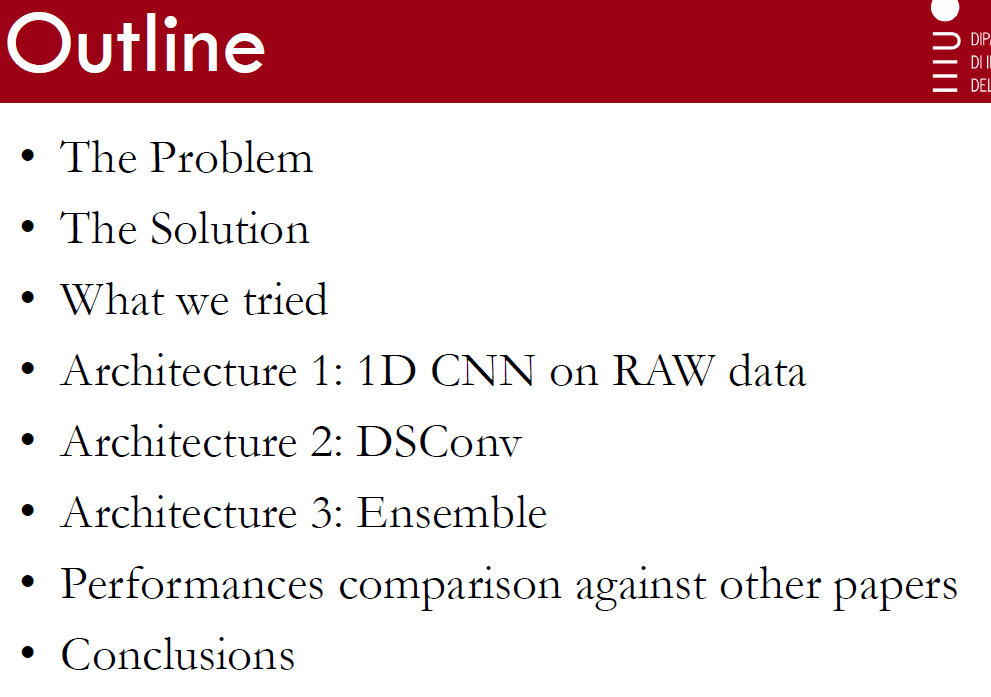
**PRESENTATION SCRIPT**

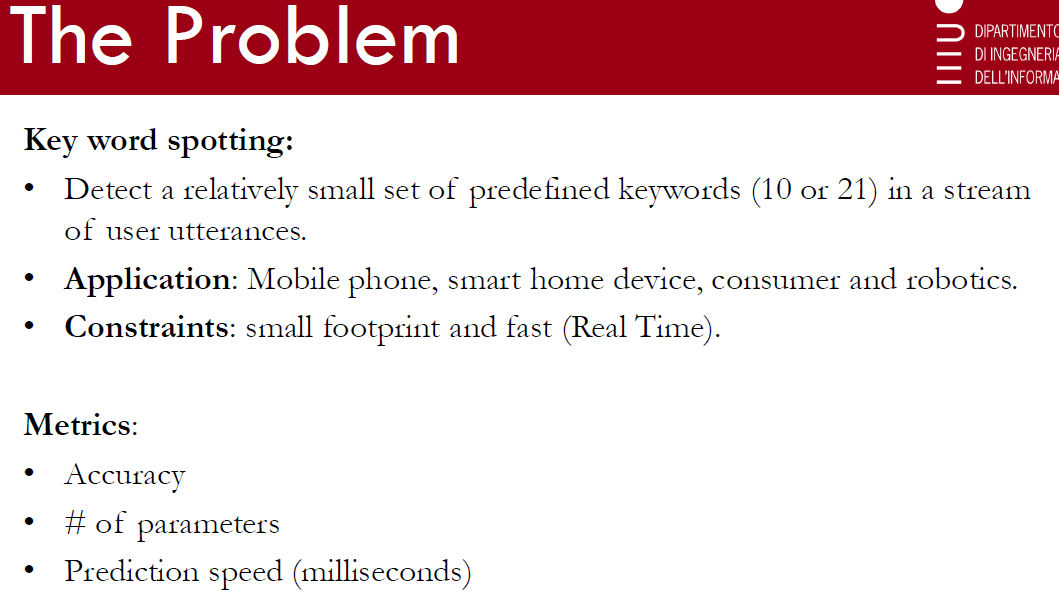
Good morning, we are Stefano Ivancich and Luca Masiero.

**Outline**



Today, we are going to talk about the Keyword Spotting Problem, we will present a possible solution for it, discuss the three architectures we made and we will compare them with other papers.

**The problem**

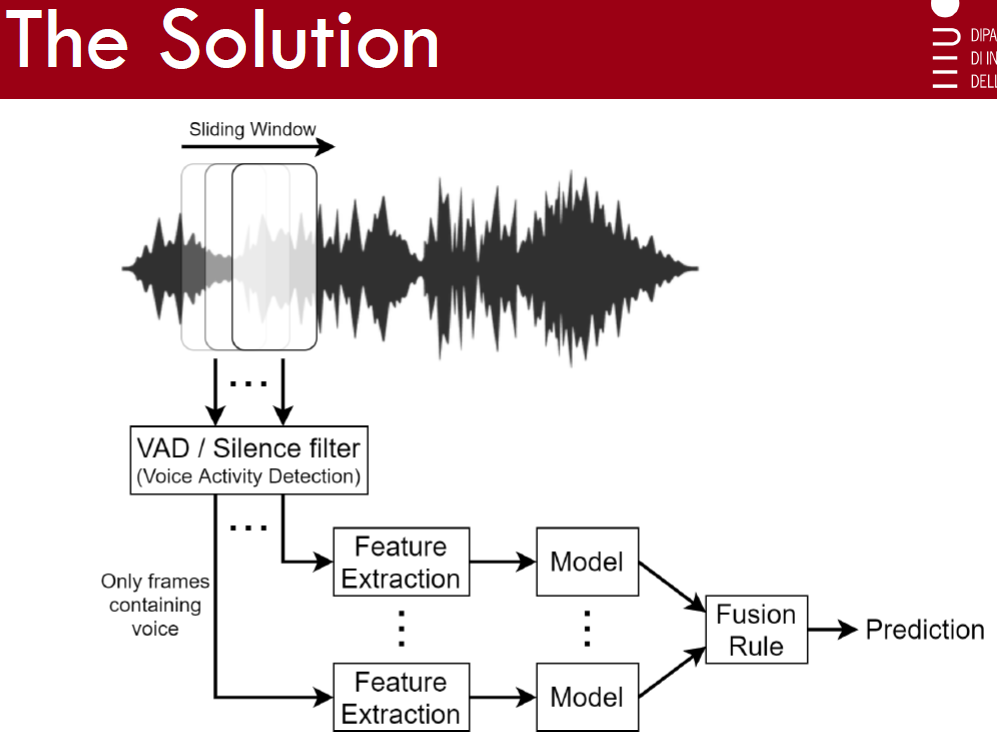


The goal of Keyword Spotting is to detect a relatively small set of predefined keywords in a stream of user utterances, usually on a mobile phone, a consumer “smart home” device or a robotics application.

The constrains are that such models should have a small footprint and a fast prediction speed so they can be deployed on low-power and low-performance-limited devices.

So, the metric we chose for the project are the accuracy of the model, it’s number of parameters and it’s prediction speed.

**The Solution**



One of the main challenges of neural networks in Audio Processing is that the length of the input sample should be fixed, but the sound captured may have various duration.

The aim of the end-to-end architecture we propose, represented in this figure, is to handle audio signals of variable lengths.

One way to avoid the constraint imposed by the input layer of the neural network, is to split the audio signal into different frames of fixed length using a sliding window of 1-second width.

However, letting the model continuously process each frame is too computationally expensive. For this reason, we can use a VAD (a Voice Activity Detection) module or a Silence Filter to let the model process only the frames that contain voice.

This component must be very compact, cheap and very fast, it could be piece of software or even a little piece of hardware.

Only a batch of subsequent windows will be processed. For each of these windows a Feature Extraction module will extract 80 Mels, 40MFCC, and this will be the input of the model.

During the classification we need to aggregate the various predictions to come up to a single decision. For this reason, different fusion rules can be used to reach a final decision, such as the majority vote or the sum rule.

**Live DEMO**

We can have different audio input:

We can record a clip from the microphone

or reding it from a numpy array or a wav file

Now I will record a 10 seconds audio.

Then we can see the waveform of the clip.

We load the model.

With the sliding window technique, we process the clip.

We found that it isn't actually needed a huge amount of windows per second, just with 2 we can have good result.

Now we will use 5 windows per second.

We have a total of 46 windows.

We can have some mistakes, but they will be fixed with the fusion rule.

The average prediction speed per window is ...

But we noticed that predicting 1 single window or a batch of 10 requires the same amount of time.

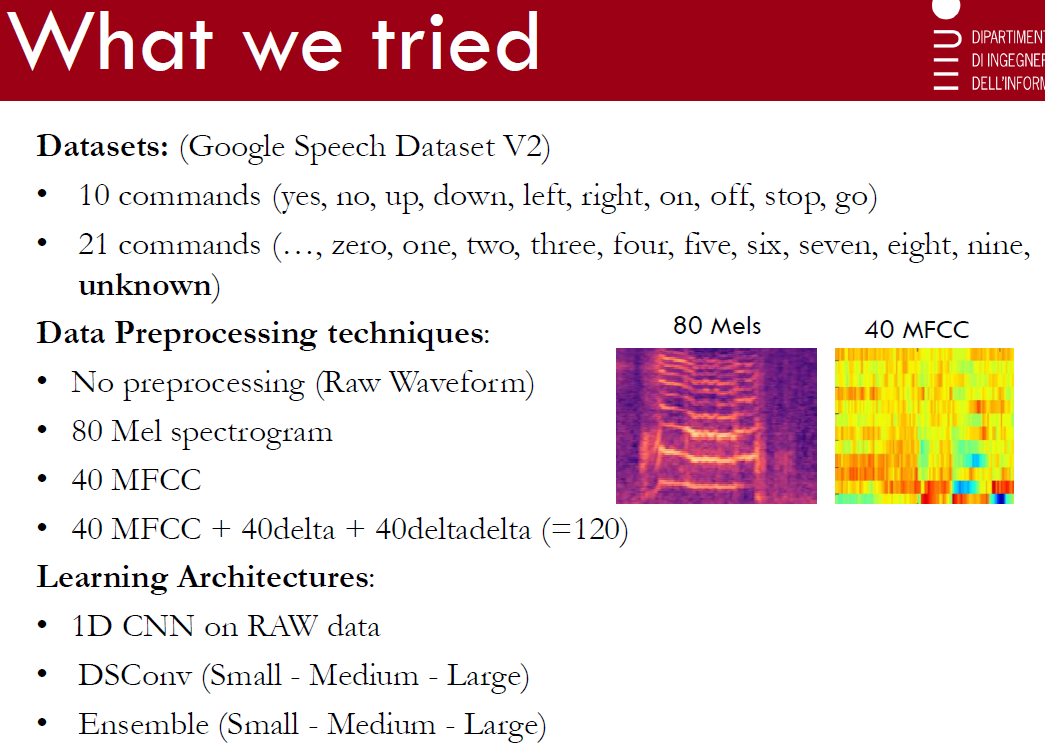
This because TensorFlow works with tensors.

Than with the Fusion Rule, we can see that all the errors are now fixed.

And finally we can plot the waveform again.

To see that all keywords are correctly predicted.

**What we tried**



We used the **Google Speech Dataset V2**. This dataset contains 100K audio files of 1 second divided in thirty classes.

We decided to create **two different datasets**: the first one is made of 10 classes, the second of 21 classes.

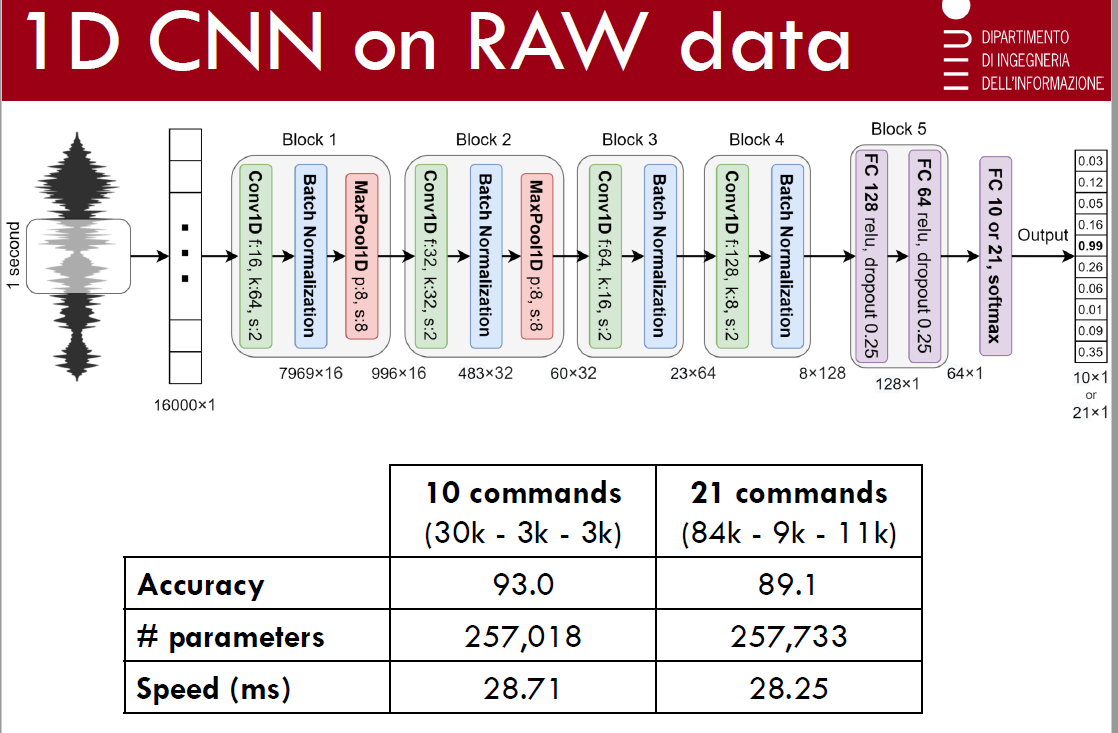
We decided to study four different types of **features** in order to compare them and try to see which was the best one able to solve the problem:

* No feature extraction: we trained a specific model (1DCNN) directly on the raw signal;
* Mel spectrogram with 80 components
* 40 Mel-frequency cepstral coefficients (MFCC);
* 40 MFCC with their first and second derivatives.

We tried different types of neural networks: 1DCNN on raw data and DSConv CNN on features and we built an ensemble of the best models.

Furthermore we implemented different model sizes in order to fit different devices.

**1D CNN on RAW data**



The 1DCNN architecture is made of 4 convolutional layers interlaced with max pooling layers and followed by two fully connected layers and an output layer.

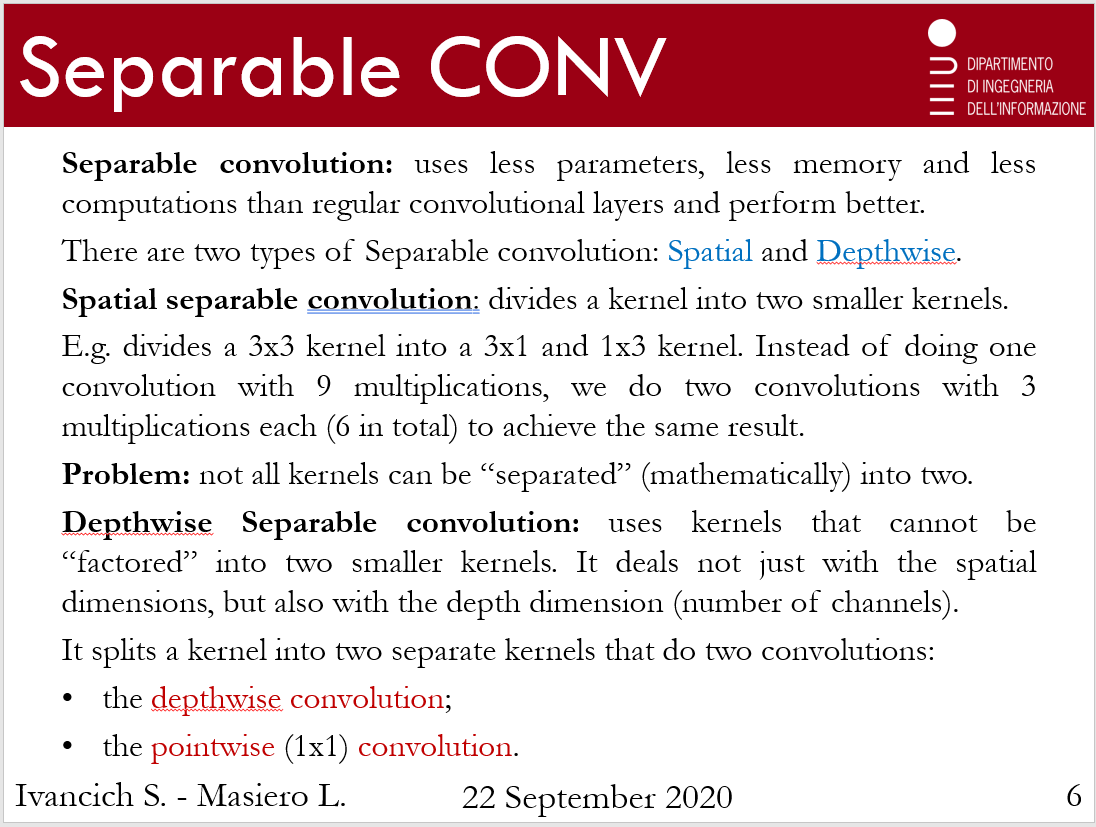
The input consists of an array of 16,000 dimensions, which represents a 1-second audio window sampled at 16kHz.

In order to reduce overfitting, batch normalization is applied after the activation function of each convolution layer and dropout is applied with a probability of 0.25 on the fully connected layers.

The performances of this model are the worst of all models:

* for the 10-commands task it reaches a 93% of accuracy and 89.1 for the 21-command one,
* it use 250K parameters
* and has a prediction speed of 28 ms, that is the fastest compared to all the other models.

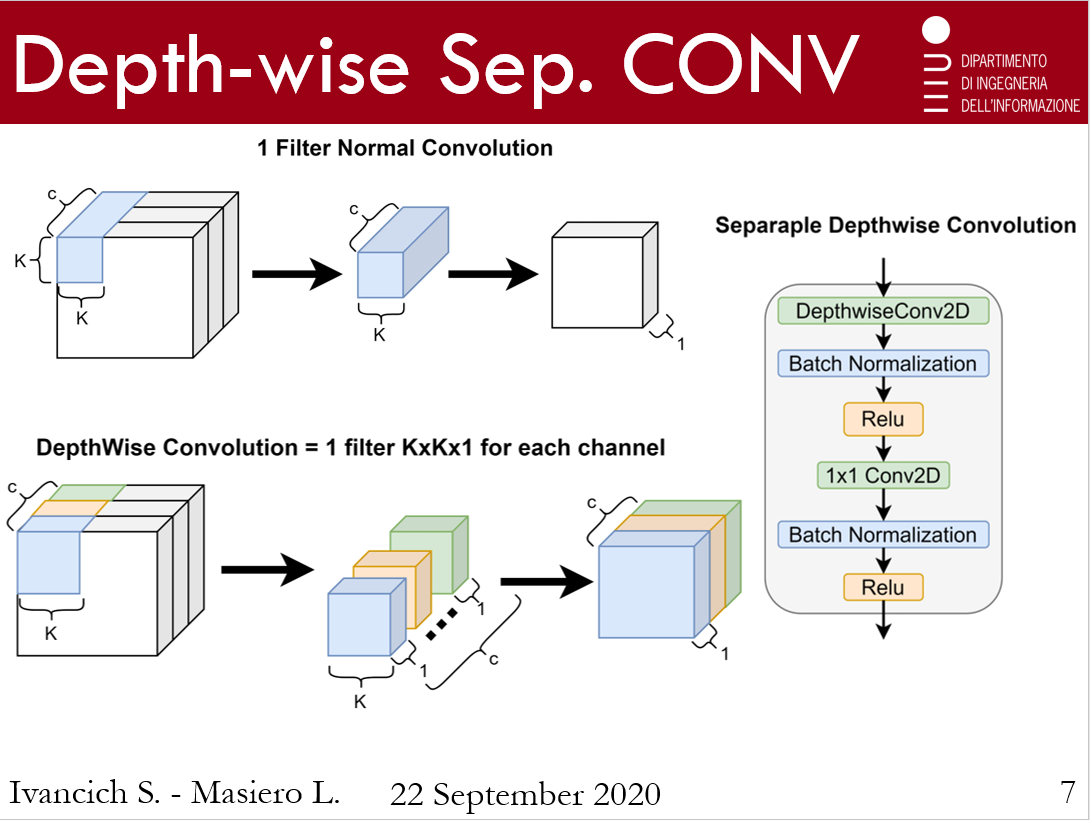
**LUCA**



**Separable convolution** uses a smaller number of parameters, a lesser quantity of memory and a smaller number of computations than convolutional layers that we are used to know, and it even performs better. There are two types of Separable convolution: *Spatial* and *Depthwise*.

**Spatial separable convolution** divides a kernel into two, smaller kernels and the *problem* is that not all kernels can be mathematically split.

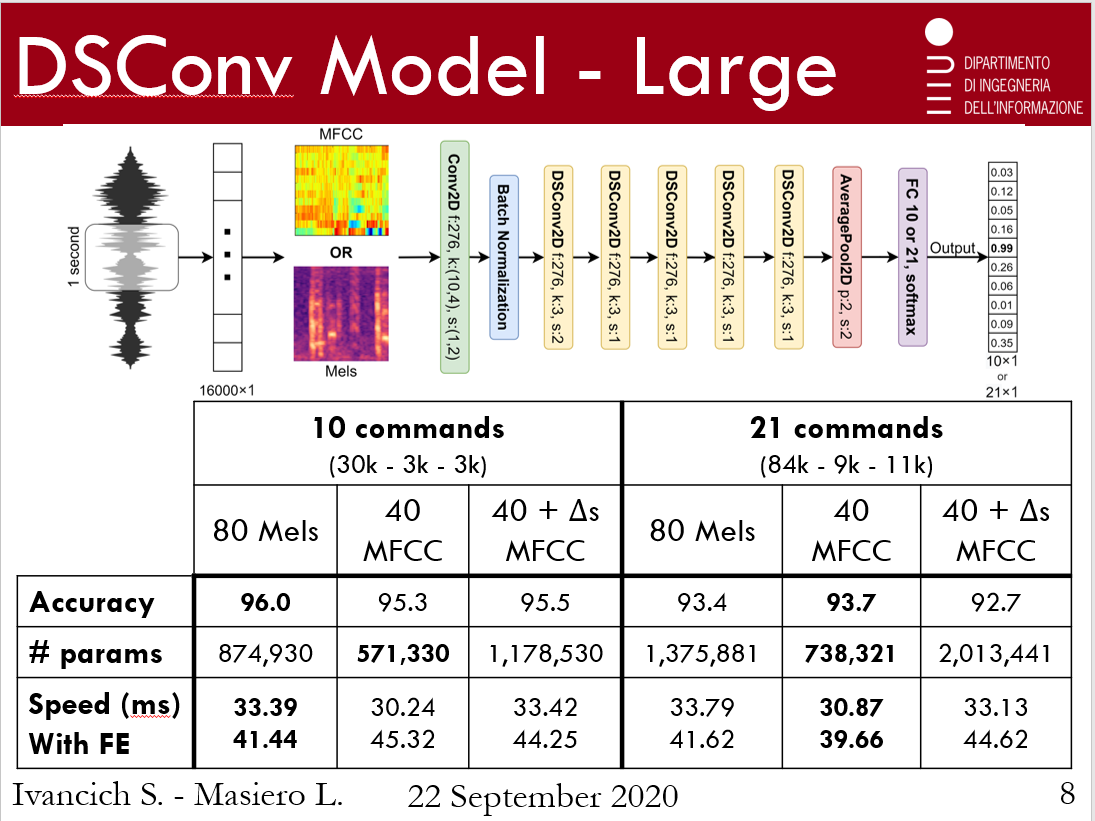
**Depthwise Separable convolution** uses, instead, kernels that cannot be “factored” into two smaller kernels. It deals not just with the spatial dimensions, but also with the depth dimension (so the number of channels). This type of convolution splits a kernel into two separate kernels that apply two more convolutions: the *depthwise* and the *pointwise* one.



In the first picture we have a normal convolution with 1 filter.

The depthwise convolution corresponds to the application of 1 filter for *each* channel.

Then, here we have the Depth Separable convolution layer that we implemented. It is made of DSConv2D and a pointwise 2Dconvolution, with batch normalization and a Relu activation fuction.



The model we implemented takes as input the Mel spectrum or MFCC with or without first and second derivatives.

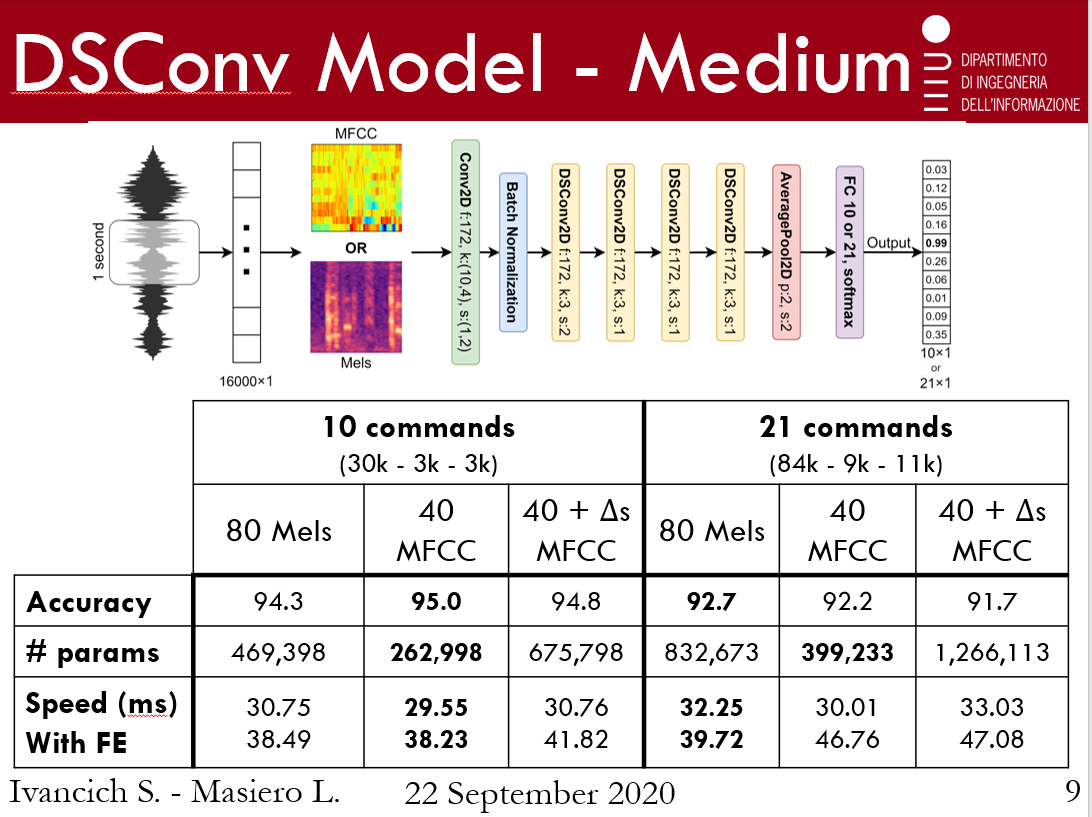
It is made of a normal 2D CNN followed by batch normalization and 5 DSConv2D layers and then we have an average pooling layer followed by a fully connected layer.

We implemented three different variants: *Large*, *Medium* and *Small*, where we tried to reduce the parameters number still maintaining a good accuracy.

For the 10-commands task:

* the best accuracy, of 96%, and speed are reached here, using 80 Mels;
* The smallest model is the 40MFCC with more than half a million of parameters.

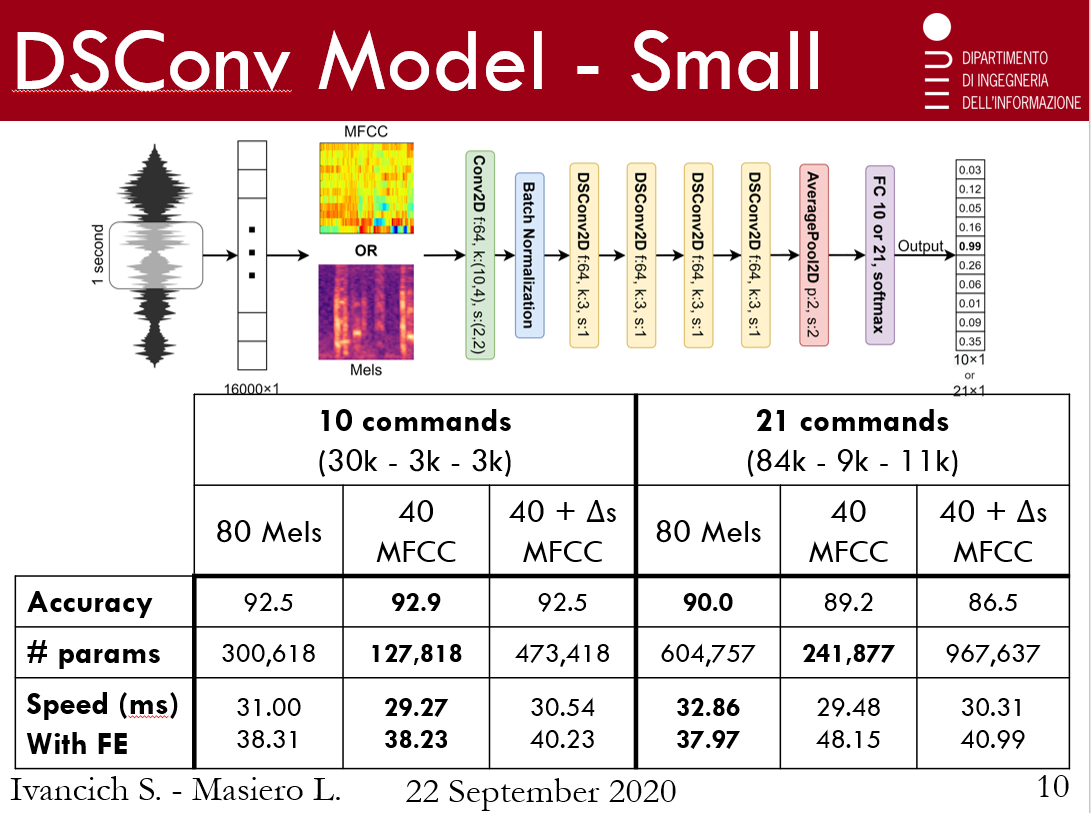
For the 21-commands task: the best model is this one considering all the three different points of view.



The *Medium* variant is the same as before but with 4 DSConv layers instead of 5 and for each convolution we used a smaller number of filters: in the *Large* model we used 276 filters while here 172.

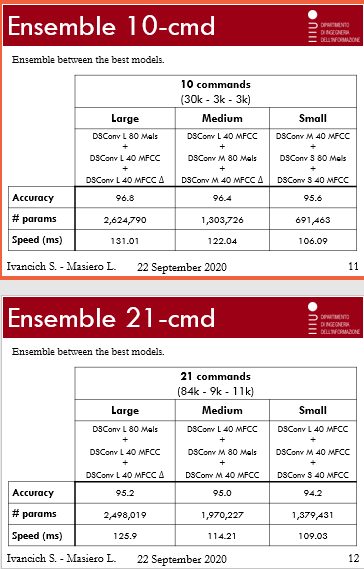
For the 10-commands task the best model 40MFCC while for the 21-commands task:

* the best accuracy of 92.7% and speed are reached using 80 Mels;
* The smallest model is the 40 MFCC with four hundred thousand of parameters.

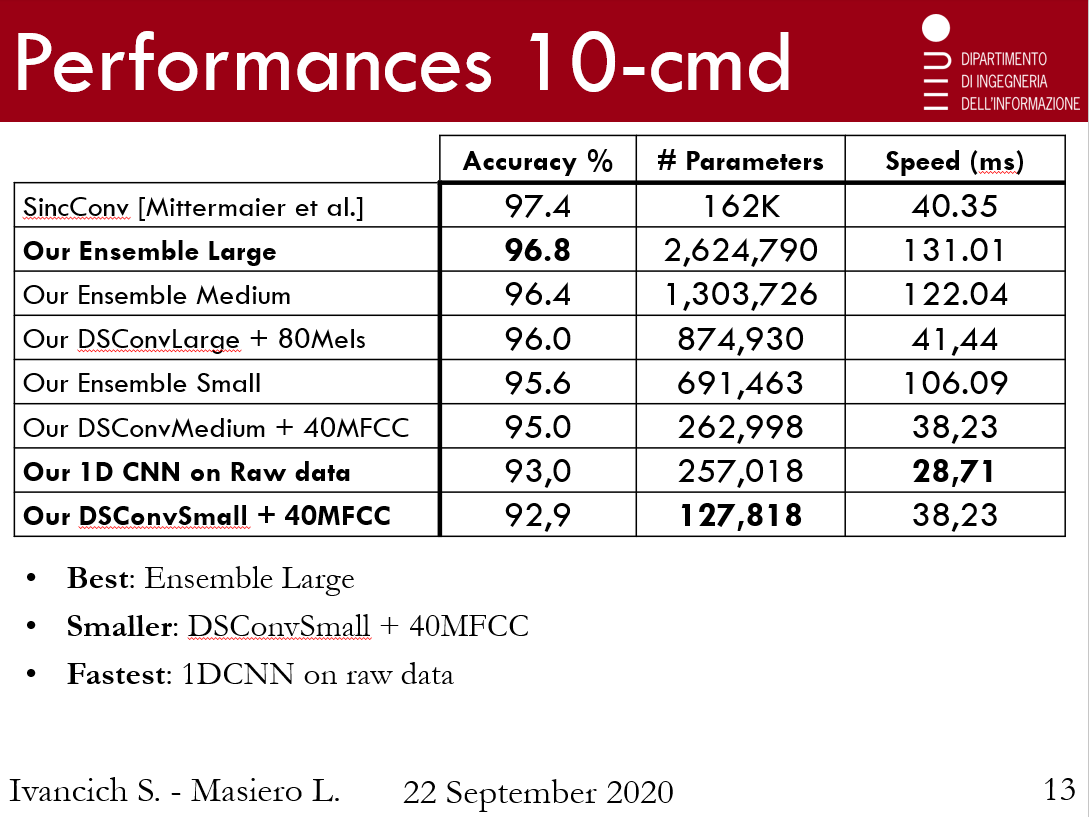


In the *Small* variant, again, we reduced the number of filters of each convolution to 64.

For the 10-commands task the best is this one: 40 MFCC; while for the 21-commands task 80 Mels would have been the best model if it wasn’t for the number of parameters.



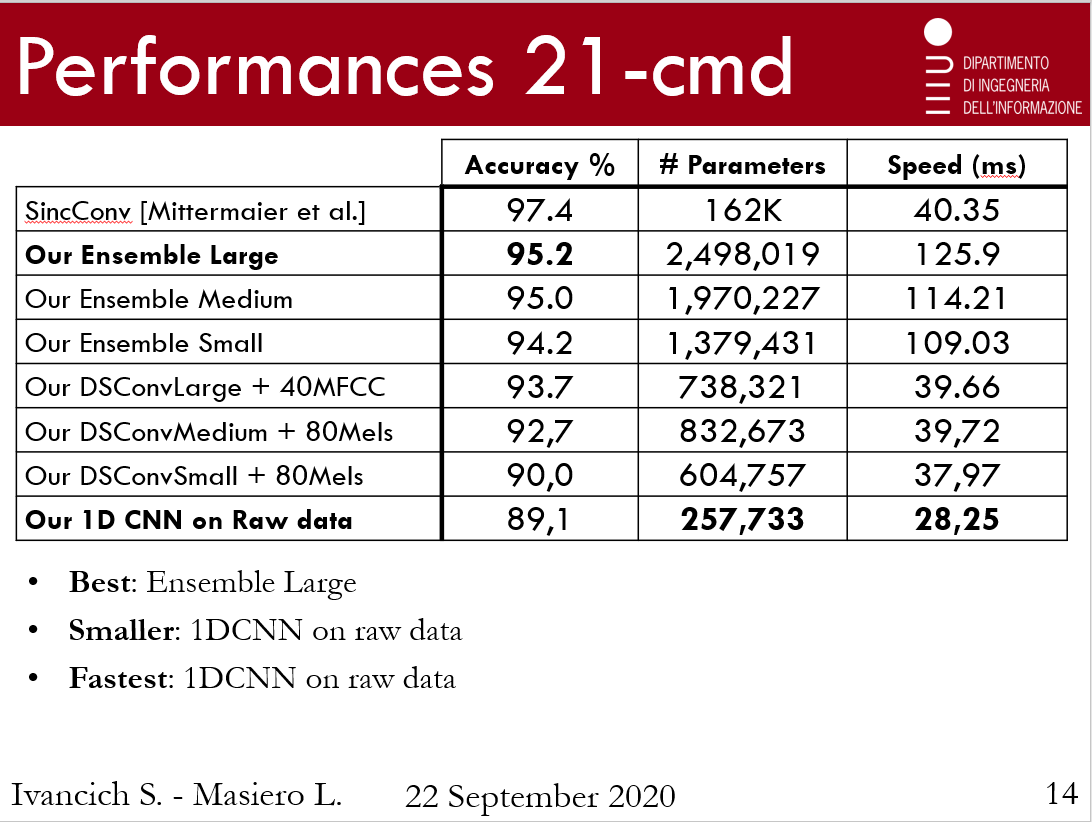
Like I said before, we built three kinds of ensembles: *Large*, *Medium* and *Small*, in order to have models that could fit different hardware sizes. The ensembles simply pick the three best models and compute the average of their final prediction, in this way we reached the best accuracy paying a high price: larger networks; and these are the results for the 10-commands dataset – and the 21-commands one.



We couldn’t reach the state of the art accuracy of 97.4% using only 127K parameters like Mittermaier’s SincConv.

For the 10-commands task:

* the best accuracy of 96.8% is reached by the *Large* Ensemble model paying, however, a high price in terms of number of parameters; a good alternative could be the *Medium* Ensemble that reaches 96.4% of accuracy with a smaller number here with respect to the *Large* ensemble.
* The smallest model is the DSConvSmall + 40MFCC that reaches 92.9% of accuracy with 127K parameters, like the state of the art model SincConv;
* In terms of speed prediction the fastest model is 1DCNN on raw data that reaches 93%.

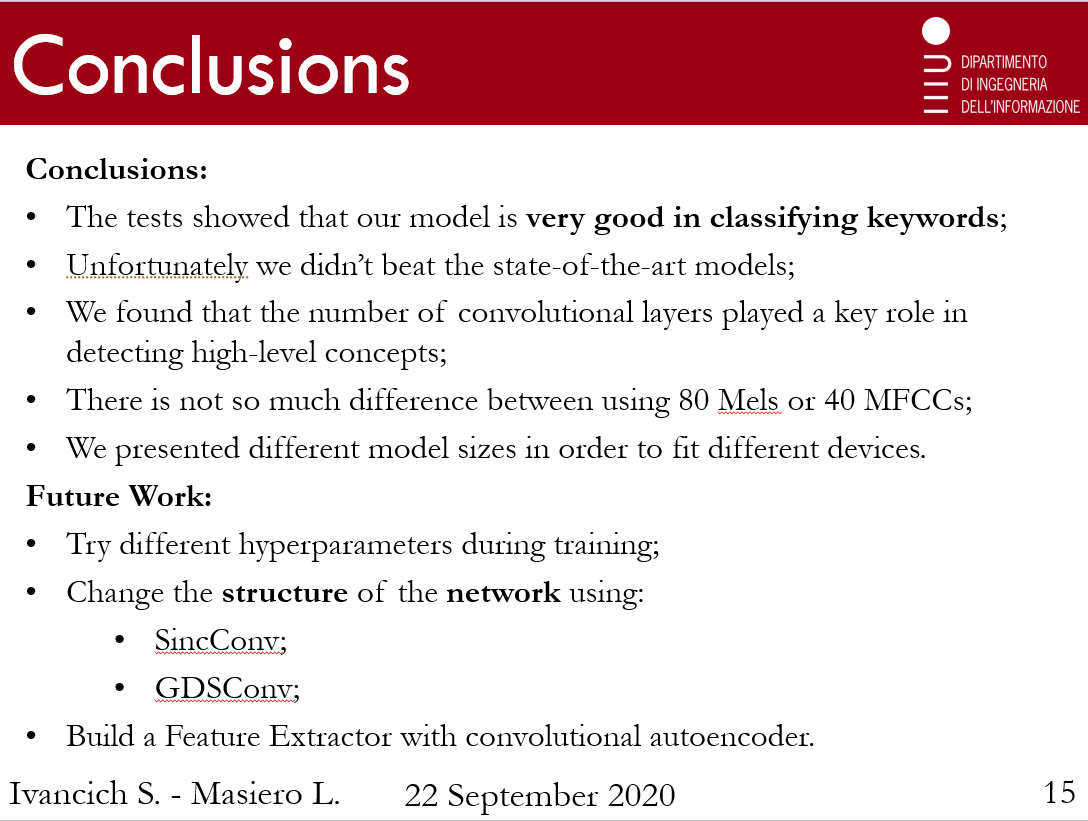


For the 21-commands task:

* the best accuracy of 95.2% is reached by the *Large* Ensemble but with 2.5M parameters;

a good alternative could be the *Small* Ensemble.

* The smallest and fastest model is the 1DCNN on raw data that reaches 89.1% of accuracy with a lower number of parameters and 28ms of speed prediction.



Before showing you a demonstration of our work, we would like to underline that:

* We presented different model sizes in order to fit different devices, that’s why the number of convolutional layers played a key role in detecting high-level concepts. The results have shown that there is not so much difference between 80 Mels or 40MFCC. 40MFCC +Δs actually always performed the worst way.
* Our test have shown that the model we implemented were very good at classifying keywords but unfortunately we didn’t beat the state-of-the-art model.

Briefly talking about future improvements of our work:

* We could try different hyperparameters during the training;
* We could change the structure of the network using SincConv or GDSConv (very similar to SincConv).

Before thanking you for your attention, I return the word to Stefano who will show you a demonstration of what we have done.