EEG Hand Movement Classification

Shanti Stewart

Information Processing Group

Oregon State University

June 28, 2020

I. Data Set

The widely used BCI Competition IV 2b data set was used to evaluate the proposed methods in this document. This data set contains EEG data from 9 healthy subjects performing 2 types of movements: left-hand and right-hand movements. The EEG signals were recorded by 3 electrodes (C3, C4, CZ) at a sampling frequency of 250 Hz. Each recorded movement is considered a trial: each trial has a total duration of around 9 seconds (the exact lengths differed by trial). For each subject, 5 sessions of data were collected; each of the first 2 sessions has 120 trials, and each of the last 3 sessions has 160 trials. Thus, each subject has a total of 720 trials. Since the beginning and end of each trial contain time for cues and resting, the data segment from 3.5 to 7 seconds was extracted – these segments are generally referred to as examples in the rest of this document.

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):

   $$\hat{R}_{xx}[k] \;=\; \frac{1}{M}\sum_{t=k}^{M} x[t]x[t-k], \qquad for\ 0 \le k \le M$$

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

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

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

   1.3. Normalize Fourier transform:

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

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:

$$S_{avg}[i] = mean\big(S\big[bins[i, 0] : bins[i, 1]\big]\big), \qquad for\ i \in \{1, \dots, num\_bins\}$$

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:

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

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

3.3. Compute the eigendecomposition of each statistical matrix:

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

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

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

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

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:

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

$$for\ w = 1\ to\ W:$$

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

4.2. Estimate power spectral density using Algorithm 0:

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

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

$$bins[i, 0] = f_i\ and\ bins[i, 1] = f_{i+1}, \qquad for\ i \in \{0, \dots, L-1\}$$

$$Z[w, :] = averagePSD(S[w, :], bins), \qquad for\ w \in \{1, \dots, W\}$$

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 $estimatePSD(\cdot)$ and $averagePSD(\cdot)$ 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).

Each subject was treated completely separately in the experiments: a separate model (CNN) instance was trained and evaluated on each subject's data separately.

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):

$$\{x_n^{(1)}[t], \dots, x_n^{(P)}[t]\} := shuffle\left(x_n^{(1)}[t], \dots, x_n^{(P)}[t]\right),$$
$$for\ n \in \{1, \dots, N\}\ and\ t \in \{1, \dots, M\}$$

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

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

$$for\ w = 1\ to\ W:$$

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

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

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

In the above steps, NumPy/MATLAB notation is used to denote array indexing and slicing. In step 1.3, the shorthand notation $combineAxis(\cdot)$ is used to denote the combining of multiple axes (which can be implemented with NumPy's 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: $Ztrain_n^{(\tilde{p})}[w, \tilde{f}]$ = standardized training set features, $Zval_n^{(\tilde{p})}[w, \tilde{f}]$ = standardized validation set features, $Ztest_n^{(\tilde{p})}[w, \tilde{f}]$ = standardized test set features

Algorithm:

2.1. Generate spectrogram features using Algorithm 3:

$$Z_n^{(\tilde{p})}[w, \tilde{f}] = createSpectrograms\left(\tilde{x}_n^{(\tilde{p})}[t], \delta, \gamma, \{0, f_1, \dots, f_L\}\right),$$
$$for\ \tilde{p} \in \{1, \dots, P\}\ and\ n \in \{1, \dots, N\}$$

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

$$Ztrain_n^{(\tilde{p})}[w,\tilde{f}], Zval_n^{(\tilde{p})}[w,\tilde{f}], Ztest_n^{(\tilde{p})}[w,\tilde{f}] \;=\; splitData(Z_n^{(\tilde{p})}[w,\tilde{f}])$$

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

$$\{Ztrain_n^{(1)}[w,\tilde{f}], \dots, Ztrain_n^{(P_{train})}\} := shuffle\left(\{Ztrain_n^{(1)}[w,\tilde{f}], \dots, Ztrain_n^{(P_{train})}\right),$$

$$\{Zval_n^{(1)}[w,\tilde{f}], \dots, Zval_n^{(P_{val})}\} := shuffle\left(\{Zval_n^{(1)}[w,\tilde{f}], \dots, Zval_n^{(P_{val})}\right),$$

$$\{Ztest_n^{(1)}[w,\tilde{f}], \dots, Ztest_n^{(P_{test})}\} := shuffle\left(\{Ztest_n^{(1)}[w,\tilde{f}], \dots, Ztest_n^{(P_{test})}\right),$$

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

2.4. Standardize data, with respect to training set:

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

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

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

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

where the mu 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 sigma 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: alternating convolutional and max pooling layers towards the shallow end (input layer), and fully connected layers towards the deep end (output layer). The architecture of the network is shown in Table 1.

Table 1: Architecture of CNN.

Layer Type	Attributes	Activation
Input	input shape: W x L x N	N/A
Conv2D	number of kernels: 12 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: 130	ReLu
Dense	number of nodes: 130	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:

   Hyperparameter tuning was performed using the average validation accuracies across subjects. A standard random search was conducted to tune the hyperparameters. The hyperparameters tuned and their optimal values are shown in Table 2.

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

Table 2: Optimal values of hyperparameters that were tuned.

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	46.0 Hz
number of frequency bins	47
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	0.032
number of training epochs	75

IV. Results

All of the results presented in this section were generated with the hyperparameter values listed in the previous section. Furthermore, these results should be viewed as intermediate rather than final results, since this research project is far from complete.

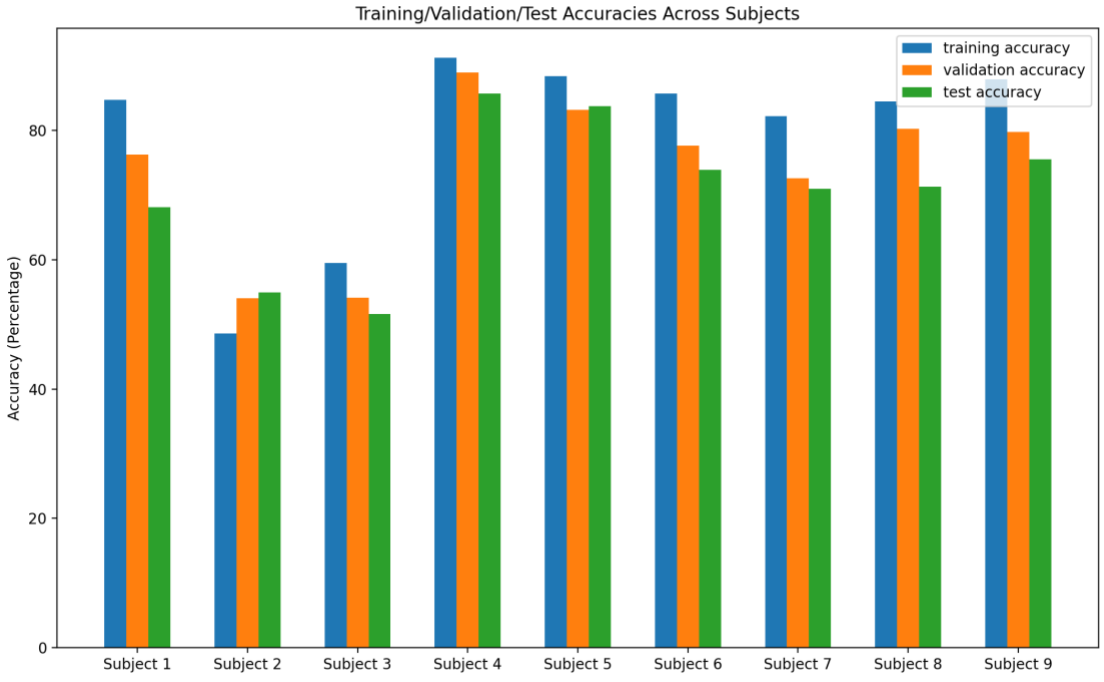
1. Performance of CNN Across Subjects:

   The binary classification accuracies of the CNN model (with the hyperparameter values listed in the previous section) on predicting right-hand vs. left-hand movement for all 9 subjects are shown in Table 3 and Figure 1.

Table 3: Training, validation, and test set accuracies of CNN for all 9 subjects.

Subject Number	Training Accuracy	Validation Accuracy	Test Accuracy
1	84.7 %	76.3 %	68.1 %
2	48.6 %	54.0 %	54.9 %
3	59.5 %	54.1 %	51.6 %
4	91.2 %	89.0 %	85.7 %
5	88.4 %	83.2 %	83.8 %
6	85.7 %	77.7 %	73.9 %
7	82.2 %	72.6 %	70.9 %
8	84.5 %	80.3 %	71.3 %
9	87.9 %	79.8 %	75.6 %

Figure 1: Bar graph of training, validation, and test set accuracies of CNN for all 9 subjects.



As can be seen, the performance of the CNN is significantly worse for subjects 2 and 3. Thus, the average accuracies across subjects are considered for two sets of subjects: all subjects, and all subjects except subjects 2 and 3.

Table 4: Average training, validation, and test set accuracies across all subjects and all subjects except subjects 2 and 3.

	Average Training Accuracy	**Average Validation Accuracy**	**Average Test Accuracy**
All Subjects	79.2 %	74.1 %	70.7 %
Subjects 1, 4-9	86.4 %	79.8 %	75.6 %

From the results of this section, it is quite clear that the CNN model's performance greatly varies from subject to subject. The CNN consistently performed the best on subject 4 and performed the worst on subjects 2 and 3. Additionally, the average test accuracy was notably less than the average validation accuracy (by around 3-4 % for both subject sets), which suggests that the hyperparameter tuning was slightly overfitted to the validation set. Excluding subjects 2 and 3 (i.e. considering them as outliers), the final test classification accuracy was **75.6 %**. These results have a large room for improvement, and there are many further improvements and possibilities to be explored.