Using CNN Networks for Music Genre Classification

Matteo Arrigo
Politecnico di Milano

June 23, 2025

Contents

1 Introduction

Music genre classification is a fundamental task in music information retrieval, with applications spanning from recommendation systems to music analytics and automated tagging. Traditionally, genre classification relied on handcrafted features such as timbre, rhythm, and harmonic content, extracted from raw audio signals. However, with the advent of deep learning, convolutional neural networks (CNNs) have emerged as a powerful tool for feature extraction, enabling end-to-end learning from raw data.

Recent studies have explored the use of **Mel-spectrograms**, a time-frequency representation of audio signals, as input to CNN architectures. These spectrograms transform sound into an image-like format, allowing computer vision models to be leveraged for classification tasks.

This report aims to reproduce the findings of a research paper [2] that employs the **You Only Look Once (YOLO)** architecture, initially designed for real-time object detection in images, by repurposing it to classify spectrograms with remarkable accuracy. Specifically, the report uses the **YOLOv4 model** for music genre classification based on Mel-spectrograms. The goal is to evaluate the effectiveness of YOLO family models in this domain, comparing its performance with conventional CNN models and analyzing different training strategies and implementations.

1.1 Motivation and Challenges

Music genre classification presents unique challenges due to:

- **Variability in musical features**: Unlike images, music has complex temporal and harmonic structures that make classification difficult.
- **Limited labeled datasets**: Unlike large-scale image datasets, annotated music datasets are relatively small, posing a risk of overfitting.
- **Spectrogram representations**: While Mel-spectrograms provide rich information, their interpretation by CNNs depends on proper preprocessing and augmentation techniques.

By addressing these challenges, this study seeks to assess **whether object detection models like YOLOv4** offer advantages over traditional CNN classifiers in the context of spectrogram-based music genre classification.

1.2 Premises on the Paper

It must be said that Yu-Huei Cheng and Che-Nan Kuo, the authors of the paper on which this project is based, are very promising in terms of the results they achieved, but not very detailed in explaining how they achieved them.

They use the mAP metric to evaluate their model and claim to obtain an average mAP greater than 97%.

However, while the description of the conversion method is quite precise, the description of the model itself is rather poor.

- The YOLOv4 architecture is presented at a high level in terms of its basic components, but no explanation is given on how these components are connected.
- Many explanations are overly general — such as how convolution and pooling layers work, or what an activation function is — resulting unnecessary in an already technical context and not helping in understanding how their specific model works.
- More advanced techniques are mentioned, but again in a general manner, without clarifying whether or how they were used in the actual implementation.
- It is unclear how they adapted the YOLO architecture — originally designed for object detection — to a classification problem.
- The training procedure is not explained. In particular, crucial pieces of information, such as the regularization techniques used, the optimizer, and the learning rate value or scheduling strategy, are missing.

In the end, I personally found it impossible to replicate the authors' work due to the lack of information about the model. As a result, after attempting to follow their paper closely in the early stages, I eventually developed my own model, which differs significantly from theirs both in structure and in the results I was able to achieve.

2 Basic Techniques

Early methods to tackle music genre classification employed signal processing techniques to compute descriptors such as zero-crossing rate, spectral centroid, MFCCs (Mel-frequency cepstral coefficients), and rhythmic patterns. These features were then used in classical machine learning algorithms like support vector machines (SVMs), k-nearest neighbors (k-NN), and decision trees.

Nowadays, convolutional neural networks (CNNs) and general deep learning techniques have become the preferred choice for this and many other tasks. This is an effective way to tacke the problem especially when paired with time-frequency representations, by converting audio signals into 2D visual patterns.

2.1 Mel-Spectrogram Representation

The **Mel-spectrogram** is a time-frequency representation of audio signals that maps the spectrum of sound onto the Mel scale, a perceptual scale that approximates the way humans perceive pitch. Unlike a standard spectrogram, which represents frequency on a linear scale, the Mel-spectrogram compresses high-frequency content and emphasizes the lower and mid-frequency ranges where most perceptually relevant audio features lie.

To compute a Mel-spectrogram, the audio waveform is first divided into overlapping frames using a short-time Fourier transform (STFT), which captures how the frequency content of the signal evolves over time. Each windowed segment is transformed into the frequency domain using the Fast Fourier Transform (FFT). Then, the linear frequency axis of the resulting spectrogram is projected onto the Mel scale using a filter bank composed of triangular filters spaced according to the Mel-frequency formula:

$$\text{Mel}(f) = 2595 \cdot \log_{10}\left(1 + \frac{f}{700}\right)$$

This transformation makes the representation more aligned with human auditory perception, as it provides higher resolution at lower frequencies and reduced resolution at higher frequencies. The resulting 2D matrix (time $\times$ Mel frequency) is typically converted to decibels (dB) to compress the dynamic range, enhancing the visibility of quieter components when visualized as an image.

The benefit of using Mel-spectrograms is that they preserve both the temporal and spectral structure of audio, enabling convolutional neural networks—originally developed for images—to learn meaningful patterns from sound data, without the need to use other more complicated models such as recurrent neural networks to take into account the time.

This approach effectively transforms the genre classification task into an image classification problem, which allows leveraging advances in computer vision architectures for music analysis.

2.2 CNN Architectures

Several studies explored various CNN architectures, including VGG, ResNet and Inception, trained on spectrogram images. Some works also experimented with

data augmentation and transfer learning to improve generalization.

The paper reproduced in this project proposed a novel approach by applying YOLOv4, an object detection model, to classify Mel-spectrograms. Instead of detecting objects, the YOLO model was adapted to output class probabilities, benefiting from its multi-scale feature extraction and real-time inference capabilities. With this approach, the results reported in the original paper demonstrated that YOLOv4 achieved competitive accuracy in music genre classification, surpassing some classical CNNs.

In my case, as a first approach to the problem I tried a Vanilla CNN, to mark a starting point for the measurements of performance. Besides that, I tried also some other pretrained CNN models, before using definitely YOLO models

2.3 YOLO Architecture

You Only Look Once (YOLO) is a state-of-the-art convolutional neural network architecture originally developed for real-time object detection. Unlike traditional object detection pipelines that rely on multiple stages—such as region proposal, feature extraction, and classification—YOLO treats detection as a single regression problem. It directly predicts bounding boxes and class probabilities from the entire image in a single forward pass through the network.

The key idea behind YOLO is that the input image is divided into a grid, and for each cell in the grid, the model predicts a fixed number of bounding boxes along with their confidence scores and class probabilities. This design enables extremely fast inference while maintaining high accuracy, making YOLO well-suited for real-time applications.

In this study, however, YOLOv4 is **repurposed for image classification** rather than object detection. Specifically, the detection head of YOLO is replaced with a **global average pooling layer** followed by some **dense layers** ending with a **softmax activation** layer, which outputs the probability distribution over the 10 music genres. The powerful feature extraction capabilities of YOLO, such as its deep backbone network and multi-scale detection layers, remain intact. These components are leveraged to capture both low-level and high-level patterns in the Mel-spectrograms.

By adapting this architecture to treat spectrograms as input images and music genres as classification labels, the model benefits from YOLO's efficiency and feature richness, potentially leading to improved performance over traditional CNN classifiers trained on the same task.

2.3.1 YOLOv4

YOLOv4 enhances earlier versions by combining a powerful backbone (**CSPDarknet53**) with techniques like **Spatial Pyramid Pooling (SPP)**, **Path Aggregation Network (PANet)**, and the **Mish** activation function. It also incorporates training optimizations such as **Mosaic augmentation** and **DropBlock regularization**. These improvements make YOLOv4 effective for extracting multi-scale features, which is beneficial when adapted for classifying Mel-spectrograms. [3]

2.3.2 YOLOv8

YOLOv8 is the latest and most advanced YOLO model, offering better accuracy and speed through a more streamlined, **anchor-free design**. It uses a modern backbone with **C2f modules**, simplified detection heads, and improved training dynamics. Designed for versatility, it natively supports classification tasks, allowing easy replacement of the detection head with a **global average pooling** and **dense softmax layer**.

Specifically, **C2f (Cross-Stage Partial Fusion)** modules are lightweight convolutional blocks introduced in YOLOv8 to improve efficiency and feature fusion. Inspired by CSP (Cross Stage Partial) connections, C2f modules split the input into multiple parts, process a subset through several convolutional layers, and then concatenate the results. This design enhances gradient flow and feature reuse while reducing computational complexity compared to deeper or wider traditional blocks. The generic basic structure is represented in Fig.1.

2.4 Performance Measurements

To evaluate the performance of the classification model, several standard metrics were employed:

- **Accuracy** measures the overall proportion of correct predictions.
- **Precision** indicates the proportion of true positive predictions among all predicted positives, reflecting the model's ability to avoid false positives.
- **Recall** (or sensitivity) measures the proportion of true positives identified among all actual positives, capturing how well the model detects relevant classes.
- **F1-score** is the harmonic mean of precision and recall, offering a balanced measure especially in cases of class imbalance.

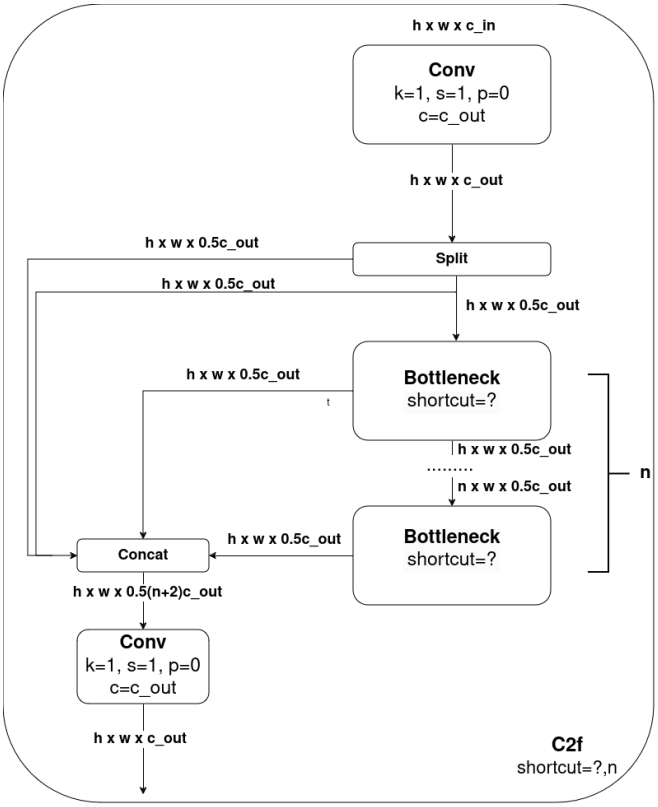

Figure 1: Basic scheme of a C2f module.

These metrics provide a comprehensive assessment of how well the model distinguishes between the 10 music genres. In particular, the reference paper makes use of **precision** in place of the most standard accuracy as main metric, in order to take into account the unbalance on data introduced by random sampling when splittind the dataset. It is reported as **mAP**, which is simply the avarage of the precision for the predictions of each class.

In addition to these metrics, a **confusion matrix** was used to visualize the distribution of predicted versus actual classes. This matrix highlights which genres are most frequently misclassified and provides insight into patterns of confusion among similar classes.

3 Dataset

3.1 GTZAN

The dataset used in this work is the **GTZAN dataset**, a widely adopted benchmark for music genre classification. It consists of 1,000 audio tracks, each lasting 30 seconds and evenly distributed across 10 different genres: classical, jazz, country, pop, rock, metal, disco, blues, reggae, and hip-hop. The dataset is balanced, with 100 tracks per genre, and has been commonly used in numerous academic papers due to its accessibility and genre diversity.

3.2 Preprocessing

To prepare the data for training with CNN models, each audio file was transformed into a visual Mel-spectrogram representation using the `librosa` Python library. The generated images were normalized from values in range $[0, 255]$ into values in range $[0, 1]$. Afterward, all spectrograms were compacted into a single matrix and saved into an HDF5 file (`.h5 format`), with samples organized as rows. This format allowed simpler loading during training and minimized preprocessing overhead during experimentation.

Output Image Size	416 × 416
Sample Rate	22050 Hz
Audio Length	30.0 s
Window Length	1024 samples
FFT Length	4096 samples
Overlap Length	512 samples
n of Mel Bands	64

(a) Version 1

Output Image Size	640 × 640
Sample Rate	22050 Hz
Audio Length	30.0 s
Window Length	1024 samples
FFT Length	4096 samples
Overlap Length	256 samples
n of Mel Bands	128

(b) Version 2

Table 1: Parameters for Mel conversion into spectograms

I tried two specific sets of parameters for the conversion, leading to 2 versions of the dataset. The first set comes from the values specified in the paper, while the second set orginates from personal experimentation. In particular, in the second version the image size was increased to match the one used in YOLOv8, while the Overlap Lenght is halved and the Number of Mel Bands are doubled in order to increase the resolution of the image.

4 Trained Models

In this section, after outlining the implementation details of the architectures, the main models trained on one of the two versions of the converted dataset are presented. These models are organized as a sequence of successive refinements leading to the best-performing model I was able to train. Failed attempts during this process are also included.

In particular, the following page contains a horizontal table summarizing all the main models I experimented with. The subsequent sections provide commentary on the motivations behind these attempts and the corresponding results.

In the **Test Precision** column, the most important performance results are highlighted in bold.

4.1 Implementation Details

The model implementation and experiments were conducted primarily on the **Kaggle platform**, which provides cloud GPU resources. Specifically, a **NVIDIA Tesla P100 GPU** was used to accelerate training. All scripts were written in **Python**, leveraging the **TensorFlow and Keras** deep learning libraries to define, train, and evaluate the neural network models. **sklearn** (scikit-learn), **seaborn** and **keras_cv** are some other used libraries.

ID	Model Changes	Val Precision	Test Precision	Test Accuracy
1	Vanilla CNN architecture	0.5439	**0.554**	0.535
2	Custom YOLOv4 implementation using dataset version 1	0.7627	**0.668**	0.61
3	From model 2, increased epochs and slightly higher dropout	0.6489	0.6401	0.61
4	YOLOv8 (fully trainable), small classification head (256–64), dropout also before output layer	0.651	**0.5681**	0.51
5	From model 4, slightly reduced dropout, none before output layer	0.6458	**0.6658**	0.66
6	From model 4, larger head (256–128–64)	0.6766	**0.6574**	0.66
7	From model 6, use of new data split (train-val-test: 800–100–100)	0.6959	**0.5907**	0.59
8	YOLOv8 with Flatten layer and medium-sized head (256–128–64), dropout and batch normalization	0.626	0.5114	0.49
9	From model 8, smaller classification head (128–64)	0.5641	0.5114	0.49
10	From model 8, larger classification head (256–128–128–64)	0.5061	0.3546	0.28
11	From model 7, increased early stopping patience and slightly reduced dropout	0.7308	**0.7126**	0.7
12	Very large classification head (256–128–128–64)	0.7174	0.5162	0.46
13	Slightly reduced classification head (256–128–64–64)	0.6474	0.4939	0.48
14	From model 11, batch normalization introduced	0.8002	0.6165	0.63
15	Batch normalization replaced by Layer Normalization	0.7772	**0.7401**	0.71
16	Training with dataset version 2	0.8216	**0.784**	0.78
17	Larger classification head (256–128–128–64)	0.812	0.727	0.69
18	From model 16, increased dropout (including before GAP), reduced learning rate, introduced class weights	0.8641	**0.8031**	0.8
19	K-fold cross-validation with zero patience	0.5916	0.6427	0.55
20	Patience increased to 30	0.7483	0.806	0.8

Table 2: Summary of Model Training Results

4.2 Baseline CNN (1)

As first attempt a Vanilla CNN was created, with the initial setup for all the other implemented models.

For all the first experiments, the dataset used is version 1. After loading and splitting the dataset, the model is created. It has 4 convolutional layers, ranging from 32 to 256 filters, a Flatten layer, a dense layer of 128 neurons with ReLU activation and the last dense layer of 10 neurons with softmax activation for the classification output. The specific important parameters used to build and train model 1 are in Tab.3.

The metric is custom (`MacroPrecision` class in the code) since during the training phase the standard tensorflow metric computes it on single minibatches, while the custom metric accumulates the values so that at the end of the epoch the precision is computed based on the whole dataset. This is the metric that is used to compute all the *Validation Precision* values of Tab.2.

All training sessions were performed using the **Early Stopping** technique, which halts training if the validation mAP does not increase for a fixed number of epochs (the **Patience** hyperparameter).

Parameter	**Value**
Learning Rate	10^{-4}
Batch Size	8
Epochs	2000
Patience	100
Optimizer	Adam
Loss Function	Categorical crossentropy
Metrics	Custom mAP
Dataset	version 1 (image size 416×416)
Train-Val-Test Split	700-100-200

Table 3: Training parameters used for model 1

This results in a fast training, with overfitting occuring very early (already about epoch 20). Still, the final performance is not that bad and there is no significant difference between validation and test precision. Hence, even with a rather simple model, the starting point for more performant models is very strong.

4.3 Custom YOLOv4 implementation (2, 3)

Trying to follow what was described in the reference paper, a YOLOv4 implementation was then used. Since it is not the latest version of YOLO and no official

library of tensorflow keras supports it, an already implemented architecture in this particular neural network environment is not easy to find. As a result, a custom implementation of the architecture based on the basic blocks described in the paper is written.

As already mentioned, the architecture in the paper is not well explained, thus many aspects were implemented from scratch by me. To reduce the total number of parameter, after the last convolutional layer a Global Average Pooling (GAP) layer is inserted. Also custom DropBlock layers are introduced, since the paper highlights its role in training. In the first version, a small classification head with (2 dense layers and output) with a single dropout layer is used, and in the second version the number of epochs and value of dropout is slightly increased.

The training is quite stable, but the model, at least with patience value of 15 used, shows its limitations. In its first version, it marks a new baseline of 66.8% for the test mAP, which is unfortunately very far from the results of the paper, even though the architecture should be more or less the same.

4.4 YOLOv8 - basic version (4 - 7)

Due to the limitations shown by the YOLOv4 model and the slow training (up to 5 minutes per epoch), the following attempt was with YOLOv8, since it is present in `keras_cv` library and, most importantly, it is pretrained on COCO [1].

Hence this model could be used as a backbone, adding a GAP layer and a classification head on top of it. On the one side, having a backbone pretrained on a big dataset should improve the training via transfer learning; on the other hand, the pretraining was done for object detection on real world images, that are very far from the spectograms. For this reason, the pretrained weights are used only as initialization, and no actual transfer learning is performed.

Various experiments with different classification heads and levels of dropout introduced are done, and the training samples are increased. In the end, the best model with respect to validation mAP has 3 dense layers, with number of neurons from 256 to 64, before the output layer. The performance are more or less the same reached by the custom implementation of YOLOv4, but with more room for improvement.

The specific parameters used with one of these models can be seen from Tab.4.

4.5 YOLOv8 with Flatten (8 - 10)

Some attempts with **Flatten layer** in place of GAP were done, with different classification heads. Also **Batch Normalization layers** are introduced in order to increase regularization. But we can notice that the training is very unstable and overall power is very limited, also in validation mAP.

Parameter	Value
Learning Rate	10^{-4}
Batch Size	8
Epochs	2000
Patience	30
Optimizer	AdamW, $\lambda = 10^{-4}$
Loss Function	Categorical crossentropy
Metrics	Custom mAP
Dataset	version 1 (image size 416×416)
Train-Val-Test Split	800-100-100

Table 4: Training parameters for model 7

4.6 YOLOv8 - improved (11 - 15)

GAP layer is kept for the next attempts. Other experiments are done with different classification heads, leading to a good trade-off. For these models, the techniques for regularization and control of the values in the compute graph of the network are improved by introducing normalization layers.

The most notable thing is the difference between model 14 and 15.

In the former, **Batch Normalization layers** are introduced, and this takes the validation mAP very high ($\approx 80\%$), but in the meantime mAP computed on test set is very lower ($\approx 60\%$). This is probably due to the way batch normalization works, computing statistics on each minibatch, that is not well customed to very small minibatches (8 elements in this model). In fact, the training is very unstable, and the generalization capabilities of the model are evidently very limited.

In the latter, batch normalization layers are replaced with **Layer Normalization**. This layers do the same jop of batch normalization, but statistics are cumulated to compute in the end statistics with respect to the whole dataset. As a result, the generalization capabilities gets rather better, with a slightly decreased validation mAP ($\approx 77\%$) but an increased test mAP ($\approx 74\%$). Now the drop from validation to test is only of $\approx 3\%$, instead of $\approx 20\%$ as before.

4.7 Training with Bigger Inputs Size (16 - 18)

The following experiments were conducted on the second version of the converted dataset, which has in particular the input size 640×640 of the original pretrained model (different from the original input size 416×416 of YOLOv4). Besides that, the resolution of the images is increased.

Moreover, class weights are used to compute the loss, taking into account class

imbalance in the training set.

This allowed to get a more overly stable training, and better results. After a bit of hyperparameter tuning, The best result of $\approx 80\%$ is reached, through the training parameters shown in Tab.5.

Parameter	**Value**
Learning Rate	10^{-5}
Batch Size	8
Epochs	2000
Patience	100
Optimizer	AdamW, $\lambda = 10^{-4}$
Loss Function	Categorical crossentropy
Metrics	Custom mAP
Dataset	version 2 (image size 640×640)
Train-Val-Test Split	800-100-100

Table 5: Training parameters for model 18

4.8 K-Fold Cross-Validation (19 - 20)

The implementation of K-Fold Cross-Validation technique is tried with these 2 models. A brief explanation of the technique is in [4]. Once removed the test data, the training data is subdivided into a fixed number of folds, and at each iteration a complete training keeping only one of these folds as validation set is performed together with early stopping. In the end, a training with the average number of epochs of the previous trainings with no early stopping is performed.

The first model has patience set to 0. As a result, the final training performs only 3 epochs, but this allows an already good result (comparable with the first complete custom YOLOv4 training).

The second model has patience increased to 30, allowing longer training, but in the end no actual improvement from the previous best model is achieved.

5 Final Results

In this section the final choice for the model and its results are shown more in depth. In particular, the plots in the following page, showing the different results, are discussed.

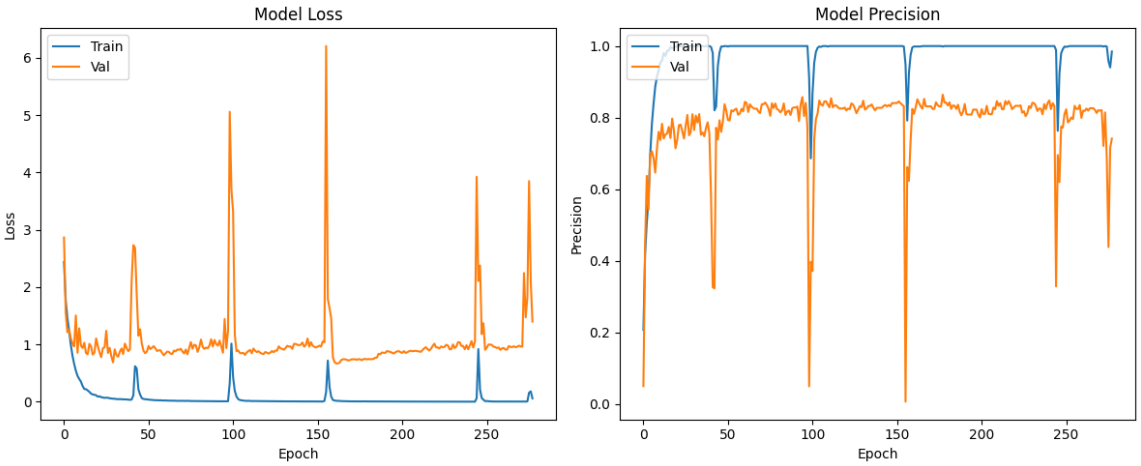

Figure 2: Plot of history of training

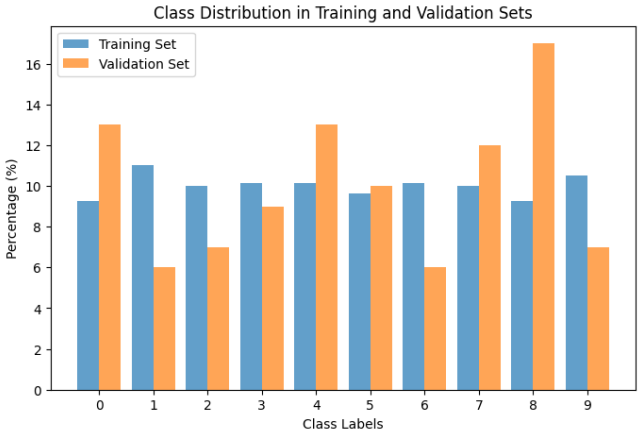

Figure 3: Distribution of samples per class

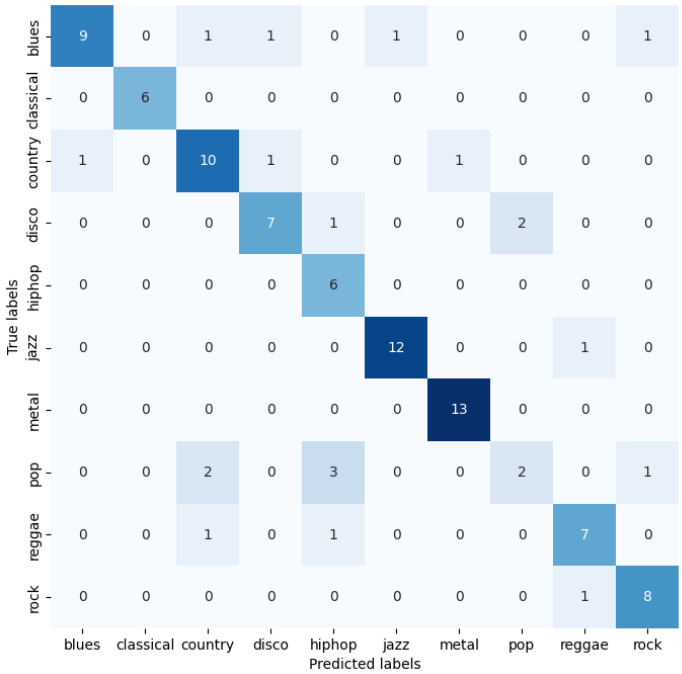

Figure 4: Confusion Matrix

Class	Precision
Blues	0.692
Classical	1.000
Country	0.769
Disco	0.700
HipHop	1.000
Jazz	0.923
Metal	1.000
Pop	0.250
Reggae	0.778
Rock	0.889

Table 6: Precision per class

The best model considered in the end is model 18 in Tab.2. It was trained on version 2 of the dataset, with the conversion parameters specified in Tab.1b, and the main training parameters listed in Tab.5.

The model uses YOLOv8 as backbone, exploiting weights of the original architecture pretrained on COCO dataset as initialized weights. A GAP layer links the backbone with the classification head, consisting in 3 dense layers with ReLU activation and neurons rangin from 256 to 64. They all have LayerNormalization after them to better control the values during trainig, and the first 2 have a dropout layer for regularization. During training, weights to consider class imbalance and early stopping for regularization are used.

The plots show the training trends for both loss and mAP. As evident, the training is relatively stable, with the presence of five noticeable spikes—sharp drops in performance that last only a few epochs. This behavior may be attributed to the strong regularization applied and the large number of parameters in the model.

From the plots, it is possible to identify the onset of overfitting, which appears shortly after epoch 150, when the highest validation mAP is reached. From that point on, the validation loss begins to increase.

Fig.3 illustrates how the samples in the training and validation sets are distributed among the classes.

To conclude, Fig.4 shows the confusion matrix for the predictions on the test set, and Tab.6 shows the computed *1-vs-rest* precision values for each class.

6 Discussion and Conclusion

The final model of the project demonstrates good performance, although it remains far from the very high performance reported in the reference paper. As previously mentioned, the inability to fully reproduce the original work makes these results not directly comparable, and for this reason, a detailed comparison is not pursued.

What stands out most is the progression of models leading to the final one. The initial baseline model exhibited performance approximately 25% lower than the final version. The improvements were achieved gradually, through a combination of exploratory hyperparameter tuning and intuitions drawn from earlier results.

In conclusion, while the room for further improvement has become narrower, the iterative process of refining the model could still be continued, potentially leading to even better performance. However, given the current approach, it is unlikely that the results of the reference paper will be fully reproduced.

References

[1] COCO dataset,
https://cocodataset.org.

[2] Yu-Huei Cheng, Che-Nan Kuo,
Machine Learning for Music Genre Classification Using Visual Mel Spectrum.

[3] Alexey Bochkovskiy, Chien-Yao Want, Hong-Yuan Mark Liano,
YOLOv4: Optimal Speed and Accuracy of Object Detection.

[4] K-Fold Cross-Validation technique,
https://scikit-learn.org/stable/modules/cross_validation.html.