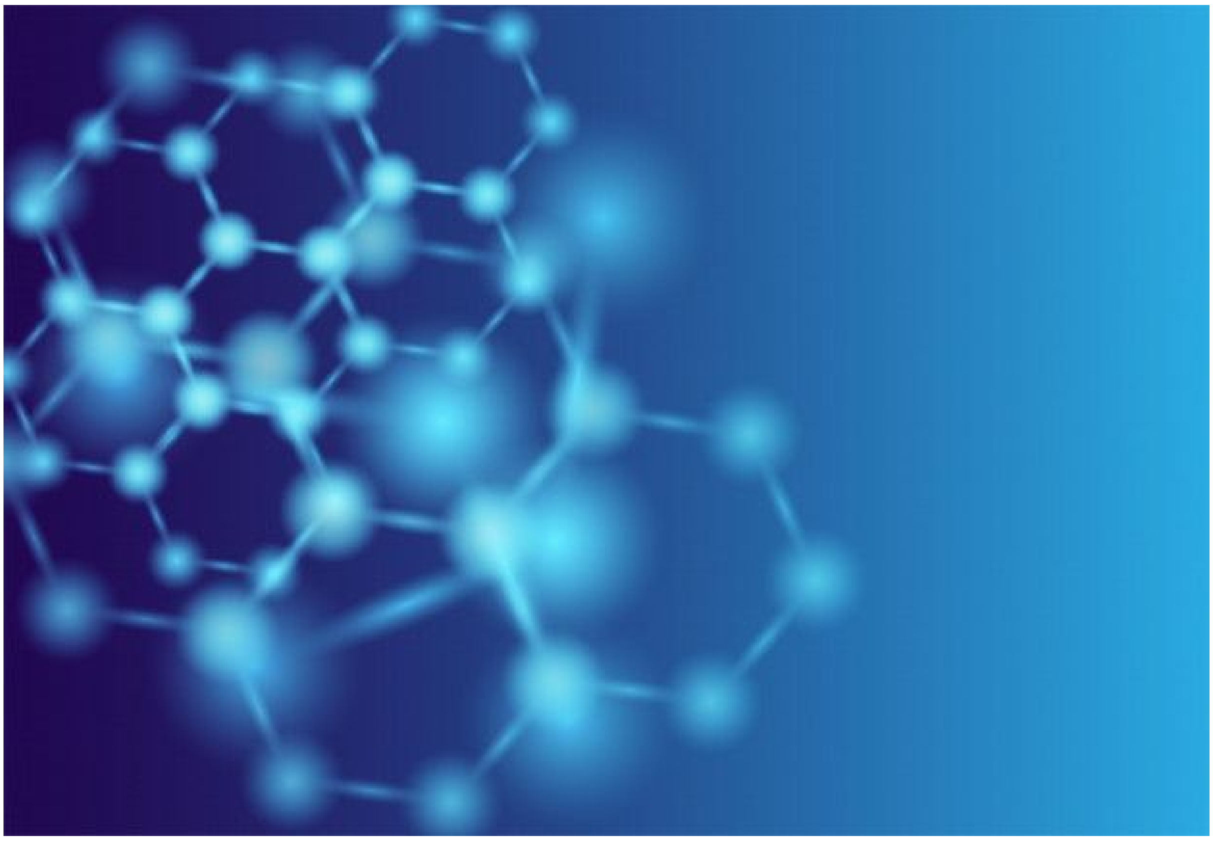
NAVYA PROJECT

Final report



BERTIN Laurina / DHILLON Prashant / MANSUY Adrien / MANSUY Alexandra / SEIFEDDINE Wassim / SHARMA Abhishek

April, 29th 2020
E4 - Embedded systems & IMC

TABLE OF CONTENTS

I - INTRODUCTION

Machine learning is a powerful tool for building models that use data to predict system behaviour and or/ control them. Its applications in all fields kept rising and we can expect it to remain this way in the near future, especially for embedded systems. Typically, the system is constrained by processing cycles, memory usage, model size, power consumption and cost, therefore it can become difficult to implement. The point of this project is to demonstrate the influence of pruning and quantization using MobileNetV2 architecture with SSD lite.

II - STATE OF THE ART

2.1 - Neural networks

Neural networks (NN) are a subfield of machine learning (ML) and it was that field that spawned Deep Learning. Deep learning approaches can be put in categories as follows: supervised, semi-supervised and unsupervised. In addition, there is a category called Reinforcement learning (RL) which uses an observer to get partial real time information from the outside environment or Deep RL.

2.1.1 - Deep supervised learning

This is the study case; supervised learning is a technique that uses labelled data. The environment has a set of inputs and corresponding outputs. After computation, the network will receive a loss score which will be used to improve the parameters for a better approximation of the desired output. After a successful training, the model will be able to get the correct answers to questions from the environment (of course, new ones). There are many different approaches in this category: deep neural networks (DNN), convolutional neural networks (CNN), recurrent Neural Networks, including Long Short-Term Memory etc.

2.1.2 - Deep semi-supervised learning

It is a type of learning that uses partially labelled datasets.

2.1.3 - Deep unsupervised learning

Unsupervised learning systems are ones that can function without the presence of labels on the data. In that case, the model learns the important internal features and representations to discover unknown relationships within the data. Some of the members of the deep learning family perform really well at this, including Auto-Encoders and the recently developed GAN.

2.1.4 - Deep reinforcement learning (DRL)

It is a learning technique used when in contact with unknown environments. It began in 2013 with Google deepmind. From then, several methods have been proposed based on this theory. Depending on the problem, one can decide which type of RL needs to be applied to solve the problem. If the problem has to be optimized, DRL is the way to go. If the problem has fewer parameters, a method without RL is good.

Deep learning is employed in several situations when machine intelligence would be useful. For example, when there is no human expert (navigation on Mars) or when humans are unable to explain their expertise (speech recognition, language understanding), the problem size is too big for our reasoning (sentiment analysis, autonomous driving). Almost all areas can have applications for deep learning, as a result this approach is often called universal learning approach.

2.1.5 - Convolutional neural network (CNN) overview

This network structure was first proposed by Fukushima in 1988. It was not used widely due to limits of hardware while training the models. In the 1990s, Y. LeCun obtained successful results for handwritten digits recognition (MNIST Dataset). After that, much progress was made. CNNs have some advantages compared to DNNs, including being highly optimized for processing 2D and 3D images, which is our study case for object detection, and also being very good at learning 2D features. Moreover, CNNs have fewer parameters than fully connected networks of similar size. Most of all, they are trained with a gradient-based algorithm and suffers less from gradient diminishing.

Gradient diminishing or gradient vanishing: as more layers using certain activation functions are added to neural networks, the gradient of the loss function approaches zero, making the network hard to train. Certain activation functions, like the sigmoid function squishes a large input space into a small input space between 0 and 1. Therefore, a large change in the input of the sigmoid function will cause a small change in the output. Hence, the derivative becomes small.

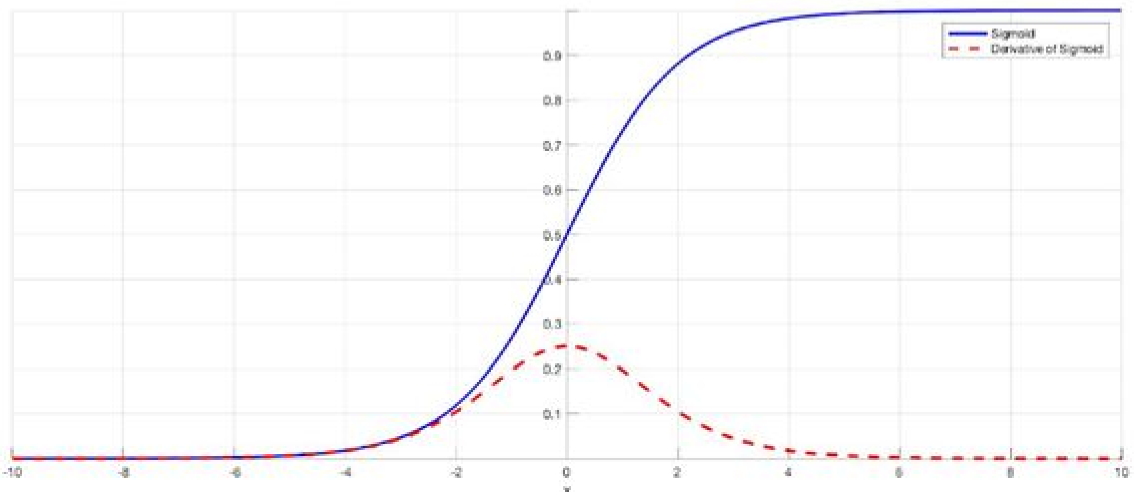


Figure 1: Situation of gradient diminishing

As an example, the sigmoid function and its derivative. Note how when the inputs of the sigmoid function become larger or smaller (when |x| becomes bigger), the derivative becomes close to zero.

For shallow networks with only a few layers that use these activations, this isn't a big problem. However, when more layers are used, it can cause the gradient to be too small for training to work effectively.

Gradients of neural networks are found using backpropagation. Simply put, backpropagation finds the derivatives of the network by moving layer by layer from the final layer to the initial one. By the chain rule, the derivatives of each layer are multiplied down the network (from the final layer to the initial) to compute the derivatives of the initial layers.

However, when n hidden layers use an activation like the sigmoid function, n small derivatives are multiplied together. Thus, the gradient decreases exponentially as we propagate down to the initial layers.

A small gradient means that the weights and biases of the initial layers will not be updated effectively with each training session. Since these initial layers are often crucial to recognizing the core elements of the input data, it can lead to overall inaccuracy of the whole network.

The simplest solution to solve this issue is to use other activation functions, such as ReLU, which doesn’t cause a small derivative.

Residual networks are another solution, as they provide residual connections straight to earlier layers. As seen in the second image, the residual connection directly adds the value at the beginning of the block, x, to the end of the block (F(x)+x). This residual connection doesn’t go through activation functions that “squashes” the derivatives, resulting in a higher overall derivative of the block.

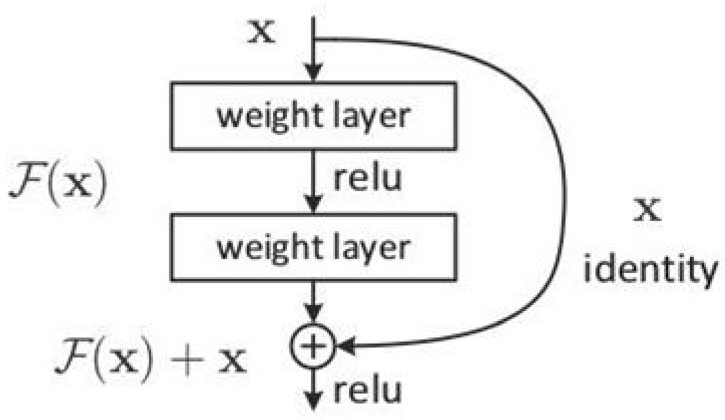

x
weight layer
$\mathcal{F}(\mathbf{x})$
relu
weight layer
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

Figure 2: Residual connections

Finally, batch normalization layers can also resolve the issue. As stated before, the problem arises when a large input space is mapped to a small one, causing the derivatives to disappear. In figure 1, this is most clearly seen at when |x| is big. Batch normalization reduces this problem by simply normalizing the input so |x| doesn’t reach the outer edges of the sigmoid function. As seen in figure 3, it normalizes the input so that most of it falls in the green region, where the derivative isn’t too small.

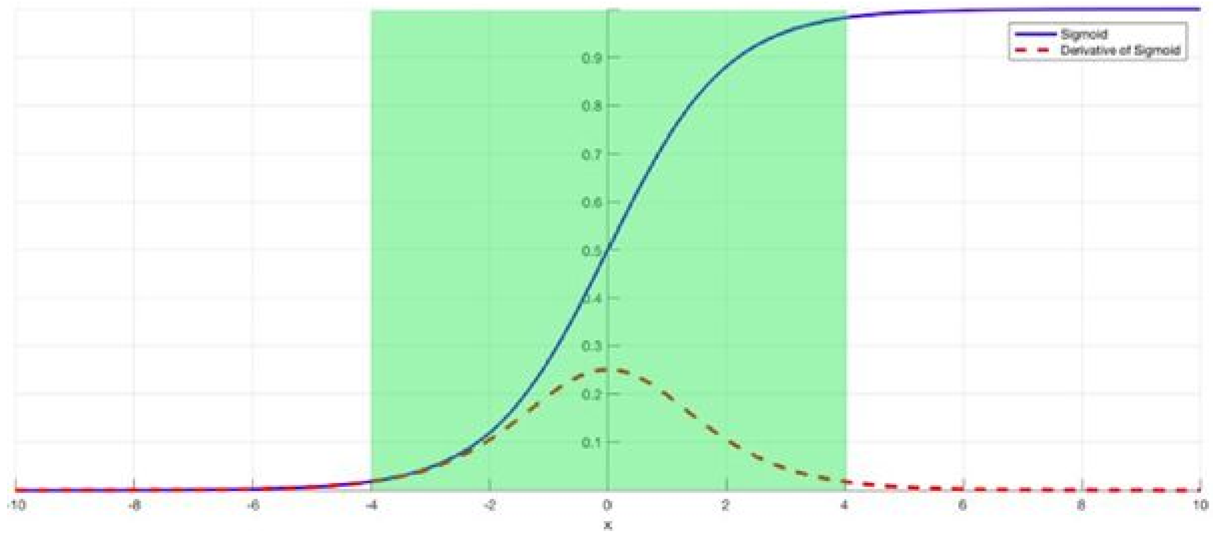


Figure 3: Batch normalization

The architecture of a CNN is divided into two parts : feature extractors and a classifier, here is an example.

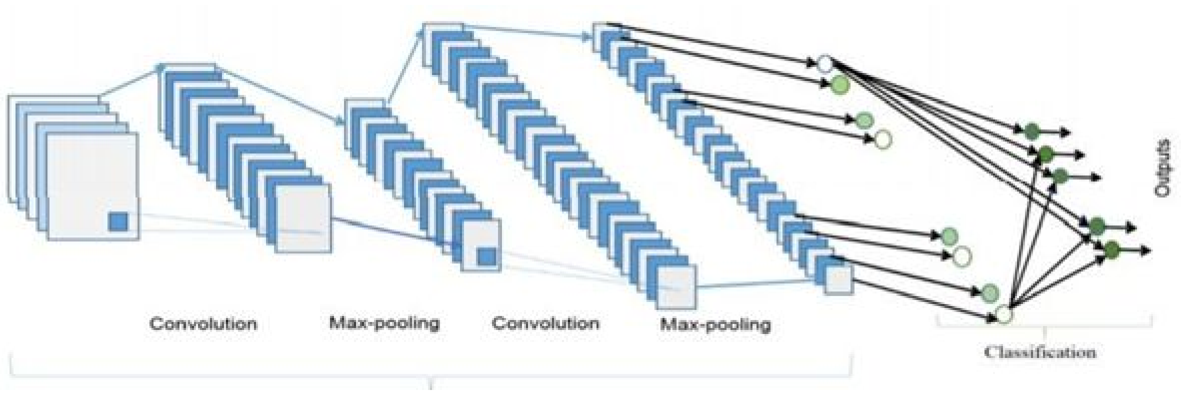


Figure 4: CNN architecture

“Max-pooling” reduces the sampling rate by calculating the average values of precalculated regions of the signal. “Convolutional layer”, in this layer, features from the previous layers are convolved with new data. The output of this operation goes through an activation function such as sigmoid, SoftMax, rectified linear etc.

“Classification layer” computes the score of each class given the previous features determined by the network.

We can now go on to analysis the architecture we are working on: MobileNetV2. It is interesting to state that we could get much better results with other networks, but we would not be able to implement it on a Raspberry PI, which would be problematic if the model is made to be embedded.

2.2 - Embedded networks

Going back to Y. LeCun’s huge progress on CNNs in the 1990s, we can start the journey of embedded networks from LeNet-5 : which consists of convolutions on the input, then subsampling (which is basically a pooling operation, so performing an average on subsections of the previous layer to reduce the input size), then applying more convolution and more subsampling which causes the data size to be smaller and smaller, before going through a series of fully connected layers and ending with gaussian connections.

Gaussian connections are a way to define loss for a classification problem (in this case, we are dealing with digits recognition). SoftMax has mostly replaced it nowadays. From our understanding, it works almost the same way as the two previous full connections but does not use ReLu.

From then, we can state some major progress in the evolution of ImageNet. AlexNet (2012) for example is the real major debut for the power of CNNs to be recognized [7]. It possesses convolution layers and fully connected layers as well. Another example, ResNet (2015) is composed of 152 residual layers and no fully connected layers. The problem starts to be clearly stated when the size of networks becomes ridiculous and we can summarize it with an image, which we think is very fitting for the situation:

©2005 FLORIDA TODAY
caglecartoons.com
JEFF PARKER
jparker@flatoday.net
Little Raspberry Pi
SOCIAL SECURITY CHINA SHOP
MIND IF MY FRIEND COMES IN WITH ME?
Big Fat 61M parameters network

Figure 5: Comparison between size of network and size of hardware [3]

These types of networks are indeed very heavy and cannot be implemented on small hardware.

With architectures like MobileNet and ShuffleNet, it is time for architectures to fit on hardware like Raspberry PIs.

The previous architectures required a huge amount of memory and computational power. In order to run image classification or object detection on mobile devices, it is a must to create lighter models with sufficient accuracy. As stated before, we could have much higher accuracy with different models, but the network size has to be limited in order to be implemented in the kind of hardware we want.

We can clearly see why it was recommended to us to use MobileNetV2, even if a V3 version of MobileNet was submitted in May 2019, based on a “combination of complementary search techniques as well as a novel architecture design”.

This new generation of MobileNet has a large and a small version aimed for high and low resources use cases. MobileNetV3-Large is about 3.2 % more accurate on ImageNet classification when reducing latency by 15% compared to MobileNetV2.

MobileNetV3-small is 4.6% more accurate while reducing the latency by 5% compared to MobileNetV2.

MobileNetV3-Large detection is 25% faster at roughly the same accuracy as MobileNetV2 on COCO detection.

MobileNetV3-Large LR-ASPP is 30% faster than MobileNetV2 R-ASPP at similar accuracy for Cityscapes segmentation.

III - MODEL ANALYSIS

3.1 - Backbone

The model used as backbone is Mobile Net V2. We could have used the V3, but we decided not to as we already started on it and we had more references available online. The backbone is created using specific operations that we are about to define in the following subsections:

3.2 - Depth wise separable convolutions

A conventional convolution transforms a Df*Df*M (input size of Df and M channels) to a

Df*Df*N block using a Dk*Dk*M*N kernel (tensor).

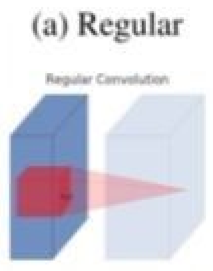


Figure 6: regular convolution [4]

The cost to compute the whole output is very large (Dk*Dk*M*Df*Df*N). MobileNetV2 introduced a new approach to convolution which basically splits the convolution into 2 steps of filtering and combination. This new approach called depth wise separable convolution results in a much lower cost (down around 9 times for Dk =3) .

Each tensor of dimension hi*wi*di can be treated as hi*wi pixels with di dimensions. It can be embedded in “low dimensional subspaces “which means that reducing the dimension of the layer would not cause information loss.

It is not so true with transformations like ReLu : actually, ReLu is only capable of preserving complete information about an input manifold if the input lies in a low-dimensional subspace.

This is why we use linear bottleneck layers, to keep the information in a low subspace.

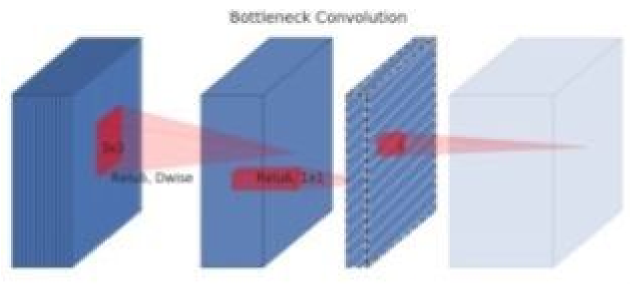


Figure 7: Bottleneck convolution [4]

Linear layers prevent non-linearities from destroying too much information. On the other hand, using non-linear layers in bottlenecks hurts the performance by several percents.

3.3 - Linear bottleneck and inverted residuals

Residual connections improve the ability of the gradient to propagate, as we saw before, it is necessary to keep the gradient under control in both exploding gradient cases AND diminishing gradient cases. Residual connections allow us to pass the gradient through layers. Residual blocks pass the gradient through expanded layers only, so layers with a large number of channels.

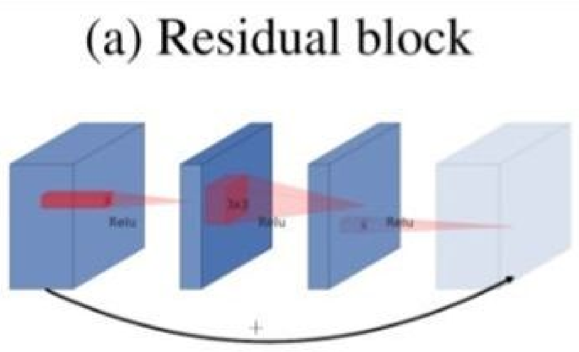


Figure 8: Residual block [4]

While inverted residual blocks pass the gradient through bottlenecks, so with a smaller number of channels, and also these are layers involving no non-linearities. This inverted approach is considerably more memory efficient.

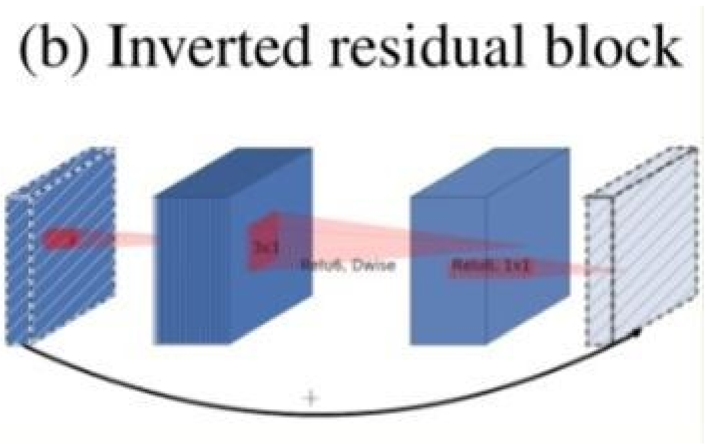


Figure 9: Inverted residual block [4]

3.4 - Effect of linear bottlenecks and inverted residuals

Combining depth wise separable convolution, linear bottlenecks and inverted residual blocks allows the computational cost per block to be much lower, and with this, input and output dimension can be relatively small. In addition, it allows a much lower memory cost which is amazing when doing embedded applications.

3.5 - MobileNetV2 architecture

We can represent the model with this diagram:

Input	Operator	t	c	n	s
224^2 x 3	Conv2d	-	32	1	2
112^2 x 32	bottleneck	1	16	1	1
112^2 x 16	bottleneck	6	24	2	2
56^2 x 24	bottleneck	6	36	3	2
28^2 x 32	bottleneck	6	64	4	2
14^2 x 64	bottleneck	6	96	3	1
14^2 x 96	bottleneck	6	160	3	2
7^2 x 160	bottleneck	6	320	1	1
7^2 x 320	Conv2d 1x1	-	1280	1	1

7^2 x 1280	Avgpool 7x7	-		1	-
1x1x1280	Conv2d 1x1	-	k	-	

Figure 10: Backbone structure

IV - GIT RESOURCES

As a starting point for the project, we have been taking this git as a base: https://github.com/qfgaohao/pytorch-ssd

It contains a pretrained MBV2 as well as a video demo. It also contains MBV1, VGG, SSD/SSD Lite implementation in pytorch. It uses Open images dataset.

As a base for quantization, we have been using this git link which contains an implementation of INT8 quantization on pytorch: https://pytorch.org/blog/introduction-to-quantization-on-pytorch/

When facing issues regarding quantization we also used this git; which gives a new definition for the model: https://github.com/pytorch/vision/blob/master/torchvision/models/mobilenet.py

V - MAIN FUNCTIONS CREATED

5.1 - VOC

We have adapted the git implementation to function with VOC2007 dataset.

5.2 - Metrics script

In order to evaluate models in a standardized way, we have implemented a few metrics that would give a good indicator of performance:

Average sparsity: after applying the pruning mask onto the tensors with different ratios, some of the values have been turned into 0s. The average sparsity returns a value corresponding to the percentage of 0s in average across all tensors. It is calculated by counting the weights equal to 0 over the number of total weights considered times 100 to convert into a percentage. We started by pruning batchnorm layers and conv2d layers but it was affecting the accuracy a lot, due to the small size of the model. So, we decided to only prune the conv2d layers.

Average precision per class: allows us to measure how accurate the model's predictions are. It is calculated using precisions and recall:

(5

$$Precision = \frac{TP}{TP + FP} \qquad TP = \text{True positive}$$

$$TN = \text{True negative}$$

$$Recall = \frac{TP}{TP + FN} \qquad FP = \text{False positive}$$

$$FN = \text{False negative}$$

$$F1 = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

Figure 11: F1 calculation [5]

With:

True positive: is an outcome where the model correctly predicts the presence of an object that exists in the real scenario.

True negative: is an outcome where the model correctly predicts the absence of an object existing in the real scenario.

False positive: is an outcome where the model detects the presence of an object that isn't truly there.

False negative: is an outcome where the model does not detect the presence of an object that exists.

$$Precision = \frac{TP}{total\ positive\ results}$$

$$Recall = \frac{TP}{total\ of\ cases}$$

Figure 12: precision and recall [5]

From there, in order to calculate AP, we need to divide the recall value from 0 to 1 into 11 points (standard, even if it doable with a different number than 11) [6] Then we compute the average of maximum precision value for these 11 recall values.

MACs:Multiplications and accumulations.

Used memory: The percentage of used memory for the whole program to be executed. (MB)

Model size in MB

FPS: We first tested the pytorch fps function but it was not accurate, so calculated it manually, first over 120 frames, later on over 10. The time considered for one frame is made of the addition of the reading time for an image, the image plotting time and the boxes' printing time.

Inference time: time taken by the model for predicting an image, when evaluating. We calculated it with two timers at the beginning and the end of the line which does the prediction. (we separated this category as mean inference time and std inference time)

We first implemented them directly into the evaluation file and used matplotlib (in the file plotter) to print the pruning levels and corresponding accuracies. Then, we thought it would be better to make it a set of functions usable at all times, for any models, even in different codes.

All of these measures are saved into a CSV file with the date included in the file's name.

5.3 - Quantization

Quantization refers to techniques for doing both computations and memory accesses with lower precision data, usually int8 compared to floating point implementations. This enables performance gains in several important areas:

- 4x reduction in model size;
- 2-4x reduction in memory bandwidth;
- 2-4x faster inference due to savings in memory bandwidth and faster compute with int8 arithmetic (the exact speed up varies depending on the hardware, the runtime, and the model) [11]

Quantization algorithms can be separated into different categories: linear and range-based. Linear algorithms quantize float values by multiplying them with a scale factor which is a numeric constant, range-based algorithms look at the range of the tensor's values to calculate the scale factor.

Asymmetric: in this mode, we map the min.max in the float range to the min/max of the integer range, it is done using a zero-point (also called quantization bias or offset) in addition to the standard scale factor.

Symmetric: instead of mapping the exact min/max to the quantized range, we choose the maximum absolute value between the min/max. In addition we zero-point. The range we then create is symmetric with respect to zero.

Post training: it consists of wrapping existing modules with quantization and de quantization operations.

Quantization-aware: applies weights quantization to convolution, fully connected and embedding modules, it will also insert FakeLinearQuantization modules after ReLU appearances. Similarly to the post training, the scale factors, zero-points are stored as buffers within their modules, so they are saved.

DoReFa: This method defines a quantization function which takes a real value between 0 and 1 and outputs a discrete value ranging from 0/(2^k -1) and (2^k -1 / 2^k -1) where k is the number of bits used for quantization.The weights are defined using a function f which takes an unbounded real value input and outputs a real value in [0;1]

PACT: This method is similar to DoReFa, described above, but the upper values of the activation functions are learned parameters instead of being hard coded to 1. This clipping value is shared between layers.

WRPN:In this method, activations are clipped to [0;1] and quantized as follows :

$$x_q = \frac{1}{2^k - 1} round\left(\left(2^k - 1\right) x_f\right)$$

weights clipped to |-1; 1] are quantized as follows

$$w_q = \frac{1}{2^{k-1} - 1} round\left(\left(2^{k-1} - 1\right) w_f\right)$$

k is the number of bits used for quantization. This method requires training the model with quantization-aware training.

Figure 15: Different quantization methods [9]

Using Pytorch we can do three kinds of quantization:

1: Dynamic: This involves not just converting the weights to int8 - as happens in all quantization variants - but also converting the activations to int8 on the fly, just before doing the computation (hence “dynamic”).

2: Post-Training Static Quantization: Static quantization performs the additional step of first feeding batches of data through the network and computing the resulting distributions of the different activations (specifically, this is done by inserting “observer” modules at different points that record these distributions). This information is used to determine how specifically the different activations should be quantized at inference time (a simple technique would be to simply divide the entire range of activations into 256 levels, but we support more sophisticated methods as well). Importantly, this additional step allows us to pass quantized values between operations instead of converting these values to floats - and then back to ints - between every operation, resulting in a significant speed-up. That's what we are going to use in our case and explain how to progress to achieve a quantized working model.

3: Quantization Aware Training: In this method all weights and activations are “fake quantized” during both the forward and backward passes of training: that is, float values are rounded to mimic int8 values, but all computations are still done with floating point numbers. Thus, all the weight adjustments during training are made while “aware” of the fact that the model will ultimately be quantized; after quantizing, therefore, this method usually yields higher accuracy than the other two methods.

We chose to start with Post static quantization, because after doing literature survey we found out that it is best advised to use post static quantization for our CNN model because of throughput limited by memory bandwidth for activations in CNNs.

In order to quantize the mobilenet architecture we had to modify it by adding QuantStub layers to inputs to quantize them, dequantstub layers to outputs to de-quantize them as well as fuse the relu, conv2d and inverted residual together. The new layer types are now the following:

Type	Role
QuantStub	Quantizing input
DequantStub	De-quantizing output
ReLu	bottleneck
Identify	Merging the conv2d, batchnorm and Relu
Batchnorm	Forces the activations to be unit standard deviation and zero mean
Conv2d	Performs 2d convolution
Inverted Residual	blocks pass the gradient through bottlenecks

Figure 16: Architecture changes for quantization

5.3.3 - Companies which use quantization and pruning

Several companies have already implemented quantization into their models, here is a non-exhaustive list:

Name	Type	Application
NVIDIA Corporation	int8	- video streaming - speech recognition - recommendation and natural language processing
Xilinx	int8	- deploying AI inference on Xilinx Edge AI platforms
Qualcomm	int8	- to implement on their chipsets
Adobe	-	- to implement ML models in their photoshop and other graphics tool
DeepScale	-	- deploying machine capable models for client’s need with squeezenets
Facebook/Google/Microsoft	-	- for variety of projects, specially their cognitive services
Mathworks	-	- Mathworks rapidly improving Matlab for Machine learning packages - to run models on matlab system instead of cloud
Amazon	-	- to improve data warehousing analysis tools using ML

5.4 - Pruning

5.4.1 - Definition

Pruning is another method used to produce models that are smaller in size, more memory efficient, more power efficient and faster at inference with a minimal loss in accuracy. Other techniques exist such as weight sharing (and quantization). In this category we are finding several different techniques that we will go over.

Method	Description	Accuracy drop on CIFAR10 (Resnet)
From scratch	Building a pruning pipeline that can be learned from randomly initialized weights	-1.20% (50%)
Adversarial Neural	Suppresses higher distorted features	-2.11% (unknown)
L1-norm	Each layer involves pruning a percentage of filters with smaller L1-norm	-0.47% (40%)
Transformable architecture search	The width and depth of the pruned network are obtained by knowledge transfer from the original	-1.85% (41.8% pruning)
Self-Adaptative	A pruning module is embedded in each layer on the convolution network, the convolution is skipped if the pruning decision is 0	Unknown
Lagrangian 10 norm	Enables the network to choose which weights to remove. The weight matrices are factorized into two smaller ones.	Unknown
Random	Randomly chooses which weights are set to 0	Unknown

Figure 17:Different pruning methods [6], [7]

We first started by experiencing with random pruning, but the accuracy was dropping too abruptly after 5% pruning so we moved on to L1 regularization also called Lasso regression [8]: this method adds a regularization term (a penalty) to some of the weights and make them turn to 0s, which makes these terms negligible and helps simplify the model. It is important to mention that it will not reduce the model's size, as it is not deleting the weights, but only setting them to 0.

VI - EXPERIMENTS

In this section we will be discussing the main difficulties of the project, besides the current lockdown which considerably impacted the project, as you would have guessed.

6.1 - Exploitation systems compatibility

One of the most problematic issues of the project has been the differences in hardware of the different members. First of all, benchmarking on different hardwares would have been not wise and would alter the performances. Then, transferring files between members using different exploitation systems caused many issues, mainly being paths and packages downloading issues. These were not too complicated to solve once they appeared a couple of times, but they were very time consuming overall.
In addition, the hardware differences really showed during the video demonstration, where the members using Linux obtained considerably better performance than the ones using MacOS or Windows.
When updating the project, an error occurred and it corrupted the pretrained model from google when the code seemed to have no issues, it was fixed by simply deleting the corrupted version and redoing the work with a backup..
During evaluation (cf part influence of pruning) it was also proven that Windows alters negatively the values of some metrics: MACs, Mean inference time, std inference time, memory consumption.

6.2 - Pruning

Pruning all layers

At first, we wanted to prune the batchnorm and the conv2d layers, but it decreased the accuracy way too much. Everytime we send a batch of images to the network, the batch normalization layer estimates the mean and variance of the batch, and during inference it applies those estimated values to single images. So when we pruned those layers the scaling on the inputs were not done properly anymore which explains the bad results.

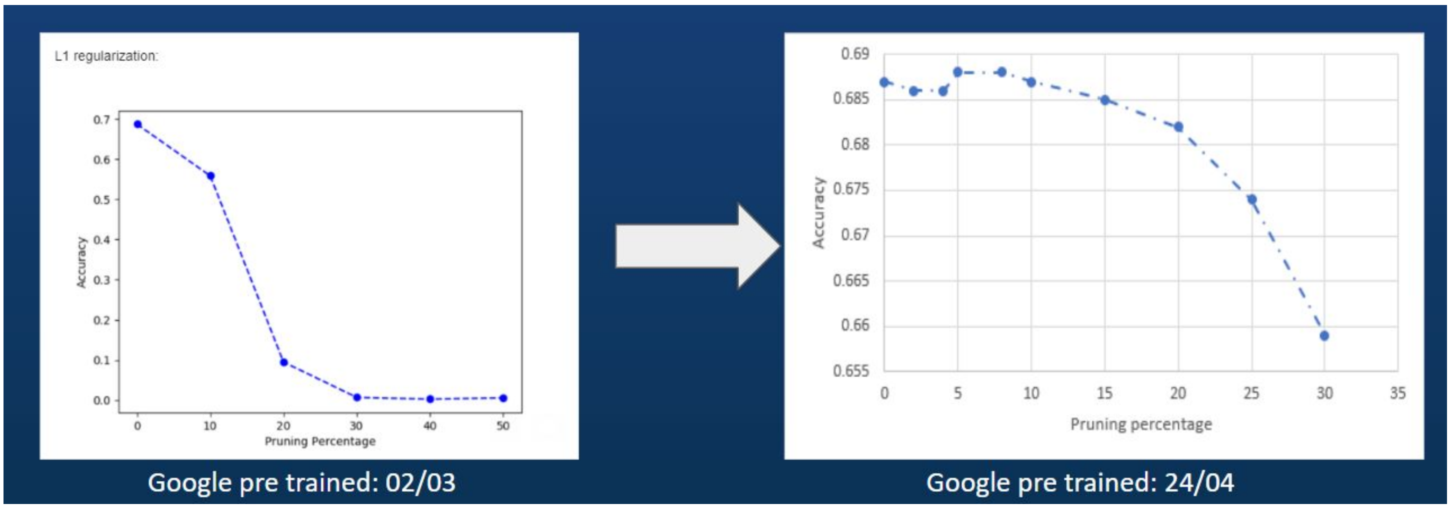


Figure 19: Comparison pruning before after

As it is shown on the diagram above we got much better results doing this method.

40%

One unsolved problem that occured is a bug when pruning at 40%. We ended up leaving this percentage out. Every other pruning level is fine, 10, 20, 30, 50% for example had no issue.

metrics

At this level of the project we already had a script for metrics in order to evaluate the model, but it was directly included in the file which generated the model. It leads to being unusable for other models than the one that was in the file (the google pre-trained from the git). In order to correct this issue we had to transform the script into actual functions, taking the model as a parameter.
Then we thought it would be more convenient to save everything in a csv file. The evaluation process for different pruning levels is very long (approximately 1h20 per computation).
The calculations of MACs had an issue with Windows users, that is why it is not included in the evaluation.

Evaluation

When performing the evaluation of the pruned models, we noticed weird behavior on certain metrics, the size of the model had doubled (this issue is detailed in the section “increase in size below), the memory consumption that was expected to decrease with the pruning percentage was constant during all the evaluation process (details in section “memory consumption” below) and the inference times (both mean and std) were expected to decrease with the pruning percentage but they were stable all along the evaluation process.

Inference time

This issue with inference time has not a clear answer from our team but we have a realistic guess: this issue might be linked with windows users as when the evaluation was partially executed by a member that was not using windows, the metrics changed to their expected behavior.

Increase in size

While pruning the navya model after fixing the metrics script, we noticed that the model size increases, which is due to how the pruning in pytorch works: we implemented L1 norm pruning, which generates a mask for the input tensor (which will be of the same dimensions as the weight matrix). The placement of the 0s and 1s of the binary mask is determined by the L1 norm of the weight matrix, a 0 is placed if the norm is small, otherwise a 1 is placed. Then, the pruning handles the multiplication between the weight tensor and the mask to return a new weight tensor which will be the output of the pruning function. The increase in size is then due to those masks being added to the model. We solved this problem by removing the masks off the model without removing the pruning outputs.

Memory consumption

In order to solve the memory consumption metric, we saved the pruned models into a separate directory and loading them in a separate script: we then got the real size of the model which is constant around **100~101 MB**

6.3 - Quantization

Model inference

After quantizing the google pretrained model there was a problem in performing model inference. This was because of different layers architecture of SSD+MNV2 module before and after architecture changes.This was solved by debugging and using following ideas: Initialize the original model then Load the pretrained weights. Quantize it and save the weights. Now, to load the quantized weights in the previous model definition we initialize the model then quantize it and load the weight.

Model definition

We found out there are some issues in model fusion. We encountered that issue because we have two distinct architectures SSD and MBV2 (the google model, and the modified navya architecture), so fusing layers must be fused in both architectures. Later we replaced the MBV2 backbone to pytorch module because of some shape errors.

Inverted residual

We needed to change the inverted residuals function for our new model definition to solve the shape mismatch error. Because there were conflicting problems between inverted residuals as they were different in SSD and MNV2.

Shape mismatch error

After changing the inverted residual blocks, the shape mismatch error was still remaining. We fixed it by changing the input channels of the first separable convolutions in both the regression and classification headers of the SSD lite from 576 to 160.

Calibration

After implementing a random weight model on the Navya pretrained model (25% accuracy when not quantized), we noticed poor performance, around 0.1% accuracy .We used calibration in order to improve the accuracy. During this stage, a small fraction of images from the validation dataset (1–5%) is used for collecting statistical information including naive min/max or optimal thresholds based on entropy theory and defining scaling factors using symmetric quantization and execution profiles of each layer [11]. The output of this stage is a calibrated model including quantized operators saved as a JSON file and a parameter file. After calibrating, we reached the same accuracy as the unquantized model.

Before: mean_average_precision,0.25103841101104996

After: mean_average_precision,0.252120306364324

APEX

During discussions about potential avenues of improving performance outside of int 8 quantization came the idea of partial quantization and float 16 quantization. In the interest of providing additional comparaison data, some experiments were carried on partial quantization. One of the more promising tools that were found during research was the Mixed Precision training tool created by NVIDIA called APEX. The study of this tool led to a better understanding of the impact of weight precision and partial training on networks, but not without its fair share of hurdles.

The basic idea behind Mixed precision is that within a network, usually all operations are executed using float 32 values. However only a small portion of these operations extract a significant benefit from the high precision offered by FP32 values. However, reducing the precision of all the operations of the network below FP32 values can have a drastic impact on the performance of a network.

The idea of the mixed precision is to have some of the operations that require the highest precision run on FP32 values, while most of the operations that do not require such runs on FP16 values. Through this selective culling, the network should be able to achieve a performance almost equal to a FP32 training while having almost all of the memory benefits of FP16 quantized networks.

Operations that benefit from high precision, such as weight updates and large reductions, are left running on FP32, while the rest of the network runs on FP16 values. Mixed precision is helped within the pytorch backend due to the availability of the .half() command on modules and tensors, which changes their parameters from FP32 to FP16.

Apex proposes a library they call amp, for Automatic Mixed Precision. The idea is to transform a network from using float 32pt to float 16 using the principles previously described.

The system is implemented within the training loop, in much the same way quantized aware training would be implemented. The amp makes modifications to the model and optimizer to convert the different operations to FP16 or leave them to FP32. In order to do this, the amp makes a couple of assumptions on the network structures, what needs to be left at FP32 and what does not. Obviously this means that for a complex structure like mobile net, there would likely be the need for fine tuning, however this would remain an interesting alternative as the drop in accuracy would be minimal.

The quantization using apex took a bit of a backseat due to it acting on the training of the model, and by our orders and limitations, we would focus in priority on methods that would not require training of the model. However, the Basic steps to implement AMP within the training loop is easy, as such we have created a variant of Training_SSD to implement it. The process is similar to what must be done to implement quantization.

In order to work, the AMP function must be called upon the model and optimizer, upon which it will use a whitelist/blacklist system to implement the mixed precision. Due to the usage of float 16 numbers within operations, it is also important to apply a scaling factor during the backward pass in order to prevent underflowing gradients. Finally, it is recommended to edit the AMP configuration in order to tune it to the network currently being used.

While the configuration of AMP within the training SSD is complete, the debugging of the program is not yet completed, mainly as the platform used for testing was windows, and that APEX is still working on their compatibility branch for windows. On top of that, most efforts when it comes to development have been directed towards quantization, being the main objective of the project. However, if more metrics are necessary or if the hardware requirements expand slightly, optimizing and running tests on the AMP could prove beneficial.

6.4 - Metrics

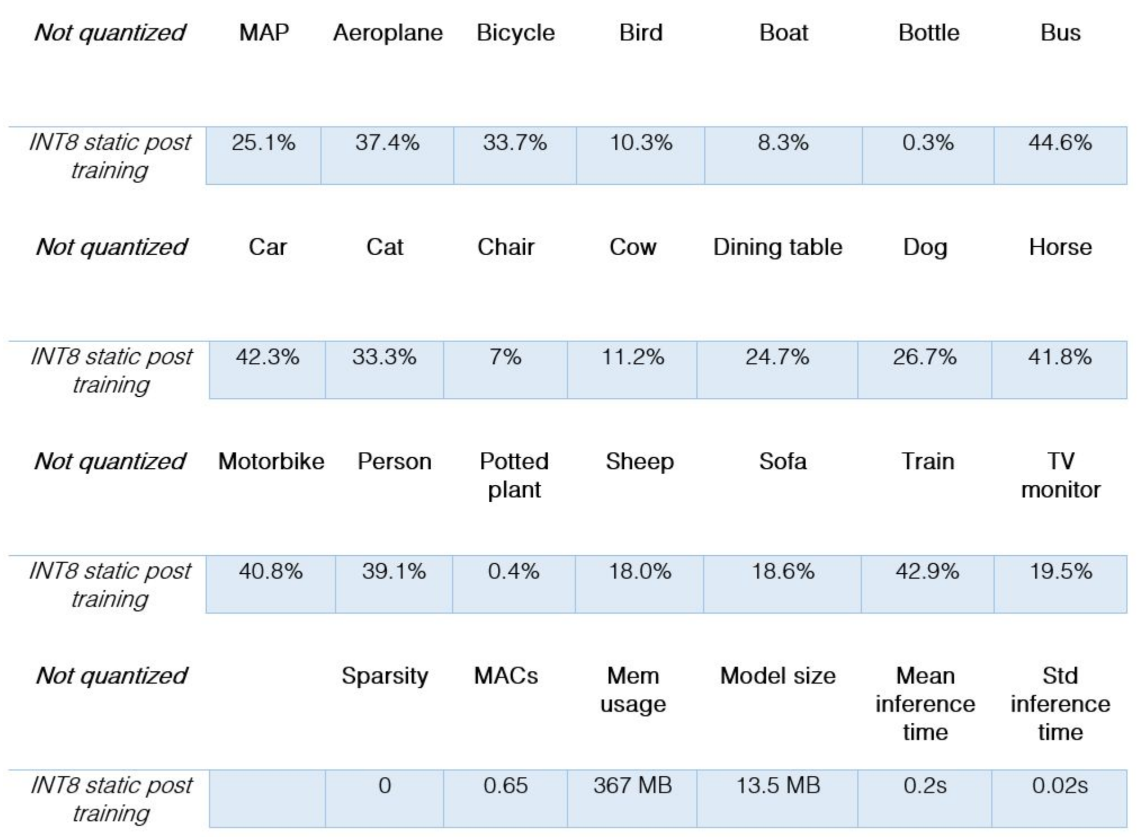
The main problems we faced during the creation of the pruning script was that some ways to calculate some metrics (MACs mostly) were not compatible with Windows. Some other functions were proved inaccurate (inference time, FPS). In order to get accurate FPS, we used two timers around the prediction.

Architecture changes

Due to the architecture changes that we instaured in order to make quantization work, the size of the input passed for the evaluation had changed . Therefore evaluating and getting the metrics of both the navya model and the google model was challenging. We solved that issue by using a backup file with its own version of the dataset.

VII - INFLUENCE OF QUANTIZATION

After modifying the architecture of the model, Navya trained a new model and provided us with the following metrics:



Not quantized	MAP	Aeroplane	Bicycle	Bird	Boat	Bottle	Bus
INT8 static post training	25.1%	37.4%	33.7%	10.3%	8.3%	0.3%	44.6%

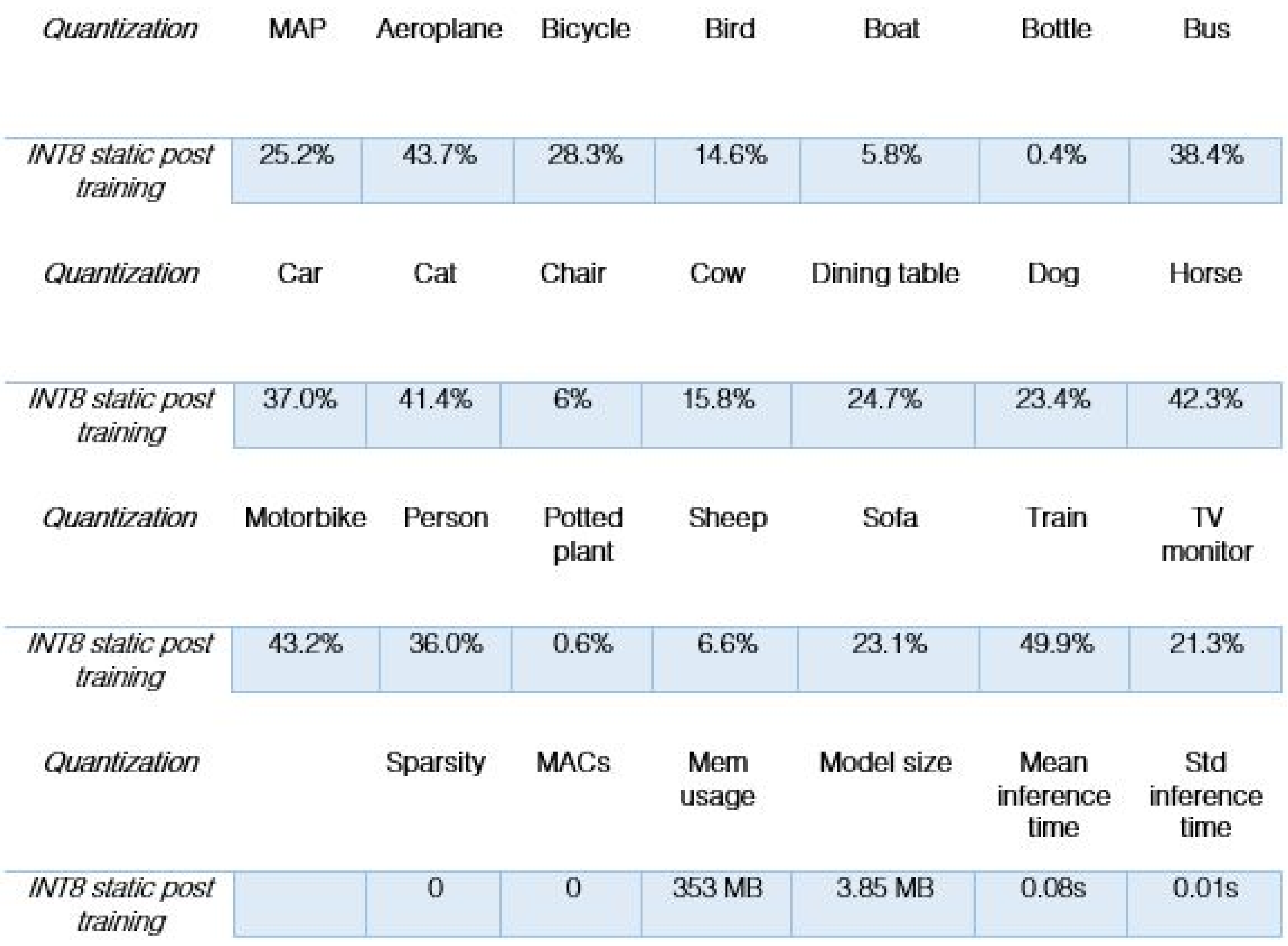
Not quantized	Car	Cat	Chair	Cow	Dining table	Dog	Horse
INT8 static post training	42.3%	33.3%	7%	11.2%	24.7%	26.7%	41.8%

Not quantized	Motorbike	Person	Potted plant	Sheep	Sofa	Train	TV monitor
INT8 static post training	40.8%	39.1%	0.4%	18.0%	18.6%	42.9%	19.5%

Not quantized		Sparsity	MACs	Mem usage	Model size	Mean inference time	Std inference time
INT8 static post training		0	0.65	367 MB	13.5 MB	0.2s	0.02s

Figure 20: Navya metrics unquantized model

We started In order to demonstrate the influence of INT-8 quantization on the Navya model (the google model proven to be unsuitable for quantization, cf 6.3) we have evaluated and calculated the following metrics after calibration :



Quantization	MAP	Aeroplane	Bicycle	Bird	Boat	Bottle	Bus
INT8 static post training	25.2%	43.7%	28.3%	14.6%	5.8%	0.4%	38.4%

Quantization	Car	Cat	Chair	Cow	Dining table	Dog	Horse
INT8 static post training	37.0%	41.4%	6%	15.8%	24.7%	23.4%	42.3%

Quantization	Motorbike	Person	Potted plant	Sheep	Sofa	Train	TV monitor
INT8 static post training	43.2%	36.0%	0.6%	6.6%	23.1%	49.9%	21.3%

Quantization		Sparsity	MACs	Mem usage	Model size	Mean inference time	Std inference time
INT8 static post training		0	0	353 MB	3.85 MB	0.08s	0.01s

Figure 21: Metrics quantized model

Instead of wanting to improve the model performance, we thought it was more interesting to measure the impact of quantization compared to the original model. We are getting very similar accuracy compared to the base model. ~25%.

The quantization evaluation was performed on Wassim's laptop, with MacOS as its exploitation system.

We can conclude on the influence of quantization: it reduces the model size, from 13.5MB to 3.85MB (divided by 3.5). It also increases FPS (x2.5) as it decreases inference time (FPS= 1/inference time). For the unquantized model, mean inference time =0.2 and Std inference time =0.02 meanwhile for the quantized model, mean inference time =0.08 and Std inference time =0.01. The accuracy remains the same ~25%.

We cannot conclude on memory consumption as we think the calculations are not accurate.

VIII - INFLUENCE OF PRUNING

In order to demonstrate the influence of pruning on the model accuracy, we have tested various levels of L1-norm pruning on the google pre-trained model

8.1 - Google pretrained model

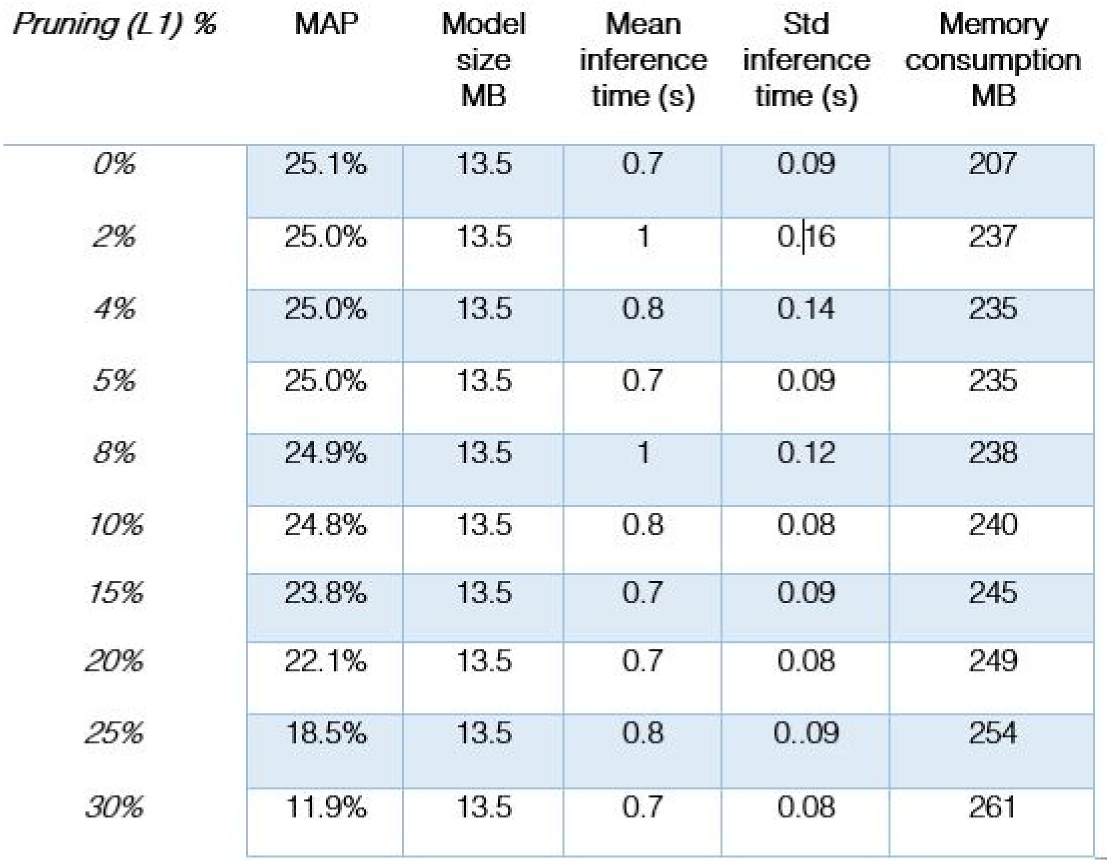
Pruning %	**Overall Accuracy**	**Car**	**Bicycle**	**Train**	**Person**	**Accuracy drop**
0%	68.7%	74.2%	77.8%	83.5%	71.6%	0
2%	68.6%	74.2%	77.7%	83.3%	71.6%	-0.1%
4%	68.6%	74.3%	77.7%	83.3%	71.5%	-0.1%
5%	68.8%	74.3%	77.8%	83.5%	71.6%	-0.1%
8%	68.8%	74.4%	78.0%	84.2%	71.6%	+0.1%
10%	68.7%	74.4%	78.3%	83.7%	71.6%	0
15%	68.5%	82.5%	78.1%	83.5%	71.6%	-0.2%
20%	68.2%	74.0%	77.1%	83.7%	71.3%	-0.5%
25%	67.4%	72.5%	77.1%	80.7%	70.6%	-1.3%
30%	65.9%	72.3%	75.7%	79.8%	69.8%	-2.8%
50%	6.3%	21.0%	19.1%	1.5%	12.0%	-62.4%

Figure 22: Accuracy drop when pruning, Google model

8.2 - Navya’s model

Pruning %	Overall Accuracy	Car	Bicycle	Train	Person	Accuracy drop
0%	25.1%	42.3%	33.7%	42.9%	39.1%	0
2%	25.0%	42.3%	33.6%	42.6%	39.0%	-0.1%
4%	25.0%	42.3%	33.6%	42.6%	39.0%	-0.1%
5%	25.0%	42.3%	33.8%	43.4%	39.4%	-0.1%
8%	24.9%	42.2%	34.0%	43.9%	38.9%	-0.2%
10%	24.8%	41.7%	33.3%	43.1%	38.6%	-0.3%
15%	23.8%	39.4%	32.3%	42.0%	37.4%	-1.3%
20%	22.1%	37.6%	29.3%	38.1%	35.2%	-3%
25%	18.5%	33.1%	23.0%	34.7%	30.1%	-6.6%
30%	11.9%	24.1%	16.5%	12.3%	24.6%	-13.2%

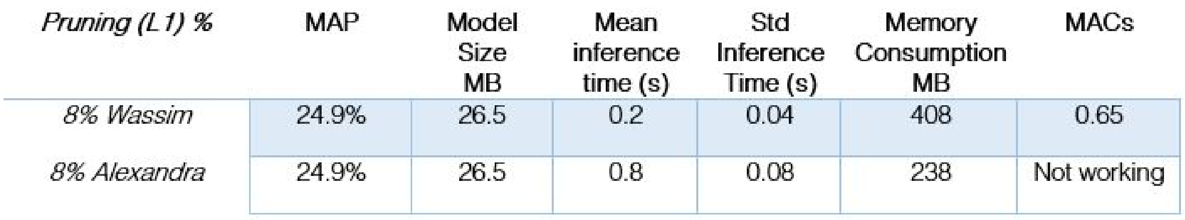
Figure 23: Accuracy drop when pruning, Navya model



Pruning (L1) %	MAP	Model size MB	Mean inference time (s)	Std inference time (s)	Memory consumption MB
0%	25.1%	13.5	0.7	0.09	207
2%	25.0%	13.5	1	0.16	237
4%	25.0%	13.5	0.8	0.14	235
5%	25.0%	13.5	0.7	0.09	235
8%	24.9%	13.5	1	0.12	238
10%	24.8%	13.5	0.8	0.08	240
15%	23.8%	13.5	0.7	0.09	245
20%	22.1%	13.5	0.7	0.08	249
25%	18.5%	13.5	0.8	0..09	254
30%	11.9%	13.5	0.7	0.08	261

Figure 24:Metrics, Navya model, Alex's laptop

The quantization evaluation was performed on a laptop, with Windows as its exploitation system. We noticed that Windows had a negative influence on some metrics, Mean inference time, std inference time and MACs. We ran the same evaluation process on Wassim's laptop and the inference time and MACs were fixed



Pruning (L1) %	MAP	Model Size MB	Mean inference time (s)	Std Inference Time (s)	Memory Consumption MB	MACs
8% Wassim	24.9%	26.5	0.2	0.04	408	0.65
8% Alexandra	24.9%	26.5	0.8	0.08	238	Not working

Figure 25:Metrics, Navya model, Wassim's laptop, before fixing the model size

In order to correct the size of the model doubling, we have done re parametrization, which removes the mask off the model without removing the output and so it lowered the model size to 13.5.

We can conclude on the influence of pruning: it does not increase nor decrease the model size, it stays at 13.5MB, if we remove the pruning mask off the network..

It also increases FPS (about x 3.5) as it decreases inference time (FPS= 1/inference time) (if run on something else than windows). For the 0% pruned model, mean inference time =0.7 and Std inference time =0.09 while for 8% (figure 25) mean inference time =0.2 and Std inference time =0.04.

When the pruned models are loaded from a different script, the memory consumption drops around 101~100 MB. (divided by 2)

IX - TEAM ORGANISATION

We have created three sub teams in order to divide the work :

Team one: Wassim, Alexandra
Team two: Prashant, Abhishek
Team three: Laurina, Adrien

Team two and three report their progress to team one and team one planned the next steps and coordinates with Navya. Wassim was in charge of git management and Alexandra of evaluating the models. Each team had a major task and assisted the others in times of need. Team 1 was in charge of pruning, team 2 quantization and team 3 of experimenting with the Apex tool.
Team two also designed the metrics script but team one had to modify it heavily in order to perform evaluation correctly. Team two was assisted by wassim while implementing quantization and calibration.
Every new function with no computation bug was pushed to git and waited for merging from either ravi or wassim.

X - FUTURE WORK

In future experiments, we could try to implement another pruning method or another type of quantization, and compare to our current results as well as the pre-trained model. We could also update the architecture on V3 as it was proven to be better inference wise and size wise. If the current architecture satisfies the constraints, we could also try to implement it into a raspberry pi and run tests on it.

XI - SOURCES

Annexe 0 : Git ressources

https://github.com/qfgaohao/pytorch-ssd

https://pytorch.org/blog/introduction-to-quantization-on-pytorch/

https://github.com/pytorch/vision/blob/master/torchvision/models/mobilenet.py

Annexe 1 : Neural networks

http://neuralnetworksanddeeplearning.com/chap1.html

Annexe 2 : Quantization

[1]https://nervanasystems.github.io/distiller/quantization.html
https://jackwish.net/neural-network-quantization-introduction.html

[2] https://nervanasystems.github.io/distiller/algo_quantization.html

https://chromium.googlesource.com/external/github.com/tensorflow/tensorflow/+/r0.10/tensorflow/g3doc/how_tos/quantization/index.md

[9] https://nervanasystems.github.io/distiller/algo_quantization.html

[11]
https://medium.com/apache-mxnet/model-quantization-for-production-level-neural-network-inference-f54462ebba05

[12] https://pytorch.org/blog/introduction-to-quantization-on-pytorch/

Annexe 3 : Embedded neural networks

[3] https://www.quantmetry.com/neural-networks-embedded-systems/

https://paperswithcode.com/sota/object-detection-on-bdd100k
https://paperswithcode.com/

[10] https://geoffpleiss.com/nn_calibration

Annexe 4 : State of the art, current researches and new features in embedded networks

http://www.laas.fr/files/LAAS/40ans_LAAS-CNRS_Sifakis_Conf.pdf
http://eurekajournals.com/embedded.html https://www.ccs-labs.org/research/
https://www.aemc.gov.au/sites/default/files/2019-06/Updating%20the%20regulatory%2
0fr ameworks%20for%20embedded%20networks%20-%20FINAL%20REPORT.PDF
https://www.lexology.com/library/detail.aspx?g=f33bf0ba-e68e-4094-be5e-e70abaa900
a7 https://warwick.ac.uk/fac/sci/eng/staff/wg/research/
https://www.claytonutz.com/knowledge/2019/september/aemcs-final-report-confirms-ex
ist ing-embedded-electricity-networks-will-be-caught-by-the-new-regulatory-regime
https://utilitymagazine.com.au/a-new-rule-for-embedded-networks/
https://www.nortonrosefulbright.com/en/knowledge/publications/b3fed4d1/embedded-n
e tworks---significant-regulatory-changes-occurring https://machinethink.net/
https://www.cisco.com/c/en/us/solutions/internet-of-things/iot-embedded-services.html

Annexe 5 : MobileNetV2

[4] https://arxiv.org/pdf/1801.04381.pdf

Annexe 6 : Embedded networks' rules

https://www.aemc.gov.au/rule-changes/embedded-networks
https://www.aemc.gov.au/sites/default/files/content/04b21dd0-521c-48ca-b575-6fbe6b
736 a37/Information-sheet.pdf

Annexe 7 : Metrics

[5]
https://medium.com/@jonathan_hui/map-mean-average-precision-for-object-detection-
45c121a31173

https://towardsdatascience.com/breaking-down-mean-average-precision-map-ae462f6
23a52.

Annexe 8 : Pruning

[6] https://arxiv.org/pdf/1608.08710.pdf

[7] https://towardsdatascience.com/l1-and-l2-regularization-methods-ce25e7fc831c

[8]
ttps://heartbeat.fritz.ai/research-guide-pruning-techniques-for-neural-networks-d9b8440ab10d