Adrián Alcolea Moreno

Three steps towards reliable and efficient systems for machine learning algorithms

Director/es
Resano Ezcaray, Jesús Javier

ISSN 2254-7606

ISSN 2254-7606

Tesis Doctoral

THREE STEPS TOWARDS RELIABLE AND EFFICIENT SYSTEMS FOR MACHINE LEARNING ALGORITHMS

Autor

Adrián Alcolea Moreno

Director/es

Resano Ezcaray, Jesús Javier

UNIVERSIDAD DE ZARAGOZA
Escuela de Doctorado

Programa de Doctorado en Ingeniería de Sistemas e Informática

2022

Three Steps Towards Reliable and Efficient Systems for Machine Learning Algorithms

Adrián Alcolea

September 19, 2022

University of Zaragoza (UNIZAR)

Computer Science and Systems Engineering Department (DIIS)
Engineering Research Institute of Aragon (I3A)
Computer Architecture Group of Zaragoza (GaZ)

PhD thesis

Three Steps Towards Reliable and Efficient Systems for Machine Learning Algorithms

Adrián Alcolea

Supervisor Javier Resano
Computer Architecture Group of Zaragoza (GaZ)
Engineering Research Institute of Aragon (I3A)
Computer Science and Systems Engineering Department
University of Zaragoza

September 19, 2022

Adrián Alcolea
Three Steps Towards Reliable and Efficient Systems for Machine Learning Algorithms
PhD thesis, September 19, 2022
Supervisor: Javier Resano

University of Zaragoza (UNIZAR)
Engineering Research Institute of Aragon (I3A)
Computer Architecture Group of Zaragoza (GaZ)
Computer Science and Systems Engineering Department (DIIS)
C/ María de Luna, nº 1
50018, Zaragoza

Abstract

This thesis compiles the work realised during four years of research in machine learning algorithms and techniques. The aim of this study lays in how to execute the inference process of machine learning algorithms in a more efficient way and how to achieve more reliable techniques. For that, this thesis consist in three main steps: first, the design and implementation on an FPGA of an accelerator that takes advantage of the optimisation opportunities offered by sparsity in DNNs, second, the design and implementation on an FPGA of an accelerator for gradient boosting decision trees in the context of hyperspectral images classification, and third, an analysis of bayesian networks for hyperspectral images classification to demonstrate how the uncertainty metrics can help us in many tasks to achieve more reliable networks.

Resumen (spanish)

Esta tesis recoge el trabajo realizado durante cuatro años de investigación en algoritmos y técnicas de aprendizaje automático. El objetivo de este estudio radica en cómo ejecutar el proceso de inferencia de los algoritmos de aprendizaje automático de una manera más eficiente y cómo conseguir técnicas más fiables. Para ello, esta tesis consta de tres pasos principales: en primer lugar, el diseño e implementación en una FPGA de un acelerador que aprovecha las oportunidades de optimización que ofrece la dispersión de datos en las DNNs, en segundo lugar, el diseño e implementación en una FPGA de un acelerador para árboles de decisión basados en la potenciación del gradiente en el contexto de la clasificación de imágenes hiperespectrales, y en tercer lugar, un análisis de redes bayesianas para la clasificación de imágenes hiperespectrales para demostrar cómo las métricas de incertidumbre pueden ayudarnos en muchas tareas para conseguir redes más fiables.

Acknowledgement

I want to thank to my parents and family for being always supporting. To the *equipo* and *rebelason*, because they had to put up with me all these years XP. To the *GazBees* for being there side by side all along this work. To the *HyperComp* for helping us with the complex world of hyperspectral images. And, of course, to my supervisor, because I can safely say that I had the best supervisor ever =).

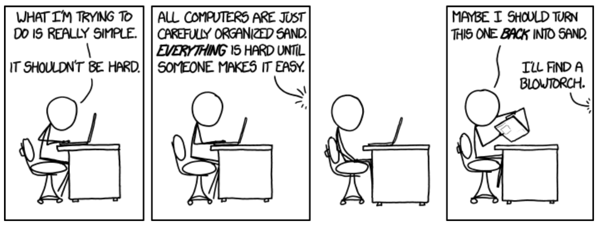
Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction


WHAT I'M TRYING TO DO IS REALLY SIMPLE.
IT SHOULDN'T BE HARD.
ALL COMPUTERS ARE JUST CAREFULLY ORGANIZED SAND. EVERYTHING IS HARD UNTIL SOMEONE MAKES IT EASY.
MAYBE I SHOULD TURN THIS ONE BACK INTO SAND.
I'LL FIND A BLOWTORCH.

— **xkcd.com**
(Just that guy, you know?)

This thesis compiles the work realised during four years of research in machine learning algorithms and techniques. The aim of this study lays in two main aspects: how to execute the inference process of machine learning algorithms in a more efficient way, that is faster and/or with less energy consumption, to meet embedded systems needs; and how to achieve more reliable techniques, i.e. not only accurate but "trustworthy" in the sense of being able to detect when the received input does not correspond to the learned information, so it is not possible to achieve a correct prediction.

Enjoy it.

1.1 Motivation and Contributions

Machine Learning (ML) techniques have made spectacular advances in recent years. The improvements and refinement of the models used, together with hardware platforms that allow the exploration of increasingly complex models, trained on larger data sets, have enabled a vast improvement in the accuracy of the models, and opened up new fields of application.

However, the size and computation requirements of these models are increasing too. The training process of these complex models is usually made with GPUs, which are capable to massively parallelise the required computation, significantly reducing the training time. In many fields this is enough, and the inference process is also carried out withing a big computational system. But in the context of embedded systems, once trained the model, the inference process is going to be executed in a small low-power device, where the size and the energy and computational requirements are critical.

In order to analyse these aspects we selected as test application the analysis of Hyperspectral Images (HI) in remote sensing, because it is a field where this challenges are very present for two reasons. The images could be very big due to the high number of features per pixel and the amount of information taken keeps growing, and that implies, not only to analyse a lot of information, but also to send it to the Earth. So efficient systems are needed in order to reduce the energy consumption for their analysis in data centres, but also for achieving on-board analysis, so it will be possible to avoid sending all the information for many cases of use.

The need for reliable techniques constitutes another major challenge in the HI analysis for two main reasons: it is very difficult to generate HI datasets, so the training information tends to be scarce and unbalanced (some classes are less represented than others); but also, in this field is a fact that unknown inputs will be received during inference in real-world applications because it is not feasible to label every single material on the Earth surface, and also each pixel comprises a large piece of terrain, which implies different and unique mixtures of materials and surfaces.

This characteristics made HI analysis a perfect target for our study, and we used it for several of our experiments. With that in mind, we analysed some critical applications in that field, including the aspects related to the inference process in the most used ML techniques and the possibilities offered by the uncertainty metrics provided by bayesian techniques; and we developed accelerators capable of improving the efficiency of the inference process, specially focused on embedded systems.

The two firs steps of our research focus on efficiency and consist in the design and implementation on FPGAs of two accelerators for the inference process of two different ML algorithms. For that we will explain first, as one of our published contributions, an analysis of the inference process of some ML techniques, and then we will describe the two accelerator designs. The third step focuses on the search of reliable ML techniques, and consist in the analysis of bayesian networks for the analysis of HI.

1.1.1 Contributions

Analysis of Machine Learning Models on Inference. Machine learning techniques are widely used for pixel-wise classification of hyperspectral images. These methods can achieve high accuracy levels, but most of them are computationally intensive models. This poses a problem for their implementation in low-power and embedded systems intended for on-board processing, in which energy consumption and model size are as important as model accuracy. In this chapter, we present an overview of the inference properties of the most relevant techniques for hyperspectral image classification. For this purpose, we compare the size of the trained models and the operations required during the inference step (which are directly related to the hardware and energy requirements). Our goal is to search for appropriate trade-offs between on-board implementation aspects (such as model size and energy consumption) and classification accuracy.

This analysis of the inference characteristics of the main models for HI classification is published in [21].

Step I: Sparse Convolutional Neural Networks Accelerator. Deep neural networks (DNNs) are increasing their presence in a wide range of applications, and their computationally intensive and memory-demanding nature poses challenges, especially for embedded systems. Pruning techniques turn DNN models into sparse by setting most weights to zero, offering optimisation opportunities if specific support is included. We propose a novel pipelined architecture for DNNs that avoids all useless operations during the inference process. It has been implemented in a field-programmable gate array (FPGA), and the performance, energy efficiency, and area have been characterised. Exploiting sparsity yields remarkable speedups but also produces area overheads. We have evaluated this trade-off in order to identify in which scenarios it is better to use that area to exploit sparsity, or to include more computational resources in a conventional DNN architecture. We have also explored different arithmetic bitwidths. Our sparse architecture is clearly superior on 32-bit arithmetic or highly sparse networks. However, on 8-bit arithmetic or networks with low sparsity it is more profitable to deploy a dense architecture with more arithmetic resources than including support for sparsity. We consider that FPGAs are the natural target for DNN sparse accelerators since they can be loaded at run-time with the best-fitting accelerator.

This work is published in [22] and the correspondent code is in a public repository [44].

Step II: Gradient Boosting Decision Trees Accelerator. A decision tree is a well-known machine learning technique. Recently their popularity has increased due to the powerful Gradient Boosting ensemble method that allows to gradually increase accuracy at the cost of executing a large number of decision trees. In this chapter we present an accelerator designed to optimise the execution of these trees while reducing the energy consumption. We have implemented it in an FPGA for embedded systems, and we have tested it with a relevant case-study: pixel classification of hyperspectral images. In our experiments with different images our accelerator can process the hyperspectral images at the same speed at which they are generated by the hyperspectral sensors. Compared to a high-performance processor running optimised software, on average our design is twice as fast and consumes 72 times less energy. Compared to an embedded processor, it is 30 times faster and consumes 23 times less energy.

For that work we published the GBDT accelerator developed [16], which also has the codes in a public repository [15].

Step III: Bayesian Neural Networks for Hyperspectral Images. Machine learning techniques, and specifically neural networks, have proved to be very useful tools for image classification tasks. Nevertheless, measuring the reliability of these networks and calibrating them accurately is very complex. This is even more complex in a field like hyperspectral imaging, where labelled data are scarce and difficult to generate. Bayesian neural networks (BNNs) allow to obtain uncertainty metrics related to the data processed (aleatoric), and to the uncertainty generated by the model selected (epistemic). On this chapter we will demonstrate the utility of BNNs by analysing the uncertainty metrics obtained by a BNN over five of the most used hyperspectral images datasets. In addition we will illustrate how these metrics can be used for several practical applications such as identifying predictions that do not reach the required level of accuracy, detecting mislabelling in the dataset, or identifying when the predictions are affected by the increase of the level of noise in the input data.

This study also justifies the need of a BNN accelerator in the context of HI classification, whose design is in process and will be considered in this thesis as future work. The study results are published in [12], and the codes to reproduce all the experiments are available in a public repository [13].

1.2 Thesis Structure

The main chapters of this thesis, Chapters 3 to 6, are self-contained and each one is dedicated to one of the contributions mentioned before. Each one of them include their related work. The thesis is structured as follows.

Chapter 2

The second chapter gathers and explain the main concepts around this thesis such as Hyperspectral Images (HI) and some of the most relevant Machine Learning (ML) techniques.

Chapter 3

The third chapter shows the analysis of the inference process of the most used ML techniques for HI classification, which is the first of the contributions mentioned before.

Chapter 4

The fourth chapter explains the second of those contributions, the design and implementation of an accelerator for sparse CNNs.

Chapter 5

The fifth chapter is dedicated to the third of the contributions, the design and implementation of the GBDTs accelerator.

Chapter 6

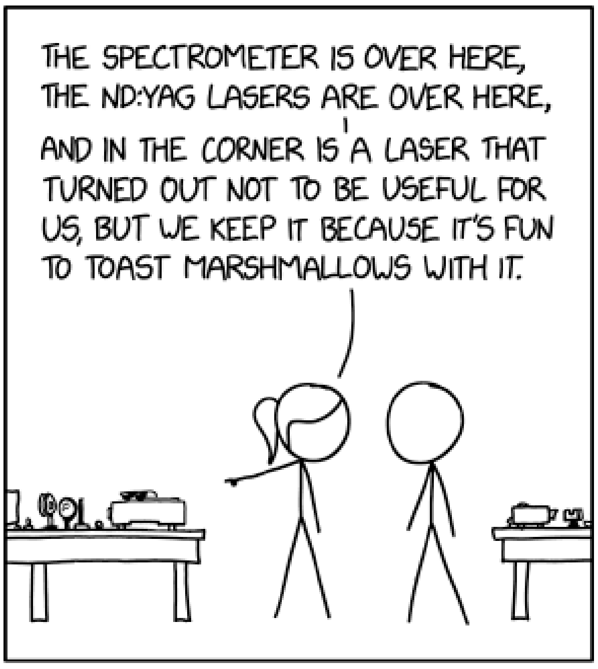
The sixth chapter explains the study of BNNs for HI classification, which corresponds to the fourth of the contributions mentioned before.

Chapter 7

The last chapter compiles the conclusions of the main contributions of this thesis.

Research Contextualisation

2


THE SPECTROMETER IS OVER HERE, THE ND:YAG LASERS ARE OVER HERE, AND IN THE CORNER IS A LASER THAT TURNED OUT NOT TO BE USEFUL FOR US, BUT WE KEEP IT BECAUSE IT'S FUN TO TOAST MARSHMALLOWS WITH IT.

EVERY LAB IN EVERY FIELD HAS SOME PIECE OF EQUIPMENT LIKE THIS.

— **xkcd.com**
(Just that guy, you know?)

In this chapter we will contextualise some important and transversal concepts and domains that will be present in the rest of the document.

First we will talk about machine learning, and specifically its particular needs for embedded systems, then we will explain what is a field programmable gate array (FPGA) and why do we use them for implementing our hardware designs, and in a final point, we will show the characteristics of hyperspectral images (HI), as one of our main target fields, and we will describe the HI datasets that we used for some of our experiments.

2.1 Machine Learning

Machine Learning (ML) techniques have been used more and more in the last years to automatise a great number of tasks in many different fields. These methods mainly consist in algorithms capable of identify patterns in a dataset used to train them, so they become able to search for these patterns in new, unseen, data and, based on that information, infer their characteristics.

One of the most frequent uses of ML, and the one selected as target in this work, is to classify images into a series of categories with supervised learning. That means to train the selected ML method with a dataset of images labelled with a collection of categories, so the trained model will be able to classify new images into this same group of categories. In particular we will focus in Hyperspectral Images (HI) classification, and we will talk about the HI datasets in Section 2.3.

The supervised training process consist in feeding the selected ML model with the labelled data, so it will produce an output. Then, according to the relation between the received output and the desired one, the parameters of the model will be slightly modified. If the parameters were modified too much, the model will only learn to fit the latest data, but the idea is that this process, called backpropagation, will be repeated thousands of times with very little modifications, so at the end the model will achieve a generic configuration to identify the features of all the data received. The learning rate is the training parameter that defines how much the model parameters will be modified on each iteration.

This training processes require a lot of computation and they are usually carried out using GPUs. This process is usually done using a framework, and most of them are currently adapted to support GPU libraries, but there are also specific accelerators to reduce the energy consumption during training.

Nevertheless, we are focusing on ML for embedded systems here, and our main concern is not the training process, but the inference. There are a lot of cases of use of this ML techniques deployed in small devices that need to perform the inference process in-place, and sometimes in real time. Images classification and pattern recognition are very used in this contexts such in autonomous vehicles, crop automation or medical equipment. To better understand the computational requirements of this task, in Chapter 3 we will explain and analyse in profundity the inference process of the ML methods most used to classify HI.

2.2 Field Programmable Gate Arrays

Our designs are meant to be deployed in a field programmable gate array (FPGA). An FPGA is a reprogrammable device where it is possible to deploy an specific logic circuit in an easy way compared to the manufacturing process of an Application-Specific Integrated Circuit (ASIC). For many years they have been used as a tool for the development process of ASICs, for research purposes and for some very specific tasks or systems that required acceleration but didn't worth the design cost of an ASIC. In recent years they have become a very useful tool for developing accelerators thanks to several factors such as the efforts to manage heterogeneous systems in data centres, the increase of its computing and on-chip memory capacity, the developing of better and better high level synthesis (HLS) languages and tools and the inclusion of ML techniques in several scenarios.

Specifically in remote sensing FPGAs are very useful, because reprogramability is a very desired quality in satellites and they use to be equipped with at least one FPGA. To develop FPGA accelerators for ML techniques can be the best option to achieve low-energy on-board analysis of multi and hyperspectral images.

Even though our designs could be implemented as an ASIC, they are specifically thought for FPGAs, as they have their own constraints. We use the Zedboard Xilinx Zynq-7000 evaluation board [10] shown in Figure 2.1, and we write our designs directly in a hardware description language (in our case VHDL) at a register-transfer level to be able to control the architecture of our designs and make low-level optimisations. We synthesise and implement using the Xilinx tool Vivado, and our codes are always available in public repositories. We will refer to them on each chapter.

The Zynq-7000 board supports software-hardware co-design using an ARM processor, so the FPGA can be exclusively dedicated to accelerate specific processes. It has dedicated DSP slices to carry out complex arithmetic computations and on-chip block RAMs to reduce the need to access to external memories. One of the reasons of using an small board as the series 7000 is that our accelerators intend to be proof of concept designs for embedded systems, so we actually want to develop small low-consuming designs. There are FPGAs with much higher capacity and speed, oriented to servers, but we are interested in low power FPGAs that can operate in a remote embedded system.

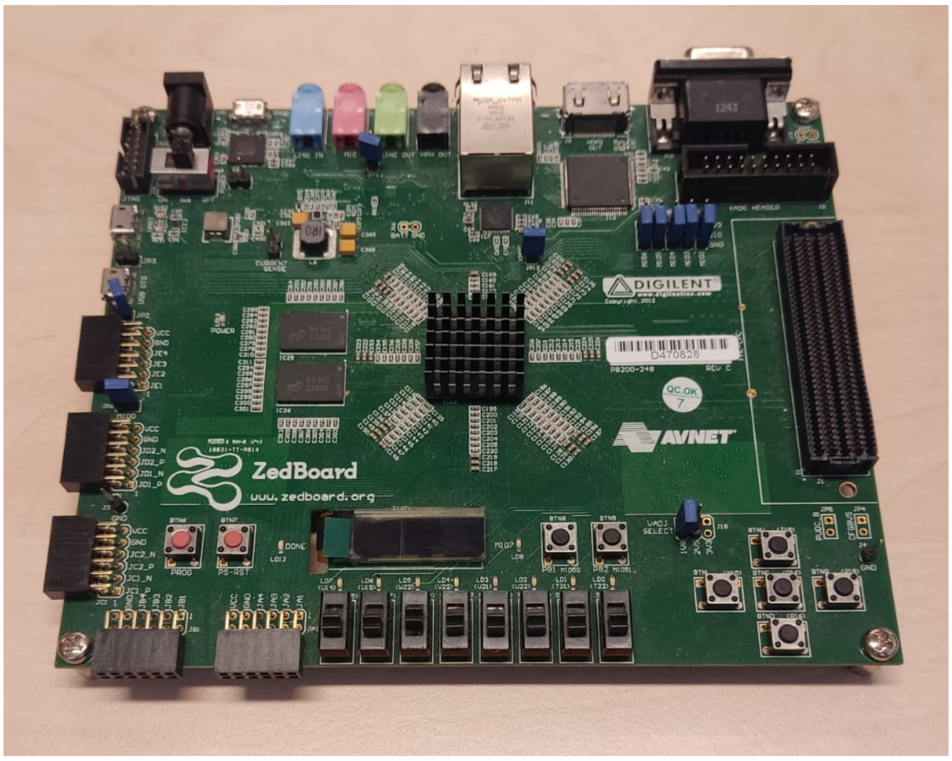


Figure 2.1: ZedBoard 7000 series.

The design and specific characteristics of our two accelerators will be described on their respective chapters. Chapter 4 will describe the sparse convolutional neural network accelerator and chapter 5 the gradient boosting decision trees accelerator.

2.3 Hyperspectral Datasets

Hyperspectral images consist of hundreds of spectral bands, where each band captures the responses of ground objects at a particular wavelength. Therefore, each pixel of the image can be considered as a spectral signature. Fig. 2.2 represents an example of the multiple bands of a hyperspectral image.

In hyperspectral images pixel classification, the input is a single pixel composed of a series of features, where each feature is a 16-bit integer. The number of features depends on the sensor used to obtain the image. The data sets that we use in our experiments are Botswana (BO) [2], Houston University (HU) [84], Indian Pines (IP) [2], Kennedy Space Center (KSC) [2], Pavia University (PU) [2] and Salinas Valley (SV) [2]. Figures 2.3 to 2.8 show an RGB representation of each image based on the algorithm provided by [24] along with the ground truth representation of

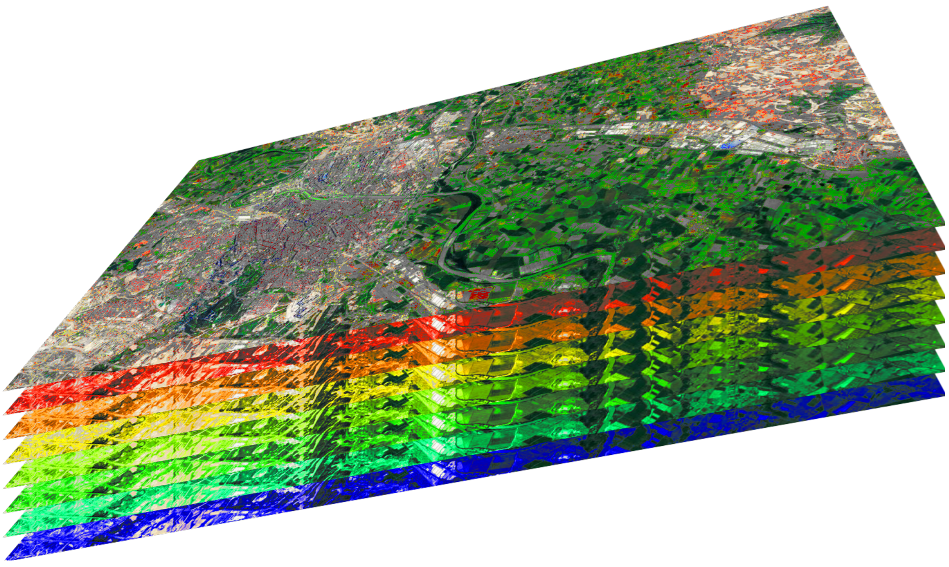


Figure 2.2: Hyperspectral image example.

each dataset. Table 2.1 shows the number of classes, labelled pixels of each class and spectral bands of every dataset, and the colours legend for the ground truth images.

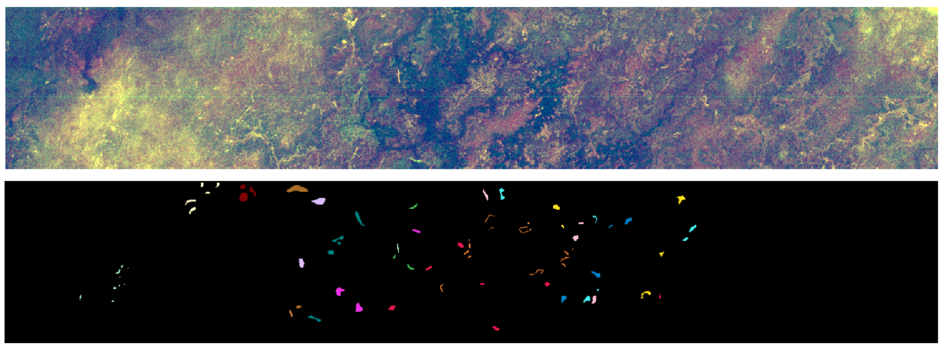


Figure 2.3: BO RGB representation and ground truth.

- The BO dataset is an image with water and vegetation collected by the Hyperion sensor over the Okavango Delta, Botswana, in 2001-2004. It has $1476x256$ pixels with 145 spectral bands after removing the uncalibrated and noisy bands that cover water absorption features. Each pixel has a 30m resolution. Only 3248 of the 377856 total pixels are labelled into 14 classes.

- The HU data set is an image of an urban area collected by the Compact Airborne Spectrographic Imager (CASI) sensor over the University of Houston, USA. It has $349x1905$ pixels with 144 spectral bands. Only 15029 of the total 664845 pixels are labelled into 15 classes. As it was the benchmark dataset for

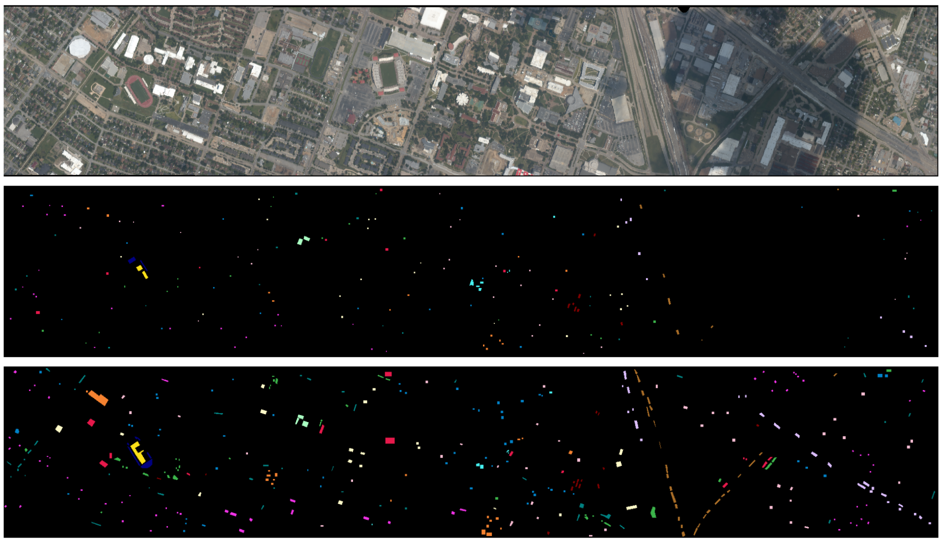


Figure 2.4: HU RGB representation, training and testing ground truth.

Table 2.1: Hyperspectral Datasets Characteristics.

	BO	HU		IP	KSC	PU	SV
		Train	Test				
Number of classes	14	15		16	13	9	16
🟥 Class 0 px.	270	198	1053	46	761	6631	2009
🟩 Class 1 px.	101	190	1064	1428	243	18649	3726
🟨 Class 2 px.	251	192	505	830	256	2099	1976
🟦 Class 3 px.	215	188	1056	237	252	3064	1394
🟧 Class 4 px.	269	186	1056	483	161	1345	2678
🟦 Class 5 px.	269	182	143	730	229	5029	3959
🟪 Class 6 px.	259	196	1072	28	105	1330	3579
🟪 Class 7 px.	203	191	1053	478	431	3682	11271
🟩 Class 8 px.	314	193	1059	20	520	947	6203
🟪 Class 9 px.	248	191	1036	972	404	-	3278
🟫 Class 10 px.	305	181	1054	2455	419	-	1068
🟨 Class 11 px.	181	192	1041	593	503	-	1927
🟥 Class 12 px.	268	184	285	205	927	-	916
🟩 Class 13 px.	95	181	247	1265	-	-	1070
🟦 Class 14 px.	-	187	473	386	-	-	7268
⬛ Class 15 px.	-	-	-	93	-	-	1807
Total labelled px.	3248	2832	12197	10249	5211	42776	56975
Spectral bands	145	144		200	176	103	204

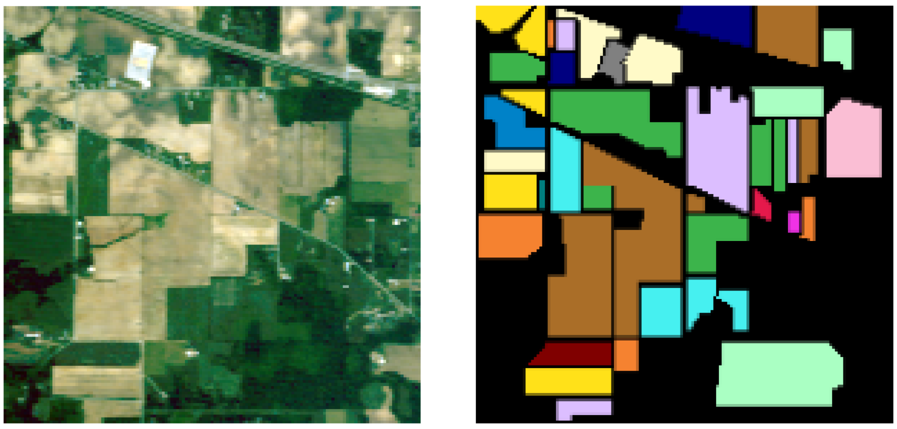


Figure 2.5: IP RGB representation and ground truth.

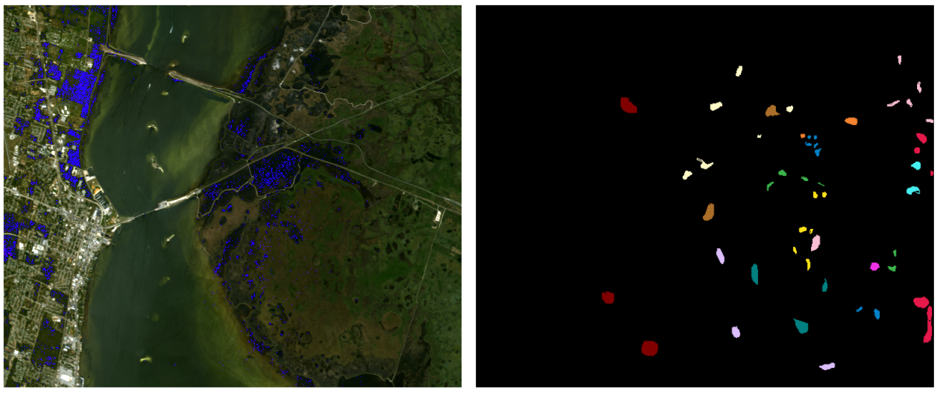


Figure 2.6: KSC RGB representation and ground truth.

a contest [87], it is already divided into training and testing sets, with 2832 and 12197 pixels, respectively.

- The IP data set is an image of an agricultural region, mainly composed of crop fields, collected by the Airborne Visible Infra-Red Imaging Spectrometer (AVIRIS) sensor in Northwestern Indiana, USA. It has $145x145$ pixels with 200 spectral bands after removal of the noise and water absorption bands. Of the total 21025 pixels, 10249 are labelled into 16 different classes.

- The KSC data set is an image with water and vegetation collected by the AVIRIS sensor over the Kennedy Space Center in Florida, USA. It has $512x614$ pixels with 176 spectral bands after removing the water absorption and low signal-to-noise bands. Only 5211 out of the available 314368 pixels are labelled into 13 classes.

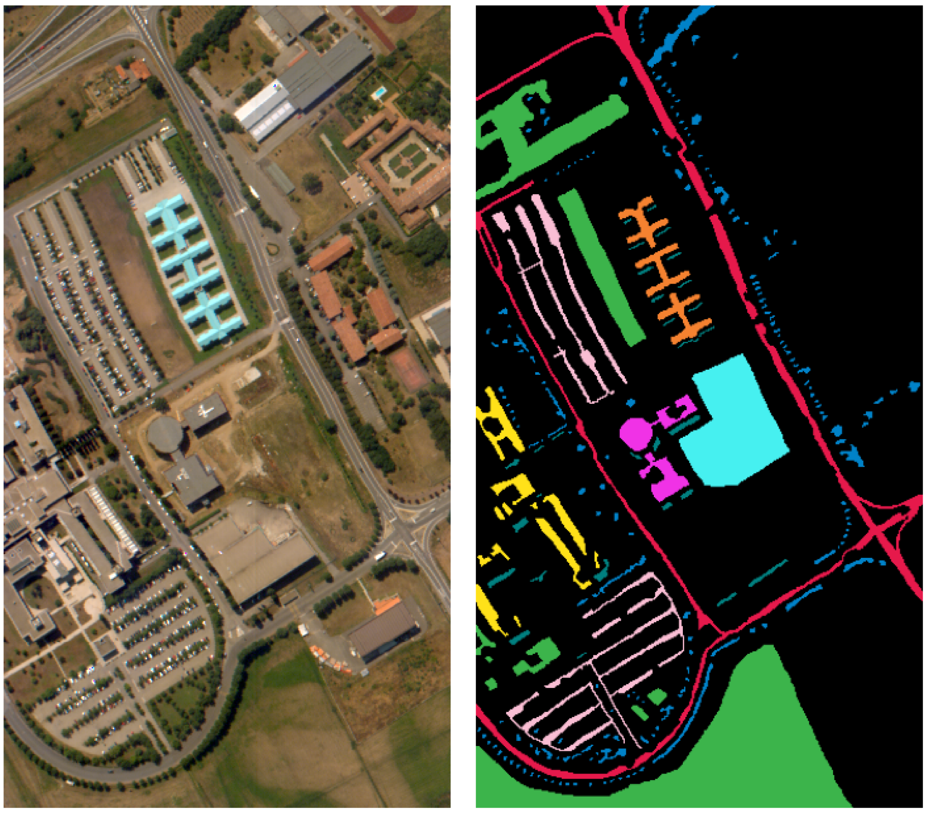


Figure 2.7: PU RGB representation and ground truth.

- The PU data set is an image of an urban area, the city of Pavia in Italy, collected by the Reflective Optics Spectrographic Imaging System (ROSIS), a compact airbone imaging spectrometer. It is composed of $610x340$ pixels with 103 spectral bands. Only 42776 pixels from the total 207400 are labelled into 9 classes.
- The SV data set is an image composed of agricultural fields and vegetation, collected by the AVIRIS sensor in Western California, USA. It has $512x217$ pixels with 204 spectral bands after removing the noise and water absorption bands. Of the total 111104 pixels, 56975 are labelled into 16 classes.

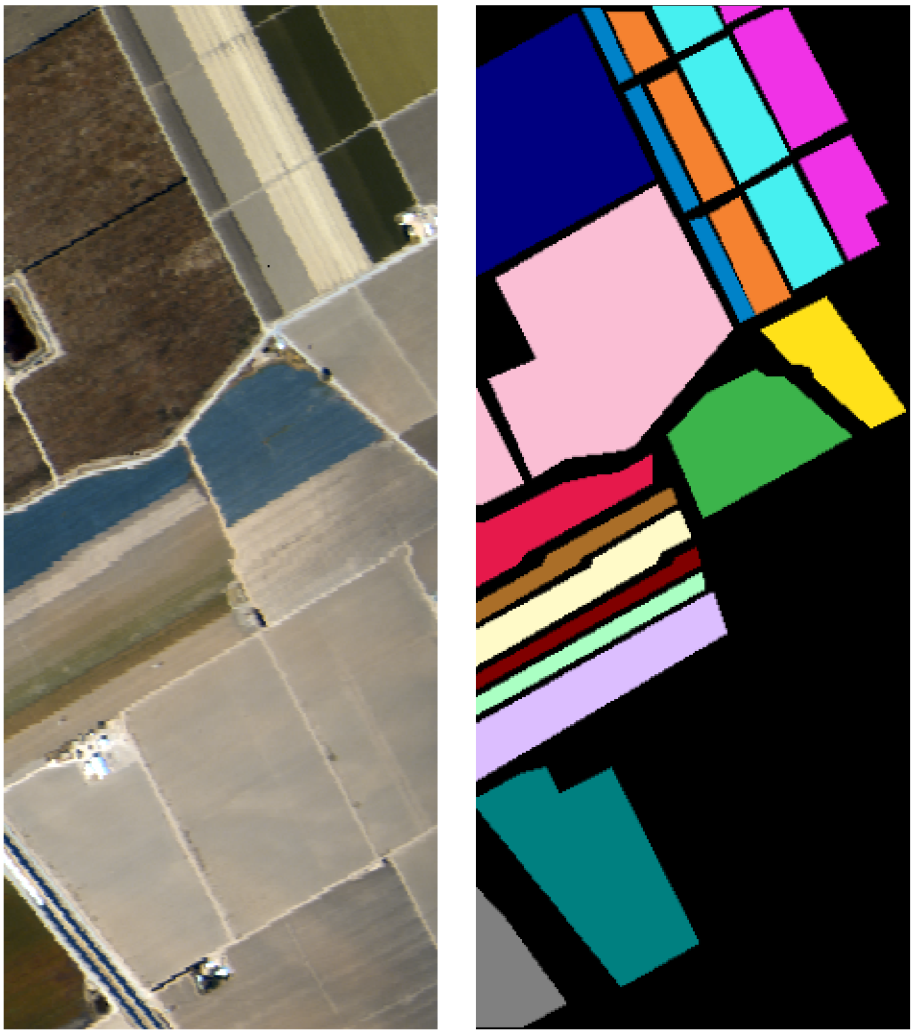
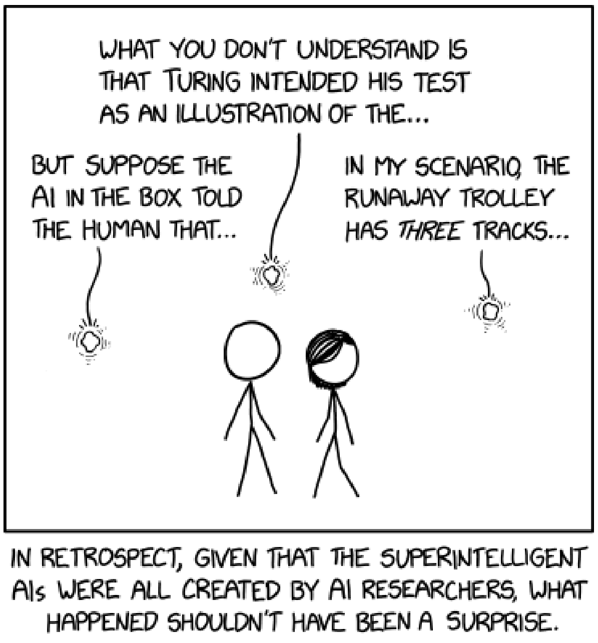


Figure 2.8: SV RGB representation and ground truth.

3 Analysis of Machine Learning Models on Inference


WHAT YOU DON'T UNDERSTAND IS THAT TURING INTENDED HIS TEST AS AN ILLUSTRATION OF THE...
BUT SUPPOSE THE AI IN THE BOX TOLD THE HUMAN THAT...
IN MY SCENARIO, THE RUNAWAY TROLLEY HAS *THREE* TRACKS...
IN RETROSPECT, GIVEN THAT THE SUPERINTELLIGENT AIs WERE ALL CREATED BY AI RESEARCHERS, WHAT HAPPENED SHOULDN'T HAVE BEEN A SURPRISE.

— **xkcd.com**
(Just that guy, you know?)

3.1 Introduction

Machine Learning (ML) techniques have become the most used mechanisms to automatically analyse and classify images. In particular, they are also used in the field of remote sensing to deal with the complexity of analysing multispectral and hyperspectral images. Nevertheless, most of the improvements on this area imply a high demand of resources and energy consumption, which could not be the best option for many problems, specifically in the context of embedded systems, such as on-board processing of remote sensing images.

In this chapter we analyse the inference process of the most used ML methods in the context of HI classification, focusing on on-board processing, which requires both performance and low-power. For that, we will first explain each method algorithm so we can then quantify the operations and type of data they require during inference. We will also train the models to classify some HI datasets so we can measure accuracy.

This comparative study of the ML techniques has been published in a high quality journal [21].

3.2 Related Work

Traditionally, the data gathered by remote sensors have to be downloaded to the ground segment, when the aircraft or spacecraft platform is within the range of the ground stations, in order to be pre-processed by applying registration and correction techniques and then distributed to the final users, which perform the final processing (classification, unmixing, object detection).

Nevertheless, this procedure introduces important delays related to the communication of a large amount of remote sensing data (which is usually in the range of GB-TB) between the source and the final target, producing a bottleneck that can seriously reduce the effectiveness of real-time applications [92]. Hereof, real-time on-board processing is a very interesting topic within the remote sensing field that has significantly grown in recent years to mitigate these limitations, and to provide a solution to these types of applications [104, 102, 94, 83, 33].

In addition to avoiding communication latencies, the on-board processing can considerably reduce the amount of bandwidth and storage required in the collection of HI data, allowing for the development of a more selective data acquisition and reducing the cost of on-the-ground processing systems [96]. As a result, low-power consumption architectures such as field-programmable gate array (FPGAs) [93, 34] and efficient GPU architectures [45] have emerged as an alternative to transfer part of the processing from the ground segment to the remote sensing sensor, which leads us to the need of an analysis of the inference operations of the most used methods.

3.3 Inference Characteristics of Models for Hyperspectral Images Classification

We selected some of the most relevant techniques for HI data classification to be compared in the inference stage. These techniques are: Multinomial Logistic Regression (MLR), Random Forest (RF), Support Vector Machine (SVM), Multi-Layer Perceptron (MLP), and a shallow Convolutional Neural Network (CNN) with 1D kernel, as well as Gradient Boosting Decision Trees (GBDT), a tree based technique that has successfully been used in other classification problems. In order to compare them, it is necessary to perform the characterisation of each algorithm in the inference stage, measuring the size in memory of the trained model and analysing the number and type of operations needed to perform the complete inference stage for the input data.

In the case of HI classification, the input is a single pixel vector composed of a series of features, and each one of these features is a 16-bit integer value. Each model treats the data in different ways so, for instance, the size of the layers of a neural network will depend on the number of features of the pixels of each data set, while the size of a tree-based model will not. We will explain the characteristics of the different models and the inference process for each of them.

3.3.1 Multinomial Logistic Regression

The MLR classifier is a probabilistic model that extends the performance of binomial logistic regression for multi-class classification, approximating the posterior probability of each class by a softmax transformation. In particular, for a given HI training set $\mathcal{D}_{train} = \{\mathbf{x}_i, y_i\}_{i=1}^{M}$ composed by M pairs of spectral pixels $\mathbf{x}_i \in \mathbb{R}^B$ and their corresponding labels $y_i \in \mathcal{Y} = \{1, \cdots, K\}$, the posterior probability $P(y_i = k|\mathbf{x}_i, \Theta)$ of the k-th class is given by Eq. (3.1) [97]:

$$P(y_i = k|\mathbf{x}_i, \Theta) = \frac{\exp\left(\boldsymbol{\theta}_k \cdot h(\mathbf{x}_i)\right)}{\sum_{j=1}^{K} \exp\left(\boldsymbol{\theta}_j \cdot h(\mathbf{x}_i)\right)} \tag{3.1}$$

being $\boldsymbol{\theta}_k$ the set of logistic regressors for class k, considering $\Theta = \{\boldsymbol{\theta}_1, \cdots, \boldsymbol{\theta}_{K-1}, 0\}$ as all the coefficients of the MLR, while $h(\cdot)$ is a feature extraction function defined over the spectral data $\mathbf{x}_i$, which can be linear, i.e., $h(\mathbf{x}_i) = \{1, x_{i,1}, \cdots, x_{i,B}\}$, or non-linear (for instance, kernel approaches [81]). In this work, linear MLR is considered.

Standardisation of the data set is needed before training, so the data are compacted and centred around the average value. This process implies the calculation of the average ($\overline{x}$) and standard deviation (s) values of the entire data set X to apply Eq. (3.2) to each pixel $\mathbf{x}_i$. In HI processing, it is common to pre-process the entire data set before splitting it into the training and testing subsets, so $\overline{x}$ and s include the test set, which is already standardised to perform the inference after training. Nevertheless, in a real environment, $\overline{x}$ and s values will be calculated from the training data and then the standardisation should be applied on-the-fly, applying these values to the input data received from the sensor. This implies not only some extra calculations to perform the inference for each pixel, but also some extra assumptions on the representativeness of the training data distribution. These extra calculations are not included in the measurements of Section 3.5.2.

$$\mathbf{x}_i' = \frac{\mathbf{x}_i - \overline{x}}{s} \tag{3.2}$$

The MLR model has been implemented in this work with the `scikit` learn Logistic Regression model with a multinomial approach and `lbfgs` solver [4]. The trained model consists of one estimator for each class, so the output of each estimator represents the probability of the input belonging to that class. The formulation of the inference for the class k estimator (y_k) corresponds to Eq. (3.3), where $\mathbf{x}_i = \{1, x_{i,1}, \cdots, x_{i,B}\}$ is the input pixel and $\boldsymbol{\theta}_k = \{\theta_{k,0}, \cdots, \theta_{k,B}\}$ correspond to the bias value and the coefficients of the estimator of class k.

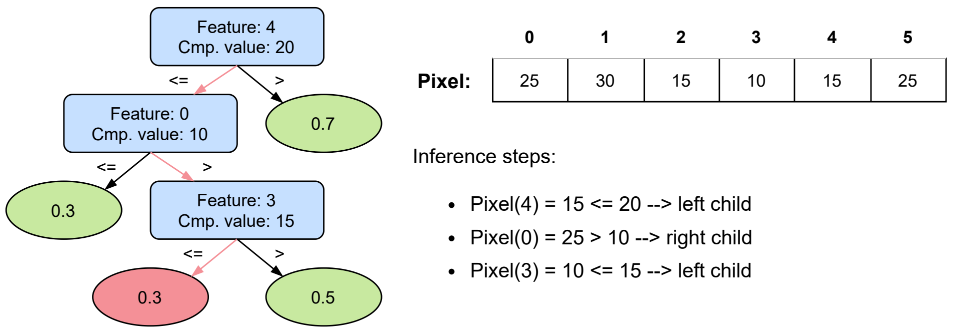
$$y_{k,i} = \boldsymbol{\theta}_k \cdot \mathbf{x}_i = \theta_{k,0} + \theta_{k,1}x_{i,1} + \theta_{k,2}x_{i,2} + \cdots + \theta_{k,B}x_{i,B} \tag{3.3}$$

As a result, the model size depends on the number of classes (K) and features (B), having $K(B+1)$ parameters. The inference of one pixel requires KB floating point multiplications and KB floating point accumulations. In this case, we have a very small model and it does not require many calculations. However, since it is a linear probabilistic model, its accuracy may be limited in practice, although it can be very accurate when there is a linear relation between the inputs and outputs.

3.3.2 Decision Trees

A Decision Tree is a decision algorithm based on a series of comparisons connected among them as in a binary tree structure, so that the node comparisons lead the search to one of the child nodes, and so on, until reaching a leaf node that contains the result of the prediction. During training, the most meaningful features are selected and used for the comparisons in the tree. Hence the features that contain more information will be used more frequently for the comparison, whether those that do not provide useful information for the classification problem will simply be ignored [109]. This is an interesting property of this algorithm since, based on the same decisions made during training to choose features, we can easily determine the feature importance. This means that Decision Trees can be also used to find out which features carry the main information load, and that information can be used to train even smaller models keeping most of the information of the image with much less memory impact.

Figure 3.1 shows the inference operation of a trained Decision Tree on a series of feature inputs with a toy example. In the first place, this tree takes feature 4 of the input and compares its value with the threshold value 20; as the input value is lower than the threshold it continues on the left child, and keeps with the same procedure until it reaches the leaf with 0.3 as output value.



Inference steps:

- Pixel(4) = 15 <= 20 --> left child
- Pixel(0) = 25 > 10 --> right child
- Pixel(3) = 10 <= 15 --> left child

Figure 3.1: Decision Tree example.

One of the benefits of using Decision Trees over other techniques is that they do not need any input preprocessing such as data normalisation, scaling or centring. They work with the input data as it is [55]. The reason is that features are never mixed. As can be seen in Figure 3.1, in each comparison the trees compare the value of an input feature with another value of the same feature. Hence, several features can have different scales. In other Machine Learning models, as we just saw in MLR for example, features are mixed to generate a single value so, if their values belong to different orders of magnitude, some features will initially dominate the result. This can be compensated during the training process, but in general normalisation or other preprocessing technique will be needed to speed up training and improve the results. Besides, the size of the input data does not affect the size of the model, so dimensionality reduction techniques such as Principal Component Analysis (PCA) are not needed to reduce the model size, which substantially reduces the amount of calculation needed at inference.

Nevertheless, a single Decision Tree does not provide accurate results for complex classification tasks. The solution is to use an ensemble method that combines the results of several trees in order to improve the accuracy levels. We will analyse two of these techniques, Random Forest (RF) and Gradient Boosting Decision Trees (GBDT). In terms of computation, most of the machine learning algorithms need a significant amount of floating point operations on inference, and most of them are multiplications. By contrast, the inference with an ensemble of Decision Trees just need a few comparisons per tree. In terms of memory requirements, the size of this models depends on the number of trees and the number of nodes per tree, but the memory accesses, and therefore the used bandwidth, is much lesser than the size of the model because Decision Trees only need to access a small part of the model to perform an inference.

In the case of hyperspectral-images pixel classification, the input is a single pixel composed of a series of features. Each node specialises in a particular feature during training, meaning that, at the time of inference, one particular node performs a single comparison between its trained value and the value of the corresponding feature. Since the feature values of hyperspectral images are 16-bit integers, each node just need an integer comparison to made their decision; i.e. left or right child. This is a very important characteristic for embedded and on-board systems. In most ML models the inputs are multiplied by a floating-point value, hence even when the input model is an integer, all the computations will be floating-point. However, a tree only need to know whether the input is smaller or greater than a given value, and that value can be an integer without any accuracy loss. So in the case of hyperspectral images pixel-classification, this technique behaves exceptionally in terms of computation. Decision Trees are fast and efficient during inference and can be executed even by low-power microcontrollers.

Random Forest

A typical approach to use as ensemble method is RF techniques, where several trees are trained for the same data set, but each one of them for a random subsample of the entire data set. Moreover, the search of the best split feature for each node is done on a random subset of the features. Then each classifier votes for the selected class [101].

The RF model has been implemented with the scikit learn Random Forest Classifier model [3]. In this implementation the final selection is done by averaging the predictions of every classifier instead of voting, which implies that each leaf node must keep a prediction value for each class, so after every tree performs the inference the class is selected from the average of every prediction. This generates big models but, as we said, it only needs to access a small part of it during inference.

Gradient Boosting

However, even better results can be obtained applying a different ensemble approach called Gradient Boosting. This technique is an ensemble method that combines the results of different predictors in such a way that each tree attempts to improve the results of the previous ones. Specifically, gradient boosting method consists in training predictors sequentially so each new iteration try to correct the residual error generated in the previous one. That is, each predictor is trained to correct the residual error of its predecessor. Once the trees are trained, they can be used for prediction by simply adding the results of all the trees [103, 55].

The GBDT model has been implemented with the LightGBM library Classifier [6]. For multi-class classification, one-vs-all approach is used in GBDT implementation, which means that the model trains a different estimator for each class. The output of the correspondent estimator represents the probability that the pixel belongs to that class, and the estimator with the highest result is the one that corresponds to the selected class. On each iteration, the model adds a new tree to each estimator. The one-vs-all approach makes it much easier to combine the results given that each class has their own trees, so we just need to add the results of the trees of each class separately, as shown in Figure 3.2. This accumulations of the output values of the trees are the only operations in floating point that the GBDT need to perform.

Due to its iterative approach, GBDT model also allows designers to trade-off accuracy for computation and model size. For example, if a GBDT is trained for 200 iterations, it will generate 200 trees for each class. Afterwards, the designer can decide whether to use all of them, or to discard the final ones. It is possible to find similar trade-off with other ML models, for instance reducing the number of convolutional layers in a CNN or the number of hidden neurons in a MLP. However, in that case, each possible design must be trained again, whereas in GBDT only one train is needed,

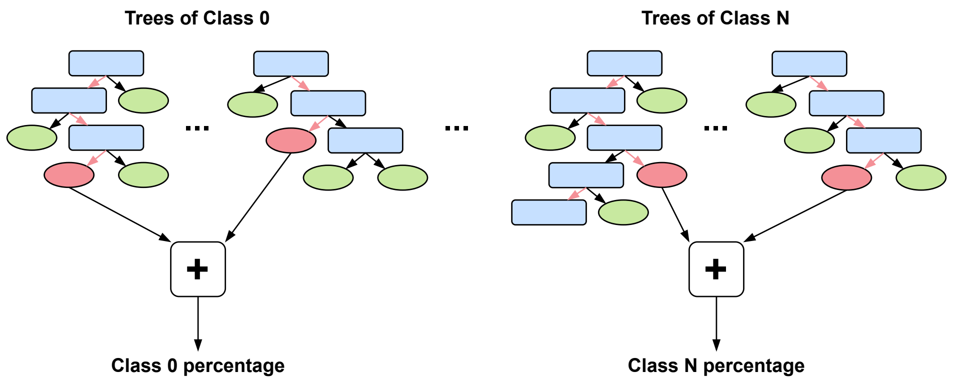


Figure 3.2: GBDT results accumulation with one-vs-all approach.

and afterwards the designer can simply evaluate the results when using different number of trees and generate a Pareto curve with the different trade-offs.

3.3.3 Support Vector Machine

A Support Vector Machine (SVM) is a kernel-based method commonly used for classification and regression problems. It is based on a two-class classification approach, the Support Vector Network algorithm. To find the smallest generalisation error, this algorithm searches for the optimal hyperplane, i.e. a linear decision function that maximises the margin among the support vectors, which are the ones that define the decision boundaries of each class [106]. In the case of pixel-based classification of hyperspectral images, we need to generalise this algorithm to a multi-class classification problem. This can be done following a one-vs-rest, or one-vs-all, approach, training K separate SVMs, one for each class, so each two-class classifier will interpret the data from its own class as positive examples and the rest of the data as negative examples [97].

The SVM model has been implemented with the scikit learn Support Vector Classification (SVC) algorithm [5], which implements one-vs-rest approach for multi-class classification. SVM model also requires preprocessing of the data applying standardisation Eq. (3.2), with the same implications explained in Section 3.3.1. According to scikit learn SVC mathematical formulation [5], the decision function is 3.4, where $\mathcal{K}(\mathbf{v}_i, \mathbf{x})$ corresponds to the kernel. We used the Radial Basis Function (RBF) kernel, whose formulation is 3.5. So the complete formulation of the inference operations is 3.6, where $\mathbf{v}_i$ corresponds to the i-th support vector, $y_i\alpha_i$ product is the coefficient of this support vector, $\mathbf{x}$ corresponds to the input pixel, ρ is the bias value and γ is the value of the $gamma$ training parameter.

$$\operatorname{sgn}\left(\sum_{i=1}^{M} y_i \alpha_i \mathcal{K}(\mathbf{v}_i, \mathbf{x}) + \rho\right) \tag{3.4}$$

$$\exp\left(-\gamma \|\mathbf{v}_i - \mathbf{x}\|^2\right) \tag{3.5}$$

$$\operatorname{sgn}\left(\sum_{i=1}^{M} y_i \alpha_i \exp\left(-\gamma \|\mathbf{v}_i - \mathbf{x}\|^2\right) + \rho\right) \tag{3.6}$$

The number of support vectors defined as M on Eq. (3.6) of the SVM model will be the number of data used for the training set. So this model does not keep too many parameters, which makes it small in memory size, but in terms of computation, it requires a great amount of calculus to perform one inference. The number of operations will depend on the number of features and the number of training data, which makes it unaffordable in terms of computation for really big data sets. Moreover, as it uses one-vs-all, it will also depend on the number of classes because it will train an estimator for each one of them.

3.3.4 Neural Networks

Neural Networks have become one of the most used machine learning techniques for images classification, and they have also proved to be a good choice for hyperspectral images classification. A Neural Network consists of several layers sequentially connected so that the output of one layer becomes the input of the next one. Some of the layers can be dedicated to intermediate functions, like pooling layers that reduce dimensionality highlighting the principal values, but the main operation, as well as most of the calculations, of a Neural Network resides in the layers based on neurons. Each neuron implements Eq. (3.7), where x is the input value and w and b are the learned weight and bias respectively, which are float values.

$$y = xw + b \tag{3.7}$$

Usually, when applying groups of Neurons in more complex layers, the results of several Neurons are combined such as in a dot product operation, as we will see for example in Section 3.3.4, and this w and b values are float vectors, matrices or tensors, depending on the concrete scenario. So the main calculations in Neural Networks are float multiplications and accumulations, and the magnitude of these computations depends on the number and size of the layers of the Neural Network. The information we need to keep in memory for inference consists in all these learned values, so the size of the model will also depend on the number and size of the layers.

Neural Network models also require preprocessing of the data. Without it, the features with higher and lower values will initially dominate the result. This can be compensated during training process, but in general normalisation will be needed to speed up training and improve the results. As for MLR and SVM models, standardisation Eq. (3.2) of the data sets was applied.

Multi Layer Perceptron

A Multi Layer Perceptron (MLP) is a Neural Network with at least one hidden layer, i.e. intermediate activation values, which requires at least two fully-connected layers.Considering the l-th fully connected layer, its operation corresponds to Eq. (3.8), where $\mathbf{X}^{(l-1)}$ is the layer's input, which can come directly from the original input or from a previous hidden layer $l-1$, $\mathbf{X}^{(l)}$ is the output of the current layer, resulting from applying the weights $\mathbf{W}^{(l)}$ and biases $\rho^{(l)}$ of the layer. If the size of the input $\mathbf{X}^{(l-1)}$ is $(M, N^{(l-1)})$, being M the number of input samples and $N^{(l-1)}$ the dimension of the feature space, and the size of the weights $\mathbf{W}^{(l)}$ is $(N^{(l-1)}, N^{(l)})$, the output size will be $(M, N^{(l)})$, i.e. the M samples represented in the feature space of dimension $N^{(l)}$ and defined by the l-th layer. In the case of hyperspectral imaging classification, the input size for one spectral pixel will be $(1, B)$, where B is the number of spectral channels, while the final output of the model size will be $(1, K)$, where K is the number of considered classes.

$$\mathbf{X}^{(l)} = \mathbf{X}^{(l-1)}\mathbf{W}^{(l)} + \rho^{(l)} \tag{3.8}$$

MLP model has been implemented with PyTorch Neural Network library [7], using the Linear classes to implement two fully-connected layers. The number of neurons of the first fully-connected layer is a parameter of the network, and the size of each neuron of the last fully-connected layer will depend on it. In the case of hyperspectral images pixel classification, the input on inference will be a single pixel ($M = 1$ according to last explanation) with B features and the final output will be the classification for the K classes, so the size of each neuron of the first fully-connected layer will depend on the number of features, while the number of neurons of the last fully-connected layer will be the number of classes.

As the input for pixel classification is not very big, this model keeps an small size once trained. During inference it will need to perform a float multiplication and a float accumulation for each one of its parameters, among other operations, so even being small the operations needed are expensive in terms of computation.

Convolutional Neural Network

A Convolutional Neural Network (CNN) is a Neural Network with at least one Convolutional layer. Instead of fully-connected neurons, Convolutional layers apply locally-connected filters per layer. These filters are smaller than the input and each one of them performs a convolution operation on it. During a convolution, the filter performs dot product operations within different sections of the input while it keeps moving along it. For hyperspectral images pixel classification, whose input consist in a single pixel, the 1D convolutional layer operation can be described with Alg. (3.1), where input pixel $\mathbf{x}$ has B features, the layer has F filters and each filter Q has q values, i.e. weights in the case of 1D convolution, and one bias ρ, so the output $\mathbf{X}'$ will be of shape $(B - q + 1, F)$. The initialisation values $LAYER\_FILTERS$ and $LAYER\_BIASES$ correspond respectively to the learned weights and biases of the layer.

```
# Layer activations and biases
$W \leftarrow LAYER\_FILTERS$ # Matrix of F filters with q weights each
$\rho \leftarrow LAYER\_BIASES$ # Array of F bias values
# Get the input
$x \equiv INPUT\_PIXEL$ # Array of B values
# Generate empty output structure
$X = zeros\_matrix((B - q + 1, F))$
# For each filter
for f in range(F):
    # Movement of the filter along the input
    for i in range(B - q + 1):
        # Dot product along the current filter and input position
        for j in range(q):
            $X[i, f] \mathrel{+}= W[f, j] \times x[i + j]$
        # Add the bias value
        $X[i, f] \mathrel{+}= \rho[f]$
# Return the output matrix
return $X$
```

Algorithm 3.1: 1D convolutional layer algorithm.

The CNN model has been implemented with PyTorch Neural Network library [7], using the Convolution, the Pooling and the Linear classes to define a Network with respectively one 1D convolutional layer, one max pooling layer and two fully connected layers at the end. The input of the 1D Convolutional layer will be the input pixel, while the input of the rest of the layers will be the output of the previous one, in the specified order. The number and size of the filters of the 1D convolutional layer are parameters of the network, nevertheless the relation between the profundity of the filters and the number of features will determine the size of the first fully connected layer, which is the biggest one. The Max Pooling layer does not affect the size of the model, since it only performs a size reduction by selecting the maximum value within small sub-sections of the input, but it will affect the number of operations as it needs to perform several comparisons. The fully connected layers are actually an MLP, as explained in Section 3.3.4. The size of the last fully connected layer will also depend on the number of classes. In terms of computation, the convolutional layer is very intensive in calculations, as can be observed in Alg. (3.1), and most of them are floating point multiplications and accumulations.

3.3.5 Summary of the relation of the models with the input

Each discussed model has different characteristics on its inference operation, and the size and computations of each one depends on different aspects of the input and the selected parameters. Table 3.1 summarises the importance of the data set size (in the case of hyperspectral images this is the number of pixels of the image), the number of features (number of spectral band of each hyperspectral pixel), and the number of classes (labels) in relation to the size and the computations of each model. The dots in Table 3.1 correspond to a qualitative interpretation, from not influenced at all (zero dots) to very influenced (three dots), regarding how each model size and number of computations is influenced by the size of the data set, the number of features of each pixel, and the number of classes. This interpretation is not intended to be quantitative but qualitative, i.e. just a visual support for the following explanations.

The number of classes is an important parameter for every model, but it affects them in a very different way. Regarding the size of the model, the number of classes defines the size of the output layer in the MLP and CNN, while for the MLR, GBDT and SVM the entire estimator is replied as many times as the number of classes. Since the RF needs to keep the prediction for each class on every leaf node, the number of classes is crucial to determine the final size of the model, and affects it much more. Regarding the computation, in the MLR, GBDT and SVM models the entire number of computations is multiplied by the number of classes, so it affects them very much. Furthermore, in the SVM model the number of classes will also affect the number of support vectors needed, because it is necessary to have enough training data for every class, so each new class not only increases the number of estimators, but also increases the computational cost by adding new support vectors. In neural networks, the number of classes defines the size of the output (fully connected) layer, which implies multiply and accumulate floating point operations, but this is the smallest layer for both models. And in the case of RF, it only affects the final calculations of the results, but it is important to remark that these are precisely the floating point operations of this model.

The number of features is not relevant for decision tree models during inference, that is why they do not need any dimensionality reduction techniques. The size of each estimator of the MLR and the SVM models will depend directly on the number of features, so it influences the size as much as the number of classes. In neural networks, it affects the size of the first fully connected layer (which is the biggest one), so the size of these models is highly influenced by the number of features. Nevertheless, in the case of the MLP, it only multiplies the dimension of the fully connected layer so it does not impact that much as in the case of the CNN, where it will be also multiplied by the number of filters of the convolutional layer. In a similar way, the number of operations of each estimator of the MLR and the SVM models will be directly influenced by the number of features. Again, for the MLP it will increase the number of operations of the first fully connected layer and for the CNN also the convolutional layer, which is very intensive in terms of calculations.

The size of the data set (and specifically the size of the training set) only affect the SVM model, because it will generate as many support vectors as the number of different data samples used in the training process. Regarding the size of the model, it implies to multiply the number of parameters of each estimator, so it will affect the size of the model as much as the number of classes. Actually, both the training set and the number of classes are related to each other. Regarding the number of operations, the core of Eq. (3.6) depends on the number of support vectors, so its influence is very high.

It is also worth noting that decision trees are the only ones that do not require any pre-processing to the input data. As we already explained in Section 3.3.1, this implies some extra calculations not included in the measurements of Section 3.5.2, but they can also be a source of possible inaccuracies because of the implications they could have once applied to a real system with entirely new data taken in different moments and conditions. For instance, applying standardisation means that we will

Table 3.1: Summary of the size and computational requirements of the considered models.

		Size dependencies			Computation dependencies		
	preprocessing	data set	features	classes	data set	features	classes
MLR	standardisation	-	●●	●●	-	●●	●●
RF	-	-	-	●●●	-	-	●
GBDT	-	-	-	●●	-	-	●●
SVM	standardisation	●●	●●	●●	●●●	●●	●●●
MLP	standardisation	-	●●	●	-	●●	●
CNN1D	standardisation	-	●●●	●	-	●●●	●

subtract the mean value of our training set to the data, and reduce it in relation to the standard deviation of our training set.

3.4 Training Configurations for the Models Comparison

The data sets selected for these experiments are the Indian Pines (IP) [2], Pavia University (PU) [2], Kennedy Space Center (KSC) [2], Salinas Valley (SV) [2] and Houston University (HU) [84]. Their characteristics have been explained in Section 2.3.

The implementation of the algorithms previously analysed have been developed and tested on a hardware environment with an X Generation Intel® Core™i9-9940X processor with 19.25M of Cache and up to 4.40GHz (14 cores/28 way multi-task processing), installed over a Gigabyte X299 Aorus, 128GB of DDR4 RAM. Also, a graphic processing unit (GPU) NVIDIA Titan RTX GPU with 24GB GDDR6 of video memory and 4608 cores has been used. We detailed in Section 3.3 the libraries and classes used for the implementation of each model: MLR with scikit learn Logistic Regression, Random Forest with scikit learn Random Forest Classifier, GBDT with LightGBM Classifier, SVM with scikit learn Support Vector Classification, MLP with PyTorch Neural Network Linear layers and CNN1D with PyTorch Neural Network Convolutional, Pooling and Linear layers.

For each dataset we trained the models applying cross-validation techniques to select the final training hyperparameters. After the cross-validation, the selected values not always correspond to the best accuracy, but to the best relation between accuracy and model size and requirements. The selected hyperparameters shown in Table 3.2 are the penalty of the error (C) for the MLR, the number of trees (n), the minimum number of data to split a node (m) and maximum depth (d) for both the RF and the GBDT, and also the maximum number of features to consider for each split (f)

Table 3.2: Selected training parameters of the different tested models.

	MLR	RF			GBDT			SVM		MLP	CNN1D				
	C	n	m	f	d	n	m	d	C	γ	h.l.	f	q	p	f1	f2
IP	1	200	2	10	10	200	20	20	100	2^{-9}	143	20	24	5	100	16
PU	10	200	2	10	10	150	30	30	10	2^{-4}	78	20	24	5	100	9
KSC	100	200	2	10	10	300	30	5	200	2^{-1}	127	20	24	5	100	13
SV	10	200	2	40	60	150	80	25	10	2^{-4}	146	20	24	5	100	16
HU	$1e5$	200	2	10	40	150	30	35	$1e5$	2^{-6}	106	20	24	5	100	15

for the RF, the penalty of the error (C) and kernel coefficient (γ) for the SVM, the number of neurons in the hidden layer (h.l.) for the MLP and for the CNN, the number of filters of the convolutional layer (f), the number of values of each filter (q), the size of the kernel of the max pooling layer (p) and the number of neurons of the first and last fully connected layers ($f1$) and ($f2$), respectively.

The final configurations of some models not only depend on the selected hyperparameters, but also on the training data set (for the SVM model) and the training process itself (for the RF and GBDT models). Table 3.3 shows the number of features (ft.) and classes (cl.) of each image and the final configurations of RF, GBDT and SVM models. For the tree models, the shown values are the total number of trees (trees), which in the case of the GBDT model depends on the number of classes of each image, the total number of non-leaf nodes (nodes) and leaf nodes (leaves), and the average depth of the trees of the entire model (depth). For the SVM model, the number of support vectors (s.v.) depends on the amount of training data.

Table 3.3: Final configurations of RF, GBDT and SVM models.

			RF				GBDT				SVM
	ft.	cl.	trees	nodes	leaves	depth	trees	nodes	leaves	depth	s.v.
IP	200	16	200	28663	28863	8,39	3200	54036	57236	5,65	1538
PU	103	9	200	33159	33359	8,63	1350	36415	55646	8	4278
KSC	176	13	200	15067	15267	8,9	3900	17883	75814	2,64	782
SV	204	16	200	23979	24179	8,46	2400	50253	52653	6,02	5413
HU	144	15	200	44597	44787	12,34	2250	67601	69851	7,31	2832

3.5 Discussion of the Results of the Models Comparison

First we will present the accuracy results for all models and images, and then we report the size and computational measurements on inference. Then, we will summarise and analyse the characteristics of each model in order to target an embedded or an on-board system.

3.5.1 Accuracy results

Figures 3.3 to 3.6 depict the accuracy evolution of each model when increasing the percentage of pixels for each class selected for training. Neural Network models always achieve high accuracy values, with the CNN model outperforming all other models, and the SVM as a kernel-based model is always the next one or even outperforming the MLP. The only behaviour out of this pattern is the high accuracy values achieved by the MLR model on the KSC data set. Except for this case, the results obtained by neural networks, kernel-based models and the other models were expected [39]. Nevertheless, it is worth to mention that, for a tree based model, GBDT achieves great accuracy values which are very close to those obtained by neural networks and the SVM, which always provide higher values than the RF, which is also a tree based model.

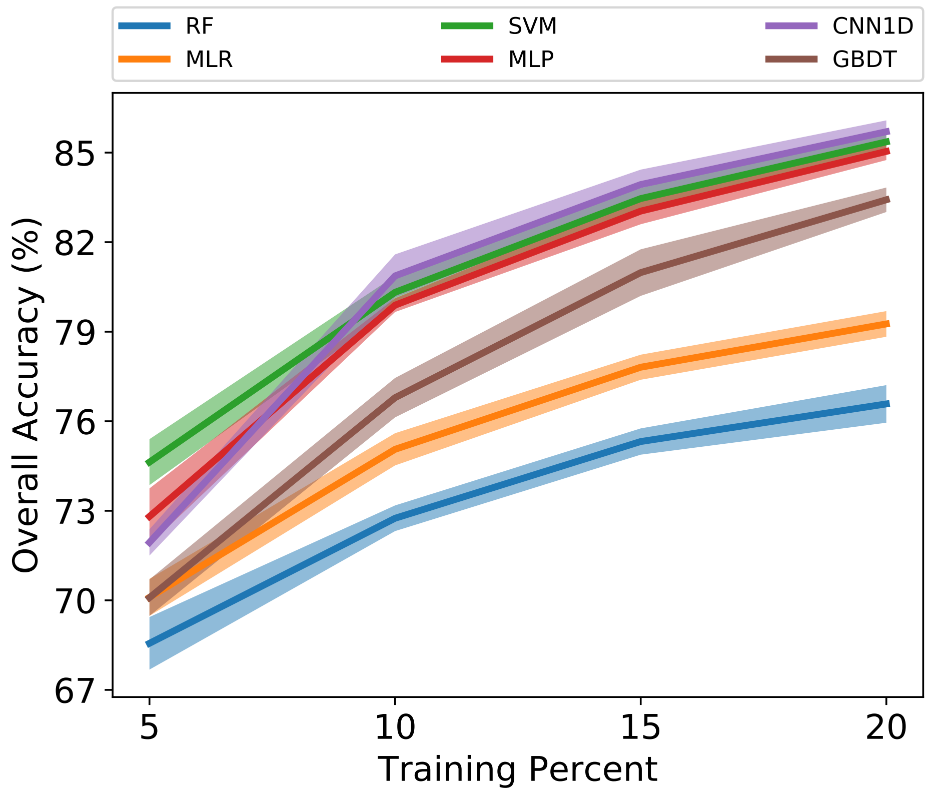


Figure 3.3: Accuracy for different training set sizes on IP.

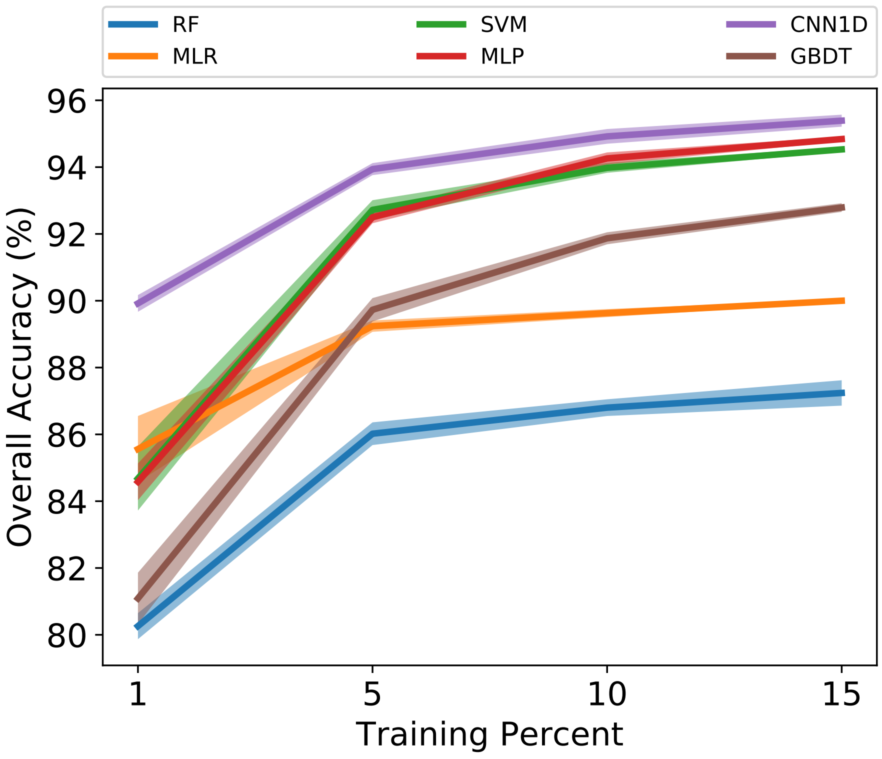


Figure 3.4: Accuracy for different training set sizes on PU.

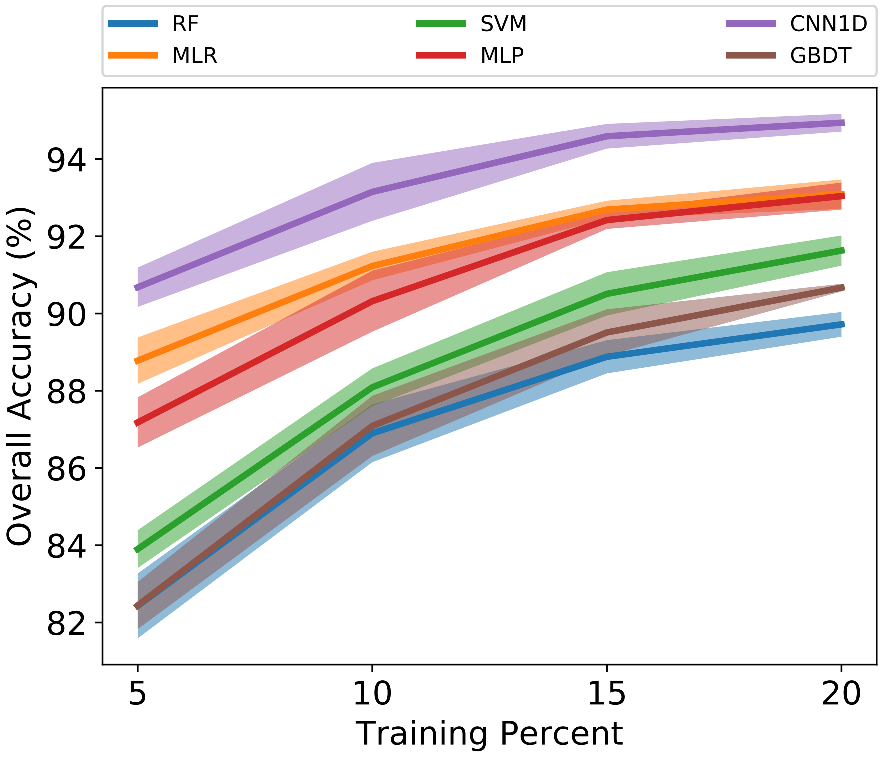


Figure 3.5: Accuracy for different training set sizes on KSC.

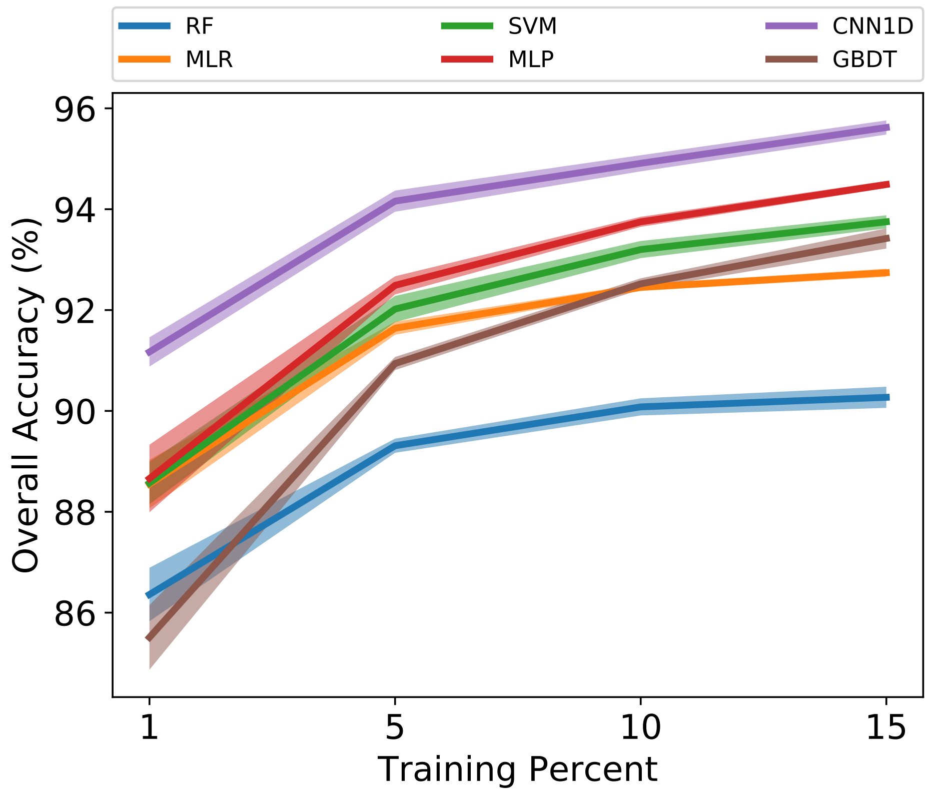


Figure 3.6: Accuracy for different training set sizes on SV.

The results obtained with the HU data set are quite particular, since it is not an entire image to work with, but two separated structures already prepared as training and testing sets. As we can observe in the values of the overall accuracy in Table 3.8, the accuracy of all models is below the score obtained for other images. But the distribution of the different models keeps the same behaviour described for the rest of the data sets, with the particularity that the MLR model outperforms the GBDT in this case.

Tables 3.4 to 3.8 show the accuracy results of the selected configurations of each model. For the IP and KSC images, the selected training set is composed of 15% of the pixels from each class, while in the PU and SV it only consists of 10% of the pixels from each class. The fixed training set for the HU image is composed of around 19% of the total pixels.

Fig. 3.7 shows the classification maps obtained for the different data sets by all the models. As we can observe, most of the classification maps have the typical *salt and pepper* effect of spectral models, i.e. classified trough individual pixels. There are some particular classes that are better modelled by certain models. For instance, the GBDT and SVM perfectly define the contour of Soil-vinyard-develop class of SV,

Table 3.4: IP data set results.

class	MLR	RF	GBDT	SVM	MLP	CNN1D
1	22.5 ± 6.71	18.0 ± 7.31	40.0 ± 5.0	40.5 ± 9.0	40.5 ± 20.27	33.5 ± 13.93
2	75.04 ± 1.17	62.73 ± 2.37	76.623 ± 1.967	80.3 ± 1.12	79.32 ± 2.6	81.52 ± 1.51
3	57.17 ± 1.76	50.14 ± 1.96	65.354 ± 1.759	70.06 ± 1.74	69.89 ± 3.22	68.07 ± 2.9
4	45.94 ± 3.64	30.5 ± 3.97	40.297 ± 2.356	67.82 ± 6.08	59.9 ± 6.03	60.99 ± 9.05
5	89.68 ± 2.32	86.18 ± 2.96	90.414 ± 1.338	93.19 ± 2.41	89.39 ± 1.79	90.27 ± 2.22
6	95.56 ± 1.5	94.78 ± 1.16	96.039 ± 1.011	95.97 ± 1.33	97.13 ± 1.41	97.39 ± 0.44
7	42.5 ± 16.75	8.33 ± 5.27	32.5 ± 23.482	71.67 ± 7.17	60.83 ± 10.07	53.33 ± 17.95
8	98.72 ± 0.42	97.74 ± 0.39	98.133 ± 0.481	97.3 ± 1.31	98.08 ± 0.57	99.16 ± 0.51
9	21.18 ± 7.98	0.0 ± 0.0	14.118 ± 8.804	47.06 ± 9.84	57.65 ± 15.96	50.59 ± 10.26
10	66.55 ± 2.76	66.31 ± 4.71	75.84 ± 4.367	75.62 ± 1.19	79.11 ± 0.44	75.38 ± 3.68
11	80.24 ± 1.27	89.08 ± 1.27	87.877 ± 1.18	84.99 ± 1.08	83.56 ± 1.32	85.05 ± 0.53
12	60.59 ± 3.36	47.96 ± 6.57	55.604 ± 1.551	76.83 ± 4.51	73.31 ± 1.97	83.25 ± 3.31
13	98.29 ± 1.02	92.8 ± 2.74	93.371 ± 1.933	98.86 ± 1.2	99.2 ± 0.28	99.2 ± 0.69
14	93.4 ± 0.64	95.61 ± 0.99	95.967 ± 0.607	94.07 ± 0.87	95.13 ± 0.3	94.89 ± 1.29
15	65.71 ± 2.06	40.91 ± 0.81	56.839 ± 2.016	64.8 ± 1.35	66.08 ± 2.68	69.06 ± 2.94
16	84.75 ± 3.1	82.5 ± 2.09	88.5 ± 3.102	87.75 ± 2.89	89.0 ± 4.77	89.0 ± 2.67
OA	77.81 ± 0.42	75.32 ± 0.44	80.982 ± 0.783	83.46 ± 0.35	83.04 ± 0.44	83.93 ± 0.5
AA	68.61 ± 1.51	60.22 ± 0.57	69.217 ± 1.627	77.92 ± 0.88	77.38 ± 2.45	76.92 ± 1.93
K(x100)	74.54 ± 0.47	71.42 ± 0.53	78.16 ± 0.897	81.08 ± 0.41	80.62 ± 0.51	81.61 ± 0.59

Table 3.5: PU data set results.

class	MLR	RF	GBDT	SVM	MLP	CNN1D
1	92.41 ± 0.86	91.35 ± 0.98	90.044 ± 0.627	93.82 ± 0.62	94.31 ± 1.09	95.37 ± 1.3
2	96.02 ± 0.21	98.25 ± 0.18	96.571 ± 0.425	98.41 ± 0.23	97.98 ± 0.39	98.16 ± 0.27
3	72.75 ± 1.13	61.51 ± 3.47	74.952 ± 1.422	78.8 ± 1.33	80.38 ± 1.12	80.55 ± 1.91
4	88.17 ± 0.74	87.2 ± 1.25	90.986 ± 1.113	93.06 ± 0.67	93.72 ± 1.02	95.43 ± 1.54
5	99.41 ± 0.3	98.43 ± 0.56	99.026 ± 0.403	98.86 ± 0.25	99.36 ± 0.48	99.8 ± 0.17
6	77.5 ± 0.72	45.2 ± 1.52	86 ± 0.837	87.97 ± 0.62	91.58 ± 1.0	92.26 ± 1.54
7	54.77 ± 4.38	75.27 ± 4.56	84.194 ± 1.245	84.58 ± 1.57	85.23 ± 2.51	89.29 ± 3.78
8	86.05 ± 0.7	88.2 ± 1.03	87.827 ± 0.805	89.67 ± 0.44	87.32 ± 1.5	88.07 ± 1.59
9	99.7 ± 0.06	99.41 ± 0.29	99.906 ± 0.088	99.53 ± 0.3	99.62 ± 0.17	99.79 ± 0.2
OA	89.63 ± 0.12	86.8 ± 0.25	91.869 ± 0.181	93.98 ± 0.15	94.26 ± 0.18	94.92 ± 0.22
AA	85.2 ± 0.54	82.76 ± 0.43	89.945 ± 0.211	91.63 ± 0.38	92.17 ± 0.16	93.19 ± 0.47
K(x100)	86.13 ± 0.17	81.98 ± 0.35	89.195 ± 0.228	91.99 ± 0.2	92.37 ± 0.24	93.25 ± 0.3

Table 3.6: KSC data set results.

class	MLR	RF	GBDT	SVM	MLP	CNN1D
1	95.92 ± 1.22	95.49 ± 1.21	95.425 ± 1.188	95.12 ± 0.77	96.23 ± 0.43	97.03 ± 1.19
2	92.27 ± 3.28	88.02 ± 2.35	86.377 ± 2.013	90.92 ± 3.56	88.89 ± 2.78	91.3 ± 4.26
3	87.25 ± 4.77	86.79 ± 2.89	86.697 ± 2.228	84.95 ± 3.46	90.92 ± 2.98	92.29 ± 1.89
4	68.09 ± 4.61	71.44 ± 4.77	64.372 ± 1.705	69.4 ± 6.88	72.74 ± 3.35	81.21 ± 8.79
5	75.18 ± 3.13	57.66 ± 5.4	55.036 ± 8.644	63.94 ± 6.95	62.04 ± 4.33	76.93 ± 5.09
6	74.97 ± 3.33	51.79 ± 3.84	61.641 ± 4.344	64.92 ± 7.66	66.87 ± 3.93	78.36 ± 5.54
7	80.67 ± 5.9	78.22 ± 4.13	82.444 ± 4.411	71.33 ± 6.06	83.33 ± 7.95	85.56 ± 8.46
8	91.77 ± 1.59	83.65 ± 2.9	86.975 ± 2.541	91.77 ± 2.62	92.7 ± 2.33	93.62 ± 3.49
9	97.01 ± 0.92	94.52 ± 2.25	93.394 ± 3.033	94.75 ± 1.59	97.6 ± 0.7	98.55 ± 0.99
10	95.99 ± 0.67	88.78 ± 0.79	93.198 ± 2.172	94.83 ± 1.5	97.44 ± 1.36	98.26 ± 0.58
11	98.1 ± 1.11	97.82 ± 1.17	94.678 ± 3.012	96.92 ± 1.59	98.26 ± 0.7	97.98 ± 0.88
12	95.09 ± 0.42	89.81 ± 1.57	93.505 ± 1.12	90.75 ± 2.81	93.83 ± 1.08	96.45 ± 1.58
13	100.0 ± 0.0	99.62 ± 0.24	99.67 ± 0.129	99.16 ± 0.46	100.0 ± 0.0	99.92 ± 0.06
OA	92.69 ± 0.23	88.88 ± 0.43	89.506 ± 0.604	90.51 ± 0.56	92.42 ± 0.23	94.59 ± 0.32
AA	88.64 ± 0.61	83.36 ± 0.83	84.109 ± 0.973	85.29 ± 1.22	87.76 ± 0.4	91.34 ± 0.59
K(x100)	91.86 ± 0.26	87.61 ± 0.48	88.308 ± 0.674	89.43 ± 0.62	91.55 ± 0.26	93.97 ± 0.35

Table 3.7: SV data set results.

class	MLR	RF	GBDT	SVM	MLP	CNN1D
1	99.19 ± 0.47	99.64 ± 0.15	99.514 ± 0.289	99.44 ± 0.36	99.64 ± 0.46	99.87 ± 0.17
2	99.93 ± 0.06	99.86 ± 0.09	99.827 ± 0.074	99.72 ± 0.15	99.83 ± 0.21	99.86 ± 0.22
3	98.85 ± 0.29	99.08 ± 0.53	98.775 ± 0.37	99.51 ± 0.12	99.47 ± 0.2	99.63 ± 0.25
4	99.39 ± 0.31	99.54 ± 0.27	99.554 ± 0.26	99.59 ± 0.11	99.62 ± 0.12	99.46 ± 0.06
5	99.19 ± 0.26	97.96 ± 0.49	98.109 ± 0.332	98.71 ± 0.51	99.11 ± 0.32	99.0 ± 0.36
6	99.94 ± 0.03	99.72 ± 0.11	99.624 ± 0.292	99.78 ± 0.12	99.84 ± 0.09	99.9 ± 0.07
7	99.74 ± 0.07	99.34 ± 0.18	99.559 ± 0.164	99.61 ± 0.16	99.71 ± 0.12	99.7 ± 0.1
8	88.07 ± 0.11	84.26 ± 0.42	85.507 ± 0.349	89.11 ± 0.34	88.84 ± 0.7	90.36 ± 1.01
9	99.79 ± 0.07	99.01 ± 0.24	99.219 ± 0.165	99.66 ± 0.21	99.88 ± 0.07	99.85 ± 0.13
10	96.34 ± 0.52	91.35 ± 0.61	93.473 ± 0.737	95.28 ± 0.87	96.32 ± 0.79	97.63 ± 0.57
11	96.9 ± 1.0	94.2 ± 1.23	94.782 ± 0.532	98.0 ± 0.71	97.75 ± 0.76	98.42 ± 0.85
12	99.79 ± 0.03	98.42 ± 0.67	99.239 ± 0.507	99.55 ± 0.34	99.8 ± 0.11	99.93 ± 0.11
13	99.05 ± 0.33	98.08 ± 0.67	97.818 ± 0.957	98.5 ± 0.52	98.55 ± 0.7	99.25 ± 0.56
14	95.89 ± 0.17	91.48 ± 1.29	95.202 ± 0.997	95.02 ± 0.81	98.17 ± 0.55	98.05 ± 0.85
15	66.85 ± 0.18	60.42 ± 1.32	74.91 ± 0.524	71.72 ± 0.69	74.61 ± 1.66	80.02 ± 2.35
16	98.45 ± 0.36	97.31 ± 0.19	97.406 ± 1.017	98.35 ± 0.17	98.7 ± 0.72	98.94 ± 0.53
OA	92.45 ± 0.07	90.08 ± 0.17	92.544 ± 0.079	93.2 ± 0.17	93.75 ± 0.1	94.91 ± 0.16
AA	96.09 ± 0.1	94.35 ± 0.16	95.782 ± 0.056	96.35 ± 0.15	96.86 ± 0.02	97.49 ± 0.15
K(x100)	91.58 ± 0.07	88.93 ± 0.19	91.696 ± 0.087	92.42 ± 0.19	93.03 ± 0.11	94.33 ± 0.18

Table 3.8: HU data set results.

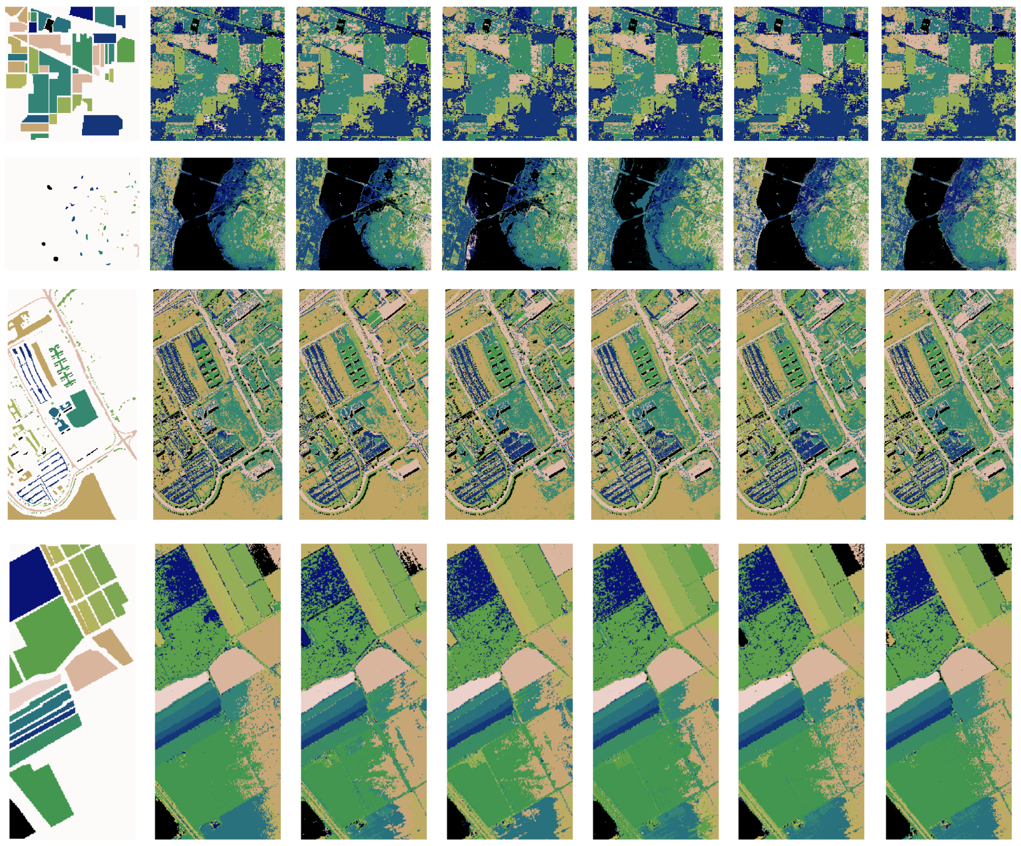
class	MLR	RF	GBDT	SVM	MLP	CNN1D
1	82.24 ± 0.35	82.53 ± 0.06	82.336 ± 0.0	82.24 ± 0.0	81.29 ± 0.32	82.55 ± 0.54
2	81.75 ± 1.13	83.31 ± 0.26	83.177 ± 0.0	80.55 ± 0.0	82.12 ± 1.18	86.9 ± 3.87
3	99.49 ± 0.2	97.94 ± 0.1	97.228 ± 0.0	100.0 ± 0.0	99.6 ± 0.13	99.88 ± 0.16
4	90.81 ± 3.67	91.59 ± 0.15	94.981 ± 0.0	92.52 ± 0.0	88.92 ± 0.55	92.8 ± 3.34
5	96.88 ± 0.08	96.84 ± 0.13	93.277 ± 0.0	98.39 ± 0.0	97.35 ± 0.32	98.88 ± 0.3
6	94.27 ± 0.28	98.88 ± 0.34	90.21 ± 0.0	95.1 ± 0.0	94.55 ± 0.28	95.94 ± 1.68
7	71.88 ± 1.26	75.24 ± 0.15	73.414 ± 0.0	76.31 ± 0.0	76.03 ± 1.74	86.49 ± 1.29
8	61.8 ± 0.68	33.2 ± 0.15	35.138 ± 0.0	39.13 ± 0.0	64.27 ± 9.17	78.02 ± 6.6
9	64.82 ± 0.23	69.07 ± 0.4	68.839 ± 0.0	73.84 ± 0.0	75.09 ± 2.27	78.7 ± 4.31
10	46.18 ± 0.35	43.59 ± 0.31	41.699 ± 0.0	51.93 ± 0.0	47.28 ± 1.07	68.22 ± 10.41
11	73.51 ± 0.33	69.94 ± 0.16	72.391 ± 0.0	78.65 ± 0.0	76.11 ± 1.07	82.13 ± 1.62
12	67.74 ± 0.26	54.62 ± 0.8	69.164 ± 0.0	69.03 ± 0.05	72.93 ± 3.56	90.85 ± 2.57
13	69.75 ± 0.72	60.0 ± 0.59	67.018 ± 0.0	69.47 ± 0.0	72.28 ± 3.6	74.67 ± 3.49
14	99.35 ± 0.49	99.27 ± 0.47	99.595 ± 0.0	100.0 ± 0.0	99.35 ± 0.41	99.11 ± 0.3
15	94.38 ± 0.89	97.59 ± 0.32	95.137 ± 0.0	98.1 ± 0.0	98.1 ± 0.48	98.48 ± 0.16
OA	76.35 ± 0.27	73.0 ± 0.07	74.182 ± 0.0	76.96 ± 0.0	78.61 ± 0.44	85.95 ± 0.94
AA	79.66 ± 0.2	76.91 ± 0.06	77.573 ± 0.0	80.35 ± 0.0	81.68 ± 0.24	87.58 ± 0.8
K(x100)	74.51 ± 0.28	70.99 ± 0.07	72.101 ± 0.0	75.21 ± 0.0	76.96 ± 0.47	84.77 ± 1.02

while CNN1D exhibits a very good behaviour on the cloudy zone in the right side of the HU data set, and both tree based models (RF and GBDT) perform very well on the swampy area on the right side of the river in the KSC data set. Nevertheless, the most significant conclusion that can be derived from these class maps is that the different errors of each model are distributed in a similar way along classes for each model, as it can be seen on Tables 3.4 to 3.8, but here we can confirm that it is consistent for the entire classification map. In general, all the classification maps are quite similar and well defined in terms of the contours, and the main classes are properly classified. We can conclude that the obtained accuracy levels are satisfactory, and the errors are well distributed, without significant deviations due to a particular class nor significant overfitting of the models.

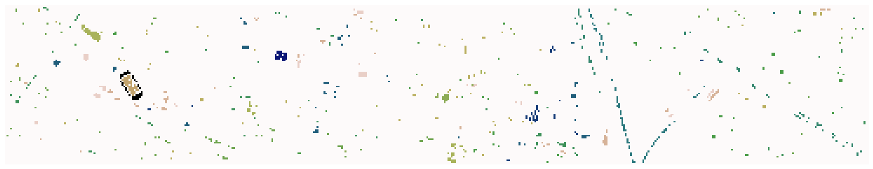
3.5.2 Size and computational measurements

To perform the characterisation of each algorithm in inference it is necessary to analyse their structure and operation. The operation during inference of every model has been explained in Section 3.3, and the final sizes and configurations of the trained models after cross-validation for parameter selection has been detailed in Section 3.4. Fig. 3.8 reports the sizes in Bytes of the trained models, while Fig. 3.9 shows the number and type of operations performed during the inference stage.

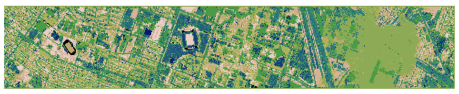
It is very important to remark that these measurements have been realised theoretically, based on the described operations and model configurations. For instance, the



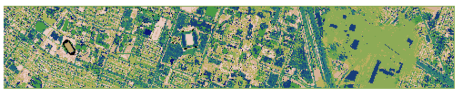
(a) GT (b) MLR (c) RF (d) GBDT (e) SVM (f) MLP (g) CNN1D



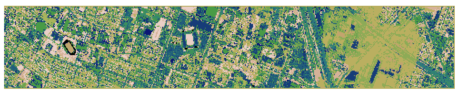
(a) GT



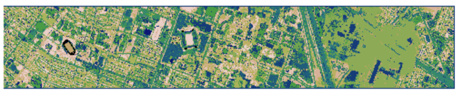
(b) MLR



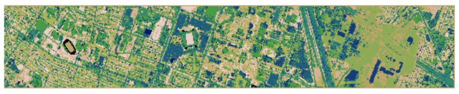
(c) RF



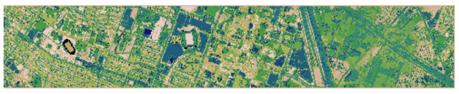
(d) GBDT



(e) SVM



(f) MLP



(g) CNN1D

Figure 3.7: Classification maps obtained for each dataset by the different models

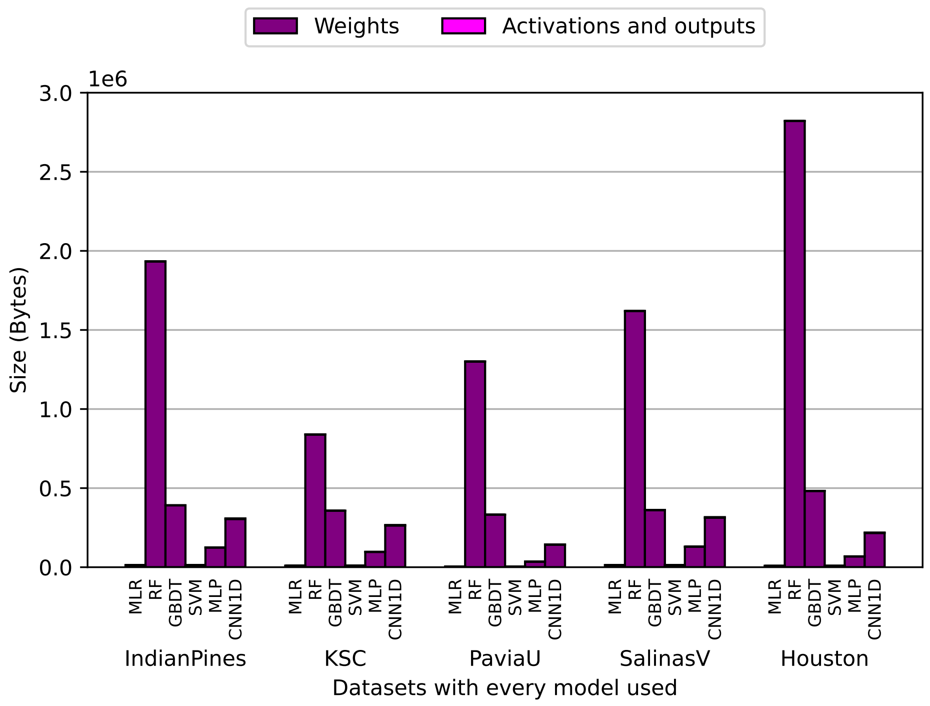


Figure 3.8: Size of the trained models in Bytes.

size measurements do not correspond to the size of a file with the model dumped on it, which is software-dependent, i.e. depends on the data structures and it uses to keep much more information for the framework than the actual learned parameters needed for inference. As a result, Fig. 3.8 shows the theoretical size required in memory to store all the necessary structures for inference, based on the analysis of the models, exactly as it would be developed for a specific hardware accelerator or an embedded system.

As we can observe, the size of RF models is one order of magnitude bigger than the others. This is due to their need to save the values of the predictions for every class on each leaf node. This is a huge amount of information, even compared to models that train an entire estimator for each class, like GBDT. Actually, the size of MLR and SVM models is one order of magnitude smaller than GBDT, MLP and CNN1D models. Nevertheless, all the models (except the RF) are below 500 kilobytes, which makes them very affordable even for small low-power embedded devices.

In a similar way, the operational measurements shown on Fig. 3.9 are based on the analysis of each algorithm, not in terms of software executions (that depend on the architecture, the system and the framework), and they are divided into four groups according to their computational complexity. The only models that use integers for the inference computations are the decision trees, and they only need integer

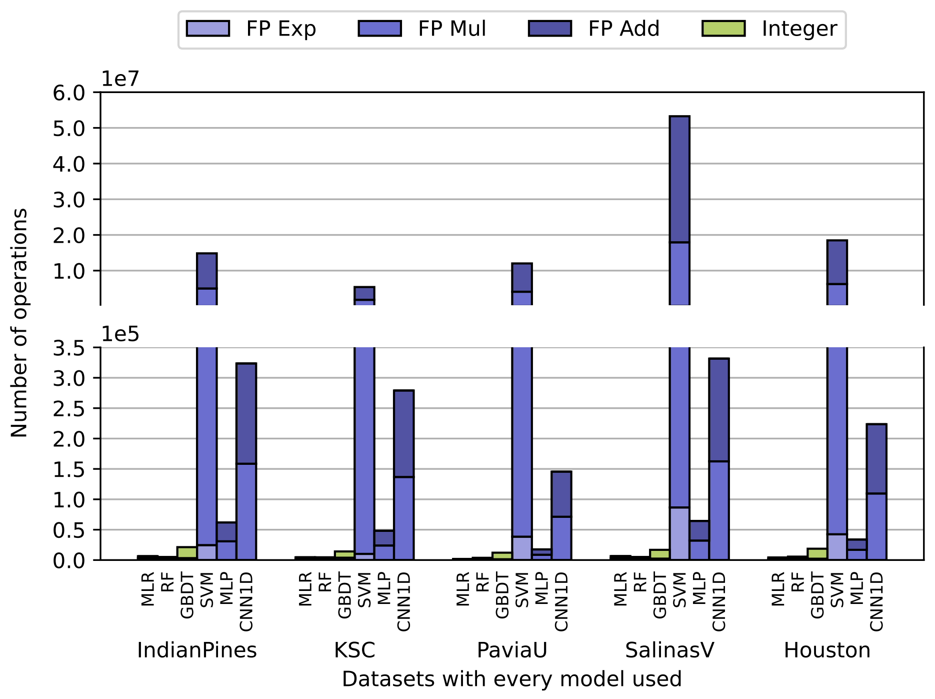


Figure 3.9: Number of operations performed during the inference stage.

comparisons. Floating point operations are the most common in the rest of the models, but they are also divided into three different categories. *FP Add* refers to accumulations, subtractions and comparisons, which can be performed on an adder and are less complex, *FP Mul* refers to multiplications and divisions, and *FP Exp* are exponential which are only performed by the SVM model. High-performance processors include powerful floating point arithmetic units, but for low-power processors and embedded devices, these computations can be very expensive.

Focusing on operations, the SVM model is two or even three orders of magnitude larger than the other models. Moreover, most of their operations are floating point multiplications and additions, but it also requires a great amount of complex operations such as exponential ones. In most of the data sets, it requires more exponential operations that the entire number of operations of the other models, except for the CNN. The number of operations required by the CNN model is one order of magnitude higher than the rest of the models, and it is basically composed of floating point multiplications and accumulations. MLR and RF models are the ones that require less operations during inference, while GBDT and MLP require several times the number of operations of the latter, sometimes even one order of magnitude more.

3.5.3 Characteristics of the models in relation to the results

In this section, we will review the characteristics of every model in relation to this results. RF and GBDT models are composed of binary trees. The number of trees of each model are decided in training time according to the results of the cross-validation methods explained above. The non-leaf nodes of each tree keep the value of the threshold and the number of features to compare with, which are integer values, while the leaf nodes keep the prediction value, which is a float. In the case of RF, leaf nodes keep the prediction for every class, which makes them very big models. Although these models are not the smallest, during inference they do not need to operate with the entire system; they just need to take the selected path of each tree. In terms of operations, each non-leaf node of a selected path implies an integer comparison, while the reached leaf node implies a float addition.

Notice that addressing operations, such as using the number of features to address the corresponding feature value, are not taken into account and are not considered in Fig. 3.9. The same occurs for the rest of the models, assuming that every computational operation needs its related addressing, so the comparison is fair.

The MLR model only requires during inference one float structure of the same size and shape as the entry, i.e one hyperspectral pixel, for each class. The operations realised are the dot product of the input and these structures and the result of each one of them is the prediction for the corresponding class.

The SVM model is small, in the same order of magnitude than the MLR, because it only needs the support vectors and the constants, some of which can be already precalculated together in just one value. But, in terms of computation, the calculation of Eq. (3.6) requires an enormous amount of operations compared to the rest of the methods.

The size and number of operations of the MLP model depends on the number of neurons in the hidden layer and the number of classes. For each neuron, there is a float structure of the same size and shape of the entry, and then for each class there is a float structure of the same size and shape of the result of the hidden layer. The operations realised correspond to all these dot products.

In the case of the CNN, the size corresponds to the filters of the convolutional layer and then the structures corresponding to the MLP at the end of the model, but this MLP is much bigger than the MLP model, because its entry is the output of the convolutional layer, which is much bigger than the original input pixel. The main difference with the MLP model (in terms of operations) lies on the behaviour of the convolutional layer. It requires a dot product between each filter and the corresponding part of the input for each step of the convolutional filters across the entire input. This model also has a max pooling layer that slightly reduces the size of the model, because it is supposed to be executed on the fly, but adds some extra comparisons to the operations.

Since embedded or on-board systems require small, efficient models, we analyse the trade-off between the hardware requirements of each model and its accuracy results. In summary, neural networks and SVMs are very accurate models, and they do not have great memory requirements, but they require a great amount of floating point operations during inference. Furthermore, most of them are multiplications or other operations which are very expensive in terms of resources. Hence, they are the best option when using high-performance processors, but they may not be suitable for low-power processors or embedded systems. In the case of the RF, the number of operations is really small, and most of them are just integer comparisons, but the size of the model is very big compared to the other models, and it also achieves the lowest accuracy values.

According to our comparison, it seems that the best trade-off is obtained for MLR and GBDT models. Both models are reasonably small for embedded systems and require very few operations during inference. GBDT is bigger, but it still has very small dimensions. In terms of operations, even if GBDT needs to perform some more operations than the MLR, its important to remark that MLR operations are floating point multiplications and additions, while most of the GBDT operations are integer comparisons, which makes them a perfect target for on-board and embedded systems. In terms of accuracy, GBDT achieves better values in most scenarios.

3.6 Conclusion

In this chapter, we show the analysis of the size and operations during inference of several state-of-the-art machine learning techniques applied to hyperspectral image classification to characterise them in terms of energy consumption and hardware requirements for their implementation in embedded systems or on-board devices. These are the main observations:

- In terms of accuracy, neural networks and kernel-based methods (such as SVMs) usually achieve higher values than the rest of the methods, while the RF obtains the lowest values on every data set. The behaviour of the MLR model is not very robust, obtaining high accuracy in some data sets and low values in others. The GBDT model always achieves higher accuracy than the RF and also gets very close to the accuracies obtained by some of the SVMs and neural networks.

- Regarding the size of the trained models, most of them are reasonably small to fit into embedded and reconfigurable small devices, except for the RF that is one order of magnitude bigger than the rest of the models. The SVM and MLR models are specially small, in some cases even one order of magnitude less than the size of the CNN, the MLP and the GBDT.

- Regarding the number and type of operations needed during inference, the RF and GBDT models clearly stand out from the rest (not only because they need very few operations during inference, but specially because most of these operations are integer comparisons). The rest of the models need floating point operations, and most of them are multiplications, which are more expensive in terms of hardware resources and power consumption. Even when some models (such as MLR and MLP) need few operations to perform the inference, the type of operations are not the most suitable for low-power embedded devices.

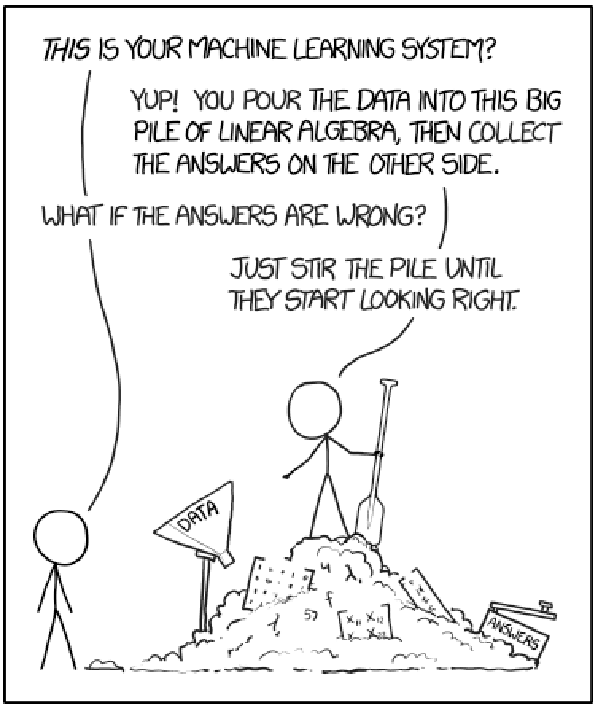
- Neural networks and SVMs, in turn, are very expensive in terms of computations (not only in terms of quantity, but also regarding the type of operations they perform). As a result, for small energy-aware embedded systems, they do not represent the best choice. Depending on the specific characteristics of the target device and the accuracy requirements of the addressed problem, an MLP could be an interesting option. The RF model is very big for an embedded system and it generally achieves low accuracy values.

- The MLR is one of the smallest models, and it also performs very few operations during inference. Nevertheless, even though the number of operations is small, they are expensive operations because it is entirely based on floating point additions and multiplications. Furthermore, it achieves high accuracy values in some data sets but low values in others, so its behaviour is very dependent on the data set characteristics. If it adapts well to the target problem, it can be a good choice depending on the embedded system characteristics.

- From our experimental assessment, we can conclude that GBDTs present a very interesting trade-off between the use of computational and hardware resources and the obtained accuracy levels. They perform very well in terms of accuracy, achieving in many cases better results than the other techniques not based in kernels or neurons, i.e. RF and MLR, while they use less computational resources than the techniques based on kernels or neurons, i.e. SVM, MLP and CNN. Moreover, most of their operations during inference are integer comparisons, which can be efficiently calculated even by very simple low-power processors, so they represent a good option for an embedded on-board system.

In the next two chapters we will present two accelerators, one for DNNs, as they are the most accurate technique, and one for GBDTs, as they represent the best trade-off between efficiency and accuracy found in this study.

4 Step I: Sparse Convolutional Neural Networks Accelerator


THIS IS YOUR MACHINE LEARNING SYSTEM?
YUP! YOU POUR THE DATA INTO THIS BIG PILE OF LINEAR ALGEBRA, THEN COLLECT THE ANSWERS ON THE OTHER SIDE.
WHAT IF THE ANSWERS ARE WRONG?
JUST STIR THE PILE UNTIL THEY START LOOKING RIGHT.
DATA
ANSWERS

— **xkcd.com**
(Just that guy, you know?)

4.1 Introduction

Deep neural networks (DNNs) have emerged as an outstanding model to solve complex problems in a wide variety of fields, such as computer vision, speech recognition, natural language processing, or audio recognition.

Convolutional neural networks (CNNs) are one of the most popular DNN models. Their core consists of several convolutional layers where the input, called activation or feature map, is convolved with a set of filters. The number of layers and filters, and the size of the activations turn out these models into computationally-intensive and memory-demanding tasks. In computer vision, state-of-the-art CNNs require many millions of multiply-and-accumulate (MAC) operations to process a single image. High-performance general purpose processors (CPUs) and graphics processing units (GPUs) have been the natural target to execute these models on non-battery-dependent systems. However, more energy-efficient architectures are needed for embedded systems.

Previous works have demonstrated that there are several strategies to reduce the computational and memory requirements of CNNs, with minimal impact in accuracy. Regarding the arithmetic, it is feasible to move from floating point to fixed point [68, 58, 60], and work with lower bitwidth [72, 64, 63]. Other works propose to speedup the MAC operations by using approximated outputs [38, 37]. Regarding the size of the models, pruning techniques force many parameters to zero, enabling data compression and reducing the number of useful MAC operations. This approach enables drastic reduction, both in model size and computational workload, of DNNs, which is essential for embedded systems.

Deep Compression [72] constitutes a good reference for these techniques. Their authors achieve network size reductions from 35x to 49x while preserving accuracy in several popular DNNs (*AlexNet*, *VGG*, and *LeNet*) by using pruning, quantization and compression. Another relevant reference is *SqueezeNet* [73]. In this study, the authors present a CNN with 50x fewer parameters than *AlexNet* with no loss in accuracy. Again, this result is achieved by using pruning and compression techniques.

Pruning techniques turn the CNNs into sparse models, and this sparsity can be exploited by including specific support to avoid useless operations, i.e., those ones where at least an operand is zero.

Although the benefits of these techniques are proved, they also introduce some challenges that must be carefully addressed in order to minimise their overhead. Compression requires support to manage the irregularities that arises in addressing and sparsity demands hardware to identify useful operations. Moreover, sparsity generates random memory-access patterns that can significantly degrade the performance of a parallel architecture due to the memory conflicts.

We have designed an accelerator that takes advantage of the optimisation opportunities offered by sparsity in DNNs. Our accelerator works with compressed filters, performs only useful operations, and retrieves only useful values from memory by identifying those operations where both operands are different from zero. To this end, we included an additional data structure, called indices tensor, consisting of a single bit per element that indicates whether that element is zero or not. This structure allows intelligent recognition of useful operations using simple bit operations, and compression of data by not storing values that are zero. Moreover, we have included specific support to reduce the memory conflicts, and we have explored the trade-offs in performance, area, and energy efficiency.

The register-transfer level design of our accelerator has been written in VHDL and has been implemented in a Xilinx Zynq UltraScale+ FPGA. We believe that reconfigurable FPGAs are the natural target for our study since in these platforms it is possible to use the same hardware resources to implement a conventional CNN accelerator or an accelerator with specific support for sparsity. We have taken both performance and power measures running two popular CNNs: *AlexNet* and *SqueezeNet*. We selected these networks as benchmarks because pruned models are available for the community [70, 69]. In addition, we have used synthetic data to characterise the behaviour of our accelerator for different sparsity ranges.

The rest of this chapter is organised as follows. Section 4.2 is an overview of the related work and Section 4.3 presents our contributions. Section 4.4 describes the compression format used. Section 4.5 introduces the procedure to identify the useful operations. Section 4.6 describes the initial architecture used as baseline and Section 4.7 explains the additional architectural support included to efficiently avoid the useless operations and reduce the memory conflicts. Section 4.8 describes the memory hierarchy and the data flow. Section 4.9 analyses the scalability of the proposed architecture. Finally, Section 4.10 evaluates the experimental results and Section 4.11 presents our final conclusions.

The content of this chapter has been published in an international journal [22] and the correspondent code is available in a public repository [44].

4.2 Related work

A recent survey [48] analyses the state of the art and future directions of DNN support in ASICs and FPGAs. This survey identifies the exploitation of sparse data as a powerful technique for reducing computational load and memory requirements in DNNs. Another survey [52] states that "there is an emerging need for the CNN-to-FPGA tools to support compressed and sparse". Although low or medium values of sparsity can be found in any model, pruning techniques are essential to build highly-sparse models. Pruning was originally proposed in [108, 107], and, recently, many other techniques has been proposed and studied [77, 61, 65]. These techniques consist in an iterative process that first identifies those weights that can be set to zero and then fine-tunes the remaining weights. Pruned models keep a similar accuracy than the original models with fewer meaningful parameters, allowing sparsity-based optimisations. However, an efficient management of sparsity must overcome the loss of regularity that arises in memory accesses. Otherwise, trying to exploit sparsity can even negatively impact in performance as explained in [62]. [31] proposes a technique to alleviate this by applying a bank-balanced pruning method designed to optimise the parallel execution of the pruned model.

Some recent works have presented custom architectures for sparse DNNs. *Cnvolutin* was the first accelerator to partially avoid useless operations [66]. The authors proposed a technique to skip those operations where the value of the input activation is zero. Han *et al.* presented "EIE: Efficient Inference Engine on Compressed Deep Neural Network" [71]. This engine manages compressed weights, and includes hardware support to compute only useful operations in fully-connected layers. *UCNN* proposes a factorisation technique to replace multiplications with additions, and takes advantage of filter sparsity [47], while SqueezeFlow [35] proposes a technique that transforms a sparse convolution into multiple effective and ineffective subconvolutions. After that the ineffective subconvolutions can be eliminated. *Cambricon-X* [74] and *NullHop* [43] are other accelerators for sparse CNNs. The first one exploits sparsity on filters but not on activations, whereas the second exploits sparsity on activations but not on the filters.

SCNN is a high-performance oriented accelerator for CNNs [59]. It includes several processors, and each of them computes a convolution in parallel by computing the Cartesian product. This is a very powerful approach that allows data reuse and avoids any operation where an operand is zero. However, this architecture requires large buffers to store partial results, and crossbars to link the multipliers and the buffers. As a result, their support for sparsity increases the area of their chip by 34%, even with a 50% smaller on-chip activation memory than their dense baseline. In terms of performance, they achieve speedups from 2.19 to 3.52 for several popular CNNs.

Other relevant works are *Eyeriss* [54] and *Eyeriss v2* [32] which are scalable architectures with hundreds of MAC units. *Eyeriss* proposes to identify when any of the operands is zero in order to gate the data path and save energy. *Eyeriss v2* compress data in sparse column (CSC) format and directly operates with the compressed data. With this approach they only read non-zero activation values. Then they look for the corresponding weights. If the weights are zero they do not carry out the computations in order to save energy. The only penalty is that this may generate some bubbles in the pipeline.

As a summary, only three previous designs include support to avoid all the operations where at least one of the operands is zero: *EIE*, designed for fully-connected layers, *SCNN*, designed for convolutional layers, and *Eyeriss v2* that can deal with both of them. Although our design supports both fully-connected and convolutional layers, we have mainly focused on convolutional layers because they are much more computationally intensive. Both *SCNN* and *Eyeriss v2* are highly parallel systems designed for high performance and include a large amount of hardware resources whereas our design is designed for embedded systems, where hardware resources and power budget are very limited. Hence, our primary goal is efficiency. In fact, our architecture reaches almost peak performance in most situations, i.e., one useful operation per clock cycle in each MAC processor, whereas *SCNN* is very far from peak performance on highly-sparse layers. For instance, when processing convolutional layers four and five in *AlexNet*, *SCNN* only reaches ∼25% peak performance [59] whereas for the same convolutions our design reaches ∼98%. Regarding *Eyeriss v2*, according to their results during the execution of *AlexNet*, one of our MAC processors carries out twice the number of operations per cycle than one of their MAC processors. This does not mean that our design is better; we just target different problems. They try to maximise the throughput using hundreds of MACs in parallel, whereas our design tries to maximise the performance of an embedded system with few MAC processors.

The evaluation methodology in all these previous works quantifies the gains in terms of performance and energy efficiency of their sparse architectures compared to dense architectures with the same arithmetic resources. This analysis is interesting, but it is not completely fair because the sparse architectures include more hardware resources than the corresponding dense architectures. In our experiments we compare architectures with similar area. To this end we include additional arithmetic resources in the dense architecture. With this approach, for a given scenario, we can identify whether is better to use hardware resources to exploit sparsity or to use them to carry out more MAC operations in parallel.

Another limitation of the previous approaches is that they use on-house high-level simulators to gather the performance metrics. Of course the authors have tried to develop accurate simulators, but it is impossible to know if they are 100% accurate since they have to model not only the accelerators, but also communications, and memory accesses. In our case, instead of using a simulator, we have implemented our design and we have measured our performance metrics during actual executions to guarantee that they are completely accurate.

4.3 Contributions

- We have designed a sparse architecture that is able to avoid all useless operations and manage filter compression both for convolutional and fully-connected layers. It also includes support to reduce the impact of the memory-bank conflicts due to the non-uniform memory access patterns. The design has been pipelined to improve the performance. Our architecture has been designed for embedded systems, and its objective is to maximise performance and reduce the energy consumption on systems with limited resources. It has been written in VHDL and is available for the community in a GitLab repository [44].
- We propose a dense/sparse evaluation methodology that attempts to compare architectures with similar area resources. To this end, we have designed a dense architecture with parameterizable arithmetic resources and for each comparison we select the dense architecture most similar in area to the sparse design.
- We have implemented both designs on an FPGA and taken performance and power consumption measurements. With this approach we can identify the trade-offs between sparse and dense architectures, and identify which architecture is better for a given scenario.

4.4 Data compression

Some popular compression formats for sparse CNNs are run length encoding (RLE), compressed sparse row (CSR) or compressed sparse column (CSC) [99]. The main idea of RLE is to store consecutive elements of the same value as a single value and the count of the repetitions, while CSC and CSR consist of two data sets: one that stores only those values that are not zero, and one that stores metadata to infer the remaining information and to calculate the addresses.

Instead of these formats, we propose to use a tensor with the same dimensions than the uncompressed data that stores a single bit per element pointing out whether they are zero or not. Figure 4.1 shows an example of these formats applied to matrices.

For the comparison we used the variation of RLE format proposed in [54], using 5 bits to codify the count of consecutive zeros. Regarding to the CSC/CSR formats, they are symmetric structures, and using one or the other will be better depending on the selected matrix representation. In our case CSR, using column indices and row pointers, is the one that reaches better results. Column indices store the column of each non-zero value, and row pointers point out the first non-zero value of each row. Its last value is the total number of non-zero elements.

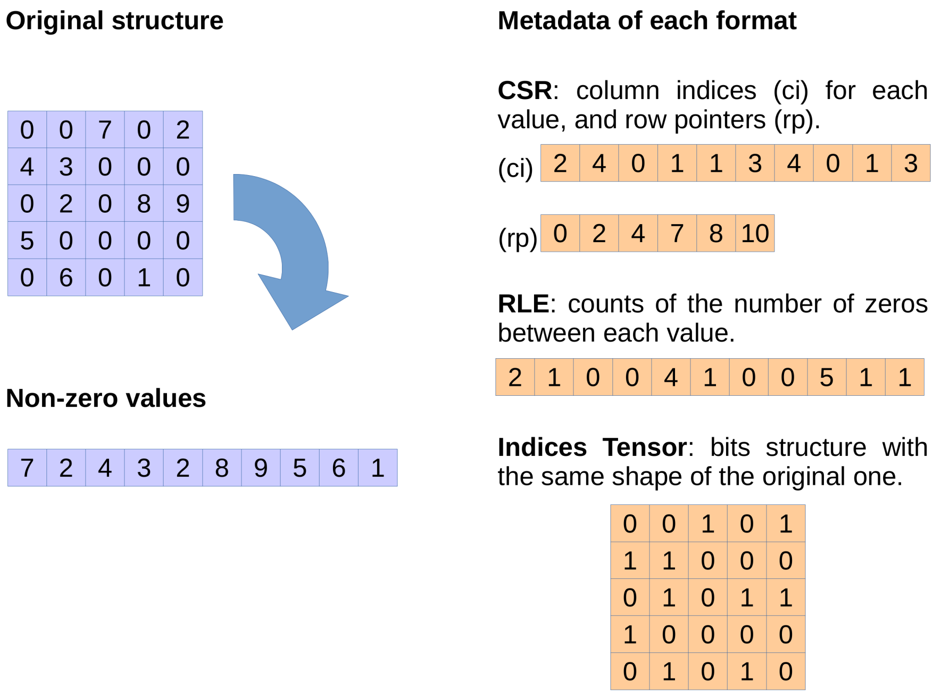


Figure 4.1: Compression formats example.

Fig. 4.2 shows how these compression formats perform as a function of the sparsity. It includes two boundaries for each format. Upper and lower bound in CSC/CSR are calculated on matrices of 9×512 with 8-bit values and 1×16 with 32-bit values, respectively, based on the sizes of the activations analysed. We have chosen the indices tensor format for two reasons: first, it allows hardware-friendly identification of those operations that are useful (further discussed in section 4.5). Second, it yields higher compression ratios than the other formats for most scenarios and, in those where it under-performs, size becomes negligible.

In our architecture, filters are compressed in order to reduce the size of the models. It is also possible to compress the activation during inference, but it demands additional resources, as it is generated on the fly. Moreover, since the compression ratio is unknown at design time, memory resources should be allocated for the worst case, so the benefits are limited and do not justify the overhead. Our design

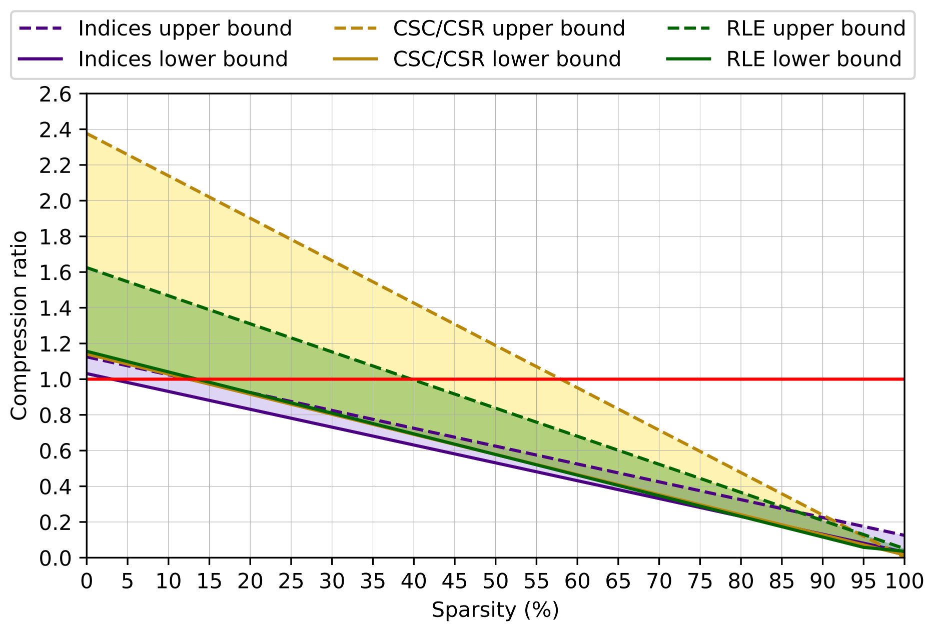


Figure 4.2: Compression ratio comparisons.

leverages the indices tensor also for the activation in order to take full advantage of this compression format to identify useful operations, at the expense of a small overhead.

4.5 Identifying useful operations

Our design avoids any useless operation in order to reduce memory accesses and arithmetic computation. An efficient identification of those useful operations relies on the compression format discussed previously. First, sections of the indices tensor of the filter and the activation are fetched. Second, they are matched through a bitwise *and* operation to find all the pairs with two non-zero values. The main steps of this process are described in Algorithm 4.1. *Activation_offset* stores the position in the section where the current pair has been found. This information is used to calculate the address of the activation value needed. *Filter_offset* indicates the number of non-zero filter values that have been skipped since the last pair found. It is required to calculate the address of the filter value as filters are compressed and only non-zero values are stored. Finally, when the last pair of the fetched sections is processed, *remaining_filter_offset* keeps track of the number of remaining non-zero filter values in the section. This number must be taken into account to compute the next filter address.

```
# Get next filter and activation sections
filter_section = get_filter_section(SECTION_SIZE)
activation_section = get_activation_section(SECTION_SIZE)
# Start mask
mask = ones_vector(SECTION_SIZE)
# Search for the first pair
match, pos, last_pair = next_pair(filter_section,
                                  activation_section)
if match:
    while match:
        # Apply mask
        filter_section = bitwise_and(filter_section, mask)
        activation_section = bitwise_and(activation_section, mask)
        # Actualise activation and filter offsets
        filter_offset = sum(filter_section[:pos])
        activation_offset = pos
        # Actualise mask
        for i in range(pos):
            mask[i] = 0
        if last_pair:
            # Actualise remaining filter offset
            remaining_filter_offset = sum(filter_section[pos+1:])
        else:
            # Search for the next pair
            match, pos, last_pair = next_pair(filter_section,
                                              activation_section)
else:
    # Default output values
    activation_offset = SECTION_SIZE
    remaining_filter_offset = sum(filter_section)
```

Algorithm 4.1: Identification of useful operations within a section.

Notice that the *next_pair* function on this pseudo-code algorithm will only return on match, activating the *last* flag if it is the last one, so the only situation in which *match* flag will be false is when there is no matches at all in the entire section, and the *else* clause is just there for this situations. After finishing with one section, the entire algorithm will be repeated. Fig. 4.3 illustrates this process with a conceptual example. The algorithm iteratively looks for pairs of non-zero values located in the same section position. Notice that these non-zero values are marked as 1. In the first iteration, the first pair is found on the section element #3. Hence, *activation_offset* is 3. *Filter_offset* is 1 since one non-zero filter value has been skipped. Finally, processed elements (from #0 to #3) are masked. In the second iteration, the same procedure applies to the unmasked elements. Notice that the value of *filter_offset* does not take on account the already masked bits. Additionally, *last_pair* flag will be active, as the last pair has been found, so *remaining_filter_offset* reports that an additional filter value must be skipped.

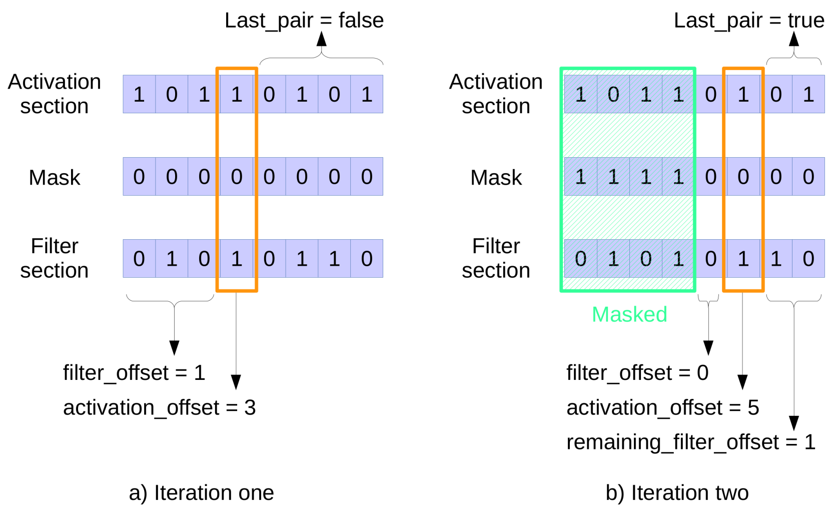


Figure 4.3: Identification of useful operations.

Fig. 4.4 depicts the hardware unit responsible for this operations. *Bitwise unit* carries out bitwise *and* operations on the filter and activation sections, and the masks generated by *Mask composer*. They are both required to identify useful operations and compute the filter offsets. The *Priority encoder* encodes the activation offset, i.e., the position where the current pair has been found. Finally, two tree adders return the filter offsets.

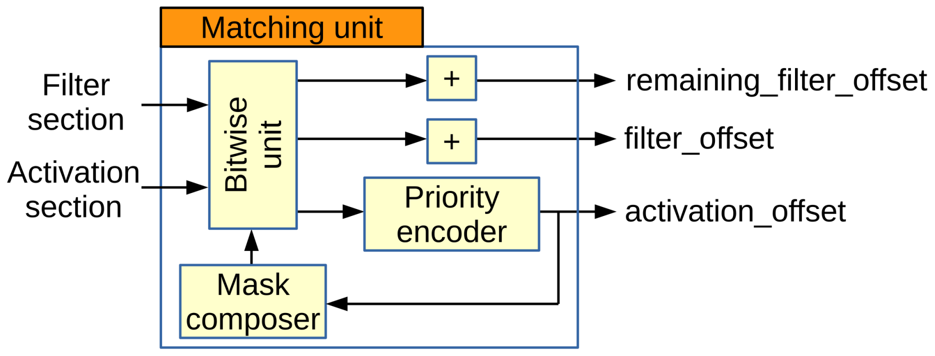


Figure 4.4: Matching unit.

4.6 Baseline: dense architecture

Assessing the benefits of exploiting sparsity in CNNs requires a baseline to compare with. We designed a dense accelerator for embedded systems able to exploit inter-filter parallelism, through *N* processing units (PUs) computing *N* different filters, and intra-filter parallelism, through *M* multipliers per PU. It also includes support for filter compression according to the format discussed in section 4.4. Both input and output activations are stored on-chip through the whole inference process in order to reduce DRAM off-chip accesses, and filters are retrieved from DRAM on demand with a prefetch policy in order to hide fetch latency.

Fig. 4.5 depicts the architecture of our dense accelerator. Each PU includes its own filter management and a MAC processor. The filter management includes support to decompress filters. The MAC processor is composed of a multiplier array, a tree adder to reduce the multipliers output, and an accumulator. *Activation values memory* is composed of two memories. When processing even layers one stores the input activation and the other one stores the output activation. When processing odd layers they swap their roles. This memory is shared among all the PUs, and there are no conflicts because all the PUs read the same data. In the figure, the pipeline is divided into four stages. These stages can also be internally pipelined in order to increase clock frequency if needed.

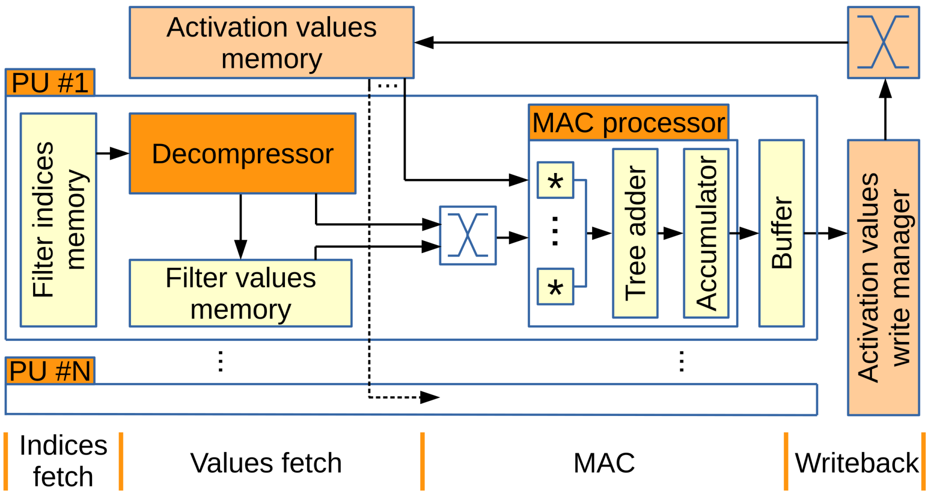


Figure 4.5: Dense architecture.

Stage 1: Indices fetch Filter indices for the next *M* operations are fetched at this stage. These indices are used at the next stage to fetch filter values.

Stage 2: Values fetch

Filter and activation values are fetched at this stage. Activation values are fetched by a global controller as they are shared among all the PUs. Filter values demand specific addressing for each PU because of compression. *Decompressor* is the module responsible for addressing the filter memory based on the filter indices.

Stage 3: MAC

Operations are performed and buffered at this stage. Filter and activation values are retrieved, the MAC operation is performed on *MAC processor*, and the value is stored in a small buffer to avoid pipeline stalls because of writings. *MAC processor* is able to carry out M multiplications in parallel and reduce them in a single cycle through a tree adder.

Stage 4: Writeback

Output activation values are stored in the output activation memory. *Activation values memory* is parameterized with M banks, therefore, arbitration is required for those setups where $N > M$. We implemented a fixed-priority arbitration on each bank in order to keep hardware overhead as low as possible because pressure on this memory is very low since there are many computations between two consecutive writings.

4.7 Sparse architecture

Based on our dense design, we made architectural changes in order to include support for sparsity. Fig. 4.6 shows an architectural overview of our sparse design, which is divided into five stages. As in the dense architecture, each stage can be internally pipelined to increase clock frequency.

Stage 1: Indices fetch

Filter and activation sections are fetched at this stage. Fetching filter indices is straightforward as they are stored in private memories. Activation indices are stored in a shared memory, and, therefore, access conflicts among PUs may arise. We included a multi-bank *Activation indices memory*, and fixed-priority arbitration for each bank. Fetching activation indices may become a bottleneck when dealing with very high sparsity ratios: the more sparsity the faster sections are processed, increasing pressure on memory. We found that including as many banks as PUs,

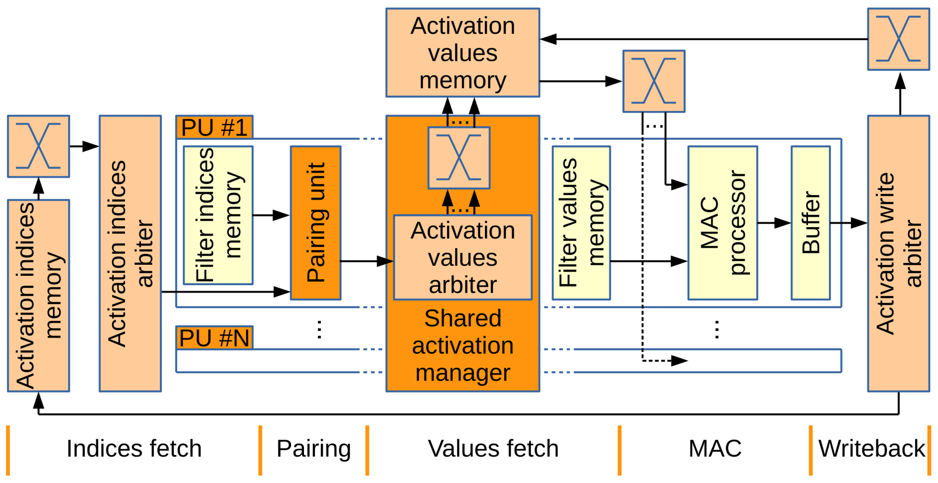


Figure 4.6: Sparse architecture.

in conjunction with a section buffer used to prefetch the next section in advance, causes minimal stalls while keeping hardware overhead low.

Stage 2: Pairing

Useful operations are identified and buffered at this stage following the procedure explained in Section 4.5. Fig. 4.7 shows the architecture of *Pairing unit*. *Sections buffer manager* stores the sections under processing and the next ones to be processed. *Matching unit* processes both filter and activation sections and returns filter and activation offsets to compute their values addresses. *Decompressor* and *Activation addressing* are responsible for computing the absolute memory addresses of filter and activation values, respectively. Finally, *Match buffer* stores these addresses. This buffer is also useful to reduce stalls on accesses to *Activation values memory*. This is further discussed in the next stage.

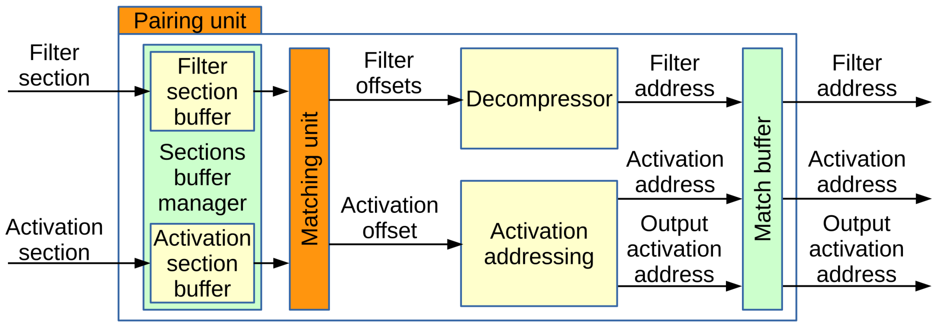


Figure 4.7: Pairing unit.

Stage 3: Values fetch

Filter and activation values are fetched at this stage. Managing accesses to the shared activation memory in our dense design is straightforward, as every PU requests the same values at a given time. However, this no longer applies to our sparse design since accessing only those values that are not zero turns regular accesses into irregular ones, and therefore arbitration is required.

Activation values memory suffers from higher pressure than *Activation indices memory*. Hence, we designed a more powerful arbitration scheme in order to prevent these accesses to become a bottleneck. This arbiter is able to explore several requests from each PU. These requests are stored in the *Match buffer*. We have included support to process the requests out of order (multiplications can be safely reordered within a convolution step), making grants more likely at the expense of a low hardware overhead. *Activation values arbiter* grants each PU one of the requests, if possible, in a fixed order from PU #1 to #N, where PU #1 has the highest priority.

Fig. 4.8 illustrates how our arbitration works on a conceptual example with four PUs. Each PU requests two addresses (i.e., the depth of the *Match buffer* is two). The activation memory is composed of four memory banks, so this memory supports up to four simultaneous accesses as long as they target different banks. The arbiter receives each pair of bank requests from each PU and grants one of them if possible. Thus, PU #1 is granted its first request (Bank #3). As a consequence, Bank #3 is masked for the remaining PUs. PU #2 requests Banks #3 and #1. Since Bank #3 is not available, the arbiter grants its second request (Bank #1). This procedure is repeated for the remaining PUs as shown in the figure.

In this example, the four PUs are granted, but this is not always possible. We empirically searched for the best trade-off between performance and hardware overhead exploring different configurations between the number of banks of the *Activation values memory* and the *Match buffer* size. As can be observed in Table 4.1, we found that providing the *Activation values memory* with twice as banks as PUs, and setting the depth of the *Match buffer* to four, i.e., the arbiter explores up to four requests from each PU, memory access conflicts rarely occur.

As an additional optimisation, we have included support to overlap the end of a convolution step with the beginning of the next one. While the requests corresponding to the last multiplications of the current convolution step are waiting to be granted, it is possible to store requests of the first multiplications of the next convolution step. The operations will need to wait until the last convolution ends, but the request information of the first operations will be loaded. Hence, when a convolution step

Table 4.1: Average conflict ratio for several configurations

Requests	Banks	Conflict ratio
1x	1x	34%
1x	2x	19.5%
2x	1x	16%
2x	2x	2%
4x	1x	6%
4x	2x	0.1%

finishes, the next one can have several requests ready for selection. This approach minimises the memory accesses conflicts.

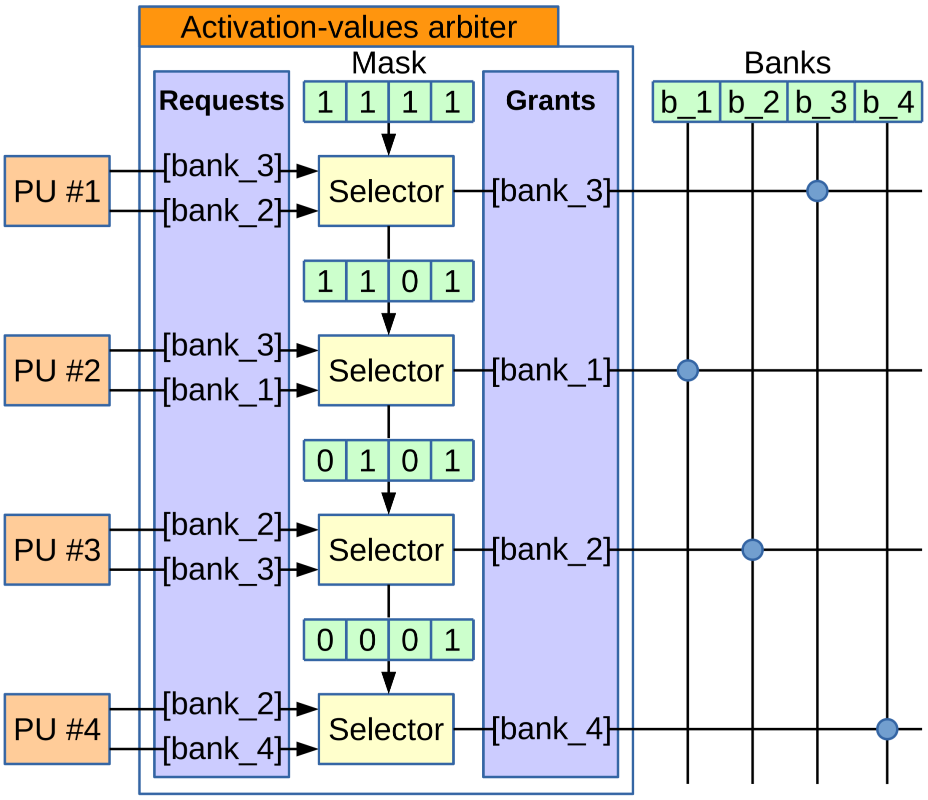


Figure 4.8: Activation values arbiter workflow.

Stage 4: MAC

Useful operations are performed and buffered at this stage. Filter and activation values are retrieved, the MAC operation is performed on a *MAC processor*, and the value is stored in a small buffer to avoid pipeline stalls because of writings.

Stage 5: Writeback

This stage is similar to the to corresponding stage in the dense architecture.

4.8 Memory requirements and dataflow

The size of the memories in our design is parameterizable, so it can be adjusted to the needs of each DNN. The memory hierarchy of our design includes the following storage elements:

- Off-chip memory: it stores all the filters of the network and the image/s to be processed. In an FPGA this will be a DDR memory.
- On-chip memory: it stores the input and output activations and the filters under processing in the following memories:
  - Filters memory: Each PU includes private resources to store both the filter for the ongoing convolution and the next one, which is preloaded to hide the fetch latency. As depicted in Figures 4.5 and 4.6, the filter values and their indices are stored in two different memories, the *Filter values memory* and the *Filter indices memory* respectively.
  - Activation memory: the *Activation values memory* and the *Activation indices memory* shown in the figure 4.6 store the activation values and their indices in two shared memories. Both of them are multi-banked and are accessed through arbiters to prevent conflicts. This memory element contains both the input and the output activation of the layer under processing.

Regarding the dataflow: first, the image is loaded into the activation memory, and one filter is loaded into the private filter memories of each PU. Once the inference begins, each PU will exploit reuse by convolving the filter across the whole input activation. This scheme reduces bandwidth with off-chip memory as filters are retrieved only once per inference. While convolving a filter, another one is retrieved from the off-chip memory in order to hide fetch latency. Each PU stores its ongoing convolution partial results in an accumulator register. No data is shared among PUs. Once the output of a layer has been stored, it becomes the input activation of the next layer.

4.9 Scalability

Inference on CNNs is a parallel-friendly task. Many filters are convolved in each layer, and many operations are performed in each filter. Both inter-filter and intra-filter parallelism are free of data dependencies, making it suitable for a custom hardware architecture to exploit it. This is indeed what we do in our dense architecture, where adding PUs exploits inter-filter parallelism, and adding multipliers to each PU exploits intra-filter parallelism.

When exploiting sparsity, we have to deal with conflicts in the access to the activation data memory, which requires hardware hard to scale. Our design needs crossbars to access multi-banked activations memories, and scaling them causes the performance-area trade-off to plummet. Hence, our sparse architecture must be small in order to be efficient. For that reason, it is more suitable for embedded systems. For other contexts, like high resolution images, there are very good massively parallel architectures with hundreds of PEs, such as *SCNN* [59] and *EyerissV2* [32], whose high throughput fits the needs of these problems.

Nonetheless, there are contexts where it is still possible to scale the design by including several instances of our architecture (i.e., cores) working in parallel on their own private activation memories. For example, if several images must be processed, they can be assigned to each of those cores. It is also possible to assign different regions of the activation to each core. In this case, there will be a small overlap among the activations, and therefore some additional control hardware would be necessary to share these overlapped data between cores.

4.10 Experimental results

We implemented our sparse and dense designs on a Xilinx Zynq UltraScale+ ZCU104 evaluation board [9]. This platform includes a SoC with an FPGA tightly coupled with a CPU, a real-time processor, and a GPU. In our experiments, only the FPGA and the CPU were used. The FPGA, which hosts our accelerators, performs all the computations, and the CPU just manages the communications with the off-chip memory.

Power consumption measurements were taken with a Yokogawa WT210 digital power meter, a device accepted by standard performance evaluation corporation [11]. Our power meter records the total consumption of the evaluation board, which includes many unused elements. Hence, we have removed the static power consumption from our measures, and we have focused on the dynamic power consumed by our design, i.e., the average difference of power consumed by the board during idle and running states. To do this, we have measured the energy consumption of our designs for one hour in each state.

The purpose of our experiments is to characterise the behaviour of our designs in terms of performance, hardware resources, and energy efficiency, as a function of the useful operations (which depends directly on the sparsity) and the arithmetic bitwidth. To analyse the impact of the sparsity, we developed a set of synthetic benchmarks with a useful operations ratio ranging from 0 to 1, i.e., we started with a scenario where each multiplication include a zero as one of its operands, and we progressively reduced the number of zeros until reaching the opposite scenario, where all the operands are different from zero. To study the impact of the bitwidth we implemented three different versions of each design using 8, 16, and 32-bit fixed-point arithmetic. All the results in the figures of this section are normalised to the baseline selected in each case study.

Our experimental setup is divided into three model designs:

a) Sparse: sparse architecture with one multiplier per PU.

b) Dense base: dense architecture with one multiplier per PU.

c) Dense equivalent: dense architecture where the number of multipliers per PU is selected in such a way that the dense design is as similar as possible in area to the sparse design.

Table 4.2 details the design parameters, hardware resources utilisation, maximum frequency, and the dynamic power consumption of each setup. In our experiments all the setups were clocked to 100 MHz. FPGAs include DSP blocks that can be used to execute MACs, and synthesis tools map these operations into them whereas the remaining functionality is implemented using look-up tables (LUTs). This makes impossible to compare ones with each others, as MACs are implemented with a different technology than the rest of the logic. Hence, we disabled DSP mapping in order to map also the MAC processors into LUTs.

We first compared *sparse* and *dense base* models, which include the same arithmetic resources (number of PUs, and multipliers per PU). Hence, both designs exploit the same degree of parallelism. We want to assess the benefits in terms of performance and energy efficiency of including support for sparsity, and quantify the hardware overhead introduced.

Table 4.2: Experimental Setups with 8 PUs

8-bits Arithmetic				
Setup	Mults / PU	Logic (LUT %)	Max. Freq	Dyn. Power
sparse	1	5.94	124 MHz	131 mW
dense	1	2.04	123 MHz	39 mW
dense eq.	8	5.41	123 MHz	206 mW
16-bits Arithmetic				
Setup	Mults / PU	Logic (LUT %)	Max. Freq	Dyn. Power
sparse	1	7.94	110 MHz	247 mW
dense	1	3.80	112 MHz	87 mW
dense eq.	4	7.58	115 MHz	472 mW
32-bits Arithmetic				
Setup	Mults / PU	Logic (LUT %)	Max. Freq	Dyn. Power
sparse	1	12.32	106 MHz	842 mW
dense	1	8.54	101 MHz	462 mW
dense eq.	2	13.02	101 MHz	1104 mW

Fig. 4.9 depicts the speedup as a function of the useful operations. As all the setups are clocked to 100 MHz, the speedup grows in inverse proportion to the percentage of useful operations since the reason for this speedup is the number of operations avoided, therefore the three different arithmetics analysed (8, 16 and 32 bits) achieve the same speedup.

Fig. 4.10 depicts the energy efficiency as a function of the useful operations. In the top of the figure the overhead in terms of logic resources is presented. Unlike speedup, the energy efficiency and the area overhead depend on the arithmetic bitwidth. The reason is that arithmetic computations require less logic resources and consume less energy for low bitwidth, whereas the area and energy consumption due to the support included for the sparsity remains the same.

Gains in performance and energy efficiency are remarkable for highly-sparse scenarios. However, achieving these results involves a logic overhead ranging from 50% to almost 200%. Hence, the question is: what if we provide our dense design with similar hardware resources? Comparison between *sparse* and *dense equivalent* models answers this question. Setups in *dense equivalent* design balance hardware resources by including more multipliers for each PU. Scaling by exploiting the intra-filter parallelism is feasible due to the size of the filters in actual CNNs, and demands little overhead (tree adders to reduce the multiplications, and a little more complex

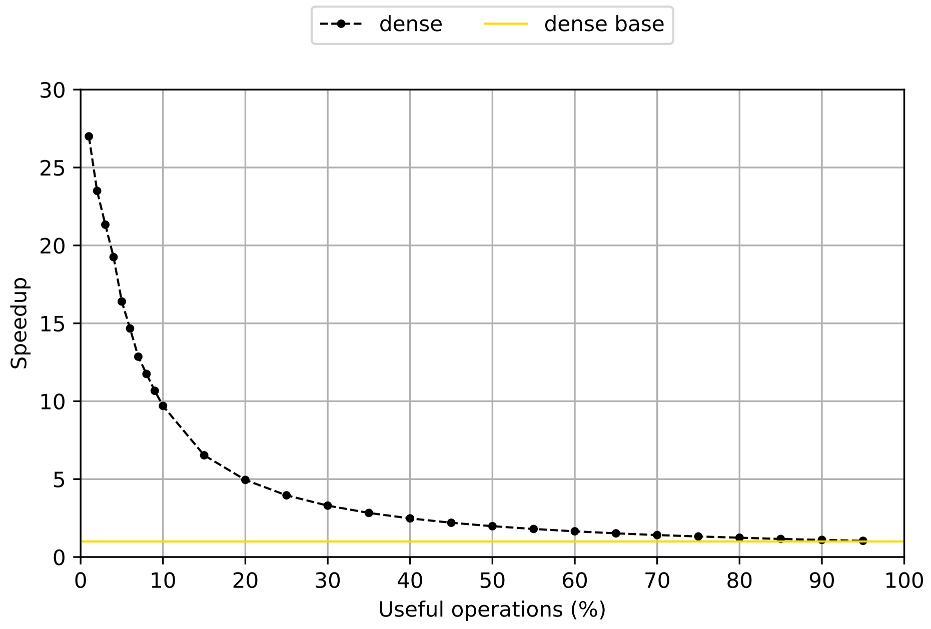


Figure 4.9: Speedup of *sparse* experimental setups normalised to *dense base*.

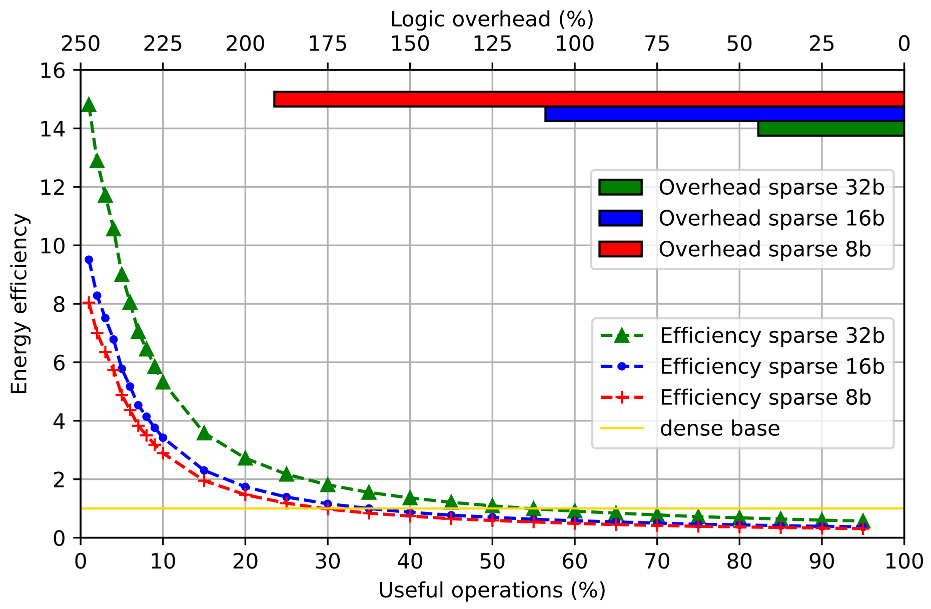


Figure 4.10: Energy efficiency of *sparse* experimental setups normalised to *dense base*.

addressing control). Our objective is to identify whether it is worthy to move from a dense to a sparse architecture for a given scenario. For that, we are going to compare our sparse design with a dense design with similar hardware resources, i.e., similar area. As can be seen in Table 4.2, for 32-bit arithmetic the dense equivalent version includes twice the number of multipliers than the sparse version, for 16-bit it includes 4x multipliers, and for 8-bit it includes 8x. These differences are due to the different sizes of the multipliers for each arithmetic bitwidth.

Results in terms of performance are shown in Fig. 4.11. The results show that the benefits of providing support for sparsity have decreased. Our *sparse* design working on 8-bit arithmetic is only worthy when the useful operations are below 10%. On 16-bit, the threshold grows up to 25%, and on 32-bits, the threshold is at 50%. Numbers of energy efficiency are more favourable to the *sparse* architecture (Fig. 4.12). The benefits show up when the useful operations are below 20%, 60% and 70%, respectively. Hence, for low-precision arithmetic, it is only profitable to include support for sparsity in aggressively pruned models.

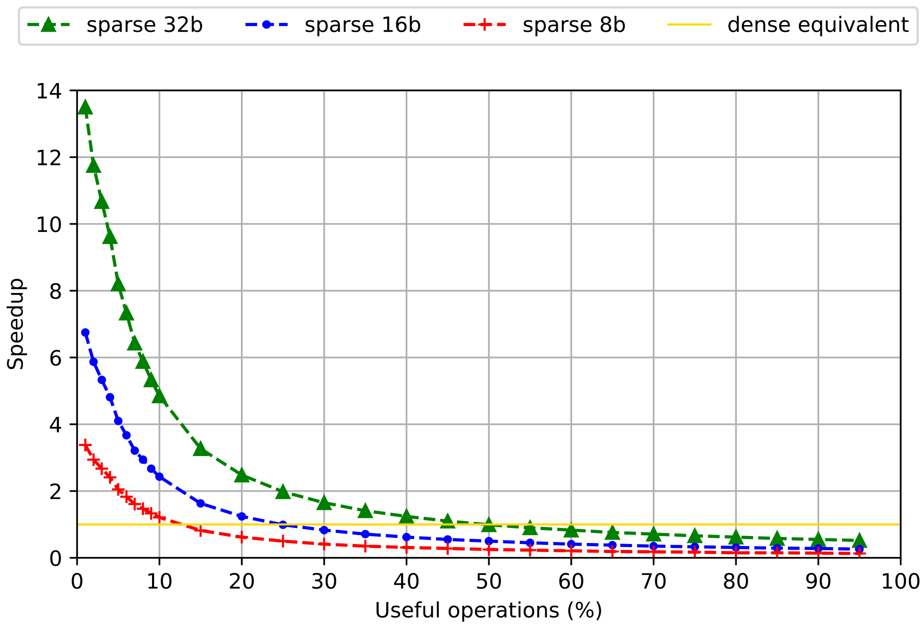


Figure 4.11: Speedup of *sparse* experimental setups normalised to *dense equivalent*.

One of the goals of our design is to maximise the utilisation of arithmetic resources. Fig. 4.13 shows that MAC utilisation is virtually 100% even for networks with a very low useful operations ratio. It only plummets when the useful operations is under $\sim$5% because, in that situation, frequently the 32-bit matching unit cannot find any useful operation in the fetched sections. Even so, this is not a undesirable scenario

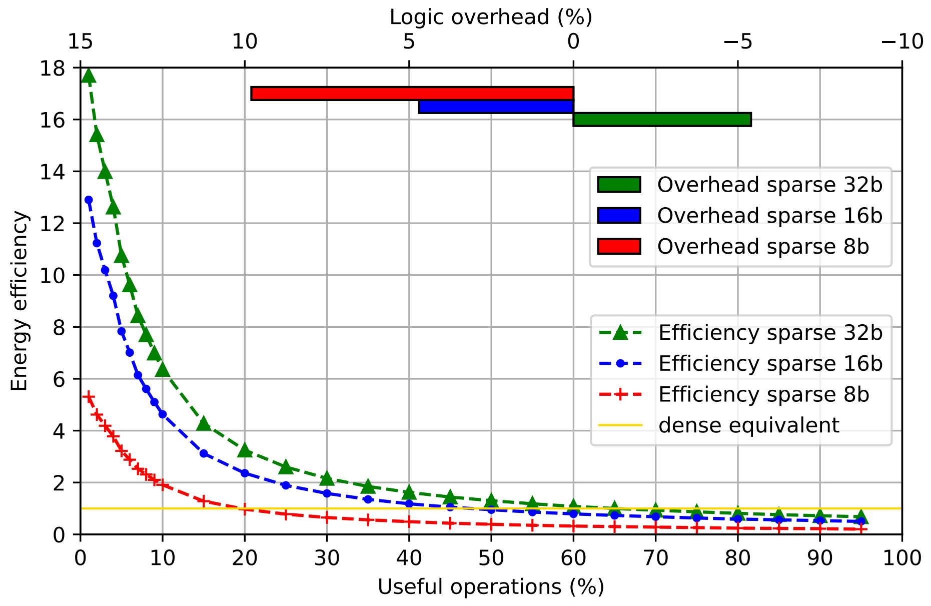


Figure 4.12: Energy efficiency of *sparse* experimental setups normalised to *dense equivalent*.

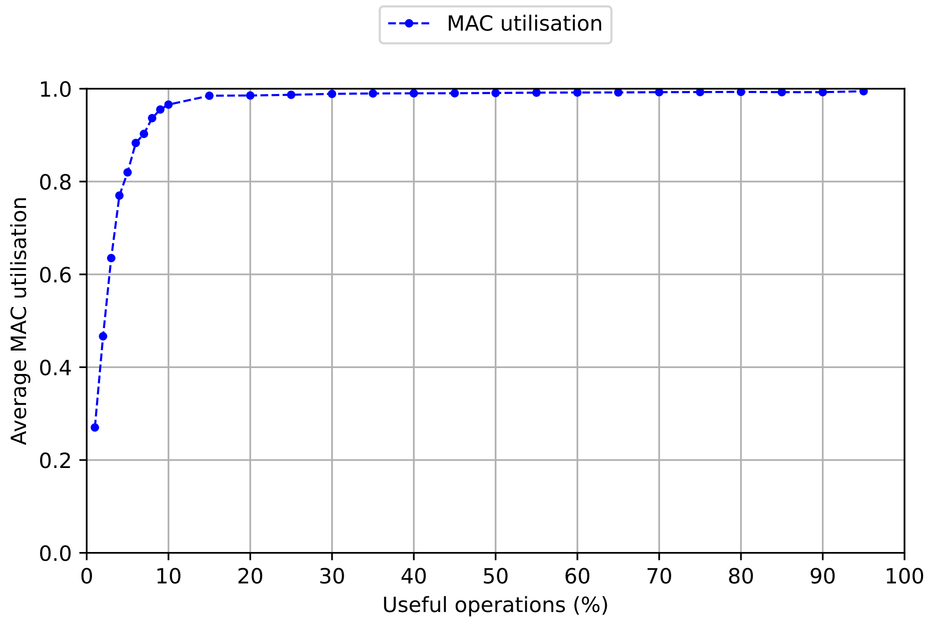


Figure 4.13: Average MAC utilisation

for our architecture. In fact, when no useful operations are found our architecture is indeed skipping 32 operations in one cycle.

4.10.1 *AlexNet* and *SqueezeNet*

The previous results have been obtained using synthetic data generated randomly for a given value for a given percentage of useful operations. However, we also wanted to try our accelerator with representative pruned models and data sets. Although many works have demonstrated the possibilities of pruning, deep learning frameworks still do not include support in their main branches. However, the authors of [72] shared two pruned models, *AlexNet* and *SqueezeNet*, on GitHub [70, 69]. These are very good benchmarks for our design since they are popular models that use the representative data set ImageNet [91], and *SqueezeNet* is especially interesting for Embedded Systems. Tables 4.3 and 4.4 detail the useful operations ratio per layer of these two pruned networks and the MAC utilisation of our sparse design for each layer. Overall, the useful operations ratio on *AlexNet* and *SqueezeNet* are 18.4% and 32.0%, respectively.

Table 4.3: Useful Operations per Layer in *AlexNet*

	Useful operations	MAC utilisation
conv1	84.3%	99.3%
conv2	6.7%	98.9%
conv3	10.9%	98.2%
conv4	13.5%	98.4%
conv5	11.3%	97.6%
fc6	3.6%	51.9%
fc7	5.8%	70.3%
fc8	7.6%	90.8%

Fig. 4.14 and 4.15 depict the execution time per layer of *AlexNet* and *SqueezeNet*, respectively. We have compared our *sparse* design working with 8-bit, 16-bit, and 32-bit arithmetic with their area equivalent dense designs, i.e., *dense equivalent* model with 8x, 4x and 2x multipliers respectively. Results vary among layers because of their different useful operations ratios, as shown in tables 4.3 and 4.4. When executing *AlexNet*, our *sparse* design outperforms its *dense equivalent* by 2.66x on 32-bit arithmetic and 1.33x on 16-bit. On 8-bit, the *dense equivalent* design with 8x multipliers is superior by 1.50x. When executing *SqueezeNet*, our *sparse* design outperforms its *dense equivalent* by 1.53x on 32-bit arithmetic whereas the *dense equivalent* design is superior on 16 and 8-bit by 1.31x and 2.61x, respectively.

Table 4.4: Useful Operations and percentage of MAC utilisation per Layer in *SqueezeNet*

	Useful operations	MAC utilisation
conv1	98.5%	99.6%
fire2	41.2%	95.5%
fire3	37.4%	96.8%
fire4	30.5%	97.8%
fire5	40.9%	98.1%
fire6	34.0%	98.1%
fire7	28.0%	98.2%
fire8	25.8%	97.9%
fire9	26.6%	98.5%
conv10	2.3%	51.9%

Given these execution times, our accelerator is able to process 227x227 images in 203 ms for *AlexNet* and in 265.2 ms for *SqueezeNet*, yielding a throughput of 3.8 and 4.9 images/s, respectively. These numbers are suitable for many embedded applications. They may not look impressive at a first glance, but, in fact, they are very close to the peak performance. For instance, processing an image with the pruned version of AlexNet [69] involves 863,740,448 multiplications. Since most of them include a zero as an operand, with our approach that number can be reduced to just 159,031,554 useful multiplications. The minimum execution time using 8 multipliers clocked at 100MHz and assuming peak performance is 198.79ms, just a bit lower than the 203ms that our sparse architecture needs.

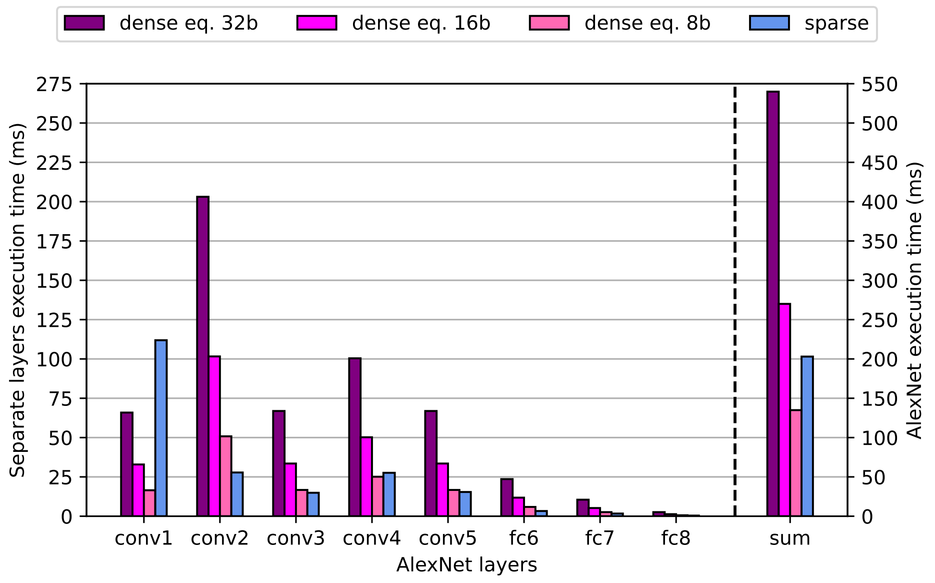


Figure 4.14: Execution time on *AlexNet*

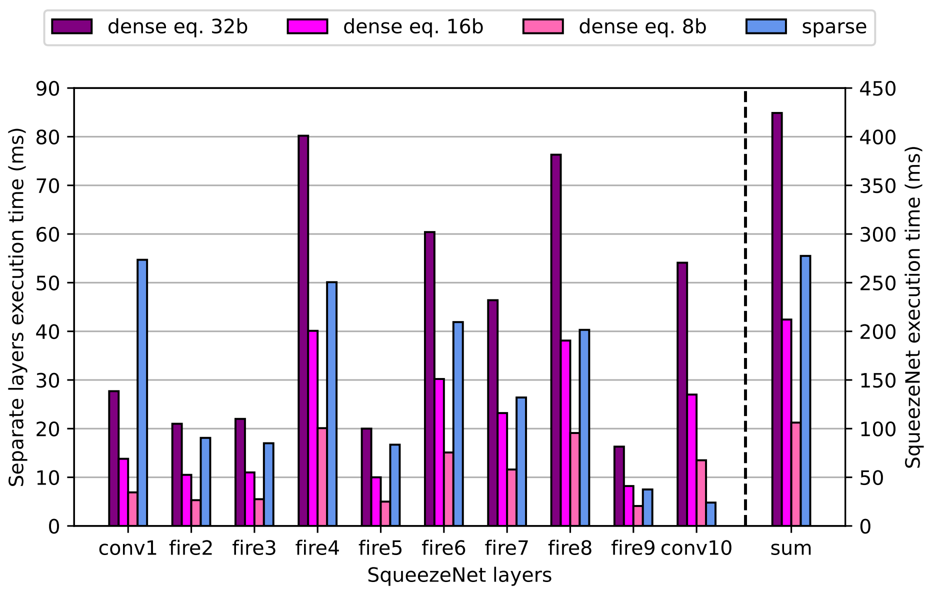


Figure 4.15: Execution time on *SqueezeNet*

4.11 Conclusion

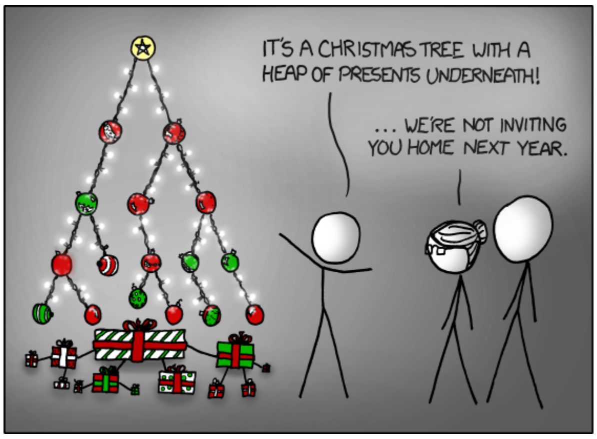
We propose a sparse architecture for DNNs that avoids those operations with zero as one of the operands and keeps almost peak utilisation of arithmetic resources, even in highly-sparse scenarios. The architecture includes support to deal with compressed filters, identify the useful operations, and reduce the memory access conflicts generated due to the non-uniform memory accesses. It has also been pipelined to improve the performance.

We have carried out comparisons between similar-in-area dense and sparse architectures in order to identify in which scenarios including support for sparsity is superior to providing a dense architecture with additional arithmetic resources. Sparsity is, as expected, the key parameter to decide whether to move from dense to sparse architectures, but arithmetic bitwidth also plays a major role. The hardware cost of MAC units does not scale linearly with the bitwidth; therefore, the overhead ratio of a sparse architecture is much larger on low precision. Our results show that the benefits of exploiting sparsity are clear on 32-bit arithmetic, whereas on 8-bit it is hard to profitably exploit sparsity given the sparsity of current state-of-the-art CNNs. Recent works show consensus on using arithmetic of at least 16-bit [66, 71, 74, 43, 59]. For this particular arithmetic bitwidth, adding support for sparsity improves energy efficiency as long as the useful operations are under 50%, and also performance, when the useful operations are under 25%. In other scenarios it is better to use the logic resources to include more arithmetic resources than to include support for sparsity.

We consider that FPGAs are currently the natural target for sparse accelerators, instead of ASICs as suggested in previous works. The benefits of including support for sparsity depend on the particular sparsity and arithmetic precision of the DNN to process, and FPGAs can be seamlessly adapted by loading the best-fitting accelerator for each profile.

Developing pruning techniques for DNNs is currently a very active research topic and we expect that many aggressively pruned models are likely to be available soon. The inclusion of support for sparsity will be needed in order to take advantage of this powerful optimisation opportunity.

5 Step II: Gradient Boosting Decision Trees Accelerator


IT'S A CHRISTMAS TREE WITH A HEAP OF PRESENTS UNDERNEATH!
... WE'RE NOT INVITING YOU HOME NEXT YEAR.

— **xkcd.com**
(Just that guy, you know?)

5.1 Introduction

Based on the result of the analysis of chapter 3, we identified that Gradient Boosting Decision Trees (GBDTs) present a very interesting trade-off between the use of computational and hardware resources and the obtained accuracy. Their accuracy results were close to those obtained with convolutional neural networks, which currently is the most accurate method, while carrying out one order of magnitude less computational operations. Moreover, most of their operations during inference are integer comparisons, which can be efficiently calculated even by very simple low-power processors, and can be easily accelerated by FPGAs. For that reason they represent a good option for embedded system.

Decision trees are a light and efficient machine learning technique that have proved their effectiveness in several classification problems. A single decision tree is frequently not very accurate for complicated tasks but, thanks to ensemble methods, it is possible to combine several trees in order to deal with complex problems. GBDTs is an ensemble method that allows to improve the accuracy gradually adding new trees in each iteration that improve the result of the previous ones [100]. Conventional implementations of GBDT suffer from poor scaling for large datasets or a large number of features, but recently some efficient implementations have overcome this drawback such as XCGBoost [67], CatBoost [40], or LightGBM [57]. For instance, LightGBM is a highly efficient open-source GBDT-based framework that offers up to 20 times higher performance over conventional GBDT. With the support of Light-GBM, GBDTs are currently considered one of the most powerful machine learning models due to its efficiency and accuracy. For example, recently they have been used for many winning solutions in several machine learning competitions [25]. They can be used for very different problems. For instance, they have been successfully used to produce accurate forecasts for the COVID-19 evolution, and to identify factors that influence its transmission rate [19]; to detect fraud from customer transactions [1]; to estimate major air pollutants risks to human health [28] using satellite-based aerosol optical depth; or to classify the GPS signal reception in order to improve its accuracy [42].

In this chapter we present an accelerator for GBDTs that can execute the GBDTs trained with LightGBM. Our accelerator has been designed for embedded systems, where hardware resources and power budget are very limited. Hence, our primary goal is efficiency. The register-transfer level (RTL) design of our accelerator has been written in VHDL, and, to demonstrate its potential, we have implemented it in a low-cost FPGA evaluation board (ZedBoard)[10], which includes an FPGA for embedded systems, and we have used our implementation for the same case study previously analysed during the comparative of the different methods. We have measured the execution time and power consumption of our accelerator and we have identified that our design can be used to process complex GBDT models even when using a small FPGA. In our case study, our accelerator can process the hyperspectral information at the same speed at which the hyperspectral sensors generate it, and the dynamic energy consumption due to the execution is an order of magnitude less in both cases, compared to a high performance CPU and compared to an embedded system CPU. Hence it could be used for on-board processing in remote sensing devices.

The content of this chapter has been published in a very prestigious journal [16], and the codes of the accelerator are available in a public repository [15].

5.2 Related work

Several previous works have targeted FPGA acceleration of Decision Trees. [95] focuses on the training processes. In our case we assume that training is carried out offline and we want to focus on inference, which will be computed online. [82] presented a custom pipeline architecture which demonstrated the potential of an accelerator for decision trees. However they do not support GBDT and they apply their techniques only to simple case studies. [85] proposes to use a high-level synthesis approach to design an FPGA accelerator. They focus on Random Forest, which is an ensemble technique that calculates the average value of several trees trained with different input data to generate a more accurate and robust final output. We have decided to focus on GBDT instead of Random Forest since recently GBDT have demonstrated an enormous potential [25] and our comparative analysis shows that GBDT provide better accuracy while using smaller models, hence we believe that it is a better approach for embedded systems. Another difference is that we have designed a custom register-transfer level (RTL) architecture instead of using a high-level synthesis that will automatically generate the RTL design from a C-code. High-level synthesis is very interesting for portability, and to reduce the design cycle, but with our RTL design we can fully design the final architecture and explore several advanced optimisation options. [90] is another work that analyses the benefits of implementing Random Forest on FPGAs. They compare the effectiveness of FPGAs, GP-GPUs, and multi-core CPUs for random forest classifiers. They conclude that FPGAs provide the highest performance solution, but they do not scale due to the size of the forest. In this sense, as explained before, GBDT models require fewer trees to obtain the same accuracy, so it is a more suitable model for FPGAs. [89] proposes to use FPGAs to accelerate the execution of decision trees used in the Microsoft Kinect vision pipeline to recognise human body parts and gestures. They use a high performance FPGA, and obtain very good results for decision tress organised as random forest. However, they identify that their design cannot be used in low-power FPGAs due to its memory requirements.

[23] is a very recent work that presents an algorithm that produces compact and almost equivalent representations of the original input decision trees by threshold compaction. The main idea is to merge similar thresholds to reduce the number of different thresholds needed, and store those values as hard-wired logic. With this approach the size of the trees can be reduced. This technique is orthogonal to our approach and can be beneficial to our design since it reduces the size of the trees, which simplifies its storage in embedded systems.

[27] is another recent work that analyses the benefits of FPGA acceleration for Gradient-boosted decision trees. In this case they evaluate the services provided by Amazon cloud, which include the access to high-performance FPGAs that can be used through high-level interfaces. Therefore, this work is complementary to ours, as it focuses on high-performance cloud servers while we focus on embedded systems.

5.3 Design architecture of the GBDT Accelerator

After considering several options, we decided to implement an accelerator for GBDTs trained with LightGBM library for being one of the latest and most used. LightGBM follows a one-vs-all strategy for classification problems that consists in training a different estimator (i.e. a set of trees) for each class, so each one of them predicts the probability of belonging to that class. With this approach each class has their own private trees, and the probability of belonging to a given class is obtained by adding the results of its trees, as shown in Figure 3.2.

Hence, during inference, each class is independent from the others, and the trees of each class can be analysed in parallel. Our accelerator takes advantage of this parallelism by including one specific module for each class.

5.3.1 Memory Format

To design an efficient accelerator, it is essential to optimise memory resources. Our goal is to store the trees in the on-chip memory resources of the FPGA to minimise data transfers with the external memory. However those resources are very limited, so a key point of the design is to optimise the format used to store the trees in order to reduce their memory requirements.

All the trees of a class are mapped into its *trees_nodes* RAM memory, which is local to the class module. Fig. 5.1 present the representation that we have selected to store the tree structure on this memory. Our objective is to include all the information of each node in a 32-bit word. Since we use generic parameters in our code, this word size can be enlarged or reduced as needed. But in our experiments we have observed that 32 bits provides a good trade-off between the accuracy to represent the trees and the storage requirements. These 32 bits follow two different formats taking into account whether they are leaf nodes or not.

The format for non-leaf includes four fields. The first and the second field store the information needed to carry out the comparison, i.e. which input will be used (8 bit), and with which value it will be compared(16 bit). Then we need to store the addresses of the child nodes. As we only have 8 bits remaining, it is not possible to store its absolute addresses. In fact, with the size of the memories that we are using, we would need almost all of the 32-bits to store that information. We have solved this issue with two solutions. First, in the *trees_nodes* memory, nodes are stored using the pre-order traversal method, i.e. the left child of a non-leaf node is always allocated in the following memory position. With this approach the address of the left child does not need to be stored, since it can be obtained adding one to the current address. Hence, we only have to store the address of the right child. Second, instead of storing the absolute address of the right child, we store a relative address that indicates its distance with the current address. This relative distance is stored in a 7-bit field. With this approach the maximum depth of a tree is 128. This is more than enough for all the trees that we have analysed, since GBDT does not rely on very large trees, but in using many of them. Finally, we have included a flag in the less significant bit of each node: This flag determines whether it is a leaf node or not.

In the case of the leaf nodes, the 32-bits memory word includes four fields. A 16-bits field stores the output of the three. The next 14-bits are used to store the address of the next tree. In our experiments 14 bits were enough for the absolute addresses. The original output of the LightGBM GBDTs is a 32-bit floating point. However, using a 16-bits fixed-point representation we obtain similar accuracy in our experiments. In any case, if needed, it is possible to use more bits for the output without increasing the size of the memory word by using relative addresses for the *@next_tree field* instead of absolute addresses. The last two bits are two flags that identify whether this is the last tree in the class, and whether the node is a leaf or not.

Figure 5.2 presents a simple example in which two trees of a class are stored. As can be seen in the figure, the root node of the first tree is stored in address 0. Then, the entire tree structure corresponding to its left child is stored, following these same rules recursively, and finally the right child is stored in the last place. The bits corresponding to *rel@_right_child* field store the relative jump to its right child. All the leaf nodes of the first tree indicate that the next tree begins in address 5. Finally, the leaf nodes of the second tree indicate that there are no more trees to process. This simple example includes 8 nodes. If we execute it in our architecture we will visit four or five of these nodes, it depends on the result of the first comparison, and we will need approximately one clock cycle to process each node. In larger trees,

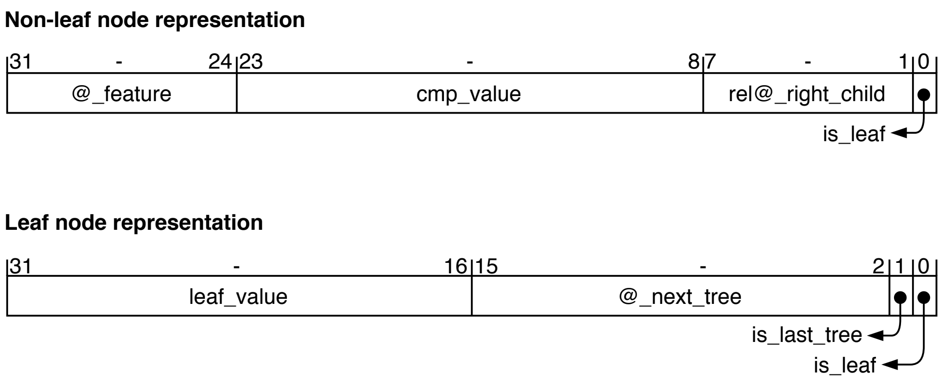
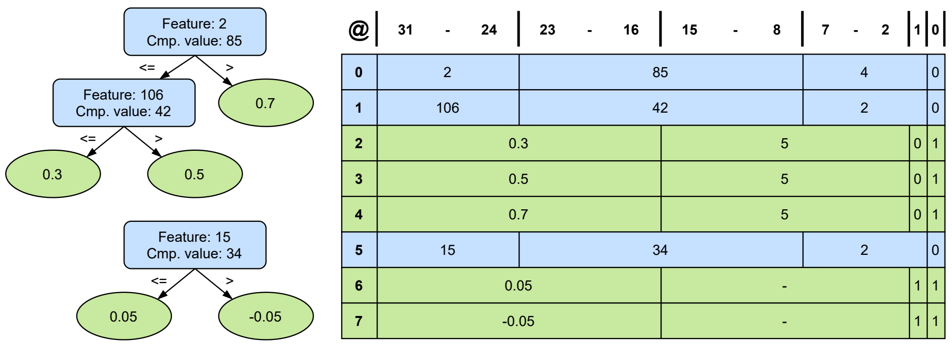


Figure 5.1: Node representation

the number of visited nodes will be much lower than the number of total nodes, and the execution time will remain approximately one cycle per visited node.



@	31 - 24	23 - 16	15 - 8	7 - 2	1	0
0	2	85		4		0
1	106	42		2		0
2	0.3		5		0	1
3	0.5		5		0	1
4	0.7		5		0	1
5	15	34		2		0
6	0.05		-		1	1
7	-0.05		-		1	1

Figure 5.2: Trees representation example

5.3.2 Accelerator Architecture

Figure 5.3 depicts the internal design of one of the modules that execute the trees of a class. The design includes the previously described *trees_nodes* RAM memory, a register, *@_last_node*, that is used to store the address of the last visited node, and the logic that carries out the comparisons, compute the next node to visit, and accumulate the results of the trees.

In this design the *@_feature* field of non-leaf nodes is used to select one feature among all the input features of the system. The selected feature is compared with the *cmp_value* field. If the value of the feature is less or equal than the *cmp_value*, the left child of the non-leaf node is selected. To this end, we add 1 to the *@_last_node*. Otherwise, we add the *rel@_right_child* to select the right child. This will generate the *@_node* that will be used to address the *trees_nodes* RAM memory in case that

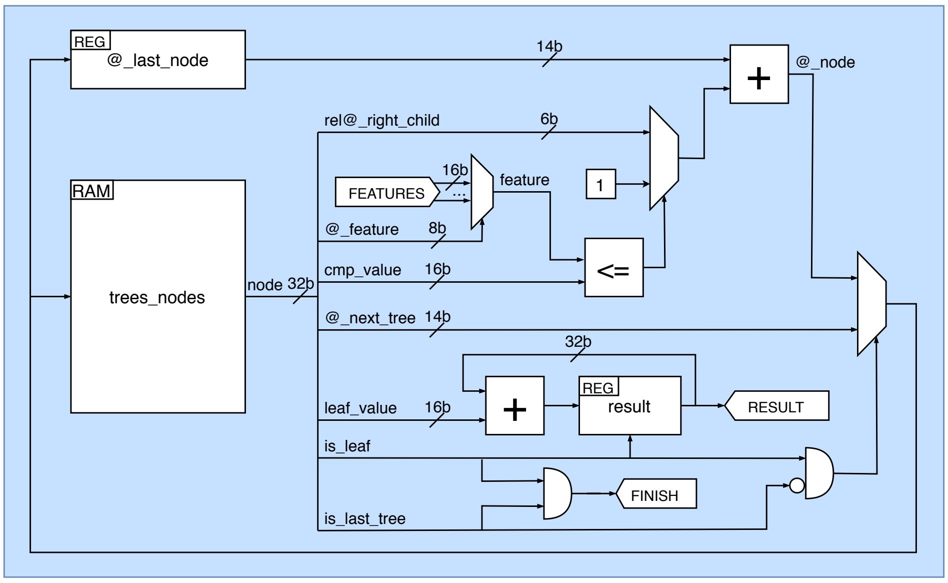


Figure 5.3: Class diagram

the current node is a non-leaf (*is_leaf* value is 0). If the current node is a leaf (*is_leaf* value is 1), and this is not the last tree (*is_last_tree* value is 0), the selected value to address the *trees_nodes* RAM memory will be the *@_next_tree* field of the leaf node. On every leaf node, the *result* register will accumulate the *leaf_value* field to the previous result value.

According to the selected memory representation of the nodes, the maximum size of the *trees_nodes* RAM will be 2^{14} words, as we dedicate 14 bits to the *@_next_tree*, and the theoretical maximum number of nodes of the same tree will be 2^6 due to the size of the relative jump *rel@_right_child*. Regarding the size of the RAM, we could address any number of trees just making the *@_next_tree* a relative address from the current node by adding it to *@_last_node*, nevertheless the current size is even bigger than our needs. In our design we dedicate 8 BRAMs of $32Kb$ to the trees of each class, which is 8192 words of 32 bits, so we are actually using the 13 less significant of the 14 bits available to address the *trees_nodes* RAM. Regarding the number of nodes of each tree, this is only a theoretical limit due to the relative jump, which actually affects only the left side of each node, i.e. the left side of each node of the tree can only have 63 nodes, so we could reach the right child in a pre-order traversal. In any case, the maximum number of nodes of our trees is 61 and the average is between 7 and 22 depending on the dataset, so this is not a problem either.

Once we receive the features of one pixel, we only need to wait until every class module has finished and then check the output of the argmax module, which selects the number of the class with the higher result. Fig. 5.4 depicts a simplified design of the accelerator showing this behaviour, where we omitted the control lines and the management of the communications. To overlap the reception of the features of the next pixel with the computation of the current one there are actually two *features* registers, so we can receive in one of them while we are working in the other.

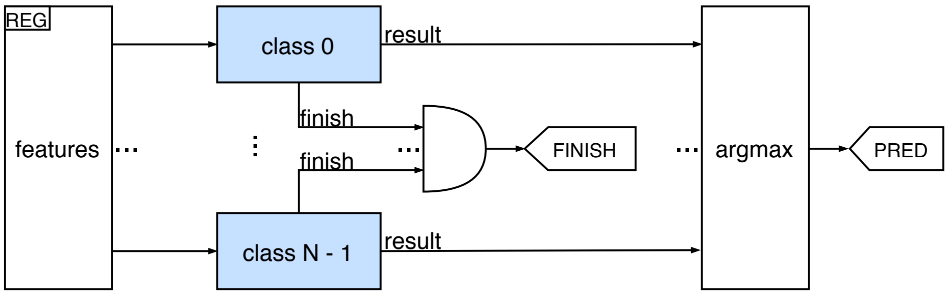


Figure 5.4: Accelerator design

5.3.3 Pipeline and Multi-threading Architecture

This design can process a node per class in each clock cycle. However, in our experiments, if we implement a system that uses most of the on-chip memory resources, the place&routing process becomes complex and the clock frequency is just 55 MHz. This can be solved by using a more modern FPGA, with more capacity, and better integration technology, but it can also be improved by applying some computer architecture optimisations similar to those that have been used to optimise the execution of general purpose processors. We will illustrate this with the following figures. In Figure 5.5 (a) we present the execution of the previously described version (single-cycle implementation). The figure depicts the execution of the nodes in one of the classes. If we have N classes all of them will be executed in parallel. In the figure three nodes are executed in three clock cycles. However, as explained before, the clock period is long and the system runs slow.

Our goal is to improve the speed, while trying to keep processing one node per cycle. To reduce the clock cycle we have designed a multi-cycle implementation. We have explored three different options: two, three and four clock cycles. In these versions, additional registers have been added to the architecture in order to split the longest combinational paths. From that analysis we selected the option that executes the nodes in three cycles, because it achieves an important clock-period reduction and at the same time provides a clear architecture, in which it is easy to identify the actions that are carried out in each cycle. With four cycles, the benefits were very small, and the resulting execution scheme was not intuitive. Figure 5.5 (b) presents the results

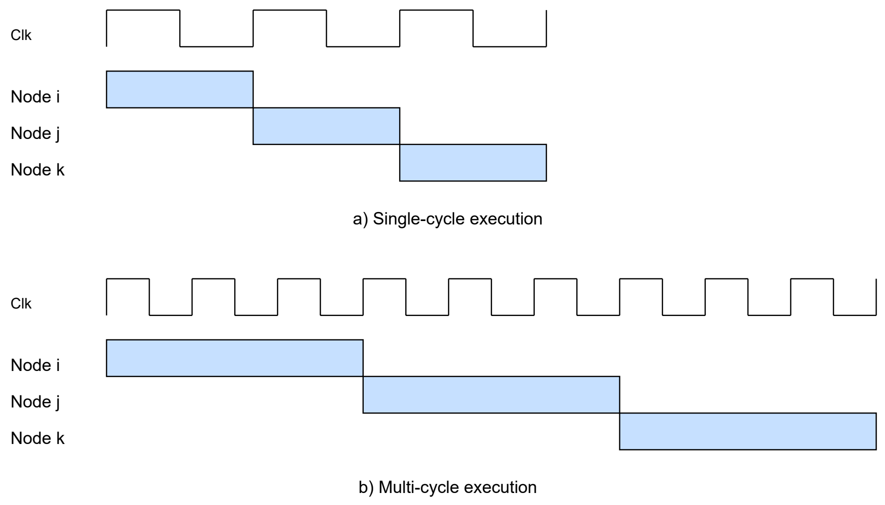


Figure 5.5: (a) Single-cycle execution scheme. (b) Multi-cycle execution scheme.

after this step. As can be seen in the figure, the clock frequency has been improved, but the system is slower than before, since we need three clock cycles for each node. However, since we have partitioned the design following a clear scheme, we could try to use a pipeline approach. As presented in figure 5.6 (a) we have three different pipeline stages, and each one of them uses different hardware resources. Hence, we can start executing a second node, as soon as the first one has finished the first stage. In our design the first step is used to read the node (fetch), the second step is used to identify the type of the node and read the needed feature (decode), and the last step is used to compare it with the comparison value, and identify the next node to execute (execution). The problem of this scheme is that we do not know the next node until the previous node has finished its execution stage. Hence we cannot fetch it in advance. Hence, a simple pipeline will not provide any benefit. This is the same problem that conventional processors have when dealing with instructions that include conditional branches. High-performance processors alleviate this problem by including complex support for speculative execution, but it is not an efficient solution for embedded systems, since it introduces important energy overheads, and it will not achieve good results, unless it is possible to identify clear patterns for branch predictions. Hence, there is no straightforward way to take advantage of the pipeline architecture.

However, this problem can be solved by combining the pipeline with a multi-threading approach. The idea is to include support to interleave the execution of three different trees. This can be done by including three @_last_node registers (that is the same as having three program counters in a processor). The trees in a class are divided in three sets, and each counter manages the execution of one

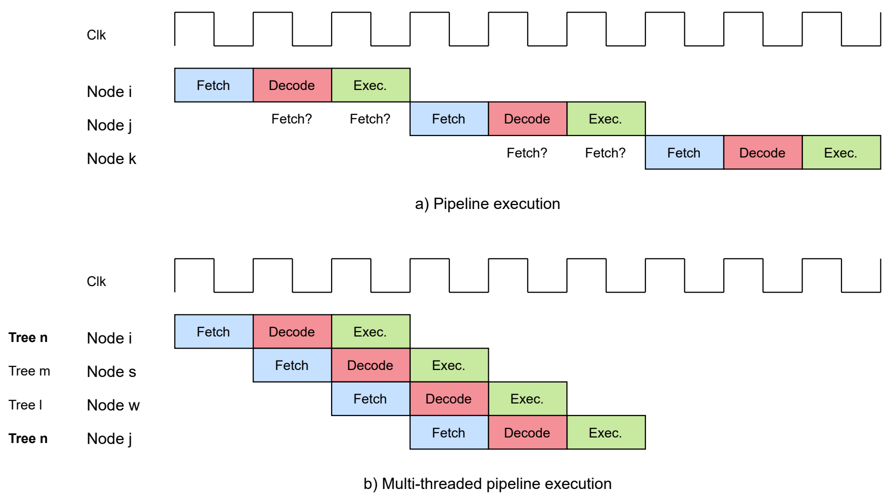


Figure 5.6: (a) Pipeline execution scheme. (b) Multi-threaded pipeline execution scheme.

of these sets. Figure 5.6 (b) depicts how the executions of the three trees are interleaved. In this example three different trees (n, m and l) are executed. With this approach we have the same period that in the multi-cycle approach, and we can execute one node per cycle, as in the single-cycle implementation.

Figure 5.7 shows the final structure of the multi-threaded class module. As can be seen in the figure, the hardware overhead is small. We only need to include a few registers to store the information needed for each stage, the additional @_last_node counters, two additional registers which indicate the starting address of the first tree in each set, three 1-bit registers which store a flag indicating whether the processing of each set has been completed, and finally modify the control unit. Moreover, the memory resources, which are the most critical in our design, are the same in both versions.

The VHDL source codes of both versions are available in [15].

5.4 Experimental results

As explained in 2.3, in hyperspectral images pixel classification, the input is a single pixel composed of a series of features, where each feature is a 16-bit integer. The number of features depends on the sensor used to obtain the image, in our case we used datasets from 103 to 224 features. Each node of the tree selects one of these features, and compares it with the value stored in the node to make their decision; i.e. whether to select the left or the right child. Our design has been written in VHDL using generic parameters to be very customisable. Hence, it can be used for different

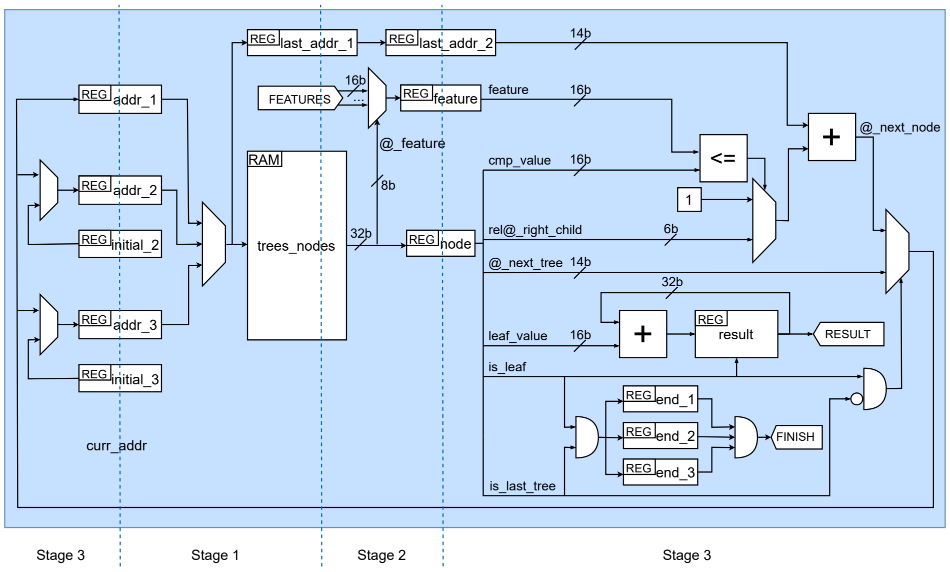


Figure 5.7: Multi-threading class diagram

images with different number of classes, of input features, of trees per class. For the evaluation of the accelerator, we used the models previously trained for Indian Pines (IP), Pavia University (PU), Kennedy Space Center (KSC) and Salinas Valley (SV), with the characteristics explained in 3.3. The models codified according to the described node representation format of Figure 5.1 are available in [15]. Table 5.1 shows the accuracy obtained executing the models with the LightGBM library, and the achieved accuracy with the equivalent models adapted for our accelerator. As can be seen in the table, the changes in the original decision trees almost have no effect on the final accuracy.

Table 5.1: Compared Top1 accuracy between LightGBM training and the accelerator.

	Accelerator	LightGBM
IP	0.802	0.805
KSC	0.894	0.894
PU	0.922	0.924
SV	0.926	0.928

The accelerator has been designed to fit in the FPGA of the Zedboard Xilinx Zynq-7000 evaluation board [10]. This is a very small FPGA with an old technology, so the needs of the design are very restrictive. The main goal of using this device is to try to reproduce the characteristics of the rad-hard and rad-tolerant FPGAs approved for embedded on-board devices. These devices do not use the latest integration technologies, hence it will not be realistic to use the latest generation of FPGAs.

Table 5.2 shows the hardware resources of single-cycle and multi-threading designs in this FPGA.

Table 5.2: Single-cycle vs multi-threading resources.

		Single-cycle	Multi-threading	incr.
IP	LUTs	15800 (29.70%)	17027 (32.01%)	1.08
	Flip-Flops	4470 (4.20%)	6250 (5.87%)	1.40
	F7 Muxes	6657 (25.03%)	6657 (25.03%)	1
	F8 Muxes	3328 (25.02%)	3328 (25.02%)	1
	BRAMs (36Kb)	128 (91.43%)	128 (91.43%)	1
KSC	LUTs	11749 (22.08%)	12641 (23.76%)	1.08
	Flip-Flops	3909 (3.67%)	5351 (5.03%)	1.37
	F7 Muxes	4578 (17.21%)	4577 (17.21%)	1
	F8 Muxes	2288 (17.20%)	2288 (17.20%)	1
	BRAMs (36Kb)	104 (74.29%)	104 (74.29%)	1
PU	LUTs	5228 (9.83%)	5714 (10.74%)	1.09
	Flip-Flops	2455 (2.31%)	3436 (3.23%)	1.40
	F7 Muxes	2016 (7.58%)	2016 (7.58%)	1
	F8 Muxes	864 (6.50%)	864 (6.50%)	1
	BRAMs (36Kb)	72 (51.43%)	72 (51.43%)	1
SV	LUTs	17462 (32.82%)	18628 (35.02%)	1.07
	Flip-Flops	4862 (4.57%)	6638 (6.24%)	1.37
	F7 Muxes	7681 (28.88%)	7681 (28.88%)	1
	F8 Muxes	3584 (26.95%)	3584 (26.95%)	1
	BRAMs (36Kb)	128 (91.43%)	128 (91.43%)	1

The only perceptible increment is the number of LUTs and Flip-Flops, the rest are null or negligible. These increments are not important for the design, since it uses a small percentage of these resources. As can be seen in the table, the bottleneck of the system are the on-chip memory resources. In fact, some images use almost all memory resources. For that reason it was very important to optimise the memory format used to store the trees.

Table 5.3 presents the performance results. In this case, the multi-threaded design provides a performance improvement of 67-85%. Hence, in this accelerator the computer architecture optimisations provide a major performance improvement with minimal area cost.

The Multi-threaded design has a small penalty, between 1% and 4%, in terms of the number of cycles needed to process each node (Avg. Cycles/Node), which leads to a higher number of cycles per pixel (Avg. Cycles/px.). The reason is that the trees are divided into three sets, which will not always be perfectly balanced. When one of the sets ends, only two of the three threads have useful work to do. To reduce this problem we grouped the trees taking into account their average depth, trying to make the workload similar in all groups. Since the depth of the trees depends on the path chosen, and will be different for each input, we can not guarantee a perfect balance, and these small penalties appear. Nevertheless, we achieved a very small penalty and the frequency increase is enough to reach better throughput.

Table 5.3: Single-cycle vs multi-threading throughput.

		Single-cycle	Multi-threading	Gain
	Freq. (MHz)	60.61	105.263	
	Avg. Cycles/px.	1658.15	1700.9	
IP	Avg. μs/px.	27.36	16.16	1.69
	Avg. px/s.	36552.78	61886.65	
	Avg. Cycles/Node	1	1.026	
	Freq. (MHz)	62.5	105.263	
	Avg. Cycles/px.	2542.76	2564.55	
KSC	Avg. μs/px.	40.69	24.36	1.67
	Avg. px/s.	24579.60	41045.41	
	Avg. Cycles/Node	1	1.009	
	Freq. (MHz)	66.67	125	
	Avg. Cycles/px.	1857.34	1938.70	
PU	Avg. μs/px.	27.86	15.51	1.80
	Avg. px/s.	35895.42	64476.20	
	Avg. Cycles/Node	1	1.044	
	Freq. (MHz)	55.56	105.263	
	Avg. Cycles/px.	1447.13	1479.37	
SV	Avg. μs/px.	26.05	14.05	1.85
	Avg. px/s.	38393.23	71153.94	
	Avg. Cycles/Node	1	1.022	

Regarding the communications, in our design we have included a DMA to read the input data from the external off-chip memory. With this approach the latency of sending the input data of one pixel is smaller than the computation itself, so it can be entirely overlapped with the processing time of the previous input data, and does not affect the throughput.

Table 5.4 shows the data provided by the VIVADO power consumption report. The dynamic power consumption of our design is between 0.402W and 0.745W, depending on the data set characteristics, hence, our accelerator provides high performance for embedded systems with a small power overhead.

Table 5.4: Power consumption.

	Static (W)	Dynamic (W)					
		Clocks	Signals	Logic	BRAMs	I/O	Total
IP	0.125(15%)	0.031(4%)	0.224(31%)	0.129(18%)	0.298(41%)	0.036(6%)	0.719(85%)
KSC	0.121(17%)	0.026(4%)	0.185(32%)	0.097(17%)	0.242(41%)	0.036(6%)	0.585(83%)
PU	0.115(22%)	0.020(5%)	0.090(22%)	0.051(13%)	0.199(49%)	0.042(11%)	0.402(78%)
SV	0.126(14%)	0.033(4%)	0.237(32%)	0.140(19%)	0.298(40%)	0.037(5%)	0.745(86%)

In order to evaluate the performance of our multi-threading design, we have executed the inference of those models in a high performance (HP) CPU, an Intel(R) Core(TM) i5-8400 CPU @ 2.80GHz, and also in an embedded system (ES) CPU, the Dual-core ARM Cortex-A9 @ 667 MHz that is part of the same system-on-a-chip as the FPGA in the Zedboard evaluation board [10]. Regarding the software, for the Intel i5-8400 we measured the execution of the LightGBM library inference, while for the Cortex-A9, we designed an equivalent C version that uses the same data format as the FPGA design, hence, it has the same computational and memory requirements. This code has been compiled with gcc 7.3.1 with -O3 optimisation level. Table 5.5 shows the total energy required for the execution of the entire test set for each image and also the performance in pixels per second. These measurements have been performed with a Yokogawa WT210 digital power meter, a device accepted by standard performance evaluation corporation (SPEC) for power efficiency benchmarks [11]. In the table we only include the dynamic energy consumption, i.e. the consumption due to the execution of the trees. To this end, we measured the power consumption of the FPGA, the Cortex-A9 and the i5-8400 during five minutes both in idle mode, and during the execution of the models, and from that measures we computed the average increase in the power consumption due to the execution of the trees in each case. The Power consumption measurements used to calculate the energy are $75W$ for the HP CPU, $1.5W$ for the ES CPU and $2W$ for the FPGA.

Table 5.5: Energy and performance comparisons.

		Energy (J)			Performance (pixels/s)		
	Test pixels	FPGA	HP CPU	ES CPU	FPGA	HP CPU	ES CPU
IP	8721	0.280	27.750	9.858	62293	23568	1327
KSC	4435	0.136	6.150	3.024	65221	53860	2200
PU	38503	1.194	58.500	15.960	64494	49336	3619
SV	48726	1.370	171.000	36.863	71133	21332	1983
Average		0.500	36.147	11.508	65706	22422	2139

According to these results, the HP CPU execution consumes in average 72 times more energy than the FPGA design to perform the inference of the test benches, while the FPGA design is twice faster. In the case of the ES CPU, it consumes 23 times more energy while the FPGA design is 30 times faster. The last comparison is interesting, because this processor is on the same chip as the FPGA, using the same main memory and a clock almost seven times faster. This demonstrates the benefits of a custom accelerator both for performance, and for energy efficiency.

Finally, one of our objectives was to process the pixels at the same speed at which they are generated by the hyperspectral sensors. According to [88] the AVIRIS sensor that was used to obtain most of our datasets must process 62873.6 pixels per second to fully achieve real-time performance. Our multi-threading design achieves this speed in three of the four images (KSC, PU and SV), and achieves 98.4% of that speed in a third one (IP). Hence, we can conclude that this design is capable of achieving real-time performance in many scenarios even using a small FPGA.

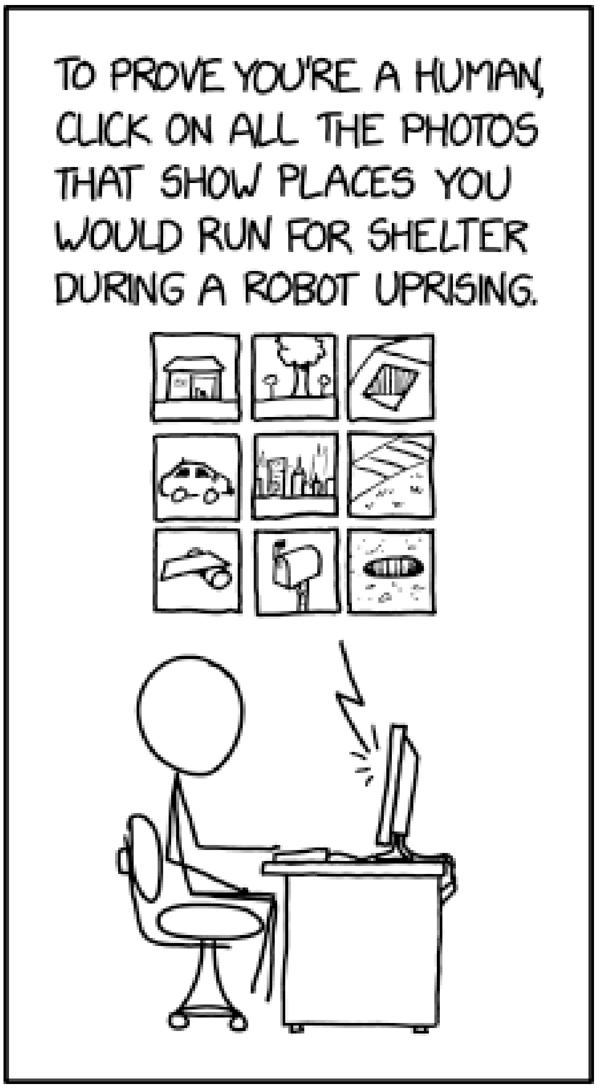
5.5 Conclusions

In this chapter we describe the architecture of a GBDTs accelerator implemented according to the great results that GBDTs obtained in the analysis of chapter 3.

According to the results of the analysis of ML techniques, we conclude than the characteristics of GBDTs make them specially indicated to be accelerated in FPGAs, since its basic operations are simple comparisons, so we developed an accelerator for them. A key factor in the design of the accelerator has been the memory restrictions. Optimising the format used to store the trees has been the key to allow complex models to be implemented in small FPGAs. Moreover, after analysing the execution of our accelerator, we have managed to find a pipeline and multi-threading execution scheme that maximises the utilisation of the FPGA resources and achieves 67-85% performance improvement compared to the single-cycle execution.

We have tested our accelerator with complex models for a relevant case-study, and we have demonstrated that it can be used to process data at run-time with a very small power-consumption overhead. Moreover, compared to the execution of LightGBM in a high performance CPU our model is capable of achieving 2x performance while consuming 72x less energy. In the case of an embedded system CPU, our design reaches a 30x performance improvement while maintaining 23x less energy consumption during the execution. Hence, this design is suitable to provide high-performance for embedded systems, that was our original target.

6 Step III: Bayesian Neural Networks for Hyperspectral Images


TO PROVE YOU'RE A HUMAN, CLICK ON ALL THE PHOTOS THAT SHOW PLACES YOU WOULD RUN FOR SHELTER DURING A ROBOT UPRISING.

— **xkcd.com**
(Just that guy, you know?)

6.1 Introduction

Machine Learning techniques have made spectacular advances in recent years. The improvements and refinement of the models used, together with hardware platforms that allow the exploration of increasingly complex models, trained on larger data sets, have enabled a vast improvement in the accuracy of the models, and opened up new fields of application.

In remote sensing, neural networks (NNs) and convolutional neural networks (CNNs) [78] have demonstrated to be one of the most useful tools for hyperspectral image classification, with many recent relevant publications [21, 36, 18, 49]. They provide state-of-the-art results, but they have some limitations. A NN model will provide very good results if it is trained with data similar to the data that it will process when deployed. However, if the training set includes data with high noise level, or it includes mislabelled data, or it is unbalanced, or the model, once in operation, receives inputs with different features, or different formats from those used in training, the predictions will not be reliable. This problem is not easy to solve since Machine Learning is based on the learn-from-data paradigm. Hence, the quality of the results of a NN model relies on the quality of the training data. However, we can deal with this problem in a more efficient way by enriching the models based on NNs or CNNs so that, together with their predictions, they report uncertainty metrics that can be used to identify these issues.

In this chapter, we will evaluate these possibilities for the problem of pixel classification of hyperspectral images. For this purpose, we will use a simple model based on NNs and upgrade it into a bayesian neural network (BNN), i.e. a model that combines neural network with Bayesian inference. BNNs use distributions to model weights and outputs. Hence, they will not generate a constant output for a given input, but a distribution, that can be analysed to measure the prediction uncertainty. Moreover, it allows to differentiate between the uncertainty generated by the model itself, and the uncertainty generated by the input data. This approach can be applied to NNs and CNNs in a similar way. We will use a simple NN model, instead of a large CNN because our goal is not to achieve the maximum accuracy, but to explore the utility of using BNNs instead of conventional NNs. This is orthogonal to the size of the model, hence, following the recommendations of the Green AI paradigm [26], we selected a model that can be trained fast with low energy consumption (we trained hundreds of different models during this research), and provides results that can be easily replicated by any researcher, without the need for specific hardware platforms to train the model.

We will demonstrate the utility of our approach, with several experiments applied to some of the most frequently used hyperspectral data sets. First, we will propose a scheme to test if the model is properly calibrated. Second, we will show how the uncertainty metric can be used to achieve a requested level of accuracy by discarding inputs that have very high levels of uncertainty. The goal is not to artificially increase the accuracy, but to identify scenarios with high uncertainty, in order to avoid using their outputs when making decisions. In fact, there are many situations in which it is better to discard the output of a network than to use an output with a high level of uncertainty thinking that it is correct. For example, the network may receive a pixel that do not belong to any of the categories, or a pixel which include a mix of them. It is critical to be able to identify such cases, either to simply ignore those outputs, mark them as unreliable outputs, use an alternative method to classify them, or to try to understand why our model fails in these situations. These situations occur in the data sets analysed, because neither the labelling, nor our models, nor the training process are perfect, but they will be more frequent if the models are used in the real world, because all kinds of new spectral signatures will appear in the input that will not belong to any of the trained categories, and therefore should not be classified. Third, we will analyse the uncertainty of the different categories of each data set. Finally, we will test the model response to two different experiments: training a new network mixing the labels of two classes, and the introduction of white noise during inference. The code used for the experiments and the trained models are available in a public repository so that they can serve as a baseline for applying this technique to other problems [13].

The results of this study have been published in a very prestigious international journal [12], and the codes to reproduce all the experiments are available in a public repository [13].

6.1.1 Motivational example

Using a simple neuron as an example we can easily illustrate the impact of bayesian models in our prediction abilities. For that, we are going to train a neuron using the training data represented on Table 6.1 that includes two different datasets, each of which has three training data for the same input $(0.5, 0.5)$. In both cases, for that input, a trained neuron will generate a function with an output about 0.7. A possible solution, is presented in Equation 6.1, where both weights are 0.7 and bias is 0, since it is the output that minimise the error for both data sets. But the inputs of datasets 1 and 2 are very different. In the first case, the training data is consistent, and the outputs are very similar for the input $(0.5, 0.5)$, while in the second case the outputs have a large variance. If the models generate the same output in both cases we will lose that information.

Table 6.1: Bayesian Neuron example inputs.

	Dataset 1			Dataset 2		
Data	Input1	Input2	Output	Input1	Input2	Output
First	0.5	0.5	0.6	0.5	0.5	0.2
Second	0.5	0.5	0.7	0.5	0.5	0.7
Third	0.5	0.5	0.8	0.5	0.5	1.2

$$O(I_1, I_2) = ReLU(I_1 \times 0.7 + I_2 \times 0.7 + 0) \tag{6.1}$$

The weights and bias of a bayesian neuron are not constant values, but distributions, as shown in Figure 6.1, so every time we call the function we will receive a different weight value according with that distribution. If we train this bayesian neuron for datasets 1 and 2, the mean value of the distributions on both models will be similar, and with the same value that in the previous cases, 0.7, but the deviation will be higher for case 2, because the distribution is wider, therefore, if we perform several inference passes over the two models, the output average will be around 0.7 on both of them, but we will be able to observe an important difference on the deviation of the outputs. For the first data set, all the inference passes will provide similar results, while for the second case, the output distribution will be wider, and the results will have larger divergence. Uncertainty metrics make it possible to differentiate between these two situations.

6.2 Related work

As explained in the introduction, NNs have demonstrated to be one of the most accurate techniques for hyperspectral image classification. CNNs can exploit both the spectral and the spatial information of hyperspectral image applying convolution filters and pooling operations to the input data. Based on the output of the previous layer, each new layer generate more complex feature vectors, which are finally processed by one or several layers of fully connected neurons. Deep CNNs achieve state of the art classification accuracy in general for image classification, and in particular for hyperspectral images. In [21] the authors analyse different machine learning methods evaluating its accuracy, its size, and the computations required. In

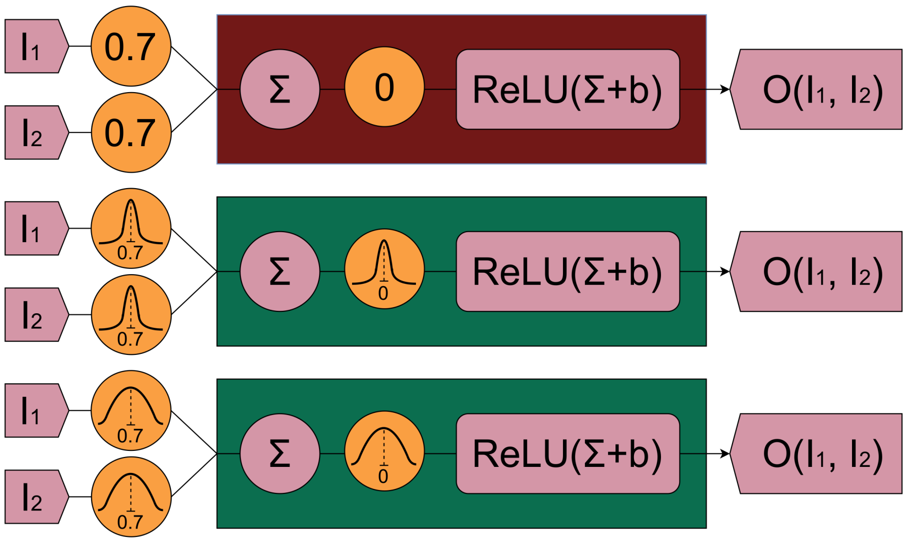


Figure 6.1: weights of normal neuron for both datasets (top) versus bayesian neuron on dataset 1 (centre) and dataset 2 (bottom).

this analysis CNNs achieve the best accuracy. The works described in [36, 14, 18, 49] are examples of state-of-the-art results using CNNs.

However, using deep CNNs for hyperspectral is very complex, as it requires a great amount of labelled training data for tuning their large number of parameters, and in remote sensing labelled samples are very difficult and expensive to collect. Moreover, the high dimensionality of the hyperspectral data further complicates the setting of the deep model parameters. This may lead to overfitting in many networks. These networks will provide good results for the training data, and for the test data if it is very similar, but their results are not generalisable for new data. This is a very complex issue, the training data can be improved by adding more samples, or by using data augmentation techniques to generate additional training samples by performing several transformation to the original data set [86].

In this context, BNNs can provide added value with their ability to generate uncertainty estimations. However, very few works have explored the possibilities of using BNNs for hyperspectral pixel classification. The most representative is the work presented in [46]. Their objective is to achieve higher classification accuracies for situations with scarce labelled data. To this end, they propose to combine BNNs with active learning for the problem of hyperspectral pixel classification. They start the training process with very few labelled pixels and then use a BNN model to select and introduce in the training set some of the unlabelled pixels in order to improve the results. Their Bayesian approach is dropout based, following the main idea described in [75].

Another work that uses BNNs for hyperspectral images was presented in [29], although in this case, they are not used to classify pixels, but to monitor water quality from an unmanned aerial vehicle. In this case, they use a BNN because it allows using a single model to generate multiple outputs, thus emulating an ensemble of models working together, but they do not measure uncertainty.

These previous works focus on increasing accuracy, which is, of course, an important goal. However, if we only look at accuracy, the advantages of using BNNs are not clear, since any result obtained with a BNN could also be obtained with an ensemble of NNs that carries out the same computations. Nevertheless, in order to use NNs for real-world problems, the reliability of the predictions is as important as accuracy itself [17], and the uncertainty metrics offered by BNNs can be the way to improve reliability [56, 20]. Therefore, we believe that it is very important to analyse the possibilities offered by BNNs to achieve more reliable neural networks in this field. Sometimes, for classification problems, softmax layers results, i.e. the output of the last layer of the NN, have been wrongly interpreted as a measure of the uncertainty of the output, but as explained in [51, 41], they cannot be interpreted in such way because their probabilities distribution are not relative to the uncertainty of the model on their predictions. Therefore, specific uncertainty metrics are needed. A good example of the utility of these metrics is [41]. In this work, the authors use the uncertainty metrics of a BNN to prove that multi-spectral images offer better reliability on classification problems than RGB images.

In this chapter, we want to analyse the opportunities offered by the uncertainty metrics provided by BNNs for the problem of hyperspectral image pixel classification. To illustrate them, we developed some experiments that demonstrate their added value.

6.3 Bayesian networks and uncertainty quantification

Neural networks (NNs) have proved to be very powerful techniques to perform image classification tasks, and they also achieved great results with hyperspectral images as we exposed in last section. Nevertheless, there are still some generalisation issues, such as overfitting, that could specially affect those areas where the datasets are scarce and difficult to generate, as remote sensing hyperspectral imaging. Probabilistic models such as bayesian neural networks allow us to analyse the model uncertainty for a given prediction due to their stochastic behaviour.

Bayesian approaches model probability distributions to express the uncertainty over the unobserved data. For that, the model starts with a prior distribution of probabilities that updates according to the observed data. After training, the generated model should represent the uncertainty about each parameter value. So, instead of weights, the calculated values to represent the network are random variables initialised with a prior distribution $p(\mathbf{w})$, and the training will consist on calculating the posterior $p(\mathbf{w}|\mathbf{D})$, where $\mathbf{D}$ represents the observed data $\mathbf{D} = \{\mathbf{y}, \mathbf{x}\}$ [105, 76, 30].

However, computing the exact posterior $p(\mathbf{w}|\mathbf{D})$ for big and complicated models as NNs is intractable. Hence, bayesian neural networks are based in approximate models. During the training phase, variational Bayesian methods are used to approximate intractable integrals. One of the most commonly used is variational inference (VI), which tries to approximate the parameters ϕ of a variational distribution $q_{\phi}(\mathbf{w})$ to minimise its Kullback-Leibler (KL) with $p(\mathbf{w}|\mathbf{D})$ [105, 30]. Theoretically, distributions could have any shape, but it is impossible to analyse the search space for that case. To solve this issue, only symmetric and tractable distributions are used. Gaussian functions are typically used for this purpose because they are defined with only two parameters, which simplifies the training process, and allows BNNs to be more compact. Therefore, compared to regular NNs, BNNs based on Gaussian functions will have twice as many parameters. This is an important overhead for large models, but, as explained in the previous section, a BNN can be seen as an ensemble of models, since each inference will generate a different output. Large NN ensembles are frequently used to reduce the variance and improve the accuracy of the output. Hence, a single BNN can replace the complete ensemble and lead to important reductions in the size of the model.

In this chapter we want to analyse the uncertainty of the predictions. Since BNNs are probabilistic models, executing T stochastic inferences over the same test dataset, will provide T different output predictions. This allows to measure the model uncertainty, but also to separately quantify aleatoric uncertainty, which is related to the quality of the dataset itself, and epistemic uncertainty, which is related to our trained model. For that analysis we will use the predictive entropy ($\mathbb{H}$), expected entropy ($\mathbb{E}_p$) and mutual information (MI) of the T resultant predictions.

Let K be the number of classes of the dataset and $\{\mathbf{c}_k\}_{k=1}^{K}$ the set of class labels. Predictive entropy ($\mathbb{H}$) represents the overall uncertainty of the model within the range $[0, \log(K)]$. We already defined $\mathbf{D}$ as the observed data, $\mathbf{w}$ as the calculated variables of the model and T as the number of stochastic forward passes. The value of $\mathbb{H}$ is given by Eq. 6.2, where $a_t = p(\mathbf{y} = \mathbf{c}_k|\mathbf{x}, \mathbf{w}_t)$ [17].

$$\mathbb{H}\left(\mathbf{y}|\mathbf{x},\mathbf{D}\right) := -\sum_{k=1}^{K}\left(\frac{1}{T}\sum_{t=1}^{T}a_t\right)\log\left(\frac{1}{T}\sum_{t=1}^{T}a_t\right) \tag{6.2}$$

Expected entropy ($\mathbb{E}_p$), given by Eq. 6.3, represents the aleatoric uncertainty. This variable can be used to analyse the dataset characteristics and will be a baseline to determine how training data alterations can affect the model [17]. High aleatoric uncertainty values indicate ambiguities in the data that can be caused by noise, mislabelled or misrepresented data, overlapping categories, or any other issues that make the dataset difficult to learn.

$$\mathbb{E}_{p(\mathbf{w}|\mathbf{D})}\left[\mathbb{H}\left(\mathbf{y}|\mathbf{x},\mathbf{w}\right)\right] := \frac{1}{T}\sum_{t=1}^{T}\left(-\sum_{k=1}^{K}a_t\log\left(a_t\right)\right) \tag{6.3}$$

Mutual information (MI) captures the epistemic uncertainty of the model and will give us information about uncertainty generated by model. The MI value is given by Eq. 6.4 and we will often refer to it as $\mathbb{H} - \mathbb{E}_p$ [17]. High epistemic uncertainty values indicate that the model is not properly trained or that it is not the right model.

$$MI\left(\mathbf{y},\mathbf{w}|\mathbf{x},\mathbf{D}\right) := \mathbb{H}\left(\mathbf{y}|\mathbf{x},\mathbf{D}\right) - \mathbb{E}_{p(\mathbf{w}|\mathbf{D})}\left[\mathbb{H}\left(\mathbf{y}|\mathbf{x},\mathbf{w}\right)\right] \tag{6.4}$$

6.4 Datasets and model characteristics

We selected for our tests five of the most used hyperspectral datasets, Botswana (BO) [2], Indian Pines (IP) [2], Kennedy Space Center (KSC) [2], Pavia University (PU) [2] and Salinas Valley (SV) [2]. Section 2.3 describes the main characteristics of the datasets.

6.4.1 Model Training

For the purposes of this research we trained for every image a NN model: a Multilayer Perceptron (MLP) with two hidden layers, the first one with 32 nodes and the second one with 16 nodes. The framework used to generate and train the networks is TensorFlow [79] with DenseFlipout layers, which implement bayesian variational inference with Flipout estimator, and we used as *kd_function* parameter the *kl_divergence* function divided by the number of labelled pixels on the dataset and *ReLU* as the activation function [8, 53]. In a previous work [21], we performed a grid search to identify the optimal size of a neural network, among other models, for each different hyperspectral image, and we found that the best results, without overfitting, ranged from 80 neurons to 140 depending on the image. However, since the goal of this research is not to identify the optimal NN configuration, but to explore the additional possibilities offered by a Bayesian network, we selected a smaller model, with only 48 neurons because it achieved results almost as good as a model with twice as many neurons, but it can be trained much faster. We used the 50% of the pixels of each class for training, with 10% reserved for validation to detect overfitting, and the other 50% for testing. We used an initial learning rate of 0.01 for every dataset. We trained the models for 50000 epochs storing the intermediate states every 100 epochs. At the end, we analysed the accuracy obtained in each of those intermediate states with the validation set, and chosen the ones with the best accuracy results. Among them, we chose those with the lowest average uncertainty. And among those with similar uncertainties, we chose the model that had been trained the least number of epochs. The training information is summarised in Table 6.2. We used *categorical_crossentropy* loss function on an Adam optimiser. During inference, we execute 100 bayesian passes to extract the uncertainty information and to average the predictions. The entire code is available on [13].

Table 6.2: Training data.

				Accuracy		**Uncertainty**	
Image	Train %	LR	Epochs	Train	Test	Train	Test
BO	50%	0.01	17000	0.94	0.91	0.24	0.26
IP	50%	0.01	22000	0.90	0.86	0.33	0.35
KSC	50%	0.01	41000	0.93	0.92	0.24	0.26
PU	50%	0.01	1800	0.95	0.95	0.16	0.17
SV	50%	0.01	4000	0.93	0.93	0.17	0.17

6.4.2 Model Calibration

Before using a model, it is necessary to evaluate its quality. Typically, this is done by analysing the accuracy of the test results. However, accuracy should not be the only concern. If we want a reliable model, we must also verify that it is well calibrated, this can be evaluated using a Reliability Diagram [56, 80, 98].

To construct a diagram for the test results of a BNN we propose a simple scheme based on defining a bin for each range of output values (from 0 to 1). For example we can define 10 bins, to store values ranging from $[0\%, 10\%)$, from $[10\%, 20\%)$, and so on. The output values will be interpreted as probabilities, and the diagram provides feedback about the accuracy of theses values. For example, a value of 0.25 would indicate that there is a 25% chance that the class is correct. What our diagram tries to check is whether, on average, these probabilities correspond to what is observed when verifying the data obtained with the test data set. What we expect for a well calibrated model is that there will be a correspondence between the average accuracy of the elements assigned to a bin, and the range of values assigned to that bin. For example, the outputs assigned to the bin ranging from $[0\%, 10\%)$ should correspond to the correct class between 0% and 10% of the time.

On every stochastic pass, the network output for each pixel will be a distribution of K probabilities, one for each class. We then calculate the average value of the T stochastic passes, so we will have one final prediction of K probabilities for each pixel. These probabilities are grouped in bins. After that, each of them is evaluated, and we compute the accuracy of each bin as the number of correct predictions divided by the number of total predictions assigned to the bin.

Figure 6.2 shows the results of the calibration of our model for the five datasets. We can observe that the curve of every image is very close to a perfect calibration. From these results we can infer that the result of averaging the T stochastic passes generates a well calibrated model.

6.5 Experimental results

6.5.1 Accuracy vs uncertainty

BNNs can be used to generate a stronger, and properly calibrated model, using the average of several stochastic passes, in the same way that ensembles of NNs are used for the same purpose. This is important, but it does not use one of the most interesting features of BNNs, the uncertainty information. In this experiment, we want to explore the relationship between uncertainty and accuracy. If there is correlation between them, once our network is already trained and tested, it will be possible to define an uncertainty threshold in order to filter out predictions with very high uncertainty values. As explained in the introduction, the objective is not to artificially increase the accuracy, but to allow our model to identify scenarios with high uncertainty, in order to avoid using their outputs when making decisions. If the level of uncertainty exceeds the set threshold, it means that our network is

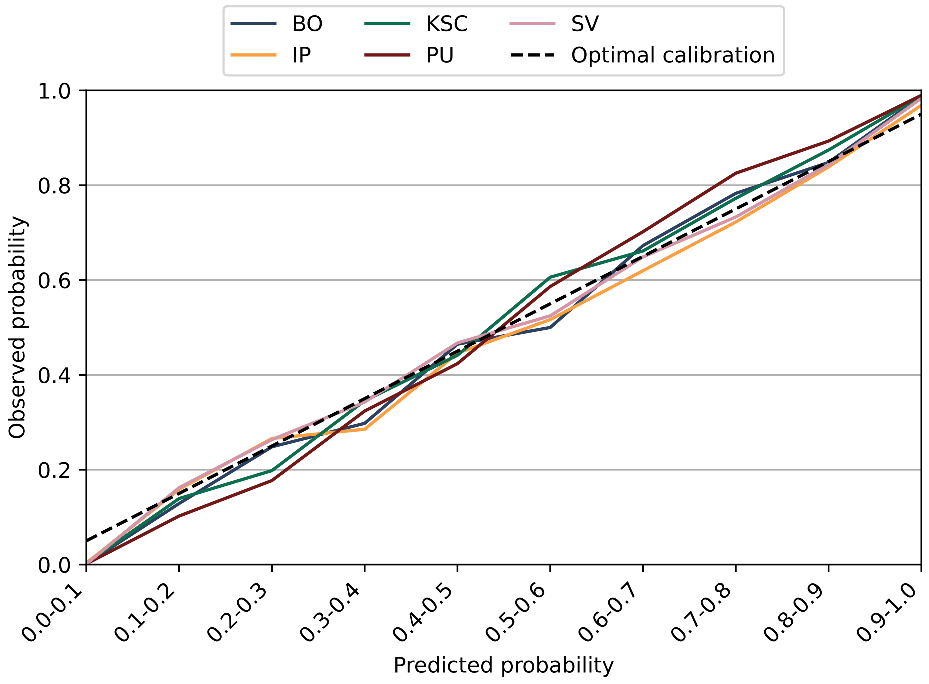


Figure 6.2: Reliability Diagram.

not able to provide an output with the required quality. This can be due to many reasons. A clear example would be that the input does not belong to any of the trained categories, or is a mixture of several of them. Other more complex cases will be discussed later.

The upper part of Figure 6.3 depicts the relationship between accuracy and uncertainty for the five datasets. As we can observe in the chart, those outputs with less than 0.1 of uncertainty achieve almost 100% of accuracy, and those with uncertainty values between 0.3 and 0.4 still maintain an accuracy of more than 90% on every image. Therefore, we can adjust the uncertainty threshold to obtain the precision we need.

The distribution of pixels with respect to the uncertainty values is also very important because setting an uncertainty threshold that optimises the accuracy will not be useful if we discard most of the pixels. At the bottom of Figure 6.3 can be seen that most of the pixels have an uncertainty value of less than 0.1 and the distribution is clearly decreasing. It can also be seen that the distribution of pixels is quite different in each dataset. In the following experiment we want to analyse this in detail, examining each of its classes separately.

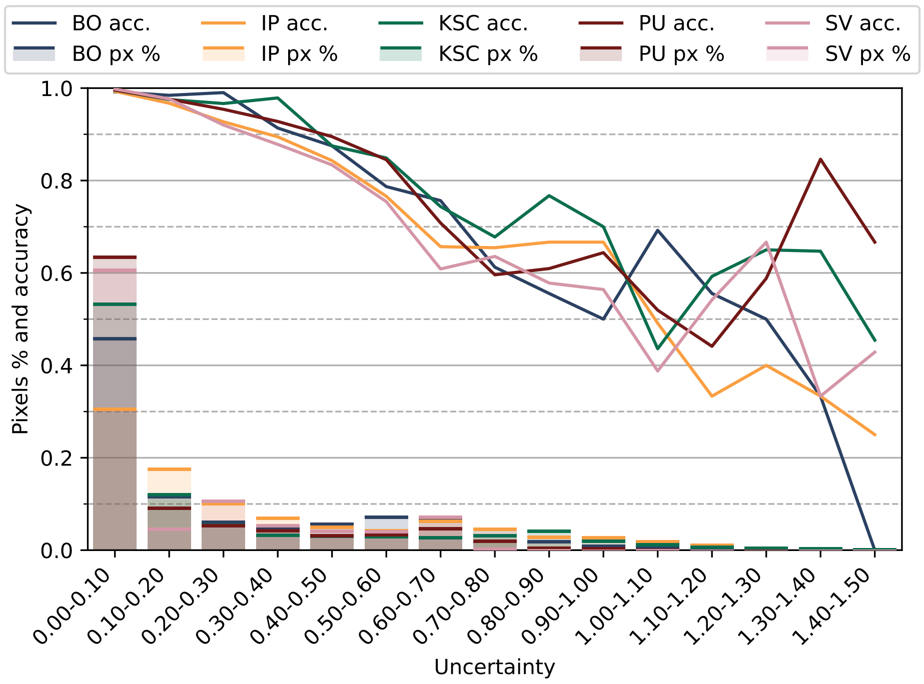


Figure 6.3: Percentage of pixels and accuracy along uncertainty groups.

6.5.2 Class uncertainty

Figures 6.4 to 6.8 represent the uncertainty values ($\mathbb{H}$) on a comfortable scale divided into two categories: $\mathbb{E}_p$, the aleatoric uncertainty that measures the uncertainty for the input data, and $\mathbb{H}-\mathbb{E}_p$, the epistemic uncertainty that measures the uncertainty of the model. The average ($\mathbb{H}$) values are between 0.2 and 0.4, which is between 7.5% and 14.5% of their theoretical maximum, as explained in Section 6.3, the uncertainty range is $[0, \log(K)]$, where K is the number of classes. We can also observe that in all the cases most of the uncertainty is due to the data. This information can be used to determine whether the selected model should be improved, either by changing it, or by training it for a longer period, or whether the results are constrained by the quality of the dataset itself. In this case using a deeper model, or training the model longer may improve the results, but only marginally, as the main source of uncertainty is the data used.

We can also observe that, on every image, there are significant differences between classes, meaning that some of them are more difficult to identify. That could be caused because there are classes with similar features, mislabelled pixels, or because the labelled pixels of a particular class are not enough to determine all their characteristics. For example, the uncertainty of class 8 of IP is 3.5 times higher than average, indicating that there is a very clear problem with the data in that

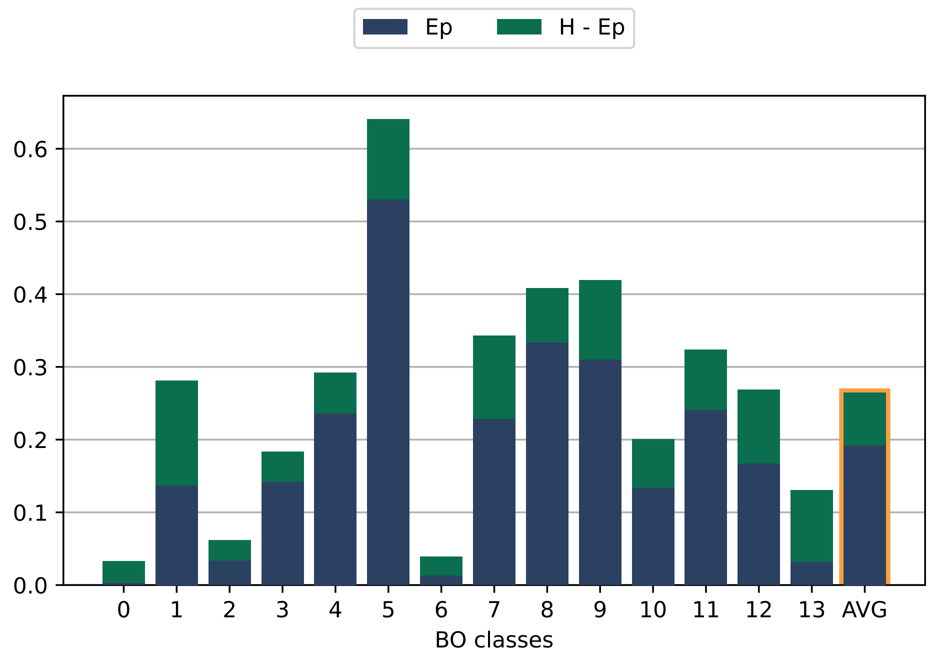


Figure 6.4: BO class uncertainty.

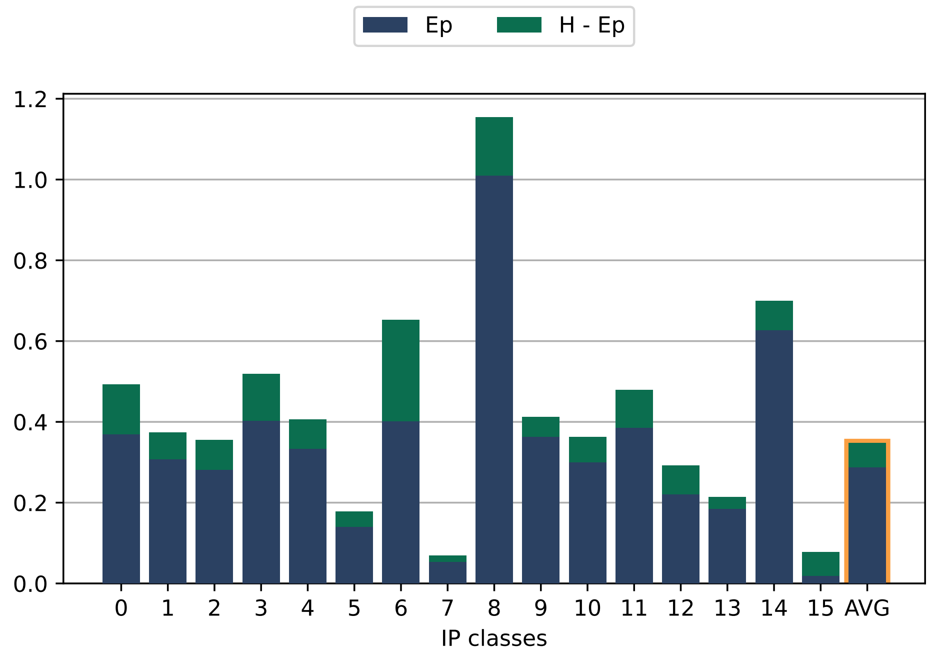


Figure 6.5: IP class uncertainty.

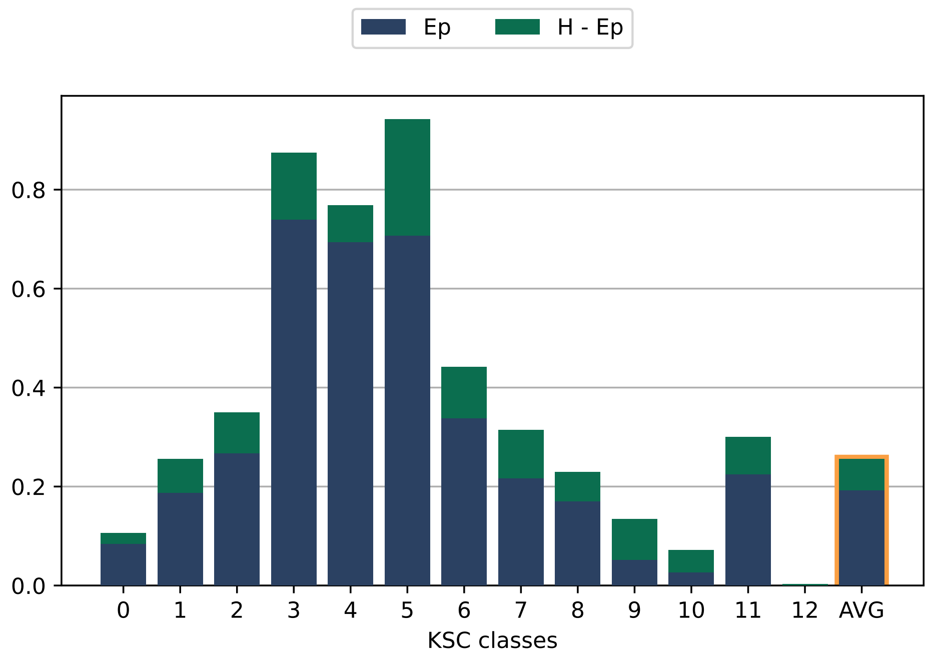


Figure 6.6: KSC class uncertainty.

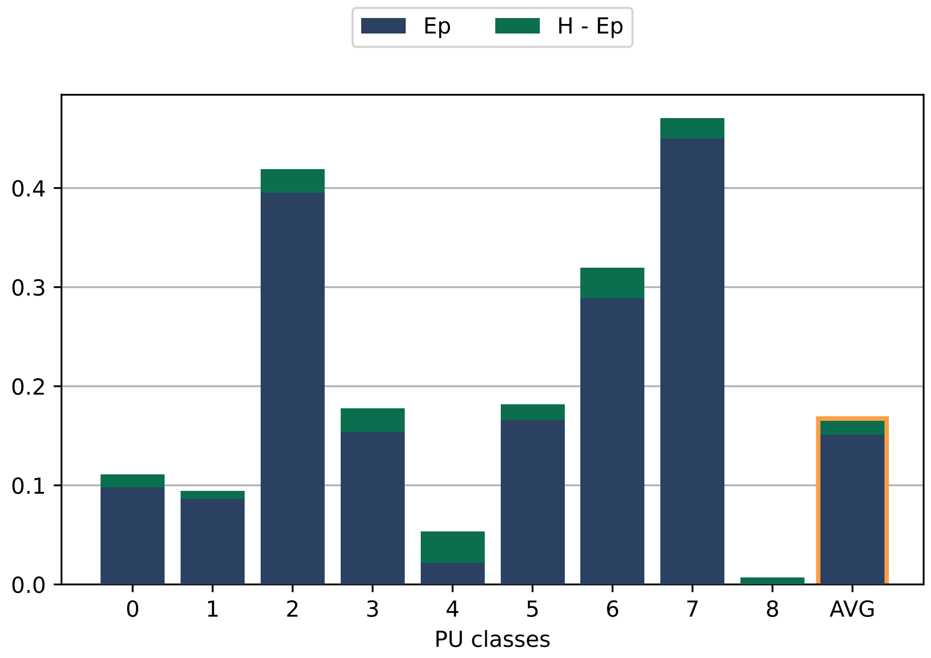


Figure 6.7: PU class uncertainty.

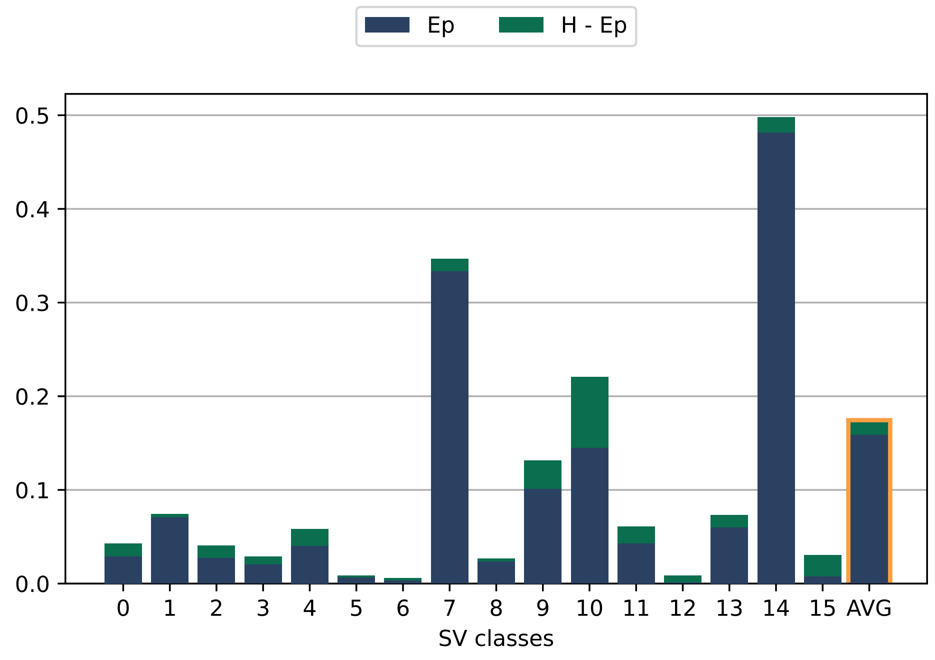


Figure 6.8: SV class uncertainty.

class. In KSC several classes duplicate the average entropy. In the description of the dataset they already warned of this problem since they have identified that certain vegetation types have similar spectral signatures. In SV classes 7 and 14 are clearly harder to identify for the BNN than the rest of the classes, and the same applies to PU classes 2 and 4. This analysis also identifies which classes work particularly well, reporting small values in the uncertainty metrics. An intensive analysis of a particular image, their classes and characteristics is out of the scope of our study, but here we can see how bayesian networks can help to identify problems in the dataset itself.

6.5.3 Uncertainty maps

A very interesting use of the information given by bayesian networks is the possibility of studying the uncertainty maps of the entire image. That give us very valuable information about how well our labelled data represents the entire scene, i.e., our prediction capabilities across the entire scene. Figures 6.9 to 6.13 show, for each dataset, an RGB representation based on the algorithm provided by [24], the graphical representation of the averaged bayesian prediction for the entire scene, the uncertainty map and the ground truth to compare with the labelled classes. The colours codification for all the images is shown in Table 6.3.

Table 6.3: Colours codification.

Classes		Ranges of uncertainty	
class 0	🟥	0.0 - 0.1	🟩
class 1	🟩	0.1 - 0.2	🟩
class 2	🟨	0.2 - 0.3	🟩
class 3	🟦	0.3 - 0.4	🟨
class 4	🟧	0.4 - 0.5	🟧
class 5	🟦	0.5 - 0.6	🟧
class 6	🟪	0.6 - 0.7	🟥
class 7	🟥	0.7 - 0.8	🟥
class 8	🟩	0.8 - 0.9	🟪
class 9	🟪	0.9 - 1.0	🟪
class 10	🟫	1.0 - 1.1	🟪
class 11	⬜	1.1 - 1.2	🟪
class 12	🟥	1.2 - 1.3	🟦
class 13	🟩	1.3 - 1.4	🟦
class 14	🟦	1.4 - 1.5	🟦
class 15	⬛		

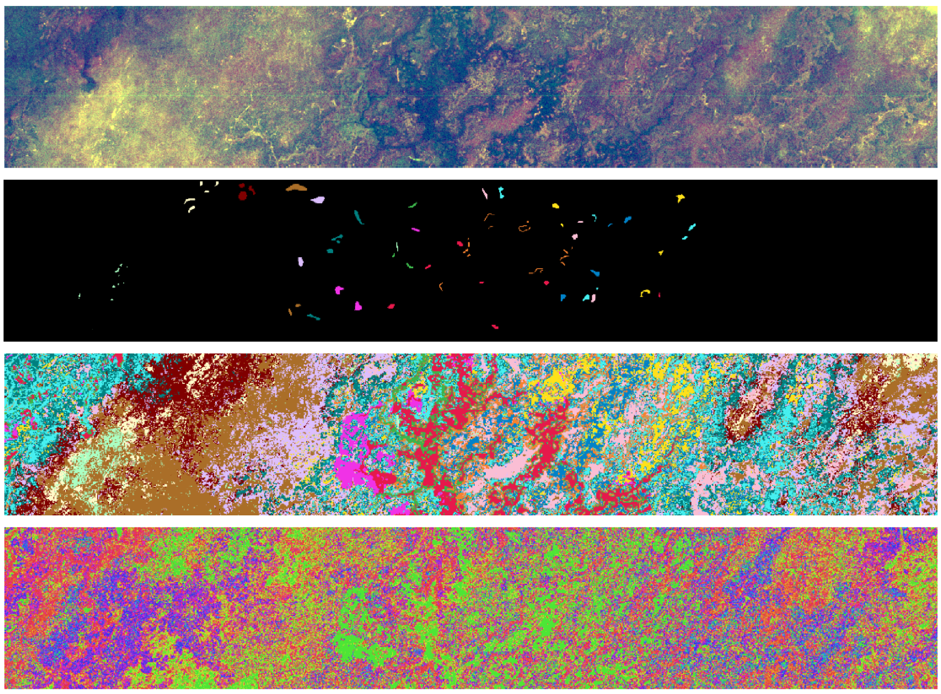


Figure 6.9: From top to bottom: BO RGB representation, BO ground truth, prediction and uncertainty maps.

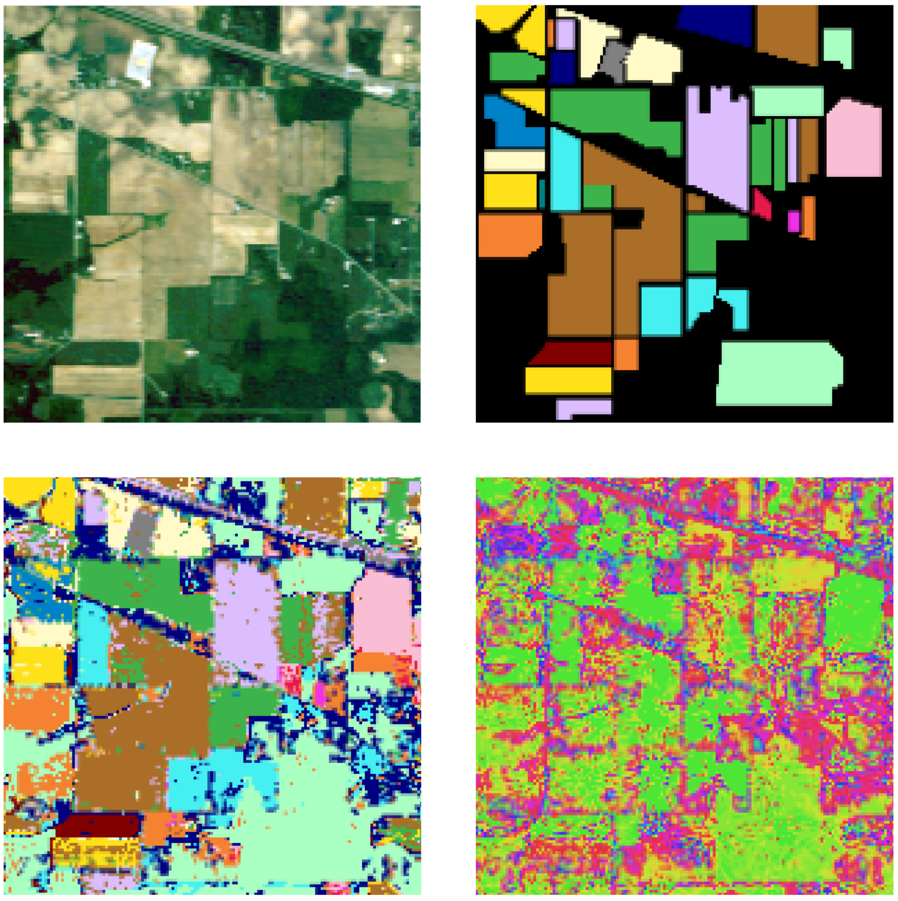


Figure 6.10: From top-left to bottom-right: IP RGB representation, IP ground truth, prediction and uncertainty maps.

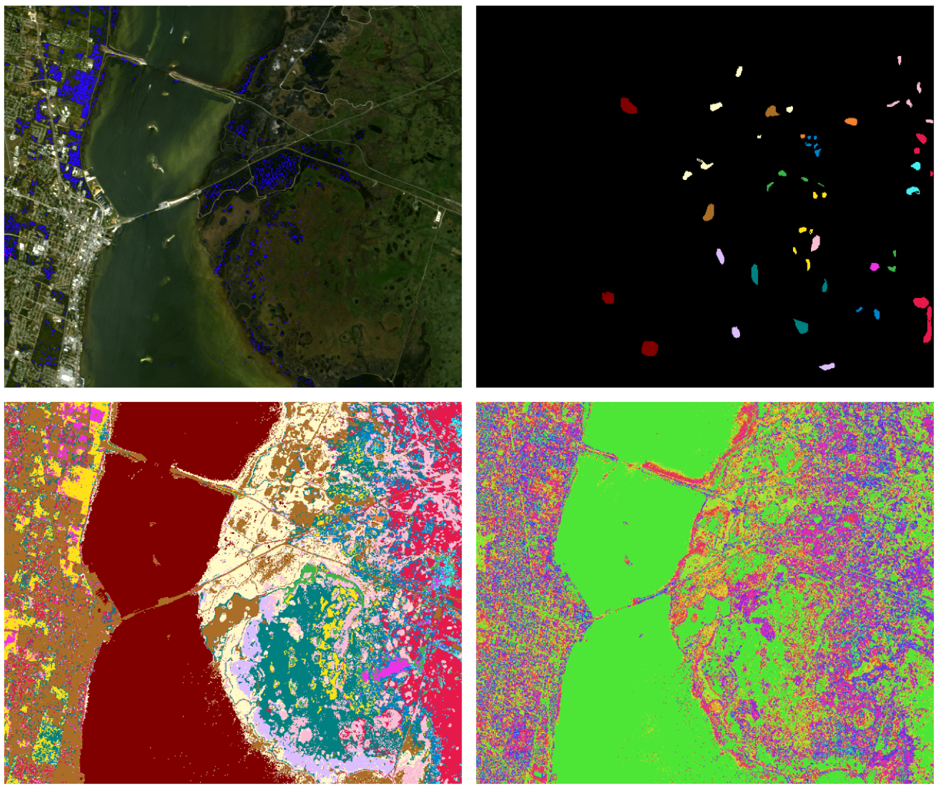


Figure 6.11: From top-left to bottom-right: KSC RGB representation, KSC ground truth, prediction and uncertainty maps.

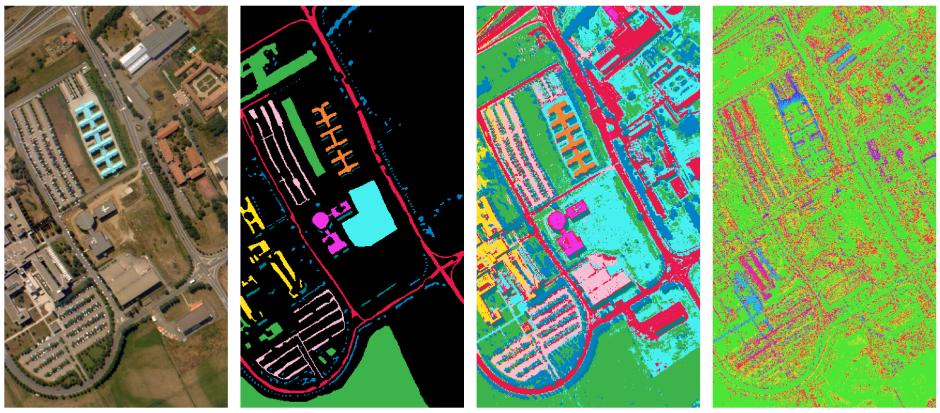


Figure 6.12: From left to right: PU RGB representation, PU ground truth, prediction and uncertainty maps.

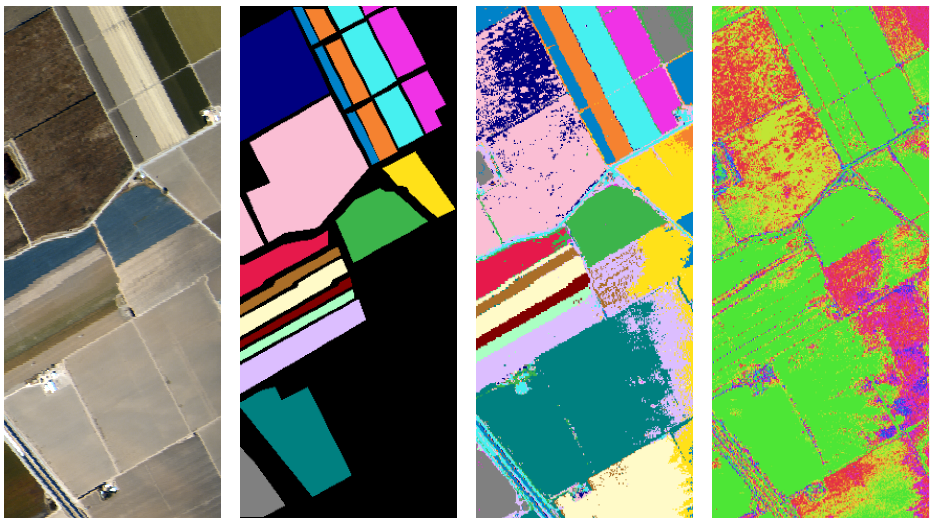


Figure 6.13: From left to right: SV RGB representation, SV ground truth, prediction and uncertainty maps.

One of the first things to notice looking at the uncertainty maps is that borders and limits are well defined an differentiated from the rest of the scene. That is expected and perfectly understandable, as this borders can imply zones with mixed classes, due to the great size of the surface that each pixel represents, or even paths or other adjacent non-labelled elements. This not only gives information about the terrain, but, being an expected behaviour, serves as a proof of the capability of this uncertainty metrics to show us whether the network is being capable of recognise some pixels or not.

Comparing with the ground truth, we can also appreciate how some of the classes are easier to recognise for the network than others. A clear example is shown in Figure 6.13. In SV, for most of the classes the accuracy is very high and the uncertainty very low. However, as previously mentioned, the model has problems in identifying class 14. As can be seen in the prediction map, it often confuses it with class 7. In these situations the uncertainty metric warns us by increasing its value as can be seen in the uncertainty map.

It is also interesting to analyse the results of the non-labelled regions. In some cases, the model clearly identifies that they belong to one of the trained classes, whereas for other regions, it provides a prediction with a high value of uncertainty, indicating that its output is not reliable. This is a useful feature in order to use the model in real-world applications, since it may receive inputs that do not belong to any of the the trained categories.

6.5.4 Mixed classes

As a proof of concept, we are going to perform two particular tests with our datasets, the first one will be to train new networks after mixing the pixels of two classes on each training dataset in order to measure its effect on the aleatoric uncertainty for these two classes. The second one will be to feed the initial BNNs with data with an increasing level of random noise and to analyse analyse its effect on the uncertainty metrics.

For the first experiment we selected two classes with a similar number of pixels for each image, shown in Table 6.4, and we aleatory mixed their labels in the training set. For example in BO, half of the 269 pixels of class 4 and half of the 269 pixels of class 5 were selected for training, but before training their labels were randomly mixed up in such a way that half of the training pixels labelled with a 4 belonged to class 5 and vice versa. Since the categories are completely mixed, the model will not be able to distinguish between them. The objective of the experiment is to test if the model identifies this problem by increasing the aleatoric uncertainty.

Table 6.4: Selected classes and number of pixels.

	First class		**Second class**	
Image	Class number	Pixels	Class number	Pixels
BO	4	269	5	269
IP	2	830	5	730
KSC	8	520	11	503
PU	3	3064	7	3682
SV	1	3726	6	3579

Table 6.5 shows the values of the aleatoric uncertainty of the BNN trainings with the mixed data ($Ep\ mixed$) compared to the same network trained with the initial data (Ep). As it was expected, the increase in the aleatoric uncertainty of the modified classes is very important. This behaviour tell us about how the expected entropy values of the different classes can be used to analyse the datasets characteristics and determine a possible lack of information for some of them.

6.5.5 Uncertainty with Noise

Another interesting application of the bayesian networks is the detection of degradation in the quality of the input data, that can be generated due to an increase in the level of noise generated during the data acquisition or the communications.

Table 6.5: Mixed classes aleatoric uncertainty (Ep).

	First class		Second class		All classes average	
Image	Ep	Ep mixed	Ep	Ep mixed	Ep	Ep mixed
BO	0.23	0.81	0.53	0.89	0.19	0.28
IP	0.28	0.98	0.14	0.83	0.29	0.50
KSC	0.17	0.93	0.23	0.92	0.19	0.41
PU	0.15	0.83	0.45	1.02	0.15	0.29
SV	0.07	1.12	0.00	0.70	0.16	0.34

The idea is that if the level of noise in the inputs increases, the uncertainty metrics should be able to detect it.

For simulating this possible situations we progressively introduced random noise on the test bench and observe the uncertainty values. To generate the noise we added to each feature of each pixel a random 16 bits signed integer value multiplied by a noise factor, which we progressively incremented by 0.01 until the uncertainty converge to high values. The normalised values of the uncertainty are shown in Figure 6.14, where each coloured line corresponds to the average uncertainty value normalised to the theoretical maximum uncertainty value of each dataset, which depends on the number of categories.

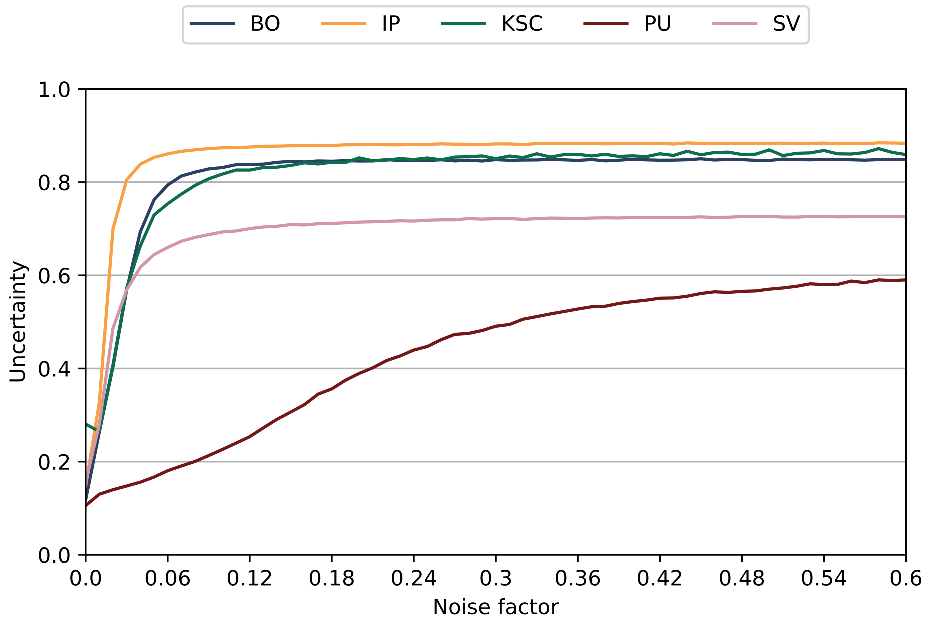


Figure 6.14: Uncertainty level, normalised to the maximum uncertainty of each dataset, when increasing noise.

As we can observe, while we increment the noise factor the uncertainty value grows. It is not our intention here to try to determine a *measure* of the noise, as this will depend on the noise source and the sensor and image characteristics, while we just used random generic noise added to the data. Analysing the figures, we can see that BO, IP, KSC and SV exhibit a similar behaviour and its uncertainty grows very fast from the beginning until it stabilises at around 0.04 noise factor. On the contrary, Pavia University (PU) dataset seems to be much more resistant to noise than the rest of datasets, with constant slower growth. Being robust to noise depends on many factors such as data structure (number of observations, number of predictors, number of classes) and model complexity [50].

The results illustrate the utility of the uncertainty metrics to verify the quality of the input data. In the context of Remote Sensing, this is a very important feature, as the sensors and the data received are exposed to many external variations, such as climatic ones.

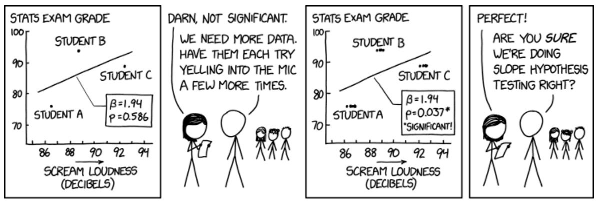
6.6 Conclusion

In this chapter we wanted to evaluate how BNN models can help us to obtain calibrated and reliable solutions for a relevant problem: pixel classification of hyperspectral images. To this end, we trained a simple BNN for five of the most used hyperspectral images datasets and then explored the additional information that BNNs provide. First, we analysed the calibration of our model, which is an orthogonal task that can be done with any NN model, but is usually neglected. In the case of BNNs, using several stochastic passes contributes to achieve a good calibration. If network outputs are going to be used for decision-making, checking that the network is well calibrated should be a mandatory step. Next, we analysed the reliability of our models using the uncertainty metrics given by the probabilistic nature of the BNNs. The clear correlation between accuracy and uncertainty makes it possible to identify the outputs that provide a requested level of accuracy by selecting an appropriate uncertainty threshold. The output of the BNNs allows the uncertainty to be decomposed into two categories: aleatoric, caused by the data, and epistemic, caused by the model. Analysing the epistemic uncertainty we can verify that our model is appropriate for the given problem and is well trained, and with the aleatoric uncertainty we can analyse the quality of the dataset themselves, and also the quality of the data received during inference. A close exploration of the data sets showed that these datasets have important uncertainty differences between classes, which indicates that some of the classes are either miss-represented on the dataset, or they are too similar, or mislabelled. This can be helpful to identify that some data classes need more samples, or that two very similar categories should be unified, since their features are indistinguishable.

In order to further illustrate the added value of using BNNs, we performed two different experiments. First, we trained new BNNs after mixing the pixels of two classes on each training dataset. This leads to a large increase in uncertainty. Hence, the BNN was able to identify that there were problems with these classes. Second, we used the original BNNs and feed them with data with an increasing level of random noise. In the experiments, we observed that uncertainty works as an excellent measure of the input data quality during inference, giving us important information about possible noise factors or communication interference.

After these experiments, we believe that the benefits of upgrading models from NNs to BNNs are very clear. BNNs provide uncertainty metrics that can be used to identify problems in their outputs, or to evaluate the quality of the training data. The tools for designing BNNs have improved a lot in recent years, and some of the most widely used design environments, such as Tensorflow or Pytorch, include them. On the downside, the models used need twice as many parameters as an equivalent NN model. This can make it difficult to train very large models. This issue is interesting, and we would like to study it in the future, but in any case, we believe it is better to have a smaller but very reliable model, than a very large deep model that slightly improves the results, but does not provide uncertainty metrics.

7 Conclusions



— **xkcd.com**
(Just that guy, you know?)

In this thesis we analysed some Machine Learning (ML) challenges, searching for reliable ML techniques and efficient computing from the point of view of embedded systems. For each one of them we made different contributions, all of them published in relevant international journals.

We analysed the inference process of Multinomial Logistic Regression (MLR), Random Forest (RF), Support Vector Machine (SVM), Multi-Layer Perceptron (MLP), Convolutional Neural Network (CNN) and Gradient Boosting Decision Trees (GBDT). Thanks to this analysis we get to know better this techniques and decided to implement an accelerator for CNNs, for being the most used and accurate, and another one for GBDTs, for presenting the best trade-off between the use of computational and hardware resources and the obtained accuracy levels.

This analysis is published in an international journal [21].

We developed an accelerator for DNNs, specially focused on CNNs, capable of taking advantage of the sparsity by avoiding all useless operations, i.e. with zero as one of the operands, and managing filter compression, both for convolutional and fully-connected layers. It keeps almost peak utilisation of arithmetic resources, even in highly-sparse scenarios.

We also proposed a novel comparison between similar-in-area dense and sparse architectures in order to identify in which scenarios including support for sparsity is superior to providing a dense architecture with additional arithmetic resources. Our results show that the benefits of exploiting sparsity are clear on 32-bit arithmetic, whereas on 8-bit it is hard to profitably exploit sparsity given the sparsity of current state-of-the-art CNNs. While, for 16-bit arithmetic, adding support for sparsity improves energy efficiency as long as the useful operations are under 50%, and also performance, when the useful operations are under 25%.

The design and implementation of this accelerator, along with the similar-in-area comparison proposal have been published in an international journal [22] and the correspondent code is available in a public repository [44].

We developed another accelerator for GBDTs achieving 2x performance while consuming 72x less energy compared to the execution of LightGBM in a high performance CPU. Compared to an embedded system CPU, our design reaches a 30x performance improvement while maintaining 23x less energy consumption during the execution.

The design and implementation of this accelerator has been published in [16], and the codes of the accelerator are available in a public repository [15].

We carried out an extensive analysis of the benefits of using the uncertainty information given by bayesian networks in the context of hyperspectral images classification. Thanks to this analysis we found that bayesian networks may help us to identify problems in the outputs, or to evaluate the quality of the training datasets. Even more, they also allow us to carry out inference in very complex contexts where some of the received inputs may not be related to the characteristics of the training dataset.

The results of this study have been published in [12], and the codes to reproduce all the experiments are available in a public repository [13].

7.0.1 Future Work

Our last analysis was focused on bayesian neural networks (BNNs) in the context of hyperspectral images (HI) classification. It shows how BNNs can be very useful to address some of the HI classification challenges. Taking that on account, our future work, already in process, will consist on the development of an specific accelerator for BNNs.

From this study we extract at least two possible future areas of interest:

- Generate a bayesian accelerator, capable of generate the stochastic weights on-the-fly during inference.

- Work around the idea of bayesian trees and the possibility of adapting our GBDTs accelerator to accept bayesian ones.

7.1 Conclusiones (spanish)

En esta tesis hemos analizado algunos retos del *Machine Learning* (ML), buscando técnicas de ML fiables, así como métodos eficientes de calcularlas desde el punto de vista de sistemas empotrados. Para cada uno de ellos hemos realizado diferentes aportaciones, todas ellas publicadas en revistas de impacto internacional.

Hemos analizado el proceso de inferencia de técnicas como *Multinomial Logistic Regression* (MLR), *Random Forest* (RF), *Support Vector Machine* (SVM), *Multi-Layer Perceptron* (MLP), *Convolutional Neural Network* (CNN) y *Gradient Boosting Decision Trees* (GBDT). Gracias a este análisis hemos podido caracterizar estas técnicas y decidido implementar un acelerador para CNNs, por ser las más utilizadas y precisas, y otro para GBDTs, por presentar el mejor *trade-off* entre el uso de recursos computacionales y de hardware y los niveles de *accuracy* obtenidos.

Este análisis está publicado en una revista interncional [21].

Hemos desarrollado un acelerador para DNNs, especialmente enfocado a las CNNs, capaz de aprovechar la *sparsity* evitando todas las operaciones inútiles, es decir, con cero como uno de los operandos, y gestionando la compresión de los filtros, tanto para las capas convolucionales como para las *fully-connected*. Mantiene una utilización casi máxima de los recursos aritméticos, incluso en escenarios de alta dispersión de datos.

También hemos propuesto un novedoso método de comparación entre arquitecturas densas y dispersas que se preocupa de mantener la similitud en área, con el fin de identificar en qué escenarios incluir soporte para la dispersión de datos es preferible a dotar a una arquitectura densa de recursos aritméticos adicionales. Nuestros resultados muestran que los beneficios de explotar la dispersión de datos son claros en la aritmética de 32 bits, mientras que en la de 8 bits es difícil explotarla de forma rentable dada la dispersión de las CNNs actuales. Mientras que, para la aritmética de 16 bits, añadir soporte para dispersión de datos mejora la eficiencia energética siempre que las operaciones útiles sean inferiores al 50%, y también el rendimiento, cuando las operaciones útiles son inferiores al 25%.

El diseño y la implementación de este acelerador, junto con la propuesta de comparación similar-en-área han sido publicados en una revista internacional [22] y el código correspondiente está disponible en un repositorio público [44].

Desarrollamos otro acelerador para GBDTs logrando el doble de rendimiento y consumiendo 72 veces menos energía en comparación con la ejecución de LightGBM en una CPU de alto rendimiento. En comparación con una CPU de un sistema empotrado, nuestro diseño alcanza una mejora de rendimiento de 30x manteniendo un consumo de energía 23x menor durante la ejecución.

El diseño y la implementación de este acelerador también se ha publicado en [16], y los códigos del acelerador están disponibles en un repositorio público [15].

Hemos llevado a cabo un extenso análisis de las ventajas de utilizar la información de la incertidumbre dada por las redes bayesianas en el contexto de la clasificación de imágenes hiperespectrales. Gracias a este análisis encontramos que las redes bayesianas pueden ayudarnos a identificar problemas en las salidas, o a evaluar la calidad de los conjuntos de datos de entrenamiento. Es más, también nos permiten realizar inferencia en contextos muy complejos donde algunas de las entradas recibidas pueden no estar relacionadas con las características del conjunto de datos de entrenamiento.

Los resultados de este estudio han sido publicados en [12], y los códigos para reproducir todos los experimentos están disponibles en un repositorio público [13].

7.1.1 Trabajo Futuro

Nuestro último análisis se centró en las redes neuronales bayesianas (BNNs) en el contexto de la clasificación de imágenes hiperespectrales (HI). Muestra cómo las BNN pueden ser muy útiles para abordar algunos de los retos de la clasificación de HI. Teniendo en cuenta esto, nuestro trabajo futuro, ya en proceso, consistirá en el desarrollo de un acelerador específico para BNNs.

De este estudio extraemos al menos dos posibles áreas futuras de interés:

- Diseñar un acelerador bayesiano, capaz de generar los pesos estocásticos al vuelo durante la inferencia.
- Trabajar en torno a la idea de los árboles bayesianos y la posibilidad de adaptar nuestro acelerador de GBDTs para aceptar árboles bayesianos.

Bibliography

[1] Chris Deotte. *IEEE-CIS Fraud Detection contest 1st Place Solution*. [online] https://www.kaggle.com/c/ieee-fraud-detection/discussion/111284. Accessed January 2021 (cit. on p. 72).

[2] GIC. *Hyperspectral Remote Sensing Scenes, Grupo de Inteligencia Computacional de la Universidad del País Vasco*. [online] http://www.ehu.eus/ccwintco/index.php/Hyperspectral_Remote_Sensing_Scenes. Accessed January 2022 (cit. on pp. 10, 30, 94).

[3] Scikit Learn. *Ensemble methods. Forests of randomized trees*. [Online] https://scikit-learn.org/stable/modules/ensemble.html#forest. Accessed November 2019 (cit. on p. 23).

[4] Scikit Learn. *Generalized Linear Models. Logistic Regression*. [Online] https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression. Accessed November 2019 (cit. on p. 20).

[5] Scikit Learn. *Support Vector Machines. Mathematical formulation*. [Online] https://scikit-learn.org/stable/modules/svm.html#svm-mathematical-formulation. Accessed November 2019 (cit. on p. 24).

[6] Microsoft. *LightGBM documentation*. [online] https://lightgbm.readthedocs.io. Accessed November 2019 (cit. on p. 23).

[7] PyTorch. *PyTorch Docs. Neural Network*. [Online] https://pytorch.org/docs/stable/nn.html#module-torch.nn. Accessed November 2019 (cit. on pp. 26, 27).

[8] TensorFlow. *TensorFlow DenseFlipout layer documentation*. https://www.tensorflow.org/probability/api_docs/python/tfp/layers/DenseFlipout. [Online; accessed August 2021] (cit. on p. 95).

[9] Xilinx. *Zynq UltraScale+ MPSoC ZCU104 Evaluation Kit, SoC and FPGA platform*. https://www.xilinx.com/products/boards-and-kits/zcu104.html (cit. on p. 61).

[10] Xilinx. *Zynq-7000, SoC and FPGA platform*. [online] https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf (cit. on pp. 9, 72, 81, 84).

[11] Yokogawa. *WT210/WT230 Digital Power Meters*. http://tmi.yokogawa.com/products/digital-power-analyzers/digital-power-analyzers/wt210wt230-digital-powermeters/#tm-wt210_01.htm. Accessed November 2019 (cit. on pp. 61, 84).

[12]Adrián Alcolea and Javier Resano. “Bayesian neural networks to analyse hyperspectral datasets using uncertainty metrics”. In: *IEEE Transactions on Geoscience and Remote Sensing* (2022). DOI: 10.1109/TGRS.2022.3205119 (cit. on pp. 4, 89, 112, 114).

[13]Adrián Alcolea and Javier Resano. *BNN for hyperspectral datasets analysis*. https://github.com/universidad-zaragoza/BNN_for_hyperspectral_datasets_analysis. Accessed on February 2022. 2022 (cit. on pp. 4, 89, 95, 112, 114).

[14]Juan M. Haut, Adrian Alcolea, Mercedes E. Paoletti, et al. “GPU-Friendly Neural Networks for Remote Sensing Scene Classification”. In: *IEEE Geoscience and Remote Sensing Letters* 19 (2022), pp. 1–5. DOI: 10.1109/LGRS.2020.3019378 (cit. on p. 91).

[15]Adrián Alcolea and Javier Resano. *FPGA accelerator for GBDT. Source code and models*. https://github.com/AdrianAlcolea/FPGA_accelerator_for_GBDT. 2021 (cit. on pp. 4, 72, 80, 81, 112, 114).

[16]Adrián Alcolea and Javier Resano. “FPGA Accelerator for Gradient Boosting Decision Trees”. In: *Electronics* 10.3 (2021). ISSN: 2079-9292. DOI: 10.3390/electronics10030314. URL: https://www.mdpi.com/2079-9292/10/3/314 (cit. on pp. 4, 72, 112, 114).

[17]Umang Bhatt, Javier Antorán, Yunfeng Zhang, et al. “Uncertainty as a Form of Transparency: Measuring, Communicating, and Using Uncertainty”. In: *Proceedings of the 2021 AAAI/ACM Conference on AI, Ethics, and Society*. AIES ’21. Association for Computing Machinery, 2021, pp. 401–413. ISBN: 9781450384735. DOI: 10.1145/3461702.3462571. URL: https://doi.org/10.1145/3461702.3462571 (cit. on pp. 92–94).

[18]Danfeng Hong, Lianru Gao, Jing Yao, et al. “Graph Convolutional Networks for Hyperspectral Image Classification”. In: *IEEE Transactions on Geoscience and Remote Sensing* 59.7 (2021), pp. 5966–5978. DOI: 10.1109/TGRS.2020.3015157 (cit. on pp. 88, 91).

[19]Simin Li, Yulan Lin, Tong Zhu, et al. “Development and external evaluation of predictions models for mortality of COVID-19 patients using machine learning method”. In: *Neural Computing and Applications* (2021). ISSN: 1433-3058. DOI: https://doi.org/10.1007/s00521-020-05592-1 (cit. on p. 72).

[20]Bofan Song, Sumsum Sunny, Shaobai Li, et al. “Bayesian deep learning for reliable oral cancer image classification”. In: *Biomed. Opt. Express* 12.10 (Oct. 2021), pp. 6422–6430. DOI: 10.1364/BOE.432365. URL: http://opg.optica.org/boe/abstract.cfm?URI=boe-12-10-6422 (cit. on p. 92).

[21]Adrián Alcolea, Mercedes E. Paoletti, Juan M. Haut, Javier Resano, and Antonio Plaza. “Inference in Supervised Spectral Classifiers for On-Board Hyperspectral Imaging: An Overview”. In: *Remote Sensing* 3 (2020). ISSN: 2072-4292. DOI: 10.3390/rs12030534. URL: https://www.mdpi.com/2072-4292/12/3/534 (cit. on pp. 3, 18, 88, 90, 95, 111, 113).

[22]Adrián Alcolea Moreno, Javier Olivito, Javier Resano, and Hortensia Mecha. “Analysis of a Pipelined Architecture for Sparse DNNs on Embedded Systems”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 28.9 (2020), pp. 1993–2003. DOI: 10.1109/TVLSI.2020.3005451 (cit. on pp. 3, 47, 112, 114).

[23] Taiga Ikeda, Kento Sakurada, Atsuyoshi Nakamura, Masato Motomura, and Shinya Takamaeda-Yamazaki. “Hardware/Algorithm Co-optimization for Fully-Parallelized Compact Decision Tree Ensembles on FPGAs”. In: *Applied Reconfigurable Computing. Architectures, Tools, and Applications*. Ed. by Fernando Rincón, Jesús Barba, Hayden K. H. So, Pedro Diniz, and Julián Caba. Springer International Publishing, 2020, pp. 345–357 (cit. on p. 73).

[24] Magnus Magnusson, Jakob Sigurdsson, Sveinn Eirikur Armansson, et al. “Creating RGB Images from Hyperspectral Images Using a Color Matching Function”. In: *IGARSS 2020 - 2020 IEEE International Geoscience and Remote Sensing Symposium*. 2020, pp. 2045–2048. DOI: 10.1109/IGARSS39084.2020.9323397 (cit. on pp. 10, 101).

[25] Microsoft. *Machine Learning Challenge Winning Solutions*. https://github.com/microsoft/LightGBM/blob/master/examples/README.md. 2020 (cit. on pp. 72, 73).

[26] Roy Schwartz, Jesse Dodge, Noah A Smith, and Oren Etzioni. “Green AI”. In: *Communications of the ACM* 63.12 (2020), pp. 54–63 (cit. on p. 88).

[27] Maxim Shepovalov and Venkatesh Akella. “FPGA and GPU-based acceleration of ML workloads on Amazon cloud - A case study using gradient boosted decision tree library”. In: *Integration* 70 (2020), pp. 1–9. ISSN: 0167-9260. DOI: https://doi.org/10.1016/j.vlsi.2019.09.007 (cit. on p. 74).

[28] Tianning Zhang, Weihuan He, Hui Zheng, et al. “Satellite-based ground PM2.5 estimation using a gradient boosting decision tree”. In: *Chemosphere* (2020), p. 128801. ISSN: 0045-6535. DOI: https://doi.org/10.1016/j.chemosphere.2020.128801. URL: http://www.sciencedirect.com/science/article/pii/S0045653520329994 (cit. on p. 72).

[29] Yishan Zhang, Lun Wu, Huazhong Ren, Licui Deng, and Pengcheng Zhang. “Retrieval of Water Quality Parameters from Hyperspectral Images Using Hybrid Bayesian Probabilistic Neural Network”. In: *Remote Sensing* 12.10 (2020). ISSN: 2072-4292. DOI: 10.3390/rs12101567. URL: https://www.mdpi.com/2072-4292/12/10/1567 (cit. on p. 92).

[30] Javier Antorán Cabiscol. “Understanding Uncertainty in Bayesian Neural Networks”. In: *Master of Philosophy (University of Cambridge)* (2019) (cit. on p. 93).

[31] Shijie Cao, Chen Zhang, Zhuliang Yao, et al. “Efficient and Effective Sparse LSTM on FPGA with Bank-Balanced Sparsity”. In: *Proceedings of the 2019 ACM/SIGDA ISFPGA*. FPGA ’19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 63–72 (cit. on p. 48).

[32] Y. Chen, T. Yang, J. Emer, and V. Sze. “Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices”. In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9.2 (2019), pp. 292–308 (cit. on pp. 48, 61).

[33] María Díaz, Raúl Guerra, Pablo Horstrand, et al. “Real-time hyperspectral image compression onto embedded GPUs”. In: *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing* 12.8 (2019), pp. 2792–2809 (cit. on p. 18).

[34]Cong Li, Lianru Gao, Antonio Plaza, and Bing Zhang. “FPGA implementation of a maximum simplex volume algorithm for endmember extraction from remotely sensed hyperspectral images”. In: *Journal of Real-Time Image Processing* 16.5 (2019), pp. 1681–1694 (cit. on p. 18).

[35]J. Li, S. Jiang, S. Gong, et al. “SqueezeFlow: A Sparse CNN Accelerator Exploiting Concise Convolution Rules”. In: *IEEE Transactions on Computers* 68.11 (Nov. 2019), pp. 1663–1677 (cit. on p. 48).

[36]Shutao Li, Weiwei Song, Leyuan Fang, et al. “Deep Learning for Hyperspectral Image Classification: An Overview”. In: *IEEE Transactions on Geoscience and Remote Sensing* 57.9 (2019), pp. 6690–6709. DOI: 10.1109/TGRS.2019.2907932 (cit. on pp. 88, 91).

[37]A. Mirzaeian, H. Homayoun, and A. Sasan. “TCD-NPE: A Re-configurable and Efficient Neural Processing Engine, Powered by Novel Temporal-Carry-deferring MACs”. In: *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. 2019, pp. 1–8 (cit. on p. 46).

[38]Ali Mirzaeian, Houman Homayoun, and Avesta Sasan. “NESTA: Hamming Weight Compression-Based Neural Proc. Engine”. In: *CRR* eprint arXiv:1910.00700 (2019). URL: http://arxiv.org/abs/1910.00700 (cit. on p. 46).

[39]ME Paoletti, JM Haut, J Plaza, and A Plaza. “Deep learning classifiers for hyperspectral imaging: A review”. In: *ISPRS Journal of Photogrammetry and Remote Sensing* 158 (2019), pp. 279–317 (cit. on p. 32).

[40]Liudmila Prokhorenkova, Gleb Gusev, Aleksandr Vorobev, Anna Veronika Dorogush, and Andrey Gulin. *CatBoost: unbiased boosting with categorical features*. 2019. arXiv: 1706.09516 [cs.LG] (cit. on p. 72).

[41]Jacob J. Senecal, John W. Sheppard, and Joseph A. Shaw. “Efficient Convolutional Neural Networks for Multi-Spectral Image Classification”. In: *2019 International Joint Conference on Neural Networks (IJCNN)*. 2019, pp. 1–8. DOI: 10.1109/IJCNN.2019.8851840 (cit. on p. 92).

[42]Rui Sun, Guanyu Wang, Wenyu Zhang, Li-Ta Hsu, and Washington Ochieng. “A gradient boosting decision tree based GPS signal reception classification algorithm”. In: *Applied Soft Computing* 86 (Nov. 2019). DOI: 10.1016/j.asoc.2019.105942 (cit. on p. 72).

[43]Alessandro Aimar, Hesham Mostafa, Enrico Calabrese, et al. “NullHop: A Flexible Convolutional Neural Network Accelerator Based on Sparse Representations of Feature Maps”. In: *IEEE transactions on neural networks and learning systems* 30.3 (2018), pp. 644–656 (cit. on pp. 48, 69).

[44]Adrián Alcolea, Javier Olivito, and Javier Resano. *Pipelined Architecture for Sparse DNNs*. https://gitlab.unizar.es/jolivito/pipelined_architecture_for_sparse_DNNs. Accessed on February 2022. 2018 (cit. on pp. 3, 47, 50, 112, 114).

[45]Juan M Haut, Sergio Bernabé, Mercedes E Paoletti, et al. “Low–High-Power Consumption Architectures for Deep-Learning Models Applied to Hyperspectral Image Classification”. In: *IEEE Geoscience and Remote Sensing Letters* 16.5 (2018), pp. 776–780 (cit. on p. 18).

[46] Juan Mario Haut, Mercedes E. Paoletti, Javier Plaza, Jun Li, and Antonio Plaza. “Active Learning With Convolutional Neural Networks for Hyperspectral Image Classification Using a New Bayesian Approach”. In: *IEEE Transactions on Geoscience and Remote Sensing* 56.11 (2018), pp. 6440–6461. DOI: 10.1109/TGRS.2018.2838665 (cit. on p. 91).

[47] Kartik Hegde, Jiyong Yu, Rohit Agrawal, et al. “UCNN: Exploiting Computational Reuse in Deep Neural Networks via Weight Repetition”. In: *Proceedings of the 45th ISCA*. ISCA ’18. IEEE Press. June 2018, pp. 674–687 (cit. on p. 48).

[48] Sparsh Mittal. “A Survey of FPGA-based Accelerators for Convolutional Neural Networks”. In: *Neural Computing and Applications* (Sept. 2018), pp. 1–31 (cit. on p. 47).

[49] M.E. Paoletti, J.M. Haut, J. Plaza, and A. Plaza. “A new deep convolutional neural network for fast hyperspectral image classification”. In: *ISPRS Journal of Photogrammetry and Remote Sensing* 145 (2018). Deep Learning RS Data, pp. 120–147. ISSN: 0924-2716. DOI: https://doi.org/10.1016/j.isprsjprs.2017.11.021. URL: https://www.sciencedirect.com/science/article/pii/S0924271617303660 (cit. on pp. 88, 91).

[50] Alby D. Rocha, Thomas A. Groen, Andrew K. Skidmore, Roshanak Darvishzadeh, and Louise Willemen. “Machine Learning Using Hyperspectral Data Inaccurately Predicts Plant Traits Under Spatial Dependency”. In: *Remote Sensing* 10.8 (2018). ISSN: 2072-4292. DOI: 10.3390/rs10081263. URL: https://www.mdpi.com/2072-4292/10/8/1263 (cit. on p. 108).

[51] Murat Sensoy, Lance Kaplan, and Melih Kandemir. “Evidential deep learning to quantify classification uncertainty”. In: *arXiv preprint arXiv:1806.01768* (2018) (cit. on p. 92).

[52] Stylianos I. Venieris, Alexandros Kouris, and Christos-Savvas Bouganis. “Toolflows for Mapping Convolutional Neural Networks on FPGAs: A Survey and Future Directions”. In: *ACM Computing Surveys* 51.3 (June 2018), 56:1–56:39 (cit. on p. 47).

[53] Yeming Wen, Paul Vicol, Jimmy Ba, Dustin Tran, and Roger Grosse. “Flipout: Efficient Pseudo-Independent Weight Perturbations on Mini-Batches”. In: *International Conference on Learning Representations*. 2018. URL: https://openreview.net/forum?id=rJNpifWAb (cit. on p. 95).

[54] Y. Chen, T. Krishna, J. S. Emer, and V. Sze. “Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks”. In: *IEEE Journal of Solid-State Circuits* 52.1 (2017), pp. 127–138 (cit. on pp. 48, 51).

[55] Aurélien Géron. *Hands-on machine learning with Scikit-Learn and TensorFlow: concepts, tools, and techniques to build intelligent systems*. O’Reilly Media, Inc, Mar. 2017. ISBN: 9781491962282 (cit. on pp. 21, 23).

[56] Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q. Weinberger. “On Calibration of Modern Neural Networks”. In: *Proceedings of the 34th International Conference on Machine Learning*. Ed. by Doina Precup and Yee Whye Teh. Vol. 70. Proceedings of Machine Learning Research. PMLR, Aug. 2017, pp. 1321–1330. URL: https://proceedings.mlr.press/v70/guo17a.html (cit. on pp. 92, 95).

[57] Guolin Ke, Qi Meng, Thomas Finley, et al. "LightGBM: A Highly Efficient Gradient Boosting Decision Tree". In: *Advances in Neural Information Processing Systems 30*. Ed. by I. Guyon, U. V. Luxburg, S. Bengio, et al. Curran Associates, Inc., 2017, pp. 3146–3154. URL: `http://papers.nips.cc/paper/6907-lightgbm-a-highly-efficient-gradient-boosting-decision-tree.pdf` (cit. on p. 72).

[58] Moritz B. Milde, Daniel Neil, Alessandro Aimar, Tobi Delbruck, and Giacomo Indiveri. "ADaPTION: Toolbox and Benchmark for Training Convolutional Neural Networks with Reduced Numerical Precision Weights and Activation". In: *CRR* eprint arXiv:1711.04713 (2017). URL: `http://arxiv.org/abs/1711.04713` (cit. on p. 46).

[59] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, et al. "SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks". In: *Proceedings of the 44th ISCA*. ISCA '17. IEEE. 2017, pp. 27–40 (cit. on pp. 48, 49, 61, 69).

[60] V. Sze, Y. Chen, T. Yang, and J. S. Emer. "Efficient Processing of Deep Neural Networks: A Tutorial and Survey". In: *Proceedings of the IEEE* 105.12 (Dec. 2017), pp. 2295–2329 (cit. on p. 46).

[61] Tien-Ju Yang, Yu-Hsin Chen, and Vivienne Sze. "Designing Energy-Efficient Convolutional Neural Networks using Energy-Aware Pruning". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2017, pp. 5687–5695 (cit. on p. 47).

[62] J. Yu, A. Lukefahr, D. Palframan, et al. "Scalpel: Customizing DNN pruning to the underlying hardware parallelism". In: *Proceedings of the 44th ISCA*. ISCA '17. 2017, pp. 548–560 (cit. on p. 48).

[63] Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. "Incremental Network Quantization: Towards Lossless CNNs with Low-Precision Weights". In: *CRR* eprint arXiv:1702.03044 (2017). URL: `http://arxiv.org/abs/1702.03044` (cit. on p. 46).

[64] Chenzhuo Zhu, Song Han, Huizi Mao, and William J. Dally. "Trained Ternary Quantization". In: *CRR* eprint arXiv:1612.01064 (2017). URL: `http://arxiv.org/abs/1612.01064` (cit. on p. 46).

[65] Michael Zhu and Suyog Gupta. "To prune, or not to prune: exploring the efficacy of pruning for model compression". In: *CRR* eprint arXiv:1710.01878 (2017). URL: `https://arxiv.org/pdf/1710.01878.pdf` (cit. on p. 47).

[66] J. Albericio, P. Judd, T. Hetherington, et al. "Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing". In: *2016 ACM/IEEE 43rd ISCA*. June 2016, pp. 1–13 (cit. on pp. 48, 69).

[67] Tianqi Chen and Carlos Guestrin. "XGBoost: A Scalable Tree Boosting System". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '16. San Francisco, California, USA: Association for Computing Machinery, 2016, pp. 785–794. ISBN: 9781450342322. DOI: `10.1145/2939672.2939785`. URL: `https://doi.org/10.1145/2939672.2939785` (cit. on p. 72).

[68] Philipp Gysel, Mohammad Motamedi, and Soheil Ghiasi. "Hardware-oriented Approximation of Convolutional Neural Networks". In: *CRR* eprint arXiv:1604.03168 (2016). URL: `http://arxiv.org/abs/1604.03168` (cit. on p. 46).

[69] Song Han. *Deep Compression AlexNet. GitHub repository*. `https://github.com/songhan/Deep-Compression-AlexNet`. [Online; accessed December 10, 2018]. 2016 (cit. on pp. 47, 67, 68).

[70] Song Han. *SqueezeNet Deep Compression — GitHub repository*. `https://github.com/songhan/SqueezeNet-Deep-Compression`. [Online; accessed December 10, 2018]. 2016 (cit. on pp. 47, 67).

[71] Song Han, Xingyu Liu, Huizi Mao, et al. "EIE: Efficient Inference Engine on Compressed Deep Neural Network". In: *Proceedings of the 43rd ISCA*. ISCA '16. IEEE Press, 2016, pp. 243–254 (cit. on pp. 48, 69).

[72] Song Han, Huizi Mao, and William J Dally. "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding". In: *International Conference on Learning Representations (ICLR)* (2016) (cit. on pp. 46, 67).

[73] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, et al. "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size". In: *CRR* eprint arXiv:1602.07360 (2016). URL: `http://arxiv.org/abs/1602.07360` (cit. on p. 46).

[74] S. Zhang, Z. Du, L. Zhang, et al. "Cambricon-X: An accelerator for sparse neural networks". In: *Proceedings of the 49th IEEE/ACM MICRO*. MICRO '16. Oct. 2016, pp. 1–12 (cit. on pp. 48, 69).

[75] Yarin Gal and Zoubin Ghahramani. "Bayesian convolutional neural networks with Bernoulli approximate variational inference". In: *arXiv preprint arXiv:1506.02158* (2015) (cit. on p. 91).

[76] Zoubin Ghahramani. "Probabilistic machine learning and artificial intelligence". In: *Nature* 521 (May 2015), pp. 452–459. DOI: `10.1038/nature14541` (cit. on p. 93).

[77] Song Han, Jeff Pool, John Tran, and William Dally. "Learning both Weights and Connections for Efficient Neural Network". In: *Advances in Neural Information Processing Systems 28*. Ed. by C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett. Curran Associates, Inc., 2015, pp. 1135–1143. URL: `http://papers.nips.cc/paper/5784-learning-both-weights-and-connections-for-efficient-neural-network.pdf` (cit. on p. 47).

[78] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning". In: *Nature* 521 (2015), pp. 436–444. DOI: `10.1038/nature14539`. URL: `https://doi.org/10.1038/nature14539` (cit. on p. 88).

[79] Martín Abadi, Ashish Agarwal, Paul Barham, et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: `https://www.tensorflow.org/` (cit. on p. 94).

[80] Mahdi Pakdaman Naeini, Gregory F. Cooper, and Milos Hauskrecht. "Obtaining Well Calibrated Probabilities Using Bayesian Binning". In: *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*. AAAI'15. AAAI Press, 2015, pp. 2901–2907. ISBN: 0262511290 (cit. on p. 95).

[81]TV Nidhin Prabhakar, Gintu Xavier, P Geetha, and KP Soman. “Spatial preprocessing based multinomial logistic regression for hyperspectral image classification”. In: *Procedia Computer Science* 46 (2015), pp. 1817–1826 (cit. on p. 20).

[82]F. Saqib, A. Dutta, J. Plusquellic, P. Ortiz, and M. S. Pattichis. “Pipelined Decision Tree Classification Accelerator Implementation in FPGA (DT-CAIF)”. In: *IEEE Transactions on Computers* 64.1 (Jan. 2015), pp. 280–285. DOI: `10.1109/TC.2013.204` (cit. on p. 73).

[83]Chunhui Zhao, Yulei Wang, Bin Qi, and Jia Wang. “Global and local real-time anomaly detectors for hyperspectral remote sensing imagery”. In: *Remote Sensing* 7.4 (2015), pp. 3966–3985 (cit. on p. 18).

[84]Christian Debes, Andreas Merentitis, Roel Heremans, et al. “Hyperspectral and LiDAR data fusion: Outcome of the 2013 GRSS data fusion contest”. In: *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing* 7.6 (2014), pp. 2405–2418 (cit. on pp. 10, 30).

[85]Rafał Kułaga and M. Gorgon. “FPGA Implementation of Decision Trees and Tree Ensembles for Character Recognition in Vivado Hls”. In: *Image Processing & Communications* 19 (Sept. 2014). DOI: `10.1515/ipc-2015-0012` (cit. on p. 73).

[86]Zoubin Ghahramani. “Bayesian non-parametrics and the probabilistic approach to modelling”. In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 371.1984 (2013), p. 20110553 (cit. on p. 91).

[87]IEEE. *IEEE GRSS Data Fusion Contest*. [online] `http://www.grss-ieee.org/community/technical-committees/data-fusion/2013-ieee-grss-data-fusion-contest/`. Accessed November 2019. 2013 (cit. on p. 13).

[88]S. Lopez, T. Vladimirova, C. Gonzalez, et al. “The Promise of Reconfigurable Computing for Hyperspectral Imaging Onboard Systems: A Review and Trends”. In: *Proceedings of the IEEE* 101.3 (2013), pp. 698–722. DOI: `10.1109/JPROC.2012.2231391` (cit. on p. 85).

[89]J. Oberg, K. Eguro, R. Bittner, and A. Forin. “Random decision tree body part recognition using FPGAs”. In: *22nd International Conference on Field Programmable Logic and Applications (FPL)*. 2012, pp. 330–337. DOI: `10.1109/FPL.2012.6339226` (cit. on p. 73).

[90]B. Van Essen, C. Macaraeg, M. Gokhale, and R. Prenger. “Accelerating a Random Forest Classifier: Multi-Core, GP-GPU, or FPGA?” In: *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*. 2012, pp. 232–239. DOI: `10.1109/FCCM.2012.47` (cit. on p. 73).

[91]J. Deng, W. Dong, R. Socher, et al. “ImageNet: A Large-Scale Hierarchical Image Database”. In: *CVPR09*. 2009. URL: `http://www.image-net.org` (cit. on p. 67).

[92]Karen E Joyce, Stella E Belliss, Sergey V Samsonov, Stephen J McNeill, and Phil J Glassey. “A review of the status of satellite remote sensing and image processing techniques for mapping natural hazards and disasters”. In: *Progress in Physical Geography* 33.2 (2009), pp. 183–207 (cit. on p. 18).

[93] Antonio Plaza and Chein-I Chang. “Clusters versus FPGA for parallel processing of hyperspectral imagery”. In: *The International Journal of High Performance Computing Applications* 22.4 (2008), pp. 366–385 (cit. on p. 18).

[94] Qian Du. “Unsupervised real-time constrained linear discriminant analysis to hyperspectral image classification”. In: *Pattern Recognition* 40.5 (2007), pp. 1510–1519 (cit. on p. 18).

[95] R. Narayanan, D. Honbo, G. Memik, A. Choudhary, and J. Zambreno. “An FPGA Implementation of Decision Tree Classification”. In: *2007 Design, Automation Test in Europe Conference Exhibition*. Apr. 2007, pp. 1–6. DOI: `10.1109/DATE.2007.364589` (cit. on p. 73).

[96] Antonio J Plaza and Chein-I Chang. *High performance computing in remote sensing*. CRC Press, 2007 (cit. on p. 18).

[97] Christopher M Bishop. *Pattern recognition and machine learning*. Springer Science+ Business Media, 2006 (cit. on pp. 19, 24).

[98] Alexandru Niculescu-Mizil and Rich Caruana. “Predicting good probabilities with supervised learning”. In: *ICML 2005 - Proceedings of the 22nd International Conference on Machine Learning*. Jan. 2005, pp. 625–632. DOI: `10.1145/1102351.1102430` (cit. on p. 95).

[99] Richard Wilson Vuduc. “Automatic Performance Tuning of Sparse Matrix Kernels”. AAI3121741. PhD thesis. University of California, Berkeley, 2003 (cit. on p. 50).

[100] Jerome H. Friedman. “Stochastic gradient boosting”. In: *Computational Statistics & Data Analysis* 38.4 (2002), pp. 367–378. ISSN: 0167-9473. DOI: `https://doi.org/10.1016/S0167-9473(01)00065-2` (cit. on p. 72).

[101] Leo Breiman. “Random forests”. In: *Machine learning* 45.1 (2001), pp. 5–32 (cit. on p. 23).

[102] Chein-I Chang, Hsuan Ren, and Shao-Shan Chiang. “Real-time processing algorithms for target detection and classification in hyperspectral imagery”. In: *IEEE Transactions on Geoscience and Remote Sensing* 39.4 (2001), pp. 760–768 (cit. on p. 18).

[103] Jerome H Friedman. “Greedy function approximation: a gradient boosting machine”. In: *Annals of statistics* (2001), pp. 1189–1232 (cit. on p. 23).

[104] Christopher M Stellman, Geoff Hazel, Frank Bucholtz, et al. “Real-time hyperspectral detection and cuing”. In: *Optical Engineering* 39 (2000) (cit. on p. 18).

[105] Radford M. Neal. *Bayesian Learning for Neural Networks*. Berlin, Heidelberg: Springer-Verlag, 1996. ISBN: 0387947248 (cit. on p. 93).

[106] Corinna Cortes and Vladimir Vapnik. “Support-vector networks”. In: *Machine learning* 20.3 (1995), pp. 273–297 (cit. on p. 24).

[107]Babak Hassibi and David G. Stork. “Second order derivatives for network pruning: Optimal Brain Surgeon”. In: *Advances in Neural Information Processing Systems 5*. Ed. by S. J. Hanson, J. D. Cowan, and C. L. Giles. Morgan-Kaufmann, 1993, pp. 164–171. URL: `http://papers.nips.cc/paper/647-second-order-derivatives-for-network-pruning-optimal-brain-surgeon.pdf` (cit. on p. 47).

[108]Yann LeCun, John S. Denker, and Sara A. Solla. “Optimal Brain Damage”. In: *Advances in Neural Information Processing Systems 2*. Ed. by D. S. Touretzky. Morgan-Kaufmann, 1990, pp. 598–605. URL: `http://papers.nips.cc/paper/250-optimal-brain-damage.pdf` (cit. on p. 47).

[109]Leo Breiman, Jerome Friedman, Richard Olshen, and Charles Stone. *Classification and regression trees*. Chapman & Hall, Taylor & Francis Group, 1984. ISBN: 9780412048418 (cit. on p. 21).

This thesis had support from

Grants (1) PID2019-105660RB-C21 / AEI / 10.13039/501100011033 from Agencia Estatal de Investigación (AEI) and European Regional Development Fund (ERDF), and (2) gaZ: T5820R research group from Dept. of Science, University and Knowledge Society, Government of Aragón. It also had support from the Government of Aragón through the European Social Fund "Construyendo Europa desde Aragón" of the European Union and from the High Performance Embedded Architecture and Compilation (HIPEAC) European network. The funders had no role in study design, data collection and analysis, decision to publish, or preparation of the thesis.