POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Hand Gesture Recognition with TinyML

Hardware Architectures for Embedded and Edge AI
Computer Science and Engineering

Authors: **Alberto Pirillo - Roberto Molgora**

Professor: **Prof. Manuel Roveri**

Teaching Assistant: **Ing. Massimo Pavan**

Academic Year: **2022-2023**

Contents

Introduction

This project aims to develop a Tiny Machine Learning (TinyML) application, specifically focusing on deploying a Convolutional Neural Network (CNN) on the Arduino Nano 33 BLE, which is a tiny resource-constrained device. The goal is to employ the developed model for real-time inference on data gathered by the board's sensors in real-world scenarios.

This project centers on the creation of a Hand Gesture Recognition (HGR) system, which is a Machine Learning task and a sub-field of Computer Vision. The aim of an HGR system is to mathematically interpret gestures performed by an individual using their hand, in order to trigger specific commands or any other action in general.

In our project, we develop a CNN following the Transfer Learning paradigm and using MobileNetV1 as a base model. Once deployed to the device, we can utilize the built-in low-power camera of the Arduino board we are using to collect an image of a gesture and classify it among a variety of gestures in real-time.

As for any other TinyML application, the main challenge of the project will be optimizing the model such that it fits into the main memory of the device, which is constrained to a mere 256 KB. This limitation concerns not only the model itself but also the input data, the storage required to store the intermediate feature maps of the network, and the libraries required for the device's functionality.

Abbreviations

- **NN:** (Artificial) Neural Networks are weighted directed graphs used for solving Artificial Intelligence (AI) problems, they are widely used in the Machine Learning and Deep Learning fields.
- **CNN:** Convolutional Neural Networks are Neural Networks specifically designed for image recognition and processing applications.
- **Arduino:** is an electronic board, equipped with a microcontroller, some serial ports, and a small amount of memory. External sensors can be connected: in particular, in this project, an OV7675 camera has been used.
- **TF:** TensorFlow [1] is an open-source software library for Machine Learning and Artificial Intelligence that focuses on training and inference of deep neural networks. It is developed by Google.
- **EI:** Edge Impulse [2] is a platform designed to facilitate the development of Machine Learning models for embedded and edge devices. It provides a range of tools and resources to create, train, and deploy Machine Learning models.

1 | Dataset

The initial phase involved the selection of an appropriate dataset for the project. From the beginning, a crucial consideration was that we would need to develop a model intended for deployment on a severely resource-constrained device with a low-resolution camera.

The primary objective during this stage was to identify a dataset containing images resembling those that the device's camera could capture. Special care needs to be taken in order to prevent the development of a model that excels on the original dataset while performing poorly on real-world data.

Initially, we evaluated some datasets in the existing literature known to be a good fit for a hand-gesture recognition problem.

1.1. The HANDS dataset

The HANDS dataset [3] has been created for human-robot interaction research. It contains 12 static single-hand gestures performed with both the right hand and the left hand, along with an additional three static two-handed gestures, for a total of 29 unique classes.

The data has been collected using a Kinect v2 camera calibrated to align RGB data to Depth data spatially. Each frame has a resolution of 960 x 540 pixels, which is not too different from the one of our OV7675 Camera, which is 640 x 480 pixels.

However, we can see that the images encompass more than just the subject's hand, encompassing the entire individual. We fear that having to both recognize hand positions and classify the gestures would be too much for a model tiny enough to fit on the device. Therefore, we decided to move on and consider a different dataset.

1.2. The HaGRID dataset

The HaGRID dataset [4] (HAnd Gesture Recognition Image Dataset) is a well-known dataset made for building hand-gesture recognition (HGR) systems.

The dataset contains 552,992 Full HD (1920 x 1080) RGB images divided into 18 classes of gestures. The dataset contains 34,730 unique individuals and at least this number of unique scenes. The subjects are people from 18 to 65 years old. The dataset was collected mainly indoors with considerable variation in lighting, including artificial and natural light.

However, despite its diversity, the HaGRID dataset shares the same limitation (in our specific setting) with the HANDS dataset: images contain the entirety of the individual rather than exclusively focusing on the hand. Additionally, in this case, the resolution of the images is also much higher w.r.t. the one of our camera, therefore we decided to look for another dataset.

1.3. The ASL dataset

The ASL dataset [5] comprises a collection of hand sign images from the American Sign Language, separated into 29 folders that represent the various classes.

The dataset contains 87,000 images which are 256 x 256 pixels. There are 29 classes, of which 26 are for the letters A to Z and 3 more classes for *space*, *delete* and *nothing*. These 3 classes can be very useful in real-world applications.

Although not originally intended for hand-gesture recognition (unlike the previous datasets), the ASL dataset suits can still fit our purpose: we can easily consider the different letters of the sign language as if they were distinct gestures.

Furthermore, the resolution of the images of this dataset can be easily handled by the camera of our device and, more importantly, the images focus solely on hands, eliminating concerns about encompassing entire individuals.

Consequently, given that the ASL dataset seems very suited for our specific setting, we decided to use it to continue our work.

2 | Exploratory Data Analysis

In this chapter, we investigate the characteristics of the information contained in the ASL dataset. The analysis was conducted in a Jupyter Notebook, relying on the TensorFlow and NumPy libraries for data manipulation and on the matplotlib, seaborn, and Plotly libraries for data visualization.

All the details about this analysis are available in the notebook *ASL_EDA.ipynb* in the *notebooks* folder of the GitHub repository [6] of the project.

The goals of this EDA are the following:

- Check for inconsistencies in the data
- Check whether the class distribution is balanced
- Visualize a reduced-dimensional representation of the data
- Identify a subset of classes for model training

2.1. Pre-processing

In the following picture, a selection of samples from the ASL dataset is displayed.

Figure 2.1: Visualization of 25 samples from the ASL dataset

Interestingly, we notice that every image presents a blue line around the borders. Worried that this may potentially introduce some bias into the model, we decided to remove this border using TensorFlow. In the following picture, we can see some samples from the pre-processed data.

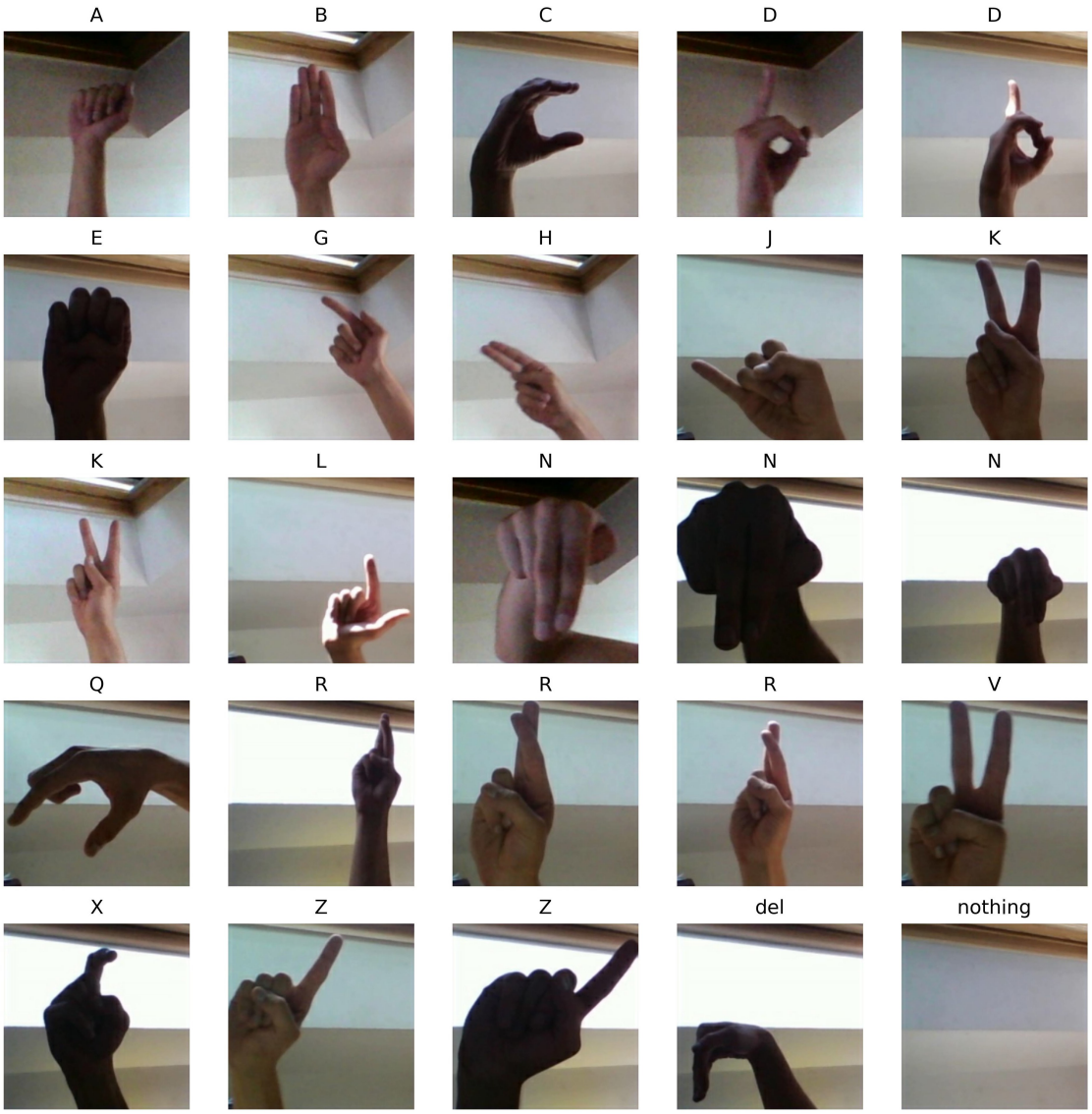


Figure 2.2: Visualization of 25 samples from the ASL dataset with the blue border removed

Subsequently, we have to check that the distribution of the classes is balanced, given that having balanced classes is a good practice when working with a classification problem. Fortunately, this dataset presents completely balanced classes, as we can see from the following chart. Hence, no further intervention is necessary.

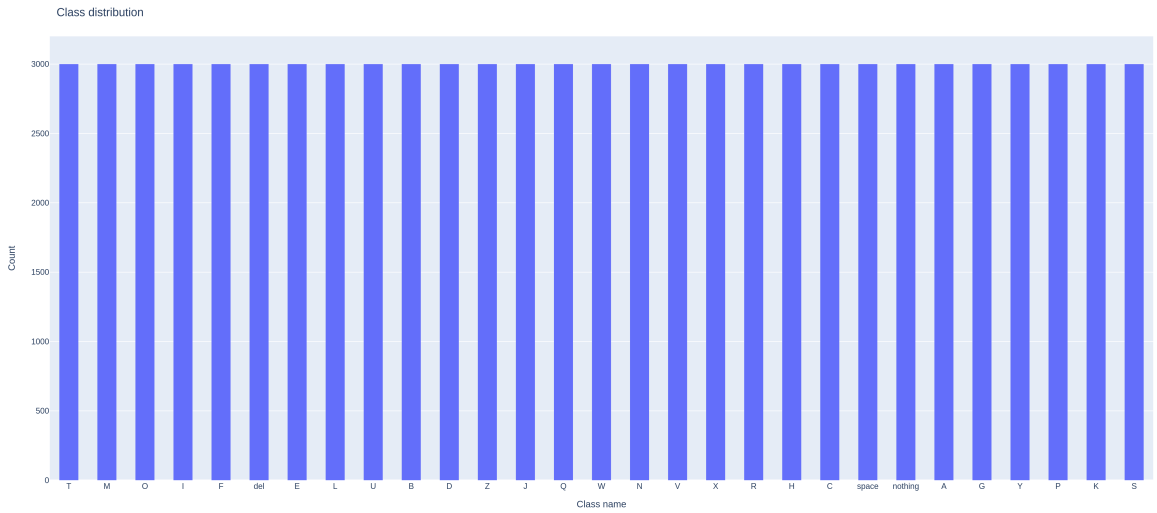


Figure 2.3: Distribution of the classes of the ASL dataset

2.2. Selecting a subset of the classes

The ASL dataset comprises 29 classes. However, there is no need for us to utilize all the classes to develop our hand-gesture recognition system. We do not need to include all 29 distinct gestures, and more importantly, we trust that having fewer classes will benefit the performance of the tiny model we are going to develop. Therefore, our objective is to select a subset of the classes to be kept and used to develop the HGR system.

We want to select a subset of the classes such that the classes inside can be easily distinguished not by a human observer, but from the perspective of a Machine Learning model.

To determine the optimal subset, we employed three increasingly complex dimensionality reduction techniques:

- Principal Component Analysis (PCA)
- t-distributed Stochastic Neighbor Embedding (t-SNE)
- Feature extraction with a pre-trained CNN

Before proceeding, we resized the images to dimensions of 96 x 96 pixels using TensorFlow, given that it is a very common input size for tiny models, and we want to be consistent with the shape of the images that the model will receive as input.

The first two techniques (PCA and t-SNE) require working with grayscale images. We performed the conversion using TensorFlow. We initiated the process with PCA, which is

a linear dimensionality reduction technique. In the following image, we can see a scatter plot of the two first principal components of each sample after PCA has been applied.

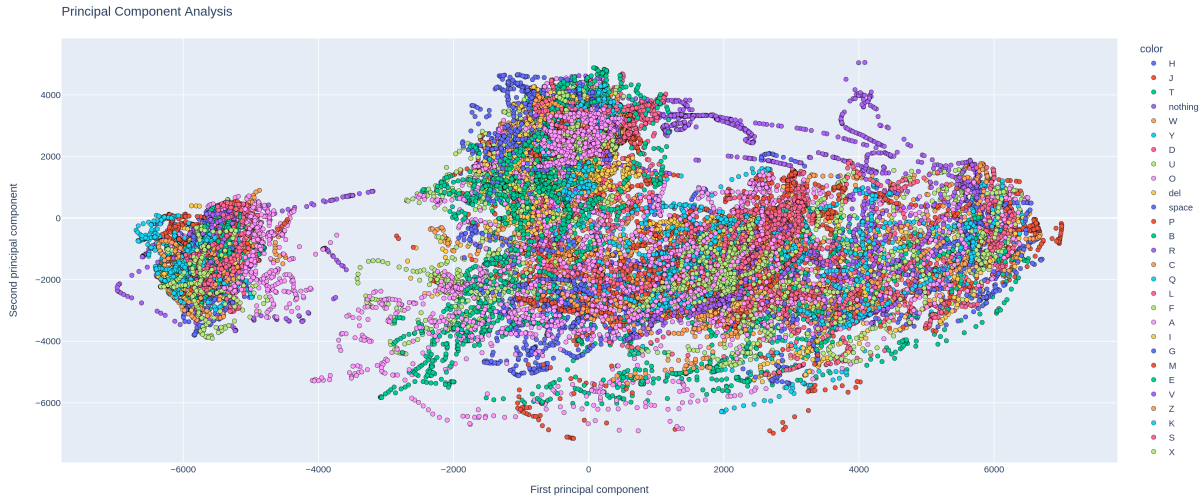


Figure 2.4: Scatter plot of the results of the PCA algorithm on the ASL dataset

The results are far from being useful for our objective. In fact, we did not expect PCA to perform well on highly-dimensional data and unstructured data like images. Therefore, we moved to a different technique, t-SNE, which is known to perform better than PCA because it is non-linear and therefore works better for unstructured real-world data. In the following image, we can see a scatterplot of the first two dimensions after t-SNE has been applied.

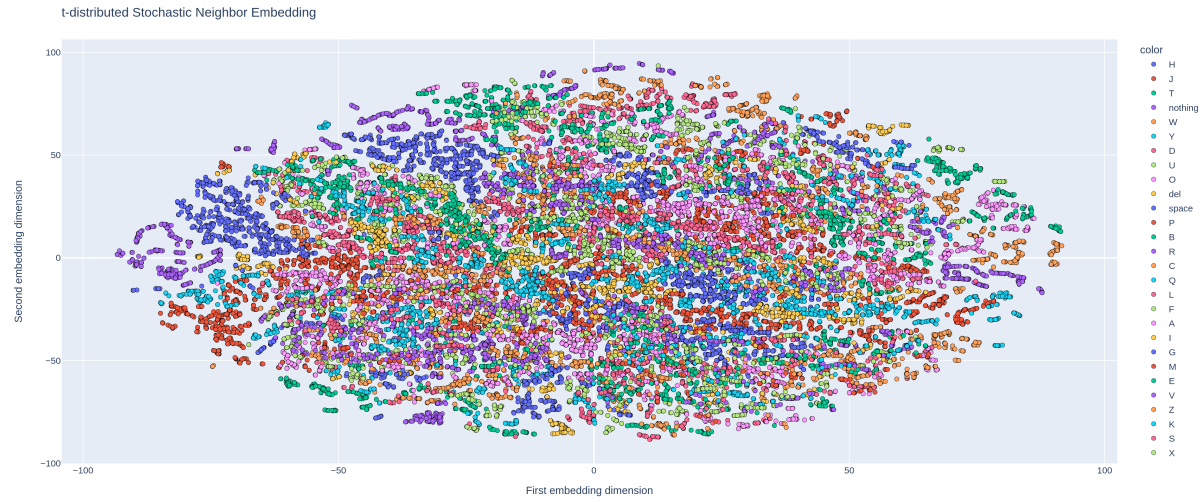


Figure 2.5: Scatter plot of the results of the t-SNE algorithm on the ASL dataset

It is worth noting that the t-SNE algorithm is computationally intensive and generally not recommended for data exceeding 50 dimensions. However, we were able to apply it regardless of our setting thanks to the RAPIDS cuML library [7] that allows us to perform the computation of the algorithm directly on the GPU, accelerating to a few seconds what would otherwise have taken multiple hours on the CPU.

Still, the results are of no interest to us since the scatter plot does not enable us to discern the class structure. Thus, in our final attempt, we decided to perform feature extraction using a pre-trained CNN. This approach offers the advantage of operating directly on RGB images (thereby processing more information) and yielding a more compact data representation. Subsequently, t-SNE can be applied to this representation for further dimensionality reduction and to make creating a visualization possible.

The model we are using as a feature extractor is EfficientNetV2B1, pre-trained on ImageNet. It is one of the models with the highest top-5 accuracy available on Keras and that does not take too long to process the images. We removed the classification head of the model and kept only the global feature extraction part. For each image, the model will output a feature vector with 1280 dimensions. In the following image, we can see a scatterplot of the first two dimensions after t-SNE has been applied.

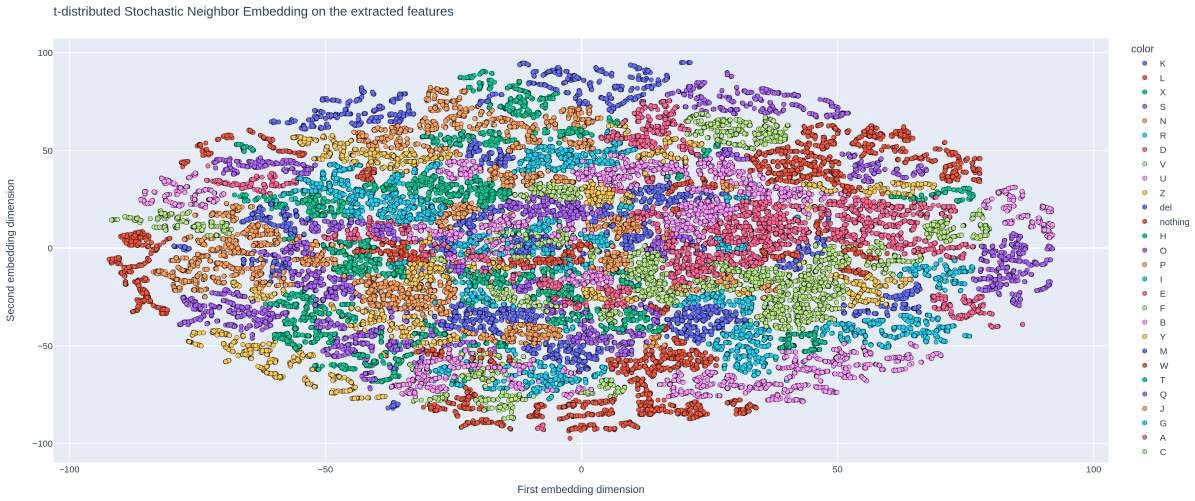


Figure 2.6: Scatter plot of the results of the t-SNE algorithm on the features extracted from the ASL dataset

This time, the data exhibits a more discernible structure. Nonetheless, identifying the most distinguishable and classifiable group of classes remains challenging. Therefore, instead of relying on dimensionality reduction techniques, we decided to first train a model to classify all 29 classes, check its performance on each class, and then keep only

the classes with the highest performance. A more detailed description of this process is covered in the following chapter.

3 | All-gestures Model

In this chapter, we discuss how the neural network to be deployed on the Arduino device was trained. We refer to this model as *all-gestures* model since it is meant to address a multi-class classification problem on all 29 classes of gestures.

Every part of this analysis was performed using TensorFlow. The source code can be found in the notebook *all-gestures model.ipynb* of the GitHub repository.

3.1. Model architecture

Following the Transfer Learning paradigm, which consists of taking features learned on one problem, and leveraging them on a new problem through a pre-trained model, the selection of the architecture falls back into choosing an appropriate pre-trained model, referred to as the *base* model.

The severe memory constraints of the target device greatly reduce the number of Convolutional Neural Network (CNN) architectures that we can choose the *base* model from. In particular, considering the device's mere 256 KB RAM, which also has to be shared with input storage, intermediate feature maps, libraries, and firmware, it is reasonable to say that the final model must not be heavier than approximately 150 KB.

In order to choose the architecture, we took inspiration from the pre-trained models available on Edge Impulse. Some of them are displayed in the following image.

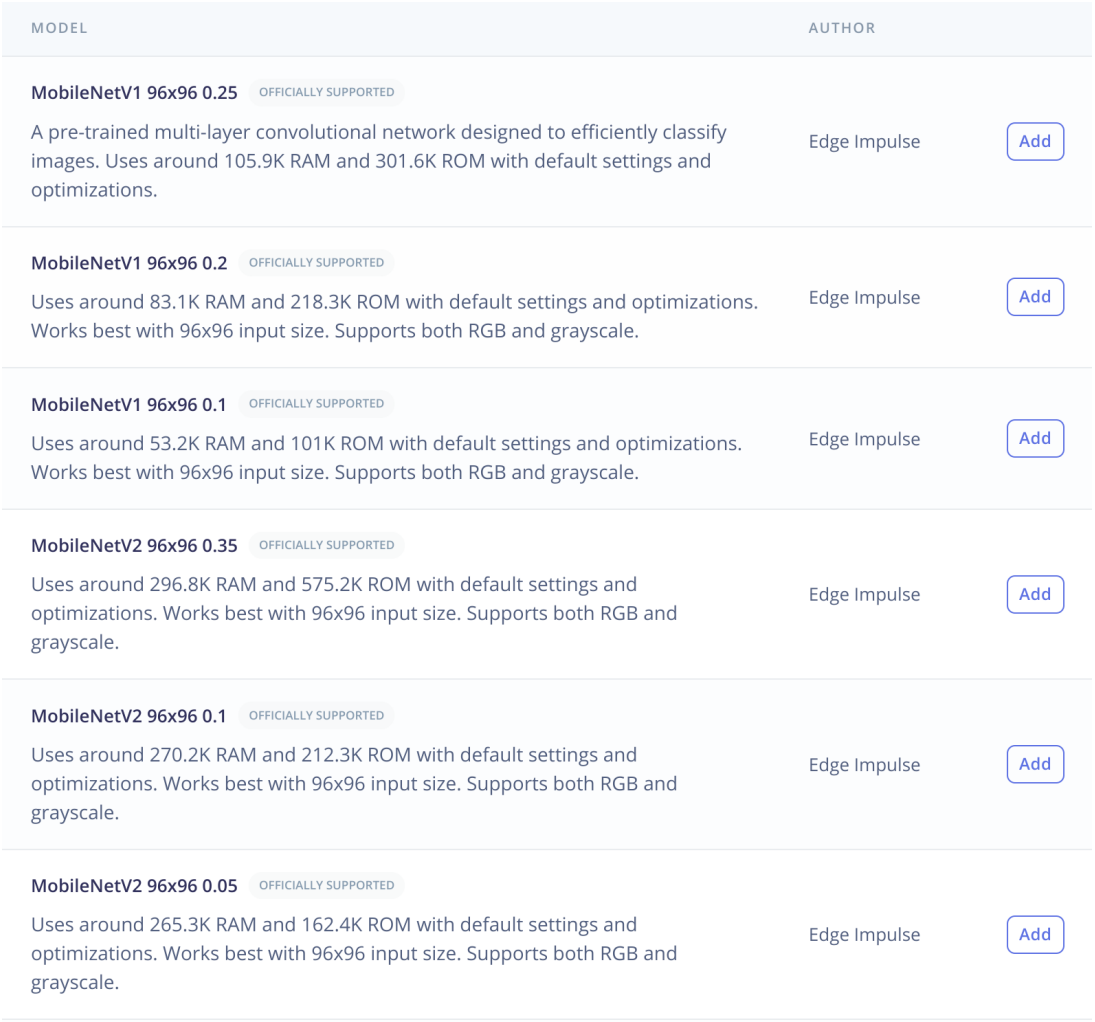


Figure 3.1: Some of the available pre-trained models from Edge Impulse.

According to Edge Impulse's estimates, the only architecture that could fit on the 256 KB of RAM of our device is MobileNetV1. Consequently, we decided to proceed with the variant of this architecture with the largest width multiplier α that could fit in our device, which is $\alpha = 0.25$.

Replicating Edge Impulse's MobileNetV1 96x96 0.25 architecture locally was preferred. Local training offers the advantages of faster training times (using GPUs) and extended training duration compared to Edge Impulse's 20-minute free plan limit, which was required in our setting since we are working with a dataset of a very large size.

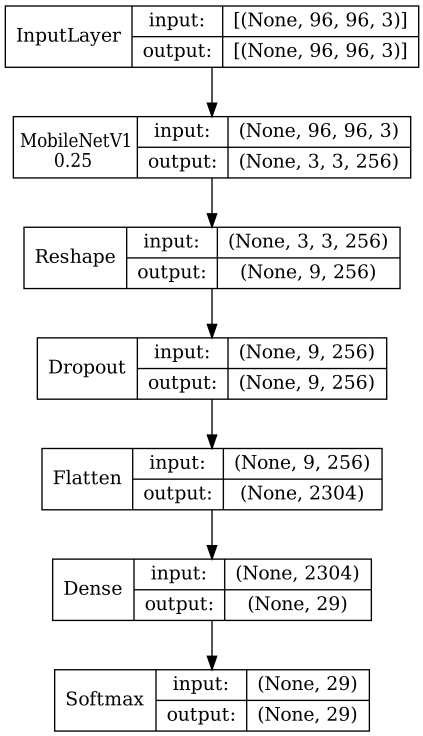


Figure 3.2: The architecture of the *all-gestures* model, with a focus on the layer type and on the input and output shapes.

3.2. Model training

With Transfer Learning, model training comprises two phases: the *fit* phase, where the base model is frozen, and only the dense classifier is trained; and the *fine-tuning* phase, where both the base model and classifier are trained.

In both steps, we are using the Adam optimizer and the Sparse Categorical Cross Entropy loss function in order to perform the gradient descent and the back-propagation.

For the *fit* phase, we are using an initial learning rate of 0.001 and training for 20 epochs. For the *fine-tuning* phase, we are using a much smaller learning rate of 0.00001 in order to avoid the phenomenon called catastrophic forgetting on the weights of the base model, and training for 20 epochs.

The final results of the training, i.e. the results of the *fine-tuning* phase, are displayed in the following charts.

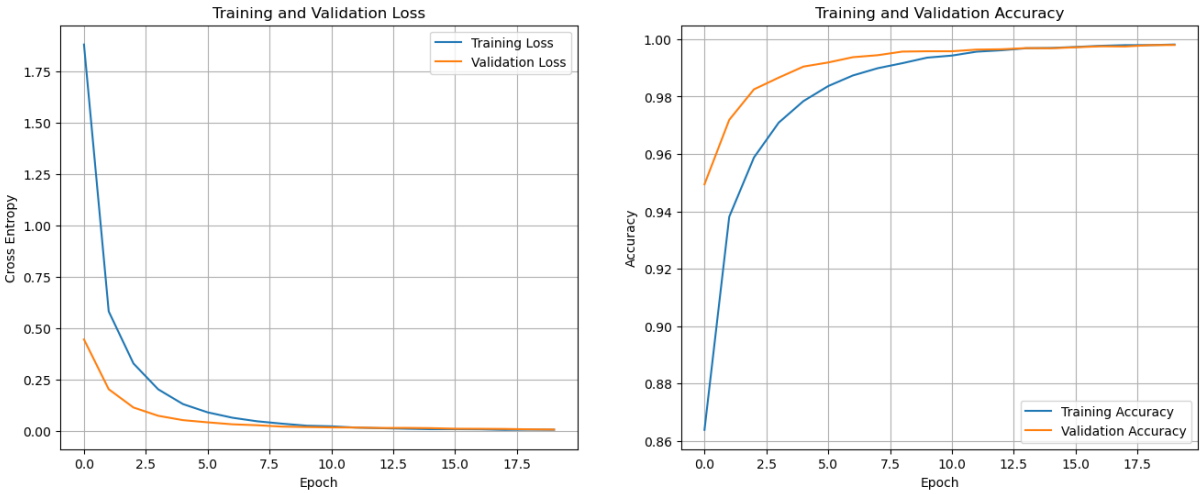


Figure 3.3: Evolution of the loss function and of the accuracy of the *all-gestures* model on both the training and the validation sets.

3.3. Model evaluation

In this section we take a closer look at the performance of the trained model, focusing on the validation set only. From the previous figure, we can see that the model achieved an extremely high performance since it was able to reach an accuracy very close to 100% without overfitting.

Satisfied with this result, our next step is to take a look at which are the classes in which the model performs better, as anticipated in the section 2.2. In order to do so, we can evaluate the model on the full validation set and generate a confusion matrix, which will highlight the samples for which the model ends up with an incorrect prediction. Then, from the confusion matrix we can compute the number of errors for each class, and finally normalize it on the number of samples for each class.

Consequently, we can easily visualize the results by using a bar plot.

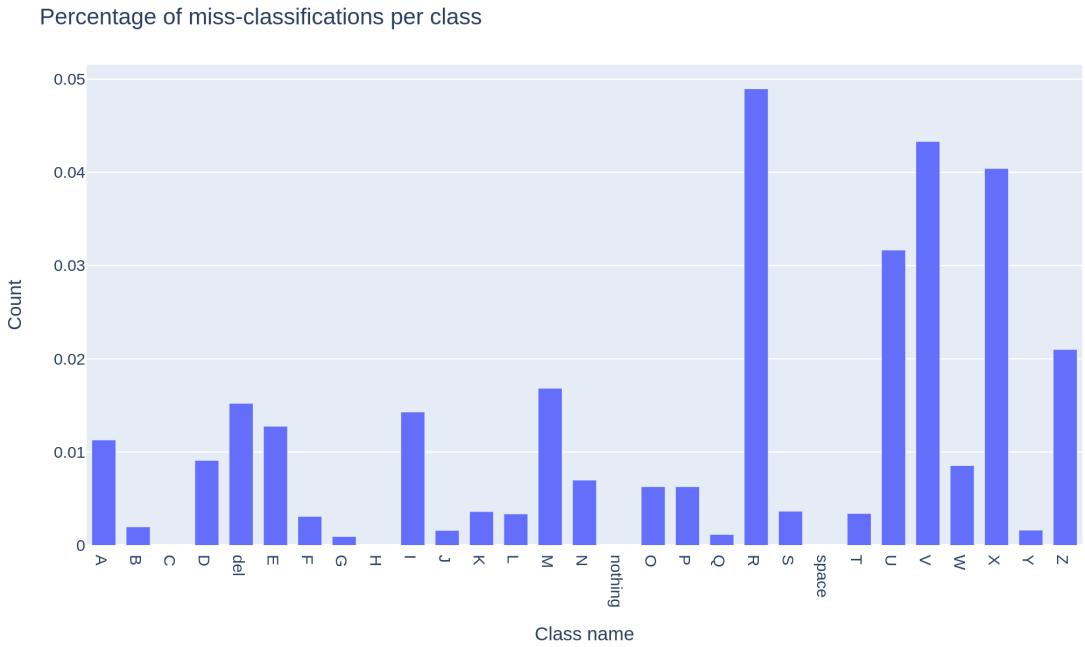


Figure 3.4: Percentage of miss-classification per class of the *all-gestures* model.

From the chart above, we can see that the model performs very well with the classes *C*, *G*, *H*, *Q* and *space*. We will also include the *nothing* class as it can be very useful in real-world applications to detect the absence of gestures or hands.

4 | 5-gestures Model

Now that we have selected a good subset of the classes, we can keep these only and train a very similar model to the one in the previous chapter. This is the model that will be deployed on the device and is referred to as *5-gestures* model.

Every part of this analysis was performed using TensorFlow. The source code can be found in the notebook *5-gestures model.ipynb* of the GitHub repository.

4.1. Model architecture

The architecture of the *5-gestures* model mirrors the *all-gestures* model, with adaptations in the dense classifier to accommodate the 6 selected classes.

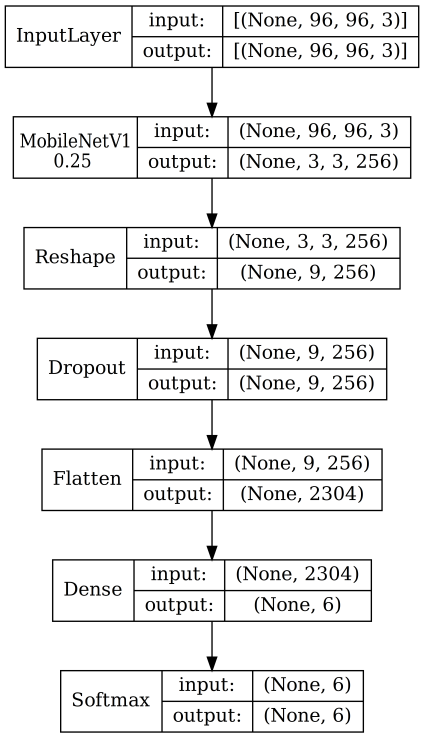


Figure 4.1: The architecture of the *5-gestures* model, with a focus on the layer type and on the input and output shapes.

4.2. Model training

The training is also very similar: it is divided in the same way into two phases, the *fit* phase and the *fine-tuning* phase.

In both steps, we are using the Adam optimizer and the Sparse Categorical Cross Entropy loss function in order to perform the gradient descent and the back-propagation.

For the *fit* phase, we are using an initial learning rate of 0.001 and training for 20 epochs. For the *fine-tuning* phase, we are using a much smaller learning rate of 0.00001 in order to avoid the phenomenon called catastrophic forgetting on the weights of the base model, and training for 20 epochs.

The final results of the training, i.e. the results of the *fine-tuning* phase, are displayed in the following charts.

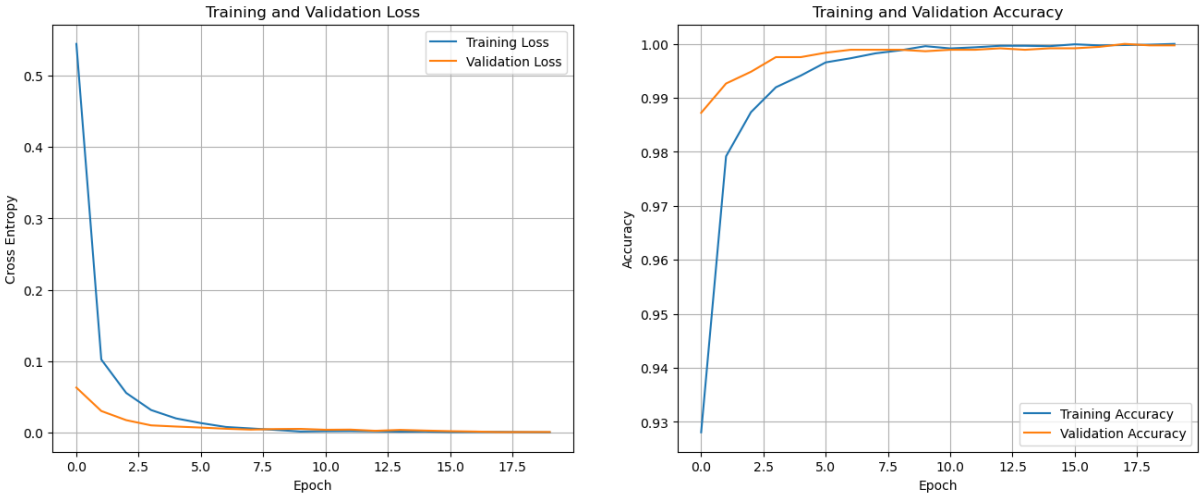


Figure 4.2: Evolution of the loss function and of the accuracy of the *5-gestures* model on both the training and the validation sets.

4.3. Model evaluation

From Figure 4.2 we can see that the model reached the highest accuracy possible, as expected since it was already achieving an excellent accuracy in the original 29-classes problem which was more complex.

In the following chart, we report a confusion matrix that represents the performance of the on a per-class basis.

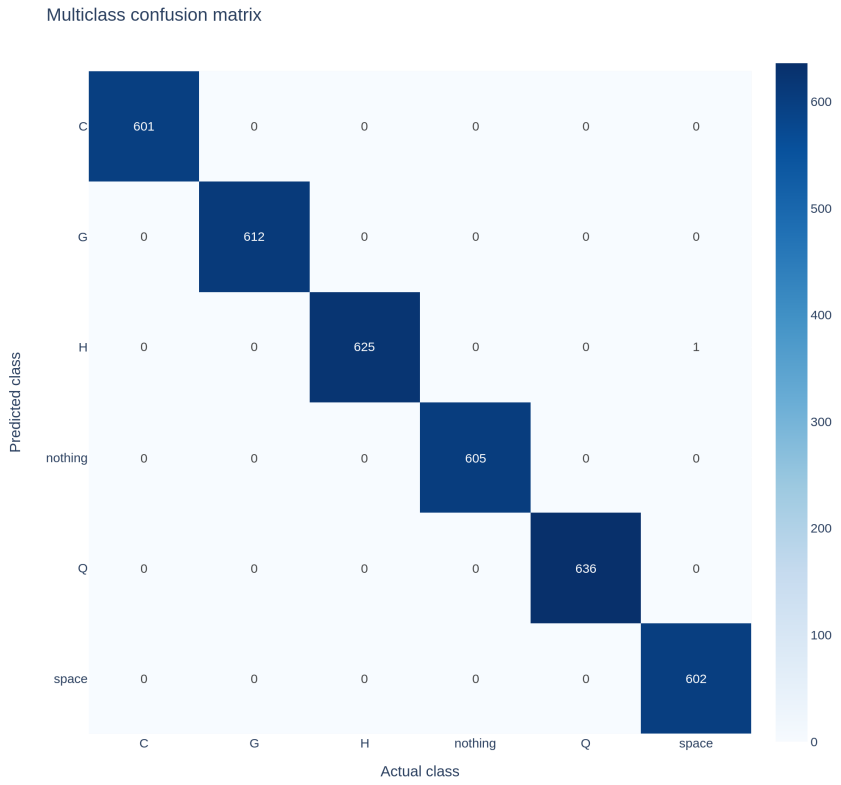


Figure 4.3: Confusion matrix on the validation set of the *5-gestures* model.

In the following charts, we can instead see the normalized error counts per class of samples that the model fails to classify correctly, for each class.

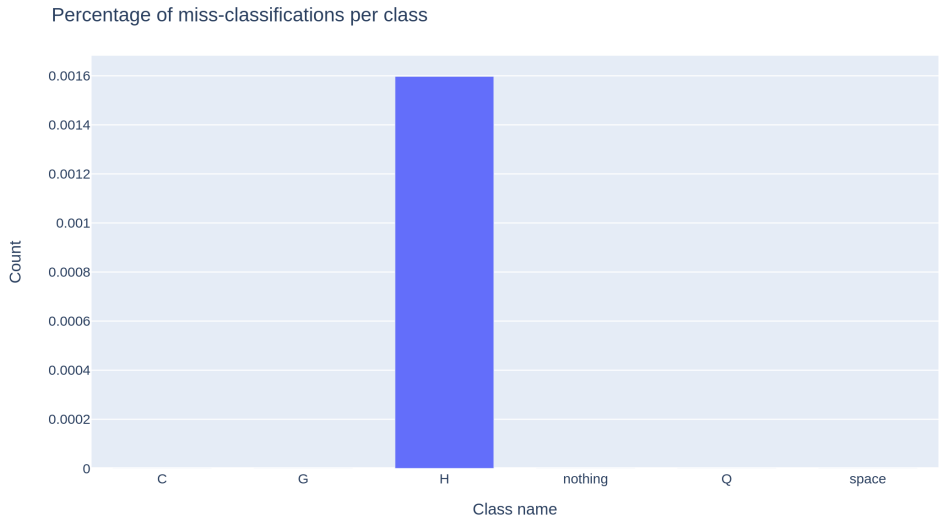


Figure 4.4: Percentage of miss-classification per class of the *5-gestures* model.

We can see that the performance of the model is excellent. Therefore, we are ready to perform the next step, which is quantizing the model in order to reduce its memory occupation. Without this step, the model would be too large to fit on the device. Quantization is covered in the next chapter.

5 | Quantization

The aim of the quantization step is to transform a 32-bit floating point model, into a smaller 8-bit model, in order to make it deployable on our Arduino device, which as already mentioned features a very small amount of memory only. This step allows also for minimizing the inference time, the dimension of the model, and the average RAM usage during the inference time, while the performance of the model can remain approximately the same.

The quantization step has initially been performed by applying Post Training Quantization (PTQ), which takes the non-quantized TensorFlow model and commutes it into a quantized TensorFlow Lite model. In order to obtain the best model possible, we generated 3 different models with 3 different "levels" of quantization.

N.B: The metric of "better" is defined not only by the accuracy but also by the time and memory demand requested during the inference and by the model dimension.

Types of quantizations:

1. **Not quantized:** this model has just been converted into a TFLite model deployable on the Arduino, without optimizations.
2. **Dynamic Range Quantization:** the weights of this model have been statically quantized from floating point to integer at conversion time, which provides 8 bits of precision. However, the outputs of the feature maps are still stored using floating point.
3. **Full Integer Quantization:** the weights have been quantized dynamically to 8 bits of precision using a representative dataset. This version enforces quantization for all operations.

Quantization type	Size [KB]	Latency [it/s]	Accuracy
No quantization	895.00	79.11	0.999
Dynamic Range quantization	283.90	56.10	0.998
Full integer quantization	313.27	59.98	0.991

Table 5.1: Statistics of the model after different kinds of quantization have been applied.

We can notice from the table above that the model with the best overall statistics is the second one. Ideally, we would choose that model, but given that the Microcontroller (MCU) of our target device supports only full integer models, we chose the third one as a final model.

Consequently, we performed full integer quantization on the original TF model, which also converted it into a TFLite model. In the following charts, a confusion matrix of the predictions and the percentage of miss-classifications of the quantized model is shown.

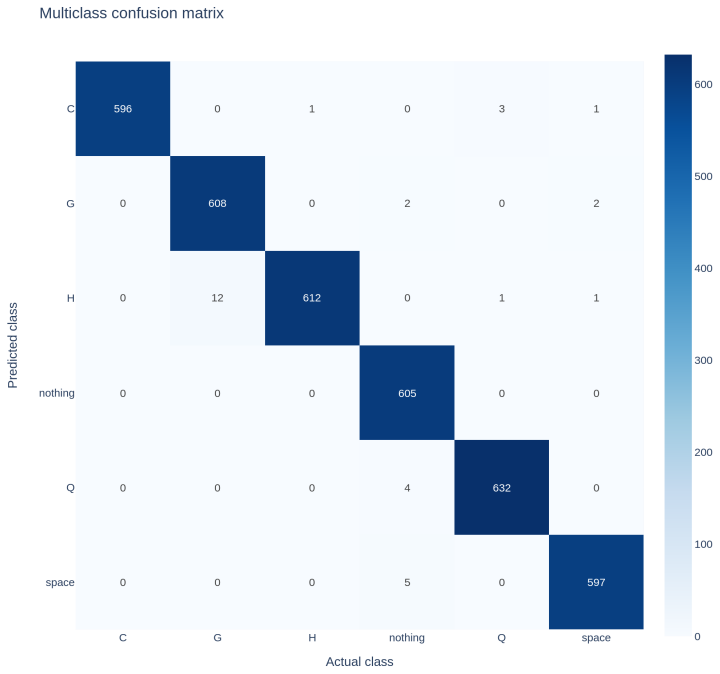


Figure 5.1: Confusion matrix on the validation set of the quantized *5-gestures* model.

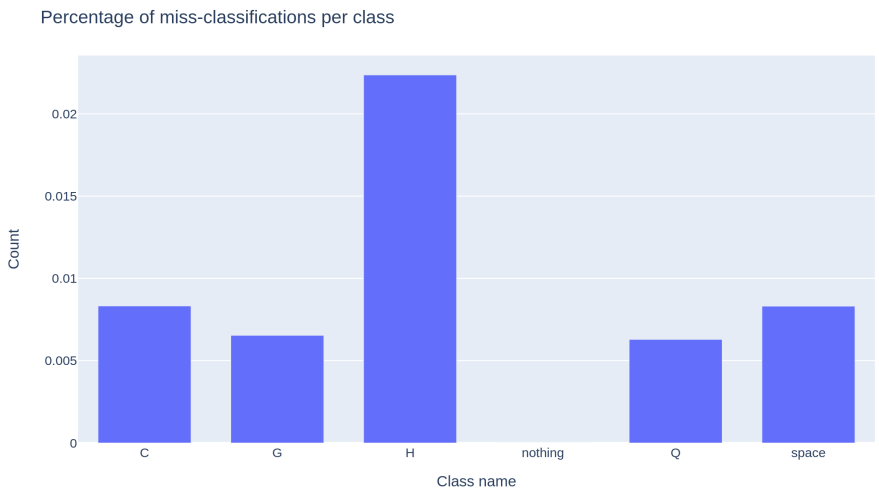


Figure 5.2: Percentage of miss-classification per class of the quantized *5-gestures* model.

We can see that as expected the quantized model performs slightly worse than the unoptimized one, but its accuracy is still very high nonetheless. We proceed to the deployment phase, in order to finally upload the model into the target device.

6 | Deployment

The deployment phase involves transferring the neural network onto the physical device. In order to do it, it's necessary to build the Arduino library and add it to the Arduino IDE libraries, so it will be possible to upload the network on the device and run it in the real world, using the Arduino Nano 33 BLE camera to take pictures and apply the inference.

To build the Arduino library, we uploaded the TFLite model we described in the previous chapter into Edge Impulse. Subsequently, Edge Impulse generated a library encapsulating both the model's weights and the necessary code for executing inference.

However, we quickly realized that, although our model fit within the device's memory, there was insufficient space to accommodate the intermediate feature maps of the model. Consequently, conducting inference on the device was unfeasible.

As a solution, we retrained the identical model on the same dataset, opting to use Edge Impulse directly instead of TensorFlow. Edge Impulse also handled the quantization process, and the deployment was executed in a similar manner. This time, it was possible to use the model on-device to perform inference on real-world data. This suggests that Edge Impulse utilizes some additional optimizations to reduce the size of the model even further.

6.1. Real-world performance

After having deployed the *5-gestures* model on the device, it was found that the accuracy was very poor. The project was tested for some time, with the hands of different individuals and using different backgrounds and lights, but the Arduino output was rarely correct.

We attribute this issue largely to the limited variability in the dataset. Given that all samples were presumably collected by a single individual in a single room, the dataset lacked the necessary diversity. This led to a search for another, more diverse dataset.

7 | Data collection

With the aim of enhancing the dataset, we looked for another dataset, with the hope of finding one created by multiple contributors with varied backgrounds.

Our search led us to the ASL Alphabet Test [8] dataset, which has 30 images for each class, for a total of 870 samples. The images are 256 x 256 8-bit photos to match the ASL Alphabet dataset. This dataset is much more variegate with very a large diversity of lighting conditions and backgrounds.

Subsequently, we employed the OV7675 Camera interfaced with the Arduino to directly capture samples. This step was possible thanks to Edge Impulse, which has a section that helps in collecting data directly from the device. It is sufficient to connect the device to the PC, establish a serial connection with Edge Impulse, and select the sensor we want. It will be possible to see the live camera output and take pictures by clicking a button; the collected image can be labeled with the class we want and added to the dataset. This process was applied in order to collect real-world data, with different types of background/light and different people's hands, so the dataset could be as general as possible.

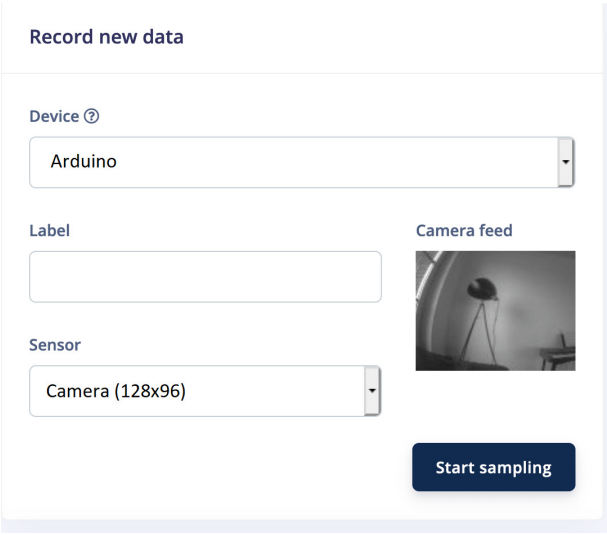


Figure 7.1: Edge Impulse data collection tool

From this moment on, we will refer to the 3 datasets we are working with using the following nomenclature:

- *ASL Classic*: the original dataset we started our work with
- *ASL Real*: the ASL Alphabet Test introduced at the beginning of this section
- *ASL Device*: the dataset we personally collected using the Camera of the device

We merged the 3 datasets into one, which we called *ASL Final* dataset. In order to introduce the least possible amount of bias in the new dataset, we decided to keep the distribution of the 3 separate datasets equal: for each gesture, we introduced 30 (which is the number of samples of the *ASL Real* dataset) samples per dataset, resulting in a total of 90 samples per class. As a consequence, the ASL Final dataset encompasses 540 samples in total.

Figure 7.2: Visualization of 49 samples from the *ASL Final* dataset

Then, we decided to test the performance of the *5-gestures* model on the new *ASL Final* dataset. We divided the dataset into a train set composed of 80% of the samples and a validation set composed of 20% of the samples. This is to allow for a fair comparison with the model we are going to build later. This analysis can be found in the last section of the *5-gestures model.ipynb* notebook of the GitHub repository.

The accuracy of the *5-gestures* model on the validation turned out to be 61.11%. As expected, it is much lower than 99%. Given that the *5-gestures* model had already seen basically all samples from the *ASL Classic* dataset during its training phase, we decided to test its performance first using samples coming from the *ASL Real* dataset only and then with samples from the *ASL Device* only.

This further examination revealed that when tested exclusively on samples from the *ASL Real* dataset, the model achieved an accuracy of 48.89%. However, its performance dramatically declined to 29.44% when assessed with samples from the *ASL Device* dataset.

From these tests, it is clear that *the 5-gestures* model overfitted the specific lighting conditions and backgrounds prevalent in the *ASL Classic* dataset, and that was why it was performing so poorly when deployed in the real world.

8 | 5-gestures-v2 Model

Now that we have a new, much more diverse dataset, we can finally train the last model, with the high hope that a good performance on our ensemble of datasets will translate into a good performance with real-world data. This model, built upon a more diverse dataset, showcases the potential to excel in practical applications and is referred to as *5-gestures-v2* model.

This analysis was performed by using both TensorFlow and Edge Impulse. TensorFlow was used in our local machine in order to experiment faster with the model architecture and hyperparameters. Once the best architecture was found, the model was trained and then deployed using Edge Impulse. The source code can be found in the notebook *5-gestures-v2 model.ipynb* of the GitHub repository.

8.1. Model architecture

The architecture of this model is the same as the *5-gestures* model (refer to Figure 4.1). However, this time we have at our disposal a much, much smaller amount of data since our dataset consists of only 90 samples per class. Normally, this amount of samples is insufficient to effectively train a Deep Learning model. However, we trust they will be enough for our specific setting, thanks to the following factors:

- The model we are working with is tiny and therefore less prone to overfitting
- We are using Transfer Learning, which is known to work well even with very little data to learn from
- We can counter overfitting by tweaking the dropout rate of the Dropout layers inside both the *base* model (MobileNetV1) and the dense classifier. A higher dropout rate will force the model to generalize better, therefore reducing overfitting
- We can use data augmentation techniques in order to artificially expand the training dataset, providing the model with more diverse examples to learn from
- We are not aiming for *perfect* accuracy: due to the diversity of this new dataset,

the *5-gestures-v2* model, even with a respectable accuracy (e.g., 70%), is expected to outperform the *5-gestures* model with 99% accuracy in real-world scenarios

Following a phase of hyperparameter tuning, we ended up using the default dropout rate (0.001) for the base model and a dropout rate of 0.35 in the dense classifier. Furthermore, we used the following data augmentation techniques, manually editing the default data augmentation pipeline that Edge Impulse offers through to its *Keras advanced mode*:

- RandomCrop
- RandomBrigthness
- RandomContrast
- RandomSaturation
- RandomHue

It is worth noting that we refrained from applying common techniques like RandomRotation and RandomFlip since they would sometimes alter the original image by too much, potentially turning a gesture into another.

Of particular note, the RandomHue technique, which adjusts the hue of the input RGB images by rotating the hue channel, proved exceptionally useful in our case since we observed that the OV7675 Camera tends to skew the colors of the image under suboptimal lighting conditions, applying a greenish tint to a significant portion of the image. We are confident that this augmentation technique will allow the model to learn from and accurately classify such scenarios.

8.2. Model training

In this case, during training, only the *fit* phase was executed, omitting the *fine-tuning* phase entirely. This is because experimental tests showed that performing the *fine-tuning* phase had no positive effect on the performance of the model, and in fact also lowered it in some cases. This outcome is likely attributed to the constrained size of the dataset, which is enough to train the dense classifier part of the model but insufficient to train the whole base model, which is much larger.

In the *fit* phase, we are using the Adam optimizer and the Sparse Categorical Cross Entropy loss function in order to perform the gradient descent and the back-propagation. We are using an initial learning rate of 0.001 and training for 100 epochs.

The final results of the training, i.e. the results of the *fit* phase, are displayed in the

following charts.

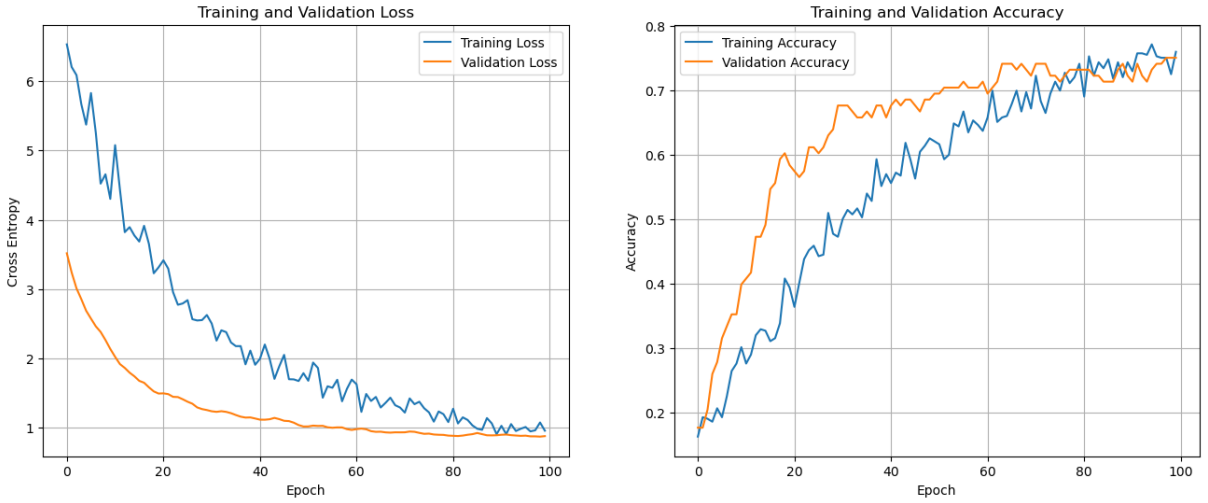
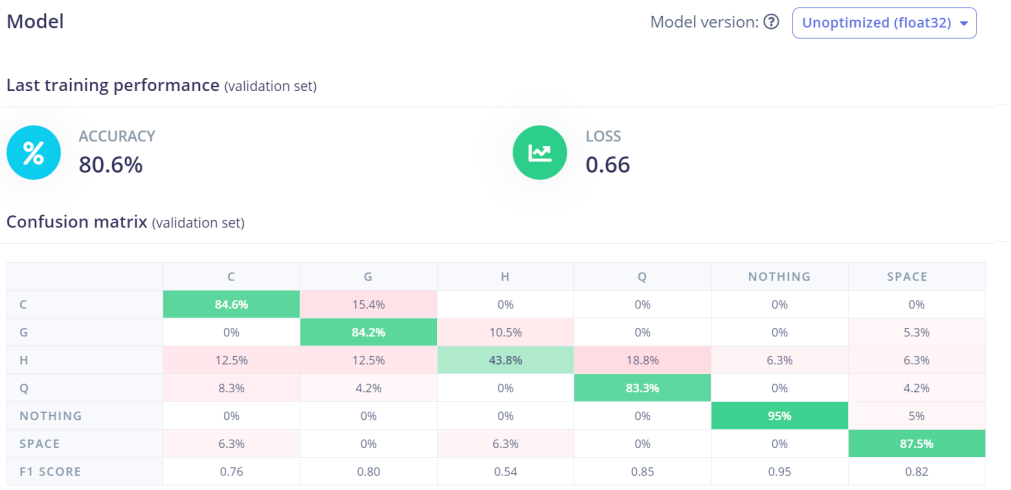


Figure 8.1: Evolution of the loss function and of the accuracy of the *5-gestures-v2* model on both the training and the validation sets.

As depicted in the above figure, the chosen hyperparameter configuration effectively mitigated overfitting, despite the restricted sample size at our disposal.

8.3. Model evaluation

In this section, we take a closer look at the performance of the trained model. This time the training phase was conducted using Edge Impulse. In the figure below we can see a confusion matrix of the predictions of the (not yet quantized) model.



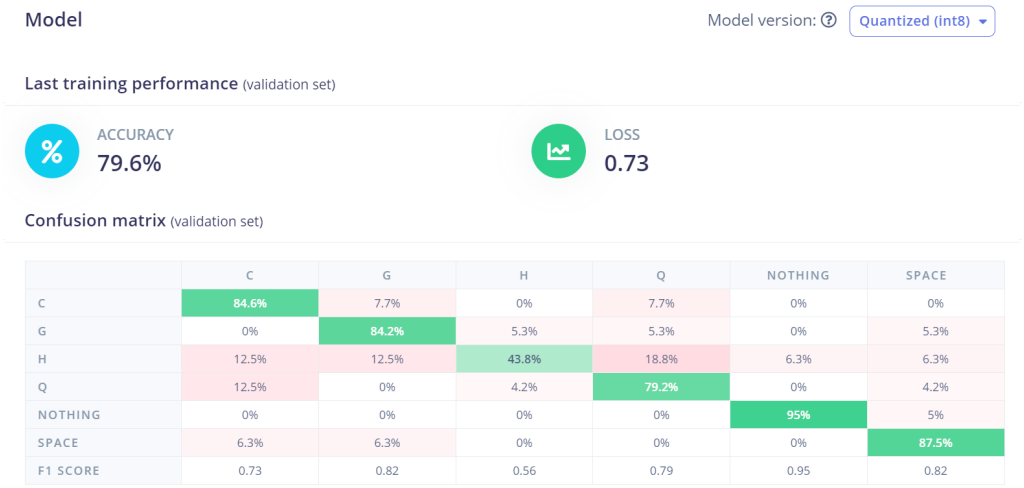
	C	G	H	Q	NOTHING	SPACE
C	84.6%	15.4%	0%	0%	0%	0%
G	0%	84.2%	10.5%	0%	0%	5.3%
H	12.5%	12.5%	43.8%	18.8%	6.3%	6.3%
Q	8.3%	4.2%	0%	83.3%	0%	4.2%
NOTHING	0%	0%	0%	0%	95%	5%
SPACE	6.3%	0%	6.3%	0%	0%	87.5%
F1 SCORE	0.76	0.80	0.54	0.85	0.95	0.82

Figure 8.2: Confusion matrix on the unoptimized 32-bit *5-gestures-v2* model.

The model was also able to reach a commendable accuracy of about 80%. Therefore, we expect that this model will perform well when deployed to the device.

8.4. Quantization

The floating point model has been quantized directly on Edge Impulse, performing full integer quantization. In the figure below we can see a confusion matrix of the predictions of the quantized model.



	C	G	H	Q	NOTHING	SPACE
C	84.6%	7.7%	0%	7.7%	0%	0%
G	0%	84.2%	5.3%	5.3%	0%	5.3%
H	12.5%	12.5%	43.8%	18.8%	6.3%	6.3%
Q	12.5%	0%	4.2%	79.2%	0%	4.2%
NOTHING	0%	0%	0%	0%	95%	5%
SPACE	6.3%	6.3%	0%	0%	0%	87.5%
F1 SCORE	0.73	0.82	0.56	0.79	0.95	0.82

Figure 8.3: Confusion matrix on the quantized *5-gestures-v2* model.

It is evident that the difference in performance w.r.t. the unoptimized version is negligible.

8.5. Deployment

In order to perform the deployment step, an Arduino library has been built, directly from Edge Impulse. Then, the library was added to Arduino IDE, and the "camera" example of that library was run on the device.

After being sure that the code was running and well working, some minor changes were made to the code, in particular, these changes were made in order to make a prediction valid only if its corresponding confidence is larger than 0.5.

This step was implemented in order to minimize the output error of the project. The value 0.5, was found after several tests, and this was found out being the best value so far.

9 | On-device demo

In order to run the project, it is sufficient to upload the code on the device, being sure to place the device with its camera facing up, and, by selecting the correct serial port and opening the serial monitor, it will be possible to see the output of the device.

It will be visible as an official output, so it is the result that the device is giving, followed by the list of possible outputs coupled with their confidence.

```
Edge Impulse Inferencing Demo
Inferencing settings:
        Image resolution: 96x96
        Frame size: 9216
        No. of classes: 6

Starting inferencing in 2 seconds...
Taking photo...
Predictions (DSP: 13 ms., Classification: 792 ms., Anomaly: 0 ms.):
Detected Class: nothing
    C: 0.00781
    G: 0.01172
    H: 0.00781
    Q: 0.01953
    nothing: 0.94922
    space: 0.00391
```

Figure 9.1: Serial monitor showing Arduino output

10 | Market

This project could be used in several market fields, in fact, the purpose of this project would be to easily control some kind of device, by using hand gestures, just by connecting them together.

For example, this project could control a television, allowing the user to turn it on/off, change the channel, and raise or lower the volume, just by making smooth moves with their hand.

It could be also connected to a PC/smartphone webcam, so the user could set some fast frequent actions that the user is used to doing, in order to perform them just by a hand gesture, for instance, such as taking an actual picture that will be saved in the gallery, but also starting a program/app too.

Another application could be gesture recognition on a car during driving, allowing the driver to do fast, actions so that the driver doesn't have to be distracted while driving and is safer.

This project has been thought mostly for a consumer market.

11 | Ethics

An important part of project development is the ethical implications: the developers must ensure to the users that their data will not be stolen or sold to anyone, and basically that the whole application is safe from an ethical perspective.

In particular, this project is ethical since:

- Code, model, and training data are freely available (open-source) in the project repository
- At inference time, the data captured by the device is not stored or sent anywhere, but locally processed and deleted immediately after
- The model is not custom but the same for every user (therefore, even if you steal the device you cannot extrapolate information about the user from the model)
- The device is constantly monitoring the user when turned on, but this is not an issue since the data is not stored anywhere and the device is not transmitting it (e.g. through the internet)
- The user has complete control of the device (can turn it off at any moment, can position it anywhere they want)

Generally speaking, the ethical implications of this project also depend on the context (market) it is inserted into. For instance, no special care needs to be taken if we are using the device to control a video on a laptop. On the other hand, if the device is being used as a Hand Gesture Recognition system inside a car, it is important that it complies with some specific laws and does not distract the user from driving. We note that the gestures used in an HGR meant to be installed in a car should also be easier to perform.

The overall conclusion is that all the data collected by the device is not stored anywhere and the device behavior is completely in the user's hands.

12 | Conclusions

In summary, the main objective of the project was reached, as we were able to deploy a Convolutional Neural Network on the target device and to have it perform inference on images collected with the board's camera in a reasonable amount of time and with reasonable accuracy.

To accomplish this, we utilized the ASL Dataset to train and fine-tune the network. A subset of the 29 gestures present in the dataset was selected by evaluating the classification performance of the *all-gestures* model.

Subsequently, the *5-gestures* model was trained and fine-tuned utilizing the data coming from the aforementioned classes only. While the model was tiny enough to fit into the main memory of the target device, it exhibited a very low performance when making inferences on data coming from the board's camera.

To address this limitation, we enriched the dataset by incorporating the ASL Test Dataset and a collection of images that we personally acquired using the board's camera. With this new dataset, which we called ASL Final, we trained the *5-gestures-v2* model and we deployed it into the device. The architecture and the size of the model are the same as the previous ones, but this one displayed a very good inference performance on data sourced from the board's camera, mainly due to the much more diverse training dataset and the data augmentation techniques used.

Future developments of this project could include reducing the inference time and the model's size even further, increasing the number of recognizable gestures, and/or increasing the accuracy. The preferred direction of these enhancements will depend on the specific use cases of the HGR system we developed.

Bibliography

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.

[2] S. Hymel, C. Banbury, D. Situnayake, A. Elium, C. Ward, M. Kelcey, M. Baaijens, M. Majchrzycki, J. Plunkett, D. Tischler, A. Grande, L. Moreau, D. Maslov, A. Beavis, J. Jongboom, and V. J. Reddi, "Edge impulse: An mlops platform for tiny machine learning," 2023.

[3] C. Nuzzi, S. Pasinetti, R. Pagani, G. Coffetti, and G. Sansoni, "Hands: an rgb-d dataset of static hand-gestures for human-robot interaction," *Data in Brief*, vol. 35, p. 106791, 2021.

[4] A. Kapitanov, A. Makhlyarchuk, and K. Kvanchiani, "Hagrid - hand gesture recognition image dataset," *arXiv preprint arXiv:2206.08219*, 2022.

[5] Akash, "Asl dataset," 2018.

[6] R. Molgora and A. Pirillo, "Hand-gesture recognition with tinyml." `https://github.com/albertopirillo/eeai-project-2023`, 2023.

[7] S. Raschka, J. Patterson, and C. Nolet, "Machine learning in python: Main developments and technology trends in data science, machine learning, and artificial intelligence," *arXiv preprint arXiv:2002.04803*, 2020.

[8] D. Rasband, "Asl dataset test," 2018.