

Classifying EEG Spectrograms by Phalangeal Articulations utilizing Long-term Recurrent Convolutional Neural Networks

**Robert Valencia**

Electrical and Biomedical Engineering IV (Co-op)

**Electrical Engineering 4OJ4/4OH4 Research Project**

Submission date: May 23, 2017

Supervisor: Dr. James P. Reilly

McMaster University

Faculty of Engineering

Department of Electrical and Computer Engineering

Table of Contents

[1. Introduction 4](#_Toc483260635)

[2. Theory of Operation 4](#_Toc483260636)

[2.1. Data 4](#_Toc483260637)

[2.2. Model 5](#_Toc483260638)

[2.2.1. Convolutional Neural Networks (CNNs) 5](#_Toc483260639)

[2.2.2. Recurrent Neural Networks (RNNs) 8](#_Toc483260640)

[2.2.3. Multilayer Perceptrons (MLPs) 10](#_Toc483260641)

[3. System Architecture 12](#_Toc483260642)

[3.1. Overview 12](#_Toc483260643)

[3.2. Model 13](#_Toc483260644)

[4. Requirements 13](#_Toc483260645)

[5. Application Programming Interface 14](#_Toc483260646)

[5.1. Training Configuration File 14](#_Toc483260647)

[5.2. ESPA Module 14](#_Toc483260648)

[5.2.1. *class* ESPAModel 14](#_Toc483260649)

[5.2.2. *class* ProgressDisplay 16](#_Toc483260650)

[5.2.3. Auxiliary Functions 17](#_Toc483260651)

[5.3. EEG Processor Module 17](#_Toc483260652)

[5.4. Converter Module 17](#_Toc483260653)

[6. Training 17](#_Toc483260654)

[6.1. Setup 17](#_Toc483260655)

[6.2. Results and Discussion 18](#_Toc483260656)

[7. Recommendations 18](#_Toc483260657)

[8. References 19](#_Toc483260658)

[9. Appendix 20](#_Toc483260659)

[9.1. training\_config.json 20](#_Toc483260660)

[9.2. espa.py 21](#_Toc483260661)

Figure 1 Sample spectrogram from 1 channel for sustained left middle finger flexion 4

Figure 2 Applications of LRC neural networks (Source: http://jeffdonahue.com/lrcn/) 5

Figure 3 A sample image, its grayscale representation, and its simplified digital representation 6

Figure 4 A sample filter and its simplified digital representation 6

Figure 5 Generation of an activation map by tiling a filter across an image and performing successive convolutions 7

Figure 6 Application of a ReLU activation layer and a MaxPool pooling layer to an activation map 7

Figure 7 General VGG-16 architecture, adapted for this model 8

Figure 8 Chain of repeating LSTM units 8

Figure 9 LSTM operations 9

Figure 10 Simple illustration of how projecting input data into a feature space enables classification. Two classes are represented by a blue square and a yellow circle. A decision surface is represented by a red square. 10

Figure 11 Sample MLP with a 2-node input layer, 3-node hidden layer, and 2-node output layer. 11

Figure 12 General system architecture 12

Figure 13 ESPA model architecture 13

Figure 14 Accuracies (average training, validation, and testing) across all three setups 18

# 1. Introduction

This design document contains information on the operation, architecture, API, and performance of the ESPA system, developed for the research on classifying EEG spectrograms by phalangeal articulations utilizing long-term recurrent convolutional (LRC) neural networks.

# 2. Theory of Operation

## 2.1. Data

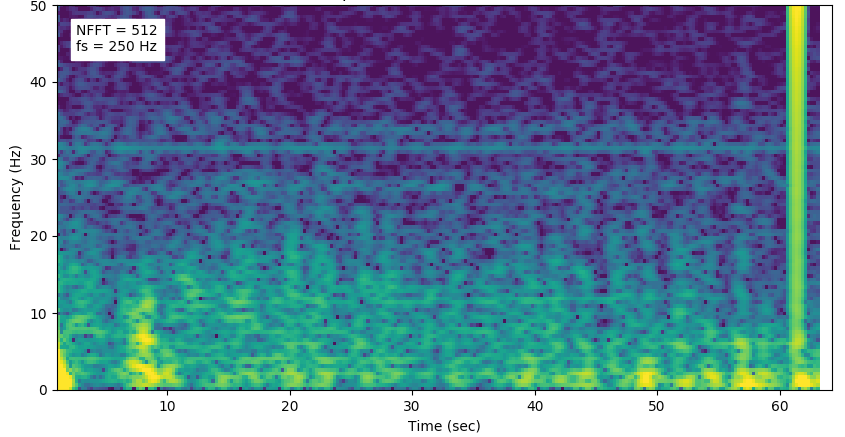


Figure 1 Sample spectrogram from 1 channel for sustained left middle finger flexion

For the inputs, the raw data consists of EEG time signals from 8 channels. The data from each channel is cleaned by performing the following steps:

1. Trim setup and teardown data
2. Remove DC offset
3. Notch mains interference
4. Bandpass filter frequencies from 1 to 50 Hz

After cleaning the data, spectrograms are computed, then partitioned into multiple samples with a dimensionality of 250 frequency points by 50 time points. Also, depending on the training run configuration, the samples are either replicated or augmented to increase the sample size. Finally, each sample is labelled with a one-hot encoded representation of its class. For example, [1., 0., 0.] would indicate class 1 in a 3-class model.

For the outputs, arrays containing predicted probabilities for each class are utilized, where the probabilities are represented in fractional form. For example, [0.75, 0.15, 0.10] would indicate 75% probability for class 1, 15% probability for class 2, and 10% probability for class 3.

## 2.2. Model

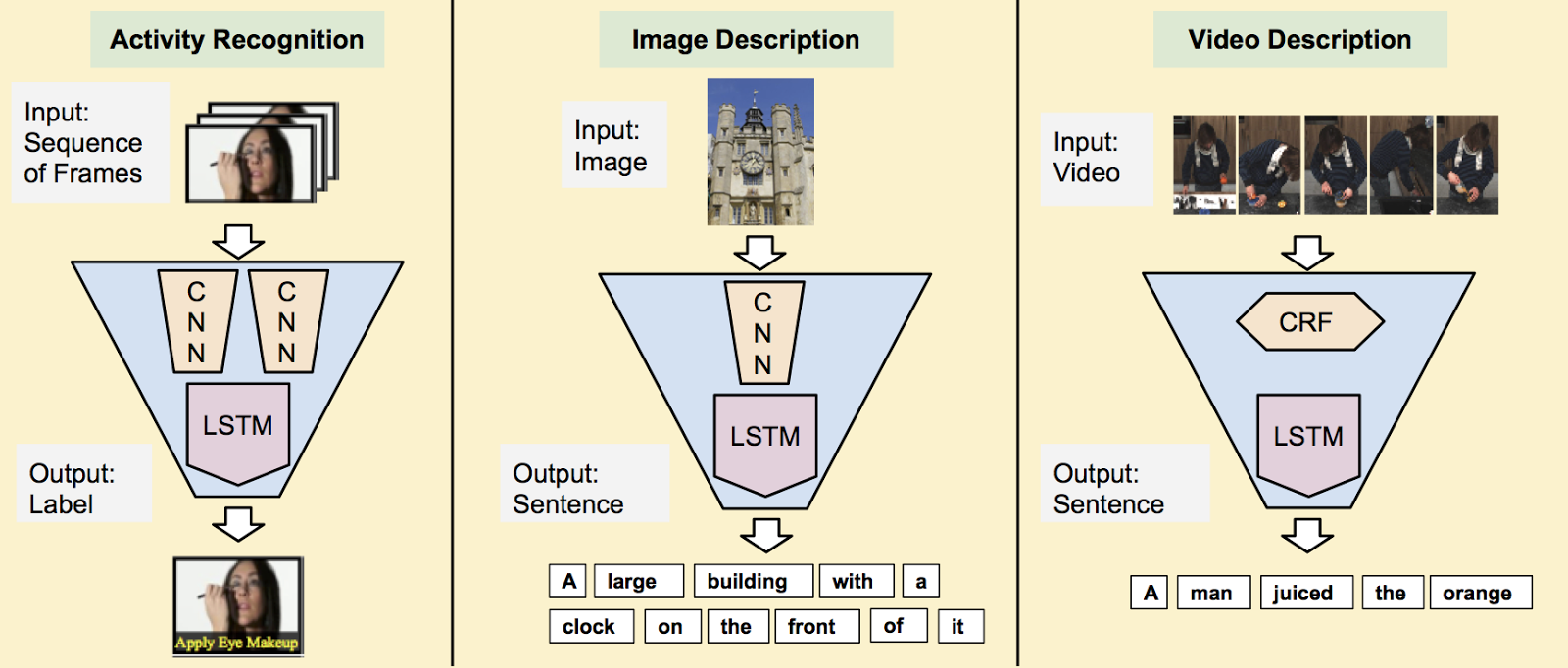


Figure 2 Applications of LRC neural networks (Source: http://jeffdonahue.com/lrcn/)

The model is based on long-term recurrent convolutional (LRC) neural networks, a class of neural networks used for visual and sequence learning [1]. It consists of a hybrid architecture of convolutional neural networks, recurrent neural networks, and multilayer perceptrons.

### 2.2.1. Convolutional Neural Networks (CNNs)

CNNs are biologically-inspired artificial neural networks that mimic the visual cortex. In a visual cortex, there are complex arrangements of cells that are sensitive to stimuli within a restricted region known as a receptive field. This region is tiled across an entire visual field, where the cells act as localized filters for detecting spatial patterns, the response to which can be approximated by a convolution operation [2][3]:

To illustrate how CNNs work, a sample 5x5 image, its grayscale conversion, and its simplified digital representation, where 1 is the maximum value instead of 255, is shown below:

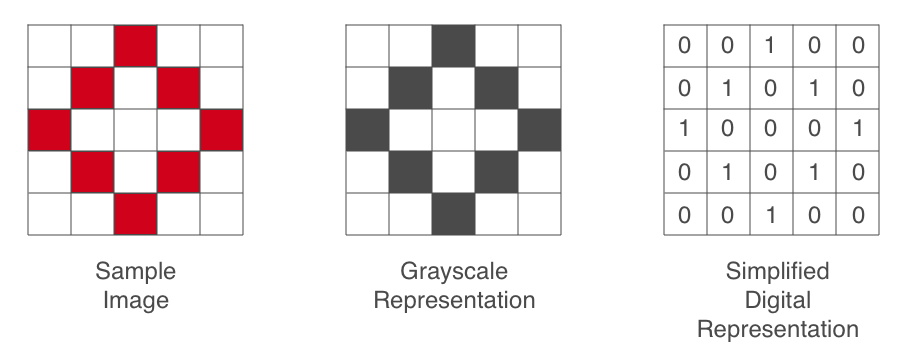


Figure 3 A sample image, its grayscale representation, and its simplified digital representation

A sample 3x3 filter and its simplified digital representation is also shown below:

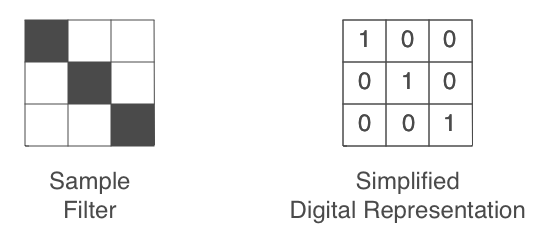


Figure 4 A sample filter and its simplified digital representation

In CNNs, convolutional filters are tiled across an image. As the filters tile across an image, receptive fields are convolved with their corresponding visual field regions, generating an activation map. In these activation maps, regions with with a high correlation with the filter pattern have high activation values, and vice versa. In this example, the filter has a stride of 1 pixel, generating a 3x3 activation map:

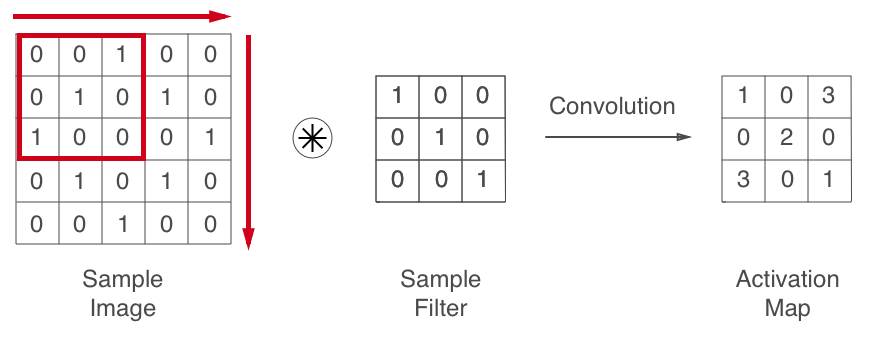


Figure 5 Generation of an activation map by tiling a filter across an image and performing successive convolutions

These activation maps are then passed through a layer of rectified linear units (ReLUs), an activation function used to improve the network’s nonlinearity:

Finally, a pooling layer downsamples the activation maps, reducing the number of parameters. In this example, a type of pooling layer called MaxPool is used, which replaces a pool of values with its maximum values, with a pool size of 2x2 and a stride of 1 pixel:

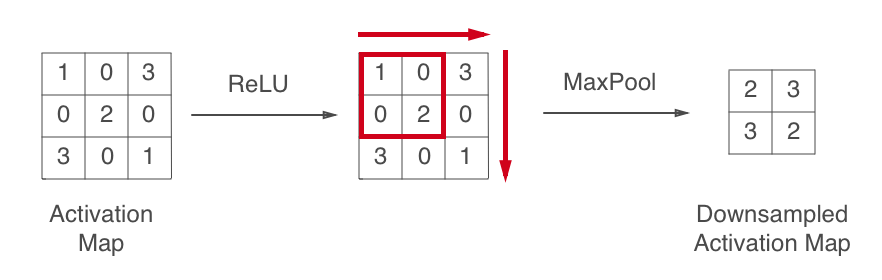


Figure 6 Application of a ReLU activation layer and a MaxPool pooling layer to an activation map

In practical applications, multiple alternating layers of convolution, ReLU activation, and MaxPool pooling are utilized. For this model, an architecture called VGG-16 [4] is used:

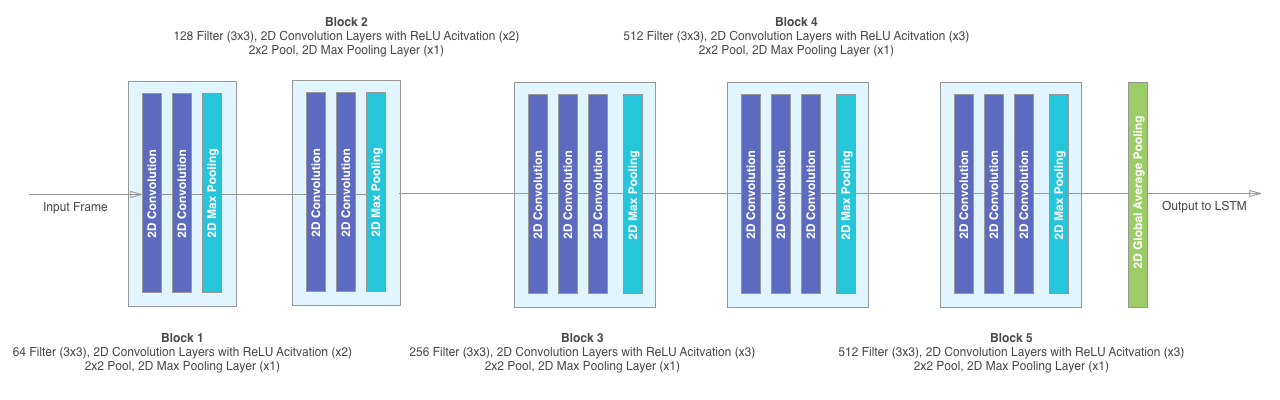


Figure 7 General VGG-16 architecture, adapted for this model

### 2.2.2. Recurrent Neural Networks (RNNs)

RNNs are artificial neural networks that are used for detecting sequential patterns. They consist of stateful memory units that are cyclically connected. One specific type of RNN, called long short-term memory (LSTM) [5] is used in the model. LSTMs, similar to regular RNNs, consist of chains of repeated LSTM units. However, unlike regular RNNs, LSTMs are well suited for data with variable gaps between events, such as variations observed in speech due to demographic and biological variability. To demonstrate how LSTMs work, diagrams and descriptions adapted from Colah’s blog [6] and DeepLearning tutorials [7] are shown below:

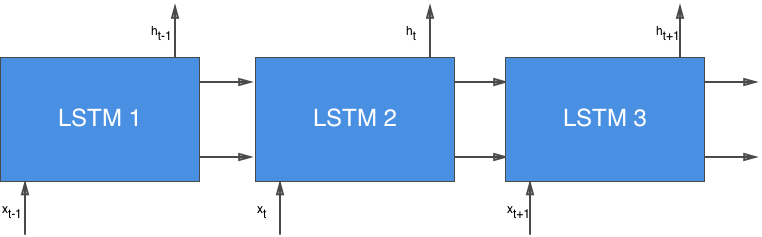


Figure 8 Chain of repeating LSTM units

Within each LSTM unit, several operations occur, which are represented by yellow circles on the diagram below:

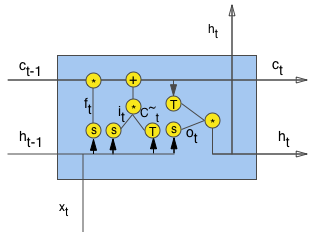


Figure 9 LSTM operations

In the diagram above, **S** represents a logistic sigmoid operation:

**T** represents a hyperbolic tangent operation:

**+** represents element-wise addition, and **\*** represents element-wise multiplication. LSTM operations also utilize weight matrices **W**, **U**, and **V**, and bias vector **b**. First, the LSTM unit selects new data to store, which involves a logistic sigmoid layer (input gate) that selects which values to update:

and a hyperbolic tangent layer that generates new candidate values:

Next, the LSTM unit selects data to forget, which involves another logistic sigmoid layer:

The layer takes in the input, , and the previous output, , then returns either 0 or 1 for each value in the cell state , where 0 represents “forget” and 1 represents “remember”. Then, the LSTM unit updates the cell state from the old state to the new state :

This operation forgets what has to be forgotten by multiplying the old cell state with the output of the forget gate , and adds new candidate values scaled by update weights by multiplying the new cell state with the output of the input gate . Finally, the LSTM unit generates the output. First, a logistic sigmoid layer selects which values of the cell state to output:

Next, the cell state values pass through a hyperbolic tangent layer, scaling the values between -1 and 1. Finally, the outputs are multiplied, resulting in a filtered cell state:

### 2.2.3. Multilayer Perceptrons (MLPs)

MLPs are artificial neural networks that consist of fully-connected layers of nodes. They map input data into outputs via a learned nonlinear transformation, which projects input data into a space where they become linearly separable, enabling classification:

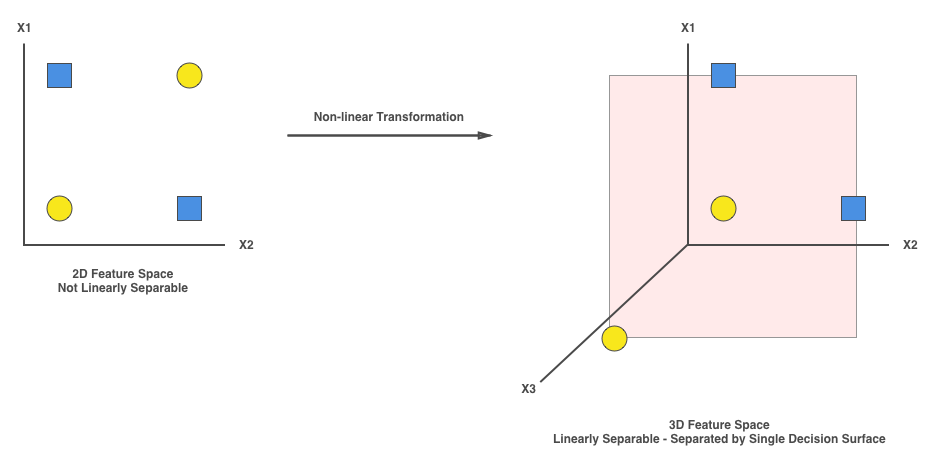


Figure 10 Simple illustration of how projecting input data into a feature space enables classification. Two classes are represented by a blue square and a yellow circle. A decision surface is represented by a red square.

MLPs consist of 3 primary stages: an input layer, hidden layers, and an output layer. With at least 1 hidden layer, an MLP becomes a universal approximator [8]. However, in practical deep learning applications, multiple hidden layers are utilized to generate more features. In the example below, an MLP with a 2-node input layer, 3-node hidden layer, and 2-node output layer is shown:

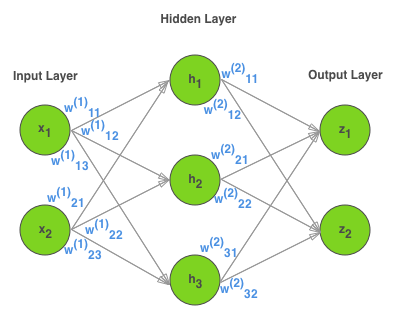


Figure 11 Sample MLP with a 2-node input layer, 3-node hidden layer, and 2-node output layer.

In MLPs, input nodes represent input features, hidden nodes represent generated features, and output nodes represent predicted class probabilities. To make predictions, an algorithm called forward propagation is used [8]:

Where **x** is the input layer vector, **h** is the hidden layer vector, **z** is the output layer vector, **b** are bias vectors, **w** are weight matrices, **s** is the hidden layer activation function, which is set to ReLU for this model, and **G** is the output layer activation function, which is set to the softmax function for multi-class classification:

Initially, the learned parameters from **w** are randomized, resulting high error and low accuracy values. To improve accuracies, parameters are learned via the backpropagation algorithm [9][10], which trains the model on labelled data and updates parameters until a cost function is minimized.

# 3. System Architecture

## 3.1. Overview

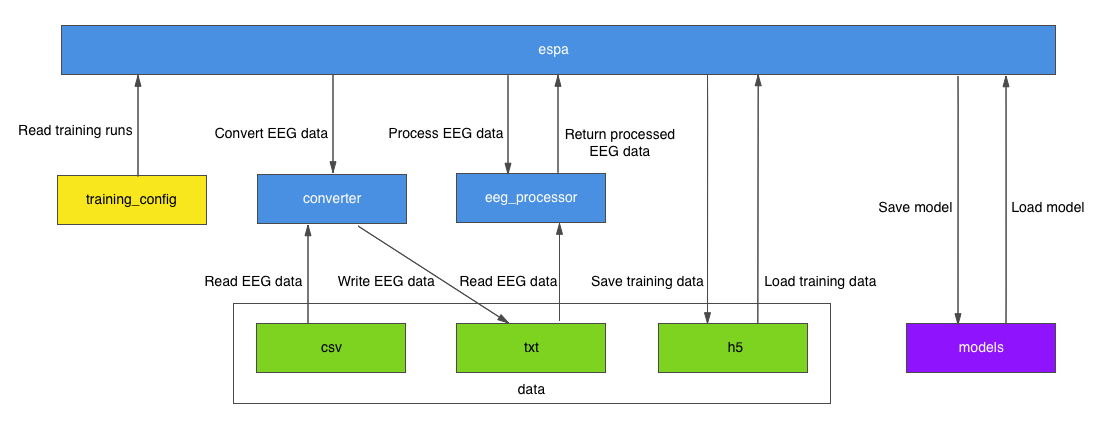


Figure 12 General system architecture

The diagram above shows the general system architecture of the ESPA system. Blue components represent Python modules, yellow components represent configuration files, green components represent data files, and purple components represent model files.

The espa module acts as the core module with which the user interfaces for executing training workflows. The converter module performs data type conversion operations for the raw EEG data. The eeg\_processor module performs preprocessing operations on the raw EEG data. The training\_config file enables the user to specify configurations for multiple training runs. The csv and txt data files store raw EEG data, while h5 data files store preprocessed training data. Finally, models store multiple model data, which come in pairs of model and weights data, stored in JSON and HDF5 files, respectively.

## 3.2. Model

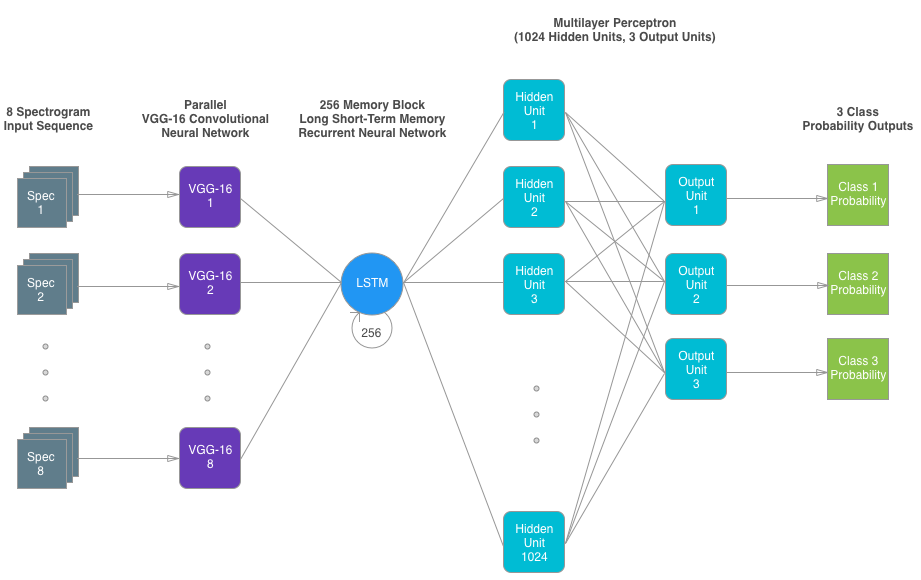


Figure 13 ESPA model architecture

The diagram above shows the ESPA model architecture. The model starts of with the input, which consists of a sequence of 8 spectrograms, each of which are replicated into 3 channels to match the expected input dimensions of the VGG-16 CNNs. Next, 8 VGG-16 CNNs processes the inputs in parallel. Then, a 256-memory block LSTM RNN processes the output of the previous stage. Finally, an MLP with a 1024-hidden unit hidden layer and 3-unit output layer processes the output of the previous layer and yields 3 class probability outputs.

# 4. Requirements

The ESPA system depends on the following:

1. Python, for general purpose programming
2. OpenCV, for computer vision
3. Keras, for implementing neural networks
4. Theano, for scientific and mathematical computations
5. SciPy, for scientific and mathematical computations
6. (Recommended) Nvidia GPU, for running computations in parallel

# 5. Application Programming Interface

## 5.1. Training Configuration File

The training configuration file, training\_config.json, contains specifications for training runs. The configuration file consists of a list of training run specification, each of which have the following format:

{

"run\_name": *string* name of training run,

"trials": *integer* number of trials,

"data\_save\_fn": *string* sample data save file path,

"validation\_ratio": *float* fraction of data allocated for validation,

"testing\_ratio": *float* fraction of data allocated for testing,

"samples\_generated\_per\_sample": *integer* number of samples generated/sample,

"augmentation": *boolean* augmentation flag,

"augmentation\_magnitude": *float* augmentation magnitude,

"freq\_points": *integer* number of discrete spectrogram frequency points,

"time\_points": *integer* number of discrete spectrogram time points,

"espa\_save\_fn": *string* ESPA model save file path,

"espa\_weights\_save\_fn": *string* ESPA model weights save file path

}

## 5.2. ESPA Module

The ESPA module, espa.py, provides the core user interface for executing training workflows.

### 5.2.1. *class* ESPAModel

Core class for instantiating and interfacing with ESPA models

#### 5.2.1.1. method constructor

Constructor for the ESPAModel class.

Arguments:

1. data\_save\_fn: *string* sample data save file path
2. validation\_ratio: *float* fraction of data allocated for validation
3. testing\_ratio: *float* fraction of data allocated for testing
4. samples\_generated\_per\_sample: *integer* number of samples generated/sample
5. augmentation: *boolean* augmentation flag
6. augmentation\_magnitude: *float* augmentation magnitude
7. freq\_points: *integer* number of discrete spectrogram frequency points
8. time\_points: *integer* number of discrete spectrogram time points
9. espa\_save\_fn: *string* ESPA model save file path
10. espa\_weights\_save\_fn: *string* ESPA model weights save file path

Returns:

None

#### 5.2.1.2. method train\_espa\_model

Trains the ESPA model

Arguments:

None

Returns:

1. metrics\_history: *Keras History object*history of training and validation metric values

#### 5.2.1.3. method test\_espa\_model

Tests the ESPA model

Arguments:

None

Returns:

1. metrics: *dictionary* testing categorical accuracy and loss

#### 5.2.1.4. generator method generate\_data

Generates data from HDF5 sample data file on demand

Arguments:

1. data\_save\_file: *HDF5 file object* sample data file
2. data\_set: *string* data set to read
3. sample\_idxs: *Numpy array* array of sample indices
4. batch\_size: *integer*number of samples for each batch

Yields:

1. Tuple of Numpy arrays containing a batch of samples and their labels

#### 5.2.1.5. method print\_espa\_summary

Prints a summary representation of the ESPA model

Arguments:

None

Returns:

None

#### 5.2.1.6. method generate\_espa\_model

Compiles the ESPA model

Arguments:

None

Returns:

None

#### 5.2.1.7. method save\_espa\_model

Saves the ESPA model and weights

Arguments:

None

Returns:

None

#### 5.2.1.8. method load\_espa\_model

Loads the ESPA model and weights

Arguments:

None

Returns:

None

#### 5.2.1.9. method process\_data

Preprocesses sample data

Arguments:

None

Returns:

None

#### 5.2.1.10. method replicate\_augment\_data

Replicates or augments sample data

Arguments:

1. X\_h: *Numpy array* pre replication or augmentation sample data
2. Y\_h: *Numpy array* pre replication or augmentation sample data labels

Returns:

1. X: *Numpy array* post replication or augmentation sample data
2. Y: *Numpy array* post replication or augmentation sample data labels

### 5.2.2. *class* ProgressDisplay

Callback class for displaying progress updates

#### 5.2.2.1. method on\_batch\_end

Displays metric values at the end of each batch

Arguments:

1. epoch: *integer* epoch number
2. logs: *dictionary* log of metrics and their values

Returns:

None

### 5.2.3. Auxiliary Functions

#### 5.2.3.1. function get\_training\_configuration

Acquires training configuration from a training configuration file

Arguments:

1. training\_config\_fn: *string* training configuration file path

Returns:

1. training\_config: *list* training configuration data

#### 5.2.3.2. function execute\_training\_runs

Executes training runs from a specified training configuration

Arguments:

1. training\_config: *list* training configuration data

Returns:

1. results: *dictionary* compiled results data from training runs

#### 5.2.3.3. function save\_results

Saves compiled results data to a JSON file

Arguments:

1. results\_save\_fn: *string* compiled results data save file path

Returns:

None

## 5.3. EEG Processor Module

The EEG processor module, eeg\_processor.py, performs preprocessing operations on raw EEG data. This module was adapted from EEGrunt.py of the EEGrunt package, developed by Curiositry. For more information, please visit the EEGrunt repository: https://github.com/curiositry/EEGrunt.

## 5.4. Converter Module

The converter module, converter.py, performs data type conversion operations for raw EEG data. This module was adapted from convert\_txt\_to\_csv.py of the EEGrunt package, developed by Curiositry. For more information, please visit the EEGrunt repository: https://github.com/curiositry/EEGrunt.

# 6. Training

## 6.1. Setup

The raw data comprises of 9, 8-channel EEG data, saved in text format, 3 for each of the following classes:

1. left index finger flexion
2. left middle finger flexion
3. left ring finger flexion

The data is then converted from text files into CSV files. Then, the data is filtered and trimmed to remove DC offset, mains interference, and setup/teardown artifacts. Next, spectrograms are calculated for each channel, each of which split into samples with 250 discrete frequency points and 50 discrete time points, and replicated 3 times to match the CNN’s input dimensions, which expects 3-colour channel RGB inputs, generating a 30 x 8 x 3 x250 x 50 training dataset. Finally, the data is saved into an HDF5 file, which is vital for on-demand loading of data as a workaround for memory resource limitations.

The following training setups were implemented, with 3 trials per setup, and 10 epochs per trial:

1. Replication, 10x sample count
2. 1% augmentation, 10x sample count
3. 5% augmentation, 10x sample count

For each setup, the data was split into the following components:

1. 60% training data
2. 20% validation data
3. 20% testing data

## 6.2. Results and Discussion

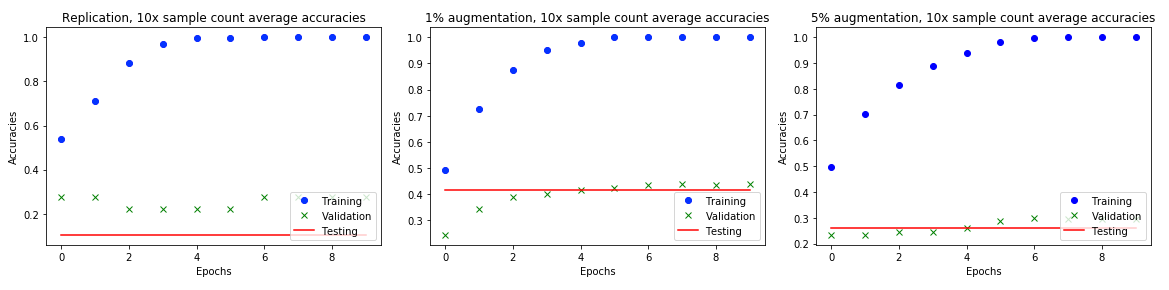


Figure 14 Accuracies (average training, validation, and testing) across all three setups

Across all three setups, it can be observed that the training accuracies ascend to high values and plateau within several epochs, while the validation accuracies lag behind the training accuracies, which could indicate overfitting. It can also be observed that both the validation and testing accuracies are significantly higher with 1% augmentation, which could indicate that 1% augmentation provides a good balance between generating independent samples and introducing excessive noise.

# 7. Recommendations

The primary issue that has to be addressed is overfitting, which prohibits the model from generalizing to new data. Some potential solutions include:

1. Increasing the raw sample size instead of depending entirely on data augmentation
2. Implementing regularization: L1, L2, and max norm
3. Implementing dropout

Other alternative changes that could potentially improve the model’s performance include the choice of CNN architecture (e.g. ResNet, Inception), RNN architecture (e.g. Gated Recurrent Unit [GRU]), MLP architecture (deeper [more hidden layers] and wider [more neurons]). With an improved model, classification could be expanded to classify finer phalangeal articulations, given sufficient training data.

# 8. References

[1] J. Donahue, L. A. Hendricks, S. Guadarrama, M. Rohrbach, S. Venugopalan, T. Darrell, and K. Saenko, “Long-term recurrent convolutional networks for visual recognition and description,” 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2015.

[2] “Convolutional Neural Networks (LeNet),” Convolutional Neural Networks (LeNet) — DeepLearning 0.1 documentation. [Online]. Available: http://deeplearning.net/tutorial/lenet.html. [Accessed: 22-Apr-2017].

[3] S. B. Damelin and W. Miller, The mathematics of signal processing. Cambridge: Cambridge University Press, 2012.

[4] K. Simonyan and A. Zisserman, “Very Deep Convolutional Networks for Large-Scale Image Recognition,” [1409.1556] Very Deep Convolutional Networks for Large-Scale Image Recognition, 10-Apr-2015. [Online]. Available: https://arxiv.org/abs/1409.1556. [Accessed: 22-Apr-2017].

[5] [Sepp Hochreiter](https://en.wikipedia.org/wiki/Sepp_Hochreiter); [Jürgen Schmidhuber](https://en.wikipedia.org/wiki/J%C3%BCrgen_Schmidhuber) (1997). ["Long short-term memory"](https://www.researchgate.net/publication/13853244_Long_Short-term_Memory). [Neural Computation](https://en.wikipedia.org/wiki/Neural_Computation_(journal)). 9 (8): 1735–1780.

[6] C. Olah, “Understanding LSTM Networks,” Understanding LSTM Networks -- colah's blog. [Online]. Available: https://colah.github.io/posts/2015-08-Understanding-LSTMs/. [Accessed: 24-Apr-2017].

[7] “LSTM Networks for Sentiment Analysis,” LSTM Networks for Sentiment Analysis — DeepLearning 0.1 documentation. [Online]. Available: http://deeplearning.net/tutorial/lstm.html. [Accessed: 22-Apr-2017].

[8] “Multilayer Perceptron,” Multilayer Perceptron — DeepLearning 0.1 documentation. [Online]. Available: http://deeplearning.net/tutorial/mlp.html. [Accessed: 22-Apr-2017].

[9] C. V. D. Malsburg, “Frank Rosenblatt: Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms,” Brain Theory, pp. 245–248, 1986.

[10] D. Rumelhart, G. Hinton, and R. Williams, “Learning Internal Representations by Error Propagation,” Readings in Cognitive Science, pp. 399–421, 1988.

# 9. Appendix

All source files can be found at the ESPA repository (https://github.com/valencra/eeg-spectrogram-phalangeal-articulation)

## 9.1. training\_config.json

[{

**"run\_name"**: "Replication, 10x sample count",

**"trials"**: 3,

**"data\_save\_fn"**: "data/h5/rep\_10x\_data.h5",

**"validation\_ratio"**: 0.2,

**"testing\_ratio"**: 0.2,

**"samples\_generated\_per\_sample"**: 10,

**"augmentation"**: **false**,

**"augmentation\_magnitude"**: 0.00,

**"freq\_points"**: 250,

**"time\_points"**: 50,

**"espa\_save\_fn"**: "models/rep\_10x\_espa\_model.json",

**"espa\_weights\_save\_fn"**: "models/rep\_10x\_espa\_weights.h5"

}, {

**"run\_name"**: "1% augmentation, 10x sample count",

**"trials"**: 3,

**"data\_save\_fn"**: "data/h5/1pcaug\_10x\_data.h5",

**"validation\_ratio"**: 0.2,

**"testing\_ratio"**: 0.2,

**"samples\_generated\_per\_sample"**: 10,

**"augmentation"**: **true**,

**"augmentation\_magnitude"**: 0.01,

**"freq\_points"**: 250,

**"time\_points"**: 50,

**"espa\_save\_fn"**: "models/1pcaug\_10x\_espa\_model.json",

**"espa\_weights\_save\_fn"**: "models/1pcaug\_10x\_espa\_weights.h5"

}, {

**"run\_name"**: "5% augmentation, 10x sample count",

**"trials"**: 3,

**"data\_save\_fn"**: "data/h5/5pcaug\_10x\_data.h5",

**"validation\_ratio"**: 0.2,

**"testing\_ratio"**: 0.2,

**"samples\_generated\_per\_sample"**: 10,

**"augmentation"**: **true**,

**"augmentation\_magnitude"**: 0.05,

**"freq\_points"**: 250,

**"time\_points"**: 50,

**"espa\_save\_fn"**: "models/5pcaug\_10x\_espa\_model.json",

**"espa\_weights\_save\_fn"**: "models/5pcaug\_10x\_espa\_weights.h5"

}]

## 9.2. espa.py

**from** **converter** **import** convert\_txt\_to\_csv

**from** **copy** **import** deepcopy

**from** **eeg\_processor** **import** EEGProcessor

**from** **keras** **import** backend **as** K

**from** **keras.applications.vgg16** **import** VGG16

**from** **keras.callbacks** **import** Callback

**from** **keras.models** **import** Model, model\_from\_json

**from** **keras.layers** **import** Dense, Dropout, Flatten, Input

**from** **keras.layers.pooling** **import** GlobalAveragePooling2D

**from** **keras.layers.recurrent** **import** LSTM

**from** **keras.layers.wrappers** **import** TimeDistributed

**from** **keras.optimizers** **import** Nadam

**from** **keras.preprocessing.image** **import** random\_rotation, random\_shift, random\_shear, random\_zoom

**from** **keras.utils.io\_utils** **import** HDF5Matrix

**from** **pprint** **import** pprint

**from** **math** **import** ceil

**from** **numpy** **import** log10

**from** **os** **import** listdir

**from** **os.path** **import** join, isfile

**from** **json** **import** dump, load

**import** **h5py**

**import** **numpy** **as** **np**

**import** **os**

**import** **sys**

**class** **ESPAModel**(object):

**def** \_\_init\_\_(self,

data\_save\_fn, validation\_ratio, testing\_ratio, samples\_generated\_per\_sample,

augmentation, augmentation\_magnitude, freq\_points, time\_points,

espa\_save\_fn, espa\_weights\_save\_fn):

K.set\_image\_dim\_ordering("th")

self.data\_save\_fn = data\_save\_fn

self.validation\_ratio = validation\_ratio

self.testing\_ratio = testing\_ratio

self.samples\_generated\_per\_sample = samples\_generated\_per\_sample

self.augmentation = augmentation

self.augmentation\_magnitude = augmentation\_magnitude

self.freq\_points = freq\_points

self.time\_points = time\_points

self.espa\_save\_fn = espa\_save\_fn

self.espa\_weights\_save\_fn = espa\_weights\_save\_fn

self.espa = None

**def** train\_espa\_model(self):

*""" Train the ESPA model*

*"""*

**print** "**\n**Training ESPA Model"

batch\_size = 32

**with** h5py.File(self.data\_save\_fn, "r") **as** data\_save\_file:

*# indices*

training\_sample\_idxs = np.random.permutation(range(int(data\_save\_file.attrs["training\_sample\_count"])))

validation\_sample\_idxs = np.random.permutation(range(int(data\_save\_file.attrs["validation\_sample\_count"])))

*# generators*

training\_sequence\_generator = self.generate\_data(data\_save\_file=data\_save\_file,

data\_set="train",

sample\_idxs=training\_sample\_idxs,

batch\_size=batch\_size)

validation\_sequence\_generator = self.generate\_data(data\_save\_file=data\_save\_file,

data\_set="val",

sample\_idxs=validation\_sample\_idxs,

batch\_size=batch\_size)

*# fit model*

progress\_display = ProgressDisplay()

metrics\_history = self.espa.fit\_generator(generator=training\_sequence\_generator,

validation\_data=validation\_sequence\_generator,

samples\_per\_epoch=len(training\_sample\_idxs),

nb\_val\_samples=len(validation\_sample\_idxs),

nb\_epoch=10,

verbose=2,

callbacks=[progress\_display],

class\_weight=None,

nb\_worker=1)

**return** metrics\_history

**def** test\_espa\_model(self):

*""" Test the ESPA model*

*"""*

**print** "**\n**Testing ESPA Model"

batch\_size = 32

**with** h5py.File(self.data\_save\_fn, "r") **as** data\_save\_file:

*# indices*

testing\_sample\_idxs = np.random.permutation(range(int(data\_save\_file.attrs["testing\_sample\_count"])))

*# generators*

testing\_sequence\_generator = self.generate\_data(data\_save\_file=data\_save\_file,

data\_set="test",

sample\_idxs=testing\_sample\_idxs,

batch\_size=batch\_size)

*# calculate steps*

sample\_count = len(testing\_sample\_idxs)

batches = int(sample\_count/batch\_size)

remainder\_samples = sample\_count%batch\_size

**if** remainder\_samples:

batches = batches + 1

*# test model*

metrics = self.espa.evaluate\_generator(testing\_sequence\_generator,

batches)

*# map metric names to metric values*

metrics = {metric\_name: metric\_value **for** metric\_name, metric\_value **in** zip(self.espa.metrics\_names, metrics)}

**print** "Accuracy: {0:>8.4f} | Loss: {1:>8.4f}".format(float(metrics["categorical\_accuracy"]),

float(metrics["loss"]))

**return** metrics

**def** generate\_data(self, data\_save\_file, data\_set, sample\_idxs, batch\_size):

*""" Generates data from HDF5 file on demand*

*"""*

**while** True:

*# determine batches*

sample\_count = len(sample\_idxs)

batches = int(sample\_count/batch\_size)

remainder\_samples = sample\_count%batch\_size

**if** remainder\_samples:

batches = batches + 1

*# generate batches*

**for** idx **in** xrange(batches):

*# incomplete batches*

**if** idx == batches - 1:

batch\_idxs = sample\_idxs[idx\*batch\_size:]

*# complete batches*

**else**:

batch\_idxs = sample\_idxs[idx\*batch\_size:idx\*batch\_size+batch\_size]

batch\_idxs = sorted(batch\_idxs)

X = data\_save\_file["\_".join(["x", data\_set])][batch\_idxs]

Y = data\_save\_file["\_".join(["y", data\_set])][batch\_idxs]

**yield** (np.array(X), np.array(Y))

**def** print\_espa\_summary(self):

*""" Prints a summary representation of the OSR model*

*"""*

**print** "**\n**\*\*\* MODEL SUMMARY \*\*\*"

self.espa.summary()

**def** generate\_espa\_model(self):

*""" Builds the ESPA model*

*"""*

**print** "**\n**Generating ESPA model..."

**with** h5py.File(self.data\_save\_fn, "r") **as** data\_save\_file:

class\_count = len(data\_save\_file.attrs["classes"].split(","))

*# input layer*

spectrograms = Input(shape=(8,

3,

self.freq\_points,

self.time\_points))

*# CNN layers*

cnn\_base = VGG16(input\_shape=(3,

self.freq\_points,

self.time\_points),

weights="imagenet",

include\_top=False)

cnn\_out = GlobalAveragePooling2D()(cnn\_base.output)

cnn = Model(input=cnn\_base.input, output=cnn\_out)

cnn.trainable = False

encoded\_spectrograms = TimeDistributed(cnn)(spectrograms)

*# RNN layers*

encoded\_spectrograms = LSTM(256)(encoded\_spectrograms)

*# MLP layers*

hidden\_layer = Dense(output\_dim=1024, activation="relu")(encoded\_spectrograms)

outputs = Dense(output\_dim=class\_count, activation="softmax")(hidden\_layer)

*# compile model*

espa = Model([spectrograms], outputs)

optimizer = Nadam(lr=0.0002,

beta\_1=0.9,

beta\_2=0.999,

epsilon=1e-08,

schedule\_decay=0.004)

espa.compile(loss="categorical\_crossentropy",

optimizer=optimizer,

metrics=["categorical\_accuracy"])

self.espa = espa

**def** save\_espa\_model(self):

*""" Save the ESPA model to an HDF5 file*

*"""*

*# delete save files, if they already exist*

**try**:

**print** "**\n**ESPA save file **\"**{0}**\"** already exists! Overwriting previous saved file.".format(self.espa\_save\_fn)

os.remove(self.espa\_save\_fn)

**except** **OSError**:

**pass**

**try**:

**print** "ESPA weights save file **\"**{0}**\"** already exists! Overwriting previous saved file.**\n**".format(self.espa\_weights\_save\_fn)

os.remove(self.espa\_weights\_save\_fn)

**except** **OSError**:

**pass**

*# save ESPA model*

**print** "**\n**Saving ESPA model to **\"**{0}**\"**".format(self.espa\_save\_fn)

**with** open(self.espa\_save\_fn, "w") **as** espa\_save\_file:

espa\_model\_json = self.espa.to\_json()

espa\_save\_file.write(espa\_model\_json)

*# save ESPA model weights*

**print** "Saving ESPA model weights to **\"**{0}**\"**".format(self.espa\_weights\_save\_fn)

self.espa.save\_weights(self.espa\_weights\_save\_fn)

**print** "Saved ESPA model and weights to disk**\n**"

**def** load\_espa\_model(self):

*""" Load the ESPA model from an HDF5 file*

*"""*

**print** "**\n**Loading ESPA model from **\"**{0}**\"**".format(self.espa\_save\_fn)

**with** open(self.espa\_save\_fn, "r") **as** espa\_save\_file:

espa\_model\_json = espa\_save\_file.read()

self.espa = model\_from\_json(espa\_model\_json)

**print** "Loading ESPA model weights from **\"**{0}**\"**".format(self.espa\_weights\_save\_fn)

**with** open(self.espa\_weights\_save\_fn, "r") **as** espa\_weights\_save\_file:

self.espa.load\_weights(self.espa\_weights\_save\_fn)

**print** "Loaded ESPA model and weights from disk**\n**"

**def** process\_data(self):

*""" Preprocesses data*

*"""*

**print** "**\n**Processing data..."

data\_dirs = sorted(["left-index-flexion",

"left-middle-flexion",

"left-ring-flexion"])

txt\_data\_dir = "data/txt"

csv\_data\_dir = "data/csv"

*# convert text data into CSV data*

**for** data\_dir **in** data\_dirs:

convert\_txt\_to\_csv(join(txt\_data\_dir, data\_dir),

join(csv\_data\_dir, data\_dir))

X = {channel\_idx:[] **for** channel\_idx **in** range(8)}

Y = []

data\_sample\_count = 0

*# iterate through all class directories*

**for** class\_idx, data\_dir **in** enumerate(data\_dirs):

class\_dir = join(csv\_data\_dir, data\_dir)

class\_files = [class\_file

**for** class\_file **in** listdir(class\_dir)

**if** (isfile(join(class\_dir, class\_file))) **and** (".csv" **in** class\_file)]

sys.stdout = open(os.devnull, "w") *# silence EEG data processing standard outputs*

*# iterate through all class files*

**for** class\_file **in** class\_files:

session\_title = " ".join(class\_file.split("-"))

eeg\_processor = EEGProcessor(class\_dir, class\_file, "openbci", session\_title)

eeg\_processor.plot = 'show'

eeg\_processor.load\_data()

*# iterate through all channels*

**for** channel\_idx, channel **in** enumerate(eeg\_processor.channels):

**print** " ".join(["Processing channel ",

str(channel\_idx + 1)])

*# load and clean channel data*

eeg\_processor.load\_channel(channel)

eeg\_processor.remove\_dc\_offset()

eeg\_processor.notch\_mains\_interference()

eeg\_processor.trim\_data(10, 10)

*# calculate spectrogram*

eeg\_processor.get\_spectrum\_data()

eeg\_processor.data = eeg\_processor.bandpass(1, 50)

spec = 10\*log10(eeg\_processor.spec\_PSDperBin)

*# accumulate sample data*

sample\_count = spec.shape[1] / self.time\_points

**for** sample\_idx **in** xrange(sample\_count):

sample = spec[0:self.freq\_points,

sample\_idx\*self.time\_points:

sample\_idx\*self.time\_points+self.time\_points]

format\_spec = **lambda** spectrogram: np.array([spectrogram]\*3)

X[channel\_idx].append(format\_spec(sample))

*# accumulate label data*

data\_sample\_count += sample\_count

label = [0]\*len(data\_dirs)

label[class\_idx] = 1

label = np.array(label)

Y.extend([label]\*(sample\_count))

sys.stdout = sys.\_\_stdout\_\_ *# stop silencing standard outputs*

*# format sample and label data*

X = [np.array([cha\_1, cha\_2, cha\_3, cha\_4, cha\_5, cha\_6, cha\_7, cha\_8])

**for** cha\_1, cha\_2, cha\_3, cha\_4, cha\_5, cha\_6, cha\_7, cha\_8

**in** zip(X[0], X[1], X[2], X[3],

X[4], X[5], X[6], X[7])]

X = np.array(X)

Y = np.array(Y)

*# save sample and label data into HDF5 file*

**with** h5py.File(self.data\_save\_fn, "w") **as** data\_save\_file:

*# partition data into training, validation, and testing sets*

sample\_idxs = np.random.permutation(range(data\_sample\_count))

training\_sample\_idxs = sample\_idxs[0:int((1.0-self.validation\_ratio-self.testing\_ratio)\*data\_sample\_count)]

validation\_sample\_idxs = sample\_idxs[int((1.0-self.validation\_ratio-self.testing\_ratio)\*data\_sample\_count):int((1.0-self.testing\_ratio)\*data\_sample\_count)]

testing\_sample\_idxs = sample\_idxs[int((1.0-self.testing\_ratio)\*data\_sample\_count):]

x\_train, y\_train = self.replicate\_augment\_data(X[training\_sample\_idxs], Y[training\_sample\_idxs])

x\_val, y\_val = self.replicate\_augment\_data(X[validation\_sample\_idxs], Y[validation\_sample\_idxs])

x\_test, y\_test = self.replicate\_augment\_data(X[testing\_sample\_idxs], Y[testing\_sample\_idxs])

data\_save\_file.attrs["classes"] = np.string\_(",".join(data\_dirs))

data\_save\_file.attrs["training\_sample\_count"] = len(x\_train)

data\_save\_file.attrs["validation\_sample\_count"] = len(x\_val)

data\_save\_file.attrs["testing\_sample\_count"] = len(x\_test)

*# training set*

x\_train\_ds = data\_save\_file.create\_dataset("x\_train",

shape=(len(x\_train), 8, 3, self.freq\_points, self.time\_points),

dtype="f")

y\_train\_ds = data\_save\_file.create\_dataset("y\_train",

shape=(len(y\_train), len(data\_dirs)),

dtype="i")

x\_train\_ds[:] = x\_train

y\_train\_ds[:] = y\_train

*# validation set*

x\_val\_ds = data\_save\_file.create\_dataset("x\_val",

shape=(len(x\_val), 8, 3, self.freq\_points, self.time\_points),

dtype="f")

y\_val\_ds = data\_save\_file.create\_dataset("y\_val",

shape=(len(y\_val), len(data\_dirs)),

dtype="i")

x\_val\_ds[:] = x\_val

y\_val\_ds[:] = y\_val

*# testing set*

x\_test\_ds = data\_save\_file.create\_dataset("x\_test",

shape=(len(x\_test), 8, 3, self.freq\_points, self.time\_points),

dtype="f")

y\_test\_ds = data\_save\_file.create\_dataset("y\_test",

shape=(len(y\_test), len(data\_dirs)),

dtype="i")

x\_test\_ds[:] = x\_test

y\_test\_ds[:] = y\_test

**def** replicate\_augment\_data(self, X\_h, Y\_h):

*""" Replicates/augments sample data*

*"""*

*# samples after replication/augmentation*

X = []

Y = []

*# spectrogram formatting function*

format\_spec = **lambda** spectrogram: np.array([spectrogram]\*3)

*# replicate/augment samples*

**for** x\_h, y\_h **in** zip(X\_h, Y\_h):

*# increase sample data via image augmentation*

**if** self.augmentation:

**for** \_ **in** xrange(self.samples\_generated\_per\_sample):

x = [] *# new sample*

**for** spectrogram **in** x\_h:

spectrogram = spectrogram[0] *# first colour channel only*

shifted\_sample = random\_shift(np.array([spectrogram]),

wrg = self.augmentation\_magnitude,

hrg = self.augmentation\_magnitude)

x.append(format\_spec(shifted\_sample[0]))

X.append(np.array(x))

Y.append(y\_h)

*# increase sample data via image replication*

**else**:

**for** \_ **in** xrange(self.samples\_generated\_per\_sample):

X.append(x\_h)

Y.append(y\_h)

*# replicated/augmented samples*

**return** np.array(X), np.array(Y)

**class** **ProgressDisplay**(Callback):

*""" Progress display callback*

*"""*

**def** on\_batch\_end(self, epoch, logs={}):

**print** " Batch {0:<4d} => Accuracy: {1:>8.4f} | Loss: {2:>8.4f} | Size: {3:>4d}".format(int(logs["batch"])+1,

float(logs["categorical\_accuracy"]),

float(logs["loss"]),

int(logs["size"]))

*# auxilliary functions*

**def** get\_training\_configuration(training\_config\_fn):

*""" Acquires training configuration from a file*

*"""*

**with** open(training\_config\_fn, "r") **as** training\_config\_file:

training\_config = load(training\_config\_file)

**return** training\_config

**def** execute\_training\_runs(training\_config):

*""" Executes training runs from specified training configuration*

*"""*

results = {}

*# iterate through training runs*

**for** training\_run **in** training\_config:

run\_name = training\_run["run\_name"]

trials = training\_run["trials"]

data\_save\_fn = training\_run["data\_save\_fn"]

validation\_ratio = training\_run["validation\_ratio"]

testing\_ratio = training\_run["testing\_ratio"]

samples\_generated\_per\_sample = training\_run["samples\_generated\_per\_sample"]

augmentation = training\_run["augmentation"]

augmentation\_magnitude = training\_run["augmentation\_magnitude"]

freq\_points = training\_run["freq\_points"]

time\_points = training\_run["time\_points"]

espa\_save\_fn = training\_run["espa\_save\_fn"]

espa\_weights\_save\_fn = training\_run["espa\_weights\_save\_fn"]

results[run\_name] = {}

**print** "**\n**".join(["="\*80,

"EXECUTING TRAINING RUN: {0}".format(run\_name),

"="\*80])

*# iterate through trials*

**for** trial **in** xrange(trials):

**print** "**\n**".join(["-"\*80,

"TRIAL: {0}".format(trial),

"-"\*80])

espa = ESPAModel(data\_save\_fn = data\_save\_fn,

validation\_ratio = validation\_ratio,

testing\_ratio = testing\_ratio,

samples\_generated\_per\_sample = samples\_generated\_per\_sample,

augmentation = augmentation,

augmentation\_magnitude = augmentation\_magnitude,

freq\_points = freq\_points,

time\_points = time\_points,

espa\_save\_fn = espa\_save\_fn,

espa\_weights\_save\_fn = espa\_weights\_save\_fn)

espa.process\_data()

espa.generate\_espa\_model()

espa.print\_espa\_summary()

training\_metrics\_history = espa.train\_espa\_model()

testing\_metrics = espa.test\_espa\_model()

espa.save\_espa\_model()

*# organize results*

results[run\_name][trial] = {}

results[run\_name][trial]["train"] = training\_metrics\_history.history

results[run\_name][trial]["test"] = testing\_metrics

**return** results

**def** save\_results(results, results\_save\_fn):

*""" Save results to a JSON file*

*"""*

**with** open(results\_save\_fn, "w") **as** results\_save\_file:

dump(results, results\_save\_file)

**if** \_\_name\_\_ == "\_\_main\_\_":

training\_config = get\_training\_configuration("training\_config.json")

results = execute\_training\_runs(training\_config)

save\_results(results, "results/results.json")