Effective processing pipeline and advanced Neural Network architectures for Small-footprint Keyword Spotting

Daniele Ninni, Nicola Zomer
Department of Physics and Astronomy ”Galileo Galilei”, University of Padua, Italy
daniele.ninni@studenti.unipd.it, nicola.zomer@studenti.unipd.it

***Abstract*—The keyword spotting (KWS) task consists of identifying a relatively small set of keywords in a stream of user utterances. This is preferably addressed using small footprint inference models so that they can be deployed on-device even on performance-limited and/or low-power devices. In this framework, model performance is not the only relevant aspect and the model footprint plays an equally crucial role. In recent years, artificial neural networks have proven particularly effective for the KWS task. In general, they are capable of outperforming most of the previous HMM-based approaches. In this work, we experiment with several neural network architectures as possible approaches for the KWS task. We run our tests on the Google Speech Commands dataset, one of the most popular datasets in the KWS context. We define a CNN model that outperforms our baseline model and we use it to study the impact of different preprocessing, regularization and feature extraction techniques. We see how, for instance, the log Mel-filterbank energy features lead to the best performance and we discover that the introduction of background noise on the train set with a reduction coefficient of 0.5 helps the model to learn. Then, we explore different machine learning models, such as ResNets, RNNs, attention-based RNNs and Conformers in order to achieve an optimal trade-off between accuracy and footprint. We find that this architectures offer between a 30-40% improvement in accuracy compared to the baseline, while reducing up to 10× the number of parameters.**

***Index Terms*—keyword spotting, convolutional neural networks, residual networks, recurrent neural networks, attention mechanism, conformers.**

I. Introduction

In the context of speech processing, the keyword spotting (KWS) task consists of identifying a relatively small set of keywords in a stream of user utterances. Typically, this task is performed by an intelligent agent on a mobile or smart home device. In general, it is preferable that the underlying inference models have a small footprint (for example, measured as the number of model parameters) so that they can be deployed on-device even on performance-limited and/or low-power devices. Furthermore, off-line WKS prevents any potential violation in terms of user privacy.

In recent years, neural networks have proven particularly well suited to the small footprint KWS task. In most cases, the challenge for researchers consists in finding the right trade-off between high detection accuracy and small footprint. In this sense, one of the most frequent approaches consists in implementing simplified variants derived from full speech recognition models.

In this work, we focus on several neural network families that have proven to be effective for the small footprint KWS in the research community. In detail, we experiment with convolutional neural networks (CNN), residual networks (ResNet), recurrent neural networks (RNN) with attention and convolution-augmented transformers (Conformer). We test our approach and implementations on the Google Speech Commands dataset, one of the most popular datasets in the KWS context. Our main contribution consists in tuning the architecture and hyperparameters of the above models to achieve an optimal trade-off between accuracy and footprint.

II. Related Work

The idea of using artificial neural networks for the KWS task is certainly not new. In general, they have proven capable of outperforming most of the previous HMM-based approaches.

Sainath and Parada [1] explored CNNs and achieved significant improvements over previous DNN-based approaches. They specifically cited reduced model footprints as a major motivation in moving to CNNs.

Moreover, Tang and Lin [2] applied deep residual learning and dilated convolutions to KWS and this led to outperform most of the models proposed by [1] in the face of a significantly smaller footprint.

Furthermore, Andrade et al. [3] introduced a convolutional recurrent network with attention for KWS. They demonstrated that the proposed attention mechanism not only improves performance but also allows inspecting what regions of the audio were taken into consideration by the network when outputting a given category.

Finally, Gulati et al. [4] proposed the convolution-augmented transformer for end-to-end speech recognition, named Conformer. They studied the importance of each component, and demonstrated that the inclusion of convolution modules is critical to the performance of the Conformer model. Moreover, they showed that the Conformer significantly outperforms the previous Transformer and CNN based models achieving state-of-the-art accuracies.

III. Processing Pipeline

First of all, the dataset is loaded using the `SciPy` library. Then, the audio signals enter the pre-processing block, which consists of 3 steps: padding and trimming, data augmentation and feature extraction.

In the first step the audio clips are reshaped, i.e. padding or trimming is performed to convert the input waveform into a one-second long signal. In the training phase, at this point the dataset is cached via the TensorFlow API [5].

After this step, some data augmentation techniques can be included. In particular, we consider the possibility of adding a background noise signal and/or performing a time shift of a maximum length of ±100ms. Finally, the signal thus obtained is converted into a two-dimensional feature, which will be given as input to a neural network model. This conversion starts for all features with a segmentation of the signal into short frames. Motivated by [6], we select a 25ms time window with 10ms overlap. At this point, the signal is assumed to be quasi-stationary within each frame, and the processing pipeline is applied to each frame independently. This stage is very delicate and strongly depends on the type of features to be extracted. Since our work is mainly focused on the log Mel-filterbank energy features, in this section we describe only this data processing technique. In this case, for each frame, the power spectrum is computed and 40 Mel-spaced filters are applied to it. Finally, 40 energy values are extracted, and their logarithm is computed. By applying this procedure to each frame and stacking the resulting coefficients horizontally, our feature vector then has shape $(99, 40)$.

The complete processing pipeline with the log energy features can be visualized in Fig. 1.

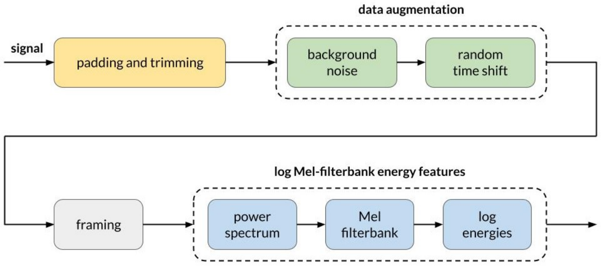


Fig. 1: Processing pipeline.

IV. Signals and Features

A. Speech Commands dataset

The Speech Commands dataset (v0.02) consists of 105,829 short (one-second or less long) audio clips of English commands, stored as audio files in the .wav format [7]. The dataset contains 35 commands in total, spoken by thousands of different people. It was collected using crowdsourcing, through AIY by Google and released under a CC BY 4.0 license.

B. Acquisition

The data was captured in uncontrolled locations and in a variety of formats, and then converted to a 16-bit little-endian PCM-encoded WAVE file at a 16000 sample rate. The audio was then trimmed to a one second length to align most utterances, using the `extract_loudest_section` tool. The audio files were then screened for silence or incorrect words, and arranged into folders based on the utterance they contain.

C. Padding and trimming

Audio clips are 1 second or less long at 16kHz, meaning that not all of them are shaped the same. Therefore, we pad the short ones to exactly 1 second (and eventually trim the longer ones) so that all samples can be processed by the same neural network, which of course requires a fixed input shape. Most of the clips have 16000 samples (1 second at 16kHz), so we use this value as fixed input shape. In particular:

- **padding**: performed by `numpy.pad` using the mean value of the signal, half on the left and half on the right;
- **trimming**: performed by simply cropping the signal to $[0, 15999]$ instead of randomly selecting a portion of it.

In this way, the input data consists of vectors all having the shape $(1, 16000)$. An example of a padded signal can be seen in Fig. 3.

D. Background noise

The `_background_noise_` folder contains a set of six longer audio clips that are either recordings or mathematical simulations of noise. To help train machine learning models to cope with noisy environments, it can be helpful to mix in realistic background audio. Therefore, we define the function `background_noise` which mixes one of these background audio signals into an input audio clip. With the argument `select_noise` it is possible to select which of the available background audio clips to use. If `select_noise` = `None` (default value), the background noise is randomly chosen between the available ones. Moreover, as these clips still have a 16kHz sample rate but are longer than 1 second, we randomly select a portion of them to match the size of the input, called s_η. We also include a noise reduction coefficient $\alpha \in [0, 1]$ with default value 0.5 such that the output signal s_{out} is given by $s_{in} + \alpha \cdot s_\eta$. Notice that $\alpha = 1$ corresponds to zero noise applied, while $\alpha = 0$ if the noisy signal is not reduced. Finally, we decide to add the background noise to each sample with a probability of 0.8 at every epoch.

E. Random time shift

Adding a random time-shift of Y milliseconds to the input data is a technique that can be found in many works on this dataset (for example [2]), motivated by the fact that in a real-time keyword detector the input signal may be not centered in the selected time window. Following their approach, we implement a function that perform this operation, where $Y \sim U(-100, 100)$. We then use padding with the average value to bring the signal back to the required size. Later we will see that this strategy does not lead to better results, and therefore will be discarded.

F. Feature extraction

The waveforms in the pre-processed dataset are represented in the time domain. We use `tf.signal.stft` by TensorFlow [5] or the `python_speech_features` [8] library to extract relevant features from them. These feature vectors form the input of the neural network models we explore. In

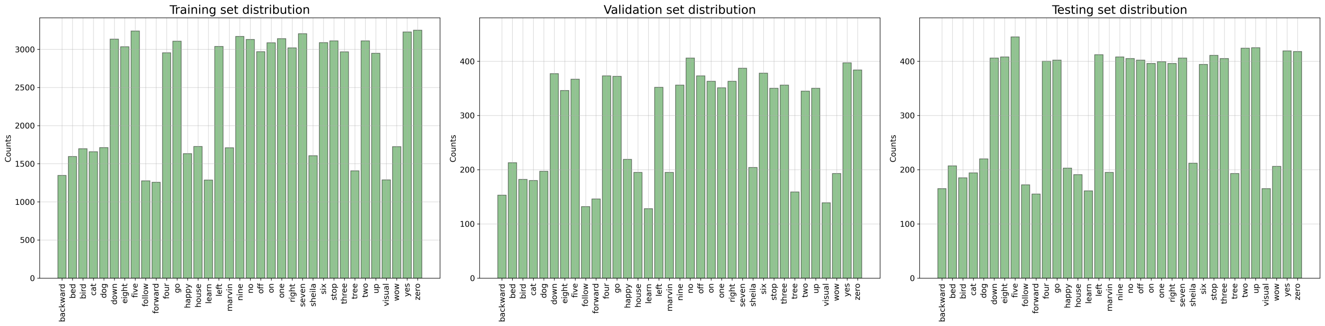


Fig. 2: Training, validation and test set distributions.

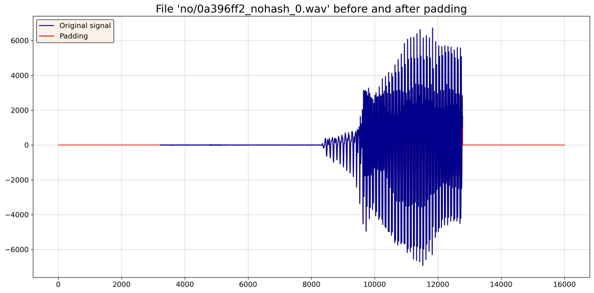


Fig. 3: Padding example.

detail, we implement several utility functions for extracting the following audio features:

- spectrogram;
- log Mel-filterbank energy features;
- MFCC features (with or without Delta features);
- discrete wavelet transform plus MFCC (with Delta).

By default, the 2D feature vectors are computed by framing the input audio signal into overlapping analysis windows of length 25ms and with a step of 10ms. This is the same approach implemented by Arik et al. [6]. Moreover, for the last three features we use 40 filters in the filterbank.

G. Partitioning: training, validation and test sets

The text files `validation_list.txt` and `testing_list.txt` (included in the Speech Commands dataset) contain the paths to all the files in each set, with each path on a new line. Any files that are not in either of these lists can be considered to be part of the training set. For the sake of comparison we remain consistent with this partition. The validation and test set sizes thus obtained are approximately 10% of the size of the complete dataset. In detail:

- **training set**: 80.17% (84843 samples)
- **validation set**: 9.43% (9981 samples)
- **test set**: 10.40% (11005 samples)

In Fig. 2 we report the distribution of each set. Notice how the classes are unbalanced.

V. Neural network architectures

The end-to-end model architecture consists of a speech feature extractor, whose pipeline is the one shown in Fig. 1, and a neural-network based classifier.

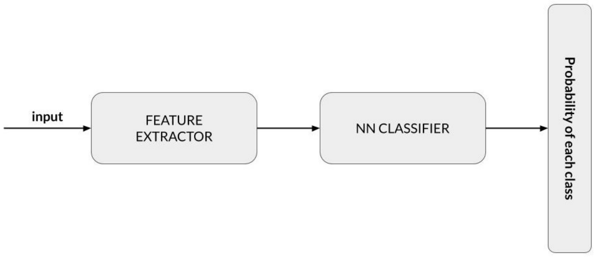


Fig. 4: Classification pipeline.

The last layer of each architecture is characterized by the softmax activation function, so that our neural network models produce the probabilities that the input belongs to each of the output classes. In general, we use the sparse categorical cross-entropy loss function with Adam optimizer for model training. Moreover, in the training procedure we use the Keras callbacks `EarlyStopping` and `ReduceLROnPlateau`, monitoring the validation loss.

A. Baseline model

We use as baseline model the CNN model `cnn-one-fpool3` presented in [6]. We train it with (99×40) log Mel-filterbank energies features vector. With this input data the model has 2.26M parameters and reaches an accuracy of 53.0% on the test set. We also implement the models `nn-trad-fpool3`, `cnn-one-fstride4` and `cnn-one-fstride8` presented in [6], but we decide to use only the first as a reference.

B. CNN

Our purpose with this architecture is to explore more complex and recent ideas in order to get an improvement over the baseline, without increasing the footprint. We test various CNN models characterized by a different number of CNN blocks, each of them consisting of a sub-block made of a convolutional layer followed by batch normalization and ReLU activation function. A number of such sub-blocks can be stacked and then followed, in order, by max pooling and droupout layers to form the final block. We also implement ideas from [6] such as striding and pooling in time. Furthermore, we perform a Bayesian optimization to tune the hyperparameter of a CNN model we named `custom_cnn_simple`. This model is characterized by the 4 previously described blocks, each consisting of a single convolutional sub-block. The first 2 blocks end with a balanced (s, s) max pooling layer, while in the last two we only pool over time. Its optimized structure is presented in Table 1, where $(m \times r)$ is the filter size, n the number of filters, $(p \times q)$ the pooling subsampling, and q is the time dimension. Each batch normalization layer in the convolutional blocks is followed by a ReLU activation function.

Blocks	Type	m	r	n	p	q	Params
-	batchnorm	-	-	-	-	-	4
block 1	conv	8	3	48	2	2	1.20K
	batchnorm	-	-	-	-	-	192
block 2	conv	8	3	32	2	2	36.9K
	batchnorm	-	-	-	-	-	128
block 3	conv	5	5	128	1	3	102.5K
	batchnorm	-	-	-	-	-	512
block 4	conv	5	5	64	1	3	204.9K
	batchnorm	-	-	-	-	-	256
-	linear	-	-	256	-	-	1311.0K
	batchnorm	-	-	-	-	-	1024
-	linear	-	-	35	-	-	9.0K
Total	-	-	-	-	-	-	1.667M

TABLE 1: Optimized `custom_cnn_simple` architecture.

This model is also used to explore the effect of dropout, batch normalization, background noise and random time shift. Last, we train the model `custom_cnn_simple` with different input audio features to perform a feature comparison.

C. ResNet

First, we implement and benchmark a model we named `resnet`. It is obtained as a slightly modified version of the `res8` ResNet architecture, presented in [2]. In detail, we introduce batch normalization as first layer, we use a kernel size equal to 5 in the convolutions, we replace the 4x3 average pooling with a 2x1 max pooling over time, we keep a convolutional layer after the residual blocks as in `res15`, setting a dilation rate equal to 8 in it, and, finally, we add dropout before the final dense layer. Then, motivated by [9], we modify the final layers of such model and the training algorithm to get a meaningful embedded representation of the input signals. We use k-NN to perform the classification task on these intermediate representations. This work is entirely based on modifying the loss function and using the Triplet Semi-Hard Loss instead of the sparse categorical cross-entropy. The last layer of this network has 256 neurons, no activation function and it is followed by L2 normalization.

D. CRNN with attention

Attention models are powerful tools to improve performance on natural language, image captioning and speech tasks. We define a neural attention model obtained from a a slight modification of the architecture presented in [3] and available in the corresponding repository. The only differences are how the pre-processing pipeline is implemented and the addition of a dropout layer after the convolution part and between the final dense layers. We test both the standard RNN and the recurrent attention model, with the idea of obtaining high performances while keeping the footprint small (less than 200k parameters).

E. Conformer

The Conformer model combines self-attention and convolution to achieve the best of both worlds, i.e. self-attention learns the global features whilst the convolutions efficiently capture the local correlations. Its architecture is shown in Fig. 5.

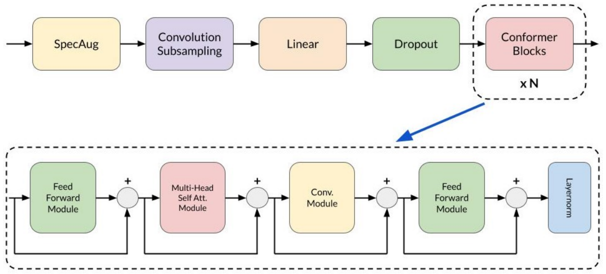


Fig. 5: Conformer encoder model architecture.

The Conformer encoder first processes the input with a convolution subsampling layer and then with a number of Conformer blocks. A Conformer block comprises of two Feed-Forward modules with half-step residual connections sandwiching the Multi-Headed Self-Attention module and the Convolution module. This is followed by a Layer normalization layer.

Thanks to the `audio_classification_models` library [10], we implement the model `conf_base`, i.e. a baseline Conformer architecture inspired by [4]. In detail, we use only one Conformer block in order to reduce the number of model parameters. Moreover, we perform hyperparameter tuning by means of Bayesian optimization in order to find, among the models with less than 2M parameters, the one that leads to the best accuracy.

VI. Learning Framework

In this section we describe some particular algorithms and learning strategies that have been implemented in the pre-processing stages or in the neural network architectures mentioned above.

A. Bayesian optimization

To tune the hyperparameters of the CNN and the Conformer model we exploit Bayesian optimization. Bayesian optimization is a strategy to find the optimum of a black-box function, whose analytic form is not known and which is expensive to evaluate. The optimization is achieved through a sequential data driven scenario, where the goal is to improve the guess of the optimum while being data efficient in terms of how many queries we ask of the black-box. We leave to the scikit-learn reference [11] and to Snoek et al. [12] for more technical information.

B. Triplet loss

Triplet loss is a loss function characterized by the fact that a reference input (anchor) is compared to a matching input (positive) and a non-matching input (negative). The model parameters are updates such that the distance between the anchor and the positive is minimized while the distance between the anchor and the negative is maximized.

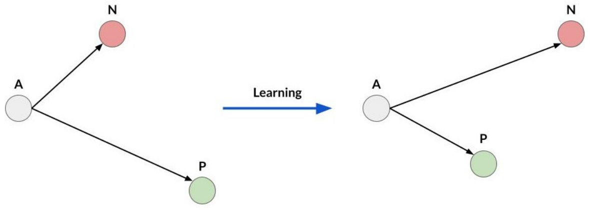


Fig. 6: Triplet loss learning.

The loss function can be described in terms of the Euclidean distance as:

$$\mathcal{L} = max\left(|f(A) - f(P)|^2 - |f(A) - f(N)|^2 + \alpha, 0\right), \quad (1)$$

where α is some margin that has the role of stopping the adjustment of the weights. The use of Semi-Hard online learning is motivated by [13], where they show that the best results are obtained when N is farther from A than P, but still produces a positive loss. Only the Semi-Hard examples in each batch are used for training. Using the loss function `TripletSemiHardLoss` from TensorFlow the implementation is straightforward.

C. Testing metrics

We mainly rely on accuracy as a comparison metric. Although it is not the best metric given the unbalanced dataset, it is probably the most used in reference papers on speech recognition, so we choose it for comparison purposes. However, for each model, we also compute precision, recall, F1-score, cross-entropy value and the Cohen's Kappa coefficient. Such values can be inspected in the notebooks in the GitHub repository of the report [14]. To account for label imbalance, precision, recall and F1-score are computed with the *weighted average*, i.e. the average of the metric values for individual classes weighted by the support (i.e. the number of true instances) of that class. See [15] for more information on these metrics in the context of multi-class classification.

VII. Results

A. Pushing the performance of CNN

Many studies on speech recognition (for example [6] and [2]) are aimed at finding models and strategies that allow to obtain good accuracy while keeping the footprint small. As a first result, we want to show how CNN performance can be boosted while keeping the number of parameters below 2M. We perform a Bayesian optimization of the model `custom_cnn_simple`, training the model using the log Mel-filterbank energy features for 30 calls, and for 20 epochs for each call. For more information about the search space see the corresponding notebook in [14]. The resulting parameters are then slightly modified to use meaningful values. The final details of each block are reported in Table 2.

In addition to the parameters presented there, we get that the best dropout amount is 0.3 after each convolutional block and 0.2 before the fully connected layers. Finally, we get an optimized learning rate of 0.01. The resulting model architecture has 1.67M parameters and can be visualized in Fig. 7.

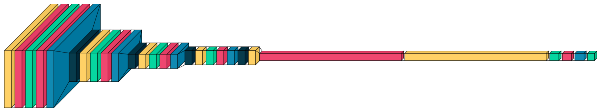


Fig. 7: `custom_cnn_simple` optimized.

Next, we create a modified version of such model, called `custom_cnn`. Here the blocks where pooling is applied in both directions are made up of 3 convolution sub-blocks each. We also add one more dense layer at the end and slightly change the kernel sizes. The resulting model is presented in table 2 and has 1.40M parameters.

Blocks	Type	m	r	n	p	q	Params
-	batchnorm	-	-	-	-	-	4
block 1 (3x conv)	conv + bn	7	3	48	-	-	-
	max pool	-	-	-	2	2	-
	total B1	-	3	48	-	-	98.5K
block 2 (3x conv)	conv + bn	7	3	32	-	-	-
	max pool	-	-	-	2	2	-
	total B2	-	3	48	-	-	75.7K
block 3 (1x conv)	conv	5	7	128	1	3	143.5K
	batchnorm	-	-	-	-	-	512
block 4 (1x conv)	conv	5	7	64	1	3	286.8K
	batchnorm	-	-	-	-	-	256
-	linear	-	-	512	-	-	655.9K
	batchnorm	-	-	-	-	-	2048
-	linear	-	-	256	-	-	131.3K
	batchnorm	-	-	-	-	-	1024
-	linear	-	-	35	-	-	9.0K
Total	-	-	-	-	-	-	1.404M

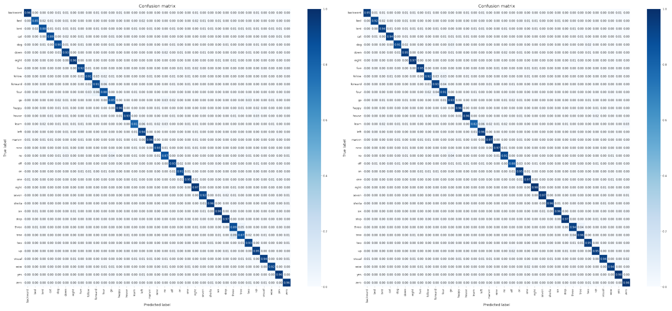
TABLE 2: CNN architecture for `custom_cnn`.

The performances of both models on the test set are presented in Table 3 and their confusion matrix is reported in Fig. 8.

Model	Parameters	Accuracy
`custom_cnn_simple`	1.67M	92.4
`custom_cnn`	1.40M	94.6

TABLE 3: CNN models performances.

Notice how increasing the number of CNN sub-blocks in the first stages helps improving the performances without affecting the number of parameters. Moreover, these CNN models show improvements of almost 40% compared to our baseline model, while keeping a relatively small footprint.



(a) `custom_cnn_simple` (b) `custom_cnn`

Fig. 8: Confusion matrix of CNN models.

B. *Studying the effect of dropout and batch normalization*

In [16] they show how dropout can be an effective strategy to prevent neural networks from overfitting. Moreover, another technique that has been shown in [17] to be really successful in improving not only stability but also learning speed is batch normalization. In Tab. 4 we present a comparison obtained by training the model `custom_cnn_simple` without the dropout or the batch normalization layers.

	Training	Validation	Testing
Default	94.8	92.8	92.4
No Dropout	97.3	92.5	91.9
No BatchNorm	3.8	3.9	3.8

TABLE 4: Dropout and batch normalization effect on the accuracy of `custom_cnn_simple`.

First, we can see that the use of dropout layers appears to be effective also in the context of speech commands recognition. Notice then that the performance drops completely without any normalization strategy. This motivates our choice of introducing batch normalization as first layer also in most of the subsequent models, thus avoiding to normalize the entire dataset a priori.

C. *Features comparison*

We train the model `custom_cnn_simple` for 20 epochs with different input audio features to perform a feature comparison. In detail, we considered the following input features:

- spectrogram;
- MFCC with Delta features;
- MFCC without Delta features;
- log Mel-filterbank energies;
- discrete wavelet transform plus MFCC (with Delta).

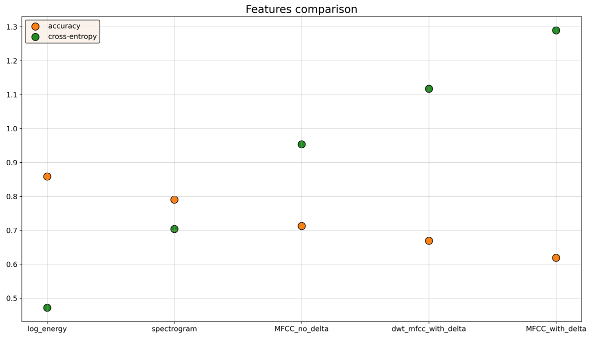


Fig. 9: Features comparison for CNN.

As shown in Fig. 9, the best results on the test set are achieved using the logarithm of the raw filterbank energies, with a relative performance improvement of more than 25% on the worst feature. It could be argued that this is due to the fact that the Bayesian optimization was done using this feature, and therefore the reason for this result is precisely that the model we have selected for the comparison is optimized for the log energies. However, from some other tests we did, on unbiased models, we got the same result shown here. From now on, we will focus our research on this feature, unless otherwise specified.

D. *Investigating the impact of data augmentation*

It is interesting to study the effect of adding background noise. To do this, we train again the model `custom_cnn_simple` applying different noise levels on the train set, while keeping noise-free the validation and test sets. For each value of α, we train the model `custom_cnn_simple` for 20 epochs.

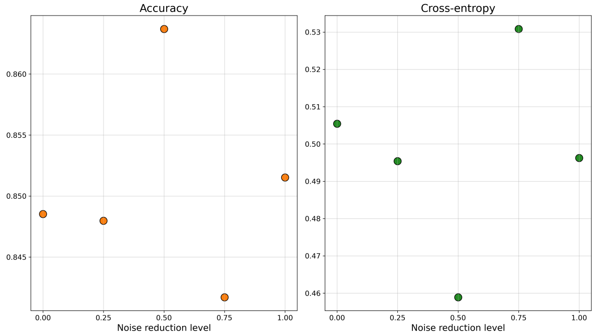


Fig. 10: Background noise effect on testing performance.

From the resulting metrics, we can see that the best performance is achieved with a noise reduction level of 0.5. This explains our choice of setting a default value of α =0.5 in the pre-processing pipeline.

We also want to see if shifting the audio signals in the train set by a random amount in $[-100, 100]$ms helps improving the model performance. Contrary to expectations, the model accuracy after 20 epochs of training decreases from 85.5% to 80.4% when including it. We are not sure at the moment if this is due to a slowdown in training of if the performance actually drops, for example because portions of the training data containing information useful to the model are cropped each time. We leave this as an object of future studies, for now eliminating this technique from our pipeline.

E. Mitigating the footprint using skip connections

Next, we want to show how skip connections and, therefore, residual network architectures can lead to a strong decrease of the footprint, maintaining a high detection accuracy. First, we train the model `resnet` for 40 epochs using the log Mel-filterbank energy features, the loss function `SparseCategoricalCrossentropy` and a learning rate of 0.01.

Then, we train a modified version of such model, called `resnet_triplet_loss`. Its purpose is not to directly classify the input data, but to obtain an interesting triplet loss based embedded representation of it. For this reason the training is performed using the loss function `TripletSemiHardLoss` instead of the sparse categorical cross-entropy.

The performance of both models on the test set are presented in Table 5.

Model	Parameters	Accuracy
`resnet`	359K	94.1
`resnet_triplet_loss`	1.74M	89.3

TABLE 5: ResNet models performances.

Notice that the model `resnet` leads to an accuracy comparable to that achieved by `custom_cnn`, but with the great advantage of having a number of parameters almost 4 times smaller. Instead, in this case, the triplet loss based approach not only leads to worse accuracy, but also to a larger model footprint. Although in [9] they show how a careful fine-tuning of this model and of the training triplets can improve the performance of a standard ResNet by 3-4 %, the sacrifice is surely in the footprint and in the complexity of the training.

Finally, we select 4900 audio signals from the test dataset (140 samples per class) and use `projector.tensorflow` to visualize their embedded representation, obtained via the `resnet_triplet_loss` model. The visualization is made using the UMAP algorithm, setting the number of neighbors to 20 and selecting a 2D representation and is shown in Fig. 11.

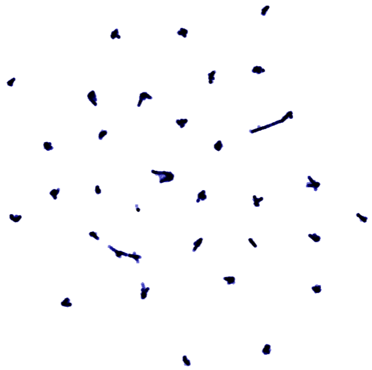


Fig. 11: Embedded representation, 2D view.

F. CRNN with attention

First, we train the standard RNN model called `RNNSpeechModel`. In this case we make some different choices regarding the optimization, motivated by [3]. First, as optimizer, we use Adam with an initial learning rate $l_0 = 10^{-3}$. Then, thanks to `LearningRateScheduler` from TensorFlow, we are able to schedule a learning rate decay according to the following function:

$$l(n) = l_0 \cdot 0.4^{\text{floor}[(1+n)/15]},$$

where n is the epoch and the lower bound for $l(n)$ is:

$$l_{\text{min}} = 4 \cdot 10^{-5}.$$

Secondly, we train a slightly modified version of such model, called `AttRNNSpeechModel`, which embodies the attention mechanism.

Since [18] and [3] already showed that attention-based RNN can lead to a good performance improvement in the context of speech recognition, we want to force such models by halving the footprint compared to our ResNet architectures. The performance of both models, together with the number of parameters, are presented in Table 6.

Model	Parameters	Accuracy
`RNNSpeechModel`	164K	87.7
`AttRNNSpeechModel`	180K	87.8

TABLE 6: Att-CRNN models performances.

We can observe that, in this case, the introduction of the attention mechanism does not lead to a significant improvement in terms of accuracy. Notice, however, how with RNN we are still able to achieve an improvement of approximately 30% with respect to our baseline, using ∼10x less parameters.

G. Conformer

First, we train the model `conf_base` for 100 epochs. As optimizer, we use Adam with a learning rate of 10^{-4} and, inspired by [4], we set $\beta_1 = 0.9$, $\beta_2 = 0.98$ and $\epsilon = 10^{-9}$.

Then, we perform a Bayesian optimization of the model `conf_base` for 20 calls and setting a maximum model size of 2M. The search space is the following:

- n_{sub}: number of filters (conv. subsampling)
- d_{sub}: kernel size (conv. subsampling)
- s_{sub}: stride (conv. subsampling)
- n_{head}: number of heads (self-attention)
- d_{head}: head size (self-attention)

- d_{conf}: kernel size (conv. module in conformer block)
- p: dropout (before and in conformer block)

The resulting model is called conf_opt. The parameter values of both models are reported in Table 7.

Model	n_{sub}	d_{sub}	s_{sub}	n_{head}	d_{head}	d_{conf}	p
conf_base	144	3	2	4	36	32	0.1
conf_opt	144	4	1	6	26	32	0.1

TABLE 7: Bayesian optimization results for Conformer.

The performance of both models on the test set are presented in Table 8.

Model	Parameters	Accuracy
conf_base	908K	88.4
conf_opt	1.68M	90.6

TABLE 8: Performance of Conformer models.

Notice how the Bayesian optimization leads to a slight performance improvement at the cost of nearly double the number of parameters. However, conf_opt performs worse than both custom_cnn_simple and custom_cnn, despite the fact that the latter two have fewer parameters.

VIII. Concluding Remarks

In this project we tackled the keyword spotting task focusing on the accuracy-footprint trade-off and comparing recent architectures with more traditional ones. We started by pushing the performance of CNNs and analyzing the impact of different pre-processing, regularization and feature extraction techniques. We then moved towards more sophisticated models, with the aim of slightly sacrificing the accuracy to reduce the footprint as much as possible. We have seen how the introduction of skip connections is definitely successful, while more advanced strategies require strong fine-tuning to lead to effective results. However, we believe it is worth continuing to research in this direction, as we saw these models make it possible to significantly reduce the number of parameters while negligibly sacrificing the performance.

Regarding the first part on the analysis of different possible pipelines, we believe this work represents a relevant contribution in this field, especially as regards its completeness. In the future it could certainly be replicated in more advanced models than CNNs. Furthermore, it would be interesting to replicate this analysis by carrying out different trials for each training with different random seed, in order to add statistical information to the results. In particular, in could be interesting to add a 95% confidence interval to them. Given the amount of papers that implement it, we believe it may be necessary to perform a more in-depth analysis on the introduction of the random time shift on the input signals in the train set. We also believe that our CNN model can be a good starting point for a more in-depth study on CNNs and ResNets in the context of speech commands recognition. For instance, it remains to be studied in detail how a different number of CNN blocks or convolutional sub-blocks impacts on accuracy and footprint.

Moving on to the second part, our most important contribution is maybe the confirmation that the use of RNN models, attention-based architectures and transformers can be fundamental for drastically and intelligently reducing the footprint. This would come in handy especially when the keyword spotting task is performed on-device rather than in the cloud. In this framework the detection of predefined keywords is carried out on a mobile phone or a smart home device, both characterized by low-power and performance limits.

IX. Extra: project conclusions

A. What we have learned

First, we learned that the accuracy (or in general the performance) is not the only relevant metrics. We already had the opportunity to understand this during the course, but from this project we really have understood and internalized how practical aspects, such as the model footprint, play an equally important role. From the technical side we want to mention for example the concept of triplet loss, the Cohen's Kappa coefficient and in general how to compute testing metrics in a multi-class task, the implementation of an efficient, flexible and reusable pre-processing pipeline, the creation of a demo application in Python, some neural network architectures such as the Conformer. But there are really many other concepts and tools we could mention. Moreover, perhaps most importantly, we had to read a lot of papers to do this work, and we believe that having gained a lot of confidence in this task is an important soft skill we acquired. Finally, being a project with an infinite number of possible developments, we understood how important is to define a plan when tackling such complex task. There are many papers out there, and certainly looking for some papers and trying to implement them without knowledge of the facts and without clear goals does not lead to the development of useful and interesting results.

B. Difficulties encountered

We believe that the greatest difficulty was understanding how to organize the work and which paths to take. We had so many ideas, models that we wanted to implement and test, but clearly we had to make choices and it was not easy. On the technical side, we believe the main problem has sometimes been to implement architectures that are not clearly described in their reference papers, or with few reference about them. Again, sometimes models had very high intrinsic complexity and required heavy fine-tuning to work, or were not so intuitive to understand.

References

[1] T. Sainath and C. Parada, "Convolutional neural networks for small-footprint keyword spotting," in *Interspeech*, 2015.

[2] R. Tang and J. Lin, "Deep residual learning for small-footprint keyword spotting," 2017. [Online]. Available: https://arxiv.org/abs/1710.10361

[3] D. C. de Andrade, S. Leo, M. L. D. S. Viana, and C. Bernkopf, “A neural attention model for speech command recognition,” 2018. [Online]. Available: https://arxiv.org/abs/1808.08929

[4] A. Gulati, J. Qin, C.-C. Chiu, N. Parmar, Y. Zhang, J. Yu, W. Han, S. Wang, Z. Zhang, Y. Wu, and R. Pang, “Conformer: Convolution-augmented transformer for speech recognition,” 2020. [Online]. Available: https://arxiv.org/abs/2005.08100

[5] Tensorflow documentation, https://www.tensorflow.org/api_docs.

[6] S. O. Arik, M. Kliegl, R. Child, J. Hestness, A. Gibiansky, C. Fougner, R. Prenger, and A. Coates, “Convolutional recurrent neural networks for small-footprint keyword spotting,” 2017. [Online]. Available: https://arxiv.org/abs/1703.05390

[7] Speech Commands Dataset, https://www.tensorflow.org/datasets/catalog/speech_commands.

[8] J. Lyons, D. Y.-B. Wang, Gianluca, H. Shteingart, E. Mavrinac, Y. Gaurkar, W. Watcharawisetkul, S. Birch, L. Zhihe, J. Holzl, J. Lesinskis, H. Almer, C. Lord, and A. Stark, “jameslyons/python_speech_features: release v0.6.1,” Jan. 2020. [Online]. Available: https://doi.org/10.5281/zenodo.3607820

[9] R. Vygon and N. Mikhaylovskiy, “Learning efficient representations for keyword spotting with triplet loss,” in *Speech and Computer*. Springer International Publishing, 2021, pp. 773–785. [Online]. Available: https://doi.org/10.1007%2F978-3-030-87802-3_69

[10] Audio Classification Models library, https://github.com/awsaf49/audio_classification_models.

[11] Scikit-optimize documentation, https://scikit-optimize.github.io/stable/.

[12] J. Snoek, H. Larochelle, and R. P. Adams, “Practical bayesian optimization of machine learning algorithms,” 2012. [Online]. Available: https://arxiv.org/abs/1206.2944

[13] F. Schroff, D. Kalenichenko, and J. Philbin, “FaceNet: A unified embedding for face recognition and clustering,” in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, jun 2015. [Online]. Available: https://doi.org/10.1109%2Fcvpr.2015.7298682

[14] GitHub repository of the report, https://github.com/NicolaZomer/Keyword_Spotting.

[15] M. Grandini, E. Bagli, and G. Visani, “Metrics for multi-class classification: an overview,” 2020. [Online]. Available: https://arxiv.org/abs/2008.05756

[16] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: A simple way to prevent neural networks from overfitting,” *Journal of Machine Learning Research*, vol. 15, no. 56, pp. 1929–1958, 2014. [Online]. Available: http://jmlr.org/papers/v15/srivastava14a.html

[17] S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift,” 2015. [Online]. Available: https://arxiv.org/abs/1502.03167

[18] J. Chorowski, D. Bahdanau, D. Serdyuk, K. Cho, and Y. Bengio, “Attention-based models for speech recognition,” 2015. [Online]. Available: https://arxiv.org/abs/1506.07503

[19] P. Warden, “Speech commands: A dataset for limited-vocabulary speech recognition,” 2018. [Online]. Available: https://arxiv.org/abs/1804.03209

[20] G. Alon, “Key-word spotting-the base technology for speech analytics.”

[21] C.-H. Li, S.-L. Wu, C.-L. Liu, and H.-y. Lee, “Spoken squad: A study of mitigating the impact of speech recognition errors on listening comprehension,” 2018. [Online]. Available: https://arxiv.org/abs/1804.00320

[22] B. Kim, S. Chang, J. Lee, and D. Sung, “Broadcasted residual learning for efficient keyword spotting,” 2021. [Online]. Available: https://arxiv.org/abs/2106.04140

[23] L. Dong, S. Xu, and B. Xu, “Speech-transformer: A no-recurrence sequence-to-sequence model for speech recognition,” in *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2018, pp. 5884–5888. [Online]. Available: https://ieeexplore.ieee.org/document/8462506