmentation map outputted by the network. The prediction of the segmentation in the yellow area in the input image requires image data within the blue area as input. In other words, the output segmentation map only contains pixels such that for each pixel (yellow area) there is full context available (blue area) in the input image. This is one of the consequences of the fact that convolutions in the U-net architecture contain no padding. Moreover, it can be seen that for the input image (left image) the region outside the white box of same size as the predicted segmentation map (right image) is a mirror image of the inside of the white box. This 'mirroring' is a way of extrapolating the missing context for pixels in the border region of the image. Fig. taken from [69].

3.5 U-net

In many visual tasks from different fields, such as biomedical image processing, there is a strong need for predictions to include localization information at the pixel level. Moreover, the size of dataset tends to be quite limited, and definitely far smaller than the typical ones on which CNNs have built their success.

In light of this, Ronneberger et al. [69] extended the FCN architecture proposed by Long et al. [53] and developed a network that strongly leverages data augmentation in order to efficiently train on small, labeled datasets like the one typically found in biomedical imaging. One modification that is worth discussing is that in the upsampling part of the network (i.e. right part of the U), a large number of feature channels are present. In combination with the presence of up-convolutions, this allows the network to propagate context information to higher resolution layers. This characteristic is also responsible for the U-shape of the architecture, from which the name U-net was coined. Another important design choice is that for each convolution, only the valid part of the convolution is used. This allows processing of large images of any size using an overlap-tile strategy (Fig. 3.12).

U-net is composed of a downsampling and upsampling part (Fig. 3.13). In the downsampling part, the high-level features from the input image (i.e. input feature map) are learnt. This is done by putting four convolutional blocks one after each other. Each

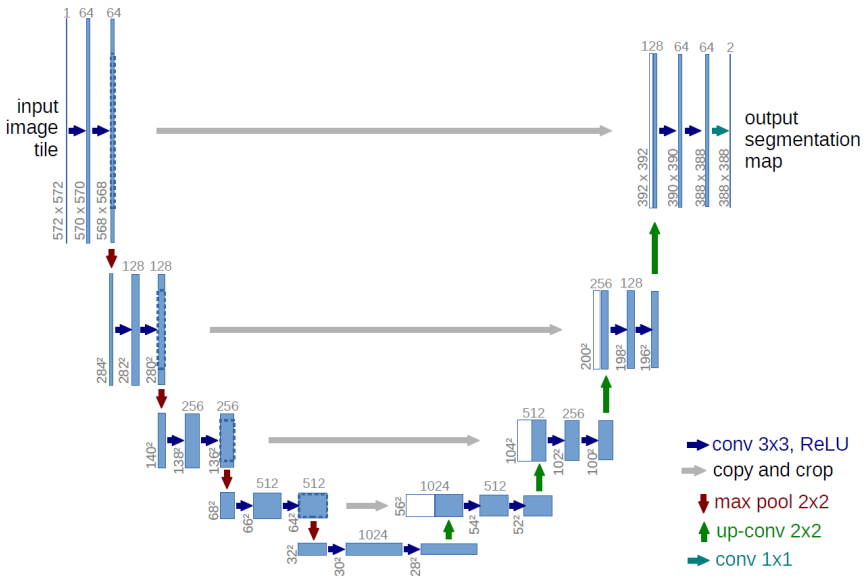

Figure 3.13: **U-net architecture**. Each blue box in the architecture representation corresponds to a multi-channel feature map. The number of channels is denoted on top of the box, while the width and height of the feature map is specified at the lower left edge of the box. White boxes represent cropped portions of feature maps that have been copied from a different feature map. The coloured arrows denote the different operations that are applied to the input image and subsequent feature maps at different stages within the network. Fig. taken from [69].

convolutional block is composed of two 3x3 unpadded convolutions, each followed by ReLu activation, and a final 2x2 max pooling layer with stride 2 connecting the output from a block to the input of the next block.

Since convolutions are unpadded, after each convolution two pixels from the border are lost. Moreover, the convolution following each max pooling operation always contains twice as many filters as the convolution before max pooling. Hence, as the feature map is processed by deeper layers in the downsampling part of the network, its size in terms of height and width becomes progressively smaller, while its depth (i.e. number of channels, which is determined by the number of filters used at each convolution) becomes larger. Hence, although the max pooling operations preserves the information that best describes the context and discards unimportant information, the subsequent convolution allows to preserve the contextual information. This in turn helps to increase the resolution of the classification down to the pixel level.

In the upsampling part, precise localization of the high-level features that have been identified in the downsampling path is achieved. Also, pixel-wise classification is achieved with a final 1x1 convolution. Again, this part is made up of four convolutional blocks. Each is composed of an upsampling operation, which is deterministic and non-learnable, a 2x2 convolution followed by ReLu activation, a concatenation with the corresponding (high-resolution) feature map from the downsampling part and two 3x3 convolutions, each followed by ReLu.

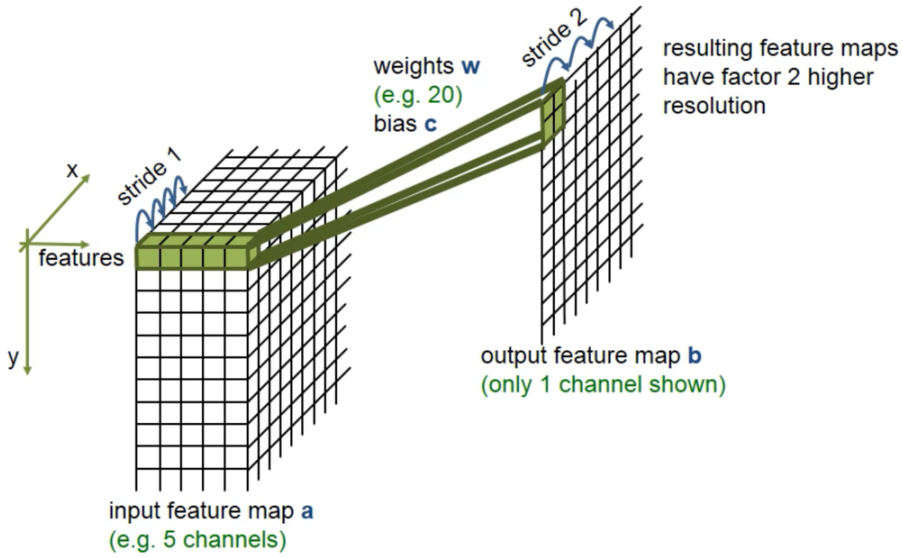

Figure 3.14: **2D up-convolution**. An up-convolution in the context of the U-net architecture can be broken down into upsampling the feature map by a factor of 2 in both width and height, followed by a 2x2 convolution that reduces the number of channels of the feature map by a factor of 2. In particular, upsampling is a way of enlarging the size of the input feature map, by simply repeating each entry in the feature map in both height and width direction. This is responsible for creating output feature maps that have a 2 times greater resolution following up-convolution. The convolution part of the up-convolution intentionally halves the number of feature channels (e.g. it uses half the number of filters used in the convolution in the previous block). Moreover, it uses a learned kernel containing $f \times 4$ weights, where f is the depth of the input feature map (i.e. 5 in the example shown, so that works out as 20 weights plus 1 bias) in order to map a single feature vector from the input feature map to a 2x2 pixel output vector in the output feature map. Fig. taken from [70].

The upsampling operation allows the size of the input data to be increased, by simply repeating both the rows and columns of the input by a specified factor. The combination of the upsampling operation followed by convolution is equivalent to the up-convolution operation. This can be simply conceived as a transformation that goes in the opposite direction of a normal convolution. It essentially uses a learned kernel to map each feature vector to a 2x2 pixel output vector (Fig. 3.14).

Also, the feature maps that are concatenated from the downsampling part are cropped. Again, this is necessary because of the loss of border pixels after each unpadded convolution that took place in between the two feature maps to be concatenated. Using this network trained on transmitted light microscopy images, the authors became the uncontested winner of the Cell Tracking Challenge at the International Symposium on Biomedical Imaging (ISBI) in 2015.

Chapter 4

Project Objectives and Methodology

4.1 Project objectives

Although there are presently no reports of the application of ML to the automatic recognition of IARs (or similar geomagnetic phenomena such as Schumann resonances) patterns from spectrogram image data, it can be envisaged that a trained neural network could be used to automatically capture the features of interest, namely IARs-associated signal, from such data. Since the IARs-associated signal displays complicated signal strength as well as localization patterns, a conventional classifier generating a single class label for a given image would appear to be unsuitable here.

Instead, a pixel-wise, binary classification with IARs signal or background as target classes should be performed in order to take localization and context into account. As described in Section 3.3, this type of classification falls within the realm of semantic segmentation.

Furthermore, given that the present labeled dataset that could possibly be used for training is restricted to only 152 examples (i.e. 152 out of 525 calendar days between September 2012 and February 2014 that, according to visual inspection, have been adequately labeled by BGS's data analysis method), it appears that the issue at hand could best be dealt with using a fully convolution network such as U-net within the context of supervised ML. This neural network is particularly suitable for the task as it was specifically designed for being trained on small datasets including as little as 30 training examples by leveraging data augmentation.

Hence, the present project proposition seeks to create an alternative method to the image detection part of the data analysis method developed by BGS described in Beggan [4] for selectively recognizing IARs signal that will exhibit:

1. greater signal to noise ratio on a 'good' day with IARs signal present
2. ability to generate null predictions on a 'bad' day with no IARs signal present

In order to achieve these objectives, the project was split into the following list of tasks:

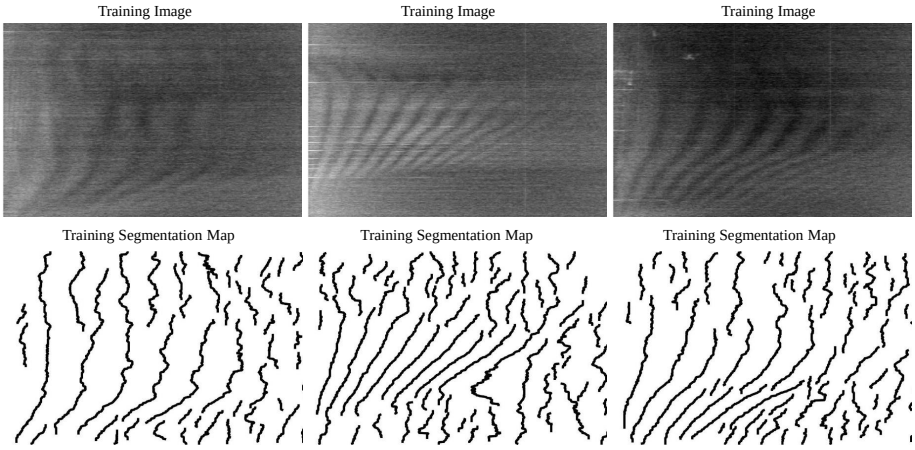

Figure 4.1: **Overview of training images and associated training segmentation maps for 3 training examples from the IARs dataset**. Top shows 3 training images drawn from the IARs dataset (i.e. training data), while bottom shows the respective segmentation maps (i.e. training labels) for each image.

1. Train U-net algorithm on IARs dataset and achieve maximum performance on the dataset
2. Identify best way of filtering prediction images as to reduce the noise generated by U-net
3. Perform negative control test with noisy test examples to confirm selectivity of U-net for ‘good‘ days
4. Obtain positive confirmation by Dr. Beggan with regards to the goodness of the newly developed ML tool

As the ML algorithm used in the project aims to essentially achieve the image segmentation task, the trained predictive model essentially constitutes a segmenter. The focus throughout the project is to estimate the performance of the trained segmenter in an accurate and unbiased way. Performance here, as in many ML projects, indicates the ability to make satisfactory predictions on new, unseen data. Following that, the model needs to be finalized by performing training on the entire dataset. After doing that, it will possible to benchmark the trained segmenter against the current method and determine whether the two project objectives have been met. In the next Sections, the proposed methodology used as part of the dissertation is extensively reviewed. This is divided into two Sections covering the methods and approaches falling into ML (Section 4.2) or HPC (Section 4.3), respectively.

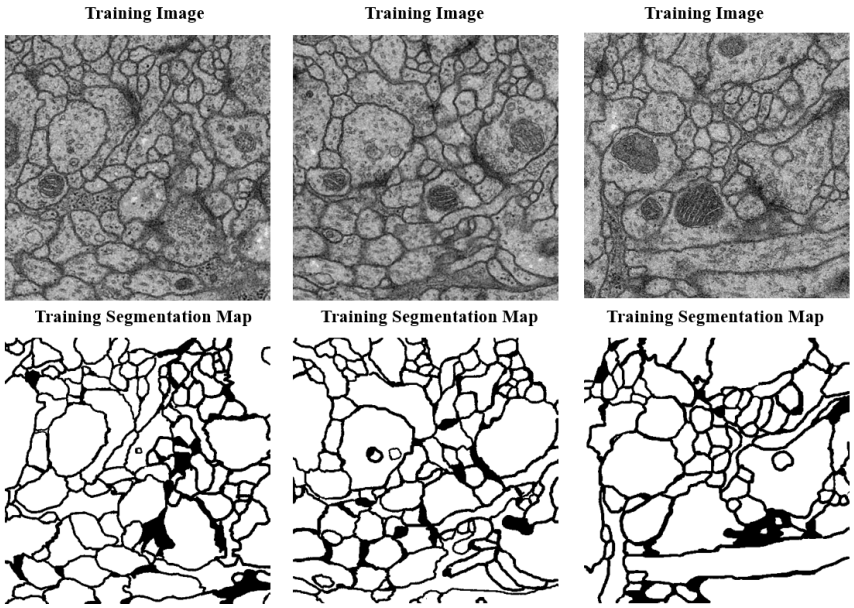

Figure 4.2: **Overview of training images and associated training segmentation maps for 3 training examples from the Cell dataset**. Top shows 3 training images drawn from the Cell dataset (i.e. training data), while bottom shows the respective segmentation maps (i.e. training labels) for each image.

4.2 Methodology: Machine Learning

4.2.1 Datasets

IARs dataset

At the beginning of the project, a curated dataset of 152 training examples (i.e. images of adequately labeled spectrograms as shown in panel b in Fig. 2.8) was shared by the industrial collaborator. In the middle of the project, this dataset was revised and it was replaced with a different dataset containing 178 training examples of better quality. This sample contains a carefully selected list of images in grayscale format and corresponding segmentation maps (with pixel values of 1 or 0 for absence or presence of IAR signal, respectively) drawn from a larger dataset for the 2313 available calendar days between September 2nd 2012 and January 1st 2019. Hence, the curated, training dataset contains 178 images, while the test dataset contains the remaining days from the larger dataset that are not part of the training dataset, namely 2135 examples.

Colour images were also available, but since grayscale images were used in the original paper where U-net was described [69], these were chosen over the versions in colour (Fig. 4.1). Both the images and the segmentation maps for a given training example have a size of 701x1101 pixels. In terms of what signal these images represent, the width represents 'frequency' and it ranges between 0.5 and 6.3 Hz from left to right in the image. The height corresponds to 'time' and it ranges between 6pm to 6am from bottom to top of image.

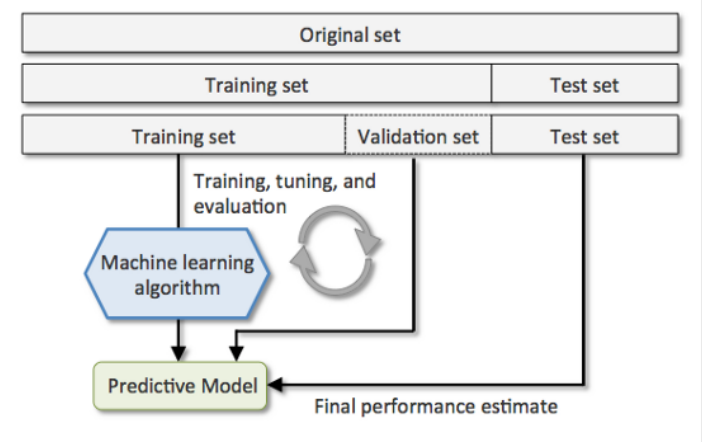

Figure 4.3: **Breakdown of dataset into train, validation and test set and their roles in model building and evaluation**. In a typical supervised ML task, the original dataset is split into training and test set. The training set corresponds to the 178 'good' days, while the test set is represented by the 135 calendar days between September 2nd 2012 and January 1st 2019 that are not part of the training dataset. The test set is used for calculating a final performance estimate, while the training set can be further divided into training and validation set. These are used in conjuction in order to train, tune and evaluate a predictive model (i.e. a segmenter) using a machine learning algorithm (i.e. U-net neural network for this project). Fig. taken from [65].

Cell dataset

As mentioned in Ronneberger et al. [69], U-net was originally developed as part of the challenge on segmentation of neuronal structures in electron microscopy stacks at the ISBI in 2015. The images that are part of this dataset are shown in Fig. 4.2. From here onwards the ISBI dataset is denoted as 'Cell' dataset.

4.2.2 Model evaluation techniques

There are several techniques that can be used to estimate model performance, such as the holdout method, k-fold cross-validation, leave-one-out cross-validation, stratification and bootstrap [42]. All these methods differ with respect to how efficiently the data are used, the amount of bias and variance introduced in the evaluation and the complexity of the procedure. That being said, all these methods rely on a key re-distribution of the available dataset, which is the splitting into training, validation and test sets (Fig. 4.3). Training data are specifically used to learn patterns from data, which translates into determining the best set of weights that minimize the loss function and in turn allow the network to make predictions on unseen (test) data. On the other hand, validation data are needed in order to understand the behaviour of the model and gauge its ability to perform well and adapt properly to new, unseen data while it is still being trained. It can be also be used to gain insights into how to tune the parameters of the model. Finally, test data are synonymous with unseen data and are used in order to

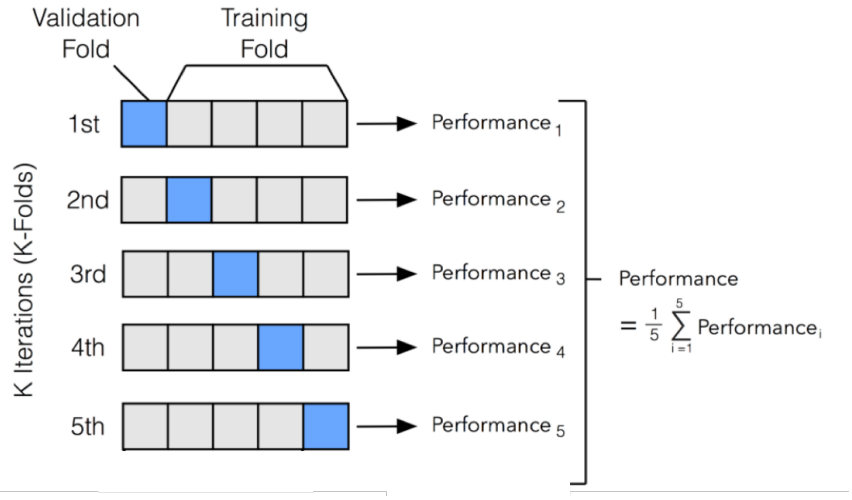

Figure 4.4: **Diagrammatic description of the k-fold cross-validation algorithm**. This figure illustrates the cross-validation algorithm when k, namely the number of folds, is set to 5. The data are split into 5 folds and at each of the 5 iterations a different fold is used as validation fold while the remaining ones are used as training folds. Hence, at each iteration, a separate, surrogate model is trained on a portion of the training dataset and a performance value is calculated. The validation loss is the performance indicator used in the project. The final performance value that is outputted by the algorithm is the average of the individual performance values calculated over the 5 iterations. Fig. taken from [51].

understand how the model would perform when deployed in the real world. Test data are paramount as they generate an unbiased estimate of model performance.

In the following two subsections, the algorithm for k-fold cross-validation is outlined, along with a discussion of the benefits of choosing this method for model performance estimation and hyperparameter tuning.

Model evaluation with k-fold cross-validation

K-fold cross-validation currently constitutes the industry standard for model performance estimation and parameter tuning. K-fold cross-validation is a resampling method where a model of interest is repeatedly re-fit to samples from the training set, in order to obtain additional information about the fitted model [32]. K-fold cross-validation is mainly used to get an idea of the test error of the model, namely the error on unseen test data. Error can be interpreted as the value of the loss function. In order to estimate the test error, cross-validation holds out a subset of the training observation from the fitting process, and then applies the trained machine learning method to those held out observations. In particular, the dataset is split into k parts and the training process is repeated k number of times (Fig. 4.4). For each of these k iterations, a different fold of the dataset is picked to be the validation set and the remaining k-1 folds are used as training set. On each of these iterations, the value for all the evaluation metrics and loss functions, including the validation set loss, are

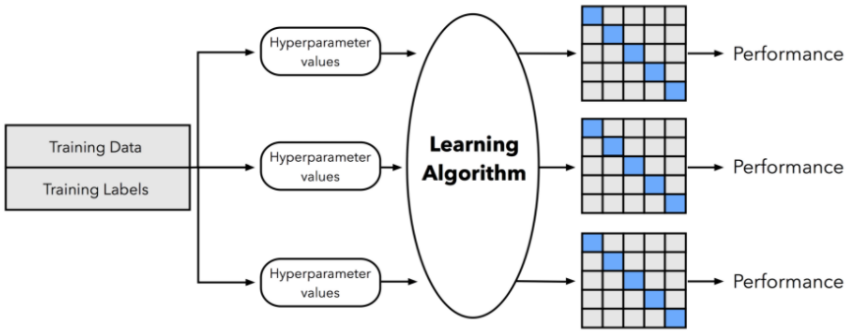

Figure 4.5: **Diagrammatic description of k-fold cross-validation applied for hyperparameter tuning**. The k-fold cross-validation algorithm described in Fig. 4.4 can be extended in order to perform hyperparameter tuning by performing k-fold cross-validation on different permutations of hyperparameter values. Exploration of parameter values can be done using grid search or random search. For each permutation of hyperparameter values, k-fold cross-validation separately calculates a performance value. The hyperparameter permutation that is responsible for the best performance (i.e. lowest validation loss) is regarded as the optimal one. Fig. taken from [51].

computed, such that the average of these values is ideally the same as the one that would be obtained if the model was to be trained on the entire dataset directly without K-fold cross-validation.

Picking a value of k is a form of bias-variance trade off. In other words, when data are split into folds, some variability is introduced just by how the split is performed, as the number of splits affects the estimate of the test error. Two types of k-fold cross-validation are typically contrasted in the ML literature: the k-fold cross-validation approach described so far (see Fig. 4.4), and leave-one-out cross-validation (LOOCV), which is a special case of k-fold cross-validation in which k is equal to the number of training examples. The general conclusion that can be drawn by consulting the relevant literature is that there is no universal agreement as to whether any of these methods exhibits lower bias and variance while estimating the test error of a ML algorithm [7, 85]. That being said, the textbook answer is that k-fold cross-validation is preferred over LOOCV because the test error estimate from k-fold cross-validation has lower variance than the estimate from LOOCV [32]. This is because in k-fold cross-validation, the test error estimate is calculated by averaging the outputs of k fitted models that are somewhat less correlated with each other compared to LOOCV. Since the mean of highly correlated quantities has higher variance than the mean of quantities that are not as highly correlated, this explains why the test error estimate from k-fold cross-validation tends to have lower variance than test error estimate resulting from LOOCV. Furthermore, k-fold cross-validation is advantageous from a computational point of view, because it requires building k independent models, where k is typically smaller than the number of training examples. On the other hand, in LOOCV the ML method must be fitted a number of times that is equal to the number of training examples.

There is not an exact or well-defined way of selecting the value of k for k-fold cross-validation. Values of k of 5 or 10 are usually recommended based on empirical tests [32]. However, the number of instances contained in a given dataset should also be factored in the decision of the value of k. For example, if a dataset contained 10 instances, 10-fold cross-validation, which is equivalent to LOOCV in this case, would not be applicable. Hence, another concern when selecting a value for k is that both the training set and validation set are drawn from the same distribution, and that both sets contain sufficient variation such that the underlying distribution is represented. For example, in a 10-fold cross-validation with only 10 instances, there would only be 1 instance in the test set and clearly this instance would not properly represent the variation of the underlying distribution. That being said, estimating how well a given fold represents the overall dataset is known to be a hard task.

Hyperparameter tuning with k-fold cross-validation and grid-search

In addition to getting an insight into the test error of a fitted model, the other objective of cross-validation is to allow the best hyperparameters of a given model to be picked (Fig. 4.5). Hyperparameter optimization is a crucial part of the training/fitting process of any ML algorithm [29, 17]. A model hyperparameter is a characteristic of a model that is external to the model and whose value cannot be estimated from data. In other words, the value of the hyperparameters must be set before the learning process begins. Therefore, the best hyperparameters for a given model and dataset cannot be known in advance. Examples of hyperparameters for neural networks are learning rate, batch size, number of epochs, training optimization algorithm, weight initialization, activation function of layers, dropout regularization and the number of neurons in a hidden layer. Once tuned, these hyperparameters help in the estimation of model parameters and, in turn, allow to generate the most accurate predictions possible for a given model and dataset. On the other hand, model parameters are an internal characteristic of a model and their value can be estimated from data. These are, for example, the weights of the neural network.

There are several methods and algorithms for finding the optimal hyperparameters of a model within the context of ML, including grid search and random search [9, 8, 45]. Grid search is an approach to hyperparameter tuning that methodically builds and evaluates a model for each combination of algorithm hyperparameters specified in a grid. Random search differs from grid search in that, rather than providing a discrete set of values for each hyperparameter to be explored, a statistical distribution for each hyperparameter is given from which values may be randomly sampled. A discussion of other, recent approaches for hyperparameter tuning including global optimization, evolutionary algorithms and hyperparameter optimization algorithms is outside the scope of this dissertation.

Although it has been shown both empirically and theoretically that randomly chosen trials are more efficient for hyperparameter optimization than trials on a grid [8], grid search still constitutes one of the most widely used strategies for hyperparameter

optimization. The reason for this is that grid search is simple to implement, its parallelization is trivial and is reliable in low dimensional space (e.g. 1 and 2 dimensions) [8]. As part of this dissertation, grid search was used in combination with k-fold cross-validation in order to find the best hyperparameters for training U-net on the IARs dataset.

4.2.3 Evaluation of experiments on U-net

After having built the U-net neural network using a computer program that specifies its layers, the network must be trained on data in order to build the sought-after predictive model that is able to generate predictions on new, unseen data. Training is monitored using evaluation metrics and training loss functions. In this Section, the evaluation metric and training loss function chosen for the project are presented, while considering the rationale that motivates their choice.

Evaluation Metric

A metric allows the performance of a ML algorithm to be evaluated in a standard manner. Accuracy is the most common evaluation metric and is computed as the number of correct predictions made as a ratio of all predictions made:

$$Accuracy = (TP + TN)/(TP + TN + FP + FN)$$

A true positive (TP) represents a pixel that is correctly predicted to belong to the given class according to the ground truth mask, whereas a true negative (TN) represents a pixel that is correctly identified as not belonging to the given class. Moreover, a false positive (FP) represents a pixel that is incorrectly predicted to belong to the given class, while a false negative (FN) represents a pixel that is incorrectly predicted as not belonging to the given class.

The main limitations of accuracy is that it becomes uninformative in imbalanced data problems, namely when classes contain different relative numbers of predictions, or when there is variation in the costs of different prediction errors [15, 30]. In these two situations, alternative metrics should be used. Accuracy is clearly not suitable for this dissertation project, as the segmentation maps in the IARs dataset typically contain about 3 times more white pixels than black pixels. In other words, if accuracy was to be used the metric would be biased in mainly reporting how well negative case are identified, namely where the class IARs signal is not present.

The Intersection over Union (IoU) metric is essentially a method to quantify the percent overlap between the target mask and the prediction output [49]. Just as the name says, this metric is calculated by counting the number of pixels that are correctly predicted

and therefore have the same location and class label in both the ground truth and prediction masks divided by the sum of the number of pixels present across both masks:

$$IoU = TP/(TP + FP + FN)$$

In a binary or multiclass problem, the IoU score is calculated for each class and then averaged over all classes, thereby yielding a mean IoU score. The IoU metric appears to be a very suitable metric for the segmentation task and is used throughout the dissertation project.

Loss function

Binary classification is a type of predictive modeling problem where examples are assigned one of two labels. Cross-entropy is the default loss function to use for binary classification problems [33]. It is intended for use with binary classification where the target values for the prediction of each class are in the set contained between 0 and 1. This loss can be extended to pixel-wise binary cross-entropy, where for every individual pixel the vector of class predictions (with depth of 2 for binary classification) is compared against the one-hot encoded vector of the corresponding pixel in the ground truth image (Fig. 4.6). One-hot encoding essentially means that for a given ground truth pixel a vector of class values can be considered, with a value of 1 assigned to the correct class and 0 to all other classes.

Pixel-wise binary cross-entropy loss is calculated as the log loss summed over two possible classes at every pixel [35]. The average value of the loss function calculated over all *N* pixels is then reported :

$$-1/N \sum_{i=1}^{N} y_{true} log(y_{pred}) + (1 - y_{true}) log(1 - y_{pred})$$

In the original U-net implementation, the authors present a custom loss weighting scheme based on binary cross-entropy that places larger weight on the borders of segmented objects. This helped them to produce binary segmentation maps on the Cell dataset that have clear borders. However, this modification of binary cross-entropy is not necessary here, as the IARs signal lines tend to be well-spaced.

Binary cross-entropy tends to be the default choice for loss function for the image segmentation task [53, 61, 3]. However, one potential drawback of using binary cross-entropy is that from a mathematical point of view it is still more closely related to accuracy than to IoU. Hence, by defining a loss function that is more closely related to IoU, it could be argued that the training process could be further improved. The Dice coefficient and the Jaccard distance loss constitute two examples of loss functions that measure the amount of overlap between prediction and ground truth,

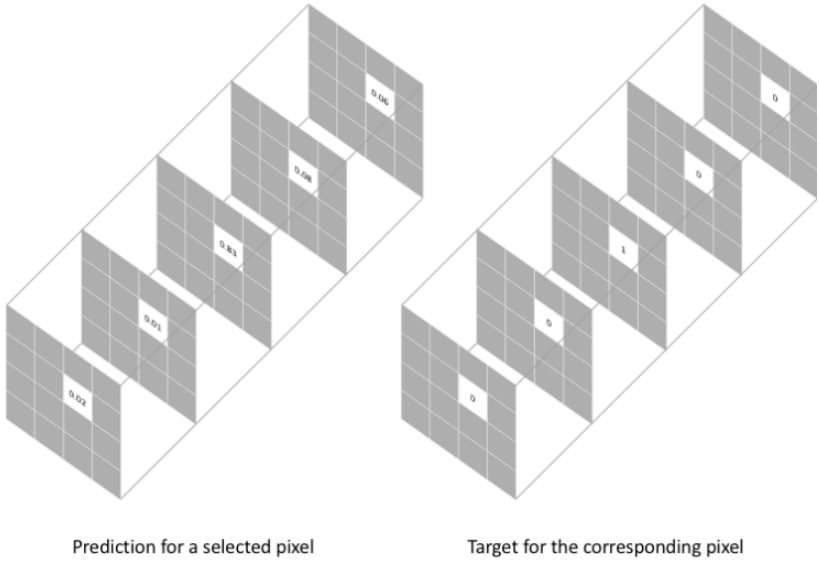

Figure 4.6: **Pixel-wise cross-entropy**. In order to achieve (binary) pixel-wise classification, U-net employs a convolution with 2 filters of size 1x1 and sigmoid activation as last layer in the network. This reduces the dimensionality in the filter dimension (from 64 to 2 filters) such that the output segmentation map has 2 feature channels, one for each class. For a multiclass segmentation problem (e.g. 5 classes in the picture shown here), class predictions, which can be represented as a depth-wise pixel vector (shown on left), are compared to a one-hot encoded target or ground truth vector (shown on right). Fig. taken from [35].

where the prediction is a collection of all the depth-wise pixel vectors populated by values generated by the trained algorithm when it is presented with validation data during training, while the ground truth is the equivalent representation containing the 'correct' label in one-hot encoded format for the validation data that is provided to the (supervised) learning algorithm during training. Considering the predicted region P and the ground truth region G, and by denoting |P| and |G| as the sum of elements in each region, the Dice coefficient [87] is twice the ratio of the intersection over the sum of areas:

$$DiceCoef(P, G) = 2(|P \cap G|)/(|P| + |G|)$$

The Jaccard distance [49] loss is denoted as follows:

$$JaccardDistance(P, G) = (|P \cap G|)/(|P \cup G|) = (|P \cap G|)/(|P| + |G| - |P \cap G|)$$

It must considered that as P and G in both the Dice Coefficent and Jaccard Distance are one-hot encoded vectors, the individual sums of their respective values is equivalent to the number of 1s (or values that are approximately equal to 1) in each respective set. This is why the vertical bar notation can be used in order to indicate the cardinality, or the number of 1s, of either P or G. Moreover, the intersection between prediction

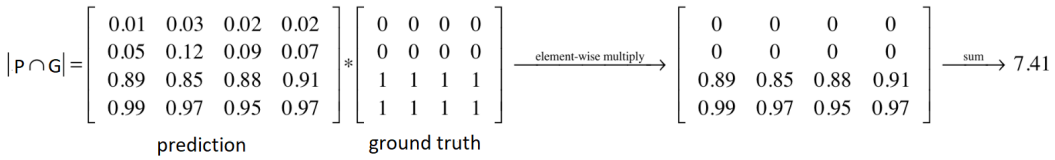

Figure 4.7: **Calculating intersection between prediction and ground truth images**. The intersection between predicted and ground truth mask in a binary classification can be approximated as the element-wise multiplication between the two masks, followed by the sum of all the elements. Fig. taken from [35].

and ground truth can be approximated as the element-wise multiplication of the two sets, followed by summing the individual values (Fig. 4.7). Furthermore, it is also worth noting that in practice the Jaccard distance loss is typically implemented as an approximation of IoU made using probabilities [80].

Since binary cross-entropy still constitutes the default loss function for the segmentation task, it was chosen as the starting point for the project.

4.2.4 Implementation tools

In this Section, an overview of the programming implementation tools used in the project is given.

Tensorflow

TensorFlow is an open source library for machine learning developed by Google Brain, which is a DL and AI research organization within Google [1]. It was originally intended for internal Google use, but then was publicly released on November 2015 under the Apache 2.0 open source license. It is written in C++, Python and CUDA. Tensorflow is a high performance numerical computation software library. Its strong support for machine learning and neural networks makes it the most popular AI software package. Tensorflow architecture allows easy computational development across a diverse range of platforms, such as CPUs, GPUs or TPUs.

Keras

Keras is an open source neural network library written in Python and developed as part of the research effort of project Open-ended Neuro-Electronic Intelligent Robot Operating System (ONEIROS) [16]. Its primary author and maintainer is François Chollet, a Google engineer. Keras was designed not to be a standalone ML framework, but rather a high-level interface on top of the most popular deep learning backends. Keras is in fact capable of running on top of popular DL platforms such as TensorFlow,

Table 4.1: **List of Git branches and associated features**. This is the finalized list of active branches that were created and maintained over the course of the project. These can be found at the online Github repository belonging to the student [55].

Branch name	Features
activations heatmaps	experiments with keract package
alternative loss functions	experiments with alternative loss functions
code profiling	timing entire code before and after HPC optimizations
image size experiments	experiments with finalized model on images of larger size
k unet cross valid	implementation of K U-net with cross-validation
master	experiments with finalized model
parallelization experiments repeated	repeats of some multithreading experiments
parallelization experiments	experiments with multithreading and GPU parallelization
hyperparameter tuning tests	experiments for identifying best hyperparameter values
x unet cross valid	implementation of X U-net with cross-validation
vanilla x unet	vanilla implementation of X U-net

Microsoft Cognitive Toolkit and Theano. Keras was created with the following guiding principles: modularity, extensibility and user-friendliness. As of 2017, Keras is supported in Tensorflow's core library.

Git

All the code was maintained and back-ed up throughout the project in a Github repository [55]. Code was split into different branches in order to aid introduction of novel features. Table 4.1 shows the finalized list of branches and associated features.

Hardware

All experiments were conducted on the University of Edinburgh Cirrus HPC cluster using 1 or more GPU nodes, or 1 or more standard compute nodes [20]. The University of Edinburgh Cirrus HPC cluster is simply shortened to 'Cirrus' from here onwards.

A standard compute node on Cirrus contains two 2.1 GHz, 18-core Intel Xeon E5-2695 (Broadwell) series processors. On the other hand, a GPU compute node contains two 2.4 GHz, 20-core Intel Xeon Gold 6148 Skylake series processers. Each of the cores in the processors in both the standard and GPU compute nodes support 2 hardware threads (also known as hyperthreading) enabled by default. A GPU compute node also contains four NVIDIA Tesla V100-PCIE-16GB Volta GPU accelerators connected to the host processors and to each other via Peripheral Component Interconnect (PCI) Express. From here onwards a single Intel Xeon Gold 6148 processor is named 'Skylake CPU', while a single Intel Xeon E5-2695 is referred as 'Standard CPU'.

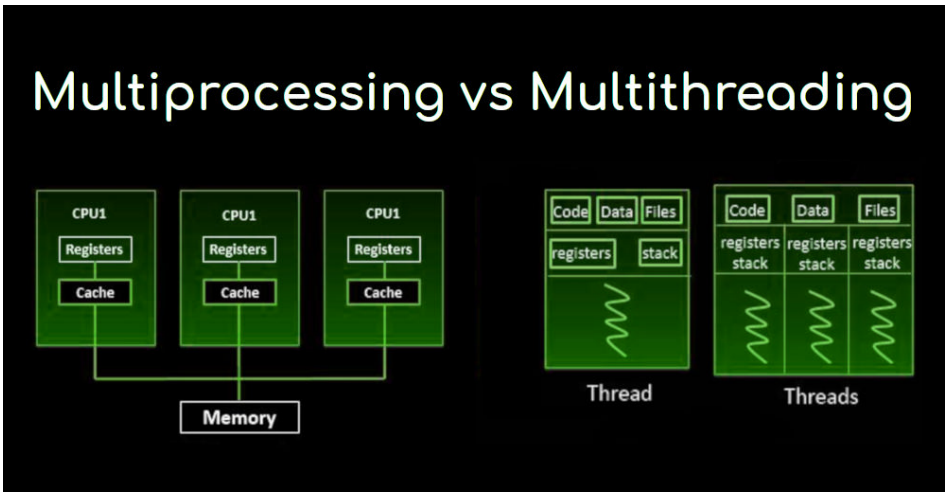

Figure 4.8: **Difference between multithreading and multiprocessing**. A multiprocessing system (left) is one which has more than two processors, and each processor, namely a CPU, has its own set of registers and cache. However, all processors share main memory. According to this model, processors can be arbitrarily added to the system to increase its computing speed. On the other hand, multithreading consists in executing multiple threads of a single, given process concurrently within the context of the process. In a single-threaded process, the instructions are executed in a single sequence, namely one single thread, and only one command is processed at a time. On a multiprocessing or multi-core system, multiple threads can execute in parallel, leading to multithreading. In the context of multithreading and parallel computing, a thread can be conceived as a single child process that shares the resources (e.g. code, data, files, memory) of the parent process but executes independently using private registers and stack that belong to an individual processor. According to this model, a single process can be made arbitrarily faster through parallelization, namely split data and/or tasks into parallel subtasks that run either concurrently on multiple cores within a CPU or in parallel on multiple cores across multiple CPUs. Fig. taken from [24].

4.3 Methodology: HPC and parallelization

4.3.1 Multithreading and Multiprocessing

Tensorflow and Keras offer several, independent avenues for achieving multiprocessing and multithreading in the context of training neural networks. Multiprocessing is the idea of executing multiple processes at the same time on multiple processors, whereas multithreading lets a process generate multiple threads to increase the computing speed of a system (Fig. 4.8). A thread is defined as a basic unit of CPU utilization. A multiprocessing system executes multiple processes simultaneously whereas a multithreading system lets multiple threads of a process to be executed simultaneously. Both these options are suitable for this project because the computational resources available on Cirrus offer several cores within a single node (i.e. 36 cores for a single standard compute node and 40 cores for a single GPU compute node) and also each of these cores is able to schedule 2 logical threads with hyperthreading.

At the heart of any Tensorflow program, there is a computation graph, which is essentially a series of connected, data-processing functions. This graph is made up of

two building blocks, namely nodes and edges. The nodes are called Operations (*Ops*) and they constitute some sort of action or computation being done on data. The edges represent the actual values that are passed to and from *Ops*. Individual *Ops* have parallel implementations, meaning that they are capable of using multiple cores in a CPU, or multiple threads in a GPU [79]. Hence, it would appear that by simply running the code responsible for training the U-net on the IARs dataset on multiple cores within one or more compute nodes on Cirrus, it should be possible to automatically get the maximum performance out of the code with multiprocessing.

Tensorflow lists two configurations in order to optimize CPU performance by adjusting the thread pools [78]. These are *intra op parallelism threads* and *inter op parallelism threads*. With *intra op parallelism threads*, *Ops* that can use multiple threads to parallelize their execution (i.e. internal parallelization) will schedule the individual pieces into a specific pool of *intra op parallelism threads*. With *inter op parallelism threads*, *Ops* that performing blocking operations are queued on a pool of *inter op parallelism threads* that can be used to parallelize their execution (i.e. concurrent parallelization). So it would appear that using these options, it should be possible to improve the performance even further through multithreading on top of multiprocessing (Fig. 4.8). In addition, Keras's fit generator function, which is used to train the model on data generated batch-by-batch by a Python generator, has some arguments that in theory should offer additional improvements in performance and will therefore be tested in the dissertation project. One argument of this function is *use_multiprocessing*, which accepts a *Boolean*. If it is set to True, this allows process-based threading. Another argument is *workers*, which accepts an *Integer*. This is the maximum number of processes to spin up when using process-based threading.

4.3.2 GPU parallelization

In order for Tensorflow code to be deployed on a GPU, the GPU-enabled version of Tensorflow (i.e. tensorflow-gpu) must be installed. Due to compatibility issues between the CUDA driver present on Cirrus and required CUDA installation for tensorflow-gpu, tensorflow-gpu was used in a local Singularity image [76] that was built using an online tensorflow-gpu image available from Dockerhub (*singularity build tfgpu.simg docker://tensorflow/tensorflow:1.12.0-gpu-py3*). Without any code changes, the same code could be easily deployed on a CPU or GPU.

4.3.3 Multi-GPU parallelization

There are two common approaches for multi-GPU parallelization: data-parallelism and device parallelism. In order to achieve parallelization across multiple GPUs with data parallelism, Keras's *multi_gpu_model* can be used. With this option enabled, Keras replicates the target model once on each device and then uses each copy to process a different fraction of the input data. On the other hand, device parallelism

consists in running different parts of the same model on different devices. This works best for models that have a parallel architecture, such as a long short-term memory (LSTM). Given that U-net does not exhibit a structure that makes it amenable for device parallelism, the data parallelism option was chosen for the dissertation project.

4.3.4 Monitoring GPU utilization

GPU utilization was separately monitored for GPU parallelization using the NVIDIA System Management Interface (i.e. *nvidia-smi*) command. This is a command line utility, based on top of the NVIDIA Management Library (NVML), intended to aid in the management and monitoring of NVIDIA GPU devices. This utility allows the user to query GPU device state at $\approx$ every second, including percentage of GPU and memory utilization [63].

Chapter 5

Experiments: ML

This Chapter presents the application of the ML methods, approaches and concepts outlined in Chapter 4 using a scientific approach on the IARs dataset. The main objective is to first get an insight into the performance of U-net with the dataset and generate compelling evidence as to whether this DL-based method could indeed be used to replace the image detection part of the current data analysis method developed by BGS.

5.1 Experimenting with default U-net implementation on Cell dataset

During the project preparation work between January and March 2019, a well-maintained Github repository that implemented the U-net architecture was reviewed [86]. The initial step was to ensure the validity and the working status of the default U-net implementation against the Cell dataset. The network was run for 5 epochs and it achieved a training loss and accuracy of 0.2287 and 0.9235, respectively. Predictions on test data are shown in Fig. 5.1.

The results obtained with the Cell dataset further strengthen the rationale for using the U-net architecture in the dissertation project, as the predicted segmentation map obtained with this dataset shares some resemblance with the type of binary classification (i.e. black lines on white background) portrayed in the segmentation maps that are associated with the IARs training examples (Fig. 4.2).

It is worth highlighting that in the implementation by Zhixu [86] both the input image and segmentation map have same size (i.e. 256x256). This is possible by first resizing the input image to the desired size and by using padded convolutions. This is a variation from the original U-net implementation described in Ronneberger et al. [69], which on the contrary used unpadded convolutions, leading to a difference of 184 pixels in both height and width between the output segmentation map and input image. As mentioned

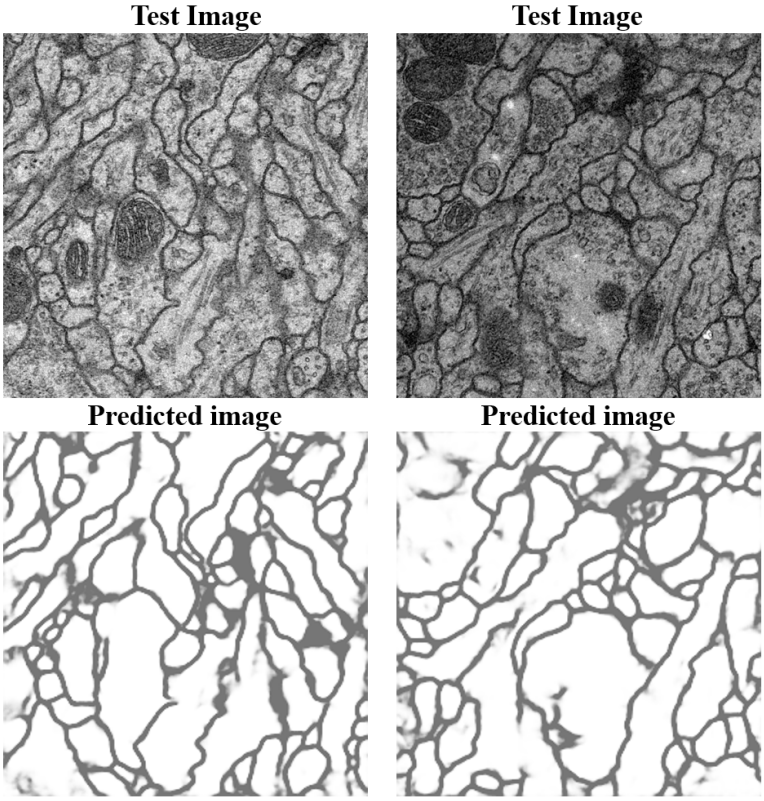

Figure 5.1: **Prediction experiment using default U-net implementation on Cell dataset**. U-net implementation proposed in [86] was trained on the Cell dataset for 5 epochs and 300 steps per epoch. Sample test images and associated predicted images are shown.

in Section 3.4, padding allows to maintain the size (i.e. height and width) of the input volume to a given convolution. However, due to the introduction of padding in all the convolutions, an additional convolution with 2 filters before the last convolution in the network had to also be introduced in the implementation by Zhixu [86] in order to allow for the output segmentation map to match the input image size.

5.2 Choosing the number of epochs for IARs dataset

The next thing to do was to run the default U-net implementation proposed in Zhixu [86] on the IARs dataset. The default U-net implementation did not employ k-fold cross-validation or other techniques in order to evaluate the model for training , and used two independent training and test sets of 30 images each. In a similar fashion, a relatively small dataset like the IARs dataset could be split into training and test sets according to a ratio 80:20 (Fig. 4.3). Given that the dataset contains 178 images, this works out as 142 images as training set and 36 as test set. The dataset files were listed alphanumerically (i.e. *CH2 2012 09 02.jpg* comes before *CH2 2012 09 05.jpg*) and the first 142 training examples were used as training data, while the last 36 were used as

Table 5.1: **Training Loss and IoU values from default U-net implementation on IARs dataset with different number of epochs**. U-net implementation proposed in [86] was independently trained on the IARs dataset for 1, 3, 5 or 10 epochs, with steps per epoch set to 71. Final training loss and IoU values are reported.

	1 epoch	**3 epochs**	**5 epochs**	**10 epochs**
Training Loss	0.3800	0.2910	0.2862	0.2767
Training IoU	0.7719	0.8193	0.8217	0.8270

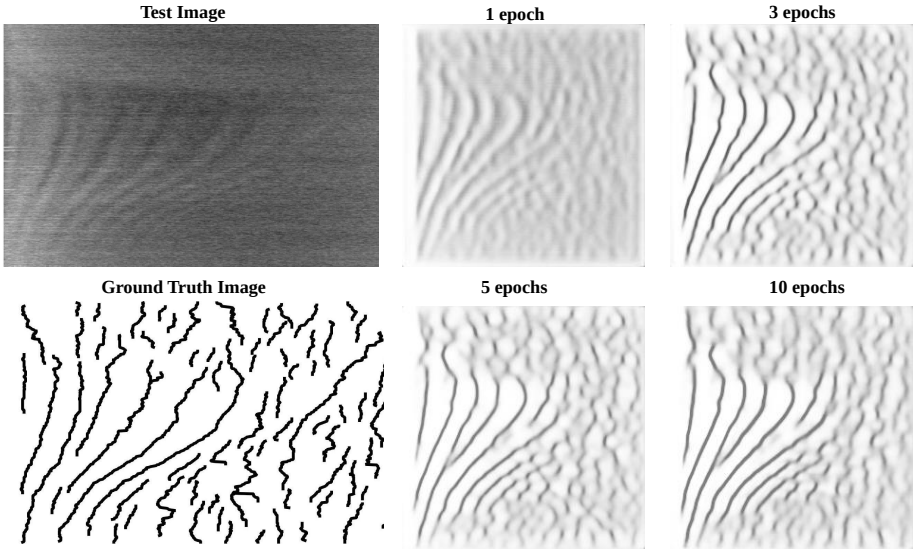

Figure 5.2: **Predicted segmentation maps from standard U-net implementation on IARs dataset**. U-net implementation proposed in [86] was independently trained on the IARs dataset for 1, 3, 5 or 10 epochs, with steps per epoch set to 71. Prediction on test image at end of training for different number of epochs is displayed. Ground truth image (not used by the algorithm) for the test image is also shown for reference. Test image belongs to 15/06/2018 (day/month/year).

test data. In the default implementation by Zhixu [86], the number of epochs was set to 1 and steps per epoch was set to 300. Moreover, the repository creator stated that after 5 epochs, the calculated value for the accuracy metric is approximately 0.97 on the Cell dataset. Since 300 appears to be quite an arbitrary number, steps per epoch was set to be 71 instead (i.e. 142 divided by batch size value, which is set at 2 in the implementation). This would reflect the fact that 1 epoch would actually coincide with one (forward and backward) pass through the dataset. The network was trained for 1, 3, 5 and 10 epochs in order to investigate whether the network could be trained on the IARs data, and whether the number of epochs could be tuned to improve the training dynamics and predictions on test data. Resulting training loss and IoU are shown in Table 5.1, while predictions on test data are shown in Fig. 5.2.

By considering the final loss and IoU values reached at the end of training along with the qualitative assessment of the predictions on test data, it appears that the network already performs very well on the IARs dataset by just setting the number of epochs to 3. Further increasing the number of epochs does not really have an impact on either the training metrics and the quality of predictions on test data.

5.3 Choosing k for cross-validation with IARs dataset

As mentioned in Section 4.2.2, k-fold cross-validation can be used in order to estimate the test set error when a model is trained on a given dataset. A diagrammatic description of the k-fold cross-validation algorithm is given in Fig. 4.4. An implementation detail of the algorithm that is worth discussing is that in order to remove any potential correlation among training examples in the IARs dataset, the dataset is shuffled using a random number generator with a set seed value prior to splitting into folds. The use of a random number generator in order to shuffle data allows for consistency for different k-fold cross-validation experiments (e.g. with different values of k) and reproducibility for different repeats of the same experiment.

A cross-validation experiment was performed using the entire IARs training set and the parameter k, which controls the number of folds in the algorithm, set to 2, 5 or 10 on 10 epochs in order to get an idea of the amount of variation between the folds and whether k had any effect on training dynamics (Fig. 5.3). As the dataset is now being split into training and validation set, the loss function value and IoU score that are computed on the respective validation fold at each iteration of the algorithm are denoted as 'validation loss' and 'validation IoU'. Although a value of 10 generated the best values for training loss, validation loss and validation IoU, it appears that a value of 10 is too large for the IARs dataset, as there is very little variation in the training loss, validation loss and validation IoU among folds as indicated by the error bars. Moreover, although a value of 5 seems to exhibit the largest amount of variation between folds, it appears to generate suboptimal values for the training loss, validation loss and validation IoU. Finally, a value of 2 seems to exhibit the greatest trade-off between training dynamics and amount of variation between folds. Considering that a value of 2 is also advantageous in terms of computational resources, all subsequent experiments in the project that involved k-fold cross-validation were conducted with k set to 2.

5.4 Experimenting with different loss functions

Although pixel-wise binary cross-entropy loss is the most commonly used loss function for the task of image segmentation, overlap based loss functions such as the Dice coefficient and the Jaccard distance loss are also suitable in theory. Hence, an experiment was carried out by independently training the network for 10 epochs with each of these losses. An explanation of these loss functions along with their equations is given in Section 4.2.3.

As binary cross-entropy exhibits the most optimal values for training loss and IoU (Table 5.2), namely lowest training loss and highest IoU, it was chosen over the other two loss functions for the rest of the project. Using binary cross-entropy appears to be advantageous as it has more resolution compared to the other two loss functions. Resolution here means that the loss function hs greater potential to optimize the loss

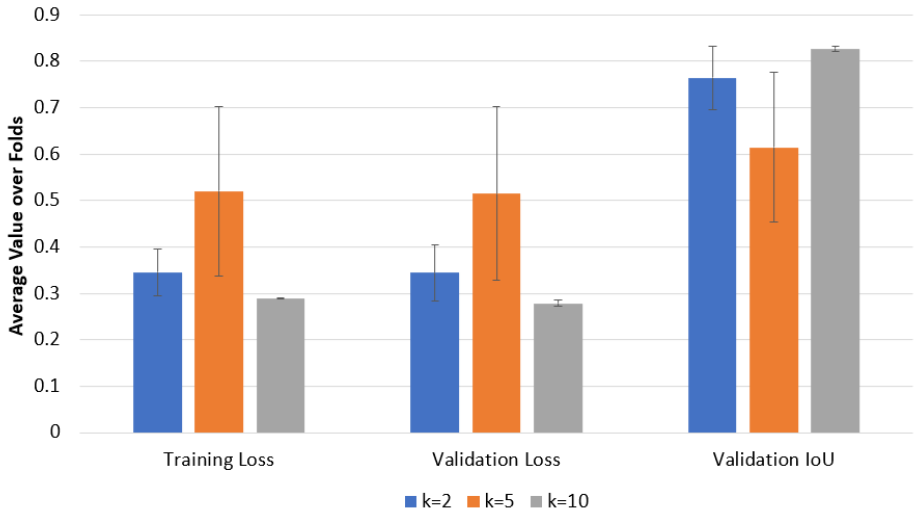

Figure 5.3: **Effect of k on training dynamics for IARs dataset**. U-net implementation proposed in [86] with additional code developed in the dissertation for k-fold cross-validation was independently trained on the IARs dataset for 10 epochs with different values for k. Final training loss and IoU values are reported. Average and standard deviation of training loss, validation loss and validation iou values over the set number of folds are reported.

Table 5.2: **Training Loss and IoU values from default U-net implementation on IARs dataset with different loss functions**. U-net implementation proposed in [86] was independently trained on the IARs dataset for 10 epochs with binary cross-entropy, Dice coefficient and Jaccard distance as loss function. Final training loss and IoU values are reported.

	Binary cross-entropy	**Dice coefficient**	**Jaccard distance**
Training Loss	0.2767	0.6201	0.4794
Training IoU	0.8270	0.4494	0.4865

and IoU value such that these can take on values during training that can span a larger range.

5.5 Comparison of training dynamics for two U-net implementations

An alternative, well-documented implementation in Keras with Tensorflow back-end described in Zak [84] was chosen for further evaluation in order to determine whether it exhibits any potential advantages with respect to the implementation made by Zhixu [86], which has been used in the project so far. After having introduced the necessary code for the IoU metric in the implementation proposed by Zak [84], both implementations were tested with the parameters laid out in the respective, original implementations while setting the number of epochs to 10 and k to 2 (Table 5.3). As it can be noted, the differences between these two implementations pertain to the the dropout value, the Keras Image Data Generator arguments, the input image size and

Table 5.3: **List of hyperparameters values, Keras Image Data Generator arguments for data augmentation and implementation features for the X and K U-net implementations**. These values are based on the proposed, original implementation of the X and K networks on the Cell dataset.

Hyperparameter	**X U-net**	**K U-net**
Batch size	2	2
Dropout	0.5	0.2
# Epochs	5	10
Optimizer	Adam	Adam
Learning rate	1e-4	1e-4
Loss function	Binary cross-entropy	Binary cross-entropy
Weight initializer	he normal	he normal
Generator argument	**X U-net**	**K U-net**
Rotation range	0.2	15
Width shift range	0.05	0.05
Height shift range	0.05	0.05
Shear range	0.05	50
Zoom range	0.05	0.2
Horizontal Flip	True	True
Vertical Flip	N/A	True
Fill mode	nearest	constant
Feature	**X U-net**	**K U-net**
Evaluation metric	Accuracy	Accuracy
Input image size	256x256	512x512
Architecture encoding	Coded each single layer manually	Divided into convolutional blocks

the architecture encoding. For simplicity, throughout the rest of this manuscript the implementation proposed by Zhixu [86] and Zak [84] are referred as X and K U-net, respectively.

K-fold cross-validation was used in order to get an estimate of the validation loss, validation IoU and validation accuracy for the two implementations of U-net. Plots of the training dynamics for X U-net and K U-net are shown in Fig. 5.4 and 5.5, respectively. The respective average of validation loss, validation IoU and validation accuracy at the end of training across the two folds was computed and is reported in Table 5.4. Total execution time for both implementations (excluding code required for importing Python packages and for plotting training dynamics) was also recorded independently when the different U-net implementations were trained on a single GPU (see Section 4.3.2). One general observation that can be made is that the X U-net has a relatively smoother curve for training and validation loss on fold 1 compared to the K-unet. Moreover, the accuracy metric does not exhibit any noticeable improvements for the X U-net and it performs worse than IoU for the K U-net. This illustrates the clear drawback of using accuracy as evaluation metric. As described in Section 4.2.3, due class imbalance, namely a large difference in the number of predictions falling into each class, the accuracy score is strongly biased towards predicting background pixels, which is the class with the largest representation in the IARs dataset, and is therefore uninformative. In light of this, the accuracy metric was not monitored any longer in subsequent experiments.

Table 5.4: **Comparison of training dynamics and execution time for X and K U-net**. U-net implementation proposed in [86] with additional code developed in the dissertation for k-fold cross-validation was independently trained on the IARs dataset for 10 epochs with k set to 2 on the IARs dataset. The reported values for validation loss, validation IoU and validation accuracy are the averages over folds. Execution time is the time taken to run the entire programme on 40 Skylake CPU cores and is the result of a single experiment.

	X U-net	**K U-net**
Validation Loss	0.34185	0.37595
Validation IoU	0.7647	0.70175
Validation Accuracy	0.4921	0.6375
Execution time (seconds)	165.51	351.97

Since the X U-net has a lower validation loss, greater validation IoU and markedly lower execution time than the K U-net (likely due to smaller input image size) on 40 Skylake CPU cores, the X U-net was chosen for further experiments.

5.6 Hyperparameter tuning with k-fold cross-validation

5.6.1 Overview of hyperparameters for training U-net

As mentioned in Section 4.2.2, k-fold cross-validation can also be used in order to find the best hyperparameters of a model. Out of the 7 hyperparameters from U-net, the loss function and number of epochs have already been explored in Sections 5.2 and 5.4. This leaves 5 hyperparameters to be explored further. A complete review of the role each of these hyperparameters has on training dynamics for CNNs is outside the scope of this dissertation. The first step of a grid search experiment is the definition of values of the hyperparameters to be explored. Batch size refers to the number of training samples propagated through the network during training and should be set to multiples for either 2, 4, 8, 16 or 32 [66]. For this dissertation, multiples of 2 up until 64 were initially chosen for exploration, but a batch size of 64 appeared to be problematic in terms of memory allocation for the GPU node on Cirrus, so it could not be tested.

Dropout is a regularization method that helps to prevent overfitting and it is applied by randomly dropping neurons along with their incoming and outgoing connections from the neural network during training [75]. The Dropout value is a hyperparameter that specifies the fixed probability at which each unit in a given layer is retained, namely it is not 'dropped out'. The optimal dropout value for a hidden layer is typically 0.5. Dropout values ranging from 0.1 to 1.0 were chosen for exploration in the dissertation. In particular, the implementation of U-net by [86] introduces Dropout with values of 0.5 after the second convolution of the fourth and fifth convolutional blocks of the U-net (Fig. 3.13).

The learning rate defines how quickly the internal parameters of the network (e.g.

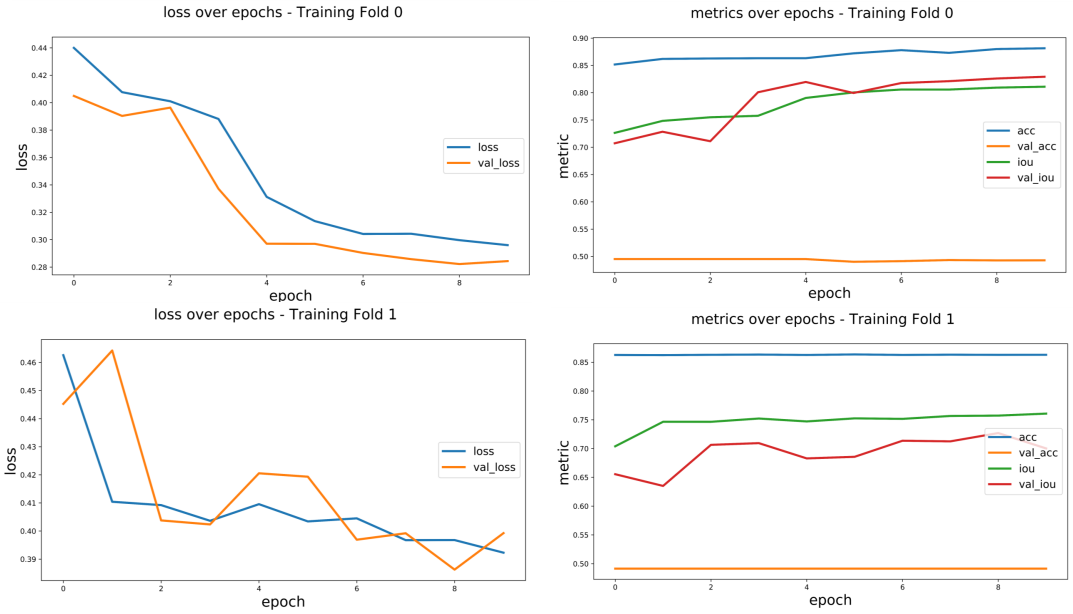

Figure 5.4: **Plots of training dynamics for X-unet**. U-net implementation proposed in [86] with additional code developed in the dissertation for k-fold cross-validation was trained on the IARs dataset for 10 epochs with k-fold cross-validation with k set to 2. Plots on the left display evolution of loss and validation loss over 10 epochs for fold 0 (top) and fold 1 (bottom). Plots on the right display evolution of accuracy, validation accuracy, IoU and validation IoU over 10 epochs for fold 0 (top) and fold 1 (bottom). The plotted values are the outputted values at the beginning of each epoch, with the first epoch being epoch 0 on the plot, and are the result of a single experiment.

weights) are updated. The range of values to be considered for the learning rate must be less than 1.0 and greater than 10^{-6} [6]. Values of 0.01, 0.001 and 0.0001 were chosen for exploration in the dissertation.

Optimizer denotes the algorithm used for optimizing stochastic gradient descent [38]. Keras offers several optimizers that are popular within the DL community [40]. In addition to the baseline implementation of SGD with momentum and Adam, which is a well-performing and currently very popular optimizer [36], another adaptive method RMSprop was chosen for exploration in the dissertation (see the Keras webpage [40] for a brief description and the references to the individual research papers behind each optimizer). Adaptive means that these methods change the learning step according to the topology of the loss function contour.

Weight initialization refers to the fact that, as part of stochastic gradient descent, the weights of the neural network must be initialized to small, random numbers. This process has been shown to have an impact on the rate of convergence [25]. Keras offers several weight initializers that are popular within the DL community [39]. All the optimizers with a literature reference present on Keras webpage were chosen, namely lecun uniform, glorot normal, glorot uniform, he normal, lecun normal, he uniform and orthogonal.

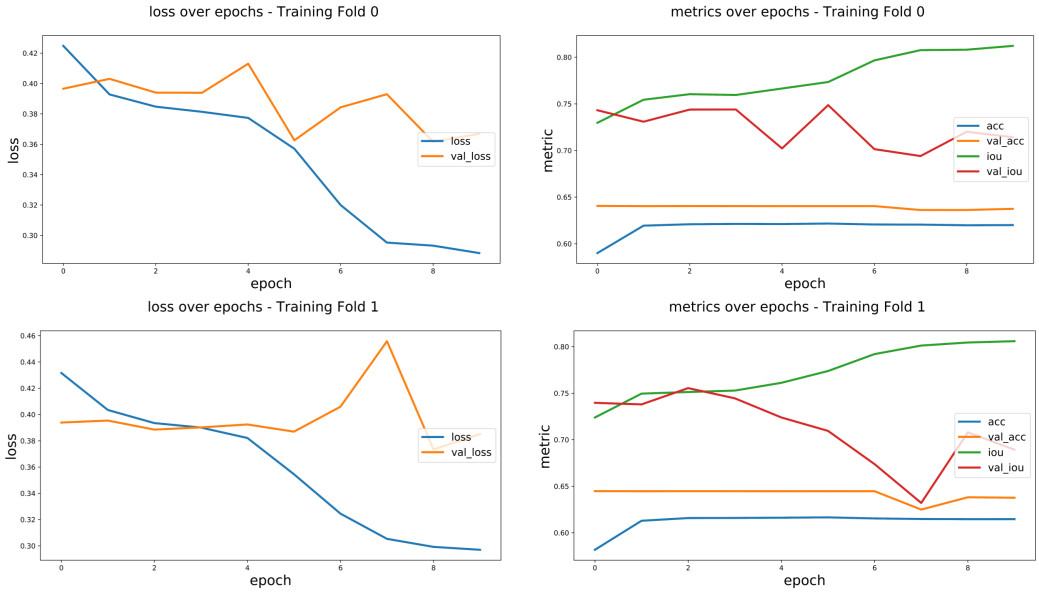

Figure 5.5: **Plots of training dynamics for K-unet**. U-net implementation proposed in [84] with additional code developed in the dissertation for k-fold cross-validation was trained on the IARs dataset for 10 epochs with k set to 2. Plots on the left display evolution of loss and validation loss over 10 epochs for fold 0 (top) and fold 1 (bottom). Plots on the right display evolution of accuracy, validation accuracy, IoU and validation IoU over 10 epochs for fold 0 (top) and fold 1 (bottom). The plotted values are the outputted values at the beginning of each epoch, with the first epoch being epoch 0 on the plot, and are the result of a single experiment.

5.6.2 Results of grid search

A diagrammatic description of the application of k-fold cross-validation for hyperparameter tuning is given in Fig. 4.5. Based on a preliminary test, training U-net with a single permutation of the 5 hyperparameters with k set to 2 and number of epochs set to 10 takes $\approx$ 100 seconds on a single GPU. Considering that the total number of permutations of the hyperparameter values is 3150 (i.e. 5 values for batch size, 10 values for dropout, 3 values for learning rate, 3 values for optimizer and 7 values for weight initialization), such experiment would take around $\approx$ 88 hours. Since Cirrus places a limitation of 6 hours as the maximum runtime of any job run on a GPU and parallelizing such computation may constitute an arduous task, the immediate solution would be to split the grid search into two non-exhaustive searches with permutations based on different subsets of the hyperparameters.

An additional implementation feature that was introduced in order to facilitate the grid search is the *EarlyStopping* callback provided by the Keras API. This is essentially a method that allows an arbitrary large number of epochs to be specified before training and to stop training once the performance of the model stop improving. In turn, this is assessed by monitoring the validation loss measured on a given validation fold during training with k-fold cross-validation. A value of 0.01 for the *min_delta* argument to the callback was chosen, meaning that an improvement to the validation loss smaller than

Table 5.5: **Results of grid search for tuning batch size, optimizer and learning rate**. U-net implementation proposed in [86] with additional code developed in the dissertation for k-fold cross-validation was independently trained on the IARs dataset with k set to 2 on the IARs dataset using each of the 45 permutations of the values for batch size, optimizer and learning rate. Keras's *EarlyStoping* callback was used in order to stop training when validation loss on a given iteration of k-fold cross-validation stopped improving. The cross-validation score is calculated as the average of validation loss over the folds for a given permutation of the hyperparameters.

Hyperparameter	Best Permutation	Second Best Permutation	Third Best Permutation
Batch size	2	2	2
Learning rate	0.0001	0.0001	0.01
Optimizer	Adam	RMSprop	SGD
Cross-validation score	0.2891	0.2926	0.2927

Table 5.6: **Results of grid search for tuning dropout and weight initializer**. U-net implementation proposed in [86] with additional code developed in the dissertation for k-fold cross-validation was independently trained on the IARs dataset with k set to 2 on the IARs dataset using each of the 70 permutations of the values for dropout and weight initialization. Keras's *EarlyStopping* callback was used in order to stop training when validation loss on a given iteration of k-fold cross-validation stopped improving. The cross-validation score is calculated as the average of validation loss over the folds for a given permutation of the hyperparameters.

Hyperparameter	Best Permutation	Second Best Permutation	Third Best Permutation
Dropout	1.0	0.5	1.0
Weight initializer	he uniform	he uniform	lecun normal
Cross-validation score	0.2889	0.2918	0.2930

this value would be classified as no improvement and it would trigger end of training on that particular iteration of the k-fold cross-validation algorithm. Using this callback is helpful as it prevents unnecessary training after the predictive model has stopped improving when trained a on given permutation, which reduces the overall execution time and also prevents unnecessary use of computational resources.

A grid search experiment using k-fold cross-validation was performed, with dropout and weight initialization mode set to values of 0.5 and he normal [39] (which are the default values for X U-net implementation), respectively, while changing values for batch size, optimizer and learning rate. The 3 permutations with the best cross-validation scores are listed in Table 5.5. An additional, non-exhaustive grid search experiment was then performed while changing values for dropout and weight initialization on top of the best permutation of values for batch size, learning rate and optimizer identified in the previous hyperparameter search. The 3 permutations with the best cross-validation scores are listed in Table 5.6.

The best permutations from the two individual grid searches have very similar cross-validation scores. However, as these experiments both technically constitute non-exhaustive grid searches, it is not possible to argue whether any of the hyperparameter values that has been identified as optimal across the two grid searches is indeed absolutely optimal for training U-net on the IARs dataset. That being said, the fact that the best permutations from both grid searches have similar cross-validation

Table 5.7: **Training loss and IoU values for network independently trained with three best permutations of hyperparameter values on entire IARs training set**. U-net implementation proposed in [86] was separately trained for 10 epochs on the three best permutations of batch size, learning rate and optimizer identified in the first grid search (see Table 5.5) on the entire IARs training set without k-fold cross-validation. The reported values for training loss and training IoU are the result of a single experiment.

Parameter Permutation	Training Loss	Training IoU
Best	0.2783	0.8265
Second Best	0.2827	0.8233
Third Best	0.2857	0.8225

scores, combined with the fact that the optimal values identified through tuning for 3 out of 5 hyperparameters correspond to the default values used so far in the project, suggests that the choice of hyperparameter values proposed by [86] is already quite suitable for training U-net on the IARs dataset.

5.7 Experiments on finalized model

In order to finalize the predictive model (Fig. 5.6), U-net was separately trained on the entire IARs training set (178 training examples) with the 3 best parameter permutations generated from the first grid search (Table 5.5) and used to generate predictions for the test dataset of 2135 days. For this, the original code from Zhixu [86] was used without cross-validation. Training loss and IoU were monitored for 10 epochs for all the 3 permutations. On all the permutations, the network exhibits very similar training dynamics, as it is able of reaching convergence very quickly after about 4 epochs (Fig. 5.7). Plots for the training dynamics when the network was trained on second and third best permutation are not shown, as they are very similar anyway to the plot with the best permutation. As expected, the best permutation has the lowest training loss and highest training IoU after 10 epochs (Table 5.7). That being said, the differences between the final training loss and IoU for the models trained on the 3 different permutations are minimal and considering the stochasticity of the training process, it can be argued that the 3 models have essentially comparable performance. From here onwards, the values associated with the best hyperparameter permutation shown in Table 5.5 will be referred as default hyperparameter values for simplicity.

In a similar fashion as the ground truth images used as part of training, the predictions on test data contain some noise represented by non-IARs specific signal. However, unlike the current data analysis method developed by BGS, the trained classifier is able of assigning a probability value to all the pixels in the segmentation map (see Section 5.11). This means that it is possible to remove some of the noise from the segmentation map by simply converting all the values that are larger than a given threshold to be 1 (i.e. a white pixel). Different thresholds in the range 0.4-0.6 were tested, but it seems that by visually comparing the thresholded prediction image against the ground truth image,

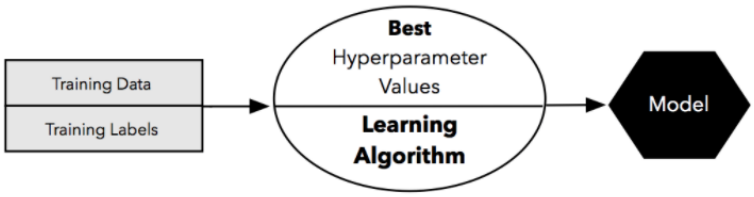

Figure 5.6: **Model finalization**. After the best hyperparameter values for a predictive model have been chosen and the performance of the ML algorithm has been reliably estimated with techniques such as k-fold cross-validation, the model must be finalized by retraining it on the entire dataset.

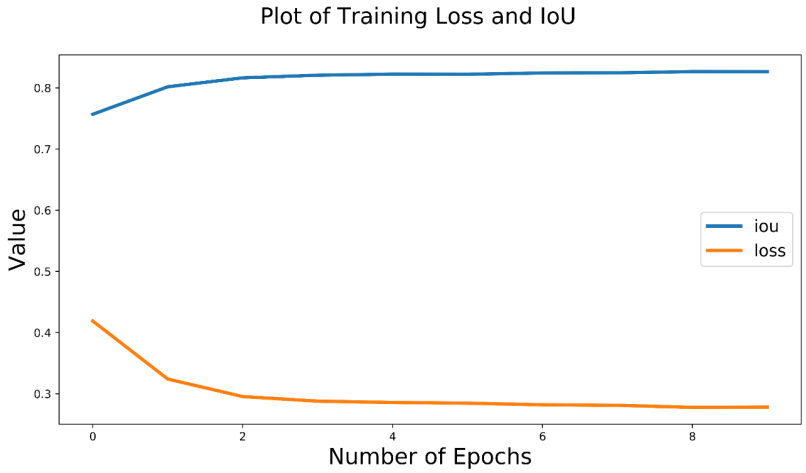

Figure 5.7: **Training dynamics for network trained with best permutation of hyperparameter values on entire IARs training set**. U-net implementation proposed in [86] was trained for 10 epochs on the best permutation of batch size, learning rate and optimizer (see Table 5.5) on the entire IARs dataset without k-fold cross-validation. The plotted values are the outputted values at the beginning of each epoch, with the first epoch being epoch 0 on the plot, and are the result of a single experiment.

a threshold of 0.4 or 0.5 offers the best trade-off between correct signal detection and noise removal (Fig. 5.8).

Another observation that can be made from inspecting Fig. 5.7 is that training for 10 epochs seems to be unnecessary, as the training loss appears to virtually stop improving after 4 epochs. As a test, the network was separately trained for 5, 7 and 10 epochs using the default hyperparameter values and the generated prediction images on test data were subsequently thresholded with a threshold value of 0.5 (Fig. 5.9). The idea of this experiment is to identify the lowest value for the number of epochs to be used for training that still allows acceptable prediction images on test data to be generated. As the thresholded prediction image generated after U-net has been trained for 5 epochs is completely white, it appears that training for 5 epochs is simply not enough. Moreover, the thresholded prediction images generated when the network is trained for 7 or 10 epochs are very similar to each other. Furthermore, the difference in the final training

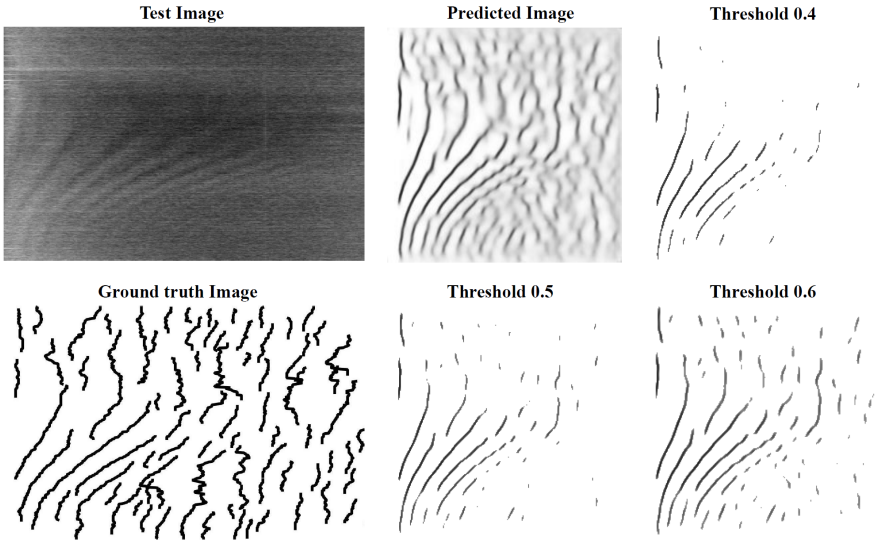

Figure 5.8: **Application of different thresholds to predicted segmentation map for an unseen example**. U-net implementation proposed in [86] was trained for 10 epochs on the default hyperparameter values on the entire IARs training set. Top left is test image, while bottom left is associated ground truth image. Remaining images show the effect of applying a threshold value of 0.4, 0.5 or 0.6 to the prediction images generated by the predictive model. Test image belongs to 15/06/2018 (day/month/year).

loss and IoU recorded on these two different epochs is minimal (Table 5.8). Although 7 epochs seems to offer a good trade-off between number of epochs and amount of IARs signal, the thresholded prediction image generated when the network is trained for 10 epochs still appears to have relatively less fragmented IARs signal. Since there is also no considerable difference between 7 and 10 epochs in terms of execution time (Table 5.8), the choice of 7 epochs over 10 does not seem to bring enough benefits to justify it and therefore using 10 epochs for training still constitutes the preferred choice.

An additional observation that can be made based on the results shown in Fig. 5.9 is that, as it would be expected, training the network for more epochs essentially results in a more certain output. This is substantiated by the fact that less black pixels disappear in the thresholded image after U-net is trained for 10 epochs compared to 7. Also the black pixels that are labeled as IARs signal with 7 epochs take on a darker black colour when the network is trained for 10 epochs, suggesting that such pixels are assigned values that are closer to 0 as a result of longer training.

One argument that could be made against training for more epochs is that it may result in the network becoming more certain about the noise as well. That being said, it appears that U-net does not face this issue, as the difference in the amount of non-IARs signal predicted after training for 7 or 10 epochs is minimal.

An additional argument that could be made against training for more epochs is that it may result in overfitting, namely the predictions of U-net may correspond too closely to the training data and as a result U-net may fail to generalize well on unseen data. This argument can be countered by considering the fact that in the plots of training dynamics

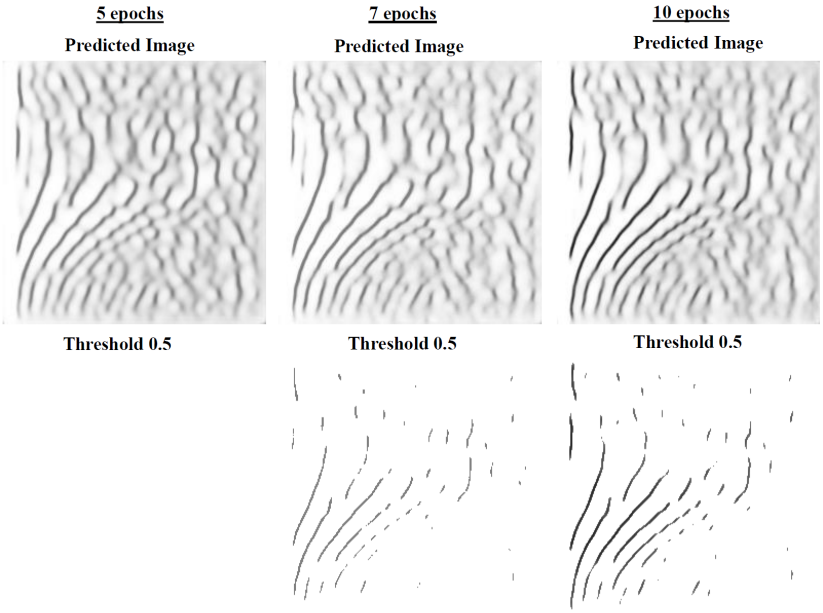

Figure 5.9: **Investigating effect of number of epochs on signal quality in prediction images**. U-net implementation proposed in [86] was independently trained for either 5, 7 or 10 epochs on the default hyperparameter values on the entire IARs training set and the resulting prediction images were thresholded with a threshold value of 0.5. Predicted image belong to 15/06/2018 (day/month/year).

generated with k-fold cross-validation, the final value for validation loss never becomes larger than the initial one 5.4. Moreover, as it can be seen in Fig. 5.10, validation loss stops improving after about 50 epochs, indicating that at that point the network is starting to overfit to the training data. This further suggests that U-net is certainly not overfitting to the IARs dataset when trained for 10 epochs.

Table 5.8: **Finding optimal number of epochs - training loss and IoU values**. U-net implementation proposed in [86] was independently trained for either 5, 7 or 10 epochs on the default hyperparameter values on the entire IARs training set. The reported values for training loss and training IoU are the result of a single experiment. Training runtime is the time recorded for section of the code required for training U-net.

# Epochs	Training Loss	Training IoU	Training Runtime (seconds)
5	0.2903	0.8200	75.20
7	0.2805	0.8253	102.41
10	0.2783	0.8265	137.59

5.8 Qualitative assessment of predictive model

In order to assess the goodness of the trained segmenter against the data analysis method developed by BGS, the thresholded prediction images on test data can be overlaid on top of the ground truth images. The images in the ‘Overlay‘ column in Fig. 5.11 illustrate this operation: red pixels indicate pixels that are labeled as (white) background in both

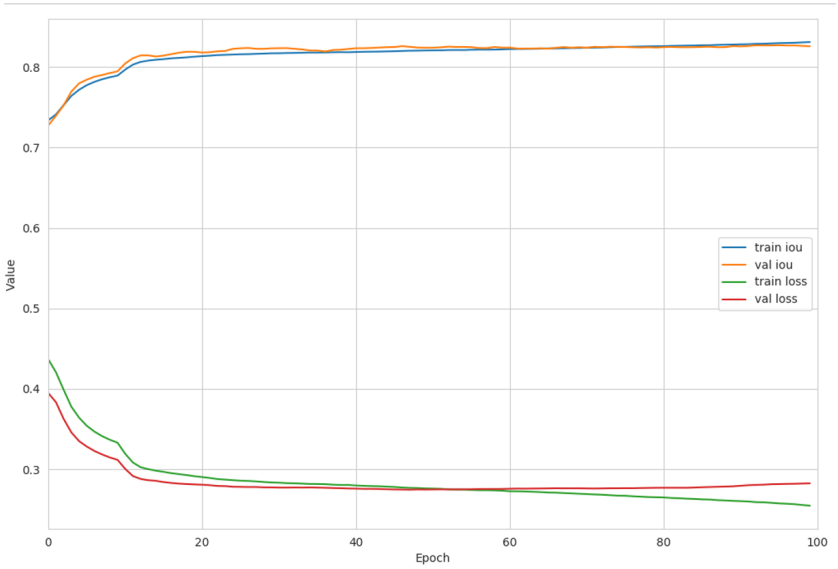

Figure 5.10: **Overfitting test on IARs dataset - fold 1**. U-net implementation proposed in [86] with additional code developed in the dissertation for k-fold cross-validation was trained for 100 epochs on the IARs dataset, with k set to 2. Training loss, validation loss, training IoU and validation IoU for fold 1 were monitored. The plotted values are the outputted values at the end of each epoch and are the result of a single experiment.

images; black pixels indicate pixels that are predicted as IARs signal in both the ground truth image and the thresholded prediction image; white pixels indicate pixels that are predicted as IARs signal in ground truth images, but as background in the thresholded prediction image. In other words, the white lines in the images in the 'Overlay' column highlight the noise that is present in the predictions made with the current data analysis method, but absent when the predictions are made on the same test images with the trained segmenter. As the white pixels tend to be associated with fragmented lines located in regions of the image where IARs signal is mostly not expected to be present anyway, the trained classifier appears to be less prone to predicting noise. It can therefore be concluded that from a qualitative point of view the classifier exhibits a lower signal to noise ratio than the current data analysis method developed by BGS.

A negative control experiment was performed in order to confirm that the network has actually been trained to detect meaningful patterns in new, unseen data. The network was presented with an image that contains no IARs signal, therefore a noisy example (Fig. 5.12). As expected, the prediction on the test image contains a lot of gray or blurred signal, suggesting that the network is unable to classify pixels with enough certainty. By applying a threshold of 0.4 or 0.5, it is possible to nearly obtain a completely white segmentation map consisting of just background, which is the desired outcome for such a noisy test image. This test provides further support for generally choosing a threshold of 0.4 or 0.5 for thresholding prediction images.

An additional negative control experiment was carried out by presenting a test image that contains a confounding signal originating from manmade interference to the trained

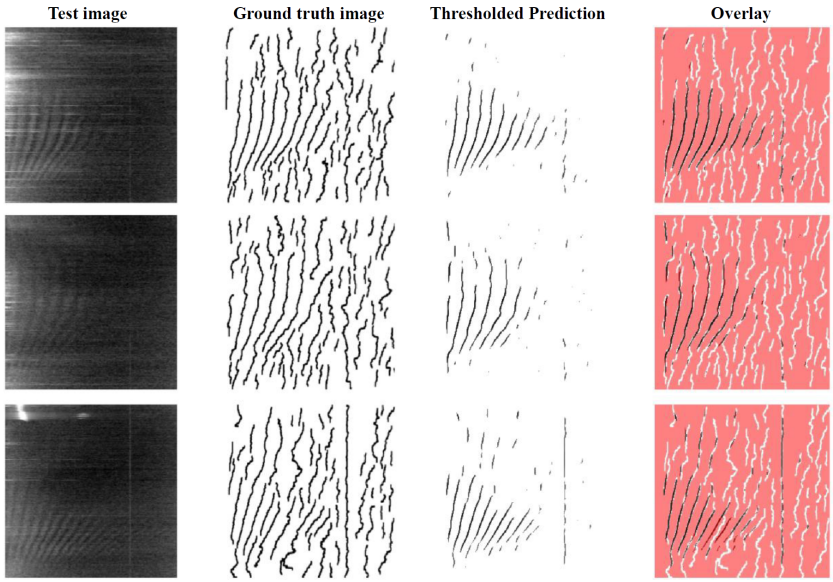

Figure 5.11: **Qualitative assessment of performance of current data analysis method developed by BGS against trained segmenter**. U-net implementation proposed in [86] was trained for 10 epochs on the default hyperparameter values on the entire IARs training set. First column from left is the test image for 3 random days drawn from the IARs test set of 2135 days. Second column is the respective ground truth image generated by the current data analysis method developed by BGS. Third column is the respective thresholded (with value of 0.5) prediction image generated by trained segmenter on the test image in the first column. Fourth column is the overlay of the thresholded prediction image in the third column on the ground truth image in the second column. Colour key for pixels in the overlay image is as follows: red pixels indicate pixels that are labeled as (white) background in both images; black pixels indicate pixels that are predicted as IARs signal in both the ground truth image and the thresholded prediction image; white pixels indicate pixels that are predicted as IARs signal in ground truth image, but as background in the thresholded prediction. From top to bottom, test and associated ground truth images belong to 05/09/2012, 06/09/2012 and 07/09/2012 (day/month/year).

segmenter (Fig. 5.13). The prediction on the test image shows that the network recognizes the confounding pattern and classifies most of it as IARs signal with enough certainty. Although this test clearly shows one shortcoming of the trained segmenter, it should be considered that this kind of test image would never be intentionally presented to the segmenter anyway. Another insight that this test provides is that the segmenter almost certainly recognizes the brightness of fringe patterns as an image feature and uses it to make predictions. In fact, the first two vertical lines with bright white colour in the test image are poorly recognized as IARs signal, while the remaining four vertical lines with less bright white colour are detected by the segmenter with enough certainty as IARs signal (Fig. 5.13). Another feature that the segmenter appears to be identifying based on this test is the orientation of the fringe lines. As mentioned, the segmenter recognizes the bright lines that have a vertical orientation, but it does not discern the lines with horizontal orientation present on the left part of the test image (Fig. 5.13)

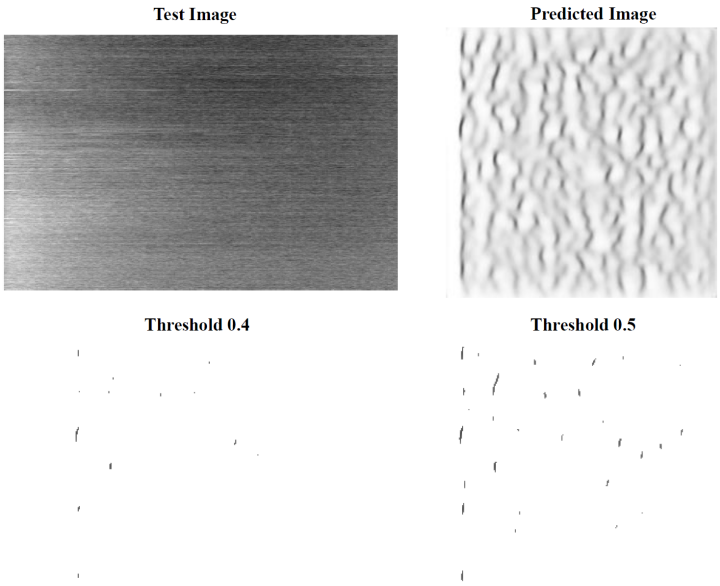

Figure 5.12: **Negative control test with noisy example - no signal**. U-net implementation proposed in [86] was trained for 10 epochs on the default hyperparameter values on the entire IARs training set. Top left is a test image for a day in the test set with no IARs signal. Top right is the resulting prediction image generated by the trained segmenter. Bottom left and right are the thresholded versions of the prediction image with a threshold value of 0.4 and 0.5, respectively.

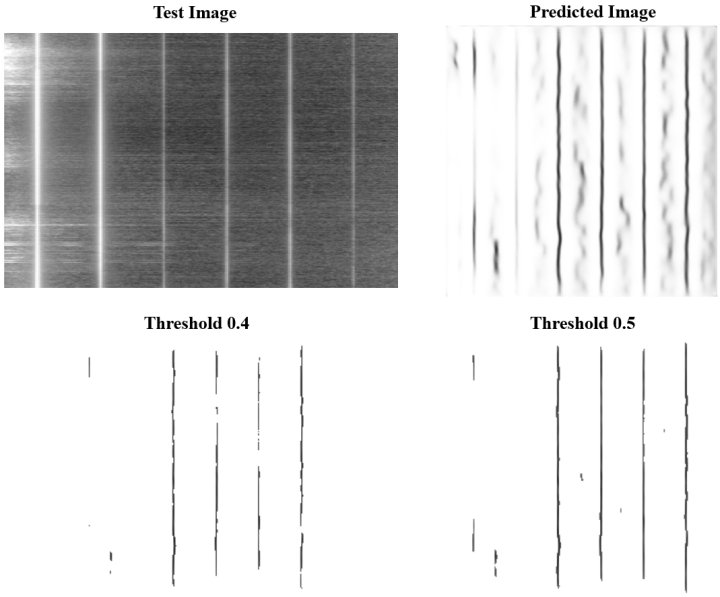

Figure 5.13: **Negative control test with noisy example - artificial signal**. U-net implementation proposed in [86] was trained for 10 epochs on the default hyperparameter values on the entire IARs training set. Top left is a test image for a day in the test set with IARs signal, but with confounding signal due to man-made interference. Top right is the resulting prediction image generated by the trained segmenter. Bottom left and right are the thresholded versions of the prediction image with a threshold value of 0.4 and 0.5, respectively.

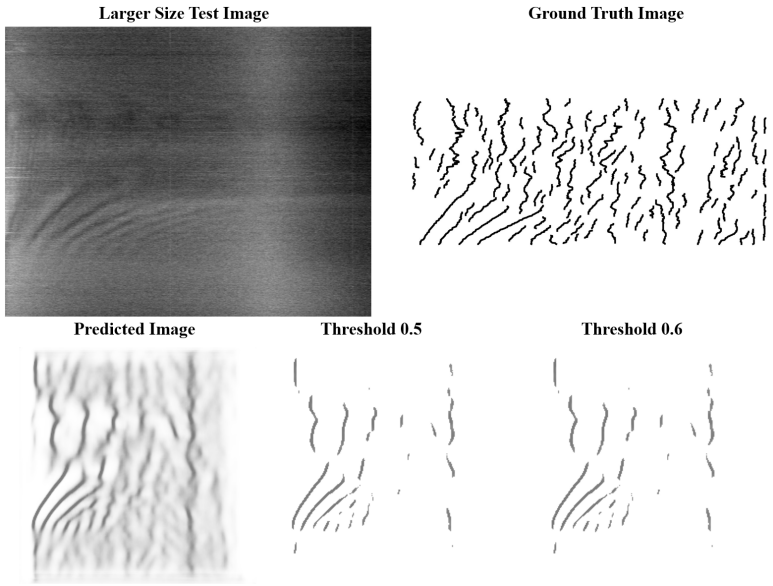

Figure 5.14: **Experiment with image of larger size when network is trained on images of smaller size**. U-net implementation proposed in [86] was trained for 10 epochs on the default hyperparameter values on the entire IARs training set. Top left is the larger size version of a test image drawn from the IARs test set of 2135 days. Top right is the associated ground truth image generated by current data analysis method developed by BGS. Bottom left is the resulting predicted image generated by the trained segmenter. Remaining images are the thresholded versions of the prediction images with a threshold value of 0.5 and 0.6. Test image belongs to 15/06/2018 (day/month/year).

5.9 Testing on images of larger size

One intermediate step in the current data analysis method developed by BGS that is responsible for the final generation of the images that were used in the project as training examples is the cropping of the images in both the x and y dimension, such that the final x axis ranges between 0.5 and 6.3 Hz and the final y axis ranges between 18:00 (i.e. 6 PM) and 06:00 (6 AM). However, if this step could be avoided, it would simplify the whole process. Moreover, since the length of night time can vary over the year (e.g. it becomes significantly longer in the winter in Scotland) and therefore the input image size may be larger in the y dimension, it would advantageous if the segmenter had the ability of accepting images of larger size as input.

Hence, as an additional experiment, the entire IARs training set of conventional size (701x1101) was used for training and the larger size version (1417x1802) of the test set was used for testing. The segmenter is remarkably able to predict some part of the IARs signal when it is presented an image of larger size (Fig. 5.14). That being said, the general impression that can be gained from the prediction and thresholded images is that the segmenter is not able to predict a signal that fully respects the proportions of the desired signal in the input test image. The fact that the network has been trained on images of smaller size is a likely explanation of this shortcoming.

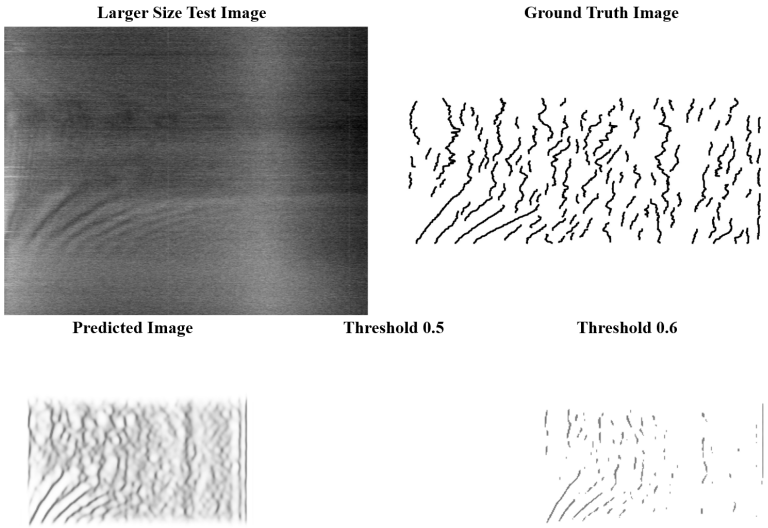

Figure 5.15: **Experiment with image of larger size when network is trained on images of larger size**. U-net implementation proposed in [86] was trained for 10 epochs on the default hyperparameter values on the entire IARs training set. Top left is the larger size version of a test image drawn from the IARs test set of 2135 days. Top right is the associated ground truth image generated by current data analysis method developed by BGS. Bottom left is the resulting predicted image generated by the segmenter after being trained with image of the same size as the test image. Remaining images are the thresholded versions of the prediction images with a threshold value of 0.5 and 0.6. Test image belongs to 15/06/2018 (day/month/year).

5.10 Training and testing on images of larger size

In order to confirm whether U-net performed poorly on test images of larger size due to the fact that it had been trained on images of smaller size, the same experiment outlined in Section 5.9 was repeated with the only difference that the network was trained on images of larger size. That is, the larger size version of the images belonging to the entire IARs training set was used for training. As it can be seen from Fig. 5.15, the quality of the prediction image is now satisfactory: the proportions of the desired signal in the test image are now maintained and there seems to be a larger agreement between the prediction image and the ground truth image (which is not used for training or testing, and is just displayed for clarity), especially when compared to the output of the the network trained with images of smaller size (see Fig. 5.14). Altogether, this experiment confirms that U-net can be trained and the resulting predictive model can be used for generating predictions for IARs test images with any size in the range between 701x1101 and 1417x1802, provided that both the training and test images have the same size.

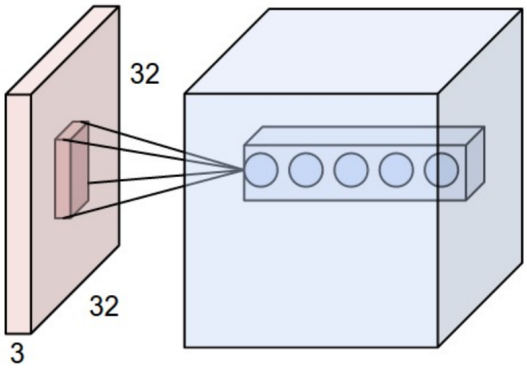

Figure 5.16: **Local connectivity and receptive field in CNNs**. This picture shows an example input volume (red) of size 32x32x3 and an example volume of neurons in the first convolutional layer (blue) of an hypothetical CNN. Each neuron in the convolutional layer is connected only to a local region in the input volume spatially, but to the full depth (i.e. all the 3 color channels). Also, there are multiple neurons (5 in this example) along the depth of the convolutional layer, all looking at the same region in the input. The depth of the output volume is a hyperparameter. It corresponds to the number of filters that will be used in the convolutional layer, each learning to look for something different in the input. A set of neurons that are all looking at the same region of the input are referred as a depth column or fibre. Fig. taken from [37].

5.11 Visualizing activations of U-net on IARs data

Neural networks are known to behave as a black box, in the sense that there is no simple link between the weights learnt by a network during training and the mathematical function that the neural network is actually trying to approximate. The Python keract [68] package has been created in order to attempt to address this issue. Keract offers a straightforward way to obtain the activations (outputs) for each layer of any given neural network moded coded in Keras.

In the downsampling part of any CNN, each layer contains a number of filters that are capable of detecting the presence of specific features or patterns that are present in the input volume to the layer. This is possible because each neuron in a given layer is connected to a localized region in the input volume, also known as ‘receptive field‘ of the neuron (Fig. 5.16). It must be noted that each neuron in a given convolutional layer is connected only to a local region in the input volume spatially, but to the full depth (i.e. all color channels). As it would be expected, every entry in the 3D output volume of a given layer is the dot product of a filter with a region in the input, which produces the response of that filter at that spatial location.

Using keract, it is possible to get a heatmap plot of the activation output from each respective filter when the input volume from the preceding layer is presented to a given convolution layer. For example, if the first convolutional layer takes as input the raw image, then different neurons along the depth dimension may activate in presence of various oriented edges, or blobs of color. Each small square created by keract displays the amount of activation associated with each filter at each position in the input volume

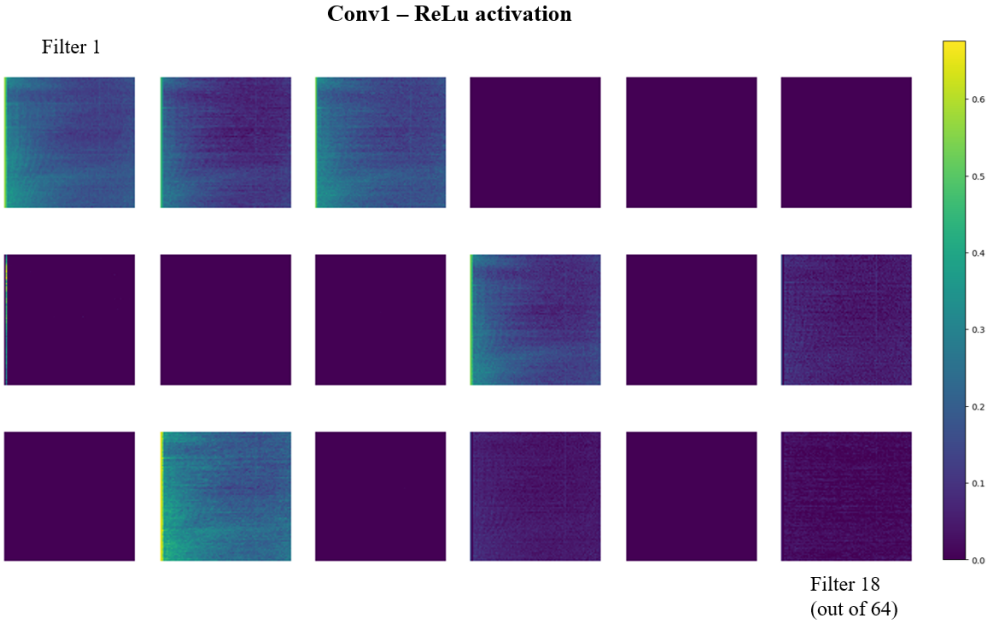

Figure 5.17: **Activations for filters of convolutional layer 1 in U-net architecture**. U-net implementation proposed in [86] was trained for 10 epochs on the default hyperparameter values on the entire IARs training set and then presented with an image from the IARs test set that belongs to 05/09/2012 (day/month/year). The activation maps associated with this test image were obtained using the keract python package. This figure shows the ReLu activation maps for the first 18 out of 64 filters applied as part of convolutional layer 1 of U-net onto the input image. The bar on the right is a key for interpreting activation values displayed within the individual maps: activations at 0.1 have a dark purple colour; activations at 0.3 have a colour between green and blue; activations at 0.4 have a green colour; activations at 0.5 have a colour between yellow and green; activations slightly above 0.5 have a yellow colour.

according to some scale that is computed based on both the type of input that is presented at each layer and the kind of activation function. For example, the activations from all the filters in the first convolution of U-net with ReLu activation span the range between 0.2 and 1.6 (Fig. 5.17). As mentioned, this is not an absolute scale.

It is widely known that the first layers in a CNN typically detect low-level features while later layers detect high-level features. In the example of the first convolution looking at the input image, its small receptive field (i.e. 3x3 kernel for U-net) forces it to only consider a very small context of the image such that it is only able to learn low-level features (e.g. blobs, edges or lines in a conventional CNN). By simply arranging neurons with such small receptive fields in layers and having each layer to process the signal originating from the previous one, the effective receptive field becomes larger at deeper convolutions in the network. This has been formalized as a linear relationship, where each extra layer increases the receptive field size by the kernel size [54].

Consistent with this, it can be seen that the filters from the first convolution mostly pick up sparse pixels (Fig. 5.17), while successive convolutions such as convolution 18, which is located in the upsampling part of U-net, detect lines by connecting the individual pixels (Fig. 5.18). As the last layer has a sigmoid activation function, the

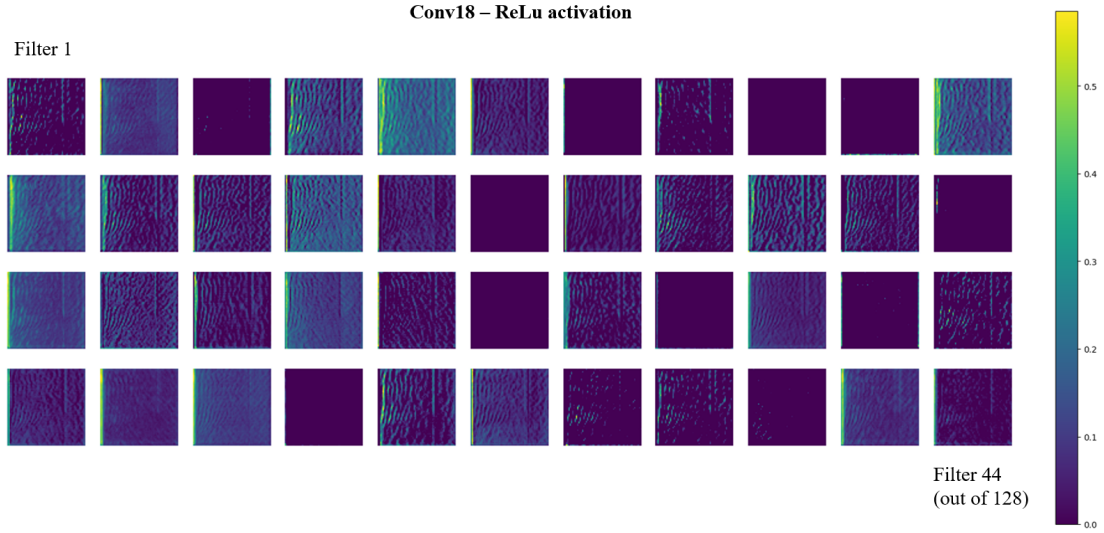

Figure 5.18: **Activations for filters of convolutional layer 18 in U-net architecture**. U-net implementation proposed in [86] was trained for 10 epochs on the default hyperparameter values on the entire IARs training set and then presented with an image from the IARs test set that belongs to 05/09/2012 (day/month/year). The activation maps associated with this test image were obtained using the keract python package. This figure shows the ReLu activation maps for the first 44 out of 128 filters applied as part of convolutional layer 18 (first convolution of the penultimate block in the upsampling part of the network) onto input volume to the convolutonal layer. See Fig. 5.17) for an explanation to the figure key on the right.

activation from this layer spans the range between 0 and 1 (Fig. 5.19), where pixels in the image that take on a probability value of 1 are represented as white (i.e. background class), while pixels in the image that take on a probability value of 0 are represented as black (i.e. IARs signal class). Moreover, as it would be expected, the pixels in the segmentation map that appear as grey or blurry and therefore do not clearly belong to either IARs signal or background class correspond to pixels that have a probability around 0.5 following sigmoid activation (i.e. greenish/bluish pixels in the keract output for conv24).

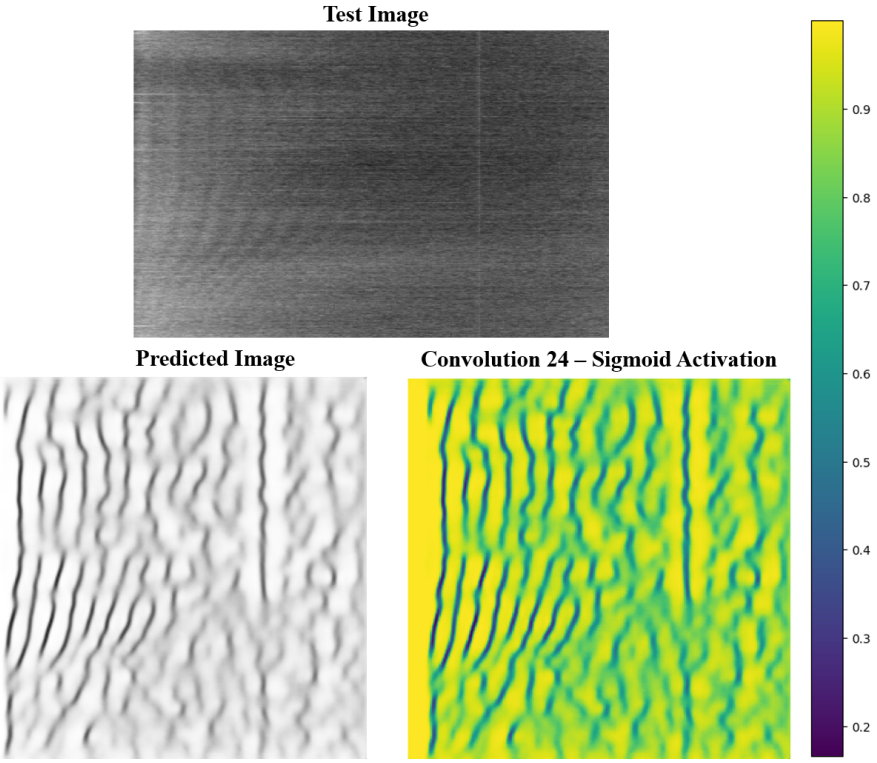

Figure 5.19: **Probability heatmap for output segmentation map of U-net**. Top image is the test image, while bottom left is the prediction image generated by the trained segmenter, with IARs signal shown as black pixels and background class shown as white pixels. Bottom right is the activation map associated with convolutional layer 24 (1x1 convolution in the last convolutional block) with sigmoid activation obtained using the keract python package. Since this activation function outputs values for the probability of belonging into either background or IARs class, the activation map values range between 0.0 (IARs class) and 1.0 (background class). Probability values below 0.2 are depicted by a dark purple colour, while yellow colour indicates probability values above 0.9. Test image belongs to 05/09/2012 (day/month/year).

Chapter 6

Experiments: HPC

In Chapter 5, U-net was clearly shown to be a valid method for automatic recognition of IARs signal in spectrogram images generated by BGS. In this Chapter, the state-of-the-art HPC methods and concepts outlined in Section 4.3, including CPU multithreading and GPU parallelization, are applied in order to speed up the execution of the computer programme that is responsible for training the U-net on the IARs dataset.

6.1 Initial code profiling

As a starting point, the script that was used to train the network on the 178 training examples without k-fold cross-validation and generate predictions for the 2135 days was timed in order to identify which parts of the code are responsible for the majority of runtime (Table 6.1). Generating predictions on test images turned out to be responsible for most of the runtime, followed by training the actual neural network. It is worth highlighting that the program is remarkably fast at generating predictions on test data, with a prediction on a single image taking only 0.21 seconds using CPUs.

Table 6.1: **Initial code profiling**. U-net was trained for 2 epochs and both runtime and percentage of runtime of each code section was recorded. Code is used to generate prediction for 2135 images on 2 Skylake CPUs for a total of 40 cores.

Code section	Runtime (seconds)	Percentage of total runtime
Python imports	4.0	0.68
Initialization	0.41	0.07
Training	133.23	22.43
Plotting	4.05	0.68
Generating predictions	452.29	76.14

The Python package imports and the plotting of training dynamics were not included in the timing for subsequent performance experiments. This is because these parts of the program do not exhibit potential to be improved by parallelism as they are very fast

already. Moreover, the part of the code that generates predictions on test data was also not included, as generating a single prediction is quite fast and the script is not expected to be used to predict such a large number of images on a regular basis anyway.

Training neural networks is known to be very computationally intensive and definitely exhibits some potential to be sped up by parallelism. The timing calls were thefore amended as to only include two parts, initialization and training, in all subsquent experiments unless otherwise stated. Initialization includes the creation of variables and model to be used during the training phase, while training includes the actual training phase. These parts were independently timed. Care was also taken as to manually delete the automatically generated pycache files (i.e. storing bytecode-compiled and optimized bytecode-compiled versions of files used by the program) and HDF5 files (i.e. storing the trained models) in between experiments, such that each experiment would entirely be independent and therefore give an accurate timing.

6.2 Tensorflow and Keras multithreading

U-net was trained for 2 epochs with different types and number of CPUs. Table 6.2 shows the timing results of the individual experiments. This test confirms that Tensorflow by default runs the code in parallel by leverging multiprocessing. Moroover, Skylake CPUs are clearly better than the Standard CPUs on Cirrus. The best time of 130.97 with native multiprocessing offered by Tensorflow was recorded using 4 Skylake CPUs spread over two nodes, for a total of 80 CPUs.

Table 6.2: **Native multiprocessing experiments conducted on different types of CPUs**. U-net was trained for 2 epochs using 2 CPUs of either Sylake or Standard type. Experiments on Skylake CPUs were conducted twice, while the experiments on Standard CPUs were conducted once. Experiments with 80 or 72 cores were run over two Cirrus nodes.

Type of CPU	# Cores	Initialization	Training
Skylake	10	0.43	168.50
Skylake	20	0.41	134.01
Skylake	30	0.41	176.79
Skylake	40	0.46	132.03
Skylake	80	0.46	130.97
Standard	36	0.65	320.53
Standard	72	0.45	314.14

The results obtained when the *use_multiprocessing* argument is set to *True* are not better than when the argument is set to *False* (Table 6.3). Additional experiments were performed with different values set for the number of workers (Table 6.4). A value of of 60 is found to be the best. That being said, the improvement in training time is marginal. One potential explanation to this is that as the *use_multiprocessing* and *workers* arguments only allow to tune the extent of parallelism exhibited by the Keras generator while creating batches of training data, this does not constitute distributed training in the sense that the several batches of input data are not all used for training at

the same. Given the small size of the dataset, the generation of batches of training data is not a bottleneck and this could explain why changing the values of these arguments only leads to marginal performance improvements.

Table 6.3: **Keras *use_multiprocessing* argument**. U-net was trained for 2 epochs using 2 Sylake CPUs. Keras has a *use_multiprocessing* argument inside the *fit_generator* method, which is by default set to *False*. The experiment with *use_multiprocessing* argument set to a different value was conducted once.

Type of CPU	# Cores	*use_multiprocessing*	Initialization	Training
Skylake	40	False	0.46	132.03
Skylake	40	True	0.41	133.34

Table 6.4: **Keras *workers* argument**. U-net was trained for 2 epochs using 2 Sylake CPUs. Keras has a *workers* argument inside *fit _generator*, which is by default set to 1. The *use_multiprocessing* argument was set to *False*. Experiments with *workers* argument set to a different value were conducted once.

Type of CPU	# Cores	*workers*	Initialization	Training
Skylake	40	1	0.46	132.03
Skylake	40	20	0.41	130.13
Skylake	40	60	0.41	129.55
Skylake	40	80	0.41	131.72

As a last test, Tensorflow's *intra_op_parallelism_threads* and *inter_op_parallelism_threads* configuration arguments were explored. Changing the value to which the *inter_op_parallelism_threads* argument is set was detrimental to performance (Table 6.6). The *inter_op_parallelism_threads* option essentially restricts how many different *Ops* can be launched in parallel. It appears that parallelizing different, non-blocking portions of the Tensorflow graph in a concurrent way likely introduces additional communication that constitutes an unnecessary overhead, especially with such a small batch size and training time for a single epoch on the IARs dataset. Using a value of 20 for *intra_op_parallelism_threads* argument, it was possible to further reduce the training time to 123.42 seconds (Table 6.5). This is currently the best time recorded with CPU multithreading.

Table 6.5: **Tensorflow *intra_op_parallelism_threads***. U-net was trained for 2 epochs using 2 Sylake CPUs. Tensorflow has a configuration option *intra_op_parallelism_threads*, which is by default set to the number of recognized logical CPU cores for a given architecture. Keras *use_multiprocessing* and *workers* arguments was set to their respective, default values. Experiments with configuration option *intra_op_parallelism_threads* set to a different value were conducted once.

Type of CPU	# Cores	intra_op_parallelism _threads	Initialization	Training
Skylake	40	N/A	0.46	132.03
Skylake	40	10	0.40	125.39
Skylake	40	20	0.40	123.43
Skylake	40	30	0.42	126.78
Skylake	40	40	0.49	131.48
Skylake	40	80	0.40	145.16

Table 6.6: **Tensorflow *inter_op_parallelism_threads***. U-net was trained for 2 epochs using 2 Sylake CPUs. Tensorflow has a configuration option *inter_op_parallelism_threads*, which is by default set to the number of recognized logical CPU cores for a given architecture. Keras *use_multiprocessing* and *workers* arguments was set to their respective, default values. Experiments with configuration option *intra_op_parallelism_threads* set to a different value were conducted once.

Type of CPU	# Cores	inter_op_parallelism _threads	Initialization	Training
Skylake	40	N/A	0.46	132.03
Skylake	40	10	0.49	>300
Skylake	40	20	0.48	>300
Skylake	40	30	0.40	292.15
Skylake	40	40	0.40	292.48
Skylake	40	80	0.40	288.57

6.3 GPU and multi-GPU parallelization

GPU parallelization was implemented while training the neural network for 2 or 10 epochs, with either 1, 2 or 4 GPUs within a single node. Each experiment was conducted 3 times and average and standard deviation for both initialization and training parts of the code are reported. As it can be seen from Fig. 6.1, although the initialization part of the code, which is responsible for replicating the model and transferring the data across GPUs, becomes sensibly larger when using 2 or 4 GPUs compared to a single GPU, the total execution time on both 2 and 10 epochs across all the experiments with different number of active GPUs is comparable. As parallelizing over multiple GPUs does not seem to give an additional benefit given the model and dataset size, parallelising over 1 GPU appears to be the most suitable option. Considering that using 1 GPU already allows to achieve a speedup of 4.45 times over the best option for CPU multithreading (Table 6.7), it seems that parallelizing the code with 1 GPU is the best option overall.

Table 6.7: **Calculating speed-up of GPU against shared memory parallelization**. U-net was independently trained for 10 epochs with either of the parallelization models. Execution time is the total time required for running the initialization and training sections of the code.

Parallelization model	Execution time	Speedup
40 Skylake CPU cores, *intra_op_parallelism_threads* set to 20	587.60	N/A
1 GPU	132.09	4.45

6.4 Final code profiling

The entire code was timed again when run on 1 GPU for 2 or 10 epochs (Table 6.8). When trained on 1 GPU for 2 epochs, training now makes up 16 % of total runtime. This constitutes a 6 % reduction in recorded runtime associated with training compared to the initial code profiling thanks to GPU parallelization. When training for 10 epochs, training and generating predictions have virtually equal contributions to runtime.

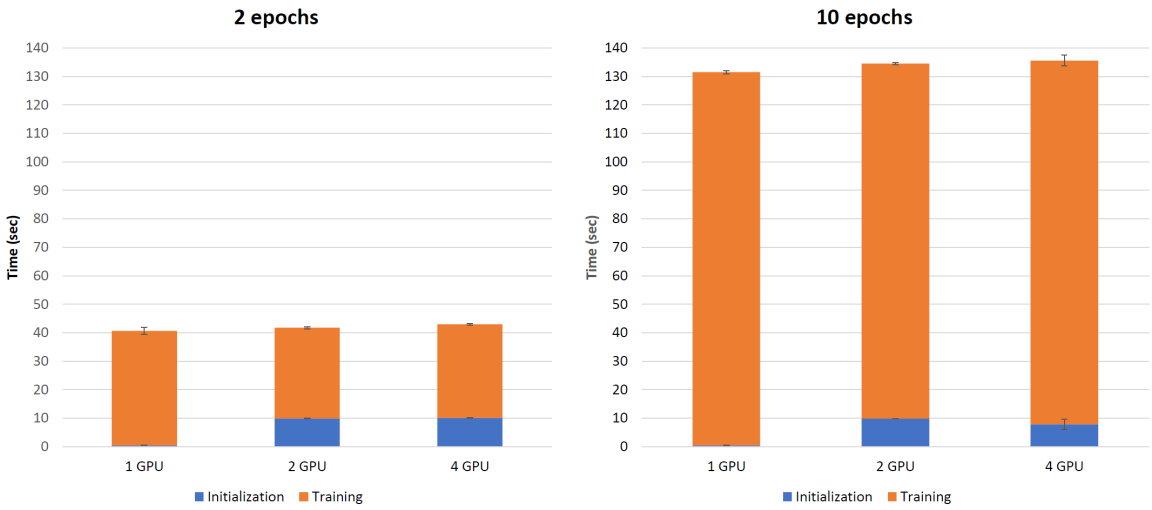

Figure 6.1: **Results of GPU parallelization experiments**. U-net was trained for 2 or 10 epochs using Keras's *multi_gpu_model* with 1, 2 or 4 GPUs. The experiments were conducted with 10 cores of a complementary Skylake CPU, as this is the minimum number of CPU cores needed for running a job on a GPU node on Cirrus. Bar plots and error bars display average and standard deviation, respectively, from 3 repeats of each experiment.

Table 6.8: **Code profiling when code is run on 1 GPU**. U-net was trained for 2 or 10 epochs on 1 GPU. Code is used to generate predictions for 2135 test images. Percentage of total runtime for each code section is reported.

Code section	% of total runtime on 2 epochs	% of total runtime on 10 epochs
Python Imports	1.55	3.55
Initialization	0.20	0.17
Training	15.96	43.58
Plotting	4.38	3.20
Generating predictions	77.90	49.51

GPU utilization was monitored using the *nvidia-smi* tool (Fig. 6.2). The clear pattern that can be seen is that for the multi GPU models, there is some imbalance in the extent of GPU utilization over the execution of the programme. In particular, in the 4 GPU model, 2 GPUs out of 4 are virtually idle. This may explain why it has not been possible to further reduce the runtime of the programme when switching from 2 to 4 GPUs. This difference cannot be attributed to potential differences in measured memory bandwidth across different GPUs, as the percentage of used memory across GPUs during program execution is virtually identical (see Fig. 8.4 and 8.5 in Appendix). One potential cause for the fact that spreading the work onto multiple GPUs does not lead to performance improvements is the small size of the training dataset. With such a small dataset, spreading the work on multiple GPUs likely incurs unnecessary overheads such as data splitting and concatenation of results that essentially nullify any potential speedup.

It is also worth mentioning that Keras is in charge of parallelising the code by spreading the work over the available GPUs and therefore no control is given to the programmer. A further investigation into an improved parallelization strategy that could better balance

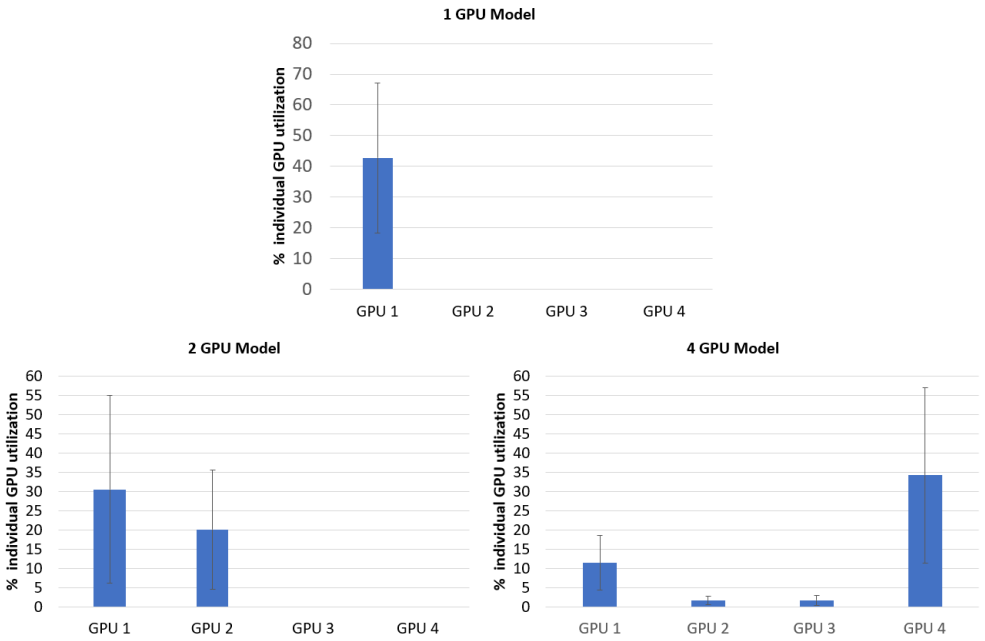

Figure 6.2: **Percentage of individual GPU utilization recorded every second and averaged over entire program execution with different Keras multi-gpu models** . U-net was trained for 2 epochs using different multi-gpu models. Bar plots and error bars display average and standard deviation, respectively. Average here indicates the approximate value of percentage of GPU utilization that is registered the most number of times during program execution. Standard deviation indicates how much the value of the percentage of GPU utilization varies during program execution.

GPU utilization when the network is trained on multiple GPUs is currently outside the scope of the dissertation. However, it could explored further in a separate, follow-up study.

Chapter 7

Discussion and Future Work

In this work, a ML algorithm was trained in order to generate a predictive model that would achieve pixel-level classification, namely segmentation, of a grayscale image depicting the IARs geomagnetic signal over both frequency and time. The objectives for this project, outlined in Section 4.1, were that the trained segmenter would have greater signal to noise ratio on a ‘good‘ day with IARs signal present compared to the original data analysis method developed by BGS (Objective 1) and also exhibit the ability to generate null predictions on a ’bad’ day with no IARs signal present (Objective 2).

As shown in Fig. 5.12, when the trained segmenter is presented with an image that contains neither IARs signal nor counfounding artificial signal, it essentially generates a null image (Objective 2), where virtually no pixels are classified as IARs signal.

It is worth noting that although a supervised machine learning approach was used in this project, the labels, in other words, the ground truth segmentation images, constitute by no means an accurate or noise free prediction of the real geomagnetic signal uncovered in the images. Since it is well-known that the quality of the predictions made by any trained classifier or segmenter is contingent on the goodness of the data that is used to train it with DL, it is understandable that the predictions on test data made by the segmenter contain some noise. Since prediction images are essentially made up of probability values between 0 and 1, thresholding such images seems to be a satisfactory way of obtaining a relatively noise-free version of the segmentation maps.

On the other hand, the predictions made by the original data analysis method developed by BGS simply contain values of 0 and 1 and therefore do not capture the level of uncertainty associated with each prediction at each pixel. Considering that the trained segmenter seems to also predict relatively less noise (see Fig. 5.11), from a qualitative point of view it could be argued that generating a segmenter using a trained ML algorithm such as the U-net neural network along with thresholding the resulting prediction images does yield an overall better performance than the original data analysis method developed by BGS (Objective 1). One way to substantiate this statement quantitatively would be to manually label the signal that is considered to be noise in the ground truth image, then generate a count for the number of pixels that

fall into the 'noise' category and, following that, calculate a percentage reduction in noise detection by the segmenter by taking the ratio between this count and the number of white pixels in the overlay image of thresholded prediction on top of ground truth image shown in Fig. 5.11.

One potential way to reduce the amount of noise detected by the segmenter that may obviate the need of thresholding the predictions is to modify the binary cross-entropy loss function used so far such that larger weight is given to predicted lines of longer length. An additional, alternative approach is to use an ensemble or recursive method, where U-net is first trained and the resulting predictive model is used in order to generate predictions, followed by repurposing these predictions as ground truth images as part of an additional training phase. Since the ground truth images used in the latter phase would exhibit less noise, this would theoretically result in a superior segmenter. Some approaches for ensemble learning applied in combination with random forest, particle swarm optimization and aggressive up-convolution have shown reasonable success in the segmentation task for a variety of complex datasets [56, 88, 77].

A sample of the prediction images generated by the trained segmenter on the test set of 2135 examples from the IARs dataset has been visually inspected by Dr. Beggan, who gave a positive assessment as to the predictive performance of the deployed model for recognizing IARs signal (see 8.1 and 8.2 in Appendix). Although there is some room for improvement as detailed in this Section, since the current predictive model is generated using the U-net ML algorithm, it still constitutes a state-of-the-art implementation for achieving semantic segmentation on this dataset.

Chapter 8

Conclusion

Deep learning has been the driving force for many great strides and recent breakthroughs in the field of computer vision. In this work, a novel application of the widely-known fully convolutional neural network U-net for the automated pattern recognition within the context of experimental, geomagnetic data analysis has been described.

During the project, the four tasks were successfully carried out and the two project objectives were achieved. In particular, strong evidence has been generated indicating that maximum performance was reached while training the U-net on the IARs dataset (Task 1). Moreover, a threshold value of 0.4 and 0.5 appeared to be optimal for filtering the prediction images and in turn remove a large fraction of the noise present in the prediction images generated by the predictive model (Task 2). Furthermore, the predictive model passed the negative control test with a 'bad' day test example, suggesting that is it selective for 'good' days as test images and therefore real IARs signal (Task 3). Finally, the predictive capability of the model has been vetted by the industrial collaborator by examining the predictions of the model made on unseen test data (Task 4). Following that, the trained neural network is deemeed to be currently fit for purpose and can be deployed on BGS's computer cluster.

From a qualitative point of view, the predictive model generated by training U-net performs better than the existing data analysis method developed by the industrial collaborator of the project as it predicts relatively less noise. Moreover, the neural network is fast: it takes as little as $\approx$132 seconds to train U-net for 10 epochs on a dataset of 178 training examples on a GPU; and the resulting predictive model takes $\approx$58 milliseconds to generate a prediction image for a single test image. Some ideas for further reducing the amount of noise detected by the predictive model, including a modified loss function and ensemble methods, have been suggested. The approach and methods that have been used in this project in order to benchmark the U-net on the IARs dataset are definitely transferable and applicable for the automatic detection of additional geomagnetic phenomena with similar spectral patterns, such as Schumann resonances.

Appendix

All the commands used for installing the required software and for training U-net are listed at the following Github pages:

https://github.com/marangiop/unet/blob/master/docs/setup-cirrus.md

https://github.com/marangiop/unet/blob/master/docs/training-testing-unet.md

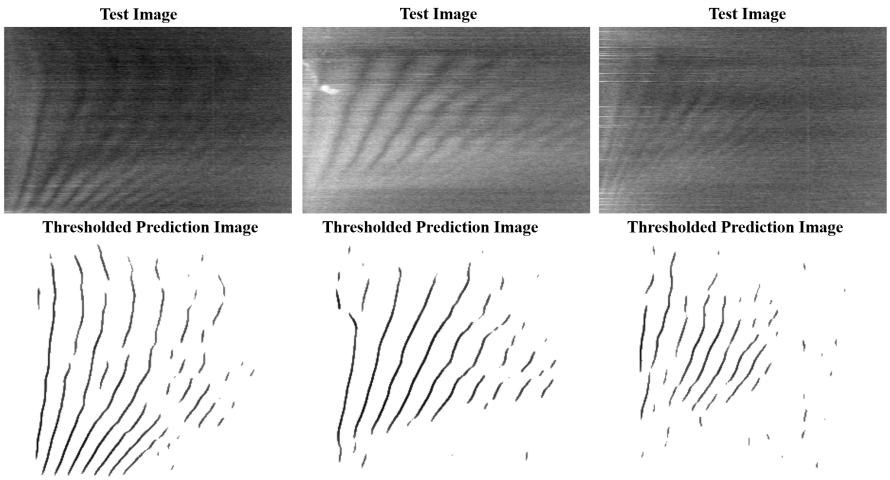

Figure 8.1: **Thresholded predictions on additional test images - part 2**. U-net was trained for 10 epochs on the default hyperparameter values. Top image is test image, while bottom image is prediction image after a threshold of 0.5 has been applied. These test images do not constitute unseen data for the network as they are part of the the dataset that was used to train the network. From left to right, these test images belong to 06/03/2016, 31/08/2015 and 26/08/2014 (day/month/year), respectively.

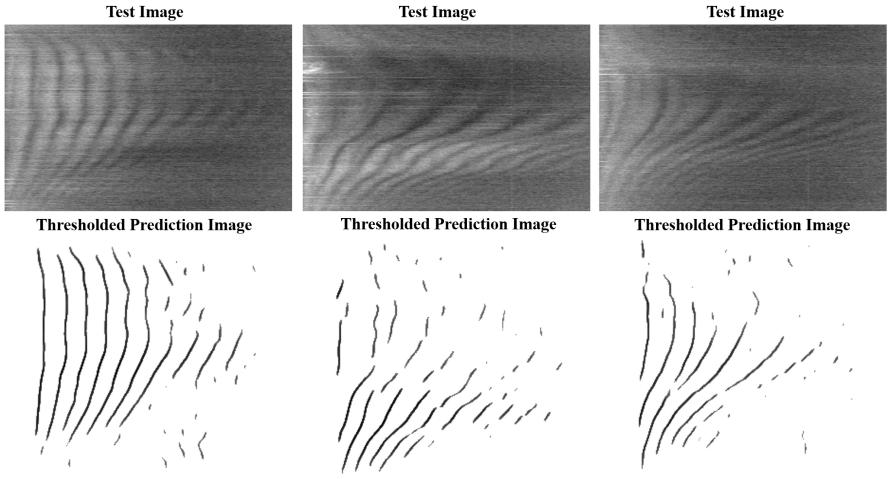

Figure 8.2: **Thresholded predictions on additional test images - part 1**. U-net was trained for 10 epochs on the default hyperparameter values. Top image is test image, while bottom image is prediction image after a threshold of 0.5 has been applied. These test images do not constitute unseen data for the network as they are part of the the dataset that was used to train the network. From left to right, these test images belong to 24/09/2012, 12/08/2017 and 08/06/2018 (day/month/year), respectively.

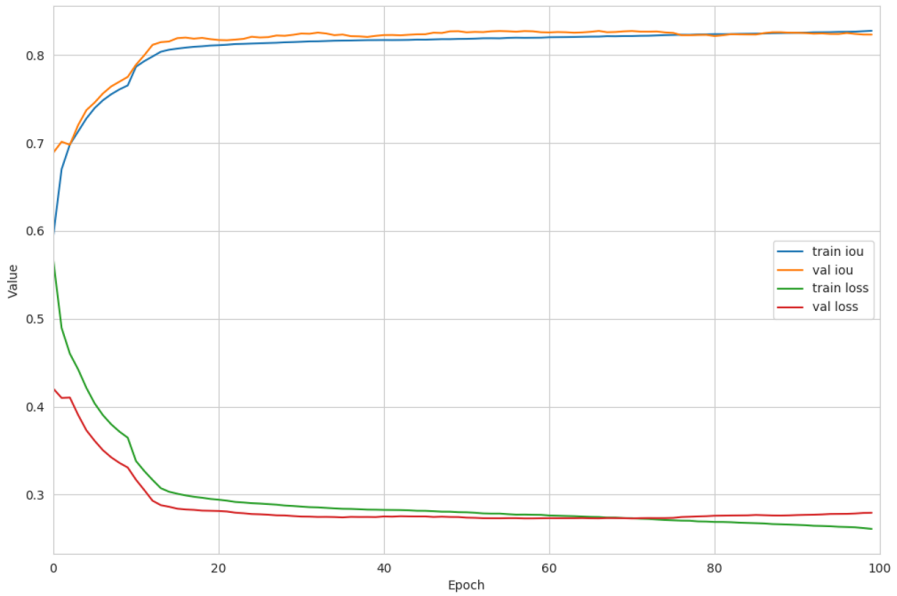

Figure 8.3: **Overfitting test on IARs dataset - fold 0**. U-net implementation proposed in [86] with additional code developed in the dissertation for k-fold cross-validation was trained for 100 epochs on the IARs dataset, with k set to 2. Training loss, validation loss, training IoU and validation IoU for fold 0 were monitored.

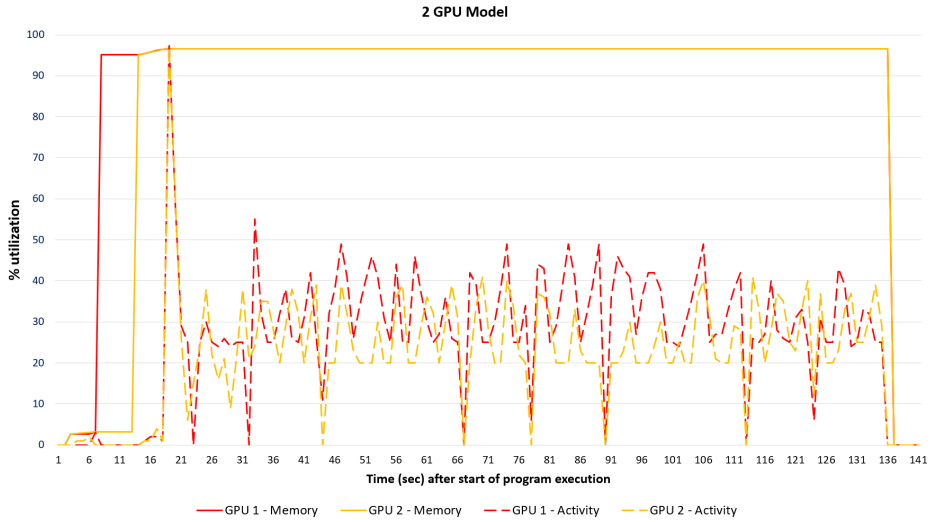

Figure 8.4: **Percentage of individual GPU activity and memory utilization recorded every second with 2 GPU model**. U-net was trained for 10 epochs on the default hyperparameter values. *Nvidia-smi* command was used in in order to query the 2 GPUs for their respective percentage of gpu and memory utilization. GPU utilization is defined as percent of time over the past second during which one or more kernels was executing on the GPU. Memory utilization is defined as amount of memory allocated by active contexts as percentage of total installed memory on the individual GPU. GPU and memory utilization were measured for every second of program execution, however, tick marks for time after start of program execution have a 5 seconds interval.

Figure 8.5: **Percentage of individual GPU activity and memory utilization recorded every second with 4 GPU model**. U-net was trained for 10 epochs on the default hyperparameter values. *Nvidia-smi* command was used in in order to query the 4 GPUs for their respective percentage of gpu and memory utilization. GPU utilization is defined as percent of time over the past second during which one or more kernels was executing on the GPU. Memory utilization is defined as amount of memory allocated by active contexts as percentage of total installed memory on the individual GPU. GPU and memory utilization were measured for every second of program execution, however, tick marks for time after start of program execution have a 5 seconds interval.

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mane, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viegas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *arXiv:1603.04467 [cs]*, March 2016. URL http://arxiv.org/abs/1603.04467. arXiv: 1603.04467.

[2] H. Alfvén. Existence of Electromagnetic-Hydrodynamic Waves. *Nature*, 150 (3805):405–406, October 1942. ISSN 1476-4687. doi: 10.1038/150405d0. URL https://www.nature.com/articles/150405d0.

[3] Vijay Badrinarayanan, Alex Kendall, and Roberto Cipolla. SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation. *arXiv:1511.00561 [cs]*, November 2015. URL http://arxiv.org/abs/1511.00561. arXiv: 1511.00561.

[4] C. D. Beggan. Automatic detection of ionospheric Alfven resonances using signal and image processing techniques. *Annales Geophysicae*, 32(8):951–958, August 2014. ISSN 1432-0576. doi: 10.5194/angeo-32-951-2014. URL https://www.ann-geophys.net/32/951/2014/.

[5] C. D. Beggan and M. Musur. Observation of Ionospheric Alfven Resonances at 1-30 Hz and Their Superposition With the Schumann Resonances. *Journal of Geophysical Research: Space Physics*, 123(5):4202–4214, May 2018. ISSN 21699380. doi: 10.1029/2018JA025264. URL http://doi.wiley.com/10.1029/2018JA025264.

[6] Yoshua Bengio. Practical recommendations for gradient-based training of deep architectures. *arXiv:1206.5533 [cs]*, June 2012. URL http://arxiv.org/abs/1206.5533. arXiv: 1206.5533.

[7] Yoshua Bengio and Yves Grandvalet. No Unbiased Estimator of the Variance of K-Fold Cross-Validation. *The Journal of Machine Learning Research*, 5:

1089–1105, December 2004. ISSN 1532-4435. URL http://dl.acm.org/citation.cfm?id=1005332.1044695.

[8] James Bergstra and Yoshua Bengio. Random Search for Hyper-Parameter Optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012. ISSN ISSN 1533-7928. URL http://www.jmlr.org/papers/v13/bergstra12a.html.

[9] James S. Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for Hyper-Parameter Optimization. In J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 24*, pages 2546–2554. Curran Associates, Inc., 2011. URL http://papers.nips.cc/paper/4443-algorithms-for-hyper-parameter-optimization.pdf.

[10] B. P. Besser. Synopsis of the historical development of Schumann resonances. *Radio Science*, 42(2), 2007. ISSN 1944-799X. doi: 10.1029/2006RS003495. URL https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1029/2006RS003495.

[11] BGS. High-frequency Magnetometers, February 2019. URL http://www.geomag.bgs.ac.uk/research/inductioncoils.html.

[12] Christopher Bishop. *Pattern Recognition and Machine Learning*. Information Science and Statistics. Springer-Verlag, New York, 2006. ISBN 978-0-387-31073-2. URL https://www.springer.com/gp/book/9780387310732.

[13] Jason Brownlee. Deep Learning & Artificial Neural Networks, August 2016. URL https://machinelearningmastery.com/what-is-deep-learning/.

[14] Jason Brownlee. A Gentle Introduction to Convolutional Layers for Deep Learning Neural Networks, April 2019. URL https://machinelearningmastery.com/convolutional-layers-for-deep-learning-neural-networks/.

[15] Nitesh V. Chawla. Data Mining for Imbalanced Datasets: An Overview. In Oded Maimon and Lior Rokach, editors, *Data Mining and Knowledge Discovery Handbook*, pages 853–867. Springer US, Boston, MA, 2005. ISBN 978-0-387-25465-4. doi: 10.1007/0-387-25465-X_40. URL https://doi.org/10.1007/0-387-25465-X_40.

[16] François Chollet and others. *Keras*. 2015. URL https://keras.io.

[17] Marc Claesen and Bart De Moor. Hyperparameter Search in Machine Learning. *arXiv:1502.02127 [cs, stat]*, February 2015. URL http://arxiv.org/abs/1502.02127. arXiv: 1502.02127.

[18] Daphne Cornelisse. An intuitive guide to Convolutional Neural Networks, April 2018. URL https://www.freecodecamp.org/news/an-intuitive-guide-to-convolutional-neural-networks-260c2de0a050/.

[19] J. Deng, W. Dong, R. Socher, L. Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer*

Vision and Pattern Recognition, pages 248–255, June 2009. doi: 10.1109/CVPR.2009.5206848.

[20] EPCC. Cirrus Hardware, 2019. URL https://www.cirrus.ac.uk/about/hardware.html.

[21] ESA. Earth's protective shield, 2014. URL http://www.esa.int/spaceinimages/Images/2014/02/Earth_s_protective_shield.

[22] Kunihiko Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36(4):193–202, April 1980. ISSN 1432-0770. doi: 10.1007/BF00344251. URL https://doi.org/10.1007/BF00344251.

[23] Brian Fulkerson, Andrea Vedaldi, and Stefano Soatto. Class segmentation and object localization with superpixel neighborhoods. In *2009 IEEE 12th International Conference on Computer Vision*, pages 670–677, Kyoto, September 2009. IEEE. ISBN 978-1-4244-4420-5. doi: 10.1109/ICCV.2009.5459175. URL http://ieeexplore.ieee.org/document/5459175/.

[24] Geekboots. Multiprocessing vs Multithreading | Geekboots Story, 2019. URL https://www.geekboots.com/story/multiprocessing-vs-multithreading.

[25] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pages 249–256, March 2010. URL http://proceedings.mlr.press/v9/glorot10a.html.

[26] GoogleDevelopers. ML Practicum: Image Classification | Machine Learning Practica, 2019. URL https://developers.google.com/machine-learning/practica/image-classification/convolutional-neural-networks.

[27] Jun Han and Claudio Moraga. The influence of the sigmoid function parameters on the speed of backpropagation learning. In José Mira and Francisco Sandoval, editors, *From Natural to Artificial Neural Computation*, Lecture Notes in Computer Science, pages 195–201. Springer Berlin Heidelberg, 1995. ISBN 978-3-540-49288-7.

[28] S. R. Hebden, T. R. Robinson, D. M. Wright, T. Yeoman, T. Raita, and T. Bosinger. A quantitative analysis of the diurnal evolution of Ionospheric Alfven resonator magnetic resonance features and calculation of changing IAR parameters. *Annales Geophysicae*, 23(5):1711–1721, July 2005. ISSN 1432-0576. doi: 10.5194/angeo-23-1711-2005. URL http://www.ann-geophys.net/23/1711/2005/.

[29] Geoffrey E. Hinton. A Practical Guide to Training Restricted Boltzmann Machines. In Grégoire Montavon, Geneviève B. Orr, and Klaus-Robert Müller, editors, *Neural Networks: Tricks of the Trade*, volume 7700, pages 599–619. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-35288-1 978-3-642-35289-8. doi: 10.1007/978-3-642-35289-8_32. URL http://link.springer.com/10.1007/978-3-642-35289-8_32.

[30] Mohammad Hossin and Sulaiman M.N. A Review on Evaluation Metrics for Data Classification Evaluations. *International Journal of Data Mining & Knowledge Management Process*, 5:01–11, March 2015. doi: 10.5121/ijdkp.2015.5201.

[31] D. H. Hubel and T. N. Wiesel. Receptive fields and functional architecture of monkey striate cortex. *The Journal of Physiology*, 195(1):215–243, March 1968. ISSN 0022-3751. URL https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1557912/.

[32] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani, editors. *An introduction to statistical learning: with applications in R*. Number 103 in Springer texts in statistics. Springer, New York, 2013. ISBN 978-1-4614-7137-0. OCLC: ocn828488009.

[33] Katarzyna Janocha and Wojciech Marian Czarnecki. On Loss Functions for Deep Neural Networks in Classification. *arXiv:1702.05659 [cs]*, February 2017. URL http://arxiv.org/abs/1702.05659. arXiv: 1702.05659.

[34] Manikandan Jeeva. The Scuffle Between Two Algorithms - Neural Network vs. Support Vector Machine, October 2018. URL https://medium.com/analytics-vidhya/the-scuffle-between-two-algorithms-neural-network-vs-support-vector-machine-16abe0eb4181.

[35] Jeremy Jordan. An overview of semantic image segmentation., May 2018. URL https://www.jeremyjordan.me/semantic-segmentation/.

[36] Andrej Karpathy. A Peek at Trends in Machine Learning, April 2017. URL https://medium.com/@karpathy/a-peek-at-trends-in-machine-learning-ab8a1085a106.

[37] Andrej Karpathy. CS231n Convolutional Neural Networks for Visual Recognition, 2019. URL http://cs231n.github.io/convolutional-networks/.

[38] Ayoosh Kathuria. Intro to optimization in deep learning: Momentum, RMSProp and Adam, June 2018. URL https://blog.paperspace.com/intro-to-optimization-momentum-rmsprop-adam/.

[39] Keras. Initializers - Keras Documentation, 2019. URL https://keras.io/initializers/.

[40] Keras. Optimizers - Keras Documentation, 2019. URL https://keras.io/optimizers/.

[41] J. Kiefer and J. Wolfowitz. Stochastic Estimation of the Maximum of a Regression Function. *The Annals of Mathematical Statistics*, 23(3):462–466, September 1952. ISSN 0003-4851, 2168-8990. doi: 10.1214/aoms/1177729392. URL https://projecteuclid.org/euclid.aoms/1177729392.

[42] Ron Kohavi. A Study of Cross-validation and Bootstrap for Accuracy Estimation and Model Selection. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'95, pages 1137–1143, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc. ISBN 978-1-55860-363-9.

URL http://dl.acm.org/citation.cfm?id=1643031.1643047. event-place: Montreal, Quebec, Canada.

[43] N. Kumari and S. Saxena. Review of Brain Tumor Segmentation and Classification. In *2018 International Conference on Current Trends towards Converging Technologies (ICCTCT)*, pages 1–6, March 2018. doi: 10.1109/ICCTCT.2018.8551004.

[44] Harshall Lamba. Understanding Semantic Segmentation with UNET, February 2019. URL https://towardsdatascience.com/understanding-semantic-segmentation-with-unet-6be4f42d4b47.

[45] Hugo Larochelle, Dumitru Erhan, Aaron Courville, James Bergstra, and Yoshua Bengio. An empirical evaluation of deep architectures on problems with many factors of variation. In *Proceedings of the 24th international conference on Machine learning - ICML '07*, pages 473–480, Corvalis, Oregon, 2007. ACM Press. ISBN 978-1-59593-793-3. doi: 10.1145/1273496.1273556. URL http://portal.acm.org/citation.cfm?doid=1273496.1273556.

[46] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998. ISSN 0018-9219. doi: 10.1109/5.726791.

[47] Yann LeCun, Patrick Haffner, Léon Bottou, and Yoshua Bengio. Object Recognition with Gradient-Based Learning. In *Shape, Contour and Grouping in Computer Vision*, pages 319–, London, UK, UK, 1999. Springer-Verlag. ISBN 978-3-540-66722-3. URL http://dl.acm.org/citation.cfm?id=646469.691875.

[48] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, May 2015. ISSN 0028-0836, 1476-4687. doi: 10.1038/nature14539. URL http://www.nature.com/articles/nature14539.

[49] Michael Levandowsky and David Winter. Distance between Sets. *Nature*, 234 (5323):34–35, November 1971. ISSN 1476-4687. doi: 10.1038/234034a0. URL https://www.nature.com/articles/234034a0.

[50] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft COCO: Common Objects in Context. In David Fleet, Tomas Pajdla, Bernt Schiele, and Tinne Tuytelaars, editors, *Computer Vision ECCV 2014*, Lecture Notes in Computer Science, pages 740–755. Springer International Publishing, 2014. ISBN 978-3-319-10602-1.

[51] Ethen Liu. model_selection, 2019. URL http://ethen8181.github.io/machine-learning/model_selection/model_selection.html?source=post_page----------.

[52] Ivan Lizarazo. SVM-based segmentation and classification of remotely sensed data. *International Journal of Remote Sensing - INT J REMOTE SENS*, 29: 7277–7283, December 2008. doi: 10.1080/01431160802326081.

[53] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3431–3440, 2015.

[54] Wenjie Luo, Yujia Li, Raquel Urtasun, and Richard Zemel. Understanding the Effective Receptive Field in Deep Convolutional Neural Networks. *arXiv:1701.04128 [cs]*, January 2017. URL http://arxiv.org/abs/1701.04128. arXiv: 1701.04128.

[55] Paolo Marangio. unet for image segmentation., March 2019. URL https://github.com/marangiop/unet. original-date: 2019-03-09T10:54:02Z.

[56] Dimitrios Marmanis, Jan D. Wegner, Silvano Galliani, Konrad Schindler, Mihai Datcu, and Uwe Stilla. Semantic segmentation of earierl images with an ensemble of CNNs. 2016. doi: 10.5194/isprsannals-III-3-473-2016.

[57] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, December 1943. ISSN 1522-9602. doi: 10.1007/BF02478259. URL https://doi.org/10.1007/BF02478259.

[58] David M Miles. Advances in Fluxgate Magnetometry for Space Physics. *University of Alberta Libraries*, 2017. doi: 10.7939/r30k26p98. URL https://era.library.ualberta.ca/items/6e0869a4-d2e4-4795-9498-02226bfc20f9.

[59] Vinod Nair and Geoffrey E. Hinton. Rectified Linear Units Improve Restricted Boltzmann Machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, pages 807–814, USA, 2010. Omnipress. ISBN 978-1-60558-907-7. URL http://dl.acm.org/citation.cfm?id=3104322.3104425. event-place: Haifa, Israel.

[60] Minh Hoai Nguyen and Fernando de la Torre. Optimal feature selection for support vector machines. *Pattern Recognition*, 43(3):584–591, March 2010. ISSN 0031-3203. doi: 10.1016/j.patcog.2009.09.003. URL http://www.sciencedirect.com/science/article/pii/S0031320309003409.

[61] Hyeonwoo Noh, Seunghoon Hong, and Bohyung Han. Learning Deconvolution Network for Semantic Segmentation. *arXiv:1505.04366 [cs]*, May 2015. URL http://arxiv.org/abs/1505.04366. arXiv: 1505.04366.

[62] M. Nose, M. Uyeshima, J. Kawai, and H. Hase. Ionospheric Alfven resonator observed at low latitude ground station, Muroto. *Journal of Geophysical Research: Space Physics*, 122(7):7240–7255, July 2017. ISSN 2169-9402. doi: 10.1002/2017JA024204. URL https://agupubs-onlinelibrary-wiley-com.ezproxy.is.ed.ac.uk/doi/abs/10.1002/2017JA024204.

[63] NVIDIA. Useful nvidia-smi Queries | NVIDIA, 2017. URL https://nvidia.custhelp.com/app/answers/detail/a_id/3751/~/useful-nvidia-smi-queries.

[64] Chigozie Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall.

Activation Functions: Comparison of trends in Practice and Research for Deep Learning. *arXiv:1811.03378 [cs]*, November 2018. URL http://arxiv.org/abs/1811.03378. arXiv: 1811.03378.

[65] Sagar Patel. Data Science essentials: Why train-validation-test data?, September 2018. URL https://medium.com/datadriveninvestor/data-science-essentials-why-train-validation-test-data-b7f7d472dc1f.

[66] Josh Patterson and Adam Gibson. *Deep Learning: A Practitioner's Approach*. O'Reilly Media, Inc., 1st edition, 2017. ISBN 1-4919-1425-4 978-1-4919-1425-0.

[67] A.S. Potapov, T.N. Polyushkina, B.V. Dovbnya, B. Tsegmed, and R.A. Rakhmatulin. Emissions of ionospheric Alfven resonator and ionospheric conditions. *Journal of Atmospheric and Solar-Terrestrial Physics*, 119:91–101, November 2014. ISSN 13646826. doi: 10.1016/j.jastp.2014.07.001. URL https://linkinghub.elsevier.com/retrieve/pii/S1364682614001643.

[68] Philippe Remy. Activation Maps (Layers Outputs) and Gradients in Keras.: philipperemy/keract, July 2019. URL https://github.com/philipperemy/keract. original-date: 2017-05-17T04:50:57Z.

[69] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-Net: Convolutional Networks for Biomedical Image Segmentation. *arXiv:1505.04597 [cs]*, May 2015. URL http://arxiv.org/abs/1505.04597. arXiv: 1505.04597.

[70] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-Net: Convolutional Networks for Biomedical Image Segmentation, 2019. URL https://lmb.informatik.uni-freiburg.de/people/ronneber/u-net/.

[71] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533, October 1986. ISSN 1476-4687. doi: 10.1038/323533a0. URL https://www.nature.com/articles/323533a0.

[72] W. O. Schumann. Uber die strahlungslosen Eigenschwingungen einer leitenden Kugel, die von einer Luftschicht und einer Ionospharenhulle umgeben ist. *Zeitschrift Naturforschung Teil A*, 7:149–154, February 1952. ISSN 0932-0784. doi: 10.1515/zna-1952-0202. URL http://adsabs.harvard.edu/abs/1952ZNatA...7..149S.

[73] Jamie Shotton, John Winn, Carsten Rother, and Antonio Criminisi. TextonBoost for Image Understanding: Multi-Class Object Recognition and Segmentation by Jointly Modeling Texture, Layout, and Context. *International Journal of Computer Vision*, 81(1):2–23, January 2009. ISSN 1573-1405. doi: 10.1007/s11263-007-0109-1. URL https://doi.org/10.1007/s11263-007-0109-1.

[74] Piotr Skalski. Deep Dive into Math Behind Deep Networks, August 2018. URL https://towardsdatascience.com/https-medium-com-piotr-skalski92-deep-dive-into-deep-networks-math-17660bc376ba.

[75] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014. URL http://jmlr.org/papers/v15/srivastava14a.html.

[76] Sylabs. Documentation & Examples, 2019. URL https://sylabs.io/docs/.

[77] T. Y. Tan, L. Zhang, C. P. Lim, B. Fielding, Y. Yu, and E. Anderson. Evolving Ensemble Models for Image Segmentation Using Enhanced Particle Swarm Optimization. *IEEE Access*, 7:34004–34019, 2019. ISSN 2169-3536. doi: 10.1109/ACCESS.2019.2903015.

[78] Tensorflow. Performance, 2019. URL https://www.tensorflow.org/guide/performance/overview.

[79] Tensorflow. TensorFlow examples. Frequently Asked Questions, August 2019. URL https://github.com/tensorflow/examples/blob/master/community/en/docs/faq.md. original-date: 2018-07-16T22:11:56Z.

[80] Floris van Beers, Arvid Lindström, Emmanuel Okafor, and Marco Wiering. Deep Neural Networks with Intersection over Union Loss for Binary Image Segmentation:. In *Proceedings of the 8th International Conference on Pattern Recognition Applications and Methods*, pages 438–445, Prague, Czech Republic, 2019. SCITEPRESS - Science and Technology Publications. ISBN 978-989-758-351-3. doi: 10.5220/0007347504380445. URL http://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0007347504380445.

[81] Athanasios Voulodimos, Nikolaos Doulamis, Anastasios Doulamis, and Eftychios Protopapadakis. Deep Learning for Computer Vision: A Brief Review. *Computational Intelligence and Neuroscience*, 2018:7068349, 2018. ISSN 1687-5273. doi: 10.1155/2018/7068349.

[82] X. Wang, S. Wang, Y. Zhu, and X. Meng. Image segmentation based on Support Vector Machine. In *Proceedings of 2012 2nd International Conference on Computer Science and Network Technology*, pages 202–206, December 2012. doi: 10.1109/ICCSNT.2012.6525921.

[83] Zhiwen Yu, Hau-San Wong, and Guihua Wen. A modified support vector machine and its application to image segmentation. *Image and Vision Computing*, 29(1): 29–40, January 2011. ISSN 0262-8856. doi: 10.1016/j.imavis.2010.08.003. URL http://www.sciencedirect.com/science/article/pii/S0262885610001113.

[84] Karol Zak. Helper package with multiple U-Net implementations in Keras as well as useful utility tools helpful when working with image semantic segmentation tasks. This library and underlying tools come from .., July 2019. URL https://github.com/karolzak/keras-unet. original-date: 2019-03-13T07:50:21Z.

[85] Yongli Zhang and Yuhong Yang. Cross-validation for selecting a model selection procedure. *Journal of Econometrics*, 187(1):95–112, July 2015. ISSN 03044076.

doi: 10.1016/j.jeconom.2015.02.006. URL https://linkinghub.elsevier.com/retrieve/pii/S0304407615000305.

[86] Hao Zhixu. unet for image segmentation., July 2019. URL https://github.com/zhixuhao/unet. original-date: 2017-04-06T01:58:15Z.

[87] Kelly H. Zou, Simon K. Warfield, Aditya Bharatha, Clare M.C. Tempany, Michael R. Kaus, Steven J. Haker, William M. Wells, Ferenc A. Jolesz, and Ron Kikinis. Statistical Validation of Image Segmentation Quality Based on a Spatial Overlap Index. *Academic radiology*, 11(2):178–189, February 2004. ISSN 1076-6332. doi: 10.1016/S1076-6332(03)00671-8. URL https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1415224/.

[88] Yan Zuo and Tom Drummond. Fast Residual Forests: Rapid Ensemble Learning for Semantic Segmentation. In *Conference on Robot Learning*, pages 27–36, October 2017. URL http://proceedings.mlr.press/v78/zuo17a.html.