

Pattern Recognition - 3rd Lab (Pre-lab)

Konstantinos Papadakis (DSML 03400149)

Step 0

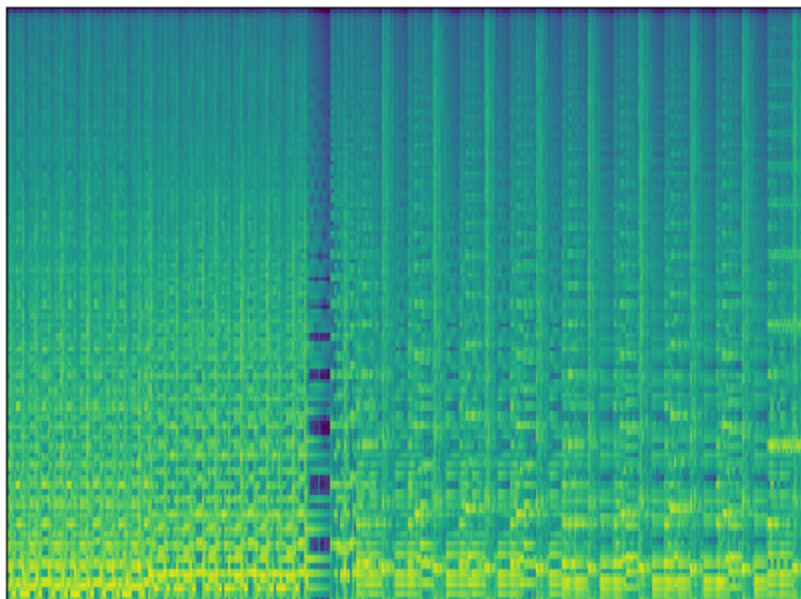
A main.py file with the same code used on kaggle has been included.

Step 1

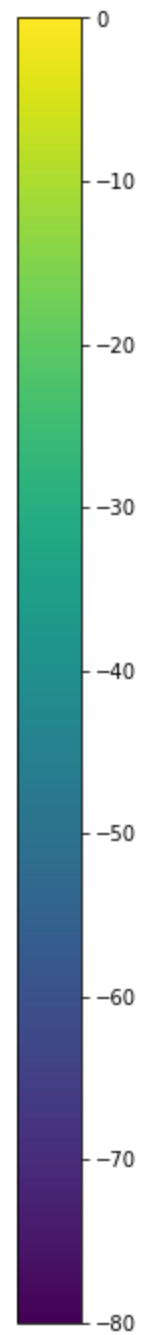
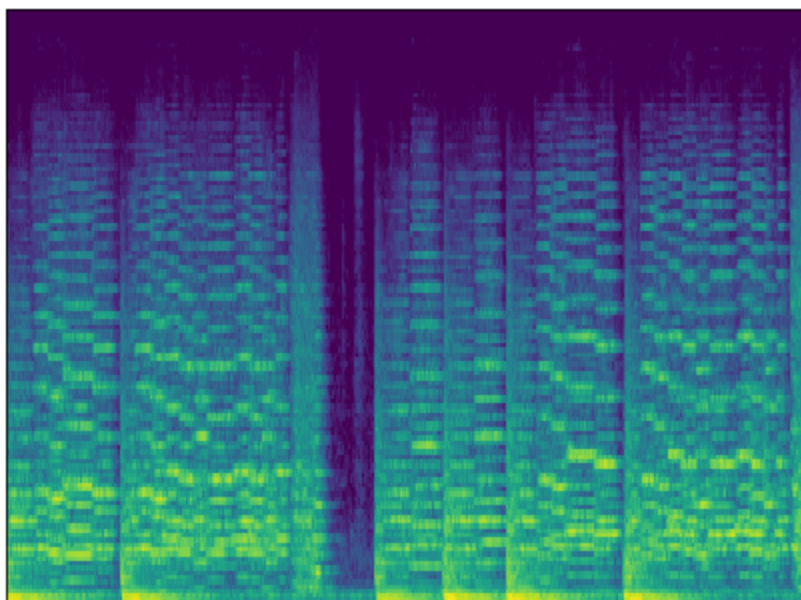
In our example we chose Electronic music vs Classical music. We see that the Electronic sample is more tightly and discretely structured, while the Classical sample is more fluid and continuous, and this holds both for the mel spectrogram and the chromatogram. Also, from the mel spectrograms we see that the Electronic sample has harmonics over the entire frequency range, while the Classical sample does not. Finally, notice the regular vertical lines in the Electronic samples which are a result of a regular rhythm

Mel frequencies (Raw)

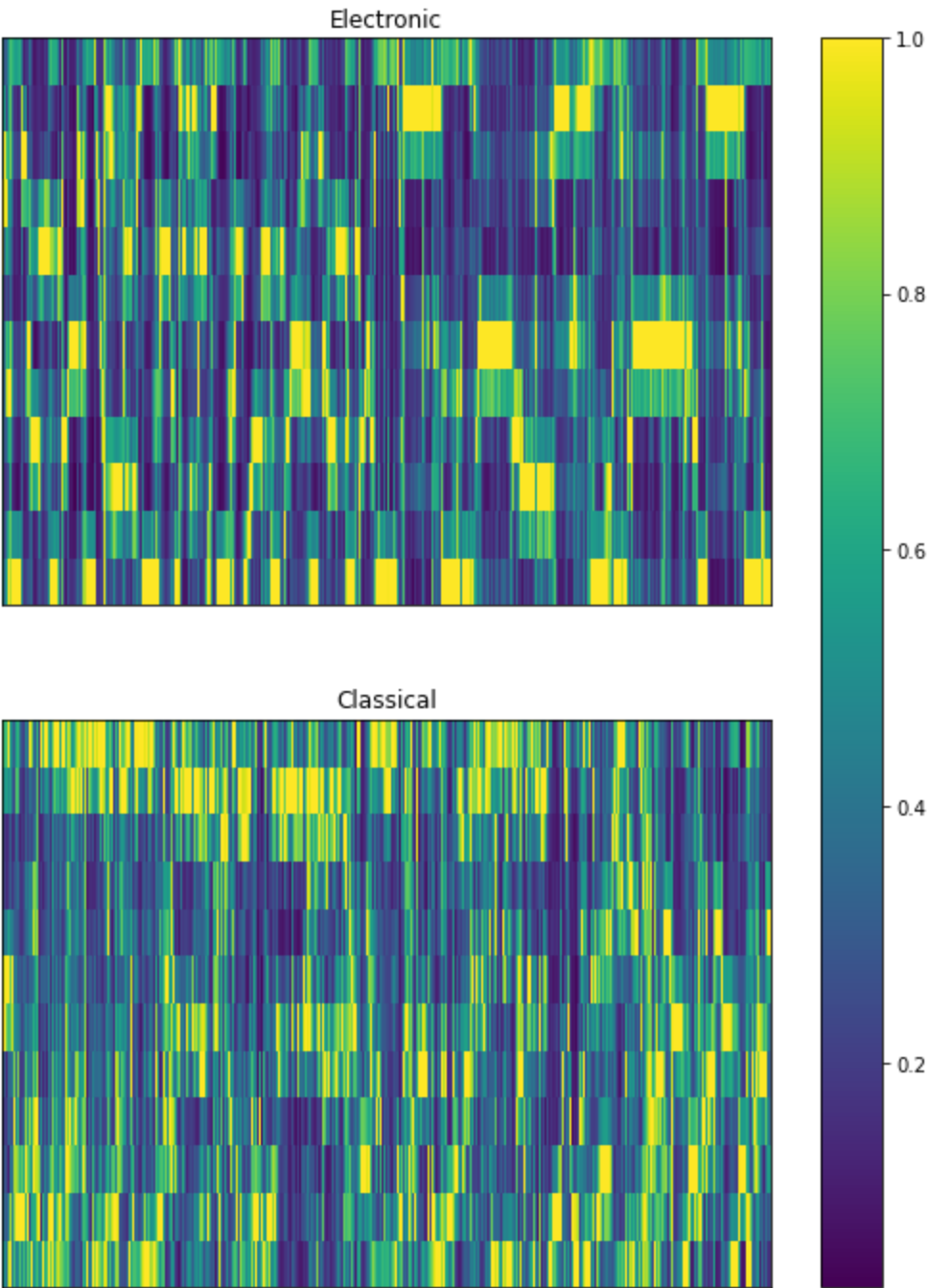
Electronic



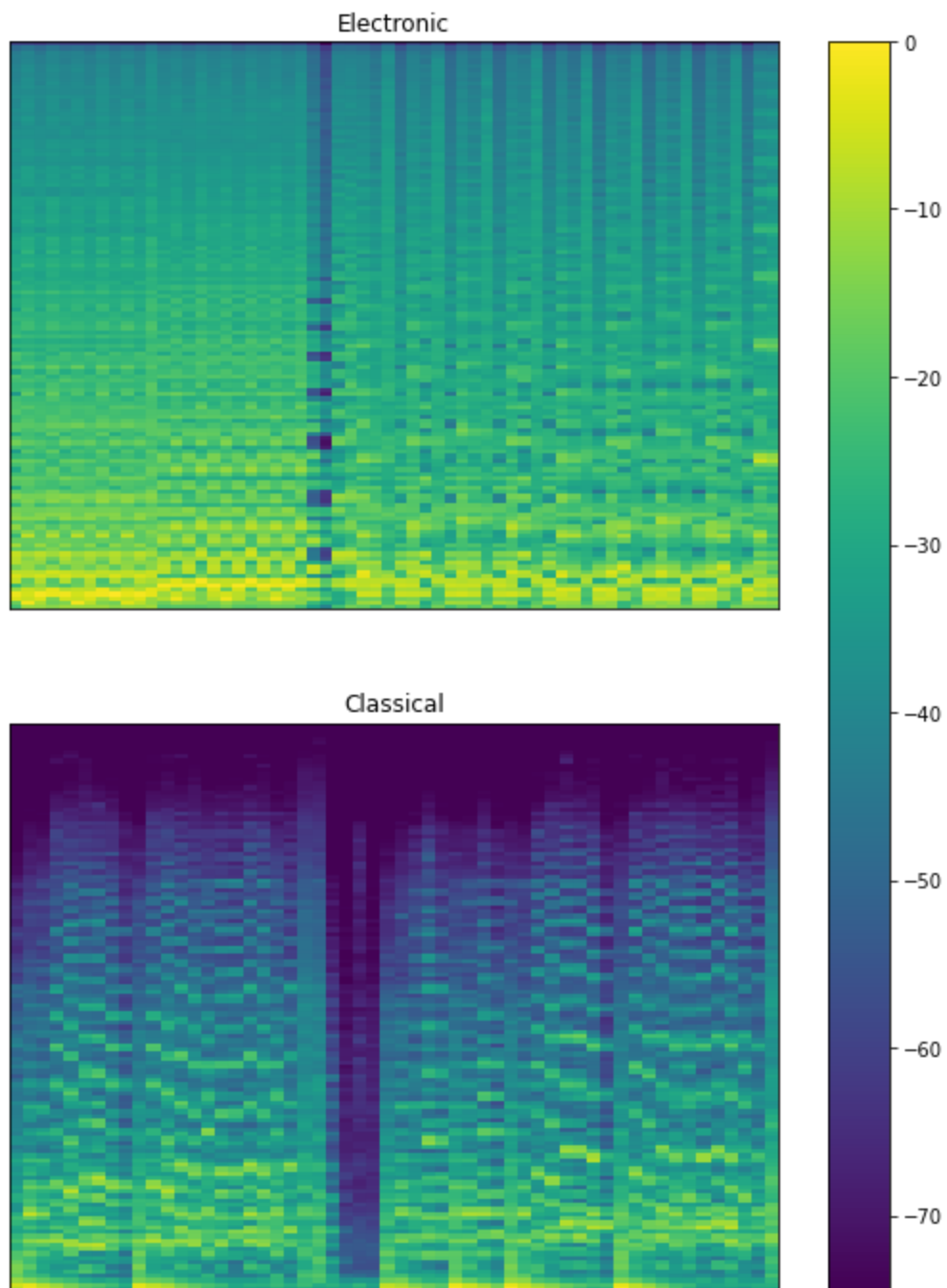
Classical



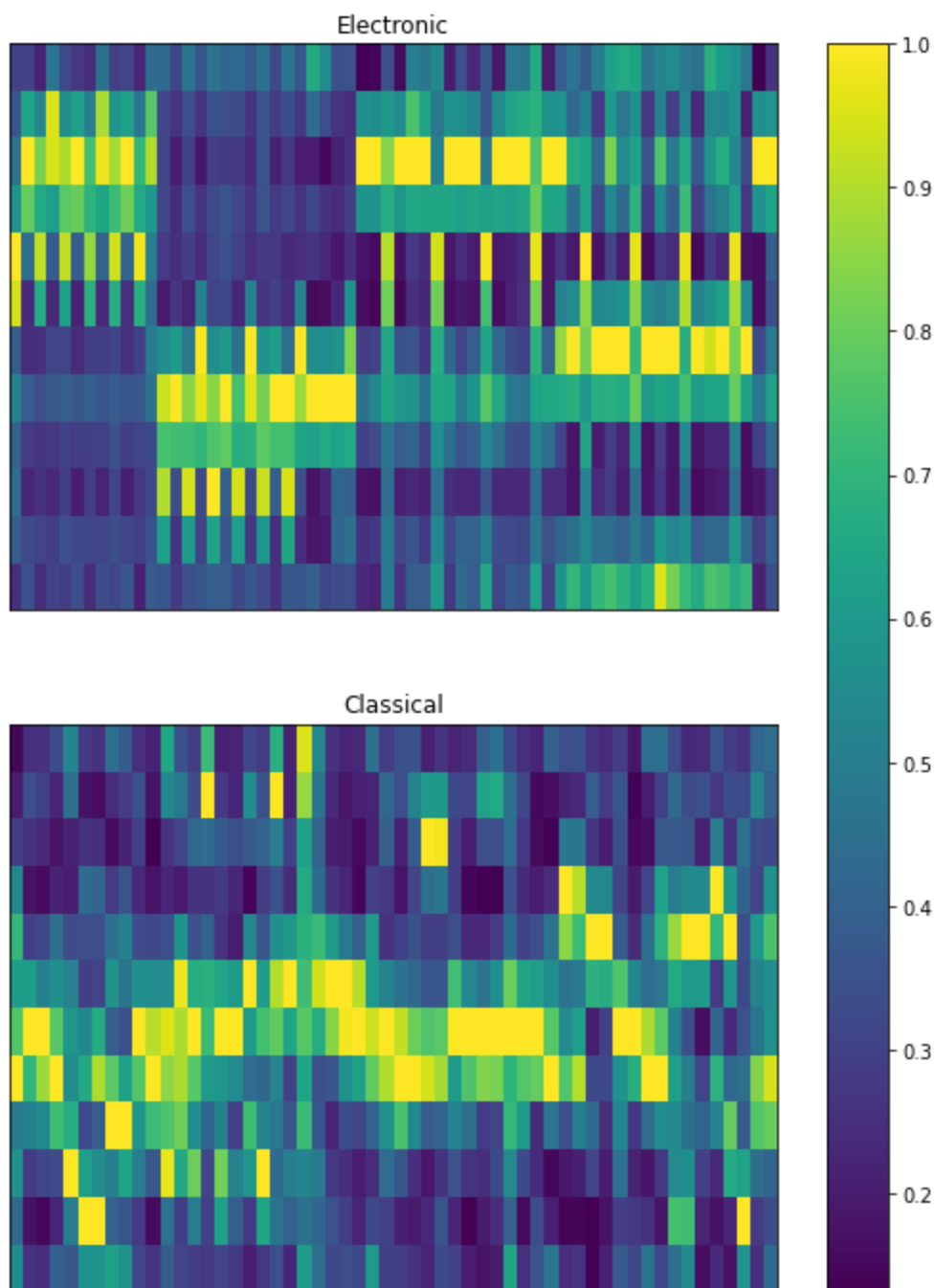
Chromagrams (Raw)



Mel frequencies (Beat-Synced)



Chromagrams (Beat-Synced)



Step 2

Mel frequencies (Raw) shape: (1293, 128)
Chromatograms (Raw) shape: (1293, 12)
Mel frequencies (Beat-Synced) shape: (62, 128)
Chromatograms (Beat-Synced) shape: (62, 12)

As we see, the size of each raw sample is above 150,000 which will require a lot of resources to train. On the other hand beat-synced samples have a size of roughly 750, which will make training easier. Also, because LSTM is a sequential model, the time dimension can't be parallelized, instead we need to compute it step by step.

The plots of the beat-synced data are shown above. We notice that much of the original information is preserved and a lot of noise has been removed.

Step 3

The plots are shown above.

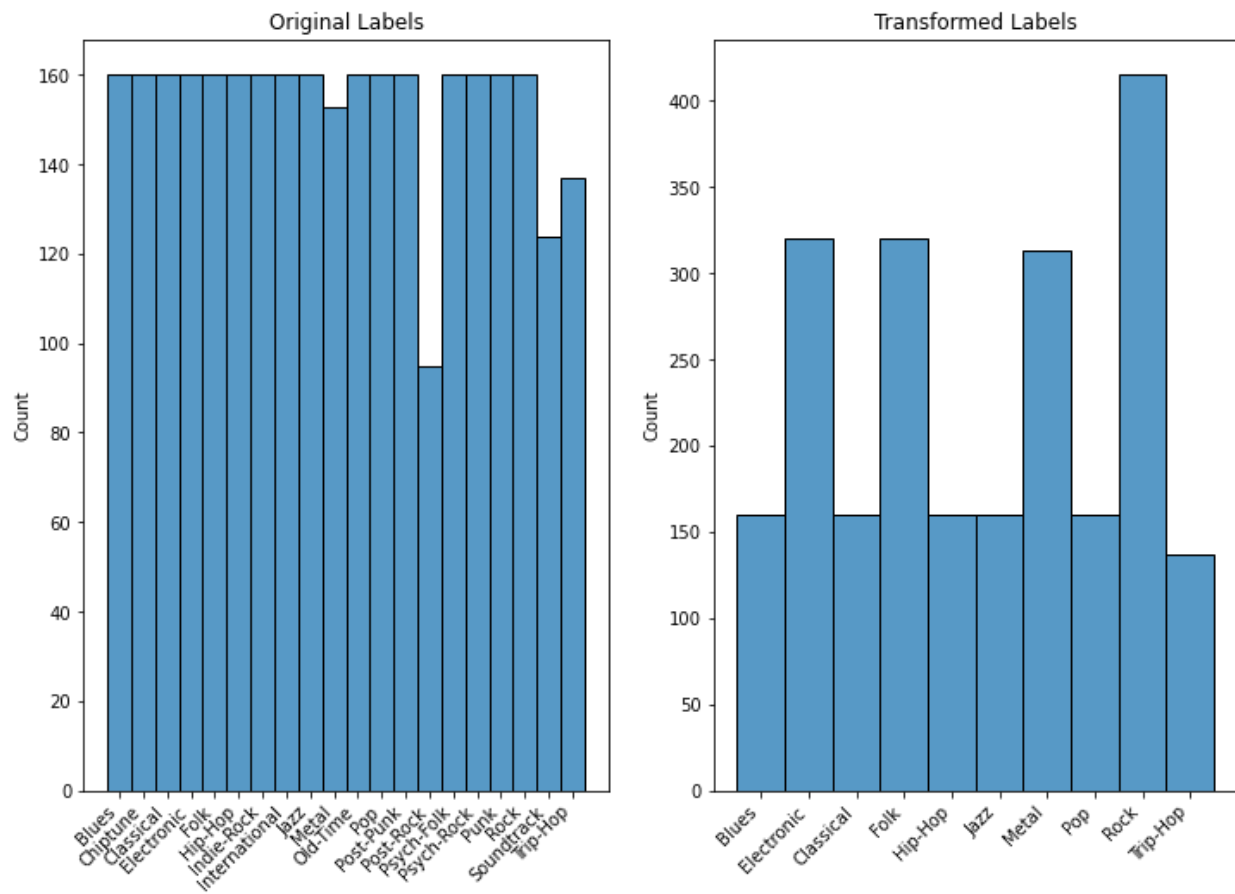
Step 4

I have reimplemented the data loading code so that the data is loaded on demand instead of all at once. This way very little RAM is used (only what's needed for one batch) and the algorithms are just as fast. Also, we perform padding on batch creation using an appropriate `collate_fn` and split our datasets using `torch's random_split`.

Considering the "TODO"s in the original code:

- QUESTION: Comment on how the train and validation splits are created.
ANSWER: We read the data in arrays, create an array of the indices, we shuffle the indices, and then we split them.
- QUESTION: It's useful to set the seed when debugging but when experimenting ALWAYS set `seed=None`. Why?
ANSWER: Because we would always be training and validating on the same data, which could make the model learn properties specific to that split and which aren't properties of the entire set.
- QUESTION: Comment on why padding is needed
ANSWER: Because PyTorch doesn't support ragged tensors.

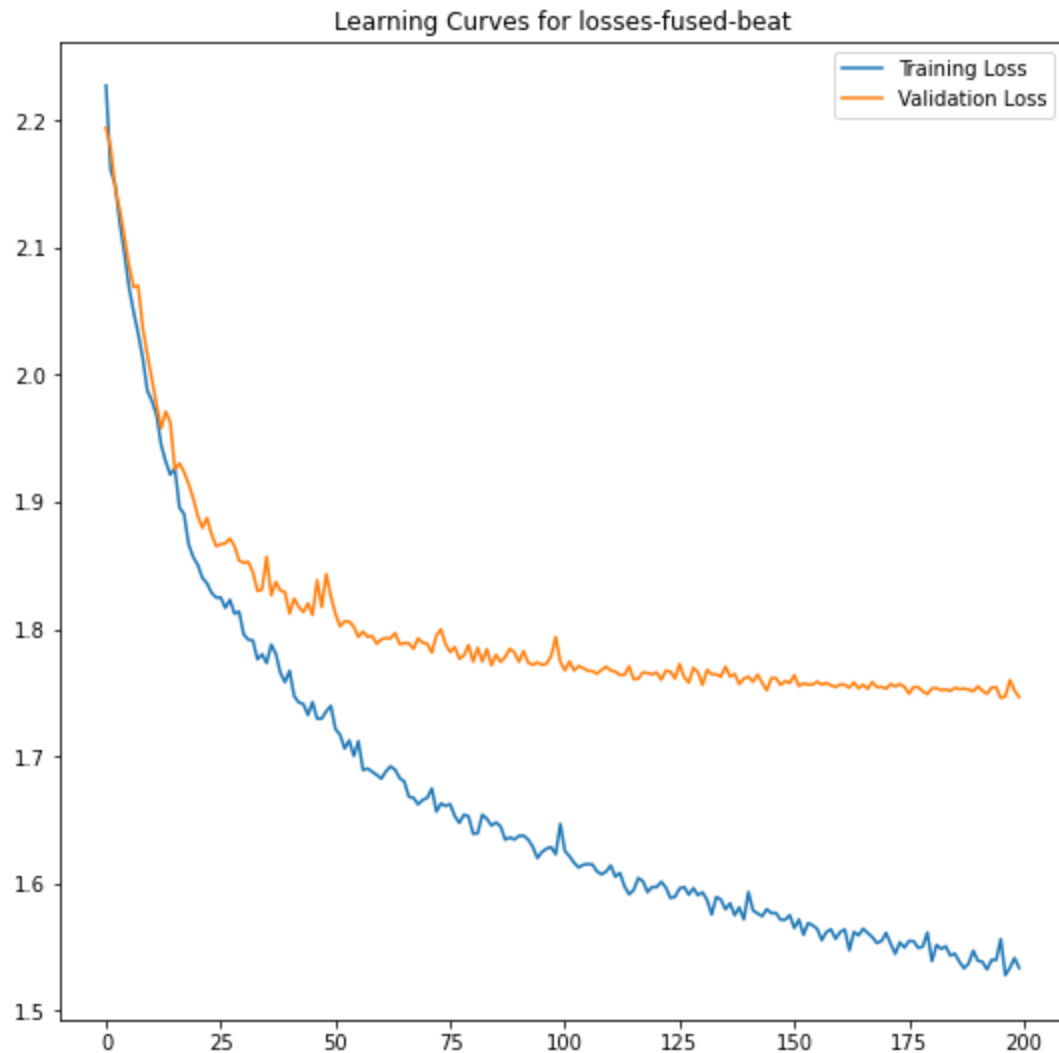
Below we can see the distribution of the labels before and after the `class_mapping`.



Step 5

I first used `overfit_batch=False` to find parameters that reduce the loss, and then used those parameters to train with the full datasets.

Below we can see the learning curves for the beat-synced fused dataset.



Step 6

To calculate precision, recall and f1 of a class C, we view our problem as a binary classification problem, with C being the positive class and all the other classes comprising the negative class. Then we have,

- **Precision:** $\text{True Positives} / (\text{True Positives} + \text{False Positives})$.
Precision is the ability of the classifier not to label as positive a sample that is negative.
- **Recall:** $\text{True Positives} / (\text{True Positives} + \text{False Negatives})$.
Recall is the ability of the classifier to find all the positive samples.
- **F1:** $\text{True Positives} / (\text{True Positives} + (\text{False Positives} + \text{False Negatives})/2)$.
F1 is a mix of recall and precision. Instead of picking either False Positives or False Negatives, we just pick their average.

When we consider all the classes we have

- **Accuracy:** $\text{correct_predictions} / \text{predictions}$

and then we can generalize the inherently binary metrics Precision, Recall and F1, by averaging them in one of the following ways:

- **Macro:** Simply taking the mean of the results
- **Weighted:** Take the *weighted* mean of the results using class frequencies as weights.
- **Micro:** We consider all the True Positives, all the False Positives and all the False Negatives of all classes, and we use their sums to calculate Precision, Recall and F1. Note that micro F1 is the same as Accuracy (if each sample belongs to exactly one class), since $(\text{False Positives} + \text{False Negatives}) / 2 = \text{Number of misclassified data}$ (a false negative for one class is a false positive for another, meaning we double-count).

In our case, all classes are equally important, so we should use their weighted average instead of the simple macro. Micro average is also a good global metric in our case.

The F1 score can be low when either the precision or recall are low. F1 differs from Accuracy mainly when the classes are imbalanced. For example, consider that we have 90 negative samples and 10 positive ones, and a classifier which predicts everything correctly except for 10 negative samples. In that case $TP=10$, $FP=0$, $FN=10$ and we have $\text{Accuracy} = 9/10$ and $F1 = 2/3$.

Micro F1 vs Macro F1 is essentially a generalization of Accuracy vs F1 in a multilabel setting. Micro F1 is equal to Accuracy (if each sample belongs to exactly one class), while the Macro F1 averages the F1 scores for each class giving every class equal weight. This means that a heavily underrepresented class will influence the macro F1 as much as the other classes, but it will have little influence on the micro F1.

Different problems require different metrics.

- We need high recall for problems where catching every positive is important, for example, identifying a severe disease, even at the cost of giving many false positives.
- We need high precision for problems where false positives are costly. For example we don't want to classify an important email as spam, even at the cost of letting some spam go to the inbox.

Below the results for each dataset are listed

Mel raw

	precision	recall	f1-score	support
0	0.00	0.00	0.00	40
1	0.11	0.03	0.04	40
2	0.23	0.68	0.35	80
3	0.32	0.62	0.42	80
4	0.00	0.00	0.00	40
5	0.00	0.00	0.00	40
6	0.33	0.37	0.35	78

7	0.00	0.00	0.00	40
8	0.29	0.25	0.27	103
9	0.00	0.00	0.00	34

accuracy			0.28	575
macro avg	0.13	0.19	0.14	575
weighted avg	0.18	0.28	0.21	575

Mel beat-sync

	precision	recall	f1-score	support
0	0.20	0.03	0.04	40
1	0.35	0.55	0.43	40
2	0.39	0.72	0.51	80
3	0.38	0.51	0.44	80
4	0.20	0.17	0.19	40
5	0.33	0.07	0.12	40
6	0.53	0.50	0.52	78
7	0.00	0.00	0.00	40
8	0.36	0.44	0.40	103
9	0.22	0.06	0.09	34

accuracy			0.38	575
macro avg	0.30	0.31	0.27	575
weighted avg	0.33	0.38	0.33	575

Chroma raw

	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	0.00	0.00	0.00	80
4	0.00	0.00	0.00	40
5	0.00	0.00	0.00	40
6	0.00	0.00	0.00	78
7	0.00	0.00	0.00	40
8	0.18	1.00	0.30	103

9	0.00	0.00	0.00	34
accuracy			0.18	575
macro avg	0.02	0.10	0.03	575
weighted avg	0.03	0.18	0.05	575

Fused raw

	precision	recall	f1-score	support
0	0.00	0.00	0.00	40
1	0.00	0.00	0.00	40
2	0.29	0.65	0.40	80
3	0.31	0.70	0.43	80
4	0.00	0.00	0.00	40
5	0.00	0.00	0.00	40
6	0.26	0.68	0.37	78
7	0.00	0.00	0.00	40
8	0.64	0.07	0.12	103
9	0.00	0.00	0.00	34
accuracy			0.29	575
macro avg	0.15	0.21	0.13	575
weighted avg	0.23	0.29	0.19	575

Fused beat

	precision	recall	f1-score	support
0	0.25	0.03	0.05	40
1	0.36	0.55	0.44	40
2	0.40	0.65	0.50	80
3	0.34	0.51	0.41	80
4	0.22	0.12	0.16	40
5	0.17	0.05	0.08	40
6	0.51	0.51	0.51	78
7	0.00	0.00	0.00	40
8	0.37	0.41	0.39	103
9	0.19	0.15	0.17	34

accuracy		0.37		575
macro avg	0.28	0.30	0.27	575
weighted avg	0.32	0.37	0.32	575