GTZAN Genre Classification Report

Aref Moradi

Abstract

In this report I discuss the GTZAN genre classification dataset and a classification task on this dataset. I use a simple CNN model to classify the tracks from their mel-spectogram. Furthermore, I discuss about a minimal dataset pipeline to allow for using much larger dataset sizes and also applying offline data augmentation to sound tracks.

1 Introduction

Music genre classification is a machine learning task concerned with tagging tracks and songs with the genre they belong to. Automating such classifications can be beneficial in many ways. For example, when building recommendation systems for songs, a very important feature of a song is the genre it belongs to. Therefore, a classifier that can reliably detect the genre of a song can be very beneficial in building such recommendation systems.

Various methods exist for audio processing. In this report I focus on applying deep learning on mel-spectograms of sound tracks. Mel-spectograms are a visual representation of a sound track created by representing the sound signal in the frequency domain using Fourier transforms. The technical details of obtaining the mel-spectogram are out of the scope of this report.

By using the mel-spectogram the classification task effectively becomes an image classification task where the images are mel-spectograms of the sound tracks.

2 Dataset

For this task I use the GTZAN genre dataset which is comprised of ten music genres and 100 tracks of 30 seconds per each genre. The size of the small dataset is around 1.2GB. In the following sections I briefly explain the process of converting the tracks to mel-spectograms. Furthermore, I explain the data pipeline designed to allow for scaling to larger datasets and also allow performing offline data augmentation.

2.1 Converting to Mel-Spectograms

To convert the wav format to mel-spectograms I use the lebrosa library which contains all the functionality needed for the conversion to mel-spectograms. For the parameters of the conversion such as window size, overlapping size, etc I use suggestions that others have used. The functions for converting the wav files to mel-spectograms are available in the *prepare_data.py* file.

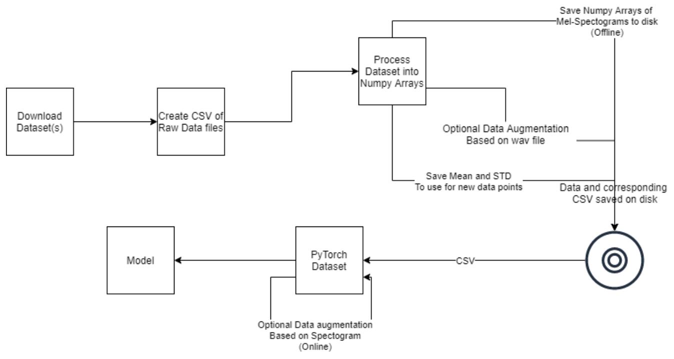

Download Dataset(s)
Create CSV of Raw Data files
Process Dataset into Numpy Arrays
Save Numpy Arrays of Mel-Spectograms to disk (Offline)
Optional Data Augmentation Based on wav file
Save Mean and STD To use for new data points
Data and corresponding CSV saved on disk
Model
PyTorch Dataset
CSV
Optional Data augmentation Based on Spectogram (Online)

Figure 1: Overview of the dataset pipeline.

2.2 Data Pipeline

The data pipeline is designed to allow for training on larger datasets than the small GTZAN dataset. This is acheived by separating each step of the data processing and implementing *PyTorch* DataLoaders which read directly from disk in an efficient manner. To achieve this, each step of the pipeline is represented by an CSV file that holds the path to the data files and the label for them. The CSV could potentially also contain other features for each track such as classic audio features. An overview of the steps are shown in Figure 1.

As it can be seen in the diagram the first step of the pipeline is downloading the dataset(s). After the raw dataset is available on disk a corresponding CSV is created which contains the path of each raw file and its respective label. The corresponding python file for these steps is the *create_dataset_csv.py* file.

Using the CSV created in the previous step the *prepare_data.py* script pre-processes the data into numpy arrays of the mel-spectograms. Optionally, this step can also augment the dataset using common audio augmentations to increase the size of the dataset. By adding a new column to the CSV file we can also easily keep track of the augmented and original datasets for further steps of the pipeline. Finally, the mean and std of the dataset is also saved in disk to allow for online normalization of new data points. Note that in this step we split the train and test datasets before augmenting and calculating the normalization values. Only the train data is used for augmenting new datasets. Also, the normalization values should not be calculated on all of the dataset as this can cause unwanted data leakage between our train and test sets.

At this point the pre-processed samples are stored on disk alongside a corresponding CSV file which contains their paths. To feed the data into models I use a custom PyTorch dataset class which takes the path of the csv file as input. In this way, no unneeded data is loaded into the memory. Also, we can conveniently use existing libraries to split the train dataset to train and validation datasets. The *__getitem__* method of the dataset class implements the loading of the actual numpy array for each file and returning it to the model. A DataLoader class can be conveniently used to wrap the Dataset class and serve the data in batches to the model. Furthermore, the DataSet class can be easily extended to perform transformations and augmentations that can be applied directly on the mel-spectogram.

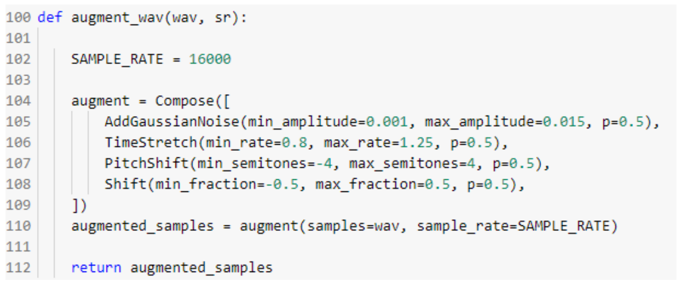
2.3 DataSet Discussions

In this section I will present alternatives to the current DataSet pipeline structure and discuss the pros and cons of the current approach.

As mentioned previously, one of the main benefits of this approach is that the training data is not directly loaded in memory. The DataLoader class will control the amount of data loaded directly into memory as well as other settings such as shuffling the dataset, etc. Moreover, the dataset is first pre-processed into mel-spectograms which will save time for future trainings since the Dataset will not need to perform pre-processing on the data for the model to be able to train on the data. Also, the data augmentation is performed offline which is a time consuming task as I saw in the course of this project.

However, there are downsides to this approach. As mentioned, the mel-spectograms are directly saved to disk and used for future trainings. Effectively, the models will be constrained to using mel-spectograms as their input in this way and developing a model which will use other representations of the wav file will require us to change the pipeline. This problem can be mitigated by creating more steps in the data pipeline in real-life scenarios. Furthermore, in the current approach if we want to test other data augmentations we have to re-run the pipeline which might not be practical for very large datasets. An alternative could be to do all augmentations online during training. I believe this is case-dependent and depends on many factors like the rate of incoming new data and the rate at which we would want to train such a model in production. A good compromise might be having to sets of augmentations for both offline and online data augmentation.

Finally, I think by creating well defined pipeline steps and defining explicit input and output requirements for each this approach can be scaled to much larger datasets.



```python
def augment_wav(wav, sr):

    SAMPLE_RATE = 16000

    augment = Compose([
        AddGaussianNoise(min_amplitude=0.001, max_amplitude=0.015, p=0.5),
        TimeStretch(min_rate=0.8, max_rate=1.25, p=0.5),
        PitchShift(min_semitones=-4, max_semitones=4, p=0.5),
        Shift(min_fraction=-0.5, max_fraction=0.5, p=0.5),
    ])
    augmented_samples = augment(samples=wav, sample_rate=SAMPLE_RATE)

    return augmented_samples
```

Figure 2: Data Augmentation Code Snippet

3 Model

Due to time constraints, I implemented and tested a simple CNN model on the normal dataset and augmented datasets. The CNN takes the mel-spectogram as input and outputs a one-hot vector of the classes. However, the structure and modularity of the code enables rapid testing of different models and different algorithms. The CNN model can be seen in the *models.py* file.

The model is implemented in PyTorch and uses the PyTorch Lightning framework. PyTorch Lightning helps avoid repeating training boilerplate and writing modular models and training pipelines. Furthermore, a GPU(s) can be used for training with zero modifications to the training code. Finally, no extra work is needed to track the training metrics in TensorBoard (or other tools) as Lightning provides easy to use tools for these purposes.

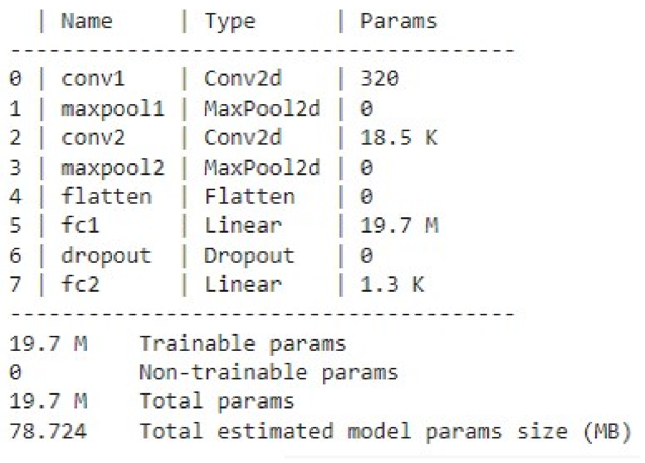
I tested training the model on a PC with no GPU and seamlessly used the same code to run the whole pipeline and model on Google Colab. The main training and evaluation logic can be seen in the *train.py* notebook.

For data augmentations, I used the Audiomentations library which contains implementations of common audio augmentation techniques. The implementations can also be seamlessly added to the training step of both PyTorch and TensorFlow frameworks. In this project, the augmentations are performed offline in the dataset pipeline before saving the data to disk. Figure 2 shows the code snippet with the applied data augmentations and their respective parameters.

The model implementation and layers can be seen in figure 3. To apply a softmax at the output layer I use the *log_softmax* layer alongside an *nll_loss* loss function. Another approach would be to not use a *log_softmax* layer and output the logits directly to a *CrossEntropyLoss* which under-the-hood combines a *log_softmax* and *nll_loss*. Using a standalone *softmax* layer is not advised as it can easily result in overflow issues in PyTorch. For the optimizer, I use an Adam optimizer with a starting learning rate of 0.001 which is the default value.

3.1 Model Discussion

Due to time constraints, I was not able to try different models and different algorithms. However, in this section I mention some next steps that can be taken to improve the model and discuss some pros and cons of the current implementation.



```
  | Name     | Type      | Params
---------------------------------------
0 | conv1    | Conv2d    | 320
1 | maxpool1 | MaxPool2d | 0
2 | conv2    | Conv2d    | 18.5 K
3 | maxpool2 | MaxPool2d | 0
4 | flatten  | Flatten   | 0
5 | fc1      | Linear    | 19.7 M
6 | dropout  | Dropout   | 0
7 | fc2      | Linear    | 1.3 K
---------------------------------------
19.7 M    Trainable params
0         Non-trainable params
19.7 M    Total params
78.724    Total estimated model params size (MB)
```

Figure 3: Model Layers

An immediate change to test is to use different k-fold strategies while training the model. This can be done relatively simply by using the *scikit* library k-fold split on the PyTorch Dataset class. Also, an exhaustive hyperparameter tuning can help to achieve higher performance using this dataset. Given, the small size of the model this can be done on Google Colab or any other PC with a GPU. For bigger models, scaled hyperparameter tuning tools such as raytune can be used.

In the current project, I use a mix of data augmentations randomly applied to the dataset. However, a more detailed study of each augmentation and their effect on the classification task can be beneficial. Also, as suggested in the assignment Ensemble models can be used to improve the performance of the classifier. One interesting approach that might be beneficial as future work is combining the features learned by the convolution layers with classical features of audio tracks in the linear layer. In this way the model can benefit from both features. Note that these features can easily be added to the CSV file of each data file during the pre-processing step.

Moreover, it would be worthwhile to test classic machine learning algorithms on the classic features to compare how they perform compared to deep learning approaches. In many cases classical and simple algorithms might be enough for our purposes.

Finally, I believe the modular implementation of the training loop and data pipeline allows for rapid prototyping of the suggested improvement.

4 Deployment

For this project, I did not attempt to solve problems related to deployment of the model. A very simple deployment can be done by using FastAPI in Python and a trained model. The pre-processing step of the data pipeline can be slightly modified to make it reusable for by the service where the model will be served on. In this way we can ensure the pre-processing steps done in training and inference steps are the same.

Model	Epochs	Batch Size	Accuracy
CNN - Raw Data	50	20	0.58
CNN - Augmented Data	100	20	0.68

Table 1: Model Results

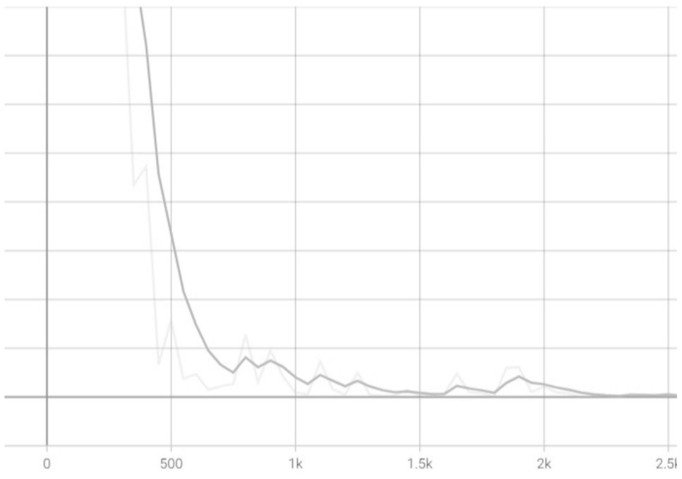


Figure 4: Training loss for original data.

Moreover, a proper CI pipeline needs to be setup to automate training, testing and deploying a model in production. An important part of this pipeline is creating simple steps to catch bugs and errors early in the procedure. One simple test can be trying to over-fit every model on a very simple dataset to ensure the model has learning capacity and bugs weren't introduced.

5 Evaluation and Results

As mentioned previously, Due to time constraints a simple CNN model was tested. The same model was tested using only the original data and the augmented data. To compare the models the accuracy metric is used. Note that in this special case the accuracy metric is useful as we have a balanced dataset (even the augmentations are done in a manner to keep the classes balanced). However, in most other classification cases the f1-score is a better measure to deal with existing class imbalances and the accuracy metric should **not** be used. The results of these two experiments are provided in table 1. The training and validation loss graphs for the original dataset are seen in figure 4 and 5. The training and validation loss graphs of the combined original and augmented dataset are seen in figure 6 and 7. As can be seen the models reach peak performance much sooner than 100 epochs and we save the best performing model according to the validation loss. Moreover, we see that the augmented dataset results in a clear improvement in the performance of the model which points to the fact that such augmentations can be beneficial for the model to learn to generalize.

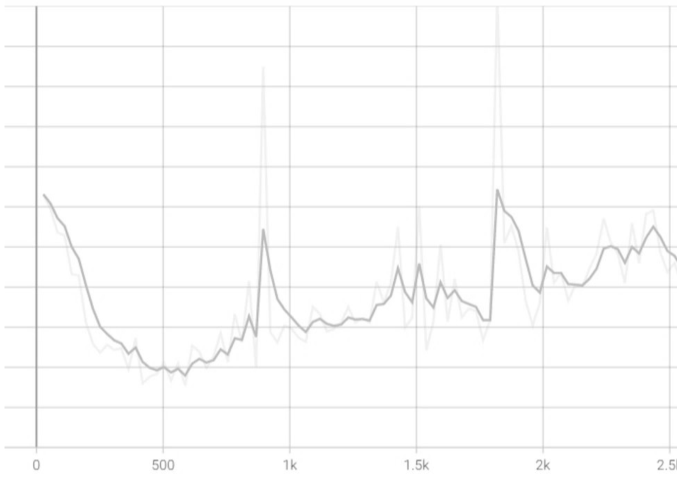


Figure 5: Validation loss for original data.

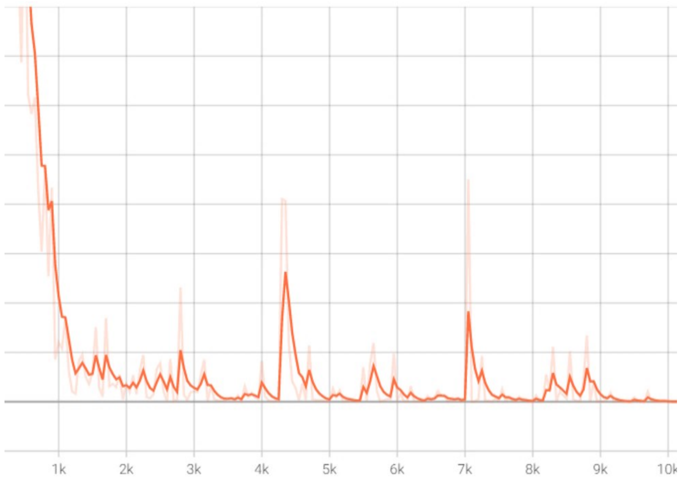


Figure 6: Training loss for original data.

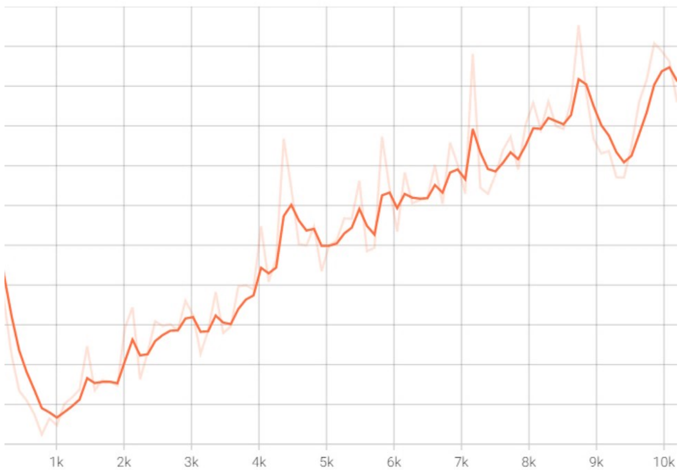


Figure 7: Validation loss for original data.