Onboard Image Classification of Biological Habitats Using Underwater Vehicles

Miguel Quinaz Pereira
Mestrado de Ciências de Computadores
Departamento de Ciências dos Computadores
2021

Orientador
Eduardo R. B. Marques, Professor Auxiliar, Faculdade de Ciências da Universidade do Porto

Supervisor
José Queirós Pinto, Assistente de Investigação, Faculdade de Engenharia da Universidade do Porto

Abstract

As time passes, biological habitats change: the conditions of the planet are constantly alternating with complex relationships. Hence, it is important to monitor this differences throughout time and space, at a time where we must face problems like global warming and mass extinction of species. To do so, a crucial task by biologists is to map the habitats by going to the field, collecting data, and then labelling areas according to standard habitat classification system, like European Nature Information System (EUNIS). This is challenging due to the massive size of the areas at stake, and, if we consider oceans, the need to collect data underwater regarding the sea floor. Autonomous vehicles are a very important tool in this regard since they can obtain optic, sonar and aerial imagery in bulk and automated manner. The data obtained can then be classified by a biologist. Still, classifying a vast number of these images is not practical, and, pushing further ahead, Machine Learning (ML) can potentially turn the classification process automatic and with very good precision. In this dissertation, we present an extension to the software toolchain of LSTS for autonomous vehicles to perform real-time habitat mapping using Convolutional Neural Networks (CNNs) over images collected from vehicles' cameras. Demonstrating the feasibility of our approach, we trained and evaluated several CNN models using underwater imagery collected by LSTS vehicles at Northern Littoral Natural Park (PNLN) in Esposende, later classified by biologists using the EUNIS standard. The software and models we developed were deployed in embedded software platforms suitable for use with autonomous vehicles, such as Raspberry PI 4 and NVidia Jetson Nano, and validated in simulation.

Keywords: habitat mapping, autonomous vehicles, machine learning, convolutional neural networks.

Resumo

À medida que o tempo passa os habitats biológicos alteram-se: as condições do planeta estão constantemente em evolução com relações complexas. É então importante avaliar estas diferenças ao longo do tempo e do espaço, numa altura em que enfrentamos fenómenos como o aquecimento global e a extinção maciça de espécies. Para tal, uma tarefa crucial por parte de biólogos é mapear os habitats indo para o terreno, colectar dados, e etiquetar áreas de acordo com sistemas padrão para o efeito, tal como o European Nature Information System (EUNIS). Esta é uma tarefa desafiante dada as áreas massivas que é preciso catalogar, e, se considerarmos oceanos, a necessidade de obter debaixo de água para o fundo do mar. Veículos autónomos são ferramentas bastante importante a esse respeito, dado que podem obter imagens de diversos tipos em grande volume e de forma automatizada. Os dados obtidos podem depois ser classificados por um biólogo. No entanto, a classificação manual de grandes volumes de imagens é impraticável, e, indo mais além, o uso de técnicas de Machine Learning (ML) pode potencialmente tornar o processo de classificação automático e com alta precisão. Apresentamos nesta dissertação uma extensão a software do LSTS para veículos autónomos para classificação automática de habitats usando redes neuronais convolucionais (CNNs) sobre imagens obtidas capturadas pelas câmaras dos veículos. Para demonstrar a aplicabilidade da aproximação, treinamos e avaliamos vários modelos baseados em CNNs usando imagens subaquáticas recolhidas por veículos autónomos do LSTS no Parque Natural do Litoral Norte (PNLN) em Esposende, depois classificadas manualmente por biólogos usando o standard EUNIS. O software e modelos desenvolvidos foram avaliados em plataformas de sistemas embutidos adequadas a veículos autónomos, como Raspberry PI e Nvidia Jetson Nano, e validadas em simulação.

Palavras-chave: mapeamento de habitas, veículos autónomos, aprendizagem automática, redes neuronais convolucionais.

Acknowledgements

I would like to thank my supervisors, Professor Eduardo Marques and José Pinto believing in me, the support they gave me throughout the semesters and the help guiding me on several different topics making it possible for me to write this dissertation. The motivation even when things were not going on correctly never cease to exist. Thanks for introducing me to LSTS and the explanation of the toolchain modules as well as the opportunity for me to work on this topic, for the professors at FCUP and DCC for the background possible to work on Machine Learning (ML), Computer Vision (CV) and algorithms.

I am also thankful for the help provided for my family during the time I took to write the dissertation and for the motivation Laura gave me and her honesty.

Contents

List of Tables

List of Figures

Listings

Acronyms

API Application Programming Interface

ASV Autonomous Surface Vehicle

AUV Autonomous Underwater Vehicle

AVA Amplitude Versus grazing Angle

CNN Convolutional Neural Network

CPU Central Processing Unit

CUDA Compute Unified Device Architecture

CV Computer Vision

DUNE Dune Uniform Navigation Environment

EUNIS European Nature Information System

FCN Fully Convolutional Network

GPU Graphics Processing Unit

HM Habitat Mapping

IMC Inter-Module Communications

LAUV Light Autonomous Underwater Vehicle

LIDAR Light Detection And Ranging

LAUV Light Autonomous Underwater Vehicle

ML Machine Learning

NN Neural Network

OBIA Object-Based Image Analysis

OMARE Marine Observatory of Esposende

PNLN Northern Littoral Natural Park

ROV Remotely Operated Vehicle

SVM Support Vector Machine

TPU Tensor Processing Unit

UDP User Datagram Protocol

UAV Unmanned Aerial Vehicle

Chapter 1

Introduction

1.1 Motivation

As time passes, biological habitats change: the conditions of the planet are constantly alternating with complex relationships. Hence, it is important to monitor this differences throughout time and space, at a time where we must face problems like global warming and mass extinction of species [16]. To do so, a crucial task by biologists is Habitat Mapping (HM). This requires going to the field, collecting data, and then labelling areas according to standard habitat classification systems like European Nature Information System (EUNIS) [4, 7]. This is challenging due to the massive size of the areas at stake, and, if we consider oceans, the need to collect data underwater regarding the sea floor.

Autonomous vehicles are a very important tool in this regard since they can obtain optic, sonar and aerial imagery in bulk and automated manner. The data obtained can then be classified by a biologist. Still, classifying a vast number of these images is not practical, and, pushing further ahead, Machine Learning (ML) can potentially turn the classification process automatic and with very good precision [5].

In the Marine Observatory of Esposende (OMARE) project, biologists classify natural habitats of Northern Littoral Natural Park (PNLN) according to the EUNIS guidelines. The coastal area at stake has 80 Km^2, of which 70 Km^2 are underwater, as illustrated in Figure 1.1. For this classification, images were collected using autonomous vehicles by LSTS/FEUP and Convolutional Neural Network (CNN) models were developed for automated HM [6, 17].

1.2 Problem statement and contributions

Previously in the OMARE project, CNNs were employed for habitat mapping only after field tests took place, however, this dissertation addresses the problem of real-time habitat mapping within autonomous vehicles using CNNs, i.e., the classification of the images on-the-fly during field tests.

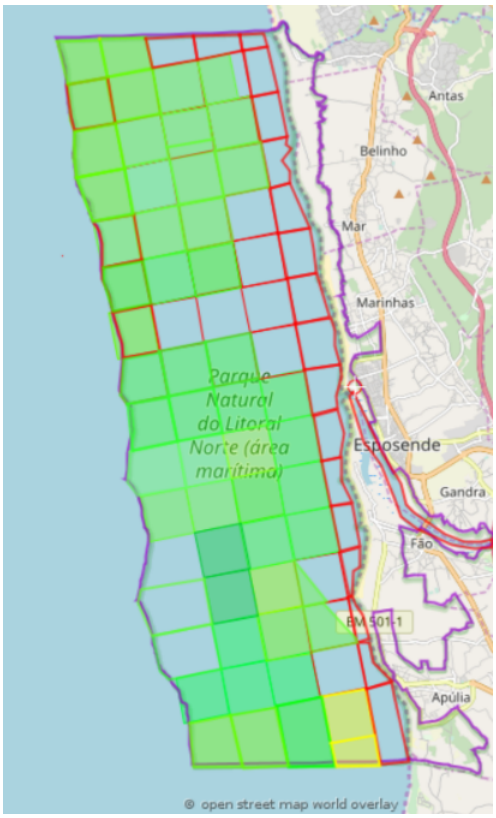


Figure 1.1: PNLN map, taken from [17]

This has the potential of accelerating the overall habitat mapping workflow. For this goal, one needs to consider the challenge of deploying CNN models onboard an autonomous vehicle, with the inherent resource constraints of the computacional platform. Hence, model inference need to be computationally efficient, without sacrificing reasonable predictive power. With this overall context and challenges in mind, this thesis puts forward the following contributions:

- We derived neural network models for habitat mapping suitable for deployment in embedded platforms used for autonomous vehicles. The models at stake include two simple models that serve as simple baselines, plus state-of-the-art deep CNNs with architectures designed for use with embedded software, like MobileNetV2 [44] through transfer learning using the TensorFlow API [41] or MnasNet CNNs [40] through the Google Auto ML Vision cloud service [3]. All models are converted for use with TensorFlow Lite [42], a specialized TensorFlow library for deep learning on resource-constrained devices.
- We developed a Python framework to run onboard autonomous vehicles that is capable of performing real-time image acquisition and habitat classification using a configurable model, sampling rate, and video source. The framework is interoperable with the LSTS toolchain [30] through a message-based protocol called IMC.
- We conducted an evaluation regarding the use of models and the onboard software running on Raspberry Pi 4 [34] and Nvidia Jetson Nano [25] devices, illustrating that it is efficient

to deploy CNNs for habitat mapping on autonomous vehicles both in terms of predictive power as well as computational performance.

1.3 Thesis structure

The remainder of this dissertation has the following structure:

- **Chapter 2 (Background)** provides the background of this dissertation regarding various key aspects for the work of this dissertation, including the EUNIS classification system, the LSTS software toolchain, and convolutional neural network fundaments. We also survey previous work in the OMARE project regarding automated habitat mapping and other works in the state-of-the-art.
- **Chapter 3 (Development)** presents our development. We describe the process of training various CNN models and then the onboard classification software for autonomous vehicles that can be used to deploy these models.
- **Chapter 4 (Results)** presents results regarding the various CNN models and the performance of the onboard classification software.
- **Chapter 5 (Conclusion)** presents a final discussion of the thesis work and of problems for future work.

Chapter 2

Background

In this chapter we provide the background of this thesis. We start by describing the Marine Observatory of Esposende (OMARE) project, where autonomous vehicles have been employed to aid habitat mapping in a collaboration between LSTS and Northern Littoral Natural Park (PNLN) (Section 2.1). The habitat mapping taxonomy follows the European Nature Information System (EUNIS) standard described next (Section 2.2). Then we provide background information regarding neural networks, CNNs, and the machine learning tools used in the thesis (Section 2.3). We then present the LSTS toolchain (Section 2.4). We end with a discussion of related work in the use of Machine Learning (ML) for automated Habitat Mapping (HM) (Section 2.5).

2.1 The OMARE project

The coastline of Esposende possesses a high level of biodiversity and to protect it PNLN was created. In association to PNLN, project OMARE has the aim of implementing a information and monitoring system for the marine biodiversity of PNLN. The goals of the project to map PNLN's seabed and habitats, to monitor the existing species by regular inspection of the state of biodiversity, identifying and recovering degraded ecosystems, implement new principles of conservation and management, and ensure awareness and training for the entire population in the sustainable development of the region. To this end, the municipality of Esposende promotes the sustainable use of sea resources to ensure that the natural heritage of the sea is protected from threats and risks [16] to the marine environment and its biodiversity such as the overexploitation of resources, the destruction of habitats and the introduction of exotic species, climate-related pressures, disorderly urbanism, human pressure and human waste.

In a recent partnership between PNLN and LSTS in the scope of OMARE, the use of autonomous vehicles has been considered to aid the task habitat mapping. There are a number of different types of vehicles like Autonomous Underwater Vehicle (AUV), Unmanned Aerial Vehicle (UAV), Remotely Operated Vehicle (ROV), Autonomous Surface Vehicle (ASV). Some of those developed at LSTS are shown in Figure 2.1. Some of the vehicles are remotely operated in

Figure 2.1: Different types vehicles created by LSTS

real-time by users, while others operate underwater for long periods without human intervention. LSTS vehicles are relatively low cost, modular, and have common software infrastructure which makes them inter-operable and able in some cases to work as a team.

Communication can be done with cables or wirelessly to transmit information. Some vehicles like ROV, AUV can use long cables of either optic fiber or conductive wires to achieve underwater real-time video although we have a limited amount of cable length and harm can be done through bad sea conditions. Wireless communications use mechanical or electromagnetic waves that possess a frequency/amplitude that is modulated to transmit analog or digital data. The main difference in both is that mechanical waves(e.g. sound) require a medium but in both occurs attenuation, absorption, reflection and collision with other waves. Depending on the wave frequency and its material properties (e.g. density, molecular form) waves can get absorbed/refracted/reflected by some material Since in the electromagnetic wave spectrum the lower the frequency, the higher the probability the wave gets reflected in the atmosphere, using low frequencies is highly impractical. Water is too dense so radio propagation is possible but not efficient for communication [33].

For habitat mapping, LSTS vehicles can obtain aerial or underwater imagery by employing side-scan sonar or video cameras [6]. Some of the possible types of images are illustrated in Figure 2.2 Video cameras are cheap and capable sensors for detecting structure, motion and features through light, however, using cameras underwater poses many problems as light is absorbed by the water and reflected on suspended particles. As an alternative, underwater vehicles typically use expensive acoustic sensors to acquire bottom data which, despite their price, cannot detect thin and/or soft objects like fishing nets, algae or marine debris which can otherwise be imaged by underwater cameras.

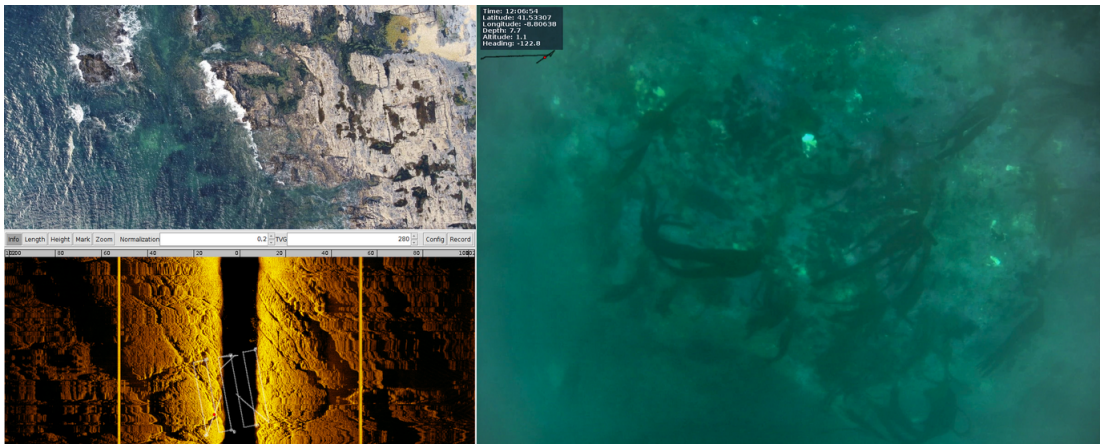


Figure 2.2: Aerial, underwater and side scan images captured by autonomous vehicles

2.2 European Nature Information System (EUNIS)

A habitat is described as "a place where plants or animals normally live, characterized primarily by its physical features (topography, plant or animal physiognomy, soil characteristics, climate, water quality etc.) and secondarily by the species of plants and animals that live there" [4].

EUNIS is a habitat classification system that gathers information from different organizations and databases that focus on the European continent and sea area, including offshore islands(Cyprus; Iceland), archipelagos of the European Union Member States(Canary Islands, Madeira and the Azores), Anatolian Turkey and the Caucasus. Overall this system can be most specific to classify a habitat based on the species identified nearby or more general such as checking if the habitat is underwater or not. In this sense, we can view it as a hierarchical system [4] where lower levels(e.g. level 1) are more general and upper levels are more specific (e.g. level 6). It tries to cover all the terrestrial and marine habitats, although such a system can never be complete due to new research or the extension of the area covered. For the case of discovering a new habitat type it is possible to make a request to the developers of the classification at the European Topic Centre on Biologic Diversity who can advise on how to code the new habitat type and include it in the next published version.

EUNIS possesses a database which can be accessed through its website interface [7] in order to visualize text descriptions, environmental parameters, relationships to other classifications and to legislative systems. Only a fraction of PNLN has been covered and making use of low-detail categories in EUNIS.

Habitats can range from small areas with less than 1 m^2 (called a microhabitat) to large ones with 10 ha (100 000m). Overall the available classifications are [4]:

A. Marine habitats

B. Coastal habitats

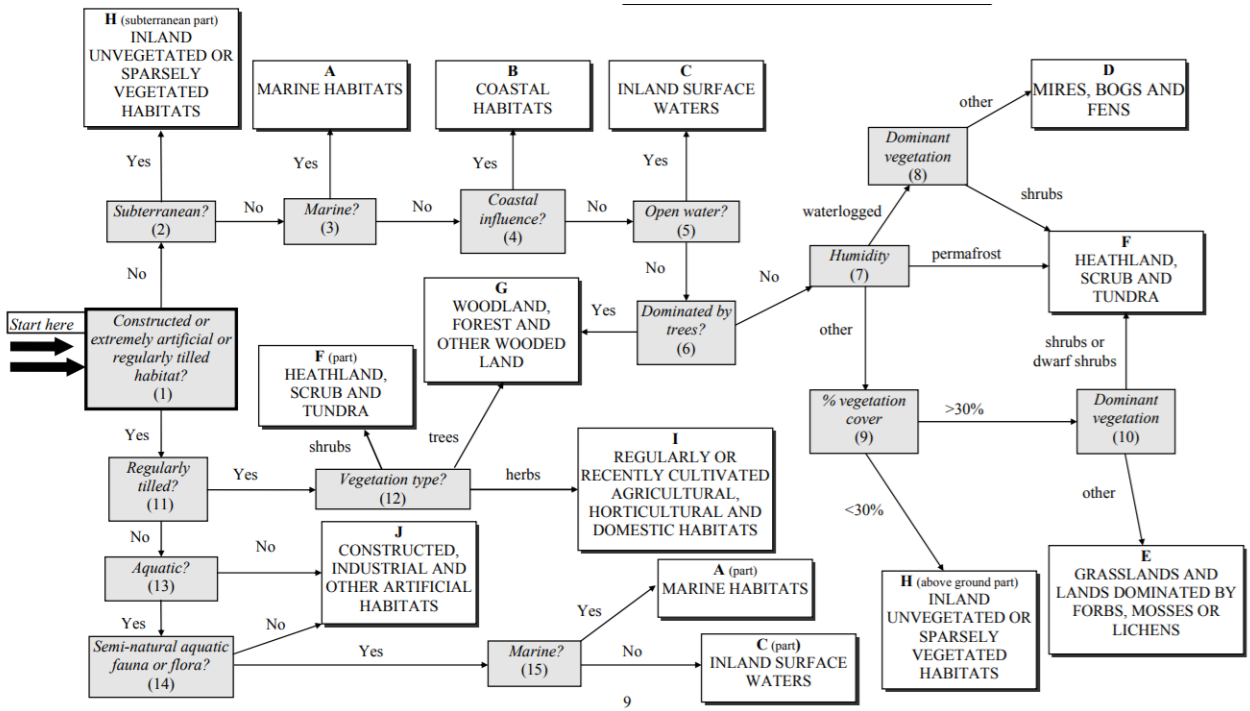


Figure 2.3: EUNIS Habitat Classification: criteria for Level 1 taken from [4]

C. Inland surface waters

D. Mires, bogs and fens

E. Grasslands and lands dominated by forbs, mosses or lichens

F. Heathland, scrub and tundra

G. Woodland, forest and other wooded land

H. Inland unvegetated and sparsely vegetated habitats

I. Regularly or recently cultivated agricultural, horticultural and domestic habitats

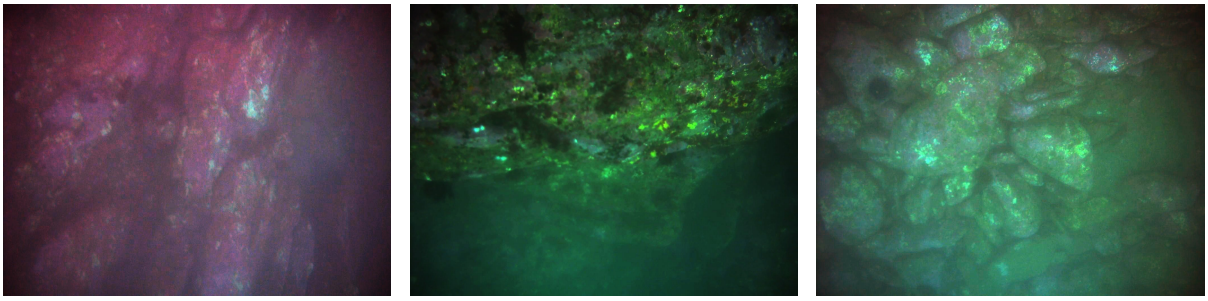
J. Constructed, industrial and other artificial habitats

To make the different classifications, specific rules are observed by biologists, for example as shown in Figure 2.3. Since we are working with underwater vehicles every classification is defined as A and there is no motivation to train a model using these labels. In terms of marine habitats, the available classifications at level 2 are:

A1. Littoral rock and other hard substrata

A2. Littoral sediment

A3. Infralittoral rock and other hard substrata



(a) A3 (b) A4 (c) A5

Figure 2.4: Example images classified by biologists using EUNIS

A4. Circalittoral rock and other hard substrata

A5. Sublittoral sediment

A6. Deep-sea bed

A7. Pelagic water column

A8. Ice-associated marine habitats

Figure 2.4 provides sample images obtained with a AUV and classified as A3, A4, and A5.

2.3 Machine learning background

The use of machine learning is becoming a popular approach for automatic habitat mapping [5]. Recent advances in Computer Vision (CV) and (deep) Convolutional Neural Network (CNN) can be employed to normalize and improve the quality of underwater imagery as well as extract and detect these features automatically. Neural networks are instances of supervised learning methods, where data is provided with labels so that the learning algorithm can improve and later on use a inferred function to make future predictions. This contrasts with unsupervised machine learning, where data is not classified and the learning algorithm finds a function to infer a hidden structure from unlabelled data, or hybrid semi-supervised machine learning algorithms. Here we describe the basic way in which neural networks work and then the specific nature of CNNs. We also provide background information on TensorFlow and Google Auto ML, the tools we used to derive CNNs in the thesis.

2.3.1 Neural networks

A simple neural network is illustrated in Figure 2.5. The neural network has three inputs x_i and two outputs $Y_i^{'}$, defining what are called the input and output layers. The network also has one intermediate layers, called a hidden layer. Except for the input layer, a neuron has input

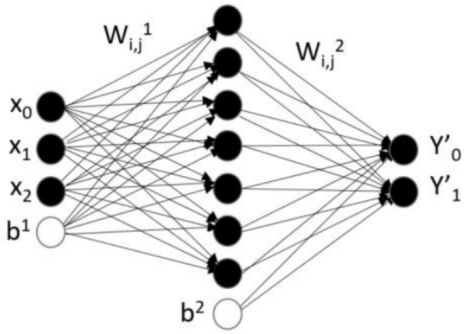


Figure 2.5: Neural Network (NN) with three inputs, two outputs, and a hidden layer (taken from [8])

connections to neurons in the preceding layer, where each connection has an associated weight ($W^1_{i,j}$ for the intermediate layer and $W^2_{i,j}$ in the output layer) and an activation function that determines its output. Each layer also has a bias value b_k that is used as offset for the activation function. The output is a function of the input values, weights, and bias. In this example we have that

$$a_j = f\left(\sum_{i=0}^{2} x_i W^1_{i,j} + b_j\right)$$

and

$$Y^{'}_j = f^{'}\left(\sum_{i=0}^{n} a_i W^2_{i,j} + b^2_j\right)$$

where f and f' are the activation functions. That is, a_j is the result from the first layer with function f, and $y^{'}_j$ uses as input a_j and gives the result of the second layer with function $f^{'}$. f and $f^{'}$ are activation functions such as logistic, rectified linear ($relu(x) = max(0, x)$) or hyperbolic tangent which maps values to a interval $[0, 1]$ to symbolise an on or off neuron. For a classification problem that has to output a probability linked to the last layer of the NN, the softmax logistic function is used:

$$\text{softmax}(x)_j = \frac{1}{1 + \sum_{k \neq j} e^{x_k - x_j}} \tag{2.1}$$

To approximate this function we pick values for the weights minimizing the cost function given by

$$C(x^i, y^i) = \sum ||y^i - y^{'}(x^i)||$$

using the labeled examples (x^i, y^i) and back propagation, which is a variation of gradient descent [10, 24]. The training process iteratively adjusts the weights to try to minimize the cost function.

2.3.2 Convolutional neural networks

CNN is a type of neural network that uses a special kind of operations called convolutions. Convolutions allow us to detected patterns, represented as two-dimensional windows for 2D images, as illustrated in Figure 2.6.

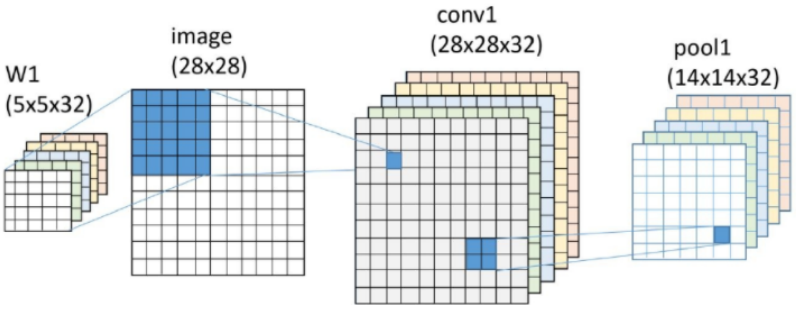


Figure 2.6: Schematic of how a CNN processes an image (taken from [8])

The figure illustrates the process of finding a 5x5 W_1 array and offset b, that can reduce a window at a particular point of the image to a new array conv as follows:

$$\text{conv}_{p,q} = \sum_{i,k=-2}^{2} W_{i,k} image_{p-i,q-k} + b \tag{2.2}$$

The conv array also applies a relu activation to its point so that it does not have any negative value and lastly a max pooling operation uses a 2×2 that reduces a particular sub block of the image by choosing the maximum value. All of these operations are run with different windows, for 2.6, 32 14x14 results, and we can view the resulting array as a transform of the original image where each of the windows W select a different set of features of the original image.

2.3.3 Tensorflow

TensorFlow is an open source library for numerical computations that can be used in Central Processing Unit (CPU), Graphics Processing Unit (GPU) and Tensor Processing Unit (TPU). It helps implementing, training, evaluating and deploying ML models, as illustrated in the framework overview in Figure 2.7.

An example use of Tensorflow to train a simple Neural Network model is given in Listing 2.1, adapted from the Tensorflow tutorials. The steps at stake comprise: (1) loading the data at stake, in this case data from the MNIST data set and preprocessing it; (2) defining a neural network model, `model`; (3) defining the method of training through a call to `model.compile()`; (4) the actual training for 5 epochs through a call to `model.fit()`; and (5), after training, evaluating the performance through a call to `model.evaluate()`. These steps are explained in detail later in Chapter 3 (§ 3.1.2 and § 3.1.3).

For the neural network at stake in this example,inputs are 28×28 images where pixel

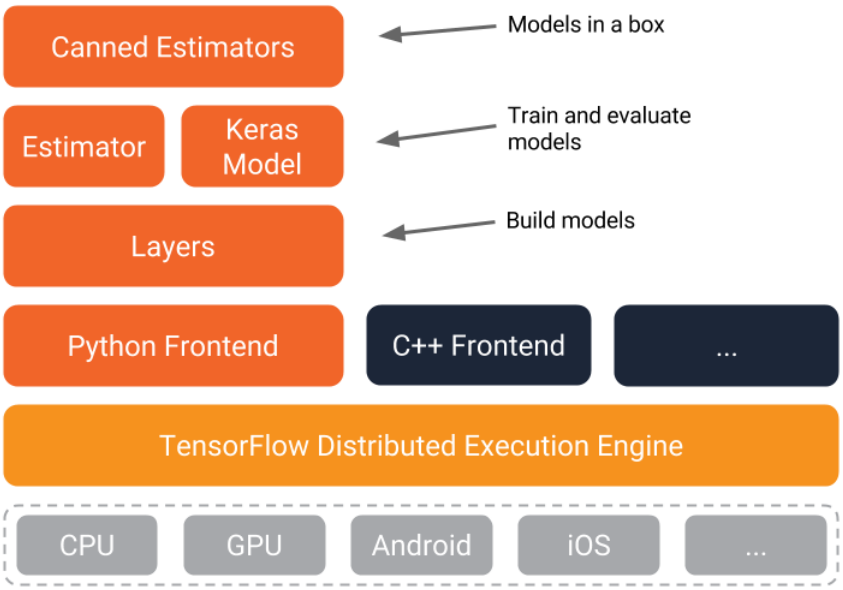


Figure 2.7: Tensorflow framework

values are converted from the $[0-255]$ RGB scale to values in the $[0-1]$ interval. This scale transformation is done before the actual training. The network takes the 28×28 pixel matrix and flattens it to a 1D-array. The following layer is a fully connected layer with a relu activation, while the final output layer is also fully connected but has a softmax activation. To prevent overfitting the training data, a "virtual" dropout layer is used. During training the use of dropout randomly sets inputs to 0 with the given probability per each epoch. After training, the dropout layer is removed.

2.3.3.1 Keras

The fragment in Listing 2.1 uses the Keras API. Keras (`tf.keras`) is a high-level Python API for Tensorflow. It allows for an easy definition and manipulation of models. Aspects such as like neural network layers, cost functions, optimizers, initialization schemes, activation functions are all defined by the API. Keras also supports a wide range of production deployment options, integration with different back-end engines (TensorFlow, CNTK, Theano, MXNet, and PlaidML) and support for multiple GPU and distributed training.

Keras has a model data structure, that can be defined with the Sequential model and with the Functional Application Programming Interface (API). The `Sequential` model in the example is a linear stack of layers that are defined in Python using `model = Sequential()`. Using the functional API, it also possible to handle models with non-linear topology, shared layers, and even multiple inputs or outputs. These layers can be customized or predefined like Dense, Activation, Dropout, Lambda, max and average pooling, recurrent layers and convolution layers that are used for image recognition. Besides the core of creating a model, Keras has samples of data sets packaged in `keras.datasets`, predefined models that allow the use of transfer learning and

```
import tensorflow as tf
mnist = tf.keras.datasets.mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(10, activation='softmax')
])

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=5)

model.evaluate(x_test,  y_test, verbose=2)
```

Listing 2.1: Tensorflow example (Keras API)

possess the deployment of models for multiple platforms.

2.3.3.2 Tensorflow Lite

TensorFlow Lite (TFLite) [42] is variant of the TensorFlow library for machine learning on mobile and embedded devices. We use TFLite in the thesis as the main API for the task of image classification using trained models. TFLite models can run entirely onboard a device like iOS or Android smartphones or Raspberry Pi embedded devices. TFLite files, with a `.tflite` extension, are saved using a special format defined by a serialization library called FlatBuffers, which is typically much more compact than the standard SavedModel format used by TensorFlow.

It is possible to convert models from SavedModel to TFLite format using the TFLite converter. During the conversion, in addition to the typical reduction in file size due do the FlatBuffers representation, it it also possible to apply techniques like post-training quantization that reduces model size further and can also improve CPU and hardware accelerator (e.g., GPU latency) without sacrificing model accuracy significantly. Quantization is a process that may for instance convert 32-bit floating point arithmetic onto lower precision 16-bit floating point arithmetic or 8-bit integer arithmetic.

2.3.4 Google Auto ML

In this thesis, we make use of the Google Auto ML cloud service [3] for automated derivation of CNN models without expert knowledge. The CNN models at stake are adequate for use in mobile and embedded devices with architectures from the MnasNet family [40], and can be deployed in a variety of formats including TFLite. Chapter 3 (§ 3.1.4) describes how 3 CNN models were derived using Google Auto ML.

2.4 The LSTS software toolchain

The Underwater Systems and Technology Laboratory (LSTS) has created a software toolchain for autonomous vehicles [30], available open-source at GitHub [19]. The three main components of the toolchain are: (1) IMC, a message-based protocol for interoperability between nodes in a network such as vehicles or human operator consoles; (2) DUNE: an on-board software platform for autonomous vehicles, and; (3) Neptus, a command-and-control infrastructure that allows users to prepare, monitor, and review vehicle operations using GUI consoles.

2.4.1 IMC

Inter-Module Communications (IMC) [23, 30] is an extensible message-based protocol, where messages have the structure shown in Table 2.1. The flat serialization format of messages is described by an XML file, from which programming language bindings can be automatically generated, e.g., C++, Java or Python. In this thesis, we extend IMC with a few messages for image classification and employ the pyIMC library [39] that provides Python bindings for IMC.

The protocol is transport-agnostic, meaning that it can be used over any type of communication link. As shown in Table 2.1, messages can identify the source and destination peers, and possibly also source and destination entities, through 16-bit abstract addresses. It is up to transport drivers to map these abstract addresses onto concrete addresses at the link level (e.g., UDP or TCP).

2.4.2 DUNE

Dune Uniform Navigation Environment (DUNE) is a onboard software for unmanned vehicles designed for efficiency, portability and flexibility, that deals with communications with the outside world by sensing and actuation, navigation, control and logging. DUNE is divided into a base library and tasks implemented in C++, and employs CMake for cross compilation to several variants of Linux, but also possibly Windows or MacOs. When DUNE starts, the initialization daemon loads a configuration that instantiates all tasks that are referred in that configuration. Tasks are then started, with one operating system thread supporting each task, and then interact

Table 2.1: IMC message structure

Packet Label	Field Name	Field type	Description
Header	Synchronization Number	uint16_t	This field marks the beginning of the packet, adding also the protocol versions for later interpretation
	Message Identification Number	uint16_t	This field is used for correct message interpretation and deserialization
	Message size	uint16_t	The size of the message data in the packet
	Time stamp	fp64_t	The time when the packet was sent
	Source Address	uint16_t	The Source IMC system ID.
	Source Entity	uint8_t	The entity generating this message at the source address
	Destination Address	uint16_t	The destination IMC system ID
	Destination Entity	uint8_t	The entity that should process this message at the destination address.
Payload	*	*	Varies according to message type
Footer	checksum	uint16_t	This field is computed using the CRC-16-IBM with polynomial 0x8005 ($x^16 + x^15 + x^2 + 1$). The data includes all preceding header and message bytes.

through an IMC message bus.

2.4.3 Neptus

Neptus Command and Control Unit is a framework for use by human operators in interface with autonomous vehicles. An example Neptus console is displayed in Figure 2.8. Overall, operators of Neptus can interact with vehicles in real time by creating plans and receiving data from the network. Neptus supports different kinds of vehicles, allowing planning and simulation of future missions, visualize previous missions as well as controlling current missions in real-time. The missions are based on the execution of maneuver plans that can be changed during execution. The functionality of Neptus can also be extended through a plugin system.

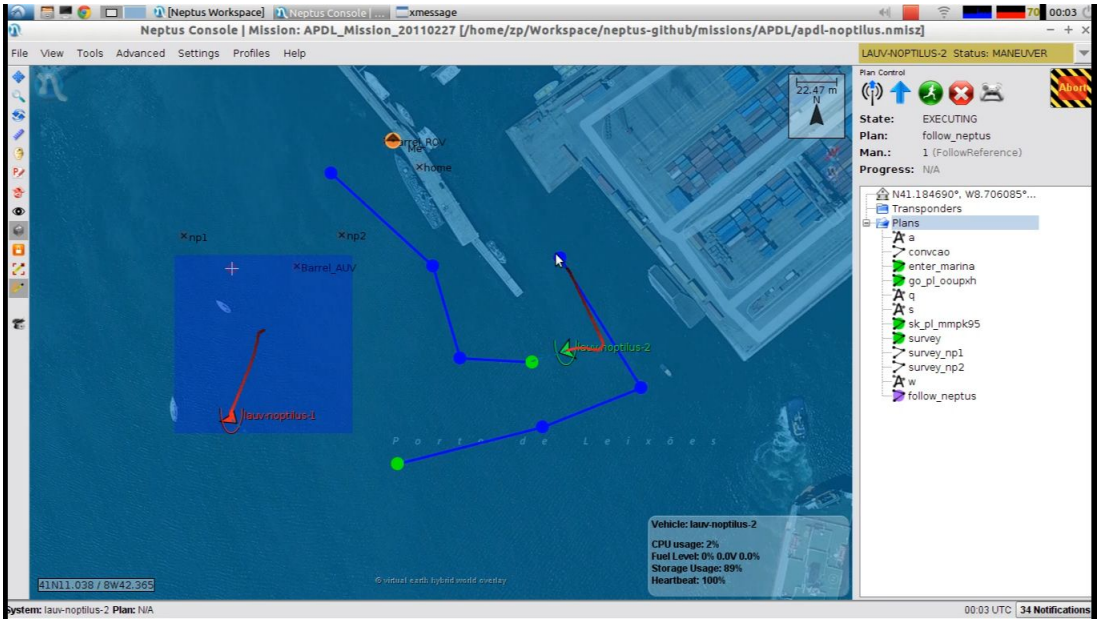


Figure 2.8: Neptus console taken from [20]

2.5 State of the art

2.5.1 Previous work in the OMARE project

In [29], an LSTS report describes the overall work in the habitat mapping of PNLN done with AUV vehicles with camera, sidescan sonar and multibeam. UAVs also mapped the territory to be labelled in the future and all the data is available to the municipality of Esposende in a cloud facility. The used CNN models were developed by Diegues et al. [6] based on side-scan sonar images obtained with a AUV and annotated by marine biologists. The deployed CNN models had a accuracy of 85.1 % for the same level 2 from EUNIS, and 69.6 % for level 3. The work also claims that no preceding work uses CNN to predict the EUNIS habitat type of underwater images to perform automatic HM according to the European Union standards. Lima et al. [17] then applied these CNN models for mapping PNLN using Light Autonomous Underwater Vehicle (LAUV) and UAV vehicles. The vehicles were equipped with side-scan sonar, multibeam and/or cameras and were controlled in groups so that they can collect data simultaneously, optimising the coverage of the territory in a mission.

2.5.2 Other related work

Early foundation work in HM was in areas of study towards species and habitat relations, for example in predictive vegetation mapping, for instance using gradient models and statistical models as described by Franklin [9] Around the year 2000, classification and regression trees, k-nearest-neighbours and decision trees were used for HM [35]. Recently ML models started to be explored in which CNN are the most used for HM due to the best obtained accuracy. There

has been a lot of work in the field but not necessarily work using a EUNIS classification [6].

In [27], Pandian et al. discuss marine HM and the associated technologies for seabed classification ranging from singlebeam echo-sounders to Light Detection And Ranging (LIDAR) bathymetry. The authors conclude that sonar-based images are preferable to optic images to train ML models especially in zones with bad conditions underwater.

In [12], Gomez-Rios et al. use CNN models to classify coral texture images. The models were trained with data augmentation since there was class imbalance in the data set and transfer learning used with architectures like Inception v3, ResNet and DenseNet. The data sets are EILAT and RSMAS were used and obtained a average of 98.2 % accuracy. The high accuracy is explained by big difference in the visuals of the textures at stake.

In [11], Guirado et al. developed CNN for detection of plant species of conservation concern using free high-resolution Google EarthTM images and data augmentation. It achieved a accuracy of 91.8 % and was a improvement to Object-Based Image Analysis (OBIA) methods up to 12 %, 30 % recall and 20 % balance between precision and recall. They also provide open source code to reproduce their work for other species detections.

In [21], Luo et al. consider two CNN models for classification of sediments using a small seabed images data set. A shallow CNN had better results than a deep CNN using side scan sonar data ranging from 93.4 % to 87.54 % compared to 66 % of the deep CNN.

The PhD thesis of Christensen [2] approached the problem of marine habitat mapping using multibeam sonar data for interpretation of the seabed substrate. The interpretation was done combining bathymetry and backscatter increasing the number of mapped classes and making geomorphologic features become visible. They tested the suitability of habitat classification schemes in the Norwegian coast for handling the data and using the previous article method. They also compared biological, geological and tidal current models to find correlation for future HM projects. They reviewed interpretation, processing, acquisition and sampling methods used in Norwegian waters and suggested a mapping procedure. They demonstrated that angular response curves could be used for sediment classification. Using theoretical models and multibeam data they also investigated the extraction of physical parameters of the seabed sediments using Amplitude Versus grazing Angle (AVA) curves; Lastly they demonstrated automatic classification based on multibeam backscatter and also determined the seabed properties like coral reefs using acoustic data.

In [18], Liu et al. compare the use of different machine learning methods for object-based wetland mapping in South Florida using images from small unmanned aircraft systems. The methods include Fully Convolutional Network (FCN) (that derive segmented images), CNN, Support Vector Machine (SVM), and random forest. They obtained the best accuracy using FCN (76.9%) but also observe that CNN can obtain comparable accuracy for larger data sets.

In [1], Berthold et al. use CNN models trained with side-scan sonar images labelled using four classes of sand sediment labelled by experts: fine, sand, coarse, and mixed sediment. They

achieved an accuracy between 11% and 83 % per class, and an average accuracy of 56 %.

Chapter 3

Development

In this chapter we describe the development of the machine learning models for habitat classification (Section 3.1) and the software framework for onboard image classification in autonomous vehicles (Section 3.2). The code and resulting models can be found in our GitHub repository [28] with contents summarised at the end of this chapter (Section 3.3).

3.1 Models

3.1.1 Dataset

To define our models, we use underwater imagery data captured by AUVs at Northern Littoral Natural Park (PNLN). The data was collected during three separated field deployments of a Light Autonomous Underwater Vehicle (LAUV) vehicle [22] in May 2018 and July 2018. In each of these deployments, the vehicle was active several hours retrieving photos using an underwater camera. Some heuristics were applied so that the vehicle did not record images on upper depths (turbidity makes it difficult to obtain good quality images).

Part of the image frames collected were annotated by biologists, belonging to three classes of level-1 marine habitats: **A3 - Infralittoral rock and other hard substrata**, **A4 - Circalittoral rock and other hard substrata** and **A5 - Sublittoral sediment**. Frames were also classified in sub-levels of A3, A4, and A5. The image labelling statistics are shown in Table 3.1, and concern 2139 images in total. We can see that the data is not evenly balanced between classes. At top level we have 22 % of the images labelled with A3, 64% with A4, and 14 % for A5. Within these top-level classes, we also have imbalanced categories, e.g., A4.1 and A4.7 within A4. Moreover, some sub-categories have very few labelled images, for instance A5.13 and A5.23 we have less than 10, and for many of the sub-categories have less than 100 images.

We decided to cover the top-level A3, A4, and A5 categories only, as these allow for a model that covers categories at the same level and there is enough data to sample in all of them. This was also done for simplicity at the beginning of our development, and over the course of our work

Table 3.1: Data labelled by biologists

Category	Images
A3 – Infralittoral rock and other hard substrata	**484**
A3.1 – Atlantic and Mediterranean high energy infralittoral rock	388
A3.11 – Kelp with cushion fauna and/or foliose red seaweeds	113
A3.7 – Features of infralittoral rock	96
A3.71 – Robust faunal cushions and crusts in surge gullies and caves	44
A4 – Circalittoral rock and other hard substrata	**1385**
A4.1 – Atlantic and Mediterranean high energy circalittoral rock	1207
A4.7 – Features of circalittoral rock	178
A4.71 – Communities of circalittoral caves and overhangs	178
A4.711 – Sponges, cup corals and anthozoans on shaded or overhanging circalittoral rock	20
A5 – Sublittoral sediment	**300**
A5.1 – Sublittoral coarse sediment	181
A5.13 – Infralittoral coarse sediment	2
A5.14 – Circalittoral coarse sediment	130
A5.2 – Sublittoral sand	85
A5.23 – Infralittoral fine sand	4
A5.25 – Circalittoral fine sand	81
A5.4 – Sublittoral mixed sediments	34
A5.44 – Circalittoral mixed sediments	34
Total	**2169**

Table 3.2: Dataset used for model development

Category	Total	Train	Validation	Test
A3 – Infralittoral rock and other hard substrata	300	210	45	45
A4 – Circalittoral rock and other hard substrata	300	210	45	45
A5 – Sublittoral sediment	300	210	45	45
Total	900	630	135	135

```
datagen = ImageDataGenerator(rescale=1.0/255.0)
train_dataset = datagen.flow_from_directory(
    'dataset/train', target_size=(64,64), class_mode='sparse')
validation_dataset = datagen.flow_from_directory(
    'dataset/validation', target_size=(64,64), class_mode='sparse'
)
test_dataset = datagen.flow_from_directory(
    'dataset/test', target_size=(64,64), class_mode='sparse'
)
```

Listing 3.1: Dataset – definition of train, validation, and test sets in Keras

we did not explore models that cover at least some of the A3/A4/A5 subcategories. The volume of data could potentially be enough in some of the sub-categories, and some lack of images could be compensated partially by image augmentation techniques for instance.

For the dataset to be used in model training, we sampled 300 images per each of the A3, A4,

and A5 categories. Per each category, we also split the data in three sets: 70% for the training set (210 images), 15% for the validation set (45), and the remaining 15% for the test set (45). The dataset was organised in three folders (training, validation, and test), with each of these containing in turn sub-folders per each of the A3/A4/A5 categories. The data at stake can be read using the Keras API, as illustrated in the code of Listing 3.1. Note that the `target_size` parameter indicates the image resolution to be used, and varies from model to model, and RGB pixels can be converted in some of the models from the $[0, 255]$ integer range to a $[0, 1]$ floating point interval.

3.1.2 Simple models

We derived two simple models that have the potential of serving as a baseline both in terms of predictive power and performance. Due to their relative simplicity, we expect them to have limited accuracy but very fast prediction times. The first model, nicknamed Mnist, has an architecture taken from a Tensorflow official tutorial [43] over the classic MNIST dataset. The second model, nicknamed Udacity, is taken from an Udacity course on deep learning by Tensorflow [45]. Using Jupyter notebooks [14] running Python, we programmed all the steps necessary to define and train these models with the Keras API, and then save them onto a convenient format for use in the onboard software later using the TFLite API.

The Mnist model is illustrated in Figure 3.1 in terms of the definition through the Keras API (left), and a corresponding visualisation using Netron [37]. As shown, the network takes a 64×64 RGB image as input, and produces a 1×3 vector output corresponding to the 3 EUNIS classes. The internal layers correspond to a preliminary flattening of an image (conversion to a 1D vector), followed by two fully-connected layers (identified as `Dense` in the Keras API). The second fully-connected layer produces the output with a softmax activation. A dropout of 0.2 is also configured to prevent overfitting during training. Note that no convolutions are used, hence technically the network is not a CNN.

The Udacity model is illustrated in Figure 3.2, again in terms of its definition using the Keras API and a visualisation in Netron. It has 14 layers and, unlike Mnist, it uses 3 pairs of 2D convolutions and max-pooling layers. The final layers are similar to Mnist though: a flattening layer is followed by two fully-connected layers. Two dropout factors are configured though, as we illustrate in Chapter 4, they are not sufficient to prevent overfitting of the training data.

Each of the two models, Mnist and Udacity, has been trained using the Keras API as shown in Listing 3.2. The model is first configured for training using the `Model.compile()` Keras API function, specifying metrics to monitor during training, the model's optimiser, and the loss function to use. As shown, in terms of metrics we monitor accuracy, the ratio of correctly predicted classes, by taking the class classified with highest probability through the final softmax activation. The loss function is used to compute the measure of the error in approximating ground truth values during training, and we use the cross-entropy loss function. If a CNN has a softmax activation for n classes with output probability values $(p_1, \ldots, p_n)$, and the ground truth

values are $[t_1, ..., t_n]$ then the cross-entropy loss is defined as $L_{CE} = -\sum_{i=1}^{n} t_i \log(p_i)$. Given that in classification problems like ours, we have one ground truth value $t_c = 1$, where c is the correct class, and all other values are 0, then $L_{CE} = -\log(p_c)$ (if p_c equals 1, the loss is of course 0). For N items with ground truth classes $c_1, ..., c_N$ Tensorflow evaluates the loss by taking the average of L_{CE} values, i.e.,

$$L = \frac{1}{N}\sum_{i=1}^{N} L_{CE}(c_i) = -\frac{1}{N}\sum_{i=1}^{N} \log(p_{c_i}).$$

The optimiser refers to the algorithm to use to adjust CNN weights and, in this case, a stochastic gradient descent method known as the "Adam algorithm" [15] is used.

```
model.compile(
    metrics=['accuracy'],
    loss=tf.keras.losses.SparseCategoricalCrossentropy(),
    optimizer='adam')
stop_early = tf.keras.callbacks.EarlyStopping(
                monitor='val_loss',
                patience=10,
                restore_best_weights=True)
max_epochs = 50
history = model.fit(
     train_dataset,
     epochs=max_epochs,
     validation_data=validation_dataset,
     callbacks=[stop_early])
```

Listing 3.2: Keras code for training a model

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(64,64,3)),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(3, activation='softmax')
])
```

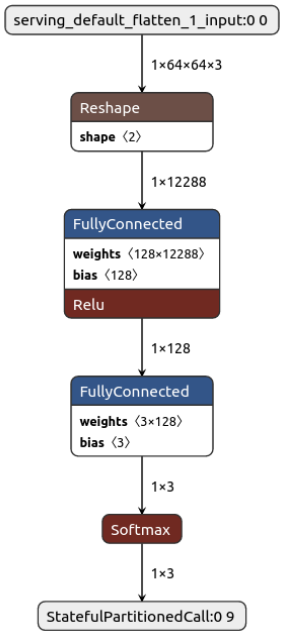


Figure 3.1: The "Mnist" model

```
model = Sequential()
model.add(Conv2D(16, 3, padding='same',
                 activation='relu', input_shape=(64,
                     64, 3)))

model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(32, 3, padding='same',
                 activation='relu'))

model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(64, 3, padding='same',
                 activation='relu'))

model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dropout(0.2))
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(3, activation='softmax'))
])
```

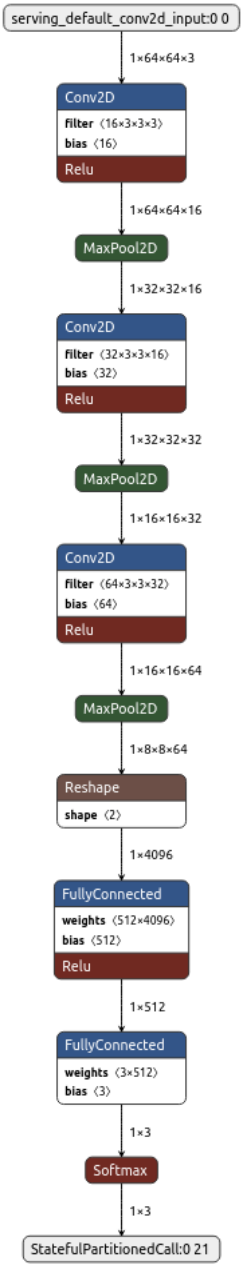


Figure 3.2: The "Udacity" model

After the model is configured for training, actual training takes place using the the `Model.fit()` Keras API function. As shown, we specify as parameters the training and validation datasets, and set 50 as the maximum number of epochs for training (50). In each epoch, the optimiser (the Adam algorithm) will adjust CNN weights taking into account the outputs for the training dataset. Weight adjustment is done in batches of 32 images (by default) during each epoch. At the end of each epoch, accuracy and loss values can be determined both for the training and validation datasets. The code shows that we define an early stop criteria so that training stops if no improvements are observed for the loss function in the validation dataset after 10 epochs. The weights of the final CNN will be those of the epoch that minimises validation loss.

```
model.save('udacity')
converter = tf.lite.TFLiteConverter.from_saved_model('udacity')
tflite_model = converter.convert()
with open('udacity.tflite', 'wb') as f:
      f.write(tflite_model)
```

Listing 3.3: Conversion of a model to TFLite format

After model training is completed, we need to convert it to TFLite format for use with the onboard classifier. This is done as illustrated in Listing 3.3. We first save the model onto the Keras format, known as SavedModel, using `Model.save()`. We then use the TFLite API to convert the model onto the TFLite format.

3.1.3 Transfer learning using MobileNet

The next model we consider has the MobileNetV2 [38] architecture, a 67-layer deep network, adjusted through transfer learning. Transfer learning reuses part of an already trained CNN

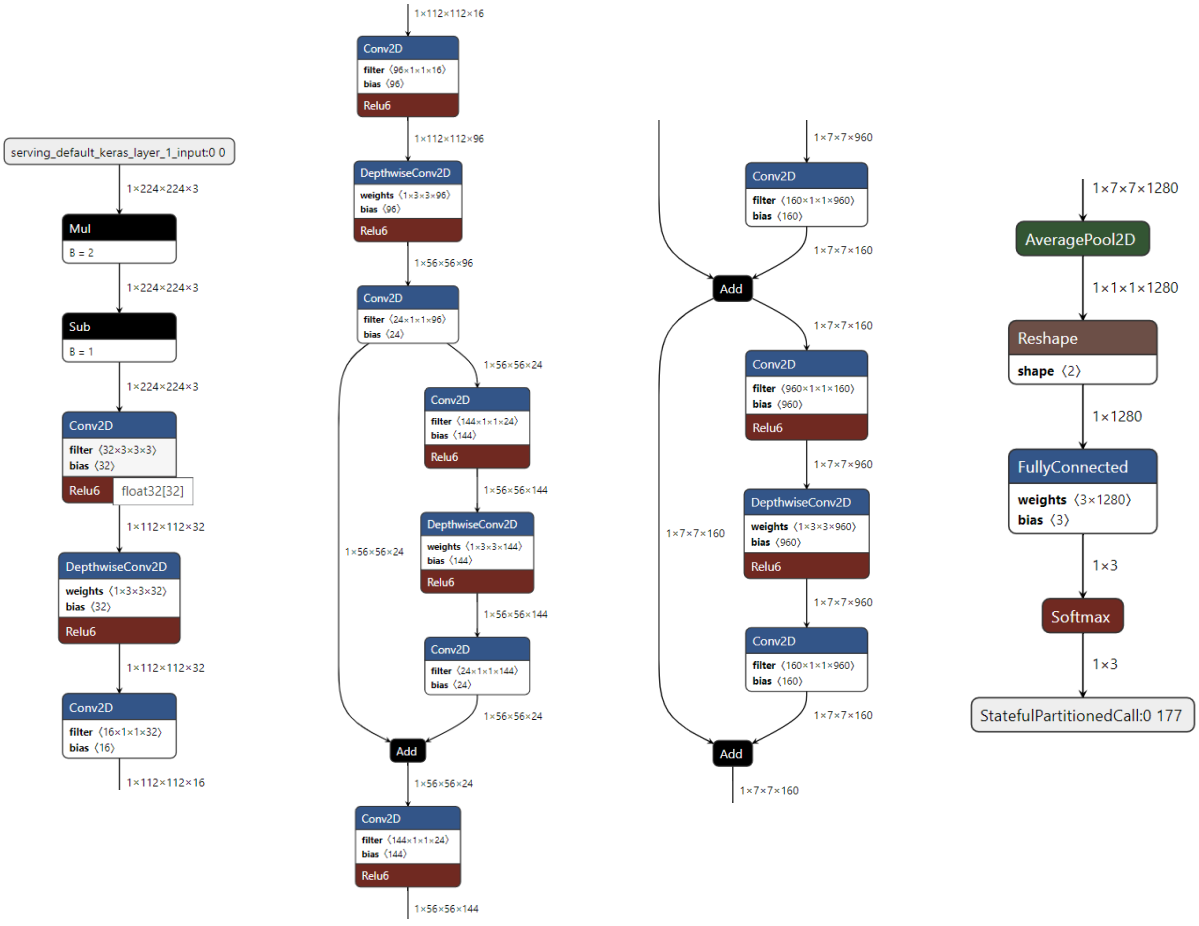


Figure 3.3: MobileNetV2 architecture (fragments)

```
model = tf.keras.Sequential([
  hub.KerasLayer(
     "https://tfhub.dev/google/tf2-preview/mobilenet_v2/feature_vector/4",
     output_shape=[1280],
     trainable=False,
     input_shape=(224,224,3)),
   tf.keras.layers.Dense(3, activation='softmax')])
```

Listing 3.4: Transfer Learning using MobileNetV2

in order to define a new model, in contrast to training a CNN with the same architecture from scratch. Quite often transfer learning results in a much shorter training time without compromising the quality of the resulting model. MobileNetV2 is an evolution of a family of CNNs, adequate for use in mobile and embedded devices known as MobileNets [13].

The code we used to define the CNN using the Keras API is shown in Listing 3.4. The model obtains a pre-trained feature vector of MobileNetV2 available at TensorFlow Hub [44], with a final output of 1280 "features", and defines a new fully-connected output layer with a softmax activation. Only the final layer is then trained as in the simple models, though we could in alternative also adjust the original weights by setting `trainable=True` rather than `False`. The training code is the same as that used for the simple models. The resulting CNN architecture is illustrated partially in Figure 3.3, with the input layer shown left, the output layer shown right, and other repeating patterns for intermediate layers shown middle.

3.1.4 Use of Google AutoML

We derived three additional models using Google AutoML Vision [3], a cloud service for automated training of deep neural networks. Figure 3.4 illustrates the necessary steps necessary to generate an image classification model suitable for use in mobile and embedded devices.

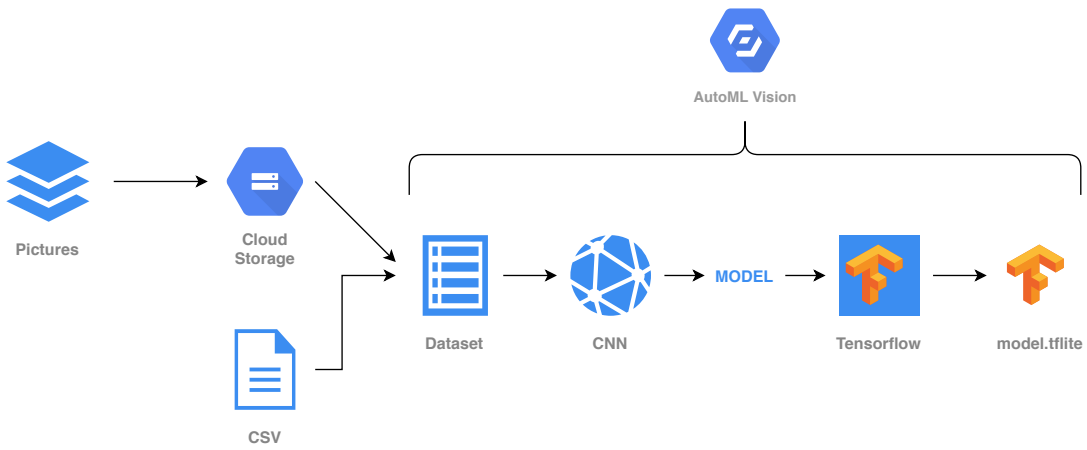


Figure 3.4: Derivation of TFLite models using Google Auto ML

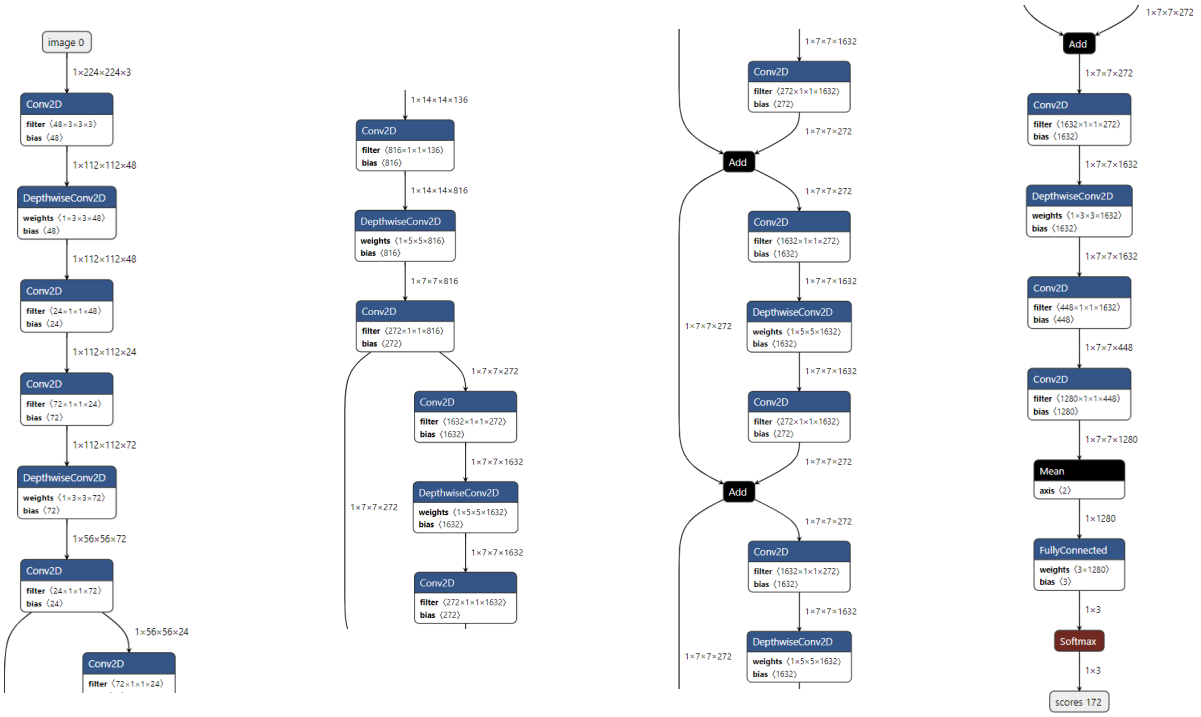


Figure 3.5: AutoML model (slow) architecture fragments

The process starts by defining a dataset through image files plus a CSV file identifying the classes for images and the association of images to training, validation, or test datasets. This data must be stored in a "storage bucket" of the Google Cloud Storage service. Once the dataset is defined, training can be performed at the click of a button, such that the user only needs to choose one of three CNN variants: one for high accuracy, another one for fast prediction times, and a final one representing a trade-off prediction accuracy and speed; we call these the "fast", "slow", and "medium" variants respectively. After training, the resulting model can be obtained in the TFLite format, among other supported formats.

We thus derived three models using the same dataset as for all other models. The slow, medium, and fast variants essentially have the same 65-layer deep CNN architecture, differing only in terms of density (number of weights / connections) at each layer, meaning that the slow variant is the most dense and the fast variant is the least dense. Figure 3.5 depicts the architecture of the slow variant, and Figure 3.3 depicts internal layers of all three variants to illustrate the difference in density per layer. The CNN architectures at stake are from the MnasNet family [40] for mobile and embedded devices, that result from an automated neural architecture search that accounts explicitly for model latency to derive models that can provide a good trade-off between accuracy and latency.

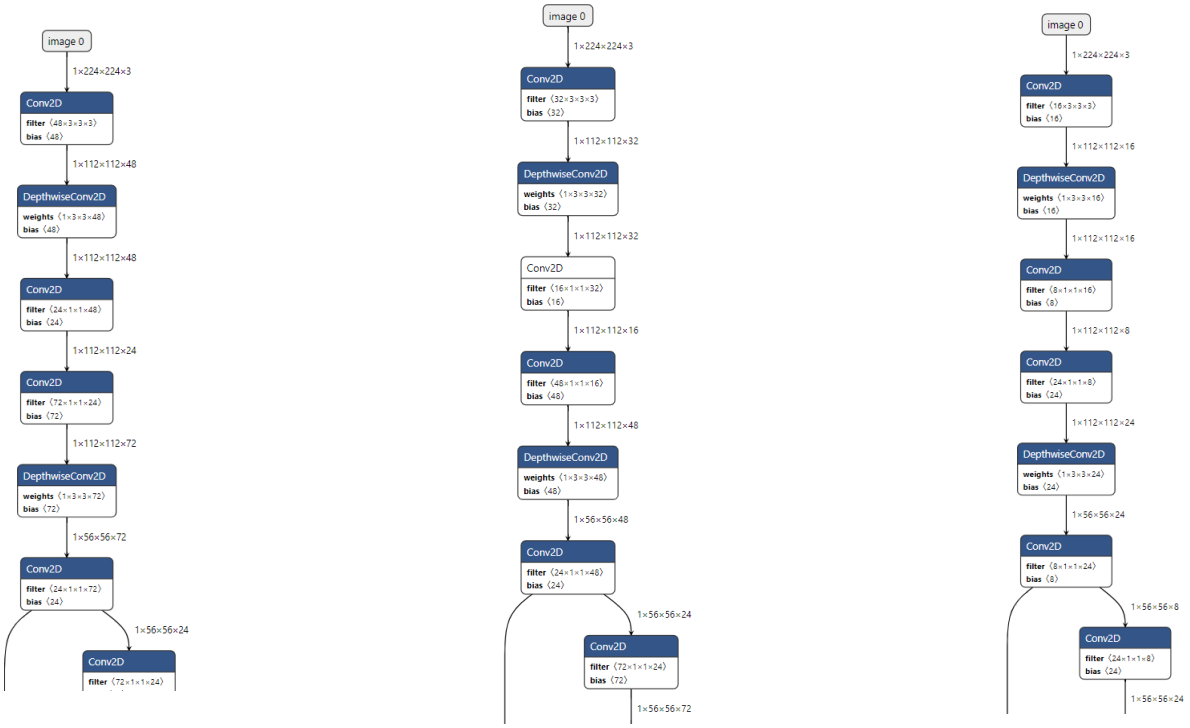


Figure 3.6: Comparison of AutoML architecture variants (slow, medium, and fast, left to right)

3.2 Onboard software

In this section, we present the onboard software for image classification in terms of overall architecture, main aspects of implementation, extensions to the IMC message standard, and helper scripts.

3.2.1 Architecture

The architecture of the software for onboard image classification is illustrated in Figure 3.7. The figure depicts the overall role of what we call the **image classification actor** (ICA). As shown, the purpose of the ICA to capture and classify video frames from a camera or alternatively, for simulation purposes, a video file.

The ICA is controlled through an IMC message called `ImageClassificationControl` that can be received over the network. This control message configures, starts or stops image classification, depending on the value of a command field. For configuration, the message specifies the model to use, the video source, and the image sampling frequency. Over time, any of these aspects may be changed. After setup, and once image classification is activated, the ICA transmits back classification results through another IMC message called `ImageClassification`, itself containing several inner `ScoredClassification` messages that indicate the confidence level per each class in the model.

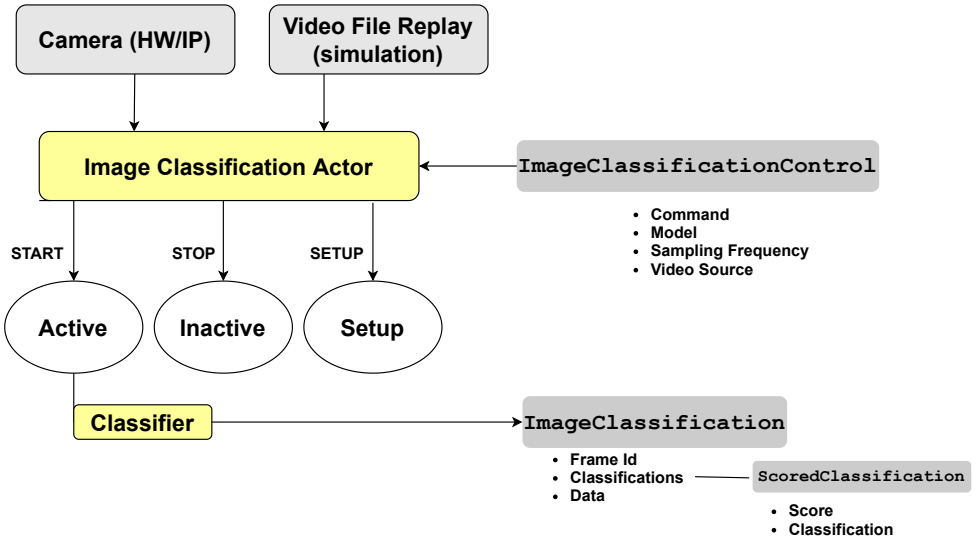


Figure 3.7: Architecture of the onboard software

The ICA interactions over the network take place using User Datagram Protocol (UDP), facilitating integration with other modules like DUNE or Neptus, since UDP is the simplest and most commonly used network protocol used by the LSTS toolchain. By default, the ICA listens on UDP port 6011 and transmits results to port 6012 to the local host. These settings can be configured (check details in the GitHub repository [28]).

3.2.2 Implementation

The ICA is implemented by a Python class called `ImageClassificationActor`, with the skeleton shown in Listing 3.5.

As illustrated by the Python `import` statements in the code, the ICA makes use of the OpenCV (`cv2`) and pyIMC (`pyimc`) libraries. OpenCV is used for video frame acquisition and image file manipulation, while pyIMC provides IMC message bindings and support for IMC-based networked interactions. The ICA also uses the TFLite library with access encapsulated in the `tfmodel` module that is programmed separately, and discussed later in this section.

The ICA class is derived from super-class `DynamicActor` that is part of pyIMC, and includes three core methods:

- `__init__` – the class constructor, that performs one-time initialisation actions.
- `on_Classification_Control` – the handler for incoming `ImageClassificationControl` messages. As shown, the reception of different commands may cause the ICA to either (re)configure classification parameters, and start or stop image classification.
- and `classification_loop` – a method that is called every 1 millisecond to perform image classification. As shown, the ICA proceeds with classification only when active and

```
...
import cv2
...
import pyimc
import tfmodel
...
from pyimc.actors.dynamic import DynamicActor
from pyimc.decorators import Subscribe, RunOnce, Periodic
from pyimc.node import IMCService
...
class ImageClassificationActor(DynamicActor):
  ...
  def __init__(self, parameters):
    ...
  @Subscribe(pyimc.ImageClassificationControl)
  def on_Classification_Control(self, msg: pyimc.ImageClassificationControl):
    logging.info('Received control message -- {}'.format(msg))
    if msg.command == pyimc.ImageClassificationControl.CommandEnum.SETUP:
      ... perform setup ...
    elif msg.command == pyimc.ImageClassificationControl.CommandEnum.START:
      ... start classifying images ...
    elif msg.command == pyimc.ImageClassificationControl.CommandEnum.STOP:
      ... stop classifying images ...
    else:
      logging.error('invalid command received')

  @Periodic(0.001)
  def classification_loop(self):
    if self.mode != self.MODE_ACTIVE:
       return
    current_time = time.time()
    if current_time < self.next_capture_time:
      return
    self.next_capture_time += 1.0 / self.fps
    ret, frame = self.video_source.read()
    if not ret:
       logging.error('failed to grab frame - end of video stream reached?')
       self.reset()
       return
    if  current_time < self.next_classif_time:
       return
    self.next_classif_time += 1 / self.setup.sampling_freq
    ... classify image ...
```

Listing 3.5: ICA class structure

according to the configured rate for classification and the video source's frame rate. The classification actions within `classification_loop` are shown in detail in Listing 3.6. It comprises three basic steps:

```
    @Periodic(0.001)
    def classification_loop(self):
        ...
        cl_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
        try:
            results = self.classifier.classify(cl_frame)
        except:
            traceback.print_exc()
            return
        classification_time = time.time() - current_time
        logging.info('Classification time: %.3f s' % classification_time)
        logging.info(results)
        self.frame_counter += 1
        img_name = '{}/f{:05d}.png'.format(self.data_dir, self.frame_counter)
        cv2.imwrite(img_name, frame)
        logging.info('{} written!'.format(img_name))

        ic_msg = pyimc.ImageClassification()
        ic_msg.frameid = self.frame_counter
        for (label, score) in results:
            sc_msg = pyimc.ScoredClassification()
            sc_msg.score = int(round(score*100))
            sc_msg.classification = label
            ic_msg.classifications.append(sc_msg)
        logging.info(ic_msg)
        compressed_frame = cv2.resize(frame, (128,128))
        ic_msg.data = cv2.imencode('.png',compressed_frame)[1].tobytes()

        self.log_message(ic_msg)
        self.send_static(ic_msg, set_timestamp=False)
        self.show_image(frame)
```

Listing 3.6: ICA classification code

1. The image pixels are converted from the default BGR (Blue-Green-Red) representation employed by OpenCV onto RGB, and then submitted to the TFLite classifier.
2. The `ImageClassification` message encoding the classification results is built and logged onto the local filesystem. The image is also saved onto disk.
3. Finally, the `ImageClassification` message is transmitted over the network.

Listing 3.7 illustrates an execution log fragment of the ICA, during which image classification is setup and then started.

```
$ ./ica.py
2021-08-05 09:14:02 - INFO - starting
2021-08-05 09:14:02 - INFO - resetting internal state
...
2021-08-05 09:14:06 - INFO - Received control message -- ImageClassificationControl
FFFF:FF -> FFFF:FF
2021/08/05 09:14:06
    command: 2
    model: autoML
    video_source: example.mjpg
    sampling_freq: 1
2021-08-05 09:14:06 - INFO - resetting internal state
2021-08-05 09:14:06 - INFO - setting up
...
2021-08-05 09:15:43 - INFO - Received control message -- ImageClassificationControl
FFFF:FF -> FFFF:FF
2021/08/05 09:15:43
    command: 0
    model:
    video_source:
    sampling_freq: 0
2021-08-05 09:15:43 - INFO - now active
2021-08-05 09:15:43 - INFO - Classification time: 0.099 s
2021-08-05 09:15:43 - INFO - [('A4', 0.4823529411764706), ('A3', 0.4196078431372549), ('A5', 0.10196078431372549)]
2021-08-05 09:15:43 - INFO - /home/edrdo/Onboard-Image-Classification/data/autoML/20210805/091541/f00001.png written!
2021-08-05 09:15:43 - INFO - ImageClassification
FFFF:FF -> FFFF:FF
    frameid: 1
    classifications: [{
      abbrev: ScoredClassification
      score: 48
      classification: A4
    }{
      abbrev: ScoredClassification
      score: 42
      classification: A3
    }{
      abbrev: ScoredClassification
      score: 10
      classification: A5
    }]
    data:
...
```

Listing 3.7: ICA example execution

3.2.3 TFLite interface

The interface with the TFLite API is encapsulated in a class called `Classifier`, defined in a module called `tfmodel`. The main parts of the code are shown in Listing 3.8. As shown, the class defines only two methods: `__init__`, the constructor, and the `classify` method for subsequent image classification. The constructor initialises the TFLite interpreter and associated data, loading the model file according to the given path and model identifier. The classification method takes an input image, resizes it to fit the model's input layer, and then finally submits the image to the TFLite interpreter. The output of the interpreter is then retrieved and returned as a list of `(class, score)` pairs ordered by confidence score (higher scores first).

```
...
import tensorflow as tf
...
class Classifier:
  def __init__(self, path, model):
    ...
    model_path = '{}/{}/model.tflite'.format(path, model)
    ...
    self.interpreter = tf.lite.Interpreter(model_path)
    self.interpreter.allocate_tensors()
    self.input_details = self.interpreter.get_input_details()
    self.output_details = self.interpreter.get_output_details()
    self.floating_model = self.input_details[0]['dtype'] == np.float32
    self.height = self.input_details[0]['shape'][1]
    self.width = self.input_details[0]['shape'][2]

  def classify(self,frame):
    input_data = np.expand_dims(cv2.resize(frame, (self.width, self.height)),
        axis=0)
    if self.floating_model:
      input_data = np.float32(input_data) / 255.0
    self.interpreter.set_tensor(self.input_details[0]['index'], input_data)
    self.interpreter.invoke()
    output_data = self.interpreter.get_tensor(self.output_details[0]['index'])
    results = np.squeeze(output_data)
    if not self.floating_model:
        results = results / 255.0
    results = sorted(zip(self.labels, results), key=lambda k: - k[1])
    return results
```

Listing 3.8: TFLite classifier

3.2.4 IMC messages

For reference, the detailed definition of the IMC messages we use with the ICA is provided in Table 3.3, and the corresponding XML specification fragment is provided in Listing 3.9. The pyimc library takes the XML specification of IMC to generate the corresponding Python bindings for each type of message.

Table 3.3: IMC messages used by the ICA

IMC message	Field Name	Field Type	Summary
ImageClassificationControl	command	uint8_t	Command flag (0: START, 1: STOP, 2: SETUP)
	model	plaintext	Identifier of model (required for SETUP command).
	video_source	plaintext	Identifier of video source path (required for SETUP command).
	sampling_freq	fp32_t	Image classification frequency in Hz (required for SETUP command)
ImageClassification	frame_id	uint32_t	Unique identifier for frame.
	classifications	message-list	Classifications, a list of messages of type ScoredClassification.
	data	rawdata	Original image frame, possibly compressed.
ScoredClassification	score	uint8_t	Classification score, from 0 to 100.
	classification	plaintext	Classification label.

```
<message id="704" name="Image Classification Control" abbrev="ImageClassificationControl" source="ccu, vehicle">
  <field name="Command" abbrev="command" unit="Enumerated" type="uint8_t" prefix="ICC">
    <value id="0" name="Start" abbrev="START"/>
    <value id="1" name="Stop" abbrev="STOP"/>
    <value id="2" name="Setup" abbrev="SETUP"/>
  </field>
  <field name="Model" abbrev="model" type="plaintext"/>
  <field name="Video Source" abbrev="video_source" type="plaintext"/>
  <field name="Sampling Frequency" abbrev="sampling_freq" type="fp32_t" min="0" unit="Hz"/>
</message>
<message id="705" name="Image Classification" abbrev="ImageClassification" source="ccu, vehicle">
  <field name="Frame Id" abbrev="frameid" type="uint32_t"/>
  <field name="Classifications" abbrev="classifications" type="message-list" message-type="ScoredClassification"/>
  <field name="Data" abbrev="data" type="rawdata"/>
</message>
<message id="706" name="Scored Classification" abbrev="ScoredClassification" source="ccu, vehicle">
  <field name="Score" abbrev="score" type="uint8_t" min="0" max="100"/>
  <field name="Classification" abbrev="classification" type="plaintext"/>
</message>
```

Listing 3.9: IMC messages used by the ICA – XML specification

3.2.5 Helper scripts

In addition to the ICA program, several utility scripts have been developed. Detailed usage info is available at GitHub [28]. Here we just provide a basic summary of their functionality:

- `icmsg.py` - can be used to send an `ImageClassificationControl` message to the ICA, e.g.

```
$ ./icmsg.py -m mobilenet -s 10 -v /dev/video0 setup
Message sent to 127.0.0.1:6011 ...
ImageClassificationControl
FFFF:FF -> FFFF:FF
2021/08/05 08:49:25
    command: 2
    model: mobilenet
    video_source: /dev/video0
    sampling_freq: 10
```

- `iclisten.py` - can be used to listen to `ImageClassification` and other IMC messages from the ICA, e.g.,

```
$ ./iclisten.py
ImageClassification
FFFF:FF -> FFFF:FF
    frameid: 1
    classifications: [{
      abbrev: ScoredClassification
      score: 48
      classification: A4
    }{
      abbrev: ScoredClassification
      score: 42
      classification: A3
    }{
      abbrev: ScoredClassification
      score: 10
      classification: A5
    }]
    data:
...
```

- `ictest.py` - can be used to test the classification of model over given input images, e.g.,

```
$ ./ictest.py -m autoML_slow examples/m*.jpg
INFO: Initialized TensorFlow Lite runtime.
--- examples/m3_1.jpg ---
```

```
1: A3 93
2: A4 4
3: A5 3
--- examples/m3_2.jpg ---
1: A3 95
2: A4 3
3: A5 2
...
--- examples/m5_2.jpg ---
1: A5 95
2: A3 3
3: A4 3
--- examples/m5_3.jpg ---
1: A5 93
2: A4 4
3: A3 4
```

- `mstats.py` – can be used to calculate the following statistics for the performance of a model applied to all files in a directory: accuracy, loss, and average classification time:

```
$ ./mstats.py mnist ds/dataset/test
Files: 135
Accuracy: 0.84
Loss: 0.49
Time p/image (ms): 2.57
```

3.3 GitHub repository

To end this chapter, we provide an overview of the contents of the GitHub repository holding our developments [28]. Table 3.4 lists the main contents. In summary:

- The top-level directory contains the Python source code for onboard image classification, including the image classification actor, and a `README.md` file explaining how to use the scripts;
- The `notebooks` directory contains the Jupyter notebooks we used to develop the Mnist, Udacity, and MobileNet models;
- The `models` directory contains the derived models in TFLite format, one sub-directory per model.

Table 3.4: Contents of the GitHub repository

Dir/File	**Description**
/	Root directory
ica.py	ICA implementation.
tfmodel.py	TFlite API interface (`Classifier` class)
icmsg.py	Utility script to send `ImageClassificationControl` messages
iclisten.py	Utility script to listen to `ImageClassification` messages
ictest.py	Utility script to test image classification
mstats.py	Utility script to derive model statistics
notebooks	Jupyter notebooks
mnist_model_train.ipynb	Derivation of "Mnist" model
udacity_model_train.ipynb	Derivation of "Udacity" model
mobilenet_model_train.ipynb	Derivation of MobileNetV2 model
models	TFlite models
mnist	"Mnist" model
udacity	"Udacity" model
mobilenet	MobileNetV2 model
autoML_fast	Google Auto ML model – "fast" variant
autoML_medium	Google Auto ML model – "medium" variant
autoML_slow	Google Auto ML model – "slow" variant

Chapter 4

Results

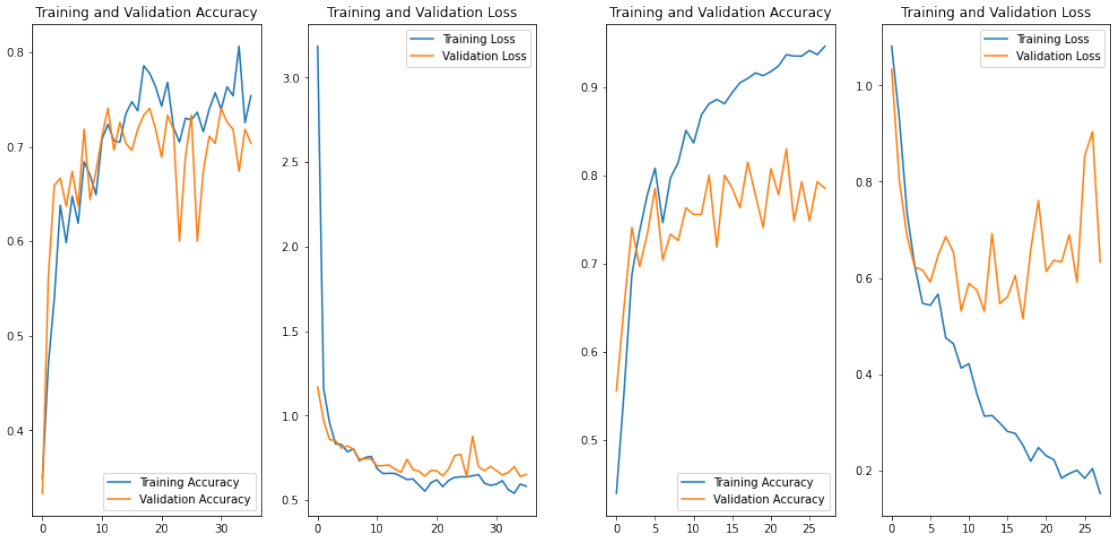
This chapter presents results we obtained for the developments described in the previous chapter. We first present training results for the CNN models (Section 4.1), This is followed by an overall comparison of all CNN models in regard to model complexity and predictive power (Section 4.2). Finally, we present performance results for the onboard classification software in three distinct hardware platforms regarding predictions times per model and per platform, and resource usage in terms of CPU and RAM (Section 4.3).

4.1 Training results

As explained in the previous chapter, we followed two distinct methodology to derive CNN models: (1) through direct use of the Keras API within Jupyter notebooks running in Google Colab for the simple Mnist and Udacity models, plus the CNN derived through transfer learning for MobileNet; and (2), the use of the Google Auto ML cloud service for the other three remaining models. For methodology (1) we gathered the data regarding the evolution of accuracy and loss for the sequence of training epochs. For methodology (2) we could only obtain some brief information regarding the training process through the Google Auto ML service. We present these now.

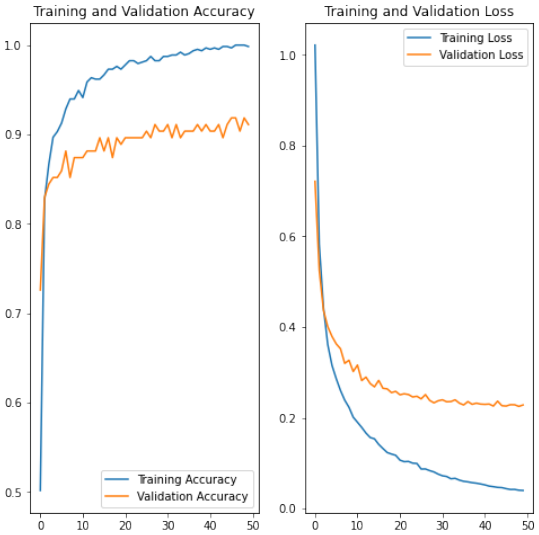
Figure 4.1 provides plots for the evolution of accuracy and loss for the train (70 % of data) and validation (15 %) data in each of the Mnist (a), Udacity (b), and Mobilenet (c) models (the remaining 15 % of the data is used as the test dataset; see next section). As explained in the previous chapter, a maximum of 50 training epochs was set, along with an early-stop criteria that prevents further training epochs if validation loss has not been improved in the last 10 epochs, and the weights of the epoch with the minimum validation loss are kept in the resulting CNN.

The number of training epochs was 36 for Mnist, 28 epochs for Udacity, and 50 for Mobilenet (the maximum possible value). The training time in a Google Colab notebook environment was of approximately 8 minutes for Udacity and Mnist, and 15 minutes for Mobilenet. For Mnist and Udacity we can observe that accuracy and loss both tend to improve more or less steadily



(a) Mnist

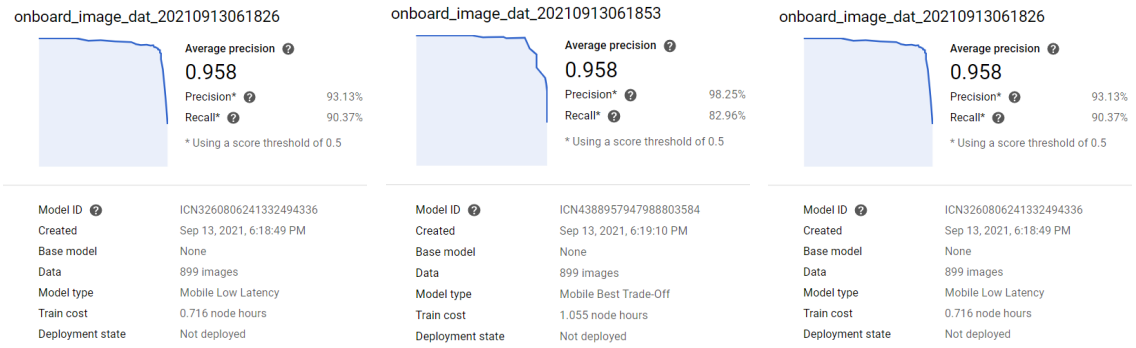
(b) Udacity



(c) Mobilenet

Figure 4.1: Training results – Mnist, Udacity and Mobilet models

for the training data , but this does not happen for the validation data. Validation accuracy and loss clearly have variability and fail to improve after some initial epochs. This is a sign that the model is over-fitting the training data. Observe the case of Udacity in particular, where for the training data accuracy evolves close to 100% and loss evolves close to 0. In contrast, the results for Mobilenet show steady improvement during training both for the training data set and the validation data set, and both for accuracy and loss. The maximum number of epochs was used in the case of Mobilenet, so it is possible that the model could be trained further to achieve better values for the metrics. Clearly, the Mobilenet model seems more promising to use in terms of predictive power, as we reach a validation data accuracy close to 90% and validation loss close to 0.3, Accuracy and loss values are put in perspective in more detail in the next section.



(a) AutoML fast (b) AutoML medium (c) AutoML slow

Figure 4.2: Training results – Google Auto ML models

Regarding Google Auto ML models, we could only obtain the UI feedback for training shown in Figure 4.2 respectively for the low, medium, and fast variants. The only information available is the number of "node hours" used during training, where a node refers to a virtual machine hidden from the user that may use a GPU or a TPU (details are not reported), along with metrics for the derived model evaluated over the test data set.

4.2 CNN architecture comparison

Table 4.1 provides a comparative summary of all developed models. The number of layers, number of parameters in all layers (input, outputs, and trainable weights in intermediate layers), along with the accuracy and loss values for test images in our dataset. For the Mnist, Udacity, and Mobilenet models, the network architecture details were directly obtained using the `Model.summary()` Keras API method before conversion to TFLite format in our model training notebooks. In the case of AutoML models, given that we had difficulties in obtaining and reading models in a Keras-compatible format, we used STM32CubeMX [31] instead and over TFLite files directly. The accuracy and loss values were calculated using a Python script.

Table 4.1: CNN architecture comparison

Model	Layers	Parameters (10^6)	File size (MB)	Accuracy (%)	Loss
Mnist	4	1.6	6.3	84	0.49
Udacity	11	2.1	8.5	86	0.36
Mobilenet	68	2.3	8.9	90	0.32
AutoML fast	66	0.5	0.6	93	0.28
AutoML medium	65	3.1	3.2	92	0.39
AutoML slow	65	5.8	5.9	93	0.24

The relation between model depth (number of layers) and classification accuracy is clear. Deeper models have higher accuracy. The AutoML and Mobilenet networks have an accuracy of

90 % or more, but in any case the simple Mnist and Udacity models still have reasonably good accuracy, 84 and 86 % respectively. The results also indicate that depth is more relevant than density (the number of parameters). For instance, the accuracy values are almost the same for all the AutoML models that have a similar architecture except for the layer densities. The AutoML fast variant has a much lower number of parameters than the medium and slow variants, and in fact also all other models, without compromising accuracy. These observations regarding model depth and density overall apply also to the loss values, except for the autoML medium variant which has a somewhat high loss value of (0.39, only the Mnist model has a higher loss value).

4.3 Onboard classification results

To evaluate the performance of the onboard classification software we took measures for model inference times, as well as the CPU and RAM usage of the image classification actor. The inference time per model was derived by measuring the average elapsed time for classifying an image in the test dataset (135 images). The CPU and RAM usage were measured during executions of the image classification actor when idle, or when running image classification for a test video at different classification frequencies. We collected results for three distinct hardware platforms: Raspberry PI 4, a Jetson Nano, and a computer laptop.

Table 4.2: Onboard classification tests – hardware platforms

Platform	CPU	RAM	GPU	Operating system
Raspberry Pi 4	ARM Cortex-A72 Quad-Core 1.5 GHz	8	Broadcom VideoCore VI	Raspbian Linux 10 (buster)
Nvidia Jetson Nano	ARM Cortex-A57 Quad-Core 1.43 GHz	4	128 core Maxwell GPU 921 Mhz	Ubuntu Linux 18.04
HP laptop	Intel i5-1035G1 octa-core 1 GHz	8	Intel Corporation Device 8a56	Ubuntu Linux 20.04.3

4.3.1 Hardware platforms

The characteristics of the hardware platforms are given in Table 4.2. Even if we list GPU characteristics per each platform, GPUs could not be activated with our framework due to technical reasons. The reason is that TFlite has a module called the GPU delegate that we could not activate properly using the Python API bindings. The use of GPUs with TFLite or TensorFlow is overall problematic both with the Raspberry Pi and NVidia. For the Raspberry Pi there are compliance problems with Compute Unified Device Architecture (CUDA) drivers, and there similar issues for NVidia Jetson (e.g., see [36]). There are custom versions of TensorFlow provided by Nvidia [26] but GPU support does not cover TFLite in these versions.

4.3.2 Prediction times

Table 4.3: Onboard classification tests – average prediction per model and platform (ms)

Model	Raspberry Pi	Jetson Nano	HP laptop
Mnist	2.47	6.36	0.74
Udacity	6.06	11.62	2.33
Mobilenet	94.54	107.02	23.39
AutoML fast	48.51	31.51	15.66
AutoML medium	127.73	86.54	43.92
AutoML slow	183.35	131.24	67.65

Table 4.3 lists the average prediction time in milliseconds (ms) when classifying images per each model and platform. The inference times show that it is feasible to deploy the models on embedded hardware platforms / autonomous vehicles. Overall the inference times are not higher than 200 ms (the worst performance obtained is 183 ms for AutoML slow on the Raspberry). This translates us an image sampling frequency of at least 5 Hz, which would be quite enough for an Autonomous Underwater Vehicle (AUV) diving underwater to provide adequate spatial resolution, as these vehicles only go as fast as 1-2 meters per second. Much higher frequencies can be obtained even for deep CNN models. For instance, the AutoML fast model can run with approximate frequency of 20 Hz frequency on the Raspberry Pi and 30 Hz on the Nvidia Jetson Nano.

Inference times are without surprise linked to model complexity. Simple models like Mnist and Udacity have clearly the fastest prediction times, lower than 12 ms in all platforms. In contrast, all the other models have much higher prediction times, comparatively per platform: 48 to 184 ms vs. 2 to 6 ms on Raspberry PI, 32 to 131 ms vs. 6 to 12 ms on Jetson Nano, and 16 to 68 ms vs. 1 to 2 ms on the HP laptop. Overall, in terms of platforms: the HP laptop is clearly the fastest, and the Raspberry PI has worse results than the Nvidia Jetson for the AutoML models but better results for the other models.

4.3.3 RAM and CPU usage

Table 4.4: Onboard classification tests – RAM and CPU usage

System + model	RAM (average GB)	CPU (idle)	CPU (1 Hz)	CPU (7 Hz)
Raspberry Pi - Mnist	1.9	33	49	98
Raspberry Pi - AutoML slow	1.8	33	64	100
Jetson Nano - Mnist	1.3	37	48	100
Jetson Nano - AutoML slow	1.3	37	59	100
Hp Laptop - Mnist	1.5	24	32	81
Hp Laptop - AutoML slow	1.5	24	37	113

Table 4.4 lists measures for RAM and CPU usage of the image classification actor during execution. The measures are taken for average values of one minute of execution of the actor in

three cases: idle operation (no classification taking place), a classification rate of 1 Hz, and a higher classification rate of 7 Hz that is equal to the frame rate of the test video. We only provide results for the most and least lightweight models, Mnist and AutoML slow, which are sufficient to illustrate performance. Also, since we observed little variations in RAM usage per model or classification rate, we only list the average value in all executions. Overall, RAM and CPU usage is low taking into account the platforms' characteristics: no more than 2 GB of RAM is used, and the CPU usage only exceeds at most one core (100% = 1 core) in only one case (AutoML slow on the HP laptop). When the actor is idle or operating with a 1 Hz classification frequency, the CPU usage does not exceed 40% and 64% in all platforms, respectively. The differences in values are more noticeable when we compare the use of Mnist and AutoML slow, especially at 1 Hz.

Chapter 5

Conclusion

We conclude with a final discussion of the thesis contributions and future work.

The main contribution of this thesis was the development of a software framework for automated image classification using machine learning that can be deployed in autonomous vehicles. This is in line with the main goal at the beginning of the dissertation, and represents an evolution of the past work in the OMARE project [6, 17] where models were not used in real-time. In conjunction, we have trained various machine learning models for habitat mapping of three EUNIS categories, based on dataset of underwater imagery. The models at stake can be employed in embedded software platforms with good performance, both in terms of computational efficiency and classification accuracy.

There are a few shortcomings and there are also several ideas for future work. We discuss these next.

First of all, since we conducted tests only in a simulated environments, it would be interesting to use the onboard software for real-life scenarios in field tests. We are confident this will not represent a major challenge, given that the simulation results were obtained for some embedded hardware platforms like Raspberry PI or Nvidia Jetson Nano that are appropriate for use in autonomous vehicles. Handling the technical problems regarding the use of a Graphics Processing Unit (GPU) in the embedded software platform will be important, however. There are potential gains in performance that may for instance allow a higher image classification rate. Our results used a sampling rate of up to 7 Hz (the frame rate of the videos used for testing). This frequency or even a lower one are more than adequate for Autonomous Underwater Vehicle (AUV) vehicles, where the speed is usually only 1-2 meters per second. But if we consider vehicles with much higher speeds, Unmanned Air Vehicle for instance, higher sampling rates may be required for appropriate spatial resolution.

The machine learning models we derived for habitat mapping, comprising several different architectures for neural networks, demonstrate the feasibility of our approach. Even simple neural network architectures with only a few layers achieved reasonable accuracy, higher than 80%. The other state-of-the-art deep neural networks we used, like MobileNet and those derived using

Google Auto ML, had very high precision, more than 90%. However, the data set we considered was only large enough to train models for three EUNIS categories using underwater imagery. Further vehicle expeditions in Northern Littoral Natural Park (PNLN) and data annotations by biologists could lead to an improved validation of the models we developed and also more refined models. These did not happen during this dissertation mainly due to difficulties related to the COVID-19 quarantine. Moreover, the consideration of other types of images, e.g., aerial or side-scan sonar imagery, may allow the definition of models for other EUNIS categories.

During the dissertation, we had discussions regarding the use of models for two other purposes distinct from habitat mapping. One was the automatic identification of man-made objects in the sea for pollution monitoring, a highly relevant application for autonomous vehicles. While we believe this is feasible in technical terms, the main challenge may be the construction of data sets for this purpose. We also had talks of using images acquired underwater by a Remotely Operated Vehicle (ROV) that would be processed onboard using machine learning for the classifier and Computer Vision (CV) to estimate the motion of the vehicle or water turbidity. In particular this would be useful to emulate the information used by a AUV but using a cheaper vehicle, since small ROVs that do not have built-in sensors for this purpose but are always equipped with an underwater camera. This could be achieved by using optical flow algorithms to estimate acceleration, velocities, vector of direction, depth to the seabed, among others together with the onboard software classifier, e.g., as in [32].

Bibliography

[1] Tim Berthold, Artem Leichter, Bodo Rosenhahn, Volker Berkhahn, and Jennifer Valerius. Seabed sediment classification of side-scan sonar data using convolutional neural networks. In *2017 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–8, 2017. doi:10.1109/SSCI.2017.8285220.

[2] Ole Christensen. *SUSHIMAP (Survey strategy and methodology for marine habitat mapping)*. PhD thesis, Norwegian University of Science and Technology, 2006.

[3] Google Cloud. Google Auto ML – Cloud Vision. https://cloud.google.com/vision/overview/docs#automl-vision. Last access: September 2021.

[4] Cynthia Davies, Dorian Moss, and Mark Hill. EUNIS habitat classification revised 2004. Technical report, European Environment Agency, 2004.

[5] André Diegues and João Borges Sousa. A survey on automatic habitat mapping. *Instrumentation viewpoint*, 20, 2019.

[6] André Diegues, José Pinto, and Pedro Ribeiro. Automatic Habitat Mapping using Convolutional Neural Networks. In *Proc. IEEE OES Autonomous Underwater Vehicle Symposium (AUV)*. IEEE, 2018. doi:10.1109/AUV.2018.8729787.

[7] Dorian Moss. EUNIS habitat classification - a guide for users, 2008.

[8] Ian Foster and Dennis B. Gannon. *Cloud Computing for Science and Engineering*. The MIT Press, 1st edition, 2017. ISBN: 0262037246.

[9] Janet Franklin. Predictive vegetation mapping: geographic modelling of biospatial patterns in relation to environmental gradients. *Progress in Physical Geography: Earth and Environment*, 19(4):474–499, 1995. doi:10.1177/030913339501900403.

[10] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*, volume 19. MIT Press, 2017. doi:10.1007/s10710-017-9314-z.

[11] Emilio Guirado, Siham Tabik, Domingo Alcaraz-Segura, Javier Cabello, and Francisco Herrera. Deep-learning Versus OBIA for scattered shrub detection with Google Earth Imagery: Ziziphus lotus as case study. *Remote Sensing*, 9:1220, 11 2017. doi:10.3390/rs9121220.

[12] Anabel Gómez-Ríos, Siham Tabik, Julián Luengo, ASM Shihavuddin, Bartosz Krawczyk, and Francisco Herrera. Towards highly accurate coral texture images classification using deep convolutional neural networks and data augmentation, 2018.

[13] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017.

[14] Project Jupyter. Jupyter. https://jupyter.org. Last access: September 2021.

[15] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *Proc. International Conference on Learning Representations (ICLR)*, 2015.

[16] William Laurance. Habitat destruction: Death by a thousand cuts. *Conservation Biology for All*, 01 2010. doi:10.1093/acprof:oso/9780199554232.003.0005.

[17] Keila Lima, José Pinto, Vasco Ferreira, Bárbara Ferreira, André Diegues, Manuel Ribeiro, and João Borges Sousa. Comprehensive habitat mapping of a littoral marine park. In *Proc. OCEANS 2019*, pages 1–6, 2019. doi:10.1109/OCEANSE.2019.8867074.

[18] Tao Liu, Amr Abd-Elrahman, Morton Jon, and Victor Wilhelm. Comparing Fully Convolutional Networks, Random Forest, Support Vector Machine, and Patch-based Deep Convolutional Neural Networks for Object-based Wetland Mapping using Images from small Unmanned Aircraft System. *GIScience & Remote Sensing*, 55, 01 2018. doi:10.1080/15481603.2018.1426091.

[19] LSTS. GitHub - LSTS software toolchain. https://github.com/LSTS, . Last access: September 2021.

[20] LSTS. TTK22 Software tool chain for networked vehicle systems. https://zepinto.github.io/ttk22/, . Last access: September 2021.

[21] Xiaowen Luo, Xiaoming Qin, Ziyin Wu, Fanlin Yang, Mingwei Wang, and Jihong Shang. Sediment Classification of Small-Size Seabed Acoustic Images Using Convolutional Neural Networks. *IEEE Access*, 7:98331–98339, 2019. doi:10.1109/ACCESS.2019.2927366.

[22] Luís Madureira, Alexandre Sousa, José Braga, Pedro Calado, Paulo Dias, Ricardo Martins, José Pinto, and João Sousa. The light autonomous underwater vehicle: Evolutions and networking. In *Proc. MTS/IEEE OCEANS 2013*, pages 1–6, 2013. doi:10.1109/OCEANS-Bergen.2013.6608189.

[23] Ricardo Martins, Paulo Sousa Dias, Eduardo R. B. Marques, José Pinto, Joao B. Sousa, and Fernando L. Pereira. IMC: A communication protocol for networked vehicles and sensors. In *Proc. Oceans Europe*, 2009. doi:10.1109/OCEANSE.2009.5278245.

[24] Michael A. Nielsen. Neural networks and deep learning, 2018.

[25] Nvidia. Jetson Nano. https://developer.nvidia.com/embedded/jetson-nano. Last access: September 2021.

[26] Nvidia. Installing TensorFlow For Jetson Platform. https://docs.nvidia.com/deeplearning/frameworks/pdf/Install-TensorFlow-Jetson-Platform.pdf, 2021.

[27] P. Pandian, J.P. Ruscoe, Mark Shields, Jonathan Side, Rob Harris, Sandy Kerr, and C.R. Bullen. Seabed habitat mapping techniques: An overview of the performance of various systems. *Mediterranean Marine Science*, 10:29–44, 12 2009. doi:10.12681/mms.107.

[28] Miguel Pereira, Eduardo R. B. Marques, and José Pinto. Onboard Image Classification of Biological Habitats Using Underwater Vehicles. https://github.com/mquinaz/Onboard-Image-Classification. Last access: September 2021.

[29] José Pinto and João Borges de Sousa. Observatório Marinho de Esposende. Ação 1 - Cartografia de Habitats EUNIS. Relatório final do contrato de prestação de serviços nº 119/2017. Technical report, Instituto de Sistemas e Robótica - Porto, 2020.

[30] José Pinto, Paulo Dias, Ricardo Martins, João Fortuna, Eduardo R. B. Marques, and João Sousa. The LSTS Toolchain for Networked Vehicle Systems. In *Proc. MTS/IEEE Oceans 2013*, 2013. doi:10.1109/OCEANS-Bergen.2013.6608148.

[31] S N Prasanth. STM32CubeMX graphical tool that analyzes our tflite models w.r.t to size. https://rarelyknows.wordpress.com/2021/06/08/how-to-analyze-tflite-models-w-r-t-size/, 2021. Last access: September 2021.

[32] Hélio Puga. Velocity Estimation for Autonomous Underwater Vehicles using Vision-Based Systems. Master's thesis, Faculdade de Engenharia da Universidade do Porto, 07 2018.

[33] Frédéric Py, José Pinto, Mónica Silva, Tor Johansen, João Sousa, and Kanna Rajan. EUROPtus: A Mixed-initiative Controller for Multi-Vehicle Oceanographic Field Experiments. In *Proc. International Symposium on Experimental Robotics (ISER)*, pages 323–340. Springer, 2016. doi:10.13140/RG.2.2.24176.25603.

[34] Raspberry Pi Foundation. Raspberry Pi 4. https://www.raspberrypi.com/products/raspberry-pi-4-model-b/. Last access: September 2021.

[35] Kalle Remm. Case-based predictions for species and habitat mapping. *Ecological Modelling*, 177:259–281, 10 2004. doi:10.1016/j.ecolmodel.2004.03.004.

[36] GitHub TensorFlow repository. Is TF Lite optimized for nvidia gpu's and Intel CPUs? https://github.com/tensorflow/tensorflow/issues/34536, 2019.

[37] Lutz Roeder. Netron: Visualizer for neural network, deep learning, machine learning models. https://netron.app/. Last access: September 2021.

[38] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In *Proc. IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4510–4520. IEEE, 2019.

[39] Oystein Sture. Python bindings for Inter-Module Communication Protocol (IMC). https://github.com/oysstu/pyimc. Last access: September 2021.

[40] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V. Le. MnasNet: Platform-Aware Neural Architecture Search for Mobile. In *Proc. IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2815–2823, 2019. doi:10.1109/CVPR.2019.00293.

[41] TensorFlow. Tensorflow – an end-to-end open source machine learning platform. https://tensorflow.org, . Last access: September 2021.

[42] TensorFlow. Tensorflow lite – deploy machine learning models on mobile and iot devices. https://tensorflow.org/lite, . Last access: September 2021.

[43] Tensorflow. TensorFlow 2 quickstart for beginners. https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/quickstart/beginner.ipynb, 2019. Last access: September 2021.

[44] Tensorflow Hub. Feature vectors of images with MobileNet V2 trained on ImageNet (ILSVRC-2012-CLS). https://tfhub.dev/google/tf2-preview/mobilenet_v2/feature_vector/4. Last access: September 2021.

[45] Udacity. Udacity model for flower classification. https://colab.research.google.com/github/tensorflow/examples/blob/master/courses/udacity_intro_to_tensorflow_for_deep_learning/l05c04_exercise_flowers_with_data_augmentation_solution.ipynb. Last access: September 2021.