Urban Sound Classification

Statistical methods for machine learning
experimental project report

Francesco Tomaselli
969853

University of Milan
Master's Degree in Computer Science
Academic year 2020/2021

Contents

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study.

1 Introduction

The goal of this project is to build a neural network to classify audio files from the *UrbanSound8k* dataset. [**?**]

This dataset contains audio divided in ten classes, each one representing a different type of city sound, for instance, we can find *car horns*, *dogs barking*, *sirens*, etc. A deeper discussion about the dataset is made at Subsection 2.1 on the following page.

The presented methodology is composed of three main parts. The first step is to extract relevant features from audio files, this is discussed on Section 2 on the next page.
The next step consists in composing and refining a neural network to classify the data obtained from the feature extraction phase. This part is discussed in Section 3 on page 6.
Lastly, results from the classification, namely accuracy and standard deviation among test sets, and possible future works are presented in Section 4 on page 11.

Project structure The project folder is structured as follows:

- *src*: this contains the source code for the project. Each sub-folder contains code for a specific part of the processing. In particular, the *data* folder holds classes to extract features and to manage the dataset, the *model* folder contains the class to create the neural network, then *utils* stores utility functions to measure performances;
- *data*: here data is stored, there is a *processed* and *raw* sub-folders, where the first stores computed datasets and the latter original data;
- *models*: trained models are saved here;
- *notebooks*: the folder contains the Jupyter notebooks used in the project, where code from src is executed.

The code is written in Python.

2 Feature extraction

This Sections presents the original dataset structure and the steps followed to create training and test sets from it.

Note that the models mentioned in this section are three layers *multilayer perceptron*, a feed forward neural network where each layer is densely connected to the following, with a reasonable number of neurons, trained for 100 epochs with default parameters for the *stochastic gradient descent optimizer*. [**?**][**?**]
The accuracy on the training is computed with a *cross-validation* approach. [**?**]
A deeper discussion about the models structure as well as the validation techniques used in the project can be found at Section 3 on page 6.

2.1 Dataset structure

The UrbanSound8k dataset contains ten folds of audio samples, each one about four seconds long. The samples are divided in ten classes.
From the total of ten folds, the number one, two, three, four and six are taken as a training set, the others are each one a test set. For this reason the following count about class numerosity considers only the five training folds.

Class name	Number of samples
air conditioner	500
car horn	208
children playing	500
dog bark	500
drilling	500
engine idling	517
gun shot	190
jackhammer	548
siren	536
street music	500

The table shows a clear class imbalance. In particular, the classes *car horn* and *gun shot* are not as numerous as the others. This can lead to poor performances on the these two categories, it is therefore taken into consideration with training.

The following table shows the number of samples in the training set and the various test sets.

Dataset	Number of samples
Training set	4499
Test set 5	936
Test set 7	838
Test set 8	806
Test set 9	816
Test set 10	837

All the operations on the datasets are performed with *Pandas* library. [**?**]

2.2 First dataset

Extracting features from audio files is not straightforward, nonetheless there are a collection of audio characteristics that are commonly used in audio machine learning applications. [**?**]

To extract information from audio files *Librosa* is used. [**?**] The library provides many methods to choose from, to keep it simple, for the first try with this dataset, the extracted features are these three ones:

1. *Mel-frequency cepstral coefficients*;
2. *Chromagram*;
3. *Root-mean-square*.

Each feature consists of an array of arrays containing measurements. A series of functions are applied to each sub-array and results are concatenated in a final feature vector. The functions applied are *minimum*, *maximum*, *mean* and *median* from the *Numpy* library. [**?**]

This approach results in 132 components feature vectors.

Parallelizing feature extraction Extracting the three features listed above is really intensive but the task is easily parallelizable, in fact, each file is independent from one another.

For this purpose *Dask* is used to speed up the computation and extract features from audio files in a multi-processing fashion. [**?**] The main idea is to build an execution plan, where each audio file is managed in parallel by a collection of workers. Improvement is great as the time to process a single fold is cutted into a third.

Feature scaling After testing some neural networks on the first dataset results are not promising. One of the reasons is the big difference in ranges among feature vector components, for instance, some audio characteristics are in the order of thousands while others range between zero and one.

To mitigate this effect a *StandardScaler* from *scikit learn* is applied. [**?**] The result is a dataset where each feature has more or less a distribution centered in zero with unit variance. This leads to an improvement on the results using the same model as before.

2.3 Extended dataset

Results using the three features named in the previous Subsection are promising but not enough, thus, to improve accuracy on the training set, new audio characteristics are extracted, namely:

1. *Zero-crossing rate*;
2. *Roll-off frequency*;
3. *Spectral flux onset strength.*

This time *standard deviation* is added to the previous functions *minimum*, *maximum*, *mean* and *median*. Each one iss applied to the feature sub-vectors and results are concatenated, leading to a total of 180 features for each audio file. Scaling yields to promising results on the first dataset, so the same approach is applied to the extended one.

After testing a network on the new training set we can see a better accuracy.

Feature selection Adding new features can lead to better results in the end but they all need to be useful to the model. For this reason the extended dataset is subject of some experiments with feature selection, in particular *PCA* algorithm from scikit learn is applied. [**?**]

The main idea is to select a reduced number of features from the total, loosing as little information as possible. This approach often leads to better results, as useless features are discarded.

The method used to select features offers the possibility to specify how much variance to preserve in the reduced dataset, this means that the number of components is not given explicitly, indeed we try to preserve 99 percent of the original variance.

This technique resulted in 102 features, unfortunately, a model applied to this new dataset failed to reach the performances obtained by the previous one, nonetheless, performances improved with respect to the scaled dataset.

3 Model definition

This Section starts by presenting how the networks used on the initial training sets are structured. We proceed to give a detailed overview of the performances on the different training sets, lastly, the hyperparameter tuning phase is described.

3.1 Neural network structure

The starting point for the neural network structure is a reasonable network in terms of hidden neurons to prevent over-fitting, indeed a high number of units in the hidden layers would end up in learning too much from the dataset, leading to poor performances on the test sets.

For this reason, the rule of thumb followed to decide hidden neurons quantity is the following:

$$\#hidden\ neurons = \frac{2}{3}\big(\#input\ neurons + \#output\ neurons\big)$$

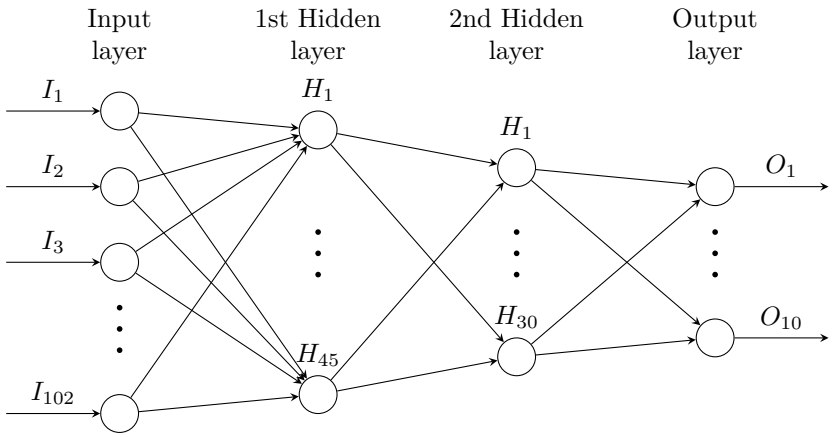
The next step is to decide the hidden layers number. As using the rule presented above gives a quite small amount of units, only two layers are considered, indeed, an higher quantity would mean having a real small number of neurons per layer.

Applying this rule ended up in the following architectures on the four different training sets, where the output layer is fixed at 10:

Training set	Input	1st Hidden	2nd Hidden
132 features unscaled	132	60	30
132 features scaled	132	60	30
180 features scaled	180	80	46
102 features reduced with PCA	102	45	30

To build the actual model *Tensorflow* and *Keras* libraries are used. [**?**][**?**]

Starting point model To give reference, this is the model used on the PCA training set, mentioned at the end of Subsection 2.3 on page 4, with 102 input features.



This last model, with the one built to predict the 180 features training set, are the ones selected for the hyperparameter tuning phase.

Activation and loss functions The activation function for the input and hidden layers is a *Relu*, typically used to prevent the *vanishing gradient problem*, and the output one uses a *Softmax* to have classification probabilities among classes. [**?**][**?**][**?**]
The loss used is the *Sparse Categorical Crossentropy loss* as it is well suited for multiclass classification and, since the classes are integers and not one-hot encoded, the sparse version is preferred. [**?**]

Choosing an optimizer When choosing an optimizer for a neural network one must take into account the cost of reaching a minimum point on the error function. Although more complex optimizers exist, build to reduce training cost or achieve better performances on deep networks, the one choosen for this model is a classic Stochastic Gradient Descent optimizer.

Testing some more advanced optimizers shows that convergence is reached faster on the networks used in the experiments, but the speed increase is negligible as the model is quite small.

3.2 Initial training set results

Section 2 on page 3 talks about the four different training sets obtained from the dataset and, without going into details, states that there is continuous improvement. We now give a more detailed look on training performances, after detailing how class imbalance is faced and the applied validation method.

Note that all the random seeds used by Tensorflow are fixed to make results reproducible. This step is necessary as many parameters initial value is random, for instance, the neural network weights.

Class imbalance To deal with the minority of some classes, balancing techniques should be applied when fitting the model. One of the possible approaches, and the one followed here, is to assign class weights.

The main idea is to penalize errors made on not well represented classes to account for their minority. Class weights computation relies on the *compute class weights* function from sklearn. [**?**]
The following are the computed quantities for the dataset classes:

Class name	Number of samples	Class weight
air conditioner	500	0.8998
car horn	208	2.1629
children playing	500	0.8998
dog bark	500	0.8998
drilling	500	0.8998
engine idling	517	0.8702
gun shot	190	2.3678
jackhammer	548	0.8209
siren	536	0.8393
street music	500	0.8998

As expected, the less numerous classes have higher class weight than the rest, in particular, the misclassification of a car horn sample counts more than double than an air conditioner one.

Stratified cross-validation To estimate performance on the training set, stratified cross-validation with five folds is used. Basically the dataset is divided into five parts and a model is repeatedly trained on four and tested on one, all while considering class distribution, indeed, the original distribution of the classes is maintained in the splits. [**?**]

The stratified approach is required as there is class imbalance on the training set. In fact, applying a classical cross-validation could show misleading results, for instance when the minority classes are more present in the test fold rather than the training ones; in such cases the loss would be higher. The mean accuracy on the test folds gives a hint about the model performance.

For this step the *Stratified KFold* class from scikit learn is used. [**?**]

Results The following are the results on the training sets, using the architectures presented at the beginning of the Subsection.

Training set	Mean accuracy	St. deviation
132 features unscaled	0.1138	0.0039
132 features scaled	0.5743	0.0324
180 features scaled	0.6363	0.0494
102 features reduced with PCA	0.6188	0.0420

There is a great improvement after scaling the training set, after that small

refinements are made. As accuracy is the best on the last two training sets, those are the two selected to perform the hyperparameter tuning.

3.3 Hyperparameter tuning

The last two tries with feature extraction lead to the best results with stratified cross-validation. Although the two models are reasonable, they can not be the final ones as many parameters are left on their default value, for instance, learning rate and momentum of the optimizer are untouched.

The main goal now is to experiments with ranges of model parameters to find a better one. From now on, the model build on the 180 features training set is called *Extended model*, while the one tested on the PCA training set is named *PCA model*.

Grid and random search comparison Two of the most commonly used strategies in hyperparameter optimization are *grid* and *random search* [**?**].

In both cases we define ranges of parameters to test different combinations, for instance, fixed the number of neurons, one could try to find the best combination of learning rate and momentum that optimize accuracy on the training set.

While similar, the two methodologies differs in the amount of exploration they do. The grid search try all the possible combinations of parameters, while the random approach fixes a number of iterations and picks an arbitrary unseen combination each time.

Obviously the first one is more computationally expensive than the second, if we fix a small amount of possible iterations, but in theory it finds a better result than going the random route. Nonetheless the grid search can led to over-fitting and in practice random search in preferred.

Random search We now run a random search with various parameters to optimize the initial models. Note that class weights are still considered and the models are evaluated again with a stratified cross-validation. The optimizer used is the stochastic gradient descent.

The considered ranges for parameters for this run are:

1. *Neurons*: input layer has dimension H_i and last layer H_o, while the two hidden layers are tested with a number of neurons respectively equals to:

$$H_1 + 2i \text{ and } H_2 + 2j, \text{ with } i, j \in \{-2, -1, 0, 1, 2\}$$

2. *Learning rate*: $0.001, 0.01, 0.1, 0.5$;
3. *Momentum*: $0.0, 0.01, 0.1, 1$;
4. *Epochs*: $60, 80, 100$;
5. *Batch size*: $32, 64$.

Where H_i, H_1, H_2 and H_o are input, first hidden, second hidden and output layer dimensions. The random search is performed on two models, with those starting dimensions:

- *Extended model*: 180, 80, 46, 10.
- *PCA model*: 102, 45, 30 and 10;

An *early stopper* with default parameters is used on the training to stop it when no progress is made with respect to the last epoch results. [**?**] The search is performed with 100 iterations for both rounds.

Final models The first round of the random search, performed on the Extended model, resulted in the following parameters:

- Neurons: 180 for input, 76 for the first hidden layer, 50 for the second and 10 for output;
- Momentum: 0.01
- Learning rate: 0.01
- Epochs: 80
- Batch size: 64

We can see an improvement in accuracy by comparing it with to starting point model:

Model	Mean accuracy	St. deviation
Initial extended model	0.6363	0.0494
Random search result	0.6497	0.0431

The second random search on the smaller PCA model, resulted in this results:

- Neurons: 102 for input, 45 for the first hidden layer, 30 for the second and 10 for output;
- Momentum: 0.01
- Learning rate: 0.1
- Epochs: 80
- Batch size: 32

As before, comparing it with the starting model, we can see some better results:

Model	Mean accuracy	St. deviation
Initial PCA model	0.6188	0.0420
Random search result	0.6270	0.0315

4 Conclusions

This last Section shows the results obtained by the best model found on the previous Section, namely the *Extended model*, on the different test sets and gives an overview about possible future works to be made.

4.1 Test set results

As stated on the first Section, the five test sets are made out of the folds number five, seven, eight, nine and ten. Evaluating the extended model found by the random search on the test sets gives the following results:

Test set	Accuracy
Fold 5	0.7425
Fold 7	0.6038
Fold 8	0.6873
Fold 9	0.6348
Fold 10	0.6858

Finally, the mean accuracy and standard deviation are:

Mean accuracy	Standard deviation
0.6708	0.0478

4.2 Future works

Results on the test sets are promising but there is definitely room for improvement.

Indeed, a more refined training set creation can be made, by exploiting more features from the Librosa library, testing different type of scalers and experimenting with different feature selection techniques.

Finally, the models used in the project are simple. More complex neural networks, for instance with convolutional layers, could perform better, also, a more extensive random search might improve results.