Pattern Recognition - 3rd Lab (Pre-lab)

Konstantinos Papadakis (DSML 03400149)

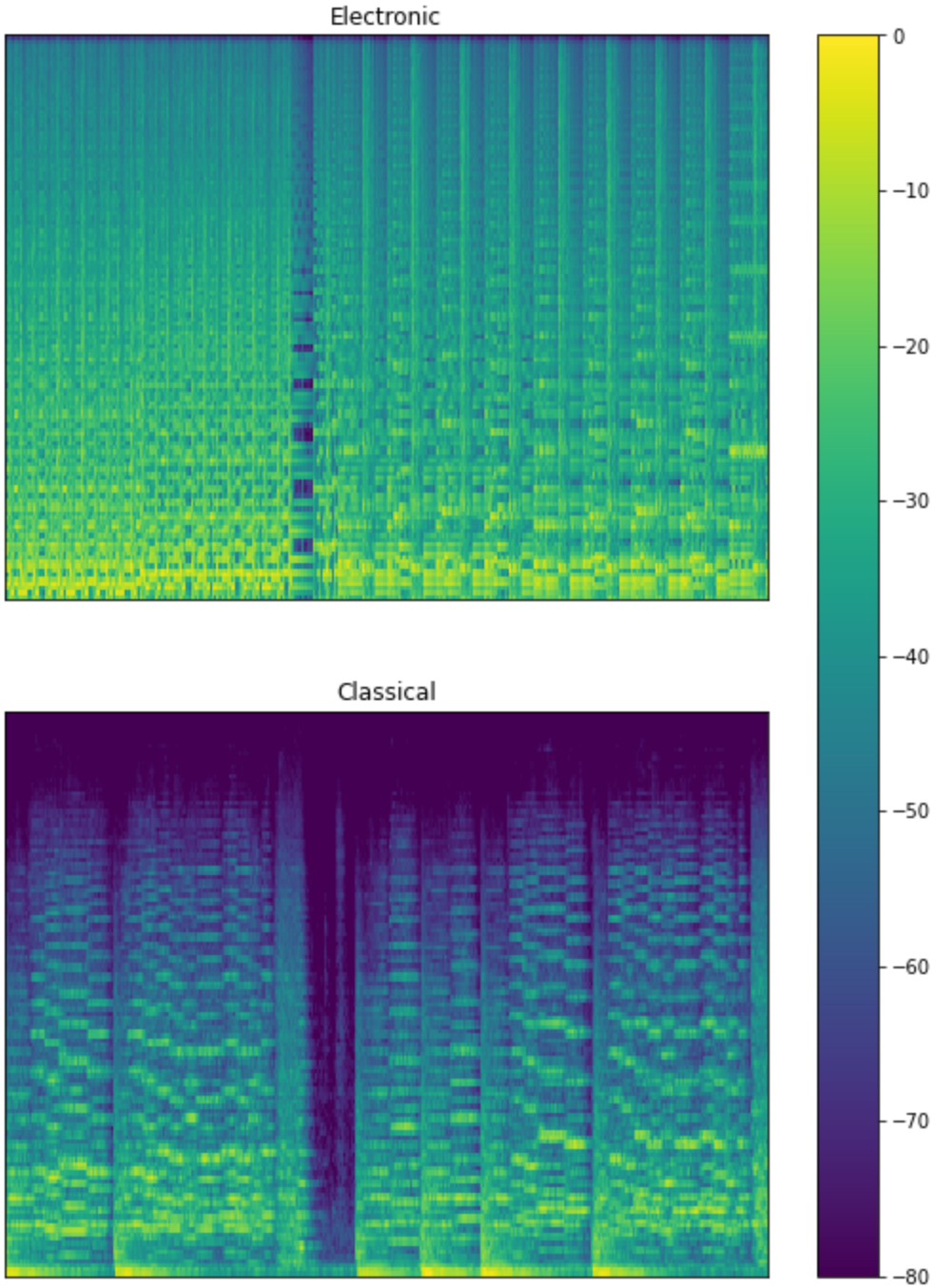
Step 0

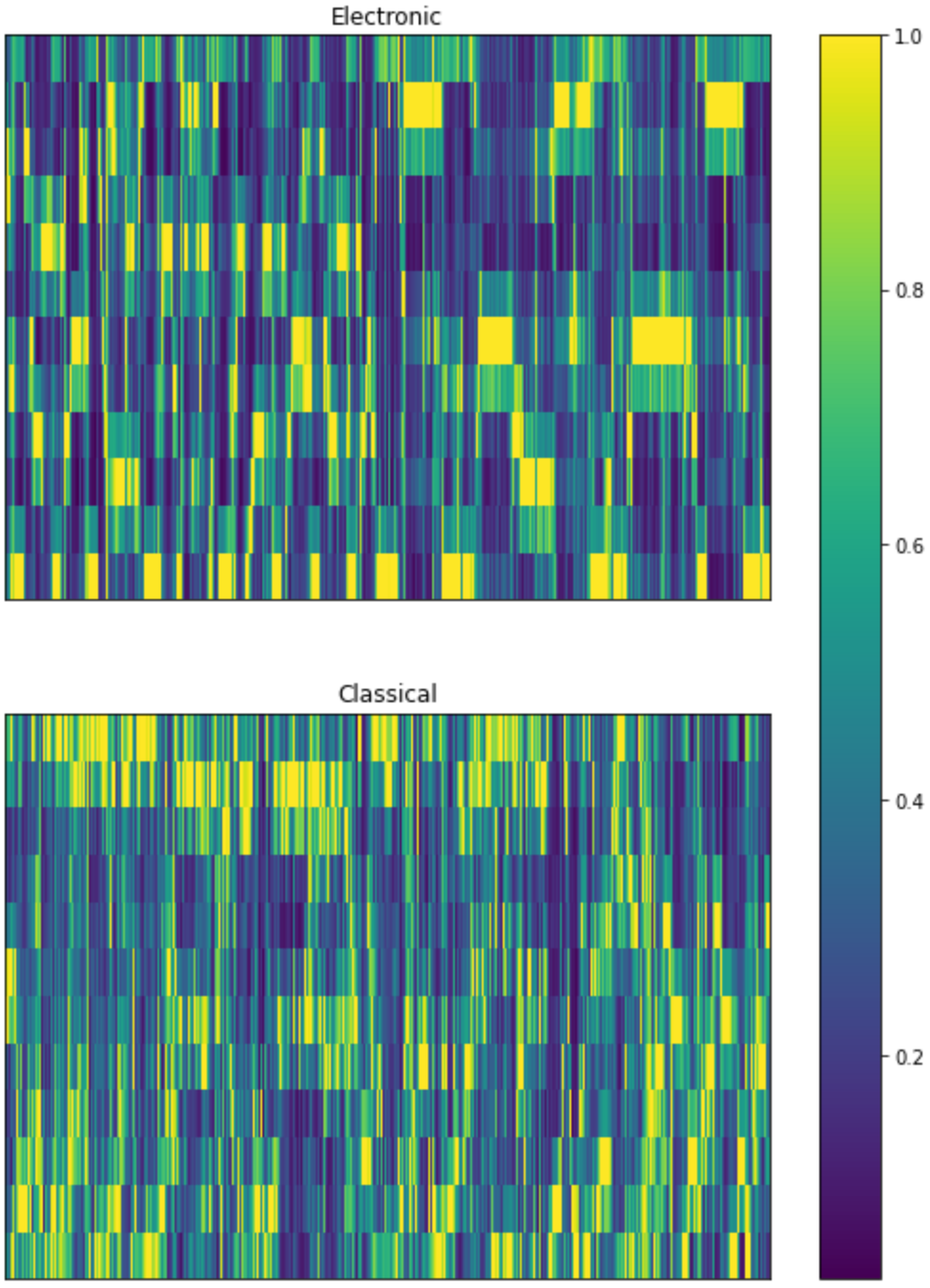
A main.py file with the same code used on kaggle has been included.

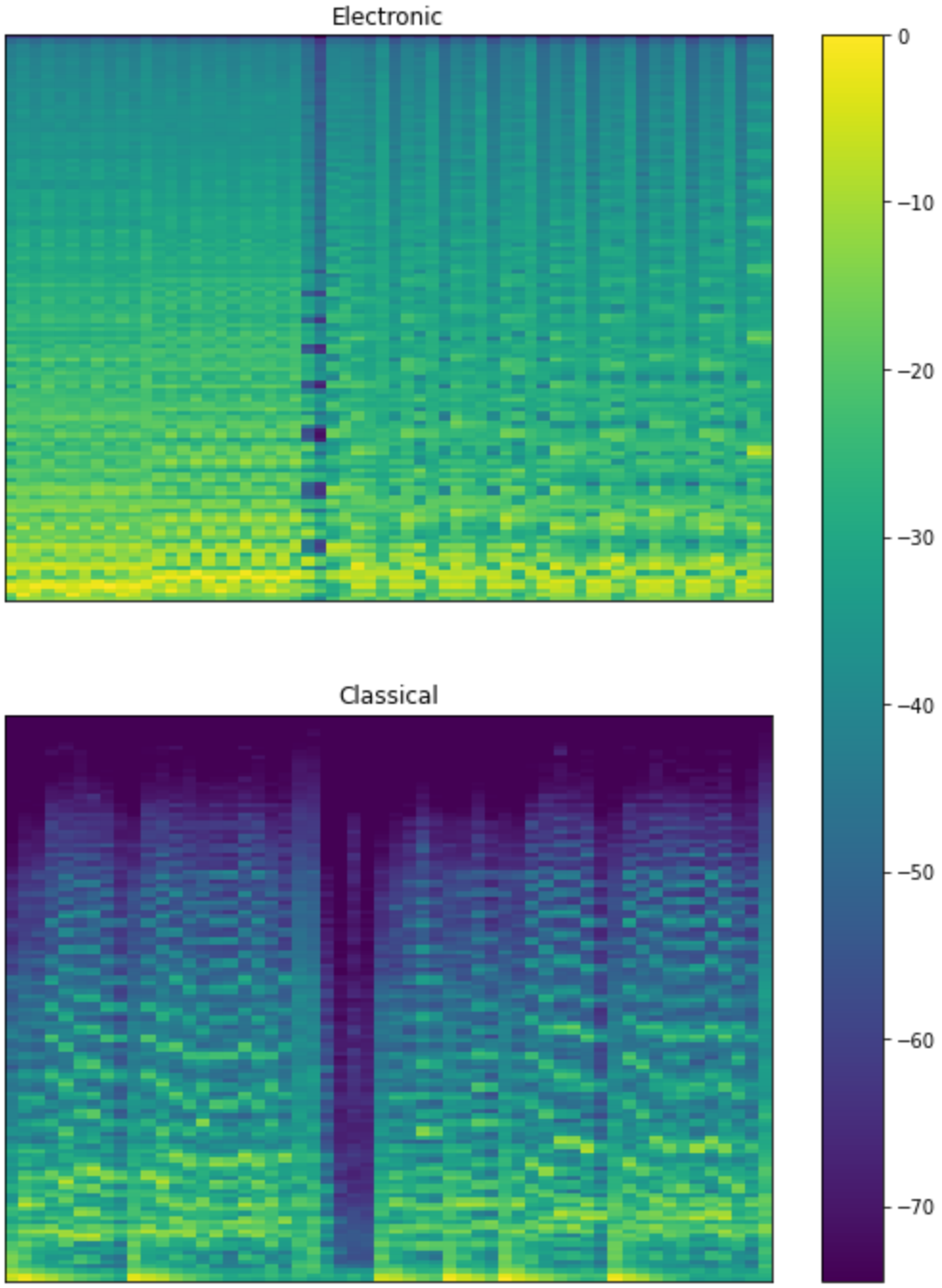
Step 1

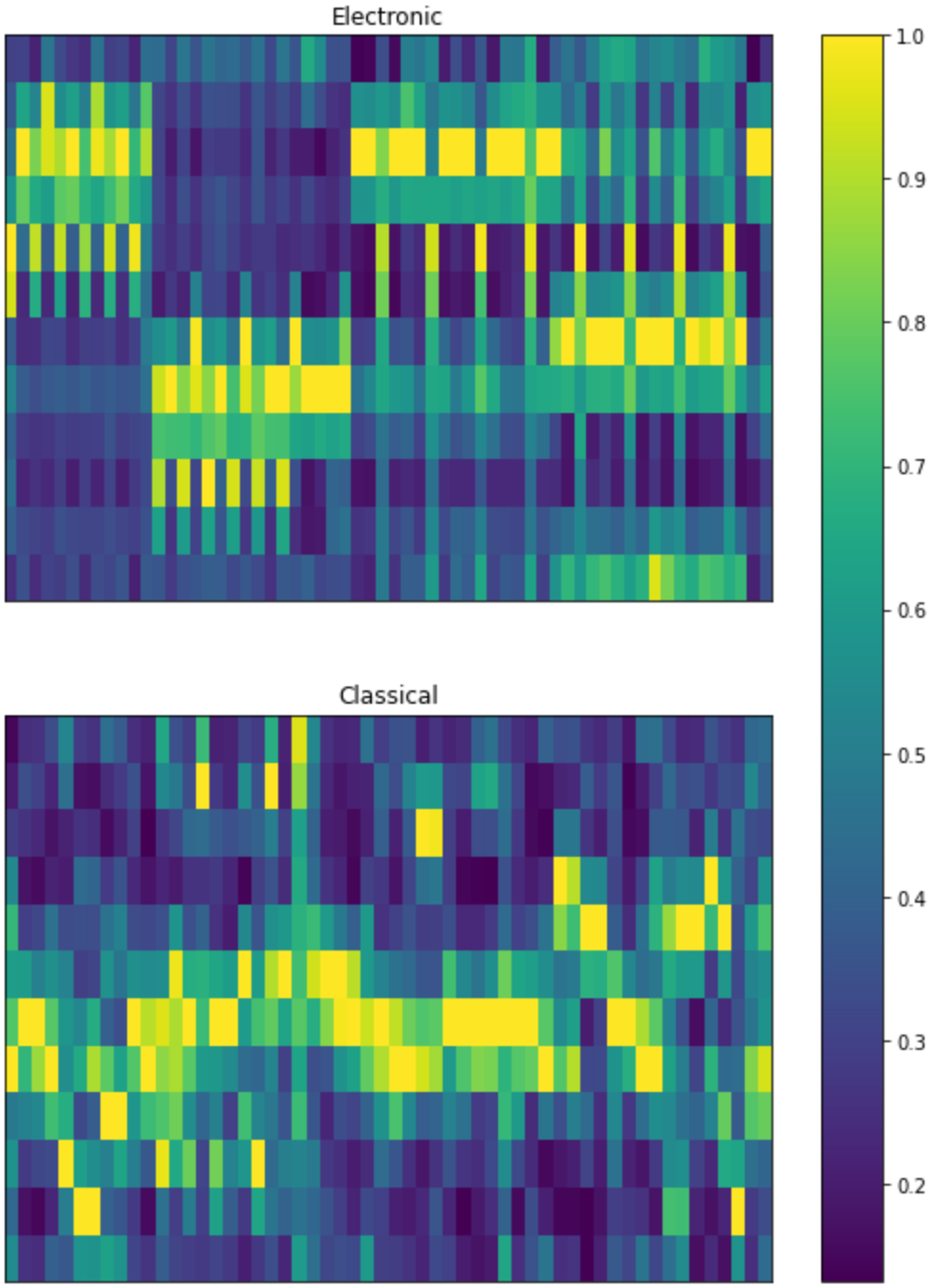
In our example we chose Electronic music vs Classical music. We see that the Electronic sample is more tightly and discretely structured, while the Classical sample is more fluid and continuous, and this holds both for the mel spectrogram and the chromatogram. Also, from the mel spectrograms we see that 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)




Chromagrams (Raw)
Electronic
Classical
1.0
0.8
0.6
0.4
0.2


Mel frequencies (Beat-Synced)
Electronic
Classical
0
−10
−20
−30
−40
−50
−60
−70


Chromagrams (Beat-Synced)
Electronic
Classical
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2

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.

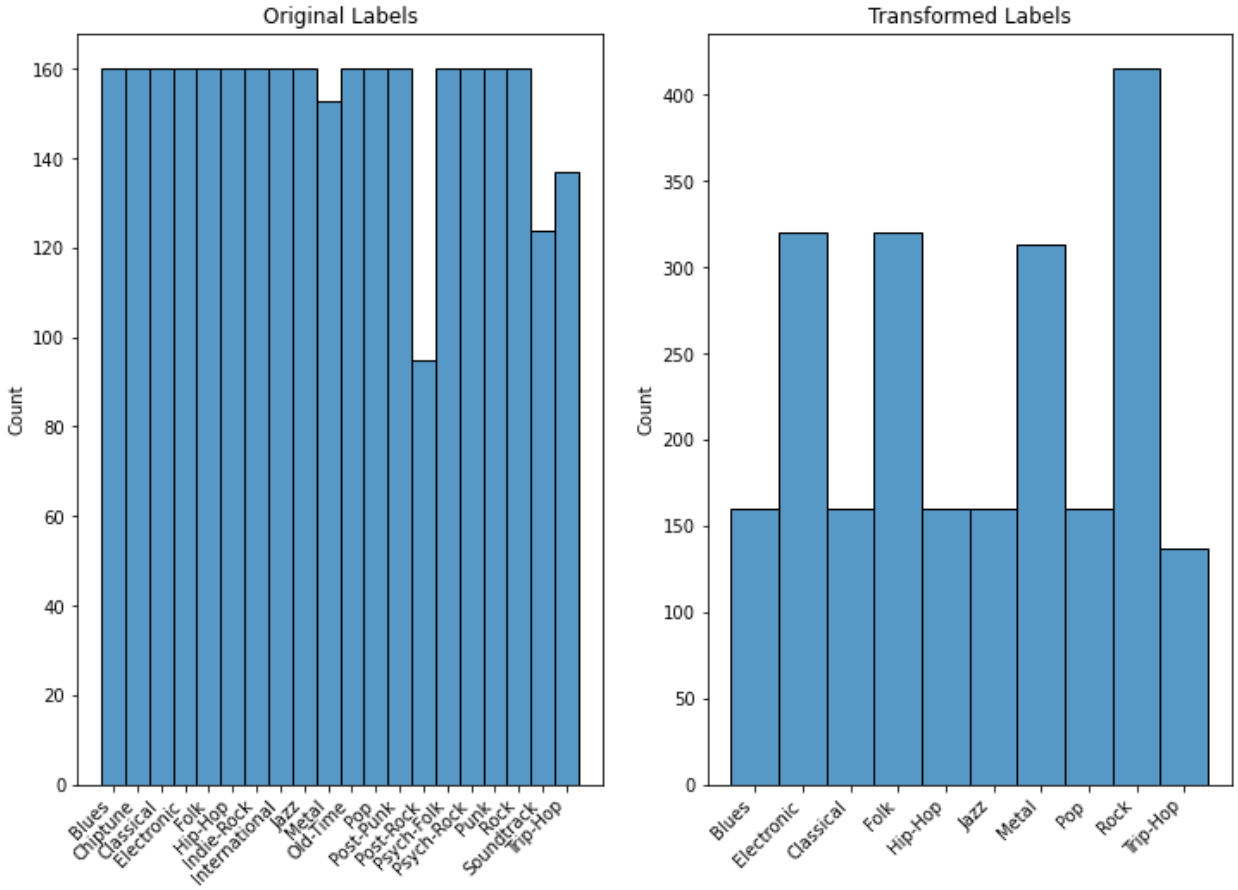
Step 4

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:

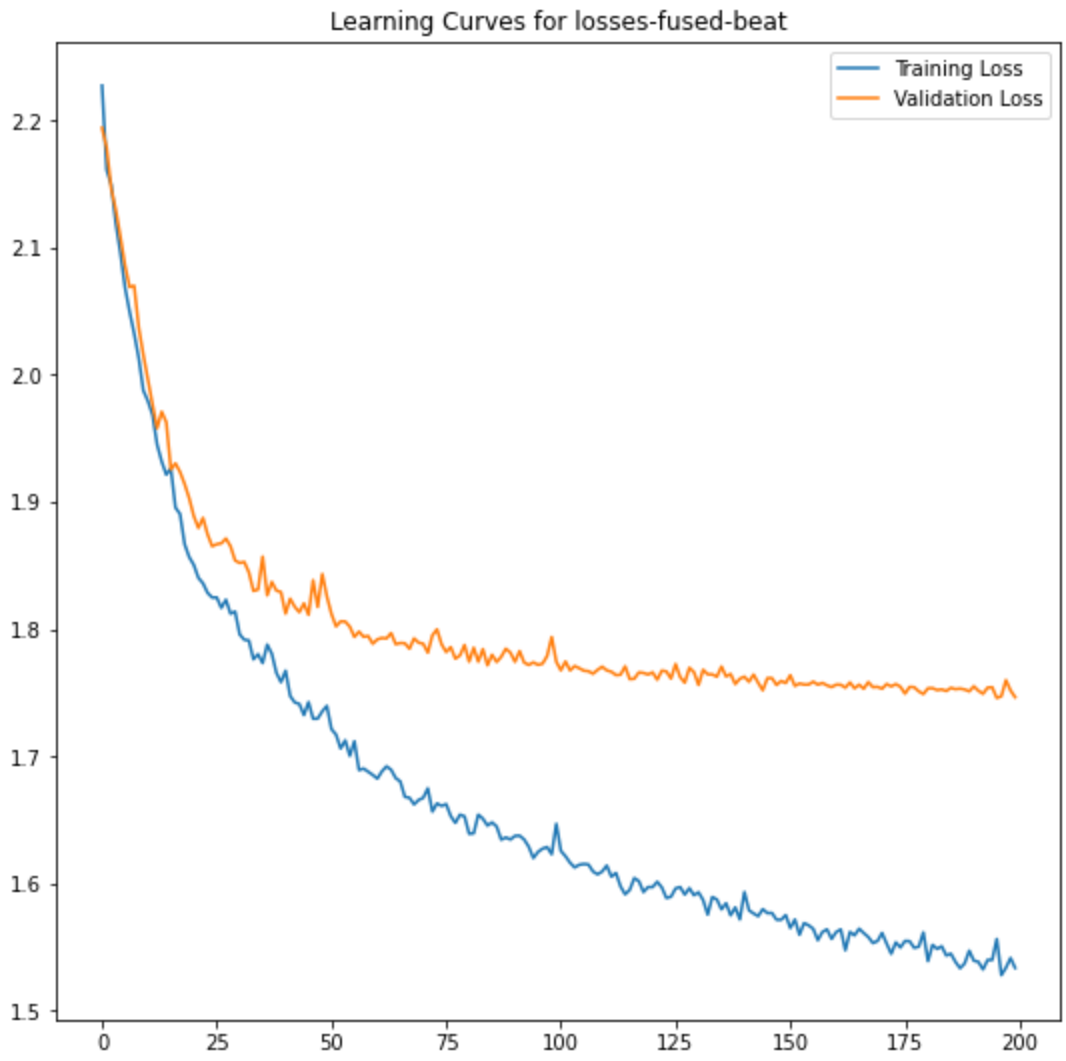
- QUESTION: Comment on how the train and validation splits are created.
  ANSWER: We read the data in arrays, create an array of the indices, we shuffle the indices, and then we split them.
- QUESTION: It's useful to set the seed when debugging but when experimenting ALWAYS set seed=None. Why?
  ANSWER: 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**: True Positives / (True Positives + False Positives).
  Precision is the ability of the classifier not to label as positive a sample that is negative.
- **Recall**: True Positives / (True Positives + False Negatives).
  Recall is the ability of the classifier to find all the positive samples.
- **F1**: True Positives / (True Positives + (False Positives + 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**: correct_predictions / 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 (False Positives + False Negatives) / 2 = 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 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

Step 7

After training a simple 2 layer convolutional network on MNIST, we view the activations on a handwritten letter “3”.

The network’s architecture is explicitly defined as follows:

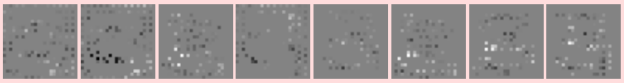
```
layer_defs = [];
layer_defs.push({type:'input', out_sx:24, out_sy:24, out_depth:1});
layer_defs.push({type:'conv', sx:5, filters:8, stride:1, pad:2, activation:'relu'});
layer_defs.push({type:'pool', sx:2, stride:2});
layer_defs.push({type:'conv', sx:5, filters:16, stride:1, pad:2, activation:'relu'});
layer_defs.push({type:'pool', sx:3, stride:3});
layer_defs.push({type:'softmax', num_classes:10});
```

In the first convolutional layer, we notice that the activations are a result of filters that detect something akin to vertical and horizontal edges (Sobel Operators).

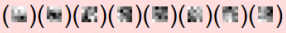
conv (24x24x8)
filter size 5x5x1, stride 1
max activation: 3.24863, min: -3.64962
max gradient: 0.0001, min: -0.00011
parameters: 8x5x5x1+8 = 208

Activations:

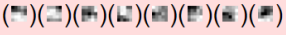
Activation Gradients:



Weights:



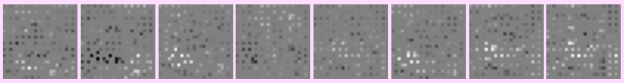
Weight Gradients:



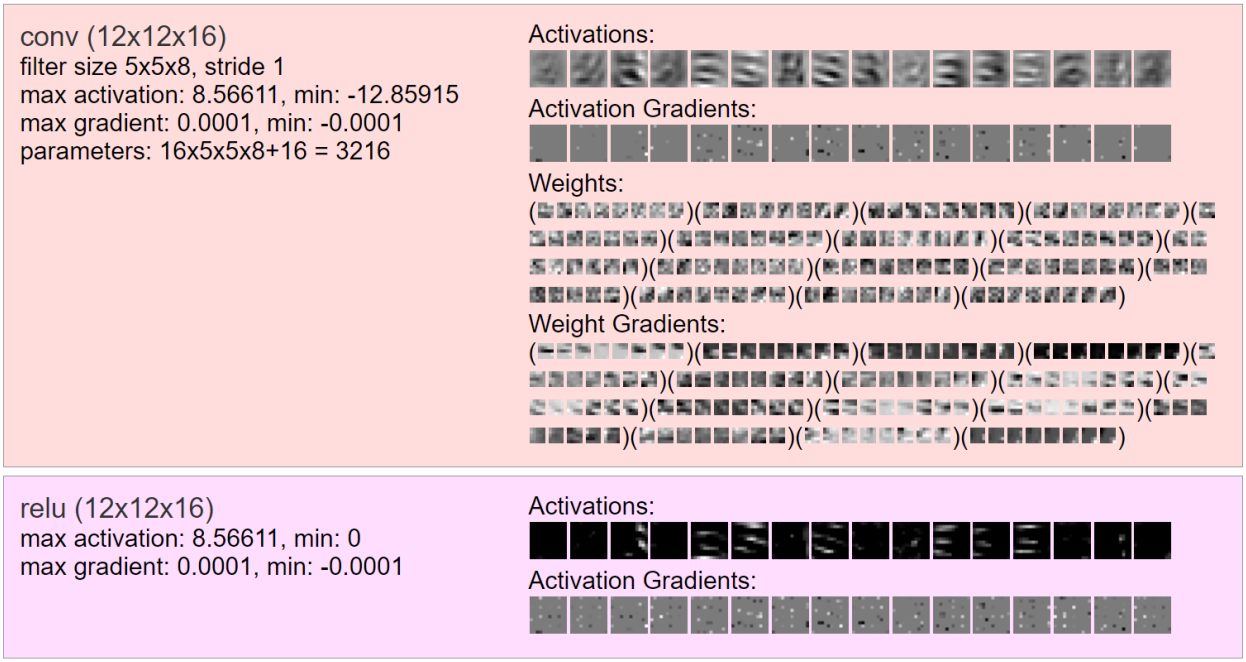
relu (24x24x8)
max activation: 3.24863, min: 0
max gradient: 0.0001, min: -0.00011

Activations:

Activation Gradients:



In the second convolutional layer we observe that more complex features are being learned. Most of the activations consist of three horizontal lines on the left side of the picture, which is a defining characteristic of “3”.



Next, we implement a convolutional neural network in the class ConvNet in main.py. Each block contains the following layers

- **2D Convolution**. A k by k by image_channels box slides over the image, computing the dot product of the box with the part of the image it covers. The weights of the box are learnable. This process is applied in parallel out_channels times (independent boxes), creating a new image of out_channels channels.
- **Batch Normalization Layer.** In this layer, when we are in training mode, we compute the mean and the variance of the nodes across the current batch. Then we normalize the node values by subtracting the mean and then dividing by the standard deviation (plus some small constant to avoid division by 0). After that, we apply an affine transformation $a_i x_i + b_i$ on each node x_i separately, where the parameters a_i, b_i are learnable. When we are in inference mode, we use a fixed estimate of the mean and variance of nodes which we have computed during the training phase.
- **RELU Layer.** In this layer nodes with value <=0 are mapped to 0, while the rest are left as is.
- **Max Pooling Layer.** On each channel separately, we slide a window (usually with stride equal to the window size, so that we have no window overlap) and we keep the maximum value of that window, resulting in a downsampling of the original image.

The results are seen below. The CNN reaches much higher accuracy than the RNN.

	precision	recall	f1-score	support
0	0.00	0.00	0.00	40
1	0.75	0.38	0.50	40
2	0.50	0.41	0.45	80
3	0.32	0.76	0.45	80
4	0.37	0.50	0.43	40
5	0.00	0.00	0.00	40
6	0.42	0.72	0.53	78
7	0.09	0.05	0.06	40
8	0.41	0.25	0.31	103
9	0.33	0.15	0.20	34
accuracy			0.38	575
macro avg	0.32	0.32	0.29	575
weighted avg	0.35	0.38	0.33	575

Step 8

Final losses (train, val)
{('cnn', 'danceability'): (0.026200693398714066, 0.03497045673429966),
 ('cnn', 'energy'): (0.03836638227105141, 0.06758967833593488),
 ('cnn', 'valence'): (0.05265886425971985, 0.06979266880080104),
 ('rnn', 'danceability'): (0.039081798270344734, 0.030843285378068686),
 ('rnn', 'energy'): (0.07389625608921051, 0.0665128231048584),
 ('rnn', 'valence'): (0.07319185778498649, 0.06853642454370856)}

Spearman Correlations
{('cnn', 'danceability'): 0.4200621872035989,
 ('cnn', 'energy'): 0.7412197341436112,
 ('cnn', 'valence'): 0.17434145197510315,
 ('rnn', 'danceability'): 0.2574231630396351,
 ('rnn', 'energy'): 0.1439822665013788,
 ('rnn', 'valence'): -0.14435019562799786}

We see that the best model, target combination is the CNN for the ‘energy’ target.

Step 9

In the paper "How transferable are features in deep neural networks?" the researchers attempt to learn task B by utilizing learning on task A.

The idea of transfer learning is that in general the initial layers learn features that are useful for many tasks (activations similar to Gabor filters), thus one can assume that those layers can be used across different tasks. Of course, it's better when the tasks are similar, as the paper shows that training on images containing only man-made entities doesn't give great results when transferring for images containing natural entities. In any case though, transferring from a dissimilar task is better than initializing with random weights!

A network of depth n=8 is trained on task A. Then after keeping the first k layers, we randomly initialize and append n-k layers. After this, we train the model either by completely freezing the first k layers, or by allowing them to be trained as well (fine tuning).

The paper discusses two distinct issues that arise when performing transfer learning.

- On the cutoff point, any previously learned co-adaptations are not helpful anymore and new co-adaptations need to be relearned. If the layer is frozen, the co-adaptations cannot be rediscovered. If the layer is not frozen (fine tuning), then new co-adaptations can be learned.
- Higher layers tend to learn features that are more task-specific, which means that they are not as transferable as lower ones.

Finally, it is demonstrated that better results are to be expected when fine tuning instead of when freezing the initial layers.

I will choose the convolutional neural network instead of the recurrent one because it had better performance. Also, because the CNN has more layers, the lower layers will probably learn features that are more transferable.

The resulting Spearman correlation, after training for 5 epochs with learning rate 0.0001 is:
SpearmanrResult(correlation=0.5409489478646622, pvalue=7.394105911178049e-10)

The result is seemingly worse than the one from step 8.

Step 10

In the paper "One Model To Learn Them All" multi-modal learning is discussed. Instead of traditional multi-task learning where each task is from the same domain (e.g. speech, vision or vision), the authors train the model on tasks of different modes (both speech, vision and text).

The main result of this paper is that multi-modal learning tends to improve performance for tasks where few data are available, while on the other hand, when the dataset is large, the results are

similar to single-task learning. This implies that seemingly unrelated tasks contain common information.

Training an LSTM on the multitask dataset gives the following results:

Valence:
SpearmanrResult(correlation=-0.06419014874773575, pvalue=0.5013305848114331)

Energy:
SpearmanrResult(correlation=0.4575604464504971, pvalue=3.935501279358577e-07)

Danceability:
SpearmanrResult(correlation=-0.04447491159482537, pvalue=0.6414814673090212)