End-to-end Keywords Spotting
A minimal deep learning approach

Alfredo Petrella[†]

***Abstract*—In this report an end-to-end approach at the Keyword Spotting task is presented. It includes a Voice Activity Detection module initially applied to the raw input audio, a feature extraction block and a neural network classifier. Different feature extraction methods have been tested, such as 24 log-Mels+frequencies filter, 40 log-Mels and 20 MFCCs, each with different sliding window size and stride. Simple Convolutional Neural Networks (CNNs) architectures have been trained, to obtain lightweight classifiers, and a residual network with both convolutional and recurrent layers has been provided to estimate the reachable accuracy starting from the selected features. The models have been tested on the Google Speech Commands Dataset V2 for 10-classes, 20-classes and 21-classes classification tasks, reaching an accuracy of 90.75% with less then 120k parameters and 92.56% in the 21-classes scenario.**

***Index Terms*—Speech recognition, Keyword Spotting, Small Footprint Keyword Spotting, Convolutional Neural Networks, Recurrent Neural Networks.**

I. Introduction

Small-footprint keyword spotting (KWS) is a crucial task in Artificial Intelligence, being very useful for several applications such as virtual assistance, smart domotic control or wearable intelligent devices. In this project Google Speech Commands Dataset V2 has been used to assess the performance of lightweight classifiers, providing a comparison between different pre-processing techniques for recognising predefined speech commands in a stream of user utterances.

In this framework, the trade-off between accuracy and footprint is key: adapting a model on devices with limited-performance from a memory, energetic and computational point of view, in fact, enables us to avoid transferring user data to the cloud, both reducing the latency of the service and preserving their privacy.

In this work, Convolutional Neural Networks (CNNs), previously and successfully applied to the small-footprint Keyword Spotting task, have been tested on different types of input features in order to minimize the number of parameters and the computational complexity of the models. Moreover, a residual network has been implemented in order to provide a fair benchmark to the simpler models.

In particular, the work consists of:

- a high level description of the whole end-to-end architecture;
- an illustration of the different tested pre-processing approaches (window size and stride, 24 Mels over a reduced frequency spectrum, 40 Mels, MFCC);
- the training of two different neural networks architectures under various conditions (minimal CNN models and residual CRNN);
- an analysis of the memory-efficiency trade-off for the studied models.

The report is structured as follows: in Section II the state of the art in the KWS field is presented, in Section III the approach chosen in this work is described and in Section IV the pre-preprocessing strategy is reported. In Section V, then, the considered architectures are detailed, while Section VI contains the obtained results. Finally, Section VII deals with some considerations about future developments and improvements.

II. Related Work

Traditional approaches in keyword spotting tasks usually involved Hidden Markov Models (HMMs) for modeling both keywords with a phoneme-based approach [1]. In recent years, Deep Neural Networks (DNNs) have demonstrated to be efficient to build small-footprint solutions, as shown, for example,in [2]. More advanced architectures have been successfully applied to KWS problems, such as Convolutional Neural Networks (CNNs), which have exhibited time and memory efficiency [3], while Recurrent Neural Networks (RNNs) have shown to be able to keep longer temporal context thanks to the gating mechanism and their internal states, as shown in [4].

More recently, both depth-wise separable convolutions, like in the model proposed in [5], and temporal convolutions, eventually integrated in a compact ResNet architecture [6], have shown better performance requiring smaller amount of memory, and for this reason, in the latter scenario, differentiable Neural Architecture Search (NAS) are leveraged in [7] to attain 97.2 % accuracy on Google Speech Command Dataset V1 with nearly 100K parameters.

This work mainly follows the path opened by [3], using simple CNNs but investigating possible improvements in term of alternative pre-processing and feature selection strategies, and hyper-parameters tuning.

Then, a slightly more complex model is introduced, partially inspired by [8] and containing both convolutional and recurrent layers in a residual-inspired architecture, in order to provide a benchmark for the metrics of the previously presented models. No significant improvement to the state of the art has unfortunately been reached.

[†]Data Science student, Department of Mathematics, University of Padova, badge n. 1206627, email: {alfredo.petrella}@studenti.unipd.it

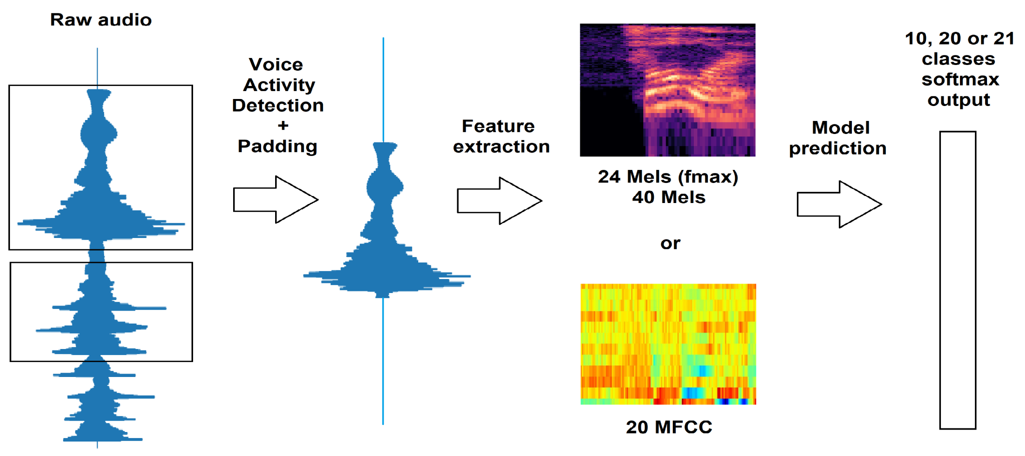

Fig. 1: Proposed end-to-end framework

III. Architecture overview

In order to be of some use, the first requirement that an end-to-end architecture for KWS needs to satisfy is to be enabled dealing with audio signals of variable lengths, and doing it in real time: a solution to this problems could be the one represented in Fig. 1.

In particular, the models that will be described in the next sections perform better when trained with a consistent fixed shape for all the input data, and this must be achieved starting from a captured recording of arbitrary size. The audio signal is then processed and segmented into audio chunks containing a sample louder than a given thresholdfrom a Voice Activity Detector, implemented with the help of the `PyDub` and `Librosa` libraries. This is a crucial point to speed up the processing and save energy and memory. At this point, from the returned chunks, log-Mels or MFCCs are extracted and used as input for the selected predicting model, which thanks to the final *softmax* activation will return the probability associated to each admissible output class.

IV. Signals and Features

A. Datasets description

The Google Speech Commands Dataset V2 contains 105,829 samples belonging to 35 classes, plus five longer tracks containing different noise types of variable duration. All the samples are in `.wav` format, with a sampling rate of 16000, while their duration is not standard: besides the five types of noise (about 10 minutes each), most of the samples are one-second long, but some of them are shorter. For this reason, the latter have been padded with silence to meet others duration. An attempt of cropping the longest ones to a smaller size was also performed with worse results, due to the fact that many utterances are not centered in the sample.

The classes that were also present in the first version of the dataset are *go, stop, on, off, up, down, yes, no, left, right*, to which one should add the digits from 0 to 9 (slightly less frequent) and the least frequent *dog, house, bed, cat, bird, visual, backward, sheila, happy, marvin, wow, learn, follow, tree, forward.*

In this work the models have been trained, validated and tested according to the corresponding split suggested in the original dataset paper [9], using the *SHA-1* hash function applied to the speaker IDs (part of the file name) to generate probability values in order to avoid their utterances to appear in more than one set, distorting the computed metrics, and to keep the split consistent over time, also when new data is added.

Moreover, three different datasets have been considered, based on the frequency of the classes they contain: the original 10-classes dataset, a 20-classes dataset including the second most frequent labels and a 21-classes dataset, different from the previous one for including the *other* class to classify all the input sequences belonging to the 15 remaining classes.

In the three cases, the datasets roughly contain respectively:

- 10-classes: 25k, 8k, 8k samples;
- 20-classes: 46k, 15k, 15k samples;
- 21-classes: 64k, 21k, 21k samples.

In all the cases, an intensive grid search for hyper-parameters tuning over the training and the validation set have been executed, and the best resulting models have then been evaluated over the actual test set.

B. Feature extraction

Being one of the points this work wanted to focus on, a quite diverse set of combinations of initial parameters have been investigated. Namely, the number of log-Mels coefficients to be extracted (24, 40, 60, 80), the width and the stride size of the FFT sliding window over the raw data (with different mutual proportion and overlap, starting from the (0.025, 0.01) pair adopted in [9]), and the options of whether to apply pre-emphasis (95%) or low/high pass filters. Some preliminary models have been evaluated over each combination, always with an eye on efficiency, penalizing small improvements with a significant time cost.

In the end, three different approaches have been elected and used to extract features from the raw input data in order to minimize the number of parameters of the resulting model:

- 24 log-Mels spectrogram ignoring all frequencies above 1760 Hz, as an upper bound to the highest notes an opera soprano is usually required to sing, before windowing (0.05 s length, 0.019 s stride performed better than what was suggested in [3]), applying the FFT, mapping to a log-Mel scale and filtering with overlapping triangular filters;
- 40 log-Mels spectrogram without masking the highest frequencies, windowing (0.05 s length, 0.025 s stride performed better here), same process as above;
- 20 MFCCs obtained from the 40 log-Mels coefficients computed as described in the previous point.

To speed up the implementation and the training and evaluation phases, `Librosa` python library have been used, and the differently pre-processed datasets have been saved in `.pickle` format in order to reduce the disk space needed to store the data and avoid assigning to the right set (train, validation, set) and extracting the same features for the inputs at each new evaluation session, consuming resources and slowing down the process.

V. Learning Framework

A. 2D-CNN

The first architecture is shown in Fig. 2. It is, in its general form, composed of only four two-dimensional convolutional layers, each followed by a batch normalization layer, to speed up the training and improve generalization, and a Rectified Linear Unit (ReLU) activation function for the hidden layers, trying to encourage the sparsity of the network and, therefore, reducing the computational cost of the final prediction. Moreover, after the first and the last two blocks, an average pooling layer is applied to reduce the shape of the model and, thus, its memory footprint. After the first activation, a small dropout is also applied in some case to regularize the training and generalize against noise.

As for the input shape, it depends on the adopted feature extraction technique and from the sliding window length and stride. In general, the first component refers to the time dimension, while the second one is related to the frequency, so the considered input shapes are `(38, 40)` for the 40 log-Mels case, `(50, 24)` for the 24 log-Mels case and `(38, 20)` for the MMFCs.

After the convolutional blocks, a flattening layer allows the single following dense layer, with `nClasses` neurons followed by a *softmax* activation, provides the output probability distribution of the input sample belonging to each class.

As a general rule, it can be said that the kernel shape is decreasing along the architecture, also due to the subsequent average pooling layers, and also the number of filters of the last convolutional layer shouldn't be too big in order to limit the total number of parameters, strongly influenced by the number of connections in the final fully connected layer.

The parameters which are not present in Fig. 2 are the ones over which a grid search has been implemented in order to find the best model for each task. In particular, the selected parameters are:

- the size `kernelSize1` of the filters in the first convolutional layer (different values based on the input shape): intuition suggests that the shape of the filters in input is key in order to catch information about the local relations inside each frame, and actually modifying this hyper-parameter is remarkably effective on the computed metrics. In particular, in some cases, a larger filter (on the frequency axis) helped improving the overall model;
- the number `nFilters4` of filters of the last convolutional layer, in [16, 32, 64]: given that the following are a flattening and a fully connected layer, this value strongly influences the total number of parameters, and it also constitute a potential bottleneck for the model. It was then key to fine-tune it's value;
- the dropout value `dropOut` after the first activation, in [0.0, 0.1]: an eventual presence of the dropout (small on purpose, given the small dimension of the studied model) is aimed to make the network stabler against the noise which is intrinsically present in the input signals;
- the L_2 regularization rate `l2Reg` multiplied by the L_2 norm of the gradient after each activation in [0, 0.05]: as above, it has been introduced to deal with the noise problem, but in all the cases the model performance resulted strongly penalized from its presence;
- the training batch size `batchSize` in [128, 256]: depending on the (variable) input and the model sizes, a smaller batch size can help avoiding overfitting.

The parameters of the best performing models for each dataset configuration and feature extraction technique are presented in Tab. 3 for the 10-classes version of the problem, in Tab. 4 considering 20 classes and in Tab. 5 adding all the missing classes as *other*.

B. 2D-residual CRNN

The second architecture is shown in Fig. 6. Like the ResNet50 architecture, it is composed by a first block including a two dimensional convolutional layer followed by a batch normalization layer, to speed up the training and improve generalization, and a ReLU activation function. The ResNet50 max pooling layer has been substituted with an average pooling layer which has shown better performance in the previous model.

After the first stage, the analogous of a ResNet50 *identity block* is inserted in the network, and it is thus consisting of two convolutional layers, each followed by a batch normalization and a ReLU activation layer, and than one more convolutional-batch normalization block; before the final activation, a shortcut from the output layer of the first stage is added, with kernel size of 1 and the same number of filters of the last layer to match its dimension, and then the ReLU activation is applied.

The exact same stage is replicated a second time, and then, after the needed reshape, a GRU layer is inserted to catch the

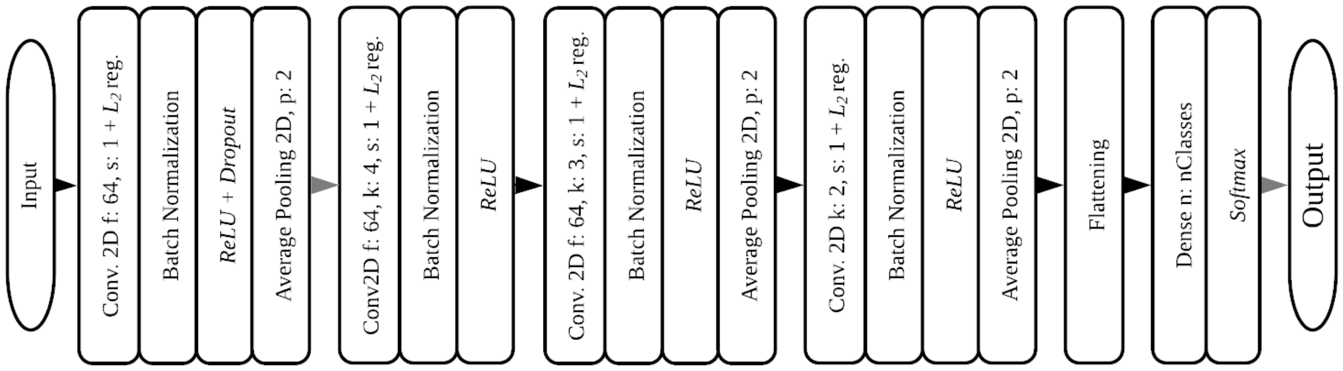

Fig. 2: Minimal 2D-CNN network model; the missing hyper-parameters are the ones varying during the implemented Grid Search.

Hyper-parameter	**24 log-Mels**	**40 log-Mels**	**20 MFCCs**
`kernelSize1`	8	[8,32]	8
`nFilters4`	16	16	16
`dropOut`	0.1	0	0.1
`l2Reg`	0	0	0
`batchSize`	256	128	128
Total number of parameters	114522	127130	113242

Fig. 3: Best performing models' hyper-parameters, 10-classes setting.

Hyper-parameter	**24 log-Mels**	**40 log-Mels**	**20 MFCCs**
`kernelSize1`	8	8	8
`nFilters4`	16	32	16
`dropOut`	0.1	0.1	0.1
`l2Reg`	0	0	0
`batchSize`	128	128	128
Total number of parameters	117412	128628	114852

Fig. 4: Best performing models' hyper-parameters, 20-classes setting.

eventually remaining temporal correlation among the features, followed by the usual ReLU activation layer.

This network has only been evaluated with the 40 log-Mels feature type, given that applying such an architecture to smaller inputs would have made little sense, so the input shape is `(38, 40)` as explained before.

At the end, as before, the final dense layer, with `nClasses` neurons followed by a *softmax* activation, provides the output probability distribution of the input sample belonging to each class. In this case, the tuned hyper-parameters have been:

- the size `kernelSize1` of the filters in the first convolutional layer in [6, 8, [6, 16], [8, 16]], for the same reasons discussed above;
- the size `kernelSize3` of the filters in the second identity block repetition, in [3, 4], to relax some assumptions of our principal model structure;
- the number `nGRU` of GRU cells in the recurrent layer, in [128, 256];
- the training batch size `batchSize` in [128, 256].

The best performing parameters can be found directly in Fig. 6.

All the described models were trained with `Keras 2.5.0`, minimizing the sparse categorical cross entropy via an *Adam* optimizer. The maximum number of epochs was fixed to 20, but early stopping with a patience of 6 epochs was implemented as well in order to avoid overfitting.

Hyper-parameter	**24 log-Mels**	**40 log-Mels**	**20 MFCCs**
`kernelSize1`	8	8	8
`nFilters4`	16	16	16
`dropOut`	0	0	0.1
`l2Reg`	0	0	0
`batchSize`	256	128	128
Total number of parameters	117701	118373	115013

Fig. 5: Best performing models' hyper-parameters, 21-classes setting.

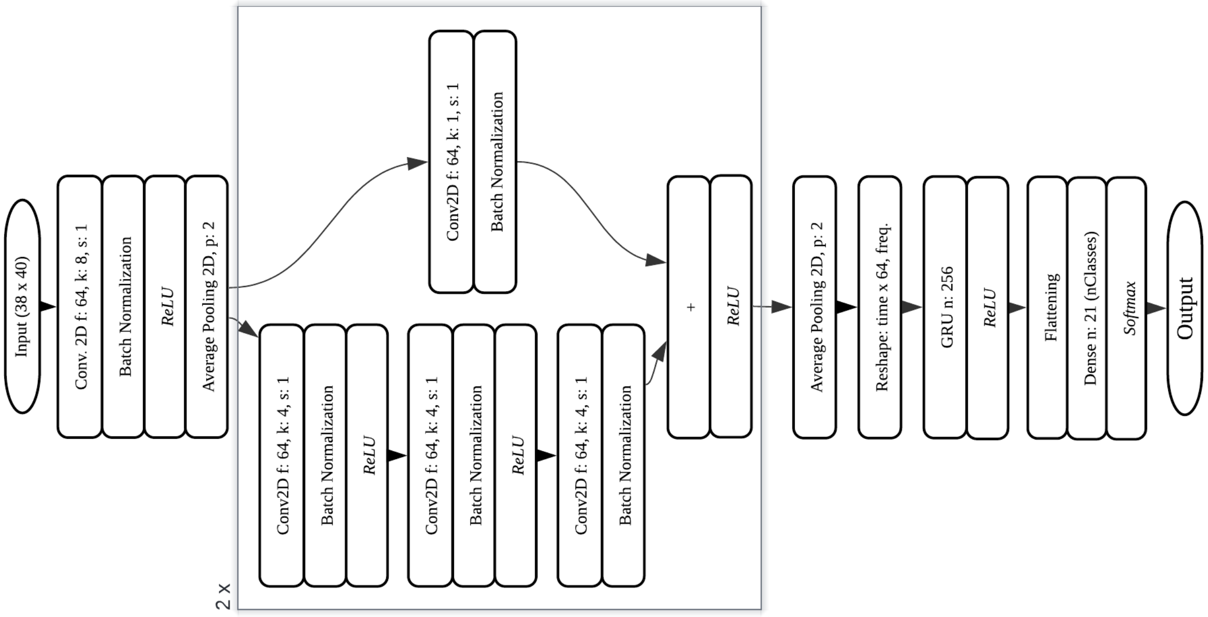

Fig. 6: 2D-residual CRNN.

VI. Results

In this section, the main results of the grid search and the metrics values on the test set will be reported and commented.

The considered metrics are the accuracy along the entire training phase, to spot and prevent eventual anomalous behaviors, and the weighted precision, recall and F1-score of the best model as well. In accordance with the goal of this work, however, a smaller number of parameters in the models played a key role when the alternative was to prefer a slightly more precise but bigger model in the election of the best candidate. As already mentioned, the models have been trained with three nested datasets of 10, 20 and 21 classes, and for each of them, the input has been processed in three distinct ways.Showing all the results for each combination of hyper-parameters would have been unfeasible, so a selection of the most interesting outcomes has been performed.

In all the cases, the presence of the batch normalization layers allowed the models to improve their performance: given the presence of relatively few parameters and a big number of input, achieving a faster training is key to avoid overfitting. Moreover, the L_2 regularization rate will not be indicated and must be considered 0, given that it uniformly worsens the models up to a 20% with respect to the corresponding not regularized ones.

As expected, the tuned 2D-CNNs generally perform better in the simpler 10-classes task and the metrics gradually deteriorate as new labels are added. In most cases, wider kernels in the first convolutional layer appear to perform better. This in not due to the possibility, for the first layer, to have a comprehensive sight on broader frequency ranges, but only to the presence of a bigger number of trainable parameters. In these cases, the bottleneck constituted by the rest of the structure, imposed to avoid the model size to grow too much, result in jumpy loss and accuracy and thus in apparently good but unstable models. For this reason, with one exception, smaller filters are preferred in the selected models, even if looking at Tables 7, 8, 9, 10, 11, 12, 13, 14, 15, it could seem inappropriate.

Results for the best models in each case are shown in Tables 16, 17, 18 for the minimal models. Clearly, their

Hyper-parameters				**Metrics** (weighted average)				
`kernelSize1`	`nFilters4`	`dropOut`	`batchSize`	**Accuracy**	**Precision**	**Recall**	**F1-score**	# param.
8	16	0.1	256	89.89	90.34	89.89	89.94	114522
[8,24]	16	0	256	91.13	91.29	91.13	91.13	126810
[8,24]	32	0	128	89.61	90.26	89.61	89.63	133866

Fig. 7: Best performing models on the validation set, 24 log-Mels, 10 classes setting.

Hyper-parameters				**Metrics** (weighted average)				
`kernelSize1`	`nFilters4`	`dropOut`	`batchSize`	**Accuracy**	**Precision**	**Recall**	**F1-score**	# param.
8	16	0.1	128	89.61	90.12	89.61	89.59	117418
[8,24]	16	0.1	128	90.37	90.86	90.37	90.39	129700
[8,24]	32	0	128	91.09	91.30	91.09	91.11	139636

Fig. 8: Best performing models on the validation set, 24 log-Mels, 20 classes setting.

Hyper-parameters				**Metrics** (weighted average)				
`kernelSize1`	`nFilters4`	`dropOut`	`batchSize`	**Accuracy**	**Precision**	**Recall**	**F1-score**	# param.
8	16	0	256	88.08	88.59	88.08	88.10	118522
[8,24]	16	0.1	128	89.02	89.52	89.03	89.16	129989
[8,24]	32	0	128	89.20	89.26	89.19	89.32	140213

Fig. 9: Best performing models on the validation set, 24 log-Mels, 21 classes setting.

Hyper-parameters				**Metrics** (weighted average)				
`kernelSize1`	`nFilters4`	`dropOut`	`batchSize`	**Accuracy**	**Precision**	**Recall**	**F1-score**	# param.
[8,32]	16	0	128	93.34	93.39	93.34	93.33	127130
8	32	0	128	92.95	93.21	92.95	92.98	122218
[8,32]	32	0.1	256	93.37	93.52	93.38	93.38	134506

Fig. 10: Best performing models on the validation set, 40 log-Mels, 10 classes setting.

Hyper-parameters				**Metrics** (weighted average)				
`kernelSize1`	`nFilters4`	`dropOut`	`batchSize`	**Accuracy**	**Precision**	**Recall**	**F1-score**	# param.
8	16	0	128	93.28	93.49	93.28	93.28	118052
8	32	0.1	128	94.00	94.07	94.00	94.00	128628

Fig. 11: Best performing models on the validation set, 40 log-Mels, 20 classes setting.

Hyper-parameters				**Metrics** (weighted average)				
`kernelSize1`	`nFilters4`	`dropOut`	`batchSize`	**Accuracy**	**Precision**	**Recall**	**F1-score**	# param.
8	16	0	128	91.62	91.74	91.62	91.61	118373
[8,32]	16	0.1	128	91.63	91.78	91.63	91.64	130661
[8,32]	32	0.1	256	91.95	92.13	91.95	91.96	141557

Fig. 12: Best performing models on the validation set, 40 log-Mels, 21 classes setting.

performance strongly depends on the number of entrances of the input features, but it is worth noting that the gap between the metrics gets smaller as the number of classes grows, indicating that the trained models are probably too

Hyper-parameters				**Metrics** (weighted average)				
`kernelSize1`	`nFilters4`	`dropOut`	`batchSize`	**Accuracy**	**Precision**	**Recall**	**F1-score**	# param.
8	16	0.1	128	88.67	88.93	88.67	88.66	113242
8	32	0.1	128	89.56	89.84	89.56	89.59	119018
10	32	0.1	256	89.84	90.14	89.84	89.83	121322

Fig. 13: Best performing models on the validation set, 20 MFCC, 10 classes setting.

Hyper-parameters				**Metrics** (weighted average)				
`kernelSize1`	`nFilters4`	`dropOut`	`batchSize`	**Accuracy**	**Precision**	**Recall**	**F1-score**	# param.
8	16	0.1	128	89.26	89.49	89.26	89.26	114852
8	32	0	128	89.73	89.92	89.73	89.68	122228
10	32	0	256	89.95	90.17	89.95	89.96	124532

Fig. 14: Best performing models on the validation set, 20 MFCC, 20 classes setting.

Hyper-parameters				**Metrics** (weighted average)				
`kernelSize1`	`nFilters4`	`dropOut`	`batchSize`	**Accuracy**	**Precision**	**Recall**	**F1-score**	# param.
8	16	0.1	128	86.76	87.26	86.76	86.77	115013
8	32	0.1	128	86.85	87.15	86.85	86.88	122549

Fig. 15: Best performing models on the validation set, 20 MFCC, 21 classes setting.

simple to efficiently solve the task, which is also proven by the results of the 2D-residual CRNN shown in Tab. 20.

The best results have been achieved in all the cases from the 40 log-Mels, surprisingly followed by the 24 log-Mels with reduced frequency and then from the 20 cepstral coefficients. The smallest selected model discriminates among 10 classes starting from 20 MFCC with an accuracy of 89.4%, while the best one is clearly the CRNN with an accuracy of 92.6% for all the 21 classes, still far from the SOTA of 97.8% reached in [10].

Looking closer at the results, the accuracy matrix of the best performing CNN model for the 21-classes problem in Fig. 19 shows that the most frequently misclassified utterances are the quickest to be pronounced (*go*, *up* and *no*). This is probably due to the dimension of the model and not to the pre-processing approach, given that the issue doesn't appear for the bigger 2D-residual CRNN. Moreover, both models' precision is partially invalidated by the fact that the dataset is unbalanced toward the *other* class.

VII. Concluding Remarks

In this work the Keyword Spotting task has been tackled on different subsets of the Google Speech Commands Dataset V2. Among the diverse tested feature extraction methods, the 40 log-Mels provided a CNN model with less then 120k parameters and an accuracy of 90.75% on the 21-classes task, while the residual CRNN best model, bigger and only trained as a benchmark, achieved 92.56% accuracy in the same scenario. In the 10-classes case the maximum accuracy of 92.62% is also reached thanks to the 40 log-Mels by a CNN model with 127130 parameters.

In both cases the performance is far from the SOTA accuracy of 97.2% for the 10-classes task reached in [7] with an even smaller model, and from the one of 97.8% in [10] for the 21-classes problem, with a bigger model although.

As a student, in fact, the main lesson I learned is the importance of relying on previous literature as the first step, when approaching a new task: I started off on the wrong foot, knowing too little about the task and trying to reach satisfying results from scratch, but clearly failing. Moreover, the apparently simple task of deciding how to pre-process the inputs turned out to be one of the most time consuming, together with the self-imposed challenge of handling the big amount of data and their structure using mostly my own code.

I also learned how a scientific paper should properly be written and how to handle compatibility issues among machines with slightly different Python environments (my laptop, the DEI cluster and Google Colab).

Further improvements could derive from the implementation of the model resulting from the DNAS performed in [7], starting from feature extraction techniques different from the one selected in the paper (MFCCs) to try refining the result based on the insights gained in this work.

Feature	**Hyper-parameters**				**Metrics** (weighted average)				
	`k1`	`f4`	`DO`	`BS`	**Accuracy**	**Precision**	**Recall**	**F1-score**	# param.
24 log-Mels	8	16	0.1	256	90.36	90.89	90.36	90.39	114522
40 log-Mels	[8,32]	16	0	128	92.62	93.09	92.62	92.67	127130
20 MFCC	8	16	0.1	128	89.43	89.73	89.43	89.44	113242

Fig. 16: Selected models' results on the test set, 10 classes setting.

Feature	**Hyper-parameters**				**Metrics** (weighted average)				
	`k1`	`f4`	`DO`	`BS`	**Accuracy**	**Precision**	**Recall**	**F1-score**	# param.
24 log-Mels	8	16	0.1	128	89.92	90.45	89.92	89.90	117418
40 log-Mels	8	32	0.1	128	92.19	92.48	92.19	92.21	128628
20 MFCC	8	16	0.1	128	89.63	89.75	89.63	89.63	114852

Fig. 17: Selected models' results on the test set, 20 classes setting.

Feature	**Hyper-parameters**				**Metrics** (weighted average)				
	`k1`	`f4`	`DO`	`BS`	**Accuracy**	**Precision**	**Recall**	**F1-score**	# param.
24 log-Mels	8	16	0	256	87.18	87.77	87.18	87.19	117701
40 log-Mels	8	16	0	128	90.75	91.01	90.75	90.78	118373
20 MFCC	8	16	0.1	128	86.79	87.15	86.79	86.78	115013

Fig. 18: Selected models' results on the test set, 21 classes setting.

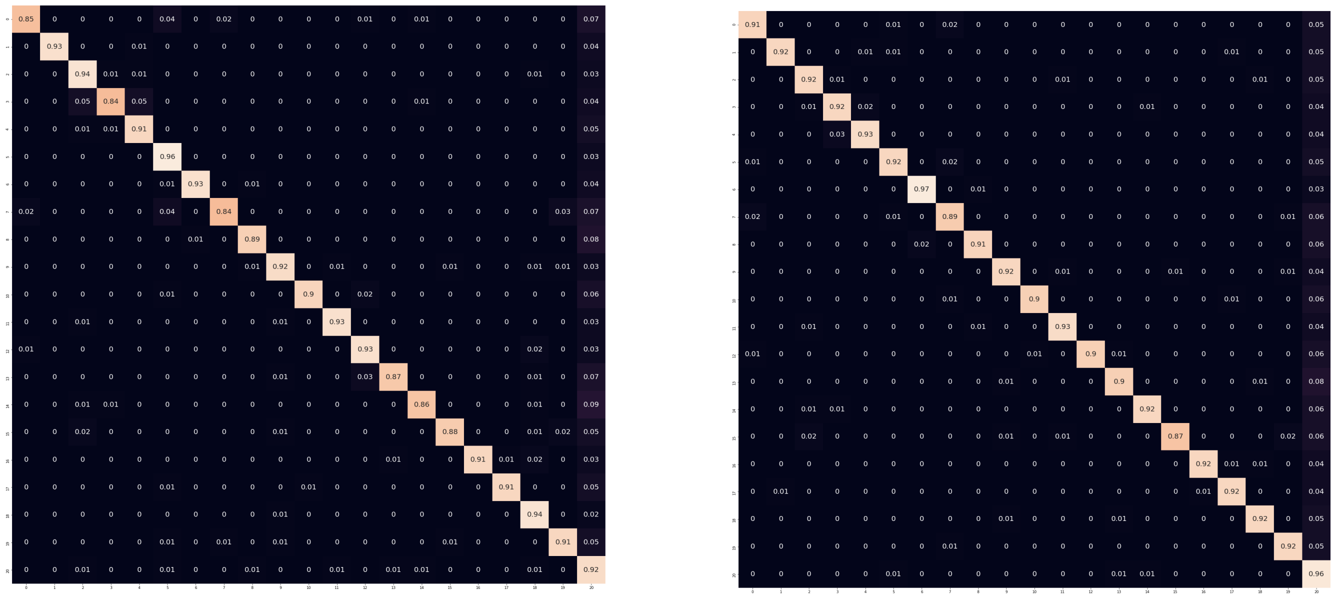

Fig. 19: Accuracy matrix for CNN model on the left, and for the residual CRNN model on the right, in the 40 log-Mels, 21-classes setting.

References

[1] R. C. Rose and D. B. Paul, "A hidden Markov model based keyword recognition system," in *Acoustics, Speech, and Signal Processing, 1990 International Conference on. IEEE*, 1990.

[2] G. Chen, C. Parada, and G. Heigold, "Small-footprint keyword spotting using deep neural networks," in *Acoustics, Speech, and Signal Processing, 2014 International Conference on. IEEE*, 2014.

[3] T. N. Sainath and C. Parada, "Convolutional Neural Networks for Small-footprint Keyword Spotting," in *INTERSPEECH*, (Dresden, Germany), Sept. 2015.

[4] M. Sun, A. Raju, G. Tucker, S. Panchapagesan, G. Fu, A. Mandal, S. Matsoukas, N. Strom, and S. Vitaladevuni, "Maxpooling loss train-

Hyper-parameters				**Metrics** (weighted average)				
`kernelSize1`	`kernelSize3`	`nGRU`	`batchSize`	**Accuracy**	**Precision**	**Recall**	**F1-score**	# param.
6	3	128	128	91.14	91.34	91.14	91.16	377045
8	4	256	128	92.95	93.38	92.95	92.92	619605
[8,16]	4	256	256	92.49	92.67	92.49	92.53	623701
8	4	256	128	92.56	92.79	92.56	92.58	619605
8	4	256	128	93.26	93.34	93.26	93.26	619348 (20)
8	4	256	128	93.32	93.45	93.32	93.32	616778 (10)

Fig. 20: Best performing models on the validation set and results of the best model on the test set, CRNN-res, 40 log-Mels, 21 classes setting. The 20 and 10 classes test results of the same model are reported below for completeness.

ing of long short-term memory networks for small-footprint keyword spotting," in *Spoken Language Technology Workshop (SLT), 2016 IEEE*, 2016.

[5] Y. Zhang, N. Suda, L. Lai, and V. Chandra, "Hello edge: Keyword spotting on microcontrollers," *CoRR*, vol. abs/1711.07128, Nov. 2017.

[6] S. Choi, S. Seo, B. Shin, H. Byun, M. Kersner, B. Kim, D. Kim, and S. Ha, "Temporal convolution for real-time keyword spotting on mobile devices," *CoRR*, vol. abs/1904.03814, 2019.

[7] B. Zhang, W. Li, Q. Li, W. Zhuang, X. Chu, and Y. Wang, "Autokws: Keyword spotting with differentiable architecture search," in *ICASSP 2021 - 2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Sept. 2020.

[8] R. Tang and J. Lin, "Deep residual learning for small-footprint keyword spotting," *CoRR*, vol. abs/1710.10361, Aug. 2017.

[9] P. Warden, "Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition," Apr. 2018.

[10] D. Seo, H.-S. Oh, and Y. Jung, "Wav2kws: Transfer learning from speech representations for keyword spotting," *IEEE Access*, May 2021.