Efficient Deployment of Deep Learning Models on Autonomous Robots in the ROS Environment

M. G. Sarwar Murshed, James J. Carroll, Nazar Khan, and Faraz Hussain

Abstract Autonomous robots are often deployed in applications to continually monitor changing environments such as supermarket floors or inventory monitoring, patient monitoring, and autonomous driving. With the increasing use of deep learning techniques in robotics, a large number of robot manufacturing companies have started adopting deep learning techniques to improve the monitoring performance of autonomous robots. The Robot Operating System (ROS) is a widely used middleware platform for building autonomous robot applications. However, the deployment of deep learning models to autonomous robots using ROS remains an unexplored area of research. Most recent research has focused on using deep learning techniques to solve *specific problems* (e.g., shopping assistant robots, autopilot systems, automatic annotation of 3D maps for safe flight). However, integrating the data collection hardware (e.g., sensors) and deep learning models within ROS is difficult and expensive in terms of computational power, time, and energy (battery). To address these challenges, we have developed EasyDLROS, a novel framework for robust deployment of pre-trained deep learning models on robots. Our framework is open-source, independent of the underlying deep learning framework, and easy to deploy. To test the performance of EasyDLROS, we deployed seven pre-trained deep learning models for hazard detection on supermarkets floors in a simulated environment and evalu-

Electronic supplementary material The online version of this chapter (https://doi.org/10.1007/978-981-16-3357-7_9) contains supplementary material, which is available to authorized users.

M. G. Sarwar Murshed (✉) · J. J. Carroll · F. Hussain
Clarkson University, Potsdam, NY, USA
e-mail: murshem@clarkson.edu

J. J. Carroll
e-mail: jcarroll@clarkson.edu

F. Hussain
e-mail: fhussain@clarkson.edu

N. Khan
Department of Computer Science, University of the Punjab, Lahore, Pakistan
e-mail: nazarkhan@pucit.edu.pk

ated their performances. Experimental results show that our framework successfully deploys the deep learning models on ROS environment.

1 Introduction

The use of autonomous robots to perform tasks that until recently were performed solely by humans (e.g., rescuing victims from hazardous environments, inventory management, identification of hazards, etc.) has seen immense interest in recent years [24]. An important reason for the surge of research in this domain is the deployment of highly accurate deep learning models that can significantly enhance robot intelligence enabling them to perform human-like activities. In the retail industry, efforts are increasingly been made to reduce human involvement in hazard detection and inventory maintenance. Robots produce large amounts of data to observe the surrounding environment and most robotics technology involves using cloud servers for data processing [22]. The use of cloud servers involves transferring raw data to the cloud, which increases communication costs and response time, and also makes any private data vulnerable to compromise. It is therefore desirable for the data to be processed as close to the source as possible.

Edge computing techniques address this problem by processing data in close proximity to the data source [17]. With the improvement in edge computing technology, many edge devices now possess enough computational and storage capabilities to perform deep learning at the network edge. For example, the Coral Dev Board can perform 4 Trillion Operations Per Second (TOPS) operating at only 2 W. Therefore, edge computing can rapidly become a pervasive technology for deploying deep learning models at the network edge. Low-power devices such as the Coral Dev Board and the NVIDIA Jetson are among popular edge devices used for designing autonomous robots capable of performing real-time deep learning tasks. Devices such as NVIDIA Jetson, the BeagleBone AI, and the Coral Dev Board have been developed with the aim of providing intelligence to the robotic fields using edge technology.

Deep Learning (DL) has revolutionized the way image, audio, and sensor data are analyzed. Efforts to deploy DL techniques on cyber-physical agents for making real-time decisions have increased significantly [31]. However, deep learning techniques are computationally expensive due to their complex deep neural architectures involving millions of parameters. It has been observed in the literature that shallow neural architectures are also available; however, deep neural architecture-based learning models have achieved better accuracy, have automatic data engineering features, and can learn to solve complex problems, such as image and voice recognition, using much larger datasets [16]. Although they are successful in solving many problem accurately, deploying deep learning models on devices with limited memory and computational capabilities (such as IoT devices, autonomous robots, etc.) remains challenging. Different model compression techniques like model pruning and quantizing can help to deploy DL models on resource-constrained devices. In this chapter, we show how to deploy DL models on different resource-constrained devices in the ROS environment. Such resource-constrained devices are increasingly being used as the main on-board computational device on autonomous robots. In the experiments, a supermarket floor hazards dataset was used to train and test different deep learning models and to show successful deployment of deep learning models on resource-constrained settings within the ROS environment using our novel open-source EasyDLROS[1] framework.

In this chapter, we first provide an overview of how to deploy deep learning models on resource-constrained edge devices and then introduce our novel framework for deploying those models on autonomous robots within the ROS environment. We evaluate our approach using a real-world use-case of identifying hazards on floors using a ROS-integrated autonomous robot.

Research contribution

A lot of autonomous robots increasingly use the Robot Operating System (ROS), but there is no established framework for streamlining the deployment of DL within the ROS environment. ROS is middleware that has emerged as a universal vehicle for robotics application development [33]. It provides a publish/subscribe model for interprocess communication and multiple libraries and packages for application development, making it useful for a wide variety of practitioners. However, there remain three major issues in deploying deep learning models on robots using ROS:

- the high energy consumption by the deep learning model
- significant computational burden on computing devices

A major barrier to deploying deep learning models on edge devices or autonomous robots is the lack of efficient algorithmic systems that are robust enough to work with the limited computational resources and low battery life. This chapter describes a new deep learning architecture for training a deep learning model and deploying intelligence to the network edge within the ROS environment, allowing resource-constrained devices to aid faster decision-making. Six other existing deep learning models that can be deployed on constrained devices are also discussed in this chapter. Finally, we describe the EasyDLROS, a framework for deploying DL models on edge devices or autonomous robots operated by ROS. The main contribution of this chapter is as follows:

- the design of *EdgeLite*, a lightweight image recognition CNN architecture for detecting the presence or absence of supermarket floor hazards
- a novel framework, EasyDLROS, for deploying deep learning models on autonomous robots integrated within the ROS environment
- a comparison of EdgeLite with six state-of-the-art deep learning models (viz. MobileNetV1, MobileNetV2, InceptionNet V1, InceptionNet V2, ResNet V1, and GoogleNet) for supermarket hazard detection when deployed on the NVIDIA Jetson TX2 showing EdgeLite to have the highest F-1 score and comparable resource requirements in terms of memory, inference time, and energy, and
- a new dataset of images of floor hazards in supermarkets
- a case-study using a real-world example with a robot in a simulated environment.

[1] https://github.com/sarwarmurshed/supermarket_hazard_detection/tree/master/EasyDLROS.

The rest of the chapter is organized as follows: Sect. 2 presents related work on hazard detection using deep learning and edge computing, and deploying deep learning models on autonomous robots. Section 3 provides a brief introduction of relevant CNN architectures, their training, and inference. Section 4 describes the design and architecture of EdgeLite, a lightweight CNN architecture for hazard detection. The architecture of our novel EasyDLROS framework is discussed in Sect. 5. Data collection and generation methods are described in Sect. 6. Experiments, performance evaluation, and analysis of EdgeLite along with six other DL models, both with and without using EasyDLROS framework, are described in Sect. 8, which is followed by the conclusion.

2 Related Work

Despite the challenge posed by the computational requirements of deep learning, several researchers have explored ways of deploying state-of-the-art DL systems in resource-constrained settings. In most of the work so far in this area, researchers have used a large computing system or cloud servers for performing deep learning on a small device or ROS-powered autonomous robot. However, most autonomous robots get energy from an on-board battery. Running a compute-heavy deep learning model on a robot puts a tremendous burden on the battery. On the other hand, using cloud servers compromises data security and increases latency. Therefore, our work focuses on using small computation devices (such as the Jetson Nano) to locally perform all deep learning inference and other tasks.

Although Hsu et al. used a large cloud server, they have developed a fall hazard detection system that generates an alert message when an object falls [9]. Their approach has three key aspects:

1. a skeleton extraction performed for building an ML prediction model to detect falls,
2. a Raspberry Pi used as an edge computing device for primary data processing and to reduce the size of videos/images,
3. and finally, falls are detected using machine learning inference on the cloud, and users are notified in appropriate cases.

In another study, a human fall detection system was developed using a convolutional neural network [32]. This fall detection system can detect human falls in video sequences.

A hierarchical distributed computing architecture has been designed for a smart city to analyze big data at the edge of the network to detect hazardous events in a city area [30]. A working prototype was constructed using a machine learning algorithm on low-power computing devices to quickly detect hazardous events to avoid potential damage. Researchers have also used deep learning models to detect small-sized hazards on roads (e.g., lost cargo) which is a vital capability for autonomous vehicles [19].

Lane et al. developed a software accelerator capable of lowering the computational resources required by deep learning and a software accelerator capable of lowering the device resources required by deep learning [14]. They used two techniques, viz, run-time layer compression and deep architecture decomposition which help to control the memory and computation run-time during the inference phase.

A service robot that assists a customer in shopping has been developed by Su et al. [27]. ROS and deep learning techniques were used in this system to detect a customer and answer some simple questions. In another study, the combination of ROS and deep learning were used to annotate maps with regions of crowded and non-crowded scenes [13]. The identification of non-crowded regions helps a drone to find paths to fly safely. However, the performance of DL models on a robot was not discussed in these studies.

Sisido et al. designed a traffic signs recognition system using deep learning and tested the system on a ROS simulator [26]. Chang et al. developed an object detection system using the Faster R-CNN algorithm and ROS [2]. They deployed the full system on a Raspberry Pi-based robot and used the Kinect sensor to capture the images. Cloud services were used in the system to detect objects in real-time. Liu et al. developed an autopilot system using deep learning and ROS distributed architecture [15]. They used three cameras to capture images of the roads and used deep learning to detect road signs, lane lines, and signal lights on those images. Then they passed detection results to the decision module and process those results. Finally, they used a control module to control the car automatically. They used 3 cameras, 3 Jeston TX1 boards, and a big industrial computer to detect objects and drive the car.

Most research on DL within the ROS environment has used public cloud computing or external computing systems to perform data processing tasks. In our experiments, we only use onboard devices for data processing, allowing us to keep all data private.

3 Deep Learning Background

This section presents a brief background discussion on the key concepts in the deployment of deep learning models on resource-constrained devices.

3.1 CNNs

The majority of modern deep learning architectures are based on *Artificial Neural Networks (ANNs)*. Artificial neural networks combined with a set of mathematical or other operations, known as convolutions, are called *Convolutional Neural Networks (CNNs)*. CNNs are one of the most widely used types of ANNs which can take images as input, learn different features of those images and classify them into categories accurately [1]. Techniques based on CNNs have made tremendous progress not only on image recognition but also on speech recognition, natural language processing making it one of the most popular method in deep learning. CNN-based deep learning techniques mimic human intelligence using ANNs and progressively learn to solve problems. CNNs consist of millions of artificial neurons and connections among those neurons. The connections between neurons carry weights. The number of neurons in deep learning networks has a great impact on the performance of the network. A CNN performs better if the depth of layers increases [3] and the training dataset is rich. However, increasing the depth of the network increases the number of the parameters in the network. The number of parameters has a great impact on computational resources and energy usage. This high computational requirement inhibits deep learning deployment on battery-powered robots. However, several models have recently been developed that maintain an excellent balance between the number of parameters and performance as well as limiting the use of computation resources [17]. MobileNet and SqueezeNet are examples of such lightweight models which use fewer parameters compared to other well-known models, such as AlexNet, but perform well in resource-constrained settings [5, 11].

3.2 Training

Figure 1 shows an overview of CNN training and inference. During training, a CNN network learns the value of its parameters, such as filters, weights, etc., from previous data based on specific task of interest. Training of a CNN is performed in two phases:

1. Forward phase: input data is fed to the CNN and passed completely through the network. The CNN makes a prediction based on the input data and calculates the error in the prediction using “loss function”.
2. Backward phase: the errors are received and weights (connections between the artificial neurons) updated based on the error values.

After training, the weights, biases, and other parameters are stored as part of a trained deep learning model. This trained model is used for the prediction and/or classification on new data during inference.

3.3 Inference

Deep learning inference is a process where a model is used to apply knowledge gained during training to make a prediction and/or classification on previously unseen data. For example, when a new image is shown to a trained model as input, the model outputs a prediction score based on the value of its weights which were learned in the training phase. Generally, a model is trained using 32-bit floating point precision and uses the same precision for inference. However, a model deployed on a battery-powered robot that is based on 32-bit operations drains too much energy for

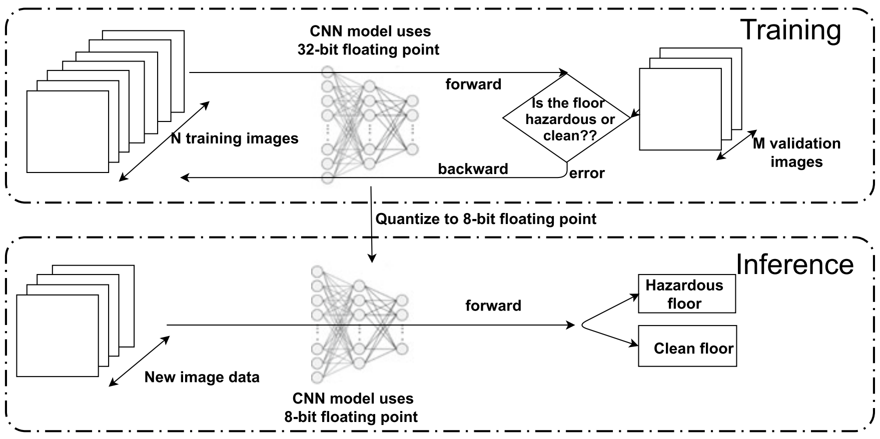

Fig. 1 CNN training and inference for resource-constrained setting. In the forward pass, input data is fed to the network. Each layer of the network uses an activation function to process the input data and calculates an error between the current output vector of the network and the expected output vector. In the backward pass, the network tries to minimize the error by repeatedly adjusting its weights and biases. 32-bit floating point format is used in this forward and backward pass. Once trained, the model is quantized to 8-bit. During inference, the quantized model is used to save computational resources and energy to enable ease deployment on resource-constrained devices

a power-scare scenario. Therefore, DL models are often quantized to 8-bits before being deployed on a robot. This conversion reduces the accuracy of the model a little bit, but sacrificing this accuracy saves energy, making the techniques feasible for resource-constrained settings. Figure 1 shows the quantization and inference of a DL model.

4 Deploying DL Models on Resource-Constrained Devices

Fitting deep learning models on edge devices is challenging due to their limited memory and computational capabilities. For example, an edge device may not have enough memory to store parameters, the weight values of CNN filters, or the input data arrays, additionally, the battery power may be insufficient for real-time model inference. Therefore, there is increasing interest in lightweight, compute-efficient CNNs with some acceptable loss in accuracy.

A typical CNN performs millions of mathematical operations to generate results, and therefore has significant energy requirements. The dominant numerical format used for CNN training and inference is 32-bit floating points. Performing mathematical operations with that precision can be very resource hungry and time-consuming. Lower-precision numerical formats can address such issues without incurring significant accuracy loss.

Among many state-of-the-art model compression techniques, transferred/compact convolutional filters, and parameter quantization are two widely adopted approaches used to fit CNNs on edge devices [4, 6, 17, 36]. The first is reducing the number of mathematical operations required for a model to improve inference time with a minimal loss in accuracy. Examples of this category are MobileNet [5], SqueezeNet [11], EfficientNet [29], and ShuffleNet [35]. Another approach is to quantize the model weights from a higher bit floating point (e.g., 32 bit) into lower bit-depth representations (e.g., 8 bit). This technique is exemplified by Binary Neural Networks (BNNs) [10] and XNOR-Net [20].

The approach we take is to train the deep learning models on a desktop machine using 32-bit floating point precision, and then quantize the model to 8-bit, before deployment on a robot using ROS. The inference is performed solely on a robot. Quantization helps not only reduce the use of computational resources but also reduces energy consumption and latency of generating results.

To test our technique, we applied it to the problem of automatically detecting of hazards on supermarket floors. We tested several pre-trained models including InceptionV1 & InceptionV2, MobileNetV1 & MobileNetV2, and ResNet, used transfer-learning technique to train them on our supermarket hazard dataset, and then developed a new architecture *EdgeLite* that outperforms other models for hazard detection on resource-constrained devices. EdgeLite, a simplified CNN model that requires fewer mathematical operations than InceptionNet was quantized to run on resource-constrained edge devices. We experimented with a varying number of layers in order to create a model that can run on resource-constrained devices (such as the Coral Dev Board and the Raspberry Pi) using as little memory as possible while retaining high accuracy.

4.1 EdgeLite Architecture

In order to have a model that can be used for inference on resource-constrained edge devices, with a low-memory footprint without any additional hardware, we developed a lightweight CNN architecture which achieved more than 90% accuracy on our hazard dataset.

EdgeLite has 19 layers, not counting the pooling layers. We used filters with multiple sizes which operate on the same level of the CNN network to extract features at different scales. The different types of filters used were of size 1×1, 3×3, and 5×5. To make the network computationally cheaper, 1×1 convolutions were used to reduce the input channel depth and an extra 1×1 convolution was used before the 3×3 and 5×5 convolutions.

Our CNN architecture, shown in Table 1, consists of convolution, max-pooling, avg-pooling, and EdgeLite layers. EdgeLite layers are incorporated into CNNs as a way of reducing computational expense through a dimensionality reduction with stacked 1×1 convolutions. Multiple kernel filter sizes are used in this layer and an extra 1×1 convolution is added whenever 3×3 and 5×5 layers are used. All

Table 1 Outline of the architecture of EdgeLite

Type	Patch size/stride	Output size
Conv	$7 \times 7/2$	$112 \times 112 \times 64$
Max pool	$3 \times 3/2$	$56 \times 56 \times 64$
Conv	$3 \times 3/1$	$56 \times 56 \times 192$
Conv	$3 \times 3/1$	$56 \times 56 \times 256$
Conv	$3 \times 3/1$	$56 \times 56 \times 480$
Pool	$3 \times 3/2$	$14 \times 14 \times 480$
5×EdgeLite_conv		$14 \times 14 \times 832$
Pool	$3 \times 3/2$	$7 \times 7 \times 832$
2×EdgeLite_conv		$7 \times 7 \times 1024$
Pool	$7 \times 7/1$	$1 \times 1 \times 1024$
Dropout		$1 \times 1 \times 1024$
Linear		$1 \times 1 \times 1000$
Softmax	Classifier	$1 \times 1 \times 2$

the kernels are ordered to operate on the same level sequentially. A max-pooling is performed in this layer and the resulting outputs are concatenated, and then sent to the next layer. EdgeLite's architecture is inspired by Inception [28]. However, we reduced the number of layers and the size of the kernels to make it suitable for resource-constrained devices.

5 Architecture of EasyDLROS

This section presents the relevant background on the Robot Operating System, the structure of our novel EasyDLROS framework, the basic components of ROS, and how those components are used in EasyDLROS.

5.1 ROS

The Robot Operating System (ROS [21]) is a meta-operating system, based on middleware, which provides a flexible framework for developing robot software. ROS provides a set of software libraries and tools that help to build robot applications on a heterogeneous computer cluster. Distributed computing, portability, and ease of testing are among the main features of ROS. These features help to implement multi-machine communication, real-time operation, and distributed computing. The software in ROS is organized in packages that help implement applications such as perception, simultaneous localization, robot models and mapping, simulation tools, and other algorithms.

A **ROS node** is a process that performs computations necessary for completing a task. A node is an executable program that resides in a ROS application. ROS allows multiple nodes in one application. A package consists of multiple nodes that communicate with each other using ROS topics, services, actions, etc.

A **ROS topic** is a data stream that helps to exchange information between nodes. Topics implement a publish/subscribe communication mechanism for exchanging data in a ROS system.

ROS has various types of drivers, libraries, and protocols for handling different types of devices attached to a robot. These components provide support for computer vision and make it possible to deploy deep learning models on a ROS-based autonomous robot to perform tasks like object recognition, voice recognition, and so on. Many of the current deep learning models take images as input. An image pipeline of the ROS framework helps with capturing images, color decoding, and other low-level operations to make an image suitable for a deep learning model.

ROS supports reusing existing drivers and code written for other robots and platforms. These features greatly help developers not only achieve a specific problem-based task but also improve the ROS community. The next section describes the node and topic design for the application layer of our novel framework.

5.2 *EasyDLROS Framework*

An overview of the architecture of our novel deep learning system is shown in Fig. 2. It shows how EasyDLROS helps to deploy deep learning models on a robot and performs all analytic tasks locally (without the use of cloud servers). The system is divided into three main parts:

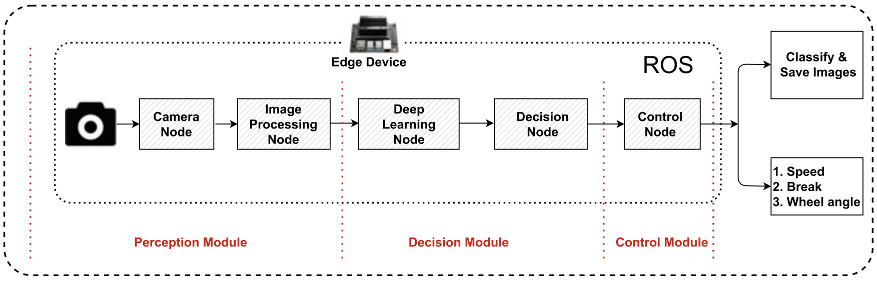

Fig. 2 The architecture of our novel EasyDLROS framework. EasyDLROS is a ROS-based on-device deep learning deployment system. First, the camera node captures raw image data using a camera mounted on a robot. Then, the image processing node processes the raw image data and makes it suitable for a deep learning model. The deep learning node deploys a pre-trained DL model and generates image classification results. The decision node then analyzes the classification results and generates the necessary commands to perform robot's subsequent tasks. Finally, the control node receives commands from the decision node and performs the necessary user-defined tasks in addition to controlling the robot

1. **Perception module**: This module includes a camera node, which is responsible for taking all necessary input images. These images are then processed by an image processing node. The main functionality of this node includes image conversion (ROS msgs/Image message to OpenCV image), brightness adjustment, resizing the pixels, etc.
2. **Decision module**: This module is the brain of our system. All images from the previous module are processed in this module using DL model. It fuses the results to the other nodes.
3. **Control module**: This module contains logics to control the robot. It gets input from the decision module and uses control logic to make a control decision. Then it generates control commands, such as for modifying speed, front wheel angles, verifying hazardous or clean floor, brake, etc. Steer and speed nodes that generate predictive robot lateral and longitudinal control commands including vehicle speed and front-wheel rotation angle are deployed in the module.

ROS standard nodes are used to achieve the functionality of these modules. In our prototype, we used one camera and one Jetson TX2 as the edge device.

Camera node

The camera node is part of the perception module (see Fig. 2) and is used to handle the camera attached to the robot. All input images are taken using this node which makes this one of the most important nodes. In our experimental setup, an 8.08MP wide-angle Leopard imaging camera, with a 136° field of view, was mounted in front of the robot to capture images. Other types of cameras can also be used on this node. Figure 3 shows the overall structure of the camera node.

ROS uses a *camera_info_manager* package for saving and restoring the camera calibration data. Again, the *Camera_info_manager* uses a *camera_package* and OpenCV to save and restore msgs/CameraInfo data. After capturing an image, this camera node publishes two outputs: an *image_raw* message and a *camera_info* message. The *image_raw* message contains all the necessary unprocessed data required for generating a digital image and the *camera_info* message contains additional information like the location of the calibrated camera, height, and width of the captured

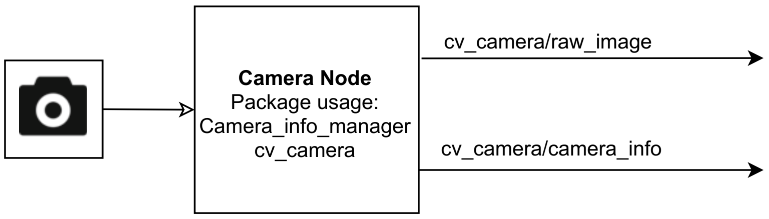

Fig. 3 The block diagram of the camera node. This node captures raw image data using robot camera and feeds it to the image processing node

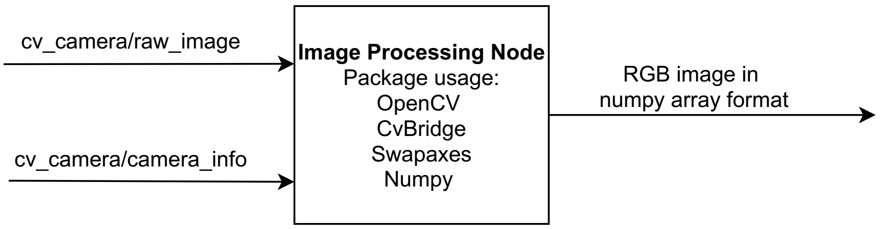

Fig. 4 Block diagram of the image processing node

image. However, if *camera_info_manager* fails to find camera devices then it starts to save and publish dummy image data, which is a common cause of hard-to-detect bugs. To handle these issues, the camera should be calibrated properly to make sure that the *camera_info_manager* only stores real images using *camera_info* message.

Our camera node is camera-device independent and can capture images with any type of camera attached to the robot. In our experiments, we have used 224 × 224 images in our deep learning model. Therefore, we keep the resolution of the camera to 224 × 224. This low resolution helps to increase image capturing and image processing speed.

Image processing node

A ROS camera node captures a raw image in a special *sensor_msgs/Image* message format. Most DL models use images in a matrix format as input, hence *sensor_msgs/Image* format data cannot be used directly as input to any DL model. Therefore, we used the OpenCV *CvBridge* library to convert these raw images. This library converts *sensor_msgs/Image* message format files into OpenCV images making them readable by DL models. Figure 4 shows the overall structure of this node and the code snippet to convert the images is shown in Listing 9.1.

CvBridge supports different types of encodings to convert ROS raw images to OpenCV images. Mono8, bgra8, rgb8 are the most popular image encodings. The deep learning models used in our experiments accept RGB images as input, so we converted all ROS raw images messages to RGB images using rgb8 encoding. An overview of the functionality of a CvBridge is shown in Fig. 5.

Listing 9.1 Image data received from a robot camera is converted to a rgb image

```
from cv_bridge import CvBridge
bridge = CvBridge()
cv_image = bridge.imgmsg_to_cv2(image_message, ‘‘rgb8’’)
```

Although we have converted the ROS image message to an RGB image, further image processing tasks such as swapaxes, ndarrary conversion, etc. are required before it can be used in a DL model. In our experiments on floor hazard detection,

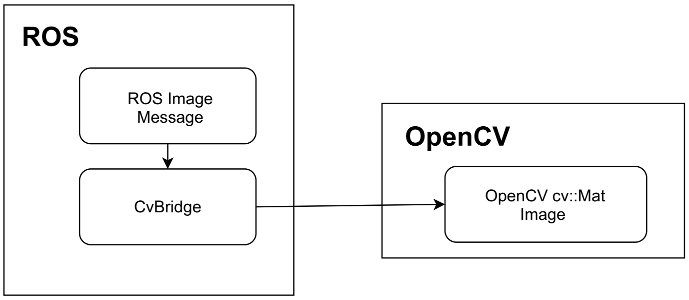

Fig. 5 A ROS camera node captures images in the ROS *sensor_msgs/Image* message format. This image format is not suitable for conventional DL models. This figure shows a block diagram of image conversion from ROS image format to OpenCV cv::Mat format using CVBridge. OpenCV cv::Mat can directly be used as input to a DL model. CVBridge provides different types of OpenCV image encodings, such as, mono8, bgr8, rgb8, rgba8, to support gray-scale and color images

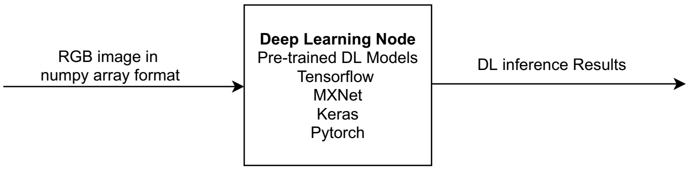

Fig. 6 This figure shows the deployment of a pre-trained DL model in the ROS environment using a ROS node. This node receives processed RGB image data as input and feeds that data to the pre-trained DL model to generate output results. Finally, it publishes the output results. The decision node receives this result and makes a decision (e.g., increase or reduce the speed of the robot) based on this result

all processing in this node is done using the CvBridge package in order to publish data in a readable format for the DL models. The deep learning node receives this data and performs classification.

Deep learning node

This node contains a pre-trained DL model as described earlier (Sect. 3.2), in the ROS environment.

Figure 6 shows the structure of the deep learning node. For our experiment, we used the DL models trained by the MXNet framework. However, models trained by other popular frameworks like TensorFlow, Keras, Pytorch can also be used in this node.

All packages required to use the MXnet model were installed on the robot's onboard computer [12]. To use the MXnet DL models on a robot, first, the pre-trained model was loaded into the MXNet module and then assigned the corresponding parameters such as input size, argument types, etc. Then, we completed the model load and parameter binding (Listing 9.2).

Listing 9.2 Load a deep learning model and bind parameters using MXNet

```
import mxnet as mx
name ='mobilenetv1−1.0'
sym,␣argp,␣auxp␣=␣mx.model.load_checkpoint(name,␣100)
mod␣=␣mx.mod.Module(symbol=sym,␣context=mx.cpu())
mod.bind(for_training=False,␣data_shapes=␣[('data', (1,3,224,224))])
mod.set_params(argp, auxp)
```

When all the binding is done, we feed the input image data received from the image processing node to the DL model. The model uses this image data to find out if there are any hazards in the input image or not. Finally, the node publishes a clean or a hazard report as an output result.

Decision node

This node first analyzes the output received from the previous node. If the decision node received "hazard" as input, it first tries to measure the area of the hazard so that the robot can avoid that hazard. In the end, this node provides some decisions to control nodes. Based on these decisions, the control node can control the robot. Another major function of this node is to publish an alert topic if it finds a hazard in an image.

Control node

The control node can be used to modify the speed, brake, and wheel angle of the robot according to the input received from the decision node.

EasyDLROS uses lightweight CNN models during training and inference, so that the framework drains less energy from the robot battery. The complete CNN training and deployment processes using EasyDLROS are shown in Fig. 7. To show the effectiveness of our framework, we deployed seven pre-trained models on a robot and evaluated the performance, the details of which are described in detail later (Sect. 8).

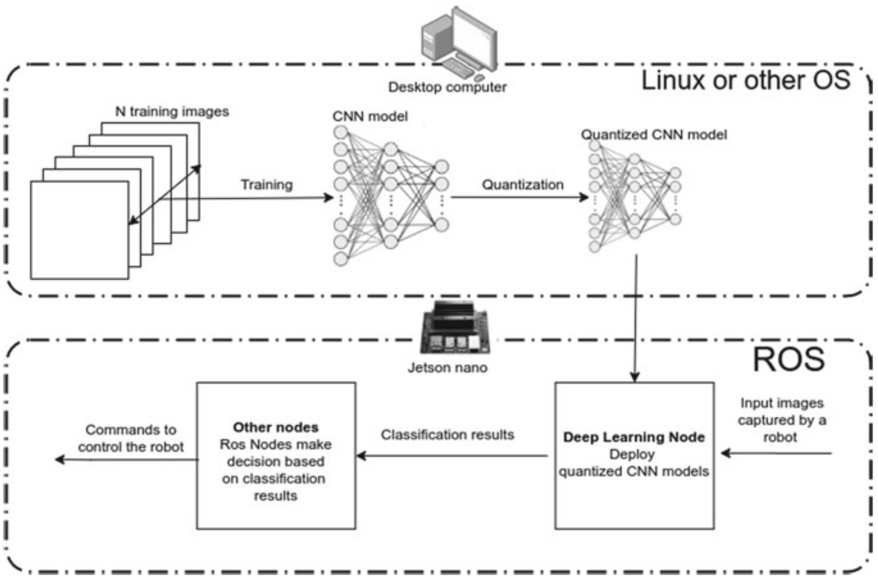

Fig. 7 This figure depicts an overview of our EasyDLROS framework. First, deep learning models are trained on a desktop computer using frameworks such as TensorFlow and MXNet. Then, the models are compressed which helps to reduce compute requirements and facilitate easy integration with ROS. Finally, the models are deployed on the robot and the output results obtained from the DL models can be used to control the robot and perform other tasks

6 Dataset

Since we know of no publicly available dataset for supermarket hazards, we built a new dataset of images showing hazards on supermarket floors. We manually collected 1180 images of clean and hazardous floors and used data augmentation and synthesis [7] to generate new images to enrich our dataset.

We generated an additional 300 images using the data synthesis technique designed by Dorr et al. [7]. First, we collected images of common grocery items (e.g., bakery, bread, broken eggs, sauces, and liquid spills). These images were cropped and resized and placed on clean floor images. In other words, images of clean floors were used as the background layer and cropped hazards were layered on them.

We also used data augmentation methods, including horizontal flip, shift, zoom, and brightness change, to generate an additional 5020 images. After data augmentation, we had 6500 images, where 3250 images for hazardous floors and 3250 for clean floors. We used 5500 images (2750 images for hazardous floors and 2750 images for clean floors) for training & validation and 1000 images (500 images for each category) for testing.

Table 2 The distribution of images in our supermarket floor hazards dataset. Out of a total of 6500 images used in our experiments, 1180 were collected manually while the remaining were generated using data augmentation and synthesis techniques

Class	Training	Validation	Testing
Hazardous floor	2224	526	500
Clean floor	2224	526	500

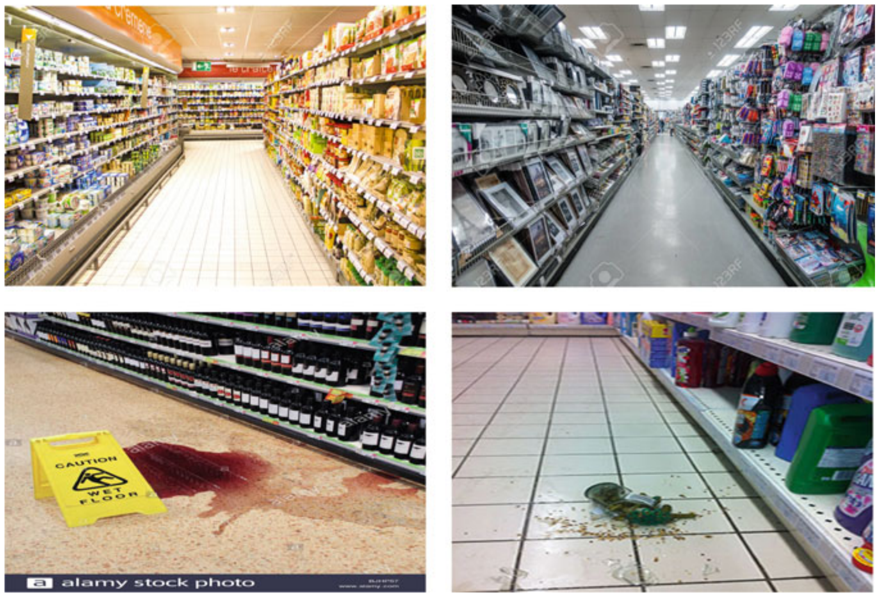

Fig. 8 Supermarket images showing clean and hazardous floors

Table 2 shows how the images were split among the training, validation, and testing sets. Figure 8 shows sample images from our dataset of clean and hazardous floors in supermarkets. Synthetically generated images are shown in Fig. 9.

7 How to Use the Code

The seven deep learning architectures listed in Table 3 can be built, trained, and tested with the code provided in our GitHub.[2] There are multiple README files in the repository with details instructions about using the models. To run the code, the following dependencies need to be met:

[2] https://github.com/sarwarmurshed/supermarket_hazard_detection.

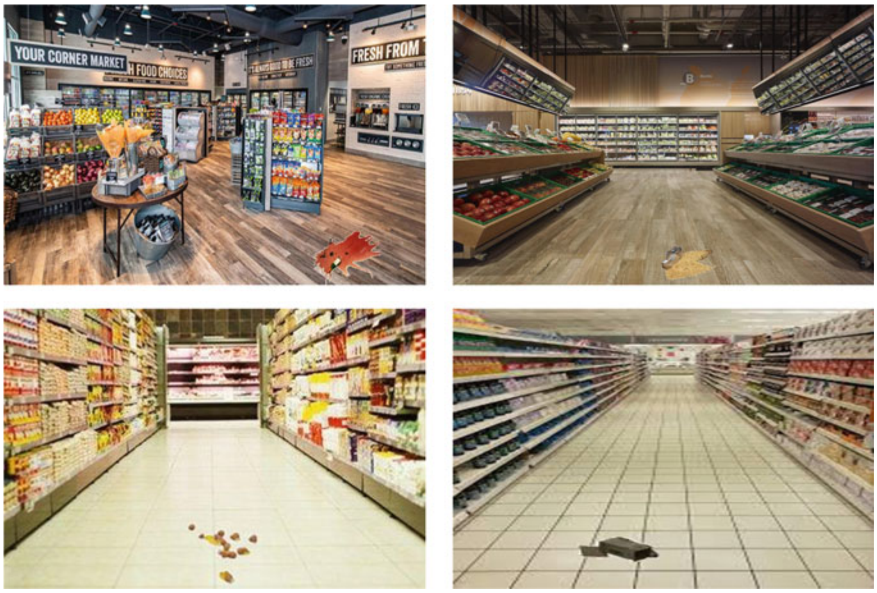

Fig. 9 Images showing hazardous floors in supermarkets that were synthetically generated using the technique of Dorr et al. [7]

Table 3 The size and the number of parameters of the deep learning models used in our experiments

Model	InceptionV1 INV1	InceptionV2 INV2	ResNet RNV1	GoogleNet GN	MobileNetV1 MNV1	MobileNetV2 MNV2	EdgeLite EL
Size (MB)	6.80	10.20	5.87	5.05	3.30	2.30	4.90
Parameters (millions)	5.98	10.17	11.18	6.02	3.21	3.51	5.26

- Install MXNet, TensorFlow
- Install any ROS distribution (our code was tested on ros-melodic)
- Jetpack > 4.2.2, jetson-inference, `ros_deep_learning`[3] (for any NVIDIA Jetson device)

[3] https://github.com/dusty-nv/ros_deep_learning.

Training DL models

The *EdgeLite*[4] and *fine-tune_existing_models*[5] directory contains code in Jupyter Notebook format for building, training, and testing all the models that we used in our experiments. To train a model, the training dataset needs to be store in *data*[6] directory and then execute the Jupyter Notebook.[7] After training, a trained model (*.params, .json* files) will be saved to the *model* directory. This trained model will be used in the EasyDLROS framework for image classification.

Deploy model on ROS environment

Any model trained on MXNet or TensorFlow can be run using EasyDLROS framework. Models trained on a different DL framework, such as Pytorch, needs to be converted to TensorFlow-compatible `.pb` format in order to run using our framework. The following is the sequence of steps to run a model and classify images with EasyDLROS:

- start `roscore`
- store the trained model and label files (`.pd`, `synset_label_file.txt` for TensorFlow, `.params`, `json` and `synset_label_file.txt` file MXNet) in a directory.
- keep `live_image_recognition.py` file in the same directory
- run `'rosrun cv_camera cv_camera_node'` to capture images using the camera mounted in a autonomous robot or in a computer
- `'python live_image_recognition.py image:=/cv_camera/image_raw'` should be run to classify images captured in the previous step
- `'rostopic echo /result'` can be used to display classification results.

8 Experiments

Our experiments are divided into two main phases. The first phase included training and evaluating DL models in a plain UNIX-based OS without ROS (Sect. 8.1). In this phase, we used NumPy, OpenCV, etc., to process data and control data flow as

[4] https://github.com/sarwarmurshed/supermarket_hazard_detection/tree/master/edgeLite.

[5] https://github.com/sarwarmurshed/supermarket_hazard_detection/tree/master/fine-tune_existing_models.

[6] https://github.com/sarwarmurshed/supermarket_hazard_detection/tree/master/fine-tune_existing_models/fine-tune_googlenet/data.

[7] https://github.com/sarwarmurshed/supermarket_hazard_detection/blob/master/fine-tune_existing_models/fine-tune_googlenet/fine_tune_with_test.ipynb.

well as to complete image classification tasks. We trained several DL models and evaluated them, by deploying and testing, in this test phase. In the second phase (Sect. 8.2), we used ROS packages or services such as Camera_info_manager, cv_camera, ROS nodes, ROS topics, etc., to process data and control the data flow. We deployed the same models on a robot using the EasyDLROS framework and evaluated model performance in a ROS environment. The robot was built with an NVIDIA TX2 device and driven by ROS.

We conducted both sets of experiments in resource-constrained settings. Our experimental results show that image classification of our dataset for the hazard detection model is possible within less than a second and with less than 3.5 Joules of energy usage on a resource-constrained robot.

8.1 Experiments on Resource-Constrained Linux Environment Without ROS

We trained six widely used CNN-based image classification models on our supermarket hazard dataset using MXNet and then compared their performance with EdgeLite (Sect. 4.1) in terms of model accuracy, execution time, memory usage, and power consumption during inference time. The six CNN models used were MobileNetV1, MobileNetV2, InceptionNet V1, InceptionNet V2, ResNet V1, and GoogleNet. A Raspberry Pi, an NVIDIA Jetson TX2, and a Coral Dev Board were used as the edge devices. When deployed on these three edge devices, EdgeLite outperformed all other models in detecting hazards in supermarket floor images while maintaining comparable performance in other metrics, i.e., energy utilization, inference time, and memory usage.

8.1.1 Hyperparameter Tuning

Hyperparameter tuning is arguably the most important factor for improving performance of CNN models. We tuned multiple hyperparameters for EdgeLite as well as the other architectures used in our experiments. We paid special attention to the momentum, learning rate, weight decay coefficients, dropout rates, and corruption bounds for various data augmentations: random scaling, input pixel dropout, and random horizontal reflections. We optimized these over a validation set of slightly more than 1,000 examples drawn from the training set. We used a grid search and varied the values of these hyperparameters and ran each network for 300 epochs on the hazard dataset. Due to the small size of our training set, we conducted extensive tuning experiments (batch size: from 8 to 128, learning rate: from 0.0005 to 0.1, momentum: from 0.0 to 0.9, decay: 0.00001 to 0.0001) and evaluated the model with the best-performing hyperparameter configuration on the test set.

All of the aforementioned hyperparameters were also tuned for other state-of-the-art DL models used in our experiment. We followed the same procedure to fine-tune all models and reported the best results of models. TensorFlow and MXNet were used for composing, training, and evaluating the models.

8.1.2 Experimental Setup for the Linux Environment Without ROS

We conducted our tests on three edge devices as well as a desktop machine, which was used solely to train the models. The experiments involved three stages:

1. training the DL models on a desktop machine,
2. compressing the model to facilitate deployment on resource-constrained devices, and
3. performing inference after deploying the model on these edge devices.

The desktop machine used for training had a 20 core Intel Xeon processor (3 GHz) and 64 GB RAM. TensorFlow and MXNet were used to build and compressed the networks. We tested using the model that performs best on the validation set. We got the best EdgeLite model by training the networks using the Adam optimizer with a momentum of 0.9, and a batch size of 32, the learning rate was 0.002, and decay of 0.00004 on the model weights. We also investigated and selected the best hyperparameters for other models used in our experiments. After training with the appropriate hyperparameters, we evaluated the performance of the models on the testing dataset.

The trained neural network was exported to Open Neural Network Exchange (ONNX) format, converted to a TensorFlow Lite flatbuffer file, and finally converted to a TensorFlow Lite model for reducing storage and inference time. This conversion reduced the model size by more than 65% compared to the original model.

To evaluate the power consumption during inference, we used NVIDIA's power measurement software [18] for the Jetson TX2 and the X-DRAGON Digital USB meter [34] for the Coral Dev Board and Raspberry Pi. NVIDIA's power measurement software can capture the power consumption of six of the primary supply rails (CPU, RAM, GPU, etc.). The X-Dragon meter can measure power consumption directly by connecting to the device's power supply.

We monitored and measured the power consumption for all edge devices before and during inference. The maximum power (P_{max}) recorded at any time during the entire period of inference (T) is used to compute an upper bound on the energy required for inference ($E_{max} = P_{max} \times T$).

8.1.3 Results for the Linux Environment Without ROS

To analyze the performance of EdgeLite and other state-of-the-art CNN models on edge devices, we measured the accuracy, the inference time, and the amount of memory used to classify an image (Table 4). We also calculated the average energy

Table 4 Comparison between the performance of existing deep learning models with EdgeLite based on experiments on the Coral Dev Board (CB), Raspberry Pi (RP) and NVIDIA Jestson TX2

Model	Inference time (sec)			Avg RAM usage (MB)			Avg. energy usage (J/img)			Accuracy(%)		
	CB	RP	TX2	CB	RP	TX2	CB	RP	TX2	CB	RP	TX2
InceptionV1	0.56	0.71	0.36	7.00	3.26	22.00	1.04	1.78	1.24	86.31	86.41	89.05
InceptionV2	0.79	0.73	0.50	9.00	3.65	22.66	1.44	1.51	1.70	85.11	85.13	83.94
ResNet	0.58	0.75	0.46	8.50	5.87	24.45	1.07	1.55	1.27	84.70	84.67	84.67
GoogleNet	0.55	0.71	0.39	8.00	7.19	20.52	0.95	1.37	1.13	88.10	88.10	91.97
MobileNetV1	0.29	0.69	0.18	6.00	2.89	20.00	0.51	0.70	0.70	87.87	86.80	87.99
MobileNetV2	0.27	0.68	0.16	5.00	2.74	14.60	0.48	0.68	0.6	89.02	88.95	91.24
EdgeLite	0.51	0.69	0.38	5.20	3.10	19.87	0.86	1.29	1.03	92.37	91.98	92.17

consumed by the models during inference. The energy consumption is computed by summing the energy consumed during the whole inference period and then dividing by the number of images.

Accuracy. EdgeLite outperformed all other models in terms of accuracy (Table 4). We achieved 92.37% accuracy on the Coral Dev board when using our supermarket hazards dataset.

Inference time. In Table 4, the fastest model is the MobileNetV2, which only took 0.16 sec, 0.268 sec and 0.68 sec for inference on the Jetson TX2, Coral Dev Board and Raspberry Pi, respectively. EdgeLite took 0.38 sec for inference on the Jetson TX2, 0.506 sec on the Coral Dev, and 0.69 sec on the Raspberry Pi.

Compared to MobileNet (both V1 and V2), EdgeLite took more time and memory for classification because the latter has more filters than MobileNet and the number of convolution filters in each layer of a CNN has a significant effect on inference time and memory usage.

Memory usage. In terms of memory usage, the Raspberry Pi outperformed all other devices as shown in Table 4. When we measured the maximum memory usage at any point during the whole inference process, we observed that all models took less memory for inference on the Pi. MobileNet V2 only took 2.74 MB and MobileNet V1 took 2.89 MB while EdgeLite took 3.1 MB during inference. One factor that has a large impact on inference time and memory usage is the number of pixels of input images. Table 4 illustrates memory usage during inference while using 224 $\times$ 224 images. MobileNet V2 used only 5 MB memory for classification on the Coral Dev Board while EdgeLite used slightly more RAM at 5.2 MB.

Energy consumption. Most real-life scenarios where resource-limited devices are deployed are usually battery-powered hence the energy available to the devices is limited [25]. However, DL models consume a lot of energy during inference due to the high levels of computation required to generate output. This inhibits the deployment of DL models on small battery-powered devices. To see how the quantized DL models performed on a resource-constrained device, we measured the actual power consumption of pruned models on the edge devices. Table 4 shows the energy consumed during the classification of supermarket floor images (calculated as described at the end of Sect. 8.1.2) on resource-constrained devices. MobileNet V2 was the most efficient on our dataset, needing, on average, 0.48 J, 0.6 J, and 0.68 J for classification on the Coral Dev Board, Jetson TX2, and Raspberry Pi respectively.

8.1.4 Analysis

Impact of device speed & memory on inference time. Model quantization allows deep learning models to be deployed on resource-scarce devices by reducing memory footprint and speeding up inference. The inference time of a model is significantly dependent on the computing power of a device. For example, the computational power of the Raspberry Pi is 1.2 GHz and available memory is 1 GB. With this computational power, MobileNet V2 used a maximum of 2.74 MB during inference on that device and the average inference time for an image was 0.68 s. On the other hand, the Coral Dev Board operates at 1.5 GHz with 32 GFLOPs 32-bit GPU and 1 GB RAM. MobileNet V2 used a maximum of 5 MB on this device and the average inference time was 0.27 s, which is less than half that of the Raspberry Pi. The Jetson TX2 has two 2-GHz CPUs and a total of 8 GB memory which helps this device to classify an image in 0.16 s. MobileNet V2 used a maximum of 14.60 MB memory during inference. The inference time of the TX2 is four times less than that of the Pi and less than half of the Dev Board. In summary, the inference time of a model significantly depends on the CPU power and available memory of a device.

Impact of model size on accuracy. More parameters do not guarantee high accuracy, and there is no linear relationship between model size, complexity, and accuracy. MobileNet V2 and EdgeLite achieved higher accuracy than other models on all devices even though they have fewer parameters than comparatively low number of parameters (Table 3). This finding is independent of the device architecture.

Impact of the device type and model architecture on energy consumption. We observed that energy consumption depends not only on the DL model architecture but also on design of the computational device on which it is executed. The use of devices specifically designed for DL inference, such as the Coral Dev Board, BeagleBone-AI, NVIDIA Jetson, is now increasingly common for deploying DL models in order to achieve high accuracy in real-time. In our experiments, the Coral Dev Board, being a specialized device for deep learning inference, performs better in terms of energy consumption than the other common embedded devices such as the Raspberry Pi. This is because it contains the Coral Edge TPU module, which is designed for fast prototyping of machine learning hardware and performing energy-saving mathemat-

Table 5 The F-1 score, accuracy, precision, recall, average inference time, maximum RAM usage, and average energy consumption of a deep learning model on the Jetson TX2 when operated in a ROS environment using the EasyDLROS framework

Model	F-1 score	Accuracy	Precision	Recall	Avg. inference time (sec)	Max. RAM usage (MB)	Avg. energy used (J/image)
InceptionV1 (INV1)	87.50	88.00	91.30	84.00	0.84	70	3.18
InceptionV2 (INV2)	83.35	83.67	82.71	84.00	1.23	120	3.16
ResNet (RNV1)	77.61	79.67	88.14	69.33	1.13	120	3.59
GoogleNet (GN)	89.04	89.00	91.54	86.67	0.94	90	3.45
MobileNetV1 (MNV1)	87.01	86.68	84.81	89.33	0.34	50	3.41
MobileNetV2 (MNV2)	85.53	84.67	80.95	90.67	0.32	30	3.28
EdgeLite (EL)	90.00	90.39	92.08	88.02	0.93	70	3.32

ical operations, making the Coral Dev Board faster than other devices. This helps to reduce the inference time of a model hence also reducing energy consumption.

Observations and recommendations. Based on our experiments, we make the following observations and recommendations for deploying deep learning models on resource-constrained devices:

- The Jetson TX2 was the fastest edge device in our experiments but it is expensive, consumes more energy and physically larger than Coral Dev Board.
- Although the Coral Dev Board performs well in DL inference, it has a number of platform dependency issues. Currently, other than TensorFlow Lite, DL frameworks like Keras, MXNet, and PyTorch cannot run on the Coral Dev Board.
- The Raspberry Pi provides more flexibility to use different DL frameworks compared to the Coral Dev Board and also has a large technical support group making it useful in a wider range of systems and applications.
- The Raspberry Pi, while the least expensive, performed poorly compared to the Jetson TX2 and the Coral Dev Board in terms of inference time and energy consumption. For building a prototype, the Raspberry Pi is a good choice but for industrial deployment where performance metrics are very important, the Coral Dev Board or the NVIDIA TX2 are better.
- To build a deep learning-based edge infrastructure at a low cost, it is advisable to use the Coral Dev Board as it uses less energy and is smaller than the Raspberry Pi and the Jetson TX2, and performs inference faster than the Raspberry Pi.

8.2 Experiments on the ROS Environment

We created a simulated supermarket environment to test the efficiency of our EasyDLROS framework. A Jetson TX2, and a Logitech C925e webcam were used in this phase of the experiment. All the necessary ROS nodes and pre-trained DL models were deployed on the Jetson TX2 device and input images were captured by the webcam. The image node of the framework is capable of capturing 32 images per second. However, since 32 images per second are not needed to monitor the supermarket environment, we stored one image per second through the camera node. Our framework classified images captured with the camera node in real-time and stored them on a hard drive. The camera node captures images in 640 $\times$ 480 pixels, then the image processing node converts those images into 224 $\times$ 224 pixels. Finally, the DL node receives those converted images, feeds the images to DL models, and generates classification results. As we mentioned before, the framework utilizes CvBridge and OpenCV to capture the input images and MXNet to deploy the DL models.

We used EasyDLROS to run DL models on a ROS environment (inside Linux) and compared the performance of this framework with the performance of a DL model run on a Linux environment without ROS. Four metrics (viz, accuracy, memory usage, inference time, and energy consumption) were used to evaluate the efficiency of EasyDLROS (Sect. 8.2.1).

During the experiment, we monitored the memory usage carefully and reported the maximum RAM usage. The inference time of the DL model was captured programmatically and reported the average time consumed by the model to generate the classification result of each image. To evaluate the power consumption of the framework, we used NVIDIA's power measurement software [18], which can capture the power consumption of six of the primary supply rails (CPU, RAM, GPU, etc.). The maximum power (P_{max}) recorded at any time (T) during the evaluation period is reported as the maximum energy consumption ($P_{max} \times T$) of the framework.

8.2.1 Results for ROS Environment

Seven widely used deep learning models were used to evaluate the effectiveness of the EasyDLROS framework as summarized in Table 5. For assessing the performance of EasyDLROS, the results of a comparative performance evaluation over an existing Linux based framework on a set of metrics including accuracy, energy usage, and inference time are shown in Fig. 10.

F-1 score. We created a simulated environment of a supermarket and captured images using the ROS camera node to build a simulated test dataset. On that dataset, EdgeLite performed best, achieving an F-1 score of 90.00. GoogleNet is the second-best model, with an F-1 score of 89.04. In the simulated environment, the input images were captured from the test-set images of the supermarket dataset using the ROS camera node. Sometimes the quality of the captured images by this simulated process is

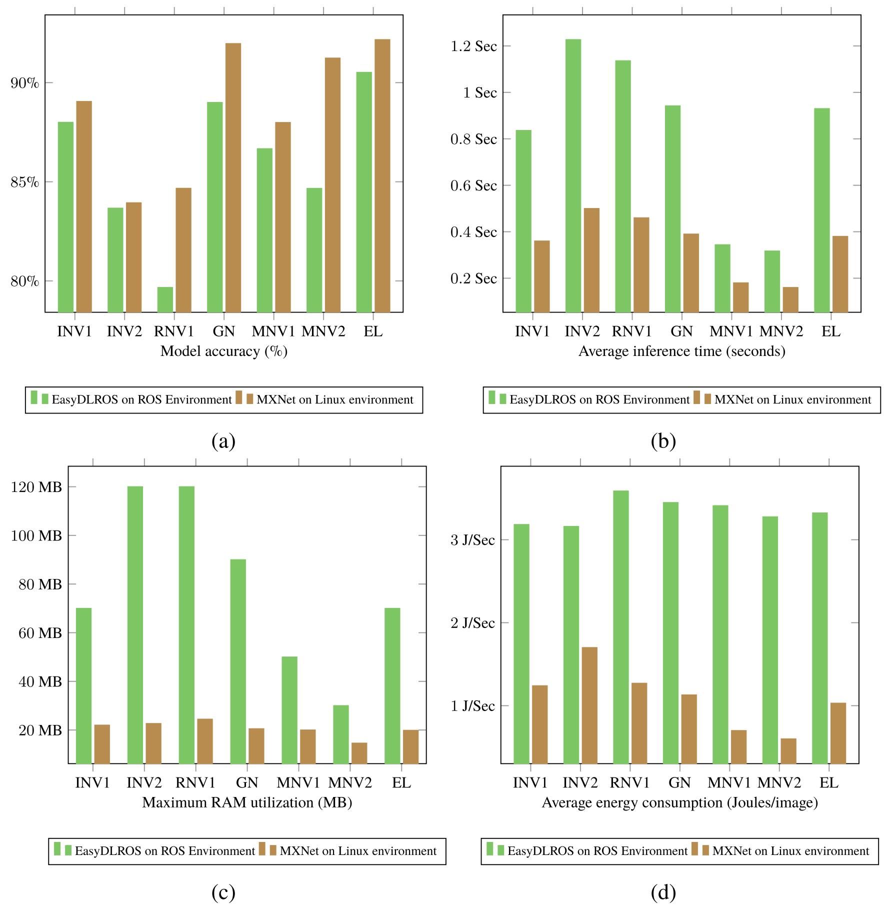

Fig. 10 A comparison of the performance of seven deep learning models on the ROS and plain Linux (without ROS) environments on the Jetson TX2. Figure 10a and Fig. 10b show model accuracy and execution time of different DL models on the supermarket hazard dataset respectively in two different environments. Model names and sizes are in Table 3. Figure 10c shows the maximum memory usage at any point during inference for the seven models in two different environments. Figure 10d shows the average energy consumption (Joules/image) during classification by the DL models in ROS and plain Linux environments

not as good as the quality of the original test-set images. This quality degradation has slightly reduced the accuracy of DL models executed in the ROS environment compared to models executed on Linux systems. The comparison results are shown in Fig. 10a. However, when we used the original test-set images as input to the DL model in the ROS environment, i.e., the images were directly copied from the dataset, we achieved the same accuracy as the model executed in the plain Linux environment.

Inference time. MobileNetV2 achieved an average inference time of 0.32 s per image on the ROS environment. Figure 10b shows that all the models in the ROS environment consumed more time than the inference time in the Linux system. The publishing, subscribing processes consume time which makes ROS systems a little slower than Linux systems.

Max RAM usage. Executing a DL model on the ROS environment consumes more memory than the Linux (without ROS) environment. The DL models consume a maximum of 120 MB RAM in the ROS environment to generate a classification result. MobileNet V2 consumes 30 MB, which is the best model in terms of memory usage. On the other hand, Inception V2 and ResNet V1 consume 120 MB to generate a result. Figure 10c shows the comparison of RAM usage by different DL models in the ROS and plain Linux environment.

Average Energy consumption. All DL models consume a little over 3 Joules energy to classify an image in the ROS environment. Inception V1 consumes the lowest energy, 3.18 Joules, among all models used in the experiment. The comparison of energy usage of DL models in two different environments is shown in Fig. 10d.

8.2.2 Analysis

We found the execution time of all DL models in the ROS environment takes more memory, energy, and time compared to the plain Linux environment (i.e., without ROS). This is primarily because different ROS nodes consume time and energy to publish and subscribe topics which make the ROS-based system a little slower and resource hungry. For example, MobileNetV2, the fastest model in the plain Linux environment, took 0.16 s when run on a TX2 device. It takes 0.32 s in the ROS environment when run on the same TX2 device. We believe that this performance cost is acceptable given the benefits of deploying DL models on autonomous robots run by ROS.

9 Conclusion

Onboard data analysis is rapidly becoming one of the key focus areas for AI researchers, but modern deep learning models typically have millions of parameters and involve a large number of complex computations, making their deployment on low-memory devices challenging. This chapter described EasyDLROS, a novel framework to deploy deep learning models on autonomous robots. First, we discussed why deep learning is important in the ROS environment and how deploying DL-based systems can help autonomous robot industries. We then described EdgeLite, a fast, lightweight, deep learning model for floor hazards detection geared toward easy deployment and inference on resource-constrained edge devices. The advantages and limitations of EdgeLite were detailed compared to over six state-of-the-art deep learning-based architectures for object detection on three resource-constrained edge devices were discussed. EdgeLite was shown to outperform six other models for detecting hazards in supermarket floors in terms of accuracy and had comparable performance in other metrics.

We introduced and described EasyDLROS, a framework for easy deployment of deep learning models on ROS-based autonomous robots. A Jetson TX2 was used to evaluate the performance of models deployed using EasyDLROS. The EdgeLite model, when deployed in a ROS environment using EasyDLROS, achieved almost the same accuracy (only 1.78% loss of accuracy) compared with the scenario in which the model was deployed on a the Jetson TX2 running plain Linux, without ROS. Also, EasyDLROS used only a little more memory and energy making it a useful tool for automating the deployment of deep learning models on edge devices without causing any significant computational overhead.

References

1. Alom, Z., Taha, T., Yakopcic, C., Westberg, S., Sidike, P., Nasrin, M.S., Hasan, M., Essen, B.V., Awwal, A., Asari, V.: A state-of-the-art survey on deep learning theory and architectures. Electronics **8**, 292 (2019)
2. Chang, Y.H., Chung, P.L., Lin, H.W.: Deep learning for object identification in ROS-based mobile robots. In: IEEE International Conference on Applied System Invention (ICASI). Chiba, pp. 66–69 (2018). https://doi.org/10.1109/ICASI.2018.8394348
3. Chen, P., Hang, H., Chan, S., Lin, J.: An efficient CNN for road scene segmentation. Asia Pac. Signal Inf. Process. Assoc. Annu. Summit Conf. (APSIPA ASC), 424–432 (2019)
4. Cheng, Y., Wang, D., Zhou, P., Zhang, T.: A survey of model compression and acceleration for deep neural networks (2017). https://arxiv.org/abs/1710.09282
5. Dhar, S., Guo, J., Liu, J., Tripathi, S., Kurup, U., Shah, M.: MobileNets: efficient convolutional neural networks for mobile vision applications (2017). https://arxiv.org/abs/1704.04861
6. Dhar, S., Guo, J., Liu, J., Tripathi, S., Kurup, U., Shah, M.: On-device machine learning: an algorithms and learning theory perspective. https://arxiv.org/abs/1911.00623
7. Dörr, L., Brandt, F., Meyer, A., Pouls, M.: Lean training data generation for planar object detection models in unsteady logistics contexts. In: 18th IEEE International Conference on Machine Learning and Applications (ICMLA-2019). Boca Raton, FL, USA, pp. 329–334. https://doi.org/10.1109/ICMLA.2019.00062
8. Feng, Z., George, S., Harkes, J., Klatzky, R., Satyanarayanan, M., Pillai, P.: Eureka: edge-based discovery of training data for machine learning. IEEE Internet Comput. **23**, 35–42 (2019)
9. Hsu, C.C., Wang, M.Y., Shen, H.C.H., Chiang, R.H., Wen, C.H.P.: FallCare+: an IoT surveillance system for fall detection. In: 2017 International Conference on Applied System Innovation (ICASI). Sapporo, Japan, pp. 921–922 (2017). https://doi.org/10.1109/ICASI.2017.7988590
10. Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., Bengio, Y.: Binarized neural networks. Adv. Neural Inf. Process. Syst. (2016)
11. Iandola, F.N., Han, S., Moskewicz, M.W., Ashraf, K., Dally, W.J., Keutzer, K.: SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size (2017). https://arxiv.org/abs/1602.07360
12. Install MXNet on a Jetson. https://mxnet.apache.org/versions/1.6/get_started/jetson_setup.html. Last accessed 13 Aug 2020

13. Kakaletsis, E., Tzelepi, M., Kaplanoglou, P.I., Symeonidis, C., Nikolaidis, N., Tefas, A., Pitas, I.: Semantic map annotation through UAV video analysis using deep learning models in ROS. In: 25th International Conference, MMM: Thessaloniki, Greece, January 8–11, 2019. Proceedings, Part II (2019). https://doi.org/10.1007/978-3-030-05716-9_27
14. Lane, N.D., Bhattacharya, S., Georgiev, P., Forlivesi, C., Jiao, L., Qendro, L., Kawsar, F.: DeepX: a software accelerator for low-power deep learning inference on mobile devices. In: 2016 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN). Vienna, Austria, pp. 1–12 (2016). https://doi.org/10.1109/IPSN.2016.7460664
15. Liu, M., Niu, J., Wang, X.: An autopilot system based on ROS distributed architecture and deep learning. In: IEEE 15th International Conference on Industrial Informatics (INDIN). Emden, pp. 1229–1234 (2017). https://doi.org/10.1109/INDIN.2017.8104950
16. Muhammad, A., Aseere, A., Chiroma, H., Shah, H., Gital, A.Y., Hashem, I.A.: Deep learning application in smart cities: recent development, taxonomy, challenges and research prospects. Neural Comput. Appl. **33**, 2973–3009 (2020)
17. Murshed, M.G., Murphy, C., Hou, D., Khan, N., Ananthanarayanan, G., Hussain, F.: Machine learning at the network edge: a survey (2019). https://arxiv.org/abs/1908.00080
18. NVIDIA. Thermal design guide (2017). https://developer.nvidia.com/embedded/dlc/jetson-tx2-thermal-design-guide. Last accessed 18 June 2020
19. Ramos, S., Gehrig, S., Pinggera, P., Franke, U., Rother, C.: Detecting unexpected obstacles for self-driving cars: fusing deep learning and geometric modeling. In: IEEE Intelligent Vehicles Symposium (IV) (2017). https://doi.org/10.1109/IVS.2017.7995849
20. Rastegari, M., Ordonez, V., Redmon, J., Farhadi, A.: XNOR-Net: ImageNet classification using binary convolutional neural networks. In: ECCV (2016)
21. Robot Operating System (ROS). https://www.ros.org/ Last accessed 31 Aug 2020
22. Saha, O., Dasgupta, P.: A comprehensive survey of recent trends in cloud robotics architectures and applications. Robotics **7**, 47 (2018)
23. Sarwar Murshed, M.G., Verenich, E., Carroll, J.J., Khan, N., Hussain, F.: Hazard detection in supermarkets using deep learning on the edge. In: 3rd USENIX Workshop on Hot Topics in Edge Computing (HotEdge–2020). https://www.usenix.org/conference/hotedge20/presentation/murshed
24. Shabbir, J., Anwer, T.: Survey of deep learning techniques for mobile robot applications (2018). https://arxiv.org/abs/1803.07608
25. Silva, E.M., Maló, P., Albano, M.: Energy consumption awareness for resource-constrained devices. In: 2016 European Conference on Networks and Communications (EuCNC). Athens, Greece, pp. 74–78 (2016). https://doi.org/10.1109/EuCNC.2016.7561008
26. Sisido, F., Goya, J., Bastos, G.S., Li, A.W.: Traffic signs recognition system with convolution neural networks. In: Latin American Robotic Symposium, 2018 Brazilian Symposium on Robotics (SBR) and 2018 Workshop on Robotics in Education (WRE). Joao Pessoa, pp. 339–344 (2018). https://doi.org/10.1109/LARS/SBR/WRE.2018.00068
27. Su, H., Zhang, Y., Li, J., Hu, J.: The shopping assistant Robot design based on ROS and deep learning. In: 2016 2nd International Conference on Cloud Computing and Internet of Things (CCIOT). Dalian, China, pp. 173–176 (2016). https://doi.org/10.1109/CCIOT.2016.7868328
28. Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., Rabinovich, A.: Going deeper with convolutions. In: 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). Boston, MA, USA, pp. 1–9 (2015). https://doi.org/10.1109/CVPR.2015.7298594
29. Tan, M., Le, Q.V.: EfficientNet: rethinking model scaling for convolutional neural networks (2019). https://arxiv.org/abs/1905.11946
30. Tang, B., Chen, Z., Hefferman, G., Pei, S., Wei, T., He, H., Yang, Q.: Incorporating intelligence in fog computing for big data analysis in smart cities. In: IEEE Transactions on Industrial Informatics (2017). https://doi.org/10.1109/TII.2017.2679740
31. Wani, M.A., Kantardzic, M., Sayed-Mouchaweh, M.: Deep Learning Applications. Springer (2020)

32. Wani, M.A., Khoshgoftaar, T.M., Palade, V.: Deep Learning Applications, vol. 2. Springer (2021)
33. Whitney, D., Rosen, E., Phillips, E., Konidaris, G., Tellex, S.: Comparing robot grasping teleoperation across desktop and virtual reality with ROS reality. In: ISRR (2017)
34. X-DRAGON USB 2.0 digital multimeter power meter tester current and voltage monitor. http://www.x-dragon.net/index.php?c=product&id=421. Last accessed 17 July 2020
35. Zhang, X., Zhou, X., Lin, M., Sun, J.: ShuffleNet: an extremely efficient convolutional neural network for mobile devices. In: 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 6848–6856
36. Zhang, Q., Zhang, M., Chen, T., Sun, Z., Ma, Y., Yu, B.: Recent advances in convolutional neural network acceleration. Neurocomputing **323**, 37–51 (2019)