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. [1]

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 testing different types of neural networks to classify the data obtained from the feature extraction phase. This part is discussed in Section 3 on page 7.
Lastly, results from the classification, namely accuracy and standard deviation among test sets, and possible future works are presented in Section 4 on page 16.

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, for the first and extended datasets 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*. [2][3]

A *convolutional neural network*, a network with a series of convolutional and pooling layers, followed by some densely connected ones, is applied to the image dataset, trained for 10 epochs. [4]
The accuracy on the training is computed with a *cross-validation* approach. [5]

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 7.

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. [6]

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. [7]

To extract information from audio files *Librosa* is used. [8] 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*: the first 20 coefficients are extracted;
2. *Chromagram*: the first 12 chromas are considered;
3. *Root-mean-square*.

Method parameters that are not specified in the above list, are left on their default values. 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. [9]

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. [10] 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* with default parameters from *scikit learn* is applied. [11] 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.*

As before methods parameters are left on default, but this time *standard deviation* is added to the previous functions *minimum*, *maximum*, *mean* and *median*. Each one is 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. [12]

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.

2.4 Image dataset

A completely different approach is followed this time to represent audio files as images, to later apply a convolutional neural network.

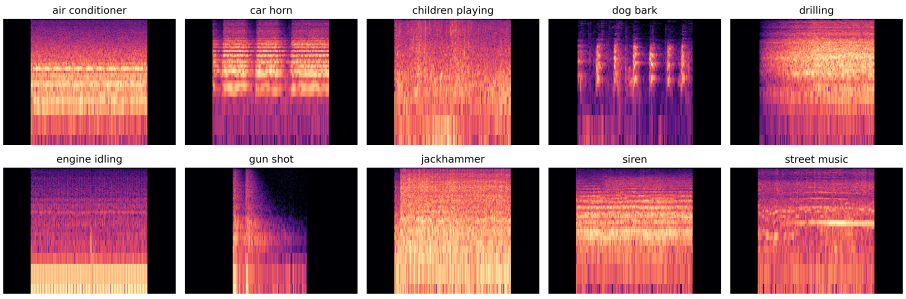

Figure 1: Short time Furier transform on samples coming from each class. The scale on the Y axis is logarithmic, the X axis represent time and the black portion on the edges is due to the applied padding.

Audio as an image Representing an audio as an image is not straightforward but can be done after some preprocessing. The main idea is to use audio features and consider them in a two dimensional space, where the value of a single cell can be viewed as a pixel. Note that some features are well suited for this kind of representation, for instance, some of the previously extracted ones are given in output as a two dimensional vector.

Usually, an image has three channels, red, green and blue, but image classification on grayscale images, with one single channel is also relevant. This means that one can extract multiple features and view them as different channels, or choose only one to have the equivalent of a grayscale image.

Short time Furier Transform For this try with image classification the Short-time Furier transform is extracted from the audio files, and then used as a single channel image. In particular, we extract 128 intervals, and, to accomodate the different lenghts of audio in the dataset, pad the result to 256 wide vectors.

The result is an 128×256 image for each sample. Figure 1 shows an example of an image obtained for each class.

We can see some difference between the classes, for instance, the *dog barking* shows short repeated sounds, *gun shot* has a declining intensity, while *air conditioner* is constant. The hope is that those subtle differences are enough lo learn and classify.

Scaling pixel values Scaling techniques are usually applied even to images to prevent high disparities among pixel value ranges, for this reason a Standard scaler with default parameters is once again applied to the training set to mitigate this effect. We can not apply it directly to the images, as they are two dimensional, but we can flatten the pixel arrays, apply scaling and reshape them in the original 128×256 format.

Testing a convolutional network on the image dataset gives results that are comparable to the ones obtained on the initial scaled training set, discussed at Subsection 2.2 on page 4.

3 Model definition

This Section gives an overview of the models used on the different training sets. We first start with the multilayer perceptrons and end with the convolutional networks used on the image dataset.

For both models we present the architecture, the initial results and the hyperparameter tuning phase.

3.1 Multilayer perceptron

The first model used in the experiments is a multilayer perceptron. The considered training sets are the ones presented on Subsection 2.2 and 2.3 namely the first and extended datasets. For the first one, scaled and unscaled versions are considered, while for the second the normal and PCA version are tested.

3.1.1 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. [13][14]

Starting point model To give reference, Figure 2 shows the model used on the PCA training set, mentioned at the end of Subsection 2.3 on page 5, 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.

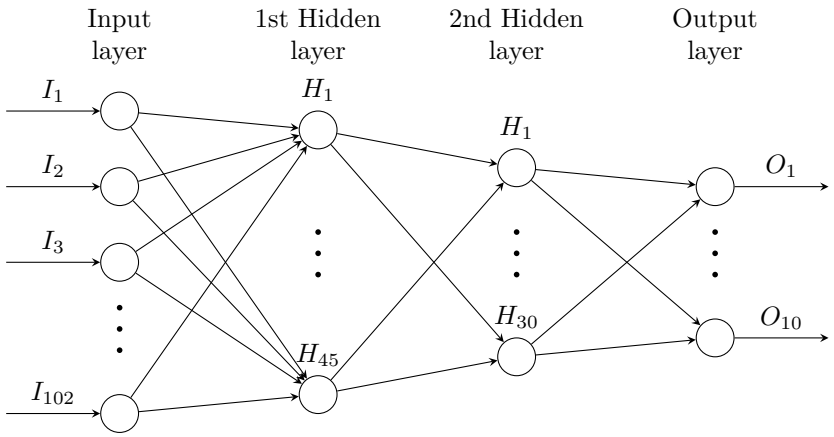

Figure 2: MLP architecure used on the PCA dataset

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. [15][16][17]
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. [18]

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.

3.1.2 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, in particular, each class has weight:

$$w_i = \frac{\#total\ samples}{\#classes * \#samples\ of\ class\ i}$$

Class weights computation relies on the *compute class weights* function with the balanced logic from sklearn. [19]
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. [20]

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. [21]

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

The runs are performed with a five fold stratified cross-validation. At each fit, epochs are fixed at 100, batch size is set at 64 and class weights, computed once on the entire training set, are considered. The optimizer is the stochastic gradient descent with zero momentum and 0.01 learning rate.

Mean accuracy and standard deviation are finally computed on the folds results.

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.

3.1.3 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* [22].

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 five fold 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 I, last layer is fixed at 10, 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*: $32, 64$.

The quantities I, H_1, H_2, which are input, first hidden and second hidden layer dimensions, depend on the initial network architecture. As stated before, the two models considered are the following:

Model name	I	H_1	H_2
Extended model	180	80	46
PCA model	102	45	30

An *early stopper* with patience equals to three is used on the training to stop it when no progress is made with respect to the last three epochs result. [23] The search is performed with 100 iterations for both rounds.

Final models The first round of the random search, performed on the Extended model, finds 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 to the starting point model:

Model	Mean accuracy	St. deviation
Initial extended model	0.6363	0.0494
Tuned extended model	0.6497	0.0431

The second random search on the smaller PCA model, finds these parameters:

- Neurons: 102 for input, 45 for the first hidden layer, 32 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
Tuned PCA model	0.6270	0.0315

Final remarks on MLP Unexpectedly PCA application lead to worse results compared to the 180 feature dataset. Even after the random search, the so called Tuned Extended model performs better. For this reason this is the final multilayer perceptron evaluated on the test set.

3.2 Convolutional neural network

This subsection presents the results obtained by a convolutional neural network trained on the image datasets presented at the end of Section 2 on page 3. We start by giving an overview of the network, to then proceed with the training results and hyperparameter tuning.

3.2.1 Network structure

The structure of a neural network for image classification consists of convolutional layers followed by pooling layers, and, at the end, densely connected ones to have the output classification.

The main idea is to extract relevant features from the images with the convolutional layers, that apply a kernel to the input to detect patterns and image features. After the convolution part there is a pooling layer that reduces dimensionality. At the end, densely connected layers classify the obtained features.

Stacking convolutions Taking inspiration from the *VGG19 architecture*, that stacks two or more convolutional layers to then apply pooling, the network is built with the following layers: [24]

- *Convolution*: 8 filters, kernel size of 5×5, Relu activation, 1 stride
- *Convolution*: 8 filters, kernel size of 5×5, Relu activation, 1 stride
- *MaxPooling*: pooling size of 2×2, 3 strides
- *Convolution*: 16 filters, kernel size of 3×3, Relu activation, 1 stride
- *Convolution*: 16 filters, kernel size of 3×3, Relu activation, 1 stride
- *MaxPooling*: pooling size of 2×2, 3 strides
- *Flatten layer*

- *Dense layers*: dimensions are 2000, 800, 200 and 60, Relu activation
- *Output*: 10 neurons and softmax activation

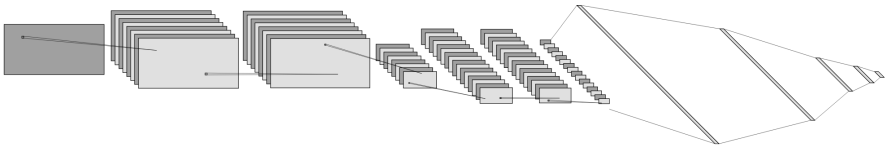

Figure 3: Diagram of the network used in the experiments, we can see the first two convolutions, followed by the max pooling layer. After that, two other convolutions and another max pooling, and finally the densely connected ones. We can see the principle of learning higher level features and fine grain ones, indeed the dimensionality increases from the first convolutions to the seconds.

Kernel sizes, filters and strides Two crucial parameters for a convolutional layers are the number of filters, the kernel size and strides dimensions. The filter number represent the number of ways a convolutional layer sees the image, while the kernel size makes the area considered by the layer bigger or smaller. One can see a small kernel size as a way of learning fine grain features, on the opposite, a bigger size capture higher level image characteristics. Finally, strides determine the reduction on the output shape, for instance, a stride equals to two, reduces the output shape in half.

One important factor in choosing the values for this parameters is the output dimensionality, in fact, while strides reduces it, higher filters and smaller kernels increase the feature space.

For this reasons, filters number is smaller and kernel size is bigger for the first convolutional layers, while we have more filters and smaller kernel for the consequent ones. The idea is to learn higher level features first, to then capture small peculiarities. For the first architecture stride on convolutional layers is kept at one, later, an higher value is tested.

3.2.2 Training set results

We now present the results on the image dataset using the model introduced in the *Stacking convolutions* paragraph. Stratified cross validation with five folds is once again used, the number of epochs is fixed at 10 and we use an early stopper with patience equals to 2, to stop training if no improvement is made on accuracy for two epochs. The principle of class weights, introduced with the previous experiments, is reused this time. The optimizer used it the Adam one, with learning rate set to 0.001.

We can compare results with previously tuned multilayer perceptrons, as the validation technique is the same.

Training set	Mean accuracy	St. deviation
Tuned extended model	0.6497	0.0431
Tuned PCA model	0.6270	0.0315
Initial CNN	0.5383	0.0395

Results with the CNN are a big downgrade from previously tested models, but we need to consider the fact that those have their hyperparameters tuned.

3.2.3 Hyperparameter tuning

Training a CNN is really expensive compared to the MLP networks. Performing a full random search is infeasible with the resources we have, therefore, to have an idea about the impact on accuracy of the network parameters, we run cross-validation on the following models, that, for simplicity, have a name.

Name	Filters	C. strides	P. strides	Dense structure
C_1	8	3	1	300, 150, 80, 25
C_2	16	1	3	5000, 2000, 800, 200, 60
C_2	16	3	3	600, 200, 80, 25

The filter quantity represent the number of filters for the first convolutions. The subsequent ones have double that quantity. Kernel size is fixed at 5×5 and 3×3 for the first two and last two convolutional layers respectively.

The *C. strides* values are the strides for convolutions, a value of one do not reduce dimensionality, while a value of three reduces the output dimensionality by two thirds.
The *P. strides* are the pooling layers strides. As before, a value of one do not reduce dimensionality, while two cut the output in half.
Lastly, the *dense structure* sums up the last layers architecture. For instance, the model C_1 have four consequent densely connected layers, with neurons 300, 150, 80 and 25, followed by a final output layer of 10 neurons.

The rest of the network is the same as the initial CNN, thus activation is always a Relu, apart from the Softmax on the output layer, pool size is fixed at 2×2.

Cross-validation results The following are cross-validation results. The settings for the cross validation are the same ones presented in the *Training set results* paragraph, apart from the epochs that this time are increased to 15. We can see that the C_3 model is the best performer, even if the differences among models are not really big.

Model name	Mean accuracy	St. deviation
Initial model	0.5383	0.0395
C_1	0.5347	0.0480
C_2	0.5285	0.0325
C_3	0.5412	0.0474

Overfitting and underfitting Cross-validation results gives an hint about overfitting and underfitting. Indeed, by looking at the trainable parameters quantity using Keras summary function we have the following results.

Model name	Trainable parameters
Initial model	13011950
C_1	256095
C_2	67957294
C_3	923789

The numbers might indicate that the C_1 model is underfitting, as the number of parameters does not create the required complexity in the model to capture the dataset, the C_2 and initial models are overfitting, as they have too many parameters thus they are learning too much from the training. C_3 seems to be at a sweet spot between the two, confirmed by the higher accuracy.

Final remarks on CNN Unfortunately, the high cost of training a CNN made impossibile to run a Random search starting from the initial network structure. Results are worse than the ones obtained with the MLP models.

4 Conclusions

This last Section presents results obtained on the test sets. We test the best multilayer perceptron and the best convolutional network found.

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.

Test set preprocessing The test sets are preprocessed with the same techniques used to extract the training sets. In particular, regarding the MLP dataset, with 180 features, the Standardscaler fitted on the training set is reused on the test folds. Similarly, the scaler used to scale pixel values on the image dataset is reused on the ones coming from the test folds.

Results The MLP model is the fine tuned one coming from the hyperparameter tuning phase, trained on the 180 features dataset, namely the *Extended model*, while the CNN one is the best performer from the tried convolutional models, the so called C_3 network.

Test set	MLP Accuracy	CNN Accuracy
Fold 5	0.7425	0.5171
Fold 7	0.6038	0.5668
Fold 8	0.6873	0.5744
Fold 9	0.6348	0.5515
Fold 10	0.6858	0.5639

The mean accuracy and standard deviation are:

Model	Mean accuracy	Standard deviation
MLP	0.6708	0.0478
CNN	0.5547	0.0202

4.2 Future works

Results are better on MLP models, but CNNs shows potential on the problem. Results could be improved, 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.

Regarding the image dataset, more channels could be considered by using more features as images, also, data augmentation techniques could be applied in order to increase the cardinality of the dataset. Finally, with more computational power, a proper random search could be performed.

References

[1] Justin Salamon, Christopher Jacoby, and Juan Bello. A dataset and taxonomy for urban sound research. 11 2014.

[2] Wikipedia. Multilayer perceptron. https://en.wikipedia.org/wiki/Multilayer_perceptron.

[3] Wikipedia. Stochastic gradient descent. https://en.wikipedia.org/wiki/Stochastic_gradient_descent.

[4] Wikipedia. Convolutional neural network. https://en.wikipedia.org/wiki/Convolutional_neural_network.

[5] Wikipedia. Cross validation. https://en.wikipedia.org/wiki/Cross-validation_(statistics).

[6] Pandas. https://pandas.pydata.org.

[7] Joel Jogy. How i understood: What features to consider while training audio files? https://towardsdatascience.com/how-i-understood-what-features-to-consider-while-training-audio-files-eedfb6e9002b.

[8] Librosa. https://librosa.org.

[9] Numpy. https://numpy.org.

[10] Dask. https://dask.org.

[11] Scikit learn. Standard scaler. https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html.

[12] Scikit Learn. Pca. https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html.

[13] Tensorflow. https://www.tensorflow.org.

[14] Keras. https://keras.io.

[15] Wikipedia. Rectifier. https://en.wikipedia.org/wiki/Rectifier_(neural_networks).

[16] Wikipedia. Softmax function. https://en.wikipedia.org/wiki/Softmax_function.

[17] Wikipedia. Vanishing gradient problem. https://en.wikipedia.org/wiki/Vanishing_gradient_problem.

[18] Kiprono Elijah Koech. Cross-entropy loss function. https://towardsdatascience.com/cross-entropy-loss-function-f38c4ec8643e.

[19] Scikit learn. Compute class weight. https://scikit-learn.org/stable/modules/generated/sklearn.utils.class_weight.compute_class_weight.html.

[20] KSV Muralidhar. What is stratified cross-validation in machine learning? https://towardsdatascience.com/what-is-stratified-cross-validation-in-machine-learning-8844f3e7ae8e.

[21] Scikit learn. Stratified kfold. https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedKFold.html.

[22] Kishan Maladkar. Why is random search better than grid search for machine learning. https://analyticsindiamag.com/why-is-random-search-better-than-grid-search-for-machine-learning.

[23] Keras. Early stopper. https://keras.io/api/callbacks/early_stopping/.

[24] Aakash Kaushik. Understanding the vgg19 architecture. https://iq.opengenus.org/vgg19-architecture/.