

EEG Hand Movement Classification

Shanti Stewart

Information Processing Group

Oregon State University

June 20, 2020

I: Data Set:

1. Data Set Notation:

Several notations about the data set should be defined at this point, which will be used throughout the remainder of this document. Let P denote the total number of examples in the data set, and N denote the number of electrode channels. A lower case p in the superscript is used to index into the examples, and a lower case n in the subscript is used to index into the channels. In addition, a lower case t and lower case f inside brackets are generally used to index into the time and frequency domains, respectively.

II: Feature Calculation:

Three different (but related) feature calculation algorithms were developed and implemented: *frequency bin-average PSD*, *PCA on PSD*, and *PSD spectrograms*. All three algorithms require the estimation of *power spectral density (PSD)*. The power spectral density estimation (Algorithm 0) and Algorithms 1 and 3 operate on each example in the data set and each channel independently -- therefore, all computations in these sections are implied to be performed on all examples and all channels. Algorithm 2 performs computations across examples and channels, which is clearly shown in the notation.

1. Power Spectral Density Estimation (Algorithm 0):

Power spectral density was estimated by computing the Fourier transform of the autocorrelation function. Let $x[t]$ be the raw EEG signal as a function of sample t , M be the number of samples, $R_{xx}[k]$ be the autocorrelation function as a function of sample lag k , and $S[f]$ be the power spectral density as a function of frequency f . Power spectral density was estimated by the following:

Input: $x[t]$ = raw EEG signal

Output: $S[f]$ = power spectral density

Algorithm:

1.1. Estimate autocorrelation function (positive part):

$$\hat{R}_{xx}[k] = \frac{1}{M} \sum_{t=k}^M x[t]x[t-k], \quad \text{for } 0 \leq k \leq M$$

1.2. Calculate (unnormalized) Fourier transform of autocorrelation function:

$$S[f] = \text{HFFT}\{\hat{R}_{xx}[k]\}, \quad \text{for } f \in \{0, \dots, F_s/2\}$$

where $\text{HFFT}\{\cdot\}$ denotes the *Hermitian Fast Fourier Transform*, and F_s is the sampling frequency of the signal.

1.3. Normalize Fourier transform:

$$S[f] := \frac{1}{M} S[f], \quad \text{for } f \in \{0, \dots, F_s/2\}$$

In step 1.2, the Hermitian FFT was used because the autocorrelation function (both positive and negative parts) is a real symmetric function, which is also a Hermitian function by definition. The Hermitian FFT was implemented by a function in NumPy's FFT library: this function takes in the positive half (by default) of a Hermitian signal as its input and produces a real-valued frequency spectrum as its output. From Fourier analysis theory, it is known that the Fourier transform of a Hermitian signal is real-valued – this property facilitates our method of power spectral density estimation. After this NumPy function is called, the nonnegative part of the frequency spectrum is extracted (i.e. 0 Hz – 125 Hz) and kept.

2. Algorithm 1: Frequency Bin-Average PSD

This algorithm is the simplest and most straightforward idea of generating features from a frequency spectrum: divide up the spectrum into discrete “bins” and compute the average PSD values in each bin. Let num_bins be the number of frequency bins, $bins[i, 0]$ be the lower frequency of bin i , $bins[i, 1]$ be the upper frequency of bin i , and $S_{avg}[i]$ be the average PSD value in bin i .

Inputs: $S[f]$ = power spectral density, $bins$ = frequency bins for which to average over

Output: $S_{avg}[i]$ = frequency bin-average PSD values

Algorithm:

2.1. Calculate average PSD values in selected frequency bins:

$$S_{avg}[i] = \text{mean}(S[bins[i, 0] : bins[i, 1]]), \quad \text{for } i \in \{1, \dots, num_bins\}$$

3. Algorithm 2: PCA on PSD

This algorithm is really an extensions of algorithm 1 – it applies principal component analysis to frequency bin-average PSD values. Three slightly different variations of this algorithm were implemented, differing by the statistical matrix computed, as is explained in step 3.2. Let $S_n^{(p)}[f]$ be the PSD value of example p , channel n , frequency f ; L be the number of selected frequencies; and $\{f_1, \dots, f_L\}$ be the selected frequencies.

Before this algorithm is run, Algorithm 1 is performed with the frequency bins delineated by the selected frequencies $\{0, f_1, \dots, f_L\}$: $bins[i, 0] = f_i$ and $bins[i, 1] = f_{i+1}$, for $i \in \{0, \dots, L - 1\}$. The frequency bin-average PSD values $S_n^{(p)}[f]$ are assumed to be already computed. Moreover, $S_n^{(p)}[f]$ for $f \in \{f_1, \dots, f_L\}$ is treated as discretized power spectral density; in other words, $S_n^{(p)}[f]$ is treated as the PSD value at frequency f , instead of the average PSD value calculated in frequency bin f .

Inputs: $S_n^{(p)}[f]$ = (discretized) power spectral density, Q = number of principal components to keep, ε = small positive parameter to prevent $\ln(0)$

Output: $W_n^{(p)}[k]$ = PCA projected PSD values

Algorithm:

3.1. Log-normalize PSD values across examples:

$$S_n^{(p)}[f] := \ln \left(S_n^{(p)}[f] + \varepsilon \right) - \ln \left(\frac{1}{P} \sum_{p=1}^P S_n^{(p)}[f] + \varepsilon \right),$$

for $p \in \{1, \dots, P\}$ and $n \in \{1, \dots, N\}$ and $f \in \{f_1, \dots, f_L\}$

3.2. Compute channel-specific statistical matrices across examples – three variations:

3.2.1. Variation 1 – Channel-specific autocorrelation matrices:

$$K_n[i, j] = \frac{1}{P} \sum_{p=1}^P S_n^{(p)}[f_i] * S_n^{(p)}[f_j], \quad \text{for } n \in \{1, \dots, N\} \text{ and } i, j \in \{1, \dots, L\}$$

3.2.2. Variation 2 – Channel-specific autocovariance matrices:

$$K_n[i, j] = \frac{1}{P} \sum_{p=1}^P \left(S_n^{(p)}[f_i] - \overline{S_n[f_i]} \right) * \left(S_n^{(p)}[f_j] - \overline{S_n[f_j]} \right),$$

for $n \in \{1, \dots, N\}$ and $i, j \in \{1, \dots, L\}$

3.2.3. Variation 3 – Channel-specific Pearson correlation coefficient autocovariance matrices:

$$K_n[i, j] = \frac{1}{\sigma_i * \sigma_j} \sum_{p=1}^P \left(S_n^{(p)}[f_i] - \overline{S_n[f_i]} \right) * \left(S_n^{(p)}[f_j] - \overline{S_n[f_j]} \right),$$

for $n \in \{1, \dots, N\}$ and $i, j \in \{1, \dots, L\}$

where the overbar denotes a mean across examples: $\overline{S_n[f_i]} = \frac{1}{P} \sum_{p=1}^P S_n^{(p)}[f_i]$; and the

sigma denotes an unnormalized standard deviation across examples: $\sigma_i =$

$$\sqrt{\sum_{p=1}^P (S_n^{(p)}[f_i] - \overline{S_n[f_i]})^2}.$$

3.3. Compute the eigendecomposition of each statistical matrix:

$v_{k,n}$ = k th eigenvector of K_n , *for $n \in \{1, \dots, N\}$ and $k \in \{1, \dots, L\}$*

$\lambda_{k,n}$ = k th eigenvalue of K_n , *for $n \in \{1, \dots, N\}$ and $k \in \{1, \dots, L\}$*

3.4. Sort eigenvectors of each statistical matrix in descending order of (the magnitude of) their associated eigenvalues:

$$\{v_{1,n}, \dots, v_{L,n}\} := \text{sort}(v_{1,n}, \dots, v_{L,n}), \quad \text{for } n \in \{1, \dots, N\}$$

3.5. Project (log-normalized) PSD values onto eigenvectors of statistical matrices (principal components):

$$W_n^{(p)}[k] = \langle S_n^{(p)}, v_{k,n} \rangle = \sum_{i=1}^L S_n^{(p)}[f_i] * v_{k,n}[i],$$

for $p \in \{1, \dots, P\}$ and $n \in \{1, \dots, N\}$ and $k \in \{1, \dots, Q\}$

4. Algorithm 3: PSD Spectrograms

This algorithm is not fundamentally different from the others – it is simply a way to transform 1-dimensional feature vectors into 2-dimensional feature images, which are required by convolutional neural networks. The idea is straightforward: generate power spectral density spectrogram images by a sliding time-window method.

Let M be the number of samples of the raw signal $x[t]$, $\tilde{x}[w, t]$ be the windowed raw signal as a function of window w and sample t , $S[w, f]$ be the power spectral density as a function of window w and frequency f , $\{0, f_1, \dots, f_L\}$ be the selected frequency bin delineators (L is the number of frequency bins), and $Z[w, \tilde{f}]$ be the PSD spectrogram image as a function of window w and frequency bin \tilde{f} .

Inputs: $x[t]$ = raw signal, δ = sliding window length (in samples), γ = sliding window stride length (in samples), $\{0, f_1, \dots, f_L\}$ = selected frequency bin delineators

Outputs: $Z[w, \tilde{f}]$ = PSD spectrogram image (size: (W, L))

Algorithm:

4.1. Use a sliding time window to segment signal:

$$\text{Number of valid windows: } W = \text{floor} \left[\frac{M - \delta}{\gamma} + 1 \right]$$

for $w = 1$ to W :

$$\tilde{x}[w, :] = x[w\gamma : w\gamma + \delta]$$

4.2. Estimate power spectral density using Algorithm 0:

$$S[w, :] = \text{estimatePSD}(\tilde{x}[w, :]), \quad \text{for } w \in \{1, \dots, W\}$$

4.3. Calculate frequency bin-average PSD values using Algorithm 1:

$$\begin{aligned} \text{bins}[i, 0] &= f_i \text{ and } \text{bins}[i, 1] = f_{i+1}, & \text{for } i \in \{0, \dots, L-1\} \\ Z[w, :] &= \text{averagePSD}(S[w, :], \text{bins}), & \text{for } w \in \{1, \dots, W\} \end{aligned}$$

In the above steps, NumPy/MATLAB notation is used to denote array indexing and slicing. In steps 4.2 and 4.3, the shorthand notations $\text{estimatePSD}(\cdot)$ and $\text{averagePSD}(\cdot)$ are used to represent Algorithms 0 and 1, respectively. Something to note is that each spectrogram image is actually 3-dimensional (a “volume”), because each example contains N channels. Thus, each spectrogram image is of size (W, L, N) – assuming a channels-last dimension order – and can be viewed as N “stacks” of spectrogram images. In addition, PCA can be applied in step 4.3 as a variant (i.e. substitute Algorithm 2 for Algorithm 1).

III: Classification:

For classifying left-hand vs. right-hand movements, a convolutional neural network (CNN) was implemented using the TensorFlow Keras API. The CNN was trained on the spectrogram features generated by Algorithm 3 (of the previous section).

1. Data Set Augmentation and Shuffling:

Due to a shortage of training data, the data set was augmented by a sliding time-window method (identical in implementation to that of Algorithm 3), in order to generate more training/test examples. Let $x_n^{(p)}[t]$ be the raw EEG signal of example p , channel n , as a function of sample t , and P be the number of examples in the original data set; M be the number of samples of each raw signal; $\tilde{x}_n^{(\tilde{p})}[t]$ be the raw EEG signal after augmentation of new example \tilde{p} , channel n , as a function of sample t , and \tilde{P} be the number of examples in the augmented data set. The sliding time-window augmentation and shuffling process is outlined by the following:

Inputs: $x_n^{(p)}[t]$ = raw data, δ = sliding window length (in samples), γ = sliding window stride length (in samples)

Outputs: $\tilde{x}_n^{(\tilde{p})}[t]$ = augmented and shuffled raw data

Algorithm:

1.1. Shuffle data across examples (p -index axis):

$$\{x_n^{(1)}[t], \dots, x_n^{(P)}[t]\} := \text{shuffle}(x_n^{(1)}[t], \dots, x_n^{(P)}[t]),$$

$$\text{for } n \in \{1, \dots, N\} \text{ and } t \in \{1, \dots, M\}$$

1.2. Use a sliding time window to augment data set:

$$\text{Number of valid windows: } W = \text{floor}\left[\frac{M-\delta}{\gamma} + 1\right]$$

for $w = 1$ to W :

$$\tilde{x}_n^{(p)}[w, :] = x_n^{(p)}[w\gamma : w\gamma + \delta], \text{ for } p \in \{1, \dots, P\} \text{ and } n \in \{1, \dots, N\}$$

1.3. Combine the p -index and w -index axes into a single \tilde{p} -index axis:

$$\tilde{x}_n^{(\cdot)}[t] = \text{combineAxis}(x_n^{(\cdot)}[:, t]), \text{ for } n \in \{1, \dots, N\} \text{ and } t \in \{1, \dots, \delta\}$$

In the above steps, NumPy/MATLAB notation is used to denote array indexing and slicing. In step 1.3, the shorthand notation $\text{combineAxis}(\cdot)$ is used to denote the combining of multiple axes (which can be implemented with NumPy's $\text{reshape}()$ function). Something that should be noted in order to avoid potential confusion is that the sliding window and stride sizes used in this algorithm are different from the ones used in Algorithm 3 – the same Greek letters were used for simplicity.

The reason that the data is shuffled before – instead of after – the sliding time-window augmentation is that adjacent windowed data “segments” (i.e. new examples \tilde{p}) can be very similar, especially if the stride length is small relative to the window length. Having adjacent windowed data segments be very similar and then shuffling them would effectively lead to multiple “copies” of the same example (i.e. almost identical examples) randomly distributed throughout the data set, which would not allow us to split the data set into training/validation/test sets in a valid manner.

2. Feature Generation and Data Set Division:

Spectrogram features were computed from the augmented data set (from the previous section). Let $\{0, f_1, \dots, f_L\}$ be the selected frequency bin delineators (L is the number of frequency bins); $Z_n^{(\tilde{p})}[w, \tilde{f}]$ be the PSD spectrogram feature of example \tilde{p} , channel n , as a function of window w and frequency bin \tilde{f} ; and P_{train} , P_{val} , and P_{test} be the number of examples in the training, validation, and test sets, respectively.

Inputs: $\tilde{x}_n^{(\tilde{p})}[t]$ = augmented and shuffled raw data, δ = sliding window length (in samples), γ = sliding window stride length (in samples), $\{0, f_1, \dots, f_L\}$ = selected frequency bin delineators

Outputs: $Z_{train}_n^{(\tilde{p})}[w, \tilde{f}]$ = standardized training set features, $Z_{val}_n^{(\tilde{p})}[w, \tilde{f}]$ = standardized validation set features, $Z_{test}_n^{(\tilde{p})}[w, \tilde{f}]$ = standardized test set features

Algorithm:

2.1. Generate spectrogram features using Algorithm 3:

$$Z_n^{(\tilde{p})}[w, \tilde{f}] = \text{createSpectrograms}(\tilde{x}_n^{(\tilde{p})}[t], \delta, \gamma, \{0, f_1, \dots, f_L\}),$$

$$\text{for } \tilde{p} \in \{1, \dots, P\} \text{ and } n \in \{1, \dots, N\}$$

2.2. Split data set into training, validation, and test sets:

$$Z_{train}_n^{(\tilde{p})}[w, \tilde{f}], Z_{val}_n^{(\tilde{p})}[w, \tilde{f}], Z_{test}_n^{(\tilde{p})}[w, \tilde{f}] = \text{splitData}(Z_n^{(\tilde{p})}[w, \tilde{f}])$$

2.3. Shuffle each set independently, across examples (\tilde{p} -index axis):

$$\{Z_{train}_n^{(1)}[w, \tilde{f}], \dots, Z_{train}_n^{(P_{train})}[w, \tilde{f}]\} := \text{shuffle}(\{Z_{train}_n^{(1)}[w, \tilde{f}], \dots, Z_{train}_n^{(P_{train})}[w, \tilde{f}]\}),$$

$$\{Z_{val}_n^{(1)}[w, \tilde{f}], \dots, Z_{val}_n^{(P_{val})}[w, \tilde{f}]\} := \text{shuffle}(\{Z_{val}_n^{(1)}[w, \tilde{f}], \dots, Z_{val}_n^{(P_{val})}[w, \tilde{f}]\}),$$

$$\{Z_{test}_n^{(1)}[w, \tilde{f}], \dots, Z_{test}_n^{(P_{test})}[w, \tilde{f}]\} := \text{shuffle}(\{Z_{test}_n^{(1)}[w, \tilde{f}], \dots, Z_{test}_n^{(P_{test})}[w, \tilde{f}]\}),$$

$$\text{for } n \in \{1, \dots, N\} \text{ and } w \in \{1, \dots, W\} \text{ and } \tilde{f} \in \{1, \dots, L\}$$

2.4. Standardize data, with respect to training set:

$$Z_{train}_n^{(\tilde{p})}[w, \tilde{f}] := (Z_{train}_n^{(\tilde{p})}[w, \tilde{f}] - \mu_n[w, \tilde{f}]) / \sigma_n[w, \tilde{f}]$$

$$Z_{val}_n^{(\tilde{p})}[w, \tilde{f}] := (Z_{val}_n^{(\tilde{p})}[w, \tilde{f}] - \mu_n[w, \tilde{f}]) / \sigma_n[w, \tilde{f}]$$

$$Z_{test}_n^{(\tilde{p})}[w, \tilde{f}] := (Z_{test}_n^{(\tilde{p})}[w, \tilde{f}] - \mu_n[w, \tilde{f}]) / \sigma_n[w, \tilde{f}]$$

for $\tilde{p} \in \{1, \dots, P\}$ and $n \in \{1, \dots, N\}$ and $w \in \{1, \dots, W\}$ and $\tilde{f} \in \{1, \dots, L\}$
where the μ denotes a mean across examples in the training set: $\mu_n[w, \tilde{f}] = \frac{1}{P_{train}} \sum_{\tilde{p}=1}^{P_{train}} Ztrain_n^{(\tilde{p})}[w, \tilde{f}]$; and the σ denotes a standard deviation across
examples in the training set: $\sigma_n[w, \tilde{f}] = \sqrt{\frac{1}{P_{train}} \sum_{\tilde{p}=1}^{P_{train}} (Ztrain_n^{(\tilde{p})}[w, \tilde{f}] - \mu_n[w, \tilde{f}])^2}$.

In the first two steps, the shorthand notations *createSpectrograms*(\cdot) and *splitData*(\cdot) are used to denote Algorithm 3 and the division of the data set into three subsets. The relative sizes of subsets were the following: training set (70 %), validation set (15 %), and test set (15 %).

3. CNN Architecture:

The architecture of the convolutional neural network used was very standard: convolutional layers intermingled by max pooling layers towards the shallow end (input layer), and fully connected layers towards the deep end (output layer).

CNN Architecture:

Layer Type	Attributes	Activation
Input	input shape: W x L x N	N/A
Conv2D	number of kernels: 10 kernel size: 3 x 3 stride size: 1 x 1	ReLu
MaxPool2D	pooling size: 2 x 2 stride size: 2 x 2	N/A
Flatten	N/A	N/A
Dense	number of nodes: 140	ReLu
Dense	number of nodes: 140	ReLu
Output	number of nodes: 1	Sigmoid

The input shape (at the input layer) references the variables used in the PSD Spectrograms algorithm (Algorithm 3) in the previous section. The number of parameters of the CNN depend on the input shape.

4. CNN Training:

The CNN was trained using *binary cross entropy* as the loss function and the *Adam* optimization algorithm. In addition, L2 regularization was applied to the fully connected (including output) layers of the network in order to reduce overfitting – the L2 regularization parameter was considered as a hyperparameter for subsequent tuning.

5. Hyperparameter Tuning:

The following hyperparameters were tuned, with their optimal values shown:

Hyperparameters	Optimal Value
window size for data set augmentation	2.5 s
stride size for data set augmentation	0.25 s
window size for spectrogram generation	0.8 s
stride size for spectrogram generation	0.05 s
max frequency	45.0 Hz
number of frequency bins	50
number of convolutional/max pooling layers	1
number of fully connected layers	2
number of convolutional kernels per layer	10
number of nodes in fully connected layers	140
regularization type	L2
L2 regularization parameter	0.035
number of training epochs	50

It should be noted that the hyperparameter search conducted was not very extensive or thorough, due to insufficient computing power and time.

IV: Results:

All of the results presented in this section were generated with the hyperparameter values listed in the previous section.