Αναγνώριση Προτύπων
Προπαρασκευή 3ης Εργαστηριακής Άσκησης
Αναγνώριση Είδους και Εξαγωγή Συναισθήματος από Μουσική

Παρέλλη Μαρία
03115155
9ο Εξάμηνο

Χαρδούβελης Γεώργιος-Ορέστης
03115100
9ο Εξάμηνο

Step 1

a

In the prelab, we work with a subset of the FMA (Free Music Archive) Dataset. The FMA dataset provides full-length and high-quality audio, pre-computed features, together with track- and user-level metadata. It was made for musical analysis with up to 161 genres consisting of 106,574 untrimmed tracks from 16,341 artists and 14,854 albums in total. All of the data is MP3-encoded. The archive offers the dataset in different sizes, where there is also a possibility to get 8,000 tracks with 8 balanced genres. In this task we classify samples according to their musical genre.

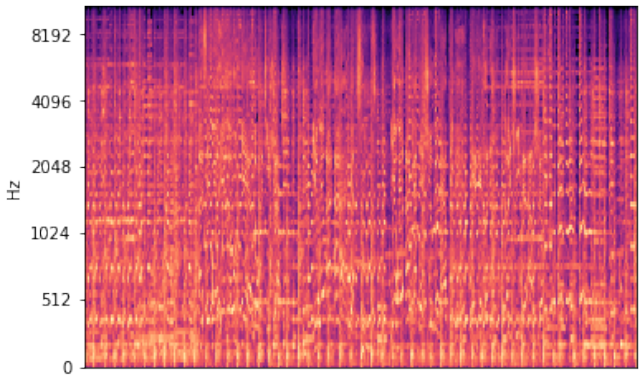
Our input consists of mel spectograms for each audio sample. Traditional MIR tasks relied heavily on MFCCs which extract frequency content in a short-window frame of audio. However, recent works are shifting towards spectrograms, which take advantage of temporal structure and have shown good performance in classification tasks. A spectrogram is a representation of frequency content over time found by taking the squared magnitude of the short-time Fourier Transform (STFT) of a signal. A mel-spectrogram is a spectrogram that has been mapped to the mel scale. This leads to lower frequencies gaining more importance, which is how humans interpret sound.
From the fma_genre_spectrogram/train_labels.txt file, we randomly pick two lines, corresponding to tracks in different music genres. We chose the tracks with labels 1043 and 10035 which belong to blues and classical genre respectively.

b

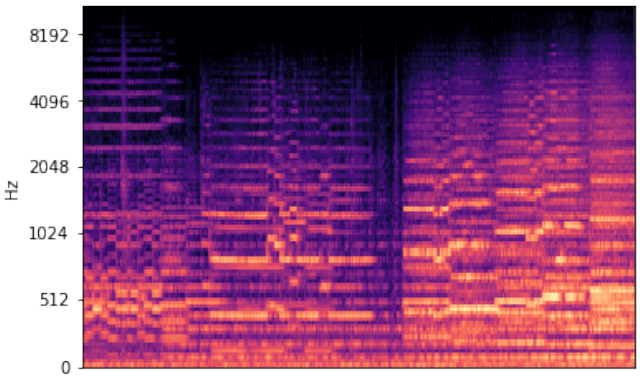
Following the given instructions, we pick the corresponding files from the fma_genre_spectrogram/trai directory and get their mel spectograms (and chromagrams needed for step 3).

c

Below we can see the mel spectograms, depicted using the librosa.display.specshow function.



(α′) Sectogram for blues track



(β′) Spectogram for classical track

In the spectograms we can see a visual representation of the spectrum of frequencies as they vary in time. The yellow speckles indicate the peaks of frequency over time. More specifically, what we notice are the tones indicated by the frequency level that are played for the corresponding short period of time.

In our first sample we have a blues song, and we observe many vertical bars which indicate short and powerful frequency peaks caused by the beat. This can possibly be attributed to the frequent use of drums in blues music.

On the contrary, in our second sample we have a classical piece, where the previous element is absent; what we observe is smaller frequencies with continuous frequency levels over bigger periods of time, as expected by the organs used (i.e. violin), as well as the overall nature of classical music. Musically speaking, chords are prominent here, whereas beat is absent.

Step 2

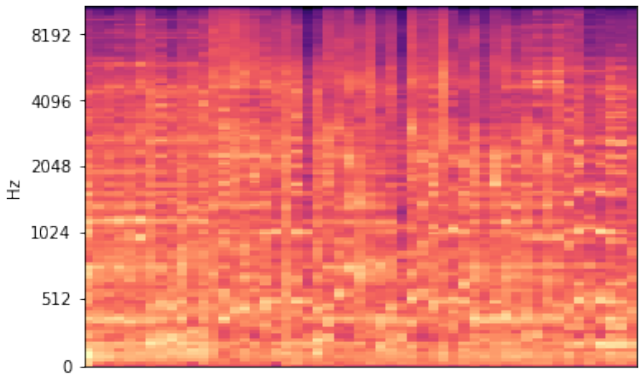
a

Printing the dimensions of the randomly chosen spectographs above, we see that there are 140 frequency steps and 1291 and 1292 time steps respectively. Thus, in the y-axis, we have hundreds of units to work with.
Many neural network applications use the above data to classify music pieces into genres. Nevertheless, even after reducing our data size by using the mel-spectograms, using said data to train an LSTM with a large dataset would be computationally expensive. Moreover, there is a lot of information that doesn't prove to be quiet useful, as well as some overlapping information, i.e. the minimum duration of a note. Studies indicate that the most useful input for genre classification is the beat of the track

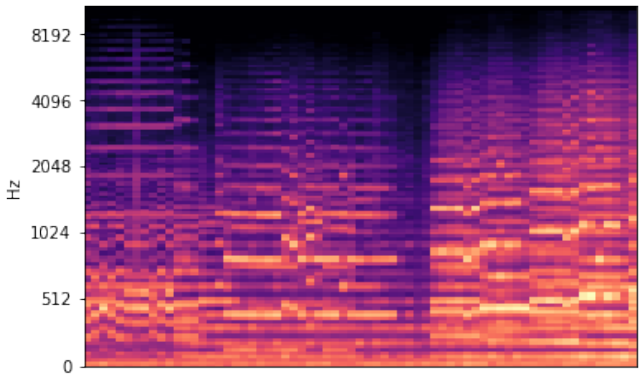
b

A solution given in order to reduce out time steps is to synchronize our spectograms with the beat. Hence, we use as our dataset the spectograms in the folder "fma_genre_spectrograms_beat", where we keep the median between each dinstictive beat.
Printing the dimensions of the current spectograms, we observe 53 and 67 time units respectively, so we reduced the size by one order of magnitude, which renders our neural network way faster more efficient.

We can observe said spectograms below:



(α′) Beat-synced spectogram for blues track



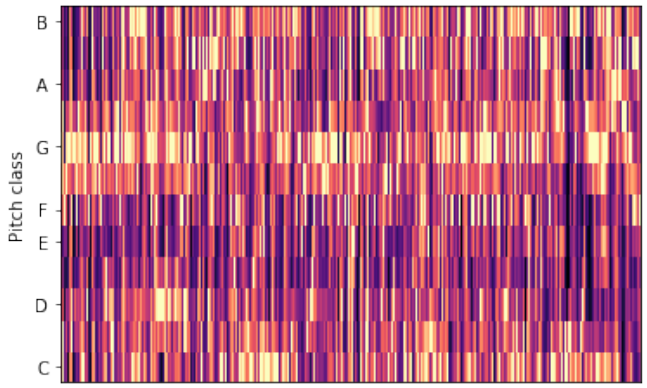
(β′) Beat-synced spectogram for classical track

As we can see, these spectograms with the ones in the previous step show many similarities, as they are almost identical. In the first one we still observe powerful peaks of frequency, whereas in the second more lasting notes. However, the same intuitive information seems to be compressed in a smaller time interval.

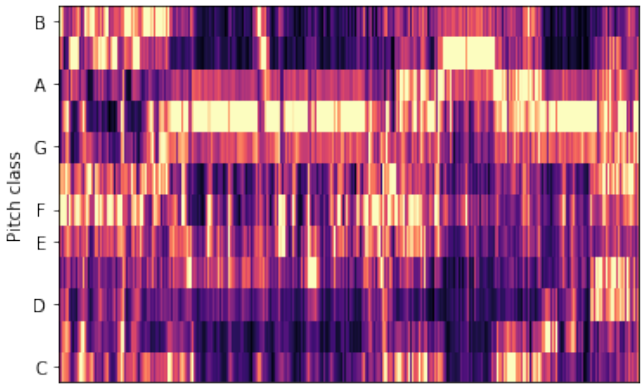
Step 3

Another mean of displaying frequencies being played over a period of time is a chromagram. Chromagrams relate to the twelve different pitch classes (C, C#, D, D#, E , F, F#, G, G#, A, A#, B). Chroma feautes can capture harmonic and melodic characteristics of music, while being robust to changes in timbre and instrumentation. Each chroma value differs an octave from the previous one (double the frequency).
More specifically, a chromagram is a time-chroma representation expressing how the representation's pitch content within a time window is spread over the twelve chroma bands.

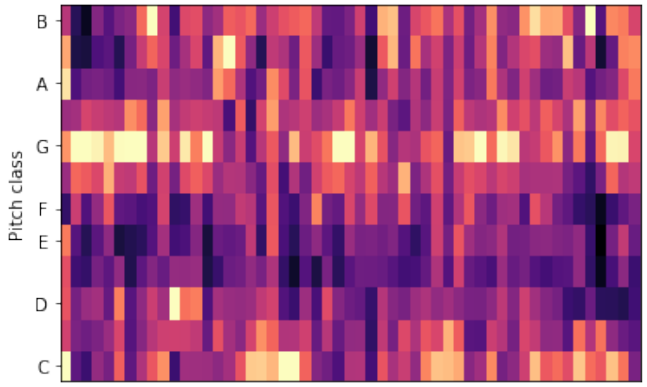
In the figures below we can see the chromagrams for the randomly chosen tracks, as well as the beat-synchronized chromagrams (produced similar as the respective spectograms).



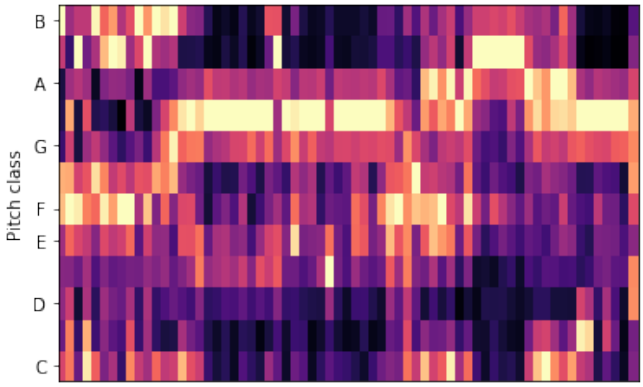
(α′) Chromagram for blues track



(β′) Chromagram for classical track



(γ′) Beat-synced chromagram for blues track



(δ′) Beat-synced chromagram for classical track

As we can see, in the blues track we have more D# and E pitches without a lot of variation, where as in the classical track we observe intense flunctuation, as there is a lot of higher pith concentration in C, C# and A#, B.
In addition, the beat-synchronized chromograms show the same behavior as the original ones, just more compacted, proving again that beat-synchronization is an effective way of reducing our input size without dismissing a lot of useful information.

Step 4

a

For this task we used the implementation of a pytorch dataset, as it was provided in the help code.

The functions and classes implemented are:

- torch_train_val_split (function): splits the dataset creating different train and validation sets. By default, the validation set is 20% of hte whole dataset
- read_mel_spectrogram (function): reads only the first 128 elements of the input, which constitute the spectogram
- read_chromagram (function): reads only the last 128 elements of the input, which constitute the chromagram
- read_fused_spectrogram (function): reads all 256 elements of the input, which constitutes the spectogram and chromagram concatenated
- get_files_labels (function): retrieves the labels of the files from the dataset
- PaddingTransform (class): implements padding, which is needed so that all samples have the same length

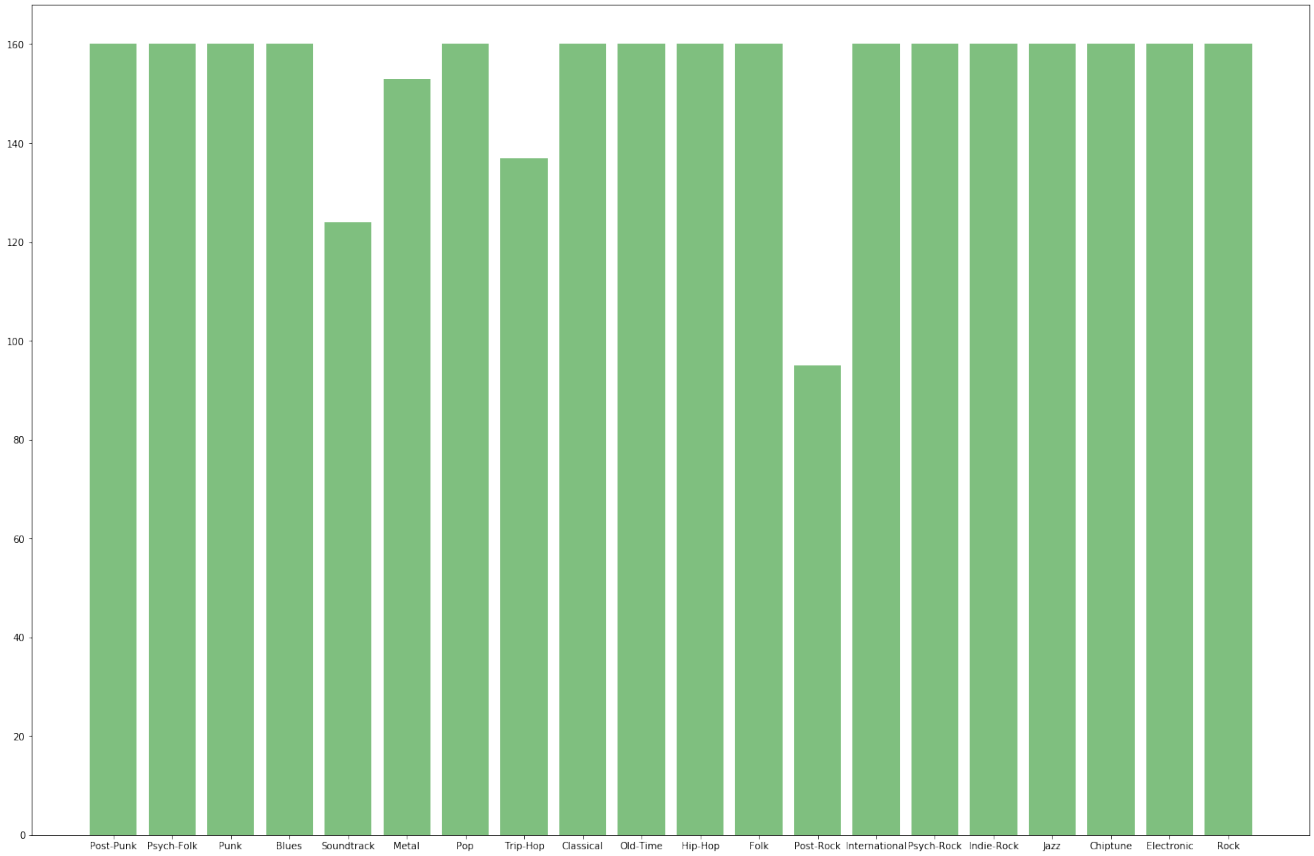
More thorough analysis for the use of the given classes and function can be seen in comments alongside the code.

b

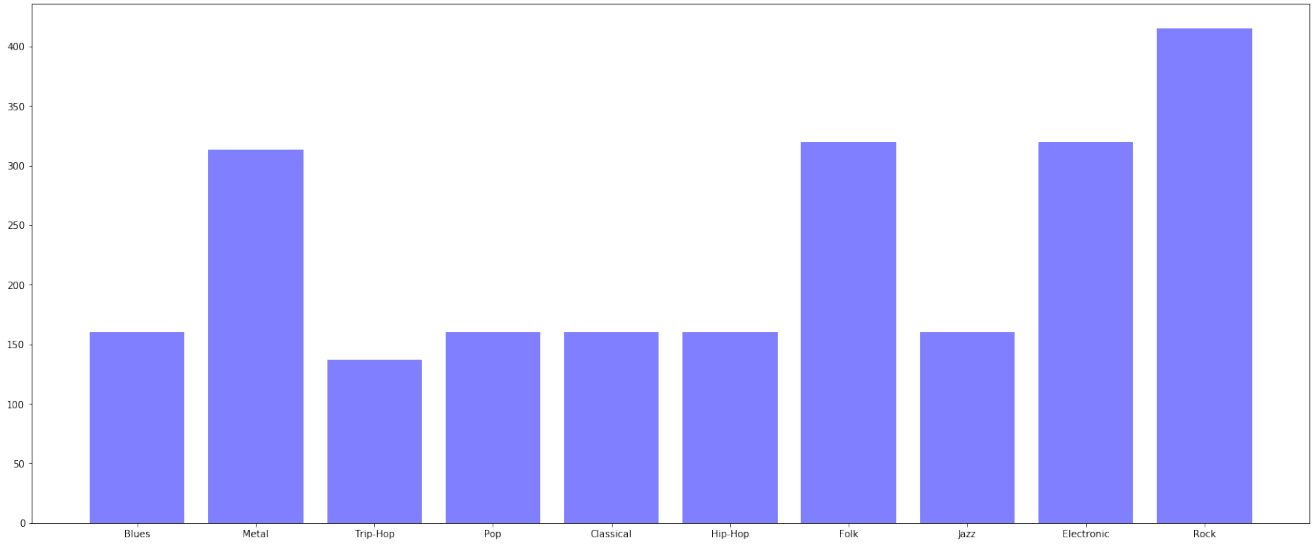
Before we initialize and train our model we prepossess our data, by merging similar classes and removing those with a small number of samples. This step is necessary, because with limited training data for a class our model would not be able to generalize and test accuracy would be lower.

c

Below we can see two histograms depicting the number of songs on each class (music genre), before the procedure of merging and removing classes and after.



(α′) Histogram before mapping



(β′) Histogram after mapping

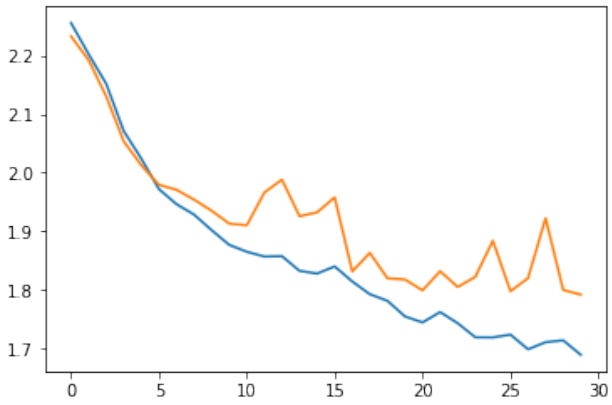
As we can see, in the first case most classes are balanced, with some exceptions that highly differ. After tampering with the number of classes, there is a higher imbalance. Nevertheless, samples with similar genres that would be hard to distinguish are now in the same class, improving eventually the classification procedure.

Step 5

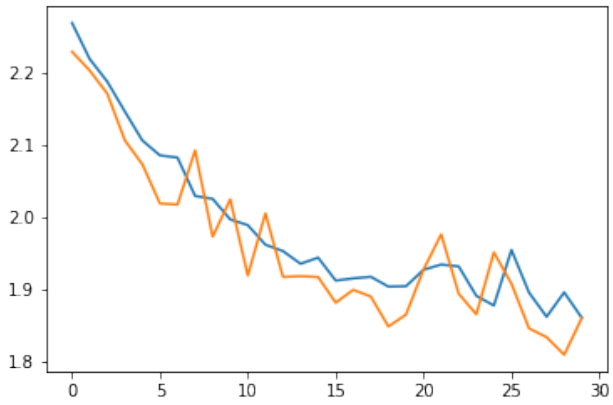
We initialized a LSTM model, which consists of two layers, where the hidden layer size is 128. Input dimensions are 128, 128, 12 and 140 for the spectograms, beat-synced spectograms, chromagrams and concatenated spectograms with chromograms respectively. We trained our

four models with a learning rate of 0.0001 and batch size=32 for 40 epochs. We added drop out (with drop probability = 0.4) and a weight decay factor (equal to 0.001). We saved the model state with the lowest validation loss.

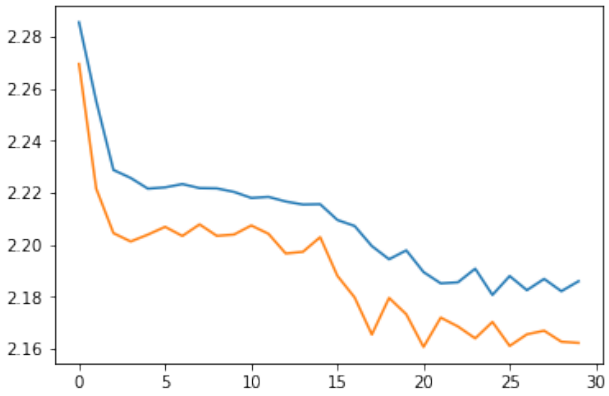
Below, we can see the learning curves for each model.



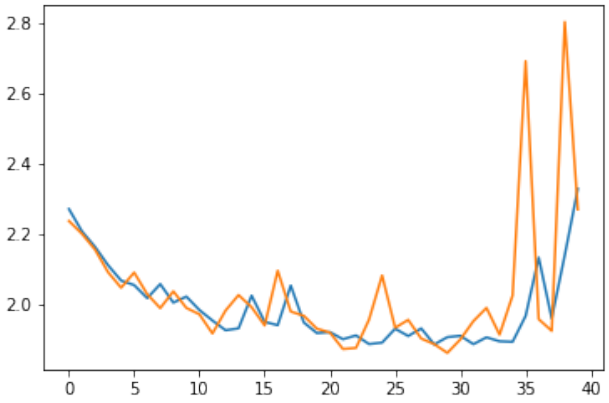
(α′) Beat-synced spectograms as input data



(β′) Spectograms as input data



(γ′) Chromagrams as input data



(δ′) Fused spectograms & chromagrams as input data

The learning curves of our trained models. The blue one is the learning curve for training loss and the orange one is the learning curve for validation loss

Step 6

Here we evaluate the models from the previous set (with spectograms, beat-synced spectograms, chromagrams and concatenated spectograms and chromagrams as training data). Using the sklearn.metrics.classification_report function, we calculated, the overall accuracy, and the precision, recall and F1-score, both for each individual class, as well as for all the classes combined (macro-averaged, micro-averaged and weighted averaged).
Note that since we don't examine multi-label or multi-class with a subset of classes data, the micro-average for all examined metrics is equal to the accuracy, and thus excluded from the classification reports which can be seen below.

	precision	recall	f1-score	support
0	0.00	0.00	0.00	40
1	0.33	0.62	0.43	40
2	0.28	0.50	0.36	80
3	0.24	0.51	0.33	80
4	0.50	0.03	0.05	40
5	0.00	0.00	0.00	40
6	0.48	0.28	0.35	78
7	0.00	0.00	0.00	40
8	0.34	0.47	0.39	103
9	0.00	0.00	0.00	34
accuracy			0.31	575
macro avg	0.22	0.24	0.19	575
weighted avg	0.26	0.31	0.25	575

(α′) Evaluation of LSTM with spectograms as input data

	precision	recall	f1-score	support
0	0.00	0.00	0.00	40
1	0.49	0.53	0.51	40
2	0.36	0.68	0.47	80
3	0.34	0.61	0.44	80
4	0.22	0.20	0.21	40
5	0.20	0.10	0.13	40
6	0.48	0.62	0.54	78
7	0.00	0.00	0.00	40
8	0.39	0.27	0.32	103
9	0.12	0.03	0.05	34
accuracy			0.37	575
macro avg	0.26	0.30	0.27	575
weighted avg	0.30	0.37	0.32	575

(β′) Evaluation of LSTM with beat-synced spectograms as input data

	precision	recall	f1-score	support
0	0.00	0.00	0.00	40
1	0.00	0.00	0.00	40
2	0.00	0.00	0.00	80
3	0.20	0.68	0.31	80
4	0.00	0.00	0.00	40
5	0.00	0.00	0.00	40
6	0.23	0.29	0.26	78
7	0.00	0.00	0.00	40
8	0.19	0.36	0.25	103
9	0.00	0.00	0.00	34
accuracy			0.20	575
macro avg	0.06	0.13	0.08	575
weighted avg	0.09	0.20	0.12	575

(γ′) Evaluation of LSTM with chromagrams as input data

	precision	recall	f1-score	support
0	0.00	0.00	0.00	40
1	0.39	0.60	0.48	40
2	0.30	0.75	0.42	80
3	0.36	0.40	0.38	80
4	0.00	0.00	0.00	40
5	0.00	0.00	0.00	40
6	0.44	0.36	0.39	78
7	0.00	0.00	0.00	40
8	0.33	0.50	0.40	103
9	0.00	0.00	0.00	34
accuracy			0.34	575
macro avg	0.18	0.26	0.21	575
weighted avg	0.24	0.34	0.27	575

(δ′) Evaluation of LSTM with fused spectograms & chromagrams as input data

All the metrics calculated root back to the confusion matrix and the actual values in relation with our predicted ones.

- The most common metric is accuracy, which shows the overall proportion of correctly classified samples.
  Concentrating on the individual classes, we can define as positive the samples actually in the class, and negative the ones not in the class. Also, we characterize true the ones that our classifier correctly guessed, and false the mistaken ones. Thus we can characterize samples as true positives (TP), true negatives (TN), false positives (FP) and false negatives (FN). In light of these definitions, we could define accuracy as the proportion if the overall TN and TP $\left(\frac{TP+TN}{TP+TN+FP+FN}\right)$.
  Precision is defined as the proportion of the positive (belong in class) of our total items identified as positive $\left(\frac{TP}{TP+FP}\right)$.
  On the other hand, recall is the proportion of the correctly identified as positive of the total true positives $\left(\frac{TP}{TP+FN}\right)$.
  In general, we strive for the maximum precision and recall scores. However, improving one of those can result in the other one lowering. That's why we also use f1-score, a metric that combines the above two into one using the harmonic mean: $f1\_score = 2\frac{precision \cdot recall}{precision + recall}$. By using this mean, it gives more weigh to the lower number.

- Generalizing the above metrics in more classes, the macro-averaged metrics (precision, recall and f1-scores) are the means of the respective individual class metrics. The difference with the weighted-average metrics is that we consider equal weights for each class (whereas

on the weighted-average ones each class' weigh is the number of samples in it).

The micro-averaged metrics use the same equations as the individual class ones extended for all our samples (true positives are considered to be the overall correctly predicted samples). As we mentioned before, it's easy to see that in our case the micro-average metrics are equal to the accuracy.

- Each metric can be better in different situations and sometimes they have large deviations. For example, accuracy and f1-score can appear really different. In general, accuracy is used when the correctly guessed samples (true positive and true negative) are more important, whereas F1-score is used when the falsely predicted values (false positives and false negatives) are crucial. Accuracy may seem the most straightforward and trustworthy metric, but it is only useful when we deal with balanced classes; otherwise it can be misleading, as high metrics don't show prediction capacity for the minority classes. On the other hand, f1-score gives equal weight to precision and recall, when most times one is more important (depending on the problem).

- Similarly, macro f1-score and micro f1-score can show large deviations. As we observed before, the macro-average f1-score will treat all the classes equally, and thus favouring minority classes, where the micro-average f1-score focuses on the overall hits and misses.

- Nevertheless, picking a good metric always depends on the data we are examining. Precision, for example, is more important if our goal is to minimize our false positives, and recall is more important if we try to minimize our false negatives. More specifically, in case we use a classifier to detect an illness, we would seek high recall so as to detect as many sick people as possible, since falsely classifying someone will lead to them not seeking additional help. On the other hand, when it comes, to movie recommendations, we would try and make sure that all (or most of) the recommended movies are relevant, and so seek higher precision.
  We see that in cases like that, accuracy and f1-score are not enough to pick the better model.

In our case, by observing the last histogram we can see that the classes are not perfectly balanced but not completely unbalanced either. Thus, accuracy is still a trustworthy metric to use. Precision seems to be more important than recall, but we mainly need overall success. So we mostly rely on accuracy and f1-score.

We can deduct that the metrics referring to all the classes are coherent and indicative. We can also see that the precision and recall metrics, especially in individual classes can be very different.

Comparing our models, the LSTM model with the beat-synced mel spectograms had better results than the rest. The reason why it proved better than the one with the mel spectograms could be that the smaller sized spectograms resulted to less overfitting. The model with chromograms as inputs had the worse results, which intuitively makes sense, obsvering the information they provide from the plots in above steps. The model with concatenated spectograms and chromograms as inputs did better than the ones with the respective inputs alone, as expected, since it had more useful data for the classification.
In general, music genre classification is a hard problem and simple LSTM neural networks are not ideal. Hence we see these poor results.

Step 7

a

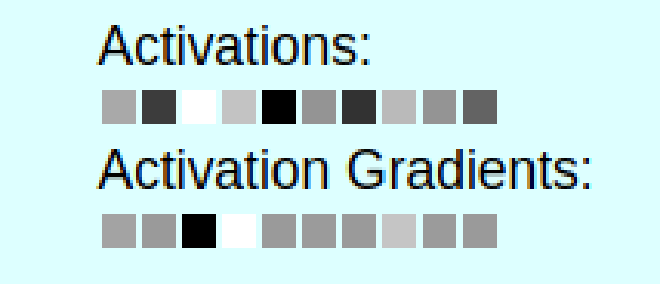
As an introduction to Convolutional Neural Networks we examine the results and training procedure of a CNN on the MNIST digits dataset. By observing the activations and the filter weights in the first convolutional layer we can see that the filters that operate directly on the raw pixel values learn to extract low-level features, such as lines, edges and corners. Eight filters are used and thus eight feature maps are extracted. The relu layer then dismisses the pixels with small values, whereas the max pooling layer summarizes the response over a 2x2 neighborhood and leads to maps of half the size. In the next CNN layer we can see that higher level shapes are learned by the filters and 16 feature maps are extracted. As we go deeper into the network, the feature maps look less like the original image and more like an abstract representation of it, due to the fact that deeper feature maps encode high level concepts.
Finally, after repeating the process, in the fully connected layer the last components are projected to the number of classes. Using that, the softmax layer computes the posterior probabilities, resulting in the classification of the sample.

Below we see some screenshots in mid-training of the model.

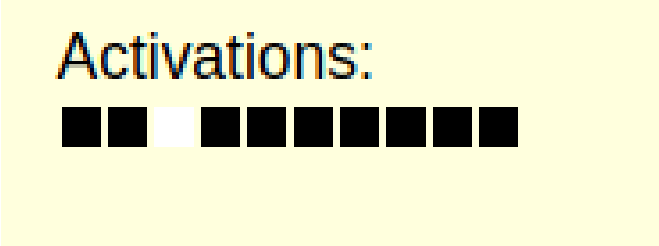
Input

Activations of the first convolutional layer

Activations of the second convolutional layer



(α′) Activations of the fully connected layer

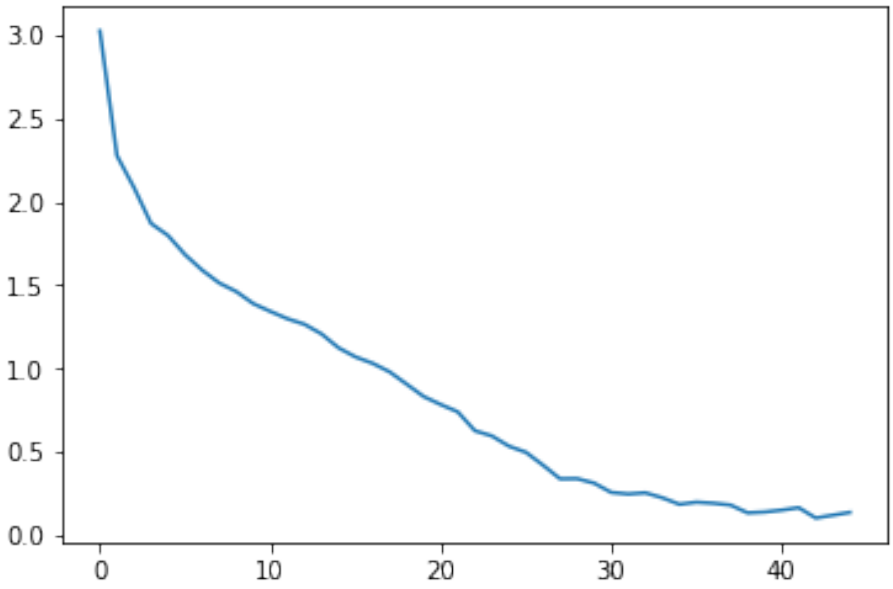


(β′) Activations of softmax

b

In this step we create a 2D CNN with 4 layers. This neural network takes the spectograms as inputs and processes them as single canal images. On each layer, the CNN realizes 2D convolution, batch normalization, ReLU activation and max pooling respectively. In the end, we flatten our resulting matrix and pass it through a fully connected layer. In the next substeps we examine the use of the above techniques. Our architecture has as follows:
The first convolutional layer produced 32 feature maps, the second 64, the third 128 and the last 256. The first 2 max pooling layers operate on 2x2 neighborhoods and the last two on 4x4 neighborhoods. We used a fixed kernel size of 3x3.

We trained the CNN with the train set (as instructed we decided not to use a validation set since we already had a small dataset). We used a learning rate of 0.001 and we employed Adam optimizer with weight decay of 0.0001.Then, by testing with our test set we obtained the results below:



Learning curve for the CNN model

	precision	recall	f1-score	support
0	0.29	0.33	0.31	40
1	0.68	0.65	0.67	40
2	0.66	0.56	0.61	80
3	0.45	0.35	0.39	80
4	0.65	0.70	0.67	40
5	0.39	0.40	0.40	40
6	0.54	0.59	0.56	78
7	0.11	0.07	0.09	40
8	0.39	0.53	0.45	103
9	0.38	0.26	0.31	34
accuracy			0.47	575
macro avg	0.45	0.45	0.45	575
weighted avg	0.47	0.47	0.46	575

Evaluation of CNN

c

Here we dive in the nature of CNNs and their functions.

2D Convolution

Convolution is a mathematical concept used heavily in Digital Signal Processing when dealing with signals that take the form of a time series. It s usually used as a mechanism to combine two functions of time in a coherent manner.
The standard definition of convolution looks like this:

$$(f * g)(x) = \sum_a f(a)g(x - a)$$

The above definition describes one-dimensional convolutions. Similarly, we can think of a two-dimensional convolution as sliding one function on top of another, multiplying and adding. By thinking of images as two-dimensional functions, there have been many applications if 2D convolutions in image processing. Many important image transformations are convolutions where you convolve the image function with a very small, local function called a kernel or filter. In practice, it is a smaller-sized matrix (in comparison to the input dimensions of the image), that consists of real valued entries. It slides to every position and computes a new pixel as a weighted sum of the pixels it floats over. In deep learning, filter weights, instead of being handcrafted, are learned by the network, so as to adapt to the task at hand.

At last, the information is transformed encoded in the pixels and a feature map is produced. We could describe a feature as a distinct and useful observation or pattern obtained from the input data that aids in performing the desired image analysis. The convolutions of different filters over the same image result in different feature maps.

The convolution is fully defined by its characteristics, also called hyperparameters.
Firsty, since we can't connect all the neurons to all the possible regions, we have to define the receptive field, which is the dimensions of the region within which the encompassed pixels are

fully connected to the neural network's input layer.
In many case, zero padding is also used, which is the technique of symmetrically adding zeroes to the input matrix. That way the dimensions of the input volume can be preserved in the output volume.
In addition, we have the stride, which constitutes the number of pixels by which we slide our kernel over the input matrix.
At last, main hyperparameters are the dimensions of the kernel (width, height, depth).

Convolution leverages three important ideas: sparse interactions, parameter sharing and equivariant representations. Sparse weights are achieved by using a kernel smaller than the input. This means that fewer parameters are stored and the memory requirements of the model are reduced. Parameter sharing is achieved in convolutional nets, by using each member of the kernel at every position of the input. This also causes the layer to have equivariance to translation. For example in edge detection, since edges can appear in many places in the image, it is practical to share parameters accross the entire image.

Batch Activation

Batch activation (or batch normalization) is a technique used for training very deep neural networks that manage to stabilize the learning process and dramatically reduce the number of training epochs required to train deep networks. In a nutshell, it standardizes the inputs to a layer for each mini-batch. Standardization refers to re-scaling data to have a mean of zero and a standard deviation of one.
Normalizing the inputs to the layer reduces the number of epochs required to train it. It also has a regularizing effect, reducing generalization error.

ReLU activation

ReLU stands for Rectified Linear Unit and it is a non-linear operation, where $ReLU(x) = max(0, x)$. It is the simplest one from many non-linear functions used at this stage.
Basically, it replaces all the negative pixel values in the feature map with zero, thus introducing non-linearity, getting the data to resemble more like real-world imperfect data. A rectified linear unit outputs zero across half its domain. This makes the derivatives remain large when the unit is active.

Max pooling

Max pooling is a special case of Spatial Pooling which reduces the dimensionality of a feature map, retaining the important information.
In max pooling, a spatial neighborhood in defined and we disintegrate the feature map in windows. Then, from each window we only keep the largest element. Pooling over spatial regions helps to make the representation invariant to translations.

d

Using this model for music genre recognition with spectograms, we can compare it with the one created in step 5a.

Below we can see together the different metrics after the evaluation of each model.

	precision	recall	f1-score	support
0	0.00	0.00	0.00	40
1	0.33	0.62	0.43	40
2	0.28	0.50	0.36	80
3	0.24	0.51	0.33	80
4	0.50	0.03	0.05	40
5	0.00	0.00	0.00	40
6	0.48	0.28	0.35	78
7	0.00	0.00	0.00	40
8	0.34	0.47	0.39	103
9	0.00	0.00	0.00	34
accuracy			0.31	575
macro avg	0.22	0.24	0.19	575
weighted avg	0.26	0.31	0.25	575

(α′) Evaluation of LSTM

	precision	recall	f1-score	support
0	0.29	0.33	0.31	40
1	0.68	0.65	0.67	40
2	0.66	0.56	0.61	80
3	0.45	0.35	0.39	80
4	0.65	0.70	0.67	40
5	0.39	0.40	0.40	40
6	0.54	0.59	0.56	78
7	0.11	0.07	0.09	40
8	0.39	0.53	0.45	103
9	0.38	0.26	0.31	34
accuracy			0.47	575
macro avg	0.45	0.45	0.45	575
weighted avg	0.47	0.47	0.46	575

(β′) Evaluation of CNN

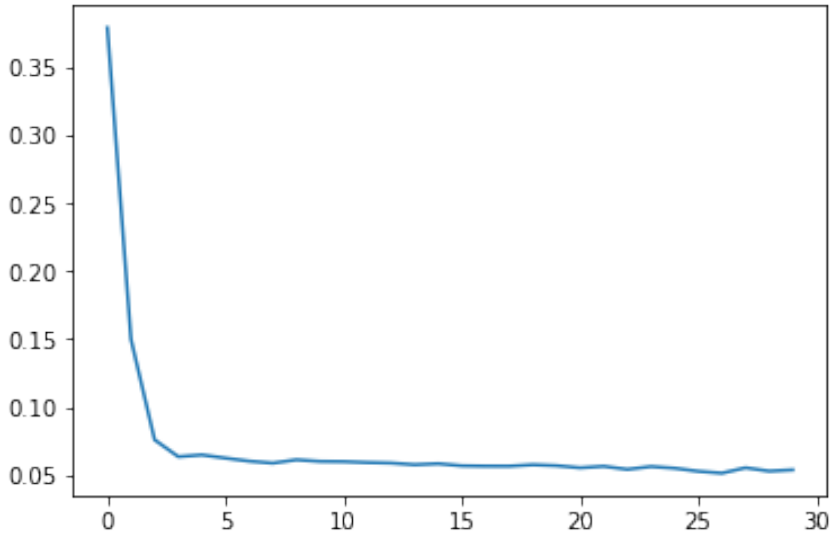
It is obvious that the current model had by far a better performance than the LSTM. This was expected based on the size of the dataset and the complexity of the task.

Step 8

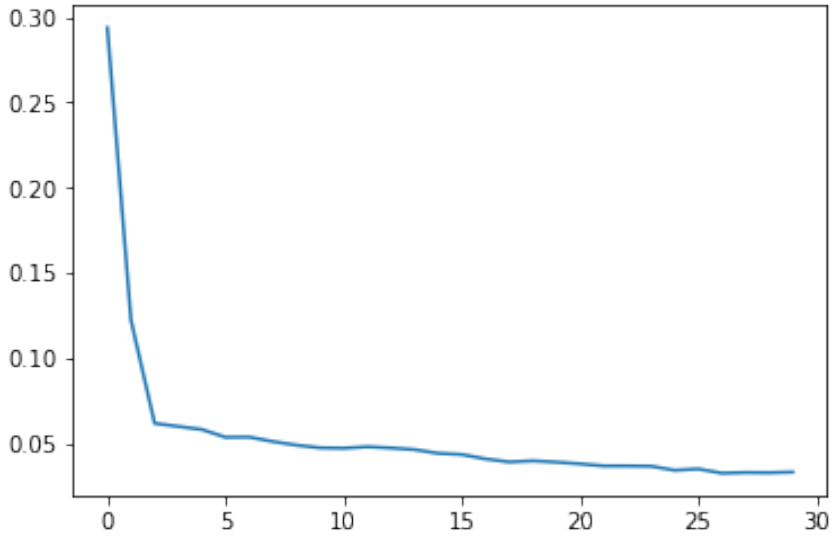
a, b, c, d

In this task we work on the multiset dataset. We train 3 seperate CNN models from the previous step and 3 seperate LSTM models from step 5 in order to predict the values of 3 features: energy, danceability and valence. The LSTM model used was the one in 5b (with the beat-synced spectograms as input data) as it was the one with the best performance.
Since this is a regression task we use minimum least squared error as our loss function. In order to evaluate the efficiency of our models we used the spearman correlation coefficient. It is a non-parametric measure statistical dependence between the rankings of two variables) and assesses how well the relationship between two variables can be described using a monotonic function.

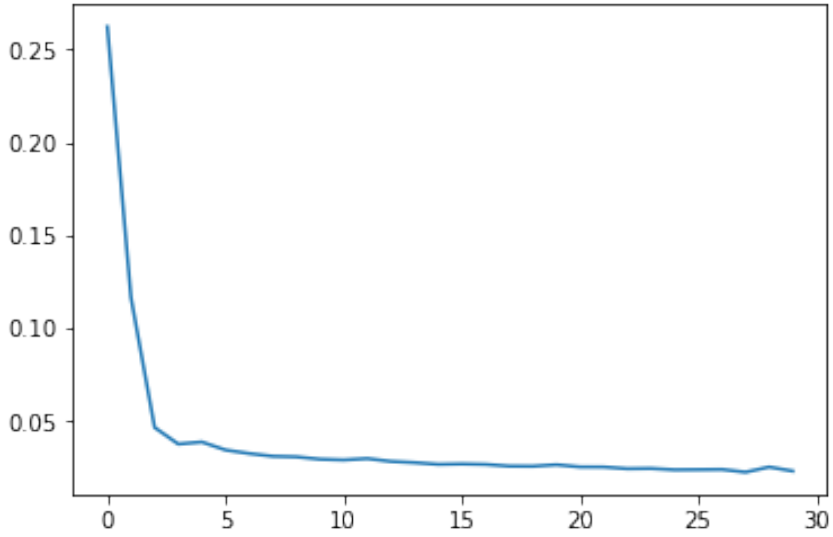
After training both the LSTM and the CNN models for all three features, the following learning curves were produced:



(α') Valence

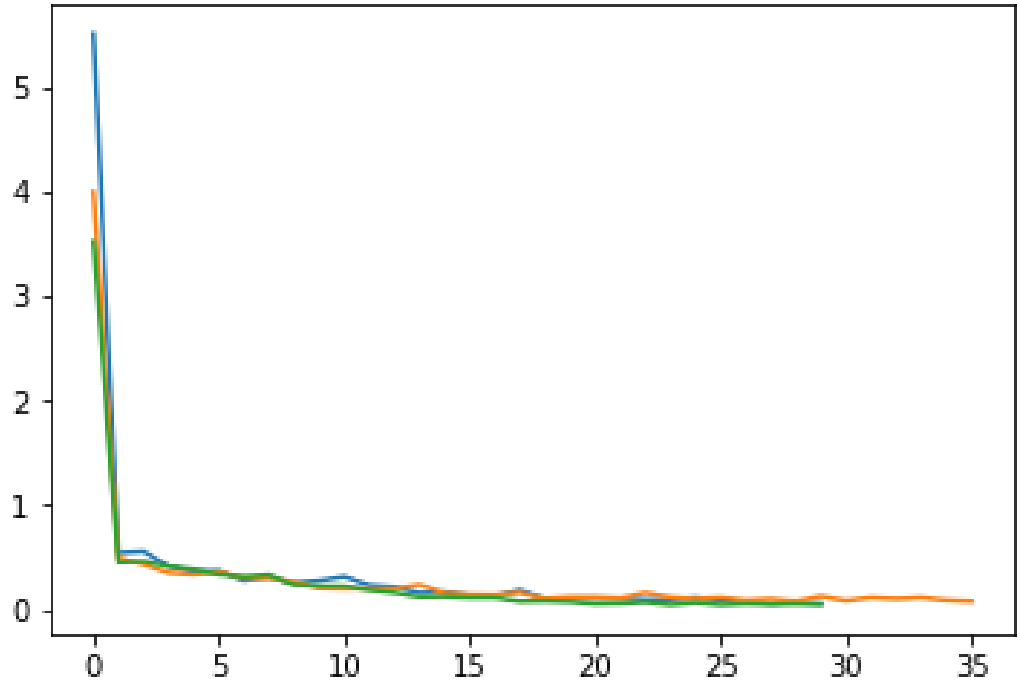


(β') Energy



(γ') Danceability

Learning Curves for the LSTM model



Learning Curves for the CNN model. With blue color we have the loss concerning the energy, with orange the valence and with green the danceability

d

Here we demonstrate the results of the above models. Specifically we present the Spearman Correlation (of each axis and the mean one) of the test sets.

- In the LSTM model we had:
  - Spearman Correlation for Energy: 0.714663
  - Spearman Correlation for Valence: 0.421551
  - Spearman Correlation for Danceability: 0.603529
  - Mean Spearman Correlation: 0.57991442642793
- In the CNN model we had:
  - Spearman Correlation for Energy: 0.757193
  - Spearman Correlation for Valence: 0.569181
  - Spearman Correlation for Danceability: 0.621002
  - Mean Spearman Correlation: 0.6491253766780033

Once again we notice that the CNN model proved to have better performance in genre classification.

Step 9a

In this step we focus on Transfer Learning. In transfer learning, we first train a base network on a base dataset and task, and then we repurpose the learned features, or transfer them, to a second target network to be trained on a significantly smaller target dataset and task. There are two strategies that can be considered: backpropagating the errors from the new task into the base features to fine-tune them to the new task, or freezing the feature layers.

a

In the provided paper by Yosinski et al., they try to quantify the generality and specificity of of neurons in the different layers of a CNN. This is inspired by the fact that on natural images, the first layer learns features similar to Gabor filters and color blobs.
More specifically, they defined a way to characterize a layer as specific or general (based on how well the features are transferred, experimentally point out issues that worsen transfer learning, compare the performance of transfer learning based on the similarity of the tasks and compare it with random weight initialization.

In order to achieve the above, they constructed pairs of non-overlapping subsets from the ImageNet dataset. They produced and tested both similar and dissimilar pairs of sets.

The basic conclusions that they reached to are that:

- optimization difficulties were proven be worse in the middle of a network than near the bottom or top. This indicates that in the middle layers features interact with each other in a complex or fragile way and their co-adaptation can not be learned solely by the upper layers alone
- More generally, the first two layers were proven quite "general" and provided a high performance, whereas for higher layers, the performance dropped significantly. This drop is due to the
  - lost co-adaptation of the neurons (as explained above)
  - the decreasing generality of the features, since higher level neurons are specialized to their original task
- transferring features and then fine-tuning them results in networks that generalize better than those trained directly on the target dataset, possibly because the effects of training with the base dataset does not completely fade out, boosting generalization performance.
- features transfer more poorly (they are more specific) when the datasets are less similar. More specifically, the transferability gap with frozen features grows quicker for dissimilar tasks than for similar ones
- even transfer learning from a dissimilar task is better than using random filters

b

For this step, we chose the CNN model developed in step 7. It was the obvious choice because between this one and the LSTM models in step 5, since the CNN model demonstrated better results. As we noticed before, CNN is a more suitable model for this task, taking into consideration its complexity and the nature of the dataset.

c

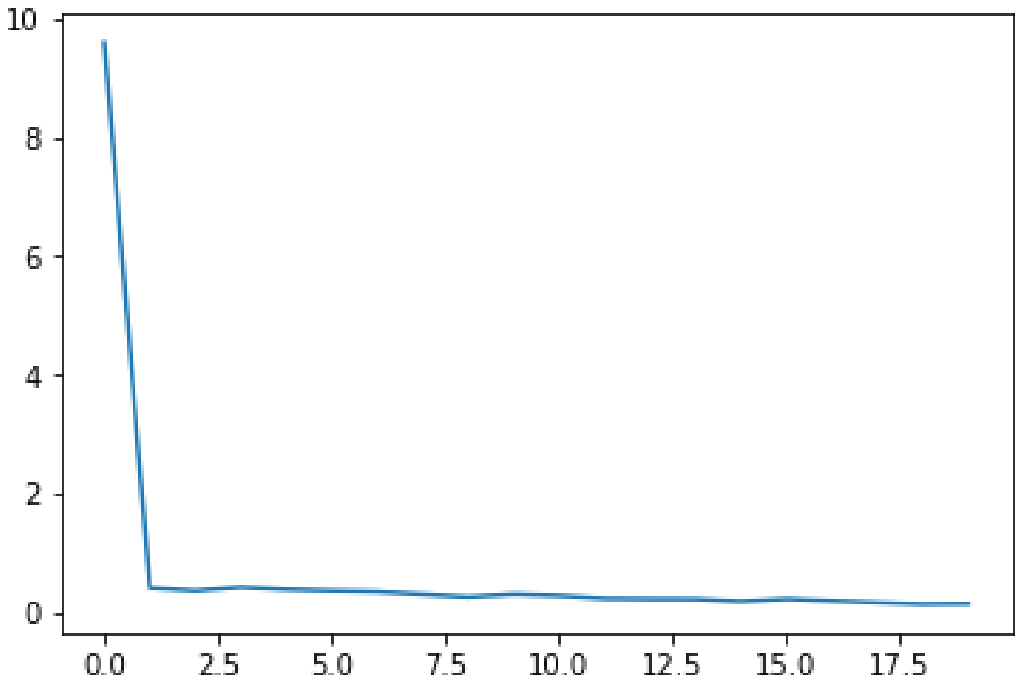
We pretrained this model in the fma_genre_spectrograms dataset in step 7b.

d

Using the resulted weights, we implement the transfer learning technique using fine tuning taking as input the multitask dataset. We freezed the covolutional layers, thus using the pretrained

net as a feature extractor and we trained the last fully connected layer in the multitask dataset for a few epochs.

Below we can see the results for the energy axis



Learning Curve for CNN after Transer Learning

In the learning curve, we notice a steep decline from the beginning of the training.

The Spearman Correlation for the energy axis is 0.702401.

e

Comparing the results with ones in step 8, we notice similar performance (slightly worse), as expected, even though the tasks are related. For transfer learning to produce better results, we usually need a much bigger dataset to do the initial training with.

Step 9b

In this step we focus on Multi-Task Learning. Multi-Task Learning in its essence involves optimizing more than one loss function, using the information from related tasks to improve. This can be done either with hard parameter sharing (where the hidden layers are shared between all tasks but there are several task-specific output layers) or with soft parameter sharing (where each task has its own model and the distance between the model is regularized).

a

In the paper provided (by Kaiser et al.), they trained different tasks using Multi-Task Learning. More specifically, they focused on training data from the ImageNet database, multiple translation tasks, image captioning, speech recognitions and an English parsing task.
For the implementation of the model, various building blocks from other research studies were

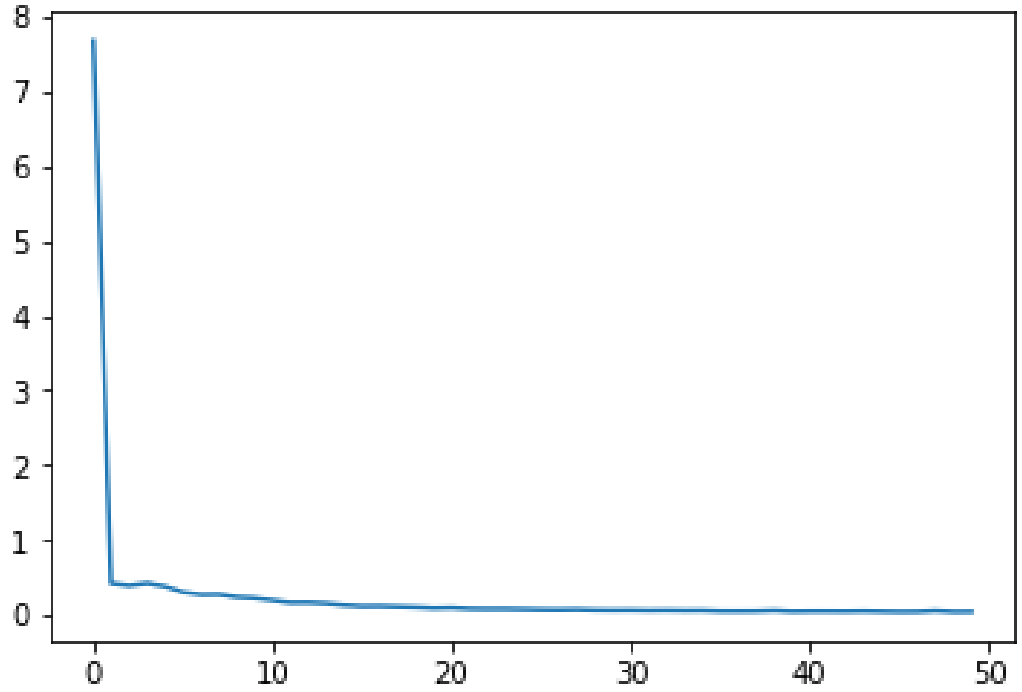
used. Their model conssisted of convolutional blocks which allow the model to detect local patterns and generalize, attention blocks which enable the model to focus on specific elements and Mixture-of-Experts blocks (a number of feed forward neural networks and a gating network that chooses a sparse combination of the previous that will process each input) that give the model capacity without adding to the computation cost. The model they developed consisted of the above building blocks had three main parts: an encoder that processes the inputs, a mixer that mixes the encoded inputs with previous outputs (autoregressive part), and a decoder that processes the inputs and the mixture to generate new outputs.

The basic conclusions that they reached to are seen below:

- Generally, the Multi-Task Learning model's results are similar to the state-of-the-art ones
- Training the tasks simultaneously has similar or better accuracy than training them seperately, especially in the cases of smaller datasets.
- From the above point it is implied that there is transfer learning from tasks with a large amount of available data to ones with more limited datastes.
- Even when exclusively seemingly unrelated tasks are trained in Multi-Task Learning, the performance is proven better than when training the tasks separately.
- Building blocks that seem unrelated or not crucial to a specific task, either don't affect the performance or, in most cases, slightly improve it
- This paper is proof that a single deep learning model can jointly learn a number of large-scale tasks from multiple and different domains.

b

Here we trained our own Multi-Task Learning Model using hard parameter sharing. More specifically, the cost function used is the sum of the individual costs for valence, energy and danceability. The CNN model used was the one from step 7. We used a learning rate of 0.001, weight decay equal to 0.0001 and number of epochs equal to 50.
Below we can see the learning curve of the model.



Σχήμα 10: Learning Curve for Multi-Task Learning Model

The results for this model have as follows:

- Spearman Correlation for Valence: 0.6188217112057082
- Spearman Correlation for Energy: 0.7659793005536225
- Spearman Correlation for Danceability: 0.7587220662621901
- Mean Spearman Correlation: 0.714508

c

Overall, in Step 8 we noticed mean spearman correlation equal to 0.6491253766780033 (for the CNN model), whereas after using Multi-Task learning we had mean spearman correlation equal to 0.714508. As expected, the performance improved. Even if in some cases the results for individual axis might worsen (in our case for the energy), in general we notice similar or better results, overall improving the performance.

Step 10

a, b, c

At last, using the model from the previous step to approximate the results for valence, energy and danceability. After forming a solution.txt file as described, we joined the kaggle competition under the name "Arrested Developers".
Taking into consideration 30% of the data, we achieved a score of 0.74187 on the Leaderboard.

d

The results were close to the ones we produced with our data. In general, using drop out added a randomness to the results, so every time we had a small deviation. Of course, the results

for all of the test data would provide a more complete idea about the performance of our model, with less deviation.