POLITECNICO
MILANO 1863

SCHOOL OF INDUSTRIAL AND INFORMATION ENGINEERING

TECHNOLOGIES FOR ARTIFICIAL INTELLIGENCE

SMART PHYSIOTHERAPIST:

UWB-RADAR BASED ARMS AND LEG MOVEMENT DETECTION

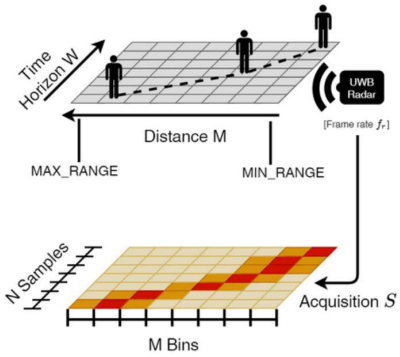
A.Y. 2024/2025

Students:

Riva Alessandro - 245140
Stefanoni Luca - 252192

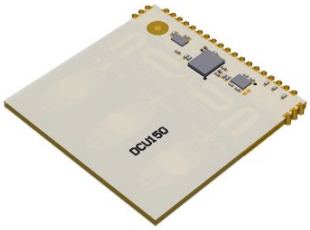
Abstract

This project aims to develop a simplified AI model that could be a first step toward the assistance of an injured person during rehabilitation activities. In the future where Artificial Intelligence will be part of our daily life, in this application, during recovery practice time, the person will be constantly monitored by a radar to carefully check the quality of the movements and suggest corrections or improvements. The scenario just described is a future hope and requires a lot of work to be put on the table, but this project aims in that direction. The radar is an optimal solution to be employed since it guarantees the privacy of the person and it is a less invasive technology. The scope of the actual project is to check the validity of the Ultra Wide Band radar technology in terms of recognising different body movements, e.g. an arm raised, a leg in movement… The results will be presented in terms of the accuracy and the inference time of the AI models to recognise various body configurations and motions. Before diving into the technical part of the project is fundamental to state the requirements of the final result: it must be runnable and deployable on tiny devices, i.e. Arduino Nano, thus the area of interest of this work is Tiny Machine Learning.

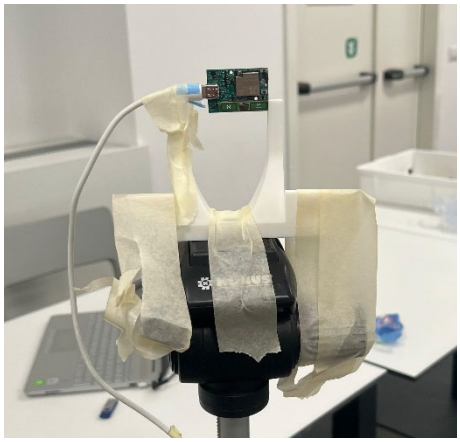


Formalisation of the problem

This chapter discusses the acquisition method, the classes of the model, and the definition of the input. As anticipated, the technology employed in the acquisition of the data is the UWB radar DCU 150 provided by Truesense shown in the figure aside. This device has two antennas mounted on it that can acquire data up to 5 meters from its position at $25\ Hz$. The functioning of this technology will be not deepened in this report, while more space will be given to the approach to the problem that has been followed.



DCU150 Radar by TrueSense



The horizontal radar configuration

The radar is connected through a USB cable to the computer, therefore how to position the device is one of the first decisions to be addressed. The set-up must be as stable as possible to prevent fallings or tiltings of the radar during the acquisition, consequently, two positions are exploitable: vertical or horizontal. For the scope of this project, the AI model will address the distinction between left and right motions of the limbs, thus a **horizontal placement** seems to be the best solution. Due to the greater distance the signal will travel, the two antennas will more easily recognize the two motions.

Project goals

Before discussing all the other aspects of the problem formalisation it is important to set the goals of the project. The work will be considered fulfilling once the developed model can classify body movements in the correct category with an accuracy of at least 95%. Fixing this value is quite ambitious, but, since this project wants to be the first step towards real-time assistance during rehabilitation activities, this threshold is required to have a solid and effective basement on where to start.

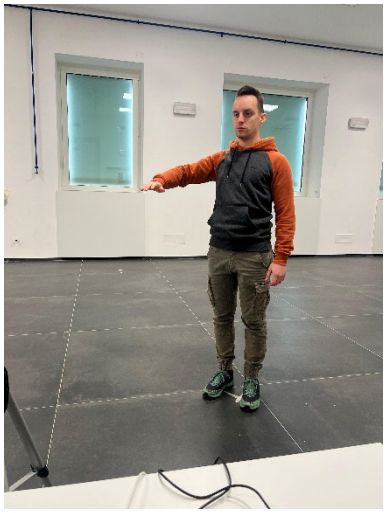
Dataset collection approach

The creation of a well-crafted dataset is a crucial step in the development of an AI model; to obtain good results it should be well-balanced and contain high-quality data. To achieve this goal it has been decided to structure the dataset into five classes each with 100 inputs inside. These classes represent five distinct body configurations, an arbitrary choice made for this project, that the AI model should be able to classify into the correct bucket.

The five classes are listed below, accompanied by explanatory pictures:

Class 1: Static standing position

Class 2: Right arm up

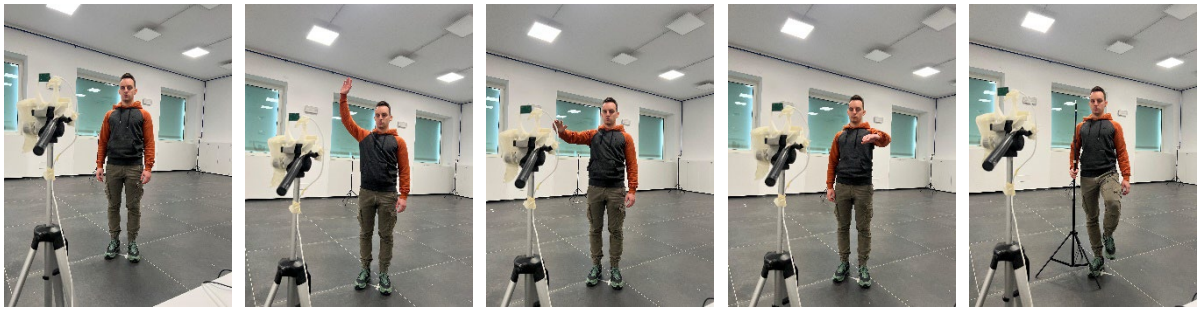


Class 3: Right arm moving

Class 4: Left arm moving

Class 5: Left leg moving

The person has been positioned at $1{,}5\ m$ in front of the radar sensor, on top of a cross placed on the floor. The body configurations and motions have been collected always in this position:



Another fundamental parameter to select is the length of the acquisition window. Some rules of thumb exist, but this variable must be chosen carefully considering the scenario in which the model will go to work. Since our application is directed towards an AI model that will assist injured people, it needs to be considered that the pain could slow down the speed of the body motion of this person. This issue is associated with the movement classes, there are no particular problems with static classes, but, given that to create a quality dataset all the classes should have the same length, a **5-second window** is selected. To simplify and speed up the collection process of the data it has been decided to acquire a stream of 25 seconds and then cut it into five windows of 5 seconds with the script *"function_prepare.py"*. All the body configurations do not suffer from possible data loss during the chopping of the windows, therefore **no overlapping** between them was implemented.

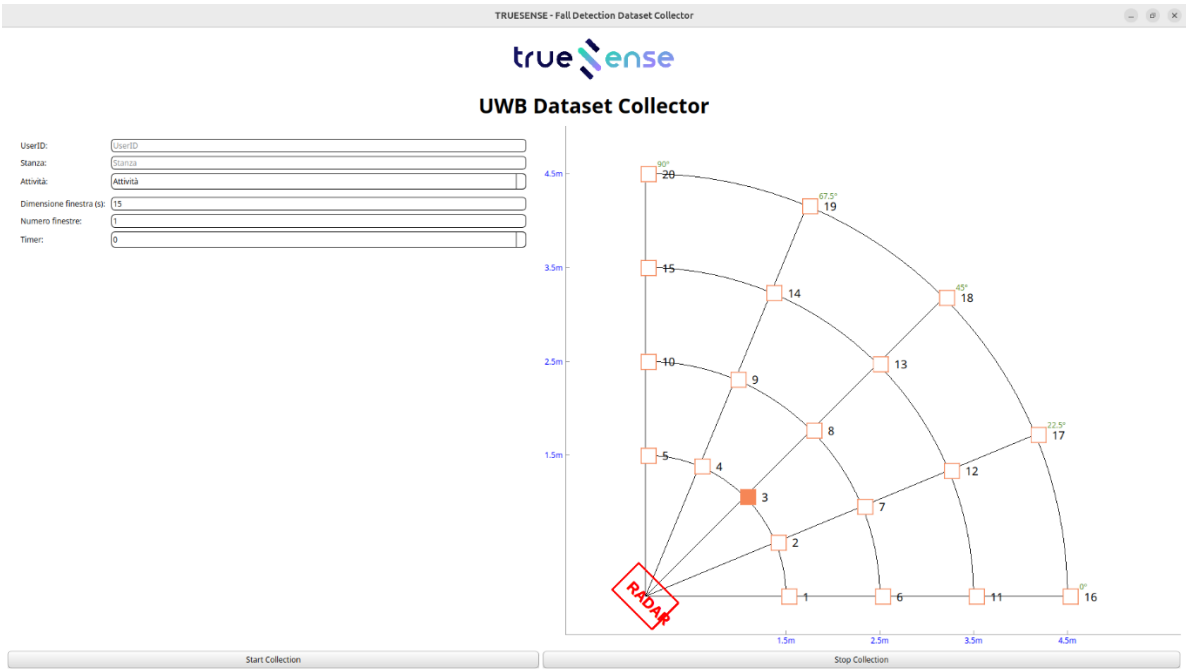
The next table summarises all the observations about the dataset until now:

Classes	Number of data collected	Length of single window
Static standing position	100	$5s$
Right arm up	100	$5s$
Right arm moving	100	$5s$
Left arm moving	100	$5s$
Left leg moving	100	$5s$
NO OVERLAPPING		

These input data will be divided randomly into two batches, one for the training of the model and the other for the testing of its accuracy. A usual split is:

- **Training set: 80 %**
- **Test set: 20%**

For this project, it has been decided to maintain this subdivision for each class to ensure the balance.



The acquisition window of the software for the radar TrueSense

The parameters like the length of the window, the choice of the class, and the number of inputs acquired per time are displayed on the left, while the right panel allows the selection of the position of the target in the space.

Input data characteristics

This section will discuss the passage from the raw data acquired by the two antennas to the input images of the neural network. The data created by the acquisition procedure are matrices of *space-time (x-y)* that can conveniently be visualised and processed as images.

The data acquired are delivered by the software with the *.npy* extension organised in this way:

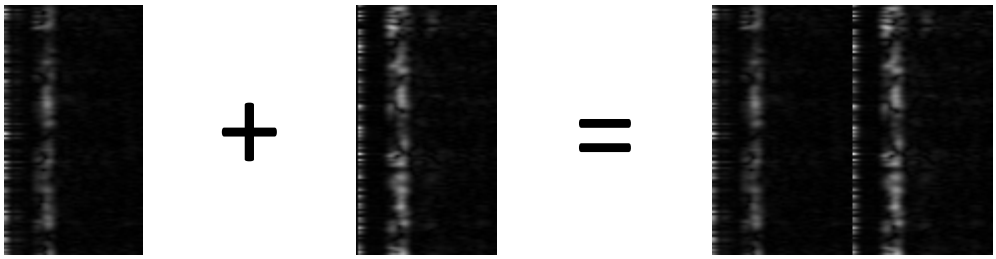
subject.action.room.[distance, angle]_date_radarmodel_antenna.npy

For each antenna, one file is created where the same acts as the transmitter and receiver. Additionally, two more files are created where instead one antenna is the transmitter and the other the receiver. The final part of the file name is labelled in this way:

- **tx** (Transmitter) + antenna's number
- **rx** (Receiver) + antenna's number

For example, the code tx0rx0 means that the first antenna acts both as a transmitter and receiver, the code tx0rx1 means that the first sends the signal that is gotten by the second, and so on.

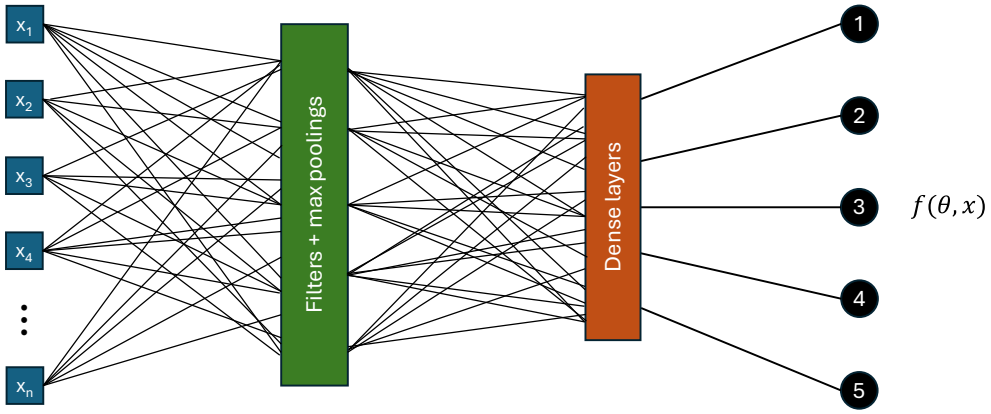
To implement our neural network, only the data coming from the same antenna are taken into account, thus two files for every acquisition window. These two files are processed again by the Python script "*function_prepare.py*" which takes *.npy* inputs and provides a combined image as output. The process is graphically explained below:



The first figure is an example of the radar acquisition of the raised right arm just using the first antenna (tx0rx0). This latter is then combined with the radar data from the second antenna (tx1rx1) into a unique image. This has now become one element of the first dataset given as input to the neural network. The final dimension of the image is $140\ x\ 125\ pixels$.

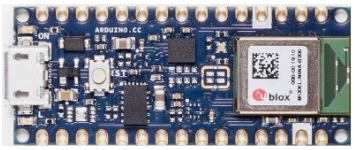
Model development

The idea is to process the radar data as images, combined following the procedure just described above. The neural network will be schematised in this way:



High-level representation of our neural network structure

The input data passes through a block containing filters and max poolings, followed by one or more dense layers; the final output is the classification result, associated with the probability of the five classes. The research on the best compromise between the parameters was conducted starting from a simple network and then trying to adjust the features, aiming for the best accuracy. As anticipated in the Abstract the AI model under development must be suitable to work on Tiny devices, such as Arduino Nano 33 BLE. This means that not only accuracy is important, but the focus must also be on the memory parameters, RAM and Flash, and the inference time of the solution. The physical limitations of the hardware, taken from the official website of Arduino, are listed below:



Arduino Nano 33 BLE

- RAM memory: $256kB$
- Flash memory: $1MB$

The space occupied on the device by the mere model is just a part of the equation, then it must be added the one related to the inputs, feature maps, firmware… Therefore, during the check of the model requirements, the previous physical limits should be corrected more conservatively:

- RAM memory: $180kB$
- Flash memory: $800kB$

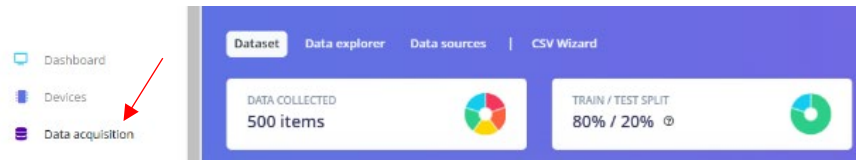
By respecting these thresholds the model should be compatible to run on the Arduino device, anyway, this will be assessed in a dedicated section at the end of this report.

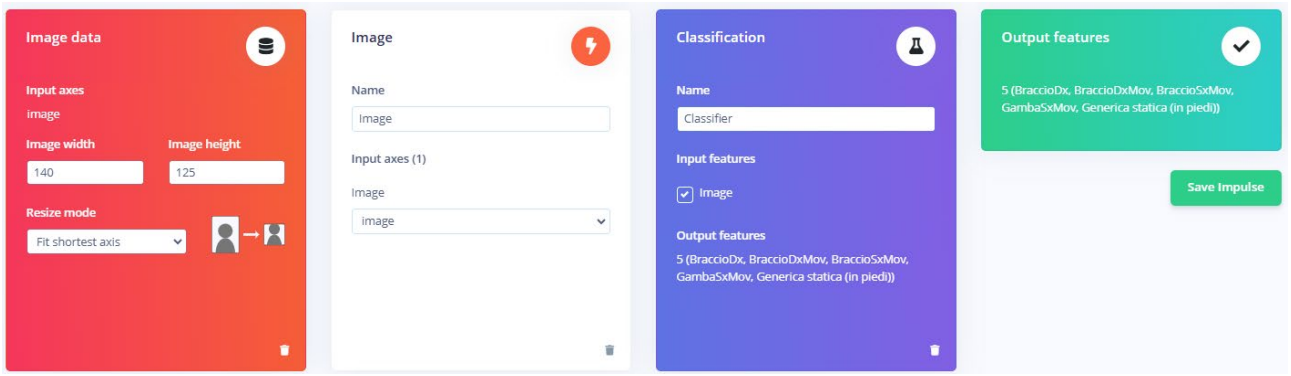
Coding platforms

Another fundamental step in developing an AI model is picking the platform and the programming language. Nowadays a lot of projects dealing with artificial intelligence exploit Python as programming language and Tensorflow, which is a powerful software library for machine learning and AI.

For this project, it has been chosen to stick with this solution, but rely on the website Edge Impulse (https://edgeimpulse.com/) which provides a user-friendly interface able to streamline all the procedures for creating simple neural networks. Edge Impulse offers an intuitive interface to import the dataset, images for the current project, and a block-based programming section where the parameters of the classifier can be set.

To import the dataset the procedure is straightforward: it is sufficient to click on the button “Data Acquisition” placed on the right and upload the image in a *.png* format for example. Once the dataset is created, left-click on the button “Create Impulse” to start the procedure of setting up the neural network.





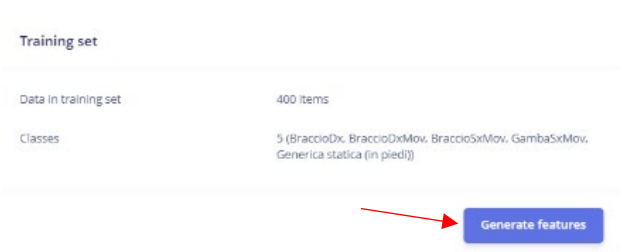
The window for the creation of an impulse on Edge Impulse

In this screen, the input characteristic can be fixed and processing and learning blocks must be added as well to respectively generate the features of the input and develop the model.

Having defined all the boundary conditions of the problem, it’s time to implement the first neural network.

First NN set-up

Before starting the configuration with the parameters of the classifier one must prepare the dataset by extracting the features from the images. To carry out this operation Edge Impulse makes available a specific section called *Image* where the data can be converted into grayscale and subsequently processed to generate the features on which the classification will be based.



Shifting to the classifier settings, the main training parameters of a neural network are:

I. Number of training cycles (epochs)
II. Learning rate (speed of learning of the NN)
III. Validation set size (inside the option *Advanced training settings*)

The I. is the complete passing through of all the datasets once and it must be sufficiently high to be suitable with the learning rate (II.), i.e. the convergence velocity of the model. This parameter is preset by Edge Impulse at 0,0005; during the work it has been observed that this number is a good compromise to obtain high accuracies while preventing overfitting, thus it is left almost unaltered throughout the model trials. Finally, the validation set size is the percentage of the training data (N.B. it is a percentage of the 80 % of data of the training set, not the test set) used for a first evaluation of the model accuracy.

Having understood the meaning of the previous parameter the attention now shifts to the set-up of the neural network. Edge impulse itself supplies a series of default values, therefore, to have a starting point from which begin, for the first neural network, they will be left unchanged.

Its structure is shown below:

The output layer is the final dense layer of the model having five outputs, equal to the number of classes. For simplicity, the Flatten and Dropout layers were not initially mentioned in the scheme of the previous page. The Flatten layer converts the data into a one-dimensional array for the following dense layers, while the Dropout layer temporarily removes some forward and backward connections with a dropout probability *p* indicated in brackets. This latter operation helps prevent overfitting.

Training this network for 30 epochs, with a learning rate of 0,0005 and 20% as validation set size the results are:

Training accuracy	*Test accuracy*
88,9%	76,8%

These numbers seem promising, however by looking at the RAM requirements and the inference time of the current model it is easy to understand that other solutions must be explored:

A target inference time demanded for this application could be under $700/800ms$.

Study on the "filters-max pooling" layers

The results with the default network are not terrible in terms of accuracy, but the aim is to reduce the network's computational requirements and even improve the accuracy further. This section will investigate the effect of changing the configuration of the two "filters + max-pooling" layers, leaving one unique dense layer with five classes as output. All the detailed results and NN set-ups are available for consultation in the appendix of this document, here the reader will find just observations and comments.

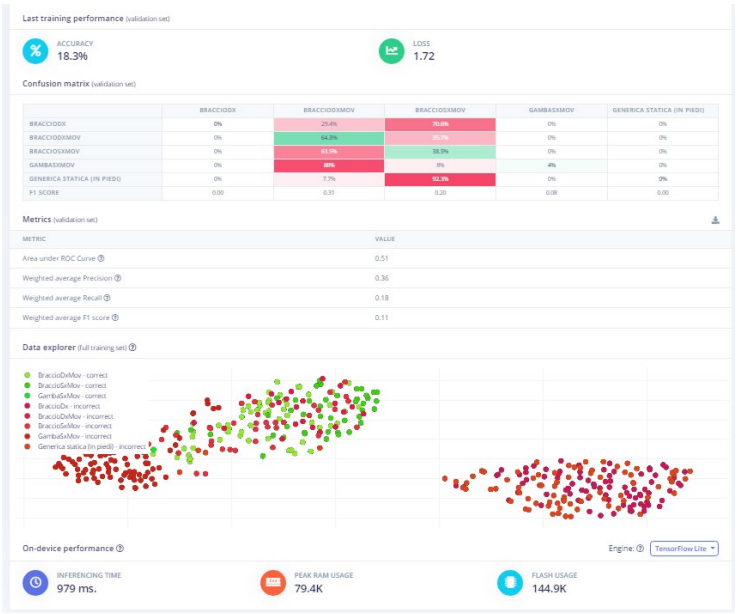
The presence of two layers of $3x3$ filters, for 48 kernels, strongly influences the RAM demand of the first model. In these further trials, it has been tried to reduce this in two ways: by reducing the number of filters in each layer or by passing from two to one block "filter + max pooling". Sometimes also the kernel size has been modified by trying configurations with $4x4$ or $2x2$ dimensions.

The outcomes of these tests are positive, meaning that most of the time the memory demand has dropped from $415.1\ kB$ to an average of almost $230\ kB$ (outlier values from too-bad models have been discarded). The requirements on the flash memory are always satisfied with this type of network, while the inference time is often over $1000ms$, which is considered an excessive number. In conclusion, the accuracies of these networks are close to 90% on the training data, and 85% on the test set, averagely speaking.

Trail with the transfer learning block of Edge Impulse

Inside the section "*Create Impulse*" of Edge Impulse a Transfer Learning block could be selected. This block facilitates the adaptation of pre-trained models to new, particularly useful working with image classification problems or keyword spotting. One fundamental characteristic of Transfer Learning is the obligation to supply squared input images to the model. The only pre-trained network that could fit the Arduino computational features is the MobileNetV1, which asks for $96x96$ input images. Therefore, since the dimensions of the combined radar images are $140\ x\ 125\ pixels$, they must be resized and squashed.

The hardware demand is suitable, but the accuracy of a network so designed is dreadful:



An accuracy of $18{,}3\%$ over five classes implies that the model is trying randomly to classify the data. Given these results, it is clear that this approach is not suitable for working with radar data and it has been decided to abandon it.

Squashed inputs network

The Transfer Learning block has not been very successful, however the way to design a neural network by reducing the input dimensions could be a valid solution, exploiting the classification block. These trials are carried out to reduce the RAM requirements, which, until now, have been too demanding by all the good models.

The input images are squashed before entering the neural network and their final dimension is decided by the user. By doing this operation the input is scaled down and the rest of the network benefits in terms of computation, but the accuracy must be verified because squashing the data carries with itself a loss of information. It has been tried with two approaches: starting from the original dimensions of $140\ x\ 125\ pixels$, factors of $0{,}8$ and $0{,}5$ have been arbitrarily chosen for the size reduction, bringing the images to $112\ x\ 100$ and $70\ x\ 63$ respectively.

All the details about the network structure and the training results can be deepened in the appendix, the next table summarises the results for the RAM usage and accuracy:

Trial	*Squashing*	*RAM Usage*	*Test accuracy*
1	80%	$135.5kB$	81,8%
2	50%	$108.8kB$	77,6%
3	50%	$55.8kB$	72.5%

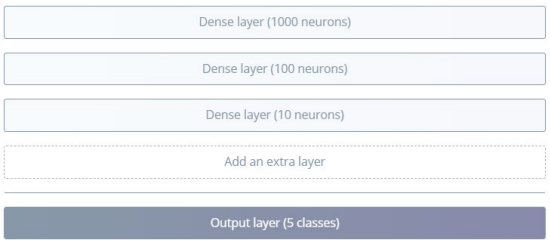
Despite all the models having a RAM demand below the $180kB$ threshold, just the first one can be deployable on the Arduino, because of the Flash memory requirements of the 50% squashing architectures. The reason for this high usage of Flash memory of this type of model will be discussed better in the next chapter, it is linked to the addition of dense layers before the output one.

The investigation on the squshed inputs has led to this conclusion: the reduction of the RAM usage is evident and the first neural network could be actually deployable, however, the accuracy seems too compromised. Another solution to reduce the input, more effective and with stunning results from all the points of view will be presented later in the report.

The addition of other dense layers

Until now the examined models have an architecture with just one dense layer block: the output one with five classes. The only exception to this rule was two 50% squashing networks as anticipated. This section will investigate the inclusion of other dense layers before the last one.

The addition of a dense layer carries an increase in the Flash memory request because the greater the neurons the greater the number of parameters (weights and biases) to be stored on board. These supplementary dense layers are added following a decreasing path in terms of the number of neurons, progressively reducing the network and channelling the connections to the final five classes. The first neural network realised with this approach involves the blocks shown below before the output layer:



Starting from 1000 neurons the next layers are scaled down by a factor of 10 for three times until the final classification. By looking at the accuracy result of this model, one can notice the evident improvement considering the old configurations:

Training accuracy	*Test accuracy*
$95{,}0\%$	$87{,}1\%$

Accuracy is surely one crucial parameter in the selection of the model, but this latter must be compatible with the hardware. For the current solution, the hardware requirements are:

Edge Impulse returns $0{,}0kB$ both for RAM and Flash memory, probably because the usage is so heavy that cannot return a valid number. This hypothesis can be confirmed also by looking at the huge inferencing time of the NN which is above 12,5 seconds. Despite the best accuracies obtained so far, this model is completely inadequate to assess the project's scope. This approach of adding extra dense layers could still be open, but the number of neurons needs to be strongly shrunk.

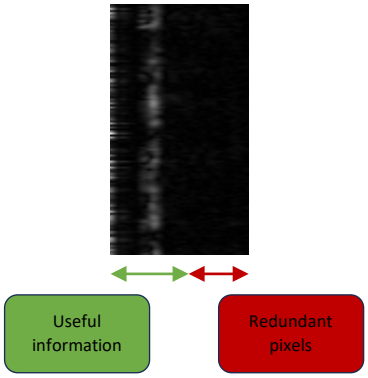
Following this direction, other trials have been carried out:

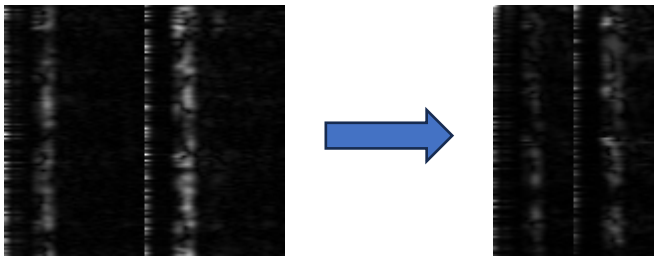
Trail 1		*Trial 2*	
Dense layer (200 neurons) Dense layer (20 neurons) Add an extra layer Output layer (5 classes)		Dense layer (300 neurons) Dense layer (50 neurons) Dense layer (10 neurons) Add an extra layer Output layer (5 classes)	
Training accuracy	90,0%	*Training accuracy*	76,3%
Test accuracy	83,2%	*Test accuracy*	50,5%
INFERENCING TIME 911 ms. PEAK RAM USAGE 209.4K FLASH USAGE 6.8M		INFERENCING TIME 599 ms. PEAK RAM USAGE 136.1K FLASH USAGE 1.7M	

The results of these analyses are not satisfactory, firstly because the accuracy is not excellent, it is significantly low in the second model, and secondly, because the network cannot be deployable on the Arduino due to the high Flash memory consumption. Also reducing the total number of neurons the Flash memory usage exceeds $1\ MB$.

Data cleaning approach

All the architectures tested until now seem to be not able to reach accuracies over 90% while respecting all the hardware limitations. To try to overcome this barrier, this chapter presents a new data-cleaning approach. From previous considerations, it is known how the reduction of the input dimensions helps in the shrinking of the network, however, the squashing has not worked properly. Instead of compressing data, in this case, the downsize operation is carried out by removing redundant information about the acquired radar signals. The antennas of the radar can measure up to a $5m$ distance, however, the person is positioned at $1{,}5m$ meaning that at least $30\ pixels$ over the total width of 70 could be removed in each image without the loss of any useful information. Theoretically, this stripping of data could be even more strong, but it has been decided to keep $30\ pixels$ as a conservative value. By doing this operation for the single antenna data, the total gain over each combined image in the dataset is $60\ pixels$!





The final size of each image has now become 80 x 125 pixels.

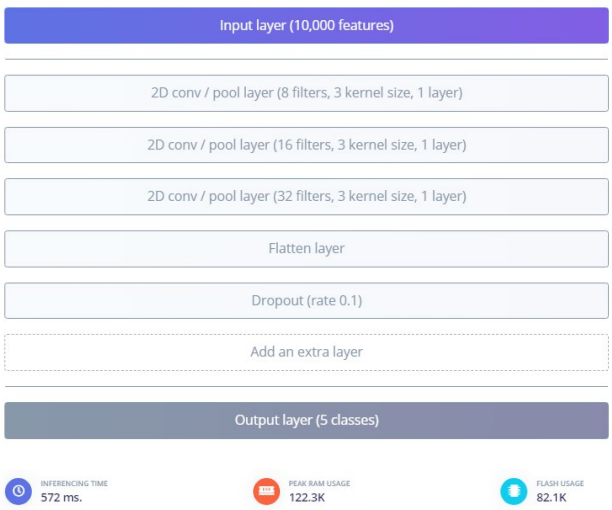
By adjusting the input in this way all the approaches used until now could be explored under new clothes, with the knowledge that the hardware requirements will be reduced. From the first test, it has been clear that every network structure has now a better ability to recognise the correct body movements more effectively and efficiently, taking up even less inference time. All the trials are detailed in the appendix, where every architecture is reported, this section makes some observations about the type of uncertainties of the networks and studies the best two.

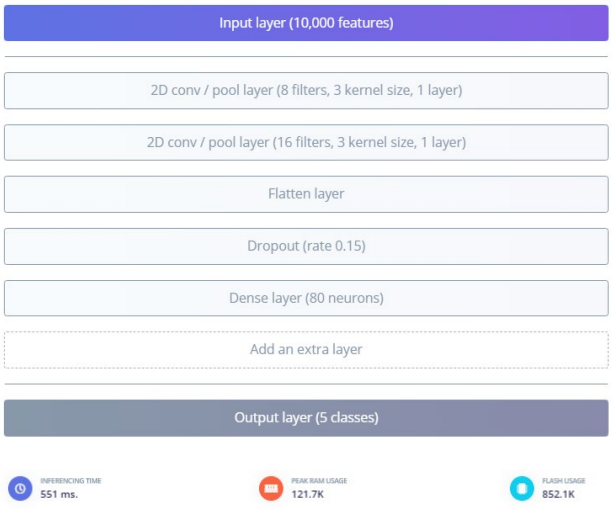
For developing this new model, the architectures with only one output layer and the others with one or more extra dense layers have been tested. In many cases, it has been noticed that most of the accuracy losses were linked to the distinction between the movement of the left and the right arm, like in the example reported here:

	BRACCIODX	BRACCIODXMOV	BRACCIOSXMOV	GAMBASXMOV	GENERICA STATIC
BRACCIODX	100%	0%	0%	0%	0%
BRACCIODXMOV	0%	93.3%	6.7%	0%	0%
BRACCIOSXMOV	0%	18.2%	81.8%	0%	0%
GAMBASXMOV	0%	0%	0%	100%	0%
GENERICA STATICA (IN	0%	0%	0%	0%	100%
F1 SCORE	1.00	0.90	0.86	1.00	1.00

The model can perfectly recognise three classes out of five, but it is slightly ambiguous between those two. Sometimes this issue has been related to the static body position, but the scenario previously described is by far the most recurrent situation. To delve deeper into this aspect, a dedicated set of trials will be prepared later.

After numerous tests, two valid models, having comparable accuracy, have been chosen to be the candidates for the deployment:





For the correct evaluation of the accuracy, one run of the program is not formally enough to obtain reliable results. The analysis must be carried out from a statistical point of view running multiple times the model and calculating the average and the standard deviation of the values at the end of this process. This operation assures the repeatability and the validation of the model.

MODEL: single dense				ARDUINO		
Run	**Accuracy Training**	**Loss**	**Accuracy Test**	**Inferencing Time [ms]**	**Peak RAM Usage [kB]**	**Flash Usage [kB]**
1	96.7%	0.07	99.01%	572	122.3	82.1
2	100.0%	0.08	98.02%	572	122.3	82.1
3	96.7%	0.09	96.04%	412	122.3	82.1
4	98.3%	0.09	95.05%	555	122.3	82.1
5	96.7%	0.09	98.02%	568	122.3	82.1
6	96.7%	0.09	92.08%	719	122.3	82.1
7	96.7%	0.09	96.04%	554	122.3	82.1
Mean	**97.4%**	**0.09**	**96.3%**	**565**		
σ_{st}	**1.3%**	**0.8%**	**2.3%**			

MODEL: + 80 neurons				ARDUINO		
Run	**Accuracy Training**	**Loss**	**Accuracy Test**	**Inferencing Time [ms]**	**Peak RAM Usage [kB]**	**Flash Usage [kB]**
1	98.3%	0.05	97.06%	551	121.7	852.1
2	100.0%	0.05	98.95%	512	121.7	852.1
3	98.4%	0.07	98.95%	498	121.7	852.1
4	96.7%	0.06	97.89%	516	121.7	852.1
5	100.0%	0.05	98.95%	510	121.7	852.1
6	96.6%	0.06	96.84%	506	121.7	852.1
7	96.7%	0.06	98.95%	373	121.7	852.1
Mean	**98.1%**	**0.06**	**98.2%**	**495**		
σ_{st}	**1.5%**	**0.8%**	**1.0%**			

At this point, the best model seems to be the second one with an accuracy higher than 98% and an inference time lower than $0{,}5s$! Despite this consideration, both the models will be deployable on the device because the inference time reported by Edge Impulse is an estimation and must be verified on board to appoint the best network. Before making the final comments on the deployment, the next chapter will thorough the distinction between left and right movements, which turned out to be the most critical type of classification.

Two classes analysis

So far, networks have been analysed and optimised for the complete version of the model: the classification of all the five body motions and positions. As anticipated, this additional section wants to focus now on a more detailed analysis of the two arm-movement classes, aiming to highlight:

1) The differences between exploiting the data coming from a single antenna versus both antennas;
2) The performance of the five-class optimized network (their architectures are reported in the latter chapter) when applied to a dataset of two classes.

The first comparison aims to determine whether increasing the number of antennas improves the recognition of right arm movement compared to the left. The patient's action is identical for both limbs but differs in symmetry relative to the body's centre. The dataset is represented by 200 not combined images, resized to $40x125\ pixels$; it is desired to continue using the optimised input. The images are evenly split into "*BraccioDxMov*" and "*BraccioSxMov*", below an example of two of them is reported:

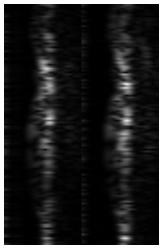
BraccioDxMov

BraccioSxMov

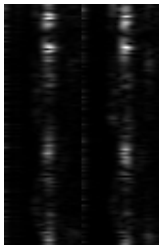
The best-performing networks have been employed, but the results reveal a significant drop in accuracy for both training and testing compared to five-class cases:

Model	Training accuracy	Test accuracy
5 classes (single dense)	96,7%	99,01%
2 classes 1 antenna (single dense)	*62,5%*	*70%*
5 classes (+ 80 neurons)	98,3%	97,06%
2 classes 1 antenna (+ 80 neurons)	75%	70%

Subsequently, the case with two antennas was analyzed.



BraccioDxMov



BraccioSxMov

The input images remain 200, but resized to $80x125\ pixels$, consistent with the training dimensions used for the five-class setup.

The results are as follows:

Model	Training accuracy	Test accuracy
5 classes (single dense)	96,7%	99,01%
2 classes 1 antenna (single dense)	62,5%	70%
2 classes 2 antennas (single dense)	*56%*	*87,18%*
5 classes (+ 80 neurons)	98,3%	97,06%
2 classes 1 antenna (+ 80 neurons)	75%	70%
2 classes 2 antennas (+ 80 neurons)	88%	89,74%

The five-class-trained model performs significantly better at distinguishing between these two specific classes compared to the model trained exclusively on them. The use of two antennas led to an increase of nearly 17 percentage points in test accuracy, suggesting that radar acquisition with dual antennas is more effective in recognizing symmetrical movements as expected.

One notable aspect is the training accuracy: while the network having an additional dense layer improves, the first achieves only 56%. Training the same architecture on two classes instead of five results in poor performance. This could be due to the network being too large for the available data or insufficient training time. It is also important to remember that a network specifically trained for five classes will almost certainly not be the best for two.

To try to address this, additional training was conducted increasing the learning rate equal to 0,001 and the number of epochs, equal to 20 in both cases.

Model	Training accuracy	Test accuracy
5 classes (single dense)	96,7%	99,01%
2 classes 2 antennas (single dense)	96%	94,87%
5 classes (+ 80 neurons)	98,3%	97,06%
2 classes 2 antennas (+ 80 neurons)	100%	92,31%

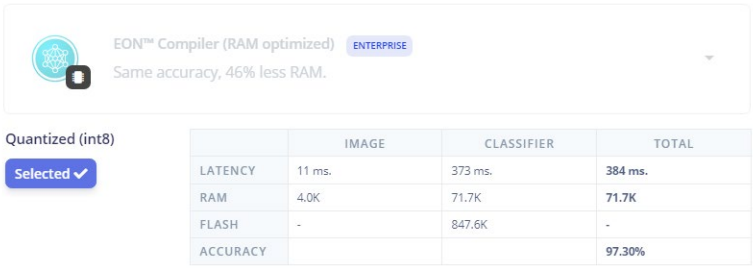
This adjustment significantly improved performance, making the results comparable to those of the five-class model, even employing just a single antenna. This led to the conclusion that the selection of training parameters is crucial for network optimization, as it depends on multiple factors, including the dataset size.

Deployment of the model on Arduino Nano 33 BLE

The final step of the project involves deploying the two best-performing models onto the Arduino Nano 33 BLE. The primary objective is to verify key execution metrics:

1) Ensuring that the inferencing time estimated by Edge Impulse matches real-world values
2) Confirming that RAM and Flash usage remain within the device's constraints

Deploying a model onto the device requires following a simple procedure. On Edge Impulse, after selecting the desired model, the Deployment section provides the option to export it as an **Arduino Library**. This choice generates a library that, once imported into the Arduino IDE, allows the use of precompiled examples to load the model directly onto the device. In particular, the EON Compiler offers an estimation of the model's key parameters, including latency, RAM usage, Flash memory usage, and accuracy.



By clicking the *Build* button, a ZIP file containing all necessary deployment data is generated.

Once the library is obtained ei-ai_project_3rd_cut-arduino-1.0.1 , the next step is loading it into the **Arduino IDE**. In this case, the "*nano_ble33_sense_camera*" example is used. This example captures an image from the camera and uses it as input for the neural network. While the captured image is not radar-based, the primary goal of this phase is to measure the execution time for **preprocessing (DSP)** and **classification**, rather than evaluating the accuracy of the results.

The final step involves compiling the example and uploading the code onto the device . Once this process is completed, the *Serial Monitor* window allows real-time monitoring of the output and analysis of execution times.

```
Starting inferencing in 2 seconds...
Taking photo...
Predictions (DSP: 16 ms., Classification: 921 ms., Anomaly: 0 ms.):
Predictions:
  BraccioDx: 0.00000
  BraccioDxMov: 0.00000
  BraccioSxMov: 0.00000
  GambaSxMov: 0.00000
  Generica statica (in piedi): 0.99609
```

Single dense model 5 classes

```
Starting inferencing in 2 seconds...
Taking photo...
Predictions (DSP: 16 ms., Classification: 703 ms., Anomaly: 0 ms.):
Predictions:
  BraccioDx: 0.00000
  BraccioDxMov: 0.00000
  BraccioSxMov: 0.00000
  GambaSxMov: 0.00000
  Generica statica (in piedi): 0.99609
```

+ 80 neurons model 5 classes

This result validates that both the model architectures are compatible with the capabilities of the Arduino Nano 33 BLE, ensuring execution without errors or interruptions. Excluding the time for the acquisition of the image, which is not of interest because it is not performed with the TrueSense radar, for the two models these are the inferencing time results:

Model	DSP time	Classification time	Total time	Estimated time
Single dense	$16ms$	$921ms$	$937ms$	$565ms$
+ 80 neurons	$16ms$	$703ms$	$719ms$	$495ms$

From this table, one can understand that the models run on the device without issues, however, the time calculated by Edge Impulse is quite underestimated compared with reality. By looking at the performance on board the second model with an additional dense layer can be considered the best one overall.

Final comments

The deployment of the neural networks on the device closes this project which has been a successful first step towards the assistance to injured patients. All the fixed goals, in terms of accuracy, restricted hardware requirements and inference time have been reached. It is hoped that this work will serve as a solid foundation from which new improvements and developments can be conducted in the future.

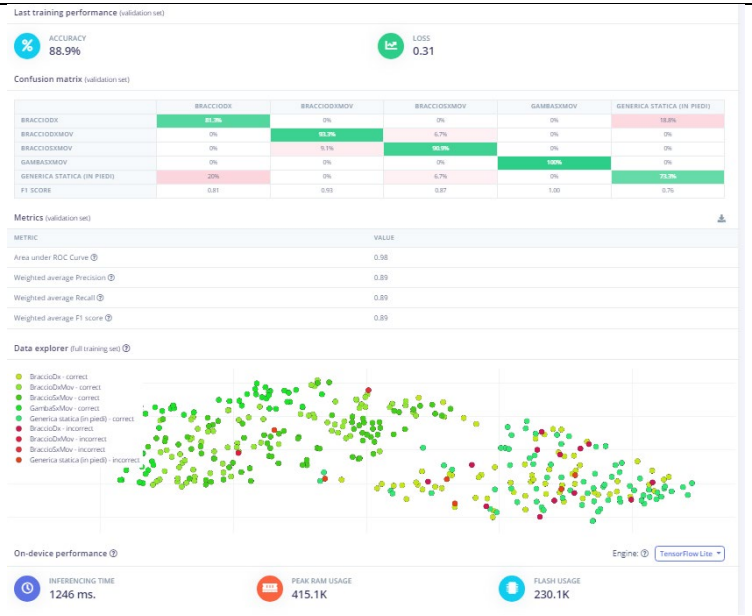
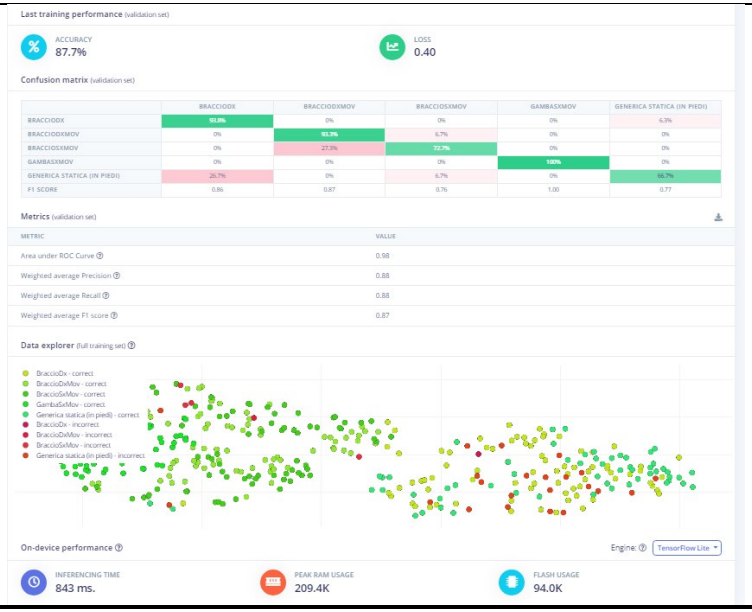
Material repository

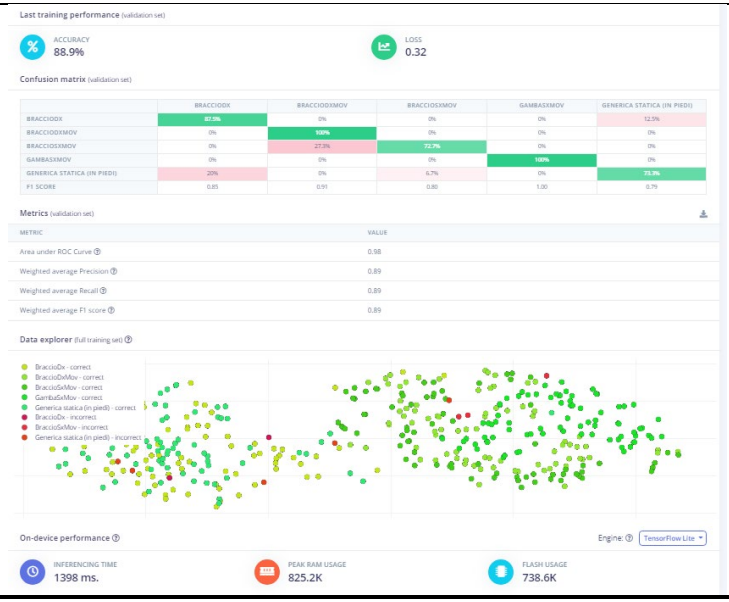
The current document with the associated datasets and all the material used in this project is Open Source and available in the following GitHub folder: https://github.com/AleRiva01/Smart-Physiotherapist.git.

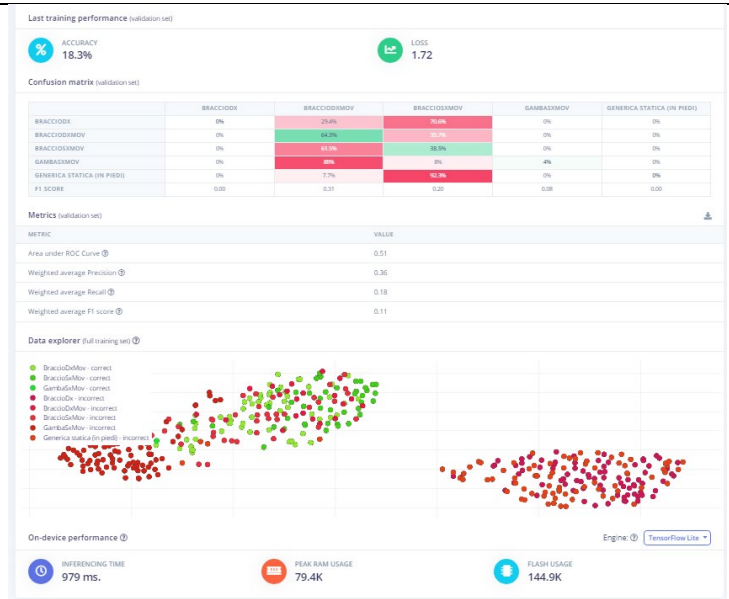
For the redistribution and use conditions please refer to *BSD 2-Clause License* on GitHub.

Appendix

The tests related to the developed models can be found below.

ARCHITECTURE (train-test 80-20)	RESULTS	ACCURACY
Img 140x125, classifier, 30 epoch, val set 20%		
Input layer (17,500 features) 2D conv / pool layer (16 filters, 3 kernel size, 1 layer) 2D conv / pool layer (32 filters, 3 kernel size, 1 layer) Flatten layer Dropout (rate 0.25)		76,77%
Img 140x125, classifier, 15 epoch, val set 20%		
Input layer (17,500 features) 2D conv / pool layer (8 filters, 3 kernel size, 1 layer) 2D conv / pool layer (8 filters, 3 kernel size, 1 layer) Flatten layer Dropout (rate 0.1) Add an extra layer Output layer (5 classes)		86,87%

Img 140x125, classifier, 15 epoch, val set 20%		
Input layer (17,500 features) 2D conv / pool layer (32 filters, 3 kernel size, 1 layer) Flatten layer Dropout (rate 0.1) Add an extra layer Output layer (5 classes)		82,83%
Img 140x125, classifier, 15 epoch, val set 20%		
Input layer (17,500 features) 2D conv / pool layer (4 filters, 3 kernel size, 1 layer) 2D conv / pool layer (4 filters, 3 kernel size, 1 layer) Flatten layer Dropout (rate 0.1) Add an extra layer Output layer (5 classes)		82,61%

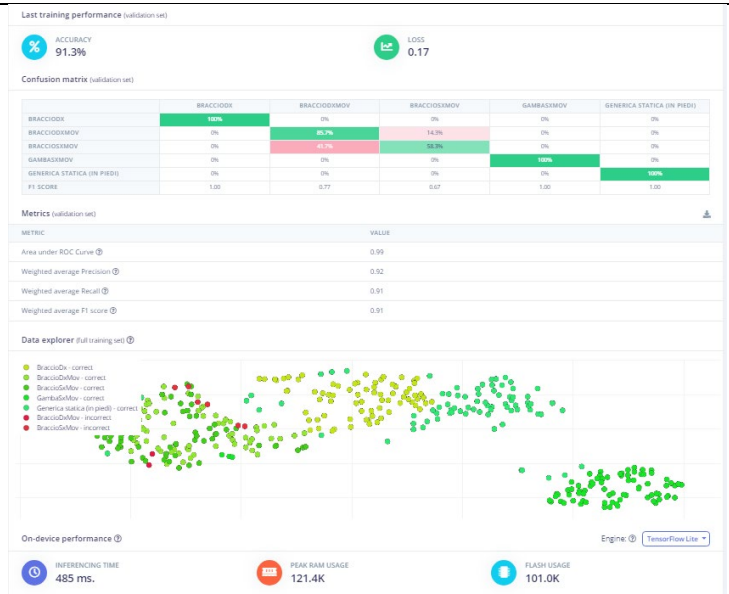
<table>
<tr><td colspan="3">Img 140x125, classifier, 15 epoch, val set 15%</td></tr>
<tr>
<td> 
</td>
<td> 
</td>
<td>84,78%</td>
</tr>
<tr><td colspan="3">Img 125x125 squash, transfer learning, 10 epoch, val set 20%</td></tr>
<tr>
<td> 
</td>
<td> 
</td>
<td>0,00%</td>
</tr>
</table>

Img 112x100 squashed (80%), classifier, 15 epoch, val set 20%

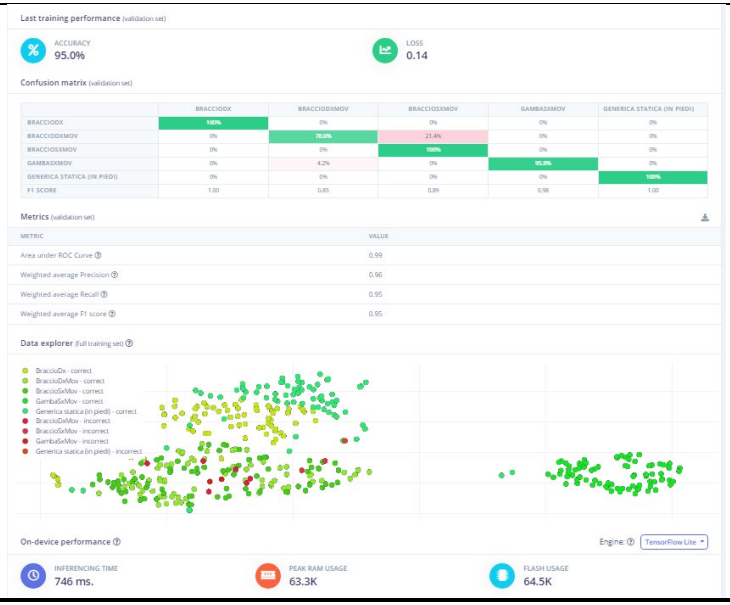
81,82%

Img 80x125, classifier, 15 epoch, val set 20%

96,00%

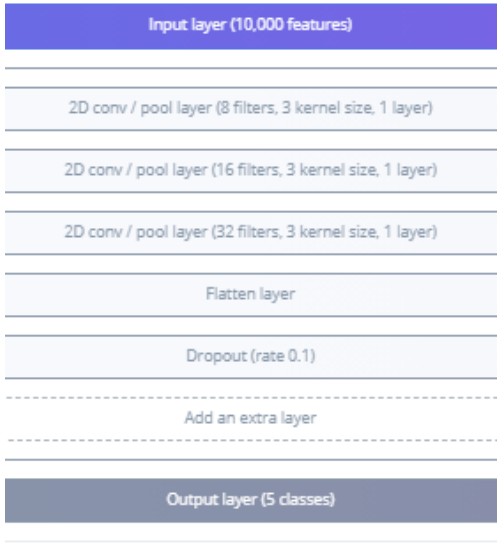
<table>
<tr><td colspan="3">Img 80x125, classifier, 15 epoch, val set 20%</td></tr>
<tr>
<td>Input layer (10,000 features)
2D conv / pool layer (8 filters, 3 kernel size, 1 layer)
2D conv / pool layer (16 filters, 3 kernel size, 1 layer)
Flatten layer
Dropout (rate 0.1)
Add an extra layer
Output layer (5 classes)</td>
<td> 
</td>
<td>95,00%</td>
</tr>
<tr><td colspan="3">Img 80x125, classifier, 20 epoch, val set 20%</td></tr>
<tr>
<td>Input layer (10,000 features)
2D conv / pool layer (8 filters, 4 kernel size, 3 layers)
Flatten layer
Dropout (rate 0.1)
Add an extra layer
Output layer (5 classes)</td>
<td> 
</td>
<td>95,00%</td>
</tr>
</table>

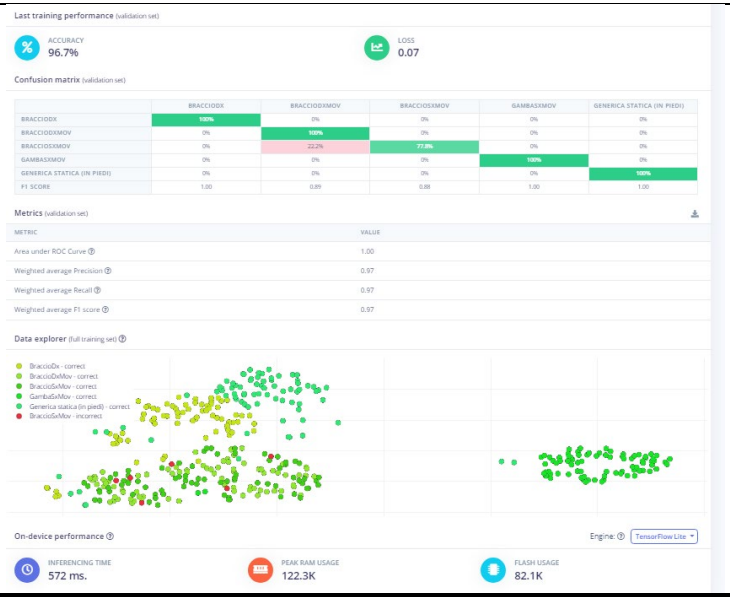
Img 80x125, classifier, 17 epoch, val set 20%



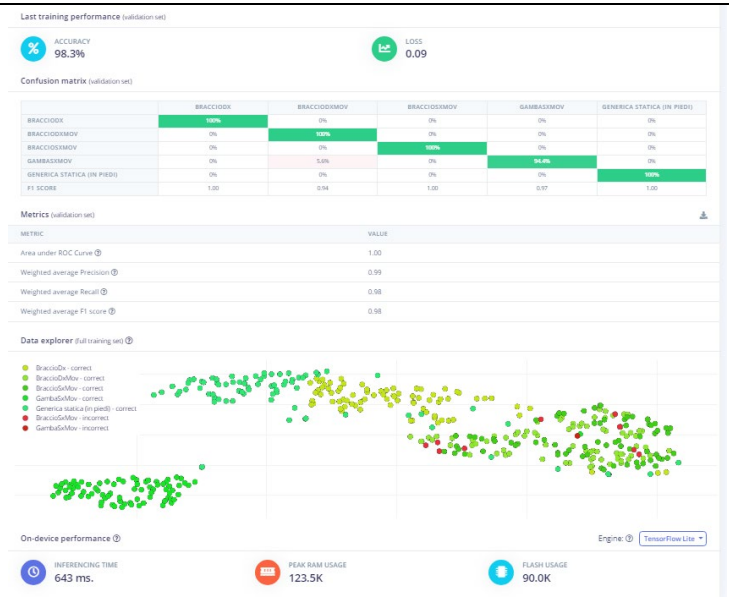
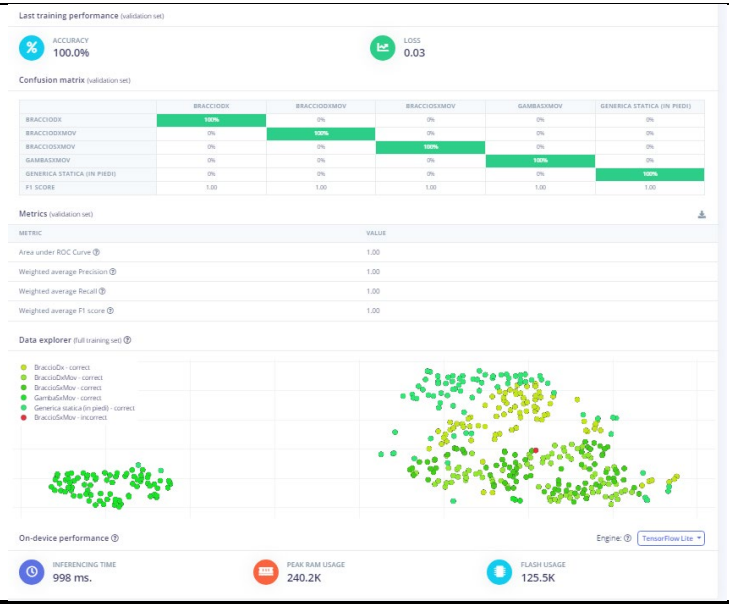
96,04%

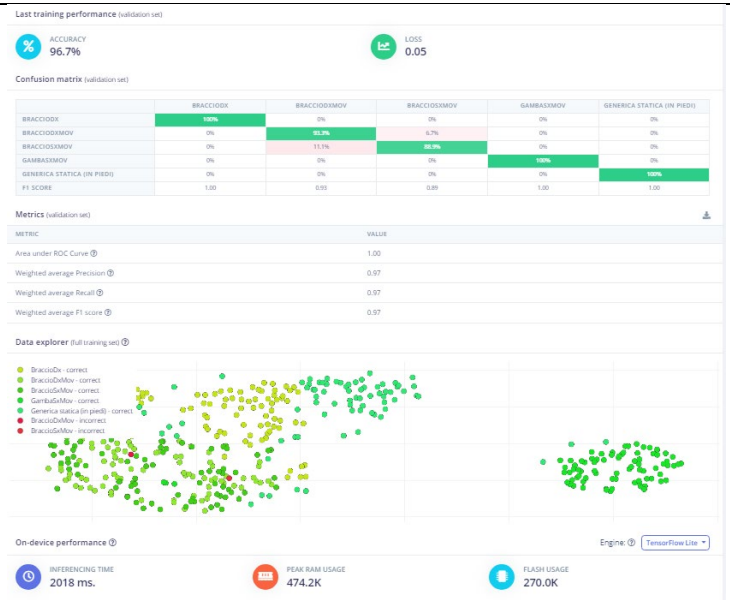
Img 80x125, classifier, 17 epoch, val set 15% BEST configuration: **Single dense model**

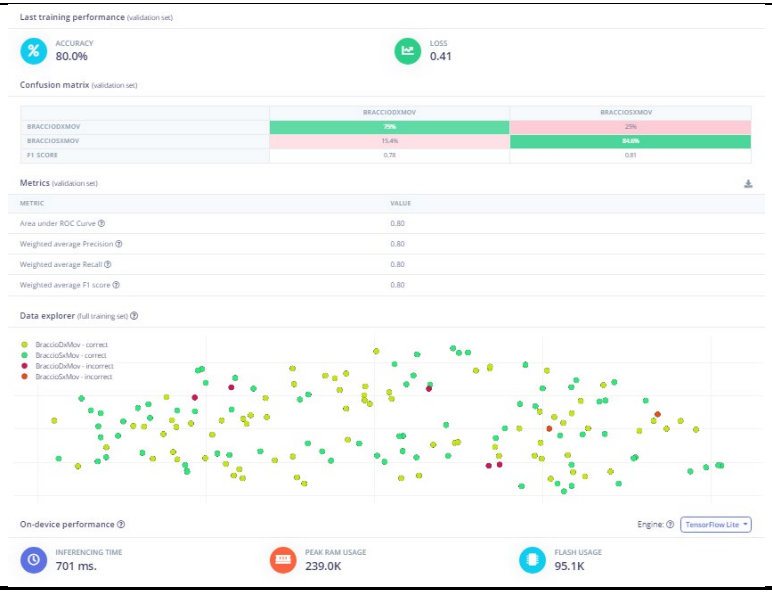


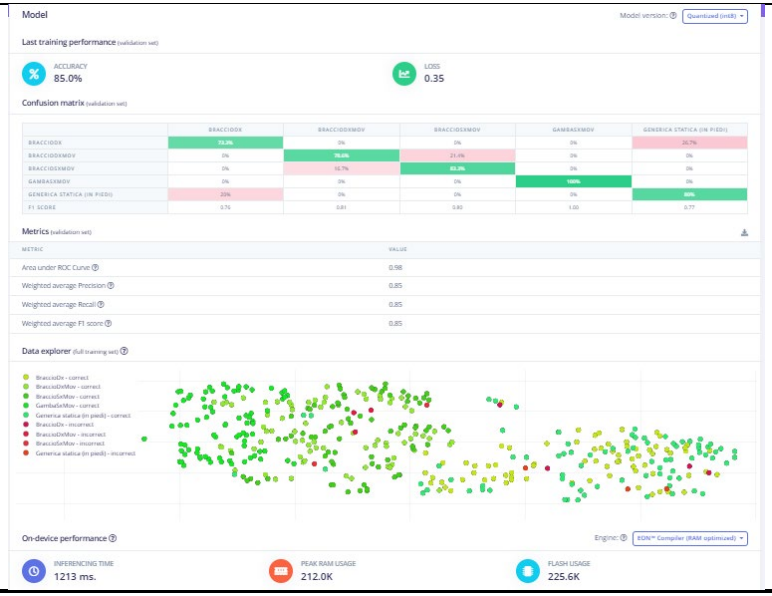
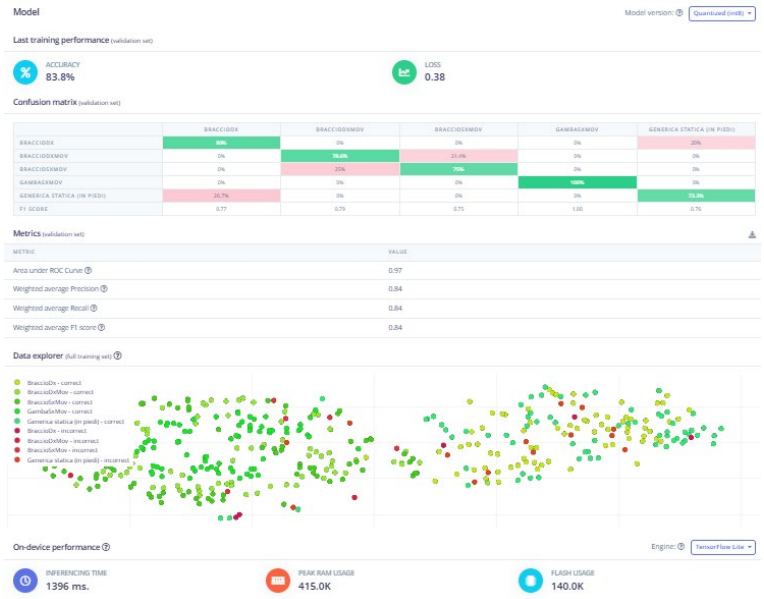


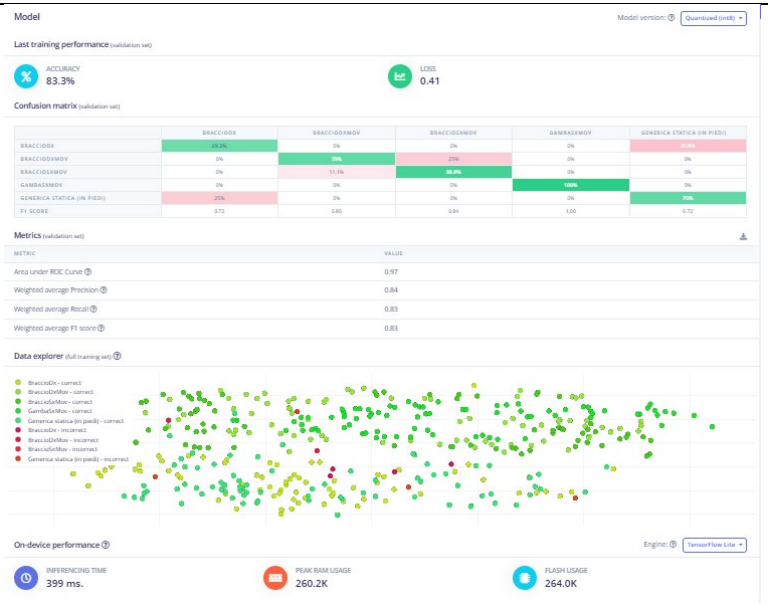
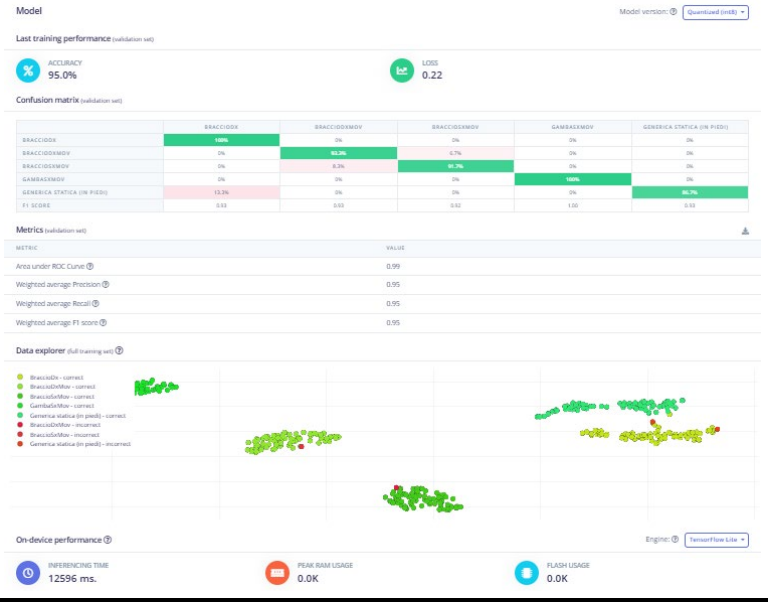
99,01%

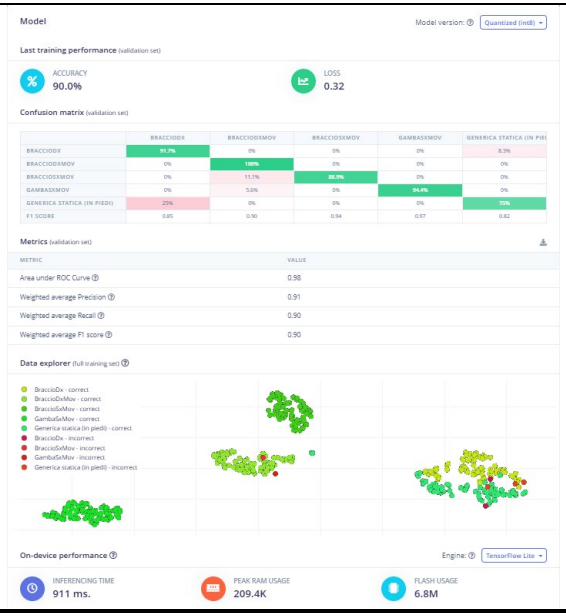
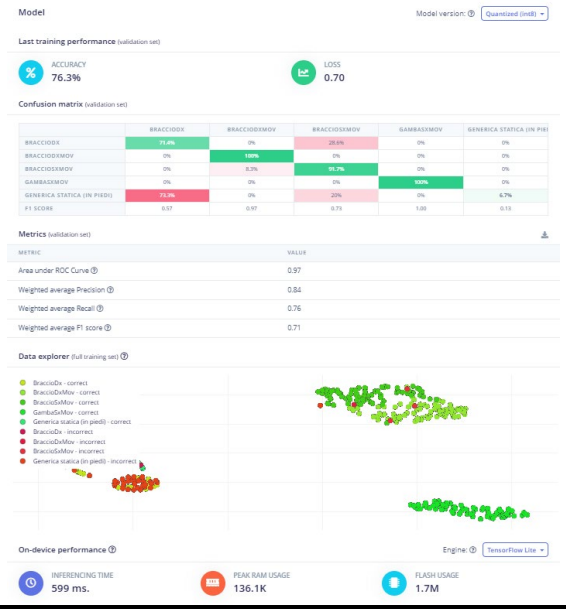
Img 80x125, classifier, 15 epoch, val set 15%		
Input layer (10,000 features) 2D conv / pool layer (8 filters, 3 kernel size, 1 layer) 2D conv / pool layer (16 filters, 3 kernel size, 1 layer) 2D conv / pool layer (32 filters, 3 kernel size, 1 layer) 2D conv / pool layer (64 filters, 3 kernel size, 1 layer) Flatten layer Dropout (rate 0.1) Add an extra layer Output layer (5 classes)		95,05%
Img 80x125, classifier, 17 epoch, val set 15%		
Input layer (10,000 features) 2D conv / pool layer (16 filters, 3 kernel size, 1 layer) 2D conv / pool layer (32 filters, 3 kernel size, 1 layer) 2D conv / pool layer (64 filters, 3 kernel size, 1 layer) Flatten layer Dropout (rate 0.1) Add an extra layer Output layer (5 classes)		96,84%

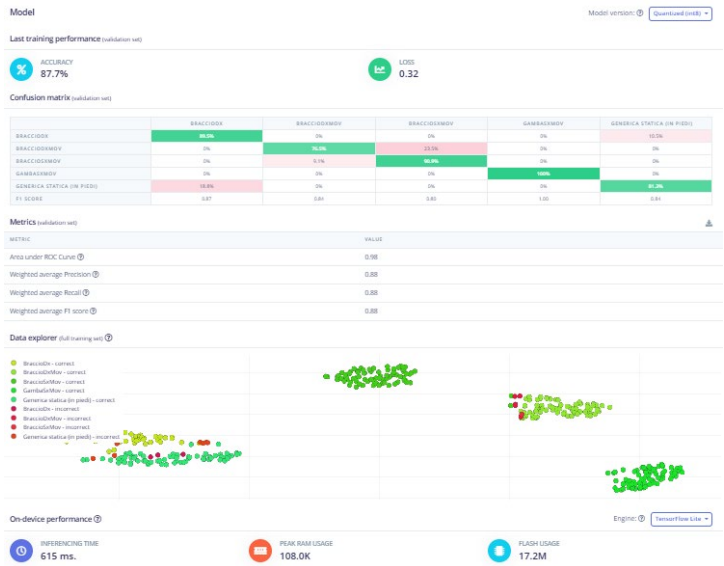
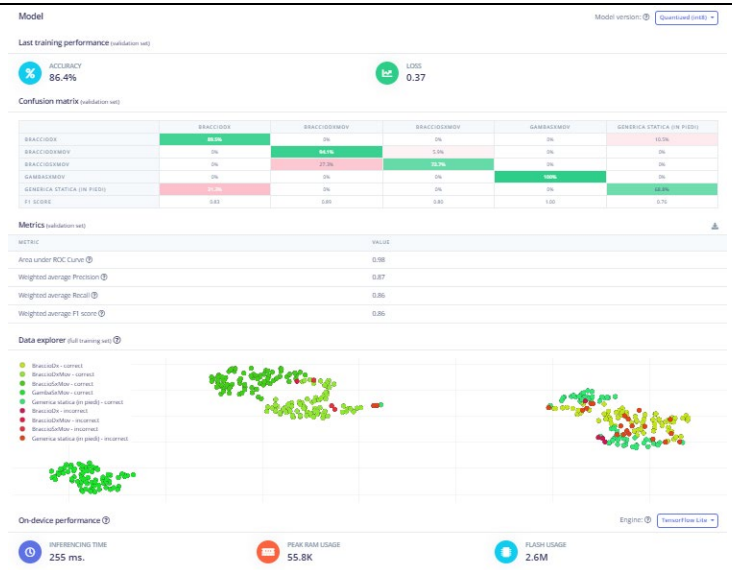
Img 80x125, classifier, 17 epoch, val set 15%			
Input layer (10,000 features) 2D conv / pool layer (32 filters, 3 kernel size, 1 layer) 2D conv / pool layer (64 filters, 3 kernel size, 1 layer) Flatten layer Dropout (rate 0.1) Add an extra layer Output layer (5 classes)			97,89%

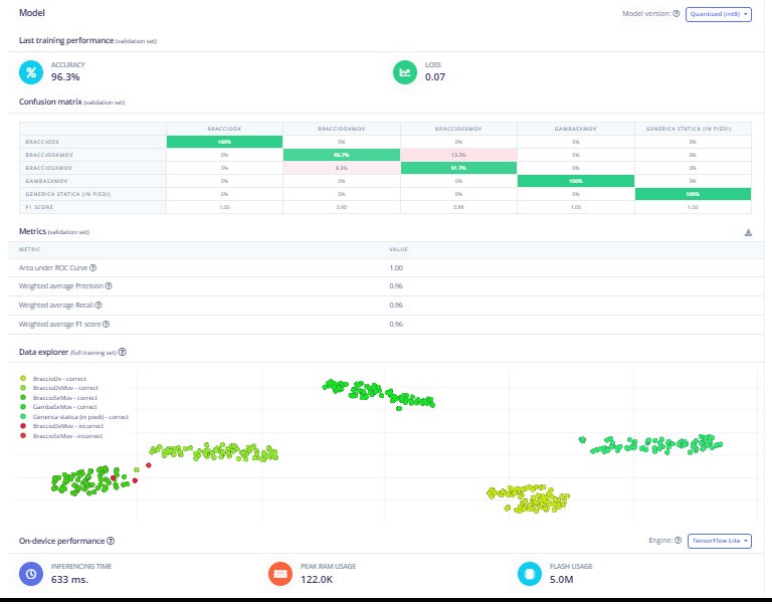
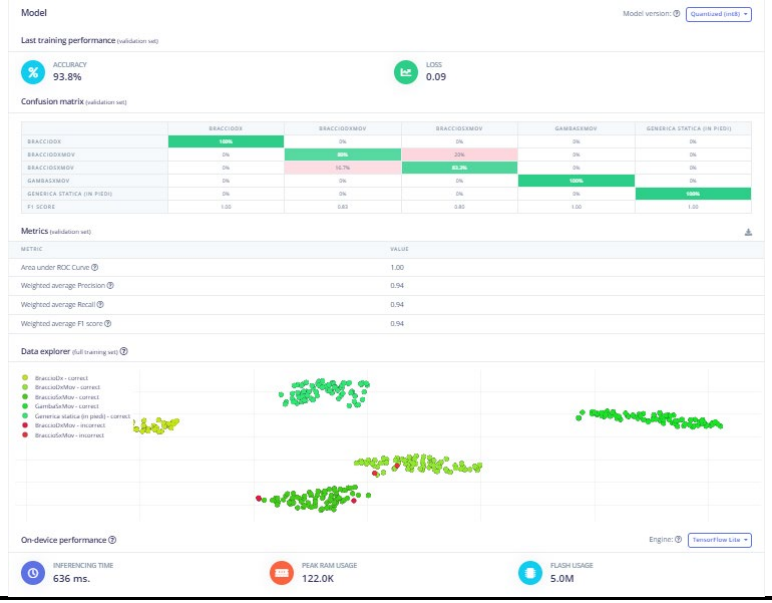
<table>
<tr><td colspan="3">Img 40x125, classifier, 15 epoch, val set 15%, 2 classes (BraccioDxMov vs BraccioSxMov), 1 antenna</td></tr>
<tr><td>Input layer (5,000 features)
2D conv / pool layer (16 filters, 3 kernel size, 1 layer)
2D conv / pool layer (32 filters, 3 kernel size, 1 layer)
Flatten layer
Dropout (rate 0.1)
Add an extra layer
Output layer (2 classes)</td><td> 
</td><td>72,50%</td></tr>
<tr><td colspan="3">Img 80x125, classifier, 15 epoch, val set 15% 2 classes (BraccioDxMov vs BraccioSxMov), 2 antennas</td></tr>
<tr><td>Input layer (10,000 features)
2D conv / pool layer (16 filters, 3 kernel size, 1 layer)
2D conv / pool layer (32 filters, 3 kernel size, 1 layer)
Flatten layer
Dropout (rate 0.1)
Add an extra layer
Output layer (2 classes)</td><td> 
</td><td>82,50%</td></tr>
</table>

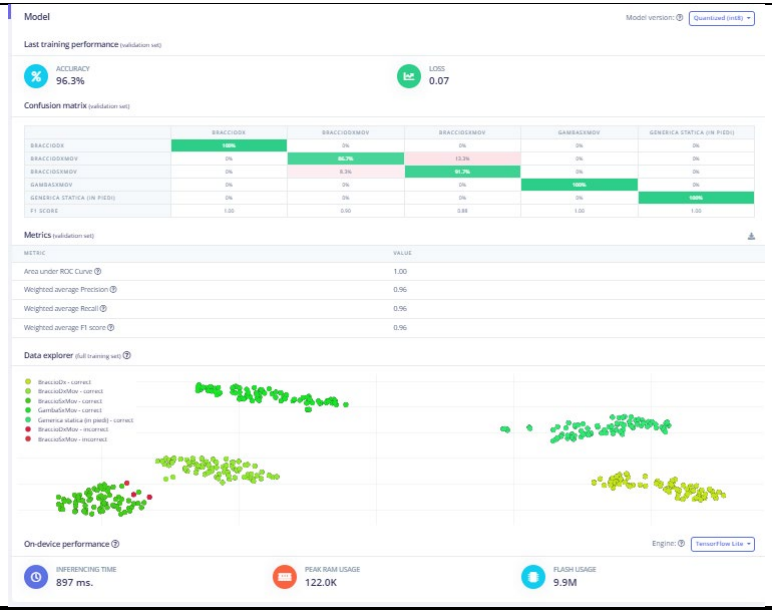
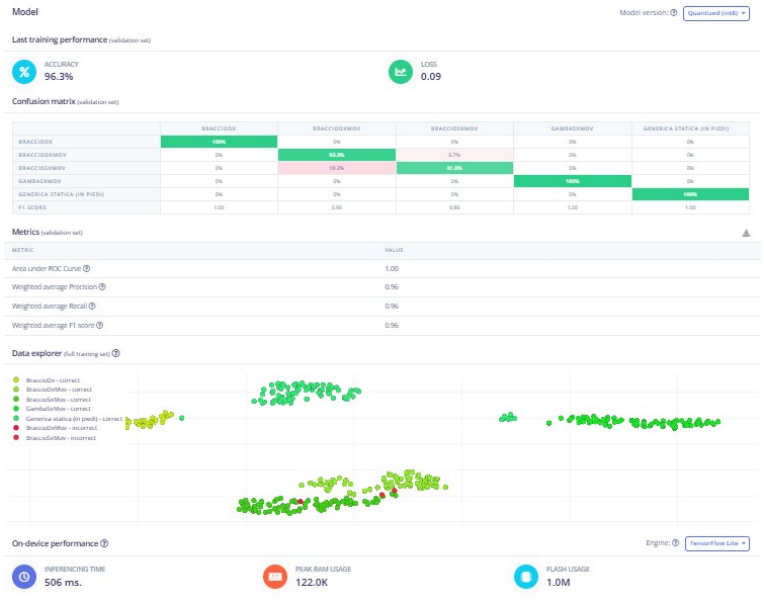
Img 140x125, classifier, 30 epoch, val set 20%		
Input layer (17,500 features) 2D conv / pool layer (16 filters, 3 kernel size, 1 layer) 2D conv / pool layer (32 filters, 3 kernel size, 1 layer) Flatten layer Dropout (rate 0.1) Add an extra layer Output layer (5 classes)		82.0%
Img 140x125, classifier, 20 epoch, val set 20%		
Input layer (17,500 features) 2D conv / pool layer (16 filters, 4 kernel size, 1 layer) 2D conv / pool layer (16 filters, 3 kernel size, 1 layer) Flatten layer Dropout (rate 0.2) Add an extra layer Output layer (5 classes)		77%

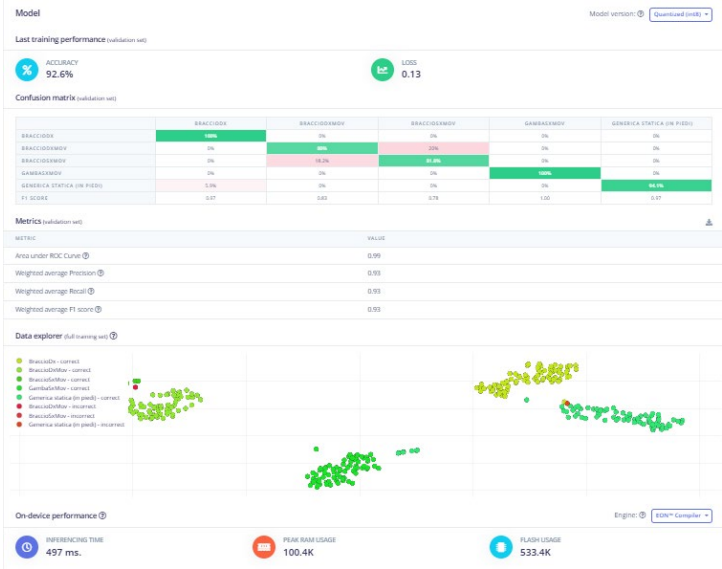
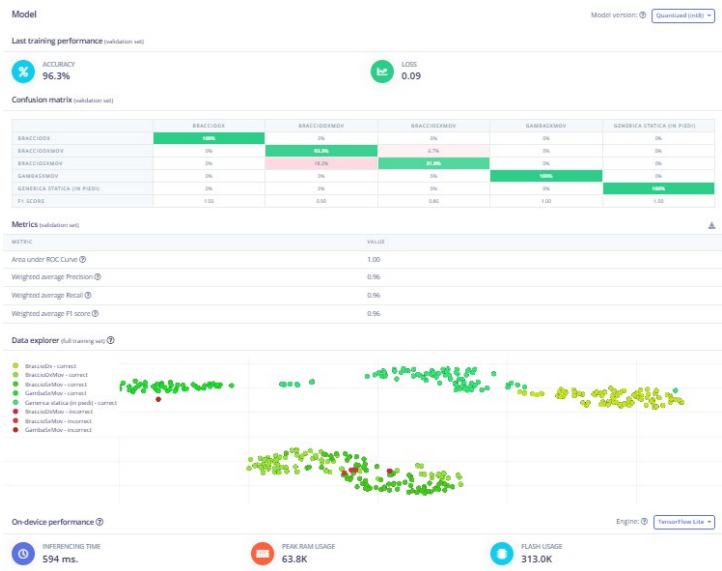
Img 140x125, classifier, 30 epoch, val set 15%		
Input layer (17,500 features) 2D conv / pool layer (10 filters, 2 kernel size, 1 layer) Flatten layer Dropout (rate 0.05) Add an extra layer Output layer (5 classes)		77.0%
Img 140x125, classifier, 10 epoch, val set 20%		
Input layer (17,500 features) 2D conv / pool layer (16 filters, 3 kernel size, 1 layer) Flatten layer Dropout (rate 0.2) Dense layer (1000 neurons) Dense layer (100 neurons) Dense layer (10 neurons) Add an extra layer Output layer (5 classes)		87.13%

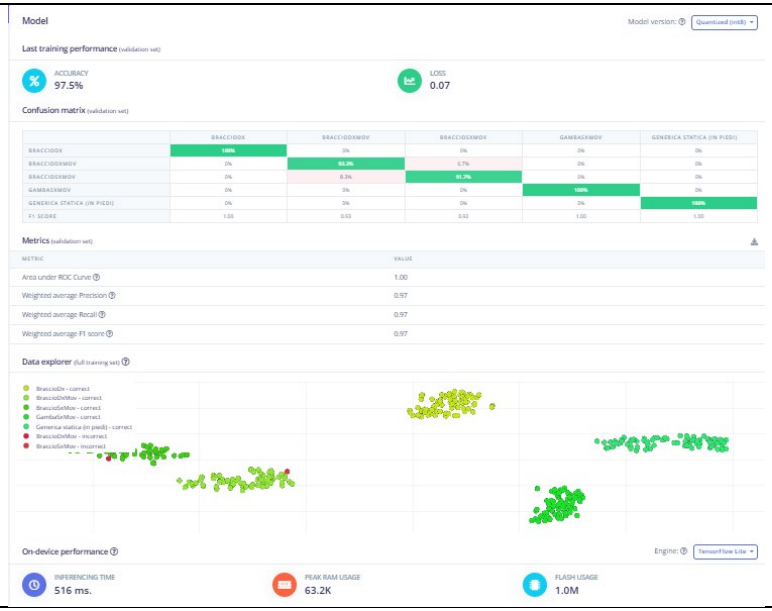
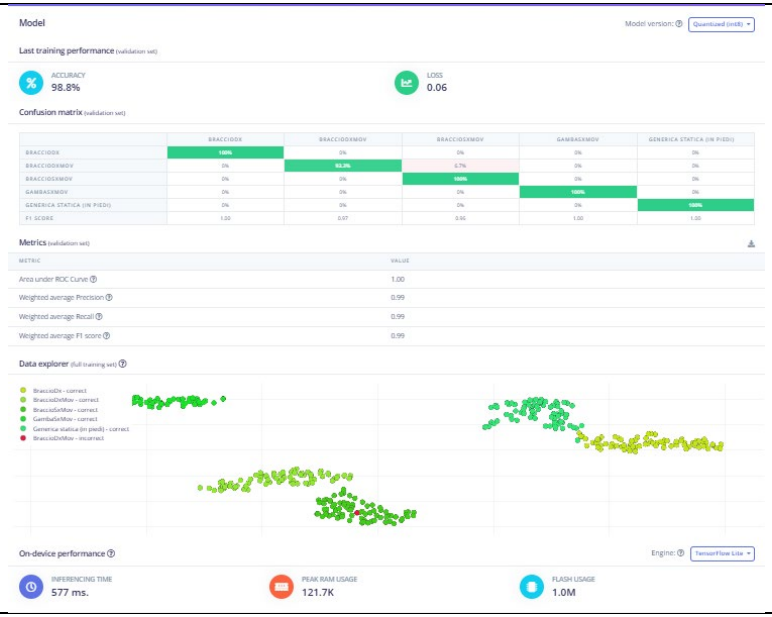
Img 140x125, classifier, 20 epoch, val set 15%		
Input layer (17,500 features) 2D conv / pool layer (8 filters, 3 kernel size, 1 layer) Flatten layer Dropout (rate 0.2) Dense layer (200 neurons) Dense layer (20 neurons) Add an extra layer Output layer (5 classes)		83,17%
Img 140x125, classifier, 10 epoch, val set 20%		
Input layer (11,200 features) 2D conv / pool layer (8 filters, 3 kernel size, 2 layers) Flatten layer Dropout (rate 0.25) Dense layer (300 neurons) Dense layer (50 neurons) Dense layer (10 neurons) Add an extra layer Output layer (5 classes)		50.50%

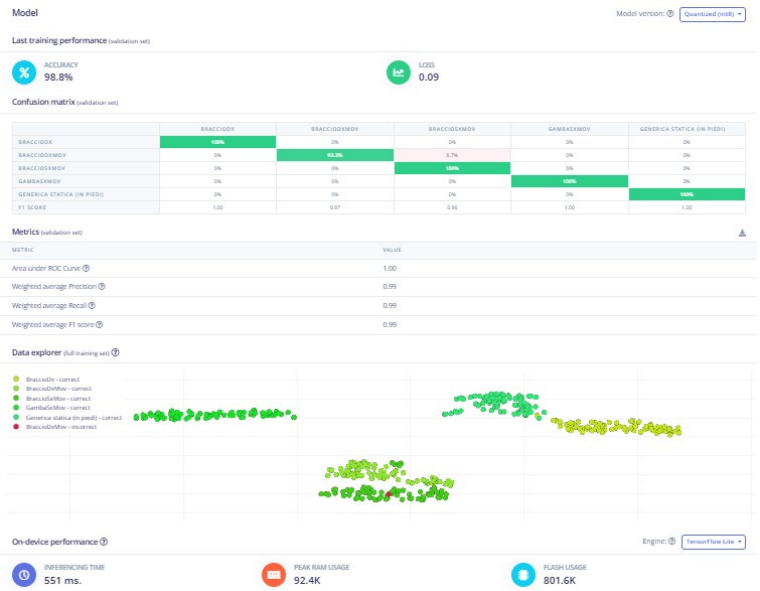
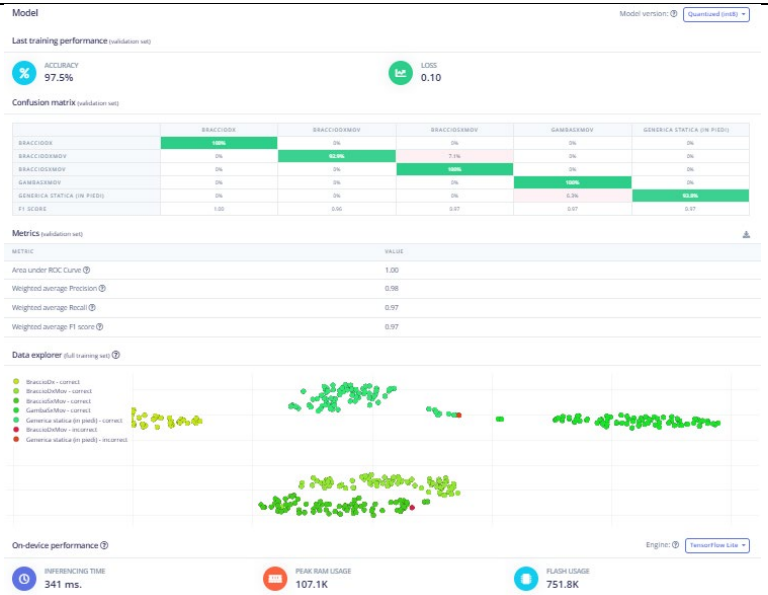
<table>
<tr><td colspan="3">Img 70x63 squashed (50%), classifier, 20 epoch, val set 20%</td></tr>
<tr>
<td>Input layer (4,410 features)
2D conv / pool layer (16 filters, 3 kernel size, 1 layer)
Flatten layer
Dropout (rate 0.2)
Dense layer (1000 neurons)
Dense layer (100 neurons)
Dense layer (10 neurons)
Add an extra layer
Output layer (5 classes)</td>
<td> 
</td>
<td>77.55%</td>
</tr>
<tr><td colspan="3">Img 70x63 squashed (80%), classifier, 15 epoch, val set 20%</td></tr>
<tr>
<td>Input layer (4,410 features)
2D conv / pool layer (8 filters, 3 kernel size, 1 layer)
Flatten layer
Dropout (rate 0.1)
Dense layer (300 neurons)
Dense layer (30 neurons)
Add an extra layer
Output layer (5 classes)</td>
<td> 
</td>
<td>72.45%</td>
</tr>
</table>

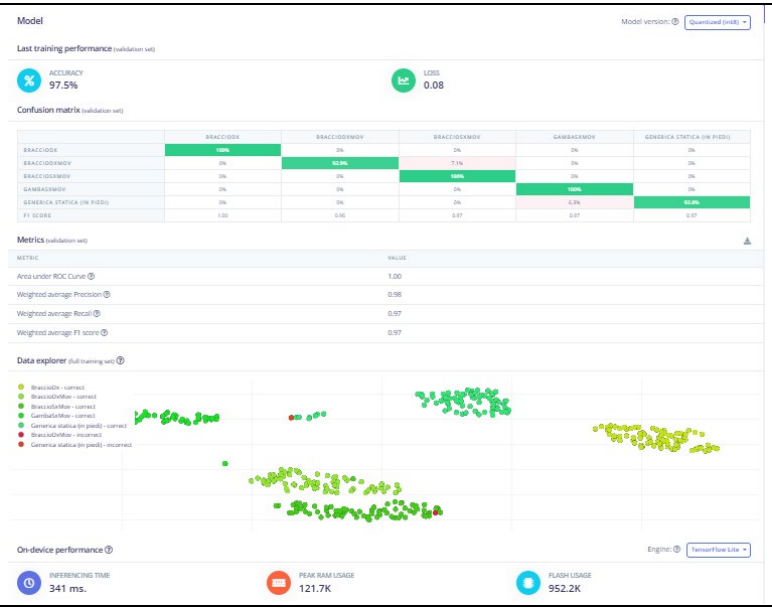
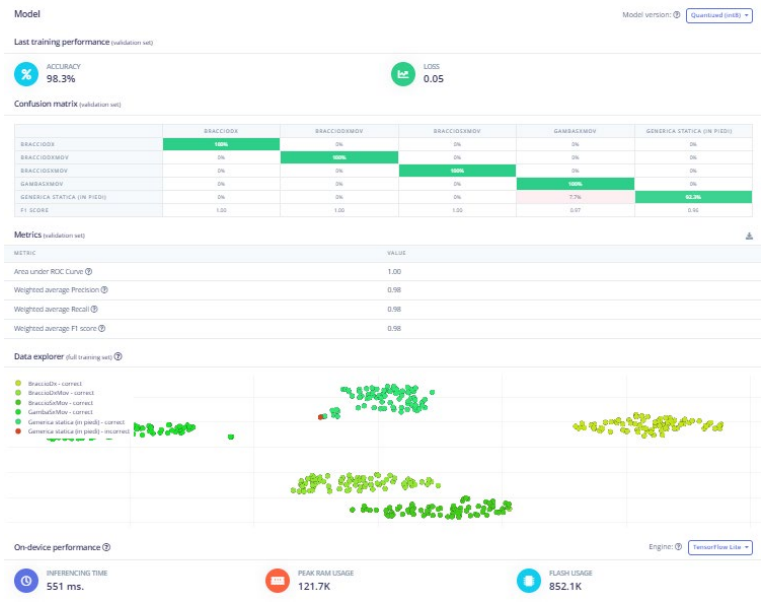
Img 80x125, classifier, 20 epoch, val set 20%		
Input layer (10,000 features) 2D conv / pool layer (8 filters, 3 kernel size, 1 layer) 2D conv / pool layer (16 filters, 3 kernel size, 1 layer) Flatten layer Dropout (rate 0.2) Dense layer (500 neurons) Dense layer (50 neurons) Add an extra layer Output layer (5 classes)		99.0%
Img 80x125, classifier, 20 epoch, val set 20%		
Input layer (10,000 features) 2D conv / pool layer (8 filters, 3 kernel size, 2 layers) Flatten layer Dropout (rate 0.1) Dense layer (1000 neurons) Dense layer (100 neurons) Add an extra layer Output layer (5 classes)		99.0%

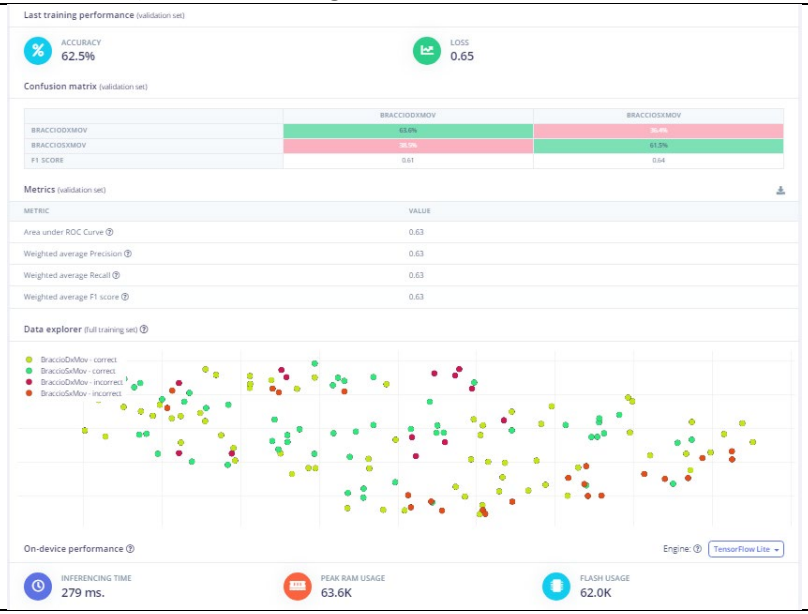
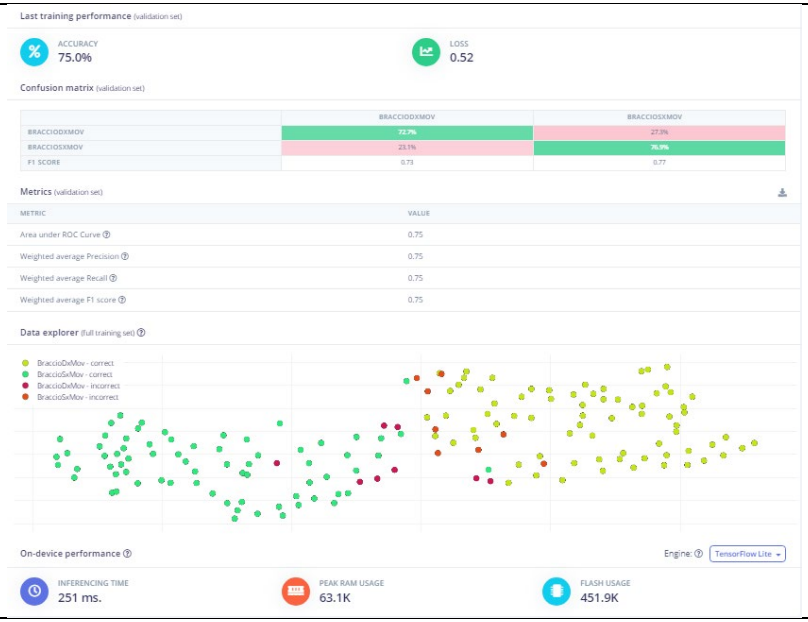
Img 80x125, classifier, 20 epoch, val set 20% Input layer (10,000 features) 2D conv / pool layer (8 filters, 3 kernel size, 1 layer) 2D conv / pool layer (16 filters, 3 kernel size, 1 layer) Flatten layer Dropout (rate 0.15) Dense layer (1000 neurons) Dense layer (100 neurons) Add an extra layer Output layer (5 classes)		99.0%
Img 80x125, classifier, 20 epoch, val set 20% Input layer (10,000 features) 2D conv / pool layer (8 filters, 3 kernel size, 1 layer) 2D conv / pool layer (16 filters, 3 kernel size, 1 layer) Flatten layer Dropout (rate 0.2) Dense layer (100 neurons) Dense layer (25 neurons) Add an extra layer Output layer (5 classes)		93.94%

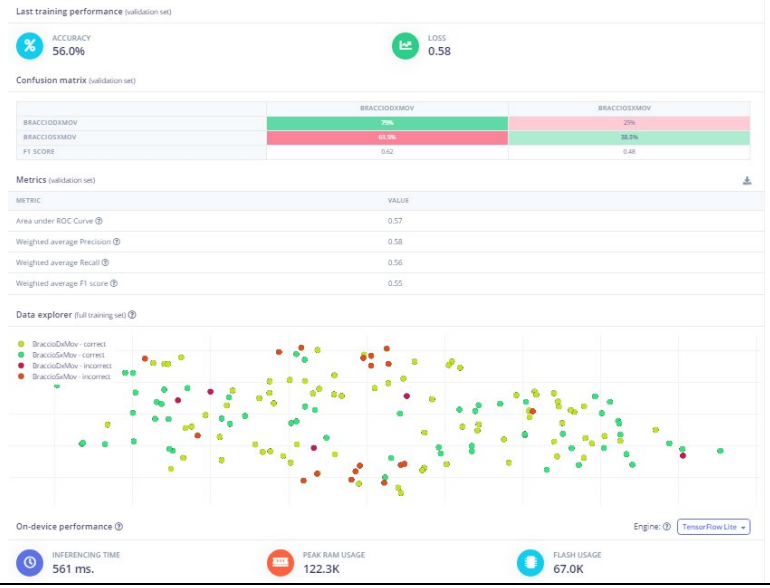
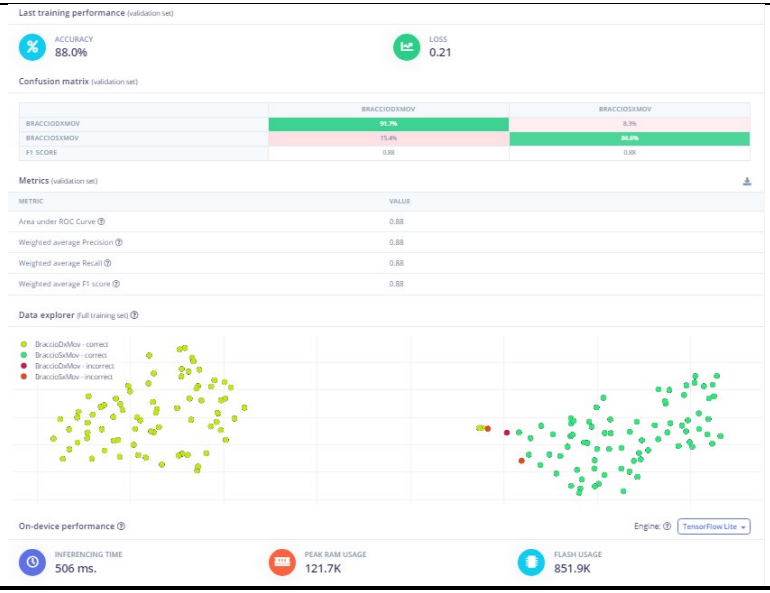
Img 80x125, classifier, 20 epoch, val set 20%		
Input layer (10,000 features) 2D conv / pool layer (8 filters, 3 kernel size, 2 layers) Flatten layer Dropout (rate 0.2) Dense layer (100 neurons) Dense layer (20 neurons) Add an extra layer Output layer (5 classes)		92.93%
Img 80x125, classifier, 20 epoch, val set 20%		
Input layer (10,000 features) 2D conv / pool layer (4 filters, 3 kernel size, 1 layer) 2D conv / pool layer (4 filters, 4 kernel size, 1 layer) 2D conv / pool layer (8 filters, 4 kernel size, 1 layer) Flatten layer Dropout (rate 0.15) Dense layer (200 neurons) Dense layer (50 neurons) Add an extra layer Output layer (5 classes)		93.94%

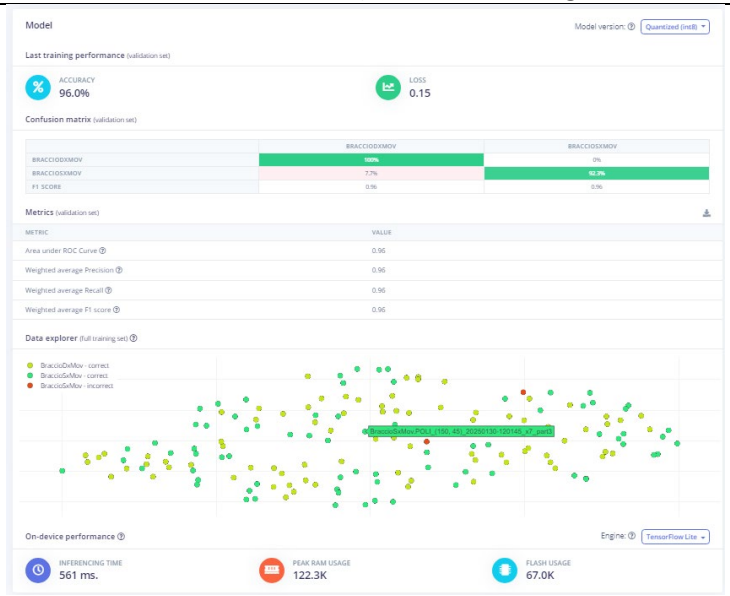
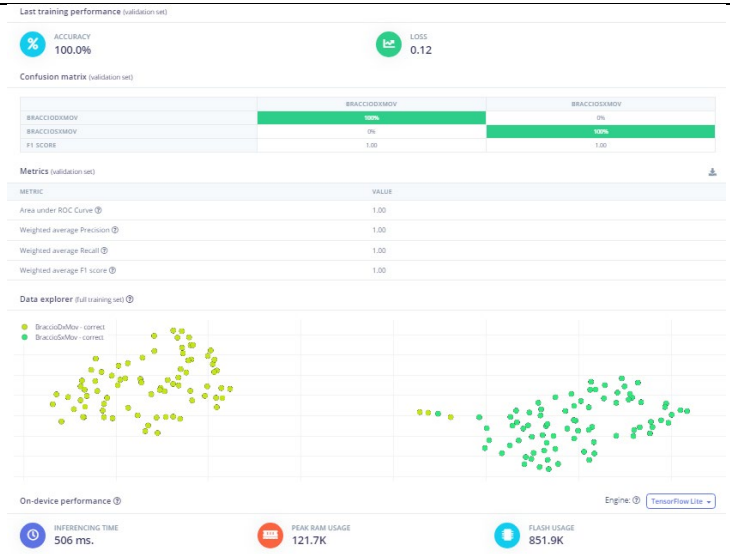
Img 80x125, classifier, 20 epoch, val set 20%		
Input layer (10,000 features) 2D conv / pool layer (4 filters, 3 kernel size, 1 layer) 2D conv / pool layer (8 filters, 3 kernel size, 1 layer) Flatten layer Dropout (rate 0.25) Dense layer (200 neurons) Dense layer (20 neurons) Add an extra layer Output layer (5 classes)		92.0%
Img 80x125, classifier, 20 epoch, val set 20%		
Input layer (10,000 features) 2D conv / pool layer (8 filters, 3 kernel size, 1 layer) 2D conv / pool layer (16 filters, 4 kernel size, 1 layer) Flatten layer Dropout (rate 0.15) Dense layer (100 neurons) Add an extra layer Output layer (5 classes)		92.0%

Img 80x125, classifier, 15 epoch, val set 20%		
Input layer (10,000 features) 2D conv / pool layer (6 filters, 3 kernel size, 1 layer) 2D conv / pool layer (12 filters, 3 kernel size, 1 layer) Flatten layer Dropout (rate 0.15) Dense layer (100 neurons) Add an extra layer Output layer (5 classes)		95.00%
Img 80x125, classifier, 20 epoch, val set 20%		
Input layer (10,000 features) 2D conv / pool layer (7 filters, 3 kernel size, 1 layer) 2D conv / pool layer (14 filters, 3 kernel size, 1 layer) Flatten layer Dropout (rate 0.15) Dense layer (80 neurons) Add an extra layer Output layer (5 classes)		97.06%

<table>
<tr><td colspan="3">Img 80x125, classifier, 20 epoch, val set 20%</td></tr>
<tr><td>Input layer (10,000 features)
2D conv / pool layer (8 filters, 3 kernel size, 1 layer)
2D conv / pool layer (16 filters, 3 kernel size, 1 layer)
Flatten layer
Dropout (rate 0.15)
Dense layer (90 neurons)
Add an extra layer
Output layer (5 classes)</td><td> 
</td><td>96.08%</td></tr>
<tr><td colspan="3">Img 80x125, classifier, 20 epoch, val set 15% <u>BEST configuration</u>: + 80 neurons model</td></tr>
<tr><td>Input layer (10,000 features)
2D conv / pool layer (8 filters, 3 kernel size, 1 layer)
2D conv / pool layer (16 filters, 3 kernel size, 1 layer)
Flatten layer
Dropout (rate 0.15)
Dense layer (80 neurons)
Add an extra layer
Output layer (5 classes)</td><td> 
</td><td>97.06%</td></tr>
</table>

<table>
<tr><td colspan="3">Img 40x125, classifier, 17 epoch, val set 15%, 2 classes (BraccioDxMov vs BraccioSxMov), 1 antenna, single dense model</td></tr>
<tr><td>Input layer (10,000 features)
2D conv / pool layer (8 filters, 3 kernel size, 1 layer)
2D conv / pool layer (16 filters, 3 kernel size, 1 layer)
2D conv / pool layer (32 filters, 3 kernel size, 1 layer)
Flatten layer
Dropout (rate 0.1)
Add an extra layer
Output layer (5 classes)</td><td> 
</td><td>70,00 %</td></tr>
<tr><td colspan="3">Img 40x125, classifier, 20 epoch, val set 15%, 2 classes (BraccioDxMov vs BraccioSxMov), 1 antenna, + 80 neurons model</td></tr>
<tr><td>Input layer (10,000 features)
2D conv / pool layer (8 filters, 3 kernel size, 1 layer)
2D conv / pool layer (16 filters, 3 kernel size, 1 layer)
Flatten layer
Dropout (rate 0.15)
Dense layer (80 neurons)
Add an extra layer
Output layer (5 classes)</td><td> 
</td><td>70,00 %</td></tr>
</table>

<table>
<tr><td colspan="3">Img 80x125, classifier, 17 epoch, val set 15%, 2 classes (BraccioDxMov vs BraccioSxMov), 2 antennas, Single dense model</td></tr>
<tr><td>Input layer (10,000 features)
2D conv / pool layer (8 filters, 3 kernel size, 1 layer)
2D conv / pool layer (16 filters, 3 kernel size, 1 layer)
2D conv / pool layer (32 filters, 3 kernel size, 1 layer)
Flatten layer
Dropout (rate 0.1)
Add an extra layer
Output layer (5 classes)</td><td> 
</td><td>71,79% (th 0,6)
87,18% (th 0,5)</td></tr>
<tr><td colspan="3">Img 80x125, classifier, 20 epoch, val set 15%, 2 classes (BraccioDxMov vs BraccioSxMov), 2 antennas, +80 neurons model</td></tr>
<tr><td>Input layer (10,000 features)
2D conv / pool layer (8 filters, 3 kernel size, 1 layer)
2D conv / pool layer (16 filters, 3 kernel size, 1 layer)
Flatten layer
Dropout (rate 0.15)
Dense layer (80 neurons)
Add an extra layer
Output layer (5 classes)</td><td> 
</td><td>87,18 % (th 0,6)
89,74% (th 0,5)</td></tr>
</table>

<table>
<tr><td colspan="3">Img 80x125, classifier, 20 epoch, val set 15%, learning rate 0,001%, 2 classes (BraccioDxMov vs BraccioSxMov), 2 antennas, Single dense model well trained</td></tr>
<tr><td>Input layer (10,000 features)
2D conv / pool layer (8 filters, 3 kernel size, 1 layer)
2D conv / pool layer (16 filters, 3 kernel size, 1 layer)
2D conv / pool layer (32 filters, 3 kernel size, 1 layer)
Flatten layer
Dropout (rate 0.1)
Add an extra layer
Output layer (5 classes)</td><td> 
</td><td>94,87 %</td></tr>
<tr><td colspan="3">Img 80x125, classifier, 20 epoch, val set 15%, learning rate 0,001%, 2 classes (BraccioDxMov vs BraccioSxMov), 2 antennas, +80 neurons model well trained</td></tr>
<tr><td>Input layer (10,000 features)
2D conv / pool layer (8 filters, 3 kernel size, 1 layer)
2D conv / pool layer (16 filters, 3 kernel size, 1 layer)
Flatten layer
Dropout (rate 0.15)
Dense layer (80 neurons)
Add an extra layer
Output layer (5 classes)</td><td> 
</td><td>92,31 %</td></tr>
</table>