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

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

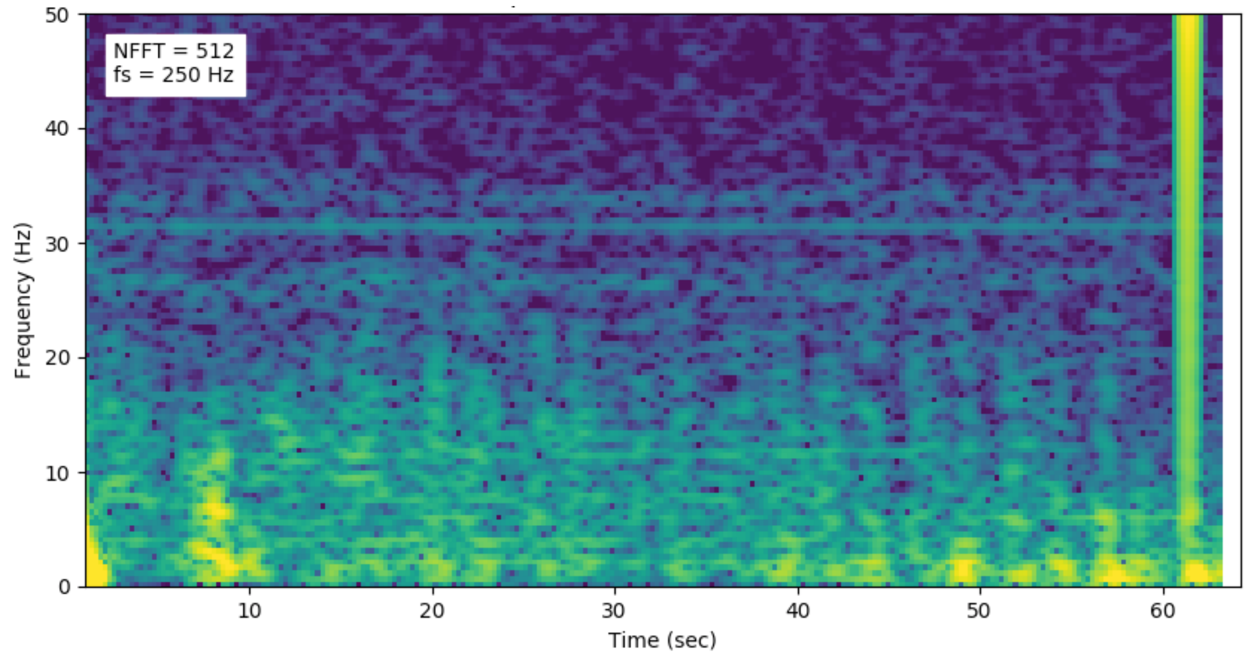


Figure 1 Sample spectrogram from 1 channel for sustained left middle finger flexion

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

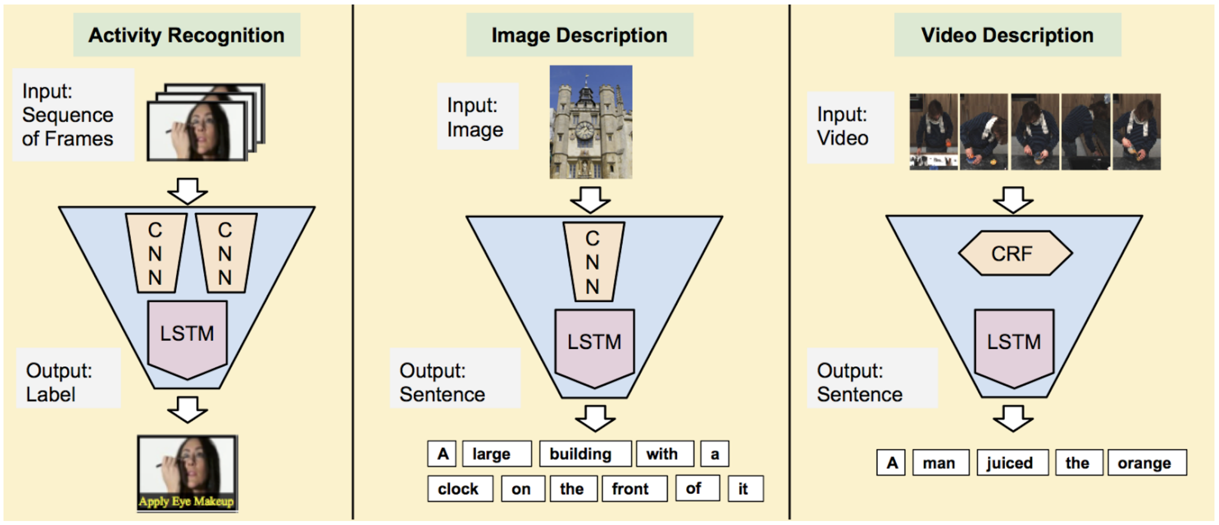


Figure 2 Applications of LRC neural networks (Source: http://jeffdonahue.com/lrcn/)

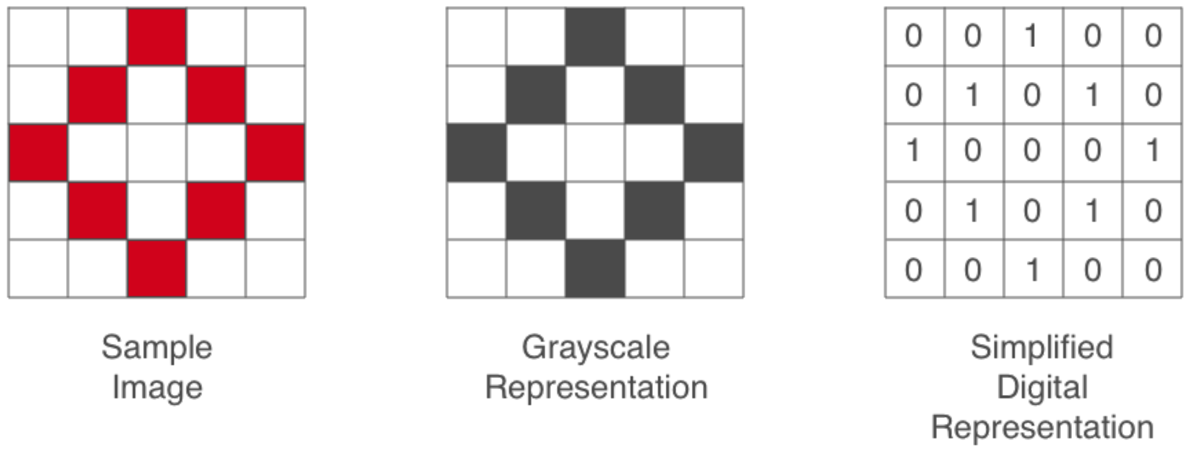
The model is based on long-term recurrent convolutional (LRC) neural networks, a class of neural networks used for visual and sequence learning [1]. It consists of a hybrid architecture of convolutional neural networks, recurrent neural networks, and multilayer perceptrons.

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]:

$$
\begin{aligned}
(f * g)[n] &\stackrel{\text{def}}{=} \sum_{m=-\infty}^{\infty} f[m]g[n-m] \\
&= \sum_{m=-\infty}^{\infty} f[n-m]g[m]
\end{aligned}
$$

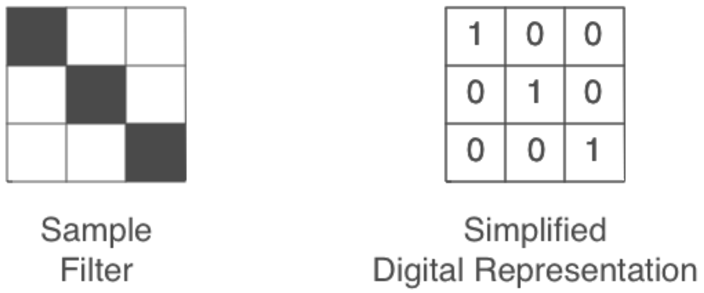
To illustrate how CNNs work, a sample 5x5 image, its grayscale conversion, and its simplified digital representation, where 1 is the maximum value instead of 255, is shown below:



0	0	1	0	0
0	1	0	1	0
1	0	0	0	1
0	1	0	1	0
0	0	1	0	0

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:



1	0	0
0	1	0
0	0	1

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:

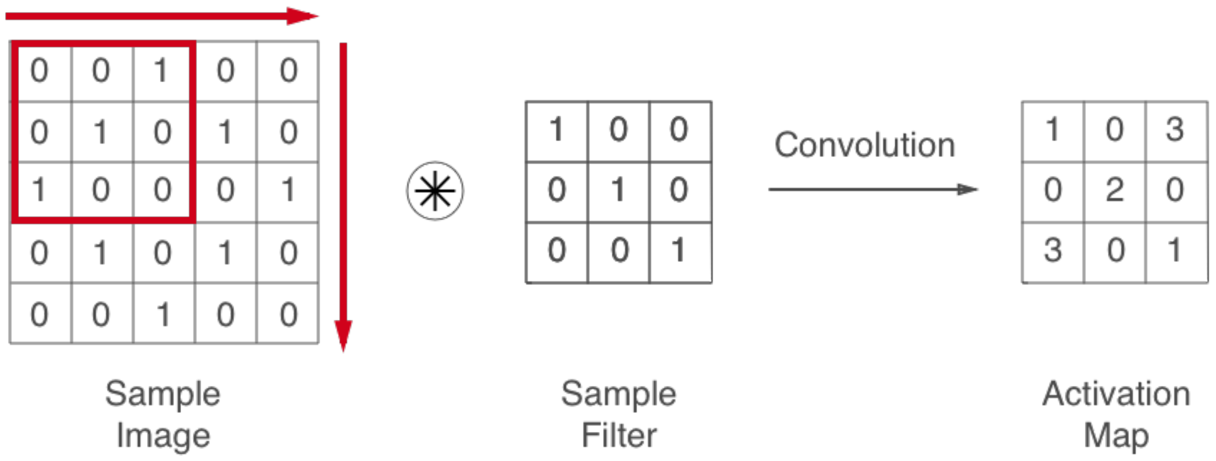


Figure 5 Generation of an activation map by tiling a filter across an image and performing successive convolutions

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

$$f(x) = \max(0, x)$$

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:

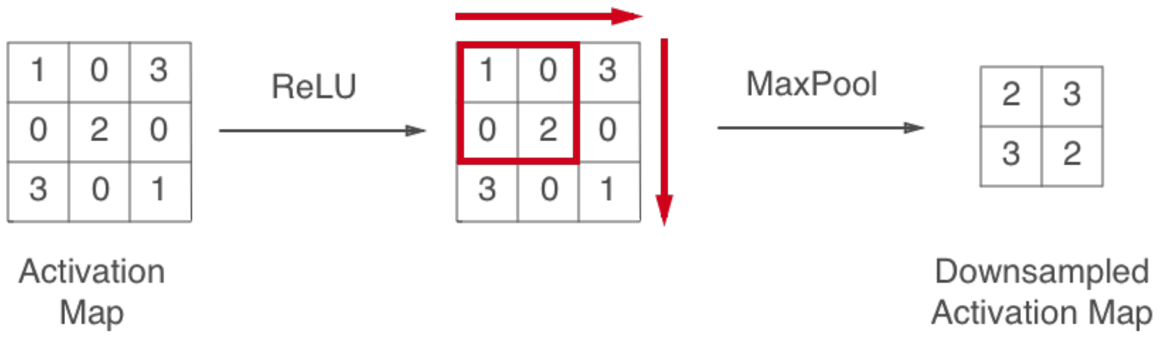


Figure 6 Application of a ReLU activation layer and a MaxPool pooling layer to an activation map

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:

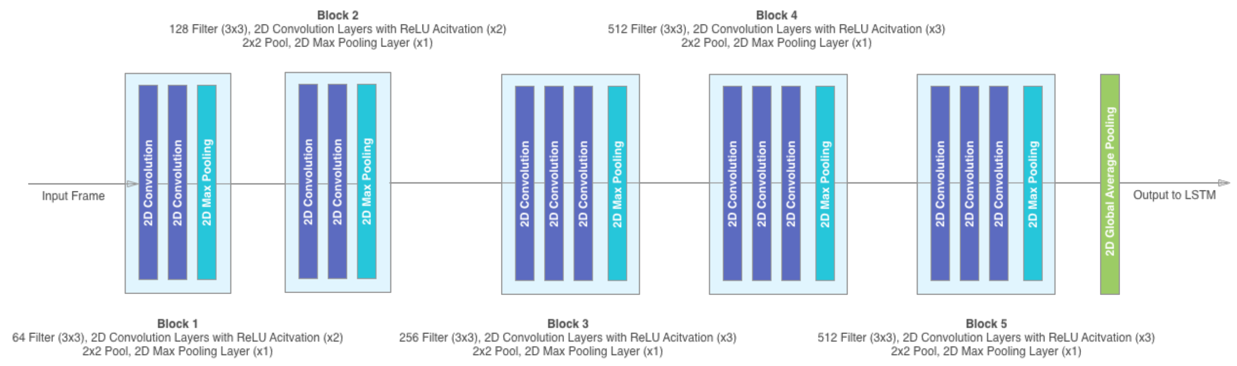

Block 2
128 Filter (3x3), 2D Convolution Layers with ReLU Acitvation (x2)
2x2 Pool, 2D Max Pooling Layer (x1)
Block 4
512 Filter (3x3), 2D Convolution Layers with ReLU Acitvation (x3)
2x2 Pool, 2D Max Pooling Layer (x1)
Input Frame
2D Convolution
2D Max Pooling
2D Global Average Pooling
Output to LSTM
Block 1
64 Filter (3x3), 2D Convolution Layers with ReLU Acitvation (x2)
2x2 Pool, 2D Max Pooling Layer (x1)
Block 3
256 Filter (3x3), 2D Convolution Layers with ReLU Acitvation (x3)
2x2 Pool, 2D Max Pooling Layer (x1)
Block 5
512 Filter (3x3), 2D Convolution Layers with ReLU Acitvation (x3)
2x2 Pool, 2D Max Pooling Layer (x1)

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:

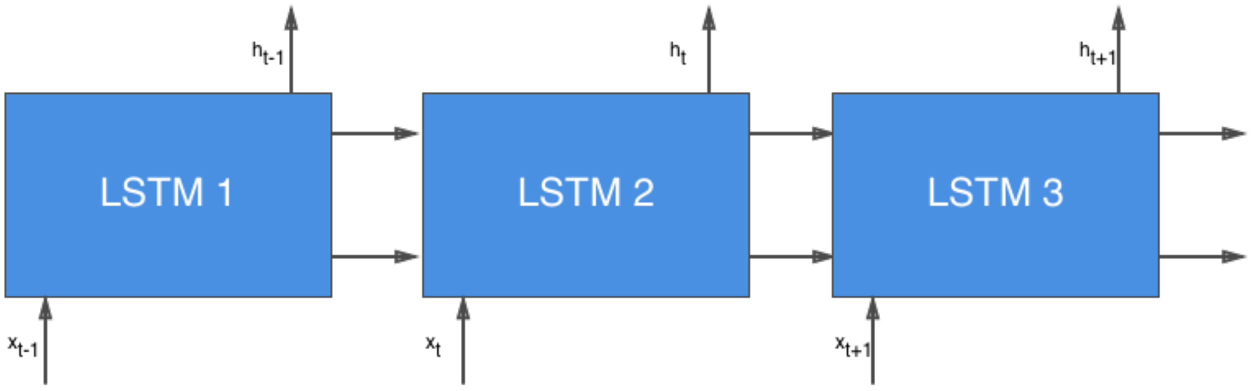


Figure 8 Chain of repeating LSTM units

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

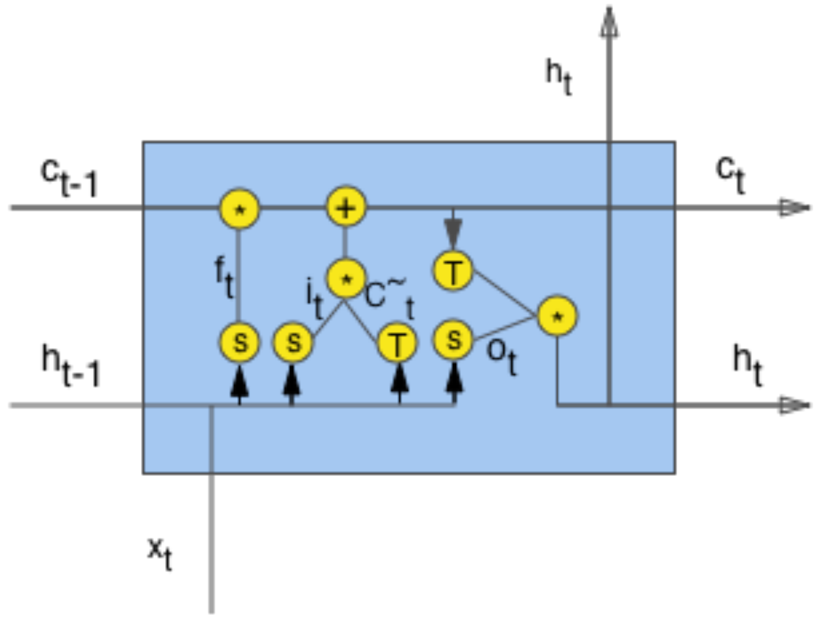


Figure 9 LSTM operations

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

$$S(t) = \frac{1}{1 + e^{-t}}$$

T represents a hyperbolic tangent operation:

$$T(t) = \frac{e^{2t} - 1}{e^{2t} + 1}$$

+ 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:

$$i_t = S(W_i x_t + U_i h_{t-1} + b_i)$$

and a hyperbolic tangent layer that generates new candidate values:

$$C_t^{\sim} = T(W_c x_t + U_c h_{t-1} + b_c)$$

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

$$f_t = S(W_f x_t + U_f h_{t-1} + b_f)$$

The layer takes in the input, x_t, and the previous output, h_{t-1}, then returns either 0 or 1 for each value in the cell state C_{t-1}, where 0 represents "forget" and 1 represents "remember". Then, the LSTM unit updates the cell state from the old state C_{t-1} to the new state C_t:

$$C_t = i_t C_t^{\sim} + f_t C_{t-1}$$

This operation forgets what has to be forgotten by multiplying the old cell state C_{t-1} with the output of the forget gate f_t, and adds new candidate values scaled by update weights by multiplying the new cell state C_t with the output of the input gate i_t. Finally, the LSTM unit generates the output. First, a logistic sigmoid layer selects which values of the cell state to output:

$$o_t = S(W_o x_t + U_o h_{t-1} + V_o C_t + b_o)$$

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:

$$h_t = o_t T(C_t)$$

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:

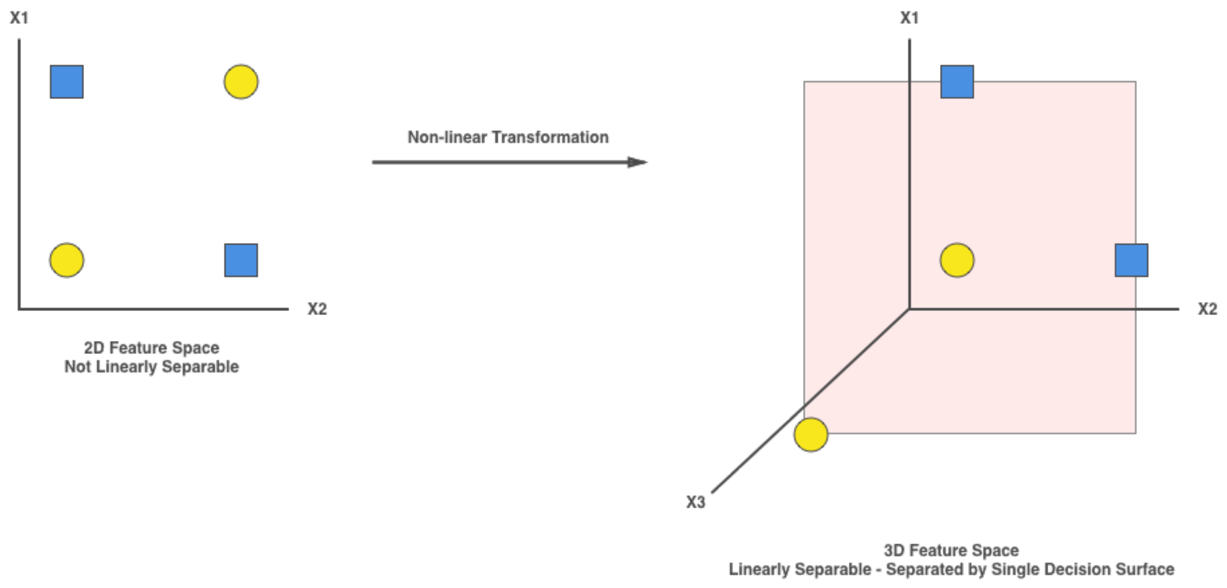

X1
X2
2D Feature Space
Not Linearly Separable
Non-linear Transformation
X1
X2
X3
3D Feature Space
Linearly Separable - Separated by Single Decision Surface

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:

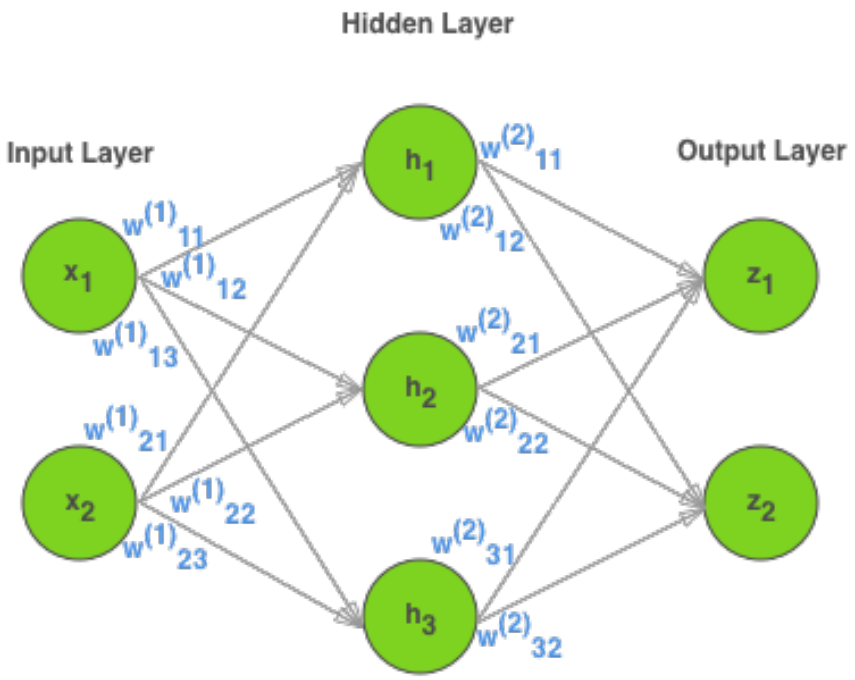


Figure 11 Sample MLP with a 2-node input layer, 3-node hidden layer, and 2-node output layer.

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]:

$$h(x) = s(b^{(1)} + w^{(1)}x)$$
$$z\big(h(x)\big) = G(b^{(2)} + w^{(2)}h(x))$$

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:

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}, j = 1, \dots, K$$

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

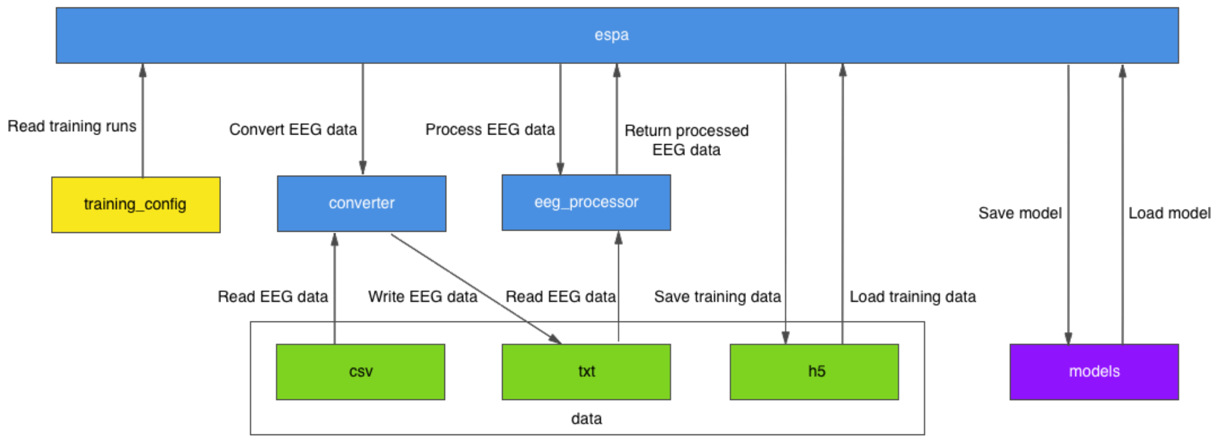


Figure 12 General system architecture

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

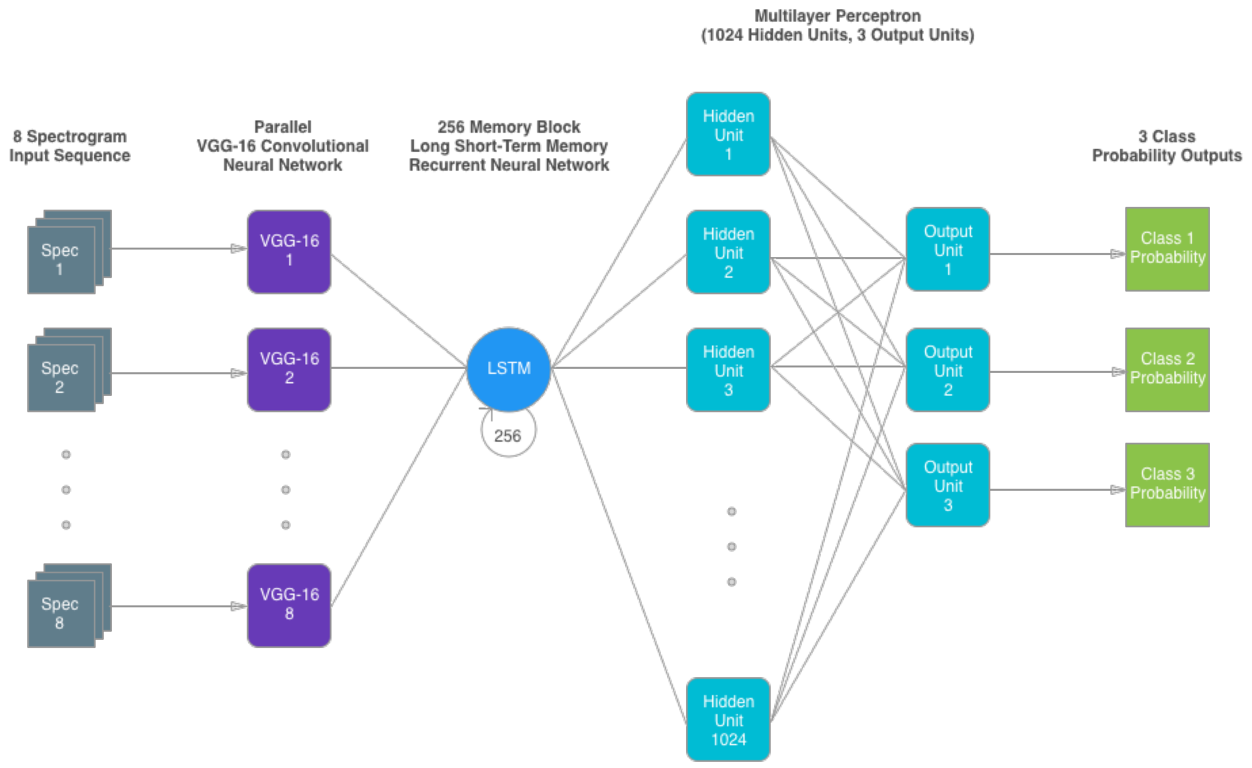


Figure 13 ESPA model architecture

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

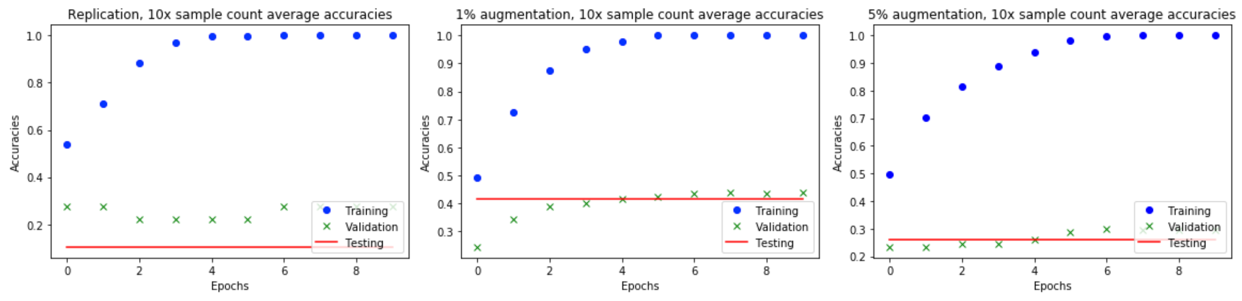


Figure 14 Accuracies (average training, validation, and testing) across all three setups

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; Jürgen Schmidhuber (1997). "Long short-term memory". Neural Computation. 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 "\nTraining 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_s
ample_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 "\nTesting 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_sa
mple_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 "\nGenerating 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 "\nESPA 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 "\nSaving 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 "\nLoading 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 "\nProcessing 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")
```