Introduction to Signal Processing for Data Science

Lecturer: Dr. Barak Sober

Written by: Yoel Graumann

Main paper: Classification of epileptic EEG recordings using signal transforms and

convolutional neural networks by Rubén San-Segundoa, Manuel Gil-Martína, Luis Fernando D'Haro-Enríqueza, José Manuel Pardoa

Link to paper:
https://www.sciencedirect.com/science/article/abs/pii/S0010482519301398

Introduction:

Signal classification plays a crucial role in various scientific and technological domains, from medical diagnostics to multimedia processing. In the medical field, electroencephalogram signal classification has been extensively studied for applications such as epilepsy detection and brain-computer interfacing. By categorizing EEG patterns, clinicians can identify neurological conditions with high precision, enabling timely medical interventions. Over the past decades, researchers have developed numerous signal processing and machine learning approaches to improve EEG classification accuracy, with particular emphasis on wavelet-based feature extraction and deep learning models like Convolutional Neural Networks.

A growing body of research demonstrates that wavelet decomposition provides an effective way to extract time-frequency characteristics from EEG signals. Studies by Kalayci & Ozdamar, Ghosh-Dastidar et al., and San-Segundo et al. highlight how discrete wavelet transforms (DWT), nonlinear chaotic measures, and deep learning architectures enhance the ability to detect abnormal EEG activity. San-Segundo et al., explores CNN-based EEG classification while systematically evaluating the impact of signal transformations, including Fourier, wavelet, and empirical mode decomposition (EMD). Their findings indicate that CNNs can easily learn discriminative patterns from transformed EEG signals, outperforming traditional approaches.

While a lot of research on CNN-based classification of transformed signals has focused on biomedical applications, similar time-frequency processing techniques are widely used in music genre classification, speech recognition, and environmental sound analysis. Many audio classification tasks rely on spectrograms, wavelet transforms, and Fourier-based representations, which bear strong similarities to EEG preprocessing techniques. Given these commonalities, this project aims to evaluate whether an EEG-based deep learning approach can generalize to an entirely different domain: music classification.

This project will reimplement the CNN-based algorithm proposed by San-Segundo et al., originally designed for EEG seizure detection, and apply it to the GTZAN dataset, a widely used benchmark dataset for music genre classification. In short, Segundo et al. use a deep neural network with two convolutional layers for feature extraction and three fully connected layers for classification to differentiate between focal and non-focal epileptic EEG signals and detect seizures. They evaluate different signal transformations as inputs to the model: Fourier, wavelet, and Empirical Mode Decomposition.

I will investigate the adaptability of deep-learning-based EEG classifiers to audio signals and determine whether time-frequency transformations enhance genre recognition. By testing a deep learning method originally designed for EEG classification on a music dataset, this study explores the transferability of signal processing techniques across domains. If successful, this approach could suggest new ways of leveraging medical signal processing algorithms for audio classification tasks and vice versa.

Data:

In Segundo's paper, one of the datasets used by the algorithm was The Epileptic Seizure Recognition Dataset which consists of EEG recordings from both healthy individuals and epilepsy patients. It offers a structured way to analyse brain activity across different

physiological states. The dataset includes EEG recordings divided into five distinct classes, each containing 100 single-channel EEG segments of 23.6 seconds duration, sampled at 173.61 Hz. The five different classes are A – EEG from healthy subjects with their eyes open, B – EEG from healthy subjects with eyes closed, C – Interictal signals from the non-epileptogenic zone of epilepsy patients', D – Interictal EEG from epileptogenic zone of epilepsy patients, E – Seizure EEG from affected brain regions during active episodes. The second dataset was the Bern Barcelona dataset, which originates from a study aimed at analysing intracranial EEG from epilepsy patients. The aim was to localize the brain areas responsible for seizures in patients with drug resistant temporal lobe epilepsy. This dataset contains 7500 signal pairs, half focal (zones in the brain with seizure onset), the other half non-focal (areas not involved at seizure onset) all of which were collected recorded for 20 seconds at 1024Hz and down sampled to 512 Hz.

In this project, I decided to implement Segundo's pipeline on the GZTAN dataset. This dataset is one of the most widely used benchmark datasets in music genre classification. GZTAN consists of 1000 audio clips, each of which is 30 seconds long, sampled at 22050 Hz and stored in mono, 16-bit PCM format. The dataset has 10 classes, with 100 clips per class / genre: Blues, Classic, Country, Disco, Hip-hop, Jazz, Metal, Pop, Reggae & finally Rock.

At first glance, EEG signals and audio signals appear to be fundamentally different, but they share key signal processing properties that make Segundo's algorithm pretty adaptable. First, Both EGG & music are time series signals, while EEG signals represent electrical activity over time, audio signal (music) represent amplitude variations over time. Both are one dimensional time series data, which makes it possible to have similar filtering, feature extraction and classification methods. Second, EEG signals contain important frequency components, such as alpha, beta & delta etc. waves which are important for distinguishing between normal and epileptic states. Similarly, Music signals have distinct spectral patterns which correspond to the genre, with different distribution of harmonics and rhythms. Techniques employed in Segundo's algorithm such as Fourier and wavelet transform, are useful for analysing both EEG and Music signals. Thirdly, the datasets are both for classification, that is, the response variables in all of the datasets are labels and not numbers etc. Making the datasets similar in the task at hand. Finally, Both datasets have Temporal patterns and structures. The EEG seizures involve characteristic patterns in time, such as sudden spikes or oscillations. Similarly, music genres also exhibit temporal structures such as rhythmic patterns and harmonic progressions. Techniques like wavelet decomposition can be adapted to extract relevant features from both EEG and music signals. I think that applying this EEG-focused pipeline to music genre classification is an unconventional but very insightful approach that shows the flexibility of signal processing techniques across different domains. While EEG and music serve different purposes in our lives, both datasets rely on temporal and spectral analysis for classification.

Paper's Training Pipeline and Algorithm:

So first I would like to say that I did not find an implementation of the paper's algorithm and pipeline online, so I had to recreate it by myself. It taught me a lot about what's actually happening inside the code and made everything more understandable. After implementing the code, I tested it on one of the datasets in the paper. specifically, I tested it on the Epileptic Seizure Recognition dataset, and I performed all 5 experiments that Segundo et al. ran for this

paper. I reran the algorithm for Fig 4 and Fig 8 in the paper. My results can be found in in the "PROOF OF CONCEPT" section in the supplementary materials section. To be completely transparent, I have to say that sometimes my implementation fared better than the author's results. Conversely sometimes my results were a little worse than the author's. There are several possible reasons for these discrepancies. Since I did not have access to the original code, I had to recreate the algorithm from the paper's description. Small differences in implementation could have impacted performance. Additionally, advancements in deep learning frameworks and hardware likely played a role. The original paper used Keras with TensorFlow 1, whereas my implementation benefits from TensorFlow 2 optimizations, improved GPU acceleration and better training techniques. These improvements may have contributed to my model performing better in some cases. Conversely, since the authors probably had access to different computational resources, that could explain why my results were sometimes lower. Finally, differences in dataset handling and train-test splits may have introduced variability. Despite my efforts to replicate the original methodology, even minor differences in cross-validation, feature extraction, or signal transformations (Fourier, which, for example was performed using octave, Wavelet, etc.) could impact performance. In summary, while my results align closely with those in the paper, natural variations in implementation and technological advancements likely explain any discrepancies.

Segundo's work in this paper could be split into two main parts, the first part is the feature extraction step, where the pipeline takes the data, and transforms it in a smart signal-processy way. The second part is the model itself and the training. It is important to understand that the task at hand is a classification task, where the pipeline takes the data, and attempts to classify the signal. In the paper, the authors tested binary and 3-way classification. In my project, I want to use the whole GZTAN dataset, better put, I will engage in a 10-way classification, as there are 10 different genres in the dataset.

Step 1: Feature Extraction:

According to chapter 3.3 in the paper, before processing the signals (including raw data), a sixth-order Butterworth filter was applied to eliminate frequencies above 60 Hz. Imagine that you're at a party, and you want to listen to an interesting conversation about Donald Trump, but there's loud music playing in the background. A butterworth filter works like a "smart" noise filter that smoothly removes the unwanted sounds, without distorting the conversation too much. Unlike other filters that may introduce sudden volume drops or resonances, this filter ensures the transition from what we want to keep (passband) to what we want to remove (stopband) is as smooth as possible. This filter has a maximally flat passband, which means that it does not amplify or suppress frequencies before the cutoff frequency. Further, the transition from passband to stopband happens smoothly. Finally, as the order n increases, the filter cuts off unwanted frequencies more steeply. The frequency response, or the way the filter behaves across frequencies is given by: $|H(\omega)|^2 = \frac{1}{1+\left|\frac{\omega}{\omega_c}\right|^{2n}}$. $H(\omega)$ is how much the signal is allowed through at frequency ω. ω_c is the cutoff frequency where the filter starts reducing signals and n is the order of the filter, with higher n meaning a sharper cutoff. We can intuitively think of n as some kind of volume knob that controls how aggressively we cut out unwanted noise. A low order like n=1 fades out noise very gently, while a higher order filter like n=6 will sharply remove frequencies past ω_c. In the classic analog derivation, a 6th order filter has six poles that are spread around a semicircle in the complex s-plane. The poles

are given by: $s_k = \omega_c e^{j\left(\frac{\pi}{2}+\frac{(2k-1)\pi}{2n}\right)}, k = 0, 1, \ldots, 6$. Where j is the imaginary unit. And since n=6, we get six poles equally spaced between 90 and 180 degrees in the left half of the s-place. This filter was ALWAYS applied in Segundo's & our implementation. After applying the filter on the signal, the signal is either sent as is ("raw") into the model or goes through one of the following transformations: One of them is the Fourier transform, in this case, the input to the model were the magnitude values. Due to the symmetry of real signals, only the first N/2 spectral points were retained as input features. Another one was Daubechies wavelets, with 8 vanishing moments and 5 filterbank iterations. A wavelet's vanishing moments capture how well it can represent polynomials of a certain degree. In our case, we have 8 vanishing moments, this means the wavelet integrates polynomials of degree up to 7 (8-1) to zero. Having 8 vanishing moments means that the wavelet is relatively smooth and can flexibly capture subtle changes in the signal. Higher vanishing moments can give cleaner subband separation and more detail in low-frequency components, at the cost of longer wavelet filters. A wavelet's filterbank iterations refers to the successive splitting of the signal into approximate subbands. At each iteration, the approximate subband is downsampled and then split again. After 5 iterations, we have subbands covering different and progressively narrower frequency rangers. In short, the wavelets decompose the signal 5 times, then collect all wavelet coefficients to feed the CNN. Another transformation was the Empirical Mode Decomposition (EMD). Since the algorithm is provided in the paper itself, ill attempt to give a more intuitive explanation. We can think of EMD like peeling off layers of an onion, where we reveal simpler wave-like components of our signal one by one, from fastest (the more outer parts of the onion) down to the slowest trend (the more inner parts of the onion). This is done in multiple steps. (1) We start with our raw signal & call it residue, (2) then we identify all the peaks and the troughs in that residue. We can imagine drawing an "upper curve" through the peaks and a "lower curve" through the troughs. (3) Then we compute the average of these two curves, which basically captures the local trend in that portion of the signal. (4) Then we subtract that average from the residue, so we basically "peel off" the most rapidly oscillating wave component (just like with an onion). If the waveform has a stable number of peaks and troughs and doesn't drift, in other words, if it's a valid Intrinsic Mode function (IMF), we accept it as a layer. If it drifts too much, we treat it as a temporary signal, repeat (2), (3) & (4) until it stabilises. (5) once we have a "layer", we remove the IMF from the residue, leaving a new residue that lacks those faster oscillations. (6) we repeat and get the next IMF. We continue until we either reach the slowest layer or until we reach the number of IMFs we wanted. Finally, our original signal is represented as a sum of these IMFs and the residue. This algorithm is powerful because we don't force data into fixed frequency bands, like in wavelets or Fourier. Instead, EMD let's the data speak ("EMPIRICAL"), peeling away whatever wiggles are present. In the paper they decided to extract 6 IMFs which are then fed into the Model, we did the same. Finally, there's the "ALL" transform. This transform is a concatenation of multiple signal transformations: Raw, Fourier, Wavelet, and Empirical Mode Decomposition, which I described above, applied to the signal. Each transform extracts unique features, which are stacked together into a unified input representation for the CNN. This approach leverages complementary information from different domains (time, frequency, and time-frequency) to enhance classification performance. In our implementation, we decided to add another type of possible input into the CNN model, the mel spectrogram. A spectrogram is a visual representation of how the energy of our signal is distributed across different frequencies over time. We can think of it like a "musical staff" the

scrolls in time. At each moment, we can see which frequencies are "lit up" (have more energy) and which aren't. In more mathy terms, for a discrete time signal $x[n]$, the spectrogram is computed from the Short Time Fourier Transform. This transform takes overlapping frames of the signal and computes the discrete fourier transform for each frame. If we denote a window function by $w[m]$ and let X(k,n) be the DFT of the n-ths frame, then we have $X(k,n) = \sum_{m=0}^{M-1} x[nR+m]w[m]e^{-j\frac{2\pi}{M}km}$. With M being the window length, R being the amount of shift between successive frames, and k being the frequency bin index in {0, 1,..,M-1}. The spectrogram is then the squared magnitude of X(k,n), that is $|X(k,n)|^2$. The spectrogram can be viewed as a 2D matrix whose axes are time (n) and frequency (k). Each matrix element contains the energy at the time-frequency point. We can imagine slicing our signal into short windows, each capturing a small time snippet, and wthinin each snippet we measure how much of each frequency band is present. Putting these slices together forms this representation. A mel spectrogram goes a bit further, as it warps the frequency axis onto a mel scale, to better match how we humans perceive frequency differences. Once we calculate $|X(k,n)|^2$ we pass it through a set of mel-spaced triangular filters. Each filter covers a frequency subband, with increasing center frequencies that follow the mel scale. If we define b to be the index of the filters {1,2, 3,....,B}. The mel-filtered energy in the b-th band at time frame n is $E(b,n) = \sum_{k=0}^{K-1}|X(k,n)|^2 H_b(k)$. Where each filter $H_b(k)$ is defined by $H_b(k) =$ 0 if $k < f_{b-1}$ or $k > f_{b+1}$, $(k-f_{b-1})/(f_b - f_{b-1})$ if $f_{b-1} \le k \le f_b$, $(f_{b+1}-k)/(f_{b+1}-f_b)$ if $f_b \le k \le f_{b+1}$. Where f_b is the center frequency of filter by, and f_{b-1} is the lower boundary of that filter and f_{b+1} is the upper boundary. The mel scale is defined by $\text{mel}(f) = 2595\log_{10}\left(1+\frac{f}{700}\right)$ where f is in Hz. Finally, we take a log compression of $E(b,n)$. So the mel spectrogram is MelSpectogram(b,n) = log($E(b,n)$+ϵ) with epsilon being a small constant to help us avoid taking the log of zero. In a more intuitive sense, we can Imagine we have a piano keyboard stretched from very low bass to very high notes. A regular spectrogram effectively gives equal width for every key, from the lowest to higher note. But our ears can more easily distinguish small shifts in pitch at the low end rather at the high end. A mel spectrogram is like warping that piano keyboard so that the bass notes are spaced out more finely, so we pay closer attention to them, while the high keys get compressed closer together, so we treat them with less resolution. Each triangular mel filter is like a "bucket" collecting energy in a specific region of the keyboard; the lower-frequency buckets are narrower, and the higher-frequency buckets are wider. After collecting all those buckets, we take a log, which makes small differences in energy more noticeable while preventing huge energy values from overwhelming the scale. In our implementation, we used 128 mel filters, covering the frequency range from 0 Hz to half the sampling rate of the recording (11025Hz). After computing the mel spectrogram, each log-magnitude coefficient is normalized to fit a fixed intensity scale of 0-255, the final product is a 2D representation in the size of 28x28. These tweaks are done to make it easier for the CNN to train. To reiterate, our addition of the mel spectrogram is to get it with the algorithm described above and then feed it as a picture to the cnn. Since CNNs were originally made for pictures, it is only natural that we would try to get a picture out of our signal and then feed it into the cnn. We expected to get decent results as CNNs have been shown to detect informative features and good for image classification, empirically.

Step 2: Model & Training:

It is important for us to reiterate that we did not find any available code online and had to write the whole code for ourselves. At first, we created a Proof-of-Concept code, where we tested the papers' findings on the Seizure Recognition dataset. Later, we moved on to the GZTAN dataset and ran our updated code & experiments. You can find the GZTAN runs in the coding section when searching "GZTAN PROJECT". Before explaining the model, I will give a quick explanation about some of the components that appear in the paper's model. Max Pooling layer: Usually put after a convolution layer. A convolution layer produces a set of feature maps, and a max pooling layer reduces the resolution of those maps. It does so by basically sliding a small window across each feature map, taking the maximum value in each window. This process makes the feature maps smaller and helps the cnn become somewhat invariant to small shifts in the input. If X is a picture of size H*W and we have a maxpool window of size r * s that moves in strides across X, then each pooling operation can be given by: $\text{Maxpool}(X_{(u,v)}) = \max\{X_{(i,j)} \mid i \in [u, u+r-1], j \in [v, v+s-1]\}$ Where (u, v) is the index of the topleft corner of the pooling window, we get one output per window. Dropout layer: When training a model, a dropout layer randomly "drops out" and sets to 0 a specified fraction of the neurons' outputs. This prevents the neural network from relying too heavily on a particular neuron for the task and forces it to learn more robust representations of the data. In a more mathy way, if we let $h = (h_1, h_2, \dots, h_n)$ be the vector of neuron activations in the layer, then a dropout creates a mask vector $m = (m_1, m_2, \dots, m_n)$, where each m_i is sampled from $m_i \sim$ 1- Ber(p), in our case p=0.2 . by Hadamard product the final output is just $(h_1 * m_1, h_2 m_2, \dots, h_n m_n)$. With proportion p dropped out. At training time, a fraction becomes zero each step. And at test time, it does nothing, per Keras' documentation. Finally, there's Softmax activation: When performing a classification task, our neural network has to assign a probability to each of the several classes, we use a softmax layer at the output to do that. A softmax turns a vector into a probability distribution over C classes, such that all probabilities sum to 1 and they all are between 0 and 1. This let's the networks final output be interpreted as "the probability that the input belongs to class C". The mathematics here are very simple, assume we have a vector $Z = (z_1, z_2, \dots, z_C)$, for each class i in {1,2,...C}, the softmax output is $p_i = \frac{e^{z_i}}{\sum_{j=1}^{C} e^{z_j}}$, and since exponentials are always positive and each pi is in the range (0,1) and because of the normalization in the denominator by the sum of exponentials the probabilities sum up to 1 so we get a probability distribution. Now, to the model:

The model structure in the paper is pretty simple, in high level, it is a model made out of 2 convolutional layers, the first is followed by a maxpooling layer and a dropout layer. The second cnn layer is followed only by a dropout. And then everything is flattened and fed into 2 shrinking in size fully connected layers, each of which is followed by a dropout layer. Finally, we have another fully connected layer which is used for the classification task (with softmax). Like the authors, we implemented the model in a way such that, if the cnn finds that it is a binary classification task, it will use the sigmoid activation, and if it is a multiclass it will use the softmax, in our case we will be using softmax since were performing a 10-label classification task. One main difference between the original model and our model, is that now instead of having the two convolutional layers be 1x5, we have them both be 5x5. Our cnn's kernels now considers a 5x5 receptive field for each convolution operation and captures both horizontal and vertical features in a larger region. The reason for this "upgrade" was because we wanted our model to also be able to take in 2D spectrogram pictures. "How do

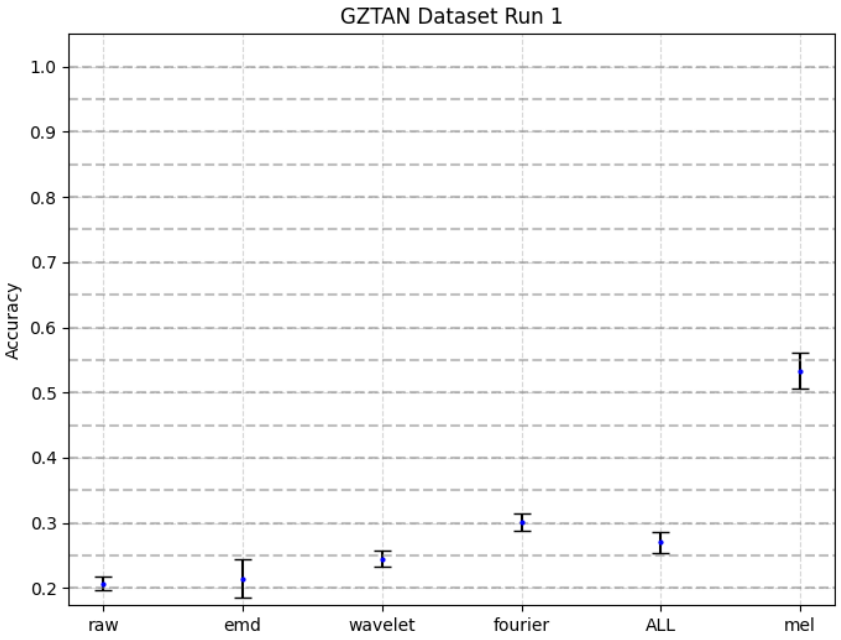
we handle the 1D signals?" you might ask. Well, before feeding the data in our implementation, we convert the 1d signals into a 2d format which means we either reshape or produce multi-row matrices. The 5x5 filter, despite being typically used for 2D images, can operate effectively on the newly restructured 1D data represented in 2D. The filter moves across both time transformation features, learning the spatial – temporal patterns. This works because the 1D signals in our GZTAN implementation are treated as narrow 2D images. The height dimension allows different representations (think Fourier coefficients, IMFs from the EMD, etc.). The width captures the sequential time domain properties. In short, we play with our 1D data, in a strategic and insightful way and turn it into a coherent 2D matrix before applying the convolutions.

For training, we used all of the hyperparameters they used, such as learning rate, optimizer etc. One thing they did not exactly tell us is the maximum number of epochs. The number of epochs were found by using a 5-fold cross validation scheme. For this cross validation scheme, a maximum number of epochs must be selected so that the algorithm knows when to stop training. Since the number of maximal epochs to try was not given in the paper, we chose to set the max number of epochs to be 50, since we were running everything locally and it takes a very long time to run the code with more. The 5-fold cross validation employed had several steps, every run / fold: we split the data into 5 parts, 3 parts are used for training, 1 part is used for validation & the remaining one for testing. The partitioning is done in such a way that across different runs, every single signal will appear in the test set exactly once. Next, we focus on the first configuration of folds and use 3 parts for training and one for validation. During training, at each epoch we track the accuracy on the validation set. This allows us to find the optimal number of epochs. It is the number of epochs where I get the highest accuracy on the validation set. After finding the best epoch, I retrain the model using the optimal number of epochs, but this time using both the three training folds and the validation fold together as a single training set, I test the accuracy on the test set. I do that 3 times, meaning, that I train the model using the 3 training and 1 validation sets using them as a single train set using optimal epochs I found. This is to reduce any influence from the random weights initialization of the neural network. For each of these three runs I calculate the accuracies and then with these accuracies I calculate the average accuracy for the fold. I do that for all 5 folds, keeping track of all predictions for the optimised models tested on the test sets. Additionally, I calculate the standard deviation over the 5 folds. In summary these are the basic steps: ***(1)*** *split the transformed data into 5 folds* ***(2)*** *for each fold, use 3 parts of it for training 1 for validation 1 for testing* ***(3)*** *train for up to 50 epochs, tracking validation accuracy each epoch, pick best epoch* ***(4)*** *retrain from scratch using the optimal epoch, merging the training and validation as a single training set, evaluating on the test fold* ***(5)*** *repeat the entire procedure three times per fold, then move to the next fold (five total folds)*. This is exactly what they do in the paper, and I implemented it. Additionally, with the tracked predictions, I create a confusion matrix, which aggregates all the predictions from the epoch-optimised models over the 5 folds / runs. So at the end of a 5fold CV, I print the confusion matrix which is normalized, where each row in the matrix represents a class's actual instances and the values in that row sum up to 1. This means that each entry represents the proportion of samples from the true class that were predicted as a given class. There are more metrics I calculate at the end of the 5-fold CV such as Macro-Avg Precision, Recall and F1. I also added a specific functionality where I can choose the random seed of the fold indexes. I have done that so I could run the whole process multiple times and ensure that I never get the

exact same fold splits every experiment. This is because in my project, I run each experiment, on all signal transforms, 10 times. The idea of running multiple experiments 10 times (In the paper they only run 1 experiment!) is to obtain a more robust statistical estimate of model performance and reduce the influence of chance in any single run. Specifically, by repeating the entire training–validation–testing procedure multiple times, each time with a different seed for the fold partitions, we can compute statistical measures such as the mean accuracy, the standard deviation, etc. This reveals how sensitive the results are to the random factors involved in training and in how data folds are split.

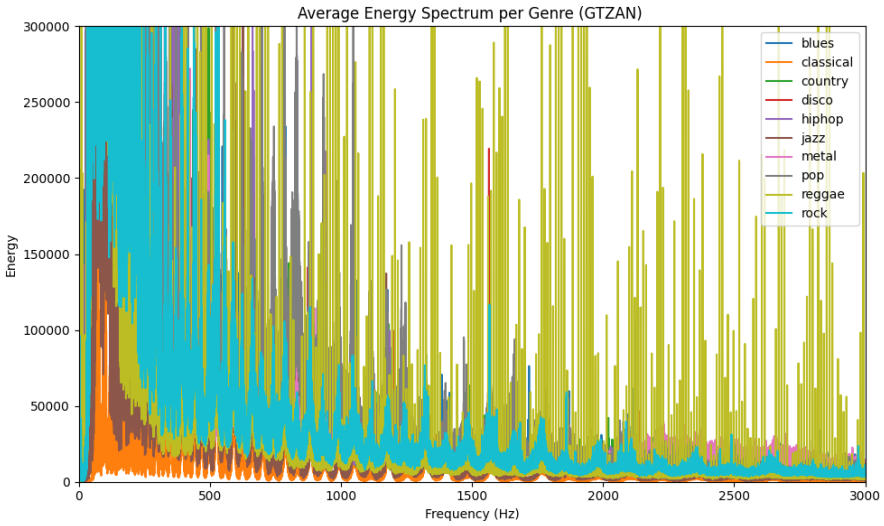
Analysis:

For the analysis, and in an attempt to prove how the model works, statistically, I ran the above experiments 10 separate time with 10 different seeds (so I get different fold allocations every time, guaranteed). Each experiment is run for all the transforms I described above. I will start by showing results for a single experiment, but the analysis is correct for all of them, since we got robust results.



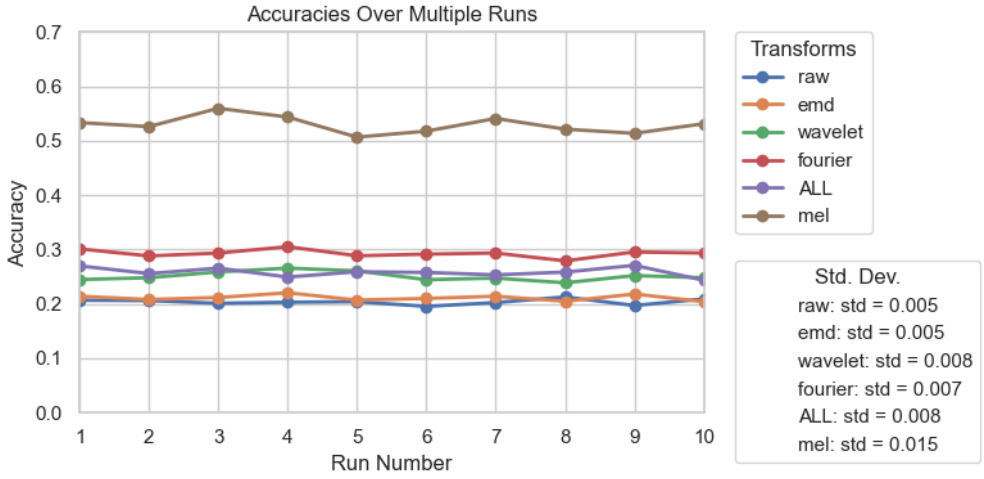
Just like in Fig 8 presented in the paper, I show the accuracies for all classes along with the standard deviations (whiskers) for each of the transforms, for a single experiment (Please see "run 1" in the code section):

In the paper's EEG experiments, the classification problems were often binary (focal vs. non-focal) or at most a few classes. So it is obvious that the same model with more classes would have a harder time getting these results of above 90% accuracy etc. What we do get is that the mel spectrogram transformation achieved the best results in accuracy, which is true for all 10 runs of the experiment. Fourier Transform Takes the 2nd place, also for all experiments. Even so, the difference in accuracies between fourier and mel is almost 2 fold. For all transformations we get better results than the "random-guess" baseline of 0.1. When inspecting the confusion matrices, which are all normalized as described in the previous chapter, a clear phenomenon is apparent; all transformations have a relatively easy time predicting classical music. More clearly, all models have a recall of atleast 0.5 for classical music. Since this is a 10-class problem, a pure random guess would yield an average of 0.1 accuracy for any given class. Thus, achieving 0.5 is five times higher than random guessing and indicates that the model is indeed finding meaningful patterns associated with classical music. This suggests that classical music has features, that are somehow extracted with any of the transformations, that are easier for the model to learn relative to other classes. Note that since we got very robust results, everything I wrote in this page is also true for all the remaining 9 experiment runs. You can see the code section for all the confusion matrices, accuracies etc for all of the



experiments. Next, I wanted to see why classical had better recall than the other genres. I graphed the average energy vs the frequency per genre for the whole dataset:

We can see that classical's average energy curve remains smoother and less spiky than, for example, metal or rock. This broad, evenly distributed range hints at reduced percussive or distorted patterns, making classical easier to isolate. The transformations consistently catch these subtle features, yielding higher recall for classical compared to genres with intense peaks or repetitive rhythms. Now I will attempt to show the consistency of my findings over the 10 different experiments. The following graph shows the final accuracies for every experiment / run, along with the standard deviations. We can see that mel and fourier are consistently best and second best respectively. We can see that wavelet and 'all' are about the same, and emd and raw are also about the same. The low standard-deviation values of raw and emd show that, although their accuracies are somewhat lower, their performance is especially consistent from run to run. Mel scores the best mean accuracy but also shows the highest variability (0.015), although it is pretty low. Overall, the relatively small standard deviations in these transforms indicate that the observed findings are indeed consistent across different runs, which means that the accuracies are not due to "luck" from splitting the data in a "lucky" way. Now that I've shown that the results are consistent over runs, I would like to prove, statistically, that some transforms are different than others. For that I would like to use the Friedman test. Remember that for every single experiment, we have a 5-fold cross validation, for each of these folds we calculate the accuracy on the particular fold, and then average them to get the final accuracies. In other words, we can see each fold accuracy as a sub-experiment result. That means that we have a total of 50 sub-experiment accuracies that could be used for the Friedman test. In short, the Friedman test is a nonparametric, rank-based method for comparing multiple techniques across repeated measures. Each Block ranks the techniques from best to worst, and the test checks if at least one technique is significantly different. We choose an $\alpha = 0.05$. After choosing the alpha, we ran the test and got the following results: pval of 3.821129896764025e-47 and statistic=227.4198. This means that there is definitely a significant difference among the transforms when viewed across all folds. The next step is to determine which of the



transforms differ significantly from one another. I will do that with a post-hoc test. I am doing a post-hoc test because the Friedman test only tells us that atleast one method is different, but it does not tell which specific pairs of transforms differ. I will be using the Nemenyi test, as it is traditionally the default for Friedman post-hoc. It controls for multiple comparisons and is often used in machine learning model evaluation or experimental studies with repeated measures.

Continuing with our $\alpha = 0.05$ we perform the Nemenyi test and get these pairwise p values:

	Raw	EMD	Wavelet	Fourier	ALL	Mel
Raw	1	0.9	0.001	0.001	0.001	0.001
EMD	0.9	1	0.001	0.001	0.001	0.001
Wavelet	0.001	0.001	1	0.001	0.7526	0.001
Fourier	0.001	0.001	0.001	1	0.04174	0.02801159
ALL	0.001	0.001	0.7526	0.04174	1	0.001
Mel	0.001	0.001	0.001	0.02801	0.001	1

First of all, we can say that we are 95% confident that the mel transform's results are different than all the others. This confirms, statistically, what we've seen in the previous plot, with the mel spectrogram being the best. Second of all, we can say the same for the fourier transform as all the pairwise comparisons are statistically significant, this is actually quite interesting because fourier seemed to be close to the "all" transformation but turns out it is statistically different from all other transformations. We also see that all and wavelet are not statistically different, this strengthens what we see in the previous graph, but now we have statistical proof. The same goes for Raw and EMD, we can see that their pairwise pvalue is 0.9, which means that its statistically insignificant. In short, from this post-hoc analysis we are getting statistical proof at the 95% confidence level of the differences between transforms that we've intuitively seen in the previous "accuracies over multiple runs" graph.

In conclusion, this analysis confirms the robustness of the findings across all experiments, with the mel spectrogram consistently achieving the highest accuracy, followed by the Fourier transform. Both were statistically superior to other transformations, with mel spectrogram nearly doubling Fourier's accuracy. Confusion matrix analysis showed that classical music was the easiest to classify, likely due to its smooth energy distribution. Statistical validation using the Friedman and Nemenyi tests confirmed significant differences among transformations, reinforcing that mel spectrogram and Fourier outperform the others.

Overall, the results demonstrate that the mel spectrogram is the most effective transformation for this classification task, with findings holding consistently across multiple experimental runs.