UNIVERSITÀ DEGLI STUDI DI MILANO

Department of Computer Science

Statistical Methods for Machine Learning

Urban Sound Classification with Neural Networks

Alice Sansoni

ACCADEMIC YEAR 2021/2022

Declaration

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.

Contents

1 Background **4**
- 1.1 Multi-Layered Perceptron . . . 4
- 1.2 Convolutional Neural Network . . . 4

2 Dataset **5**

3 Features Pre-processing **7**
- 3.1 Libraries . . . 7
- 3.2 Data Pre-processing . . . 8
- 3.3 Feature extraction . . . 8
  - 3.3.1 MFCCs (Mel-frequency Cepstral coefficients): . . . 9
  - 3.3.2 MelSpectrogram . . . 9
  - 3.3.3 Chromagram . . . 10
- 3.4 Feature normalization . . . 10
- 3.5 Dataset creation . . . 10

4 PCA Analysis **11**

5 Optimizer and Loss Function **12**
- 5.1 Optimizer . . . 12
- 5.2 Loss Function . . . 13

6 Layers **14**
- 6.1 Convolutional Layer . . . 14
- 6.2 MaxPooling . . . 15
- 6.3 Flatten . . . 15
- 6.4 Dense . . . 16
- 6.5 Dropout . . . 17

7 Hyperparameters **18**

8 Architectures **20**
- 8.1 Artificial Neural Network . . . 20
- 8.2 Convolutional Neural Network . . . 23
- 8.3 Transfer Learning . . . 26

9 Feature Distribution **30**

10 Results **31**

Introduction

The aim of this project is to build and train Neural Networks to classify urban sounds events based on audio files from the UrbanSound8K Dataset. TensorFlow was used with Python to develop the code. Different feature extraction methods, architectures and training parameters must be used to show how they affect the final predictive performance. The source code of the project is available here `https://github.com/AlySansoni/SMML.git`

Chapter 1

Background

A Neural Network is a computer architecture in which a number of processors are interconnected in a manner suggestive of the connections between neurons in a human brain and which is able to learn by a process of trial and error.
Neural networks can adapt to changing input; so the network generates the best possible result without needing to redesign the output criteria. When using more than one hidden layer, a NN enters the Deep-learning domain, where features are automatically extracted and the network learns by examples. The task of the project, being a classification problem, requires supervised learning where true labels are compared with the predicted ones.

1.1 Multi-Layered Perceptron

In a Multi-layered Perceptron (MLP), perceptrons are arranged in interconnected layers. The input layer collects input patterns. The output layer has classifications or output signals to which input patterns may map. Hidden layers fine-tune the input weightings until the neural network's margin of error is minimal. It is hypothesized that hidden layers extrapolate salient features in the input data that have predictive power regarding the outputs. Layers are usually fully connected if dropout is not applied. If it is, some of the neurons are turned off to force the Network to learn higher-level features.

1.2 Convolutional Neural Network

CNN is a class of Deep Learning algorithms especially suitable for images. It usually takes an image or something that can be shaped as an image and assigns importance (weights and biases that are learnable) to various aspects/objects in the image. In this project CNNs should work better because they can successfully capture the Spatial and Temporal dependencies in an image through the application of relevant filters. The role of a CNN is to reduce the images into a form that is easier to process, without losing features that are critical for getting a good prediction.

Chapter 2

Dataset

The UrbanSound8K dataset contains 8732 labeled sound excerpts of urban sounds, for a total of 8.75 hours of annotated sound events, for 10 different classes: **air conditioner**, **car horn**, **children playing**, **dog bark**, **drilling**, **engine idling**, **gun shot**, **jackhammer**, **siren**, and **street music**. These classes are chosen from a pre-defined taxonomy. "Children playing" and "gunshot" are added for variety, while all other classes are selected due to the high frequency in which they appear in urban noise complaints. All records are taken from freeSound3, using their original format, with all real field recordings from an urban environment. The maximum length of each sound is about 4 seconds, which was defined as a sufficient length for subjects to identify environmental sounds with good accuracy. Longer occurrences are split into slices, with a limit of 1000 slices per class to avoid large differences in the Dataset. That's the reason why important care was taken when creating the folds and splitting them according to these criteria (trying also to balance the number of slices-per fold for each sound class), otherwise, slices from the same recording could be used both from training and testing. For the same reason, folders need to be kept separated when learning and testing the Dataset. Sounds signals can have a different sampling rate, a different number of channels, and a different length, so, before extracting the features, they need to be re-elaborated to be comparable.

In addition to the sound excerpts, a CSV file containing metadata about each excerpt is provided. This includes:

- **slice_File_Name**: the name of the audio file. The name takes the following format: [fsID]-[classID]-[occurrenceID]-[sliceID].wav
  - fsID: the Freesound ID of the recording from which this excerpt is taken
  - classID: a numeric identifier of the sound class
  - occurrenceID = a numeric identifier to distinguish different occurrences of the sound
  - sliceID = a numeric identifier to distinguish the same occurrences split into slices of the sound within the original recording
- **fsID**: see what is written above
- **start**: The start time of the slice in the original Freesound recording
- **end**: The end time of the slice in the original Freesound recording

- **salience**:A (subjective) salience rating of the sound. 1 = foreground, 2 = background.
- **fold**: The fold number (1-10) to which this file has been allocated.
- **classID**: A numeric identifier of the sound class:
  - 0 = air_conditioner
  - 1 = car_horn
  - 2 = children_playing
  - 3 = dog_bark
  - 4 = drilling
  - 5 = engine_idling
  - 6 = gun_shot
  - 7 = jackhammer
  - 8 = siren
  - 9 = street_music
- **class**: The class name

The approximate size of the Dataset is 6.60 GB.
Number of images per class:

- Id = 0 Air_conditioner: 1000
- Id = 1 Car_horn: 429
- Id = 2 Children_playing: 1000
- Id = 3 Dog_barking: 1000
- Id = 4 Drilling: 1000
- Id = 5 Engine_idling: 1000
- Id = 6 Gun_shot: 374
- Id = 7 Jackhammer: 1000
- Id = 8 Siren: 929
- Id = 9 Street_music: 1000

Chapter 3

Features Pre-processing

3.1 Libraries

To compute all the different steps that are explained in the next paragraphs, some libraries were used.

- **Keras**: a deep learning API written in Python, running on top of the machine learning platform TensorFlow. It provides an approachable, highly-productive interface for solving machine learning problems, with a focus on modern deep learning. It provides essential abstractions and building blocks for developing and shipping machine learning solutions with high iteration velocity. [3]
- **Numpy**: a Python library used for working with arrays. It also has functions for working in the domain of linear algebra, Fourier transform, and matrices. [4]
- **Matplotlib**: a comprehensive Python library for creating static, animated, and interactive visualizations. [5]
- **Pandas**: a Python library for data analysis. It offers a powerful and flexible quantitative analysis tool. It is built on matplotlib for data visualization and NumPy for mathematical operations. [7]
- **Scikit-learn**(sklearn):It contains a lot of efficient tools for machine learning and statistical modeling including classification, regression, clustering, and dimensionality reduction. It is built on NumPy, SciPy, and matplolib. [2]
- **Librosa**:a Python package for music and audio analysis. It is mainly used when working with audio data like in music generation, or Automatic Speech Recognition. It provides the building blocks necessary to create the music information retrieval systems.[1]
- **Pickle**: it is a module used for serializing and de-serializing python object structures. So, it is used to convert a Python object into a byte stream to store it in a file/database, maintain program state across sessions, or transport data over the network. [9]

3.2 Data Pre-processing

In order to create the Dataset, the first thing that is needed is to load the file audio. To do that, the CSV file containing audio metadata is used. For each line of the file the number of the folder and the *slicefilename* are extracted. Both are attached to the path file linked to the Dataset. Then, the sound file and the sampling rate are obtained with:

```
sound_file, sampling_rate = librosa.load(audio_name, sr = None,
    mono=False)
```

- audioname = is the one generated as written above
- **sr** = None and **mono** = False means to keep the original sampling rate and the number of channels when loading the audio.

To produce the same input size for the Network, several steps are applied to each raw data:

1. Resampling: if the original sampling rate is different from the target one (44100 kHZ) the audio file is resampled.

```
sound_file = librosa.resample(y=sound_file, target_sr=
    SAMPLING_RATE, orig_sr=sampling_rate)
```

2. One_channel audio: if the original file is stereo audio, with 2 channels, a mean is computed between the two values, to obtain one channel only.
3. Same_length audio: A target number of samples is chosen considering the target sampling rate multiplied by the desired number of seconds: 44100*4 = 176400 samples.
   - If the length of the file is greater than the target number of samples, the vector signal is cut at the target number of samples.
   - If the length of the file is less than the target number of samples, it is padded with 0 (silence).

```
sound_file = np.pad(sound_file,(0,NUM_SAMPLES-sound_file.
    shape[0]))
```

3.3 Feature extraction

Instead of raw data, several features are extracted to feed the Neural Network, which can be time-domain, frequency-domain features, or time-frequency-domain features. The latter means that they can express both the temporal and the frequency aspects with a single representation.

3.3.1 MFCCs (Mel-frequency Cepstral coefficients):

The mel-frequency cepstrum (MFC) is a representation of the short-term power spectrum of a sound, based on a linear cosine transform of a log power spectrum on a nonlinear Mel scale of frequency [10]. The MFCCs are coefficients that collectively create an MFC. They are derived from a cepstral representation of the audio clip, split into overlapping windows (frames). A cepstrum is a non-linear "spectrum-of-a-spectrum", obtained by applying the Discrete cosine Transform to the logarithm of the spectrum, which is obtained by applying DFT (Discrete Fourier transform) to the time domain signal with a subsequent mapping onto the mel scale. MFCCS are the amplitudes of the resulting spectrum.
For a more robust analysis, the first and the second order frame-to-frame derivative are computed, approximated using the difference of the values between two frames and the difference of the obtained differences. The three vectors are then concatenate together to create a single matrix of features.

```
mfccs = librosa.feature.mfcc(y=sound_file, n_mfcc = NMFCC, sr=
    SAMPLING_RATE)
#first and sec der of mfccs = delta and delta delta mfccs
delta_mfccs = librosa.feature.delta(mfccs)
delta2_mfccs = librosa.feature.delta(mfccs, order = 2)

mfccs_tot = np.concatenate((mfccs, delta_mfccs, delta2_mfccs), axis
    =0)
```

- n_mffc: traditionally the first 13 coefficients contain the most relevant info so in the project the chosen number is 15, but another version was done using 50 coefficients (as suggested in [8]).

3.3.2 MelSpectrogram

MelSpectrogram is a mel-scaled Spectrogram. The Spectrogram is obtained by applying the short-time Fourier transform to the audio signal on overlapping windowed segments of the signal and stacking the results on top of each other [11]. It is a way to visually represent the loudness or the amplitude of a signal, as it varies over time. The Mel scale was designed because humans do not perceive frequencies on a linear scale so it was created in such a way that equal distances in pitch sounded equally distant to the listener. So, a Mel Spectrogram is a spectrogram where the frequencies are converted to the mel scale.

```
mel_spectogram = librosa.feature.melspectrogram(y=sound_file, sr=
    SAMPLING_RATE, n_fft=NFFT, hop_length=HOP_LEN, n_mels=NMEL)
```

- n_fft: is the length (number of samples) of the FFT window to split the signal
- hop_length: is the number of samples between successive frames, for the overlapping window.
- n_mels: is the number of bins for the construction of the mel-filterbank.

3.3.3 Chromagram

A Chromagram is computed from a waveform or power spectrogram. It contains the normalized energy for each chroma bin at each frame. Chromagram is defined as the whole spectral audio information mapped into one octave.

```
chroma = librosa.feature.chroma_stft(y=sound_file, sr=SAMPLING_RATE
    , n_fft=NFFT, hop_length=HOP_LEN, n_chroma=12)
```

- n_fft and hop_length: are the same as written above.
- N_chroma: is the number of chroma bins to produce

3.4 Feature normalization

To be able to compare features across different files and to reduce the intra-class variance, data need to be normalized. Each feature requires a proper normalization. For the MFCCs a Standard Scaler is chosen, which is first fitted to the training data and then applied to transform both training and test data. It standardize features by removing the mean and scaling to unit variance. The standard score of a sample x Is calculated as: $z = (x - u)/s$ where u is the mean of the training samples and s is the std of the training samples

```
scaler = StandardScaler().fit(train_mfccs.reshape(-1, train_mfccs.
    shape[-1]))
train_mfccs = scaler.transform(train_mfccs.reshape(-1, train_mfccs.
    shape[-1])).reshape(train_mfccs.shape)
```

To normalize melSpectogram data *librosa.util.normalize* method is used. Given a norm, and an axis, the input array is so scaled that: $norm(S, axis = axis) == 1$. This is applied to each melSpectogram array singularly.

```
for i in range(len(train_melSpect)):
    train_melSpect[i]=librosa.util.normalize(train_melSpect[i])
```

Chromagram features don't need to be normalized because the output itself is already the normalized energy for each bin.

3.5 Dataset creation

Each feature is saved separately using a pickle file and split into training and test set, according to the folder id attached to the file. The training set is composed of all files whose folder id is in [1,2,3,4,6] while the others [5,7,8,9,10] are used separately for the test phase.
The different architectures are then trained and tested using all these features alone and with some combinations of them: mfcc + melSpectrogram, mfcc + chromagram, mfcc + melSpectrogram + chromagram. The PCA technique is applied to the latter, to perform a dimensionality reduction.

Chapter 4

PCA Analysis

Principal Component Analysis [6] is a dimensionality-reduction method that is used to reduce the dimensionality of large Datasets, by transforming a large set of variables into a smaller one that still contains most of the information in the large set. This allows a faster computation of machine learning algorithms.
In the project, PCA was exploited for two reasons:

1. Using 2 as the number of components all features could be projected into two dimensions to obtain an approximate representation of where blobs of points for each class are located in the 2D space and so have a better understanding of the results, exploiting the visual reference [See 9]

2. To overcome memory issues, when concatenating many features, a PCA analysis is applied to reduce the dimensionality of the Dataset, checking that a good amount of information are still included in the data with *pca.explained_variance_ratio* field. It is applied on train data alone and then on each folder of test data.

```
pca = PCA(n_components=PCA_COMPONENT, svd_solver='full')
sel_feat = pca.fit_transform(sel_feat.reshape(-1, sel_feat.shape
    [-1])).reshape((sel_feat.shape[0],sel_feat.shape[1],
    PCA_COMPONENT))
```

Chapter 5

Optimizer and Loss Function

5.1 Optimizer

Optimizers in Neural Networks are algorithms or methods used to change the attributes of the NN such as weights and learning rate to reduce the losses, for the gradient descent. Every optimizer defines how these attributes should change to provide the most accurate results possible.
The one that is used in the project is **Adam** (Adaptive Moment Estimation), which is efficient when working with a large problem involving a lot of data or parameters. Adam optimization is a stochastic gradient descent method that is based on the adaptive estimation of first-order and second-order moments. It is a combination of the *gradient descent with momentum* algorithm and the *RMSP* algorithm. Momentum is used to accelerate the gradient descent algorithm by taking into consideration the exponentially weighted average of the gradients so that it can converge towards the minima at a faster pace.

$$w_{(t+1)} = w_t - \alpha m_t \text{ where } m_t = \beta m_{t-1} + (1-\beta)[\delta L/(\delta w_t)]$$

m_t = aggregate of gradients at time t; w_t = weights at time t; α = learning rate; δL = derivative of loss function; δW = derivative of weights at time t; β = moving average parameter.

5.2 Loss Function

The loss function in NN quantifies the difference between the expected outcome and the outcome produced by the machine learning model. Starting from that, it is possible to derive the gradients which are used to update the weights. In the project, being a multi-class classification problem, the Keras method *Sparse Categorical Crossentropy* is chosen.

```
model.compile(loss = tf.keras.losses.SparseCategoricalCrossentropy
    (),optimizer = tf.optimizers.Adam(learning_rate = lr), metrics
    =['accuracy'])
```

Sparse is used because the input labels are not one-hot encoded but integers. The formula of the Categorical Crossentropy is:

$$\frac{-1}{N}\sum_{n=1}^{N}\log P_{model}[y_i \in C_{y_i}]$$

where C_{y_i} represents the classes and y_i is the predicted label

Chapter 6

Layers

All layers are added to a Sequential model with

```
tf.keras.Sequential()
```

where Sequential means that the computation proceeds from the input layer progressively toward the output layer by following the topological order in the network. So, it is a linear stack of layers without any backward link.

6.1 Convolutional Layer

This layer creates a convolution kernel that is convolved with the layer input to produce a tensor of outputs (img 6.1). It is the main building block of a CNN. It contains a set of filters (or kernels), that are parameters that the network learns during the training. The output volume of the convolutional layer is generated by stacking the activation maps of every filter along the depth dimension. Due to the local connectivity of the convolutional layer, the network is forced to learn filters that have the maximum response to a local region of the input.

- **filters**: it specifies the dimensionality of the output space (i.e. the number of output filters in the convolution).
- **kernel_size**: it specifies the height and width of the 2D convolution window. Usually, the size of the filters is smaller than the actual image. Each filter convolves with the image and creates an activation map. For convolution the filter slid across the height and width of the image and the dot product between every element of the filter and the input is calculated at every spatial position.
- **strides**: it specifies the strides of the convolution along with the height and width. It is a parameter that modifies the amount of movement over the image or video.

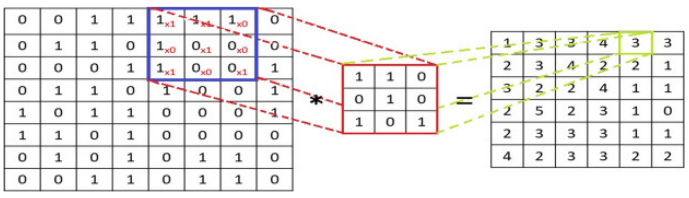


Figure 6.1: Example of how the convolution works in a Convolutional layer

- **padding**: it specifies the type of padding to apply. *same* results in padding with zeros evenly to the left/right or up/down of the input. Padding is needed to compute convolution also along with the perimetral values, increasing the effective size of the original image.

6.2 MaxPooling

It is a max-pooling operation for 2D spatial data. It means that the input is down-sampled along its spatial dimensions (height and width) by taking the maximum value over an input window, for each channel of the input. The window is then shifted by strides along each dimension. (img 6.2)

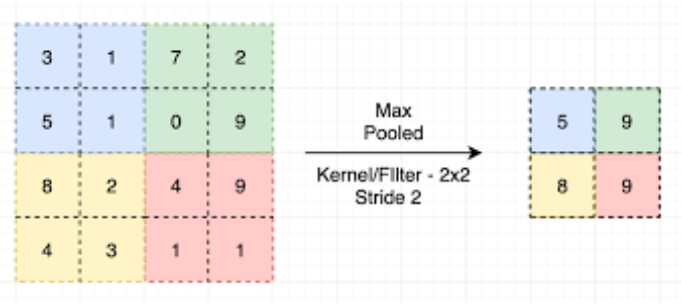


Figure 6.2: Example of Max Pooling with filter_size = 2 and stride 2

- **pool_size**: it specifies the window size over which to take the maximum.
- **strides**: as for the convolutional layer, it specifies how far the pooling window moves for each pooling step.
- **padding**: as for the convolutional layer, it is needed to compute the pooling on the borders without losing information. *same* results in padding evenly to the left/right or up/down of the input such as the output has the same height/width dimension as the input

6.3 Flatten

This layer flattens the input without affecting the batch size. It is needed as a pre-processing step to obtain 1D data from more dimensional data, to input the dense layer. If inputs are shaped without a feature axis, then flattening adds an extra channel dimension and the output shape will include the batch size.(img 6.3)

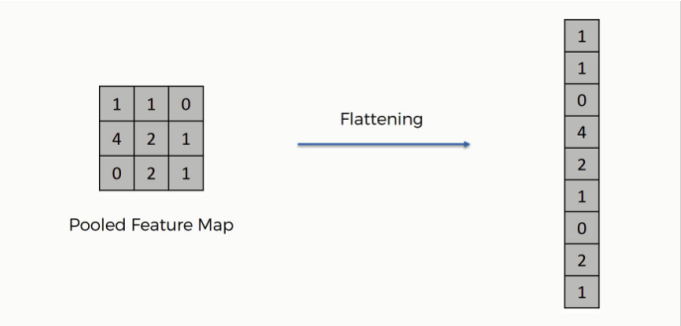


Figure 6.3: Example of a Flatten Layer

6.4 Dense

A dense layer is the regular deeply connected neural network layer. Dense implements the operation: $output = activation(dot(input, kernel) + bias)$ where *activation* is the element-wise activation function passed as the activation argument; a *kernel* is a weights matrix created by the layer, and *bias* is a bias vector created by the layer (only applicable if use_bias is True).

- **units**: it specifies the dimensionality of the output space.
- **activation**: it specifies the activation function to use. If you don't specify anything, no activation is applied.

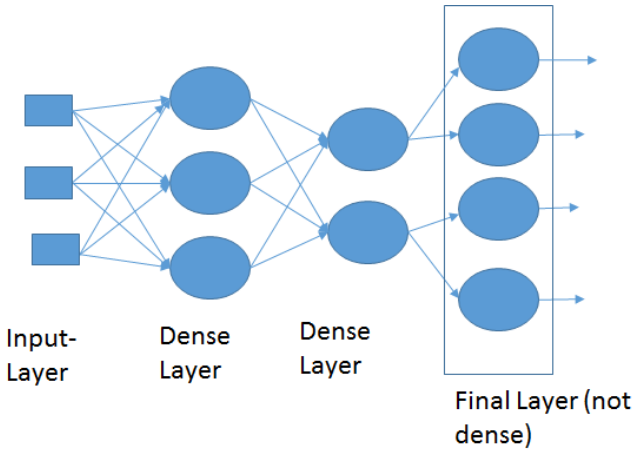


Figure 6.4: Example of a Dense Layer

In the project, **ReLu** activation is used for almost all layers. ReLu stands for Rectified Linear Unit and is a non-linear activation function that returns the input directly if positive or 0 if negative (img 6.5). The main advantage is that it doesn't activate all the neurons at the same time.

For the last dense layer instead **SoftMax** activation is used (img 6.6). Softmax is a mathematical function that converts a vector of numbers into a vector of probabilities, where the probabilities of each value are proportional to the relative scale of each value in the vector. Specifically, the network is configured to output N values, one for each class in the classification task, and the softmax function is used to normalize the outputs, converting them from weighted sum values into probabilities that sum to one. Each value in the output of the softmax function is interpreted as the probability of membership for each class.

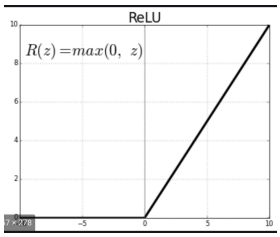


Figure 6.5: Graphic representation of ReLu activation

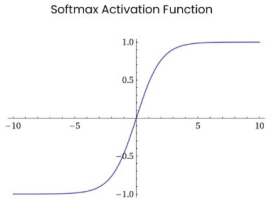
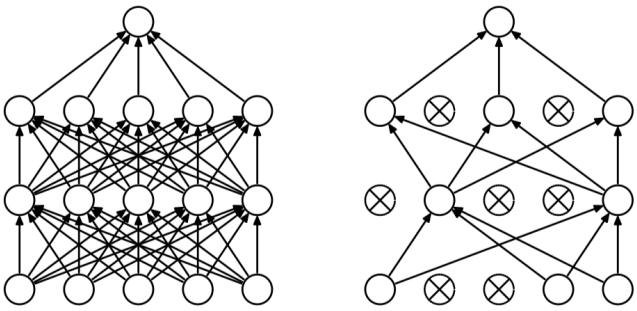


Figure 6.6: Graphic representation of Softmax activation

6.5 Dropout

This layer applies Dropout to the input. It randomly sets input units to 0 with a frequency of rate at each step during training time, which helps prevent overfitting. Inputs not set to 0 are scaled up by $1/(1 - rate)$ such that the sum over all inputs is unchanged. (img 6.7)

- **Rate**: a number between 0 and 1 that determines the number of neurons to turn off.



(a) Standard Neural Net (b) After applying dropout.

Figure 6.7: Example of Dropout

Chapter 7

Hyperparameters

A possible type of hyperparameters could be the different input values passed as parameters to Librosa functions, to obtain different shapes in the output features. The only features that were extracted twice are the mfccs coefficients, one with 15 and another one with 25 as the number of coefficients to use. This was done to see if an increase in the number of coefficients leads to an increase in the performances, so if more information is included, or if, instead, the bigger computation effort isn't followed by a higher accuracy.
While training the other Neural Networks, the hyperparameters that should be considered are:

- **Batch size**: it defines the number of samples that will be propagated through the network before updating the weights of the network. It allows to use a less amount of memory and the training is usually faster because of a lower number of samples and because weights are updated only after the propagation. In the project this is fundamental due to the limited amount of memory of the used machine. The problem is a less accurate estimate of the gradient while reducing the batch size.

- **Validation Split**: it represents the portion of the training set to be used for validation. Being the training set already not that big, the validation split is chosen to be steady at 0.1, to allow the Network to learn as much information as possible.

- **Number of epochs**: it defines the number of times that the learning algorithm works through the entire training Dataset. One epoch means that each sample in the training Dataset has had an opportunity to update the internal model parameters. An epoch is made of one or more batches.

- **Learning Rate**: (also called step size) it controls how much to change the model in response to the estimated error each time the model weights are updated. If it is too small may result in a long training process that could get stuck, whereas a value too large may result in learning a sub-optimal set of weights too fast or an unstable training process. The choice applied in the project is to start with a learning rate of 0.001.

To deal with the number of epochs and the learning rate, the callbacks parameter is added when fitting the model. A callback is an object that can perform actions at various stages of training (e.g. at the start or end of an epoch, before or after a single batch, etc). It can be used to customize the training process.

```
history = model.fit(Xtrain, Ytrain, batch_size=batch_size, epochs=
    n_epochs, verbose='auto', callbacks=[earlyStopping, mcp_save,
    reduce_lr_loss], validation_split=0.1, shuffle=True)
earlyStopping = EarlyStopping(monitor='val_loss',patience=20,
    verbose=0,mode='min')
mcp_save = ModelCheckpoint('.mdl_wts.hdf5', save_best_only=True,
    monitor='val_loss', mode='min')
reduce_lr_loss = ReduceLROnPlateau(monitor='val_loss', factor=0.1,
    patience=7, verbose=1, min_delta=1e-4, mode='min')
```

- **ReduceLROnPLateau**: it adjusts the learning rate when a plateau in model performance is detected, e.g. no change for a given number of training epochs. This callback is designed to reduce the learning rate after the model stops improving with the hope of fine-tuning model weights.
  - Monitor: is the metric to monitor during the training.
  - Factor: is the argument the learning rate is multiplied by when changing, so it is reduced by 1/10 every time.
  - Patience: Specifies the number of the training epochs to wait before triggering the change in learning rate
  - Mindelta: threshold for measuring the new optimum, to only focus on significant changes.
  - Mode: In 'min' mode, the learning rate is reduced when the quantity monitored stops decreasing.
- **EarlyStopping**: to avoid overfitting, it stops training when a monitored metric has stopped improving. A *model.fit()* training loop checks at the end of every epoch whether the metric that has to be improved is no longer decreasing/increasing; once it's found no longer improving, *model.stop_training* is marked True and the training terminates. The parameters are the same as written above for *ReduceLROnPlateau*. The quantity to be monitored needs to be available in logs dict. To make it so, the loss or metrics have to be passed at *model.compile()*.
- **ModelCheckPoint**: it is used in conjunction with training using *model.fit()* to save a model or weights (in a checkpoint file) at some interval, so the model or weights can be loaded later to continue the training from the state saved.
  - Save_best_only: Whether to only keep the model that has achieved the "best performance" so far, or whether to save the model at the end of every epoch regardless of performance. 'Best' is related to the quantity to monitor and if it should be minimized or maximized.
  - The other parameters have the same meaning as explained before.

Chapter 8

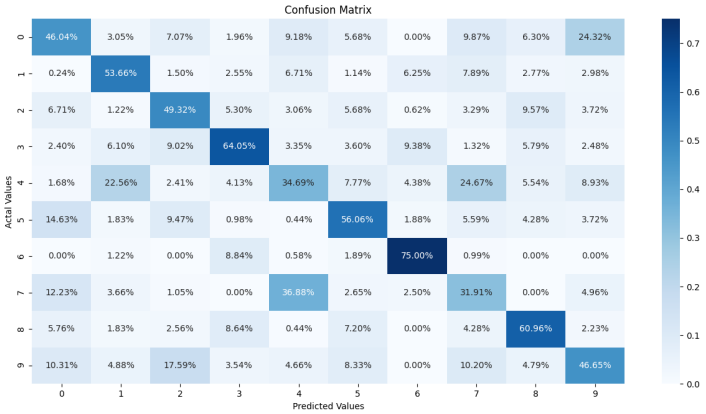
Architectures

For each neural network, all features are used and then compared. For what concerns mfcc, two versions are tested, one extracting 15 coefficients and one with 25 coefficients in order to see if there's an improvement in terms of accuracy. Then, a combination of features is chosen. First MFCCs (the version with 15 coefficients) + Spectrogram, then MFCCS + Chromagram, and then all three combined with a PCA (n_components = 50) pre-processing step to reduce their dimensionality. For each of these modalities, two batch sizes are tried, 64 and 128, but in the report just the confusion matrix related to the batch size with the best performance is included and the corrisponding accuracy plot. All other images can be found in their specific folder on GitHub.

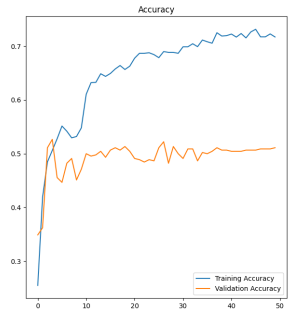
8.1 Artificial Neural Network

The first is an easy model inserted to try to prove the difference in terms of performances between a fully connected Multilayer perceptron and a Convolutional neural network (which we expect should work better)

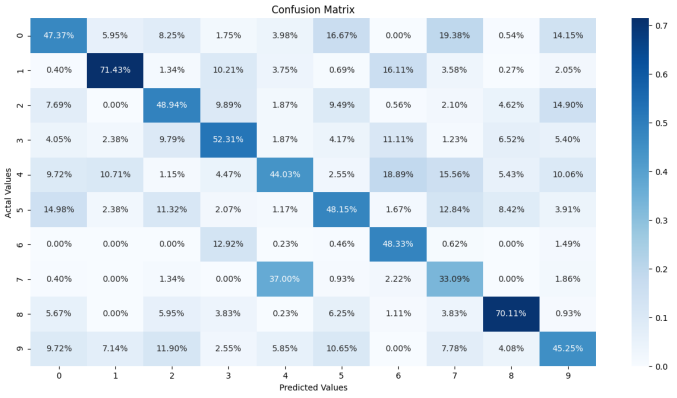
```
denseModel = tf.keras.Sequential([
    layers.Flatten(input_shape=(params['dim'][1], params['dim'][2],
    params['n_channels'])),
    layers.Dense(1024, activation='relu'),
    layers.Dropout(0.5),
    layers.Dense(2048, activation='relu'),
    layers.Dropout(0.5),
    #layers.Dense(params['dim']/2, activation='relu'),
    #layers.Dense(params['n_classes'], activation='Softmax'),
    layers.Dense(params['n_classes'],activation='softmax'),
])
```



(a) Confusion Matrix of MFCC with 15 coefficients. BS = 128

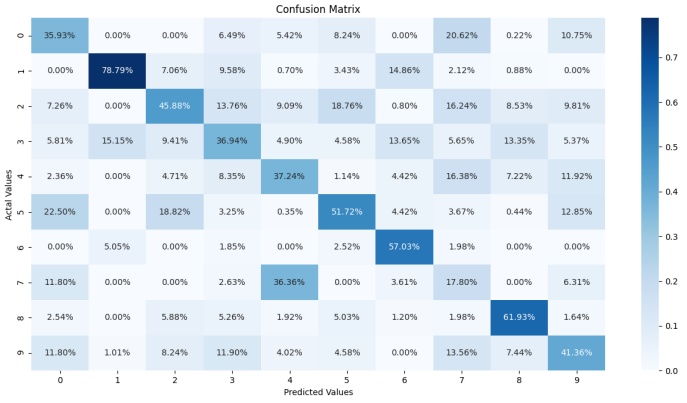


(b) Accuracy plot of MFCC with 15 coefficients. BS = 128

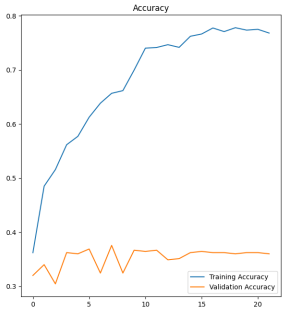


(c) Confusion Matrix of MFCC with 25 coefficients. BS = 64

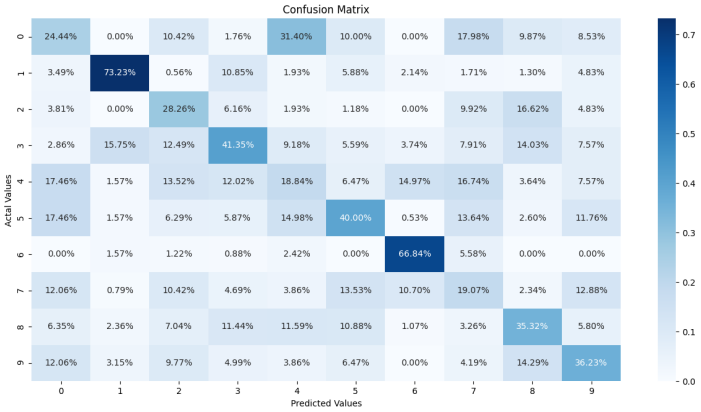
(d) Accuracy plot of MFCC with 25 coefficients. BS = 64



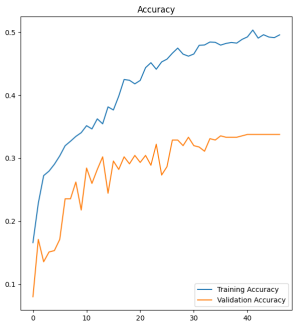
(e) Confusion Matrix of MelSpectrogram. BS = 64



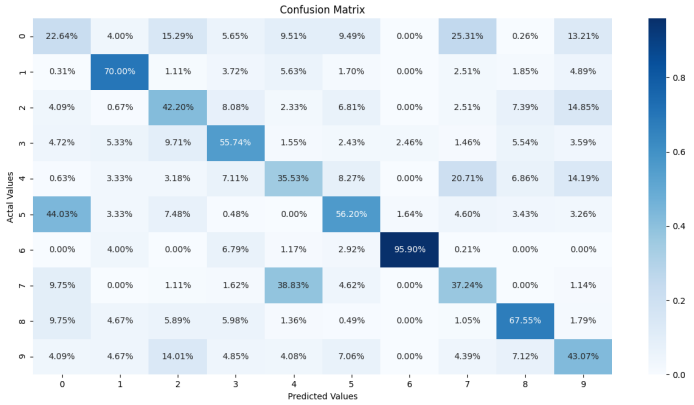
(f) Accuracy Plot of MelSpectogram. BS = 64



(g) Confusion Matrix of Chroma. BS = 128

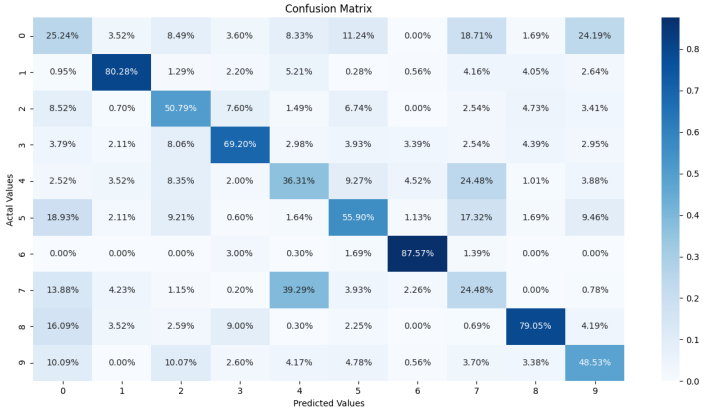


(h) Accuracy plot of Chroma. BS = 128

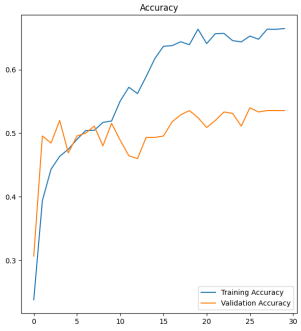


(i) Confusion Matrix of MFCC + MelSpect. BS = 128

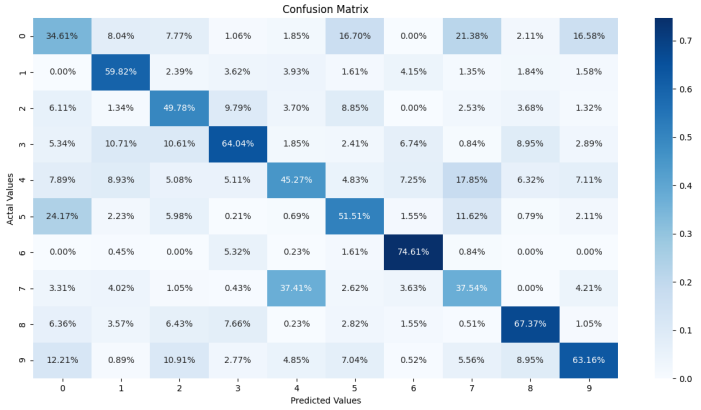
(j) Accuracy plot of MFCC + MelSpect. BS = 128



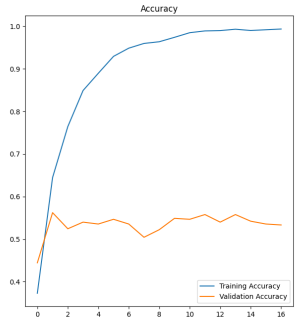
(k) Confusion Matrix of MFCC + Chroma. BS = 128



(l) Accuracy plot of MFCC + Chroma. BS = 128



(m) Confusion Matrix of PCA (50 components) of MFCC + MelSpect + Chroma. BS = 128



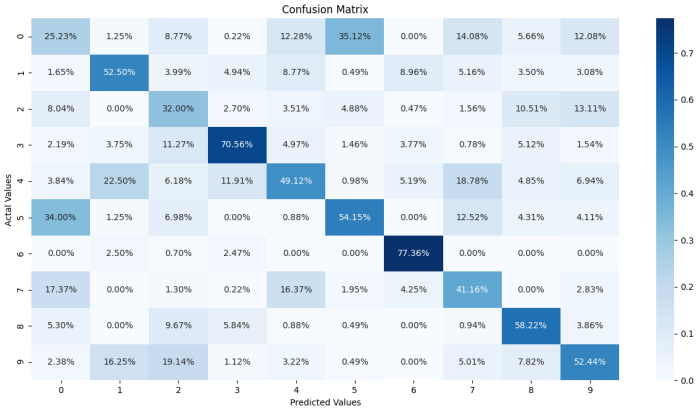
(n) Accuracy plot of PCA (50 components) result of MFCC + MelSpect + Chroma. BS = 128

Figure 8.1: Confusion Matrix + accuracy plot using an Artificial Neural Network for all features and their combination

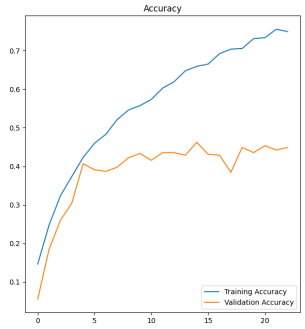
8.2 Convolutional Neural Network

In this second Neural network, some convolutional layers are added. The obtained features can be seen as images and so a 2D convolution can try to detect a correlation between them. The more number of layers are inserted, the more higher level features can be extracted, to learn a better representation of the data. The cost of this is a higher computational cost.

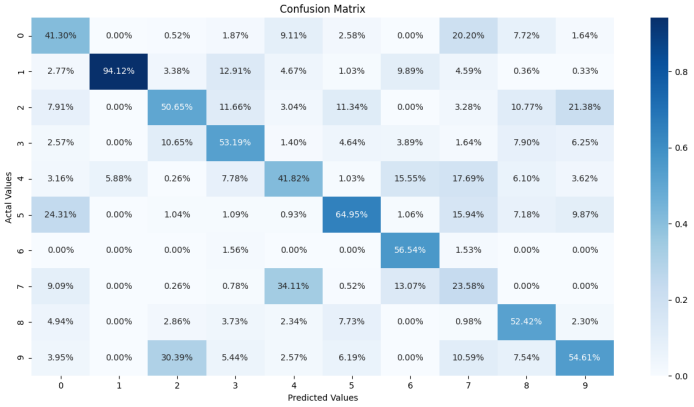
```
model2D = tf.keras.Sequential([
        layers.Conv2D(32, kernel_size=3, activation='relu',
    input_shape=(params['dim'][1], params['dim'][2], params['
    n_channels']),padding='same', strides=1),
        layers.MaxPooling2D(pool_size=(2),padding='same'),
        layers.Dropout(0.5),
        layers.Conv2D(64, kernel_size=3, activation='relu',strides
    =1,padding='same'),
        layers.MaxPooling2D(pool_size=(2),padding='same'),
        layers.Dropout(0.3),
        layers.Conv2D(64, kernel_size=3, activation='relu',strides
    =1,padding='same'),
        layers.MaxPooling2D(pool_size=(2),padding='same'),
        #layers.Conv2D(128, kernel_size=3, activation='relu',
    strides=1,padding='same'),
        layers.Dropout(0.5),
        layers.Flatten(),
        layers.Dense(64, activation='relu'),
        layers.Dropout(0.5),
        layers.Dense(32, activation='relu'),
        layers.Dense(params['n_classes'], activation='softmax'),
])
```



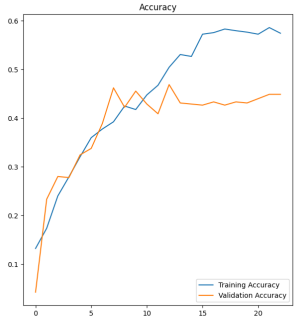
(a) Confusion Matrix of MFCC with 15 coefficients. BS = 64



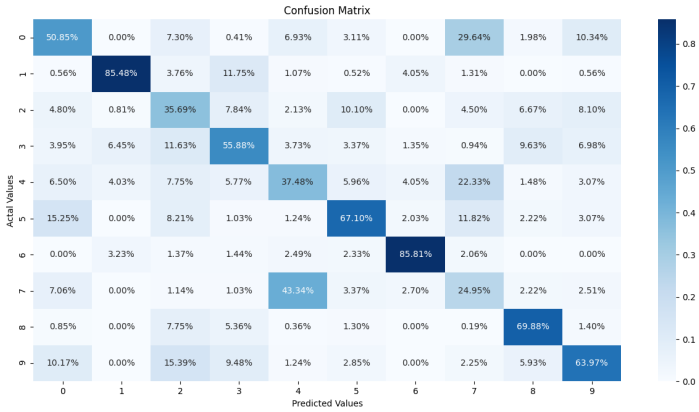
(b) Accuracy plot of MFCC with 15 coefficients. BS = 64



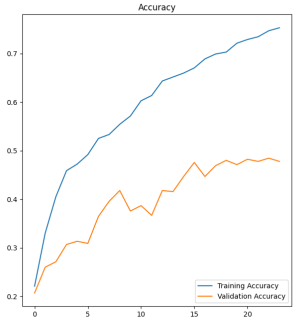
(c) Confusion Matrix of MFCC with 25 coefficients. BS = 128



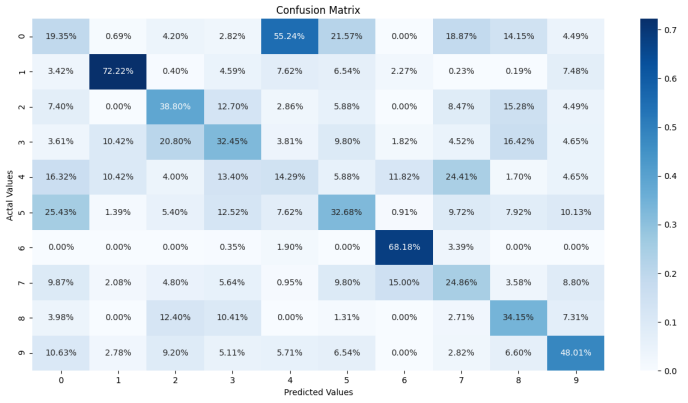
(d) Accuracy plot of MFCC with 25 coefficients. BS = 128



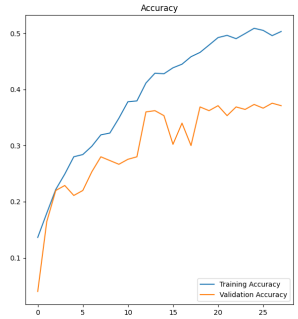
(e) Confusion Matrix of MelSpectrogram. BS = 64



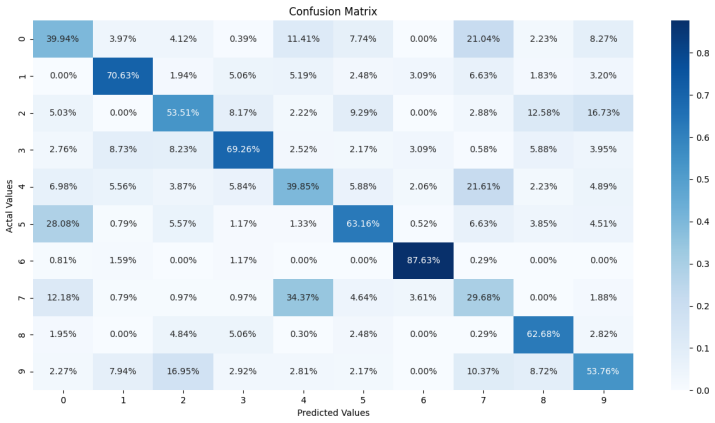
(f) Accuracy Plot of MelSpectogram. BS = 64



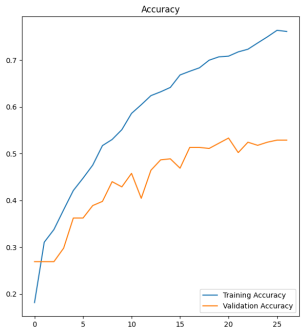
(g) Confusion Matrix of Chroma. BS = 128



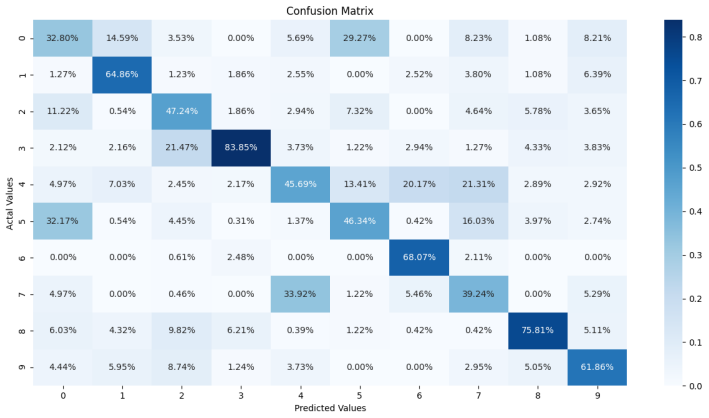
(h) Accuracy plot of Chroma. BS = 128



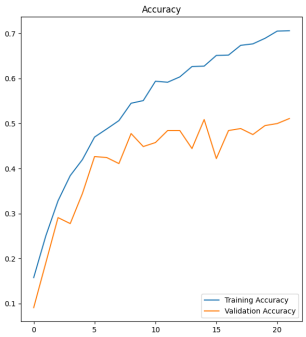
(i) Confusion Matrix of MFCC + MelSpect. BS = 128



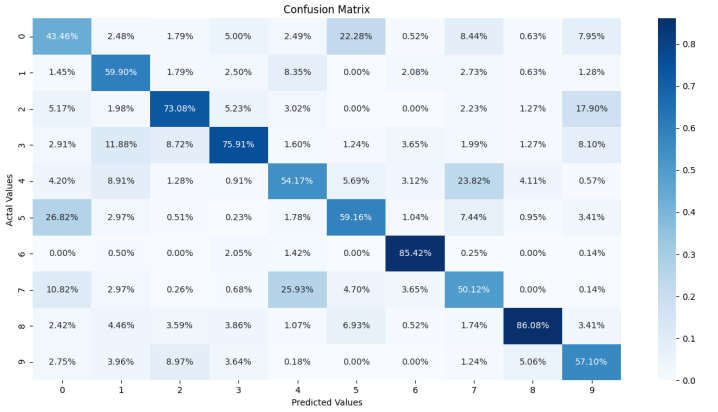
(j) Accuracy plot of MFCC + MelSpect. BS = 128



(k) Confusion Matrix of MFCC + Chroma. BS = 128



(l) Accuracy plot of MFCC + Chroma. BS = 128



(m) Confusion Matrix of PCA (50 components) of MFCC + MelSpect + Chroma. BS = 128

(n) Accuracy plot of PCA (50 components) result of MFCC + MelSpect + Chroma. BS = 128

Figure 8.2: Confusion Matrix + accuracy using a Convolutional Neural Network. All features and their combinations

8.3 Transfer Learning

Transfer learning is a valid alternative when data to train the CNN aren't sufficient or not enough computational resources to train the CNN are available. Because usually the first layers of the networks extract low-level features, even if the final task is different these low-level features could be exploited. What has to change is the final layer, to adapt the Network for the proposed target.
In this project, the pre-trained VGG network is used. The basic idea is to abstract from neurons and layers to blocks and later concatenate them. Each block is composed of convolutional layers followed by max-pooling operations with different filter sizes. (img 8.4)

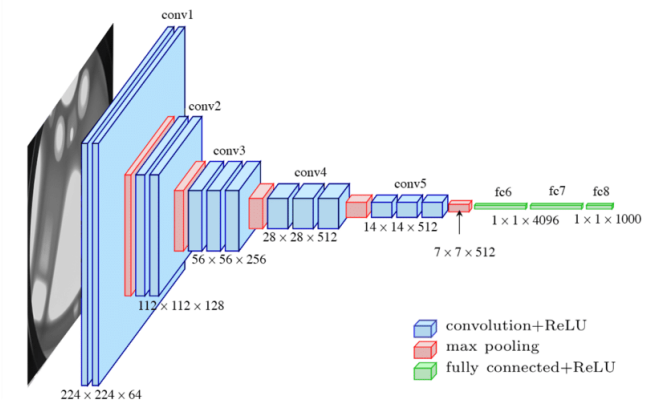


Figure 8.3: VGG architecture

Because originally it was made for image classification, the required input needs to have three channels. To overcome this issue, each matrix of features is duplicated two times to satisfy the requirements.

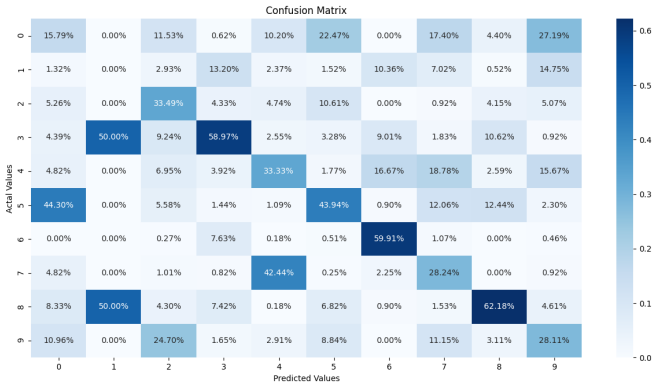
```
#To apply also for all test folders
sel_feat=np.repeat(sel_feat[..., np.newaxis], 3, axis=3)

vgg16_model = tf.keras.applications.vgg16.VGG16(weights='imagenet',
    include_top = False, input_shape=(params['dim'][1], params['dim
    '][2], params['n_channels']))
```

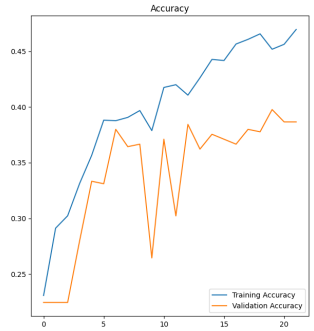
Starting from that, the model has to be adapted to the sound classification task. First, all the layers from the Vgg model are added to the custom model except for the last one. Then, they are frozen so that the weights can't be trained or modified, to be used as they are. Finally, four layers are added, to try to train the model focusing on this Dataset and then to have the right type of classification.

```
vgg16_custom_model.add(layers.Flatten(name='lastFlatten'))
vgg16_custom_model.add(layers.Dense(256, activation='relu'))
vgg16_custom_model.add(layers.Dropout(0.5))
vgg16_custom_model.add(layers.Dense(params['n_classes'], activation
    ='softmax'))
```

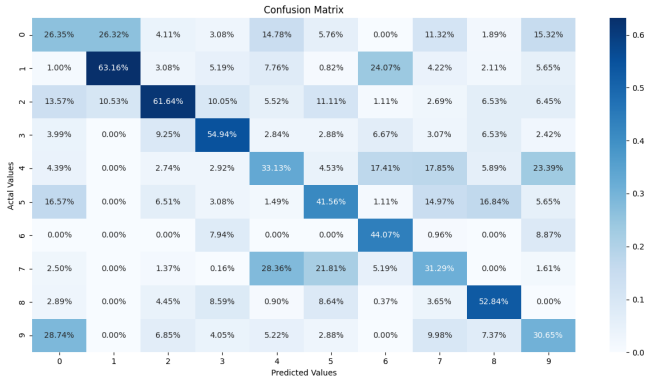
Chromagram features alone can't be used with this model because the minimum input size require is 32x32 while Chroma's first dimension is 12. Because of the bad performances using this model in general, no reshape of any kind is applied and they are just skipped, not being considered that relevant.



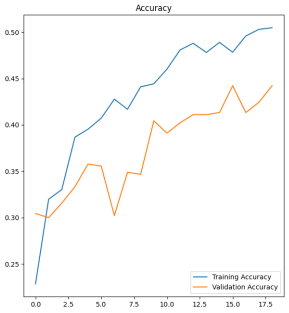
(a) Confusion Matrix of MFCC with 15 coefficients. BS = 64



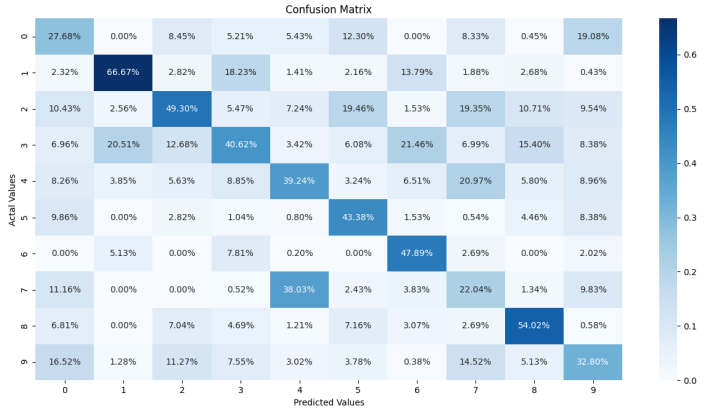
(b) Accuracy plot of MFCC with 15 coefficients. BS = 64



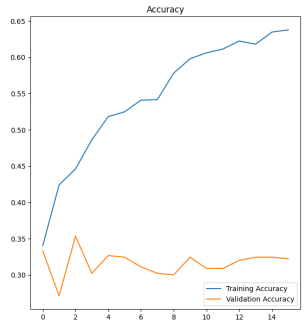
(c) Confusion Matrix of MFCC with 25 coefficients. BS = 128



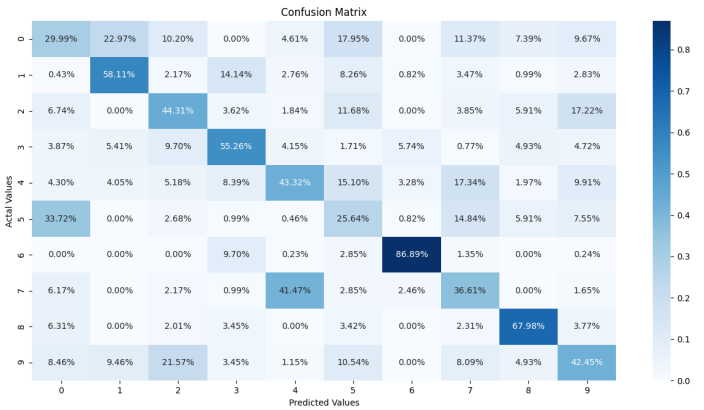
(d) Accuracy plot of MFCC with 25 coefficients. BS = 128



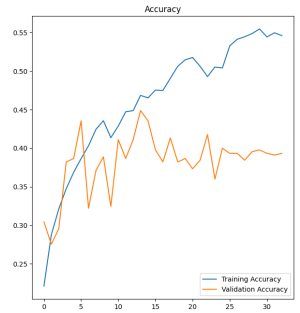
(e) Confusion Matrix of MelSpectrogram. BS = 64



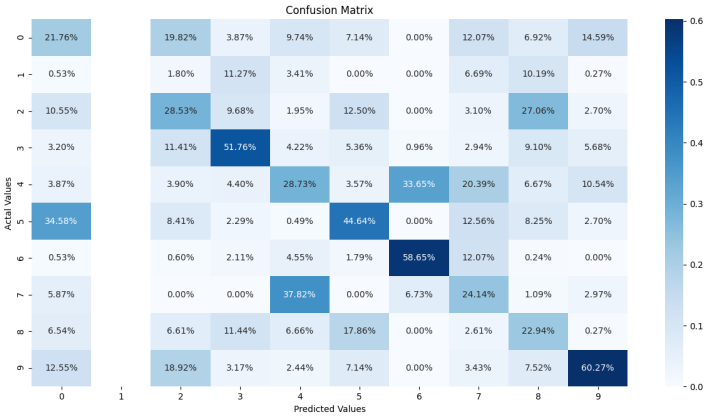
(f) Accuracy Plot of Mel-Spectogram. BS = 64



(g) Confusion Matrix of MFCC + MelSpect. BS = 128

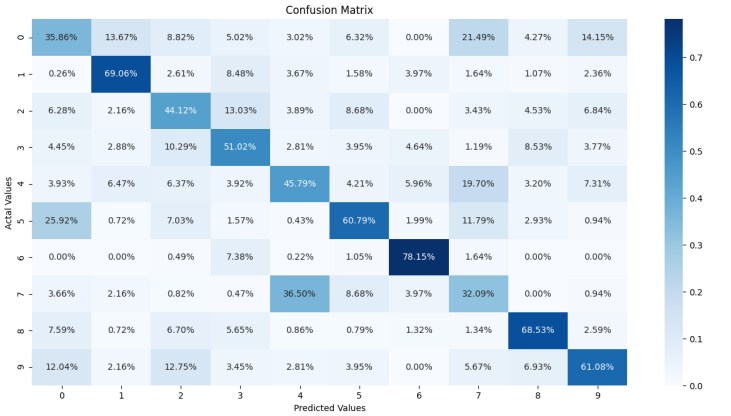


(h) Accuracy plot of MFCC + MelSpect. BS = 128

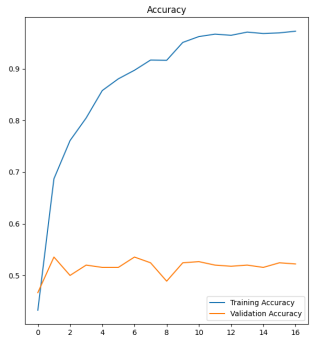


(i) Confusion Matrix of MFCC + Chroma. BS = 128

(j) Accuracy plot of MFCC + Chroma. BS = 128



(k) Confusion Matrix of PCA (50 components) of MFCC + MelSpect + Chroma. BS = 64



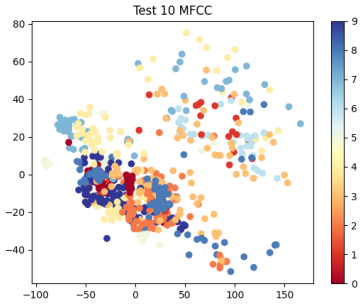
(l) Accuracy plot of PCA (50 components) result of MFCC + MelSpect + Chroma. BS = 64

Figure 8.4: Confusion Matrix and relative accuracy using transfer learning on VGG16 with different features or combinations of them

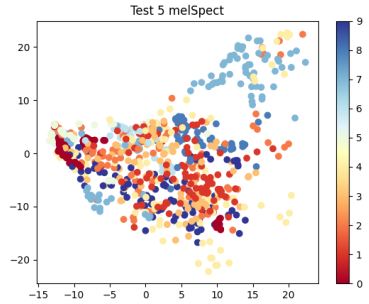
Chapter 9

Feature Distribution

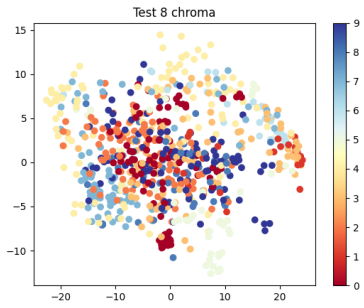
Plotting data samples as extracted features with only 2 dimensions is not fully representative of the data, because they could be split among other dimensions. Anyway, this could give an idea of how blobs of points are located in the 2D space, if they can be grouped together or if they are sparse around the plot.
This could be linked with the class accuracy in the reported confusion matrix, where sometimes is possible to see a correlation between high or low accuracy and how respective samples are distributed. Some plots are reported as an example, while the others can be found in the related folder on GitHub



(a) MFCC samples distribution in folder 10



(b) MelSpec samples distribution in folder 5



(c) Chroma samples distribution in folder 8

Figure 9.1: Examples of features distribution after a 2D projection across folders

Chapter 10

Results

Mean accuracy and std across all test folders

Feature	BS	ANN	CNN	VGG
MFCC_15	64	49.20% ± 5.78	46.07% ± 4.70%	39.55% ± 5.84%
MFCC_15	128	50.11% ± 3.25	45.42% ± 3.69%	39.02% ± 5.44%
MFCC_25	64	47.67% ± 5.71	44.14% ± 3.11%	36.78% ± 7.32%
MFCC_25	128	45.13% ± 3.62	45.04% ± 3.37%	39.93% ± 4.54%
MelSpec	64	40.78% ± 4.41%	49.88% ± 3.74%	38.32% ± 2.94%
MelSpec	128	39.50% ± 4.40%	42.27% ± 2.12%	38.31% ± 5.49%
Chroma	64	22.83% ± 4.45%	34.82% ± 3.23%	-
Chroma	128	32.91% ± 4.12%	35.15% ± 3.98%	-
MFCC+Mel	64	47.26% ± 2.17%	49.91% ± 6.09%	42.81% ± 5.99%
MFCC+Mel	128	47.73% ± 4.25%	53.39% ± 3.76%	44.64% ± 4.74%
MFCC+Chroma	64	47.10% ± 2.47%	47.23% ± 5.93%	24.06% ± 2.27%
MFCC+Chroma	128	50.64% ± 2.65%	51.49% ± 3.02%	32.50% ± 2.59%
All+PCA	64	49.30% ± 5.55%	61.00% ± 3.05%	50.23% ± 6.34%
All+PCA	128	52.51% ± 4.99%	61.25% ± 2.50%	47.15% ± 5.85%

Table 10.1: Table of comparison across all models and all features with both batch size. The values are the average mean and std considering all test folders

Looking at table 10.1 is possible to see that the best performances are obtained by exploiting MFCC MelSpectrogram and Chroma features together, applying a PCA Analysis, and using a CNN.
Once that results were observed, another tentative was done with a higher number of PCA components, 100 (img 10.1). That allowed to increase classification accuracy per each class, even class 0 [Air_conditioner] and 7 [Jackhammer], which usually are the ones most difficult to detect.
Chroma features are those that perform worst, also because of their smaller shape (12,345) with respect to the others. But if combined with MFCCs, the CNN can produce a better result, so this means that the information it carries could still be important for the classification. Generally, the combination of different features shows to gain the best results, because more information is available and can be used to split samples into classes.
The project presents different kinds of architectures and, as assumed, the use of a Convolution Neural Network is a good choice also for sound classification.
However, the use of VGG, even if it is based on convolutional blocks, doesn't improve the overall accuracy. The reason could probably be that it is pre-trained on images

and these features have a completely different aspect, so it could be difficult to find a common pattern. The accuracy plot also proves that sometimes the Network is not even learning from the data or that random values are generated after the VGG layers and the model starts to learn within the last added ones.
MFCC alone gains the best performance with the simple ANN, without any convolution, while MelSpectogram alone in the CNN.
Class 4 never reaches a peak of accuracy and this can be seen also having a look at its distribution in space, which is rarely located in a single area.
Class 1 [car horn] has a peak of accuracy in the CNN using MFCC with 25 coefficients as input, with a 94.12% of accuracy.
Class 6 [gun_shot] reaches the best accuracy in the ANN using MFCC combined with Chroma features, with a 93.55% of accuracy.
Class 0 [Air_conditioner] is often confused, for example with 4 [Drilling], 5 [Engine_idling], or 7 [Jackhammer] as some confusion matrices prove in both ANN and CNN models.
So, classes 0 [Air_conditioner],4 [Drilling] and 7 [Jackhammer] are those whose accuracy on average is lower. This is reflected also in their space distribution, because their blobs don't have a specific location while for example in fig 9.1a it's possible to observe that samples of class 9 [Street_music] or 8 [Siren] have a more defined area. When considering the best model, all classes except for 7[Jackhammer] have an accuracy percentage above 50% and classes, 3 [dog_barking], 5 [Engine_idling], 6 [Gun_shot] and 8 [Siren] have an accuracy above 70%.

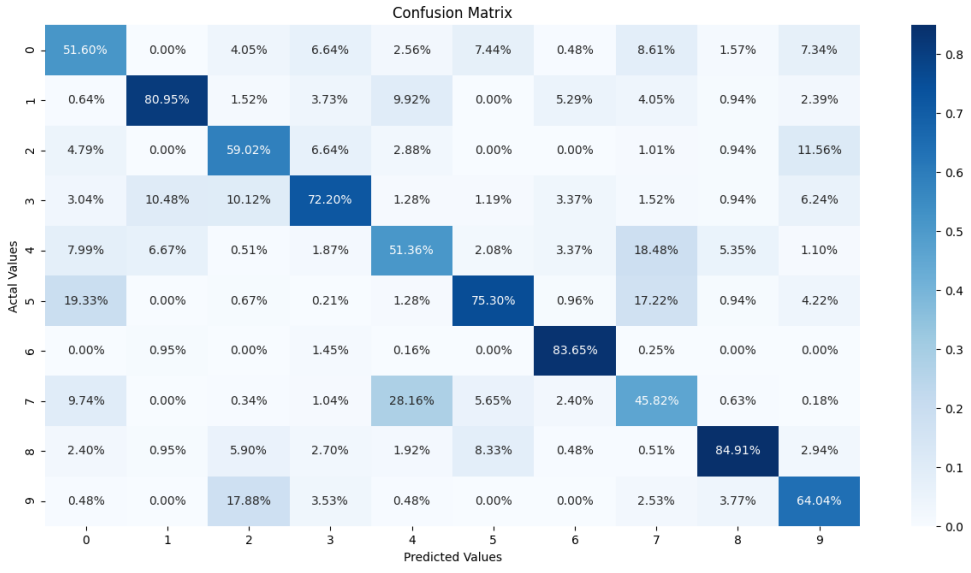


Figure 10.1: Confusion Matrix of the model with the best average accuracy, All features combined + PCA Analysis (100 components)

Bibliography

[1] Brian McFee, Colin Raffel, Dawen Liang, Daniel P.W. Ellis, Matt McVicar, Eric Battenberg, and Oriol Nieto. librosa: Audio and Music Signal Analysis in Python. In Kathryn Huff and James Bergstra, editors, *Proceedings of the 14th Python in Science Conference*, pages 18 – 24, 2015.

[2] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122, 2013.

[3] Francois Chollet et al. Keras, 2015.

[4] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, September 2020.

[5] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[6] Zakaria Jaadi. A step-by-step explanation of principal component analysis (pca), 2021.

[7] Wes McKinney et al. Data structures for statistical computing in python, pandas. In *Proceedings of the 9th Python in Science Conference*, volume 445, pages 51–56. Austin, TX, 2010.

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

[9] Guido Van Rossum. *The Python Library Reference, release 3.8.2, Pickle.* Python Software Foundation, 2020.

[10] Wikipedia contributors. Mel-frequency cepstrum — Wikipedia, the free encyclopedia, 2022.

[11] Wikipedia contributors. Mel-frequency cepstrum — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Mel-frequency_cepstrum&oldid=1088378317`, 2022. [Online; accessed 21-May-2022].