POLITECNICO DI TORINO

Master's Degree course in Electronic Engineering

Master's Degree Thesis

Tensorflow-driven neural network quantization for resource-constrained devices

Supervisors

prof. Luciano LAVAGNO

prof. Mihai Teodor LAZARESCU

Candidate

Luca BATTIATO

ID:265602

Academic year 2019/2020

Summary

Nowadays the Machine Learning has a primary role in a lot of aspect of our life. The model family increasingly has been enriched by new solutions and algorithms, making possible to generalize very difficult tasks, from Computer Vision to Medical, from Natural Language Processing to Gaming, just to mention some.

In particular, the usage of Image classification and Object Detection have increased significantly and have brought out new challenges and new opportunities. It is common by now to find these two specific ML applications in "Mobile environments", such as cars or smartphones. Often mobile devices can not rely on unconstrained resources, like large-scale distributed systems of hundreds of servers with computational devices such as GPUs, and main power grid supplies. Hence, Neural Networks must be optimized for the available resources, memory, energy consumption and processing capabilities, while providing high throughput and high accuracy. The attention is now shifted to minimize and to streamline the most promising neural network models, that in general are large and deep, with minimum performance loss.

The aim of this thesis work is to explore some optimization techniques applied on different neural network models and to analyze their performance against the baselines. In particular, will be analyzed some discretization techniques, thanks to which it is possible to quantize the model's variables on reduced numbers of bits. The techniques have been applied to some neural network models from a research project at DET called "Indoor human localisation using Capacitive sensing". The models used belong to three different neural network architectures with the very same aim: to infer and track the position of a human inside a room using the data coming from capacitive sensors. Previous works explored the best neural network architectures and optimized their training. These promising results would need to be deployed inside an embedded system like an FPGA or a microcontroller to optimize the localization system. But since these systems are resource-constrained

devices, the neural networks have to be optimized. The Quantization Technique explored in this work will achieve memory and processing requirements optimization, reducing the size of the variable representations and allowing hardware accelerations. In particular, by representing all the network data on a few bits, are optimized the multiply-and-accumulate operations, that represent the main operations inside a Neural Network implementation. The data size reductions can bring significant improvements in terms of throughput, since more operations can be executed in parallel inside the same arithmetic unit, or can be used smaller arithmetic units saving energy and resources, two crucial aspects of Embedded Systems Design.

The techniques chosen to perform the quantization came from the *Tensorflow Model Optimization* module provided by the Tensorflow API and the *Tensorlow Lite* environment. These specific modules offer different optimizations, but only two are taken in consideration in this work: *Quantization Aware Training* (QAT) and *Post-Training Quantization* (PTQ). The approach followed to produce the experimental results is straightforward: the two techniques have been applied for every neural network that was performing best in the previous experiment, and the results obtained have been compared with the results collected in the previous work. In particular, QAT let us choose the number of bits in the weight and activation representations and so it has been explored the performance degradation for many quantization numbers.

The neural networks used belong to these architectures:

- Multilayers Perceptron (MLP)
- 1D-Convolutional Neural Networks (CNN)

Long Short-Term Memory networks were also investigated, but it was not possible to apply any discretization tecniques becouse Tensorflow optimization modules do not yet support Recurrent Neural Networks. The results collected are promising since the Average Euclidean Distance Error (ADE), the comparison metric between the model using floating point representation on 32 bits, already trained, and the quantized integer version, shows minimal losses for the CNN, within the 5%, and even a performance improvements by around 10% for the MLP. The remarkable improvements in terms of resource usage and the promising results in the accuracy demonstrate the advantages of using these optimization techniques with those networks.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In order to trace the movements of a person inside a room, it has been chosen to use low-cost capacitive sensors and suitable Machine Learning algorithms to covert the noisy and environmental-dependent data coming from the sensors into the position and trajectory of the person. In the following, a brief analysis of the Neural Networks used and their functionalities is presented, followed by the previous work conducted and the thesis contribution.

1.1 Machine Learning

The machine learning activities are increasingly growing in a lot of technological and health environments. This thesis work rely on advance optimization techniques that can improve some aspects of the inference efficiency of the model. These techniques are strictly connected with the underneath environment that define and actually makes working a Neural Network, the Machine Learning algorithms precisely. Hence, in order to proceed with the analysis and usage of these optimization techniques, it is important to know and understand the basic concepts behind this branch of Computer Science.

1.1.1 Basic knowledge behind Machine Learning Algorithms

The Machine Learning actually is a set of mechanisms that can make improving the performance of a machine during time. A first formal definition of this term has been introduce by Arthur Samuel in 1959, inside an article titled *"Some Studies in Machine Learning Using the Game of Checkers"* [1].

The mechanism that regulates the learning process can be divided into two different approaches as explained in [2]:

- **Supervised learning**: the Model is fed with x_i inputs and y_j outputs samples and it has to derive the function f that with those inputs can reproduce, as precisely as possible, the outputs provided as $f(x_i) = y_j$. The output producds can be of two different types, depending on the task to be addressed:
  - classification tasks, where the outputs produced are discrete and finite values well defined at start, or continuous values in the form of a probability for a specific class label. Typical applications can be the *Email spam detector* or the *Plant species classification*, as an example.
  - regression tasks, where the outputs produced are continuous variables that represent quantities such sizes or amounts. Typical applications can be *House pricing prediction* or *Speech recognition*, as an example.
- **Unsupervised learning**: the Model is fed only with input values x_i and it has to extrapolate the underneath structure of the data set. The output produced can be of two forms:
  - clustering type, when the input data are grouped in clusters that share some regularities. Typical applications can be in the Health branch, like *Tumor classification*.
  - non-clustering type, when the underneath structure has to be found in the entire data set.

The *Supervised learning*, used in this thesis, is performed with two subsequent phases:

1. **training**: in this step the learning algorithm is fed with the input samples and the output is adjusted iteratively till convergence, if the problem is well constructed. During this phase can be provided also a *validation data set* to check the learning trend.
2. **testing**: in this step the Model trained is tested against a *test data set* that the Model has never seen before. In this way the effectiveness of the algorithm to generalize the function f is checked.

The learning algorithm, in order to improve its output accuracy predictions, uses a *cost function* or *loss function* that provides a value of how much the output predicted is wrong w.r.t. the ground truth output labels. During several steps of training, the algorithm minimizes this function J_{train}, actually bringing the predicted output closer to the target output by tuning its inner *trainable variables*. In this phase, it is important to check the quality of the learning with the *validation data set*, a restricted data set used only for this purpose, to avoid two particular generalization issues:

- *underfitting* the data, e.g. when the complexity of the problem is badly generalized and both the error on the training and validation data set are high. To avoid this situation can be enlarged the Network or trained longer, or migrate it to a different architecture.

- *overfitting* the data, e.g. when the model is too complex w.r.t. the problem to generalize and, even if the error on the training data set is low, the error on the validation data set is high and they are not following a similar trend. To avoid this situation can be used more data during the training or some *regularization* techniques.

The optimization algorithm is in charge to minimize the *loss function* by tuning the variables, like the weights and the bias, basing its actions on the *gradients* computed by the backpropagation algorithm, a special function that derives the loss function against each trainable variable, from the output to the input. The change applied to each variable are fixed by the *learning rate*, an important hyper-parameter that actually controls how much the algorithm can modify the weights from one iteration to an other.

There are several types of *built-in loss functions* and also custom ones can be used, but in the thesis work will be used the *Mean Squared Error* or MSE, defined as follow:

$$MSE = \frac{\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}{N} \tag{1.1}$$

where y represents the target output, the $\hat{y}_i$ is the predicted output and N is the number of samples provided.

Especially during the test phase, some metrics can be used to control the model accuracy, depending of what the output represents and what task is addressed. As better explained in Section 1.2, the models used inside this thesis work are able to localize a person inside a room. In particular, they take four values as input and they provide two output values representing the X and Y coordinates of the person. The metric used to evaluate the final Model accuracy is the *Average Euclidean Distance error*, defined as follow:

$$ADE = \frac{\sum_{i=1}^{N} \sqrt{(x_i^{pred} - x_i^{ref})^2 + (y_i^{pred} - y_i^{ref})^2}}{N} \tag{1.2}$$

where x_i^{pred} and y_i^{pred} are the predicted trajectories points, while x_i^{ref} and y_i^{ref} are the target ground-truth trajectories points, all in X-Y coordinates. This metrics actually represents how much the output are far from the target in m along a trajectory. The usage of this particular metric together with the MSE is useful to recognize the Models that are generalizing well the task proposed.

1.1.2 Neural network architectures

Neural Networks used in the following are a particular family of machine learning algorithms and can address several tasks in many different branches. Their structure is similar to the brain model and with it they share also some terminologies. In fact, as inside a brain, the basic element of every neural network is the neuron, the computational unit that takes the input (dentrites) and produces the output (axons). Those output are transmitted to other neurons till a network is formed. In Figure 1.1 is shown the typical neuron inside a neural network with three input and one output.

The functional role of each term, from left to right, is the following:

- The inputs x_i represents the numerical values coming from the external world, if this neuron is in the first layer, or from other neurons of the Network.
- The weights w_i represents the "amount of attention" to reserve to the associated input x_i
- The *Node* is the computational part and performs two different operations in the following order:
  1. the "multiplication and accumulation" between each input and their weights and the biasing of the result with the b term, unique for every node of the Network.

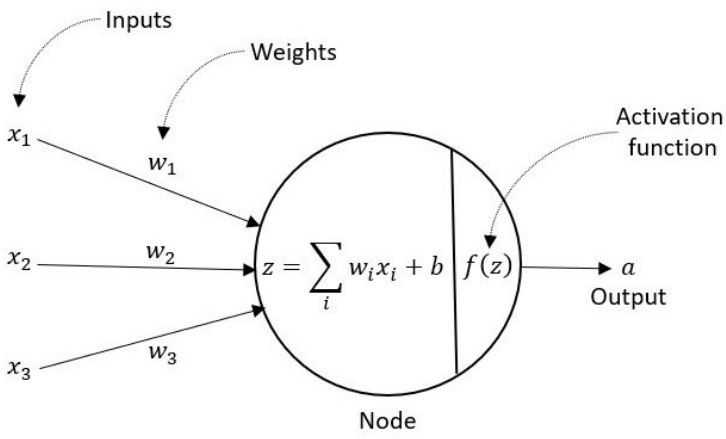


Figure 1.1: Typical neuron shape inside a neural network algorithm from [3]

2. the non-linear Activation function $f(z)$ applied to the previous result, like *sigmoid* or *ReLU*

- the output a propagation

The importance to use non-linear function at the end of each node is justified by the typical problems to be generalized, that are non-linear by definition, as reported in [3].
When several of those basic blocks are stacked toghether to form a network, it is common to call the resulting network *Multilayer perceptron*. The main characteristic of these architectures is the grouping of some neurons inside a so-called *hidden layer*. As shown in the Figure 1.2, the number of layers and neurons inside each one, are design parameters and can change the Network performances.

1.1.3 Convolutional neural network architectures

An other family of Machine Learning algorithm are the Convolutional Neural Networks, used in the thesis work and analysed in this section.

Their usage is more related to tasks that need to produce the output basing on sequence of input or input that are related with each other with some spacial or temporal relationships. Their architecture is then perfect

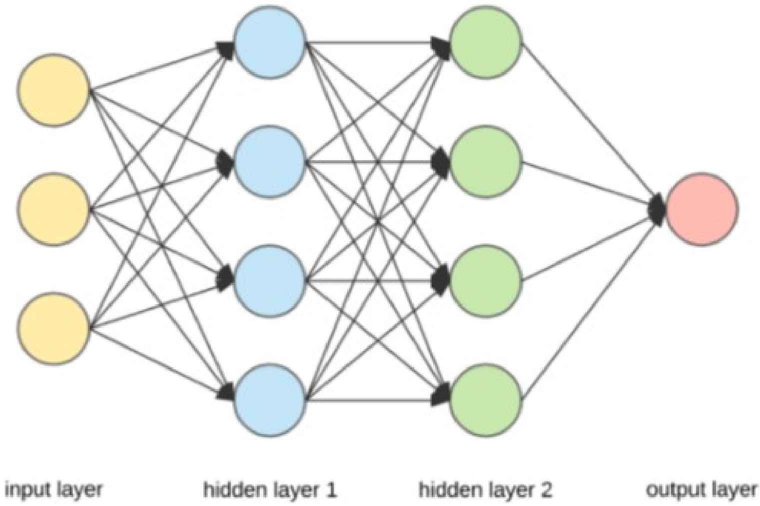


Figure 1.2: Multilayer perceptron example layout and connections between nodes

for image-driven pattern recognition tasks. As explained in [4] and [5], the *convolutional* mathematical operation used in some layers gives the name to this networks. Their backbone is made by several hidden layers stacked one on the other that can be:

- *Convolutional layer*: this is the layer that performs the scalar product between the weights and the region connected to the input volume, hence no more one input value per weight but several input values that can be seen as a volume. The typical usage is with 2-D input shapes, but can be also used with 1-D, in this case it is called *temporal convolutional layer*.

- *Pooling layer*: inside this layer the inputs are downsampled to reduce the number of parameters that otherwise would grow exponentially. There are several types of pooling layers, the most common is the *MaxPool* that actually takes the maximum value in a certain input region and creates a new output matrix with those values.

- *Fully-connected layer*: this is the same layer explained in 1.1.2, with typical several neurons inside, that produce class scores to be used for classification.

As an example, the Figure 1.3 shows the common shape of a CNN architecture, with several layers stacked together. The width and height of each square represented are the actual dimensions of the matrices produced by that layer.

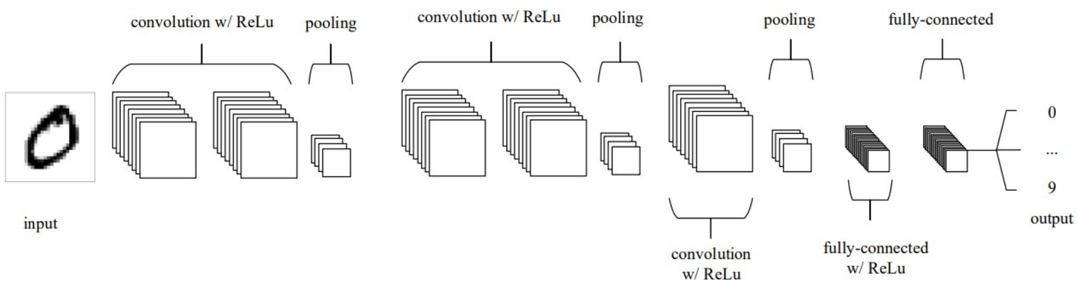


Figure 1.3: Commmon structure of a Convolutional Neural Network for the MNIST recognition provided by [5]

The main advantage coming from the usage of this kind of Networks is the good generalization of complex task but with higher computational cost due to the quick-growing number of activations to compute during filtering with CNN layers.

1.1.4 Long-Short Term Memory architecture

This kind of neural network belongs to the borader family of the Recurrent Neural Networks. Their main characteristic is the ability to find correlations inside a stream of data and for this reason are widely used in applications like speech recognition or time series forecasting.

The unfolded version of this particular network in Figure 1.4 shows the reuse of the same weight for subsequent data input. In fact, each node takes as input the new incoming data together with the activation of the previous data, always with the same function h, and so with the same weights and bias.

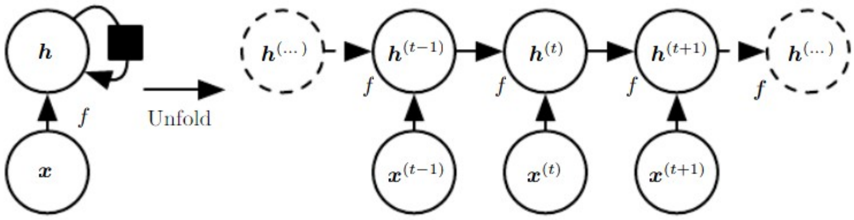


Figure 1.4: Recurrent Neural Network structure in the unfold version from [6]

A typical application of RNNs is the Long Short-Term Memory, as reported by [6]. This particular Networks are also called Gated due to their weighted paths that link the cell to other input backward in time, as shown in Figure 1.5. The main gates used are:

- Input Gate, allowing the accumulation of the input into the node state.
- Forget Gate, controlling the weight of each input in the inner self-loop.
- Output Gate, forcing the cell output to extinguish if needed

These cells can be stacked together to controls wider input windows.

1.2 Previous work

This thesis work fits in a wide project that combines different knowledge from different branches, whose aim is to develop an indoor human localization system for assisted living people. The entire system, represented in Figure 1.6 and explained in [7], is able to measure the capacitance values seen from each of the four Plates with an oscillator, then to send these values to the base station trough wireless connection and to post-process the received data. In the end, thanks to a Machine Learning algorithm, the position of the person inside the room is inferred.

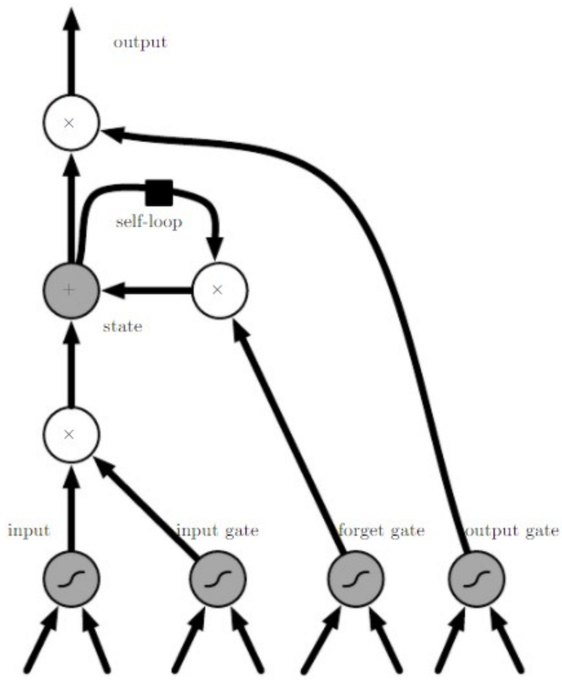


Figure 1.5: Typical usage of one LSTM cell with the gated weights from [6]

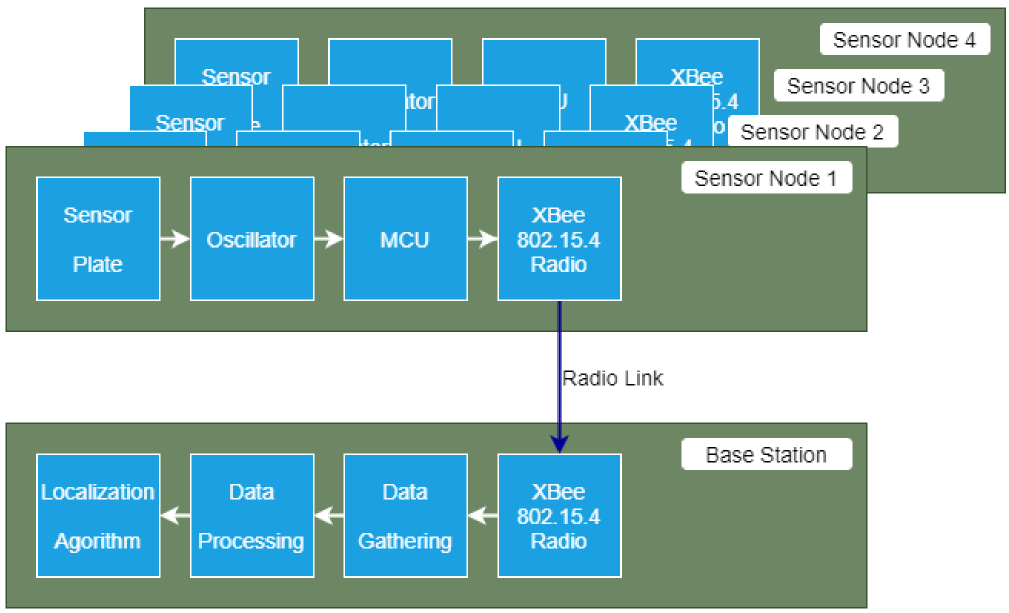


Figure 1.6: Main macro blocks for the system extrapolated from [7]

Several researches have been conducted in each of these macro blocks in order to find the best implementation or the feasibility of some approaches.

The movement of a person is tracked by the system by an indirect measure of frequency with a timer-based oscillator, that translates the capacitance value seen by the plate. Moreover, different kind of Machine Learning algorithms have been tested, in order to prove that this approach can be suitable to bring good results in the inference of the person position, due to the high environment noise brought in by the Capacitive sensors [7].

In this direction it has been conducted a deep Design Space Exploration by [8] to find the best Neural Network architecture and layout to pursuit the inference. The Models and their parameters are summarized in the Table 1.1. All of these NNs have been trained and validated with a data set collected using the four capacitive sensors system for the input labels, and an ultrasound system with four sensors to produce the most accurate ground-truth X-Y tuples, as reported by [9] and shown in Figure 1.7.

Each Model it has been trained using Tensorflow v2.2 API and its tools, like Keras, that actually make simpler the Neural Network development [10].

The provided Models are completed with the architecture layout and parameters in *.py* format, the weights trained in *.h5* format and the performance achieved with several *.csv* documents.

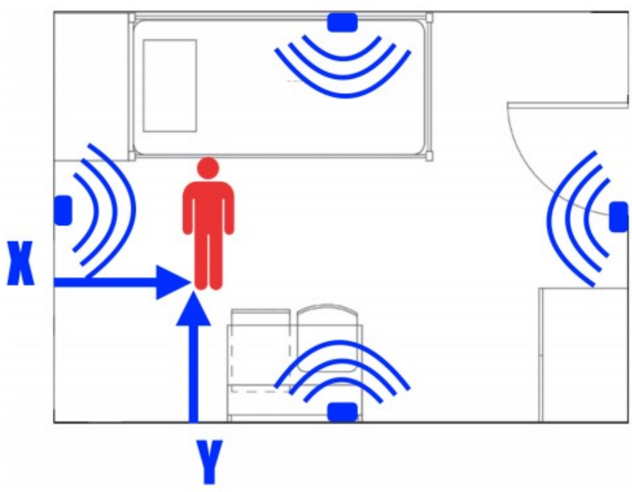


Figure 1.7: The four capacitive sensors positions inside the room during data collection reported by [7]

Architecture	Main parameters	Best MSE(1) (m^2)	Best ADE(2) (m)	Complexity (Num. of param.)
MLP	1 hidden layer, 32 neurons, Adamax optimizer	0.125	0.433	226
CNN	2-layer 1D-CNN, L2 regularization, 10 sec window size, 32-32 Neurons MLP, 3x1 Kernel size, 64 filters, Adamax optimizer	0.047	0.273	34818
LSTM	1-layer 2 neurons LSTM, 10 sec window size, Adamax optimizer	0.056	0.301	62

(1): *Mean Squared Error*, explained in 1.1
(2):*Average Euclidean Distance Error*, explained in 1.2

Table 1.1: Summary table of all the "best models" architecture trained and reported by [8]

1.3 Thesis contribution

The aim of this thesis work is to test whether the Discretization Techniques can bring good results, even if the Model has lost representational precision.

Among all the Optimization techniques, only the *Quantization-aware Training* and *Post-training Quantization* will be used and tested against some Models. The reason why to explore this optimization area is to understand how the performance degrades while using Discretization, in order to better know how to deploy in the future the Best Models for "Indoor human localization". In fact, all the Models that have been used during the development of this Thesis come from a pool of "best-performing" Neural Networks designed and reported by [8] and summarized in Table 1.1. The aim of all the Networks is to infer the exact position of a person inside a room of $3m \times 3m$ using capacitive sensors, as described in [7]. In the end the entire system will be deployed on a device which will define the available hardware. Since this device will be an embedded system, it is important to maintain the computational task under control, in particular avoiding those arithmetic operations that could be too expensive to a limited device, like big matrix multiplications and accumulations. In order to optimize this particular aspect, the quantization of a neural network is the only method that can address this particular requirement.

The radical differences in the neural networks archietctures used suggest to apply the optimization techniques chosen one model at a time, in order to collect the results and to compare them against the un-optimized ones. Especially for the *Quantization-aware Training* method, since the model has to be re-trained, all the main training hyper-parameters have been left as reported by the previous work [8]. In this way, it is possible to compare more closely the results and to effectively see the perfomance trend. In particular, the metric used to judge the optimized model accuracy is the *Mean Squared Error* (MSE) and the *Average Euclidean Distance Error* on the same dataset used with the un-optimized ones.

In the folowing parts, firstly will be presented more in detail the Discretization Techniques selected, how they are used and how they work, and then will be shown the optimized models results, in terms of accuracy achieved for hyper-parameters used.

Chapter 2

Optimization Techniques

The inference efficiency is a critical concern when it is necessary to deploy a Machine Learning model in edge-devices, like mobile and Internet of Things applications. In these cases it becomes crucial other important parameters like latency, memory utilization and also the power consumption due to the limited device's resources.

Following there will be a small summary about some optimization techniques that can address the previous concerns and which and why some of them have been chosen in the development of the thesis work. All of these techniques are available in the Tensorflow v2 API. Then, the methods selected will be further analysed and discussed, in order to explain their practical usage.

2.1 Model optimization techniques to control the inference efficiency

The main techniques adopted to optimize a model can operate in different ways and directions, basing on which part of the model has to be improved or modified to fit some constraints. The covered areas are:

- Reduce parameter count
- Update the original model topology to a more efficient one with reduced parameters or faster execution
- Reduce representational precision

The first two techniques are more related to the topology of the model and their aim is to optimize how the graph is built and how much parameters, e.g. weights, are necessary to maintain or even improve the model accuracy. The last one is a more intrusive technique that allows the developer to modify the structure of the model, migrating to its quantized version.

2.1.1 Sparsity and Pruning technique

The main goal of this technique is to trim the insignificant weights inside the model, inducing sparsity in a deep neural network. The sparsity regularization inside a neural network is nowadays very common, since it can address alone a sensible reduction in the computational cost with no to minimal loss in the accuracy.

As reported by [11], there can be different ways to effectively prune the redundant variables inside a deep neural network. As the Figure 2.1 shows, this technique can act on the single nodes, e.g. removing from the network all the connection of that node, or on some connections between different nodes, e.g. masking some weigths inside the neuron.

The Pruning Technique provided by Tensorflow allows to act on the weights, and so inducing connection sparsity. At the end, when the desired level of sparsity has been reached, the model will show significant improvements in terms of Model compression and latency during inference mode, since all the zeros parameters can be skipped in the calculations. The performance of two well-known networks, as reported by [12], are summarized in the Table 2.1. As it can be seen, this technique is very powerful with larger networks due to the high number of unnecessary weights that can be easily stuck-at zero.

The compression factor can reach 6x against the normal model, saving a lot of memory space.

	InceptionV3		MobileNetV1 224	
Sparsity	Accuracy	Non-Zero Parameters	Accuracy	Non-Zero Parameters
0%	78.1%	27.1M	70.6%	4.21M
50%	78%	13.6M	69.5%	2.13M
75%	76.1%	6.8M	67.7%	1.09M
87.5%	74.6%	3.3M		

Table 2.1: Performance degradation of InceptionV3 and MobileNetV1 224 tested against Imagenet as reported by [12]

The better accuracy achieved between large-sparse models and small-dense ones with the same memory footprint demonstrates that this technique can bring better results in terms of parameters reduction with minimal loss in accuracy.

The usage of this optimization is very easy since is embedded in Tensorflow Model Optimization Module and can be tuned with very few hyper-parameters. Accordingly to the scale of sparsity to achieve, the pruning will be applied during training epochs on the Model's layers and each weight too close to zero will be masked and removed after the forward step. This will be applied gradually during the training, in a even less aggressive way during the epochs, interleaving some training epochs without pruning further to allow the model to recover from the killed weights. The Figure 2.2 shows the trend of the sparsity growth during a training model session and how the learning rate decays gradually till the very fine-tuning in the last epochs.

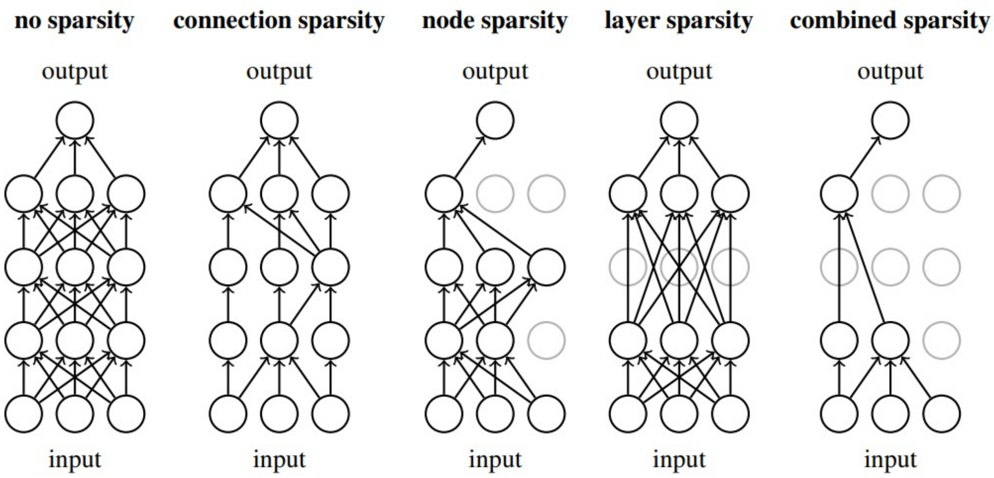


Figure 2.1: Feed-forward step inside Neural Networks with different types of sparsity level reported in [11]

2.1.2 Weight sharing technique

The weight sharing technique inside Tensorflow Model Optimization module allows the developer to group the weights in a neural network reducing the number of unique weights. In this way the optimized model will be compressed further, resulting in a thinner model that can be deployed in

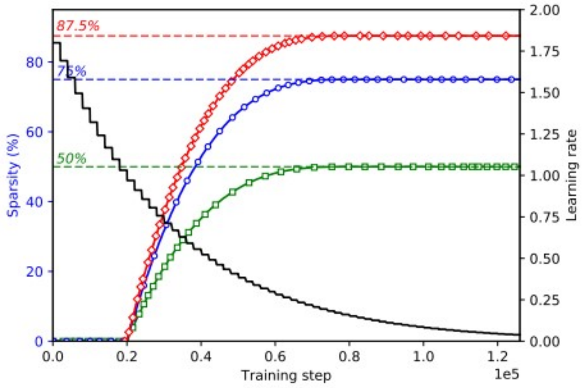


Figure 2.2: Sparsity and learning rate trends during taining steps for sparse-InceptionV3 model reported by [12]

such devices with limited resources. As reported in [13] this technique can be even more powerful if combined with weights quantization and can lead to significant improvement in terms of memory occupied. These promising results are collected in the Table 2.2. The models used are well-known neural networks and have been trained with different number of clusters and different number of layers selected.

Original		**Clustered**			
Top-1 accuracy	Size of compressed .tflite	Configuration	# of clusters	Top-1 accuracy	Size of compressed .tflite
MobileNetV1					
71.02	14.96	Selective (last 3 Conv2D layers)	256,256,32	70.62 (-5.6%)	8.42 (-43.7%)
		Full (all Conv2D layers)	64	66.07 (-7.0%)	2.98 (-80.1%)
MobileNetV2					
72.29	12.90	Selective (last 3 Conv2D layers)	256,256,32	72.31 (+0%)	7.00 (-45.7%)
		Full (all Conv2D layers)	32	69.33 (-4.1%)	2.60 (-79.8%)

Table 2.2: Performance degradation of MobileNetV1 and MobileNetV2 tested against ImageNet as reported by [13]

Weight sharing limits the number of effective weights needed to store by forcing multiple connections to share the same weight, and then fine-tune those shared weights during the back-propagation. The compression rate r grows as the cluster k factor decreases, with n connections inside the Network represented by b number of bits, as shown in (2.1) below.

$$r = \frac{nb}{n\log_2(k) + kb} \tag{2.1}$$

Even if [13] reports the usage of this technique together with the weights quantization, the only requirement is that the model has to be already trained and so its weights are already representing a proper value. In fact, the technique uses the centroid value of the weights cluster to fix the shared weight value, and then applies this value during the feed-forward and fine-tunes them during back-propagation, as shown in Figure 2.3. The weights inside the same layer are grouped together using the *within-cluster sum of squares* minimization (2.2), hence the weights w closer to the cluster value c_i will belong to the same cluster.

$$min\sum_{i=1}^{k}\sum_{w\in c_i}|w - c_i|^2 \tag{2.2}$$

It is also important how the centroids values are initialized in order to avoid clusters initialization in "unfair" points that can bring less accuracy. Figure 2.4 shows how the proposed initialization methods works with a representative weights distribution. Since larger weight absolute values play a more important role than smaller ones, the two over three methods analysed, forgy and density-based initialization, produces very few centroids with large absolute value which results in poor representation of these few large weights and so in the accuracy. Instead, linear initialization does not suffer from this problem as reported in the experimental results in [13], since it span linearly in the $[min, max]$ weights range.

2.1.3 Quantization techniques

As reported in [14], the Tensorflow Module from which are taken these techniques, there are two quite different approaches to effectively make a Neural Network quantized among a fixed number of bits: *Post-training quantization* and *Quantization-aware training.*

Post-training quantization technique

The easiest and quickest one to use is the post-training quantization, since it only needs the already trained model and its weights. It is able to

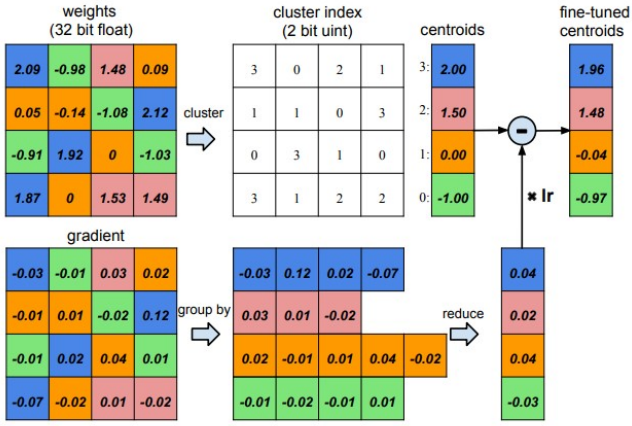


Figure 2.3: Weight clustering and fine-tune during back-propagation reported in [13]

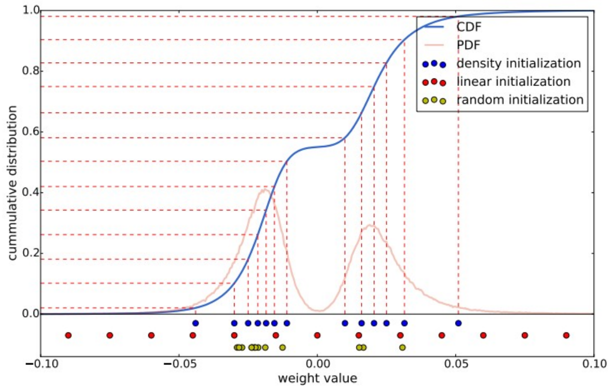


Figure 2.4: Three different methods to initialize the centroids with a representative weights distribution

reduce CPU usage and hardware accelerator latency, processing time, energy consumption and model size, with little degradation in model accuracy using different quantization modes: *Dynamic Range quantization*, *Full integer quantization* and *Float16 quantization*. As reported by [15], these different types of quantization ensure the performance collected in Table 2.3. The "data requirements" entry represents the need to supply to the method a representative sample of the input data. In this way the quantization will be more accurate and will bring better results in terms of accuracy.

Technique	Data requirements	Size reduction	Accuracy
Float16 quantization	No data	Up to 50%	Insignificant acccuracy loss
Dynamic range	No data	Up to 75%	Accuracy loss
Full integer	Unlabelled representative sample	Up to 75%	Smaller accuracy loss

Table 2.3: Performances summary for different Post-training quantization types

The way to choose the right modes is related to the characteristics of the model and the performance to achieve. If the Deployed Device supports Float32 operations, i.e. it can rely on a GPU, the *Float16 quantization* will quantize all the parameters to Float16 and the other operations will remain to Float32. In this way the computational cost is reduced with negligible accuracy losses.

On the other hand, in order to migrate to integer operations only, if there is a representative data set of the inputs, the model can be effectively limited to Int8 operations, with both weights and activations quantized to integers thanks to *Full integer* mode. Finally, if it is not possible to ensure that all the operations can be limited to integer only, the *Dynamic Range* mode will track at inference, with floating point variables, the trend of the weights and will implement those unsupported operations using Float32. The output will be still stored using Float32 values.

The technique has been applied to well-known models by [15] and the results have been collected in Table 2.4.

Model	**Top-1 Original** Accuracy	**Top-1 PTQ** Accuracy	**Latency Original** (ms)	**Latency PTQ** (ms)
MobleNetV1-1-224	0.709	0.657	124	112
MobileNetV2-1-224	0.719	0.637	89	98
InceptionV3	0.78	0.772	1130	845
ResNetV2-101	0.770	0.768	3973	2868

Table 2.4: The benefits of Post Training quantization for some selected CNN Networks as reported by [15]

The main advantages of this technique are the simple use, with very few code lines it is possible to quantize a Model, and the conversion time, the tool only needs to visit all the layers in the model one time and to apply the quantization. The main disadvantages are the limited quantization options, only Int8 resolution, and the effectiveness of the quantization, especially for smaller networks the post-training discretization can bring a huge accuracy loss.

Quantization-aware Training

The quantization-aware training technique is the best way to achieve the highest performance with minimal accuracy loss. The method cooperates during the Training with the Tensorflow API, in order to quantize the Model during this phase instead after the training has been concluded. Doing so, the method takes trace of how the weights change over the epoch training and will better find the best range values to quantize them in.

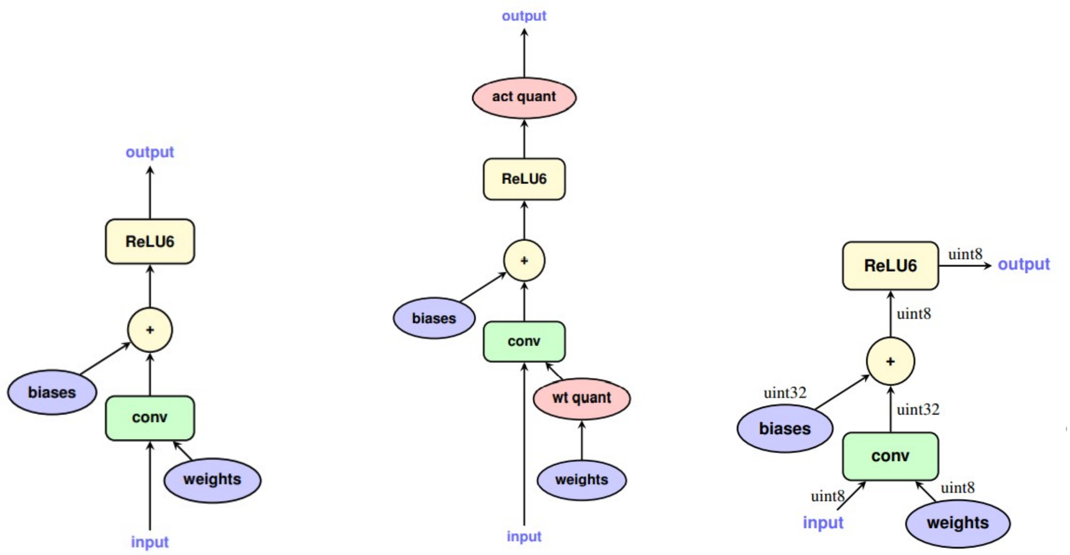
As shown in the Figures 2.5a and 2.5b, a common convolutional layer is quantized inserting some *fake quantization* Tensorflow nodes that perform the quantization of the weights before the *conv* layer and after the activation function action. The bias are maintained on 32bit. This graph structure is maintained during the training phase only, since during inference the graph is converted in the fully-integer shape as shown in Figure 2.5c.

Unlike the post-training technique, it is possible to choose on how many bits the weights have to be quantized, in the range $[2-8]$ bits. As reported by [16], the results coming from well-known models, listed in Table 2.5 are really promising with very minimal accuracy loss.

Model	**Top-1 Accuracy non-quantized**	**Top-1 Accuracy 8-bit quantized**
MobileNetV1 224	71.03%	71.06%
ResNetV1 50	76.3%	76.1%
MobileNetV2 224	70.77%	70.01%

Table 2.5: The benefits of quantization-aware training for some selected CNN networks as reported by [16]

The main advantages of this technique are the accurate model quantization, since it is performed during the training, and the performance achieved, with minimal loss in the accuracy especially for smaller networks. The disadvantages are the needed to perform a complete training session, even



(a) Common layer without quantization **(b)** Quantized common layer layout during training (QAT) **(c)** Quantized common layer layout during inference (QAT)

Figure 2.5: Quantization steps for a common Convolutional layer during *Quant-aware training* reported by [17]

for models already trained, and the substantial amount of code lines to add and to integrate in the training scripts.

2.2 Discretization techniques

With the promising results reported by [15] and [16] and reported in the Tables 2.4 and 2.5, the *post-training quantization* and the *quantization-aware training* techniques will be now analysed. In particular will be reported the basic concepts under these methods and how their hyper-parameters affect the performances.

2.2.1 Post-training quantization analysis

The main function of this method is to quantize all the variabes inside the model under-optimization to integer 8-bit values using the following formula:

$$real\_value = (int8\_value - zero\_point) * scale \tag{2.3}$$

where the $real\_value$ represents the value in floating point to be quantized, $int8\_value$ the value after the quantization, $zero\_point$ the position of the zero in the range $[-128, 127]$ and the $scale$ factor that represents the effective passage from real to integer and it depends on the input data. The effective values of the $scale$ and $zero\_point$ represent a freedom degree during the conversion of the model from the fully-float32 to the int8. Thanks to the *TFLiteConverter* module provided by Tensorflow Lite, during the conversion all the weights are forced to fall inside the int8 range, with the zero point strictly equal to 0. This symmetric structure ensures to reduce the computational cost as demonstrate below. Taken

A as a $m \times n$ matrix of Activations values,

B as a $n \times p$ matix of Weigths values,

during inference the matrix product is computed as follow:

$$\begin{gathered} a_j \cdot b_k = \textstyle\sum_{i=0}^{n} a_j^{(i)} b_k^{(i)} = \textstyle\sum_{i=0}^{n} (q_a^{(i)} - z_a)(q_b^{(i)} - z_b) = \\ \sum_{i=0}^{n} q_a^{(i)} q_b^{(i)} - \sum_{i=0}^{n} q_a^{(i)} z_b - \sum_{i=0}^{n} q_b^{(i)} z_a + \sum_{i=0}^{n} z_a z_b \end{gathered} \tag{2.4}$$

The q_a,z_a and q_b,z_b are the quantized values and the zero-points for A and B respectively.
The final four terms of the product evolution represent the computational cost of every neural networks but some optimizations can improve the total cost. In fact, the

$$\sum_{i=0}^{n} q_b^{(i)} z_a$$

and

$$\sum_{i=0}^{n} z_a z_b$$

represent only products between constants that are the same every time the model has to produce the prediction, and thus can be pre-calculated.

Instead, the

$$\sum_{i=0}^{n} q_a^{(i)} z_b$$

term can be avoided if the zero-point of the weights is equal to 0. Hence, thanks to the symmetry, it is possible to reduce the computational cost to only the un-avoidable term

$$\sum_{i=0}^{n} q_a^{(i)} q_b^{(i)}.$$

.

For the *scale* factor, the tool applies two different optimizations depending on the actual operation the tensor represents, the *per-axis* and *per-tensor* quantization. The first one uses the same *scale* factor and/or the *zero-point* per slice in the tensor, ensuring more granularity in the quantization, while the second one uses the same *scale* factor and/or the *zero-point* for all the tensor values.

In order to produce a quantized model, it is necessary to got the trained neural network and to convert it with the *TFLiteCoverter* module to a Tensorflow Lite Model. Then, some parameters have to be tuned depending on the results to achieve.
Firstly, the *tf.lite.Optimize* settings, that actually tells the module in what direction has to optimize the model. Even if there are three choices, in the last version they have been grouped all in one, the *tf.lite.Optimize.DEFAULT*, that directs the optimization strategy toward size and latency reduction.
The other parameter to set is the *tf.lite.OpsSet*, the Operations available to generate the quantized model. Actually, there are four different operations list:

- the *TFLITE_BUILTINS* default that allows to use only the Tensorflow Lite built-in operations that make the model quantized using Int8 ops when allowed and Float32 when not, without any error message;

- the *TFLITE_BUILTINS_INT8* that allows to use Integer-only operations on 8 bits throwing an error if some operations are not supported

- the experimental *EXPERIMENTAL_TFLITE_BUILTINS_ACTIVATIONS _INT16_WEIGHTS_INT8* that allows to use the built-in "Integer-only" operations but with the weights quantized to Int8, the activations to Int16 and the bias to Int64. This last particular optimization setting is designed to those models that have particularly noisy input or hard tasks, like image de-noising, HDR reconstruction and super-resolution, but it is only compatible with CPU execution;

- the *SELECT_TF_OPS* that uses the current operations defined in Tensorflow, so without any quantization.

The other settings concern the *inference_input_type* and the *inference_output_type* that informs the *TFLiteCoverter* module the types to use for the input and output Tensors between *tf.int8*, *tf.uint8* and default *tf.float32*. The first two are reserved for "Full-integer" quantization only, while the default setting is intended to maintain the interface between the float32 external environment and the quantized Model.
In Table 2.6 are summarized the performance achievement with these different settings.

2.2.2 Quant-aware training analysis

The main aim of this method is to make a Keras description model quantization aware and then to perform a training session during which the Model will be quantized effectively. Unlikey for the *post-training quantization*, this technique allows the developer to even choose the actual representational precision to quantize the model in.

Actually *Tensorflow Model Optimization* module offers three different quantizers. The mathematical base on which it relies every single quantizer is very similar as reported in (2.3), with the only difference that, except for the default settings, the integer value is not fixed to 8 bits, but can be in the range [2,8] bits and the *scale* factor is chosen with different strategies [18].

- The *LastValueQuantizer* takes the max and the min values for the input tensor during the training and uses this interval to quantize the output tensor.

- The *MovingAverageQuantizer* takes the max and min values for the input tensor and calculate the effective range using the average among the last values

- The *AllValuesQuantizer* takes the max and the min values during initilization and continues to use those for all the input tensors, all the time is called to quantize.

tf.lite.converter Attributes	tf.lite Settings	Performance
optimizations	*Optimize.DEFAULT*	Optimization strategy for memory and latency reduction
	Optimize.OPTIMIZE_FOR_SIZE	Deprecated, same ad *DEFAULT*
	Optimize.OPTIMIZE_FOR_LATENCY	Deprecated, same ad *DEFAULT*
target_spec	*OpsSet.SELECT_TF_OPS*	Allow the normal Tensorflow operations without apply quantization
	OpsSet.TFLITE_BUILTINS	Apply quantization to Int8 when supported, otherwise on Float32 without any message
	OpsSet.TFLITE_BUILTINS_INT8	Applies quantization with ONLY Int8 operations, throwing an error otherwise
	OpsSet.EXPERIMENTAL_TFLITE_BUILT INS_ACTIVATIONS_INT16_WEIGHTS_INT8	Applies quantization with Int8 weights, Int16 activations and Int64 bias
inference_input_type	*tf.float32*	Default setting, maintain the input dtype accepted to float32 even if the model is quantized differently
	tf.int8	Accept input only with Int8 dtype, ensuring compatibility with integer only devices (*preferred option for symmetric quant.*)
	tf.uint8	Accept input only with uInt8
inference_output_type	*tf.float32*	Default setting, produces the output dtype to float32 even if the model is quantized differently
	tf.int8	Output produced to Int8
	tf.uint8	Output produced to uInt8

Table 2.6: Main settings for post-training quantization using Tensorflow Lite module and their impact on the model

Actually, there is also a "blank" quantizer, the *Quantizer* that can be used to take control over the quantization strategy and to implement a custom one.

The other parameters that can be set are common for all the quantizers and they are summarized in Table 2.7 below.

Arguments	Usage
num_bits	Number of bits for quantization in the range [2,8]. Default *8bits*
per_axis	Boolean to make the quantization *per-axis* (*True*) or *per-tensor* (*False*) aware $^{(1)}$. Default *False*
symmetric	If *True* makes the quantization range symmetric between the upper and the lower limit, otherwise calculates the Max and the Min with the selected strategy. Default *False*
narrow_range	In case of 8bit resolution only, forces the range to be $[-127,127]$ to induce abetter symmetry with the 0 exact in the center $^{(2)}$. Default *False*

$^{(1)}$: as reported by [15] and the Section 2.2.1, the *per-axis* quantization can bring better results inducing more granularity since the range resolution is calculated for every Tensor slices in the last dimension instead of using only one range for all the Tensor slices.
$^{(2)}$: forcing the symmetric quantization, especially for the weights, can bring better performance in the computational cost as reported by [17] and explained in Section 2.2.1 equation 2.4

Table 2.7: Hyper-parameters for the *Tensorflow Quantizers* and their effects

Chapter 3

Model optimizations

In this chapter will be analysed the experimental results obtained from the models quantized with the two optimization techniques described in Chapter 2 Section 2.2. The models used in this Section come from the best performing models in [8], one for each different architecture:

- *Multilayer Perceptron*, 1 hidden layer, 32 and 41 neurons;
- *Convolutional Neural Network*, 2 hidden layers with 1D-CNN with kernel size 3 and 64 filters, 2 hidden layer MLP with 32 neurons;
- *Long Short-Term Memory*, 1 hidden layer with 2 LSTM cells

For each model will be applied both techniques, except for the *LSTM* becouse the last version of Tensorflow used during the development of this thesis (tf v2.2) does not yet offer support for this kind of recurrent neural network, both for quant-aware training and for post-training quantization. The results collected will be compared with the un-optimized reference models. Before the optimizations, it has been conducted a data analysis in order to find the best hyper-parameters settings for both the techniques, while all the other parameters have been left as reported by [8]. In this way, all the results collected can be compared with the ones obtained in the previous work.

A Design Space Exploration (DSE) with the MLP has been conducted at the start of this work, in order to be familiar with the Tensorflow API and its tools.

The data set used in all the experiments belongs to the data-acquisition campaign conducted during 2016 and reported in [9] and it is divided as follow:

- Training set: composed of 976 training examples used only during the DSE with MLP networks and QAT
- Validation set: composed of 325 validation samples, used to choose the best model during DSE and QAT
- Test set: composed of 325 test samples, used in the test phase to evaluate the performance of the model after the optimizations and for the DSE with MLP

The entire data set is composed by 4 values for the input and 2 values for the output. The input labels represent the values coming from the capacitive sensors, while the output labels represent the ground truth X and Y positions of the person inside the room coming from the ultrasound sensors.

The training, during DSE and QAT, is performed for 1000 epochs without any regularization, like "early-stopping", to bring the model to the full convergence, action suggested especially during QAT to better quantize the parameters and to leave the model the time to recover from the loss of representational precision. During the test phase some curves have been produced to see the performances trend of the *Mean Squared Error* parameter and the *Average Euclidean Distance Error*, the comparative terms between the un-optimized models and the quantized ones. These two metrics have been explained in (1.1) and (1.2).

3.1 Multi-layer perceptron neural network design space exploration

In this section are shown some results related to a design space exploration conducted with the MLP neural network architecture. In particular, it has been explored one hidden layer layout and tested with different number of neurons inside, in order to check if the "best-model" reported in [8] was thoroughly optimized. The experiments have been conducted in the following order:

1. Parameterization of the MLP in terms of neurons per layer

2. Autonomous training session with the number of neurons spanning into [22,52]

3. Results analysis

3.1.1 Neural network parameterization

The model layout with only 1 hidden layer is shown in Figure 3.1. In order to make easier the exploration, some parameterization on the neural network has been conducted, in particular the number of neurons inside the layer. Every model has 4 input nodes and 2 output nodes, with a *Keras Dense* layer in the middle. The kernel initializer chosen is the *keras.initializers.glorot_uniform()*. The other training parameters, like the number of epochs fixed to 1000 and the *Adamax* optimizer with learning rate $lr = 1e^{-4}$, are the same reported by [8] during the experiments.

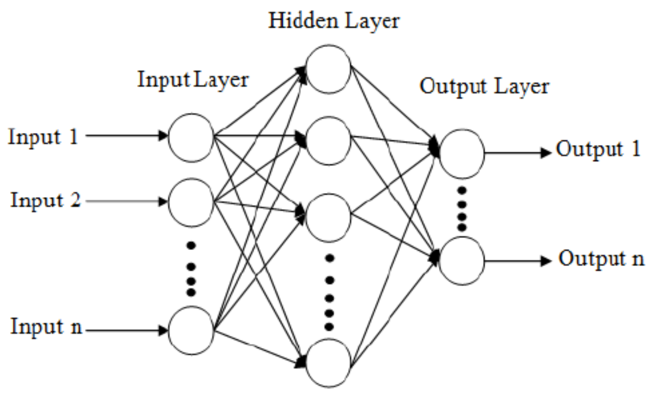


Figure 3.1: MLP layout with one hidden layer and multiple input/output nodes

3.1.2 Automated design space exploration

With the NN parameterized and the help of some *scripts*, it has been possible to describe and train different NNs at the same time. For every model layout with different number of neurons inside, the training has been conducted 10 times in order to remove sporadic results due to the random initialization of the variables. The results collected are the *loss* curves for the training and validation data set, to check the quality of the training, and the values of the *best MSE* (1.1) and *best ADE* (1.2) on the test dataset. Also the trajectories inferred have been printed out to better check the quality of the trained models.

3.1.3 Result analysis

The results collected have been summarized in Table 3.1 below. The interesting point from all those slightly different models performances is the relationship between the number of neurons and the improvements in the ADE, the main quality parameter. This relationship is underlined in Figure 3.2.

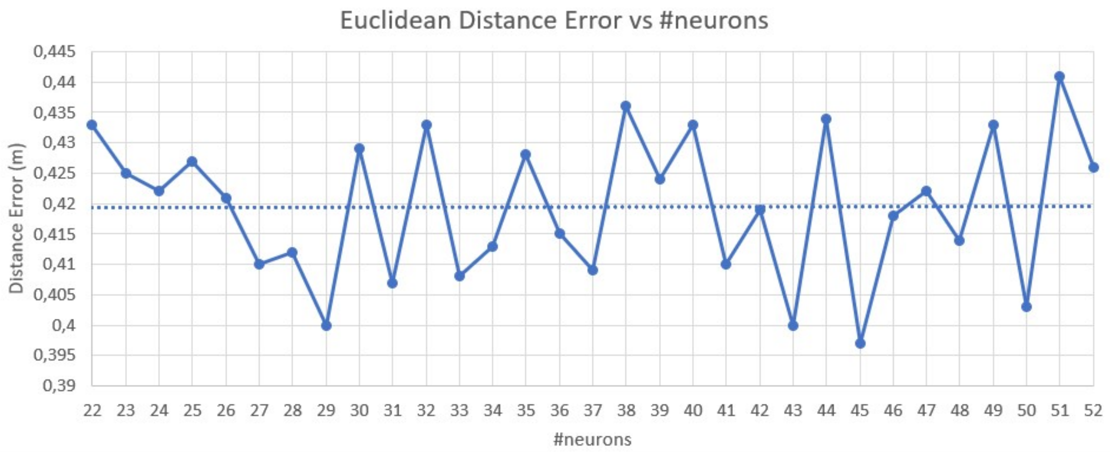


Figure 3.2: DSE results for every MLP models with different number of neurons in terms of Average Distance Error

As it can be seen, there is actually no correlation between the number of neurons and the ADE achieved. The *best model* is the one with 45 neurons. Except for the first iterations, from 22 to 29 neurons in which it is possible to recognise a more regular path, increasing the number of neurons does not decrease the ADE, as could be expected. In fact, even if the best model is with 45 neurons, this is only a local optimum with no evidence that, if trained for more time, also other points could achieve lower ADE values.

Another important evidence of lack of correlation is the *linear regression line*, represented in Figure 3.2 by the dotted line. This line has a slope so near to 0 that actually represents the *mean value*, since it is almost flat.

The loss function, reported in Figure 3.3a, shows a good convergence of

Neurons	Best epoch	Test MSE (m^2)	Test ADE (m)	N. of parameters
22	230	0,123	0,433	156
23	462	0.117	0,425	163
24	241	0,117	0,422	170
25	415	0.124	0.427	177
26	203	0,117	0,421	184
27	253	0,115	0,410	191
28	307	0,113	0,412	198
29	332	0,107	0,400	205
30	395	0,123	0,429	212
31	268	0,111	0,407	219
32	193	0,125	0,433	226
33	278	0,108	0,408	233
34	352	0.114	0.413	240
35	198	0.120	0.428	247
36	265	0.114	0.415	254
37	240	0.110	0.409	261
38	225	0,128	0,436	268
39	343	0,118	0,424	275
40	243	0,124	0,433	282
41	317	0,110	0,410	289
42	216	0,115	0,419	296
43	252	0,107	0,400	303
44	307	0.124	0.434	310
45	**229**	**0.104**	**0.397**	**317**
46	282	0,118	0,418	324
47	340	0,119	0,422	331
48	239	0,113	0,414	338
49	168	0,124	0,433	345
50	165	0,108	0,403	352
51	169	0,130	0,441	359
52	177	0,119	0,426	366

Table 3.1: Design space exploration results for multilayer perceptron network with 1 hidden layer and neuron numbers from 22 to 52

the training curve, e.g., the error the optimizer minimizes among the epochs. The validation curve, drawn looking at the error distance among the epochs for the validation data set, follows the training curve only for the first 100 epochs and then starts to diverge form the training curve. This divergence between validation and training curves shows the presence of the *overfitting problem*, i.e. when the model starts to not generalize enough the task and to stop learning from the training example. Even if the training is performed for 1000 epochs, at he end of each epoch only if the model has a lower validation loss value is saved and evaluated, otherwise it is discarded. This method allows to effectively produce the best model with the lowest validation loss, and so with the minimum overfitting problem. The model with 45 neurons learns better the task compared with the 32-neurons model, but anyway it does not represent the best model to use in order to generalize the task, since it stops quite soon to learn from the training examples.

The aim of these experiments was only to explore the space around 32 neurons optimum reported by [8] and collected in Table 3.2. In fact, the reported model was not the best model layout and, even if with a not standard number of neurons (45), the found model can infer more precisely the indoor human location.

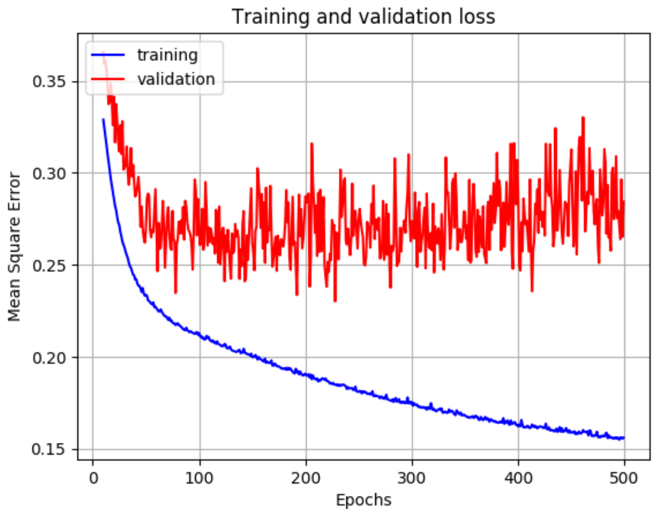
Model	Neurons	Best Epoch	Test MSE (m^2)	Test ADE (m)	N. of parameters
MLP from [8]	32	193	0.125	0.433	226
MLP from DSE	45	229	0.104	0.397	317

Table 3.2: Comparison between the performance from [8] and the best model found during design space exploration

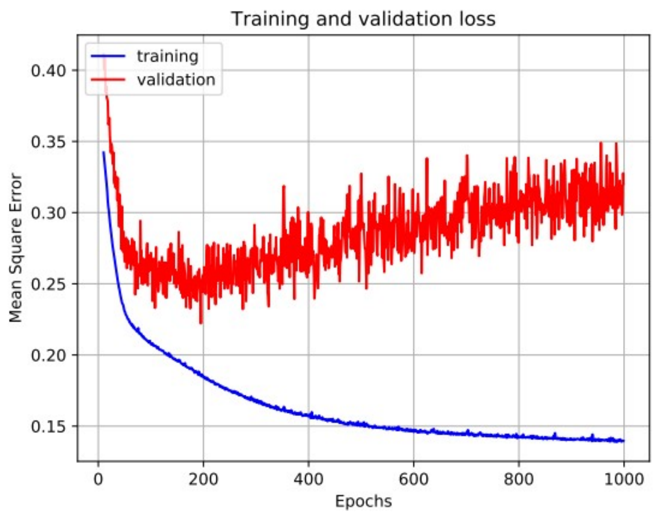
3.2 Quantizer and setting parameter optimization for the quant-aware training

In order to understand the best usage of the quantizers offered by Tensorflow model optimization module and how their hyper-parameters affects the results, it has been conducted a design space exploration among all the different settings. The model used to perform this exploration is the convolutional neural network reported by [8] with these characteristics:

- 2-layer 1D-Convolutional,



(a) Training and validation loss functions for 45 neurons network



(b) Loss and validation function for 32 neurons network

Figure 3.3: Comparison between the training and validation loss function for the multilayer perceptron network with 45 and 32 neurons

- kernel size 3×1,
- 64 filters,
- window size of 10 s (30 input data at a time processed)

- 2-layers MLP with 32 neurons,
- L2 regularization

The quantizers analysed are reported in Chapter 2 Section 2.2.2. In order to optimize the exploration and not to waste resources, only some number of bits have been chosen as a reference. In particular, weights and activations resolutions that are the more promising for hardware implementation, e.g. 4 bits in the weight and 4, 8, 16 bits in the activation. With these quantization schemes it is possible to exploit better the hardware resources, allowing parallel execution or reducing the bit parallelism in the arithmetic unit.

The parameters explored are:

- *symmetric*, to force the quantization range to be symmetrically distributed around the 0 position or asymetrically if not;
- *narrow_range*, to force the quantization range with 8 bits only to span uniformly between [-127,127] with the 0 positioned exactly in the center;
- *quantizer*, actually the *LastValue quantizer* that calculates the scale factor using the last values range or the *MovingAverage quantizer* that calculates the scale factor based on the average among the previous quantized tensors.

The quant-aware training has been performed 10 times for each parameters combinations to search an optimal training. The results are collected in Table 3.3 below.

From these results, the best choice is to use the *MovingAverage* quantizer for the weight and the *LastValue* quantizer for the activation. The quantizers actually affect more the result than the other parameters. In the case of the weight quantization, the quantizer selected is more suitable to track the variation among the epochs in a slighter way since it uses an average value. For the activation, instead, the best choice is to track only the last value from the previous epoch.

The other parameters concern the internal shape of the quantization range. Looking at the results, it seems that the *symmetric* quantization on the weigth brings better results, additionally to the *asymmetric* quantization on the activation. This difference is partially discussed in [17], where the symmetric quantized weights together with the asymmetric qauntized activations have already brought better results. The last parameter refers to

Weight quantizer		Activation quantizer		weigth-activation (#bits)	Test ADE (*m*)
symmetric	***narrow_range***	***symmetric***	***narrow_range***		
MovingAverage		*MovingAverage*		4-4	0.360
				4-8	0.338
True	*True*	*False*	*False*	4-16	0.315
MovingAverage		*MovingAverage*		4-4	0.344
				4-8	0.353
False	*True*	*False*	*True*	4-16	0.330
MovingAverage		*MovingAverage*		4-4	0.329
				4-8	0.353
False	*False*	*False*	*False*	**4-16**	**0.285**
LastValue		*LastValue*		4-4	0.354
				4-8	0.344
False	*True*	*False*	*True*	4-16	0.340
MovingAverage		*LastValue*		**4-4**	**0.321**
				4-8	**0.291**
True	*True*	*False*	*False*	4-16	0.304
MovingAverage		*LastValue*		4-4	0.323
				4-8	0.307
False	*False*	*False*	*False*	4-16	0.302
MovingAverage		*LastValue*		4-4	0.350
				4-8	0.324
True	*True*	*True*	*True*	4-16	0.329

Table 3.3: Different quantization settings results for convolutional neural network with 4-4, 4-8, 4-16 bits in weight-activation

the 8bit quantization only, and in fact performs better when enabled in the int4-int8 weight-activation resolution model.

The best way to perform the quantization using the quant-aware training approach is summarized in the list below, and used in the rest of experiments:

- *MovinAverage* quantizer for weight, with *symmetric* and *narrow_range* parameters enabled

- *LastValue* quantizer for activation, with *symmetric* parameter disabled and *narrow_range* parameter enabled

3.3 Multilayer perceptron quantization optimization

In this section the optimization techniques described in Section 3.2 will be applied to the *Multilayer Perceptron* neural network with 1 hidden layer and 32 neurons.

3.3.1 Post-training quantization

The results collected from the post-training quantization are summarized in Table 3.4. As expected, the average distance error increases due to the reduced representational precision, but around 1% worse than the float32 model.

ADE Error Ground-truth vs float32 (m)	**ADE Error** Ground-truth vs int8 (m)	**ADE Error** Int8 vs float32 (m)	**.h5 Model** (KB)	**.tflite Model** (KB)
0.433	0.438	0.0254	23.688	2.096

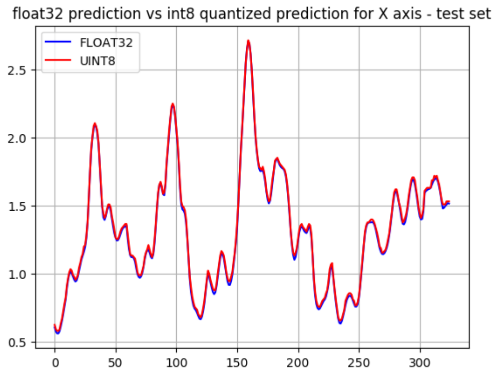
Table 3.4: Post-training quantization results with int8 resolution for multilayer perceptron model

The quantization has been conducted with a *representative data set* spanning into the complete training data set and *tf.lite.SetOps.TFLITE_BUILTINS _INT8* selected. The good performance achieved by the quantized model can be easily seen in Figures 3.4a and 3.4b, where the float32 model and the int8 model predictions are compared side by side and they appear almost identical. Since the model is used in a floating point environment, the model has an input quantizer that converts to int8 the incoming float32 data and an output de-quantizer to float32 to produce floating-type data.

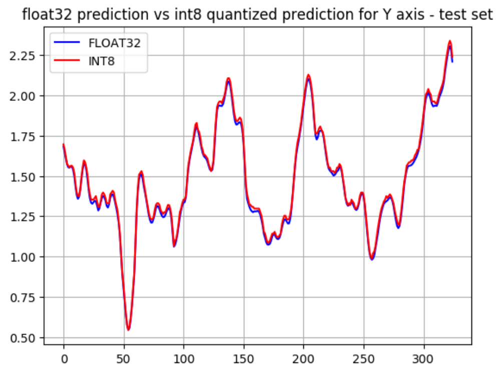
3.3.2 Quant-aware training

In order to better explore the entire design space, an automatic *design space exploration* has been developed with these parmeters:

- *Moving average quantizer* for the weight, with *symmetric quantization* and *narrow range* enabled.



(a) X trajectory inference comparison int8 vs float32



(b) Y trajectory inference comparison int8 vs float32

Figure 3.4: Trajectories comparison between the fully-integer 8 bits quantized model and the float32 model

- *Last value quantizer* for the activation, with *asymmetric quantization* and *narrow range* disabled
- Number of bits spanning from to 2 to 16 bits for weights and activations for a total of 225 different trained models

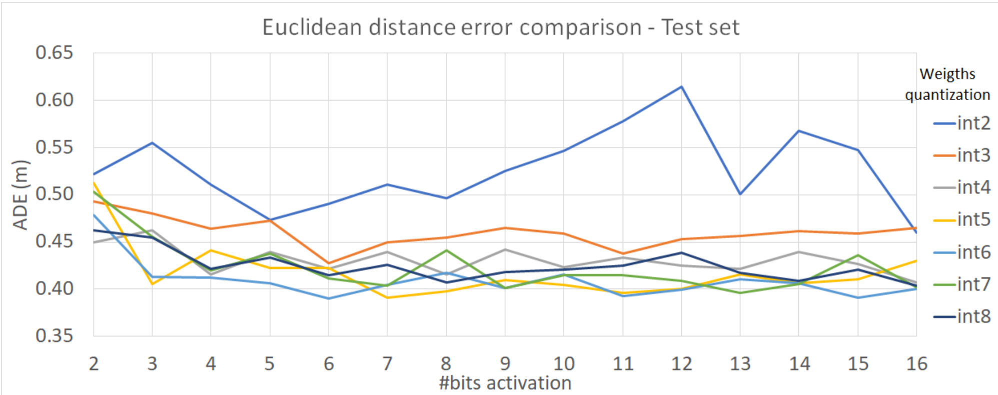
The training settings are the following:

- Adamax optimzer with learning rate $lr = 0.0001$
- 1000 epochs per training session, repeated 10 times per model

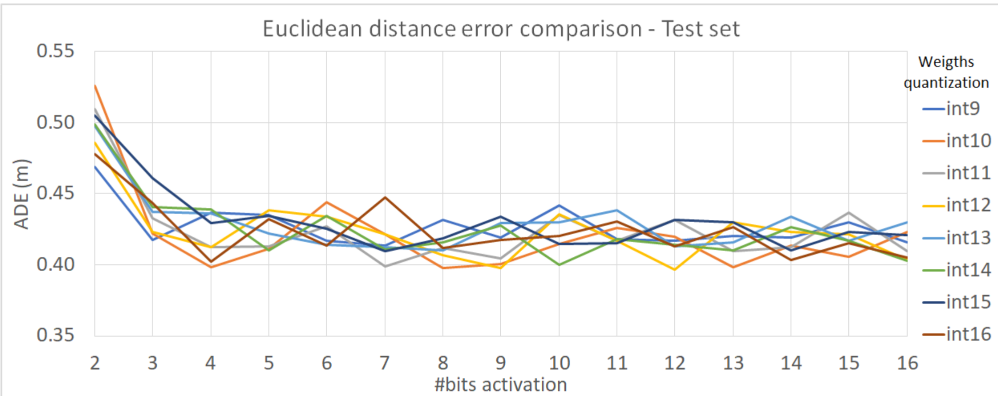
Since this kind of network is not so promising, in order to save processing time, the training is performed only 10 times to search the optimum training, and 30 times for the most hardware-interesting couple of weight-activation bits resolution like 4-4, 4-8 and 4-16.

The Figure 3.5a shows the performance trend for the quantized models from 2 to 8 bits resolution in weights. The general behaviour is a quite flat repsonse for all the models, within the [0.39,0.45] range for the ADE. Only for the 2 and 3 bit weight quantization the trend is always over this range, actually reaching unacceptable accuracy loss values. The flatness of this curve also suggests that with this network and those quantization parameters, there is no substantial difference in term of ADE value between the analysed models, allowing to select networks with very low resolution with similar accuracy achievement.

The Figure 3.5b, on the other hand, shows the performance trend of quantized models within the range [9,16] bits weight resolution. Here, the flatness of the curves is granted from about 4 bits in the activation resolution. After this limit, the ADE values for all the models is within the [0.40,0.45] threshold, as for the previous models. The shapes appears more regular with those models, thanks to a better representational precision in the weights.



(a) Performance comparison for weight in the range [2,8] bits



(b) Performance comparison for weight in the range [9,16] bits

Figure 3.5: Performance comparison for all the multilayer perceptron models quantized during the design space exploration

In Figure 3.6, the performance surface for all the models is presented. Since the compactness of the network (only one hidden layer with 32 neurons), could be convenient to look also to the models with higher resolutions, since the bits saving is still good but with more accuracy during inference. It can be seen that the blue region, the one with the best ADE values, is more concentrated in the range [5,7] bits in the weight, with some local optimum also with higher bits number, but more isolated and less interesting since the ADE achievement in those points is comparable with the previous region with less resolution.

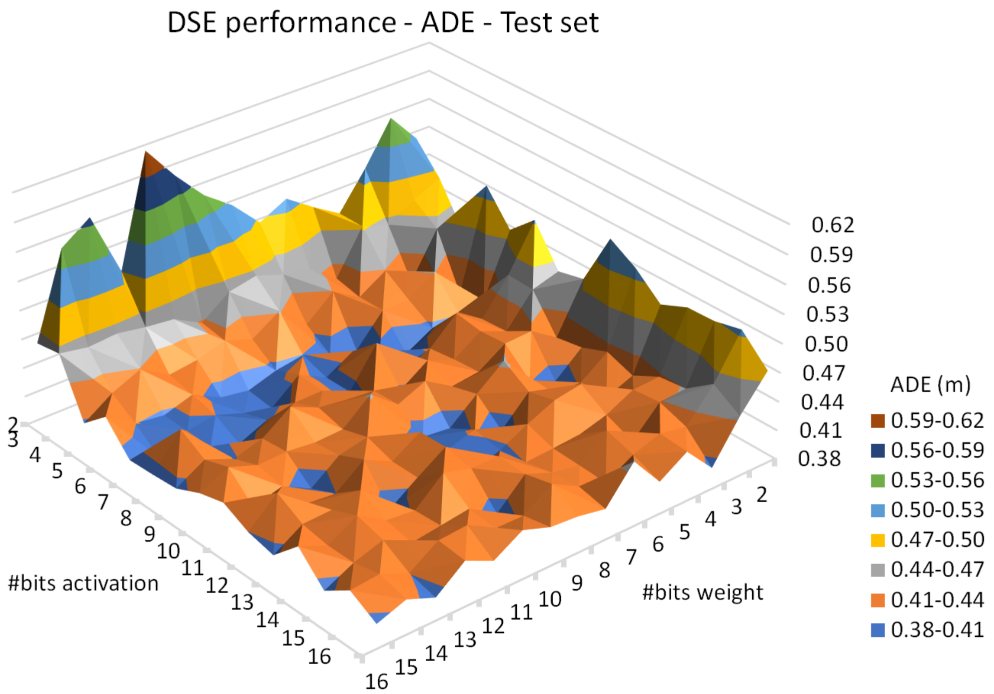


Figure 3.6: Performance surface for all the quantized models

Thanks to Figure 3.7 it is possible to explore the best performance achieved for every bits number in the activation. The labels on the line represent the number of bits in the weight (upper value) and the ADE value achieved (lower value). From these data we can deduce that:

- the weight resolution is more important than the activation resolution, since it significantly affects a lot of models;
- too low resolution in the activation brings always bad results, but in the $[4-16]$ bits range the performance is stable.

In Table 3.5 are summarized the results obtained with this network architecture using the quant-aware training technique, compared to the

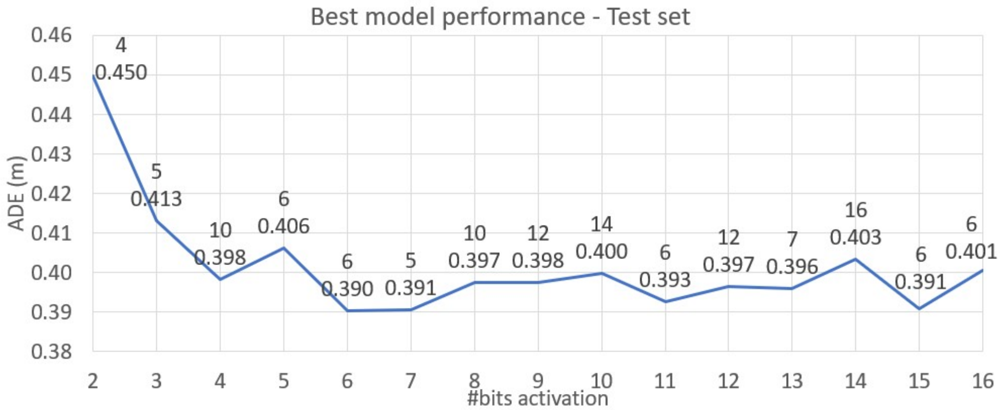


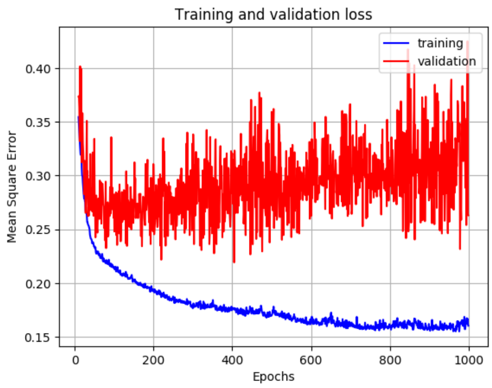
Figure 3.7: Best performance comparison between all quantized multilayer perceptron models

performance obtained by the float32 model from [8]. The ADE values produced by this technique are really promising since it achieves better results than the float32 model using less bits. The reason is in the lower resolution itself, that makes the inferred trajectories sharper and more oscillating and so nearer in average to the ground truth traces (see Chapter 4 for details).

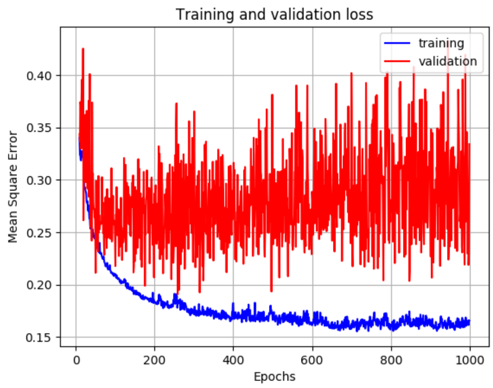
	Test MSE (m^2)	Test ADE (m)	
weight: int6; **activation**: int11	0.098	0.393	3^{rd}
weight: int5; **activation**: int7	0.096	0.391	2^{nd}
weight: int6; **activation**: int6	0.097	0.390	1^{st}
		$\Uparrow -9.9\%$	
float32	0.125	0.433	

Table 3.5: Performance comparison between the best float32 model and the best three multilayer perceptron quantized models

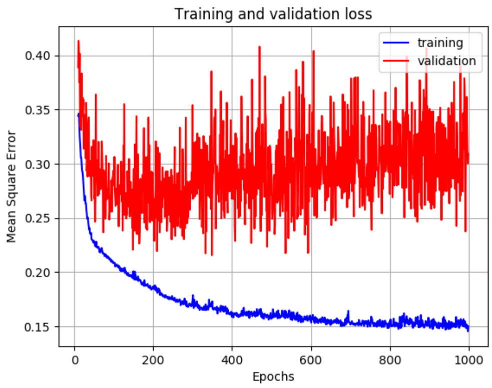
The training and validation curves from this three models are reported in Figure 3.8 below. The validation loss curve is extremely noisy for all the trained models and shows a good learning in the first epochs and then a quick stabilization and even a slight divergence from the optimal value for Fig. 3.8a. If compared with Fig. 3.3b, the float32 model already trained, the curves reach better values with a more stable trend but with a higher variance.



(a) Loss curves for int6 weight and int11 activation model



(b) Loss curves for int5 weight and int7 activation model



(c) Loss curves for int6 weight and int6 activation

Figure 3.8: Training and validation loss curves for the best three multilayer perceptron models

3.3.3 Multilayer perceptron optimization results summary

The optimization techniques used bring very interesting results, since the accuracy loss with the quantized model not only is negligible using the post-training method (around 1% less precision), but also improves significantly using the quant-aware training method. In Table 3.6, the comparison between the float32 model and the quantized models with the two techniques is shown.

Since all the training parameters, like the optimizer, the learning rate and even the number of training repetition, have been kept equal to [8], this

	Test MSE (m^2)	Test ADE (m)
float32	0.125	0.433
post-trained model: int8	0.127	0.438 (+1.15%)
quant-aware trained model: int6	0.097	0.390 (-9.93%)

Table 3.6: Multilayer perceptron network optimization results summary

method is really effective not only to reduce the resolution of the variables, but also to improve the performances of the network. To better understand how these promising results change with quantization, Figure 3.9 shows the distribution of the models quantized with this method during the DSE. 67% of all the trained models actually have improved their accuracy or at least they achieve the same ADE value w.r.t. the float32 model. This shows of the effectiveness of this method that, with this kind of network, behaves really good.

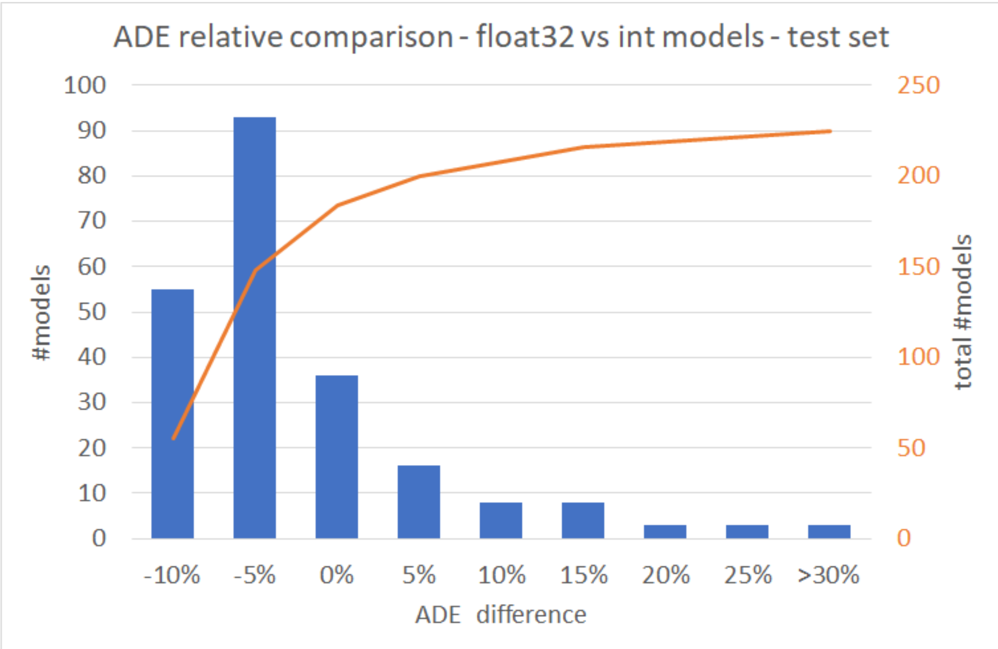


Figure 3.9: Models distribution with percentage performance comparison between all the multilayer perceptron quantized models and the float32 model

ADE Error Ground-truth vs float32 (m)	**ADE Error** Ground-truth vs int8 (m)	**ADE Error** Int8 vs float32 (m)	**.h5 Model** (KB)	**.tflite Model** (KB)
0.273	0.284	0.057	454.160	44.752

Table 3.7: Post-training quantization results for convolutional neural network model

3.4 Convolutional neural network quantization optimization

In this section will be applied the same optimization techniques to a convolutional neural network with this specifications:

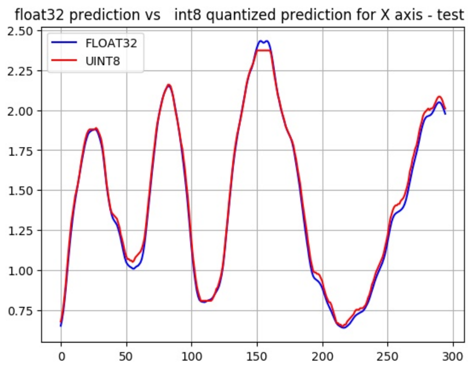
- 2-layer 1D-Convolutional,
- kernel size 3 × 1,
- 64 filters,
- window time of 10 s (30 input data at a time processed)
- 2-layers MLP with 32 neurons,
- L2 regularization

3.4.1 Post-training quantization

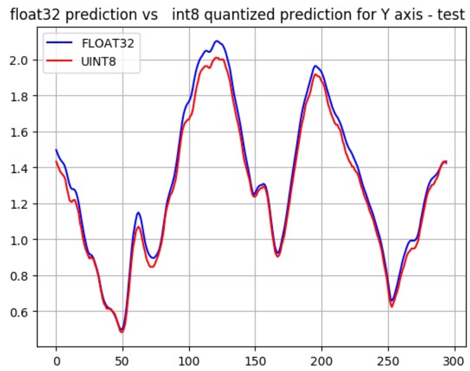
The results collected after the post-training quantization are summarized in Table 3.7. Also with this neural network, the post-training technique brings excellent results in terms of ADE error, within 4% of the float32 reference. The compression rate instead reaches 90% less of memory occupied.

In this case, the inferred trajectories by the quantized model are similar to the float32, as we can see in Figures 3.10a and 3.10b.

Even if quantized, the model is still producing and accepting floating point data type.



(a) X trajectory inference comparison Int8 vs Float32



(b) Y trajectory inference comparison Int8 vs Float32

Figure 3.10: Trajectories comparison between the fully-integer 8 bits quantized CNN model and the float32 CNN model

3.4.2 Quant-aware training

Due to the amount of possible number of bits to quantize in the model during the quant-aware training, a design space exploration has been conducted in order to find the best quantization implementation.

The DSE has been conducted with these specifications:

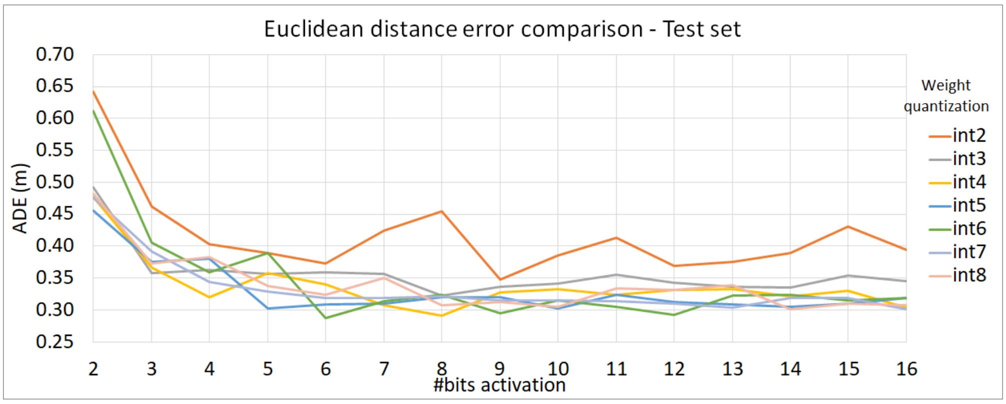
- *Moving average quantizer* for the weigths, with *symmetric* and *narrow_range* parameters enabled.
- *Last value quantizer* for the activations, with *symmetric* parameter disabled and *narrow_range* enabled.
- Number of bits spanning from 2 to 16 bits for both weight and activation, with a total of 225 different quantized models.

The training has been performed with:

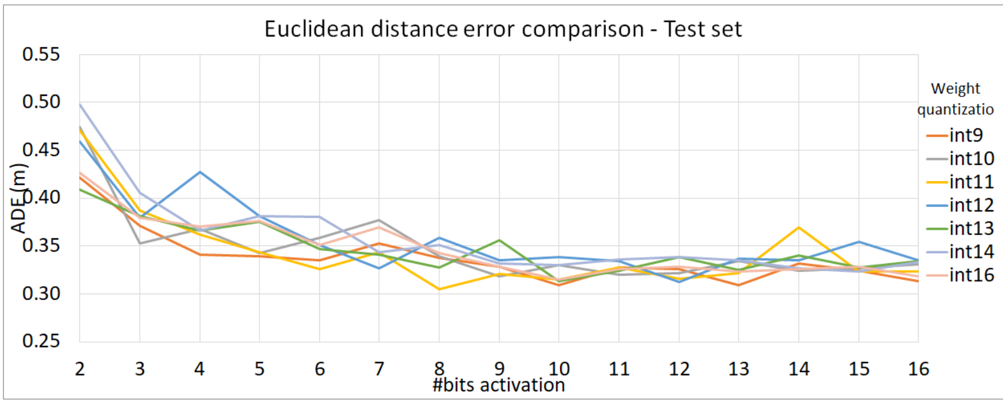
- Adamax optimizer with learning rate $lr = 0.0001$
- Training session during 1000 epochs, repeated 30 times for each model

During the exploration with the quantizers and their settings (Section 2.2.2), several training sessions have been conducyed with the CNN model. Thanks to this addditional trainigs, the models with 4 bits in weights and multiples of 4 in activations, have been quant-aware trained for a total of 90 times.

For graphical reasons, all the models results have been split in half, the first ones with the activation resolution within [2,8] range and the second ones within [9,16] range. In Figures 3.11a and 3.11b the direct comparison between the models results is proposed. The ADE error trend, as expected, assumes a quite regular dropping while the activation resolution starts to grow for every different training. Moreover, the most consistent accuracy loss is quite always constrained in the range [2,4] bits resolution for the activation, while after this limit all the models reach a more stable performance improvement.



(a) Models performance comparison for weight in the range [2,8] bits



(b) Models performance comparison for weight in the range [9,16] bits

Figure 3.11: Performance comparison for all the convolutional neural network models quantized during the design space exploration

The most promising models for hardware implementation, thanks to their lower representational precision, are the ones in Figure 3.11a. The worst model, as expected, is the one with only 2 bits in the weight representation: the lack of resolution is the reason why the ADE curve for this model is always higher than the others. All the other models have similar performance in the range [8,16] bits for activation, within the range [0.30,0.35]m for the ADE. This means that, with almost the same performance, it is possible to choose a model with lower representational precision, saving hardware resources.

Even more clearly, in Figure 3.12 it is presented the performance surface for the previous models. This 3D graph underlines the presence of a "pool" of local minimum points (orange region) where it is more convenient to quantize the CNN model.

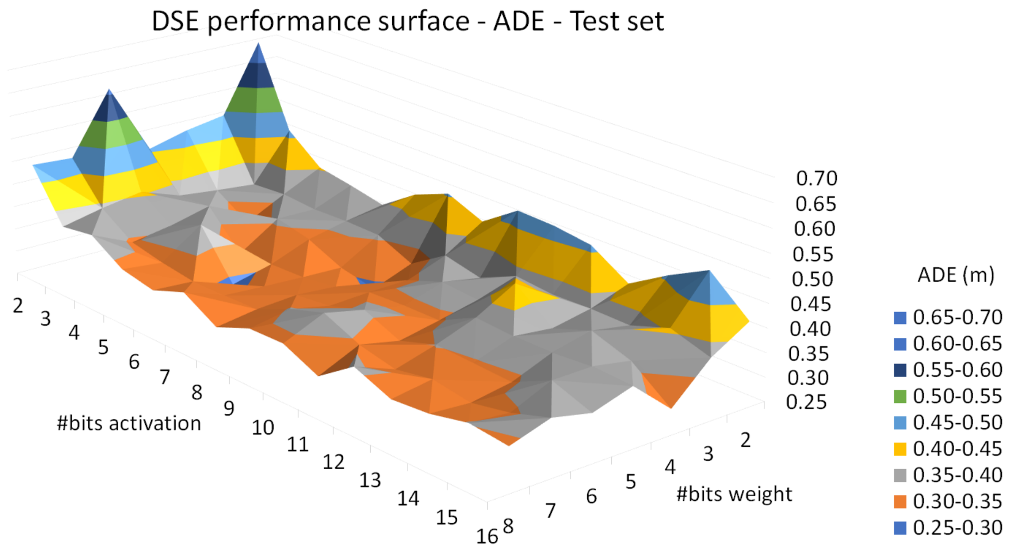


Figure 3.12: Performance surface for quantized models with [2,8] bits in the weight

In Figure 3.13 are summarized the best performances achieved by all the quantized models. In order to build this graph it has been taken the minimum ADE value for each model with progressively high number of bits in the activation. Actually, the points indicated are all local optimum for the selected couple weight-activation resolution. In each label on the line have been inserted the number of bits used in the weight (upper value) and the ADE achieved (lower value). It is interesting to notice that the best ADE values have been achieved by lower representetional precision, demonstrating that higher quantization resolution it is not always a good solution. In fact, the general trend among all the models is to have better results if quantized on fewer bits, actually using less than 10 bits in the activation and 8 bits in the weight. From this graph it is also possible to extrapolate some general rules while choosing the number of bits to quantize the model variables in:

- too low resolution in the activation brings too much accuracy accuracy loss, while it is acceptable in the range $[5 - 16]$;
- the activation resolution should be at least equal to the weight resolution, with better results when it is about double;

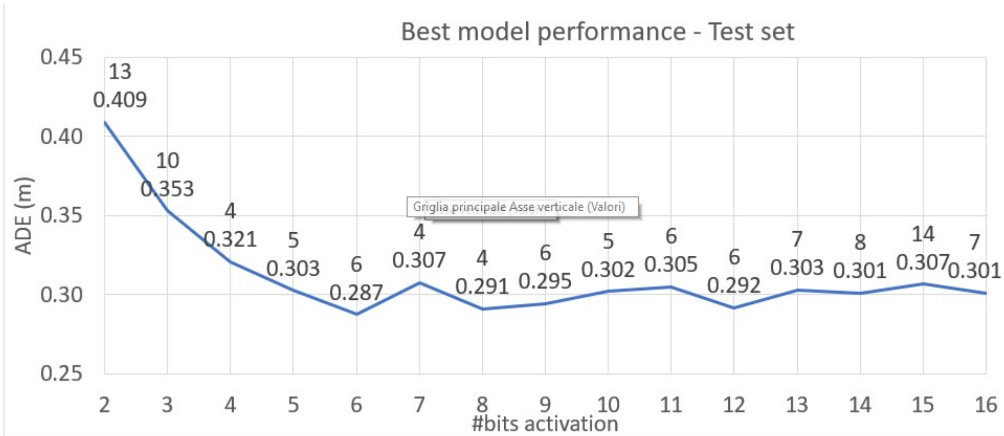


Figure 3.13: Best performance achieved by every quant-aware trained convolutional neural networks

In order to compare the obtained results with the float32 model, the Table 3.8 summarizes the best three models performance discussed in this section and the best float32 CNN model from [8].

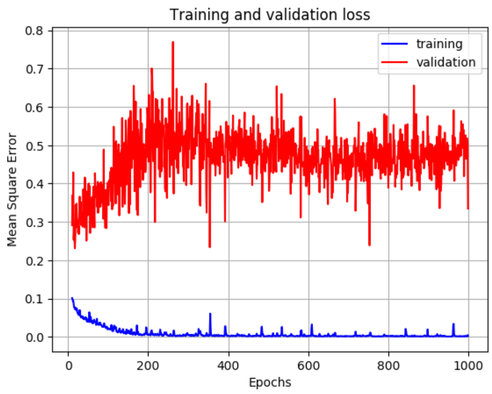
The interesting aspect from this result is the good performance achievement with only 6 bits in the weight and activation, actually using about 5× less memory and make feasible the usage of this kind of network inside an embedded system.

The training quality comparison between the float32 model and the quant-aware trained models is shown in Figures 3.14 and 3.15. The quantized models show higher variance in the validation curve, but with the same overfitting problems affecting also the float32 model. The validation curve

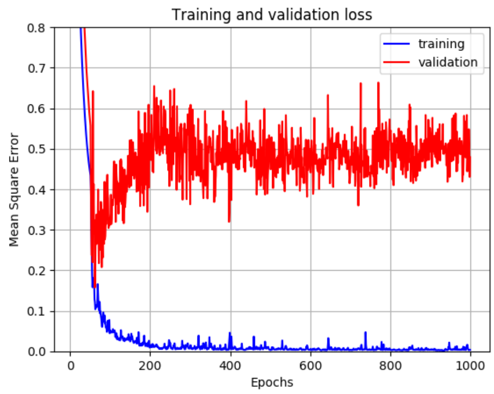
	Test MSE (m^2)	**Test ADE** (m)	
weight: int6; **activation**: int12	0.061	0.292	3^{rd}
weight: int4; **activation**: int8	0.058	0.291	2^{nd}
weight: int6; **activation**: int6	0.057	0.287 $\Uparrow$ +5%	1^{st}
float32	0.047	0.273	

Table 3.8: Performance comparison between the best float32 CNN model and the best three quantized CNN models

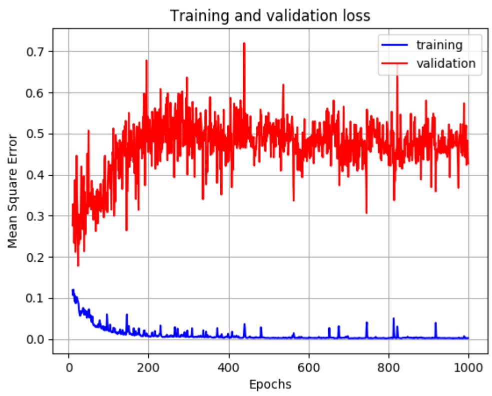
does not follow the decreasing training loss and, after an initial increase, it stabilizes around a certain value after a few hundreds epochs. In fact, all these quant-aware trained models stop learning in the very first epochs, as the float32 model.



(a) Loss curves for int6 weight and int12 activation model



(b) Loss curves for int4 weight and int8 activation model



(c) Loss curves for int6 weight and int6 activation

Figure 3.14: Training and validation loss curves for the best three convolutional neural networks

3.4.3 CNN optimization results summary

The results coming from both the techniques are promising since they show an accuracy loss from the fully-float model within 5%, allowing to save memory and resources without damaging too much the inference precision. In Table 3.9 are summarized the best results obtained.

The post-training technique is able to maintain, even with this large network, the accuracy loss within 4%, but it relies to an already trained model and so to the quality of the previous training. The quant-aware

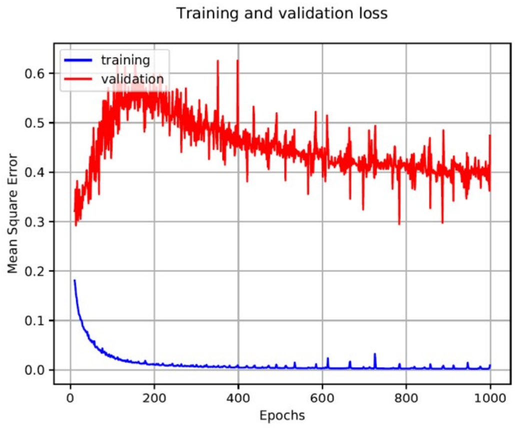


Figure 3.15: Training and validation loss curves for the convolutional neural network float32 model from [8]

training, on the other hand, it is able to choose how to quantize this large model and, with a custom training session, achieve the best accuracy loss around 5% w.r.t. the float32 model.

	Test MSE (m^2)	**Test ADE** (m)
float32	0.047	0.273
post-trained model: int8	0.061	0.284 (+4.03%)
quant-aware trained model: int6	0.057	0.287 (+5.13%)

Table 3.9: Convolutional neural network optimization reasults summary

In Figure 3.16 it is show the distribution of all the quantized models with the QAT technique in terms of percentage distance from the best float32 model ADE value. The majority of the quantized models achieve an accuracy within 30% from the best float32 model, demonstrating that, if the loss is acceptable, this convolutional neural network can be quantized on less bits. The graph groups together the quantized models that achieve an ADE value bigger than 70% of the float32 model, since the accuracy loss in that region is unacceptable.

Even if in the theory discussion the quant-aware training technique was presented as the most accurate method to actually quantize a neural network, here the results show that the post-training technique achieves the best results, considering also the processing time needed to convert the model into the quantized version. The reason could rely on the network depth, that is not so high to actually justify the fine-tuning with the quant-aware training method since the post-training is able to manage well this kind of network.

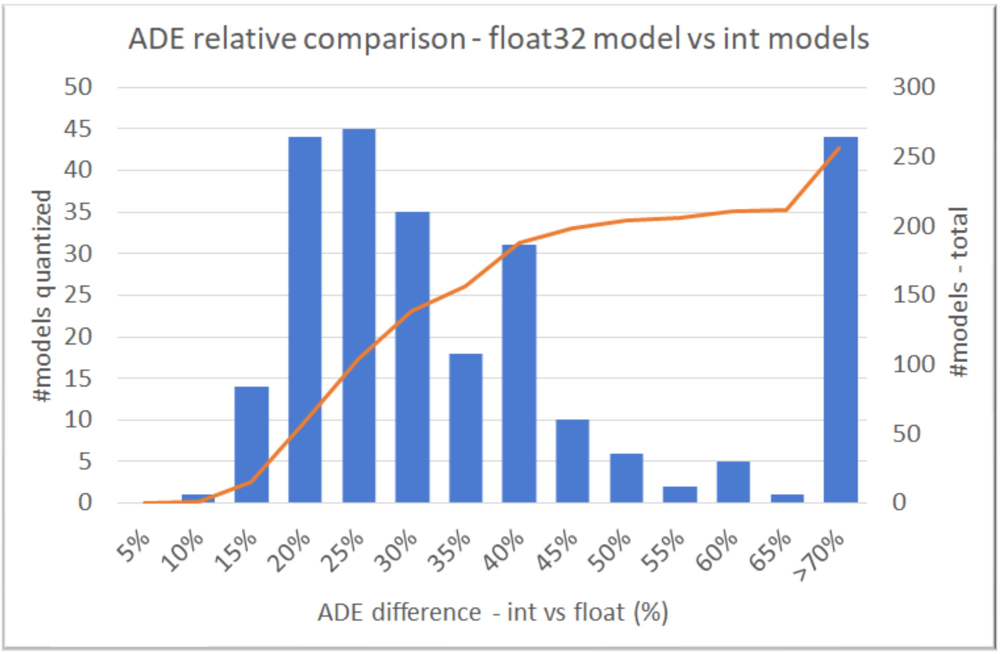


Figure 3.16: Models distribution with percentage performance comparison between all the quantized models and the float32 model

From the promising results reported above, it is also interesting to summarize the good results obtained from weigth-activation resolution couples that are more suitable to the final hardware implementation: the 4-4, 4-8 and 4-16 quantized models. In Table 3.10, the training results have been summarized and compared with the float32 model, in terms of percentage difference between the ADE value from the quantized model and the float32 model. Actually, the model that performs better is the one with 4 and 8 bits in the weight and activation respectively. It is worse than the float32 for less than 7%, but looking at how many resources can be saved with this quantization scheme, this is a remarkable result. Moreover, this model is the second best model among all the others, and not so far from the post-training quantized model, that actually uses int8 quantization for both weight and activation.

		4 bits weight		
		Test MSE (m^2)	Test ADE (m)	Deterioration (%)
activation	4 bits	0.068	0.321	17.58%
	8 bits	0.058	0.291	6.59%
	16 bits	0.068	0.304	11.36%

Table 3.10: Models results using best quantization scheme for hardware deployment

Chapter 4

Result summary

In this chapter all the results coming from the optimized models have been summarized and commented. In Table 4.1 the best results from both the techniques are shown. The best quantization scheme for both the quant-aware trained models is with 6 bits in the weight and in the activation and, in the case of the CNN model, performs as the post-training quantized model, or even better than PTQ method with the MLP model. The main goal of these techniques was to reduce the resolution of the varaibles, keeping the accuracy losses at minimum. Looking at the results, this goal has been achieved.

Neural network	**Optimization technique**	**Test MSE** (m^2)	**Test ADE** (m)	**weight-activation** (bits)
	PTQ	0.127	0.438	8-8
MLP	QAT	0.097	0.390	6-6
foat32 reference MLP model		0.125	0.433	
	PTQ	0.061	0.284	8-8
CNN	QAT	0.057	0.287	6-6
foat32 reference CNN model		0.047	0.273	

Table 4.1: Results summary with all the optimized models and the reference models

The results already presented are the best quantization points produced by the optimization techniques, but they do not represent the entire explored space. In order to show the relationship between the number of bits used during quantization and the performance achieved, the quantized models from both neural network architectures have been filtered using these criteria:

- the minimum ADE value on the test data set achieved
- the minimum number of bits used, summing up the resolution in both weight and activation

The resulting models have been collected in the graphs below. Figure 4.1 contains the pareto-points of the explored space together with the other local optimum points (grey crosses). The orange squares represents the hardware optimum points, e.g. the 4-4, 4-8, 4-16 quantization schemes that are more suitable for hardware deployment, while the green dot represent the result coming from the post-training quantization method. All the points under the light-orange line are performing better than the float32 model, the reference neural network. Looking at the graph, all the three hardware optimum points performs better using less bits than the float32 and, for this reason, they can be considered as the best choices.

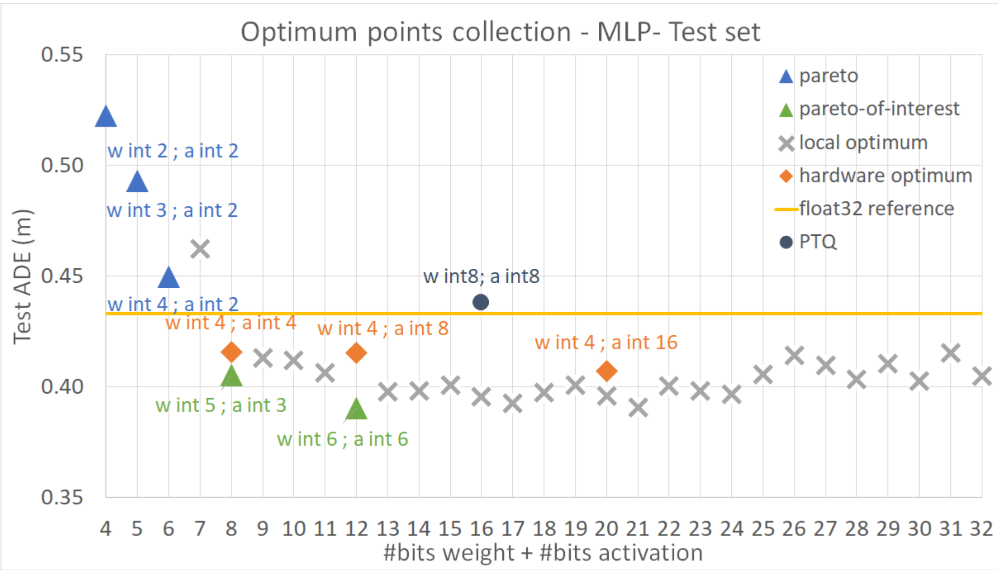
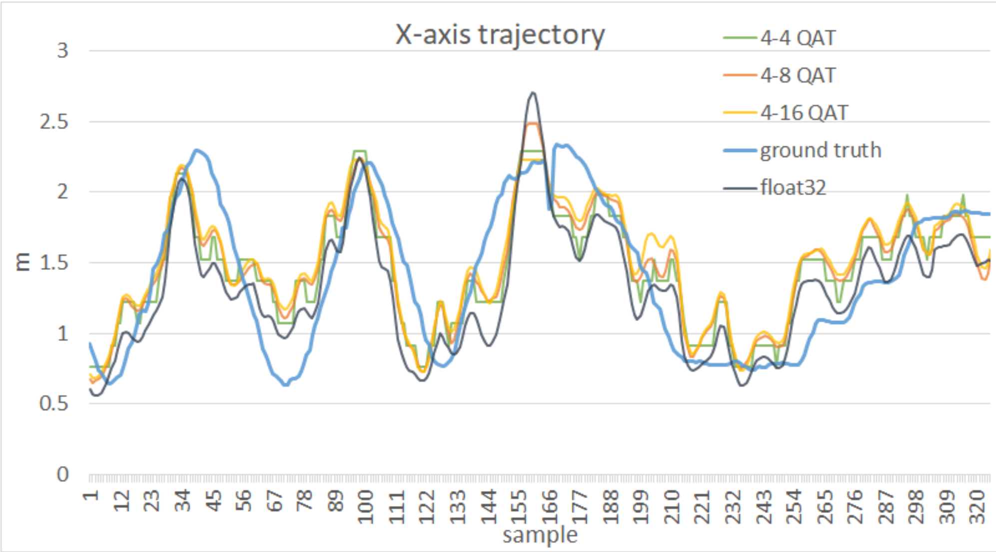
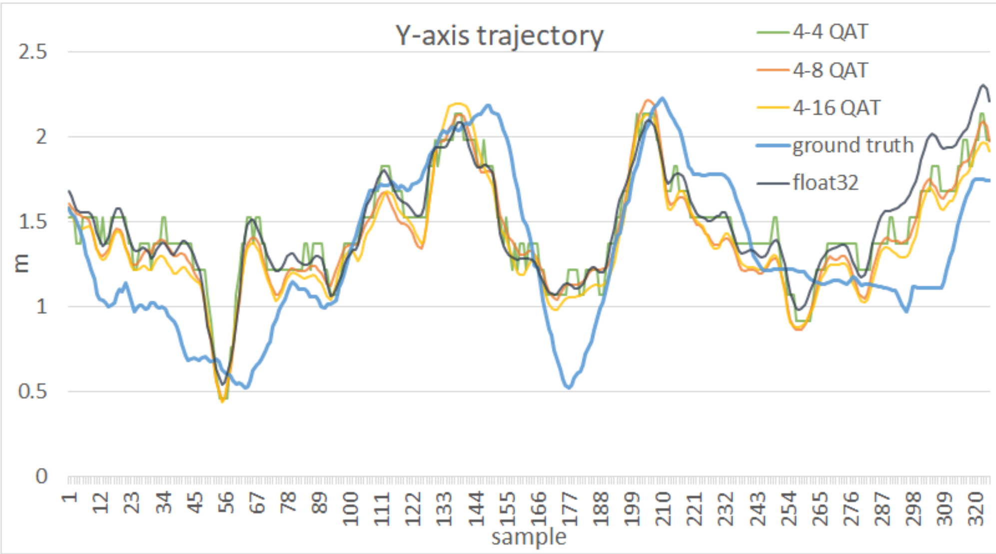


Figure 4.1: Best quantization points collected from the experiments for the multilayer perceptron network

Figure 4.2 shows the X and Y trajectories inferred by the three hardware optimum models and the ground truth reference. The smoothest trace is the 4-16 and even the more similar to the reference. Actually, the 4-4 trace is really sharp due to the lower resolution and, even if the training results report a better ADE value, this quantization scheme could not be acceptable for the indoor tracking task.



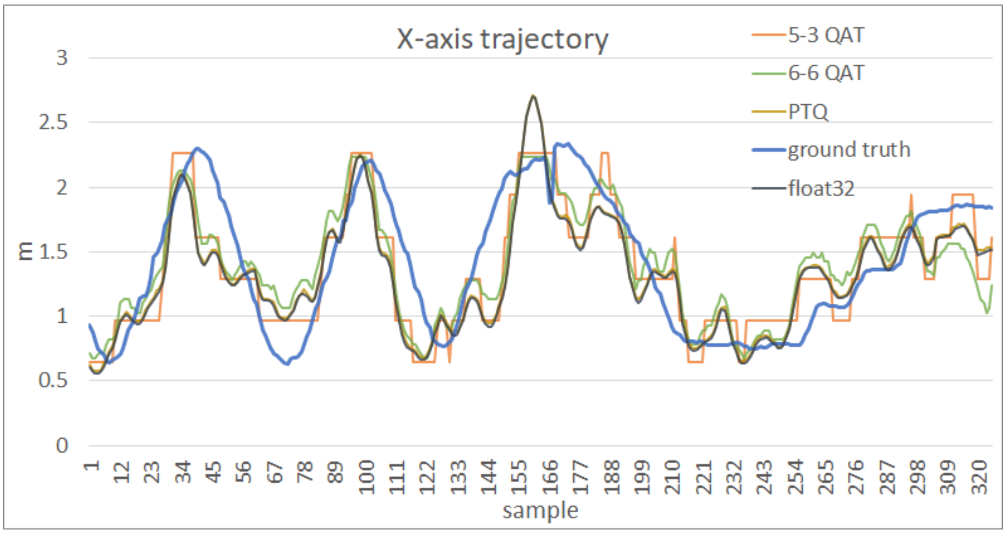
(a) X-axis predicted trajectories and ground truth



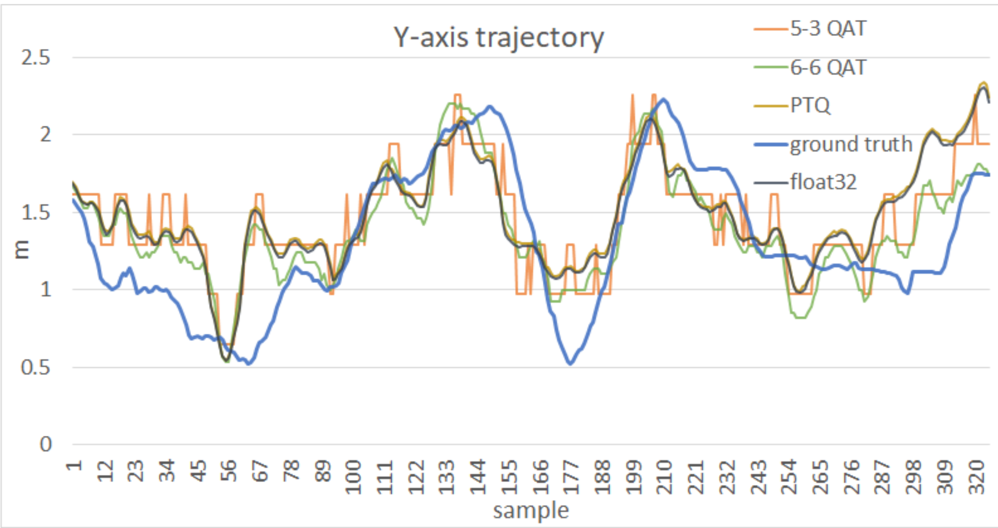
(b) Y-axis predicted trajectories and ground truth

Figure 4.2: Inferred trajectories by hardware promising quantized multilayer perceptron models

In Figure 4.3 the same trajectories graphs are shown but with the 2 models



(a) X-axis predicted trajectories and ground truth



(b) Y-axis predicted trajectories and ground truth

Figure 4.3: Inferred trajectories by pareto quantized multilayer perceptron models

in the pareto-points under the reference line reported before, the 6-6 and 5-3 quantized models. The 5-3 traces discretize too much the output and with too much oscillating behaviour. Instead, the 6-6 model traces are the most promising till now, because they are the closest to the reference and also the smoothest. The post-training quantization traces have a similar quality but they are farther from the reference than the 6-6, that actually uses less bits. Anyway the poor precision achieved by all these models is caused by the model itself, since the MLP architecture was already less suitable for the task.

The same discussion is done also for the convolutional neural network, the most promising architecture so far analysed. In Figure 4.4 is shown the graph with all the optimum points together with the pareto points and the hardware optimum.

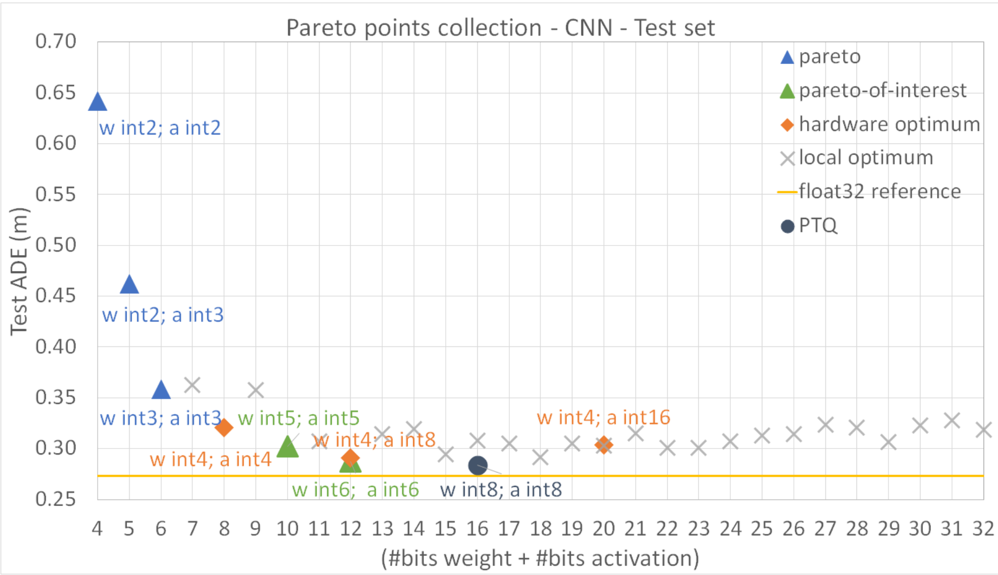
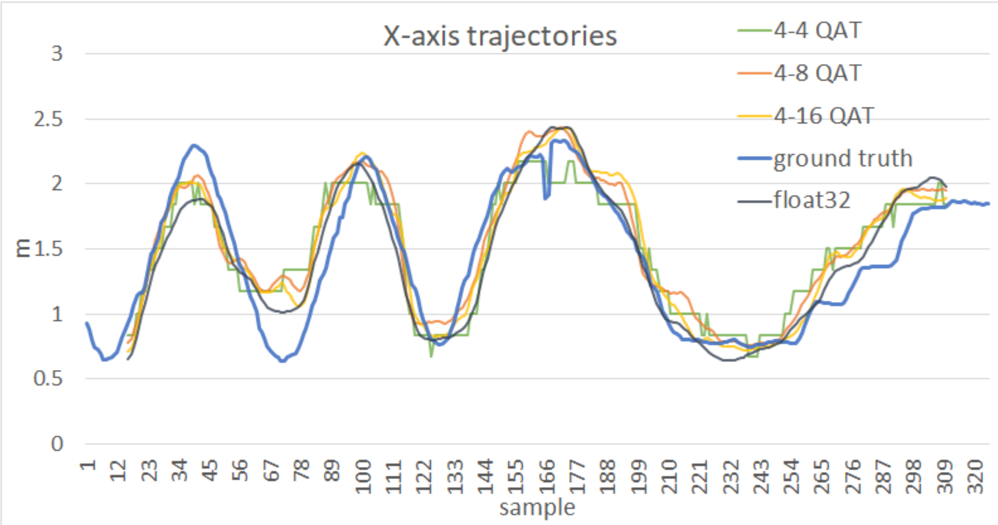


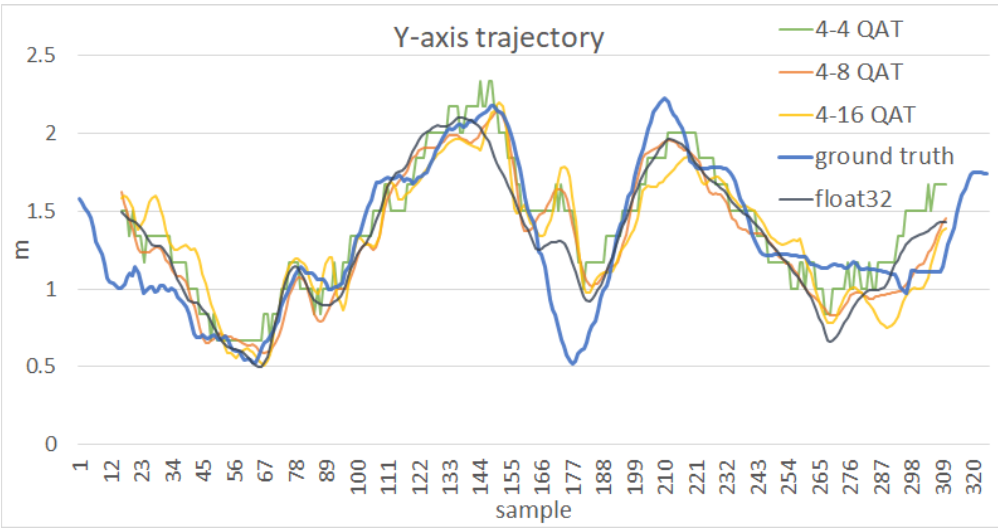
Figure 4.4: Best quantization points collected from the experiments for the convolutional neural network

For this neural network architecture, the quantized models perform slighty worse than the float32 reference, as expected. The three hardware optimum points are really close to the pareto points, as for the 4-8, or even they are themselves pareto points, as the 4-4 and 4-16 models. Even for this network architecture, the hardware optimum points can be labeled as best choices, since they perform really similar to other quantization schemes but they are more suitable for hardware deployment.

In Figure 4.5 are shown the X and Y traces inferred by the hardware optimum models. The stair shape in traces for models quantized on few bits it is always present, as for the MLPs. In particular, for the 4-4 model the inferred trajectories are affected too much by this bad behaviour, while the other two models shows more smooth traces and so can be suitable for hardware deployment.



(a) X-axis predicted trajectories and ground truth

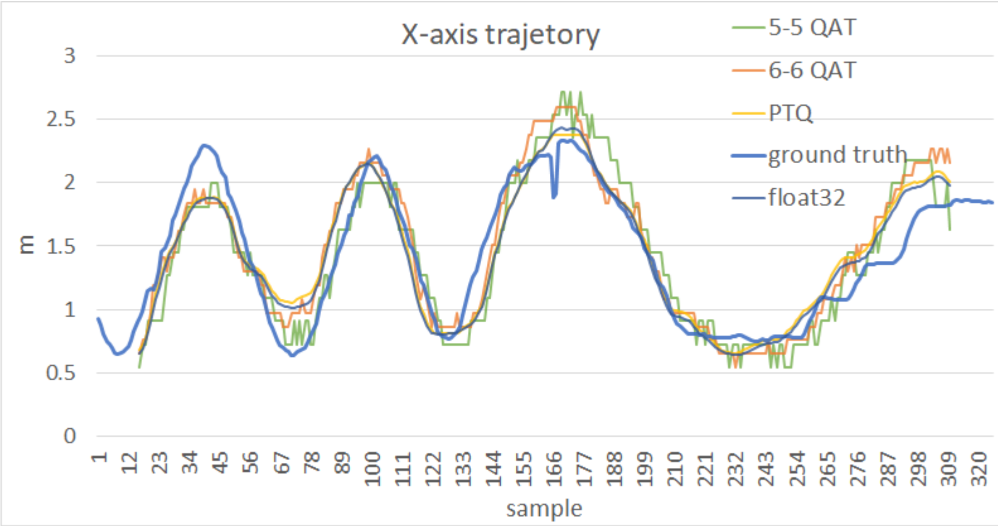


(b) Y-axis predicted trajectories and ground truth

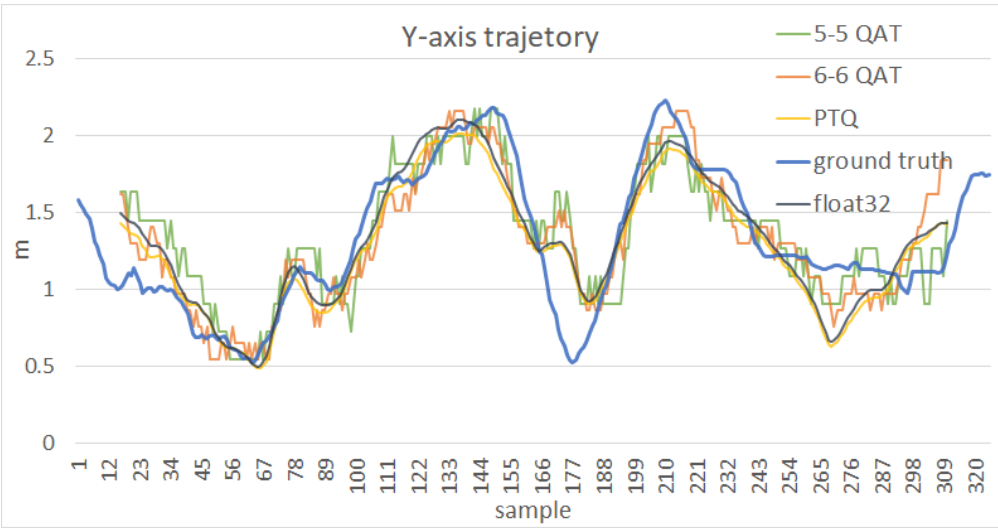
Figure 4.5: Inferred trajectories by different hardware promising quantized convolutional neural networks

Looking at Figure 4.6 the only model that can be competitive with the others already presented, is the post-training quantized one, that shows smoother traces and also closer to the reference.

The reason why a lot of quantized models perform better than the float32,



(a) X-axis predicted trajectories and ground truth



(b) Y-axis predicted trajectories and ground truth

Figure 4.6: Inferred trajectories by pareto quantized convolutional neural networks

looking at the MSE and ADE, is of poor resolution itself: degrading the representational precision, the quantized models in average start to be wrong less than the float32 thanks to their stair shape inferred trajectories. For this reason it is not convenient to look only at these metrics to evaluate the quality of the quantization, but also to look at shape of the trajectories produced during the test phase.

Chapter 5

Conclusion

The main results obtained with quantization techniques applied to convolutional neural network and multilayer perceptron network are:

- with the small MLP network the quantized version with the QAT technique performs better than the PTQ method using less bits
- for the larger CNN model the PTQ technique performs slightly better than the QAT, but this last one uses less bits

The quantization techniques analysed are very different in their general behaviour and brings different performance loss. For the QAT, about 220 different quantized MLP and CNN models have an accuracy within 10% from the float32 reference model. In particular, about 150 MLP networks were performing better than the reference, but the trajectories inferred have too high first and second derivatives. So, even if with a loss of accuracy, the quantization techniques explored can be used effectively to optimized the proposed models in a faster and cheaper implementation, that uses less hardware resources and reduces the latency.

Future work can be focused on trying other quantization algorithms with other neural network architectures. In particular, it can be exploited better the Tensorflow "blank" quantizer that can be used to implement a custom quantization algorithm.

Bibliography

[1] A. L. Samuel. «Some Studies in Machine Learning Using the Game of Checkers». In: *IBM Journal of Research and Development* 3.3 (1959), pp. 210–229. DOI: 10.1147/rd.33.0210 (cit. on p. 1).

[2] Andrew Ng. *Machine Learning*. URL: https://www.coursera.org/learn/machine-learning/ (cit. on p. 2).

[3] Andrew Ng. *Neural Networks and Deep Learning*. URL: https://www.coursera.org/learn/neural-networks-deep-learning/ (cit. on p. 5).

[4] Andrew Ng. *Deep Learning Specialization*. URL: https://www.coursera.org/specializations/deep-learning (cit. on p. 6).

[5] Keiron O'Shea and Ryan Nash. *An Introduction to Convolutional Neural Networks*. 2015. arXiv: 1511.08458 [cs.NE] (cit. on pp. 6, 7).

[6] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016 (cit. on pp. 8, 9).

[7] O. Bin Tariq, M. T. Lazarescu, J. Iqbal, and L. Lavagno. «Performance of Machine Learning Classifiers for Indoor Person Localization With Capacitive Sensors». In: *IEEE Access* 5 (2017), pp. 12913–12926. DOI: 10.1109/ACCESS.2017.2721538 (cit. on pp. 8–11).

[8] Nicoletta Sportillo. «Neural network optimization for indoor person localisation using capacitive sensors». Master's degree Thesis. Politecnico di Torino, 2020. URL: http://webthesis.biblio.polito.it/id/eprint/14506 (cit. on pp. 10–12, 27–29, 32, 40, 41, 47, 50).

[9] Osama Bin Tariq Alireza Ramezani Akhmareh Mihai Lazarescu and Luciano Lavagno. «A tagless indoor localization system based on capacitive sensing technology». In: *Sensors* (2016). DOI: 10.3390/s16091448 (cit. on pp. 10, 28).

[10] François Chollet et al. *Keras*. https://keras.io. 2015 (cit. on p. 10).

[11] Mohamed Hebiri and Johannes Lederer. *Layer Sparsity in Neural Networks*. 2020. arXiv: 2006.15604 [cs.LG] (cit. on pp. 14, 15).

[12] Michael Zhu and Suyog Gupta. *To prune, or not to prune: exploring the efficacy of pruning for model compression*. 2017. arXiv: 1710.01878 [stat.ML] (cit. on pp. 14, 16).

[13] Song Han, Huizi Mao, and William J. Dally. *Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding*. 2016. arXiv: 1510.00149 [cs.CV] (cit. on pp. 16–18).

[14] *TensorFlow model optimization*. Tensorflow.org. 2020-08-12. URL: https://www.tensorflow.org/model_optimization/guide (cit. on p. 17).

[15] *Model optimization*. Tensorflow.org. 2020-08-12. URL: https://www.tensorflow.org/lite/performance/model_optimization (cit. on pp. 18, 19, 21, 26).

[16] *Quantization aware training*. Tensorflow.org. 2020-08-22. URL: https://www.tensorflow.org/model_optimization/guide/quantization/training (cit. on pp. 20, 21).

[17] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. *Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference*. 2017. arXiv: 1712.05877 [cs.LG] (cit. on pp. 21, 26, 34).

[18] Pulkit Bhuwalka and Alan Chiao. *Tensorflow model optimization Quantization Operations*. URL: https://github.com/tensorflow/model-optimization/blob/master/tensorflow_model_optimization/python/core/quantization/keras/quant_ops.py (cit. on p. 24).