

École polytechnique de Louvain

Polyp classification for detection of colorectal cancer with capsule endoscopy

Author: **Sami GHARIB**
Supervisors: **David BOL, Martin LEFEBVRE**
Reader: **Christophe DE VLEESCHOUWER**
Academic year 2019–2020
Master [120] in Electrical Engineering

Abstract

In addition of being one of the most frequent types of cancer, colorectal cancer also represents one of the major causes of cancer deaths worldwide. Colorectal cancer can be diagnosed by detecting the presence of polyps inside the colon or the rectum, and due to its non-invasive properties, this is commonly done by using Wireless capsule endoscopy (WCE). In order to provide help to the doctors, WCE can be accompanied by computer-aided diagnosis (CAD). This allows saving time analyzing the images.

On the other hand, deep learning has proven to be efficient when it comes to image recognition, and embedded inference has become popular as it answers concerns such as latency, security, and energy wasted on raw data transmission.

In this work, we combine CAD and embedded inference by proposing a CNN based approach for embedded polyp detection inside a microcontroller. We started by designing a CNN for polyp classification, and we could achieve $4\times$ model compression to match microcontrollers memory constraints. This was done by reducing the model parameters to 8-bit fixed point numbers, at the cost of a 3% and 6% recall and accuracy deviation. The final model provides 92% recall and 90% accuracy on a dataset generated from the ETIS-Larib polyp database.

Acknowledgements

I would like to thank my supervisors David Bol and Martin Lefebvre, for the opportunity to work on this project and their support throughout this work.

I would also like to thank professor Christophe De Vleeschouwer for agreeing reading this thesis.

Contents

1	Introduction	5
2	Background	7
2.1	Colorectal cancer	7
2.2	Wireless Capsule Endoscopy	8
2.3	Convolutional Neural Networks	10
2.3.1	Neural Networks	11
2.3.2	Convolutional Neural Network	15
2.4	Embedded Neural Networks	20
2.4.1	Network quantization	22
2.4.2	Convolutional Neural Network libraries	23
2.5	Related works	25
2.6	Summary	27
3	Database	28
3.1	Available databases	29
3.2	Database creation	30
3.2.1	Data augmentation	30
3.2.2	Labeling	33
3.3	Summary	35
4	Model implementation and optimization on PyTorch	36
4.1	Google Colab	36
4.2	PyTorch	37
4.3	Model architecture	39
4.4	Hyperparameters tuning	43
4.4.1	Different methods for hyperparameters tuning	43
4.4.2	Grid search with reduced complexity	44
4.5	Summary	52

5	Inference on microcontroller	53
5.1	Implementation on CMSIS-NN	53
5.1.1	From PyTorch to CMSIS-NN	54
5.2	Code architecture	57
5.3	Device selection	59
5.4	Summary	62
6	Conclusion	63
A	Histograms of polyp sizes	65
B	Network without dropout	66
C	Computation of forward/backward size	67
D	Comparison of learning rates operation for SGD and Adam	69

Acronyms

Adam Adaptive Moment Estimation.

API Application Programming Interface.

CAD Computer-Aided Diagnosis.

CBA Convolution, Batch-Normalization, Activation.

CMYK Cyan, Magenta, Yellow, Black.

CNN Convolutional Neural Network.

CPU Central Processing Unit.

CRC Colorectal cancer.

CSV Comma-Separated Values.

CUDA Compute Unified Device Architecture.

DL Deep Learning.

DSP Digital Signal Processor.

FC Fully-Connected.

FN False Negative.

FP False Positive.

GPU Graphics Processing Unit.

HSV Hue, Saturation, Brightness.

HWC Height, Width, Channels.

IDE Integrated Development Environment.

im2col Image to Column.

LSB Least Significant Bit.

MAC Multiply-and-Accumulate.

MICCAI Medical Image Computing and Computer Assisted Intervention.

NCHW Number of images in batch, Channels, Height, Width.

NHWC Number of images in batch, Height, Width, Channels.

NN Neural Network.

ONNX Open Neural Network Exchange.

RAM Random Access Memory.

ReLU Rectified Linear Unit.

RGB Red, Green, Blue.

RoI Region Of Interest.

ROM Read-Only Memory.

SGD Stochastic Gradient Descent.

SIMD Single Instruction on Multiple Data.

SVM Support Vector Machine.

TN True Negative.

TP True Positive.

TWN Ternary Weight Network.

VM Virtual Machine.

WCE Wireless Capsule Endoscopy.

Chapter 1

Introduction

One of the most diagnosed types of cancer is the colorectal cancer (CRC). It is generally characterized by the appearance of polyps in the sidewalls of the colon or the rectum. A polyp is a small clump of cells that grows inside your body, and based on their size, color, and texture, they can either be malignant and lead to cancer or be benign.

Colonoscopy is the procedure used to examine the colon or the rectum by the use of a colonoscope, which is a long flexible tube ended by a camera. It is inserted at the back of the patient so that the doctor can see the inside of the entire colon and assess whether it contains malignant polyps.

However, due to its invasive nature, this process has a low acceptance in the general population, and this lead to the emergence of Wireless capsule endoscopy (WCE), which uses a non-invasive camera for the same diagnosis.

An emerging concept that is often solicited for image recognition problems is deep learning (DL). It has known a growth since 2010 with the increasing availability of powerful servers and GPUs (Graphics Processing Unit) [42].

The application of deep learning in the context of polyp detection is referred to as a computer-aided diagnosis (CAD). The machine helps by giving the first diagnosis while the doctor has to make the final decision.

This thesis focuses on applying deep learning to the problem of colorectal cancer diagnosis by performing polyp classification. To do so, a convolutional neural network (CNN) is embedded inside a WCE camera and acts as a first soft filtering layer that only transmits polyp images. It has to be soft to make sure not to miss polyp-containing images. This helps the doctor by narrowing the set of images to be observed to the ones that may contain a polyp.

This thesis nevertheless comes with some limitations. First, the images used

for training and testing the network are derived from the ETIS-Larib Polyp DB database. This also implies classification of images with one polyp only. Then, we limit the prototyping devices to the ones provided by the Arm DS IDE tool.

Let us now introduce how this work is presented.

First, **Chapter 2: Background** briefly introduces colorectal cancer and wireless capsule endoscopy . After this, convolutional neural networks and embedded neural networks concepts are detailed. Then, this chapter ends by exploring works related to polyp classification and embedded inference.

Subsequently, **Chapter 3: Database** browses available databases of polyp images to feed our model. It then explains how the final model dataset is created by augmenting existing ones and labeling them.

Next, **Chapter 4: Model implementation and optimization on PyTorch** introduces Google Colab and Pytorch, which are the frameworks used to implement the model. It then explains how the model is architected and details the method used to select its hyperparameters. The best model is then selected based on performance.

Finally, **Chapter 5: Inference on microcontroller** tackles the network embedded inference by first enumerating the requirements to convert a network from Python to C. It then reduces the network size to make it embeddable by quantizing it, and compares performance with the full-precision solution. Finally, it implements the network under CMSIS-NN and ends by the selection of the device that suits the best for the application.

Chapter 2

Background

In this chapter, we will go through the concepts required to address the content of this work. We will first tackle the medical scope by introducing colorectal cancer and wireless capsule endoscopy.

In addition to this medical background, this thesis especially focuses on convolutional neural networks and how to make them embeddable. We will thus have an introduction about convolutional neural networks and embedded neural networks. We will finally go through works related to polyp detection and network embedding.

2.1 Colorectal cancer

Colorectal cancer is a cancer starting in the colon or the rectum. According to [30], the overall lifetime risk of developing colorectal cancer was about 4.4% for men and 4.1% for women in 2018, making it one of the major causes of cancer deaths worldwide. Most of them start as a growth on the inner lining of the colon or rectum and these growths are called polyps [31].

Some polyps can turn into cancer over time depending on which of the two main types it is. Adenomatous polyps (adenomas) are the ones that can change into cancer while hyperplastic polyps and inflammatory polyps are more common, but they are generally not pre-cancerous. Other factors may indicate that a polyp contains cancer, including if a polyp is larger than 1cm, if more than two polyps are found in the same area, or if dysplasia is seen in the polyp after it is removed. Dysplasia is an area in a polyp or the lining of the colon or rectum where the cells look abnormal. This work focuses on detecting any of these types of polyp as a first step in colorectal cancer diagnosis.

2.2 Wireless Capsule Endoscopy

Endoscopy is the medical procedure where a tube-like instrument called an endoscope is used to look at organs inside of the body. There are several types of endoscopies depending on which part of the body is being looked at. When it comes to detecting CRCs, we observe the inside of the colon and the rectum, and it is called a colonoscopy. In this case, the doctor uses a colonoscope, which is a long and flexible tube with a light and small camera on its end. It is put through the back and into the rectum and colon as shown in Fig. 2.1. This way, it can be used to scout for polyps.

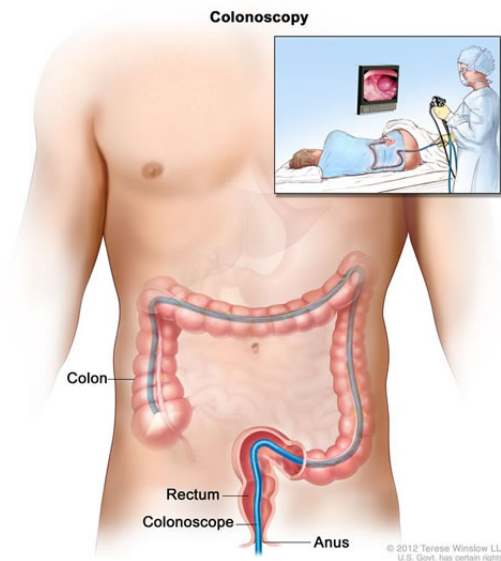


Figure 2.1: In colonoscopy, a colonoscope is inserted through the back to get images from the inside of the colon [21].

Wireless capsule endoscopy uses a tiny wireless camera that is ingested by the patient to take pictures of the digestive tract and it helps doctors see inside of the small intestine which is a non easily reached area with wired endoscopy. This way, as it travels through the digestive tract, the camera takes pictures and transmits them to a receiver.

Then, WCE camera features that are interesting to notice for this work are the imager resolution and the image sampling rate. They respectively can be used to determine the neural network input size resolution and the device frequency requirement. A table showing several WCE capsules is provided in Fig. 2.2.

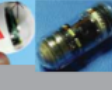
Model													
	PillCam SB (M2A)	PillCam SB 2	PillCam SB 3	PillCam ESO	PillCam ESO 2	PillCam ESO 3	PillCam Colon	PillCam Colon 2	MiroCam® MC1000-W	EndoCapsule	EndoCapsule 10	OMOM®	CapsoCam SV1
Manufacturer	Given Imaging Ltd. (Yokneam, Israel) (location)								Intromedic Co. (Seoul, Korea)	Olympus Inc. (Tokyo, Japan)		JinShan Science and Technology Co. (Chongqing, China)	CapsoVision, Inc. (Saratoga, CA, USA)
Operative region	SB			ESO			Colon		SB	SB		SB	SB
Mass (g)	3.3	2.9	3.0	3.7	2.9	2.9	2.9	2.9	3.25	3.8	3.3	≤6	N/A
Dimensions (mm)	11 × 26	11 × 26	11 × 26	11 × 26	11 × 26	11 × 26	11 × 31	11 × 31	10.8 × 24.5	11 × 26	11 × 26	13 × 27.9	11 × 31
Battery (h)	7 ± 1	≥8	≥8	0.3	0.5	0.5	9±1	10	12	8	12	8 ± 1 (2 fps)	~15
Illumination	6 LED	4 LED	4 LED	2 × 6 LED	2 × 4 LED	2 × 4 LED	2 × 4 LED	2 × 4 LED	6 LED	6 LED	4 LED	6 LED	16 LED
Lighting	Standard	Automatic	Automatic	Standard	Automatic	Automatic	Automatic	Automatic	Standard	Automatic	Automatic	Standard	Automatic
Field of view (°)	140	156	156	140	169	172	156	172	170	145	160	140 ± 10	360
Depth of field (mm)	0-30	0-30	0-30	0-30	0-30	0-30	0-30	0-30	7-20	0-20	0-20	0-35	0-30
Image sensor	1 × CMOS	1 × CMOS	1 × CMOS	2 × CMOS	2 × CMOS	2 × CMOS	2 × CMOS	2 × CMOS	1 × CMOS	1 × CCD	1 × CCD	1 × CMOS	4 × CMOS
Imager resolution	256 × 256	256 × 256	320 × 320	256 × 256	256 × 256	256 × 256	256 × 256	N/A ¹	320 × 320	1920 × 1080	1920 × 1080	QVGA (320 × 240) and VGA (640 × 480)	N/A
Image sampling rate (fps)	2	2 or 4 ⁴	2 or -6	4 or 14 ⁵	19	35	2 per camera	4 or 35	3	2	2	2 fps (QVGA) or 0.5 fps (VGA)	3 or 5 per camera
Adaptive frame rate	No	No	Yes	No	No	No	No ⁶	Yes	No	No	No	No	Yes
Data transmission	RF	RF	RF	RF	RF	RF	RF	RF	HBC	RF	RF	RF	USB
Rx antennas	8	8	8	3	3	3	8	8	9	8	8	14	0
Real-time monitor	No	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes, USB and Wi-Fi	Yes	Yes	Yes	No
Localization	Yes ⁸	Yes ⁸	Yes ⁸	No	No	No	Yes ¹¹	Yes ¹¹	No	No	3D Track	No	No
US FDA approval	Yes (2001)	Yes (2007)	Yes (2013)	Yes (2004)	Yes (2007)	Yes (2011)	No	Yes (2014)	Yes (2012)	Yes (2007)	No	No	No

Figure 2.2: Different WCE camera specifications from [39]. Their dimensions are typically around $10 \times 30 \text{ mm}^2$ and their imager resolution are generally equal to 256×256 pixels. One capsule from the table operates in the colon, it is the *PillCam Colon(2)* and its battery lifetime is around 10h so that it has enough autonomy to take pictures while traveling the digestive track.

Finally, WCE can be accompanied by computer-aided diagnosis. In the context of polyp detection, CAD consists in providing help to the doctor by highlighting frames of interest which may contain a polyp. As wireless communication is power-hungry, pre-processing the pictures before transmitting them also presents the advantage to save power by only sending the ones that potentially contain a polyp instead of continuously transmitting all pictures that are taken.

2.3 Convolutional Neural Networks

Machine learning can reproduce a pattern thanks to algorithms that are fed by a large amount of data. Based on distinct patterns, the algorithm learns decisions to make and creates a model. As machine learning progresses, the human footprint disappears. Traditionally, as shown in Fig. 2.3, engineers had to extract hand-crafted features to train the algorithm and apply on them.

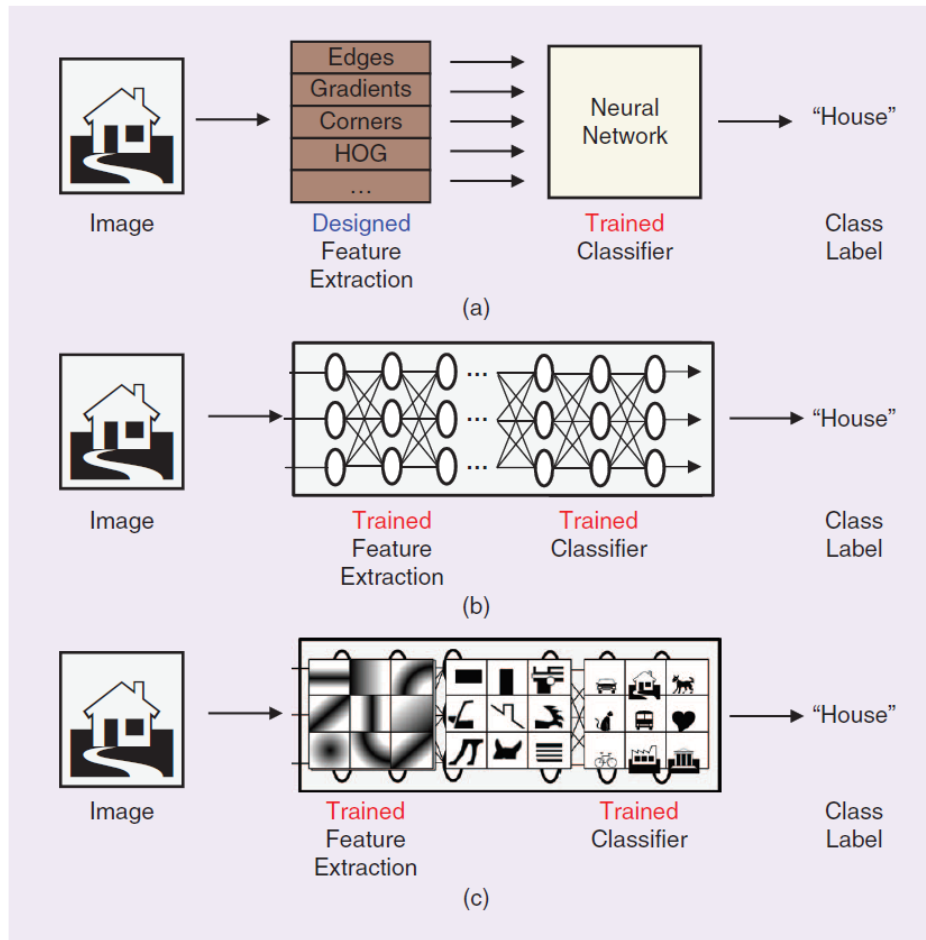


Figure 2.3: (a) Traditionally, machine learning classifiers were trained and applied on hand-crafted features. (b) Deep learning allowed the network to learn and extract the optimal feature sets. (c) Such a network trains itself to extract objects from an image. This image comes from [42].

Today, the human task rather lies in designing a model and collecting the data on which it will train to extract features instead.

One distinguishes three types of learning: the supervised learning where we have labeled data, the unsupervised learning with no labeled set, and the reinforced learning where the algorithm is rewarded for positive results and punished for negative results, allowing it to learn. Supervised learning is used to solve problems such as regression or classification, while unsupervised learning rather copes with problems such as clustering, adaptive filtering, and visualization.

In this work, we see the task as a classification problem where we have labeled data to train on to decide whether there is a polyp in an image.

2.3.1 Neural Networks

In classification and pattern recognition, neural networks (NNs) stand as an efficient tool. A neural network is a deep learning computing system inspired by the human brain learning behavior.

It is composed of several layers. The first layer is the input layer and it receives raw input data, then come the hidden layers that take as input the output of the previous layer. At the end, the last layer is the output layer, and in image classification it generally outputs the probability of the input image to belong to each class. We call network depth the number of hidden layers, and it is defined by the programmer. The set of such parameters that have to be chosen by the programmer are called the **hyperparameters**.

Then, each layer contains a set of artificial neurons as shown in Fig. 2.4. These neurons are called nodes and are connected to neurons from previous layer as in Fig. 2.5. In addition, each connection between nodes has a weight that indicates the strength and direction of each node input.

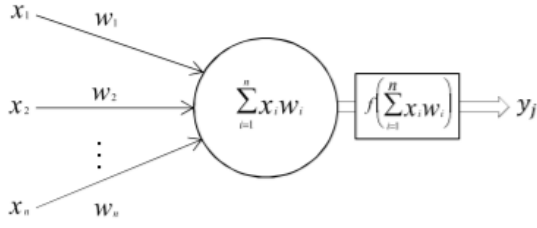


Figure 2.4: Nodes compute the sum of inputs x_i multiplied by their weights w_i . Biases might also be added to this result at each node. They are then followed by an activation function $f(\sum x_i * w_i)$ that decides whether the node is activated for next layer and provides an input for the next layer y_j . Image taken from [43].

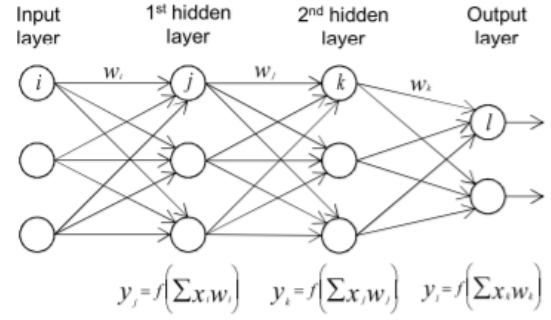


Figure 2.5: Example of a fully-connected neural network with 2 outputs. Every node is connected by all the previous nodes. These connections have a weight and the nodes have an activation function. Image taken from [43].

In image processing, NNs are used for several applications. Let us introduce them. First, **image classification** is the task we target in this work. It consists in taking an image as input and outputting a class number to classify the image. Then, **object localization** is not only producing a label but also a box that contains the object of interest, to show where it is in the image. Similarly, **object detection** provides object localization on all the objects in the image. There are several bounding boxes, one for each object. And finally, **object segmentation** consists in outputting a class label and a contour for each object in the input image.

Training a neural network

Training the network is the first step of deep learning classification problems. Before the NN starts, the weights are randomized or initialized and we call **back-propagation** the training process that allows to adjust them.

We can separate backpropagation into **4 steps**. First step, the **forward pass** is when a training image passes all the way through the network, and what will make the network learn after this step is the **loss function**, or cost function. As labels accompany training data, the network can see how far from the correct solution it is.

The loss function can vary depending on the problem. For binary classification problem as in our case, the problem translates as predicting the likelihood of a sample to belong to a class and the corresponding loss function is the **Cross-Entropy**. It is also known as Logarithmic loss.

For n training samples and d output neurons, the cross-entropy writes:

$$C = -\frac{1}{n} \sum_{k=1}^n \sum_{j=1}^d [y_j \times \log(o_j) + (1 - y_j) \times \log(1 - o_j)]$$

where C is the loss function, y is the indicator (1 or 0) if class label d is the correct classification for the observation, and o is the predicted probability that the observation is of class d .

During the first iterations, the loss is high and we want to minimize it by updating the weights that contribute the most to this loss.

The third step is the **backward pass** through the network. It determines the weights that contribute the most to the loss and adjusts them to decrease it.

This describes the expression $\frac{dC}{dW}$ where W are the weights at a particular layer and C the loss.

Once this derivative computed, the last step is the **weights update**.

Here, we update all the weights and biases of the filters via an **optimizer** to reduce the final loss. Thus, the weights change in the opposite direction of the gradient. We call this learning process **gradient descent** and it updates the weights and biases as

$$w \leftarrow w - \eta \frac{dC(w, b)}{dw}$$

$$b \leftarrow b - \eta \frac{dC(w, b)}{db}$$

w represents the weights, b the biases, $C(w, b)$ the loss function, and η the **learning rate**.

The learning rate is a hyperparameter that defines how big are the steps during the weights update. Thus, bigger learning rates take less time for the model to converge but may also make large steps that may not be precise enough to reach the optimal weights and biases.

All the 4 steps that have just been described define one training iteration, and they are repeated until convergence or after several epochs.

In practice, for one iteration, only a small subset of sample, called a batch, is considered at each gradient descent step. And one complete pass over the training set is called an **epoch**. The size of a batch is also a hyperparameter. It defines how frequently the network updates its weights, or on how many samples it bases its generalization. Such an alternative to gradient descent is called Stochastic Gradient Descent (SGD) and is used by the **SGD optimizer**.

Another efficient common optimizer is the **Adam optimizer** [25]. Adam stands

for Adaptive Moment Estimation. This optimizer uses momentum and adaptive learning rates to converge faster. We can explain momentum using the analogy of a ball rolling down a hill, and that rolls faster towards the local minima. This is introduced in the network by the following mathematical expression:

$$v_t = \gamma v_{t-1} + \eta \nabla J(\theta)$$

$$\theta = \theta - v_t$$

where θ is the parameter of the network, η the learning rate, J the objective function that we try to optimize, and γ a momentum term close to 1. γ is usually initialized to 0.9.

The intuition behind adaptive learning rate is that we start with big steps and decrease the steps as we get close to the local minimum.

Testing a neural network

Testing the neural network comes after training. During the test, the filters are frozen, which means that the weights and the biases are no more updated, and the network stops learning. Also, the performance of the network is evaluated by comparing the outputs with raw input. In the case of classification, we compare the classes assigned to input images by the network to their labels.

We use metrics to express the performance of a network. Here are the ones we will measure in this work.

The loss: It measures the performance of a classification model whose output is a probability value between 0 and 1. The loss increases as the predicted probability diverges from the actual label. Small loss results in a globally accurate network. In our case, we will focus on the cross-entropy as loss function.

The accuracy: It computes $\frac{TP+TN}{TP+TN+FP+FN}$. An accuracy close to 1 indicates that the network classifies well true positive (TP) and true negative (TN) samples.

The precision: A precision close to 100% means a few number of false positives (FP) compared to the true positives as it computes $\frac{TP}{TP+FP}$.

The recall: It represents $\frac{TP}{TP+FN}$. A large recall means a small number of false negatives (FN).

TP stands for true positive, TN for true negative, FP for false positive, and FN for false negative. These are defined in Table 2.1.

		Actual value	
		positive	negative
Predicted value	positive	true positive	false positive
	negative	false negative	true negative

Table 2.1: True positives/negatives are the samples that are correctly predicted positive/negative by the network, while false positives/negatives are the samples whose positive/negative prediction is false

Overfitting

An **overfitted** model is a model that contains more parameters than can be justified by the data. In machine learning, overfitting is the struggle of a model to generalize features from data, and the network generalizes from noise instead.

Note that it is not easy to detect when a model starts to overfit. However, preventing from overfitting can be done by reducing the network layers, or reducing the elements in hidden layers by applying regularization or using dropout layers.

Regularization consists in imposing constraints on a NN to prevent overfitting. For example, L2-regularization refers to penalizing large weights by adding a term

$$R = \frac{1}{2}\lambda \sum w^2$$

to the loss functions. λ controls the magnitude of the regularization term by multiplying the sum of the squares of the weights w .

Other regularization methods exist such as L1-distance or "Manhattan distance" that does the same as L2-regularization but without squaring the weights.

Another way to control overfitting is the use of **dropout layers** that drop out a random number of activations by setting them to zero. This way, the network does not fit too much to the training data and helps to avoid overfitting. Note that we only use this layer for training, when weights are updated.

2.3.2 Convolutional Neural Network

A convolutional neural network is a special type of neural network that is designed to process images. CNNs contain four main types of layers as hidden layers: convolutional layers, activation or ReLU (Rectified Linear Unit) layers, pooling layers, and fully-connected (FC) layers. Figure 2.6 depicts an example of a typical architecture for a CNN.

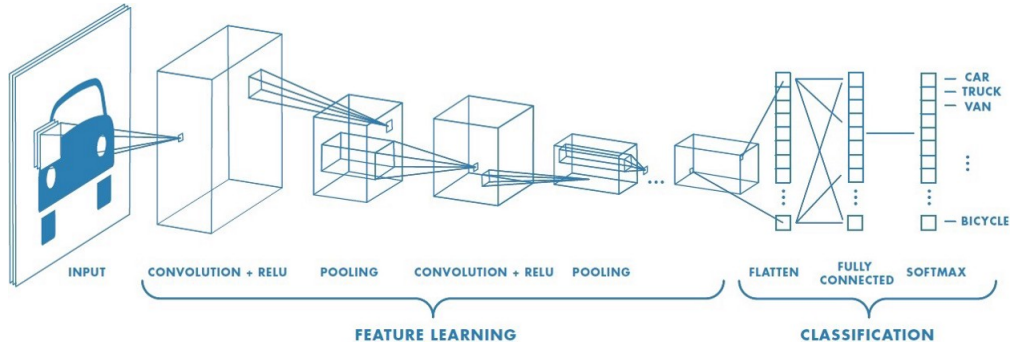


Figure 2.6: Typical architecture of a CNN from [40]. It is first composed of a feature learning part that contains convolutional layers, activation layers, and pooling layers. Then, there is a classification part that contains fully-connected layers.

Layers of a convolutional neural network

First, the **input image** is characterized by its height and width, and its number of channels. The number of input channels may vary depending on the input image format. Indeed, an image can be represented as grayscale, RGB (red, green, blue), HSV (hue, saturation, brightness), CMYK (cyan, magenta, yellow, black), etc... In this work, we will focus on RGB images. Figure 2.7 shows a 3 channel (Red, Green and Blue) image of dimensions 4x4 pixels.

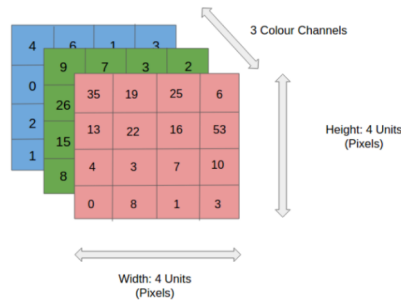


Figure 2.7: 4×4×3 RGB image from [40]. One can see the red, green, and blue components of the image that correspond to its channels. This image contains 3 components of dimensions 4×4.

Then, the **convolutional layer** convolves each channel of the input or the feature map with a filter, which is also referred to as a kernel. We call this a convolution as an identical filter is applied all along the same input channel with a defined stride and padding. The values of the filters correspond to its weights, we can also add a bias at the end of the convolution.

Then, the output of a convolutional layer is called a feature map.

Figure 2.8 shows the feature map resulting from the convolution of a 5×5 image with a 3×3 filter. Note that it is a convolution of one channel only. Typically, there are several channels, and each is convoluted by a corresponding filter.

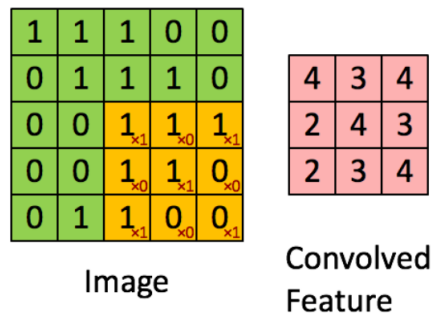


Figure 2.8: Convolution of a 5×5 image in green with a 3×3 kernel in yellow to get a 3×3 convoluted feature that is shown in red. One can notice that convolution corresponds to an element-wise product between the filter and the image components. Image from [40].

In a convolutional layer, the **stride** controls how the kernel travels around the input as it corresponds to the shift step of the convolution. Figure 2.9 shows input and output for a stride value of 1 in (a) and 2 in (b).

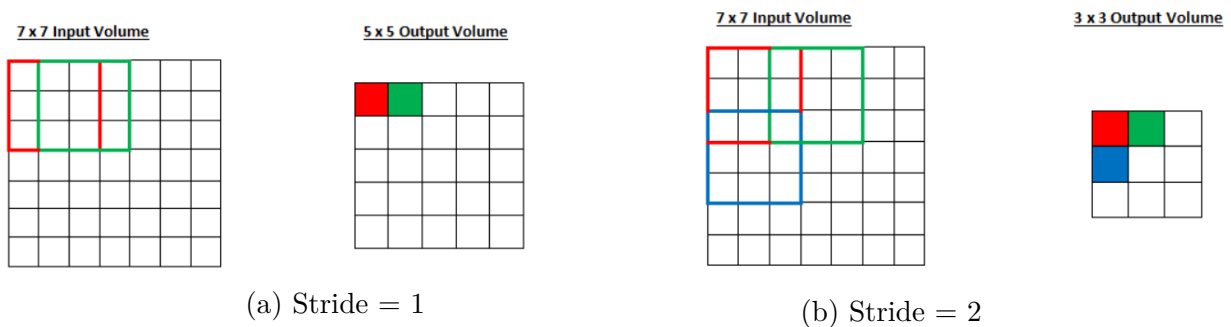


Figure 2.9: Illustration of input/output for (a) stride = 1, and (b) stride = 2. During the convolution, the filter is shifted by the value of the stride. Image taken from [17].

Padding consists in adding zeros at the edges of an image as shown in Fig. 2.10 for a padding of 1.

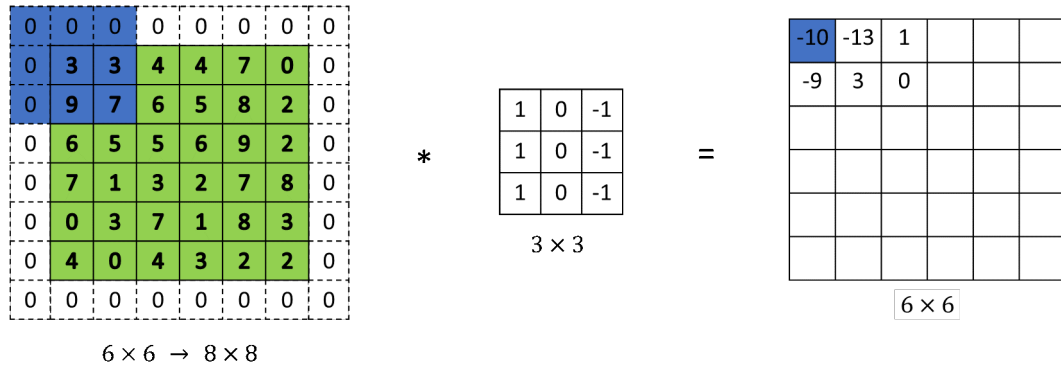


Figure 2.10: Example of a padding of 1. It consists in adding extra zero pixels at the borders of an image. It is used to increase output dimensions by preserving values at the edges of the image. Image from [6].

To have the pixels at the limits multiplied by the filter the same amount of times as the central pixels, one should have a padding of $\frac{K-1}{2}$, where K is the filter size. This is true when the stride is equal to 1. Otherwise, it should be $\frac{(S-1)*W-S+K}{2}$ where S is the stride, and W the height or the width of the input.

The padding, stride, and kernel size are parameters that allow to adjust the **output size**, based on input dimensions, and for square-shaped input and filters, we get the output dimensions as

$$O = \frac{W - K + 2P}{S} - 1$$

where O is the output height/width, W is the input height/width, K the filter size, P the padding, and S the stride.

An **activation layer** or **ReLU layer** conventionally comes after a convolutional layer and is used to introduce non-linearity in the system. The ReLU layer applies the function $f(x) = \max(0, x)$ to all the values in the input volume.

Subsequently, **pooling layers** come after activation layers, and they are used to decrease the size of computations by downsampling the input. For example, max-pooling combines a cluster of neurons by taking the maximum value over it. Pooling layers can also help to control overfitting as they reduce the number of parameters of the network. There are several types of pooling such as max-pooling or average-pooling that output the maximum or average over a cluster. Figure 2.11 depicts an example of their effect.

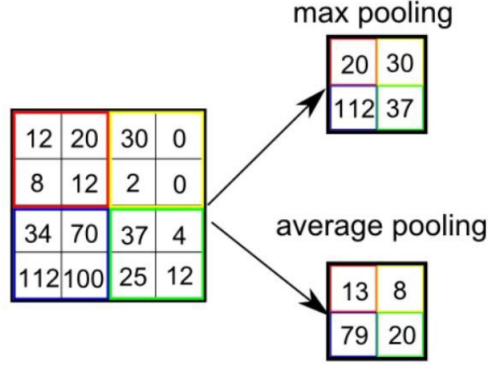


Figure 2.11: Different types of pooling from [40]. For a kernel size of 2, dimensions are reduced from 4x4 to 2x2.

At the end, **fully-connected layers** connect every neuron of one layer to every neuron of its next layer. They are expressed as

$$y = f(x \times w + b)$$

where f is an activation function, x the input, w the weights, and b the biases. They come at the end of the network, take an input volume and output a vector of length equal to the number of different labels. Before entering these layers, we flatten the input in a single vector.

The FC layers predict the correct labels by looking at the class that is the most correlated to a feature. Then, the output layer gives the final probability for each label.

Finally, a technique that helps training convergence of CNNs is **batch-normalization**. It helps by recentering and scaling the data inbetween layers and is written as

$$BN(x) = \gamma \frac{x - \mu_x}{\sqrt{\sigma_x^2 + \epsilon}} + \beta$$

with μ_x the mean value of the samples, σ_x the mean variance of the samples, γ a scaling factor and β a bias factor.

[22] states that batch-normalization is used to improve speed of training as it allows using higher learning rates.

2.4 Embedded Neural Networks

As said previously, WCE can be accompanied by computer-aided diagnosis. In the framework of this project, the CNN will be embedded in a microcontroller to diagnose beforehand in the capsule, and we will see how this is done in this section.

There are two main parts in embedded deep learning, which are training and inference. During training, power and real-time performance is not a concern. Then, we call inference the step after training, when the network is embedded and executes its task. In our case, we want inference to be embedded, as the first diagnosis will be done inside a capsule, and here we have new constraints that apply to the system.

To match this constraint difference and optimize the processes, training and inference are executed on separate processing platforms. Typically, as the training runs for a longer time on large data, it requires more powerful tools. Thus, it is executed offline on servers and GPUs.

Then, concerns such as reliability, latency, privacy, bandwidth, and transmission power lead to push inference on an embedded device. Indeed, some applications cannot rely on an internet connection, or require low latency. Other may have private data stored on an external device. In our case, we want to process data inside the device to filter what is sent and save transmission energy.

As going from a platform to the other may require some adaptations, embedding a network can be seen in three steps as in Fig. 2.12, which are training, format conversion, and inference. We will follow these steps in this work.

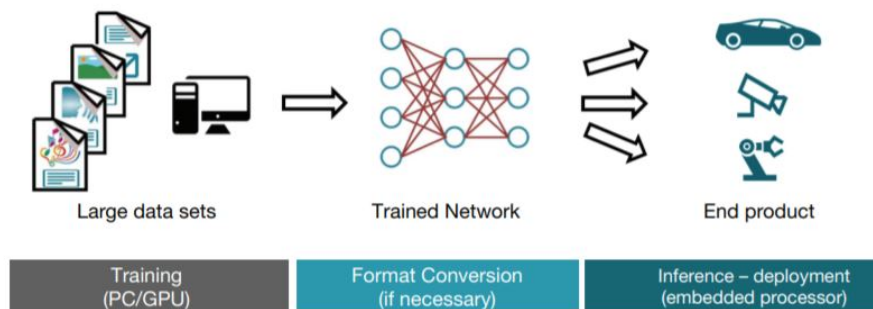


Figure 2.12: First, the training is executed on a desktop or a GPU. Then, the network format is converted to match the requirements of the deployment on an embedded processor. Image from [32].

In this section, we will see that embedded devices are challenged by microprocessors energy consumption. We will thus refer to the format conversion by talking about some techniques used to reduce this consumption, and specifically about network quantization. Then, in the same spirit, we will browse the different libraries that allow deploying a neural network on an embedded processor.

Paper [42] addresses the topic and introduces challenges for embedded deep inference as well as algorithmic and architectural techniques for energy efficiency. As explained, concerns such as user privacy, latency, and energy wasted on raw data transmission, being the main bottlenecks of offline inference, lead to push inference step from the cloud to embedded devices. As shown in Fig. 2.13(b), embedded inference presents a new drawback, which is the microprocessor energy consumption.

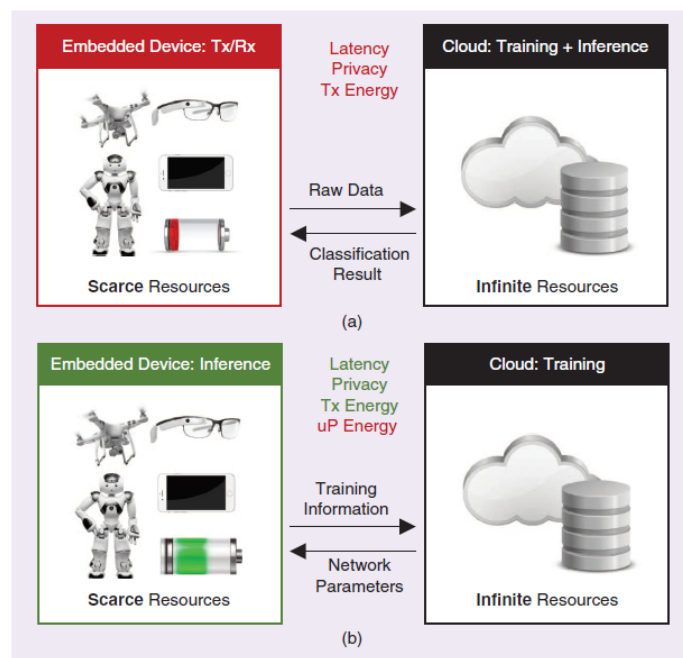


Figure 2.13: The push of DL inferences from (a) the cloud to (b) the embedded device solves the issues of latency, privacy, and transmission energy. However, the concern of microprocessor energy consumption appears.

Tx/Rx means transmitter/receiver and uP stands for microprocessor. Image from [42].

To cope with this, [42] proposes techniques for energy efficiency that consist in exploiting network structure, fault tolerance, and network sparsity to solve memory and computational bottlenecks. Figure 2.14 enumerates these techniques.

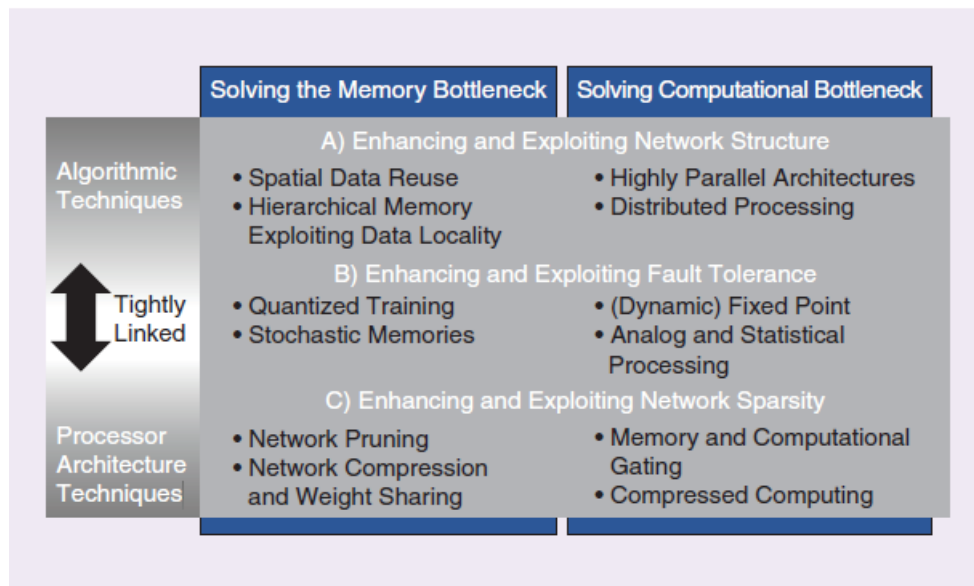


Figure 2.14: Algorithmic and architectural techniques for energy efficiency are used to solve both memory memory and computational bottlenecks. Image from [42].

These techniques exploit several aspects of neural networks. First, neural networks have special structures. By exploiting networks parameters such as layers input and output dimensions, number of channels, kernel dimensions, and pooling, one can operate spatial data reuse or parallelism. Then, neural networks are also known to be tolerant to approximations. This allows introducing concepts such as quantized training and fixed-point arithmetic. Last observation they make is that neural networks show large sparsity. We can use network pruning to remove elements that have a small impact.

For solving microprocessor energy bottleneck, we will focus on network quantization and fixed-point arithmetic as we can easily find frameworks that support them for our work. Let us now introduce the concept of network quantization.

2.4.1 Network quantization

A way to exploit fault tolerance is quantizing network parameters and activations. By doing so, one can significantly reduce the memory requirement of a network with a typically small counterpart in resulting performance.

In what follows, let us assume that values are expressed in 32 bits before quantization.

For example, [27] introduces Ternary Weight Network (TWN), which is a NN with weights constrained to +1, 0, and -1. They optimize a threshold-based function to minimize the euclidian distance between the full precision weights and the ternary ones. This way, as they end up with 2-bit values, they achieve up to $16\times$ model compression and require less multiplications than the full precision counterpart, as multiplication by +1 or -1 is the same as addition or subtraction. In addition, the advantage compared to the binary precision model that uses +1 and -1 weights resides in stronger expressiveness and in the 0 weights computations cancellation.

Also, network quantization can be done after training or during training, respectively called post-training quantization and quantization-aware training. Quantization-aware training trains the network by taking into account the fact that the values should be quantized. For example, if we process down to 8-bit quantization-aware training, the computations take place in 32-bit arithmetic but the values are then rounded to simulate the effect of 8-bit integer quantization. On the other hand, post-training quantization consists in training the network in full range 32-bit values and then quantizing it at the end. The latter might suffer from accuracy degradation as values could be clipped if all the values from different observations are not in the same range. Quantization-aware training prevents from clipping.

Paper [26] proves the better performance of quantization-aware training compared to post training quantization. To do so, it introduces an overview of techniques for quantizing CNNs with integer weights and activations. By exploring a wide variety of CNN architectures, they observe a 2% deviation for classification accuracies in the case of post-training 8-bit weights quantization. Post-training 8-bit quantization also reduces model sizes by a factor of 4 even when 8-bit arithmetic is not supported. They also observe a latency speedup of $2\times$ - $3\times$ on CPUs (Central Processing Unit) and DSPs (Digital Signal Processor). On the other hand, they observe a precision gap reduction of 1% at 8-bit precision for quantization-aware training. They also note a 2% to 10% accuracy loss range for 4-bit precision quantization. Smaller networks providing higher accuracy drop.

2.4.2 Convolutional Neural Network libraries

It requires an optimized implementation to build an energy efficient CNN with an improved runtime on microcontroller. Fortunately, libraries have been developed to construct such efficient embedded CNNs.

Even though libraries may not always perfectly translate a network from an offline platform to a microcontroller, they remain undeniably useful. Indeed, one can still either adopt a top-down approach by designing a model with layers already implemented in libraries, or try to copy a network by adding missing code to match its behaviour. In any case, available libraries are a good starting point to avoid implementing a network from scratch. In a time-constrained project such as a master thesis, using an existing library is an interesting option as it allows saving time.

CMSIS-NN [28] is an example of such a library. It is introduced as an efficient set of kernels that are optimized for NN applications on Arm Cortex-M processors, which are suited for intelligent IoT devices. Inference based on CMSIS-NN library enhances runtime by $4.6\times$ and energy efficiency by $4.9\times$.

The $4.6\times$ speed boost comes from the fact that ARM Cortex-M4 and M7 microcontrollers support specific SIMD (Single Instruction on Multiple Data) instructions such as 16-bit MAC (Multiply-and-Accumulate) instructions that accelerate matrix multiplication. This way, microcontrollers that support DSP instructions run faster with a special instruction set architecture.

The CMSIS-NN kernel code consists of two parts: **NNFunctions** and **NNSupportFunctions**. NNFunctions contains all the typical NN layers such as convolution, activation, pooling, and FC layers, while NNSupportFunctions contains utility functions for data conversion and activation function tables, which are used in NNFunctions. Figure 2.15 shows an overview of the neural network kernel structure.

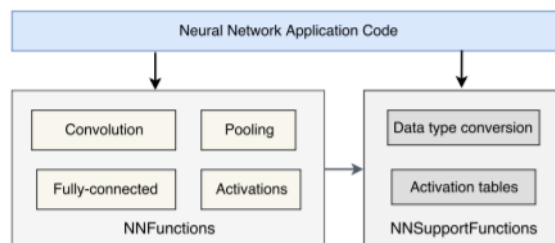


Figure 2.15: CMSIS-NN kernel structure from [28]. NNFunctions implements the different layer used in neural networks for inference, and it uses utilities from NNSupportFunctions in its codes.

We saw earlier that neural networks demonstrate fault tolerance. Thus, as inference does not require the 32-bit precision values that are generally used for training, the library adopts fixed-point quantization, and kernels allow working with 8-bit or 16-bit data. Also, convolution functions work with HWC (Height, Width, Channels) data layout as it requires less im2col (Image to Column) runtime according to [28].

Also, the API (Application Programming Interface) and the project are available at [29] and [7] respectively.

Finally, more specific purpose libraries exist or are emerging. For example, the recently released **CMix-NN** [14] library which is a memory-constrained device-oriented library, and therefore provides lower-precision integer arithmetic. By implementing 8, 4 and 2-bit weights and activations arithmetic, they provide an interesting inference library for low bitwidth quantized networks.

Unfortunately, the library implementation is still ongoing and does not provide all typical layer functions in their open-source solution, which is available at [13].

2.5 Related works

This section introduces works that are related to this thesis. It covers polyp classification or detection works as well as network embedded inference projects.

Earlier works usually first extract a region of interest (RoI) from features such as shape and texture, and then feed it to a classifier to detect candidate polyps. For instance, [34] investigates a strategy for polyp classification in WCE. They base on 2D and 3D information acquired to identify benign or malignant polyps. They first use 2D shape features to extract a RoI and then evaluate them with a boosting-based method using textural features. After this 2D pre-selection step, 3D parameters of remaining images are fed to a support vector machine (SVM) in order to classify the polyp as "benign" or "malignant".

This paper was released in 2013, and CNNs demonstrated to be very efficient for classification problems since then. Therefore, recent classification works rather use CNNs.

For example, the lately published paper [33] compares the performance of state-of-the-art classification models for polyp classification. They trained six models for the classification of hyperplastic and adenomatous polyps, and they test them on two different sets. Set-1 only contains cropped polyps, and set-2 contains cropped polyps and also about 55% background images. Performance comparison is shown in Fig. 2.16 and the metrics are described in Fig. 2.17.

Model	TP	TN	FP	FN	Ade (%)	Hyper (%)	Acc (%)	Err (%)	Pre-1 (%)	Pre-2 (%)	F1-1 (%)	F1-2 (%)	AUC (%)
VGG-19(set-1)	2424	1149	680	466	83.87	62.82	75.71	24.28	78.09	71.14	80.88	66.72	76.43
VGG-19(set-2)	2419	1346	483	471	83.70	73.59	79.78	20.21	83.35	74.07	83.52	73.83	84.80
VGG19-BN(set-1)	2071	1440	389	819	71.66	78.73	74.40	25.59	84.18	63.74	77.42	70.45	78.58
VGG19-BN(set-2)	2295	1345	484	595	79.41	73.53	77.13	22.86	82.58	69.32	80.96	71.37	82.20
ResNet50(set-1)	2350	1222	607	540	81.31	66.81	75.69	24.30	79.47	69.35	80.38	68.05	77.25
ResNet50(set-2)	2042	1305	524	848	70.65	71.35	70.92	29.07	79.57	60.61	74.85	65.54	76.27
DenseNet(set-1)	2246	1282	547	644	77.71	70.09	74.76	25.23	80.41	66.56	79.042	68.28	79.28
DenseNet(set-2)	2065	1306	523	825	71.45	71.40	71.43	28.56	79.79	61.28	75.39	65.95	78.65
SENet(set-1)	2230	1320	509	660	77.16	72.17	75.22	24.77	81.41	66.66	79.23	69.30	72.78
SENet(set-2)	2338	1138	691	552	80.89	62.21	73.65	26.34	77.18	62.21	78.99	64.67	82.05
MnasNet(set-1)	2239	1213	616	651	77.47	66.32	73.15	26.84	78.42	65.07	77.94	65.69	73.32
MnasNet(set-2)	2115	1242	587	775	73.18	67.90	71.13	28.86	78.27	61.57	75.64	64.58	77.11

Figure 2.16: Performance of state-of-the-art models on two different sets. VGG-19 outperforms other models for this polyp classification problem. Image from [33].

	Polyp classification
True Positive(TP)	Numbers of adenomatous polyps that are correctly classified
True Negative(TN)	Numbers of hyperplastic polyps that are correctly classified
False Positive(FP)	Numbers of hyperplastic polyps that are incorrectly misclassified as adenomatous
False Negative(FN)	Numbers of adenomatous polyps that are incorrectly classified as hyperplastic
Sensitivity	% of actual adenoma are correctly classified. Also termed as recall and accuracy of adenoma. $\frac{TP}{TP+FN} \times 100$
Specificity	% of actual hyperplastic are correctly classified. Also termed as recall and accuracy of hyperplastic. $\frac{TN}{TN+FP} \times 100$
Precision(Adenoma)	% of predicted adenoma that are truly adenoma. $\frac{TP}{TP+FP} \times 100$
Precision (Hyperplastic)	% of predicted hyperplastic that are truly hyperplastic. $\frac{TN}{TN+FP} \times 100$
Accuracy	Overall accuracy of both classes. $\frac{TP+TN}{TP+TN+FP+FN} \times 100$
F1-Score	Weighted average of precision and recall. $2 \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} \times 100$
Error	$\frac{1 - \text{Accuracy}}{100}$
ROC	Receiver operating characteristic curve
AUC	Area under the curve (of ROC)

Figure 2.17: Evaluation metrics used in the comparison. Precision and F1-score are computed for both classes. Image from [33].

They first notice that except for VGG models, all models present higher accuracy on set-1 than on set-2. Then, they also highlight the fact that VGG-19 outperforms other models for both sets. The specificity of the VGG Net [38] model architecture is that the depth of its layers are doubled between each convolutional layer, while max-pooling layers reduce the feature maps dimensions. We will keep that in mind when designing our own model.

This article gives us an idea of which architecture performs better for polyp classification. However, it does not cover the embedded operation that we target.

Fortunately, we can refer to work [15], with codes available at [5], which shows a tutorial on how to run a deep learning model on microcontroller with the CMSIS-NN library that we introduced previously. This example project allows to better visualize tensors dimensions and parameters that are fed to CMSIS-NN network functions.

This example is configured for uVision Simulator as well as the STM32F4 DISCOVERY board, and they explain how to run the CIFAR10 image classification on an ARM microcontroller. We cannot use the same simulator as theirs because its memory is limited to 32KB of code. Also, their application works on 32×32 pixels images while we target a higher scale, so we also need to revise the device selection. Still, we are going to learn from it to deploy our own application on CMSIS-NN.

2.6 Summary

In this chapter, we started by a brief introduction about colorectal cancer, and we investigated WCE by explaining how it works and by highlighting different capsules features.

Then, we explored CNNs by presenting how they learn and how they are tested. We also briefly discussed some difficulties that can occur during their training and how to cope with them. We finished this introduction about CNNs by reviewing the typical layers that compose them.

After exploring CNNs, we were interested in the reasons that push embedding neural networks on a third-party device as well as the challenges that implies such a deployment. We introduced different techniques to overcome these challenges by focusing on network quantization and libraries that suit the task.

Finally, we went through works related to the field of this thesis. We noticed that now they involve CNNs when it comes to classification problems. We also saw that efficiency vary from a model to another for polyp classification, and we ended by briefly introducing an example of neural network deployment on microcontroller.

Chapter 3

Database

Databases take a relevant part of machine learning as they represent the information that allows the model to learn and to be evaluated. A good model requires a good database. Databases are generally characterized by their size, which corresponds to the number of samples in the whole set, and the distribution of data according to the different classes it contains. Also, the more a database presents diversity, the more the model will be robust.

In order to build a model, one needs a dataset for each step. Typically, two distinct datasets are used for the training and the test. And then, the training dataset is split in two for training and validation.

The training dataset is used to fill the model parameters, such as weights connecting neurons. If the training set is not large enough, the training may suffer from overfitting as there will not be enough data to justify the number of parameters of the model. An intuitive way to describe what happens is that the network finishes learning from the data and loses its ability to predict by behaving as a table. Note that complex models require a larger dataset to avoid overfitting. Thus, in a first step, the model learns on a training set using supervised learning. The second step, the validation, uses a validation set that provides an unbiased evaluation of the trained model. This allows tuning the hyperparameters of the model as the validation dataset allows to keep track of the model generalization during training. Then, the third step is the testing, which applies on a test set that contains distinct data than used during training. It provides an evaluation of the final fit model.

To adopt supervised learning and have an idea of the performance of the model, we should compare expected outputs with the results for different inputs. In our case, the target outputs are called labels, and we call labeling the process of providing these correct answers.

This being introduced, this chapter will enumerate some available polyp databases and explain how our model database is created by using data augmentation and by labeling it.

3.1 Available databases

Databases are not always free access. In this section, we will present four databases that are accessible or for which access can be granted. Most of them were published during the MICCAI 2015 sub-challenge on polyp detection [11].

CVC-ColonDB [10] is a database of 15 video sequences of colonoscopy videos from 15 different studies, with a polyp in every sequence. RoIs are annotated for 300 frames amongst the whole sequences set and the database consists of 1200 different images divided in four groups, as shown in Fig. 3.1.

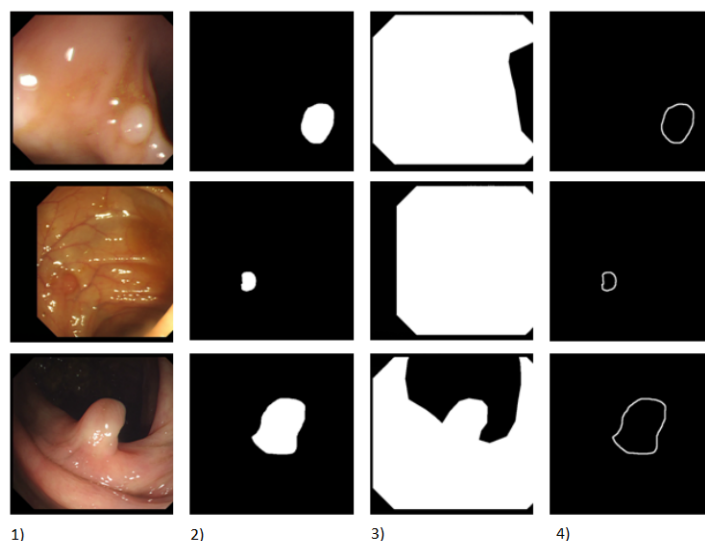


Figure 3.1: Examples of CVC-ColonDB image content. It contains 1) original images, 2) polyp masks, 3) non-informative region masks, and 4) contour of the polyp masks. Image from [19].

To use this database, one must register following instructions on this link [18].

CVC-ClinicDB [12] is a database of frames extracted from colonoscopy videos. The database is composed of 612 polyp-containing frames with original images and their corresponding ground-truth version, which are masking the region covered by a polyp in the image.

The database can be downloaded from this link [1].

Images from folder ‘Original’ are property of Hospital Clinic, Barcelona, Spain and images from folder ‘Ground Truth’ are property of Computer Vision Center, Barcelona, Spain.

ETIS-Larib Polyp DB [2] is a database of 196 polyp-containing images. Every original image is accompanied by its corresponding ground-truth version with the polyp region masked. ETIS-Larib Polyp DB database can be downloaded from this link [3]. Images from folder ‘Original’ are property of Lariboisière Hospital-APHP, France and ETIS laboratory, ENSEA, University of Cergy-Pontoise, France and images from folder ‘Ground Truth’ are property of ETIS laboratory, ENSEA, University of Cergy-Pontoise, France.

Last database, **ASU-Mayo Clinic Colonoscopy Video (c) Database** [4] is the first, largest and a constantly growing set of colonoscopy videos. As other databases, it contains original images and corresponding polyp masks. The ASU-Mayo Clinic Colonoscopy Video (c) Database is copyrighted. Using it requires contacting Prof. Jianming Liang at Arizona State University.

3.2 Database creation

The datasets of the project were created from the ETIS-Larib database because it is free access and that it has large images, allowing more crops on them. It initially contains 196 images of polyps with resolution 1225×966 . Of course, the 196 images of this database are not enough for teaching a polyp classification problem. However, the images having large dimensions, they can be cropped and still provide diversity in the new set that contains cropped images. Also, to enlarge the set and keep diversity, we make use of data augmentation on the initial set before cropping. We target a set of 10k training images, which should be suited content for the model to learn.

Another parameter to be taken into account is the distribution of positive and negative samples constituting the training set.

Let us now see how the dataset has been augmented.

3.2.1 Data augmentation

Depending on the application purpose, it is not always possible to collect large amount of image data. In the case of polyp databases, they are filled with images taken from inside a human body. So it is relatively tedious to collect these images. In order to cope with the lack of data, one can apply data augmentation and generate a bigger dataset based on a limited initial one.

Data augmentation is promising and can prevent from overfitting [36]. However, basic techniques remain an effective way to increase accuracy in classification tasks [44], so this work will not go beyond some traditional methods.

Let us take the original image of Fig. 3.2 as example to show the different modifications that were applied to the original dataset.

The first step consists in **removing the black edges** as done in Fig. 3.3. This results in images of resolution 800×966 pixels.

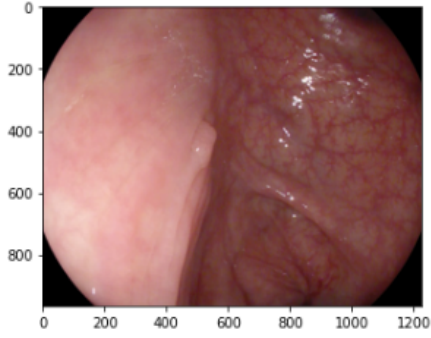


Figure 3.2: Example of original image from ETIS-Larib Polyp DB.

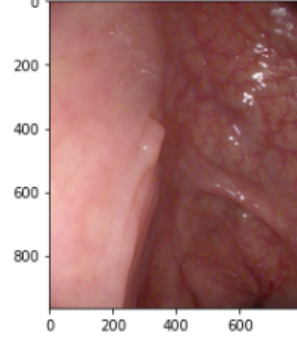


Figure 3.3: Image with black edges removed has dimensions 800×966

Rotations of 0, 90, 180, and 270 degrees are then applied in order to produce $4 \times$ more images. Result is shown in Fig. 3.4.

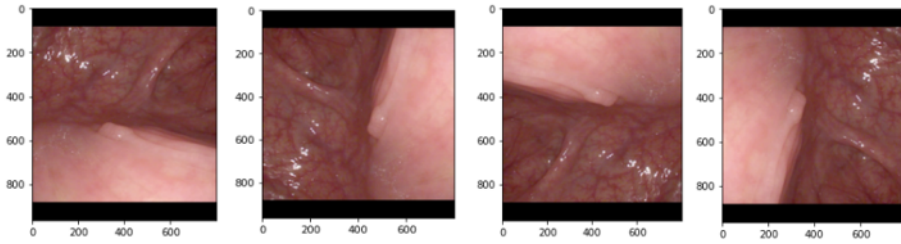


Figure 3.4: Rotated images allow increasing the number of images by $4 \times$.

Then, we apply **color enhancement** that adjusts the color balance of an image with factors 0.0, 0.5, 1.0, and 1.5, as shown in Fig. 3.5. The 0.0 enhancement factor gives a black and white image and 1.0 enhancement factor the original image. After this step, the number of images is multiplied by 16.

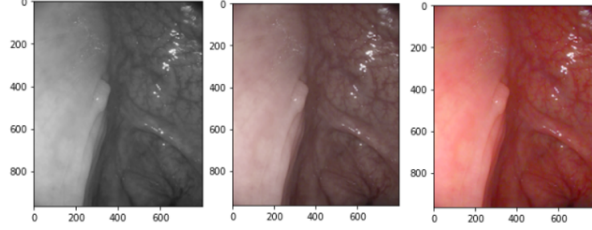


Figure 3.5: Color enhanced images. Left image has a color enhancement factor of 0.0, the middle one 0.5, and the right one 1.5.

Next step, the **brightness enhancement**, controls the brightness of an image with factors 0.0, 1.0, and 1.5 as shown in Fig. 3.6. Again this makes a set 3 times larger.

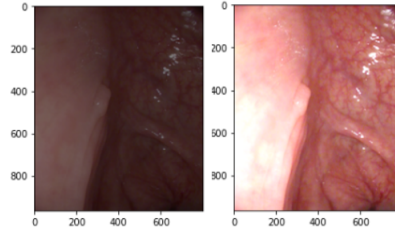


Figure 3.6: Images resulting from brightness enhancement with factors 0.0 for the left image and 1.5 for the one on the right.

Then, the images are randomly and uniformly **cropped**. Five crops of dimensions 250×250 are taken uniformly across the resulting rotated and enhanced images. This produces the well-sized images of the dataset as the ones depicted in Fig. 3.7.

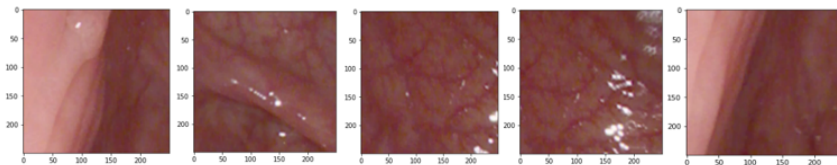


Figure 3.7: Example of cropped images

A step does not figure in the transformation process. After removing the black edges, the images were downsampled by only keeping one row and column out of two. This divides the images size by 4. Doing this presents the advantage of reducing the processing time of the different transformations, which helps as the number of images to be processed increases a lot.

Finally, the images are shuffled as the dataset will be split in three for training, validation, and test. At the end of the data augmentation process, the database size is $240\times$ larger. However, all images are not kept.

Indeed, from all the created images, a criterion was defined in order to control the set balance. To keep a distribution of polyp images higher than 50% of the dataset, polyp-labeled images are directly added to the final dataset while non-polyp ones are only added if the positive samples are higher than 50% of the dataset. This leads to a 68% positive images dataset.

After this, the final augmented dataset contains 25k images, 20k for training and validation, and 5k for testing. However, due to limited time and computing resources, 10k images will be used for training, 1000 for validation and 2500 for testing.

One may have noticed that even though the three sets contain different images, they all are generated from the same database. A better implementation would require the trained model to confront an external dataset for testing.

Also note that several transformations can be added such as sharpness and contrast enhancement or even mirroring in order to add diversity in the dataset.

3.2.2 Labeling

Once the new data have been created, the next step is to label its content. In order to decide whether an image contains a polyp or not, a criterion based on the size of the white pixels of a raw image polyp-mask has been set.

Appendix A shows the histogram of the original image polyps sizes and the one of the augmented dataset. The minimum polyp size of the original set is 1257 pixels, and as the images are downsampled, the polyp sizes are divided by 4. So the new minimum polyp size of the set becomes 314 pixels.

This limit is used in the labeling to decide whether a polyp in an image is large enough to be labeled as positive. When an image has been labeled, its label value is written in a CSV (Comma-Separated Values) file to complete the creation of the new database.

A typical labeling process is depicted in Fig. 3.8.

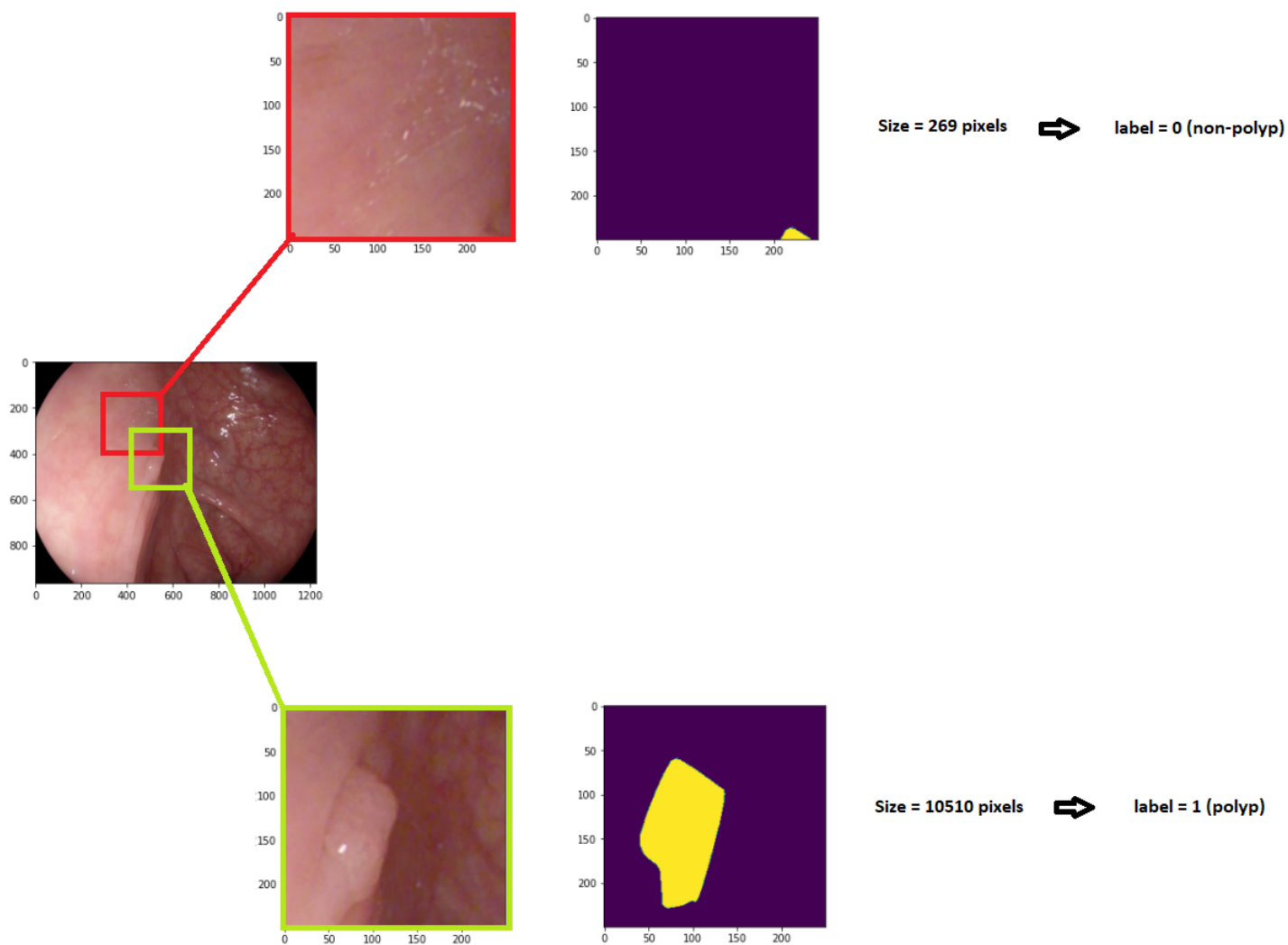


Figure 3.8: This shows the label decision based on raw version image for a cropped image. The sample on the top of the figure has a too small part of a polyp and is thus labeled as non-polyp. The other one is large enough and is thus labeled as polyp.

3.3 Summary

In this chapter, we first introduced the relevance of datasets in neural network applications. We said that they are characterized by their size and their distribution, and we also said that these characteristics can have an impact on the performance of a model.

Then, we introduced CVC-ColonDB, CVC-ClinicDB, ETIS-Larib Polyp DB, and ASU-Mayo Clinic Colonoscopy Video (c) Database, which are easily accessible polyp databases. Amongst these databases, we kept the ETIS-Larib Polyp DB one to build our own application dataset. To do so, we used data augmentation, and by applying several transformations on the original images, we could enlarge the dataset from 196 images of size 1225×966 pixels to 25k images of dimensions 250×250 .

Finally, we labeled the dataset that we created by using the raw images information. Indeed, based on raw polyp masks sizes, we could define a threshold to determine if a small part of a polyp should be considered as one.

Chapter 4

Model implementation and optimization on PyTorch

4.1 Google Colab

Colab is a product from Google Research that allows writing and executing Python code. It was quite a relevant choice for this work as it is especially fit for machine learning applications requiring small training data, and it provides free access to computing resources such as Google GPUs. The service works as Jupyter and stores notebooks in Google Drive. The dataset is also stored in Google Drive and can be accessed from a Colab notebook.

A drawback of the free version of Google Colab is that it limits its time and memory resources. Indeed, virtual machines (VMs) lifetime and GPU types available vary over time in order to ensure sharing resources fairly among all users. For example, GPUs are sometimes prioritized for users that have recently used less resources in Colab. The maximum VM lifetime is set to be around 12 hours and one will then have to wait before executing a code. Training a network may take a lot of time, so a way to cope with it would be to save the current value of the network parameters in the Drive before losing access to the VM to then resume training.

Another limit that should be taken into account while processing machine learning on Colab and especially if it implies working with large datasets is the 12GB total memory limitation.

4.2 PyTorch

There exist several Python frameworks for deep learning applications. The most common ones are Keras, TensorFlow and PyTorch. Each framework has different characteristics, and depending on what you want to develop, it may be useful to use one or the other. Following information come from [41], [8], and [35].

Keras is a neural network API in Python. It can run on top of CNTK, TensorFlow, and Theano. Keras is known for allowing fast experimentation with deep neural networks, for being user-friendly and for its easily understandable syntax. It only provides high-level computations making it work with another library for low-level. Also, it is designed for small dataset applications.

TensorFlow is an end-to-end deep learning framework comprising both high-level and low-level APIs allowing vast operations for DL models. Finally, it is rather used for its computation ability and for large dataset management.

Pytorch is a deep learning framework providing performance similar to TensorFlow in terms of computation speed and memory. However, it only supplies low-level API. Due to its flexibility, it can be adapted to save training time.

As TensorFlow and PyTorch are more suited for performance than Keras in terms of database size and computation speed, let us take a look at a comparison between both of them.

First, in terms of performance, both are similar in speed and in managing large datasets. Then, TensorFlow is more popular than PyTorch. It thus benefits from a bigger community support. However, PyTorch offers better debugging capabilities and flexibility allowed by its low-level API.

TensorFlow also provides better visualization which can also be helpful for debugging while PyTorch does not really focus on visualization.

Then, as TensorFlow provides a higher level of API, it allows more diversity in choices. For example, TensorFlow supports working with NHWC (Number of images in batch, Height, Width, Channels) and NCHW (Number of images in batch, Channels, Height, Width) model formats instead of NCHW only in PyTorch. This has its significance when a model needs to be extracted and translated in another language as it allows extracting data without needing to process it beforehand, to match the required format.

Despite TensorFlow providing more diversity, the choice was finally made to work with PyTorch.

Figure 4.1 shows a typical PyTorch workflow for implementing deep learning models.

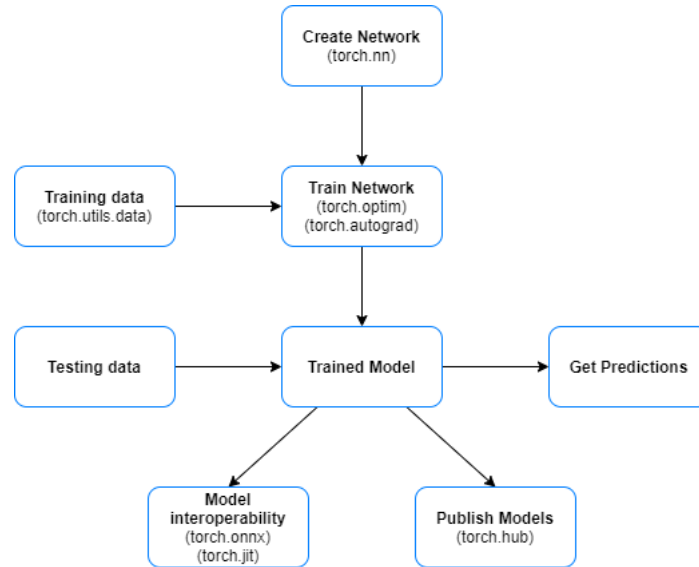


Figure 4.1: Typical PyTorch workflow. PyTorch offers tools for data processing, network creation, and network training. It also allows going further with a trained model by providing interoperability. Image adapted from [37]

First, **torch.nn** module contains utilities dedicated to creating neural networks. It provides the different layers as well as activation and loss functions. After this, for training, **torch.optim** provides the techniques for weights and biases updating, and **torch.autograd** provides classes and functions for automatic differentiation. **torch.utils.data** utility provides two important classes, which are **Dataset** and **DataLoader**. Dataset is used for custom dataset and allows to create a new dataset based on image data. DataLoader is used to load data from a Dataset in background to prepare it for use.

Also, PyTorch allows tensor computing with the ability for accelerated processing via GPUs matching with Google Colab on top of the **torch.Tensor** module. One can easily switch between CPU or GPU tensor such as CUDA (Compute Unified Device Architecture) for example.

Finally, **torch.onnx** and **torch.jit** respectively allow interoperability by exporting a trained network to ONNX (Open Neural Network Exchange) and by transitioning a PyTorch code to a TorchScript that can be run independently on C++ for example.

4.3 Model architecture

Since the applicative purpose is to perform inference on a microcontroller, there are two different strategies for designing it. First, the "bottom-up" approach would be to select the chip on which inference will be run to then design the model based on specified memory limitations.

Second, its "top-down" counterpart is to model the network by keeping in mind that it should not be too large and then selecting the device on which it could be executed.

Here, the second solution is adopted as we do not want to limit the model's architecture and therefore its performance by a device selection but rather find the device that matches a model that performs well.

Provided that the majority of WCE capsules depicted in Fig. 2.2 have image resolutions of 256×256 pixels, the model will be designed for 250×250 RGB images. So the model takes a 250×250 pixels RGB image, making 3 channels as input and outputs a vector of length 2 corresponding to the classes "*polyp*" and "*non-polyp*". In fact, the input image resolution of 250×250 was set to have a round figure. However, using 256×256 pixels images would have a more coherent choice as 256 is still a multiple of 2.

Also, the network outputs 2 classes, "*polyp*" and "*non-polyp*", but the truth is that it is a single "*polyp*" class problem. So the output should rather be a single neuron indicating if it is a polyp.

Then, the internal layers of the network are arranged as follows: It first has **3 sequences of CBA repetition followed by a max. pooling layer** called "MaxPool" in incoming figures. Note that CBA corresponds to the combination of a **convolution layer**, **batch normalization layer** and a **ReLU activation layer**. This convolution - batch normalization - activation combination is quite typical in CNNs, as mentioned in [45] that revises this repetition.

At the end of this repetition, there is an **average-pooling layer**, rescaling the data from $[C, H, W]$ to $[C, 1, 1]$. This layer is then followed by a **flatten layer** that makes the output dimensions match the input of the **first FC layer** that comes after. So dimensions are reshaped from $[C, 1, 1]$ to $[C]$ after the flattening. Finally, there is a **second FC layer** predicting the image class.

The first convolution layer has 32 output channels. This is an arbitrary choice based on typical value for first convolution output. Then, the number of output channels is doubled at every convolutional layer. VGG Net [38] does a similar thing as the number of filters doubles after each max. pooling layer. This reinforces the idea of shrinking spatial dimensions, but growing depth. This allows keeping the same amount of information but having it distributed differently. During design, we keep in mind that the final network should not be too memory consuming.

An image of the network is provided in Fig. 4.2 in order to have a better view of the different layers as well as their parameters and Fig. 4.3 also provides a view of the model layers sizes as well as the model's shape.

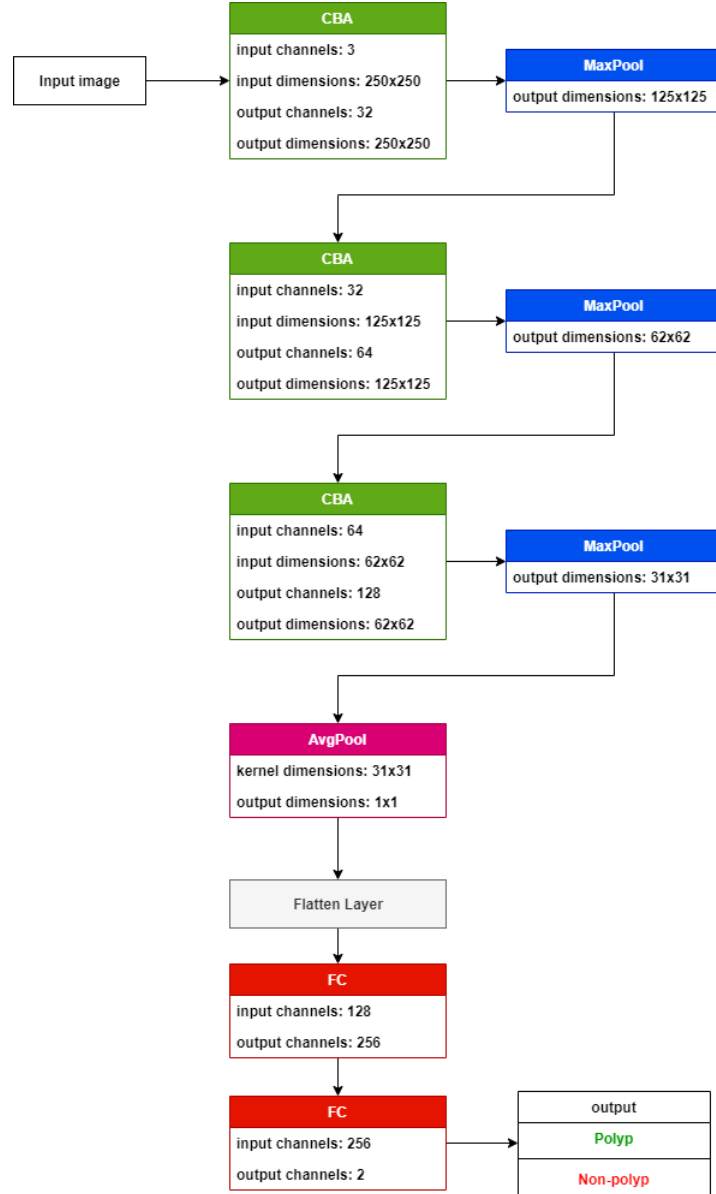


Figure 4.2: Imaging of the model's layers and their parameters. Convolution layers have padding of 3, kernel dimensions of 7×7 , and stride of 1. Max-Pooling layers are referred as MaxPool and have padding of 0, kernel dimensions of 2×2 , and stride of 2.

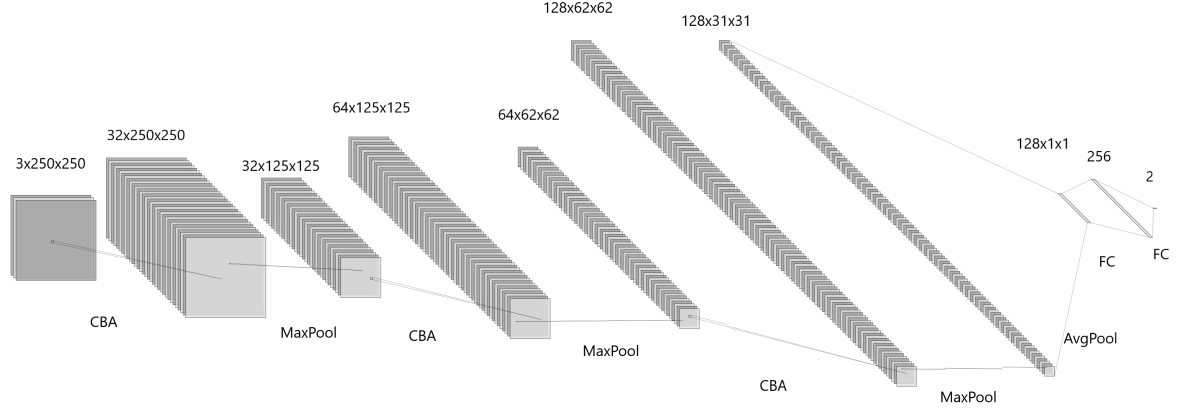


Figure 4.3: Network layers and sizes generated via NN-SVG.

In order to prevent the network from overfitting as depicted in Appendix B, **dropout layers** are added before both FC layers as they are known to work well in FC layers [20].

Finally, Table 4.1 shows output shapes and number of parameters for each layer as it summarizes the total number of parameters and the total size of the network for 32-bit values. The number of parameters are distributed as follows, 93.7% correspond to convolution layers, 6.2% to the linear layers, and 0.1% to the normalization layers. Making a total of 540,674 parameters with size 2.06MB.

One could also have noticed that there is a "Forward/backward pass size" element that consumes a lot of memory. This corresponds to the memory used for the forward pass, when the network makes predictions, and the backward pass, when the gradients are computed based on prediction errors. More information about how the 113.21 MB value is computed can be found in Appendix C. Of course, we want to avoid such amount of memory requirement in an embedded application. This issue will be tackled in next chapter.

Layer (type)		Output Shape	Param #
CBA	Conv2d	[32, 250, 250]	4,736
	BatchNorm2d	[32, 250, 250]	64
	ReLU	[32, 250, 250]	0
MaxPool2d		[32, 125, 125]	0
CBA	Conv2d	[64, 125, 125]	100,416
	BacthNorm2d	[64, 125, 125]	128
	ReLU	[64, 125, 125]	0
MaxPool2d		[64, 62, 62]	0
CBA	Conv2d	[128, 62, 62]	401,536
	BatchNorm2d	[128, 62, 62]	256
	ReLU	[128, 62, 62]	0
MaxPool2d		[128, 31, 31]	0
AvgPool2d		[128, 1, 1]	0
FlattenLayer		[128]	0
Dropout2d		[128]	0
Linear		[256]	33,024
Dropout2d		[256]	0
Linear		[2]	514
Total params: 540,674			
Trainable params: 640,674			
Non-trainable params: 0			
Input size (MB): 0.72			
Forward/backward pass size (MB): 113.21			
Params size (MB): 2.06			
Estimated Total Size (MB): 115.98			

Table 4.1: Model summary generated by the *summary* function from PyTorch. 93.7% of the network parameters come from convolution layers, 6.2% from linear layers and 0.1% correspond to normalization layers.

4.4 Hyperparameters tuning

Now that we have a model architecture, we need to train it to make it work. Getting optimal performance from training a network is challenging. Indeed, for the same network architecture, training results may vary depending on the hyperparameters given to the model. In order to optimize these results, one has to select the best hyperparameters.

4.4.1 Different methods for hyperparameters tuning

To do so, several methods exist and they all present different advantages as well as drawbacks. Here are some recurrent ones and their description from [9] and [23]:

Manual tuning: It consists in manually playing with the hyperparameters, allowing to have an intuition or an experience of how the model would react to the different parameters. This stops when the user is satisfied by the results of the training.

Even though this method provides an understanding of the model behavior with respect to the parameters, it is not optimal as it requires manual work and is based on intuition and thinking rather than properly checking all the possibilities.

Grid search: This is the method that will cover all prospective models. Grid search requires an exhaustive and relevant set for each parameter to iterate over them and evaluate every combinations of them.

The algorithm implies a consequent time complexity as it consists in brute force iterations over all parameters. However, the advantage of this method is that it is easy to implement. It is quite worth the time complexity as it makes sure not to miss the best set of parameters among the ones that are evaluated. This does not necessarily mean that an optimal solution is found as it will depend on how fine is the grid.

Random search [23]: This method starts as in grid search as it requires to prepare a set for every parameters to be tested. It then copes with the high complexity by only evaluating a predefined number of contenders taken randomly amongst the whole set and holds the best performing one.

The advantage here is that we can limit the runtime of the tuning but it implies not ensuring finding the optimal result.

Bayesian search: In Bayesian optimization, it starts from random and narrows the search space based on Bayesian approach. So after randomly evaluating some parameters, we go further for parameters around the most promising ones.

With this method, we could find an optimal solution but we might also be trapped at a local optimum.

Amongst the methods cited above, easier implementations are grid search and randomized search. Even though randomized search can provide more optimal results than grid search, grid search stays overall more effective as it can go through the whole search space. On the other hand, randomized search risks missing important points.

4.4.2 Grid search with reduced complexity

For this project, due to limited resources of training, grid search would not be the best choice unless we could find a way to alleviate the total searching procedure complexity.

What defines the time complexity of the training is **the training data size, the number of training epochs and the number of candidates to be trained**. The solution adopted to reduce it does not only lie in having relatively small and relevant sets of candidate parameters but also in reducing the training set during the optimization process. In this case, the problem consists in choosing the **optimizer, batch size, learning rate and dropout rate for both dropout layers** that give the best accuracy on the validation set.

SGD and Adam optimizers are selected as they are known for being effective. In general, batch size of 32 is a good starting point and a large batch size can lead to a degradation in the model's ability to generalize according to [24]. So the batch size will be iterated over [32, 64].

Then, typical learning rates are in [1e-1, 1e-5] and correspond to step size in the training process. Note that different learning rate sets are evaluated depending on which optimizer is being used as Adam optimizer can operate with lower learning rates than SGD as shown in Appendix D. Here are the parameters sets that are evaluated:

1. optimizer: { SGD, Adam }
2. batch size: { 32, 64 }
3. learning rate:
 - (a) SGD: { 1e-1, 3e-2, 1e-2, 3e-3, 1e-3, 3e-4, 1e-4, 3e-5 }
 - (b) Adam: { 1e-2, 3e-3, 1e-3, 3e-4, 1e-4, 3e-5, 1e-5, 3e-6 }

4. dropout1:{ 0.25, 0.5, 0.75 }

5. dropout2:{ 0.25, 0.5, 0.75 }

However, with this configuration, we still have a total of 288 models to be trained, which is not convenient for the resources available with Google Colab.

Another technique that has been adopted in order to keep reducing the complexity of the model optimization is **learning rate scheduling**.

It answers the problem of training the model just enough by lowering the learning rate while training to avoid overfitting. It is generally used to reduce magnitude of weight updates by decreasing the learning rate when generalization gap increases. In order to implement learning rate scheduling, the user first has to define a trigger on which a trained model would be considered underperforming and have its learning rate lowered. Then, a number of epochs of evaluation has to be set in order to look at the resulting model after every N epochs of training. This number of epochs should on the one hand be large enough to ensure the network has enough time to change between two observations while, on the other hand, we do not want it to be too large so that not much time is spent on training the network in a wrong direction.

Note that learning rate scheduling method relies on detecting when the network performance starts to decrease and when it overfits. However, as explained in background, it is not an easy task to detect it. Sometimes, the model metrics can oscillate and produce poorer results when being observed without meaning that it is on a decline. This can be fixed by using a moving average rather than only looking at a fixed-point.

Another issue is that as we are not training the models with the full and large dataset in a first step, a gap between training and validation results may appear at convergence due to low number of samples.

To cope with this, one should set softer constraints on the maximum error gap between training and validation metrics. In this work, the trigger is set to lower learning rates when either a 15% error margin is exceeded between one of the training metrics and its corresponding validation metrics or if the training loss did not decrease since last evaluation. In which case the training either converged or overfitted. The evaluation epoch step is set to 25 epochs.

Of course, the error margin and epoch step should be considered as hyperparameters in order to select the best ones. In fact, these parameters are selected based on how early the models learning rates are decreasing. We want to make sure that the network's training has benefited from every learning rate decay. To illustrate the meaning of benefiting from a learning rate, one can imagine a network for which learning rate scheduling is triggered and the learning rate decreased while it would not be the case if we had let it run for a few more epochs. As "learning steps" are

getting smaller when learning rate decreases, we do not want to skip a learning rate earlier than when the network really stops improving. Here, grid search has not been implemented on them, but rather manual tuning. So the choices are based on experience and trial and error. The grid search algorithm would otherwise not benefit from learning rate scheduling in terms of time complexity reduction. Indeed, in addition to preventing overfitting, learning rate scheduling provides complexity reduction. In this context, it allows not to iterate over learning rates and thus reduces the number of iterations of the grid search. Thus, instead of having several models trained for each learning rate, only one model is trained and its learning rate is modified during training.

A pseudo-code translating the learning rate scheduling implementation that was used is shown in Algorithm 1

Algorithm 1 Learning rate scheduling

```

1: procedure OVERFITS(margin)                                ▷ checks that the model does not overfit
2:   return  $|train\_metrics - validation\_metrics| < margin$ 
3: end procedure
4: procedure IMPROVED                                          ▷ Based on loss
5:   return  $current\_loss < previous\_loss$ 
6: end procedure
7: procedure TRAIN(learning_rate_limit)
8:   for batch size in [32, 64] do
9:     for dropout1 in [0.25, 0.5, 0.75] do
10:      for dropout2 in [0.25, 0.5, 0.75] do
11:         $learning\_rate = 0.1$  for SGD or 0.01 for Adam
12:        while  $learning\_rate > learning\_rate\_limit$  do
13:          Train model for 25 epochs
14:          if OVERFITS or not IMPROVED then
15:             $learning\_rate \leftarrow learning\_rate / 3$ 
16:          end if
17:        end while
18:      end for
19:    end for
20:  end for
21: end procedure

```

First procedure, *OVERFITS* checks if the gap between training metrics and validation ones are above a margin. Remember that this error margin has been set to 15%. Then *IMPROVED* verifies that generalization error decreases during training by verifying that the training loss decreases at each observation. Again, remember that the performance is monitored every 25 epochs for training. Finally, procedure *TRAIN* trains the model following learning rate scheduling method.

For all candidate models, the learning rate starts from an initial value of 0.1 or 0.01 depending on which optimizer is in use and it trains the model during 25 epochs steps. If the error gap between validation and training is higher than 15% or if the training loss did not decrease during last 25 epochs, the learning rate is

decreased and last 25 epochs of training are discarded. In this case, the learning rate is decreased and the model resumes training from previous state. Otherwise, the current state of the network is saved and it keeps going until a learning rate limit of 10^{-5} or 10^{-6} is reached depending on optimizer.

This way, models are trained until reaching the minimum learning rate allocated depending on the optimizer used. As we do not really know the impact of the optimizer on the behavior a model would have by using the full dataset, the best performing models using Adam and SGD optimizers are kept for full training on the 10k images training set.

At the end of the grid search, the model parameters that are kept for the SGD optimizer are batch size of 32 and dropout probabilities of 50%. For the Adam optimizer, the better performing model is the one with batch size of 64 and dropout probabilities of 25%. Figure 4.4 shows the selection training metrics for all the models.

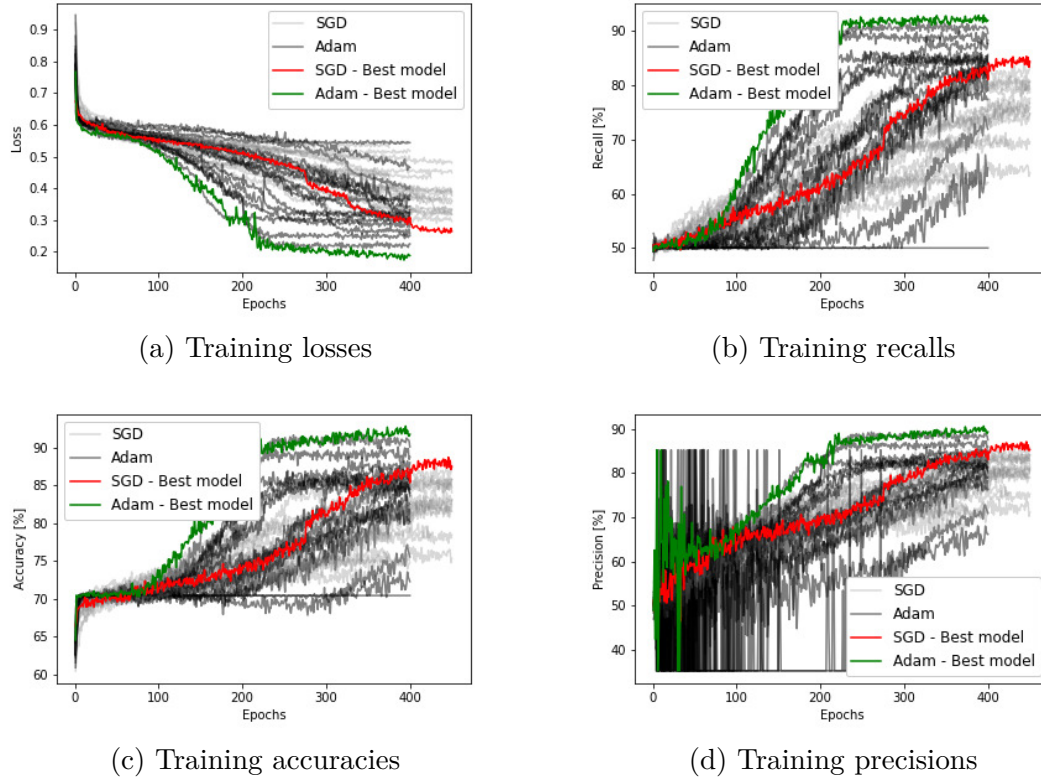
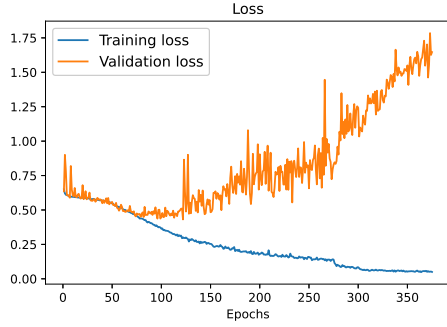
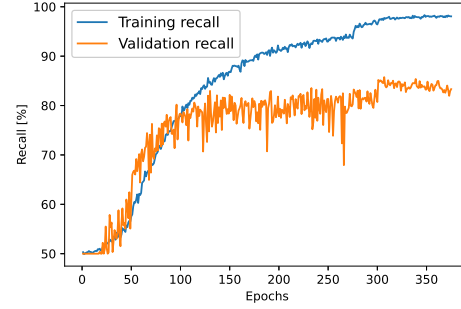


Figure 4.4: Training metrics resulting from the learning rate scheduling algorithm. Best parameterized models are drawn in red for SGD optimizer and green for Adam optimizer. Transparent black and gray curves represent non-optimal parameters models for Adam and SGD optimizers respectively. They are thus discarded and not further investigated.

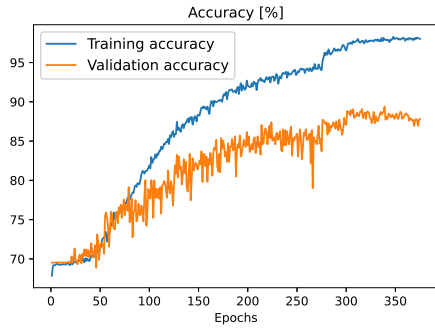
Finally, by training the best "hyperparametrized" models, we end up observing in Fig. 4.5 for the Adam optimizer model and Fig. 4.7 for the SGD optimizer one that the best-performing model is the one with the **SGD optimizer, batch size of 32 and dropout probabilities of 50%**.



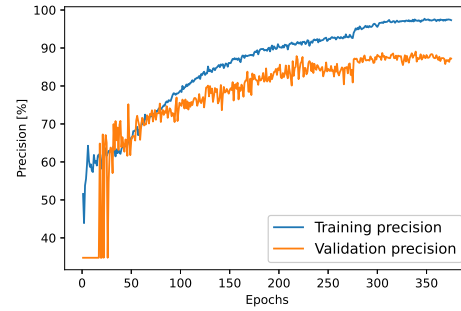
(a) Training and validation losses



(b) Training and validation recalls



(c) Training and validation accuracies



(d) Training and validation precisions

Figure 4.5: Training and validation metrics resulting from the full training of the model with **Adam optimizer, batch size of 64 and dropout probabilities of 25%**

Indeed, as the model using an Adam optimizer seemed to be better performing during early training, its validation metrics appear to converge earlier and loose the track of the training metrics. At convergence, its training metrics are around 99% while the validation ones stabilize around 90%.

In fact, what happens here is that the model seems to overfit. This is marked by the validation loss increase while the training one keeps decreasing and this might be due to small dropouts rates. First round selection did not predict it as validation metrics were following training ones and as it does not check validation loss. Fig. 4.6 shows validation loss of the first round selection and one can see that it does not follow training loss during the process. The learning rate scheduling algorithm could be improved by detecting this.

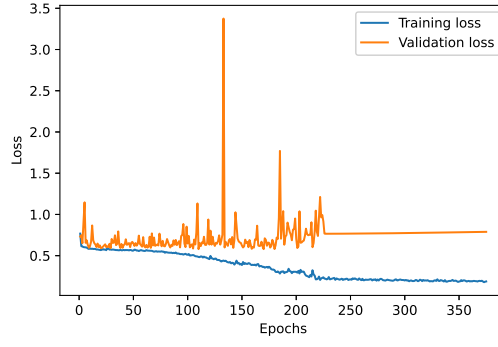
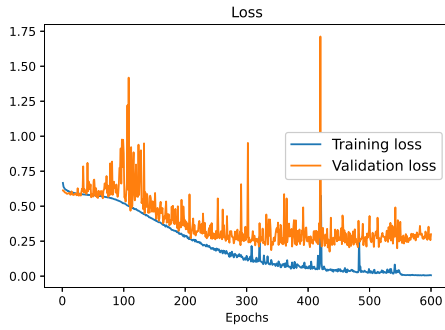
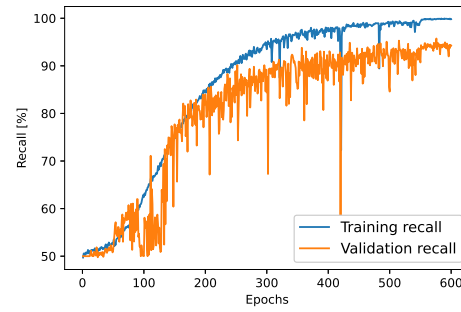


Figure 4.6: Loss evolution for Adam optimizer, batch size of 64 and dropout probabilities 25%. Validation loss increases very slightly while training loss decreases. This is a typical behavior of an overfitting network.

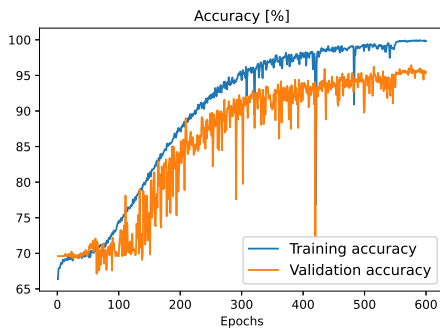
On the other hand, good news it that the model with the SGD optimizer provides generally steadier results as validation metrics follow training ones. It results in **validation recall, precision and accuracy around 95% when training converges at 99%**.



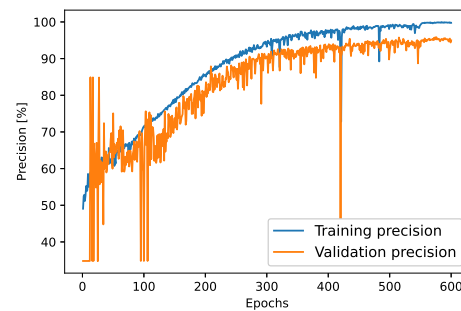
(a) Training and validation losses



(b) Training and validation recalls



(c) Training and validation accuracies



(d) Training and validation precisions

Figure 4.7: Training and validation metrics resulting from the full training of the model with **SGD optimizer, batch size of 32 and dropout probabilities of 50%**.

Note however that we can observe either some peak errors or some sudden improvements. These can be explained by learning rate changes that occur during learning rate scheduling training. Indeed, the models have been trained following the same learning rate scheduling algorithm as described in Algorithm 1. The only difference is the number of images in the training and validation datasets as they contain 4 times more samples.

In this context, we do not want to miss any polyp. In other words, we want to minimize the false negatives and thus maximize the recall. Remember, the recall is expressed as $\frac{TP}{TP+FN}$.

To do so, another parameter that comes into account is the use of **weighted loss cross-entropy function**. The function is implemented as follows in PyTorch:

$$L_c(x) = w_c \left(-x_c + \log \left(\sum_j \exp(x_j) \right) \right)$$

L_c represents the loss for a given class c , and w_c the weight of class c . The losses are averaged accross observations for each minibatch, giving

$$L = \frac{\sum_{i=1}^N L_{c_i}[i]}{\sum_{i=1}^N w_{c_i}}$$

L is the averaged loss and c_i denotes the i^{th} class.

This allows the loss function to promote one class over others during decision. In order to improve the model towards maximizing the recall, the class weights are compared to the unweighted loss function with distributions **class weights equal to [0.75, 0.25] and [0.25, 0.75]**. Class at first position is the "*polyp*" one, so [0.75, 0.25] promotes polyps decision while [0.25, 0.75] does the opposite. The impact on the training recall evolution can be observed in Fig. 4.8 and final results with the **testing set** in Table 4.2. As a reminder, the testing set is composed of 1250 images with 850 positive samples and 400 negative ones.

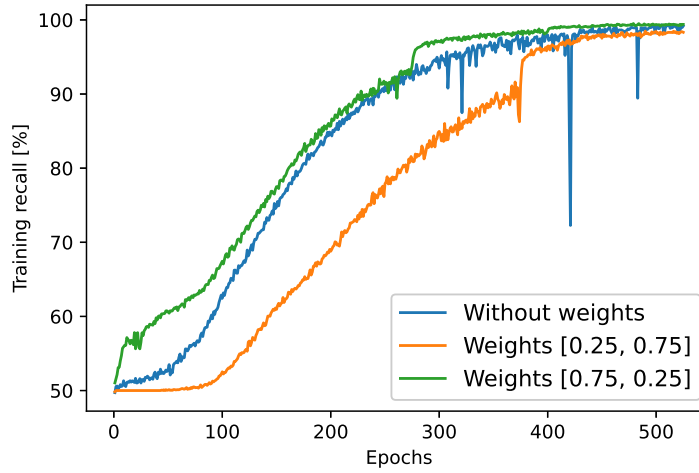


Figure 4.8: Recall evolution for SGD optimizer, batch size of 32 and dropout probabilities of 50%. Weights [0.75, 0.25] give 93% recall, weights [0.25, 0.75] give 91% recall and unweighted one gives 95% recall.

Class weights \ Final test metric	Loss	Accuracy [%]	Precision [%]	Recall [%]
[0.5, 0.5] unweighted	0.23	96	96	95
[0.75, 0.25]	0.26	94	92	93
[0.25, 0.75]	0.24	93	93	91

Table 4.2: Final results on testing set. Note that these results are taken from the model after training until convergence. So the final results are not necessarily the best ones observed during the whole system learning.

Finally, the non-weighted loss function model is kept as it is the one performing better for all metrics on the testing set.

One can observe that when promoting "*polyp*" class, the recall stands higher than the one that promotes the "*non-polyp*" class. This was expected as less polyps are classified as non-polyp when the polyp class is promoted and this decreases the false negatives and thus increases the recall. However, the recall is only slightly higher than the one of the non-weighted loss function model, and it even gets lower at the end. When no class is promoted, the model still has a bigger recall and this can be explained by the fact that the learning rate scheduling algorithm does not run the same scenario for all models. For example, it is possible that this lower recall is due to a learning rate change that occurred either earlier or later in one

model's training than in the other.

As in this case the training process is non-deterministic, the correct way that allows to draw conclusions would be to run each training several times, to extract mean values and standard deviations. Otherwise, we could be looking at outliers.

4.5 Summary

In this chapter, Google Colab was selected to run the model for its neural networks related facilities. However, due to memory and computing limitations, its free version is not fit for large dataset applications and we had to cope with it by trying to minimize the different algorithms timing complexity.

Then two frameworks, Tensorflow and PyTorch stood out from others, and Pytorch was selected to implement the model.

After selecting the tools and frameworks, the model has been architected and introduced. Some parameters still needed to be set and this was done through hyperparameters tuning. For this, the method used was grid search accompanied by learning rate scheduling. Learning rate scheduling was brought for its contribution as a training algorithm and also for reducing the time complexity of the search by avoiding iterating over learning rates in the grid itself.

The different candidate models were trained on a reduced training set and the best performing ones were selected for a full-set training.

A last parameter that came into account was the class weights distribution in the loss function and after evaluating its effect, a final model has been selected for inference. This model provides 95% recall and 96% accuracy on our data.

Chapter 5

Inference on microcontroller

This chapter introduces the way the network is implemented on a microcontroller, how the relevant data is extracted to deploy the model, and how we select the device on which the network will be deployed.

First, for the network implementation, we exploit the CMSIS-NN library because it provides the functions required to build our network in C. It also allows saving time by not coding the network from scratch, but only bringing minor changes to the source code instead.

Then, as the data are expressed in 32-bit values, we saw in previous chapter that the parameters are heavy in memory. That is the reason why we quantize the values down to 8 bits. This matches the 8-bit fixed-point arithmetic used by CMSIS-NN and allows to divide the total memory requirement by 4. Indeed, as seen in background, quantization is a straightforward and systematic optimization step when it comes to memory-hungry applications.

Also, we noticed in previous chapter that forward pass parameters occupy a lot of memory. We will see how to reduce this memory requirement and save RAM (Random Access Memory) by coding cleverly.

Finally, we will select a device that matches the application needs based on the memory information we will have gathered so far, and we will predict the final result performance.

5.1 Implementation on CMSIS-NN

CMSIS-NN has already been introduced in background, so let us start by talking about the fixed-point arithmetic that it uses.

CMSIS-NN functions work with buffers of 8-bit values as layers inputs and outputs. Then, in order to represent binary fractional values, CMSIS-NN makes use of fixed-point arithmetic, also known as **Q number format**. These numbers are

written $Qm.n$, where Q designates that the number is in Q format, m is the two's complement integer portion of the number, and n is the number of bits used to designate the fractional portion of the number. $Qm.n$ numbers lie in range $[-2^m, 2^m - 2^{-n}]$ and have a resolution of 2^{-n} .

Then, as all values are represented as 8-bit integers, CMSIS-NN uses a system of shift to adjust the difference between quantized values that do not have the same Q format. For example, if we take x an 8-bit integer value that is accompanied by a right shift of n , this indicates that x is a $Q(7-n).n$ value.

This way, functions take as argument ***bias_shift***, the amount of left-shift for the biases, and ***out_shift***, the amount of right-shift for the outputs.

For example, if a convolution function has an input of format $Q5.2$, weights in $Q3.4$, and bias in $Q4.3$, the multiplication of input and weights gives a $Q25.6$ 32-bit value. To match the fractional part precision when adding the bias, the bias should be left-shifted by $bias_shift = 3$, making it a $Q25.6$ value when represented in 32 bits. Then, if the next layer requires an input in $Q5.2$, the result after addition of bias should be right-shifted by $out_shift = 4$, and the 7 LSBs (Least Significant Bit) are kept to represent the output in 8 bits. The eighth bit being the sign bit. This being said, we can go to the next step, which consists in converting a network from PyTorch to CMSIS-NN.

5.1.1 From PyTorch to CMSIS-NN

This subsection introduces the different adaptations required in order to fully translate the network from PyTorch to CMSIS-NN.

First, as CMSIS-NN works with tables, the data are already flattened, and there is thus no need to implement a flatten layer function before fully-connected ones as in the PyTorch network. Also, dropout layers do only work during training, when weights are not frozen. So these two types of layers do not figure in our CMSIS-NN implementation.

In addition, another layer that is not implemented by CMSIS-NN is the batch-normalization layer. However, it is still possible to include its effect in the final network in C. To do so, we express the convolutional layers as a fusion of convolution and batch-normalization by redefining the weights as

$$W = W_{BN} \times W_{conv}$$

and biases as

$$b = W_{BN} \times b_{conv} + b_{BN}$$

where W_{BN} and W_{conv} respectively are the batch-normalization and convolution weights, and b_{BN} and b_{conv} are the batch-normalization and convolution biases.

Also, as volumes are expressed in NHWC format in CMSIS-NN and NCHW in PyTorch, kernels dimensions should also be rearranged from (1,2,3,4) to (1,4,3,2) order before exporting them.

Then, the weights and biases of the filters should be quantized to match the Q format from CMSIS-NN, and for this, we use post-training quantization because quantization-aware training would have slowed down the network training. We need to find the quantization scale that minimizes the approximation error for each layer parameters. The scale refers to the resolution of the quantized value. A scale of 2^{-6} means that the 8-bit quantized value is expressed in Q1.6 format.

The algorithm to determine the quantization scales of the weights and biases for the network after training is shown in Algorithm 2.

It consists in finding the set of scales that minimize the sum of differences between 32-bit values and 8-bit quantized ones.

Algorithm 2 First-order error minimizing quantization

```

1: procedure BINARY_QUANTIZATION(table) ▷ Returns table quantized with the scale that minimizes the first
   order error
2:   scales = [ $2^{-7}$ ,  $2^{-6}$ ,  $2^{-5}$ ,  $2^{-4}$ ,  $2^{-3}$ ,  $2^{-2}$ ,  $2^{-1}$ ]
3:   scale_quant =  $2^0$ 
4:   min_diff =  $\sum_i |table[i] - quantize(scale\_quant, table)[i]|$ 
5:   for scale in scales do
6:     diff =  $\sum_i |table[i] - quantize(scale, table)[i]|$  ▷ quantize(scale, table) returns table quantized with
       scale
7:     if diff < min_diff then
8:       min_diff  $\leftarrow$  diff
9:       scale_quant  $\leftarrow$  scale
10:    end if
11:  end for
12:  return quantize(scale_quant, table)
13: end procedure

```

We end up with the Q formats in Table 5.1 for the layers weights and biases.

Layer	Weights	Biases
1st convolution	Q1.6	Q3.4
2nd convolution	Q0.7	Q2.5
3rd convolution	Q0.7	Q1.6
1st FC	Q0.7	Q0.7
2nd FC	Q0.7	Q0.7

Table 5.1: Best Q formats for the weights and biases of layers that store parameters.

Finally, CMSIS-NN requires knowing in advance the Q format of the output for each layer and it implies computing the scales that will match the best the general outputs of the network. Indeed, the output of layers are not fixed as for parameters, so there is no set of quantization scales that minimizes the error for every possible output. Algorithm 2 can thus not be used in this case. However, the other option here is to search for the scale that gives the best performance during test when targeting the recall. Finding such a set is implemented as a brute grid search over all the possible quantizing scales on the already trained model. The layers for which the scale between their input and output may vary are the convolutional layers, the FC layers, and the average-pooling layer. Max-pooling layers and ReLU layers keep the same scale as their input for their output. This makes a total of 6 scales to determine within $[2^{-7}, 2^{-6}, 2^{-5}, 2^{-4}, 2^{-3}, 2^{-2}, 2^{-1}, 1]$. The best set is then extracted after iterating over these scales, and their Q formats are shown in Table 5.2.

Layer	Output Q format
input	Q0.7
1st convolution	Q3.4
2nd convolution	Q1.6
3rd convolution	Q2.5
average-pooling	Q0.7
1st FC	Q2.5
2nd FC	Q5.2

Table 5.2: Q formats for each layer output that maximize the recall on the testing set.

Finally, by testing the model with quantized weights and biases, and by quantizing the output of layers with the scales found, we obtain the metrics shown in Table 5.3 for our testing dataset.

Network\Metric	Loss	Accuracy [%]	Precision [%]	Recall [%]
Full-precision network	0.23	96	96	95
8-bit quantized network	0.42	90	88	92

Table 5.3: Testing metrics comparison between the full-precision network and the 8-bit quantized one. We minimized the recall gap as it was our target.

The scales were selected to maximize the recall, we thus notice a 3% recall deviation, and a 6% and 8% gap for accuracy and precision between the 32-bit values network and the 8-bit quantized one on the testing set.

5.2 Code architecture

The way a program is coded may save RAM memory if it is well thought. In our case, we noticed in the previous chapter that forward pass consumes a lot of memory, and to cope with it, two buffers are created and reused among layers, as done in [5]. This allows the forward pass data to switch between these two buffers and overwrite what is not used anymore. These two buffers are stored in ***scratch_buffer*** which contains the intermediate layer outputs. We also have ***col_buffer*** which stores the im2col output for convolutional layers.

According to [16], the formula to predict the required size for ***col_buffer*** across all convolutional layers is

$$2 \times 2 \times (\text{number of convolution filters}) \times (\text{kernel width}) \times (\text{kernel height})$$

and in our case, the maximum number of filters is 128 and the kernels dimensions are 7x7. We thus have a size of 24.5kB for the ***col_buffer***.

Then, Fig. 5.1 shows how to determine the size required for ***scratch_buffer***. The figure also shows how the different layers switch between the two buffers ***img_buffer1*** and ***img_buffer2***. Note that ReLU layers do not figure in the diagram as they do not switch between buffers. Instead, they directly overwrite their input because they are not source of resizing.

We figured how the maximum sizes required for ***img_buffer1*** and ***img_buffer2*** among all the layers is determined, and ***scratch_buffer*** size is then defined by joining them. We thus need 2.385MB for storing layer outputs.

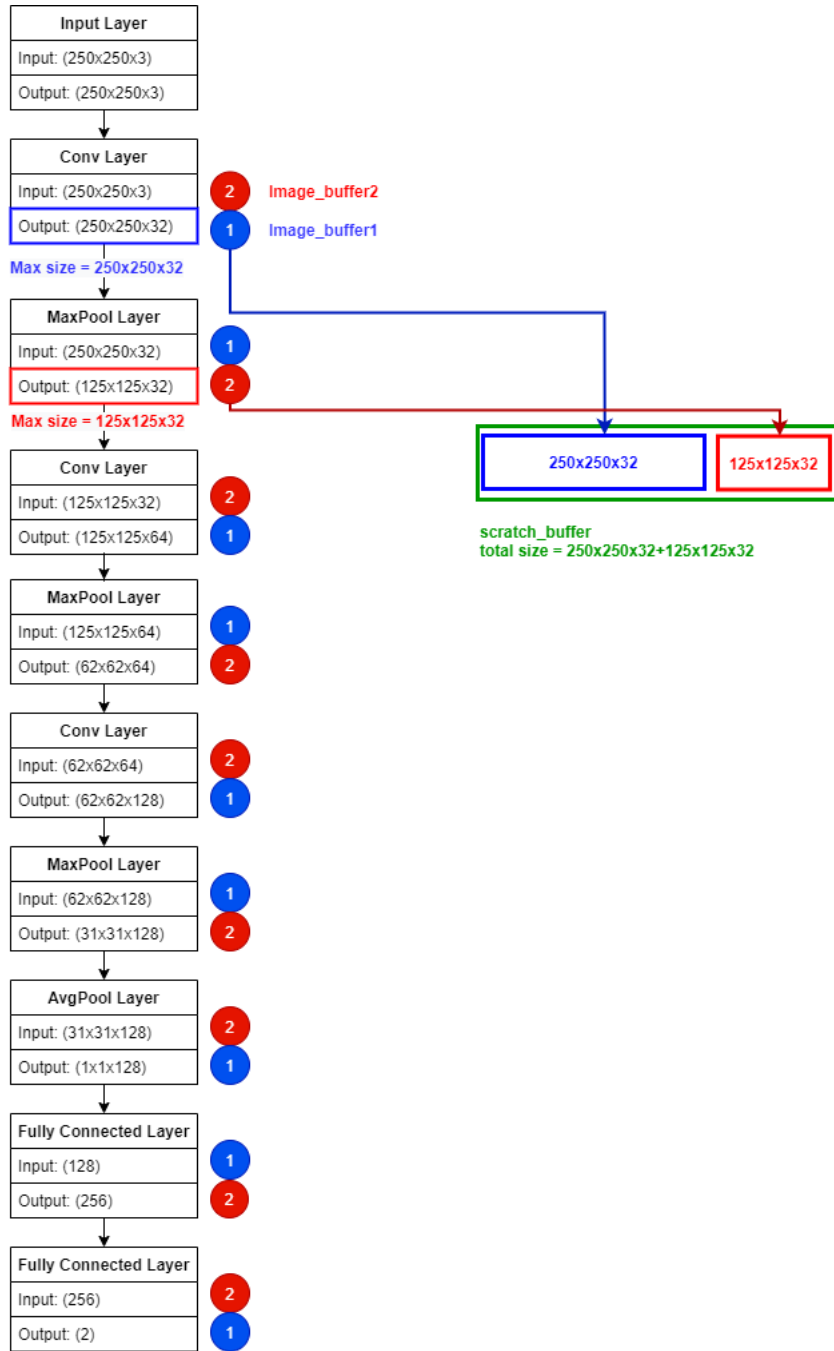


Figure 5.1: The code architecture allows to save forward pass memory by reusing two tables to store it. This way, layers outputs switch between `image_buffer1` and `image_buffer2` depending on which one can be overwritten. This figure shows how these buffers sizes are determined.

5.3 Device selection

Once the network is implemented in CMSIS-NN and it is fed with its parameters, the next step is to select the device on which inference will be run.

We need to predict the memory requirements of the network as a first step. Table 5.4 shows the memory requirements for each layer weights and biases.

Layer	Parameter	Dimensions	Size[kB]
Input image		3x250x250	183.105
1st convolution	weights	32x3x7x7	4.593
	biases	32	0.031
2nd convolution	weights	64x32x7x7	98
	biases	64	0.062
3rd convolution	weights	128x64x7x7	392
	biases	128	0.125
1st FC	weights	256x128	32
	biases	256	0.25
2nd FC	weights	2x256	0.5
	biases	2	0.002

Table 5.4: Sizes of each layer weights and biases in kB. As expected from the previous chapter, convolutaional layers require more memory than fully-connected ones.

As the input image should not be taken into account in the ROM (Read-Only Memory) memory, this makes a total **ROM memory requirement of 527.563 kB**. Then, elements of RAM memory have been reviewed in the previous section. By summing the *scratch_buffer* and *col_buffer* sizes, we obtain a total of **2.41MB of RAM required**.

Now that we know the total size of the network parameters, we can select on which device the inference will be run. The criteria are a ROM capacity larger than 527.563kB and a RAM storage larger than 2.41MB.

Table 5.5 shows all the devices taken from the ARM database that match the application memory constraints cited above.

Supplier	Device	Core	ROM	RAM	Frequency [MHz]
NXP	MIMXRT685S	ARM Cortex-M33	896kB	18080kB	300
NXP	MIMXRT633S	ARM Cortex-M33	896kB	7840kB	300
Mediatek	MT2523	ARM Cortex-M4	4MB	4224kB	208
ARM	SSE-200-MPS3	ARM Cortex-M33	8MB	4MB	50
ARM	IOTKit_CM33/23	ARM Cortex-M33/23	4MB	4MB	25
ARM	CMSDK_CM7/4	ARM Cortex-M7/4	4MB	4MB	25
ARM	ARMCA9/7/5	ARM Cortex-A9/7/5	1056MB	128MB	12
ARM	VE_CA9/7/5	ARM Cortex-A9/7/5	1056MB	128MB	12

Table 5.5: Devices matching the network memory constraints. High frequency is required for real-time applications, so the ARM devices are not fit for this.

The application requires a device that can operate at a high frequency. However, the NXP boards from Table 5.5 seem overkill as they have a lot more RAM memory than what the program requires. Also, CMSIS-NN is implemented for Cortex-M4 and Cortex-M7 devices but can still be manually brought to a project. The disadvantage is that it would not benefit the optimizations that are specific to Cortex-M4 and Cortex-M7 devices. For these reasons, the **Mediatek MT2523 board** seems to be a good choice as it has an ARM Cortex-M4 core.

Unfortunately, the Arm Development Studio tool on which the network is implemented does neither provide the Mediatek target SoC, nor the NXP ones that are cited in Table 5.5. Thus, debugging was simulated on a CMSDK_CM4 board which also has an ARM Cortex-M4 core. However, the device is operating at 25MHz and the execution time of the different layers is shown in Table 5.6.

Operation	Time[s]
Preprocessing	1.45
1st convolution	23.92
1st ReLU	0.05
1st max-pool	10.2
2nd convolution	25.04
2nd ReLU	0.01
2nd max-pool	5.31
3rd convolution	22.44
3rd ReLU	0.02
3rd max-pool	2.6
Average-pool	0.17
1st FC	0.00
2nd FC	0.01
Total	91.22

Table 5.6: Execution time of the different layers of the inference run on CMSDK_CM7 device. As expected, convolution operations take majority of the runtime. They indeed occupy 78% of it. The other layers take the remaining 22% as follows: 20% for the pooling layers, 1.5% the preprocessing, and the others take almost no time in comparison.

We have a total runtime of 91.22 seconds, and if we assume that the runtime is proportional to the inverse of the operating frequency, we obtain for the MT2523 board that operates at 208MHz

$$t = \frac{91.22 \times 25}{208}$$

and we have a **theoretical runtime of 10.96 seconds**.

However, we also have to keep in mind that we are in a real-time application. Typically, Pillcam Colon capsules, from Fig. 2.2 in background, have an imaging sampling rate of 4 frames per second. So if we want our network to match this speed, it should process an image in less than 0.25 seconds.

Finally, the behavior of the CMSIS-NN network has been validated by comparing the outputs to the ones of the quantized network from PyTorch for each layer, and for several images. The test consists in comparing each element of their outputs and asserting that no one differs by more than 1, which could be justified by elements being rounded in C as it works with integer values.

5.4 Summary

In this chapter, we first introduced the fixed-point arithmetic that is used in CMSIS-NN by explaining its Q format and the shifts that accompany the values.

Then, we converted our network from PyTorch to CMSIS-NN. This required fusing the batch-normalization layers in their respective convolutional layers to copy their effects. Volumes format also had to be changed to match the NHWC format from CMSIS-NN. Then, we quantized the network weights and biases with the binary scales that minimize the first order difference between full-precision and 8-bit elements. After this, we could extract these parameters to use them in our embedded network.

As CMSIS-NN also requires knowing the layer output Q formats, we made a grid search to iterate over the different scales and chose the set of scales that provides the best performance. This lead to a 92% recall, or a 3% gap in the target metric.

After this, we saw how we could save RAM memory by switching between two buffers during the execution, and this allowed our implementation to require 2.41MB of RAM memory for these buffers and the im2col output for convolutional layers, and a ROM memory capacity of 527.563kB for the network weights and biases.

Once we knew the memory requirements of our model, we could select a device for deployment. We ended up selecting Mediatek MT2523 board for its Cortex-M4 core that matches CMSIS-NN targets. The device has 4MB of RAM, 4224kB of ROM, and can operate at a frequency of 208MHz.

Chapter 6

Conclusion

This research aimed to create a convolutional neural network which would be embedded in a WCE camera to perform a computer-aided diagnosis. The neural network had to detect polyps by classifying them in the frames captured by the camera. As we did not want to miss any potential polyp, we targeted a high recall and accuracy classifier. Also, as the network should be embedded, we wanted it to have low memory requirements.

We answer this by proposing a model with 92% recall and 90% accuracy on a custom testing set. This model requires a 528kB ROM storage and a 2.41MB RAM memory.

To do so, we first built our custom dataset based on the ETIS-Larib Polyp DB images. We chose this database because it was easily accessible in addition of containing large images, and by using data augmentation, we could create a dataset of 25k samples from the 196 images that it contains.

However, this dataset could be improved in two ways. First, the training and testing sets both come from this dataset. Even though they do not contain the same images, it would have been better to confront our network to a third-party dataset during test. Testing on an existing dataset also allows situating the work with respect to state-of-the-art results. Secondly, we used a threshold value as minimum size of polyp on the polyp masks to label the images of the created set. This threshold value could have been processed as a hyperparameter to determine its best value. This would have allowed to calibrate with performance as criterion.

When we had our dataset, the following step was to build our model. We took inspiration from the VGG Net network that proved to be efficient for polyp classification. It has the specificity of doubling the number of channels after each convolutional layer. We did the same by first using 3 repetitions of CBA and max-pooling layers. We then added an average-pooling layer and a flatten layer that is followed by 2 FC layers. Then, we highlighted that our network needed

dropout to avoid overfitting. This led to the addition of dropout before each FC layer, where it proved to be efficient.

We used grid search to choose the hyperparameters and the optimizer of our model. Here, we could have used random search instead, which sometimes provides better results. Also, we could not do an exhaustive search due to Colab's limited memory and computational speed. We rather opted for learning rate scheduling during optimization training. Learning rate scheduling was a good choice as, in addition of being a training algorithm, it also allowed not iterating over the learning rates in our hyperparameters search, and thus reduce its timing complexity.

After inspecting weighted-loss cross-entropy function, we ended up concluding that the best hyperparameters among the ones we investigated were, for the SGD optimizer, a batch size of 32, dropout probabilities of 50%, and no weights for the loss function. The training of this model provided 95% recall and 96% accuracy for our testing set.

Once our model was built, we still had to embed it. By quantizing its weights and biases with 8-bit values, we could divide its size by 4, with a 6% and 3% deviation in accuracy and recall as counterpart. The 3% gap on the target metric matches the results of paper [26]. This paper also stated that quantization-aware training could have provided better results, while we opted for post-training quantization to save training time.

At the end, as we could not directly test our implementation on the Mediatek MT2523 target SoC, we predicted that our application would take 10.96 seconds to process an image at 208MHz, based on a simulation on the CMSD_CM4 device. And we saw that, if we compare this to the speed of the Pillcam Colon cameras that capture 4 frames per second, our model could not follow.

This might seem to be far results, but remember that we opted for a top-down approach when designing our model. The bottom-up counterpart, which consists in building the model based on timing and memory constraints could have provided suited results.

We proposed several ways to go further in this work, however there is still room for improvement to reach an embedded solution that matches real-time constraints. Also, this work could be adapted to match other purposes than polyp detection.

Appendix A

Histograms of polyp sizes

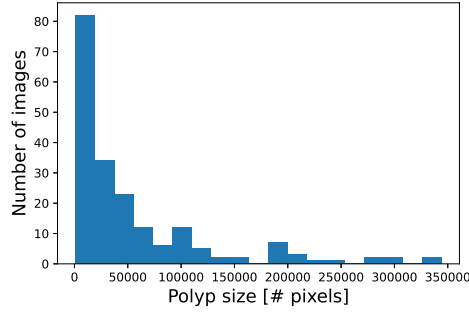


Figure A.1: Histogram of polyp sizes from the original database. The minimum pixel size is 1257 pixels and more than 50% of the images lies within the 20k pixels range.

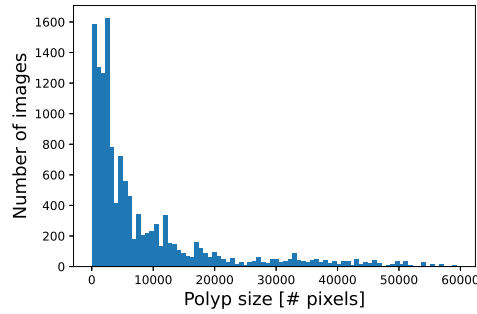


Figure A.2: Histogram of polyp sizes of the created database. There are 5400 images with 0 polyp pixels, but they are not shown here. Again, most of the samples lie in the first tiers, and by setting a threshold of minimum 300 pixels to be classed as a polyp, we get 32% of non-polyps.

Appendix B

Network without dropout

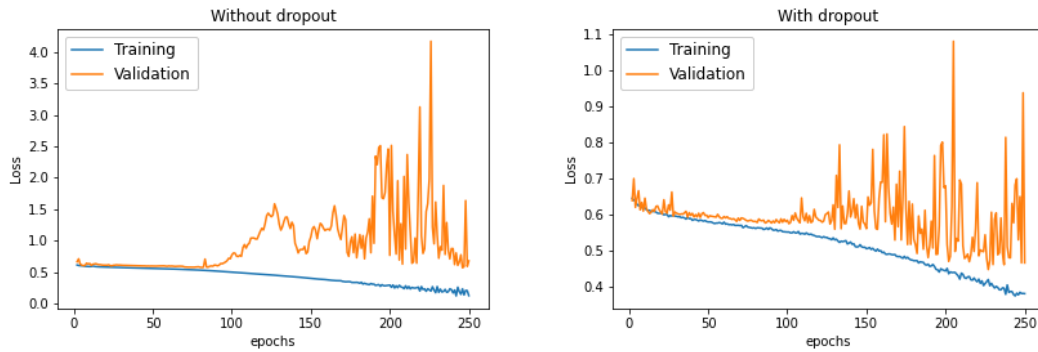


Figure B.1: The left figure shows the evolution of the loss when no dropout is applied to the network and the right one corresponds to the loss when a dropout probability of 50% is added before each FC layer. The loss exceeds a value of 4 without dropout, and is bounded to 1.1 in presence of dropout.

The overfitting behavior of the left figure is highlighted by the fact that the training loss reaches very small values, while the validation one does not follow. This can be interpreted as if the model knows the training data very well but can not use what it learned to predict validation data.

However, we notice that the validation losses have similar shapes. This behavior might be due to the data batch that is evaluated rather than the network itself.

Appendix C

Computation of forward/backward size

This appendix explains how the 113.21 MB forward/backward pass size comes from and how it is computed.

First, PyTorch sees each function as a layer. So here for example, the CBA is distinguished from its components. Figure C.1 shows the total number of values that compose the forward/backward pass memory.

PyTorch vision of the network		
Layer (type)	Output Shape	
Conv2d-1	[-1, 32, 250, 250]	} → 4*32*250*250
BatchNorm2d-2	[-1, 32, 250, 250]	
ReLU-3	[-1, 32, 250, 250]	
CBA-4	[-1, 32, 250, 250]	
MaxPool2d-5	[-1, 32, 125, 125]	+ 32*125*125
Conv2d-6	[-1, 64, 125, 125]	} → + 4*64*125*125
BatchNorm2d-7	[-1, 64, 125, 125]	
ReLU-8	[-1, 64, 125, 125]	
CBA-9	[-1, 64, 125, 125]	
MaxPool2d-10	[-1, 64, 62, 62]	+ 64*62*62
Conv2d-11	[-1, 128, 62, 62]	} → + 4*128*62*62
BatchNorm2d-12	[-1, 128, 62, 62]	
ReLU-13	[-1, 128, 62, 62]	
CBA-14	[-1, 128, 62, 62]	
MaxPool2d-15	[-1, 128, 31, 31]	+ 128*31*31
AdaptiveAvgPool2d-16	[-1, 128, 1, 1]	+ 128
FlattenLayer-17	[-1, 128]	} → + 2*128
Dropout2d-18	[-1, 128]	
Linear-19	[-1, 256]	} → + 2*256
Dropout2d-20	[-1, 256]	
Linear-21	[-1, 2]	+ 2
		Total number of output values: 14,838,050

Figure C.1: The total number of values stored in the forward/backward size memory is composed of all the output layers of the network. Image obtained via *model.summary* from PyTorch.

Then, once we know that the total number of values for the forward/backward pass is equal to 14,838,050, we obtain the size in MB as

$$\begin{aligned}Size &= \frac{2 \times 4 \times 14838050}{1024^2} \\ &= 113.205\end{aligned}$$

We multiply by 2 as we have values for prediction (forward pass) and gradients computation (backward pass). The $\times 4$ comes from that PyTorch uses 4 bytes values, and we divide by 1024^2 to get the result in MBs.

Appendix D

Comparison of learning rates operation for SGD and Adam

In order to determine ranges of learning rate operation for both optimizers, we look at the mean of losses obtained for the hyperparameters over which we iterate in our model tuning. Thus, for batch sizes in $[32, 64]$, dropouts probabilities in $[25\%, 50\%, 75\%]$, and learning rates in $[1e-1, 1e-2, 1e-3, 1e-4, 1e-5, 1e-6]$, we train the models for 100 epochs on a reduced set. Then, we average the losses of models that have the same learning rate. Figure D.1 shows the evolution of the mean loss as well as their standard deviation with respect to learning rate.

We observe that the Adam optimizer mean loss behaves as a quadratic function with a minimum at a learning rate of $1e-4$. On the other hand, the SGD optimizer mean loss goes down as the learning rate increases. It then rises slightly for a learning rate of $1e-1$.

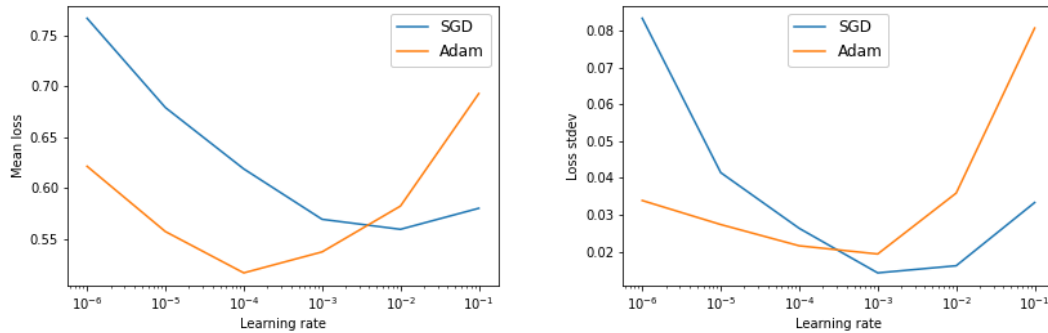


Figure D.1: The left figure shows the mean losses with respect to the learning rate for SGD and Adam optimizers. Their standard deviations are shown on the right figure. Roughly, we observe that SGD optimizer works better with higher learning rates.

Bibliography

- [1] <https://www.dropbox.com/s/p5qe9eotetjnbmq/CVC-ClinicDB.rar?dl=0>.
- [2] Juan S. Silva, Aymeric Histace, Olivier Romain, Xavier Dray, Bertrand Granado, Towards embedded detection of polyps in WCE images for early diagnosis of colorectal cancer. *International Journal of Computer Assisted Radiology and Surgery*, Springer Verlag (Germany), 2014, 9 (2), pp. 283-293.
- [3] <https://www.dropbox.com/s/j4nsxijf5dhzb6w/ETIS-LaribPolypDB.rar?dl=0>.
- [4] N. Tajbakhsh, S. R. Gurudu and J. Liang, "Automated Polyp Detection in Colonoscopy Videos Using Shape and Context Information," in *IEEE Transactions on Medical Imaging*, vol. 35, no. 2, pp. 630-644, Feb. 2016.
- [5] July 2018. https://github.com/Tony607/arm_nn_examples/tree/master/cifar10.
- [6] 004 CNN Padding, November 2018. <http://datahacker.rs/what-is-padding-cnn/>.
- [7] ARM-software/CMSIS_5, July 2020. https://github.com/ARM-software/CMSIS_5.
- [8] Keras vs TensorFlow vs PyTorch, 2020. <https://www.educba.com/keras-vs-tensorflow-vs-pytorch/>.
- [9] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for Hyper-Parameter Optimization. In *NIPS*, 2011.
- [10] J. Bernal, J. Sánchez, and F. Vilariño. Towards automatic polyp detection with a polyp appearance model. *Pattern Recognition*. Volume 45, Issue 9. pages 3166–3182, September 2012.

- [11] J. Bernal, N. Tajkbaksh, F. J. Sánchez, B. J. Matuszewski, H. Chen, L. Yu, Q. Angermann, O. Romain, B. Rustad, I. Balasingham, K. Pogorelov, S. Choi, Q. Debar, L. Maier-Hein, S. Speidel, D. Stoyanov, P. Brandao, H. Córdova, C. Sánchez-Montes, S. R. Gurudu, G. Fernández-Esparrach, X. Dray, J. Liang, and A. Histace. Comparative Validation of Polyp Detection Methods in Video Colonoscopy: Results From the MICCAI 2015 Endoscopic Vision Challenge. *IEEE Transactions on Medical Imaging*, 36(6):1231–1249, 2017.
- [12] Jorge Bernal, Francisco Javier Sánchez, Gloria Fernández-Esparrach, Debora Gil, Cristina Rodríguez de Miguel, and Fernando Vilariño. WM-DOVA maps for accurate polyp highlighting in colonoscopy: Validation vs. saliency maps from physicians. *Computerized Medical Imaging and Graphics*, 43:99–111, 2015.
- [13] A. Capotondi, M. Rusci, M. Fariselli, and L. Benini. CMix-NN, March 2020. <https://github.com/EEESlab/CMix-NN>.
- [14] A. Capotondi, M. Rusci, M. Fariselli, and L. Benini. CMix-NN: Mixed Low-Precision CNN Library for Memory-Constrained Edge Devices. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 67:871–875, May 2020.
- [15] Chengwei. How to run deep learning model on microcontroller with CMSIS-NN, July 2018. <https://www.dlology.com/blog/how-to-run-deep-learning-model-on-microcontroller-with-cmsis-nn/>.
- [16] Chengwei. How to run deep learning model on microcontroller with CMSIS-NN, July 2018. <https://www.dlology.com/blog/how-to-run-deep-learning-model-on-microcontroller-with-cmsis-nn-part-2/>.
- [17] Adit Deshpande. A Beginner’s Guide To Understanding Convolutional Neural Networks Part 2, July 2016. <https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks-Part-2/>.
- [18] ©2010 2020 Machine Vision Group. Colonoscopy database Registration. <http://mv.cvc.uab.es/projects/colon-qa/cvccolondb/colonoscopy-database-registration>.
- [19] ©2010 2020 Machine Vision Group. CVC colon DB. <http://mv.cvc.uab.es/projects/colon-qa/cvccolondb>.

- [20] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *ArXiv*, abs/1207.0580, 2012.
- [21] National Cancer Institute. <https://www.cancer.gov/>.
- [22] Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *ArXiv*, February 2015.
- [23] Pier Paolo Ippolito. Hyperparameters Optimization, September 2019. <https://towardsdatascience.com/hyperparameters-optimization-526348bb8e2d>.
- [24] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima. *ArXiv*, abs/1609.04836, 2017.
- [25] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *CoRR*, abs/1412.6980, 2015.
- [26] Raghuraman Krishnamoorthi. Quantizing deep convolutional networks for efficient inference: A whitepaper. *ArXiv*, abs/1806.08342, 2018.
- [27] Fengfu Li and Bin Liu. Ternary Weight Networks. *ArXiv*, abs/1605.04711, 2016.
- [28] Lai Lianghzen, Suda Naveen, and Chandra Vikas. CMSIS-NN: Efficient Neural Network Kernels for Arm Cortex-M CPUs. *ArXiv*, abs/1801.06601, January 2018.
- [29] Arm Ltd. CMSIS-NN Version 1.3.0, April 2020. <https://www.keil.com/pack/doc/CMSIS/NN/html/index.html>.
- [30] The American Cancer Society medical and editorial content team. Key statistics for colorectal cancer, February 2018. <https://www.cancer.org/cancer/colon-rectal-cancer/about/key-statistics.html>.
- [31] The American Cancer Society medical and editorial content team. What is colorectal cancer?, February 2018. <https://www.cancer.org/cancer/colon-rectal-cancer/about/what-is-colorectal-cancer.html>.
- [32] Mark Nadeski. *Bringing machine learning to embedded systems*. Texas Instruments, 2019.

- [33] Krushi Patel, Kaidong Li, Ke Tao, Q. Wang, Ajay Bansal, A. Rastogi, and G. Wang. A comparative study on polyp classification using convolutional neural networks. *PLoS ONE*, 15, 2020.
- [34] O. et al Ramin. Towards a multimodal wireless video capsule for detection of colonic polyps as prevention of colorectal cancer. *13th IEEE International Conference on BioInformatics and BioEngineering, Chania*, 2013.
- [35] Sayantini. Keras vs TensorFlow vs PyTorch : Comparison of the Deep Learning Frameworks, June 2020. <https://www.edureka.co/blog/keras-vs-tensorflow-vs-pytorch/>.
- [36] Connor Shorten and Taghi M. Khoshgoftaar. A survey on Image Data Augmentation for Deep Learning. *Journal of Big Data*, 6:1–48, 2019.
- [37] Vishwesh Shrimali. PyTorch for Beginners: Basics, May 2019. <https://www.learnopencv.com/pytorch-for-beginners-basics/>.
- [38] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. *CoRR*, abs/1409.1556, 2015.
- [39] LJ Sliker and G. Ciuti. *Flexible and capsule endoscopy for screening, diagnosis and treatment*. Expert Rev Med Devices, 2014;11(6):649-666.
- [40] Saha Sumit. A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way, December 2018. <https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53>.
- [41] John Terra. Keras vs Tensorflow vs Pytorch: Understanding the Most Popular Deep Learning Frameworks, July 2020. <https://www.simplilearn.com/keras-vs-tensorflow-vs-pytorch-article>.
- [42] M. Verhelst and B. Moons. Embedded Deep Neural Network Processing: Algorithmic and Processor Techniques Bring Deep Learning to IoT and Edge Devices. *IEEE Solid-State Circuits Magazine* 9, pages 55–65, Fall 2017.
- [43] Sandra Maria Gonçalves Vieira, Walter H. L. Pinaya, and Andrea Mechelli. Using deep learning to investigate the neuroimaging correlates of psychiatric and neurological disorders: Methods and applications. *Neuroscience Biobehavioral Reviews*, 74:58–75, 2017.
- [44] Jason Wang and Luis Perez. The Effectiveness of Data Augmentation in Image Classification using Deep Learning. *ArXiv*, 2018.

- [45] Gangming Zhao, Zhaoxiang Zhang, He Guan, Peng Tang, and Jingdong Wang. Rethinking ReLU to Train Better CNNs. *2018 24th International Conference on Pattern Recognition (ICPR)*, pages 603–608, 2018.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | www.uclouvain.be/epl