Imperial College London
Department of Computing

Learning from Biosignals

Akara Supratak

Submitted in partial fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing Research of Imperial College London
May 2018

Abstract

A long-standing goal of the biosignal analysis is to develop tools that can continuously collect quality biosignals, and algorithms to extract meaningful information from them. This can lead to a better understanding of our health that allows us to adjust our daily activity to be suitable for our well-being, and provide treatments promptly. However, the conventional approaches to analyze biosignals rely on the development of algorithms to extract features from the signals (i.e., hand-engineering features). It can be labor-intensive and time-consuming to develop algorithms to extract such features for particular applications repeatedly. Also, the existing machine learning based systems assume that there are annotations associated with the particular patterns of biosignals; such annotations are very expensive to obtain.

In this thesis, our objective is to develop models that can automatically learn features from biosignals without utilizing any hand-engineering features and can extract meaningful information from biosignals even when the labels of biosignals are not available. Our first contribution is that we propose a method that can remotely and accurately estimate walking speeds for people with walking impairments. This work also motivates us to use deep learning to automate the expensive feature engineering process in the remaining contributions. Our second contribution is that we propose a model that can automatically extract meaningful features from raw scalp EEG signals for epileptic seizure detection. We demonstrate that it can detect seizures without utilizing any seizure annotations. Our third contribution is that we propose a model that can automatically learn features that are useful for sleep stage scoring from raw single-channel EEG data, and achieved similar performance compared to the state-of-the-art hand-engineering ones. We also demonstrated that our model can generalize to two sleep datasets that have different properties without any modifications to the model architecture. Our final contribution is that we propose a novel data-driven approach that employs a signal transcription model to capture the relationships between signals from multiple domains to detect desynchronized biosignals as anomalies without utilizing any annotations.

Copyright Declaration

The copyright of this thesis rests with the author and is made available under a Creative Commons Attribution Non-Commercial No Derivatives license. Researchers are free to copy, distribute or transmit the thesis on the condition that they attribute it, that they do not use it for commercial purposes and that they do not alter, transform or build upon it. For any reuse or redistribution, researchers must make clear to others the license terms of this work

Declaration of Originality

I hereby declare that the contents in this thesis are the works of my own, and are written by myself. All of the previous works mentioned in this thesis have been properly referenced and listed in the bibliography.

Acknowledgements

I would like to take this opportunity to express my thanks to all of those who have always been by my side and supported me from various aspects.

Firstly, I must thank my supervisor, Yike Guo, who taught and advised me through out my PhD. Without his support and professional guidance, this thesis would not be possible. I am also very pleased to Yike Guo for teaching me how to think critically and the art of communicating ideas.

I would like to thank Douglas G. Mcilwraith, Paul M. Matthews, and Ling Li who help me revise the contents of this thesis.

I would like to thank my close collaborators at Data Science Institute, Imperial College London who I have had the distinct pleasure of working with and learning from. This especially includes Hao Dong who has always encouraged me and supported me when I was in trouble, Axel Oehmichen who has supported me to run my experiments, and David T. E. Akroyd who has helped me with the hardware issues.

I have to thank many people who have contributed to my day-to-day life and who have made my experience at Imperial College London and in UK so pleasant.

I am thankful to the Faculty of Information and Communication Technology, Mahidol University for funding my MSc and PhD studies at Imperial College London.

Finally, I would like to thank my family, especially my dad who always guide me to become a better person.

Dedication

To my dearest mom and dad.

Life can only be understood backwards; but it must be lived forwards.

– Søren Kierkegaard

Contents

Abstract **i**

Acknowledgements **vii**

1 Introduction **1**

1.1 Motivation and Challenges 1

1.2 Related Works 2

1.3 Objectives 3

1.4 Contributions and Thesis Organization 4

1.5 Publications 7

2 Background **11**

2.1 Biosignals 11

2.1.1 Electroencephalogram 11

2.1.2 Electrocardiogram 13

2.1.3 Electrooculogram 14

2.1.4 Electromyogram 15

2.1.5 Gait Acceleration 15

2.2 Supervised Learning 16

2.2.1 Linear Regression 17
2.2.2 Linear Classification 17
2.2.3 Support Vector Machine 18
2.2.4 Support Vector Regression 20
2.3 Unsupervised Learning 21
2.3.1 k-means Clustering 22
2.3.2 Principal Component Analysis 22
2.3.3 Anomaly Detection 24
2.4 Gradient Descent 25
2.5 Backpropagation 26
2.6 Regularization 28
2.7 Hyperparameter Selection 30
2.8 Evaluation 31
2.8.1 Binary Classification 31
2.8.2 Multi-class Classification 33
2.9 Deep Learning 33
2.9.1 Loss Functions 35
2.9.2 Activation Functions 38
2.9.3 Fully-connected Neural Networks 40
2.9.4 Convolutional Neural Networks 41
2.9.5 Recurrent Neural Networks 51
2.9.6 Autoencoders 61
2.9.7 Residual Learning Framework 64

2.10 Generative Models 66

2.10.1 Generative Adversarial Nets 66

2.10.2 Conditional Generative Adversarial Nets 68

2.11 Practical Methodologies 69

3 Remote Gait Monitoring for People with Multiple Sclerosis 73

3.1 Introduction 73

3.2 Related Works 75

3.3 Methods 76

3.3.1 Participants 76

3.3.2 Actigraphy Devices 77

3.3.3 Data Collection 78

3.3.4 Subject-specific Gait Calibration 80

3.3.5 Walking Speed Estimation 83

3.3.6 Data Analysis 84

3.4 Results 85

3.4.1 Sample Characteristics 85

3.4.2 Baseline Methods 86

3.4.3 Walking Speed Estimation Performance 86

3.4.4 Home Monitoring 86

3.4.5 Device Comfort 88

3.4.6 Comparison on Step Features 89

3.5 Discussion 90

3.6 Conclusion 93

4 Epileptic Seizure Detection **95**

4.1 Introduction . . . 95

4.2 Related Works . . . 97

4.3 Possibility of Stacked Sparse Autoencoders . . . 98

4.3.1 Patient-specific Seizure Detector . . . 99

4.3.2 Two-step Training Algorithm . . . 99

4.3.3 Generating Seizure Alarms . . . 101

4.3.4 Preliminary Results . . . 101

4.3.5 Discussion . . . 107

4.4 Patient-Specific Seizure Detection System . . . 109

4.4.1 Anomaly Detector . . . 109

4.4.2 Anomaly Classifier . . . 112

4.4.3 Generating Seizure Alarms . . . 114

4.4.4 Preliminary Results . . . 117

4.4.5 Discussion . . . 127

4.5 Conclusion . . . 129

5 Automatic Sleep Stage Scoring **131**

5.1 Introduction . . . 132

5.2 DeepSleepNet . . . 134

5.2.1 Representation Learning . . . 134

5.2.2 Sequence Residual Learning . . . 138

5.2.3 Sleep Stage Classification . . . 140

5.2.4 Model Specification 141

5.3 Two-Step Training Algorithm 142

5.3.1 Pre-training 142

5.3.2 Fine-tuning 142

5.3.3 Loss Function 143

5.3.4 Regularization 144

5.4 Results 145

5.4.1 Data 145

5.4.2 Experimental Design 146

5.4.3 Performance Metrics 147

5.4.4 Tests for Statistical Significance 147

5.4.5 Training Parameters 149

5.4.6 Implementation 150

5.4.7 Initial Experiments 150

5.4.8 Sleep Stage Scoring Performance 151

5.4.9 Comparison with State-of-the-Art Approaches 152

5.4.10 Sequence Residual Learning 155

5.4.11 Model Analysis 155

5.5 Discussion 163

5.6 Conclusion 165

6 Towards Desynchronization Detection **167**

6.1 Introduction 168

6.2 Related Works . . . 169
6.3 Method . . . 172
6.3.1 Cross-domain signal transcription . . . 172
6.3.2 Desynchronization Detection . . . 175
6.4 Results . . . 177
6.4.1 Dataset . . . 177
6.4.2 Experimental Design . . . 177
6.4.3 Network Architecture and Training Parameter . . . 178
6.4.4 Performance Metrics . . . 179
6.4.5 Baseline Methods . . . 180
6.4.6 Cross-domain Signal Transcription Performance . . . 180
6.4.7 Desynchronization Detection Performance . . . 181
6.5 Discussion . . . 189
6.6 Conclusions . . . 190

7 Conclusions **193**

8 Future Works **197**

Bibliography **199**

List of Tables

2.1 Typical EEG subbands and their clinically relevant activity 13

2.2 Confusion matrix for binary classification problem. 32

2.3 Performance metrics for binary classification. 32

3.1 29 features derived from each step data. 82

3.2 Ranges of hyperparameters of the SVR model supplied during model calibration for each subject. 83

3.3 Comparison between the walking speed estimation performance of the baseline (**BASE**) and our subject-specific calibration (**PROPOSED**) approaches on the VALID data. The values in the table are mean (standard deviation). **Measured** indicates the walking speed manually measured by the researcher during the data collection. **Estimated** indicates the walking speed automatically estimated by each approach. **Error** refers to the difference in walking speed between the measured and estimated speed (i.e., Estimated - Measured). **Low**, **Moderate** and **High** refers to the MS volunteers with EDSS scores of 1.0-3.5, 4.0-5.5 and 6.0 respectively. 87

3.4 MS subject demographics, their average walking speeds measured during Timed 25-Foot Walk with normal voluntarily-set speeds in the clinic (T25FW-NORMAL), and their average of the maximum sustained walking speeds estimated daily from the home environment (MAX-SUS-HOME). 91

4.1 Summary of the randomly selected six patients from the CHB-MIT dataset. There are 39 seizure in total. 103

4.2 Summary of the sensitivity and latency of our seizure detector when we varied the channel threshold (i.e., Γ in Equation 4.3). The total number of seizures are 39. . . . 108

4.3 Summary of the randomly selected nine patients from the CHB-MIT dataset. There are 67 seizures in total. 119

4.4 Performance of our seizure detection system in the unsupervised setting when ξ=0.1, γ_a=15, Γ_a=15 and T_d=30 121

4.5 Performance of our seizure detection system in the supervised setting when ξ=0.1, γ_a=10, Γ_a=15, T_d=1, γ_s=3, Γ_s=7 and T_w=5 125

4.6 A general comparison between the proposed system and the existing seiuzre onset detection systems across sensitivity, false detection rate and mean detection latency. . . 128

5.1 Number of 30-s EEG epochs for each sleep stage from two datasets 146

5.2 The mean (standard deviation) of number of training and test samples in each cross-validation from two datasets 146

5.3 Confusion matrix obtained from 31-fold cross-validation on F4-EOG (Left) channel from the MASS dataset. PR, RE and F1 are per-class precision, per-class recall and per-class F1-score respectively. 151

5.4 Confusion matrix obtained from 10-fold cross-validation on Fpz-Cz channel from the Sleep-EDF dataset. PR, RE and F1 are per-class precision, per-class recall and per-class F1-score respectively. 151

5.5 The mean (standard deviation) of performance metrics obtained from repeatedly performing the 31-fold and the 10-fold cross-validation on the randomly shuffled lists of subjects for 15 times on the MASS and Sleep-EDF dataset. The per-class precision (PR) and recall (RE) are not shown here as the per-class F1-score (F1) is a summary of the precision and recall using a harmonic mean. 153

5.6 Comparison between DeepSleepNet and other sleep stage scoring methods that utilizes hand-engineering features across overall accuracy (ACC), macro-F1 score (MF1), Cohen's Kappa (κ), and Per-class F1-Score (F1). We only compare the scoring performance with the methods that evaluated using the independent training and test sets. The bootstrap 95% confidence intervals (CIs) of the performance metrics are calculated for the Fpz-Cz channel from the Sleep-EDF dataset and the F4-EOG (Left) channel from the MASS dataset. The top and bottom numbers in parentheses are the lower and upper bounds of the CIs respectively. 154

5.7 Confusion matrix obtained from 31-fold cross-validation on the F4-EOG (Left) channel from the MASS dataset using DeepSleepNet without Sequence Residual Learning . . . 156

6.1 Summary of the cross-domain signal transcription performance in terms of $c1$, $c2$, $c3$ and $c4$. Each row shows the average $\pm$ standard deviation of each metric for each subject. The last row shows the overall performance from all subjects. 182

6.2 Comparison between our proposed and the baseline methods across degree of synchrony of each class, detection accuracy (ACC), F1-score (F1), and area under ROC curves (AUC) 184

6.3 Confusion matrices obtained by applying BASE-1, BASE-2 and our proposed method on simultaneous recordings from all test subjects. Each row and column represent the number of 30-s epochs of the no-eye and the eye classes, classified by the criteria mentioned in Section 6.4.2 and by each method respectively. The numbers in bold indicate the number of epochs that were correctly classified. 184

List of Figures

2.1 Scalp electrode positioning in the 10-20 International System [1]. 12

2.2 An example of 15-s EEG epoch from 20 channels. 12

2.3 An example of ECG of a heart in normal sinus rhythm (NSR) [2]. 14

2.4 An example of EOG generated by horizontal eye movements [3]. A pair of electrodes are placed at the outer side of the left and right eyes. 15

2.5 An example of raw 3D acceleration data collected from a tri-axial accelerometer attached around the waist and close to the center of mass. This accelerometer measures three signals simultaneously, which are vertical (red), horizontal (green) and forward (blue) acceleration. Each peak of *forward* acceleration preceding a change of the sign from positive to negative can be thought of as an instance of a foot-contact [4]. 16

2.6 An example of backpropagation along a computational graph [5]. During forward pass, vectors x and y are forwarded into a node that 1) compute vector z using some fixed function such as $z = x \odot y$, where $\odot$ is an element-wise multiplication; and 2) compute its local gradients (or Jacobian matrices) $\frac{\partial z}{\partial x}$ and $\frac{\partial z}{\partial y}$. The vector z is then forwarded to the later node in the graph. At the end of the computational graph, the total loss J (a scalar value) is computed. During backward pass, the global gradient on $\frac{\partial J}{\partial z}$ is multiplied with the local gradients, resulting in the global gradients for x ($\frac{\partial J}{\partial x} = \frac{\partial z}{\partial x}\frac{\partial J}{\partial z}$) and y ($\frac{\partial J}{\partial y} = \frac{\partial z}{\partial y}\frac{\partial J}{\partial z}$). These gradients are multiplied (or "chained"), in turn, through functions that compute x and y until the inputs have been reached. 28

2.7 Dropout Neural Network Model [6]. **Left**: a standard (or base) neural network with 2 hidden layers. **Right**: an example of a sub-network sampled from the base network on the left. Dropout trains an ensemble consisting of an exponential number of different sub-networks that can be produced by temporarily removing neurons from an underlying base network (Crossed neurons have been temporarily dropped). 30

2.8 An example of a comparison between the area under two ROC curves [7]. In this figure, the classifier B has a larger AUC than the classifier A, which means B achieves a better average classification performance. True positive rate and false positive rate are the recall and false positive rates in Table 2.3, respectively. The bottom-left point (0,0) means a classifier never predicts the positive classes (i.e., no false positive and no true positive). The top-right point (1,1) means a classifier always predicts the positive classes. The top-left point (0,1) represents the perfect classification. 33

2.9 An example of (a) a cartoon drawing of a biological neuron and (b) its mathematical model [8]. An input signal travels along the axons (e.g., x_0), which is then interacted with dendrites of the other neuron (e.g., $w_0 x_0$) based on the synaptic strength. There are synapses (e.g., w_0) between the axon and the dendrite that control the strength of influence of one neuron to another. The dendrites carry the weighted input signals to the cell body. In the biological model, the cell body accumulates the results and sums them. If the sum is greater than a certain threshold, the neuron can fire which sends a spike along its axon. In mathematical model, the cell body accumulates the results, sums them, and then applies an element-wise function σ to fire an output signal ($\sigma\left(\sum_i w_i x_i + b\right)$, where i is the index of the input and weight vector) via an output axon. If the sigmoid activation is used, the output signal can be interpreted as the average firing rate of the neuron. Finally, the axon propagates the signals along its branches to the dendrites of other neurons. 34

2.10 ReLU activation function outputs zero when $z_i < 0$ and linear with slope 1 when $z_i > 0$ [8]. 38

2.11 Sigmoid activation function squashes z_i to range between 0 and 1 [8]. 40

2.12 Tanh activation function squashes z_i to range between -1 and 1 [8]. 40

2.13 An example of a 3-layer fully-connected neural network (only count the number of hidden and output layers). There are one input layers of neurons (or units), two hidden layers of four neurons each, and one output layers of two neurons. 41

2.14 Local vs Full Connectivity. In (a), **s** is computed by convolution with a filter of size (or width) 3. For example, only three input x_2, x_3 and x_4 (green circle) affects the output s_3 (blue circle), and these units are known as the **receptive field** of s_3. In (b), **s** is computed by matrix multiplication and all inputs affect s_3 44

2.15 The receptive field of the units at the deeper convolutional layers is larger than the units at the shallow layers. In this example, even though each convolutional layer only connect to 3 input units, the output unit g_3 may *indirectly* connect all or most of the input units **x**. 44

2.16 An example of convolving a 3×5 input tensor with a 2×2 filter. Each element of the feature map is obtained by computing a dot product between the filter and the corresponding local part of the input tensor and then adding the result with a bias. The filter is slid with a stride size of one across all positions that the filter can lie entirely within the input tensor (i.e., no zero-padding). This result in the feature map of size 2×4. 45

2.17 Convolution with zero-padding and strides of different sizes. In this example, an input vector of size 5 is padded with one zeros on both size and is convolved with a filter of size 3. **Left:** Convolution with a stride size of 1 results in an output vector of size 5, which is equal to the size of the input vector. **Right:** Convolution with a stride of size 2 results in an output vector of size 3. This can be seen as we are skipping some positions of the filter to reduce the spatial size of the output tensor (i.e., *downsampling*) and the computational cost at the expense of not finely extracting features. 47

2.18 An example of convolving a 256×1×2 filter over a 30-s epoch of EEG from two channels with the sampling rate of 256 Hz (i.e., a 7680×1×2 input tensor). Using a padding of size 64 on the border of the input and a stride of size 128, there are 60 spatial positions, each of which computes a dot product between the filter and the local part of the input followed by adding the result with a bias, resulting in a feature map of size 60×1×1. As there is more than one filter in the convolutional layer (e.g., 32 filters), each of them will independently perform convolution in the same way, resulting in a set of feature maps stacked along the depth (e.g., a 60×1×32 output tensor). 49

2.19 Max pooling layer with a pool width of 3 and a stride size of 2. This pooling layer downsamples an input vector of size 6 into an output vector of size 3 using the maximum operation. Even though the rightmost pooling has a size smaller than the pool width, we have to include this so that we do not ignore some elements at the border of the input vector. 50

2.20 Unfolding a computation graph of a recurrent neural network with no outputs. This RNN incorporates the information from the input $\mathbf{x}^{(t)}$ into the previous state $\mathbf{h}^{(t-1)}$, producing the current state $\mathbf{h}^{(t)}$ that is pass forward through time. **Left:** a circuit diagram, where the red and the green boxes represent the input and the hidden vectors, respectively, and the black box indicates an interaction with a delay of a single time step. **Right:** an unfolded computation graph (representing the same network), where each column is associated with one particular time step. Typically, RNNs will have output layers that read information from the state $\mathbf{h}$ to make predictions, which will be discussed later in this section. 52

2.21 Comparison among a feedforward neural network and different RNN patterns [8]. Rectangles denote vectors and arrows denote function dependencies. **From left to right**: **(1)** a feedforward neural network that accepts an input vector (blue), transforms it to an output vector (blue) using one hidden layer (green) (e.g., image classification), **(2)** a RNN that accepts an input vector and generates a sequence of output vectors (e.g., image captioning), **(3)** a RNN that accepts a sequence of input vectors and generates an output vector (e.g., sentiment analysis), **(4)** a RNN that accepts a variable-length sequence of input vectors and generates a variable-length sequence of output vectors noting that the length of the input and output sequences can be different (e.g., machine translation), and **(5)** a RNN that accepts a sequence of input vectors and *simultaneously* generates an output vector for each input (e.g., sleep stage classification which annotates each signal epoch based on the previous sequences). Note that the RNNs **(2)**-**(5)** utilize a *fixed* recurrent hidden layer (green) to summarize the information from the previous input vectors, which enables us to process any length of the input sequences. 53

2.22 A computational graph to compute the training loss of an RNN that accepts a sequence of input vectors $\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(\tau)}\}$ and simultaneously generates an output vector for each input $\{\mathbf{o}^{(1)}, \ldots, \mathbf{o}^{(\tau)}\}$ (i.e., many-to-many design pattern). The input-to-hidden and hidden-to-output connections are parameterized by the weights, which are $\mathbf{W}_{xh}$ and $\mathbf{W}_{hy}$ respectively. The hidden-to-hidden recurrent connections are parameterized by the weights $\mathbf{W}_{hh}$. These weights are the same across time steps. A loss L quantifies the disagreement between the output $\mathbf{o}$ and the corresponding training target $\mathbf{y}$. For instance, the loss can be the cross-entropy loss that treats the output $\mathbf{o}$ as the unnormalized log probabilities. It internally computes $\hat{\mathbf{y}} = \text{softmax}(\mathbf{o})$ and compare it with the target $\mathbf{y}$. The forward pass of this RNN is defined in Equation 2.45. **Left:** The computational graph of RNN drawn with recurrent connections. **Right:** The time-unfolded computational graph of RNN, where each column represents the computation of one time step. 54

2.23 A LSTM model. LSTM maintains two states: cell and hidden states at every time step. The cell state is an internal memory and is not exposed to the other layers in the network. The hidden state, on the other hand, can be revealed to the other layers. For each time step t, LSTM computes gate values: $\mathbf{f}^{(t)}$, $\mathbf{i}^{(t)}$, $\mathbf{g}^{(t)}$ and $\mathbf{o}^{(t)}$ using the current input $\mathbf{x}^{(t)}$ and the previous hidden state $\mathbf{h}^{(t-1)}$. The $\mathbf{f}^{(t)}$, $\mathbf{i}^{(t)}$ and $\mathbf{g}^{(t)}$ are used to update the current cell state $\mathbf{c}^{(t)}$ as following. The forget gate $\mathbf{f}^{(t)}$ decides which information of the previous cell state to be forgotten. The input gate $\mathbf{i}^{(t)}$ decides which new information from the candidate $\mathbf{g}^{(t)}$ to be stored in the cell state. The output gate $\mathbf{o}^{(t)}$ controls which parts of the cell state $\mathbf{c}^{(t)}$, squashed by tanh, to be exposed as the current hidden state $\mathbf{h}^{(t)}$. Any output layers can be stacked on top of the LSTM model to read information from the hidden state $\mathbf{h}$ to make predictions. 60

2.24 A computational graph to compute the training loss of a bidirectional RNN that accepts a sequence of inputs $\mathbf{x}$ and produce a sequence of outputs $\mathbf{o}$. A loss L measures the disagreement between the output $\mathbf{o}$ and the corresponding training target $\mathbf{y}$. The same sequences of inputs are fed to both RNNs, each of which summarizes the information in different directions. The forward RNN $\mathbf{h}$ processes the input sequences forward in time from left to right, while the backward RNN $\mathbf{g}$ process the input sequences backward in time from right to left. So, at each time step t, the bidirectional can utilize the summaries of both the past $\mathbf{h}^{(t)}$ and the future $\mathbf{g}^{(t)}$ that is most sensitive to the input values around t to determine the output $\mathbf{o}^{(t)}$. 62

2.25 Degradation problem [9]. The training (left) and test (right) error on the CIFAR-10 dataset from a 20-layer *plain* network are lower than a 56-layer network. Intuitively, by making the network deeper, the performance should not decrease as the added layers can learn an identity function, and the others are copied from the shallower network (i.e., a 20-layer network). 65

2.26 Comparison between the plain and the residual networks. The plain network directly transforms an input vector x with two non-linear layers (using ReLU). The residual network introduces a shortcut connection to explicitly let the two layers learn a residual function (i.e., learn a function to modify the input vector). The output of the residual function is added (element-wise) to the input vector, followed by a ReLU function. . . 65

2.27 Image-to-image Translation [10]. **Left**: U-net trains the generator to synthesize the output image that corresponds to the input image in another domain such as from a map image to an aerial image. **Right**: Several examples of translated images: from semantic label to photo, day to night and sketch to photo. 69

3.1 Actigraphic Device. We use the AX3-Axivity tri-axial accelerometer protected with silicone backing to remotely and continuously collect data from each subject. 77

3.2 An example of raw 3D acceleration data collected from the accelerometer consisting of vertical (red), horizontal (green) and forward (blue) acceleration. 78

3.3 Preprocessing. Tilt-adjustment algorithm [11] is applied to each walking data to transform from the arbitrarily tilted accelerometer into a relevance sagittal plane (or a horizontal-vertical orthogonal plane). Then the gravitational force (i.e., 1 g) is removed from the vertical acceleration. After that, a low-pass filter with a fourth-order zero-lag Butterworth filter at 20 Hz is applied on each walking data. This is to remove noise and artifact at the higher frequency that are not related to the gait or walking patterns. 80

3.4 Step extraction. The forward acceleration is first low-pass-filtered with a fourth-order zero-lag Butterworth filter at 4 Hz to smooth the data. A peak of the filtered acceleration preceding a change of the sign from positive to negative are used to identify an instance of a foot-contacts [4]. The acceleration data between two foot-contacts are extract as step data. 81

3.5 Examples of extracting step duration and maximum features from step data. 82

3.6 Comparison of correlations between manually measured walking speeds for the full group of patients and either that estimated from actigraphy data using **(a)** the BASE model (R-value=0.85, p-value=4.74×10^{-10}) or **(b)** the PROPOSED model (R-value=0.98, p-value=1.85×10^{-22}). The white-filled, grey-filled circles, and black-filled circles denote MS subjects from low (EDSS=1.0-3.5), moderate (EDSS=4.0-5.5) and high (EDSS=6.0) disability subgroups. 87

3.7 Plots of differences between the measured and estimated walking speeds relative to **(a)** their means (or Bland-Altman plots) and relative to **(b)** EDSS scores. The top graphs are the results from the BASE model, while the bottom ones are from the PROPOSED model. Greater estimate error is evident for patients with higher disability (i.e., slower speeds) with the BASE model. 88

3.8 Examples of estimated walking speeds (top) and the number of steps (bottom) from two days of home data from one MS subject. The walking speeds and the number of steps are estimated from sustained walks. The grey areas indicate the lay-down periods (including sleeps). The maximum sustained walking speeds of Day-1 and Day-2 are 1.46 and 1.40 m/s, respectively. 89

3.9 Correlations between the average of the maximum sustained walking speeds estimated from home environment (**MAX-SUS-HOME**) and the average walking speeds measured during T25FW with normal voluntarily-set speeds in the clinics (**T25FW-NORMAL**) across subjects (R-value=0.89, p-value=4.34×10^{-8}). These results suggest good agreement across the range of disability. 89

3.10 Testing for systemic bias in agreement between the MAX-SUS-HOME and the T25FW-NORMAL speeds. First, we assessed the difference between the MAX-SUS-HOME speeds and the T25FW-NORMAL speeds using a Bland-Altman plot **(a)**. The differences did not appear to change with walking speeds. Second, we tested variation of such differences relative to EDSS scores **(b)**. No significant change was seen, consistent with clinical meaningfulness of the EDSS score measure across the disability range. . 90

3.11 Scatter plots of the first and second PCA components of step features. These features are extracted from several normal (a) and slow (b) walking data during T25FW and 2MIN of randomly chosen MS and healthy subjects. 92

4.1 Patient-specific seizure detector. It receives a 1-s EEG epoch as an input and outputs the probability of such input being of the ictal class. It consists of two sparse-autoencoder layers (SSAEs) and a binary logistic classifier. Each **sparse-autoencoder** layer uses a fully-connected layer as an encoder and a decoder function (see more in Section 2.9.6), consisting of 500 hidden units with a sigmoid activation function. The *encoder* function is used during both the training and the detection as a kind of a feature extraction function. The *decoder* function, on the other hand, is only used during the training process and is discarded during the seizure detection. The **logistic** layer is a fully-connected layer consisting of one hidden unit with a sigmoid function that outputs the probability that the input is of the ictal class. For each input, if the output probability is greater than a specified thresholded (e.g., 0.5), the model outputs an ictal label. Otherwise, it outputs an interictal label. The output classes is then post-processed to generate a seizure alarm. 100

4.2 Compute the number of seizure channels for generating seizure alarms. The trained seizure detector is used to determine whether the input 1-s EEG epochs from k EEG channels $\mathbf{x}^{(i,j)}$ are of the ictal ($\hat{y}^{(i,j)} = 1$) or the interictal ($\hat{y}^{(i,j)} = 0$) class (see Equation 4.2). The number of seizure channel at the i-th EEG epoch $s^{(i)}$ is computed by summing the output labels at the i-th EEG epoch from k EEG channels (see Equation 4.3). We refer to this process as the *post-processing*. A seizure alarm is generated when $s^{(i)}$ is greater than a channel threshold Γ. In this example, we highlight the computation of the number of seizure channels at the 3806-th EEG epoch, which is 4. 102

4.3 An example of a seizure pattern in scalp EEG signals from 23 channels from chb01. The red dash-line indicates the seizure onset at 3801 seconds. 105

4.4 An example of a seizure pattern in scalp EEG signals from 23 channels from chb03. The red dash-line indicates the seizure onset at 2996 seconds. 105

4.5 An example of the scalp EEG signals of chb01 (top) associated with the number of seizure channels (bottom) determined by our seizure detector. The red dash line indicates the seizure onset. The number of the representative channels is five and their channel indices are 2, 3, 19, 20 and 21. 107

4.6 False detection rates per hour (i.e., specificity) for three channel thresholds across six patients. 108

4.7 Patient-specific seizure detection system, consisting of two main components: anomaly detector and anomaly classifier. The **anomaly detector** detects anomalies in EEG signals when patient-specific seizure data are not available. It consists of three sparse autoencoders trained to reconstruct its input 1-s EEG epoch. It annotates the input 1-s EEG epoch as anomalous when the difference between the input and the reconstructed is more than a threshold. Such difference is measured by the root-mean-squared error (*rms-error*). The outputs from the anomaly detector are anomaly annotations. The **anomaly classifier** determines anomaly classes or types of EEG signals. This anomaly classifier is an extended version of our previous seizure detection in Section 4.3.1 to support multi-class classification. Specifically, we replace the logistic classifier with the *softmax* classifier. The number of hidden units of the softmax layer, c, depends on the number of classes we would like to classify. The outputs from the anomaly classifier are anomaly classes. **Unsupervised setting:** when the seizure labels are not available, only the anomaly detector is functional. So only the sequence of anomaly annotations is post-processed to generate a seizure alarm. **Supervised setting:** when the seizure labels (and other class labels) are available, these labeled data can be used to train the anomaly classifier. Thus *both* the anomaly detector and classifier are functional. The sequence of anomaly annotations and anomaly classes are post-processed to generate a seizure alarm. 110

4.8 Examples of the original (blue) and its reconstructed (red) 1-s EEG epochs from the SSAEs in the anomaly detector. The reconstruction of the interictal epoch (a) is similar to the original epoch. In contrast, the the reconstruction of the ictal epoch (b) is different from the original epoch. 112

4.9 Examples of the anomaly channel computed from 1-s EEG epochs during a seizure (start at 2382 s). The top graphs show raw signals from 23 EEG channels. The bottom graphs show the value of anomaly channel computed using Equation 4.9. The black dash line indicates the seizure period. 115

4.10 Examples of the anomaly channel and the seizure channel computed from EEGs during ictal (a) and interictal (b) periods. The top graphs show raw EEGs from multiple channels. The middle graphs show the value of anomaly channel (see Equation 4.9). The bottom graphs show the type degrees indicating the number of channels annotated with non-interictal types such as seizures. The type degrees of the seizures is the same the seizure channel mentioned in Equation 4.11. 117

4.11 An example of EEG data from Patient F during a seizure (between 2817 s and 2839 s), and the corresponding anomaly channel determined by the seizure detector. 120

4.12 ROC curves for different hyperparameters in the unsupervised seizure detection setting. These graphs show the changes in sensitivity versus false detection rate according to the changes of each hyperparameter. 122

4.13 Mean detection latency according to the change of each hyperparameter in the unsupervised seizure detection setting. 123

4.14 Examples of the anomaly channel and the seizure channel computed from EEGs during ictal (a) and interictal (b) periods. The top graphs show raw EEGs from multiple channels. The middle graphs show the anomaly channel. The bottom graphs show the type degrees indicating the number of channels annotated with non-interictal types such as seizures. The type A (seizure) is the same the seizure channel. The type B (extra) is the class that we manually annotated. 125

4.15 ROC curves for different hyperparameters in the supervised seizure detection setting. These graphs show the changes in sensitivity versus false detection rate according to the changes of each hyperparameter. 126

4.16 Mean detection latency according to the change of each hyperparameter in the unsupervised seizure detection setting. 126

5.1 An overview architecture of DeepSleepNet consisting of two main parts: **representation learning** and **sequence residual learning**. The representation learning part consists of two CNNs that have the first convolutional layers with small (CNN-small, left-side) and large (CNN-large, right-side) filter sizes. The specifications of the first convolutional layers depend on the sampling rate (Fs) of the EEG data (see Section 5.2.4). Each rectangular box represents one layer in the model. **conv** is a convolutional layer [*filter_size* conv, *n_filters*, /*stride_size*]. **max-pool** is a max-pooling layer [*pool_size* max-pool, /*stride_size*]. **dropout** is a dropout layer [*probability* dropout]. The sequence residual learning part consists of bidirectional-LSTMs and a shortcut connection. **bidirect-lstm** is a bidirectional LSTM layer [*forward_hidden_units*/*backward_hidden_units* bidirect-lstm]. The **shortcut connection** in the sequence residual learning that branches off from the output of the two CNNs reformulates this part into a residual function. **fc** is a fully-connected layer [*hidden_units* fc]. The arrows indicate the flow of data from raw single-channel 30-s EEG epochs to sleep stages, where the softmax layer, **softmax** [*n_classes* softmax] is used to compute the output probability of each sleep stage. Each trainable layer is a layer containing parameters to be optimized during a training process, while each non-trainable layer only performs its operation without any parameters to be optimized. 135

5.2 The 1D-convolution operation to extract time-invariant features. For simplicity, we will use the 1D-convolution operation performed at the first convolutional layers of the two CNNs, where an input is a raw single-channel 30-s EEG epoch (**blue signal**), as an example. Each filter (i.e., the weights of each neuron $s_1, \ldots, s_n$ in the convolutional layer) (**red signal**) is convolved with the EEG epoch. Each rectangular box highlights the computation of one value in a feature map (**magenta signal**), which is a dot product between a filter and a chunk of the EEG epoch. The dash-line rectangular box shows how each filter is slid across the EEG epoch during the convolution (i.e., stride). The resulting feature maps are concatenated before forwarding to the next layer. The 1D-convolution operations at the downstream layers are the same as in this example, except that there will be more than one input channel, depending on the number of feature maps. In this case, the filters of such convolutional layers will also have the same number of channels as the inputs allowing the computation of one value in a feature map to cover all input channels. As each filter is slid across the EEG epoch, each convolutional layer can learn a set of filters to detect certain patterns in the EEG epoch, regardless of when such patterns appear (i.e., **time-invariant**), via backpropagation. 136

5.3 **bidirect-lstm** layer. **Left:** A part of our sequence residual learning in the DeepSleepNet model that has two bidirectional-LSTM layers. **Right:** The unfolded version of the bidirectional-LSTM with no outputs. Each bidirectional-LSTM consists of two LSTMs that process the input sequence in the forward (**LSTM-forward**) and backward (**LSTM-backward**) directions. Each LSTM maintains its own cell state ($\mathbf{c}_f^{(t)}$ and $\mathbf{c}_b^{(t)}$) and hidden state ($\mathbf{h}_f^{(t)}$ and $\mathbf{h}_b^{(t)}$) at every time step t to encode the lossy summary of the input sequence. This allows the bidirectional-LSTM to exploit the summaries of both the past and the future that are most sensitive to the input value around each time step to determine the sleep stage. The input $\mathbf{a}^{(t)}$ is the output from the representation learning part (see Equation 5.3). 139

5.4 Examples of the hypnogram manually scored by a sleep expert (top) and the hypnogram automatically scored by DeepSleepNet (bottom) for Subject-1 from the MASS dataset. 153

5.5 Examples of the average of the sum of the filter activations from the first **conv** layers of the CNN-small and CNN-large obtained by feeding our model with data from 3 subjects. The filter activations from the small filters are on the left (a), and the larger filters are on the right (b). Each image has 5 rows and 64 columns, corresponding to 5 sleep stages and 64 filters respectively. Each pixel represents the scaled value of $u_{c,k}$ from Equation 5.15, where 1 (i.e., active) is white and 0 (i.e., inactive) is black. Each row corresponds to the 64-dimension vector (i.e., k is 64) for each sleep stage c. The first row is from stage W, and the last row is from stage REM. Each image also has labels indicating which filters are most active for which sleep stages. It can be seen that, for instance, some filters were most active for stage N2-N3, some were most active for stage W-N1-REM, and some were either most active or inactive at all stages. . . . 157

5.6 Outputs of 1D-convolution using the filters from the first **conv** layers from the CNN-small and CNN-large. Here, we illustrate the outputs (magenta signals) from convolving the raw input 30-s EEG epochs from sleep stage N3 and REM (blue signals) with the corresponding filters (red signals) (see Figure 5.2 for more details). These EEG epochs from the F4-EOG (Left) channel from one subject in the MASS dataset. We found that most of the small and large filters were mixed frequency signals, and some of them are similar. Also, some of the small filters exhibit either an increasing or decreasing trends, which produce opposite trend of output signals. 158

5.7 Interpretable LSTM cells. These two plots show the same sequence of sleep stages (capital characters), which are the the predictions from our model on one subject data, arranged through time from-left-to-right and from-top-to-bottom. Each stage in the sequence is colored in background using the corresponding value of one node (i.e., one LSTM cell) from $tanh(\mathbf{c})$, where +1 is blue and -1 is red. In **(a)**, this LSTM cell mostly activates (blue color) when our model predicts stage W or N1. In **(b)**, this cell turns off or becomes inactive (red color) when our model outputs a long sequence of REM stage (i.e., R in the plot). This suggests that the LSTMs inside the sequence residual learning part learn to keep track of the current status of a subject. 161

5.8 A large portion of cells from LSTMs, that process the input sequences in the forward (a) and backward (b) directions, are not easily interpretable. Note that these LSTM cells are from the same subject as in Figure 5.7. 162

6.1 Comparison between the traditional and our approaches. The traditional approaches (top) measure the degree of synchrony directly from these three signals. Specifically, they compute the degree of synchrony among all possible pairs of signals. Then degrees of synchrony from all pairs are summarized (e.g., computing the average) into a single value. In contrast, our approach (bottom) first transcribes input signals into their expected, simultaneous signal. The degree of synchrony is then estimated by computing the similarity between the real and the transcribed signals. 171

6.2 An overview of the architecture of our cross-domain signal transcription model and its training framework. The encoder-generator network G transcribes signals from the source domain S to its simultaneous signals in the target domain T, $G : \mathbf{x}_s^{(i)} \rightarrow \hat{\mathbf{x}}_t^{(i)}$. It consists of an encoder, a residual cross-domain transfer, and a generator. The encoder is a CNN that encodes signals from S into a feature representation. The residual cross-domain transfer is a residual network that transfers from the feature representation in the source domain to the representation in the target domain. The generator is a deconvolutional network that generates or synthesizes expected, simultaneous signals in T. The discriminator network D is a CNN that determines whether the input simultaneous signals from both domains are real or fake, $D : (\mathbf{x}_s^{(i)}, \mathbf{x}_t^{(i)} | \hat{\mathbf{x}}_t^{(i)} | \mathbf{x}_w^{(i)}) \rightarrow \{0, 1\}$. L_D, L_G and L_R (highlighted in red) are losses used during the training. Note that the D network is only used during training to provide useful gradients for synthesizing the expected, simultaneous signals, and is not used during the signal transcription. 173

6.3 Examples of EEG signals (the last row) synthesized from two EOG signals using the transcription model. The blue signals refers to the signals from EOG domain, while the red signals refers to the signals from EEG domain. As the transcription model is trained with the signals that do not contain eye-movements, the synthesized signal in (a) is more similar to the real EEG-Fz signal (the third row) compared to the synthesized one in (b). This is because the EOG signals from the right contain eye-movements which cannot be explained by the model. 176

6.4 Examples of the EEG-Fz signals synthesized from the EOG signals from the no-eye class from Subject B that have low r-correlations. The r-correlation annotated at the top-right of the last row refers to the Pearson's R coefficient between the real (the third row) and the synthesized EEG-Fz (the last row). 183

6.5 Histograms of the degree of synchrony from the no-eye (red) and the eye (blue) epochs quantified using BASE-1 (top), BASE-2 (middle) and our method (bottom) 185

6.6 ROC curves of BASE-1, BASE-2 and our proposed method obtained by varying ε from -1 to 1 for BASE-1 and the propose method, and from 0 to 1 for BASE-2. Each bold circle marker indicates the threshold that gives TPR and FPR closest to the perfect classification point (i.e., TPR=1 and FPR=0) in terms of the Euclidean distance. . . 186

6.7 Examples of the EEG-Fz signals synthesized from the EOG signals from the no-eye class from Subject A. 187

6.8 Examples of the EEG-Fz signals synthesized from the EOG signals from the eye class from Subject A. 188

6.9 Examples of the spectrograms from the real and the synthesized EEG-Fz signals from the no-eye epoch in Figure 6.7a (a) and from the eye epoch in Figure 6.8a (b). It can be seen that the changing trends of the synchronized signals coincide, while the trends of the desynchronized signals are different. 191

1

Introduction

"If you sharpen your electrical sense to generators in the body,
If you listen to body sounds emerging from depths of the body,
If you look through a fragment of the body,
If you feel the skin pulsation of the body,
You are to gain a valuable knowledge of the bodys well-being ..."

– Eugenijus Kaniusas, *Biomedical Signals and Sensors I (2012)*

1.1 Motivation and Challenges

For centuries, there have been a number of studies trying to develop methods to understand the current status of our health. A better understanding of our health allows us to adjust our daily activity to be suitable for our well-being and can lead to well-informed diagnosis and effective treatments. This can save us an enormous amount of time, money and resources from repeatedly visiting clinics or hospitals for additional diagnosis and treatments.

One of the most common methods to assess our well-being is through the analysis of what is called biomedical signals (or *biosignals*). Biosignals are signals or time-series data recorded from the human body, which contain information about physiological phenomena which reflect human health and well-being [12]. They have become an important indicator for medical diagnosis, subsequent therapy, and passive health monitoring. Examples of biosignals are brain activity, heart rate, blood pressure,

eye movements, and gait acceleration. Thus it is essential to develop tools or systems for analyzing biosignals that can assess the current status of our health as close to the actual one possible. This can lead us to a better understanding of the physiological status of individual so that appropriate treatments can be provided promptly.

The typical pipeline of the system for biosignal analysis consists of three phases: *preprocessing*, *feature extraction*, and *model construction*. Firstly, biosignals are preprocessed to remove noise, unwanted components or artifacts from signals such as signal filtering [13]. This is to ease the subsequent feature extraction process. Secondly, features that are meaningful to a certain problem are extracted or derived from the preprocessed signals. The algorithms to extract these features are usually hand-engineered by domain-specific experts who know which features are useful for particular problems. Finally, machine learning algorithms are employed to train and construct models that understand relationships between input (i.e., extracted features) and their desired output (i.e., labels or annotations), such as Support Vector Machine (SVM) or Neural Networks (NNs). Depending on the domain for which these models are trained, the trained models are then employed to, for example, monitor walking speeds at home [14, 15], classify the human's brain stages [16, 17], or detect harmful symptoms [18, 19].

Challenges. One of the main challenges in the biosignals analysis is to develop a way to *learn* or extract information about our well-being from biosignals (i.e., feature extraction). This is mainly because the measurement of biosignals is usually hindered by other biological activities, noise, and interference [20]. For example, in the brain activity recorded with electroencephalography (EEG) signals from people with epilepsy, they may contain seizure and the brain's background activities simultaneously. The seizure activity can also be further altered by artifacts such as eye blinks or body movements.

1.2 Related Works

Most of the existing works have developed algorithms to extract different features from biosignals. For instance, in epileptic seizure detection, the discrete wavelet transform (DWT), or the short-time Fourier transform (STFT) have been employed to decompose EEG signals into EEG subbands [18, 19, 21]. Similarly, in sleep stage scoring, the DWT, STFT, Hilbert-Huang Transform (HHT) and complex wavelet convolution have also been applied to extract the frequency components that correspond different sleep stages [22–25] defined by the sleep manuals [26, 27]. The output values of these subbands

are then used to compute, for instance, statistical features such as mean, maximum, energy and entropy. The derived features and their corresponding labels (e.g., seizures or sleep stages) are used to build a model using machine learning algorithms such as linear classifier, SVM or NNs that can distinguish between different classes.

There are other techniques developed by experts to study the relationship between simultaneous biosignals such as cross-correlation [28], coherence [29], mutual information [30], mean phase coherence (MPC) [31], and intrinsic phase synchrony [32]. For example, MPC has been applied to measure phase synchrony between iEEG subbands, and changes in the phase synchrony from the selected iEEG subbands were used to predict epileptic seizures [33].

As different feature extraction techniques are suitable for different applications (i.e., application-specific), we will further discuss the related works on feature extraction techniques in each chapter. Nevertheless, most of them are similar to the techniques mentioned above. They first decompose signals into frequency bands of interest and then compute features that are useful for different applications.

Remaining Challenges. However, these hand-engineering[1] features can be labor-intensive and time-consuming, as it requires *prior knowledge* from the experts to develop algorithms to extract features. Also, such algorithms may not generalize well to a larger population due to the heterogeneity among subjects and recording hardware. This is because the algorithms have commonly been developed based on the characteristics of the available datasets. It would be better if the model can automatically learn to extract such useful features from biosignals. Moreover, most of these works require annotations or labels associated with the particular patterns of biosignals (e.g., sleep stages or seizure periods) to train or construct the model. These labels, however, are very expensive to obtain. It would require domain-specific experts to manually go through the signals, and annotate their corresponding labels or classes. It would be better if we develop a model that can still perform such tasks without relying on the annotations from human experts.

1.3 Objectives

In this thesis, we aim to develop methods can achieve two main objectives as follows:

[1] In this thesis, the terms *feature engineering* or *hand-engineering features* are used to refer to the process of developing algorithms by experts to extract features from biosignals.

- **Automate feature engineering process.** Our first objective is to fully utilize deep learning to develop a model that can automatically learn features that are useful for particular tasks from biosignals. In other words, we would like to build a model that can be trained "end-to-end". This is different from the traditional approaches consisting of many stages of processing including preprocessing, feature extraction and model construction. With the end-to-end training, we train a deep learning model to perform the entire analysis process from the input biosignals to the outputs by only specifying the objective of the training (e.g., classifying sleep stages). The hidden layers of this model can be thought of as they are trained to perform multiple stages of processing, in which their parameters are jointly optimized over the same objective, instead of being separately optimized in each stage of processing. We believe that it would be better if the model can learn not only to perform given tasks, but also to extract useful features for such tasks.

- **Unsupervised learning.** Our second objective is to build a model that can learn to extract meaningful information from raw biosignals even when the labels of biosignals are not available. An example of knowledge of that we can learn from unlabeled biosignals is anomalies. We believe that it would be beneficial to the clinical research if we can extract some knowledge from biosignals without utilizing any labels that are very expensive to acquire.

1.4 Contributions and Thesis Organization

In this thesis, we develop models that can automatically learn features from biosignals without utilizing any hand-engineering features and propose a novel cross-domain signals analysis approach that can extract meaningful information from biosignals that have not been annotated. In particular, we develop neural network architectures and the ways to train them end-to-end efficiently, and demonstrate their usefulness on epileptic seizure detection and sleep stage scoring. This thesis is organized as follows:

- In **Chapter 2**, we discuss the basic and applications of different types of biosignals. We provide relevant background for supervised and unsupervised learning, optimization algorithms, and deep learning. We also summarize a typical procedure for applying deep learning to biosignal applications.

- In **Chapter 3**, we propose a method for subject-specific gait calibration using the actigraphy data from wearable, tri-axial accelerometers for automated estimation of walking speed for people with Multiple Sclerosis (MS). We develop algorithms to extract features that are useful for walking speed estimation from acceleration data. These features and their corresponding walking speed annotated in the control environment are used to calibrate a subject-specific model. The calibrated model can be used to transform from features extracted from acceleration data into walking speeds. *Lessons learned*: we found the drawbacks of the conventional hand-engineering feature approaches, and this has motivated us to develop methods that can automate the process of hand-engineering features. Some contents in this chapter have been published in [34].

- In **Chapter 4**, we propose a patient-specific system that employs a deep learning model to automatically learn features that are useful for epileptic seizure detection from raw scalp EEG. The model can learn to detect seizures *without* utilizing any seizure annotations (i.e., unsupervised learning). Whenever the seizure annotations become sufficiently available, our system can utilize such annotations to improve the detection performance. *Lessons learned*: we found several limitations of our current approach including missing some seizures and high detection latency. These problems were related to our choice of the model architecture and the nature of the epileptic seizures. In particular, we found that our model was not computationally efficient to increase spatial and temporal information so that the model perceives similar information as neurologists do when they are labeling seizure periods. Such increase can also make our model prone to overfitting. This was because our model consists of fully-connected layers (see more in Section 2.9). We also found that it is difficult to investigate the errors made by the model as epileptic seizures exhibit inconsistent patterns in EEG signals. It may be better to try with a problem that has well-defined and consistent signal patterns. This has motivated us to change our model architecture and move from epileptic seizure detection to sleep stage scoring. Some contents in this chapter have been published in [35].

- In **Chapter 5**, we propose a deep learning model, named DeepSleepNet, for automatic sleep stage scoring based on raw single-channel scalp EEG. In DeepSleepNet, we utilize Convolutional Neural Networks, which is more computationally efficient and less prone to overfitting (see more in Section 2.9), to extract time-invariant features and Bidirectional Long Short-Term Memory to learn transition rules among sleep stages automatically from EEG epochs. We also develop a two-step training algorithm that can efficiently train the model end-to-end via backpropaga-

tion, while preventing the model from suffering class imbalance problem. For the first time, we demonstrated that our DeepSleepNet can automatically learn time-invariant and temporal features from raw single-channel EEG that are useful for sleep stage scoring. Also, without changing model architecture, it can generalize to two different single-channel EEG signals from two datasets that have different characteristics, and achieve similar scoring performance compared to the state-of-the-art supervised feature engineering approaches. *Lessons learned*: we found a better choice of model architecture that can efficiently extract features from single-channel EEG signals. Next, we will apply what we have learned so far to develop a model that can extract useful information from unlabelled biosignals. The contents of this chapter have been published in [36].

- In **Chapter 6**, we propose a novel data-driven approach to detect desynchronization among biosignals from different domains as anomalies, without utilizing any prior knowledge about the synchronized patterns and annotations in the dataset. Our approach is to train a model to learn correlation or synchronized behaviors between two biosignals from two domains by transcribing signals from one domain (e.g., eye movements) to their simultaneous signals in another domain (e.g., brain activity). Desynchronization can, therefore, be detected by comparing the similarity between the measured (or real) simultaneous biosignal and the transcribed (or synthesized) one. If the degree of synchrony is less than a threshold, it is very likely that the desynchronization is happening. With this approach, we simplify the problem of "desynchronization detection in biosignals from multiple domains" into the problem of "comparing the correlation or the similarity between biosignals in the same domain". This is because we can apply the existing techniques for measuring the similarity between two signals from the same domain, instead of from different domains) to estimate the degree of synchrony. The detection of the desynchronization periods can provide useful information in addition to the analysis of each model separately, such as an early sign of harmful diseases. We demonstrate its usefulness via an example application of eye-movement detection, which is useful for many problems such sleep stage scoring. Some contents of this chapter have been published in [37].

- In **Chapter 7** and **Chapter 8**, we identify remaining challenges and discuss the potential future works.

1.5 Publications

A. Supratak, Ling Li, and Yike Guo, "Feature extraction with stacked autoencoders for epileptic seizure detection," in *36th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, 2014, pp. 4184-4187.

In this paper, we investigate the possibility of utilizing unsupervised feature learning, the recent development of deep learning, to automatically learn features from raw, unlabeled EEG data that are representative enough to be used in seizure detection. We develop patient-specific seizure detectors by using stacked autoencoders and logistic classifiers. We also implement a two-step training consisting of the greedy layer-wise and the global fine-tuning to train our detectors.

A. Supratak, C. Wu, H. Dong, K. Sun, and Y. Guo, "Survey on Feature Extraction and Applications of Biosignals," in *Machine Learning for Health Informatics: State-of-the-Art and Future Challenges*, vol. 9605, 2016, pp. 161-182.

In this paper, we review different types of biosignals and the main approaches to extract features from the signal in the context of biomedical applications. We also discuss challenges and limitations of the existing approaches, and possible future research.

A. Supratak, G. Datta, C. Wu, S. Yu, C. DArcy, R. Nicholas, Y. Guo, and P. Matthews, "Remote Actigraphy for Quantitative Assessment of Walking Speed in People with MS," *Neurology*, vol. 86, no. 16 Supplement, 2016.

In this paper, we propose a subject-specific gait calibration approach to build a walking speed estimation model for people with MS. Our model utilizes artigraphic data from a single tri-axial accelerometer attached to the lower-back position to estimate walking speed.

H. Dong, A. Supratak, W. Pan, C. Wu, P. M. Matthews, and Y. Guo, "Mixed Neural Network Approach for Temporal Sleep Stage Classification," *IEEE Trans. Neural Syst. Rehabil. Eng.*, 2017.

In this paper, we propose a practical approach to addressing limitations posed by using of single-channel electroencephalography (EEG) for sleep stage classification. We utilize rectifier neural network for detecting hierarchical features and long short-term memory (LSTM) network for sequential data learning to optimize classification performance with single-channel recordings.

A. Supratak, H. Dong, C. Wu, and Y. Guo, "DeepSleepNet: a Model for Automatic Sleep Stage Scoring based on Raw Single-Channel EEG," *IEEE Trans. Neural Syst. Rehabil. Eng.*, 2017.

In this paper, we propose a deep learning model, named DeepSleepNet, for automatic sleep stage scoring based on raw single-channel EEG. We utilize Convolutional Neural Networks to extract timeinvariant features, and bidirectional-Long Short-Term Memory to learn transition rules among sleep stages automatically from EEG epochs. We implement a two-step training algorithm to train our model efficiently.

H. Dong, A. Supratak, L. Mai, F. Liu, A. Oehmichen, S. Yu, and Y. Guo, "TensorLayer: A Versatile Library for Efficient Deep Learning Development," *ACM Multimedia*, no. 1, pp. 1-4, 2017.

In this paper, we propose TensorLayer which is a Python-based versatile deep learning library. TensorLayer provides high-level modules that abstract sophisticated operations towards neuron layers, network models, training data and dependent training jobs. In spite of offering simplicity, it has transparent module interfaces that allows developers to flexibly embed low-level controls within a backend engine, with the aim of supporting fine-grain tuning towards training. Real-world cluster experiment results show that TensorLayer is able to achieve competitive performance and scalability in critical deep learning tasks. TensorLayer was released in September 2016 on GitHub. Since after, it soon become one of the most popular open-sourced deep learning library used by researchers and practitioners.

A. Supratak, S. Schneider, H. Dong, L. Li and Y. Guo, "Towards Desynchronization Detection in Biosignals," *NIPS Time Series Workshop*, 2017.

This study presents a novel data-driven approach to detect desynchronization among biosignals from two modalities. We propose to train a deep neural network to learn synchronized patterns between biosignals from two modalities by transcribing signals from one modality into their expected, simultaneous or synchronized signal in another modality. Thus, instead of measuring the degree of synchrony between signals from different modalities using traditional linear and non-linear measures, we simplify this problem into the problem of measuring the degree of synchrony between the real and the synthesized signals from the same modality using the traditional measures. Desynchronization detection is then achieved by applying a threshold function to the estimated degree of synchrony. We demonstrate

the approach with the detection of eye-movement artifacts in a public sleep dataset and compare the detection performance with traditional approaches.

2

Background

In this chapter, we start with a brief introduction to different types of biosignals that are commonly analyzed for diagnosis and treatment of diseases. After that, we will discuss necessary technical background on machine learning including supervised learning, unsupervised learning and deep learning, which will be used through this thesis. We recommend the following materials for more comprehensive details on machine learning [5, 8, 38].

2.1 Biosignals

Biosignals are signals or time-series data that can be continually measured from the human body. They contain information about physiological phenomena which reflect human health and well-being [12]. They have become an important source of information for medical diagnosis, subsequent therapy, and passive healthy monitoring. They can be either electrical, such as electroencephalogram (EEG), electrocardiogram (ECG), electrooculogram (EOG) and electromyogram (EMG), or non-electrical, such as breathing and movements. In this section, we will discuss examples of biosignals that have been widely used in clinical applications.

2.1.1 Electroencephalogram

Electroencephalogram (EEG) is an electrical biosignal of brain activity recorded from electrodes placed on a subject's scalp. This signal reflects the potential differences resulting from the activity of neurons

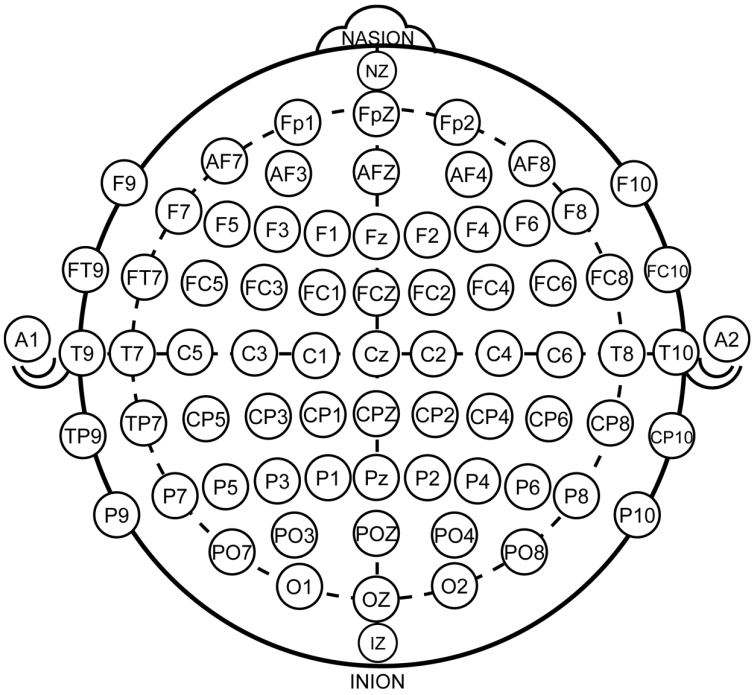


Figure 2.1: Scalp electrode positioning in the 10-20 International System [1].

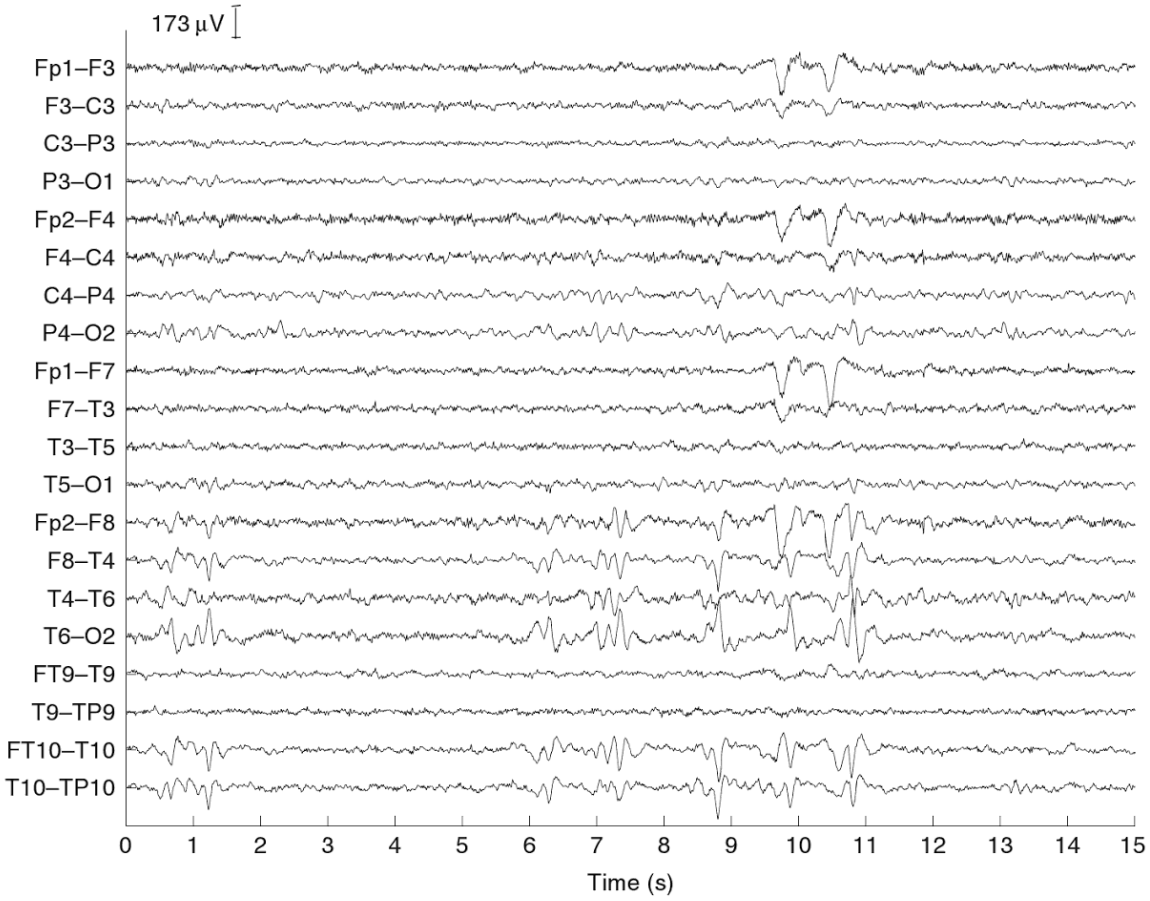


Figure 2.2: An example of 15-s EEG epoch from 20 channels.

Table 2.1: Typical EEG subbands and their clinically relevant activity

Name	Frequency band	Brain state
Delta-waves	0-4 Hz	Stage III and IV sleep, young persons
Theta-waves	4-8 Hz	Upon waking and just before falling sleep
Alpha-waves	8-15 Hz	Relaxed, alert state of consciousness
Beta-waves	12-30 Hz	Busy or anxious thinking, active concentration
Gamma-waves	> 30 Hz	Higher mental tasks, imaginary singing

in the brain and are often contaminated by external interference outside the brain, such as eye movements, muscular activity and cardiac activity [20]. The EEG is typically recorded by positioning the electrodes following the 10-20 International System of scalp electrode placement [1] (see Figure 2.1). Figure 2.2 shows an example of 15-s EEG epoch from 20 channels.

Traditionally, EEG is decomposed into different frequency bands (e.g., using STFT or DWT) for analyzing clinically relevant activity [39]. The most common frequency bands are briefly summarized in Table 2.1 [20]. The information from these bands or other features extracted from EEG has been used to help diagnose and monitor many conditions affecting the brain such as the recognition of epileptic seizures [18] or the different sleep stages [22].

2.1.2 Electrocardiogram

Electrocardiogram (ECG) is an electrical biosignal of heart activity recorded from electrodes placed on the subject's skin. These electrodes detect the electrical signal generated by the pattern of depolarizing and repolarizing of the heart muscle during each heartbeat [20]. A healthy heart will produce a regular, well-organized electro-mechanical activity known as normal sinus rhythm (NSR).

Figure 2.3 illustrates an example of ECG of a heart in NSR. Typically, a single heartbeat consists of three main components: the P wave, QRS complex, and the T wave. Ventricular depolarization (or when the heart is pumping blood to the lungs and the rest of the body) causes the impulsive QRS complex. Ventricular repolarization and the subsequent relaxation of the ventricular muscles are associated with T wave. Atria depolarization (or when the heart is transferring blood from atria to ventricles) produces the P wave, but their repolarization is believed to happen during the ventricular contraction, which is obscured by the stronger signal from the QRS complex. The patterns of a healthy heart are predictable. Thus feature such as the duration from the beginning of the P wave to the

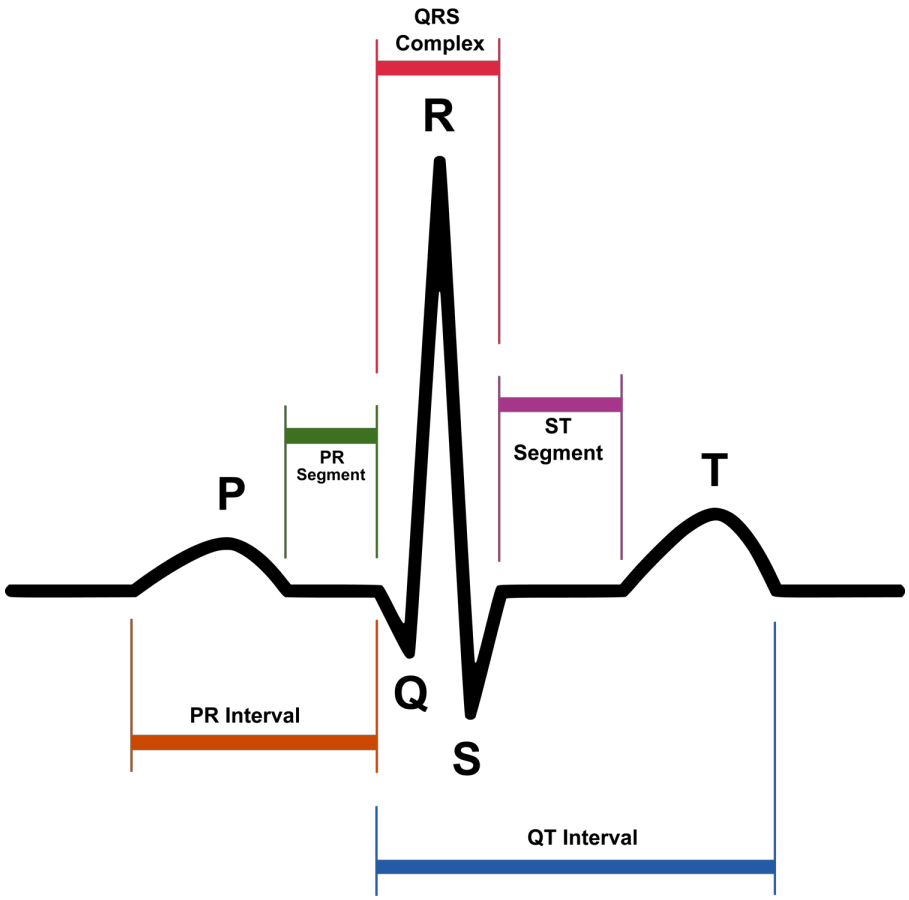


Figure 2.3: An example of ECG of a heart in normal sinus rhythm (NSR) [2].

beginning of the QRS complex (i.e., PR interval) and the duration of QRS complex can be monitored and diagnosed to detect deviation from NSR such as cardiac arrhythmia [40].

2.1.3 Electrooculogram

Electrooculogram (EOG) is an electrical biosignal of eye movements recorded from bipolar electrodes placed around the eyes. This signal measure the potential difference between the anterior pole (positive side) and the posterior pole (negative side) of the eyes when the eyes move [3]. EOG has been used to help identify different sleep stages such as Rapid Eye Movement (REM) [27]. Figure 2.4 shows an example of EOG generated by horizontal eye movements measured. This EOG is measured by placing a pair of electrodes at the outer side of the left and right eyes. The rotation of the eyes to the right results in a positive-trending change in the potential difference, while the rotation to the left results in a negative-trending change.

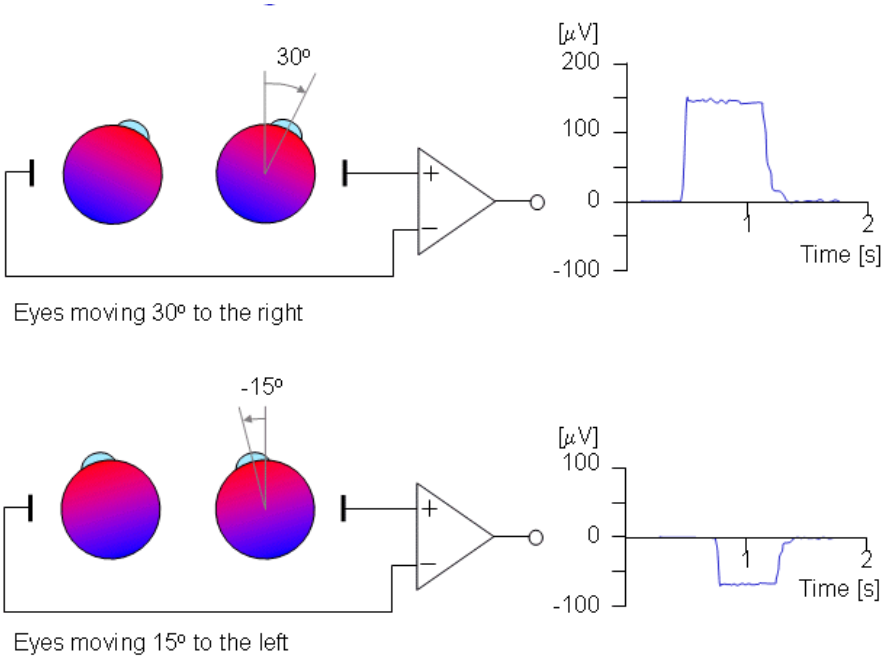


Figure 2.4: An example of EOG generated by horizontal eye movements [3]. A pair of electrodes are placed at the outer side of the left and right eyes.

2.1.4 Electromyogram

Electromyogram (EMG) is an electrical biosignal produced by skeletal muscles when these cells are electrically or neurologically activated. There are two kinds of EMG: surface EMG (sEMG) and intramuscular EMG [41]. sEMG measures muscle activity from the surface above the muscle on the skin such as the speed and strength of signals traveling between two or more points (i.e., electrodes). Intramuscular EMG records the electrical activity in that muscle by inserting needle electrodes into different points of the muscle. Typically, EMG has been used to help diagnose muscle disorders or disorders that affect the motor neurons and to identify neuromuscular diseases [42]. EMG has also been used in the sleep analysis to provide additional information in classifying sleep stages [27].

2.1.5 Gait Acceleration

Gait acceleration is a non-electrical biosignal that is measured by (multi-axis) accelerometers attached to different parts of the human body such as wrists, ankles, and waist. The accelerometers measure acceleration force in the g-force unit (where $g \approx 9.81 m/s^2$). Figure 2.5 illustrates an example of 3-dimension gait acceleration collected from a tri-axial accelerometer attached around the waist and close to the center of mass. The gait acceleration can be analyzed, for example, to build or calibrate

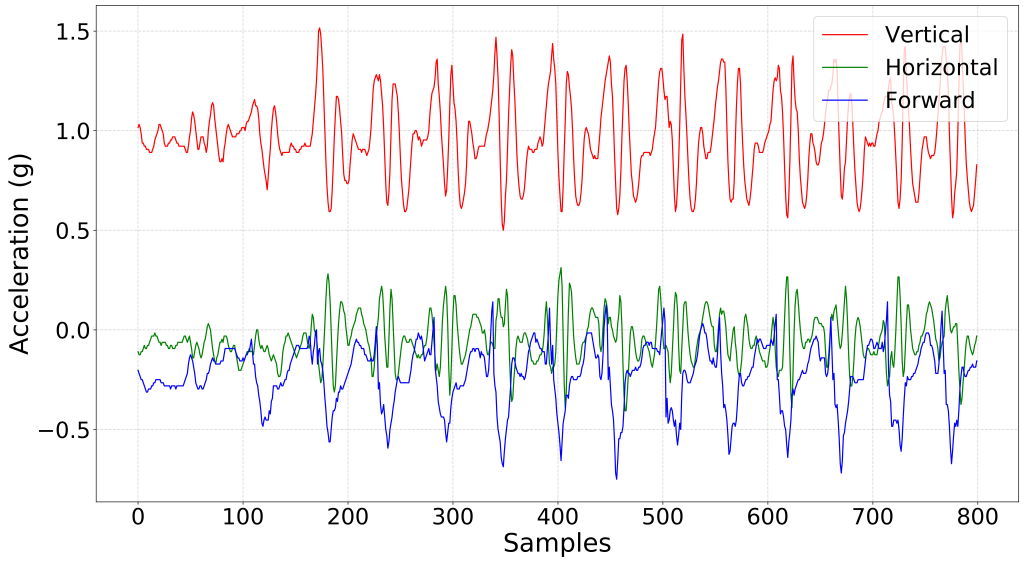


Figure 2.5: An example of raw 3D acceleration data collected from a tri-axial accelerometer attached around the waist and close to the center of mass. This accelerometer measures three signals simultaneously, which are vertical (red), horizontal (green) and forward (blue) acceleration. Each peak of *forward* acceleration preceding a change of the sign from positive to negative can be thought of as an instance of a foot-contact [4].

a model for estimating the number of steps and walking speed in people with walking impairments such as Multiple Sclerosis (MS) [43, 44].

2.2 Supervised Learning

Suppose there are m training examples $\left\{(x^{(i)}, y^{(i)})\right\}_{i=1}^{m}$ (i.e., a training set), where $x^{(i)}$ is the **input** variable, also called input features, $y^{(i)}$ is the **output** or target variable that we are trying to predict, $(x^{(i)}, y^{(i)})$ is a training example, and the superscript "(i)" is an index into the training set. The goal of the supervised learning algorithm is to learn a function $f_\theta : X \to Y$, where θ is the parameters of the function f, X is the space of input values and Y is the space of output values. For instance, in the sleep analysis, X could be the space of EEG signals and Y could be a set of five sleep stages {W,N1,N2,N3,REM} indicating one of the possible class of the input EEG signals. The function f_θ is considered as a "good" predictor for the corresponding value of y when the disagreement between a predicted value $\hat{y}^{(i)} = f_\theta(x^{(i)})$ and a target value $y^{(i)}$ is very low. Such disagreement can be quantified using a scalar-valued **loss function** (or **cost function**) $L(\hat{y}, y)$. Thus the objective in supervised

learning is to find θ that minimizes $L(\hat{y}, y)$ as follows:

$$\theta^* = \arg\min_{\theta} J(\theta) = \arg\min_{\theta} \frac{1}{m} \sum_{i=1}^{m} L(f_\theta(x^{(i)}), y^{(i)}) \ . \tag{2.1}$$

In other words, the objective is to find θ^* that minimizes the defined loss over training examples. The learned function f_{θ^*} will be then used to predict samples of Y given X.

When the space of Y is a set of discrete values, such as sleep stages, we call such learning problem a **classification** problem. When the space of Y is continuous, such as walking speed, we call it a **regression** problem.

2.2.1 Linear Regression

Consider a simple example when there are 100 training examples ($m = 100$), where $\mathbf{x}^{(i)} \in \mathbb{R}^2$ and $y^{(i)} \in \mathbb{R}$. The linear function f_θ to approximate $y^{(i)}$ from $\mathbf{x}^{(i)}$ is defined as follows:

$$f_\theta(\mathbf{x}^{(i)}) = \mathbf{W}^T \mathbf{x}^{(i)} + b \ , \tag{2.2}$$

where $\mathbf{W} \in \mathbb{R}^2$ and $b \in \mathbb{R}$. In this case, the parameters of the function f_θ are $\mathbf{W}$ and b (i.e., $\theta = \{\mathbf{W}, b\}$). The $\mathbf{W}$ and b will be optimized based on the defined loss function. A commonly used loss function in the regression problem is the mean squared difference between the predicted and the target value, $L(\hat{y}^{(i)}, y^{(i)}) = (\hat{y}^{(i)} - y^{(i)})^2$. Therefore, the objective becomes:

$$\theta^* = \arg\min_{\mathbf{W}, b} \frac{1}{m} \sum_{i=1}^{m} (\mathbf{W}^T \mathbf{x}^{(i)} + b - y^{(i)})^2 \ . \tag{2.3}$$

2.2.2 Linear Classification

In classification problem, instead of predicting a scalar value from an input, we assign the input a discrete category. Suppose we want to map each of 100 training examples to one of the three possible classes (i.e., $Y = \{\text{C1,C2,C3}\}$). In classification problem, it is common to interpret the output vector of $f_\theta(\mathbf{x}^{(i)}) = \mathbf{W}^T \mathbf{x}^{(i)} + b$ as **logits**. The probabilities of the three classes can be obtained by passing this logit vector through the *softmax* function, which takes a vector $\mathbf{z}$ and outputs a vector of the same size $\mathbf{p}$; where $p_j = e^{z_j} / \sum_{c=1}^{C} e^{z_c}$, j is the index of the vector and C is the number of output

classes ($C = 3$ in this example). Note that the vector $\mathbf{z}$ can contain arbitrary real values, but the vector p is normalized so that all of its elements are between 0 and 1 and they sum to 1 (e.g., [0.1,0.25,0.65]). This vector $\mathbf{p}$ is essentially our prediction $\hat{\mathbf{y}}$ from the linear classifier and the *correct* label $\mathbf{y}$ is a 3-dimensional vector that is all zero except for a single 1 at the index of the true class. For example, if the correct class is C1, then $y = [1, 0, 0]$. The most commonly used loss function in the classification problem is the cross-entropy between two distributions $H(p, q) = -\sum_x p(x) \log q(x)$. Thus the *cross-entropy* loss can be defined as follows:

$$L(\hat{\mathbf{y}}^{(i)}, \mathbf{y}^{(i)}) = -\sum_{j=1}^{C} y_j^{(i)} \log \hat{y}_j^{(i)} \ . \tag{2.4}$$

As the correct label $\mathbf{y}^{(i)}$ is all zeros except for a single 1 at the index of the true class, this equation (2.4) can be simplified into $L(\hat{\mathbf{y}}^{(i)}, \mathbf{y}^{(i)}) = -\log \hat{y}_{y=1}^{(i)}$, where the subscript $y = 1$ is the index of the single correct class. With this loss, we are effectively minimizing the negative log probability of the correct class. This can also be interpreted as maximizing the log likelihood of the class $\mathbf{y}$ conditioned on the input $\mathbf{x}$.

In many practical problems, the linear function is typically not sufficient to model the mapping between X and Y. A more complex function f_θ such as **neural networks** (or **deep learning**) is usually applied to predict the outputs $\mathbf{y}$ given the inputs $\mathbf{x}$. We will discuss more about deep learning later in this chapter.

2.2.3 Support Vector Machine

Support vector machine (SVM) [45] is an example of large-margin classifier that try to find a **decision boundary** (or a separating hyperplane) that has the largest distance (or margin) to the nearest training examples of any classes. In particular, given m training examples $\{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^m$, where $\mathbf{x}^{(i)} \in \mathbb{R}^n$ and $y^{(i)} \in \{-1, 1\}$, we are trying to solve the following optimization problem [46]:

$$\begin{aligned} \min_{\mathbf{W},b,\xi} \quad & \frac{1}{2} \|\mathbf{W}\|_2^2 + C \sum_{i=1}^{m} \xi_i \\ \text{subject to} \quad & y^{(i)}(\mathbf{W}^T \mathbf{x}^{(i)} + b) \geq 1 - \xi_i, i = 1, \ldots, m \ , \\ & \xi_i \geq 0, i = 1, \ldots, m \ , \end{aligned} \tag{2.5}$$

where $\mathbf{W} \in \mathbb{R}^n$, C is a regularization parameter that control tradeoff between maximizing the margin width (i.e., allow some examples to have margin equal to $1 - \xi_i$ with $\xi_i > 0$) and ensuring that most examples have function margin at least 1. After the problem is solved, SVM predicts the output class by computing:

$$\mathbf{W}^T\mathbf{x} + b \ . \tag{2.6}$$

If the output value ≥ 0, SVM outputs $\hat{y} = 1$. Otherwise, it outputs $\hat{y} = -1$.

In addition to performing linear classification, SVMs can efficiently perform a *non-linear classification* using the **kernel trick** to implicitly mapping their inputs into high-dimensional feature spaces. In particular, we first rewrite the linear function into the following form [38]:

$$\mathbf{W}^T\mathbf{x} + b = b + \sum_{i=1}^{m} \alpha_i y^{(i)} \langle \mathbf{x}, \mathbf{x}^{(i)} \rangle \ , \tag{2.7}$$

where $\mathbf{x}$ is an input vector, $(\mathbf{x}^{(i)}, y^{(i)})$ is the i-th training example, $\boldsymbol{\alpha}$ is a vector of coefficients, and $\langle \mathbf{x}, \mathbf{x}^{(i)} \rangle$ is a dot product (i.e., $\mathbf{x}^T\mathbf{x}^{(i)}$). With this formula, we then replace $\mathbf{x}$ with the output of a given feature function $\phi(\mathbf{x})$, and replace $\langle \mathbf{x}, \mathbf{x}^{(i)} \rangle$ with a **kernel** function $k(\mathbf{x}, \mathbf{x}^{(i)}) = \phi(\mathbf{x})^T\phi(\mathbf{x}^{(i)})$. Thus, by replacing the dot product with the kernel function, SVM instead predicts the output class by computing:

$$b + \sum_{i=1}^{m} \alpha_i y^{(i)} k(\mathbf{x}, \mathbf{x}^{(i)}) \ . \tag{2.8}$$

With this rewritten decision function, we instead solve the dual optimization problem (can be found in [46]) to find the vector $\boldsymbol{\alpha}$ that maximize the margin.

The most commonly used kernel to achieve the non-linear classification is the **Gaussian kernel**:

$$k(u, v) = \exp(-\frac{\|u - v\|_2^2}{2\sigma^2}) \ , \tag{2.9}$$

where σ is a hyperparameter. This Gaussian kernel can be interpreted as a similarity measure between u and v, in which $k(u, v)$ is close to 1 when u and v are close according to Euclidean distance, and near 0 when u and v are far apart.

SVM with the Gaussian kernel is, therefore, performing a kind of *template matching*, where u and v in Equation 2.9 are replaced with an input $\mathbf{x}$ and the i-th training example $\mathbf{x}^{(i)}$ respectively. In particular, each training example $\mathbf{x}^{(i)}$ associated with training label $y^{(i)}$ becomes a template for class

$y^{(i)}$. The process to evaluate the decision function (Equation 2.8) is basically combining many such training labels weighted by the similarity (computed with Gaussian kernel) of the corresponding training examples.

A major drawback of using kernel machines is the process to evaluate the decision function, as its cost is linear in the number of training examples. SVM mitigates this problem by learning the vector $\boldsymbol{\alpha}$ that mostly contains zeros. Thus the process to evaluate the decision function only requires evaluating a few of training examples that have non-zero α_i. These training examples are called **support vectors**.

2.2.4 Support Vector Regression

In addition to the classification problem, SVM can be applied in the regression problem as well, and its variant is called Support Vector Regression (SVR) [47]. In particular, suppose there are m training examples $\{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{m}$, where $\mathbf{x}^{(i)} \in \mathbb{R}^n$ and $y^{(i)} \in \mathbb{R}$. We are trying to solve the following optimization problem [46]:

$$\begin{aligned} \min_{\mathbf{W},b,\xi} \quad & \frac{1}{2}\|\mathbf{W}\|_2^2 + C\sum_{i=1}^{m}\xi_i + C\sum_{i=1}^{m}\xi_i^* \\ \text{subject to} \quad & \mathbf{W}^T\mathbf{x}^{(i)} + b - y^{(i)} \leq \varepsilon + \xi_i \ , \\ & y^{(i)} - \mathbf{W}^T\mathbf{x}^{(i)} - b \leq \varepsilon + \xi_i^* \ , \\ & \xi_i, \xi_i^* \geq 0, i = 1, \ldots, m \ , \end{aligned} \tag{2.10}$$

where ε is a hyperparameter. SVR is trying to find an ε tube so that if the predicted value is within a distance ε from the actual value, the loss is zero. If an example is outside the tube, the loss is increased by $C\xi_i$ or $C\xi_i^*$. Similar to the SVM, we can apply the kernel trick to achieve the non-linear regression as well (see Section 2.2.3). After the problem is solved, SVR estimates the output value by computing:

$$b + \sum_{i=1}^{m}(-\alpha_i + \alpha_i^*)k(\mathbf{x}, \mathbf{x}^{(i)}) \ . \tag{2.11}$$

With this decision function, we instead solve the dual optimization problem (can be found in [46]) to learn the vector $\boldsymbol{\alpha}$ and $\boldsymbol{\alpha}^*$.

2.3 Unsupervised Learning

The goal of unsupervised learning is to extract information from the **unlabeled** dataset (i.e., there are no labels). The training set now becomes a training set, $\left\{\mathbf{x}^{(i)}\right\}_{i=1}^{m}$. This unsupervised learning is motivated by the fact that we want to learn something from the data without having humans to label them. This is typically the case for most of the applications as it is very expensive to annotate the data.

Unlike supervised learning, there is no a single generic objective in unsupervised learning as it varies depending on the tasks and the nature of the available dataset. We typically consider the tasks of density estimation, clustering the data into groups of similar examples, or learning to draw samples from a distribution as unsupervised learning.

The general task of unsupervised learning is to find "good" representations of the data [38]. Representations are typically considered as "good" when they can preserve as much information about the data as possible while maintaining some constraints defined to keep the representations *simpler* or more meaningful than the original data themselves. There are many approaches to define a simpler representation including low-dimensional representations, sparse representations, and independent representations. The first approach is to compress the original data as much as possible into a smaller representation, such as multivariate Gaussian distribution or autoencoder [48]. The second approach is to transform the data into representations whose elements are mostly zeros, such as sparse autoencoder [49]. This technique usually requires a larger representation, so that it can preserve information about the original data even most of its elements are zeros. This can result in a representation that tends to distributed data along the dimensions of the representation space. The third approach is to disentangle the source of variation underlying the data distribution. This can result in a representation whose dimensions are statistically independent, such as Principal Component Analysis (PCA) [50]. Note that some techniques can result in the representations that have more than one properties such as the representations can be both in lower dimensions whose elements are independents such as PCA.

Note that most of the approaches in this thesis are based on deep learning, in which its strength is to automatically learn representations that are useful for certain tasks. Most of the remaining contents in this chapter will later present other representation learning algorithms (e.g., deep learning) that implicitly develop these approaches in different ways. Next, we will discuss three examples of unsupervised learning.

2.3.1 k-means Clustering

k-mean clustering [38] is an algorithm to group the training set into k different cluster of examples that are near each other. The k-mean algorithm starts by randomly initializing k different centroids $\{\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k\}$. Then it alternates between the cluster assignment step and the move centroid step until convergence. In the cluster assignment step, each training example is assigned to cluster j, where j is the index of the nearest centroid $\boldsymbol{\mu}_j$. The common choice of function to measure the distance is Euclidean distance. In the move centroid step, each $\boldsymbol{\mu}_j$ is moved to the mean of all training examples $\mathbf{x}^{(i)}$ that have been assigned to cluster j.

Note that this k-mean clustering may converge to a local optimum. One possible solution is to re-run the k-mean clustering many times, and choose the set of clusters that gives the lowest average of all Euclidean distances between each example and its assigned cluster centroid.

2.3.2 Principal Component Analysis

Principal Component Analysis (PCA) [38, 50] learns an orthogonal, linear transformation of the data that projects a set of (possibly correlated) training examples to a set of uncorrelated representations. Such linear transformation can be obtained as follows. Suppose there are m unlabeled training examples $\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}\}$, where $\mathbf{x}^{(i)} \in \mathbb{R}^n$. Firstly, we need to perform mean normalization (i.e., subtracting the mean of each feature dimension from every example). Then we compute the sample covariance matrix $\boldsymbol{\Sigma}$ as follows:

$$\boldsymbol{\Sigma} = \frac{1}{m-1} \sum_{i=1}^{m} (\mathbf{x}^{(i)})(\mathbf{x}^{(i)})^T \ , \tag{2.12}$$

where $\boldsymbol{\Sigma} \in \mathbb{R}^{n \times n}$. Then we compute the eigenvector and the engenvalues of $\boldsymbol{\Sigma}$ using *eigendecomposition*:

$$\boldsymbol{\Sigma} = \mathbf{W} \boldsymbol{\Lambda} \mathbf{W}^T \ , \tag{2.13}$$

where $\mathbf{W}$ is an orthogonal matrix composed of eigenvectors of $\boldsymbol{\Sigma}$, $\boldsymbol{\Lambda}$ is a diagonal matrix whose $\Lambda_{j,j}$ is the eigenvalue associated with the eigenvector in the column j of $\mathbf{W}$, $\mathbf{W} \in \mathbb{R}^{n \times n}$, and $\boldsymbol{\Lambda} \in \mathbb{R}^{n \times n}$. Note that the first column of the matrix $\mathbf{W}$ (i.e., $\mathbf{w}_1$) is the principal eigenvector corresponding to $\Lambda_{1,1}$ which is the largest eigenvalue or has the highest variance, $\mathbf{w}_2$ is the second eigenvector corresponding to $\Lambda_{2,2}$ which is the second-largest eigenvalue, and so on.

An alternative approach is to apply *singular value decomposition (SVD)* to extract $\mathbf{W}$ and $\boldsymbol{\Lambda}$ from

the training data, which is more generally applicable and numerically stable compared to the eigen-decomposition. Nevertheless, these two approaches can give us the same transformations that can be used in PCA.

The eigenvectors in $\mathbf{W}$ are called principal axes or principal directions of the data, and the projections of examples on the principal axes are called **principal components**. These eigenvectors can be used to transform each training example $\mathbf{x}^{(i)}$ into a representation $\mathbf{z}^{(i)}$ whose the individual dimensions are *mutually uncorrelated*. They can also be applied to reduce the dimensions of the training examples from n to k, where $k < n$. In particular, we can form a matrix $\mathbf{W}_{reduce}$ using the first k eigenvectors (i.e., columns) of the matrix $\mathbf{W}$. The *compressed* representation of $\mathbf{x}^{(i)}$ can be obtained as follows:

$$\mathbf{z}^{(i)} = \mathbf{W}_{reduce}^{T}\mathbf{x}^{(i)} \ , \tag{2.14}$$

where $\mathbf{z}^{(i)} \in \mathbb{R}^k$ and $\mathbf{W}_{reduce} \in \mathbb{R}^{n \times k}$. The *reconstruction* of the compressed representation can be obtained as follows:

$$\hat{\mathbf{x}}^{(i)} = \mathbf{W}_{reduce}\mathbf{z}^{(i)} \ , \tag{2.15}$$

where $\hat{\mathbf{x}}^{(i)}$ is the reconstructed examples and $\hat{\mathbf{x}}^{(i)} \in \mathbb{R}^n$.

The choice of k is application-specific. For instance, we can apply PCA to compress multi-dimensional data for data visualization. In this case, we typically compress the data into 2D or 3D (i.e., $k = 2, 3$) and visualize them on graphs. Also, we can apply PCA to reduce the number of input dimensions to speed up the training process. In this case, the choice of k can be based on the percentage of variance retained. The **percentage of variance retained** [51] can be estimated using the eigenvalues of Σ, where $\Lambda_{j,j}$ is the eigenvalue λ_j corresponding to the eigenvector $\mathbf{w}_j$ (which are sorted in descending order). We can compute the percentage of variance retained (range from 0 to 1) of k principal components as follows:

$$\frac{\sum_{j=1}^{k} \lambda_j}{\sum_{j=1}^{n} \lambda_j} \ . \tag{2.16}$$

With this, we can pick the smallest value of k that gives the the percentage of variance retained greater than or equal to a given value (e.g., 0.99 if we want to retain 99% of the variance).

2.3.3 Anomaly Detection

Anomaly detection refers to the problem of finding examples whose patterns do not conform to or are different from the expected normal ones [52]. These examples are often referred as *anomalies* or *outliers* depending on application domains. Typically, there would be a very small number of labeled data (normal or anomalous) available, as the acquisition of a labeled set of anomalous examples that cover all possible patterns of anomalies is more difficult than getting labels for normal examples. Also, the anomalous pattern is usually not consistent, and new anomalous patterns may arise. Therefore, the anomaly detection is commonly considered as an unsupervised learning problem even though we might have some labels of anomalous examples.

A possible approach to detect anomalies is first to estimate the probability density function $p_\theta(\mathbf{x})$ that generated dataset based on the available training examples. The estimated density function is then applied to determine the probability of a new example. If its probability is less than a threshold ε (i.e., $p_\theta(\mathbf{x}) < \varepsilon$), it is annotated as *anomaly*. The probability density function is typically estimated by using **multivariate Gaussian distribution** [53]. It is a generalization of one-dimensional (or univariate) Gaussian distribution to more than one dimensions, which is parameterized by a mean vector $\boldsymbol{\mu}$ and a covariance matrix $\boldsymbol{\Sigma}$ (i.e., $\theta = \{\boldsymbol{\mu}, \boldsymbol{\Sigma}\}$). In particular, given an unlabeled training set, $\left\{\mathbf{x}^{(i)}\right\}_{i=1}^{m}$, where $\mathbf{x}^{(i)} \in \mathbb{R}^n$, n is the number of dimensions and m is the number of examples, we first estimate mean $\boldsymbol{\mu}$ and covariance matrix $\boldsymbol{\Sigma}$ of the examples as follows:

$$\boldsymbol{\mu} = \frac{1}{m}\sum_{i=1}^{m}\mathbf{x}^{(i)} \ , \tag{2.17}$$

$$\boldsymbol{\Sigma} = \frac{1}{m}\sum_{i=1}^{m}(\mathbf{x}^{(i)} - \boldsymbol{\mu})(\mathbf{x}^{(i)} - \boldsymbol{\mu})^T \ , \tag{2.18}$$

where $\boldsymbol{\mu} \in \mathbb{R}^n$ and $\boldsymbol{\Sigma} \in \mathbb{R}^{n \times n}$. These two parameters can be considered as *low-dimensional representations* of the training examples as they can be used to approximate the density function of the data generation function. The estimated $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ are used to determine the probability of a given input $\mathbf{x}$ with the density function of the multivariate Gaussian distribution as follows:

$$p_{\boldsymbol{\mu},\boldsymbol{\Sigma}}(\mathbf{x}) = \frac{1}{(2\pi)^{\frac{n}{2}}|\boldsymbol{\Sigma}|^{\frac{1}{2}}}\exp\left(-\frac{1}{2}(\mathbf{x}-\mu)^T\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\mu)\right) \ , \tag{2.19}$$

where $|\boldsymbol{\Sigma}|$ denotes the determinant of $\boldsymbol{\Sigma}$ and $\boldsymbol{\Sigma}^{-1}$ denotes the inverse of $\boldsymbol{\Sigma}$. If the probability of an example is less than a threshold, $p_{\boldsymbol{\mu},\boldsymbol{\Sigma}}(\mathbf{x}) < \varepsilon$, it are annotated as anomaly. It should be noted that

the number of examples must be greater than the number of features (i.e., $m > n$), and the features must not be linearly dependent. Otherwise, $\mathbf{\Sigma}$ will be non-invertible.

The question then arises: as we do not have labels to evaluate the performance of our model, how can we choose the threshold ε using the cross-validation. Even though we are considering the anomaly detection as an unsupervised learning problem, we typically assume that we have some labeled data (e.g., $y = 0$ if normal and $y = 1$ if abnormal) that we can use as the validation and test sets. These validation and test sets can be applied for selecting hyperparameter via cross-validation and estimating the generalization error, respectively, as mentioned in Section 2.7. Note that the training data are still unlabeled, and it is still fine in practice even there are a few anomalous examples in the training data.

2.4 Gradient Descent

In this thesis, we are trying to solve an optimization problem of the form $\theta^* = \arg\min_\theta J(\theta)$, where θ is a vector of parameters, and $J(\theta)$ is the average loss of all training examples. As most of the approaches in this thesis developed by using deep learning (which represents a differentiable function), we will focus only on the most common optimization algorithm for deep learning, called **gradient descent (GD)**.

In the GD algorithm, a search start with a random θ, then it alternates two steps. The first step is to compute what is called **gradient**, $\nabla_\theta J(\theta)$. The gradient is a vector of partial derivatives, which gives us the slope of J along every dimension of θ or the *direction of increase* of J. The second step is to update θ by adding a small amount of the *negative* gradient direction to minimize J.

In practice, we will not compute the gradient using all training examples as the size of the training data can be very large (e.g., more than one million training examples). Instead, only a small minibatch of examples, such as 100 examples, is used to approximate the gradient at a time. This allows us to start making progress faster than scanning through the entire training examples before making an update. Thus we will perform many approximate updates instead of a few exact updates. This version of GD algorithm is called **stochastic gradient descent (SGD)**. Note that SGD may never converge to the minimum, and the parameters θ may keep oscillating around the minimum of $J(\theta)$. However, in practice, most the values near the minimum are a reasonably good approximation to the

true minimum. The SGD algorithm is summarized in the context of supervised learning as follows:

Repeat until convergence:

1. Sample a minibatch of m examples $\left\{(\mathbf{x}^{(i)}, y^{(i)})\right\}_{i=1}^{m}$ from training data
2. Compute the gradient estimate: $\nabla_\theta J(\theta) \approx \nabla_\theta \left[\frac{1}{m} \sum_{i=1}^{m} L(f_\theta(\mathbf{x}^{(i)}), y^{(i)})\right]$
3. Compute the update: $\Delta\theta := -\alpha \nabla_\theta J(\theta)$
4. Perform a parameter update: $\theta := \theta + \Delta\theta$

An important parameter in SGD is a **learning rate**, α. If α is too large, the optimization may not converge or even diverge. If it is too small, the optimization will take too long. In practice, the "good" value of α is application-specific, and we normally determine this value using cross-validation.

Recently, there have been several studies trying to speed up the training process by modifying the computation of $\Delta\theta$ in step 3 of the SGD algorithm to use the knowledge from previous steps, such as Adagrad [54], RMSProp [55] and Adam [56]. The intuition of these methods is to take larger steps of parameter update when parameters receive small gradients and to take smaller steps when parameters receive large gradients.

So far, we have not yet discussed how we compute the gradient. One way to compute the gradient is to manually work out the equation for the partial derivatives of $J(\theta)$. Fortunately, as deep learning models are a differentiable function, the gradient can, therefore, be computed using **backpropagation**, which will be discussed in the next section.

2.5 Backpropagation

As the function that we are going to compute the gradient of is function $J(\theta)$, which takes the parameters θ as input, we are, therefore, interested in the gradient $\nabla_\theta J(\theta)$ with respect to the parameters θ to perform the parameter update.

Backpropagation is the process to *compute gradients* of scalar-valued functions with respect to their inputs. This is achieved by using the chain rule from calculus to recursively compute gradients. We will explain the backpropagation using the following toy example. Suppose we have a function $f(x) = (7x - 9)^4$ that we want to compute $\partial f / \partial x$. In the context of deep learning, x would be a vector of network parameters and $f(x)$ would be the loss. By using the chain rule, we can introduce

intermediate variables that we can compute gradients with respect to its input. Then the gradients of the $f(x)$ can be calculated by multiplying the gradients of all intermediate variables. In this example, we introduce $a = 7x$, $b = a - 9$, and $f(x) = b^4$. The gradients of each intermediate variables are $\partial a/\partial x = 7$, $\partial b/\partial a = 1$, and $\partial f/\partial b = 4b^3$. Once we have these (local) derivatives, we can obtain the (global) derivative $\partial f/\partial x$ by repeatedly applying the chain rule as follows: $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial b}\frac{\partial b}{\partial a}\frac{\partial a}{\partial x}$.

To generalize the toy example above, suppose there is an input x_0 that is forwarded to multiple layers of transformation $x_i = f_i(x_{i-1})$ where $i = 1, ..., l$ and the last x_l is a scalar value. If we assume that the gradient (or at least the sub-gradient) exists, the final gradient can be computed via the *matrix product* of the Jacobians matrices of all intermediate transformation $\frac{\partial x_l}{\partial x_0} = \prod_{i=1}^{l} \frac{\partial x_i}{\partial x_{i-1}}$, where each matrix $\frac{\partial x_i}{\partial x_{i-1}}$ tell us the sensitivity of every dimension of x_i on every input dimension of x_{i-1} [5].

In summary, the SGD algorithm that uses backpropagation to compute gradients of the output with respect to the input consists of two main phases: *forward* and *backward* passes. During the forward pass, the minibach of data $\left\{(x^{(i)}, y^{(i)})\right\}_{i=1}^{m}$ are forwarded to the neural network with the current parameters θ to compute all intermediate values (caching them for later use) and the loss J. During the backward pass, the loss or the error is propagated backward through all intermediate transformations to recursively apply the chain rule to compute the gradients all the way to the input (i.e., backpropagation). At each transformation, we compute its local gradients using the cached intermediate values and multiply (or "chain") them with the gradients from the above.

In deep learning application, the computational process of all transformations or functions from inputs to outputs can also be viewed as a directed acyclic graph (DAG) of functions or **computational graph**. In the computational graph, vectors flow along edges and nodes represent differentiable functions or transformations that receive some vectors and combine them into one vector that then flows to other nodes. Figure 2.6 illustrate an example of backpropagation along a computational graph [5]. During forward pass, vectors x and y are forwarded into a node that 1) compute vector z using some fixed function such as $z = x \odot y$, where $\odot$ is an element-wise multiplication; and 2) compute its local gradients (or Jacobian matrices) $\frac{\partial z}{\partial x}$ and $\frac{\partial z}{\partial y}$. The vector z is then forwarded to the subsequent node in the graph. At the end of the computational graph, the total loss J (a scalar value) is computed. During backward pass, the computational process proceeds in the reverse order to recursively apply the chain rule to compute the gradients all the way to the input. In particular, this node multiplies the global gradient on $\frac{\partial J}{\partial z}$ (which tell us the influence of z on the final output or the loss J) onto the local gradients for each input. This results in the global gradients for x ($\frac{\partial J}{\partial x} = \frac{\partial z}{\partial x}\frac{\partial J}{\partial z}$)

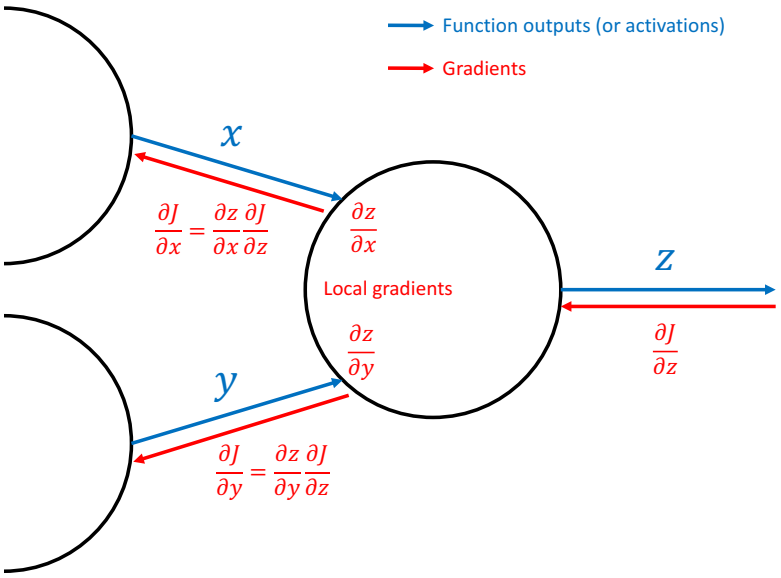


Figure 2.6: An example of backpropagation along a computational graph [5]. During forward pass, vectors x and y are forwarded into a node that 1) compute vector z using some fixed function such as $z = x \odot y$, where $\odot$ is an element-wise multiplication; and 2) compute its local gradients (or Jacobian matrices) $\frac{\partial z}{\partial x}$ and $\frac{\partial z}{\partial y}$. The vector z is then forwarded to the later node in the graph. At the end of the computational graph, the total loss J (a scalar value) is computed. During backward pass, the global gradient on $\frac{\partial J}{\partial z}$ is multiplied with the local gradients, resulting in the global gradients for x ($\frac{\partial J}{\partial x} = \frac{\partial z}{\partial x}\frac{\partial J}{\partial z}$) and y ($\frac{\partial J}{\partial y} = \frac{\partial z}{\partial y}\frac{\partial J}{\partial z}$). These gradients are multiplied (or "chained"), in turn, through functions that compute x and y until the inputs have been reached.

and y ($\frac{\partial J}{\partial y} = \frac{\partial z}{\partial y}\frac{\partial J}{\partial z}$). These gradients are "chained", in turn, through functions that compute x and y until the inputs have been reached. As the inputs for the optimization process are parameters θ, the gradients, therefore, specify how to adjust θ to reduce the loss J.

2.6 Regularization

Typically, in the machine learning application, we only have access to training examples, which are assumed to be independent and identically distributed (i.i.d.) samples from a **data generating distribution** D; i.e., $(\mathbf{x}^{(i)}, y^{(i)}) \sim D$. As we do not have access to all possible samples of D, it is challenging to optimize the objective in Equation 2.1. For instance, there can be many possible functions (i.e., a function parameterized by different parameters θ) that can achieve the same loss under Equation 2.1 (i.e., there is no unique solution), but their *generalization* outside the training data can vary. The question then arises: how do we choose among all possible functions that can achieve the same loss in Equation 2.1?

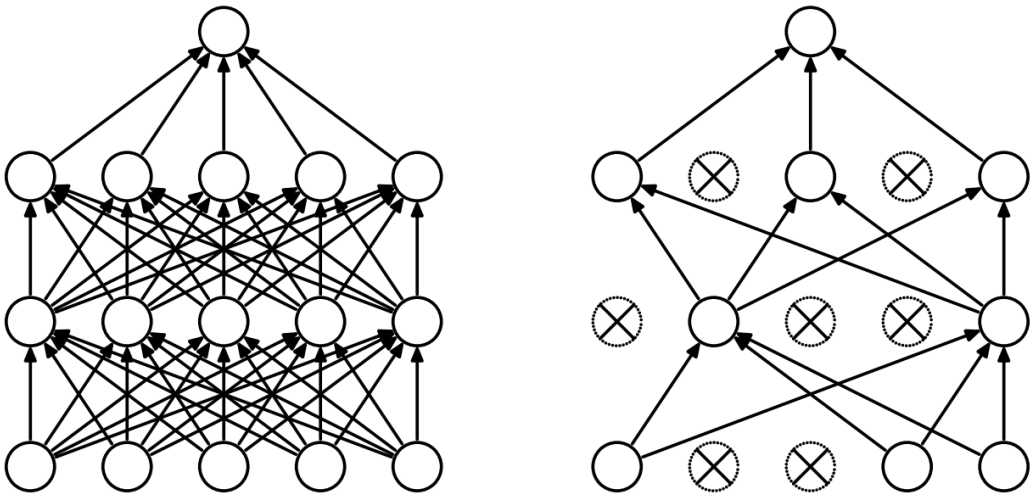
Regularization is introduced to encode preferences for some functions over the others [38]. The preferred functions are commonly selected based on the Occam's razor principle stating that "*Suppose there exist two explanations for an occurrence. In this case, the simpler one is usually better.*" This means that the regularization encourages the simpler solution. Regularization is typically used to the field of machine learning to prevent overfitting.

Weight Decay. One way to encode such preferences into the learning algorithm is to introduce a regularization term that put preferences on smaller θ, independent of their fit to the training data, into the objective as follows:

$$\theta^* = \arg\min_{\theta} \frac{1}{m} \sum_{i=1}^{m} L(f_\theta(\mathbf{x}^{(i)}), y^{(i)}) + \lambda\Omega(\theta) \ , \tag{2.20}$$

where $\Omega(\theta)$ is a parameter norm penalty that drives the θ to be smaller and λ is a scalar value that controls the strength of the penalty. One of the simplest and most commonly used parameter norm penalty is L2 weight decay $\Omega(\theta) = \|\mathbf{W}\|_2^2$. So setting the λ to 0 results in no regularization, and the larger λ restricts the θ to become smaller. This is due to its influence of $\frac{\partial\|\mathbf{W}\|_2^2}{\partial\theta} = 2\mathbf{W}$, given that $\theta = \{\mathbf{W}, b\}$, on the global gradients $\frac{\partial J}{\partial\theta}$ when we are trying to minimize the objective function in Equation 2.20. This weight decay technique give a preference to diffuse weights over peaky weights, encouraging the network to use all of its inputs rather that some of its inputs.

Dropout. There are also other techniques for defining preferences apart from the weight decay. For instance, Dropout [6] has been widely applied to deep neural networks with a larger number of parameters, as it provides a computationally inexpensive but powerful method for regularization. The key idea is to randomly drop units (along with all of their incoming and outgoing connections) from the neural network during training, which is equivalent to sampling a sub-network from a neural network (see Figure 2.7). This technique aims to *approximate the training and evaluating of a bagged ensemble of exponentially many neural networks*, in which k different models are trained using k different datasets. In particular, Dropout samples many sub-networks from the base network, and only update the parameter of the sampled network based on the minibatch-size of input data. During training, dropout randomly retains each neuron (or unit) with a fixed probability p independent of other neurons. During testing, a single base network is used without dropout. Also, as none of the neurons is dropped during test time, the outputs of the neurons from all layers that we apply Dropout must also be multiplied by p. This is to ensure that, for any hidden neurons, the *expected* output



(a) Standard Neural Net (b) After applying dropout.

Figure 2.7: Dropout Neural Network Model [6]. **Left**: a standard (or base) neural network with 2 hidden layers. **Right**: an example of a sub-network sampled from the base network on the left. Dropout trains an ensemble consisting of an exponential number of different sub-networks that can be produced by temporarily removing neurons from an underlying base network (Crossed neurons have been temporarily dropped).

(under the distribution used to drop neurons) at training time is roughly the same as the actual output at test time. With Dropout, a prediction from the model can be interpreted as an averaged prediction across the exponentially many sub-networks, but at a cost of only one forward pass. There has not yet been any theoretical argument for the accuracy of this approximation, but it performs very well in practice [38].

The choices of regularization techniques and their parameters are application-specific and are commonly decided based on the *cross-validation*, which will be discussed in the next section.

2.7 Hyperparameter Selection

Hyperparameters in machine learning are settings that we can specify to control the model's behavior. These parameters are not learned or adjusted by the optimization algorithm. As we previously see from both supervised and unsupervised learning, there are many hyperparameters such as the learning rate α, the number of layers of a neural network, the weights of the regularization λ, and so on, which we need to specify before training.

Validation set. These hyperparameters can be selected based on the model performance on the

validation set, which is typically obtained from a small portion of the training set (e.g., 20%). Note that this validation set should not be the same as the *test* set. The test set is prepared to estimate the generalization error of the model after we obtain the best hyperparameters for the model.

Cross-validation. In some problems, we may not have enough samples to be split into the training, validation and test sets. If the number of examples in the test set is too small, the report performance on the test set may not reflect its actual performance on the given task. An alternative approach is to perform k-fold cross-validation. In this approach, the dataset is partitioned into k *folds* of equal-sized examples. Then one fold is withheld for validation, and the remaining $k-1$ folds are used for training. This process is repeated for k times, with each of the k folds used exactly once as the validation data. The generalization error can be estimated using the average of the validation performance across all folds.

2.8 Evaluation

The choices of performance metrics for evaluating the model are application-specific. Different problems propose different metrics for evaluations. In this section, we discuss some of the most commonly used performance metrics for classification problems that will be used in most chapters of this thesis.

2.8.1 Binary Classification

In binary classification, **accuracy** is the most commonly used performance metrics as it measures the overall performance of a classifier. However, when the number of positive and negative classes are not balanced (e.g., 5% are from the positive class and 95% are from the negative class), the accuracy alone is not sufficient to tell whether our model is performing well or not. For instance, in an extreme case, the bad model that classifies every example as the negative class can get an accuracy of 95%. Note that we typically treat the rare class as the positive class.

Additional performance metrics that we can report in addition to the accuracy are **precision**, **recall**, **specificity**, **false positive rate** and **f1-score**, which can be computed using the values from a confusion matrix (see Table 2.2). With these additional metrics, we can have a better understanding of the performance of the model. Table 2.3 summarizes how to compute each of the performance

Table 2.2: Confusion matrix for binary classification problem.

	Predicted class	
	Positive	Negative
Positive	true positive (tp)	false negative (fn)
Negative	false positive (fp)	true negative (tn)

Table 2.3: Performance metrics for binary classification.

Metric	Equation	Interpretation
Accuracy	$\frac{\text{tp+tn}}{\text{tp+fp+tn+fn}}$	Overall performance of a classifier
Precision	$\frac{\text{tp}}{\text{tp+fp}}$ or $\frac{\text{tp}}{\text{\# predicted pos}}$	Agreement between the true positive labels and the positive labels predicted by a classifier
Recall	$\frac{\text{tp}}{\text{tp+fn}}$ or $\frac{\text{tp}}{\text{\# actual pos}}$	Effectiveness of a classifier to identify positive labels
Specificity	$\frac{\text{tn}}{\text{tn+fp}}$ or $\frac{\text{tn}}{\text{\# actual neg}}$	Effectiveness of a classifier to identify negative labels
False Positive Rate	$\frac{\text{fp}}{\text{tn+fp}}$ or $\frac{\text{fp}}{\text{\# actual neg}}$	Ineffectiveness of a classifier to identify negative labels
F1-score	$2\frac{\text{(Precision)(Recall)}}{\text{Precision+Recall}}$	Summary of the precision and recall that will be high when both precision and recall are high

metrics. Thus, in the previous example, if the model always outputs negative classes regardless of the inputs, the recall will be equal to 0.

The **receiver operating characteristics (ROC) analysis** [7] has also been used to report the classification performance. In this analysis, we plot a ROC curve which depicts tradeoffs between true positives and false positives (i.e., recall and false positive rates in Table 2.3, respectively) as a threshold that a classifier used to make predictions is varied. Each point on the ROC curve represents a true-false positive pair corresponding to a particular decision threshold. Then we estimate the area under the ROC curve (AUC) [57] to reduce the ROC plot into a single scalar value (ranging from 0 to 1) that we can use to compare the performance of different classifiers. A model that achieves the largest AUC is the model that gives the best average classification performance. This ROC curve is insensitive to changes in class distribution, which makes it a good classification performance metric when we have the imbalance-class problem. Figure 2.8 illustrates an example of a comparison between the area under two ROC curves [7].

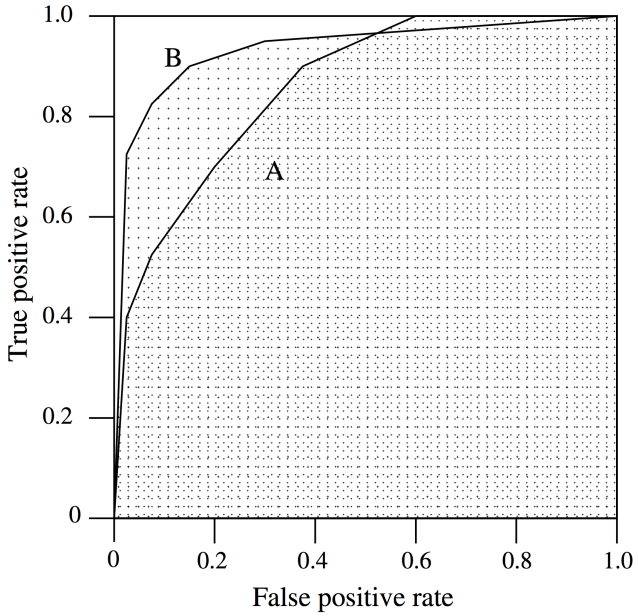


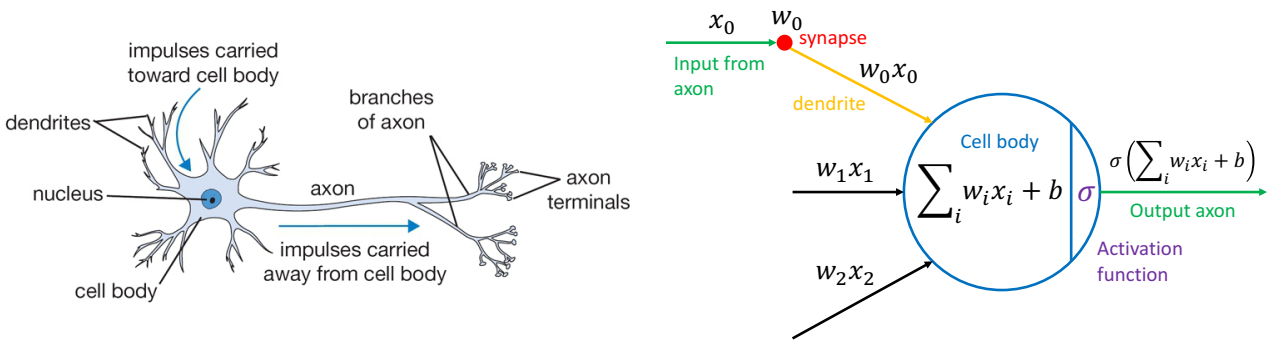
Figure 2.8: An example of a comparison between the area under two ROC curves [7]. In this figure, the classifier B has a larger AUC than the classifier A, which means B achieves a better average classification performance. True positive rate and false positive rate are the recall and false positive rates in Table 2.3, respectively. The bottom-left point (0,0) means a classifier never predicts the positive classes (i.e., no false positive and no true positive). The top-right point (1,1) means a classifier always predicts the positive classes. The top-left point (0,1) represents the perfect classification.

2.8.2 Multi-class Classification

In multi-class classification problem, we are typically interested in **overall accuracy**, **per-class precision**, **per-class recall**, **per-class F1-score** and **macro F1-score**. These metrics have the similar interpretation, except that we are now in the multi-class classification problem. The overall accuracy is computed by dividing the sum of true positives from all classes with the total number of examples. The per-class precision, recall and F1-score can be computed in the same ways as in Table 2.3. The macro F1-score provides a summary of the precision and recall over all classes. It is computed in the same way as F1-score in Table 2.3, except that it uses the average of per-class precision and recall [58].

2.9 Deep Learning

Deep learning, also called neural network, is a branch of machine learning that utilizes multiple layers of linear and non-linear functions to transform from inputs into **representations** that are useful for subsequent tasks such as classification and regression. It can be trained to approximate function



(a) Cartoon drawing of a biological neuron **(b)** Mathematical model

Figure 2.9: An example of (a) a cartoon drawing of a biological neuron and (b) its mathematical model [8]. An input signal travels along the axons (e.g., x_0), which is then interacted with dendrites of the other neuron (e.g., w_0x_0) based on the synaptic strength. There are synapses (e.g., w_0) between the axon and the dendrite that control the strength of influence of one neuron to another. The dendrites carry the weighted input signals to the cell body. In the biological model, the cell body accumulates the results and sums them. If the sum is greater than a certain threshold, the neuron can fire which sends a spike along its axon. In mathematical model, the cell body accumulates the results, sums them, and then applies an element-wise function σ to fire an output signal ($\sigma\left(\sum_i w_i x_i + b\right)$, where i is the index of the input and weight vector) via an output axon. If the sigmoid activation is used, the output signal can be interpreted as the average firing rate of the neuron. Finally, the axon propagates the signals along its branches to the dendrites of other neurons.

f_θ in both supervised and unsupervised settings. For example, in the classification problem, a deep learning model can be trained to approximate a mapping $\mathbf{y} = f_\theta(\mathbf{x})$ by adjusting the parameters θ that minimizes a given loss function, as mentioned in Section 2.2.

Biological Inspiration. The neural network is originally inspired by the way biological nervous systems process information. Figure 2.9 illustrates an example of a cartoon drawing of a biological neuron and the mathematical model of a single neuron (or a hidden unit) in a neural network. Each neuron receives input signals (the input vector $\mathbf{x}$) from its dendrites and produces output signals along its axon. The dendrites carry the signals to the cell body, in which the strengths of the input signals from the other neurons are controlled by synapses (the weights $\mathbf{w}$). The cell body accumulates the signals and sums them. If the sum is greater than a certain threshold, the neuron can fire which sends a spike along its axon. The axon propagates the signals along its branches to the dendrites of other neurons. In the mathematical model, the synaptic strengths (the weights $\mathbf{w}$) are learnable, which control the strength of influence of one neuron on another. If the activation function is sigmoid non-linearity function (σ) that squashes it to range between 0 and 1, the output signals can be interpreted as the average firing rate of the neuron. The mathematical neuron represents each neuron in neuron networks that we will discuss in the following sections.

Concepts of Layers. The neural network is built by connecting these neurons. These neurons are commonly organized in layers as they allow us to efficiently calculate **activations** of all neurons in each layer with a *matrix multiplication*. The neural network can be viewed as a directed acyclic graph describing how a sequence of layers (i.e., functions) are combined to form the network.

Power of Depth. Making a neural network model deeper is believed to enable the network to reuse features from the top layers (closer to the input layer), and can lead to learning more abstract features at the bottom layers (closer to the output layer) [49]. For instance, in the supervised learning, we can think of the last layer of the network as a learning classifier, such as a softmax classifier, and the remaining layers above as they are trying to provide representations to this classifier. Each layer learns the representations that are useful for subsequent tasks defined by a loss function which provides gradients via backpropagation.

In the following sections, we will first introduce loss (or cost) functions for training deep learning models for both classification and regression problems. Then we will discuss commonly used activation functions in deep learning models, and go through different types of neural networks or layers that can be stacked on top of each other to form a powerful model to perform a given task.

2.9.1 Loss Functions

Loss functions are scalar-valued functions defined to quantified the *error* or *disagreement* between the predicted values and the target values. For classification problems, the most commonly used loss functions are logistic regression and cross-entropy losses. For regression problems, the mean squared error loss function is mostly used.

Throughout this section, we will treat a neural network as a model consisting of two parts.

1. The first part consists of input and hidden layers that transforms from an input $\mathbf{x}$ into a hidden feature $\mathbf{h}$ (i.e., $\mathbf{h} = f_\theta(\mathbf{x})$).
2. The second part is the output (or the last) layer that provides some additional transformation from the feature to complete the task that the network must perform such as classification or regression.

Mean Squared Error loss. It is commonly used for regression, in which the output variable takes continuous values. In this case, we can design a deep learning model with the output layer that

has the number of neurons (or units) equal to the number of dimensions of the targets and a linear activation function (i.e., no nonlinearity). In particular, we use a linear layer to compute an *affine transformation*:

$$\hat{\mathbf{y}} = \mathbf{W}^T\mathbf{h} + \mathbf{b} \ , \tag{2.21}$$

where $\mathbf{W}$ and $\mathbf{b}$ are the weights and biases of the output layer, $\hat{\mathbf{y}}$ is the output vector of the model. We can then measure the performance of the model by computing the mean squared error loss:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\left(\hat{\mathbf{y}}^{(i)} - \mathbf{y}^{(i)}\right)^2 \ . \tag{2.22}$$

Logistic Regression loss. It is commonly used for binary classification, in which there are only two classes, class 0 and class 1. In this case, we just need to specify the probability of one of these classes p as another one can be computed from $1-p$ (i.e., the probabilities of these two values must add up to 1). We can design a deep learning model to output the probability $P\left(y=1|\mathbf{x};\theta\right)$ by using the output layer with one unit that squashes its output value into a range of 0 and 1:

$$\hat{y} = \sigma\left(\mathbf{W}^T\mathbf{h} + b\right) \ , \tag{2.23}$$

where $\mathbf{W}$ and b are the weights and bias of the output layer, and σ is the *sigmoid* function, which will be described in Section 2.9.2. This sigmoid output unit can be thought of as having two components. The first component is a linear layer to compute $\mathbf{z} = \mathbf{W}^T\mathbf{h} + b$. The second component is the sigmoid function to squash $\mathbf{z}$ into the range of 0 and 1, which is interpreted as a probability $P\left(y=1|\mathbf{x};\theta\right)$. Thus the probability of each class can be computed as follows:

$$P\left(y=1|\mathbf{x};\theta\right) = \hat{y} \ , \tag{2.24}$$

$$P\left(y=0|\mathbf{x};\theta\right) = 1-\hat{y} \ . \tag{2.25}$$

These two equations can be written compactly as:

$$p\left(y|\mathbf{x};\theta\right) = (\hat{y})^y(1-\hat{y})^{1-y} \ . \tag{2.26}$$

With this setup, we can train the model with maximum likelihood. In particular, suppose there are

m training examples drawn independently from the data-generating distribution $\left\{(\mathbf{x}^{(i)}, y^{(i)})\right\}_{i=1}^{m}$. If X represents all inputs and Y represents all targets, we can compute the likelihood of the parameters $\mathcal{L}(\theta; X, Y)$ as follows:

$$\mathcal{L}(\theta; X, Y) = \prod_{i=1}^{m} p\left(y^{(i)}|\mathbf{x}^{(i)}; \theta\right) \tag{2.27}$$

$$= \prod_{i=1}^{m} (\hat{y}^{(i)})^{y^{(i)}} (1 - \hat{y}^{(i)})^{1-y^{(i)}} \ . \tag{2.28}$$

Instead of trying to maximize this likelihood computed from the product over many probabilities, it is easier and less prone to numerical underflow [38] if we maximize the following log-likelihood:

$$\log \mathcal{L}(\theta; X, Y) = \sum_{i=1}^{m} y^{(i)} \log(\hat{y}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}) \ . \tag{2.29}$$

Thus the logistic regression loss is simply the negative log-likelihood:

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^{m} y^{(i)} \log(\hat{y}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}) \ . \tag{2.30}$$

Cross-entropy loss. It is commonly used for multi-class classification, in which there are C classes. In this case, we can design a deep learning model to output the probability distribution over C different classes. This can be done by using the output layer with the number of units equal to the number of classes and a *softmax* function. In particular, we first use a linear layer to compute:

$$\mathbf{z} = \mathbf{W}^T \mathbf{h} + \mathbf{b} \ , \tag{2.31}$$

where $\mathbf{W}$ and $\mathbf{b}$ are the weights and biases of the output layer, and $\mathbf{z}$ is interpreted as a vector of *unnormalized log probabilities*. The softmax function can then exponentiate and normalize $\mathbf{z}$ to obtain a vector of the probability distribution over C classes $\hat{\mathbf{y}}$, with $\hat{y}_i = P(y = i|\mathbf{x})$ as follows:

$$\hat{y}_i = \text{softmax}(\mathbf{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)} \ . \tag{2.32}$$

where the subscript i and j is the i- and j-th element of a vector. This softmax function not only squashes the output value of each unit to be between 0 and 1 but also normalizes the value so that the outputs from all units sum to 1. The softmax function can be seen as the generalization of

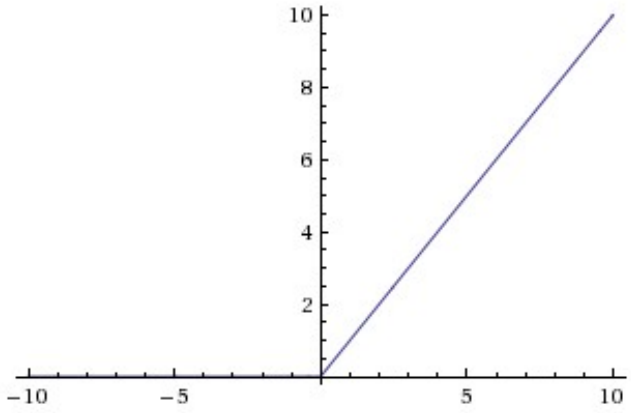


Figure 2.10: ReLU activation function outputs zero when $z_i < 0$ and linear with slope 1 when $z_i > 0$ [8].

the sigmoid function from binary to multi-class classification [38]. Similar to the logistic regression, given m training examples that are independent, we can train a model with maximum likelihood by minimizing the negative log-likelihood of the target class. Thus the cross-entropy loss is:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\log(\hat{y}^{(i)}_{y^{(i)}}) \ . \tag{2.33}$$

2.9.2 Activation Functions

The design of activation functions is an active research area and does not yet have definitive guides. So far, we have mentioned some of the activation functions that can be applied to add non-linearity into a deep learning model. In this section, we will list the activation functions that have been applied in this thesis and some of the basic intuitions motivating each choice of the activation function.

It should be noted that the intuitions we mention here can help decide when to try which function. The best choice of activation function requires the process of trial and error with the validation set as mentioned in Section 2.7.

Most activation functions can be described as accepting a vector of inputs $\mathbf{x}$, computing an affine transformation $\mathbf{z} = \mathbf{W}^T\mathbf{x} + \mathbf{b}$, and then applying an *element-wise* non-linear function $g(\mathbf{z})$. Most of the functions differ from each other only by the choice of the non-linear function $g(\mathbf{z})$.

Rectified Linear Unit. A Rectified Linear Unit (ReLU) [38] uses the following non-linear function

(see Figure 2.10):

$$g(\mathbf{z})_i = \max\{0, z_i\} \ . \tag{2.34}$$

The ReLU has become a default choice for the activation functions of the hidden layers. The hidden units with the ReLU function are easy to optimize because when the units are active, the gradients are not only large but also consistent. However, there is still one drawback to the rectified linear units as there is no gradient for examples whose activation is zero. This means that these units cannot learn via gradient-based methods (such as gradient descent) on these examples. Several generalizations of the ReLU has been proposed to address this problem using a non-zero slope α_i when $z_i < 0$:

$$g(\mathbf{z}, \boldsymbol{\alpha})_i = \max(0, z_i) + \alpha_i \min(0, z_i) \ . \tag{2.35}$$

For instance, a **leaky ReLU** [59] that uses a small fixed-value α_i such as 0.01 or a **parametric ReLU** or **PReLU** [60] that treats α_i as a trainable parameter. Nevertheless, there has not yet been any substantial evidences suggesting that these generalizations can yield better results compared to the ReLU.

Logistic Sigmoid. A logistic sigmoid (or, simply, sigmoid) activation function uses the following non-linear function (see Figure 2.11):

$$g(\mathbf{z})_i = \frac{1}{1 + \exp(-z_i)} \ . \tag{2.36}$$

The sigmoid function takes an input and squashes it into range between 0 and 1. Historically, prior to the introduction of ReLU, the sigmoid function has been widely used in most neural networks due to its nice interpretation of the firing rate of a neuron. However, the sigmoid function is now rarely used as the activation function of hidden layers because it can saturates and kill gradients. Specifically, when z_i is very positive or negative, the activation saturates at either tail of 1 and 0, respectively. As the gradients at these regions is almost zero, they can kill the gradients during backpropagation. This saturation can make gradient-based learning become very difficult. Therefore, the use of sigmoid function as the activations of the hidden layers is now discouraged.

Hyperbolic Tangent. A hyperbolic tangent (tanh) activation function uses the following non-linear function (see Figure 2.12):

$$g(\mathbf{z})_i = \tanh(z_i) = 2 \cdot \text{sigmoid}(2z_i) - 1 \ , \tag{2.37}$$

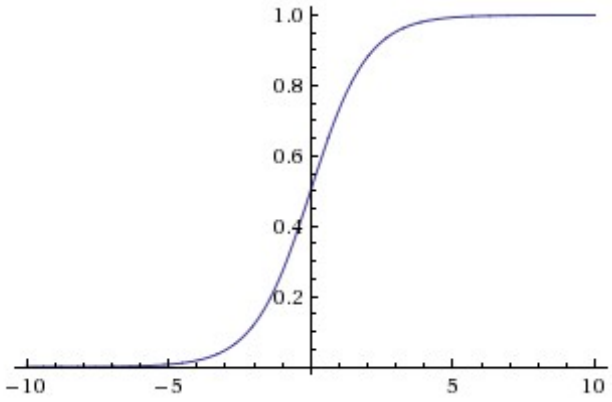


Figure 2.11: Sigmoid activation function squashes z_i to range between 0 and 1 [8].

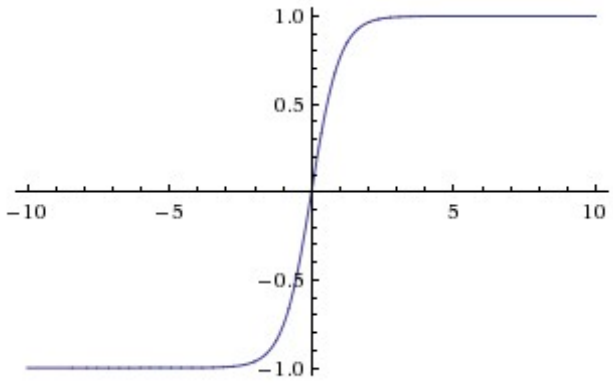


Figure 2.12: Tanh activation function squashes z_i to range between -1 and 1 [8].

where sigmoid is the sigmoid function from Equation 2.36. The tanh function is closely related to the sigmoid function that takes an input and squashes it into range between -1 and 1. In practice, this tanh function is typically preferred when a sigmoid function must be used [38]. This is mainly because the tanh outputs are zero-centered, which resembles the identity function when z_i is close to 0. Therefore, the gradient-based training becomes easier compared to the sigmoid function.

2.9.3 Fully-connected Neural Networks

A fully-connected neural network is the simplest form of neural networks. Each neuron between two adjacent layers are fully pair-wise connected, and there is no connection between the neurons in the same layer. Figure 2.13 illustrate an example of a 3-layer fully-connected neural network. This network can be implemented with a repeating sequence of matrix multiplications and element-wise non-linear

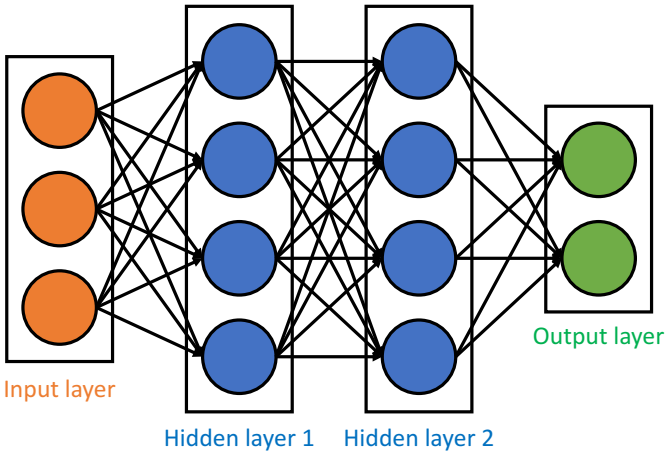


Figure 2.13: An example of a 3-layer fully-connected neural network (only count the number of hidden and output layers). There are one input layers of neurons (or units), two hidden layers of four neurons each, and one output layers of two neurons.

activations as follows:

$$f_\theta(\mathbf{x}) = f_{\theta_3}(f_{\theta_2}(f_{\theta_1}(\mathbf{x}))) = \mathbf{W}_3\sigma(\mathbf{W}_2\sigma(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2) + \mathbf{b}_3 \ , \tag{2.38}$$

where f_{θ_1} is the first hidden layer, f_{θ_2} is the second hidden layer, f_{θ_3} is the output layer, $\theta = \{\theta_1, \theta_2, \theta_3\}$, and $\theta_l = \{\mathbf{W}_l, \mathbf{b}_l\}$. The typical choices of activation function σ are, for example, identity function (i.e., no non-linearity), sigmoid function, tanh function, and rectified linear unit (ReLU) function. The last layer normally use the identify function, as most of the loss functions have already included a non-linearity function such as the cross-entropy loss.

2.9.4 Convolutional Neural Networks

A convolutional neural network (CNN) [8, 61] is a neural network that is specifically designed for processing data that has a grid-like topology or a structure such as biosignals (1D grid taking samples at regular time interval) and images (2D grid of pixels). Typically these data would have a large number of input dimensions. For example, a 300×300 RGB images will have the total number of input values of 300×300×3 (i.e., 300×300 pixel values from red, green and blue channels), and a 30-s epoch of EEG from 20 channels with a sampling rate of 256 Hz will have the total number of (30×256)×20. In these cases, the fully-connected neural network would not scale well to full input dimensions. For example, if the input is the RGB image above, there would be 300×300×3 weights in each neuron of the first hidden layer and, even worse, there would be many of such neurons depending on the number

of hidden units. This large number of parameters can lead to a very expensive computation and overfitting. The CNN, instead, employs two special schemes: **local connectivity** and **parameter sharing** to alleviate such problems. Before we go into the details of the CNN, we first describe what convolution is.

Convolution Operation

Generally, convolution is an operation on two functions (e.g., x and w) to produce a third function (e.g., s). It is typically denoted with asterisk or star ($*$), which can be computed as follows:

$$s(t) = (x * w)(t) = \int x(a)w(t-a)da. \tag{2.39}$$

The convolution can be described as a time-series of one signal weighted by another signal that are flipped backward and slided along the first signal. Suppose the function $x(t)$ represents a single-channel EEG signal that gives the reading of the brain activity at time t, and the function $w(a)$ represents a weighting function that gives more weight to recent measurements, where a is the age of a measurement. The resulting function $s(t)$ would be a smoothed estimate of the brain activity. In CNN terminology, the function x in the convolution is referred to as the **input**. The function w is referred to as the **filter** or **kernel** that can be updated by a learning algorithm. The output function (i.e., the function s) is referred to as the **feature map**. As convolution is commutative, the following operation also gives the same output function:

$$s(t) = (w * x)(t) = \int x(t-a)w(a)da. \tag{2.40}$$

Usually, when we process the data on a computer, time will be discretized, and the sensor to record the EEG signal would provide data at regular interval (e.g., depending on the sampling rate). Thus it is more common to use the discrete convolution:

$$s(t) = (w * x)(t) = \sum_{a=-\infty}^{\infty} x(t-a)w(a). \tag{2.41}$$

Also the convolution can be applied to more than one axis. For example, if we have an input image I that has two dimensions, we can use a two-dimension filter K to convolves the input as follows:

$$S(i,j) = (K * I)(t) = \sum_m \sum_n I(i-m, j-n)K(m,n). \tag{2.42}$$

In the machine learning application, the convolution operation used in the CNN does not correspond precisely to this definition. Many deep learning libraries typically implement a *cross-correlation*, but call it convolution:

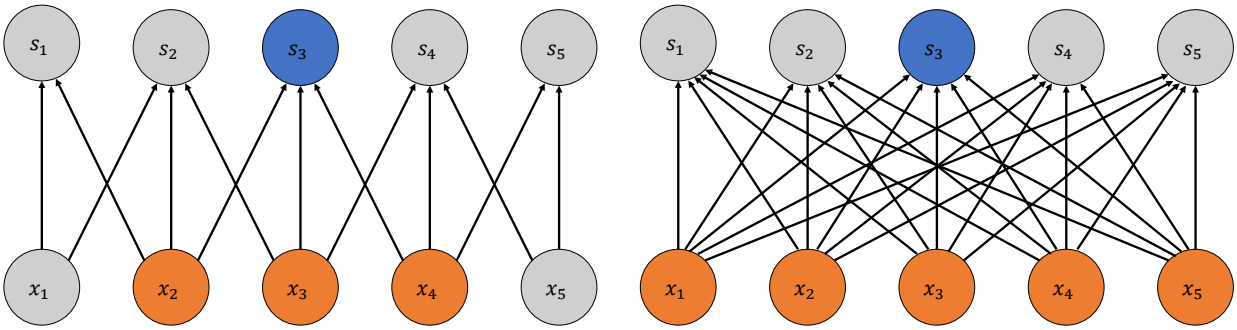
$$S(i,j) = (K * I)(t) = \sum_m \sum_n I(i+m, j+n)K(m,n). \tag{2.43}$$

The difference is that cross-correlation does not flip the kernel. As a learning algorithm will learn the appropriate values of the filter in the appropriate place, a model that implements convolution with filter flipping will learn a filter that is flipped relative to the filter learned by a model without filter flipping. Later, we will explain the other two features of the convolution that are specific to the CNN implementation: **zero-padding** and **stride**.

Convolutional Layer

The CNN implements the two schemes, local connectivity and parameter sharing, in the form of a computational building block, called **convolutional layer**.

Local connectivity. The convolutional layer implements the local connectivity scheme by restricting the size of the filters to be smaller than the input. Thus the results of each dot product between the filter and the corresponding local part of the input (i.e., one element of a feature map) can be considered as the output of one neuron that is connected to only the local part (i.e., the same size of the filter) of the input. This is different from the fully-connected layers that use matrix multiplications between the input and the weight matrices, meaning that every output unit interacts with every input unit. This can significantly reduce the number of parameters and the computation. Figure 2.14 illustrates the difference between the local and full connectivity. In the convolutional layer, each output unit are affected by three input units, and these units are known as the **receptive field** of that output unit. In the fully-connected layer, each output unit are affected by all input units. When we stack multiple convolutional layers on top of each other (i.e., deep convolutional neural networks), units in the deeper layers may *indirectly* connect to a larger portion of input units, as shown in Figure 2.15. This enables the network to efficiently describe complicated interactions between many units even though each unit



(a) Local Connectivity. **(b)** Full Connectivity.

Figure 2.14: Local vs Full Connectivity. In (a), **s** is computed by convolution with a filter of size (or width) 3. For example, only three input x_2, x_3 and x_4 (green circle) affects the output s_3 (blue circle), and these units are known as the **receptive field** of s_3. In (b), **s** is computed by matrix multiplication and all inputs affect s_3

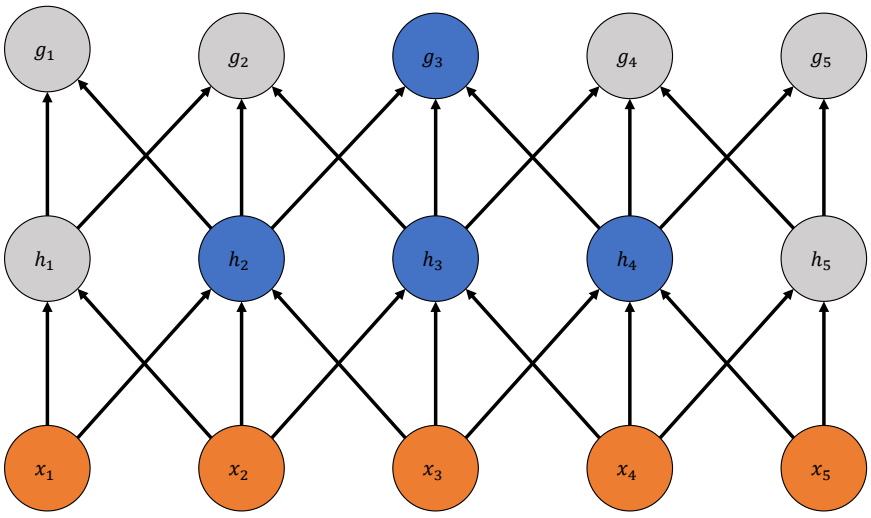


Figure 2.15: The receptive field of the units at the deeper convolutional layers is larger than the units at the shallow layers. In this example, even though each convolutional layer only connect to 3 input units, the output unit g_3 may *indirectly* connect all or most of the input units **x**.

in the convolutional layer only connects a small portion of input units.

Parameter sharing. The convolutional layer implements the parameter sharing scheme by convolving the same filter across all spatial positions of the input to generate a feature map. This means that instead of learning a separate set of filters (i.e., parameters) for every spatial position of the input, the convolutional layer constraints itself to only learn one set of filters. By sharing parameters among neighboring neurons in the same feature map, we assume that the feature that is useful for some spatial positions should also be useful for other positions as well. This does not affect the computation time of forward propagation, but it further reduces the memory requirements of the model in addition to the local connectivity.

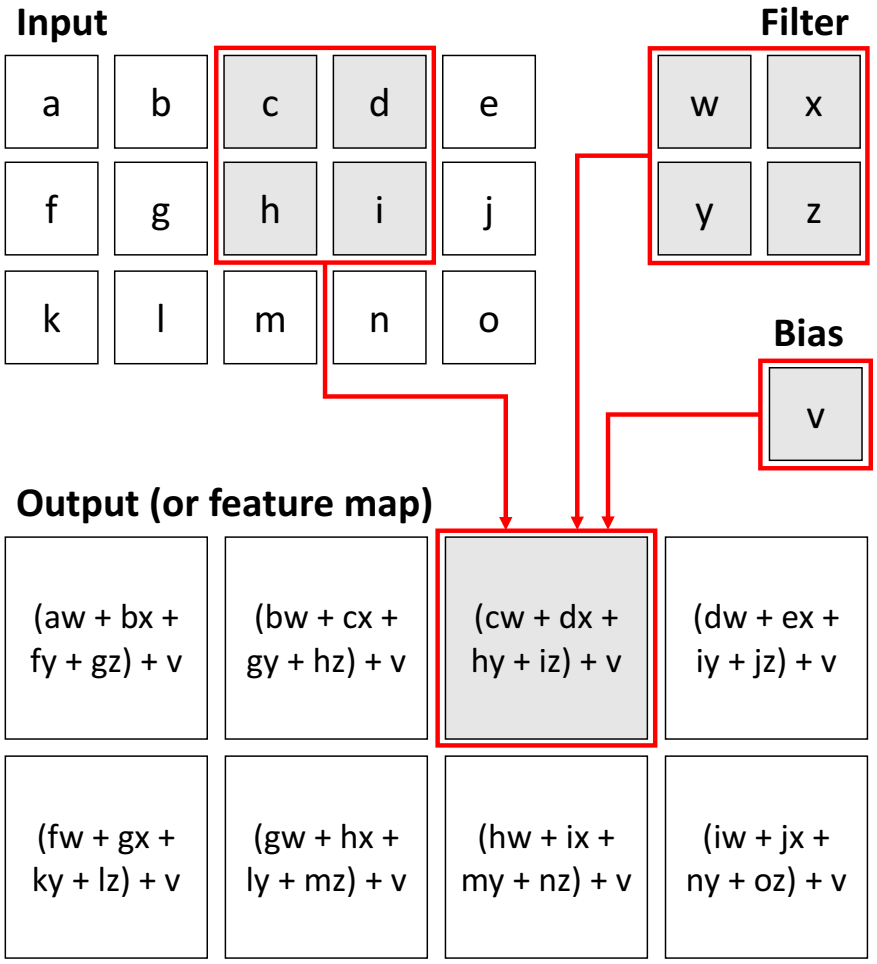


Figure 2.16: An example of convolving a 3×5 input tensor with a 2×2 filter. Each element of the feature map is obtained by computing a dot product between the filter and the corresponding local part of the input tensor and then adding the result with a bias. The filter is slid with a stride size of one across all positions that the filter can lie entirely within the input tensor (i.e., no zero-padding). This result in the feature map of size 2×4.

In summary, each convolutional layer takes a multi-dimensional input array (i.e., an input tensor[1]) and *convolves* it with a set of *trainable filters*, each of which is also a tensor whose size is smaller than the input. In particular, it slides each filter across all spatial positions of the input tensor and computes a dot product between the filter and the local part of the input. Each filter can also be thought of as the weights of each neuron in the convolutional layer that are shared across the spatial positions of the input tensor, but the terms *filter* or *kernel* are more commonly used. Figure 2.16 illustrates an example of convolving a 3×5 input tensor with a 2×2 filter. Convolving each filter results in a *feature map*. The feature maps from all filters of the convolutional layer are stacked to produce an output tensor. Intuitively, this layer is trying to learn filters that detect (or "activate") when they see certain features in any local spatial positions of the input tensor. As a CNN can be trained with backpropagation, it can learn to filters that are useful for a given task based on the training data. Note that it is possible to apply an element-wise non-linear transformation (as discussed in Section 2.9.2) to the output tensor.

Output arrangement. In the convolutional layer, there are three hyperparameters that can be used to control the size of the output tensor:

1. **Stride.** It is the step size to slide the filters during the convolution operation. When the stride is 1, we move the filters one element at a time, which is the basic convolution operation. With the stride size of more than one, we move the filters multiple elements at a time (i.e., we are skipping the computation at some spatial locations) as we slide them around the input tensor. This can reduce the spatial size of the output tensor and the computational cost at the expense of not finely extracting features.

2. **Zero-padding.** It is the number of zeros padded at the border of the input tensor, allowing us to slide the filters at the border of the input tensor. Without this zero-padding, the size of the output tensor will shrink (the shrinking size depends on the width of the filter) at each convolutional layer. This limits the number of convolutional layers to be included in a model. In the case of an input image, as convolutional layers are added, the spatial dimension of the network will eventually reduce to 1x1. Zero-padding enables us to control the size of the output tensor and the size of the filter independently. This means that there can be as many convolutional layers as desired.

[1]The word *tensor* is used in different ways in different fields. In the context of machine learning, a definition of a tensor is a multi-dimensional grid of numbers.

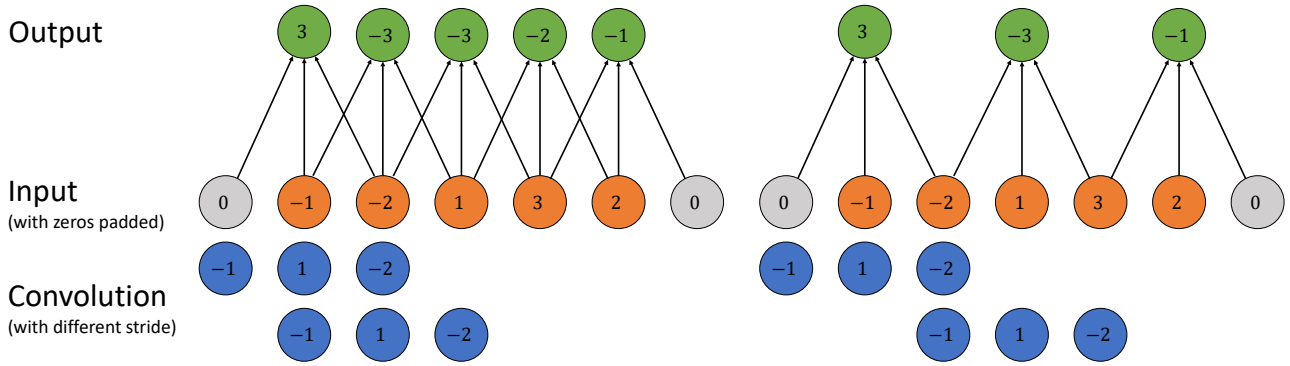


Figure 2.17: Convolution with zero-padding and strides of different sizes. In this example, an input vector of size 5 is padded with one zeros on both size and is convolved with a filter of size 3. **Left:** Convolution with a stride size of 1 results in an output vector of size 5, which is equal to the size of the input vector. **Right:** Convolution with a stride of size 2 results in an output vector of size 3. This can be seen as we are skipping some positions of the filter to reduce the spatial size of the output tensor (i.e., *downsampling*) and the computational cost at the expense of not finely extracting features.

3. **Number of filters.** It is the number of filters (or kernels) that the convolutional layer uses to slide across the input tensor. Each filter produces one feature map. The feature maps from all filters are stacked to form an output tensor. Thus the number of filters corresponds to the number of filter maps, which represents one dimension in the output tensor.

Figure 2.17 shows examples of convolution with zero-padding and strides of different sizes. We can compute the size of an output tensor as a function of the size of the input tensor, the size of the filters, the stride and the amount of zeros padded as following. Suppose the convolutional layer receives an input tensor of size $W_1 \times H_1 \times D_1$ (note that, in the case of biosignals, H_1 is always equal to one and D_1 is equal to the number of signals or channels). It generates an output tensor of size $W_2 \times H_2 \times D_2$, where $W_2 = ((W_1 - F_W + 2P)/S) + 1$, $H_2 = ((H_1 - F_H + 2P)/S) + 1$, D_2 is the number of filters of the convolutional layer, $F_W \times F_H$ is the filter size (i.e., the receptive field), P is the number of zeros padded to one side of the input border, and S is the stride size. Note that the d-th output tensor ($W_2 \times H_2$) is the feature map computed using the d-th filter and d-th bias. The D_2, F, P and S are hyperparameters of the convolutional layer. The number of parameters in each filter is $F_W * F_H * D_1$ (note that the filter always span to the full depth of the input tensor D_1). The total number of parameters of the convolutional layer is $(F_W * F_H * D_1) * D_2$ weights plus D_2 biases.

Concrete example. Suppose an input tensor is a 30-s epoch of EEG from two channels with a sampling rate of 256 Hz (i.e., a (30×256)×1×2 tensor), and there are 32 filters of size 256×1×2 in a convolutional layer. If we do pad the border of the input tensor with 64 and the stride size is 128, the

output of this convolutional layer is a $60\times1\times32$ (see Figure 2.18). In this example, the total number of weight and bias parameters is $256*1*2*32+32=16,416$. On the other hand, if this layer is a fully-connected layer with the number of hidden units equal to the output size of the convolutional layer, the total number of weight and bias parameters would be $(30*256*1*2+1)*(60*1*32)=29,493,120$. This very large number of parameters would make the model suffer from overfitting.

Interpretation on biosignal applications. When applying a 1D convolutional layer with multiple filters on an epoch of single-channel EEG, it can be described we are trying to something similar to the short-time Fourier transform (STFT) [39]. Imagine that each filter is a sine wave of one frequency. By convolving a set of sine waves with different frequencies over the EEG signal, we are trying to determine the degree to which each frequency is similar to the EEG signal (ignoring phases here as there is no complex component). The main difference is that we do not fix or predefine the filters just like STFT, but we instead learn filters that are useful for a given task based on the training data.

Pooling layers

In CNN, there is also another component that helps alleviate the overfitting problem, called **pooling layer**. This layer takes an input tensor and downsamples it by applying a summary statistics such as maximum (also called **max pooling** [62]) to the local part of each feature map independently. Figure 2.19 illustrates an example of a pooling layer with a pool width of 3 and a stride between pools of 2. This pooling layer helps make the representation approximately *invariant to local translations of the input* [38]. Such invariance to translation can be useful when we are more interested in whether some feature is present than where they are. For instance, when we are determining whether an image contains a human face, we care more about whether there is an eye of each side of the human face (left and right) than the exact location of both eyes. The pooling layers can also reduce the number of inputs to be processed by the next layer (e.g., the fully-connected layers which based on matrix multiplication) without introducing any additional parameters. This helps improve the computational efficiency and reduce the memory requirements for storing the parameters of a model.

Network architecture

A convolutional neural network typically consists of multiple convolutional layers (CONV) and, perhaps, interleave with some pooling layers (POOL). Towards the end of the networks before the output

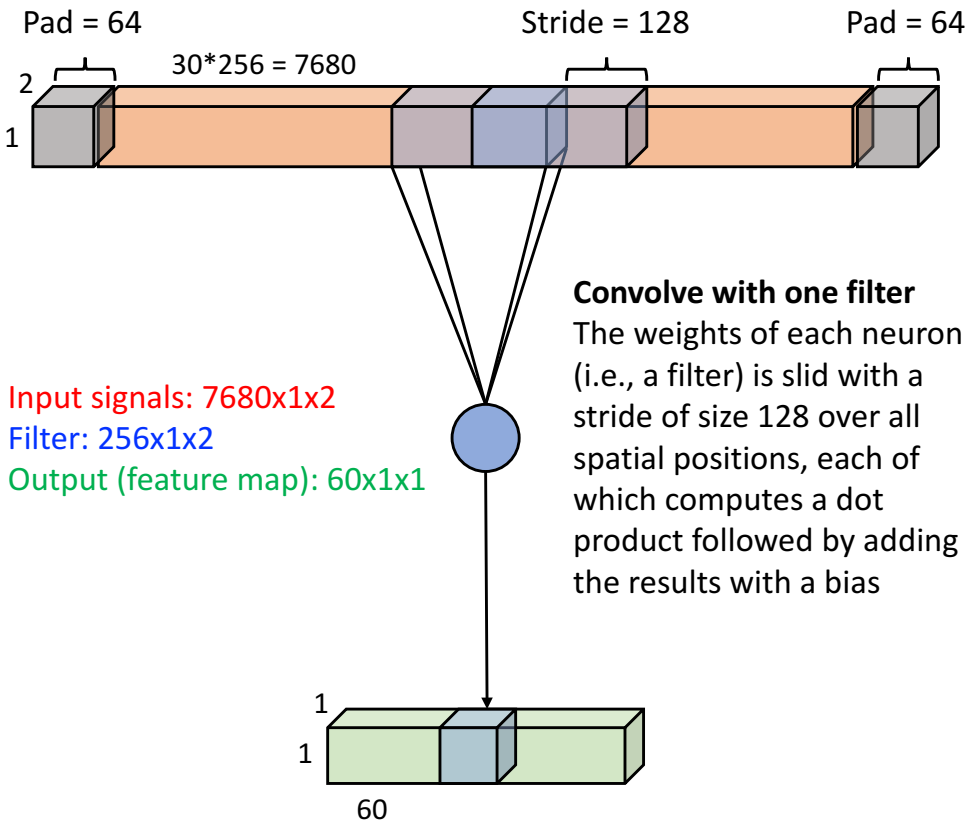


Figure 2.18: An example of convolving a 256×1×2 filter over a 30-s epoch of EEG from two channels with the sampling rate of 256 Hz (i.e., a 7680×1×2 input tensor). Using a padding of size 64 on the border of the input and a stride of size 128, there are 60 spatial positions, each of which computes a dot product between the filter and the local part of the input followed by adding the result with a bias, resulting in a feature map of size 60×1×1. As there is more than one filter in the convolutional layer (e.g., 32 filters), each of them will independently perform convolution in the same way, resulting in a set of feature maps stacked along the depth (e.g., a 60×1×32 output tensor).

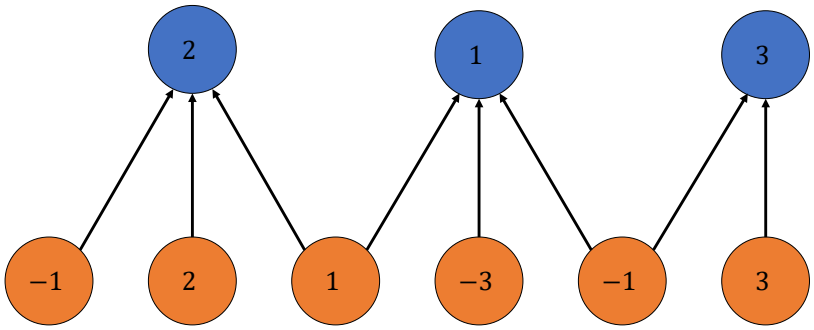


Figure 2.19: Max pooling layer with a pool width of 3 and a stride size of 2. This pooling layer downsamples an input vector of size 6 into an output vector of size 3 using the maximum operation. Even though the rightmost pooling has a size smaller than the pool width, we have to include this so that we do not ignore some elements at the border of the input vector.

layer, we usually stack with several fully-connected layers (FC). As we can apply an element-wise non-linearity function to the output the convolutional and fully-connected layers, we will explicitly write the name of the activation function as a layer to specify which non-linear function is used at which CONV layer (e.g., RELU for a ReLU activation function). If the CNN is for classification, a softmax classifier (SOFTMAX) is commonly used as the final output layer. A possible sequence of layers for a CNN can be:

INPUT → [CONV → RELU → POOL]*2 → FC → RELU → SOFTMAX

where INPUT denotes an input tensor and * indicates repetition. It is also mentioned in [8] that a stack of small filter CONV layers is preferred over one larger filter CONV layer because of the two main reasons. First, stacking CONV layers with smaller filters allows us to express more powerful features of the input, as there can be multiple non-linearities at the end of each CONV layer. Second, it can reduce the number of parameters of a model, while still maintaining the same receptive field as one larger filter. This is because a neuron at the deeper layer can be thought of as indirectly connect to a large number of units at the layer below (see Figure 2.15). Therefore another possible sequence of layers for a CNN can be:

INPUT → [CONV → RELU → CONV → RELU → POOL]*2 → FC → RELU → SOFTMAX

This stack of small filter CONV layers has one practical drawback in that we may need more memory to temporarily store all intermediate values during backpropagation.

2.9.5 Recurrent Neural Networks

Recurrent neural networks (RNNs) [38, 63] are a family of neural networks that are specialized for processing sequential data, such as sentences and biosignals. Just as CNNs that are specialized for processing a grid of values such as an image with large width and height, RNNs can is specialized for processing a long sequence of values $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(\tau)}$ that would be impractical for networks without sequence-based specialization. Most RNNs use the following equation to define the value of their hidden units or what is called **hidden state**, $\mathbf{h}$, as follows:

$$\mathbf{h}^{(t)} = f_\theta(\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}) \tag{2.44}$$

where t is the time step index (we use t instead of i to introduce the notion of time here), f_θ is a **recurrence formula** that is used to update the current hidden state $\mathbf{h}^{(t)}$ based on the previous state $\mathbf{h}^{(t-1)}$ and the current input data $\mathbf{x}^{(t)}$. $\mathbf{h}^{(0)}$ is an initialize state, which is commonly treated as either a zero vector (i.e., $\mathbf{h}^{(0)} = \vec{0}$) or as parameters and learn the initial state that are useful for particular tasks. The recurrent formula f_θ is one of the most important elements of RNNs which defines how do we update the hidden state $\mathbf{h}^{(t)}$ using $\mathbf{h}^{(t-1)}$ and $\mathbf{x}^{(t)}$. The mathematical form of f_θ varies from model to model, which will be discussed later in this section. Also, RNNs will typically have additional output layers that read information from the state $\mathbf{h}$ to make predictions, which will be discussed later in this section.

When the RNN is trained to perform a task such as predicting the future from the past, it typically learns to encode a kind of *lossy summary* of the past sequence of inputs up to time step t into state $\mathbf{h}^{(t)}$. Generally, this summary is necessarily lossy because it summarizes the task-relevant features, depending on the training criterion, of an arbitrary length of the past sequence $\{\mathbf{x}^{(1)}, ..., \mathbf{x}^{(t)}\}$ into a fixed length vector $\mathbf{h}^{(t)}$.

Parameter sharing. It is important to note the parameters of RNNs, θ, are the *same* at every time step t. This means that they can be reapplied as many times as we want. This allows RNNs to be able to process all time steps from any number of input sequences, instead of learning a separate model for all possible time steps.

Unfolding computational graphs. Equation 2.44 is recurrent because the definition of the state $\mathbf{h}$ at time t relies on the same definition at time $t-1$. For a finite number of time steps τ, the graph

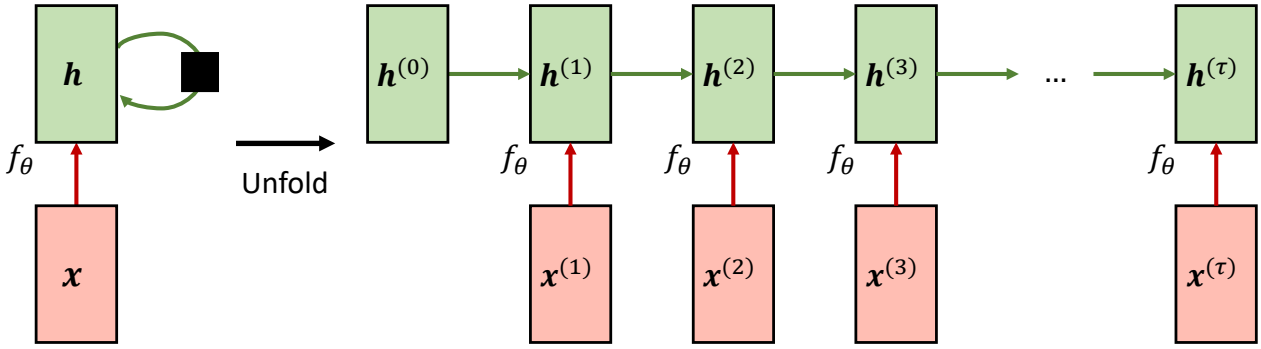


Figure 2.20: Unfolding a computation graph of a recurrent neural network with no outputs. This RNN incorporates the information from the input $\mathbf{x}^{(t)}$ into the previous state $\mathbf{h}^{(t-1)}$, producing the current state $\mathbf{h}^{(t)}$ that is pass forward through time. **Left:** a circuit diagram, where the red and the green boxes represent the input and the hidden vectors, respectively, and the black box indicates an interaction with a delay of a single time step. **Right:** an unfolded computation graph (representing the same network), where each column is associated with one particular time step. Typically, RNNs will have output layers that read information from the state $\mathbf{h}$ to make predictions, which will be discussed later in this section.

can be *unfolded* by repeatedly computing the state in the same way at each time step for $\tau - 1$ times, as illustrated in Figure 2.20. Such unfolding results in an expression that does not have any recurrent components, which can be then represented by a directed acyclic computational graph.

Design patterns of RNNs. With the graph unfolding and parameter sharing, we can design a variety of RNNs that process the sequences of input and output vectors in different patterns to satisfy the needs of different applications. Figure 2.21 illustrates four different patterns of RNNs and their applications.

Concrete example. To give a concrete example, we will explain one design pattern of an RNN, called many-to-many, which is the rightmost diagram in Figure 2.21. This many-to-many RNN is a recurrent network that produces an output at each time step and has recurrent connections between hidden units. Also, we will discuss one possible mathematical form of the recurrent formula (see Equation 2.44), and how we can add output layers to read information from the hidden state $\mathbf{h}$ to make a prediction at each time step. Suppose we have an RNN with one input layer, one hidden layer and one output layer, illustrated in Figure 2.22, that processes a sequence of input $\{\mathbf{x}^{(1)}, ..., \mathbf{x}^{(\tau)}\}$. Here we assume that the output of the network is discrete (e.g., predict the next words or characters), in which the output of the hidden layer $\mathbf{o}$ is treated as the unnormalized log probabilities of each possible value of the discrete variable. The softmax function is then applied as a post-processing step to obtain the normalized probabilities over the output $\hat{\mathbf{y}}$. Specifically, for each time step from $t = 1$

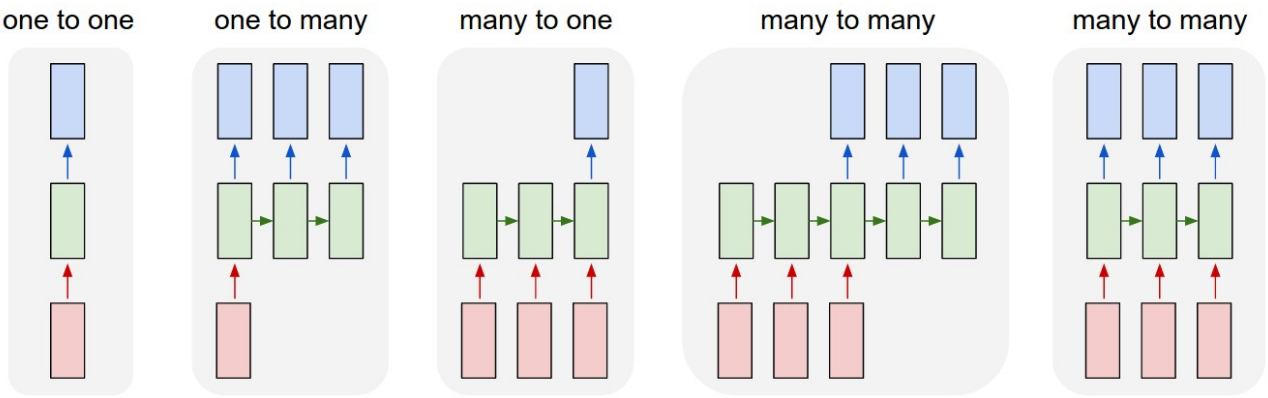


Figure 2.21: Comparison among a feedforward neural network and different RNN patterns [8]. Rectangles denote vectors and arrows denote function dependencies. **From left to right**: **(1)** a feedforward neural network that accepts an input vector (blue), transforms it to an output vector (blue) using one hidden layer (green) (e.g., image classification), **(2)** a RNN that accepts an input vector and generates a sequence of output vectors (e.g., image captioning), **(3)** a RNN that accepts a sequence of input vectors and generates an output vector (e.g., sentiment analysis), **(4)** a RNN that accepts a variable-length sequence of input vectors and generates a variable-length sequence of output vectors noting that the length of the input and output sequences can be different (e.g., machine translation), and **(5)** a RNN that accepts a sequence of input vectors and *simultaneously* generates an output vector for each input (e.g., sleep stage classification which annotates each signal epoch based on the previous sequences). Note that the RNNs **(2)**-**(5)** utilize a *fixed* recurrent hidden layer (green) to summarize the information from the previous input vectors, which enables us to process any length of the input sequences.

to $t = \tau$ with the initial state $\mathbf{h}^{(0)} = \vec{0}$, the RNN processes the input as follows:

$$\mathbf{z}^{(t)} = \mathbf{W}_{xh}\mathbf{x}^{(t)} + \mathbf{W}_{hh}\mathbf{h}^{(t-1)} + \mathbf{b}_h \ , \tag{2.45}$$

$$\mathbf{h}^{(t)} = \tanh(\mathbf{z}^{(t)}) \ , \tag{2.46}$$

$$\mathbf{o}^{(t)} = \mathbf{W}_{hy}\mathbf{h}^{(t)} + \mathbf{b}_y \ , \tag{2.47}$$

$$\hat{\mathbf{y}}^{(t)} = \text{softmax}(\mathbf{o}^{(t)}) \ , \tag{2.48}$$

where parameters are weight matrices $\mathbf{W}_{xh}$, $\mathbf{W}_{hh}$ and $\mathbf{W}_{hy}$, respectively, for input-to-hidden, hidden-to-hidden and hidden-to-output, and bias vectors $\mathbf{b}_h$ and $\mathbf{b}_y$. Here the recurrent formula is defined in Equation 2.45 and 2.46. The total loss for a sequence of input $\mathbf{x}$ paired with a sequence of target $\mathbf{y}$ is the sum of the losses over all time steps. In this case, we can use the cross-entropy loss (see Section 2.9.1), where the loss at each time step is the negative log-likelihood of $y^{(t)}$ given $\mathbf{x}^{(1)}, ..., \mathbf{x}^{(t)}$. Thus the total is then,

$$J(\theta) = L\left(\{\mathbf{x}^{(1)}, .., \mathbf{x}^{(\tau)}\}, \{\mathbf{y}^{(1)}, .., \mathbf{y}^{(\tau)}\}\right) \tag{2.49}$$

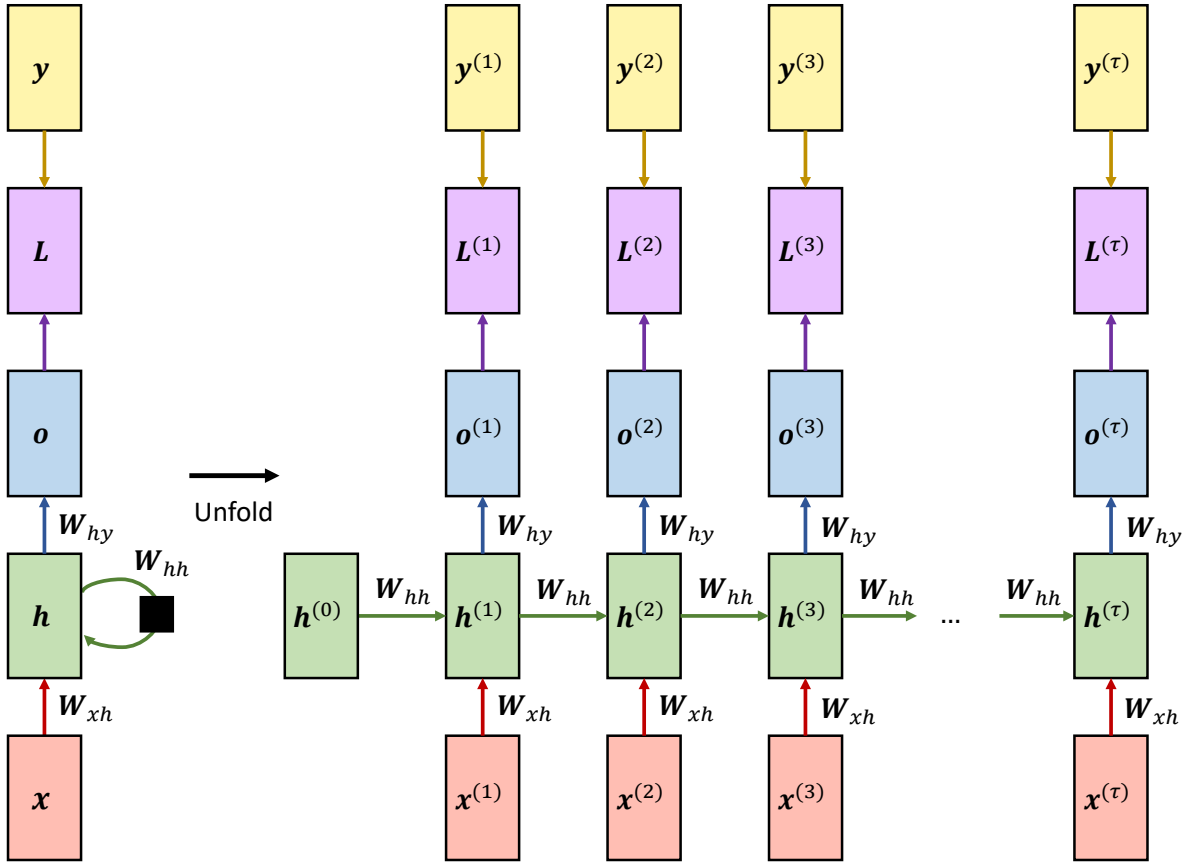


Figure 2.22: A computational graph to compute the training loss of an RNN that accepts a sequence of input vectors $\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(\tau)}\}$ and simultaneously generates an output vector for each input $\{\mathbf{o}^{(1)}, \ldots, \mathbf{o}^{(\tau)}\}$ (i.e., many-to-many design pattern). The input-to-hidden and hidden-to-output connections are parameterized by the weights, which are $\mathbf{W}_{xh}$ and $\mathbf{W}_{hy}$ respectively. The hidden-to-hidden recurrent connections are parameterized by the weights $\mathbf{W}_{hh}$. These weights are the same across time steps. A loss L quantifies the disagreement between the output $\mathbf{o}$ and the corresponding training target $\mathbf{y}$. For instance, the loss can be the cross-entropy loss that treats the output $\mathbf{o}$ as the unnormalized log probabilities. It internally computes $\hat{\mathbf{y}} = \text{softmax}(\mathbf{o})$ and compare it with the target $\mathbf{y}$. The forward pass of this RNN is defined in Equation 2.45. **Left:** The computational graph of RNN drawn with recurrent connections. **Right:** The time-unfolded computational graph of RNN, where each column represents the computation of one time step.

$$= \sum_{t=1}^{\tau} L^{(t)} \ , \tag{2.50}$$

$$L^{(t)} = -\log p\left(y^{(t)} \mid \{\mathbf{x}^{(1)}, .., \mathbf{x}^{(t)}\}; \theta\right) \tag{2.51}$$

$$= -\log(\hat{y}^{(t)}_{y^{(t)}}) \ , \tag{2.52}$$

where $\hat{y}^{(t)}_{y^{(t)}}$ is the estimated probability for $y^{(t)}$ from the network's output vector $\hat{\mathbf{y}}^{(t)}$.

Backpropagation through time. Computing gradients through a RNN is simply to apply the backpropagation (see Section 2.5) to the unrolled computation graph. In particular, the gradients are

computed by performing a forward pass moving from left to right through the unfolded computation graph in Figure 2.22, followed by a backward pass moving from right to left through the graph. This technique is commonly referred to as **backpropagation through time** (BPTT). To give some intuitions of how to compute gradients by BPTT, we will compute the gradients of the RNN (Figure 2.22) in the example above. During the forward pass, we apply Equation 2.45 and 2.49 sequentially from $t = 1$ to $t = \tau$. During the backward pass, we need to compute the gradient of the parameters $\mathbf{W}_{xh}$, $\mathbf{W}_{hh}$, $\mathbf{W}_{hy}$, $\mathbf{b}_h$ and $\mathbf{b}_y$, recursively, from $t = \tau$ to $t = 1$. We start the recursion with the node preceding the total loss:

$$\frac{\partial L}{\partial L^{(t)}} = \frac{\partial}{\partial L^{(t)}} \left(L^{(1)} + \cdots + L^{(t)} + \cdots + L^{(\tau)} \right) \tag{2.53}$$

$$= 1 \ . \tag{2.54}$$

At time step t, the gradient on the outputs is computed as follows:

$$\frac{\partial L}{\partial \mathbf{o}^{(t)}} = \left[\frac{\partial L}{\partial o_1^{(t)}}, \ldots, \frac{\partial L}{\partial o_{n_y}^{(t)}} \right] \ , \tag{2.55}$$

where n_y is the number of units of the output layer or the number of elements in the output vector $\mathbf{o}^{(t)}$ and,

$$\frac{\partial L}{\partial o_i^{(t)}} = \frac{\partial L}{\partial L^{(t)}} \frac{\partial L^{(t)}}{\partial o_i^{(t)}} = \hat{y}_i^{(t)} - \mathbf{1}_{i=y^{(t)}} \ , \tag{2.56}$$

where the subscribe i specifies the i-th element of $\mathbf{o}^{(t)}$. At the final time step $t = \tau$, there is only the gradient from $\mathbf{o}^{(\tau)}$ backpropagated to $\mathbf{h}^{(\tau)}$, so its gradient is just:

$$\frac{\partial L}{\partial \mathbf{h}^{(\tau)}} = \frac{\partial L}{\partial \mathbf{o}^{(\tau)}} \frac{\partial \mathbf{o}^{(\tau)}}{\partial \mathbf{h}^{(\tau)}} \tag{2.57}$$

$$= \mathbf{W}_{hy}^{\top} \left(\frac{\partial L}{\partial \mathbf{o}^{(\tau)}} \right) \ . \tag{2.58}$$

Once we get the gradients at the final time step, we can then backpropagate the gradients backward in time from $t = \tau - 1$ to $t = 1$. For the time step $t < \tau$, there is another gradient from $\mathbf{h}^{(t+1)}$ in addition to $\mathbf{o}^{(t)}$ that are backpropagated to $\mathbf{h}^{(t)}$. So the gradient of $\mathbf{h}^{(t)}$ (for $t < \tau$) is computed as follows:

$$\frac{\partial L}{\partial \mathbf{h}^{(t)}} = \frac{\partial L}{\partial \mathbf{o}^{(t)}} \frac{\partial \mathbf{o}^{(t)}}{\partial \mathbf{h}^{(t} + \frac{\partial L}{\partial \mathbf{h}^{(t+1)}} \frac{\partial \mathbf{h}^{(t+1)}}{\partial \mathbf{h}^{(t)}} \tag{2.59}$$

$$= \mathbf{W}_{hy}^{\top} \left(\frac{\partial L}{\partial \mathbf{o}^{(\tau)}} \right) + \frac{\partial L}{\partial \mathbf{h}^{(t+1)}} \frac{\partial \mathbf{h}^{(t+1)}}{\partial \mathbf{z}^{(t+1)}} \frac{\partial \mathbf{z}^{(t+1)}}{\partial \mathbf{h}^{(t)}} \tag{2.60}$$

$$= \mathbf{W}_{hy}^{\top}\left(\frac{\partial L}{\partial \mathbf{o}^{(\tau)}}\right) + \mathbf{W}_{hh}^{\top}\left(\frac{\partial L}{\partial \mathbf{h}^{(t+1)}}\right)\text{diag}\left(1-(\mathbf{h}^{(t+1)})^2\right) \quad , \tag{2.61}$$

where $\text{diag}\left(1-(\mathbf{h}^{(t+1)})^2\right)$ is the diagonal matrix where the (i,i)-th element contains $1-(h_i^{(t+1)})^2$. This is the Jacobian matrix of hyperbolic tangent associated with the hidden unit i at time $t+1$. With the gradients obtained from above ($\frac{\partial L}{\partial \mathbf{o}^{(t)}}$ and $\frac{\partial L}{\partial \mathbf{h}^{(t)}}$), we can now compute the gradients of the RNN's parameters. As these parameters are *shared* for many time steps, the gradients of these parameters are, therefore, the sum of the contribution of the parameters from these time steps as follows [38]:

$$\frac{\partial L}{\partial \mathbf{W}_{xh}} = \sum_{t=1}^{\tau}\sum_{i=1}^{n_h}\frac{\partial L}{\partial h_i^{(t)}}\frac{\partial h_i^{(t)}}{\partial \mathbf{W}_{xh}} \tag{2.62}$$

$$= \sum_{t=1}^{\tau}\text{diag}\left(1-(\mathbf{h}^{(t)})^2\right)\left(\frac{\partial L}{\partial \mathbf{h}^{(t)}}\right)\mathbf{x}^{(t)^{\top}} \quad , \tag{2.63}$$

$$\frac{\partial L}{\partial \mathbf{W}_{hh}} = \sum_{t=1}^{\tau}\sum_{i=1}^{n_h}\frac{\partial L}{\partial h_i^{(t)}}\frac{\partial h_i^{(t)}}{\partial \mathbf{W}_{hh}} \tag{2.64}$$

$$= \sum_{t=1}^{\tau}\text{diag}\left(1-(\mathbf{h}^{(t)})^2\right)\left(\frac{\partial L}{\partial \mathbf{h}^{(t)}}\right)\mathbf{h}^{(t-1)^{\top}} \quad , \tag{2.65}$$

$$\frac{\partial L}{\partial \mathbf{W}_{hy}} = \sum_{t=1}^{\tau}\sum_{i=1}^{n_y}\frac{\partial L}{\partial o_i^{(t)}}\frac{\partial o_i^{(t)}}{\partial \mathbf{W}_{hy}} \tag{2.66}$$

$$= \sum_{t=1}^{\tau}\frac{\partial L}{\partial \mathbf{o}^{(t)}}\mathbf{h}^{(t)^{\top}} \quad , \tag{2.67}$$

$$\frac{\partial L}{\partial \mathbf{b}_h} = \sum_{t=1}^{\tau}\frac{\partial L}{\partial \mathbf{h}^{(t)}}\frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{b}_h^{(t)}} \tag{2.68}$$

$$= \sum_{t=1}^{\tau}\text{diag}\left(1-(\mathbf{h}^{(t)})^2\right)\frac{\partial L}{\partial \mathbf{h}^{(t)}} \quad , \tag{2.69}$$

$$\frac{\partial L}{\partial \mathbf{b}_y} = \sum_{t=1}^{\tau}\frac{\partial L}{\partial \mathbf{o}^{(t)}}\frac{\partial \mathbf{o}^{(t)}}{\partial \mathbf{b}_y^{(t)}} \tag{2.70}$$

$$= \sum_{t=1}^{\tau}\frac{\partial L}{\partial \mathbf{o}^{(t)}} \quad , \tag{2.71}$$

where n_h is the number of hidden units.

Challenge of long-term dependencies. The RNN architecture that we discuss so far is subject to several problems when it is trained to learn long-term dependencies via BPTT [64, 65]. It has been shown that the gradients tend to vanish (most frequent) or explode (less frequent) exponentially when they are backpropagated through time. This makes the training process very unstable [66]. In particular, from Equation 2.61, it can be seen that the second term of the summation involve the multiplication between the weight $\mathbf{W}_{hh}$ and the gradients backpropagated from the next time

step. When this RNN is trained with a very long sequence, we may end up with multiplying the same weight $\mathbf{W}_{hh}$ many times, yielding either the exponentially smaller or larger gradients. This will result in either **vanishing gradients** or **exploding gradients**. The exploding gradient problem is rare, and can be mitigated by applying the heuristic gradient clipping [67] that forces (or clip) the gradients to when their norm is greater than some maximum value. On the other hand, the vanishing gradient problem is more common and can be mitigated by using an alternative RNN architecture that introduces paths through time that do not diminish the gradient flow, named *Long Short-Term Memory (LSTM)*, which will be discussed later in this section.

Truncated backpropagation through time. BPTT can be computationally expensive when we have a very long input sequence (i.e., τ is very large). The gradient computation includes performing a forward pass and a backward pass, whose runtime is $O(\tau)$ and cannot be reduced by parallelization. This is because the computation graph is sequential. For instance, during a forward pass, each time step can be computed only after the previous time step. Also, all hidden states obtained in the forward pass must be stored until they are used during a backward pass. This incur the memory cost of also $O(\tau)$. Truncated backpropagation through time [68] has been proposed to mitigate this problem, which generally works well in practice. It splits the very long input sequence into small sub-sequences, and performs a forward and a backward pass over each sub-sequence sequentially instead of the whole sequence. For instance, suppose we have an input sequence of length 20,000. We may split the input sequence into 200 sub-sequences each of length 100. Then we compute the gradients and update the RNN's parameters over each sub-sequence sequentially. The final hidden state obtained from the forward pass, which contains the summary of the previous input sequence, is also carried forward in time by using it as the initial hidden state $\mathbf{h}^{(0)}$ in the following sub-sequence. With this approach, we still perform the exactly same forward pass as we were to do with the whole sequence, except that we perform the backward pass over small chunks of the sequence instead of the whole. Truncated BPTT can, therefore, help us reduce both the computation and the memory cost for training RNNs over a very long sequence.

Long Short-Term Memory

Long-Short Term Memory (LSTM) [69] is a variant of RNNs that introduces *a path through time* where the gradients can flow for a very long time. This allows the gap between the relevant information from the previous data and the time point when it is needed to be very large. In addition to learning

which information at the current time point would be useful for the later time point, LSTM also allows the model to decide which information from the past to be forgotten (i.e., learn to forget irrelevant information).

In particular, LSTM introduces additional internal memory, called **cell state**, in addition to the hidden state and utilizes gate mechanisms or **gated recurrent units** to define a more sophisticated recurrent formula, facilitating the ability to read, forget and add information to/from the cell state.

Forget gate. LSTM uses forget gate units $\mathbf{f}^{(t)}$ to decide which information of the previous cell state to be forgotten, computed as follows:

$$\mathbf{f}^{(t)} = \text{sigm}(\mathbf{W}_{xf}\mathbf{x}^{(t)} + \mathbf{W}_{hf}\mathbf{h}^{(t-1)} + \mathbf{b}_f) \ , \tag{2.72}$$

where $\mathbf{x}^{(t)}$ is the current input vector and $\mathbf{h}^{(t-1)}$ is the previous hidden state. The $\mathbf{W}_{xf}$, $\mathbf{W}_{xf}$ and $\mathbf{b}_f$ are input weights, recurrent weights and biases for the forget gates, respectively. The forget gate uses the sigmoid function (sigm) to squash the output values to be between 0 and 1, where 0 and 1 means completely forget and keep the information respectively.

Input gate. LSTM uses input gate units $\mathbf{i}^{(t)}$ to decide which new information to be stored in the cell state, computed as follows:

$$\mathbf{i}^{(t)} = \text{sigm}(\mathbf{W}_{xi}\mathbf{x}^{(t)} + \mathbf{W}_{hi}\mathbf{h}^{(t-1)} + b_i) \ , \tag{2.73}$$

where $\mathbf{W}_{xi}$, $\mathbf{W}_{hi}$ and $\mathbf{b}_i$ are the parameters of the input gate. The sigm function is to squash the output value to be between 0 and 1, such that it can be used to control the amount of information to be stored in the cell state. LSTM also computes a vector $\mathbf{g}^{(t)}$ which contains candidate values that may be used to update the current cell state,

$$\mathbf{g}^{(t)} = \tanh(\mathbf{W}_{xg}\mathbf{x}^{(t)} + \mathbf{W}_{hg}\mathbf{h}^{(t-1)} + b_g) \ , \tag{2.74}$$

where $\mathbf{W}_{xg}$, $\mathbf{W}_{hg}$ and $\mathbf{b}_g$ are the parameters to decide the new candidate values from the $\mathbf{x}^{(t)}$ and $\mathbf{h}^{(t-1)}$. Here the tanh function is used instead of the sigm to squash the output value to be between -1 and 1.

Update cell state. LSTM uses the forget gate units $\mathbf{f}^{(t)}$ to select some information from the previous cell state by multiplying the previous state by $\mathbf{f}^{(t)}$. Then it adds the new information obtained by

multiplying the new candidate values $\mathbf{g}^{(t)}$ by the input gates $\mathbf{i}^{(t)}$, which specify the amount of new information to be added to the cell state. The cell state is updated with the following equation:

$$\mathbf{c}^{(t)} = \mathbf{f}^{(t)} \odot \mathbf{c}^{(t-1)} + \mathbf{i}^{(t)} \odot \mathbf{g}^{(t)} \ , \tag{2.75}$$

where $\mathbf{c}^{(t)}$ and $\mathbf{c}^{(t-1)}$ is the current and the previous cell state respectively, and $\odot$ is an element-wise multiplication.

Output gate. Once LSTM has updated the current cell state, it then uses the output gate units $\mathbf{o}^{(t)}$ to decides which information from the current cell state to be revealed as the current hidden state, computed as follows:

$$\mathbf{o}^{(t)} = \text{sigm}(\mathbf{W}_{xo}\mathbf{x}^{(t)} + \mathbf{W}_{ho}\mathbf{h}^{(t-1)} + b_o) \ , \tag{2.76}$$

where $\mathbf{W}_{xo}$, $\mathbf{W}_{ho}$ and $\mathbf{b}_o$ are the parameters of the output gate. Again, the sigm function squashes the output values to be between 0 and 1 as a binary gate to output parts of the current cell state as the current hidden state. The current hidden state is computed as follows:

$$\mathbf{h}^{(t)} = \mathbf{o}^{(t)} \odot \tanh(\mathbf{c}^{(t)}) \ , \tag{2.77}$$

where $\mathbf{h}^{(t)}$ is the current hidden state. Here, the cell state is squashed by tanh into a range of -1 and 1 before multiplied by $\mathbf{o}^{(t)}$.

In summary, LSTM maintains two states: cell and hidden states at every time step (see Figure 2.23). The cell state is an internal memory and is not exposed to the other layers in the network. The hidden state, on the other hand, can be revealed to the other layers. For each time step t, LSTM computes gate values: $\mathbf{f}^{(t)}$, $\mathbf{i}^{(t)}$, $\mathbf{g}^{(t)}$ and $\mathbf{o}^{(t)}$ using the current input $\mathbf{x}^{(t)}$ and the previous hidden state $\mathbf{h}^{(t-1)}$. The $\mathbf{f}^{(t)}$, $\mathbf{i}^{(t)}$ and $\mathbf{g}^{(t)}$ are used to update the current cell state $\mathbf{c}^{(t)}$. The $\mathbf{o}^{(t)}$ is used to control which parts of the cell state $\mathbf{c}^{(t)}$, squashed by tanh, to be exposed as the current hidden state $\mathbf{h}^{(t)}$. Similar to the simple RNN architecture mentioned earlier, any output layers can be stacked on top of LSTM to read information from the state $\mathbf{h}$ to make predictions (e.g., see Equation 2.47 and 2.48).

Improve gradient flow. It is important to note that, unlike the simple RNN architecture, the LSTM modifies the current state $\mathbf{c}^{(t)}$ using the additive and the element-wise multiplication interactions (see Equation 2.75). During backpropagation through time, the additive interaction allows the gradient to be *equally distributed*, which is the critical element of the LSTM representing the path through

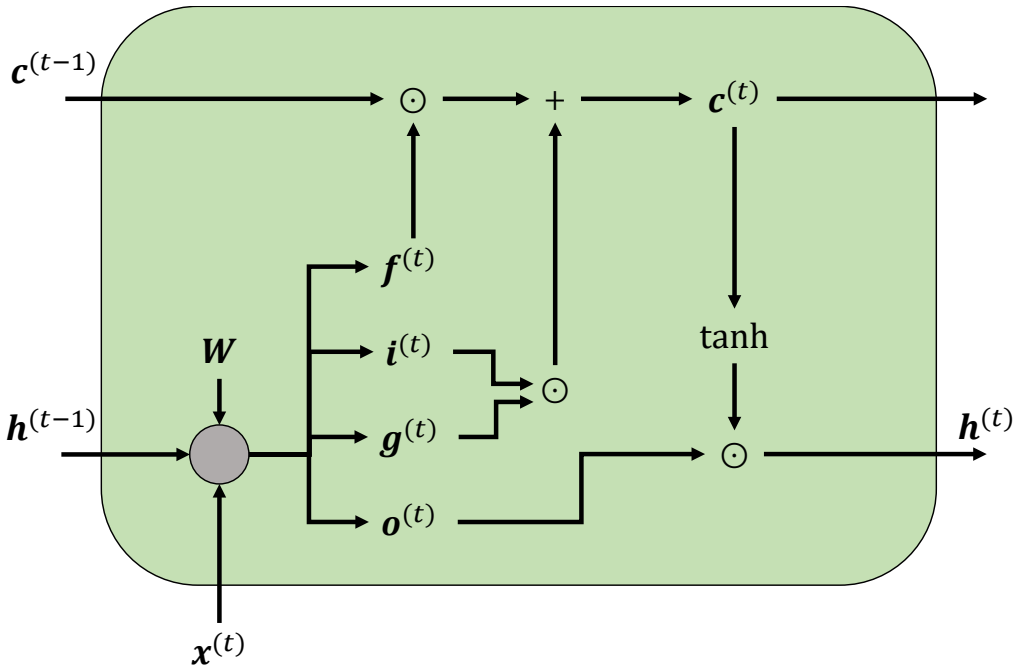


Figure 2.23: A LSTM model. LSTM maintains two states: cell and hidden states at every time step. The cell state is an internal memory and is not exposed to the other layers in the network. The hidden state, on the other hand, can be revealed to the other layers. For each time step t, LSTM computes gate values: $\mathbf{f}^{(t)}$, $\mathbf{i}^{(t)}$, $\mathbf{g}^{(t)}$ and $\mathbf{o}^{(t)}$ using the current input $\mathbf{x}^{(t)}$ and the previous hidden state $\mathbf{h}^{(t-1)}$. The $\mathbf{f}^{(t)}$, $\mathbf{i}^{(t)}$ and $\mathbf{g}^{(t)}$ are used to update the current cell state $\mathbf{c}^{(t)}$ as following. The forget gate $\mathbf{f}^{(t)}$ decides which information of the previous cell state to be forgotten. The input gate $\mathbf{i}^{(t)}$ decides which new information from the candidate $\mathbf{g}^{(t)}$ to be stored in the cell state. The output gate $\mathbf{o}^{(t)}$ controls which parts of the cell state $\mathbf{c}^{(t)}$, squashed by tanh, to be exposed as the current hidden state $\mathbf{h}^{(t)}$. Any output layers can be stacked on top of the LSTM model to read information from the hidden state $\mathbf{h}$ to make predictions.

time that we mentioned earlier. Also, the element-wise multiplication interaction with the forget gate helps mitigate the vanishing and exploding gradient problems. The gradients that are element-wise-multiplied with the forget gate that *varies for each time step* are less likely to vanish or explode compared to the simple RNN where the gradients are matrix-multiplied by the *fixed* weight matrix repeatedly. These interactions allow the gradients of the cell state that contribute to the gradients of the RNN's parameters to be propagated backward through time uninterrupted for long period of times. This allows the LSTM to learn long-term dependencies in a much more easier way compared to the simple RNN.

Bidirectional RNNs

So far we have seen RNNs that utilizes information from the past $\{\mathbf{x}^{(1)}, ..., \mathbf{x}^{(t-1)}\}$ and at the current time $\mathbf{x}^{(t)}$ to produce an output vector. However, in many applications, we may also want to utilize

the information from the *whole* input sequence $\{\mathbf{x}^{(1)}, ..., \mathbf{x}^{(T)}\}$ (i.e., past, current and future).

A bidirectional recurrent neural network (bidirectional RNN) [38, 70] is another variant of RNNs consisting of **two** RNNs that summarizes the input sequence in **both forward and backward directions**. One RNN processes the input sequence forward through time starting from the start of the sequence, and another RNN processes the input sequence backward through time starting from the end of the sequence. It has been shown that the future information can help improve the performance of many applications such as speed recognition [71] and bioinformatics [72]. For instance, in the speech recognition, when there are two possible interpretations of the current word that are both acoustically plausible, a summary of the future phonemes (in addition to the past phonemes) can help differentiate between them because of the co-articulation [71].

Figure 2.24 illustrates how a bidirectional RNN produces an output vector based on based on the information from both the past and future at each time step. It consists of two RNNs: one processes the input sequences forward in time from left to right with the hidden state $\mathbf{h}^{(t)}$, and another one processes the input sequences backward in time from right to left with the hidden state $\mathbf{g}^{(t)}$. So, at each time step t, the bidirectional RNN can utilize the summaries of both the past and the future that are most sensitive to the input values around t to determine the output $\mathbf{o}^{(t)}$.

As the main difference between (one-directional) RNNs and bidirectional RNNs is that the bidirectional RNNs utilizes an additional RNN to process the input sequence backward, any variant of the (one-directional) RNNs such as LSTM can be used in the bidirectional RNNs.

2.9.6 Autoencoders

An autoencoder is a neural network that is trained to attempt to reconstruct its inputs (i.e., unsupervised learning) [38]. By training it to reconstruct the inputs, it can learn representations that are useful for the reconstruction of the data, and this can help us to learn a better representation of the data as mentioned in Section 2.3. This network consists of two components: an encoder function $f_\theta(\mathbf{x})$ that transforms an input into a **code h** and a decoder function $g_\theta(\mathbf{h})$ that generates a reconstructed input $\hat{\mathbf{x}}$. The function f and g are defined as follows:

$$f_\theta(\mathbf{x}) = \sigma(\mathbf{W}_f \mathbf{x} + b_f) \ , \tag{2.78}$$

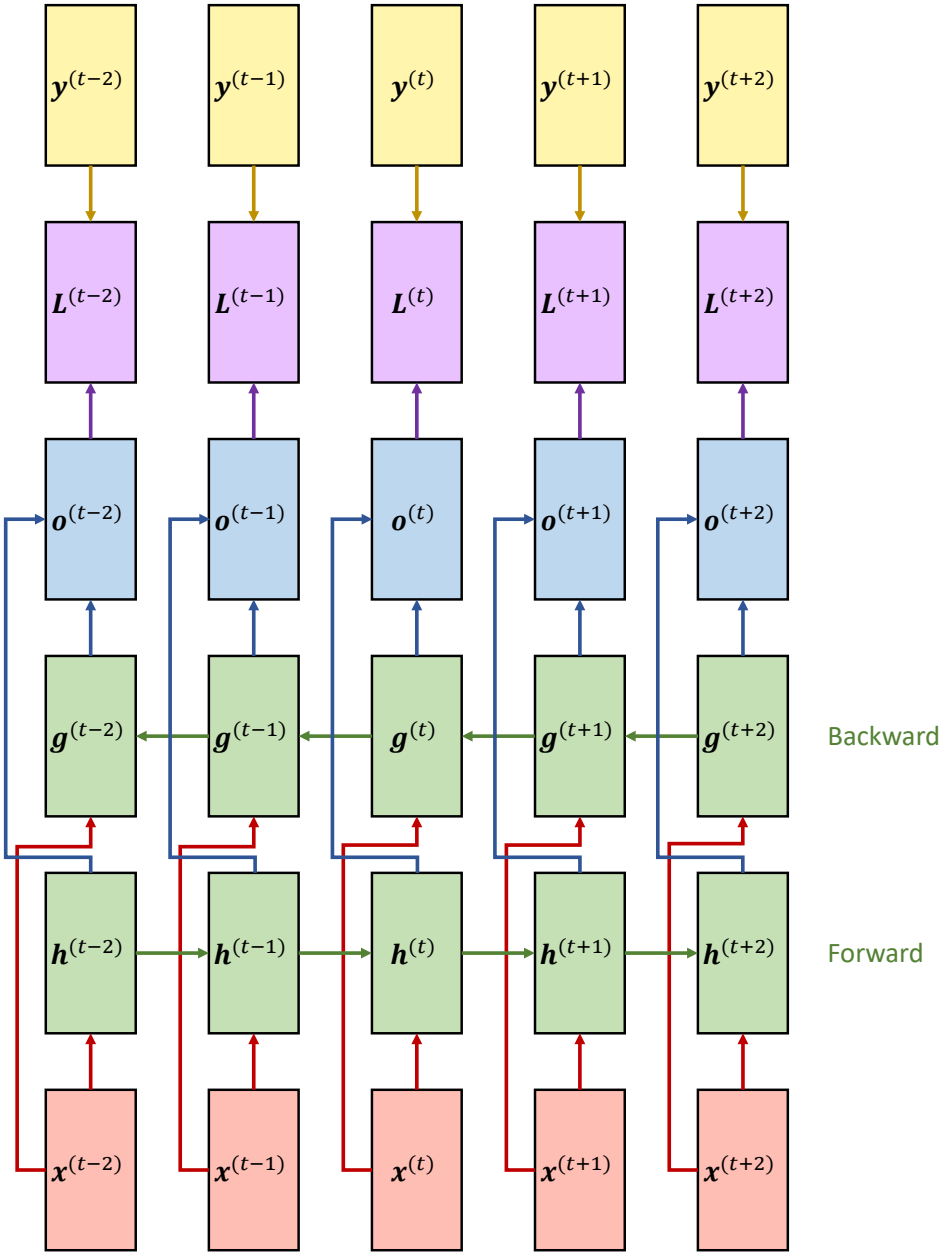


Figure 2.24: A computational graph to compute the training loss of a bidirectional RNN that accepts a sequence of inputs $\mathbf{x}$ and produce a sequence of outputs $\mathbf{o}$. A loss L measures the disagreement between the output $\mathbf{o}$ and the corresponding training target $\mathbf{y}$. The same sequences of inputs are fed to both RNNs, each of which summarizes the information in different directions. The forward RNN $\mathbf{h}$ processes the input sequences forward in time from left to right, while the backward RNN $\mathbf{g}$ process the input sequences backward in time from right to left. So, at each time step t, the bidirectional can utilize the summaries of both the past $\mathbf{h}^{(t)}$ and the future $\mathbf{g}^{(t)}$ that is most sensitive to the input values around t to determine the output $\mathbf{o}^{(t)}$.

$$g_\theta(\mathbf{x}) = \sigma(\mathbf{W}_g\mathbf{x} + b_g) \ , \tag{2.79}$$

We usually enforce some constraints on the autoencoder to make it unable to learn to reconstruct the input perfectly. This is because we want the network to learn only to keep information that is useful

for input reconstruction. In this way, the network can learn useful properties of the data.

One way is to reduce the dimension of $\mathbf{h}$ to be smaller than $\mathbf{x}$. In this way, the autoencoder will learn to capture only the most important features of the training data. As the autoencoder is trained to reconstruct its input, it can be trained in the *unsupervised* setting. In particular, a **reconstruction loss** $L(\hat{\mathbf{x}}, \mathbf{x})$ that quantifies the dissimilarity between an input and its reconstruction is used to optimize parameters θ of function f and g. An example of the loss function that we can use is the squared difference between the input and its reconstruction, $L(\hat{\mathbf{x}}, \mathbf{x}) = (\hat{\mathbf{x}} - \mathbf{x})^2$.

Another way is to allow the dimension of h to be equal or larger than the input, and introduce a regularization term, named a *sparsity penalty* $R(\mathbf{h})$, in addition to the reconstruction loss. Such autoencoder with the sparse penalty is called **sparse autoencoder**. Suppose there are m training examples $\left\{\mathbf{x}^{(i)}\right\}_{i=1}^{m}$, the unsupervised learning objective of the sparse autoencoder is to:

$$\theta^* = \arg\min_{\theta} \frac{1}{m} \sum_{i=1}^{m} L(g_\theta(f_\theta(\mathbf{x}^{(i)})), \mathbf{x}^{(i)}) + \beta R(f_\theta(\mathbf{x}^{(i)})) \ , \tag{2.80}$$

where β is the weight of the sparsity penalty. One common sparsity penalty is the Kullback-Leibler (KL) divergence:

$$R(h) = \sum_{j=1}^{n_h} \mathrm{KL}\left(\rho || \hat{\rho}_j\right) = \sum_{j=1}^{n_h} \rho \log(\frac{\rho}{\hat{\rho}_j}) + (1-\rho)\log\frac{(1-\rho)}{(1-\hat{\rho}_j)} \ , \tag{2.81}$$

where ρ is a hyperparameter used to specify the degree of sparsity, $\hat{\rho}_j = \frac{1}{m}\sum_{i=1}^{m} h_j^{(i)}$ is a mean activation of j-th element of $\mathbf{h}$, and n_h is the number of dimensions (or the number of hidden units) of $\mathbf{h}$. The ρ is typically set to a small value (e.g., 0.05). This is to force most of the values in $\mathbf{h}$ to be zero (i.e., sparse).

So far, we have shown the autoencoder that consists of a single-layer encoder (Equation 2.78) and a single-layer decoder (Equation 2.79). However, it is possible to use a deeper encoder and a deeper decoder. In particular, it is possible to stack multiple autoencoders on top of each other to form a *deep autoencoder*. Deeper autoencoders can be trained to represent more complex functions with a more compact network size (i.e., less number of parameters θ) as they can reuse features from the previous layer. It has also been shown that deeper autoencoders have a better compression capacity compare to the shallow autoencoders [38]. Deep autoencoders with and without the sparsity penalty can also be called as **stacked autoencoders (SAE)** and **stacked sparse autoencoders (SSAE)**

respectively. A common algorithm to effectively train deep autoencoders is the *greedy layer-wise unsupervised pre-training* [73].

Apart from learning useful features from the training data, the parameters θ of the trained encoder f_θ can also be used to initialize a deep learning model. As the encoder function of a deep autoencoder can also be viewed as a deep fully-connected neural network, using the parameters from the trained encoder as a starting point of the training for the subsequent task (e.g., classification) may be a better than randomly initializing θ. This can lead to an improvement in the performance on many tasks.

2.9.7 Residual Learning Framework

As previously mentioned that a deep neural network can reuses features from the top layers (closer to the input layer), and can lead to learning more abstract features at the bottom layers (closer to the output layer). Such deep network, however, is very difficult to train due to the degradation problem (see Figure 2.25) [9, 74].

A residual learning framework [9] has been proposed to alleviate such problems. In particular, instead of using multiple layers of non-linear transformations to directly approximate some functions, we use these layers to approximate a residual function. Suppose that the function that the network is trying to approximate is $H_\theta(\mathbf{x})$. We can explicitly let the network approximate to learn a **residual function** $f_\theta(\mathbf{x})$, defined as follows:

$$H_\theta(\mathbf{x}) = f_\theta(\mathbf{x}) + \mathbf{x} \ , \tag{2.82}$$

where $\mathbf{x}$ is an input vector. This reformulation is achieved by a shortcut connection and an element-wise addition (see Figure 2.26). Note that such shortcut connection requires the output vector to have the same dimension as the input vector. If we want the dimension of the output vector to be different from the input, we can perform a linear projection via the shortcut connection as follows:

$$H_\theta(\mathbf{x}) = f_\theta(\mathbf{x}) + \mathbf{W}_s x + b_s \ , \tag{2.83}$$

It is recommended that the linear projection should be applied only when the dimension of the input and output vectors are different, as a linear projection introduces additional parameters ($\mathbf{W}_s$ and b_s) to be optimized and computation complexity.

Notice that this residual building block introduces the additive interaction to the input vector. This

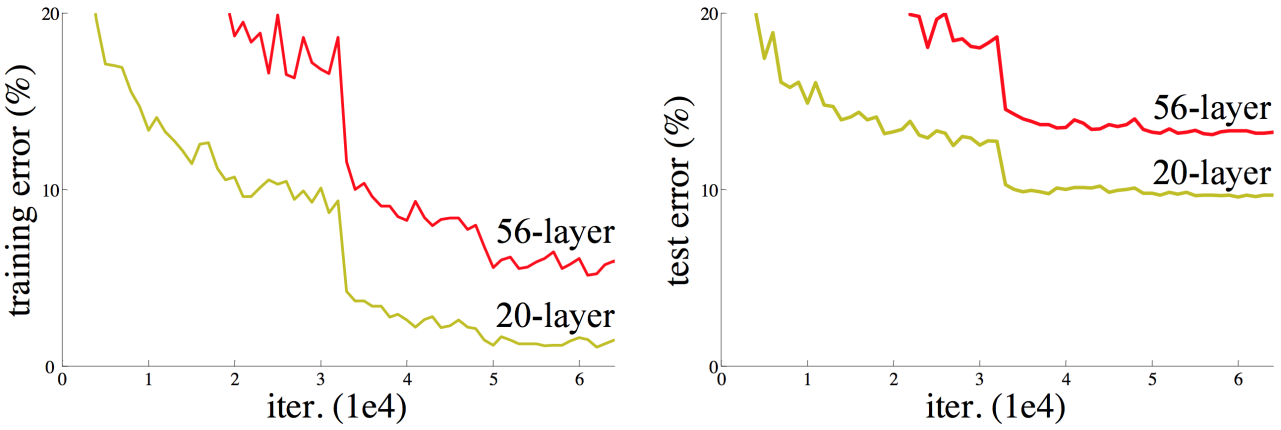


Figure 2.25: Degradation problem [9]. The training (left) and test (right) error on the CIFAR-10 dataset from a 20-layer *plain* network are lower than a 56-layer network. Intuitively, by making the network deeper, the performance should not decrease as the added layers can learn an identity function, and the others are copied from the shallower network (i.e., a 20-layer network).

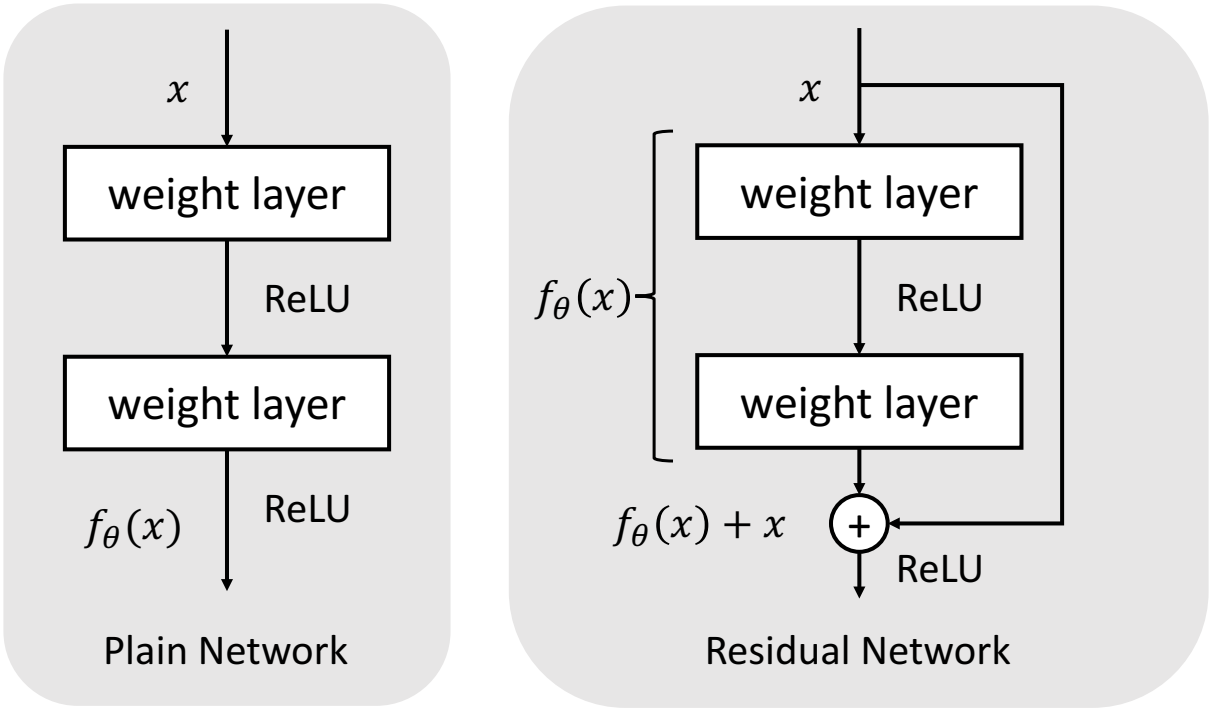


Figure 2.26: Comparison between the plain and the residual networks. The plain network directly transforms an input vector x with two non-linear layers (using ReLU). The residual network introduces a shortcut connection to explicitly let the two layers learn a residual function (i.e., learn a function to modify the input vector). The output of the residual function is added (element-wise) to the input vector, followed by a ReLU function.

additive interaction allows the gradients to easily flow from the layer below back to the layer above. Thus, by reformulating our model to learn a residual mapping, we can train a very deep network much more effectively, which alleviate the degradation problem.

2.10 Generative Models

In this section, we will cover the background knowledge for the techniques we develop in Chapter 6 for cross-domain signal transcription. In particular, we first briefly mention about what is generative models. Then we explain a Generative Adversarial Nets (GANs) framework for estimating generative models and its applications in domain translation, which is similar to the technique we develop in Chapter 6 to transcribe biosignals from one domain into another.

As the term "generative model" can be used in many different ways, in this thesis, we refer to any model that learn to *generate samples*. In particular, we will focus on the model that learns to estimate the distribution of the training set, consisting of examples drawn from a distribution p_{data}. The probability distribution that the model learns is p_{model}. The trained model can be used to generate samples from p_{model}.

Recently, there have been an increasing amount of interests to employ generative models to help improve the performance in many machine learning applications including image super-resolution [75], image-to-image translation [10], text-to-image synthesis [76]. This is due to the development of a novel training framework, called *Generative Adversarial Nets (GAN)* [77]. It has been shown that this framework can help generate the gradients for *multimodal distribution* (i.e., there are many possible correct ouputs corresponding to a single input) that cannot not be realized by traditional approaches such as mean square error (MSE) [10, 78].

2.10.1 Generative Adversarial Nets

The Generative Adversarial Nets (GAN) [77] framework has been proposed to improve the approximation of generative models. It consists of two models: a generator G and a discriminator D. The discriminator tries to distinguish real training data from the generated (or synthesized) data, while the generator tried to generate data that can fool the discriminator. In particular, this framework

Algorithm 2.1 Minibatch stochastic gradient descent for adversarial training [77]

1: **for** number of training iterations **do**
2: **for** k steps **do**
3: Sample minibatch of m noise samples $\{\mathbf{z}^{(1)}, \ldots, \mathbf{z}^{(m)}\}$ from noise prior $p_z(\mathbf{z})$.
4: Sample minibatch of m samples $\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}\}$ from data generating distribution $p_{data}(\mathbf{x})$.
5: Update the discriminator (fix θ_g) by minimizing the following loss function $J_d(\theta_d, \theta_g)$:

$$J_d(\theta_d, \theta_g) = -\frac{1}{m}\sum_{i=1}^{m} \log D(\mathbf{x}^{(i)}) - \frac{1}{m}\sum_{i=1}^{m} \log(1 - D(G(\mathbf{z}^{(i)}))) \tag{2.85}$$

6: **end for**
7: Sample minibatch of m noise samples $\{\mathbf{z}^{(1)}, \ldots, \mathbf{z}^{(m)}\}$ from noise prior $p_z(\mathbf{z})$.
8: Update the generator (fix θ_d) by minimizing the following loss function $J_g(\theta_d, \theta_g)$:

$$J_g(\theta_d, \theta_g) = -\frac{1}{m}\sum_{i=1}^{m} \log(D(G(\mathbf{z}^{(i)}))) \tag{2.86}$$

9: **end for**

setup a two-player minimax game, in which D and G play the following game on $V(D, G)$:

$$\min_G \max_D V(D, G) = \mathbb{E}_{\mathbf{x}\sim p_{data}(\mathbf{x})}[\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}}(\mathbf{z})}[\log(1 - D(G(\mathbf{z})))] \tag{2.84}$$

where $p_{\mathbf{z}}$ is a prior distribution for input noise vectors $\mathbf{z}$; $G(\mathbf{z})$ generates a sample of $\mathbf{x}$ drawn from the generator's distribution p_{model}; $D(\mathbf{x})$ is a scalar-valued function that returns the probability that $\mathbf{x}$ came from p_{data} rather than p_{model}; and p_{data} is an unknown data-generating distribution, from which the training data are sampled. By letting D and G to compete with each other, it can drive both of them to optimize their parameters until D cannot distinguish the samples generated by G from the real ones.

This game is suggested to be implemented by alternating between k steps of optimizing D and one step of optimizing G [77]. This is to keep D near its optimal solution so that it can provide useful feedback (i.e., gradients) for G. Also, it is pointed out that the loss for G in Equation 2.84 may not provide enough gradients for G to learn well in practice. If G generates samples poorly at the early stage of the training, D can reject samples from G with high confidence. In this case, the gradients from $\log(1 - D(G(\mathbf{z})))$ saturates meaning that G does not get any feedback about how to generate samples that can fool D. The author suggests that, instead of training G to minimize $\log(1 - D(G(\mathbf{z})))$, we can train G to maximize $\log D(G(\mathbf{z}))$ (i.e., flipping the target labels). In the case that D and G are neural networks, we can train them via backpropagation. This has led us to an algorithm to perform the adversarial training shown in Algorithm 2.1.

Even though it has been proved theoretically in [77] that this minimax game has a global optimum when $p_{model} = p_{data}$, such global optimum is difficult to be realized in practice. There have been several studies trying to find a way to achieve the optimum point [79–81], but it is still an ongoing research.

This framework can generalize to other applications in addition to generating images that look like to real ones. This generalization is typically achieved by modifying the generator G and discriminator D to conditions on extra information. Such modification allows the GAN to learn a loss function that would traditionally require very different kind of loss functions [10]. In the next section, we will discuss several studies that have extended the GAN framework to improve the performance of some tasks.

2.10.2 Conditional Generative Adversarial Nets

In previous section, we have discussed the generative adversarial nets (GANs) framework that can be used to train a generator G to synthesize images from random noise vectors $\mathbf{z}$ sampled from a prior distribution, such as the Gaussian distribution. In this section, we will discuss its extended version to condition on some additional information, named **conditional generative adversarial nets** or **conditional GANs**. In particular, we want to extend from training the generator $G : \mathbf{x} \rightarrow \mathbf{y}$ to the generator $G : \{\mathbf{x}, \mathbf{z}\} \rightarrow \mathbf{y}$, where $\mathbf{x}$ can be, for instance, as labels [82], texts [76], and images [10].

Recently, there have been a number of research employing the conditional GANs to train generators that are useful for many applications. As, in this thesis, we are only interested in the domain translation applications, we will discuss about the state-of-the-art techniques in this area.

Domain Translation. An example application of the conditional GANs is domain translation. It has been shown in the field of computer vision that we can use the conditional GANs to translate examples between different domains. For instance, [10] has proposed U-net that can translate images between different domains. In particular, instead of training the generator G to generate output images $\mathbf{y}$ from random noises $\mathbf{z}$ (i.e., $G : \mathbf{x} \rightarrow \mathbf{y}$), they train the generator G to generate $\mathbf{y}$ from input images $\mathbf{x}$ and random noises $\mathbf{z}$ (i.e., $G : \{\mathbf{x}, \mathbf{z}\} \rightarrow \mathbf{y}$). They also modify the discriminator D to receive pairs of images, instead of single images, and classify between real and generated (or synthesized) pairs (i.e., $D : (\mathbf{x}, \mathbf{y}) \rightarrow \{0, 1\}$). Figure 2.27 shows how they train the U-net and some examples of translated images. In addition to image-to-image translation, the conditional GAN can also be applied to perform

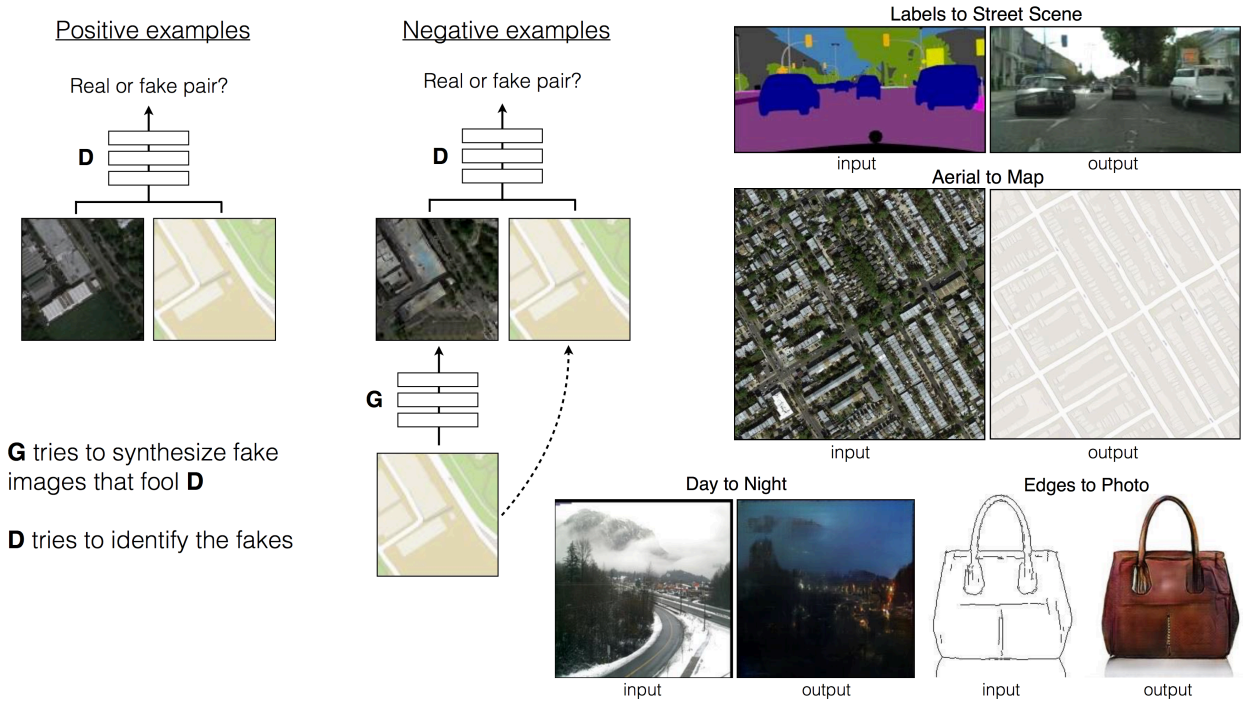


Figure 2.27: Image-to-image Translation [10]. **Left**: U-net trains the generator to synthesize the output image that corresponds to the input image in another domain such as from a map image to an aerial image. **Right**: Several examples of translated images: from semantic label to photo, day to night and sketch to photo.

text-to-image translation [76]. Here, they condition the generator with text-embedded representation (instead of images) and random noises. They also introduce a mismatch loss to Equation 2.85, so that the discriminator is able not only to guide the generator to synthesize images that look like the training data but also to guide the generator to synthesize images that match with the given text description [76].

2.11 Practical Methodologies

In this chapter, we discuss different types of common biosignals that have been analyzed to learn more our health's status to provide appropriate treatments promptly. We also discuss a variety of machine learning techniques that can be used to train a model to perform a given task. Here, we provide a summary of the typical procedure for applying deep learning (or neural networks) in some applications (including biosignals). It consists of six main stages as follows:

- **Defining goals.** We first define the goal of the problems that are trying to solve including

what are the error metrics to use and what are your expected values. In the classification, for example, the typical performance metrics are accuracy, precision, recall, and F1-score. In the regression problem, the mean square error is commonly applied.

- **Preparing data.** Next, we obtain the dataset to evaluate our goal. Then we split the dataset into three separate sets: training, validation and test sets (typically 80%, 10% and 10% of the dataset respectively). The training set will be used for training a model, the validation set will be used for hyperparameter selection, and the test set will be used for Model evaluation. In the biosignal applications, we may also want to remove the artifact from the signals such electrical powerline noises (60 Hz in the United States and 50 Hz in many other countries) [83].

- **Data preprocessing.** We may need to preprocess the data before training to improve its convergence [84]. The common preprocessing techniques including data normalization, in which every training example is subtracted by mean and divided by standard deviation. Note that these statistics must be calculated from the training data, and are used as fixed values to normalize the validation and test sets. This is to simulate its generalization to the real-world situation.

- **Designing model architecture.** We design an architecture of the network and implement a working model to approximate our desired functions such as the function f that we mentioned in the supervised learning. Typically, we are trying to implement a prototype or a baseline model that can overfit a small portion of the training data. For instance, in this thesis, we are trying to consolidate different layers to form a network that can be trained end-to-end to perform particular tasks. If the model can overfit such small training data (i.e., the training loss is close to zero), it implies that our implementation has the potential to approximate the desired functions.

- **Hyperparameter selection.** Next, we find the set of model hyperparameters that achieves the best performance on the validation set. If the available data are not sufficient to be partitioned into the three sets, we can select the hyperparameters using k-fold cross-validation instead. Note that, for deep learning models, this hyperparameter selection may also include modifications to the network architecture as well.

- **Model evaluation.** Finally, we apply our best model based on the performance on the validation set to the test set and report its performance. The performance on the test set can also be improved via model ensembles. In particular, we may train multiple deep learning models using

different sets of hyperparameters or different initializations, and report the average results.

In the next chapter, we will discuss our method to improve the walking speed estimation performance in people with walking impairments such as Multiple Sclerosis (MS).

3

Remote Gait Monitoring for People with Multiple Sclerosis

In this chapter, we discuss our method to improve the walking speed estimation performance in people with walking impairments such as Multiple Sclerosis (MS). This contribution also illustrates an example of the common pipeline of the traditional hand-engineering feature approach, which motivates us to instead employ deep learning to automate the feature engineering process in the remaining chapters.

3.1 Introduction

Tools for the sensitive assessment of clinically meaningful measures of disability and progression in PwMS are needed [85]. A major challenge has been to define measures that meaningfully reflect the concept of interest and are sensitive to change over time for the full range of disability. The most commonly accepted measure to quantify the disability in MS is the Expanded Disability Status Scale (EDSS) [86]. The EDSS shows variable sensitivity to change over time across its range of 0-10 in 0.5 step increments. Walking speed has been increasingly considered as an additional, objective outcome measure of the walking impairments in PwMS [14, 44, 87]. Many studies have also shown that walking difficulties are strongly correlated with progression of MS [88,89], and reduced quality of life [44,90,91].

At present, walking speed is evaluated only in the context of short test walks in clinics standardized as the Timed 25-foot walk (T25FW) or the Six-Minute Walk (6MW) [92]. Even though these tests are

believed to be reliable measures, there are still outstanding questions concerning the extent to which the walking speed measured in the clinic correlates with the speed in longer walking periods or with free walking in the home environment. This is because they are typically sparsely acquired (a single, short time period in the day during an assessment visit), and can be influenced by time-of-day effects such as fatigue [44, 93–95].

Mobile actigraphy is promising as a reproducible, quantitative approach to remote assessment of walking speed for people with MS [44, 96, 97]. Many recent studies have utilized single or multiple actigraphy devices, such as tri-axial accelerometers, worn on different parts of human body such as wrist and waist to collect gait-related data [14, 15, 43, 98, 99]. These actigraphic data can be utilized to build a walking speed estimation model. For instance, machine learning algorithms such as Support Vector Regression (SVR) have been applied to build models that capture the relationships between the gait-related data and their corresponding walking speed. Such models can be then used to automatically estimate speeds for the future data [98].

However, there are three main limitations to these existing approaches. Firstly, some of them have used the gait data from the wrist-worn position. We believe that the wrist-worn position is influenced significantly by factors other than gait movements. This can lead to high walking speed estimation errors. Greater accuracy can be achieved by attaching the device close to the center of mass to reduce the non-gait-related factors that can influence the actigraphic data, and to also capture the asymmetric in gait pattern in the sagittal plane [98]. It has also been shown that, in the controlled environment, the waist-worn position outperformed the wrist-worn position in counting the number of steps [100, 101]. Secondly, some of them utilize actigraphic data from multiple devices, which, in practice, cannot be comfortably worn over several days. It would be clinically more practical of a walking speed model to be built by utilizing actigraphic data from a single device. This is because it can be comfortably worn over several days. Lastly, most of these approaches used actigraphic data from a limited number of MS or healthy-control subjects and/or their demographic information, such as weight, height and step length, to build a single generic model for walking speed estimation. These approaches cannot provide accurate walking speed estimation when the MS subjects have high EDSS scores (e.g., more than 5.0); it has been shown that the ways people walk (i.e., gait patterns) change with impairment and disability [89, 102]. It would be better to build a subject-specific model using actigraphic data and the corresponding walking speeds measured from each individual for subject-specific calibration.

In this chapter, we propose a subject-specific gait calibration approach to build a walking speed es-

timation model for PwMS. Our model utilizes actigraphic data from a single tri-axial accelerometer attached to the lower-back position to estimate walking speeds. We evaluate the estimation performance with 32 MS volunteers, and compare its performance with the baseline generic model. The volunteers are also asked to wear the acceleromter in the home environment up to seven days. These home data are analyzed to investigate the possibility of extracting clinically meaningful information from quantitative gait monitoring in the home environment.

Our main contributions are as follows:

- We demonstrate that our approach can remotely and accurately estimate walking speed independent of disability level in PwMS.
- We show the feasibility and tolerabilty of wearing the accelerometer directly on the lower-back position, close to the body's center of mass, over several days for continuous home gait monitoring.
- We found that the maximum sustained free walking speed in the home environment strongly correlated with the speed measured by the T25FW in the clinic.

3.2 Related Works

There have been several previous studies trying to utilize mobile actigraphy devices to estimate walking speed in home environment. Most of them have tried to apply the existing actigraphy devices, such as actibelt, Fitbit, and Apple iPhone [14, 101], and compare their performance against the measured metrics in the clinics. However, these devices estimate walking speed based on the generic model that assume applicable equally well to all subjects. Such devices cannot provide accurate walking speed estimation for MS subjects with high disability levels. Our approach differs from these works in that we calibrate a model for each MS subject by utilizing his/her actigraphic data collected during standard walking tests at the clinics.

Recently, there has been a study investigating a single generic model for estimating walking speed in PwMS [99]. They have investigated different combinations of the number of actigraphy devices and their locations on subjects' body to determine the best combination for walking speed estimation. They have found that a single device attached to the sacrum (or around the waist) position gave

the best walking estimation performance for PwMS. However, we found that their walking speed estimation errors in MS subjects with EDSS score equal to 6.0 are still relatively high. We believe that this is because gait biomechanics of healthy controls are different from the MS patients with the high disability level. Our approach differs from this work in that we utilize the subject-specific calibration instead of the generic calibration, and we validate our method using freely-moving walking data instead of treadmill walking.

3.3 Methods

3.3.1 Participants

We recruited participants from the MS clinics of Imperial College Healthcare Trust (ICHT). The first set of volunteers were recruited from the MS clinic at the Charing Cross Hospital, London, UK. The second set of volunteers were recruited from the MS clinics at Hammersmith hospital, London, UK. The participants provided demographic information and then underwent a screening evaluation for generating an Expanded Disability Status Scale (EDSS) score. The four criteria for inclusion were: a) clinically-supported diagnosis of multiple sclerosis; b) EDSS score up to and including 6.0 at the screening evaluation; c) able to understand and follow the study instructions; and d) capable of independent walking (with or without an assistive device). An actigraphic device was then attached to each participant at the lower-back position for continuous data collection.

The clinical study protocol for the evaluation of the device and our approach has been approved by the London Bromley National Research Ethics Committee (14/LO/0292). All participants provided written informed consent before beginning the study.

We also collected data from seven healthy subjects in our lab. The data from these subjects will be used to calibrate a generic model. This generic model will be used as a baseline model and applied to the data from the MS volunteers to compare the estimation performance with our subject-specific approach.

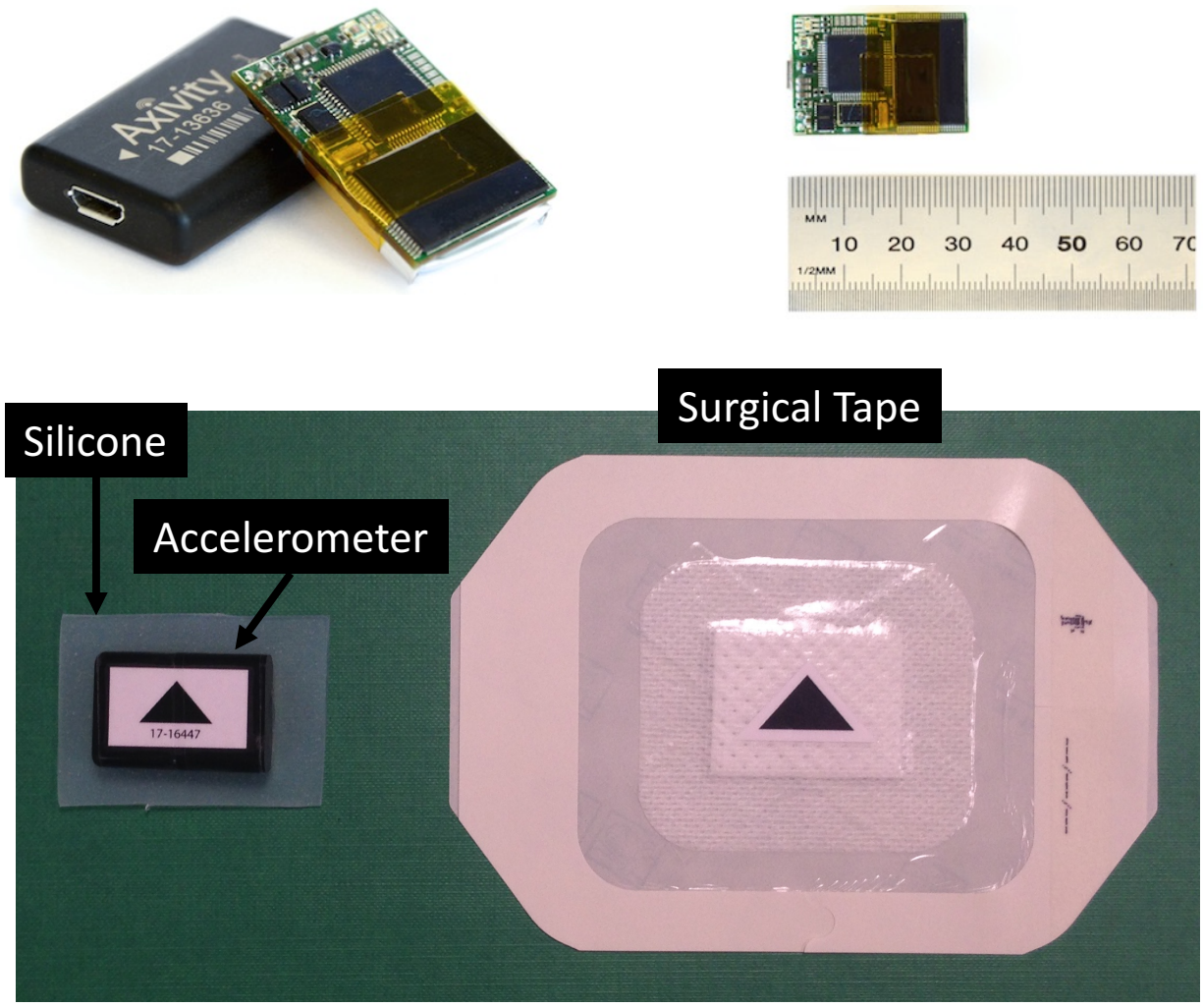


Figure 3.1: Actigraphic Device. We use the AX3-Axivity tri-axial accelerometer protected with silicone backing to remotely and continuously collect data from each subject.

3.3.2 Actigraphy Devices

An AX3-Axivity tri-axial accelerometer protected with silicone backing (see Figure 3.1) is used in this study. This device is attached to each subject at the lower-back position with a form-fitted surgical tape, which allows up to seven days of continuous data collection. This accelerometer measures acceleration forces from three dimensions in units of gravitational force (g-force), where 1 g is equal to the acceleration of a standard gravity (approximately 9.81 m/s^2).

The AX3-GUI software (https://github.com/digitalinteraction/openmovement/wiki/AX3-GUI) provided by Axivity was used to configure and download raw acceleration data from the accelerometer. This software was also used to convert raw acceleration data into comma-separated-value (CSV) files. Each row in the CSV files consists of the timestamp, and the g-forces from three directions: vertical,

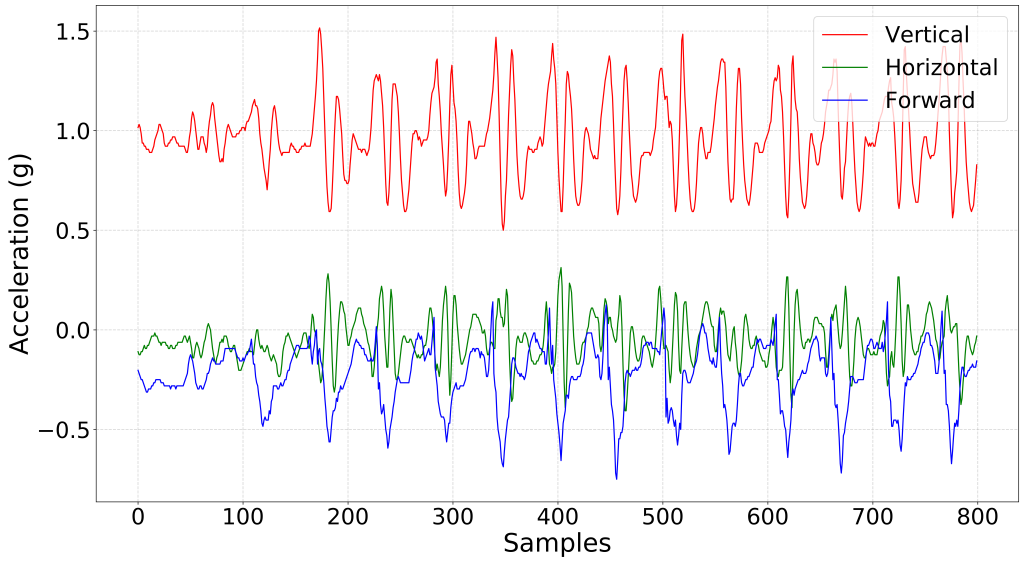


Figure 3.2: An example of raw 3D acceleration data collected from the accelerometer consisting of vertical (red), horizontal (green) and forward (blue) acceleration.

horizontal and forward (referenced to the Earths gravitational field). These CSV files will be used to calibrate the subject-specific model for walking speed estimation, which will be discussed in the next section. Figure 3.2 shows an example of raw 3D acceleration data collected from the accelerometer.

In our initial experiment, we found the sampling rate of 100 Hz (i.e., the resolution of g-force data per one second) and the range of the g-force between -8 g and +8 g were sufficient to capture the gait-related movement data for walking speed estimation.

3.3.3 Data Collection

Each volunteer was asked to complete three phases of the data collection: *calibration*, *validation* and *home monitoring*. The acceleration data from the calibration were used to build a subject-specific model for walking speed estimation. The data from the validation were used to evaluate the estimation performance of the subject-specific model relative to the earlier approach. The data from the home monitoring were used to investigate the possibility of learning clinically meaningful information about the MS volunteers from home environment.

During the data collection at the clinics, a researcher walked along side the volunteer for safety and to record the time and distance walked (meters) using a measuring wheel. The speed of each walk was calculated as distance (meters) / time (seconds). We provided instructions to stand still for at least

five seconds at the beginning and the end of each walk to separate the acceleration data of each walk. At the end, each participant was asked to provide feedback about the comfort of the device (yes/no question), and suggestion for improvement.

Calibration

Volunteers walked in the clinic in two different tests: a Timed 25-Foot Walk (T25FW) and a 2-Minute Walk (2MIN). A researcher recorded the time and distance walked, and calculated the walking speed.

- **Timed 25-Foot Walk (T25FW).** Volunteers walked for 25 feet at least 8 times at slow and normal, voluntarily-set walking speeds.
- **2-Minute Walk (2MIN).** Volunteers walked for 2 minutes in an open corridor twice at slow and normal, voluntarily-set walking speeds.

Validation

Volunteers recruited from two hospitals walked in two different settings: first and second at their preferred speed (**VALID**). The researcher recorded the time and distance walked, and calculated the walking speed. The attached devices remained in the same position as the volunteers leaved the clinic.

- **First setting.** Volunteers completed a single circuit with their preferred speed along the rectangular corridor around a lift.
- **Second setting.** Volunteers walked for two minutes along the straight corridor.

The corridors of these hospitals were clear of obstructions, but there were minor foot traffic.

Home Monitoring

Volunteers wore devices continuously for up to 7 days (including sleeping, if tolerated). The device was then posted back to our study lab. This phase is optional. The volunteers can choose not to wear the device at home and return it to the researcher in the clinics.

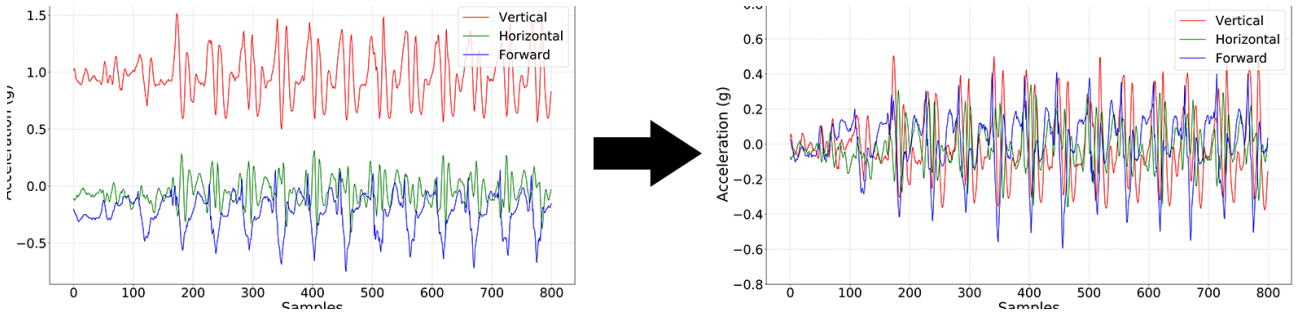


Figure 3.3: Preprocessing. Tilt-adjustment algorithm [11] is applied to each walking data to transform from the arbitrarily tilted accelerometer into a relevance sagittal plane (or a horizontal-vertical orthogonal plane). Then the gravitational force (i.e., 1 g) is removed from the vertical acceleration. After that, a low-pass filter with a fourth-order zero-lag Butterworth filter at 20 Hz is applied on each walking data. This is to remove noise and artifact at the higher frequency that are not related to the gait or walking patterns.

3.3.4 Subject-specific Gait Calibration

For each subject, the data collected during the calibration phase were used to build a subject-specific model for walking speed estimation. There are four main steps in the subject-specific gait calibration: preprocessing, step extraction, feature extraction, and model calibration.

Preprocessing

This step is to process the collected raw acceleration data into the format that can be used to calibrate a model. In particular, the acceleration data during the calibration phase are first split into the data of each walk period (i.e., *walking data*) using the standing periods between each walk. Each walking data is then tilt-adjusted using the algorithm in [11] to transform data from an arbitrarily tilted accelerometer into a relevance sagittal plane (or a horizontal-vertical orthogonal plane). After that, the gravitational force (i.e., 1 g) is removed from the vertical acceleration. Next, a low-pass filter with a fourth-order zero-lag Butterworth filter at 20 Hz is applied on each walking data. This is to remove noise and artifact at the higher frequency that are not related to the gait or walking patterns. Figure 3.3 shows an example of preprocessed walking data.

Step extraction

This step is to extract what we call *step data* from the walking data. For each walking data, the forward acceleration data is used to identify the timestamp between each foot-contact (see Figure 3.4).

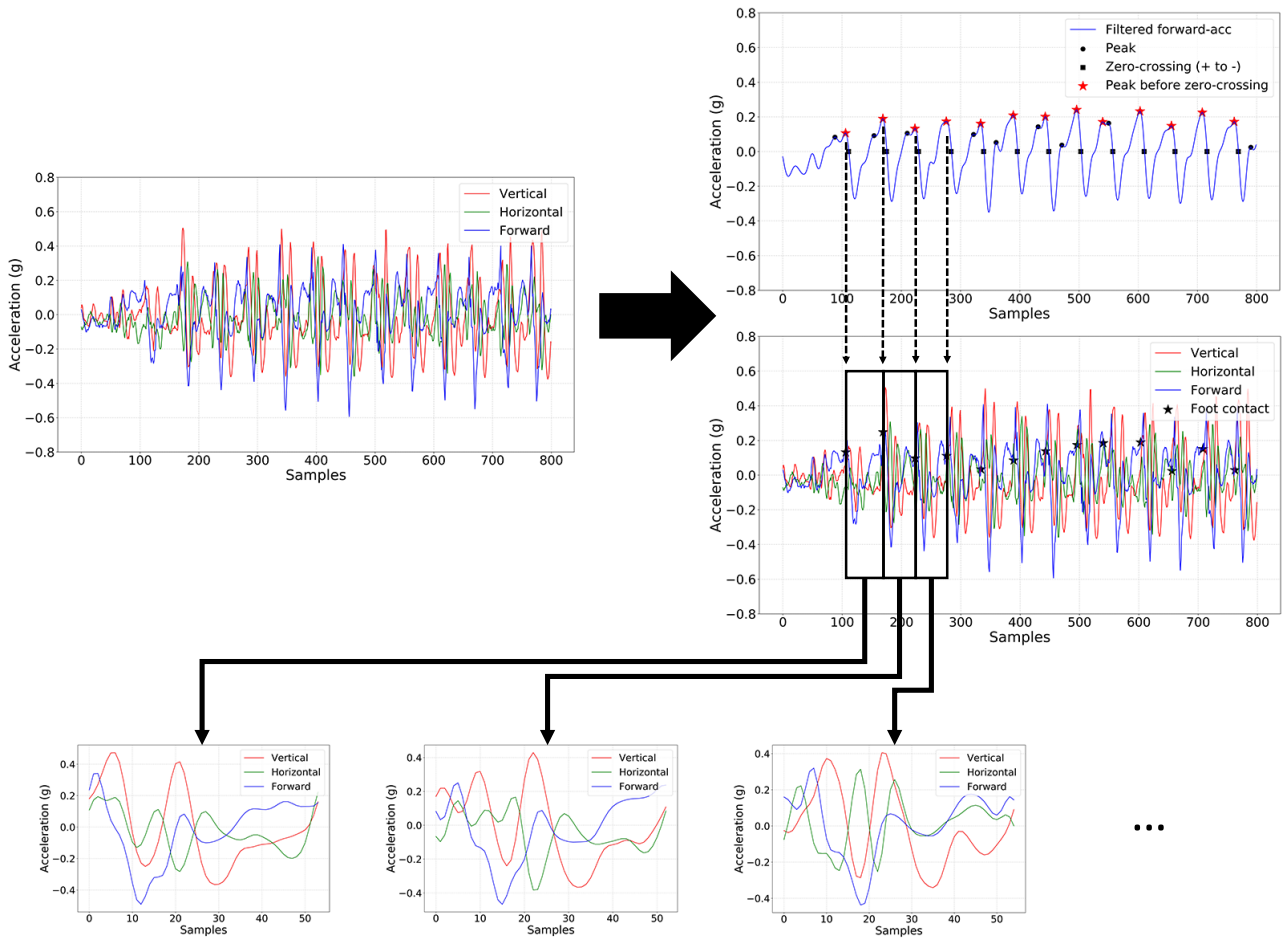


Figure 3.4: Step extraction. The forward acceleration is first low-pass-filtered with a fourth-order zero-lag Butterworth filter at 4 Hz to smooth the data. A peak of the filtered acceleration preceding a change of the sign from positive to negative are used to identify an instance of a foot-contacts [4]. The acceleration data between two foot-contacts are extract as step data.

Specifically, a peak of low-pass-filtered forward acceleration preceding a change of the sign from positive to negative is marked as an instance of a foot-contact [4]. The acceleration data between two foot-contacts are extracted as step data. As each walking data is annotated with the speed manually measured by the researcher (see Section 3.3.3), all step data extracted from the same walking data are annotated with the same walking speed.

Feature extraction

This step is to extract features that captures gait patterns from each step data. Table 3.1 shows 29 step features that are extracted from each step data in this study. Figure 3.5 shows examples of how to extract some of these features. Most of these features are inspired by [98].

Table 3.1: 29 features derived from each step data.

Feature	Equation	Dimension of acceleration to compute features
Mean	$\frac{\sum_{i=1}^{n} s_i}{n}$	*energy* acceleration
Variance	$\frac{\sum_{i=1}^{n}(s_i - \text{Mean})^2}{n}$	*ver*, *hor*, fwd and *energy* acceleration
Maximum	$\max(s_1, \ldots, s_n)$	*ver*, *hor*, fwd and *energy* acceleration
Minimum	$\min(s_1, \ldots, s_n)$	*ver*, *hor*, fwd and *energy* acceleration
Range	Maximum - Minimum	*ver*, fwd and *energy* acceleration
Maximum - Mean	Maximum - Mean	*ver*, *hor*, fwd and *energy* acceleration
Step amplitude	$\frac{\text{Maximum} - \text{Mean}}{n}$	*ver*, *hor*, fwd and *energy* acceleration
Root mean square (RMS)	$\sqrt{\frac{\sum_{i=1}^{n} s_i^2}{n}}$	*ver*, *hor*, fwd and *energy* acceleration
Step duration	$\frac{n}{F_s}$	Number of samples from any dimension

Note: Each step data consists of n values of $s_1, \ldots, s_n$ from three dimension: vertical (*ver*), horizontal (*hor*) and forward (fwd). *energy* acceleration can be computed from $e_i = \sqrt{ver_i^2 + hor_i^2 + fwd_i^2}$. F_s is the sampling rate of the recording device.

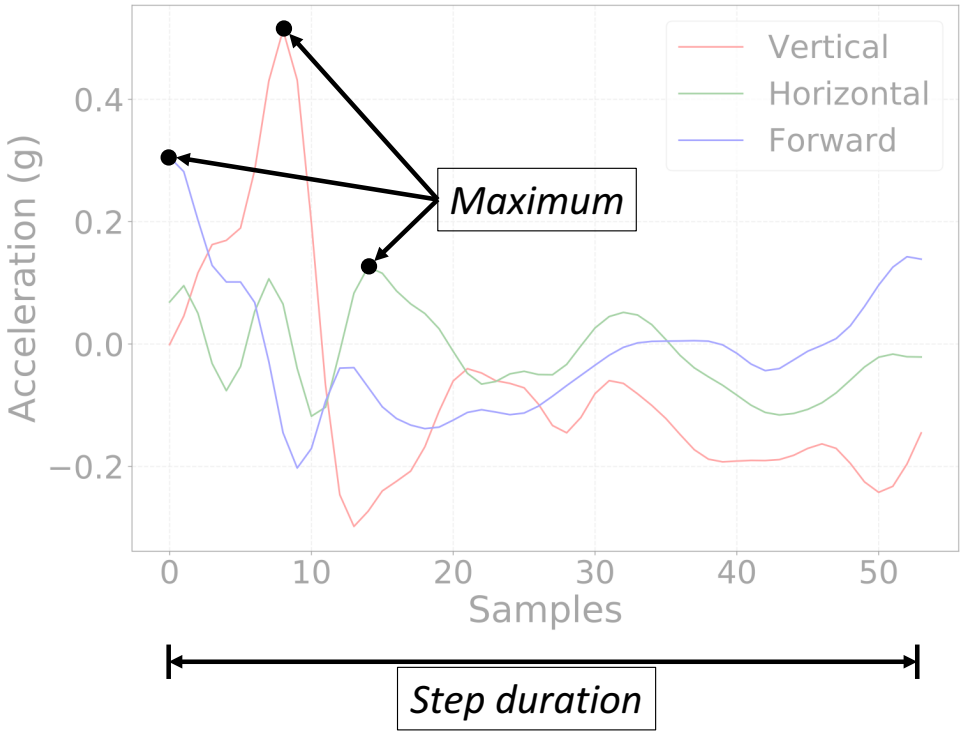


Figure 3.5: Examples of extracting step duration and maximum features from step data.

Table 3.2: Ranges of hyperparameters of the SVR model supplied during model calibration for each subject.

Hyperparameter	Range
Kernel	{"Linear", "Gaussian"}
C	$\{2^{-8}, 2^{-6}, \ldots, 2^{8}\}$
epsilon (ε)	0.01
gamma (γ) (for Gaussian kernel)	$\{2^{-8}, 2^{-6}, \ldots, 2^{8}\}$

Model calibration

Support Vector Regression (SVR) [47], one of the machine learning algorithms, is used to build a model that estimates walking speed from the input step features. Formally, suppose there are m step data annotated with their corresponding walking speed $\{(\mathbf{x}^{(1)}, y^{(1)}), \ldots, (\mathbf{x}^{(i)}, y^{(i)}), \ldots, (\mathbf{x}^{(m)}, y^{(m)})\}$ in the training data[1], where $\mathbf{x}^{(i)}$ is the i-th step feature, $\mathbf{x}^{(i)} \in \mathbb{R}^n$, n is the number of step features ($n = 29$ in this study), and $y^{(i)}$ is a floating-point number of the corresponding walking speed (i.e., ground-truth). The SVR model represents a function that estimates a walking speed for each step data $\mathbf{x}^{(i)}$ as follows:

$$\hat{y}^{(i)} = \mathrm{SVR}_\theta(\mathbf{x}^{(i)}) \tag{3.1}$$

where $\hat{y}^{(i)}$ is the estimated walking speed of $\mathbf{x}^{(i)}$, and θ is the SVR model parameters.

During the model calibration, the parameters θ are optimized using the training data such that the SVR model can estimate $\hat{y}^{(i)}$ that deviates from $y^{(i)}$ by a value no greater than an epsilon ε for each $\mathbf{x}^{(i)}$. This calibration process stops when it has reached convergence. The hyperparameter of the SVR model such as kernel function, C and ε are chosen via a grid search with 10-fold cross-validation on the training data. The best hyperparameters are the ones that give the lowest mean-squared-error (MSE), and they vary across subjects. Table 3.2 shows a range of the hyperparameters supplied during the model calibration.

3.3.5 Walking Speed Estimation

Once a model has been built and calibrated for each subject, it will be applied to the upcoming acceleration data from the same subject. As our model estimates the walking speed for each step

[1] As we only have one walking speed per walk (manually measured by the researcher, see Section 3.3.3), we used the *same* walking speed for all step data extracted from that walk. A measuring wheel that can estimates both the distance and the speed of walking can be used to provide a more fine-grained speed for each step. We leave this investigation as our future work.

feature (see Equation 3.1), the speed of each walking period can be obtained by computing the *average* of the speed estimated from all steps during such walking period. Formally, suppose there are w step features extracted from one walk from one subject $\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(w)}\}$. The speed of this walk is computed as follows:

$$\hat{y}_{walk} = \sum_{i=1}^{w} \text{SVR}_\theta(\mathbf{x}^{(i)}) \tag{3.2}$$

$$= \sum_{i=1}^{w} \hat{y}^{(i)} \ , \tag{3.3}$$

where $\hat{y}_{walk}$ is the estimated speed for a walk, and SVR_θ is the calibrated model for a subject.

Our model estimates walking for each step data instead of the whole walking data because of two main reasons. Firstly, people can walk with different pace speeds for one sustained walk. It would be more accurate to estimate walking speed for each step, in which the walking speed for the sustained walk is the average of all step speeds, than for each walk. Secondly, splitting a sustained walk into steps helps ease the feature extraction process as it is easier to derive interpretable features from the step data than from the sustained walk data.

3.3.6 Data Analysis

Walking speed estimation performance. The VALID data from each MS subject were used to evaluated the walking speed estimation performance of our approach. In particular, we computed the estimation errors between *the speeds manually measured by the researcher* and *the speeds automatically estimated from the calibrated model*. The errors were obtained by subtracting the measured walking speed from the estimated one (i.e., estimated - measured). The positive and negative errors implied over- and under-estimations respectively. To analyze the estimation errors, we generated plots of the errors relative to either *walking speed* or *disability*. In particular, we plot the errors against the average between the speed manually measured and the speed automatically estimated (i.e., (measured + estimated)/2). This plot is called *Bland-Altman* plot [103], which is typically used to test for systematic bias in the estimation errors. For instance, it can be used to identify whether most of the estimation errors are from the overestimations of slow walks or the underestimations of fast walks. We also plot the errors across EDSS scores to test whether the errors increase with disability levels. Linear regression analyzes were then performed by regressing the measured walking speed (dependent variable) on the estimated walking speed (independent variable) across MS subjects. We analyzed the

R-value, computed using Pearson's R coefficient [104], to determine the predictability of the calibrated model. A good model would have the R coefficient close to 1.

Home Monitoring. For each MS subject, we compared *the average of the maximum sustained walking speeds remotely estimated from the home environment* (**MAX-SUS-HOME**) with *the average walking speeds measured during T25FW with normal voluntarily-set speeds in the clinics* (**T25FW-NORMAL**). The MAX-SUS-HOME was computed by averaging the daily maximum *sustained* walking speeds. The sustained walking speeds refer to the speeds estimated from the walks whose periods were longer than 20 seconds. We considered only the sustained walks to avoid the short periods of body movements that were not related to walking. The comparison between MAX-SUS-HOME and T25FW-NORMAL were performed using Bland-Altman plots, plots of the speed differences versus EDSS scores and linear regression analyzes.

Device comfort. We computed the percentage of the volunteers who felt comfortable wearing our device against the total number of recruited volunteers. Suggestion for the improvement of our device were also reported.

3.4 Results

3.4.1 Sample Characteristics

There were 32 MS volunteers (15 men/17 women, age: 39.9 ± 8.6) recruited in this study, in which 25 of them were from the Charing Cross Hospital and seven of them were from the Hammersmith Hospital. Based on the screening evaluation, the median EDSS score was 2.5 with a range between 1.0 and 6.0. The volunteers were categorized into three groups based on their EDSS score as follows: low (EDSS=1.0-3.5, n=20), moderate (EDSS=4.0-5.5, n=6), and high (EDSS=6.0, n=6). The six volunteers who had the EDSS score of 6.0 used a single-point walking aid devices (cane or crutch). There were 22/32 MS volunteers (11 men/11 women, age: 39.8 ± 8.9) who agreed to wear our device for up to one week in home environment. 13, 4 and 5 of them were from low, moderate and high disability group respectively.

3.4.2 Baseline Methods

To compare the speed estimation performance with the existing approaches, we utilize the data obtained from the seven healthy subjects in our research group (7 men/1 woman, age: 29.7 ± 3.9). They were asked to perform T25FW at three different speeds (slow, normal and fast) at least 12 times in a control environment. The time and distance walked were recorded and used to calculate walking speeds. The mean ± standard-deviation (SD) of the slow, normal and fast speeds are 0.63 ± 0.10, 1.10 ± 0.15, and 1.62 ± 0.30 m/s, respectively. These walking data and their corresponding walking speeds were used to calibrate a single generic model.

3.4.3 Walking Speed Estimation Performance

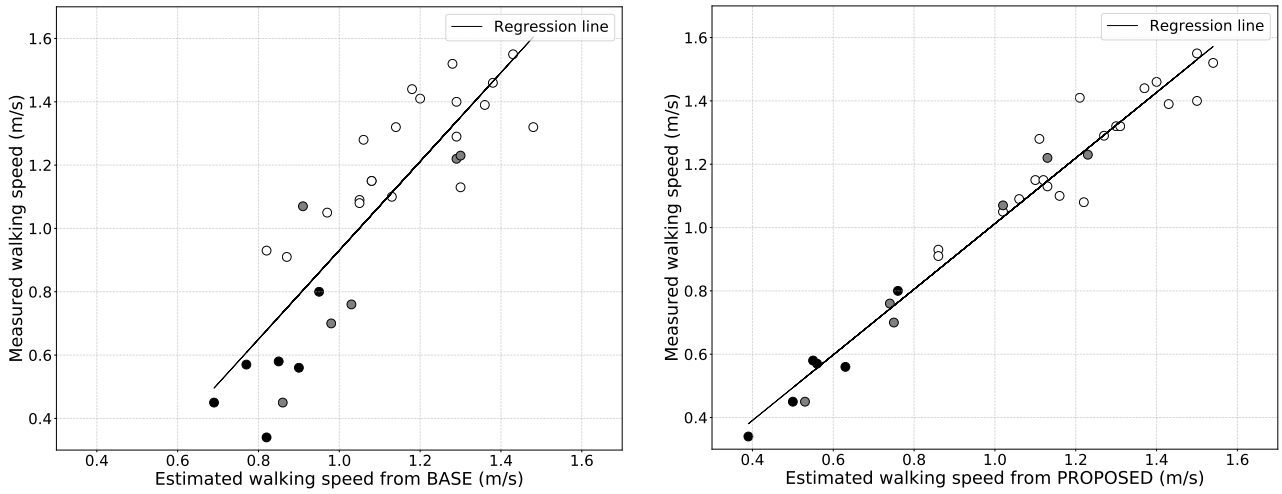
We estimated walking speed for the MS patients based on the VALID data using either a model derived from gait characteristics of the healthy subjects (**BASE model**) or one calibrated individually for each of the patients based on the calibration data in the clinic (**PROPOSED model**). We found that the estimation error of the BASE model increased according to the disability level (see Table 3.3). The errors in estimation of gait speed from the baseline approach were almost four-fold greater in the group with highest relative to the lowest disability. We also found that the BASE model (R-value=0.85, p-value=4.74×10^{-10}) (Figure 3.6a) had a poorer accuracy for measure of walking speed on the VALID data than did the PROPOSED model (R-value=0.98, p-value=1.85×10^{-22}) (Figure 3.6b). The reported p-values were from hypothesis tests whose null hypothesis is that the slope is zero. The plots of differences between the measured and estimated walking speeds relative to their means highlight a bias towards overestimation with those walking more slowly (Figure 3.7a top), and the plots relative to EDSS scores highlight an overestimate of speed for those with higher disability (Figure 3.7b top) using the BASE model. In contrast, the PROPOSED model estimated walking speeds with high accuracy across the full range of walking speed (Figure 3.7a bottom) and disability (Figure 3.7b bottom) in our study population.

3.4.4 Home Monitoring

We then used the calibrated model of our approach to estimate MAX-SUS-HOME speeds in the home environment from actigraphy measures acquired during subjects' usual activities. Maximum sustained walking speeds over a single day of one MS subject are illustrated in Figure 3.8. We compared these

Table 3.3: Comparison between the walking speed estimation performance of the baseline (**BASE**) and our subject-specific calibration (**PROPOSED**) approaches on the VALID data. The values in the table are mean (standard deviation). **Measured** indicates the walking speed manually measured by the researcher during the data collection. **Estimated** indicates the walking speed automatically estimated by each approach. **Error** refers to the difference in walking speed between the measured and estimated speed (i.e., Estimated - Measured). **Low**, **Moderate** and **High** refers to the MS volunteers with EDSS scores of 1.0-3.5, 4.0-5.5 and 6.0 respectively.

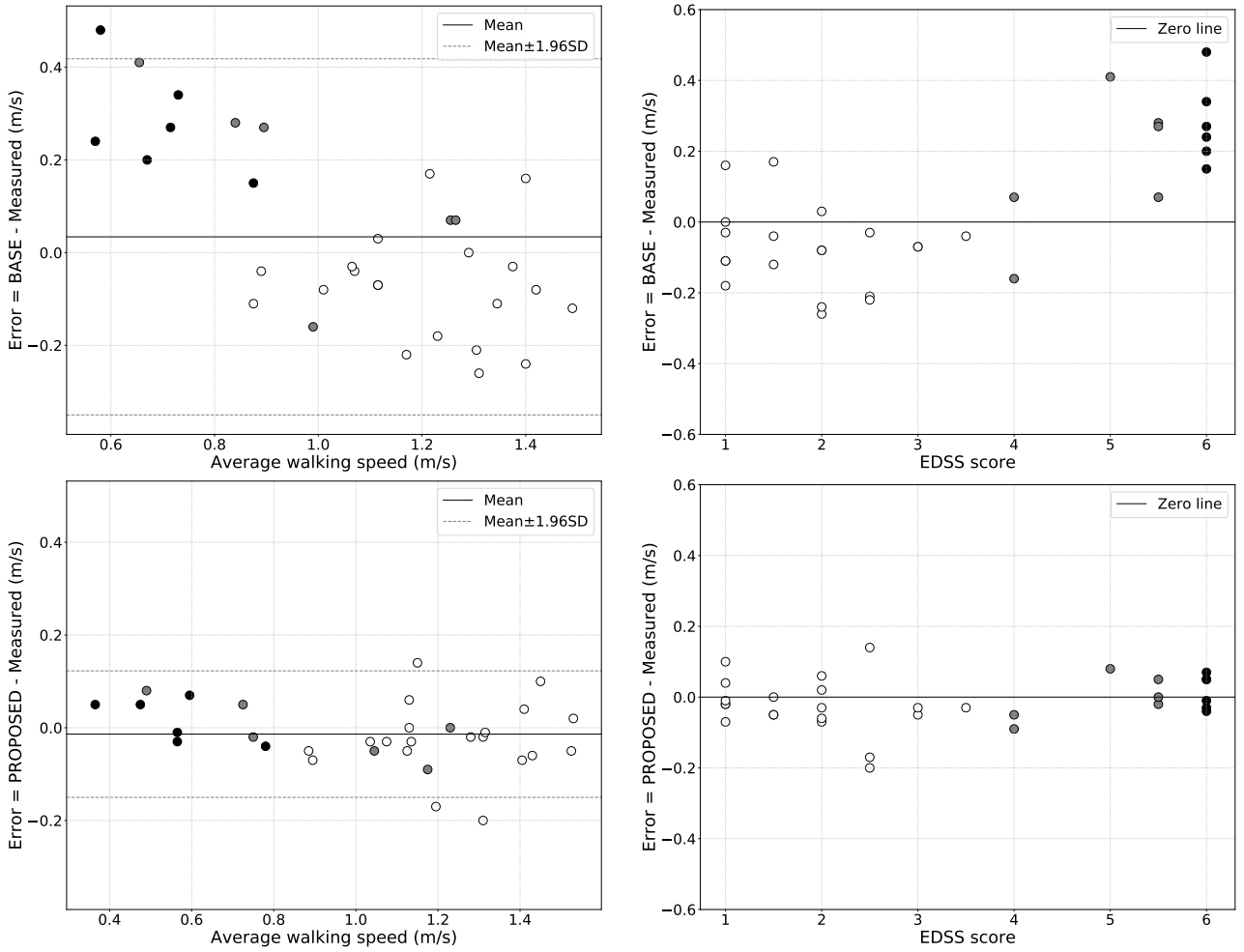
Group	Measured (m/s)	BASE (m/s)		PROPOSED (m/s)	
		Estimated	Error	Estimated	Error
Overall (n=32)	1.05 (0.35)	1.09 (0.21)	0.03 (0.20)	1.04 (0.33)	-0.01 (0.07)
Low (n=20)	1.25 (0.19)	1.17 (0.18)	-0.08 (0.12)	1.22 (0.20)	-0.03 (0.08)
Moderate (n=6)	0.91 (0.32)	1.06 (0.19)	0.16 (0.20)	0.90 (0.27)	-0.01 (0.06)
High (n=6)	0.55 (0.15)	0.83 (0.09)	0.28 (0.12)	0.57 (0.12)	0.02 (0.05)



(a) BASE model **(b)** PROPOSED model

Figure 3.6: Comparison of correlations between manually measured walking speeds for the full group of patients and either that estimated from actigraphy data using **(a)** the BASE model (R-value=0.85, p-value=4.74×10^{-10}) or **(b)** the PROPOSED model (R-value=0.98, p-value=1.85×10^{-22}). The white-filled, grey-filled circles, and black-filled circles denote MS subjects from low (EDSS=1.0-3.5), moderate (EDSS=4.0-5.5) and high (EDSS=6.0) disability subgroups.

MAX-SUS-HOME speeds in the home environment with the T25FW-NORMAL speeds in the clinic for each subject to test how well the clinic measure predicts home performance (see Table 3.4). There was a strong correlation between maximum sustained walking speeds at home and the T25FW walking speed measured in the clinic (R-value=0.89, p-value=4.34×10^{-8}) (Figure 3.9). Also, we did not find evidence for a bias in the accuracy of the clinic T25FW prediction of the maximum sustained home walking speed with either gait speed or disability (Figure 3.10).



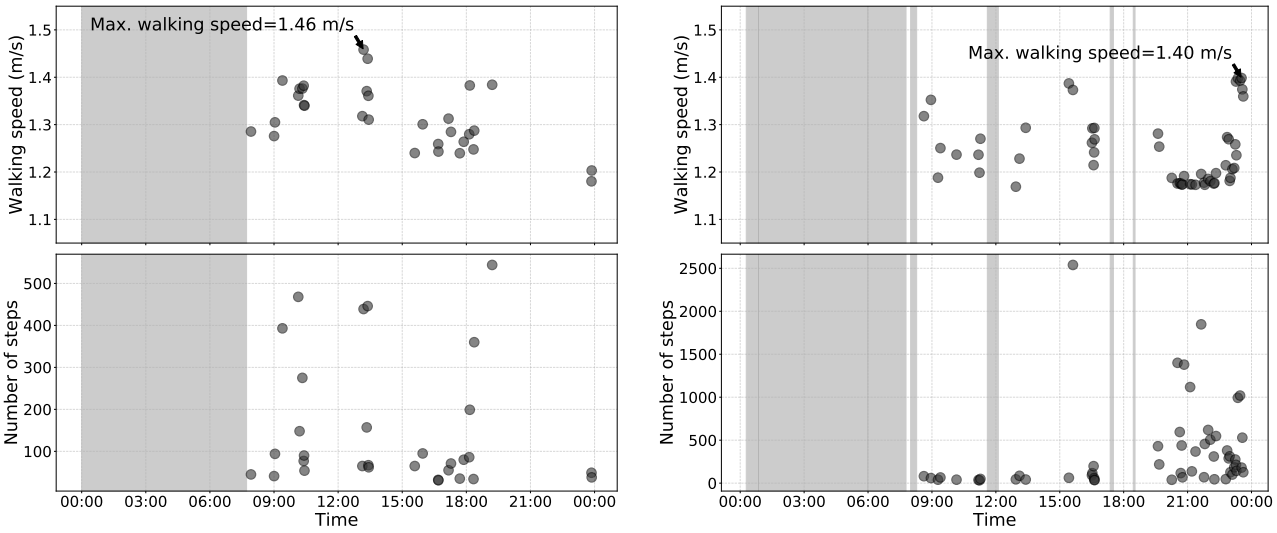
(a) Bland-Altman plots.

(b) Estimation errors across EDSS.

Figure 3.7: Plots of differences between the measured and estimated walking speeds relative to **(a)** their means (or Bland-Altman plots) and relative to **(b)** EDSS scores. The top graphs are the results from the BASE model, while the bottom ones are from the PROPOSED model. Greater estimate error is evident for patients with higher disability (i.e., slower speeds) with the BASE model.

3.4.5 Device Comfort

29/32 participants (or approximately 91%) reported that they felt comfortable while wearing the device for extended periods in both the clinic and home environments. This feedback suggested that device could be worn on the lower back continuously without concern by most subjects for at least a week. Three participants felt uncomfortable wearing the device. One participant reported that the device pressed uncomfortably on the back when sitting on a chair. Another experienced an uncomfortable subjective sensation of "warmth". The third participant was unable to reposition the device without assistance when at home.



(a) Day-1 **(b)** Day-2

Figure 3.8: Examples of estimated walking speeds (top) and the number of steps (bottom) from two days of home data from one MS subject. The walking speeds and the number of steps are estimated from sustained walks. The grey areas indicate the lay-down periods (including sleeps). The maximum sustained walking speeds of Day-1 and Day-2 are 1.46 and 1.40 m/s, respectively.

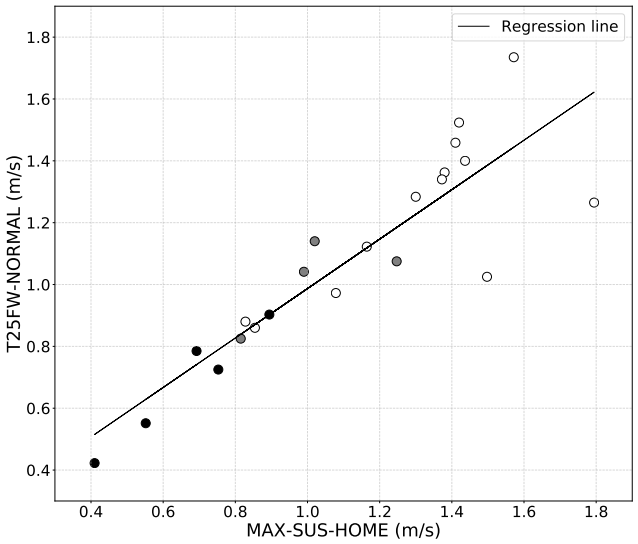
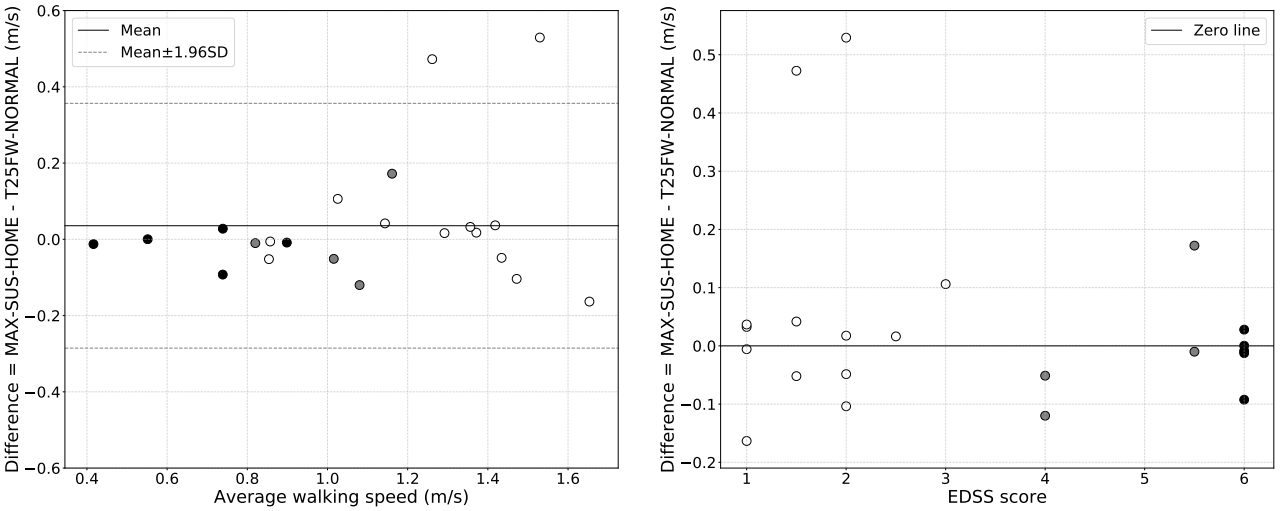


Figure 3.9: Correlations between the average of the maximum sustained walking speeds estimated from home environment (**MAX-SUS-HOME**) and the average walking speeds measured during T25FW with normal voluntarily-set speeds in the clinics (**T25FW-NORMAL**) across subjects (R-value=0.89, p-value=4.34×10^{-8}). These results suggest good agreement across the range of disability.

3.4.6 Comparison on Step Features

We performed a Principal Component Analysis (PCA) [50] of step features to investigate whether there were any different in gait patterns among healthy subjects, and MS subjects with low and high



(a) Bland-Altman plots.

(b) Difference across EDSS.

Figure 3.10: Testing for systemic bias in agreement between the MAX-SUS-HOME and the T25FW-NORMAL speeds. First, we assessed the difference between the MAX-SUS-HOME speeds and the T25FW-NORMAL speeds using a Bland-Altman plot **(a)**. The differences did not appear to change with walking speeds. Second, we tested variation of such differences relative to EDSS scores **(b)**. No significant change was seen, consistent with clinical meaningfulness of the EDSS score measure across the disability range.

disability levels. Specifically, we first selected several subjects randomly from the healthy, low disability and high disability groups. We then extracted step features from walking data during T25FW and 2MIN, and applied the PCA to compute the first and second principal components for visualization. Figure 3.11 shows scatter plots of the first and second principal components of the normal and slow walking data among the three groups (healthy, low disability and high disability). It can be seen that the second component of walking speeds from the higher disability groups only fall in a range of 0-5 (yellow triangles), while the others vary from approximately 0 to 11 (blue circles and red squares). This implies that gait patterns of the subjects from the high disability group differ from those of the subjects that have no or low disability. We also found that there was not obvious difference between features from the slow walking speed. This might be because the trunk acceleration during slow walks are similar regardless of the disability level.

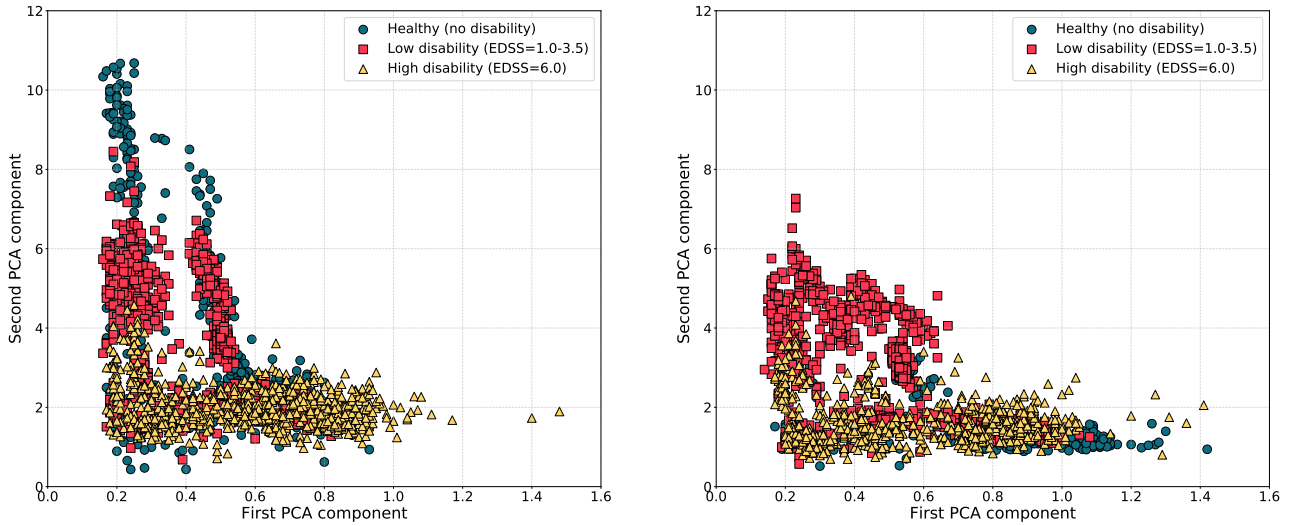
3.5 Discussion

We propose the subject-specific gait calibration model that utilizes 3D acceleration data from a tri-axial acceleromter to estimate walking speed for PwMS. We have demonstrated that our approach can estimate walking speed remotely and accurately independent of the subject's disability level. We

Table 3.4: MS subject demographics, their average walking speeds measured during Timed 25-Foot Walk with normal voluntarily-set speeds in the clinic (T25FW-NORMAL), and their average of the maximum sustained walking speeds estimated daily from the home environment (MAX-SUS-HOME).

Subject	EDSS	Age	Gender	T52FW-NORMAL speed (m/s)	MAX-SUS-HOME speed (m/s)
1	2	45	M	1.5	1.4
2	2	30	F	1.1	-
3	5.5	43	M	0.8	-
4	1	30	F	1.3	-
5	5	47	M	0.8	-
6	2.5	33	M	1.2	-
7	3	37	F	1.1	-
8	2.5	36	F	1.0	-
9	3	53	F	1.0	1.1
10	2	42	M	1.5	1.4
11	6	48	F	0.8	-
12	1	41	F	0.9	0.9
13	2	29	M	1.4	1.4
14	1.5	34	M	0.9	0.8
15	4	46	F	1.1	1.0
16	3.5	52	M	1.0	-
17	4	35	F	1.0	1.0
18	6	54	F	0.9	0.9
19	2	33	F	1.3	1.8
20	1	46	F	1.6	-
21	2.5	39	F	1.3	1.3
22	1.5	32	F	1.1	1.2
23	6	47	F	0.6	0.6
24	6	41	M	0.8	0.7
25	6	49	M	0.4	0.4
26	6	59	M	0.7	0.8
27	1	28	F	1.7	1.6
28	5.5	32	M	0.8	0.8
29	5.5	35	F	1.1	1.2
30	1.5	28	M	1.0	1.5
31	1	29	M	1.3	1.4
32	1	45	M	1.4	1.4

have also confirmed that the baseline or the standard generic model for transformation of acceleration data to walking speed lead to inaccuracies in PwMS; such inaccuracies increase with disability level. We believe that this is due to the gait variability of the trunk among MS subjects across EDSS as



(a) Normal speed. **(b)** Slow speed.

Figure 3.11: Scatter plots of the first and second PCA components of step features. These features are extracted from several normal (a) and slow (b) walking data during T25FW and 2MIN of randomly chosen MS and healthy subjects.

shown in Figure 3.11. Therefore, the use of subject-specific calibrated model can mitigate this gait variability problem. We have also demonstrated that our device can be comfortably worn for up to one week. Feedback about the comfort of the device will be used to improve the design of our device in the future work.

To our knowledge, this is the first study that has attempted to validate the walking speed measured in the clinic against the maximum sustained walking speed in the home environment. According to our analysis, we found that the maximum sustained walking speed in home environment strongly correlated with the speed measured by the T25FW in the clinic. However, there were two MS subjects from the low disability group (Subject 19 and 30 in Table 3.4) whose the maximum sustained walking speed at home were significantly different from the walking speed at the clinics. We speculate that this may have arisen, e.g., because of lower relative effort in the clinic or other factors such as fatigue at the time of the testing or greater reserve gait speed capacity. Further studies are needed to investigate factors influencing the predictive accuracy of clinical walking performance measures. Nevertheless, these results suggest that there was a very good agreement between the clinic and home walking speed measures overall.

Even though the subject-specific model can provide more accurate speed estimation compared to the generic model, it requires subject-specific data for the model calibration. This means that the initial data collection is required for new MS subjects. We believe that this is problem can be overcome as

follows. The generic model calibrated with healthy controls can be first used to estimate walking speed new MS subjects. After the subjects undergo their regular diagnosis of the disease progression with our device attached, the collected data can then be used to improve the speed estimation performance. This technique can also be used to update the model when there is a change in the disease progression of each subject.

In the future, we plan to extend our study to a larger MS population to strengthen our findings of the validity of the walking speed from T25FW tests across a range of disability. We may also ask volunteers to also complete short diary on daily activities or self-report questionnaires [105] while they are wearing our device at home. The actigraphic data and the patients' reports can be used not only to identify the causes of differences between the walking speed in the clinical and in home environments, but also to define clinically meaningful for changes in walking speeds estimated in the home environment associated with the patient questionnaires. This could be related directly to effects of treatments. We could also assess sensitivity of both gait speed and gait characteristics to changes over time across a range of disability. Moreover, there are also some other gait-related metrics (as mentioned in Section 3.4.4) that can be extracted from home data such as activity counts[2], the number of interruptions during sleep, and the amount of lay-down periods in each day. These metrics might help us to have a better understanding of the potential factors that affect the daily walking performance in PwMS (i.e., time-of-day effects). Future work can also be to explore whether continuous home monitoring of these gait metrics provides a better index of treatment responses than does the sparsely sampled clinic measure, or a more sensitive measure of disability progression over time.

3.6 Conclusion

We propose a subject-specific gait calibration approach to build a walking speed estimation model for PwMS. Our model utilizes actigraphic data from a single tri-axial accelerometer attached to the lower-back position to estimate walking speed. According to our results, we have demonstrated our approach can remotely and accurately estimate walking across a range of disability, with the overall error of -0.01 $\pm$ 0.07 m/s. This has shown that the subject-specific calibration is a better approach for walking speed estimation in PwMS compared to the generic calibration. We have also, for the first time, found that the average of the maximum sustained walking speed in home environment

[2]https://actigraph.desk.com/customer/en/portal/articles/2515580-what-are-counts-

estimated from our model strongly correlated with the speed measured by the T25FW in the clinic. Even though additional studies are still required to strengthen our findings, the results nevertheless suggested that T25FW tests may be an appropriate test that can estimate the maximum sustained walking speed in the home environment. These results support the application of mobile actigraphy for remote gait monitoring for PwMS, which allows doctors to have a better understand of the MS disease progression, and can, therefore, make appropriate treatment decisions.

It can be seen from this chapter that there are several ways to extract features such as our approach (see Section 3.3.4) and the one mentioned in [99]. Both of them requires domain-expert knowledge to develop algorithms to extract features from acceleration data that are useful for walking speed estimation. It can be labor-intensive and time-consuming to come up with features that are useful for any given tasks. The recent study [49] has shown that an alternative choice of machine learning algorithms, named **Deep Learning**, can be used to automate the feature engineering process through multiple layers of non-linear transformations given that we have sufficient training data. We believe that it would be better to include this feature engineering process as a part of machine learning algorithm that can be trained end-to-end based on a target loss function. This has motivated us to develop deep learning models for analyzing biosignals in the remaining chapters. Also, as the annotated acceleration data we collected from each MS subject are not sufficient to effectively train a deep learning model to extract features for walking speed estimation, the remaining chapters will focus on developing deep learning models for the problems that typically have a plenty amount of training data such as epileptic seizure detection and sleep stage scoring.

4

Epileptic Seizure Detection

In the previous chapter, we found that it can be labor-intensive and time-consuming to hand-engineer useful features. There are also many possible ways to develop algorithms to extract features that are useful for any given tasks, and these algorithms are typically application-specific.

In this chapter, we discuss our proposed model that utilizes deep learning to automate the feature extraction process, and demonstrate that such model can be used to detect epileptic seizures from unlabeled scalp EEG signals.

4.1 Introduction

Almost 50 million people around the world suffer from epilepsy [106]. It is one of the most common neurological disorders associated with transient, recurrent and unpredictable epileptic seizures. These seizures are periods of abnormal synchronous neuronal activity in the brain [107]. During epileptic seizures, people might loss consciousness or have uncontrollable muscle movements, leading to a greater risk of accidents or physical injuries. According to the statistics from [108], only 7-8% of people with epilepsy can be cured by surgery, and another 2/3 of them have to use medications to control their seizures. This leaves more than 20% of them whose their seizures cannot be controlled by any currently available treatments.

Typically, these seizures can be diagnosed and detected by long-term monitoring of electroencephalogram (EEG) signals. This information is very important for neurologists to quickly provide treatments

for patients. This long-term monitoring of EEG signal is, however, an expensive, repetitive and time-consuming process. This is mainly because neurologists have to go through the recorded EEG signals visually to identify the periods of seizures (i.e., labeling process) [109, 110]. Thus a system capable of automatically identifying epileptic seizures in real-time would help neurologists reduce their workload on labeling seizure periods, which can lead to an improvement in the therapy of epilepsy.

A number of researchers have proposed methods for seizure detection to help improve the quality of life for people with epilepsy. The majority of them utilize expert knowledge to develop algorithms for extracting useful features from scalp EEG [18, 21, 109, 111–118] or intracranial EEG [19, 119, 120] signals. These methods are typically referred to as *hand-engineering features*. Most of the existing seizure detection methods try to identify distinguishable features between ictal (i.e., seizure) and interictal (i.e., independent of the seizure occurrence) states. These extracted features are then used to train a classifier with machine learning algorithms such as Support Vector Machine (SVM).

However, these existing methods may not perform well in practice for two main reasons. Firstly, they depend on the prior knowledge to develop algorithms to extract meaningful features from raw signals. We believe that this process would not generalize well to a larger population due to the heterogeneity among subjects and recording devices. As there is no standard and pre-defined pattern for seizures and the presentation of seizures in EEG signals vary across seizure types and subjects [107, 109, 110], these methods may only perform well for specific patterns of seizures or for specific environment settings. It would be better to build algorithms that can automatically learn meaningful features from raw signals. Secondly, most of the existing systems require patient-specific seizure data in order to train detectors [18, 110, 117]. This means that they might not be able to detect seizures in new patients, who have had few or no seizures, due to the insufficient amount of training data. Also since seizures by their nature are temporary and unpredictable, it may require a significant amount of time and effort to collect and label seizure data from each new patient. It would be better if the seizure detection methods can detect seizures by learning from normal or interictal, or unlabelled EEG data, which is much more easier to collect in practice compared to the ictal data.

One promising technique that can be used to overcome these limitations is *deep learning* [48, 49, 121]. It is capable of learning meaningful features from data by utilizing multiple linear and non-linear transformations to transform input data into more useful representations of the data. Recently, some studies also applied deep learning to extract features from raw EEG signals for EEG classification [24, 122] and for anomaly detection in EEG signal without having seen such anomalies before [123]. Some studies

proposed deep recurrent-convolutional networks to learn features from image-based representation of EEG signals [110, 124]. These works have shown that it is possible to train deep learning to automatically learn features from raw and image-based EEG data. However, none of them has investigated whether deep learning can be trained in an unsupervised setting to automatically learn features from raw EEG signals for seizure detection.

In this chapter, we propose a patient-specific seizure detection system that can automatically learn meaningful features from raw scalp EEG signals. This system is developed using stacked sparse autoencoders (SSAEs) [49], one of the deep learning approaches. The main contributions of this work are as follows:

- We demonstrate that our system can be trained to detect seizures (or ictals) using only raw, unlabeled EEG signals (especially the normal or interictal EEG) from the training data. The results showed that, without utilizing any annotations and feature extraction algorithms, our system could achieve a comparable detection performance compared to the state-of-the-art that were trained in the supervised setting.
- Whenever the seizure annotations become sufficiently available, our systen can also utilize these annotations to improve the detection performance, especially reduce the false detection rate.

The remaining contents in this chapter are organized as follows. We firstly investigate the possibility of applying SSAEs to automatically learn features from raw scalp EEG signals for seizure detection in Section 4.3. Then we extend this work to develop a patient-specific seizure detection system that can utilize raw, unlabeled EEG signals (especially the interictal EEG) to detect seizures in Section 4.4.

4.2 Related Works

Some researchers have employed Fourier transforms to extract features from EEG data [18, 111]. However, this approach has become less popular compared to wavelet transforms in the recent years. This is because it is believed that the wavelet transform can capture the non-stationary behavior in EEG signals [21], and there has also been a comparison between these two transformations, in which the wavelet transform provided better seizure detection performance compare to the Fourier transform [125, 126].

Later, the wavelet transforms have become more popular and been utilized to generate discriminating features for seizure detection [21,109,112–118]. For instance, Zandi *et al.* [117] have applied a wavelet packet (WP) transform, which is a generalization of a discrete wavelet transform (DWT), to compute a normalized index, called combined seizure index (CSI). They found that the CSI value increased significantly after the onsets of most seizures. By computing the CSI value for each EEG segment in every channel, most seizures were recognized by detecting sudden increases in the CSI values across multiple channels. Ahammad *et al.* [21] have utilized the wavelet transform and statistical measures to generate features for seizure event detection and seizure onset detection. Wavelet-based features such as entropy, SD, maximum, minimum and mean at different EEG sub-bands were used to train a linear classifier to perform seizure event detection. In seizure onset detection, interquartile range (IQR) and mean absolute deviation (MAD) were computed in addition to the wavelet-based features. These features were similarly used to train a linear classifier to detect onsets of seizures. Our work differs from many of these approaches in that our approach does not utilize any hand-engineered features.

Recently, there has been a study tried to apply deep learning to learn features from the labeled, image-based representations of EEG signals [110]. These image-based representations were generated by transforming EEG signals and their spatial information into sequences of RGB images. The three color channels of these images correspond to FFT-based power spectrum from three frequency bands (0-7, 7-14, 14-49 Hz). These image-based representations were then used as input data to the recurrent convolutional neural network. Their results showed that the patient-specific detection performed better than the cross-patient detection. Our work differs from this study as our model is trained to extract features from raw EEG signals directly without the image-based EEG transformation, and we will discuss how our model can detect seizures and other anomalies without utilizing any seizure labels (i.e., unsupervised learning) in Section 4.4.

4.3 Possibility of Stacked Sparse Autoencoders

We develop a patient-specific seizure detector that utilizes (i) stacked sparse autoencoders (SSAEs) to extract features from EEG signals; and (ii) a binary logistic classifier to distinguish between ictal and interical states based on the features from SSAEs. To efficiently train the SSAEs, we implement a two-step training algorithm. The first step is to pretrain the SSAEs using the greedy layer-wise unsupervised pretraining [127]. This is to initialize the weights of the SAAEs, which has been shown

to provide a better starting point compared to randomly initialized weights. The second step is to perform a supervised fine-tuning.

The architectural design of the SSAEs and other hyperparameter settings were the ones that gave the best seizure detection performance based on our initial experiment, which will be discussed in Section 4.3.4.

4.3.1 Patient-specific Seizure Detector

Our seizure detector receives a 1-s EEG epoch as an input and outputs the probability of such input being of the ictal class. It consists of two sparse-autoencoder layers (SSAEs) and a binary logistic classifier (see Figure 4.1). The two SSAEs can be thought of as a kind of feature extraction algorithm, which transform from a 1-s EEG epoch into a feature useful for the seizure detection. The logistic layer determines the probability of each extracted feature being of the ictal class. If the output probability is greater than a specified thresholded (e.g., 0.5), the model outputs an ictal label; otherwise, it outputs an interictal label. We train a seizure detector for each patient using his/her labeled EEG signals (i.e., **patient-specific**). This means that the architecture and the hyperparameters of the seizure detector are the same for all patients, but with different model parameters as it is trained with patient-specific EEG signals. The sequence of output classes is then post-processed before generating a seizure alarm. The next two sections will explain the algorithm we used to train patient-specific detectors (Section 4.3.2) and the post-processing technique we used to generate an alarm (Section 4.3.3).

4.3.2 Two-step Training Algorithm

The two-step training algorithm consists of *greedy layer-wise unsupervised pre-training* and *supervised fine-tuning* [73]. The first step is an unsupervised training, which pre-trains each layer of SSAEs separately to learn sparse representations of 1-s EEG epochs. The second step is a supervised fine-tuning, which trains the pre-trained SSAEs and the logistic classifier to classify whether each 1-s EEG epoch is of seizure class.

Formally, suppose there are m labeled 1-s EEG epochs from k EEG channels $\{(\mathbf{x}^{(1,1)}, y^{(1)}), \ldots, (\mathbf{x}^{(i,j)}, y^{(i)}), \ldots, (\mathbf{x}^{(m,k)}, y^{(m)})\}$; where $\mathbf{x}^{(i,j)}$ is the i-th 1-s EEG epoch from the j-th channel, $\mathbf{x}^{(i,j)} \in \mathbb{R}^n$, n is the number elements in each 1-s EEG epoch, $y^{(i)} \in \{0, 1\}$, and $y^{(i)}$ is a label of $\{\mathbf{x}^{(i,1)}, \ldots, \mathbf{x}^{(i,k)}\}$. In the first step, the SSAEs is pre-trained one layer at a time (excluding the logistic layer) on unlabeled

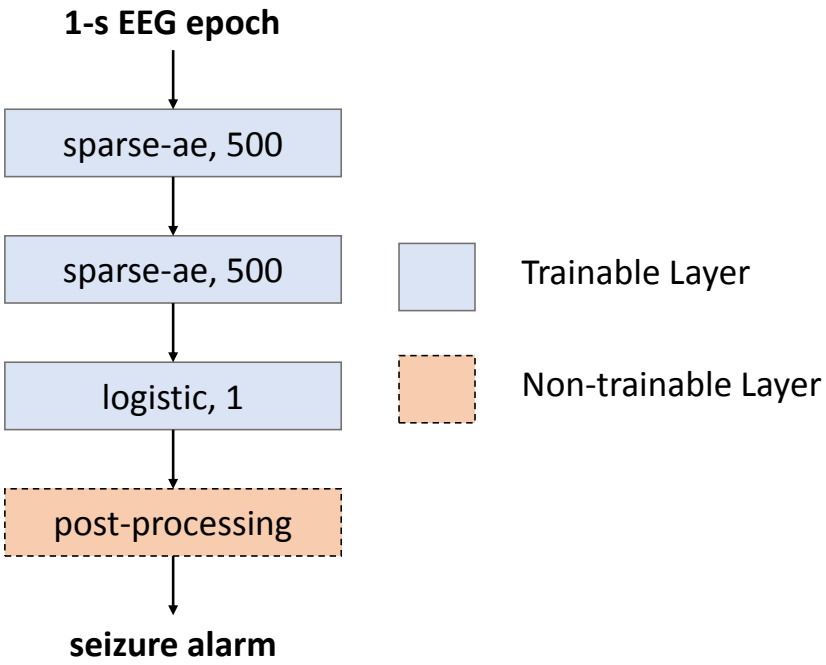


Figure 4.1: Patient-specific seizure detector. It receives a 1-s EEG epoch as an input and outputs the probability of such input being of the ictal class. It consists of two sparse-autoencoder layers (SSAEs) and a binary logistic classifier. Each **sparse-autoencoder** layer uses a fully-connected layer as an encoder and a decoder function (see more in Section 2.9.6), consisting of 500 hidden units with a sigmoid activation function. The *encoder* function is used during both the training and the detection as a kind of a feature extraction function. The *decoder* function, on the other hand, is only used during the training process and is discarded during the seizure detection. The **logistic** layer is a fully-connected layer consisting of one hidden unit with a sigmoid function that outputs the probability that the input is of the ictal class. For each input, if the output probability is greater than a specified thresholded (e.g., 0.5), the model outputs an ictal label. Otherwise, it outputs an interictal label. The output classes is then post-processed to generate a seizure alarm.

1-s EEG epochs to minimize the objective in Equation 2.80 (with the KL divergence as the sparsity penalty), while freezing parameters of other layers. Specifically, the first layer of SSAEs is pre-trained with $\{\mathbf{x}^{(1,1)}, \ldots, \mathbf{x}^{(i,j)}, \ldots, \mathbf{x}^{(m,k)}\}$. Once a layer of SSAEs has been pre-trained, it is then used to generate m activations $\{\mathbf{a}^{(1,1)}, \ldots, \mathbf{a}^{(i,j)}, \ldots, \mathbf{a}^{(m,k)}\}$, which will be used as the pre-training data for the subsequent layer of SSAEs. In the second step, the SSAEs and the logistic classifier is fine-tuned with the labeled 1-s EEG epochs to minimize the following cost function, $J_l(\theta)$:

$$J_l(\theta) = \frac{-1}{m \cdot k} \sum_{i=1}^{m} \sum_{j=1}^{k} \left[y^{(i)} \log(\sigma(f_\theta(\mathbf{x}^{(i,j)}))) + (1 - y^{(i)}) \log(1 - \sigma(f_\theta(\mathbf{x}^{(i,j)}))) \right] + \lambda \Omega(\theta) \ , \tag{4.1}$$

where the first term is a logistic regression loss and the second term is a regularization term. In the first term, the f function refers to our model consisting of the SSAEs and the logistic layer, θ is the weight and bias parameters of the model, $f_\theta(\mathbf{x}^{(i,j)})$ is the score vector from the model, σ is a logistic or

sigmoid function, and $\sigma(f_\theta(\mathbf{x}^{(i,j)}))$ is the output probability of $\mathbf{x}^{(i,j)}$ being a ictal class. In the second term, the L2 weight decay $\Omega(\theta) = \frac{1}{2}\|\mathbf{W}\|_2^2$ is used as a regularization term, $\mathbf{W}$ is the matrix of all weight parameters, and λ is a weight of the regularization term.

4.3.3 Generating Seizure Alarms

The seizure detector generates a seizure alarm when what is called **the number of seizure channels** is greater than a threshold. The number of seizure channels can be computed as following. Suppose there are m 1-s EEG epochs from k EEG channels $\{\mathbf{x}^{(1,1)}, \ldots, \mathbf{x}^{(i,j)}, \ldots, \mathbf{x}^{(m,k)}\}$. The seizure detector annotates each 1-s EEG epoch $\mathbf{x}^{(i,j)}$ with the ictal (1) or the interictal (0) class follows:

$$\hat{y}^{(i,j)} = \begin{cases} 1, & \text{if } \sigma(f_\theta(\mathbf{x}^{(i,j)})) > 0.5 \\ 0, & \text{otherwise} \end{cases}, \tag{4.2}$$

where $\hat{y}^{(i,j)}$ is the output class of $\mathbf{x}^{(i,j)}$ from the detector. Here, we label $\mathbf{x}^{(i,j)}$ as the ictal class only when its probability of being of the ictal is greater than 0.5. The output labels $\{\hat{y}^{(1,1)}, \ldots, \hat{y}^{(i,j)}, \ldots, \hat{y}^{(m,k)}\}$ are then used to compute the number of seizure channels (see Figure 4.2) as follows:

$$s^{(i)} = \sum_{j=1}^{k} \hat{y}^{(i,j)} \tag{4.3}$$

where $s^{(i)}$ is the number of seizure channels at th i-th EEG epoch, and k is the number of EEG channels. This Equation 4.3 is what we refer to as **post-processing**. A seizure alarm is generated when:

$$s^{(i)} \geq \Gamma \tag{4.4}$$

where Γ is a **channel threshold** used to generate a seizure alarm.

4.3.4 Preliminary Results

Data

Scalp electroencephalogram (EEG) data from the CHB-MIT dataset [128] were used to investigate the possiblity of SAEs for seizure detection. This dataset contains EEG recordings from pediatric patients at Boston Children's Hospital, who were suffering from intractable seizures. Recordings are

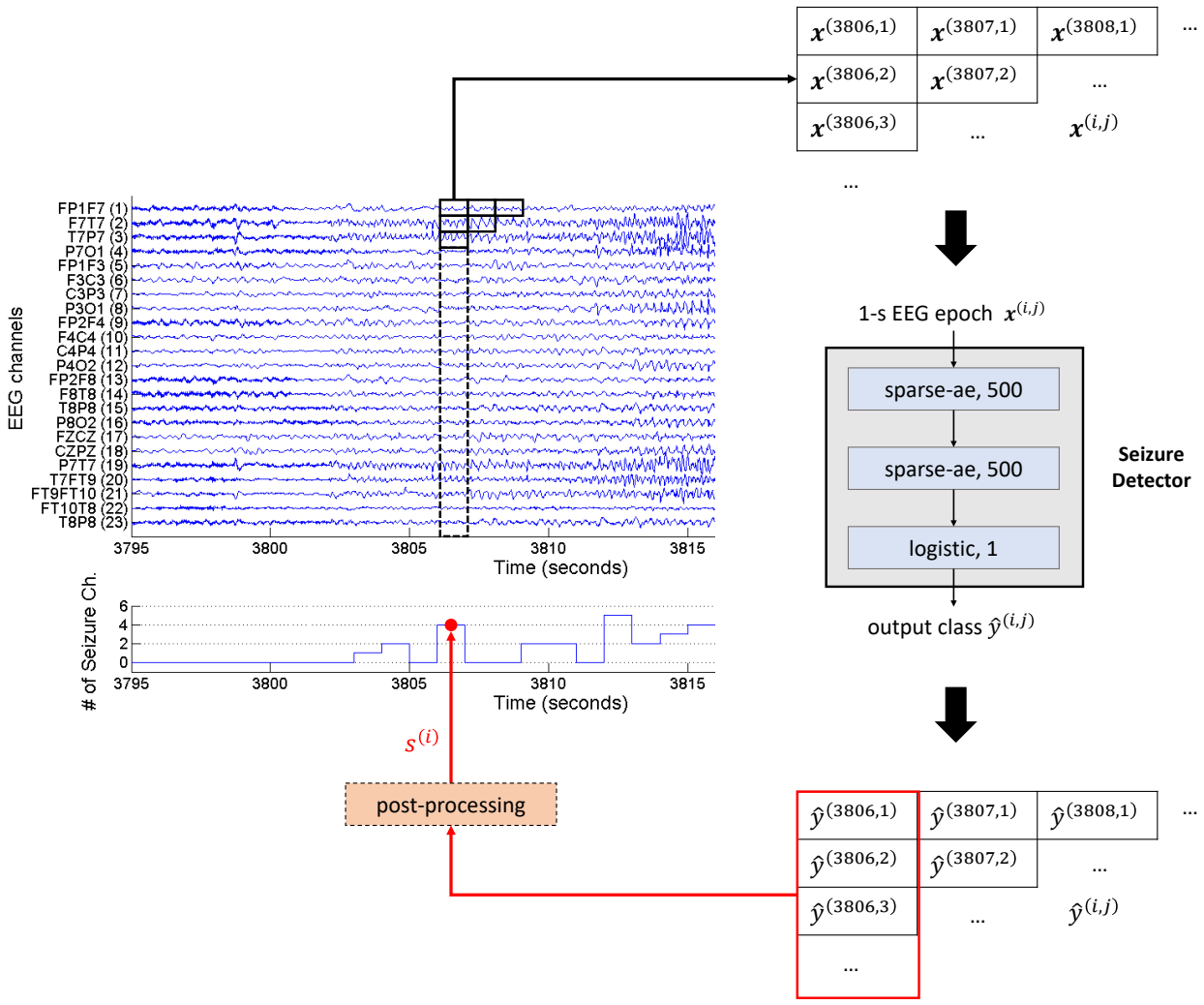


Figure 4.2: Compute the number of seizure channels for generating seizure alarms. The trained seizure detector is used to determine whether the input 1-s EEG epochs from k EEG channels $\mathbf{x}^{(i,j)}$ are of the ictal ($\hat{y}^{(i,j)} = 1$) or the interictal ($\hat{y}^{(i,j)} = 0$) class (see Equation 4.2). The number of seizure channel at the i-th EEG epoch $s^{(i)}$ is computed by summing the output labels at the i-th EEG epoch from k EEG channels (see Equation 4.3). We refer to this process as the *post-processing*. A seizure alarm is generated when $s^{(i)}$ is greater than a channel threshold Γ. In this example, we highlight the computation of the number of seizure channels at the 3806-th EEG epoch, which is 4.

grouped into 23 cases, and they were collected from 22 subjects (5 males, ages 3-22; and 17 females, ages 1.5-19). It is noted that case chb21 was obtained 1.5 years after case chb01, from the same female subject. These EEG data were recorded using the international 10-20 system of EEG electrode positions and nomenclature [1], and they were sampled at 256 Hz with 16-bit resolution. Most of these EEG data contain 23 channels, and a few of them contain 24 or 26 channels. The EEG data of each patient are stored in what are called seizure and non-seizure files. The term *seizure files* is used to refer to the files that contain one or more seizure phases, and the term *non-seizure files* for the files

Table 4.1: Summary of the randomly selected six patients from the CHB-MIT dataset. There are 39 seizure in total.

Patient	**Age**	**Gender**	$N_{seizure}$	**Total Duration (sec)**	
				Seizure Files	**Seizure Periods**
chb01	11	F	7	23925	449
chb03	14	F	7	25200	409
chb05	7	F	5	18000	563
chb08	3.5	M	5	18000	924
chb10	3	M	7	50464	454
chb20	6	F	8	20036	302

that do not contain any seizures respectively.

In this study, we randomly chose six patients to evaluate the performance of our seizure detector, and we only used seizure files as it was sufficient to investigate the performance of our detector. There were 39 seizures in total. Table 4.1 shows a summary of the number of seizures and the total duration of the seizure files and the seizure periods for each patient. Before the training, we pre-processed raw EEG data with mean normalization and scaling with standard deviation to speed up the gradient descent optimization in the training algorithm, and then segmented the data into 1-s EEG epochs.

In this dataset, at the beginning of most ictal states, a set of EEG channels start to develop different rhythmic activities, which typically contains a number of frequency components. The patterns of the rhythmic activity also appear differently in different EEG channels across patients. In this study, only EEG data from the manually selected representative channels were used to train and test the detector for each patient. In particular, Figure 4.3 and Figure 4.4 show examples of seizures from two patients: chb01 and chb03. The vertical red dash-lines indicate the onset of the seizures. chb01's seizure begins at 3801 seconds, and chb03's seizure begins at 2996 seconds. From these two figures, it can be seen that the appearances of seizure activities of these two patients are most prominent on the different channels. An example of a set of the representative channels of chb01 and chb03 can be T7-P7(3) and P7- T7(19), and FP2-F8(13) and F8-T8(14) respectively. The manually selected representative channels for different patients might not be the same depending on the appearance of the prominent seizure patterns.

Experimental Design

To estimate the detection performance, a leave-one-file-out cross-validation scheme adapted from [18] was used. Suppose there are N_s seizure files for each patient. N_s-1 seizure files are used to train the detector. The trained detector was then applied to detect seizures in the remaining seizure file that is not used in training. This process is repeated N_s times so that each seizure file is tested. Then the outputs from all folds are combined to compute the performance metrics, which will be discussed in the next section.

Performance Metrics

We evaluated the performance of our seizure detection in terms of sensitivity, specificity and latency. *Sensitivity* refers to the detection accuracy of the seizures in the test set. The test seizure is detected when there is a seizure alarm during the seizure period. *Specificity* refers to the number of false seizure alarms per hour. *Latency* refers to the delay between the actual seizure onset annotated in the dataset and the seizure alarm. Note that not all performance metrics mentioned in Section 2.8.1 were be used to evaluate our system. As we were only interested in the onset of the seizures, there was no true and false negative to compute some metrics.

Initial Experiments for Parameter Selection

The training and test data from the first three folds of the cross-validation were used to design the architecture and parameters of our seizure detector, and the training parameters. The following is the setting that gave us the best performance.

Model Parameters. As the sampling rate of EEG is 256 Hz, each 1-s EEG epoch from one representative channel contained 256 data points. The input layer of the SSAEs of the seizure detector was set to 256. The number of hidden nodes of the two sparse-autoencoder layers was set to be larger than the input size, which were 500. Thus the seizure detector for each patient had the 256-500-500-1 architecture (see Figure 4.1).

Training Parameters. The mini-batch gradient descent was used in the two-step training to train the seizure detector. In the first step of the training, the mini-batch size was set to 10. The learning rate was set to 0.01 for the first sparse-autoencoder layer, and 0.05 for the second layer. β and ρ in

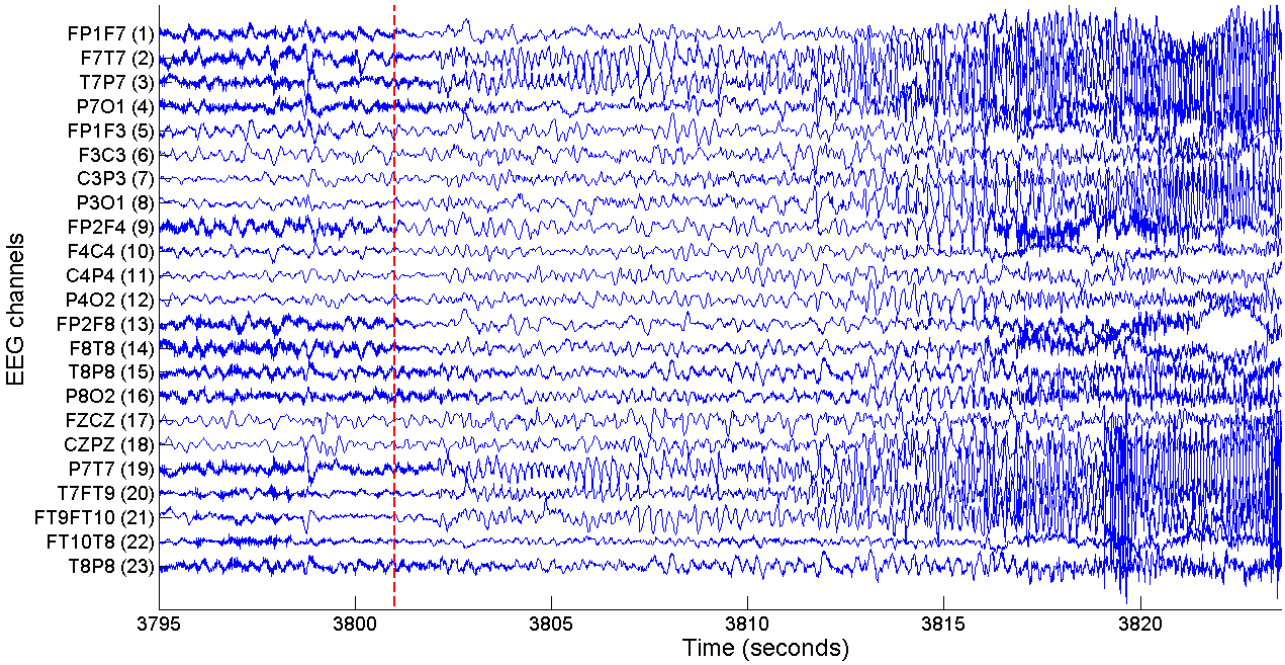


Figure 4.3: An example of a seizure pattern in scalp EEG signals from 23 channels from chb01. The red dash-line indicates the seizure onset at 3801 seconds.

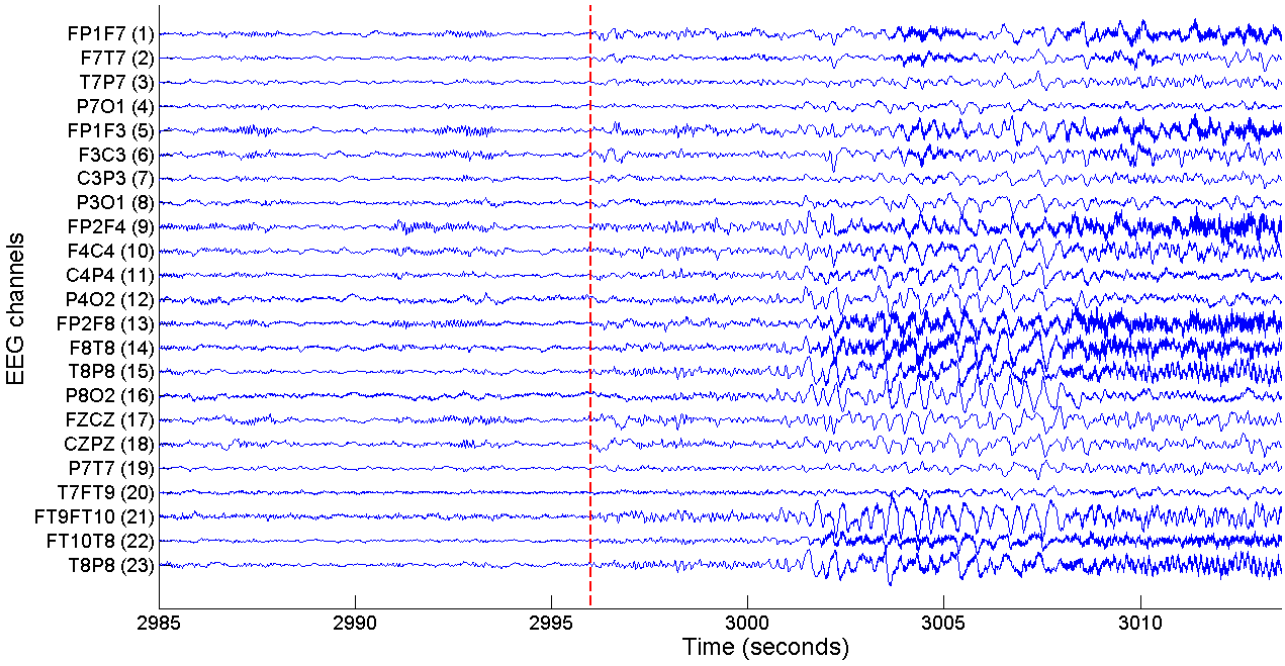


Figure 4.4: An example of a seizure pattern in scalp EEG signals from 23 channels from chb03. The red dash-line indicates the seizure onset at 2996 seconds.

Equation 2.80 were set to 3 and 0.1 respectively when training each sparse-autoencoder layer. In the second step of the training, the mini-batch size was set to 30. The learning rate was initially set to 0.05. Then it was gradually decreased to 0.01 from the 20^{th} epoch to 30^{th} epoch. The weight of the regularization λ was set to 0.003. In both training steps, the training set was randomly shuffled at the beginning of each training epoch.

Implementation

The seizure detector was implemented using MATLAB 2014a and evaluated on a computer equipped with the Intel(R) Core(TM) i7-2600 CPU 3.40 GHz and the RAM of 16.0 GB. The training time varied depending on the amount of training examples, which ranged from 2 to 5 hours. The run time of the detector was about 10 ms for each 1-s EEG epoch.

Seizure Detection Performance

Figure 4.5 shows an example of the scalp EEG signals from chb01 (top), and the number of seizure channels (bottom). In this example, the number of the representative channels was five and their channel indices were 2, 3, 19, 20 and 21. If the channel threshold is set to 1 (i.e., $\Gamma = 1$ in Equation 4.3), our detector was able to detect this seizure onset with 2-s delay (i.e., at 3803 second). If the channel threshold is set to 2, the seizure onset was detected with 3-s delay (i.e., at 3804).

In our experiment, three channel thresholds: 1, 3 and 5, were used to evaluate the performance of our seizure detector. Table 4.2 shows the summary of the sensitivity and latency evaluated by using these three channel thresholds, and Figure 4.6 shows the number of false detections per hour (i.e., specificity) for the three channel thresholds across six patients. Our detector was able to detect all of 39 test seizures with a mean latency of 2.78 seconds when the channel threshold was set to 1, but the false positive rates were considerable high. When the channel threshold was increased to 5, our detector was able to detect most of the seizures with a high latency of 18.27 seconds and a low false positive rate.

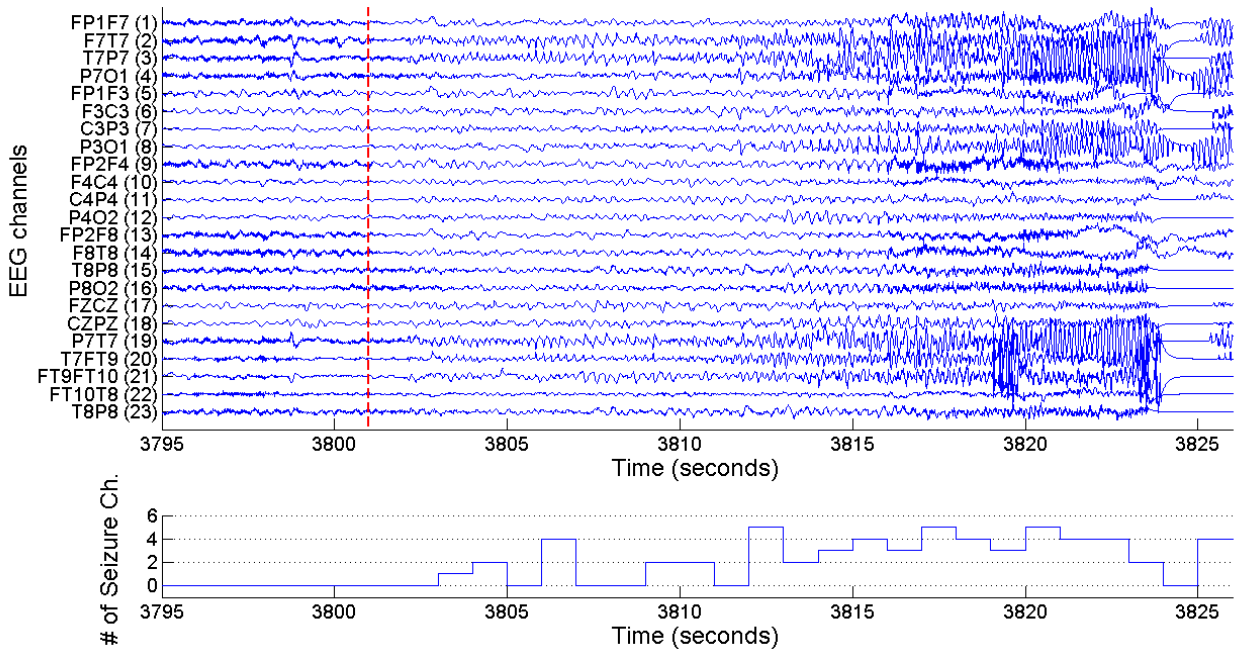


Figure 4.5: An example of the scalp EEG signals of chb01 (top) associated with the number of seizure channels (bottom) determined by our seizure detector. The red dash line indicates the seizure onset. The number of the representative channels is five and their channel indices are 2, 3, 19, 20 and 21.

4.3.5 Discussion

The preliminary results demonstrated that our seizure detector were able to detect most of the seizures with a low false detection rate and a reasonable detection latency. When a high value of channel threshold was used, our detector missed some seizures and became slower in declaring the ictal onset. This is as expected because of two main reasons. Firstly, most of the prominent seizure activities in the scalp EEG signals start to appear long after the seizure onset annotated by the experts (see Figure 4.5). Secondly, our detector will generate a seizure alarm only when it observes seizure activity from most of the representative channel (i.e., when it is very confident). Even though the higher threshold makes our detector less sensitive to the noise in the EEG signals that has a similar pattern to the seizure activity than the lower threshold, some of the seizures, however, might not be detected or be detected with a high latency. The channel threshold can be used to specify *tradeoff* between sensitivity, specificity and latency.

Even though the preliminary results show the promising seizure detection performance, however, our approach still relies on the seizure labels or annotations available in the training data (i.e., supervised training). These annotations might not be available or limited in practice as it is difficult, labor-

Table 4.2: Summary of the sensitivity and latency of our seizure detector when we varied the channel threshold (i.e., Γ in Equation 4.3). The total number of seizures are 39.

Channel Threshold	Sensitivity (%)	Mean Detection Latency (sec)
1	100	2.78
3	100	7.86
5	88.09	18.27

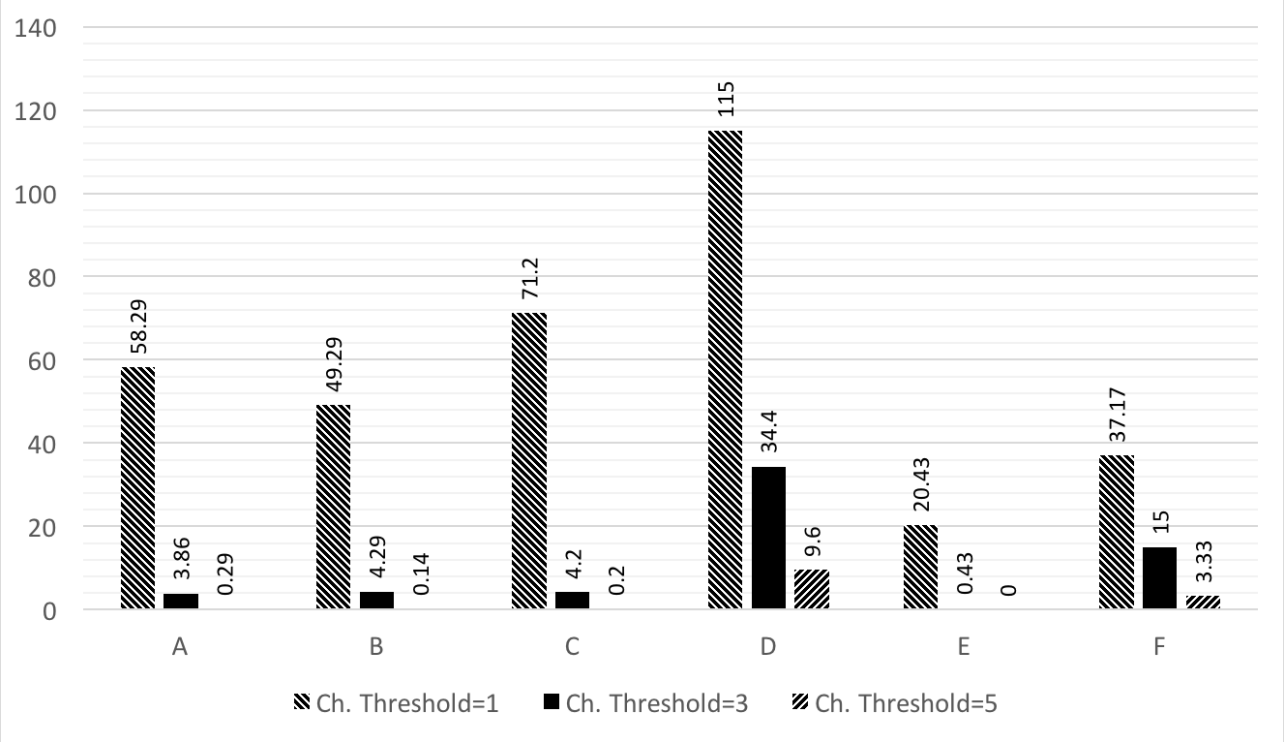


Figure 4.6: False detection rates per hour (i.e., specificity) for three channel thresholds across six patients.

intensive and time-consuming to collect and annotate EEG signals during seizures. It is, therefore, better to develop a seizure detector that can generate seizure alarms by utilizing unlabeled EEG signals. One potential solution is to extend our seizure detector to learn to detect seizures and other abnormal EEG patterns by utilizing only unlabeled EEG signals (i.e., unsupervised learning). We will also remove the process of selecting the representative channels as this process relies on the seizure annotations. In the next section, we will discuss the extended version of our seizure detector for unsupervised seizure detection.

4.4 Patient-Specific Seizure Detection System

We develop a patient-specific seizure detection system based on SSAEs that can utilize a plenty amount of raw, unlabeled EEG signals to detect seizures and other abnormal EEG patterns. Whenever the seizure annotations become sufficiently available, our system can also utilize this information to improve the detection performance. This system is an extension of SSAEs from the previous section, which consists of two main modules: **anomaly detector** and **anomaly classifier** (see Figure 4.7). The anomaly detector determines which period of EEG signals has an abnormal pattern (including seizures), and it can function without utilizing any annotations or labels from the training data (i.e., unsupervised learning). The anomaly classifier determines types of EEG patterns, and it can function when annotations or labels are available in the training data (i.e., supervised learning). Therefore, when there are insufficient amount of seizure labels or no seizure label, our system can use the anomaly detector to detect abnormal EEG patterns and generate alarms. When the seizure and the other anomaly (if any) labels are available in the training data, the system can use these labels to train the anomaly classifier to determine the class of abnormal EEG patterns. The trained anomaly classifier can be used to determine the types of anomalies (e.g., seizures) in addition to the anomaly alarms.

4.4.1 Anomaly Detector

Our anomaly detector is designed to detect anomalies in EEG signals without having encountered them before. Typically abnormal patterns in EEG signals such as seizures are rare and transient, and it may be difficult and time-consuming in practice to collect *all* abnormal patterns from each new patient. It is better therefore for the detectors to detect abnormal patterns by learning general (i.e., normal) EEG patterns from a large amount of unlabeled EEG signals.

We hypothesize that SSAEs trained to reconstruct EEG signals would learn to reconstruct normal EEG signals (such as background and interictal) well, which are prevalent in the training data. This is because the SSAEs should see few or no anomalies in the input data, and would therefore be unlikely to learn to reconstruct them. Thus the periods of EEG signals that have reconstruction errors higher than a specified threshold would be treated as abnormal EEG patterns.

We implement the anomaly detector using SSAEs (see Figure 4.7). The number of nodes in the input layer is equal to the number of data points in each 1-s EEG epoch, and the number of nodes in

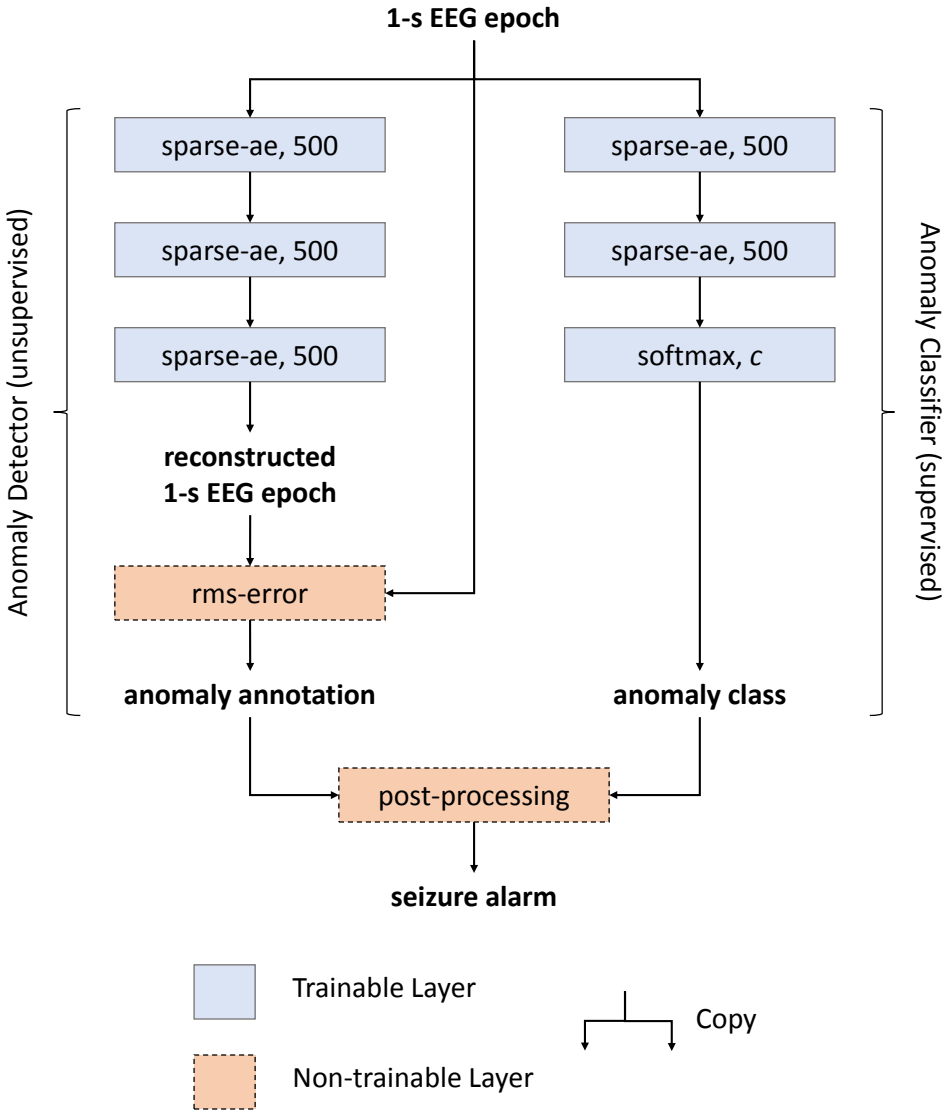


Figure 4.7: Patient-specific seizure detection system, consisting of two main components: anomaly detector and anomaly classifier. The **anomaly detector** detects anomalies in EEG signals when patient-specific seizure data are not available. It consists of three sparse autoencoders trained to reconstruct its input 1-s EEG epoch. It annotates the input 1-s EEG epoch as anomalous when the difference between the input and the reconstructed is more than a threshold. Such difference is measured by the root-mean-squared error (*rms-error*). The outputs from the anomaly detector are anomaly annotations. The **anomaly classifier** determines anomaly classes or types of EEG signals. This anomaly classifier is an extended version of our previous seizure detection in Section 4.3.1 to support multi-class classification. Specifically, we replace the logistic classifier with the *softmax* classifier. The number of hidden units of the softmax layer, c, depends on the number of classes we would like to classify. The outputs from the anomaly classifier are anomaly classes. **Unsupervised setting:** when the seizure labels are not available, only the anomaly detector is functional. So only the sequence of anomaly annotations is post-processed to generate a seizure alarm. **Supervised setting:** when the seizure labels (and other class labels) are available, these labeled data can be used to train the anomaly classifier. Thus *both* the anomaly detector and classifier are functional. The sequence of anomaly annotations and anomaly classes are post-processed to generate a seizure alarm.

the output layer is the same as the input layer. The number of *sparse-autoencoder* layers and the number of nodes in each layer are chosen such that the SSAEs can extract useful features for the reconstruction. The architecture of the SSAEs is designed based on our initial experiment with the data from one patient in the CHB-MIT dataset. We found that three sparse-autoencoder layers with 500 hidden nodes each gave us the best results.

Training

We use a two-step training consisting of pre-training and fine-tuning to train the anomaly detector for each patient. The pre-training step is similar to the one mentioned in Section 4.3.2. The fine-tuning step is different as the SSAEs is fine-tuned to reconstruct the input 1-s EEG epochs (i.e., regression problem). Specifically, it is trained to minimize the following cost function, $J_r(\theta)$:

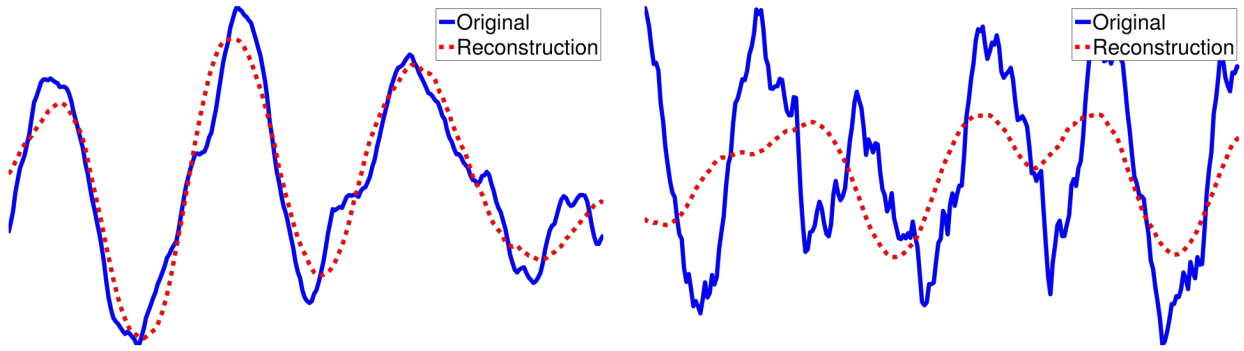
$$J_r(\theta) = \frac{1}{m \cdot k} \sum_{i=1}^{m} \sum_{j=1}^{k} (g_\theta(\mathbf{x}^{(i,j)}) - \mathbf{x}^{(i,j)})^2 + \lambda \Omega(\theta) \ , \tag{4.5}$$

where the first term is the mean-square reconstruction loss and the second term is a regularization term. In the first term, the g function refers to the SSAEs in the anomaly detector, θ is the parameters of the SSAEs, $\mathbf{x}^{(i,j)}$ is the i-th 1-s EEG epoch from the j-th channel, and $g_\theta(\mathbf{x}^{(i,j)})$ is the score vector or the reconstructed 1-s EEG epoch of the input $\mathbf{x}^{(i,j)}$. In the second term, we use the same L2 weight decay mentioned in Section 4.3.2.

Figure 4.8 shows two reconstruction examples of the interictal (Figure 4.8a) and the ictal (Figure 4.8b) 1-s EEG epochs from our initial experiment. It can be seen that the reconstruction of the interictal epoch is similar to the original epoch, while the the reconstruction of the ictal epoch is different from the original epoch. The following section will discuss how our anomaly detector computes the error between the input and its reconstruction, which will be used to determine whether each 1-s EEG epoch is of the normal or abnormal classes.

Detection

The anomaly detector decides whether each 1-s EEG epoch from a single channel is an anomaly by computing its reconstruction error using root mean square (RMS) error, and comparing it with a threshold. This step refers to the *rms-error* layer in Figure 4.7. Formally, suppose there are m 1-s EEG



(a) Interictal (or normal) **(b)** Ictal (or abnormal)

Figure 4.8: Examples of the original (blue) and its reconstructed (red) 1-s EEG epochs from the SSAEs in the anomaly detector. The reconstruction of the interictal epoch (a) is similar to the original epoch. In contrast, the the reconstruction of the ictal epoch (b) is different from the original epoch.

epochs from k EEG channels $\{\mathbf{x}^{(1,1)}, \ldots, \mathbf{x}^{(i,j)}, \ldots, \mathbf{x}^{(m,k)}\}$. The anomaly detector first reconstructs each of the input epoch $g_\theta(\mathbf{x}^{(i,j)}) = \hat{\mathbf{x}}^{(i,j)}$. Then it computes RMS error and compare with a threshold ξ. Each $\mathbf{x}^{(i,j)}$ is annotated with the abnormal (1) or the normal (0) class as follows:

$$\phi^{(i,j)} = \begin{cases} 1, & \text{if } \sqrt{\frac{1}{n}\sum_{p=1}^{m}(\hat{x}_p^{(i,j)} - x_p^{(i,j)})} > \xi \\ 0, & \text{otherwise} \end{cases}, \tag{4.6}$$

where $\phi^{(i,j)}$ is the output class of $\mathbf{x}^{(i,j)}$, $x_p^{(i,j)}$ is the p-th data point of the $\mathbf{x}^{(i,j)}$, n is the number of data points in each 1-s EEG epoch, and ξ is a positive threshold used to differentiate abnormal EEG data from the normal ones. The threshold ξ is chosen based on our initial experiment with the data from one patient in the CHB-MIT dataset. We found that 0.1 gave us the best results.

4.4.2 Anomaly Classifier

Our anomaly classifier is designed to determine types of EEG signals. The number of EEG types the classifier can determine depends on the availability of the annotations or labels provided by experts or users in the training data (i.e., supervised training).

We hypothesize that SSAEs can learn to extract meaningful features from different types of EEG signals when there are *sufficient* amount of labeled training data. This is because a considerable amount of training data is required to train deep neural networks to learn useful representations from raw data [49, 129].

We implement the anomaly classifier using SSAEs and a softmax classifier (see Figure 4.7). The

softmax classifier is a generalization of the logistic classifier for multi-class classification. The number of nodes in the input layer is equal to the number of data points in each 1-s EEG epoch. The number of *sparse-autoencoder* layers and the number of nodes in each layer are chosen such that the SSAEs can extract useful features for the classification. The number of nodes in the *softmax* layer depends on the number of EEG types available in the training data. The architecture of the SSAEs is designed based on the same initial experiment as the anomaly detection (see Section 4.4.1). We found that two sparse-autoencoder layers with 500 hidden nodes each gave us the best results.

Training

We use the same two-step training algorithm mentioned in Section. 4.3.2 to train the anomaly classifier for each patient. The only difference is the cost function as the anomaly classifier uses softmax function for multi-class classification. Specifically, the anomaly classifier is trained to minimize the following cost function, $J_s(\theta)$:

$$J_s(\theta) = \frac{-1}{m \cdot k} \sum_{i=1}^{m} \sum_{j=1}^{k} \log \left(\frac{e^{h_\theta(\mathbf{x}^{(i,j)})_{y_i}}}{\sum_{q=1}^{c} e^{h_\theta(\mathbf{x}^{(i,j)})_q}} \right) + \lambda \Omega(\theta) \ , \tag{4.7}$$

where the first term is a softmax loss and the second term is a regularization term. In the first term, the h function refers to the anomaly classifier, θ is the parameters of the SSAEs and the softmax layer in the anomaly classifier, $h_\theta(\mathbf{x}^{(i,j)})$ is the score vector of the input $\mathbf{x}^{(i,j)}$, $h_\theta(\mathbf{x}^{(i)})_q$ is the score for the q-th class index, y_i is the ground truth or the correct class index, and c is the number of output classes. In the second term, we use the L2 weight decay mentioned in Section 4.3.2.

Classification

The anomaly classifier determines the probability of each class (or type) for each 1-s EEG epoch from a single channel and outputs the class with the highest probability. Formally, the anomaly classifier first computes the score vector for each class index $h_\theta(\mathbf{x}^{(i,j)})$. Then it outputs the class index that has the maximum score value. Each $\mathbf{x}^{(i,j)}$ is annotated with the class index as follows:

$$\hat{y}^{(i,j)} = \arg\max_q h_\theta(\mathbf{x}^{(i,j)})_q \ . \tag{4.8}$$

where $\hat{y}^{(i,j)}$ is the output class index of $\mathbf{x}^{(i,j)}$, and q is the index of the score vector.

4.4.3 Generating Seizure Alarms

Our seizure detection system utilizes information from the anomaly detector and the anomaly classifier to generate seizure alarms. This step refers to the **post-processing** layer in Figure 4.7.

- **Unsupervised setting.** When the seizure annotations or labels are limited or not available in the training data, our system uses anomaly annotations from the anomaly detector to generate seizure alarms
- **Supervised setting.** When the seizure annotations become sufficiently available, our system uses both anomaly annotations and classes to reduce the number of false seizure alarms

Unsupervised Setting

Our system utilizes anomaly annotations from the anomaly detector to compute what is called *anomaly channel*. The anomaly channel is the number of channels of 1-s EEG epoch that the anomaly detector annotates as abnormal. Formally, the anomaly channel of the i-th 1-s EEG epoch, $a^{(i)}$, is defined as follows:

$$a^{(i)} = \sum_{j=1}^{k} \phi^{(i,j)} \ , \tag{4.9}$$

where $\phi^{(i,j)}$ is an abnormal annotation computed from Equation 4.6, and k is the number of EEG channels.

In our initial experiment, we found that the anomaly channel gradually or suddenly increased during the seizure's onset. For instance, Figure 4.9 shows an example of EEGs and the anomaly channel. It can be seen that the anomaly channel increases from around 10 to more than 20 after the seizure's onset at 2382 second.

In order to detect an increase in the anomaly channel, our system computes *CUSUM* values from the sequences of the anomaly channels. Formally, suppose there are m anomaly channels $\{a^{(1)}, \ldots, a^{(m)}\}$. The recursive one-side CUSUM algorithm [130, 131] is used to calculate the CUSUM at the i-th 1-s EEG epoch as follows:

$$U_a^{(i)} = \max\{0, (a^{(i)} - \gamma_a) + U_a^{(i-1)}\} \ , \tag{4.10}$$

where $U_a^{(0)} = 0$, and γ_a is a positive threshold to detect a significant increase in $a^{(i)}$. When $U_a^{(i)} > \Gamma_a$, a transient period of an increase in the anomaly channel is detected. Our system generates a seizure

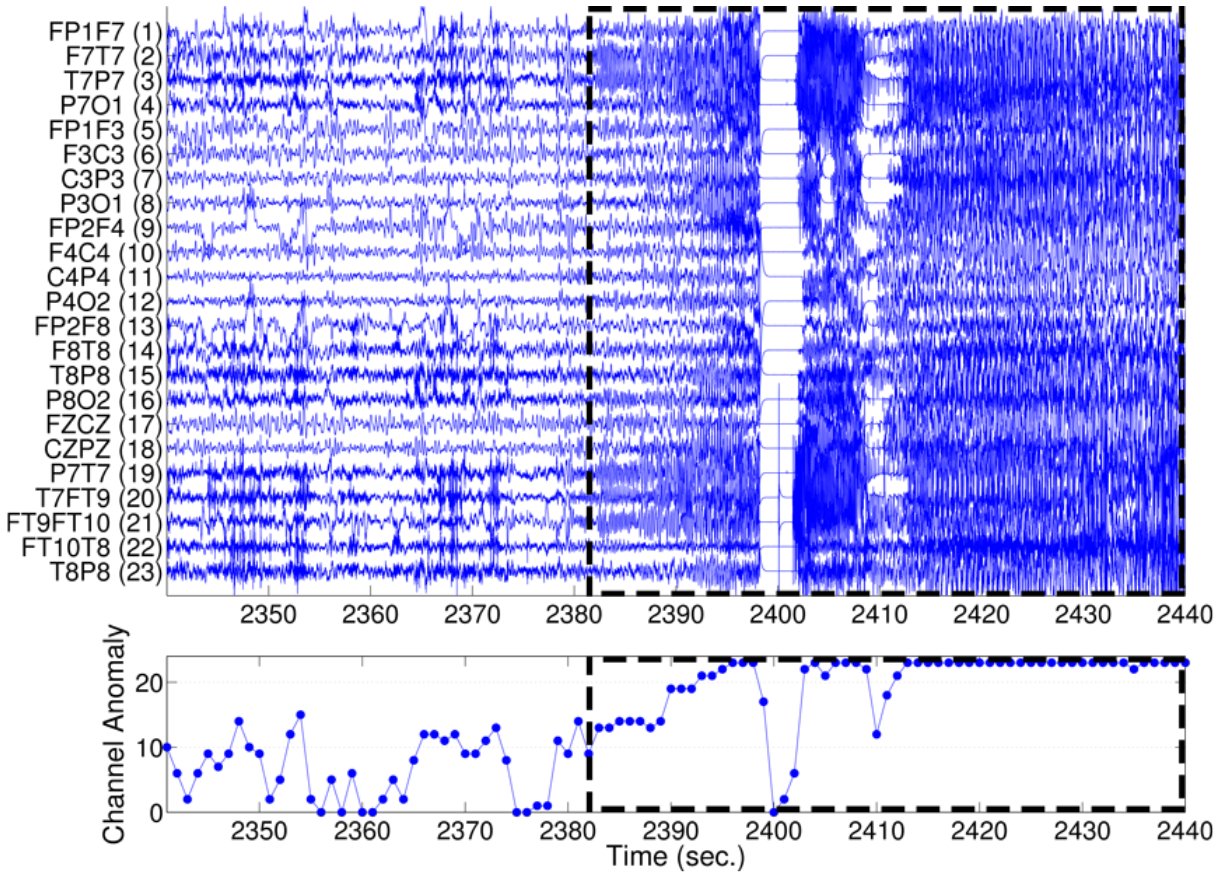


Figure 4.9: Examples of the anomaly channel computed from 1-s EEG epochs during a seizure (start at 2382 s). The top graphs show raw signals from 23 EEG channels. The bottom graphs show the value of anomaly channel computed using Equation 4.9. The black dash line indicates the seizure period.

alarm and reset the $U^{(i+1)}$ to zero to quickly detect any subsequent increases. Γ_a is another positive threshold used to detect sudden increases in the anomaly channel to generate seizure alarms. In Figure 4.9, if γ_a, and Γ_a are set to 10 and 15 respectively, a seizure alarm will be generated at 2386 seconds (i.e., 4 seconds delay). γ_a and Γ_a are the hyperparameters of the seizure detection system, and can be different among patients depending on the anomaly patterns during seizures.

The results from our initial experiment also show that if the period of a seizure is long, our system will generate multiple seizure alarms for one seizure as the value of anomaly channel still continues to be high. We also observes the same behavior from other abnormal EEG patterns that are not seizures. To prevent these repetitive alarms, we introduce what is called, *alarm delay* (T_d), to reduce the number of repetitive alarms. With this delay, our system will not generate additional seizure alarms until the end of the alarm delay.

Supervised Setting

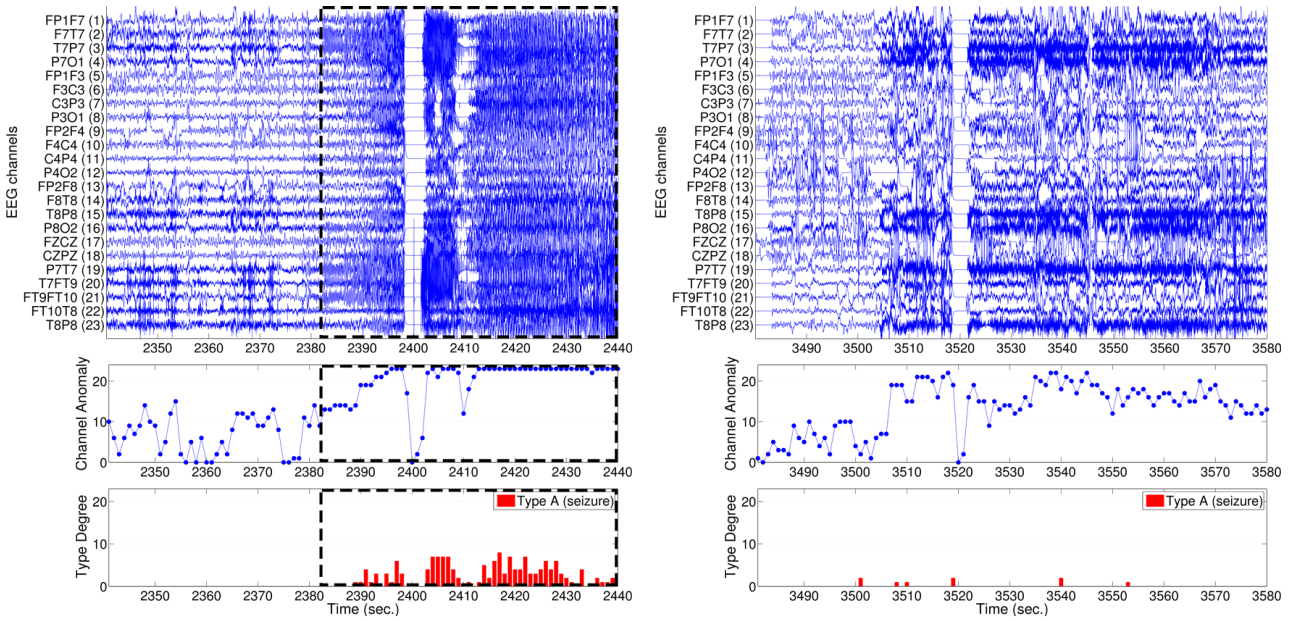
When the seizure annotations become available in the training data, our system can utilize these annotations to train the anomaly classifier to determine types or classes of EEG signals such as interictal, ictal and other artifacts. This information about the anomaly class can help reduce the number of false seizure alarms. Our system utilizes anomaly classes to compute what is called *seizure channel* in addition to the anomaly channel. The seizure channel is the number of channels of 1-s EEG epoch that the anomaly classifier annotates as seizure or ictal. Whenever our system observes an increase in either the anomaly channel or the seizure channel, it further waits for a period of T_w seconds to see whether there is an increase in the other one. If so, it generate a seizure alarm. Formally, The seizure channel of the i-th 1-s EEG epoch, $s^{(i)}$, is defined as follows:

$$s^{(i)} = \sum_{j=1}^{k} \mathbf{1}\left\{(\hat{y}^{(i,j)}) == c_{seizure}\right\} \quad , \tag{4.11}$$

where $\mathbf{1}\{\cdot\}$ is an indicator function that returns 1 when $\mathbf{1}\{\text{true}\}$ and 0 otherwise, $\hat{y}^{(i,j)}$ is the output class index from the anomaly classifier (see Equation 4.8), and $c_{seizure}$ is the class index of the seizure label.

Similar to the detection of an increase in the anomaly channel, our system computes CUSUM values from the sequence of the seizure channels $\{s^{(1)}, \ldots, s^{(m)}\}$. By replacing $a^{(i)}$ and γ_a with $s^{(i)}$ and γ_s in Equation 4.10 respectively, U_s^i can be calculated. When $U_s^{(i)} > \Gamma_s$, a transient period of an increase in the seizure channel is detected, and the $U_s^{(i+1)}$ is reset to 0. γ_s and Γ_s are also the hyperparameters of the seizure detection system, and can be different among patients depending on the anomaly patterns during seizures.

To illustrate how this supervised seizure detection works, we compute the anomaly channel and the seizure channel from EEG signals during ictal (start at 2382 s) and interictal periods as shown in Figure 4.10a and Figure 4.10b respectively. The top graphs show raw EEG signals. The middle graphs show the value of anomaly channel. The bottom graphs show the type degree indicating the number of channels annotated with different EEG types such as seizures. The type degree of the seizure is the same the seizure channel mentioned in Equation 4.11. In Figure 4.10a, it can be seen that following the onset of the seizure at 2382 s, both the anomaly channel and the seizure channel increase. When parameters ξ, γ_a, Γ_a, T_d, γ_s, Γ_s, and T_w are set to, 0.1, 10, 15, 1, 1, 3, and 5 respectively, our system generates a seizure alarm at 2393 seconds (i.e., have a detection latency of



(a) Ictal EEGs (start at 2382 s) **(b)** Interictal EEGs

Figure 4.10: Examples of the anomaly channel and the seizure channel computed from EEGs during ictal (a) and interictal (b) periods. The top graphs show raw EEGs from multiple channels. The middle graphs show the value of anomaly channel (see Equation 4.9). The bottom graphs show the type degrees indicating the number of channels annotated with non-interictal types such as seizures. The type degrees of the seizures is the same the seizure channel mentioned in Equation 4.11.

11 s). In Figure 4.10b, it can be seen that only the anomaly channel increases. Our system, therefore, will not generate a seizure alarm for these EEG signals, which can help reduce the number of false alarms.

4.4.4 Preliminary Results

Dataset

Scalp EEG data from the CHB-MIT dataset were used to evaluate our seizure detection system. The details of this dataset can be found in Section. 4.3.4. We randomly chose nine[1] out of 22 patients to evaluate the performance of our seizure detection system. There were 67 seizure events in total. In this study, we used all EEG channels in the dataset which was different from our previous study (Section 4.3) that used only the representative channels. For each channel, EEG data were segmented into 1-s EEG epochs, containing 256 data point in each segment (i.e., sampling rate is 256 Hz), before being used to train and evaluate our system. Table 4.3 shows a summary of the number of seizures and the total duration of the seizure files and the seizure periods for each patient.

[1]We will explain later in Section 4.4.5 why we did not use all of the 22 patients to evaluate our system.

Experimental Design

The evaluation was carried out under two scenarios: unsupervised and supervised seizure detection. In the unsupervised detection, only normal or interictal EEG data from randomly-selected non-seizure files were used to train the anomaly detector. In our initial experiment, we found that at least two hours of randomly selected interictal EEG data were required to train the detector. The trained anomaly detector for each patient was used to detect seizures in the seizure files. In the supervised detection, we applied the leave-one-file out cross-validation, which is similar to Section 4.3.4 except that we also included the non-seizure files into the training data. In particular, suppose there are N_s seizure files and N_{ns} non-seizure files from each patient. $N_s - 1$ seizure files were used to train the anomaly classifier, and four randomly-selected non-seizure files (i.e., 2-h EEG data) were used to train the anomaly detector. The system post-processes the information from both the trained anomaly detector and classifier to detect seizure in the withheld seizure file. This process was repeated for N_s times so that each seizure file was tested.

Implementation

Each component of our system was implemented using MATLAB 2014a and Python 2.7. We ran the experiments on a computer equipped with the Intel(R) Core(TM) i7-2600 CPU 3.40 GHz and the RAM of 16.0 GB. The time for training and detection will be discussed later for different scenarios.

Performance Metrics

The performance metrics are sensitivity, specificity and latency, which are the same as the ones in Section 4.3.4.

Initial Experiment

We chose the architectures and the model parameters of the SSAEs for the anomaly detector and classifier via cross-validation over two patients: chb01 and chb10. For the anomaly detector, we found that three *sparse-autoencoder* layers with 500 nodes each could reconstruct normal 1-s EEG epochs with low RMS error. We also found that at least two hours of normal EEG data were required to train

Table 4.3: Summary of the randomly selected nine patients from the CHB-MIT dataset. There are 67 seizures in total.

Patient	Age	Gender	$N_{seizure}$	Total Duration (sec)	
				Seizure Files	Seizure Periods
chb01	11	F	7	23925	449
chb03	14	F	7	25200	409
chb05	7	F	5	18000	563
chb08	3.5	M	5	18000	924
chb10	3	M	7	50464	454
chb14	9	F	8	25200	177
chb20	6	F	8	20036	302
chb21	13	F	4	13790	203
chb24	N/A	N/A	16	43200	539

an anomaly detector for each patient. For the anomaly classifier, we found that two *sparse-autoencoder* layers before the *softmax* layer could distinguish between interictal and ictal 1-s EEG epochs.

Unsupervised Seizure Detection Performance

Training parameters. EEG data were first pre-processed using mean normalization and scaling with standard deviation such that the data had zero mean and unit variance. After the pre-processing, these data were used to train the detector with the two-step training. In the first step, β and ρ for the KL divergence penalty were set to 3 and 0.1 respectively. In the second step, the weight of the regularization λ in Equation 4.7 was set to 0.003. In both training steps, the mini-batch gradient descent with a batch size of 10 and the learning rate of 0.05 was used. The number of epochs for the first and the second steps were 10 and 20 respectively.

After training, the seizure detection system for each patient was used to detect seizures in the remaining seizure file of that patient. The training time was ∼3.5 h, and the run time of the system for each 1-s EEG epoch from each channel was ∼11 ms.

Results. Table 4.4 shows a summary of the performance of the unsupervised seizure detection when parameters of the system ξ, γ_a, Γ_a and T_d were set to 0.1, 15, 15 and 30 respectively. Our system achieved the sensitivity of 82.09% (i.e., detected 55 of 67 seizures) with the false positive rates of 3.45 alarms per hour and the mean detection latency of 10.38 seconds. Our anomaly detector could detect

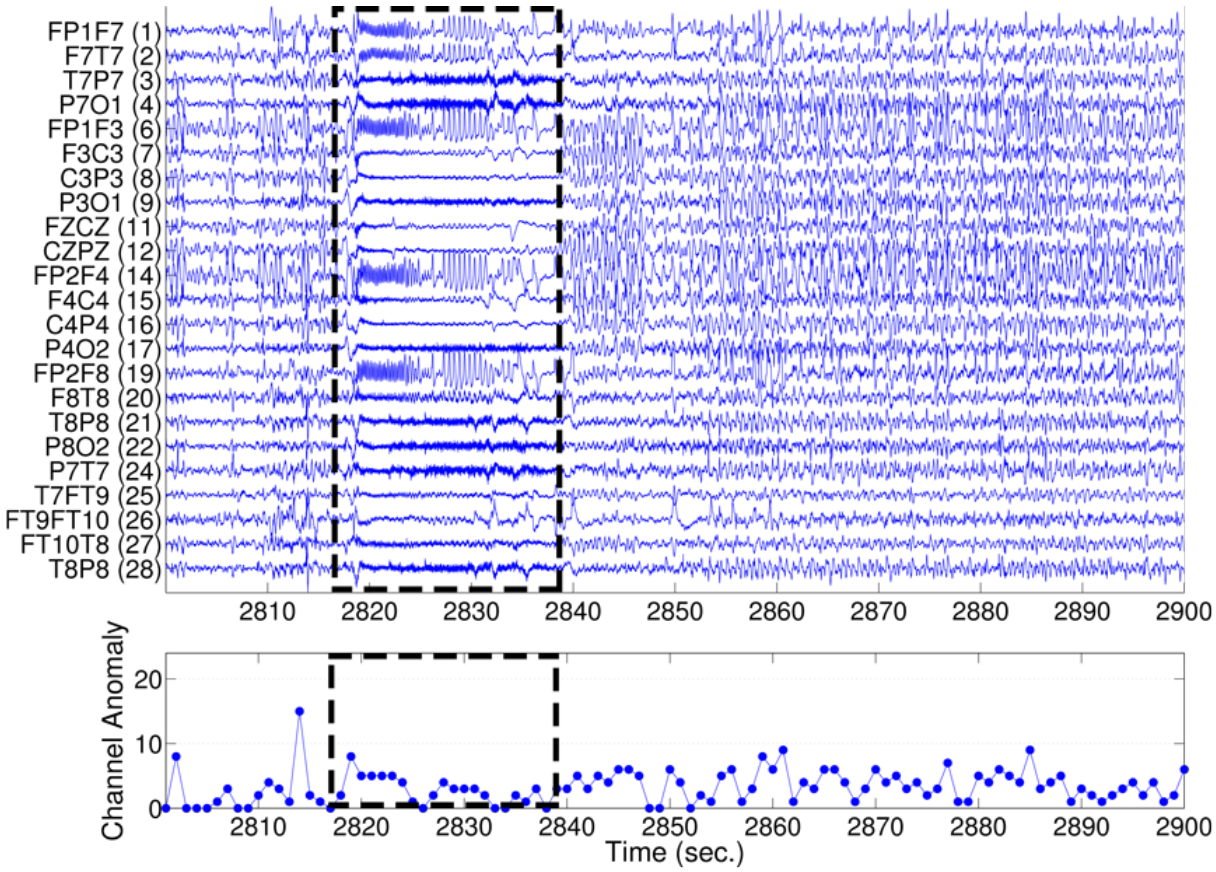


Figure 4.11: An example of EEG data from Patient F during a seizure (between 2817 s and 2839 s), and the corresponding anomaly channel determined by the seizure detector.

most of the seizures except the ones from chb14 and chb21. We found that the amplitudes of the EEG data during these seizures were relatively low (see Figure 4.11). Thus the RMS errors during this seizures were low even though the reconstructed EEG data were not similar to these seizures. This resulted in a low value of anomaly channel during such seizures.

It can also be seen from Table 4.4 that the false detection rate was high in some patients. This was because there were many abnormal EEG patterns that were not seizures and they were not labeled or annotated (see Figure 4.10b). These results were as we expected because our system should generate alarms not only for seizures, but any abnormal EEG patterns. The mean detection latency (or delay) was also high. This was because the synchronized abnormal patterns started to appear prominently after the seizures had been occurring for a while (see Figure 4.10a).

To demonstrate the effect of changing hyperparameters, we plot receiver operating characteristic (ROC) curves for different hyperparameters: γ_a, Γ_a, ξ and T_d. We change one parameters at a time while keeping the other unchanged. Figure 4.12 shows changes in sensitivity versus false detection

Table 4.4: Performance of our seizure detection system in the unsupervised setting when ξ=0.1, γ_a=15, Γ_a=15 and T_d=30

Patient	$N_{seizure}$	N_{detect}	**Sensitivity (%)**	**FDR (per hour)**	**MDL (sec)**
chb01	7	7	100	0.14	9.14
chb03	7	6	85.71	10.86	13.83
chb05	5	5	100	6.20	4.20
chb08	5	5	100	2.60	11.00
chb10	7	7	85.71	1.71	13.17
chb14	8	0	0	0	0
chb20	8	8	100	10.33	13.13
chb21	4	2	50	0.25	10.50
chb24	16	16	100	1.92	8.94
Total	67	55	82.09	3.45	10.38

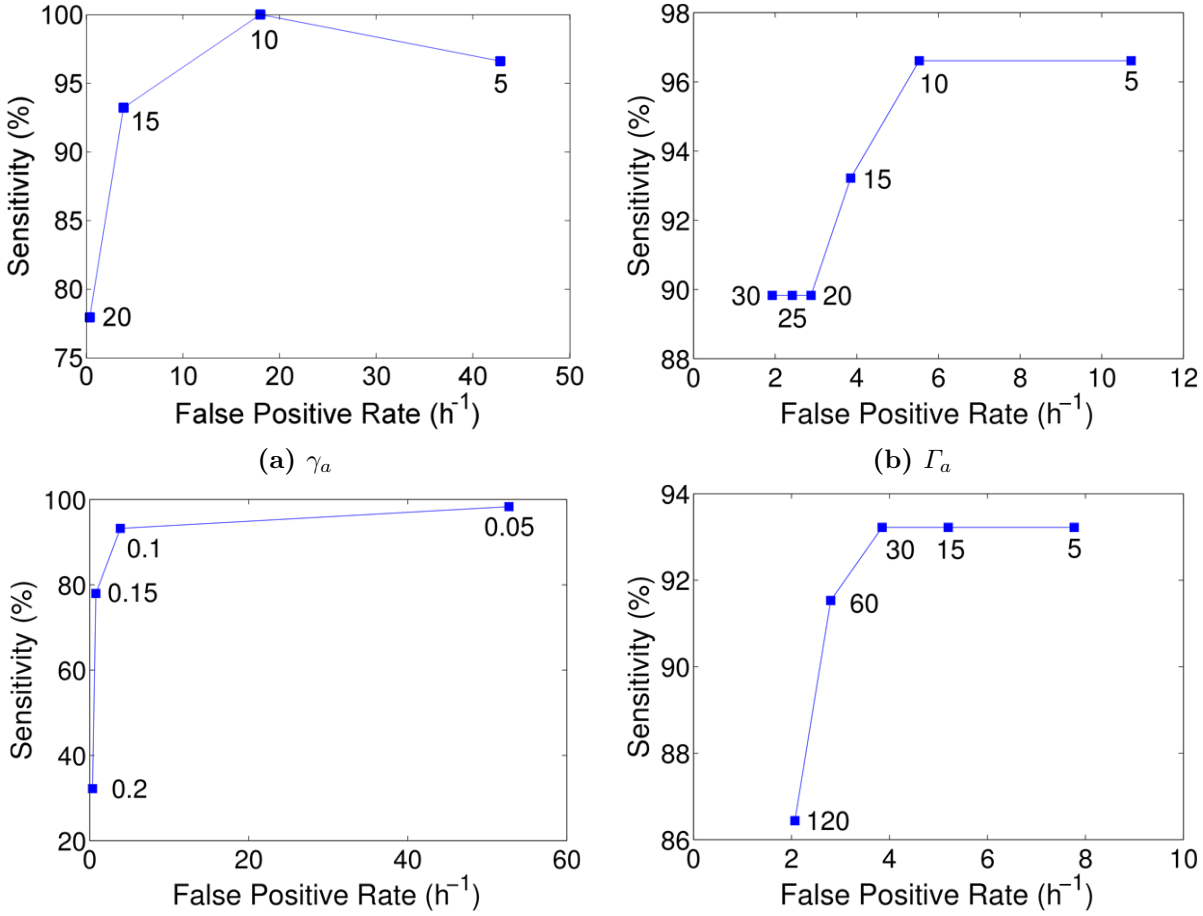
FDR: false detection rate (i.e., specificity), and MDL: mean detection latency

rate according to the changes of each hyperparameters, and Figure 4.13 shows the corresponding mean detection latency. When these parameters are set to low values, our system will generates more false alarms. For instance, when ξ is set to 0.05, the anomaly detector will treat most 1-s EEG epochs as abnormal, leading a high false positive rate.

Some of the ROC curves do not monotonically increase or decrease. This is due to the combination of the CUSUM algorithm and the alarm delay. For instance, when the values of γ_a and/or Γ_a are set to low values, our system will generate alarms even when the value of the anomaly channel is not high. Thus when a false alarm is generated just before the actual onset of the seizure, which has a duration shorter than the alarm delay, our system will miss such seizure.

Supervised Seizure Detection Performance

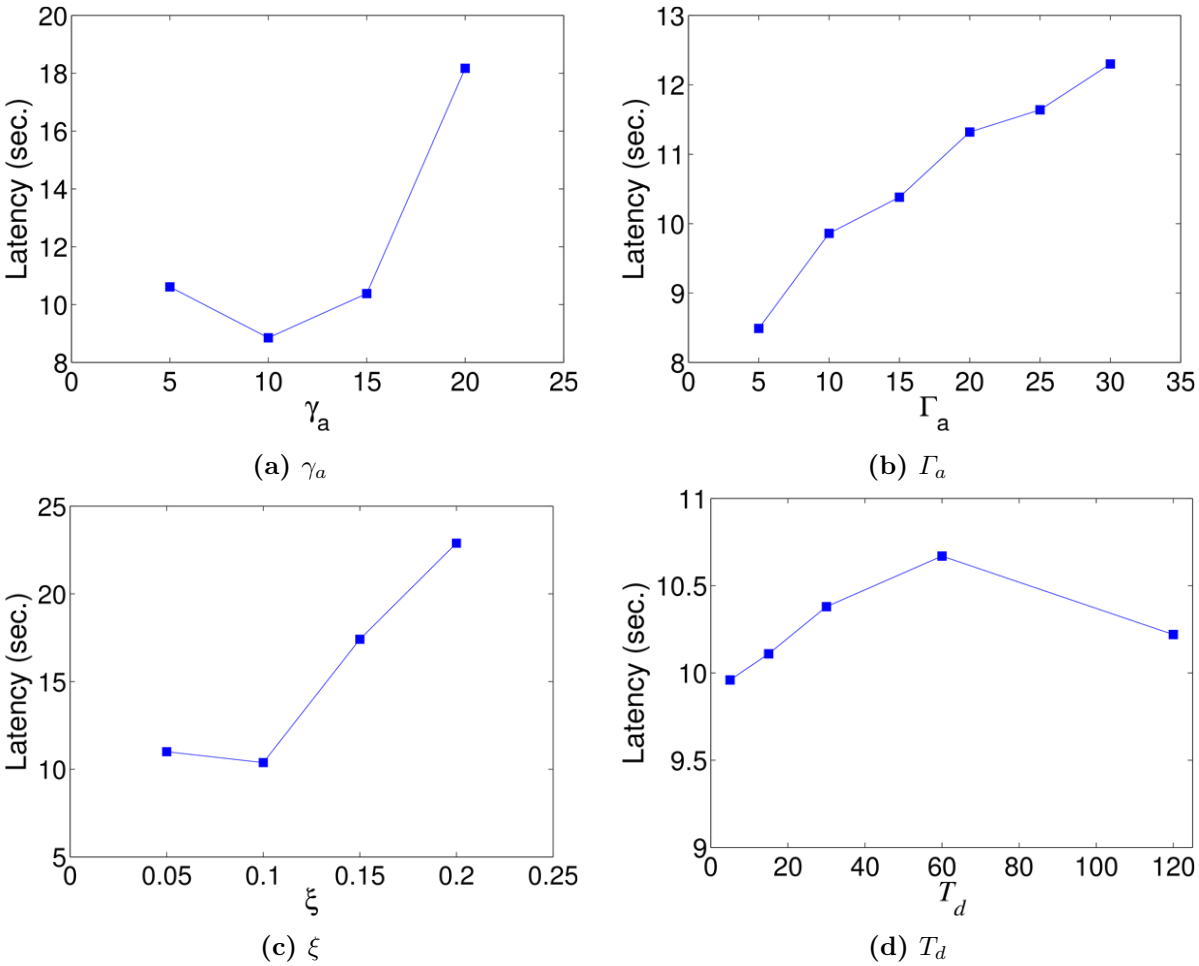
In this experiment, we annotated an extra label in addition to the seizure labels from the CHB-MIT dataset. This was because we would like to test whether the anomaly classifier were capable of identifying multiple types of EEG data (i.e., not only non-ictal and ictal). EEG data would be labeled with the extra label if they met three criteria. Firstly, those data that had patterns different from normal EEG data (e.g., background or interictal) and they were not annotated as seizures. Secondly, those data that had the high value of anomaly channel. Thirdly, those data that had the previous two characteristics for a period of time. An example of the EEG patterns that were labeled with the



(a) γ_a **(b)** Γ_a

(c) ξ **(d)** T_d

Figure 4.12: ROC curves for different hyperparameters in the unsupervised seizure detection setting. These graphs show the changes in sensitivity versus false detection rate according to the changes of each hyperparameter.



(a) γ_a **(b)** Γ_a

(c) ξ **(d)** T_d

Figure 4.13: Mean detection latency according to the change of each hyperparameter in the unsupervised seizure detection setting.

extra label is shown in Figure 4.10b.

Training Parameters. The training process of the anomaly detector is similar to the one mentioned in the unsupervised setting. The training process of the anomaly classifier is as follows. EEG data were first pre-processed using mean normalization and scaling with standard deviation before being used in the two-step training. In the first step, the batch size of 10 and the learning rate of 0.05 were used. β and ρ were set to 3 and 0.1 respectively. In the second step, the batch size of 30 was used. The learning rate was initially set to 0.05 and was then gradually decreased to 0.01 from the 20th to 30th epoch. The number of epochs for the first and the second steps were 30 and 50 respectively.

After training, the seizure detection system for each patient was used to detect seizures in the remaining seizure file of that patient. The training time of the anomaly detector was the same as in the unsupervised setting. The training time of the anomaly classifier varied from 3.5 h to 15 h depending on the number of seizure files, N_s, in each patient. The run time of the system for each 1-s EEG epoch from each channel was $\sim$17 ms.

Results. Table 4.5 shows a summary of the performance of the supervised seizure detection when the parameters of the system ξ, γ_a, Γ_a, T_d, γ_s, Γ_s and T_w were set to 0.1, 10, 15, 1, 3, 7 and 5 respectively. Our system achieved the sensitivity of 82.09% (i.e., detected 55 of 67 seizures) with the false positive rates of 0.97 alarms per hour and the mean latency of 10.33 seconds.

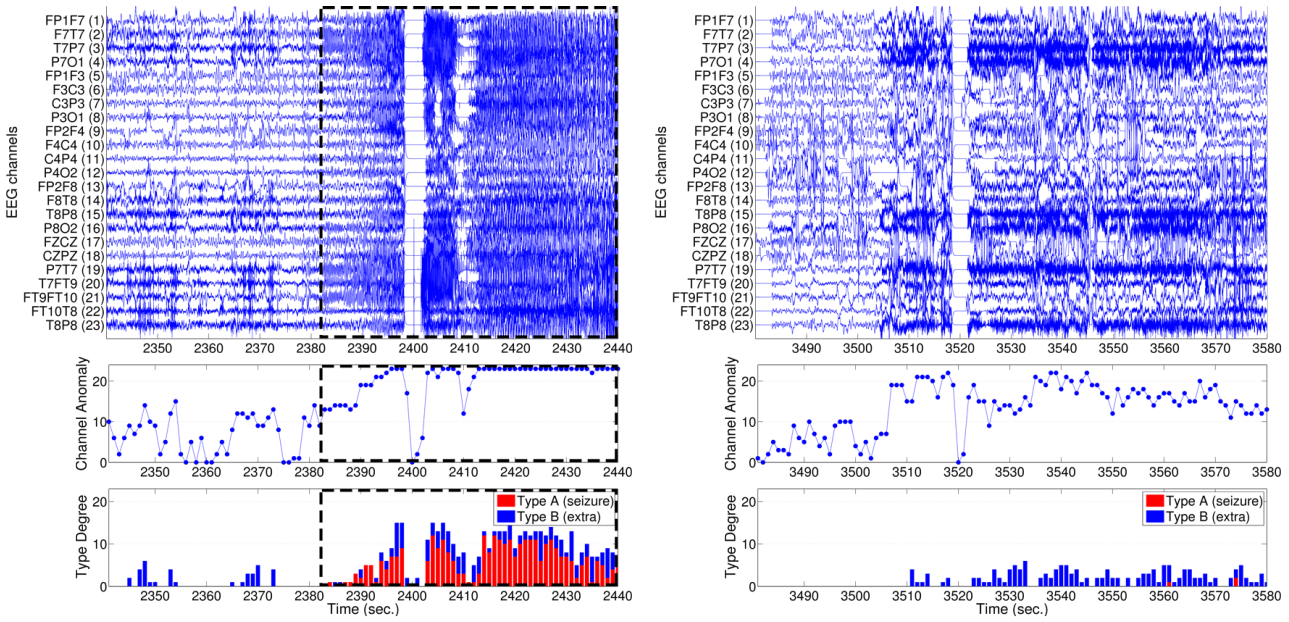
Comparing to the unsupervised setting, it can be seen that the false detection rate was significantly reduced. This was because our system utilized the anomaly classes from the anomaly classifier to prevent the system from generating alarms for other anomalies. It was not surprised that our system achieved the same sensitivity as the unsupervised setting. This was because our system still relied on the anomaly annotations from the anomaly detector. Even though we lowered the value of γ_a and T_d to 10 and 1 respectively so that our system could quickly generate alarms for anomalies (including seizures), the mean detection latency was similar to the unsupervised setting. This was because the seizure channel still started to increase after the seizures had occurred for a while. Also, our system still performed poorly for chb14 and chb21.

Figure 4.14 shows examples of the anomaly channel and the type degree computed from 1-s EEG epochs during ictal and interictal periods. In Figure 4.14a, the type degree of seizure (or seizure channel) increases during the seizure period. In Figure 4.14b, the type degree of extra increases during the interictal period. From these two figures, it can be seen that the anomaly classes from the

Table 4.5: Performance of our seizure detection system in the supervised setting when ξ=0.1, γ_a=10, Γ_a=15, T_d=1, γ_s=3, Γ_s=7 and T_w=5

Patient	$N_{seizure}$	N_{detect}	**Sensitivity (%)**	**FDR (per hour)**	**MDL (sec)**
chb01	7	7	100	0.29	4.71
chb03	7	7	100	1.57	13
chb05	5	5	100	3	5.8
chb08	5	5	100	1	9.4
chb10	7	7	85.71	0.05	17.17
chb14	8	0	0	0	0
chb20	8	8	100	2.5	13.63
chb21	4	1	25	0.05	10
chb24	16	16	100	1.42	9.13
Total	67	55	82.09	0.97	10.33

FDR: false detection rate (i.e., specificity), and MDL: mean detection latency



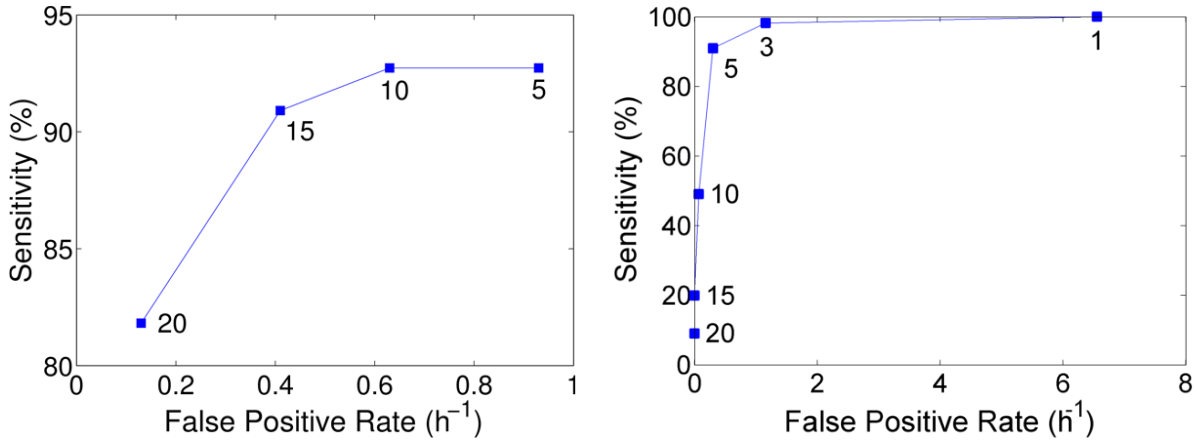
(a) Ictal EEGs (start at 2382 s)

(b) Interictal EEGs

Figure 4.14: Examples of the anomaly channel and the seizure channel computed from EEGs during ictal (a) and interictal (b) periods. The top graphs show raw EEGs from multiple channels. The middle graphs show the anomaly channel. The bottom graphs show the type degrees indicating the number of channels annotated with non-interictal types such as seizures. The type A (seizure) is the same the seizure channel. The type B (extra) is the class that we manually annotated.

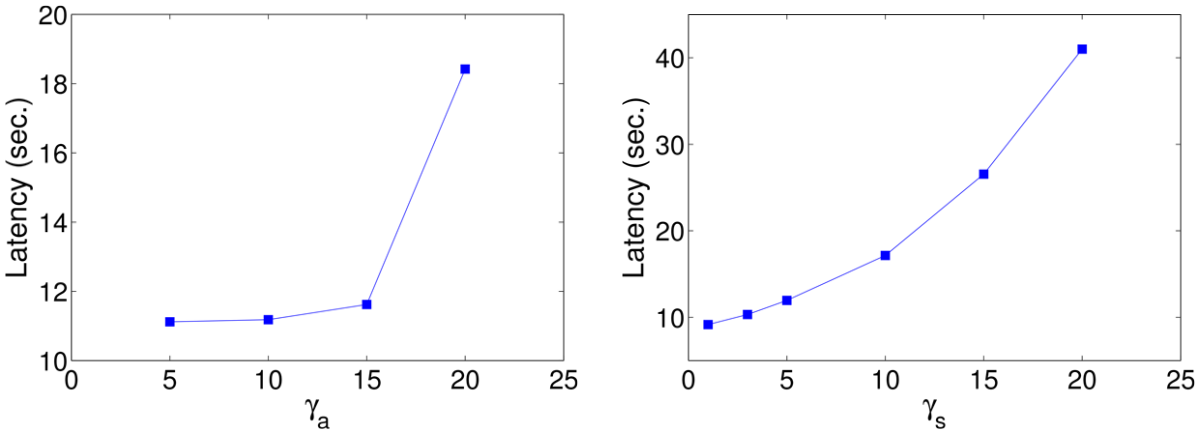
anomaly classifier can be used to reduce the number of false seizure alarms and can learn to classify EEG epochs into different types.

We also plot ROC curves to demonstrate the effect of changing γ_a and γ_s. Figure 4.15 shows changes



(a) γ_a **(b)** γ_s

Figure 4.15: ROC curves for different hyperparameters in the supervised seizure detection setting. These graphs show the changes in sensitivity versus false detection rate according to the changes of each hyperparameter.



(a) γ_a **(b)** γ_s

Figure 4.16: Mean detection latency according to the change of each hyperparameter in the unsupervised seizure detection setting.

in sensitivity versus false detection rate according to the changes of each parameter, and Figure 4.16 shows the corresponding mean detection latency. When we increase the value of γ_a, the sensitivity gradually decrease. This is because it becomes more difficult to detect an increase in the anomaly channel unless 1-s EEG epochs from most of the EEG channels are abnormal. When we increase the value of γ_s to be more than 5, the sensitivity significantly decrease. This may be because the seizure patterns in most of the patients do not appear prominently or do appear inconsistently across all EEG channels.

Comparison with State-of-the-Art Approaches

Table 4.6 shows a general comparison between the proposed seizure detection system and the state-of-the-art ones across sensitivity, specificity (i.e., false detection rate) and mean detection latency. They have evaluated the detection performance on either the same CHB-MIT dataset, or their private datasets. It can be seen that, without utilizing any labels or feature extraction algorithms, our system in both unsupervised and supervised settings achieved a comparable detection performance compared to the other systems. Even though our system showed no improvement in the detection performance, we believe that our system can be useful when the labeled seizure EEG signals are not available or limited. This is because our system can detect seizures by utilizing unlabeled EEG signals. When the seizure signals become sufficiently available, our system can utilize these annotations to reduce the number of false seizure alarms.

It should be noted that we only compare our proposed system with the state-of-the-art ones that were trained in the supervised setting to detect the onset of the seizures. The systems that detect the onset of the seizures are different from the ones that classify EEG epochs into non-ictal and ictal classes [132], and they also use different evaluation metrics. The former focuses on generating seizure alarms as soon as possible after seizures have started, while the latter focuses on the classification accuracy of EEG epochs.

4.4.5 Discussion

In this study, we develop the patient-specific seizure detection system that utilize raw unlabeled EEG signals to detect seizures and other anomalies. We also investigate how the availability of the seizure annotations can be used to improve the detection performance.

The preliminary results showed that, without utilizing any seizure annotations and feature extraction algorithms (i.e., unsupervised learning), our system achieved a comparable seizure detection performance compared to the state-of-the-art seizure onset detection systems that were trained in the supervised setting. As expected, our system in the unsupervised setting gave the fairly high false detection rate compared to the existing systems as our system is designed to generates alarms for any anomalies including seizures. When the seizure annotations become sufficiently available, the results showed that our system could utilize this information to reduce the number of false seizure alarms. We

Table 4.6: A general comparison between the proposed system and the existing seiuzre onset detection systems across sensitivity, false detection rate and mean detection latency.

Methods	Dataset	Sensitivity (%)	FDR (per hour)	MDL (sec)
Ref. [109]	Private	76	0.34	10
Ref. [116]	Private	81	0.60	16.9
Ref. [117]	Private	90.5	0.51	8.37
Ref. [18]	CHB-MIT	96	2.0 (per day)	4.6
Ref. [21]	CHB-MIT	98.5	-	1.76
Proposed (unsup.)	CHB-MIT	82.09	3.45	10.38
Proposed (sup.)	CHB-MIT	82.09	0.97	10.33

FDR: false detection rate (i.e., specificity), and MDL: mean detection latency

also show that our anomaly classifier could classify more than two types of EEGs such as interictal, ictal and extra.

In our experiment, we found that our system could not detect seizures in some patients, especially chb14 and chb21. This is because the amplitudes of the EEG signals during the seizure are relatively low. When our system computes RMS errors between the reconstructed EEG epochs and the observed seizure EEG epochs, the error between these epochs are low even though their patterns are not similar. This results in the low values of the anomaly channel during such seizures and therefore prevents our system from generating alarms. We also found that the mean detection latency of our system was relatively high compared to the existing approaches. This is because seizure activity starts to appear prominently after the seizures have begun for a while, which caused the anomaly channel to increase slowly.

These problems may well be because of two main reasons. Firstly, we only use the mean-square reconstruction loss (see Equation 4.5) to train the SSAEs in the anomaly detector. It is possible that the SSAEs ignores the EEG signals with low amplitudes as their reconstruction errors are low even thought the patterns are different. Thus the SSAEs only learns to reconstruct the EEG signals that have high amplitude. Secondly, for each patient, we use a single anomaly detector to detect anomalies in EEG signals from all channels. This means that 1-s EEG epochs from all channels are fed into one anomaly detector. Comparing to human experts, our SSAEs lack temporal (across time) and spatial (across channels) information that the experts utilize or consider while they are labeling EEG signals. This can prevent the SSAEs from recognizing the seizure activities that appear over a period of time across multiple channels. Because of these problems, we, therefore, decided not to proceed with our

approach on the remaining 13 patients in the dataset.

There are three potential improvements to the anomaly detector that can alleviate these problems. Firstly, we can incorporate the Generative Adversarial Nets (GANs) framework into the training process to improve the quality of the reconstructed signals. It has been shown in computer vision that GANs can help provide additional gradients that are useful for generating realistic images [10, 77]. Secondly, we can utilize Convolutional Neural Networks (CNNs) to efficiently increase the input dimension (across time and channels) of the model for more temporal and spatial information. When the number of input dimension is very large, CNNs are more computationally efficient and less prone to overfitting compared to SSAEs. Thirdly, we can utilize information from the other biosignals such as Electrocardiogram (ECG). We suspect that using only EEG signals are not sufficient for unsupervised seizure or anomaly detection, as it may be difficult to detect seizures via using the reconstruction error. It may be better if we detect desynchronization among biosignals from more than one domains as we can use other signals as a reference for the expected normal behavior. It has also been shown in several studies that there might be a correlation or synchronization between EEG and ECG signals prior or during seizures [133, 134] respectively.

Therefore, we believe that a better approach would be to train a CNN-based deep learning model in the *unsupervised setting* with the GAN framework to detect *desynchronization* (i.e., anomaly) among simultaneous biosignals. For the anomaly classifier in the *supervised setting*, we can also utilize the CNNs to extract features from multiple EEG channels simultaneously to improve the classification performance. However, we believe that the deep learning model based on supervised training might not be suitable for seizure detection due to the limited availability of the seizure data and the patterns of seizures within the same patient can be inconsistent. A suitable problem, that we can train deep learning in the supervised setting to extract meaningful features from *raw* biosignals, would be the problem that has well-defined and consistent signal patterns such as sleep stage scoring.

4.5 Conclusion

We firstly investigate the possibility of applying SSAEs to automatically learn features from raw EEG signals for seizure detection. Our preliminary results showed that our seizure detector could detect most of the seizures with a low false detection rate and a reasonable detection latency. Then we extend this work to develop a patient-specific seizure detection system that can utilizes raw,

unlabeled EEG signals to detect seizures and other abnormal EEG patterns (i.e., unsupervised seizure detection). Whenever the seizure annotations become sufficiently available, our system can also utilize this information to improve the detection performance. Our preliminary results demonstrated that, without utilizing any seizure annotations and feature extraction algorithms, our system achieved a comparable detection performance (Sensitivity: 82.1%, FDR: 3.45 per hour, MDL: 10.38 sec) compared to the state-of-the-art seizure onset detection systems that were trained in the supervised setting (Sensitivity: 96.0%, FDR: 2.0 per day, MDL: 4.6 sec). The results also showed that our system could utilize the seizure annotations to reduce the number of false seizure alarms (FDR: 0.97 per hour).

However, from our experiments, we found that our system could not detect seizures that exhibit abnormal activities with a relatively low amplitude in some patients. We suspect that using only EEG signals might not be sufficient for unsupervised seizure detection. We believe that a better approach would be to train a deep learning model in an unsupervised setting to detect *desynchronization* (i.e., anomaly) among simultaneous biosignals. This is because the model can utilize signals from other domain as a reference for the expected normal behavior. We also believe that a suitable time-series-based problem for a deep learning model based on supervised learning would be the problem that has well-defined and consistent signals patterns such as sleep stage scoring.

In the next chapter, we will discuss our work that has confirmed our hypothesis about a more suitable problem for deep learning trained in the supervised setting.

5

Automatic Sleep Stage Scoring

In the previous chapter, we found that our deep learning model based on supervised training had limitations in the epileptic seizure detection problem. This is because of two main reasons:

1. There are only limited amount of seizure data (i.e., class imbalance problem) and the patterns of seizures are not consistent. It would be difficult for the deep learning model to *automatically* learn useful features from biosignals without utilizing any hand-engineered features such as image-based representations of EEG signals [110]. It is better to first try to develop a deep learning model for the problem that has well-defined and consistent patterns such as sleep stage scoring.

2. The SSAEs are fully-connected neural networks. When we would like to increase the input dimensions of the model to increase the temporal (across time) and spatial (across channels) information of EEG signals, it would be too computationally expensive as the number of the model parameters will increase significantly. This can also lead to the overfitting problem. It is better to utilize CNNs to efficiently increase the input dimension.

Therefore, in this chapter, we develop a deep learning model for sleep stage scoring problem that typically has sufficient amount of annotated training data (easier to collect and annoate compare to epileptic seizures). In this problem, the patterns of biosignals are also consistent among each stages as they are properly defined in the standard sleep manuals [26, 27]. We utilize a more computational efficient network architecture to automatically extract both time-invariant (i.e., CNN) and temporal (i.e., LSTM) features for sleep stage scoring.

It should be noted that we only focus on the *supervised* training in this chapter as we would like to first develop a model that can learn useful features from raw biosignals. We will later discuss our proposed approach based on *unsupervised* training to detect desynchronization among biosignals from multiple domains in the next chapter.

5.1 Introduction

Sleep plays an important role in human health. Being able to monitor how well people sleep has a significant impact on medical research and practice [135].

Typically, sleep experts determine the quality of sleep using electrical activity recorded from sensors attached to different parts of the body. A set of signals from these sensors is called a polysomnogram (PSG), consisting of an electroencephalogram (EEG), an electrooculogram (EOG), an electromyogram (EMG), and an electrocardiogram (ECG). This PSG is segmented into 30-s epochs, which are then classified into different sleep stages by the experts according to sleep manuals such as the Rechtschaffen and Kales (R&K) [26] and the American Academy of Sleep Medicine (AASM) [27]. In most of the cases, the experts classify the PSG epochs in five sleep stages: Stage W (wakefulness), Stage N1 (Non-REM 1), Stage N2 (Non-REM 2), Stage N3 (Non-REM 3), and Stage REM (Rapid Eye Movement). This process is called sleep stage scoring or sleep stage classification. This manual approach is, however, labor-intensive and time-consuming due to the need for PSG recordings from several sensors attached to subjects over several nights.

There have been a number of studies trying to develop a method to automate sleep stage scoring based on multiple signals such as EEG, EOG and EMG [17, 136, 137], or single-channel EEG [22, 138, 139]. These methods firstly extract time-domain, frequency-domain and time-frequency-domain features from each recording epoch. In the case of multiple signals, the features from all signals in one epoch were concatenated into one feature vector. The features are then used to train classifiers to identify the sleep stage of the epoch. However, we believe that these methods may well not generalize to a larger population due to the heterogeneity among subjects and recording hardware. This is because these features were hand-engineered based on the characteristics of the available dataset.

Recently, deep learning, a branch of machine learning that utilizes multiple layers of linear and non-linear processing units to learn hierarchical representations or features from input data, has been

employed in sleep stage scoring. For instance, the authors in [24] have investigated a capability of Deep Belief Nets (DBNs) to learn probabilistic representations from preprocessed raw PSG. Convolutional Neural Networks (CNNs) have also been applied to learn multiple filters that are used to convolve with small portions of input data (i.e., convolution) to extract time-invariant features from raw Fpz-Cz EEG channel [140]. However, the results from the literature showed that applying deep learning on hand-engineered features performed better than on raw signals [22, 24]. This might well be because the authors did not consider temporal information that sleep experts use when they determine the sleep stage of each epoch.

Only a few number of literature have explored Recurrent Neural Networks (RNNs) in sleep stage scoring. RNNs are capable of conditioning outputs on all previous inputs, as they maintain internal memory and utilizes feedback (or loop) connections to learn temporal information from sequences of inputs. The main advantage of RNNs is that they can be trained to learn long-term dependencies such as transition rules [27] that sleep experts use to identify the next possible sleep stages from a sequence of PSG epochs. Elman RNNs have been applied on energy features from the Fpz-Cz EEG channel [141]. In our previous work [23], we also applied Long Short-Term Memory (LSTM) on time-frequency-domain features from the F4-EOG and Fp2-EOG channels separately. Even though the reported results were promising, these methods still rely on hand-engineered features.

This paper introduces *DeepSleepNet*, a model for automatic sleep stage scoring based on raw single-channel EEG, which is different from the existing works that develop algorithms to extract features from EEG. We aim to automate the process of hand-engineering features by utilizing the feature extraction capabilities of deep learning. The main contributions of this work are as follows:

- We develop a new model architecture that utilizes two CNNs with different filter sizes at the first layers and bidirectional-LSTMs. The CNNs can be trained to learn filters to extract time-invariant features from raw single-channel EEG, while the bidirectional-LSTMs can be trained to encode temporal information such as sleep stage transition rules into the model.
- We implement a two-step training algorithm that can effectively train our model end-to-end via backpropagation, while preventing the model from suffering class imbalance problem (i.e., learning to classify only the majority of sleep stages) presented in a large sleep dataset.
- We show that, without changing the model architecture and the training algorithm, our model could automatically learn features for sleep stage scoring from different raw single-channel EEGs

from two datasets, that have different properties (e.g., sampling rate) and scoring standards (AASM and R&K), without utilizing any hand-engineered features.

5.2 DeepSleepNet

The architecture of DeepSleepNet consists of two main parts as shown in Figure 5.1. The first part is representation learning, which can be trained to learn filters to extract time-invariant features from each of raw single-channel EEG epochs. The second part is sequence residual learning, which can be trained to encode the temporal information such as stage transition rules [27] from a sequence of EEG epochs in the extracted features. This architecture is designed for scoring 30-s EEG epochs following the standard of AASM and R&K manuals.

5.2.1 Representation Learning

We employ two CNNs to extract what we call *time-invariant* features from raw single-channel 30-s EEG epochs. We design these two CNNs to have the first convolutional layers with small (**CNN-small**) and large (**CNN-large**) filter sizes. CNN-small is one the left-side of the representation learning part, and CNN-large is on the right-side (see Figure 5.1). This architecture is inspired by the way signal processing experts control the trade-off between temporal and frequency precision in their feature extraction algorithms [39]. The small filter is better to capture temporal information (i.e., when certain of EEG patterns appear), while the larger filter is better to capture frequency information (i.e., frequency components).

Each of the two CNNs consists of four convolutional layers (**conv**) interleaved with two max-pooling layers (**max-pool**).

- **conv.** Each convolutional layer receives inputs and performs three operations sequentially: 1D-convolution with its filters, batch normalization [84], and applying the rectified linear unit (ReLU) activation function (i.e., $\text{relu}(x) = \max(0, x)$). The 1D-convolution operation convolves a set of filters across its inputs, producing a set of feature maps which can be interpreted as time-invariant features (see Figure 5.2). The batch normalization is used to preprocess the output feature maps from the 1D-convolution operation before applying the ReLU function. The main idea is to make normalization as a part of the model, in which the normalization is performed for

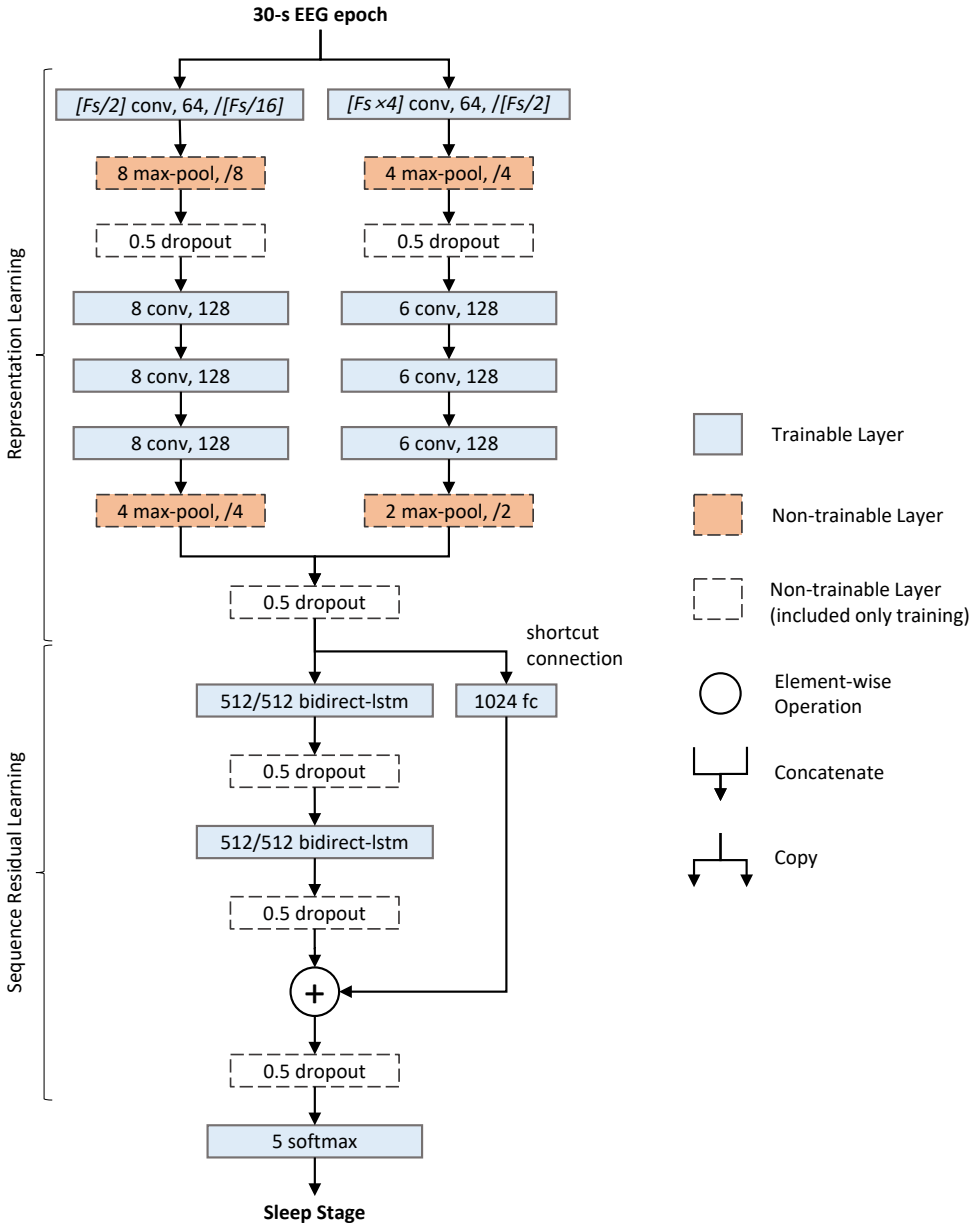


Figure 5.1: An overview architecture of DeepSleepNet consisting of two main parts: **representation learning** and **sequence residual learning**. The representation learning part consists of two CNNs that have the first convolutional layers with small (CNN-small, left-side) and large (CNN-large, right-side) filter sizes. The specifications of the first convolutional layers depend on the sampling rate (Fs) of the EEG data (see Section 5.2.4). Each rectangular box represents one layer in the model. **conv** is a convolutional layer [*filter_size* conv, *n_filters*, /*stride_size*]. **max-pool** is a max-pooling layer [*pool_size* max-pool, /*stride_size*]. **dropout** is a dropout layer [*probability* dropout]. The sequence residual learning part consists of bidirectional-LSTMs and a shortcut connection. **bidirect-lstm** is a bidirectional LSTM layer [*forward_hidden_units*/*backward_hidden_units* bidirect-lstm]. The **shortcut connection** in the sequence residual learning that branches off from the output of the two CNNs reformulates this part into a residual function. **fc** is a fully-connected layer [*hidden_units* fc]. The arrows indicate the flow of data from raw single-channel 30-s EEG epochs to sleep stages, where the softmax layer, **softmax** [*n_classes* softmax] is used to compute the output probability of each sleep stage. Each trainable layer is a layer containing parameters to be optimized during a training process, while each non-trainable layer only performs its operation without any parameters to be optimized.

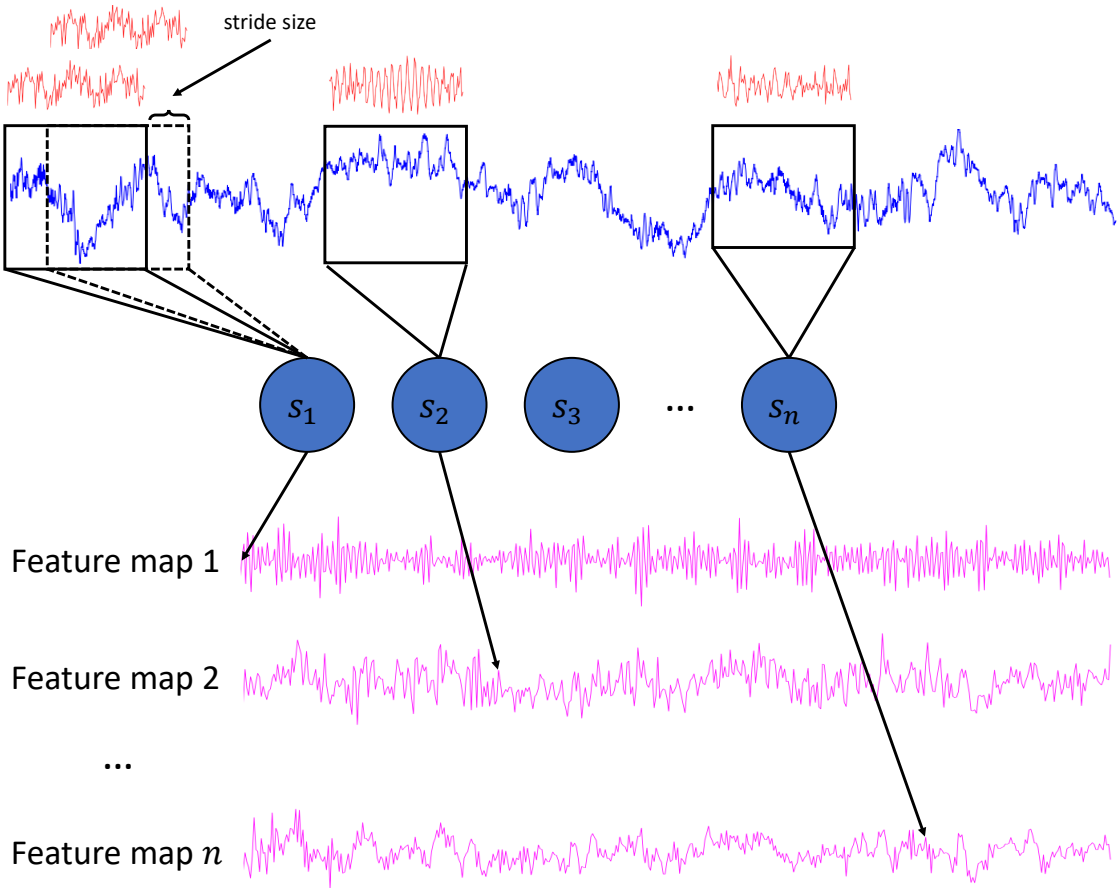


Figure 5.2: The 1D-convolution operation to extract time-invariant features. For simplicity, we will use the 1D-convolution operation performed at the first convolutional layers of the two CNNs, where an input is a raw single-channel 30-s EEG epoch (**blue signal**), as an example. Each filter (i.e., the weights of each neuron $s_1, \ldots, s_n$ in the convolutional layer) (**red signal**) is convolved with the EEG epoch. Each rectangular box highlights the computation of one value in a feature map (**magenta signal**), which is a dot product between a filter and a chunk of the EEG epoch. The dash-line rectangular box shows how each filter is slid across the EEG epoch during the convolution (i.e., stride). The resulting feature maps are concatenated before forwarding to the next layer. The 1D-convolution operations at the downstream layers are the same as in this example, except that there will be more than one input channel, depending on the number of feature maps. In this case, the filters of such convolutional layers will also have the same number of channels as the inputs allowing the computation of one value in a feature map to cover all input channels. As each filter is slid across the EEG epoch, each convolutional layer can learn a set of filters to detect certain patterns in the EEG epoch, regardless of when such patterns appear (i.e., **time-invariant**), via backpropagation.

each mini-batch of training data. This allows us to explicitly force the activations throughout the model to approximately have the mean of zero and the variance of 1. The batch normalization can help alleviate a problem of initializing the model parameters and choosing the right learning rate, accelerating the training of deep learning models [84]. Also, the batch normalization is a differentiable transformation facilitating the model to learn normalizing parameters to preprocess its input via backpropagation. The specifications of the filter sizes, the number of filters, and the stride sizes can be found in Figure 5.1 [*filter_size* conv, *n_filters*, /*stride_size*].

- **max-pool.** Each max-pooling layer downsamples inputs using max operation. This pooling layer can help alleviate the overfitting problem, make the features more time-invariant, and reduce the dimensions of the inputs to be processed by the next layer, without introducing any additional parameters to the model. This layer is useful as we are more interested in the presence of particular patterns (e.g., sleep spindles or k-complex) than when they occur in each EEG epoch. The pooling sizes and the stride sizes can be found in Figure 5.1 [*pool_size* max-pool, /*stride_size*].

There are also several **dropout** layers interleaved throughout our model. These dropout layers are used as a regularization technique which will be explained later in Section 5.3.4.

Our representation learning part processes the raw single-channel 30-s EEG epoch as following. Suppose there are N 30-s EEG epochs $\{\mathbf{x}^{(1)}, ..., \mathbf{x}^{(N)}\}$ from single-channel EEG. We use the two CNNs to extract the i-th feature $\mathbf{a}^{(i)}$ from the i-th EEG epoch $\mathbf{x}^{(i)}$ as follows:

$$\mathbf{h}_s^{(i)} = CNN_{\theta_s}(\mathbf{x}^{(i)}) \tag{5.1}$$

$$\mathbf{h}_l^{(i)} = CNN_{\theta_l}(\mathbf{x}^{(i)}) \tag{5.2}$$

$$\mathbf{a}^{(i)} = \mathbf{h}_s^{(i)} || \mathbf{h}_l^{(i)} \tag{5.3}$$

where CNN represents a function that transform a 30-s EEG epoch $\mathbf{x}^{(i)}$ into a feature vector $\mathbf{h}^{(i)}$ using a CNN, θ_s and θ_l are parameters of the CNN-small and CNN-large respectively, and $||$ is a concatenate operation that combines the outputs from two CNNs together. These concatenated or linked features $\{\mathbf{a}^{(1)}, ..., \mathbf{a}^{(N)}\}$ are then forwarded to the sequence residual learning part.

5.2.2 Sequence Residual Learning

We apply the residual learning framework [9] to design our sequence residual learning part. This part consists of two main components: bidirectional-LSTMs [70] (**bidirect-lstm**) and a shortcut connection (**shortcut connection**) (see Figure 5.1).

- **bidirect-lstm.** Each bidirectional LSTM layer consists of two LSTMs that summarizes the sequence of features extracted from CNN-small and CNN-large in both forward and backward directions independently (see Figure 5.3). This allows us to exploit the summaries of both the past and the future that are most sensitive to the input value around each time step to determine the sleep stage. We also use peephole connections [142, 143] in the LSTMs which allow their gating mechanism to inspect their current memory cell before the modification. Each LSTM maintains its cell state and hidden state to encode the lossy summary of the input sequence (as discussed in Section 2.9.5), whose sizes are the same as the number of hidden units. We employ two layers of bidirectional-LSTMs to learn temporal information such as stage transition rules [27] which sleep experts use to determine the next possible sleep stages based on the previous stages. For instance, the AASM manual suggests that if a subject is in sleep stage N2, continue to score epochs with low amplitude and mixed frequency EEG activity as N2 even though K complexes or sleep spindles are not present. In this case, the bidirectional-LSTMs might learn to remember that it has seen the stage N2, and continue to score successive epochs as N2 if they still detect the low amplitude and mixed frequency EEG activity. The number of hidden units of each LSTM can be found in Figure 5.1 [*forward_hidden_units*/*backward_hidden_units* bidirect-lstm].

- **shortcut connection.** We use a shortcut connection to reformulate the computation of this part into a residual function [9]. This enables our model to add temporal information it learns from the past and the future of input sequences into the feature extracted from the two CNNs. We also use a fully-connected layer (**fc**) in the shortcut connection to transform the features from the CNNs into a vector that can be summed with the output from the bidirect-lstm layer. This fully-connected layer performs matrix multiplication with its weight parameters, batch normalization, and applying the ReLU activation sequentially. The number of hidden units of this layer can be found in Figure 5.1 [*hidden_units* fc].

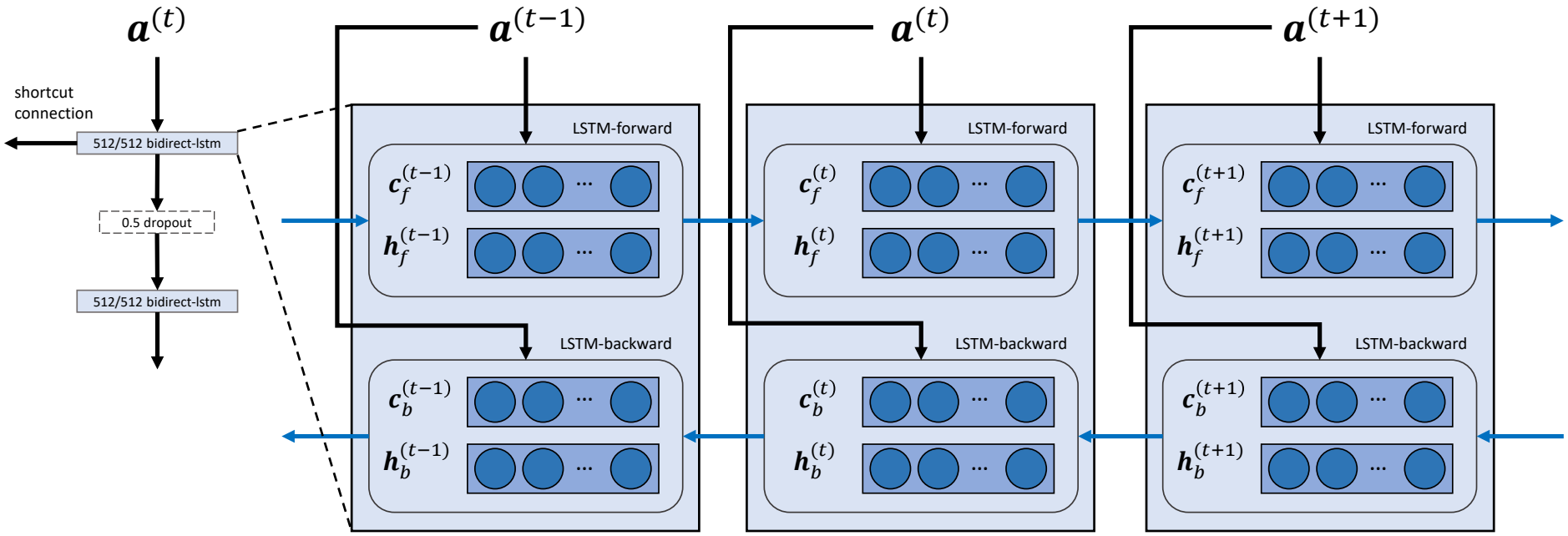


Figure 5.3: bidirect-lstm layer. **Left:** A part of our sequence residual learning in the DeepSleepNet model that has two bidirectional-LSTM layers. **Right:** The unfolded version of the bidirectional-LSTM with no outputs. Each bidirectional-LSTM consists of two LSTMs that process the input sequence in the forward (**LSTM-forward**) and backward (**LSTM-backward**) directions. Each LSTM maintains its own cell state ($\mathbf{c}_f^{(t)}$ and $\mathbf{c}_b^{(t)}$) and hidden state ($\mathbf{h}_f^{(t)}$ and $\mathbf{h}_b^{(t)}$) at every time step t to encode the lossy summary of the input sequence. This allows the bidirectional-LSTM to exploit the summaries of both the past and the future that are most sensitive to the input value around each time step to determine the sleep stage. The input $\mathbf{a}^{(t)}$ is the output from the representation learning part (see Equation 5.3).

Suppose there are N input samples from the representation learning $\{\mathbf{a}^{(1)}, \ldots, \mathbf{a}^{(t)}, \ldots, \mathbf{a}^{(N)}\}$ arranged sequentially, our sequence residual learning processes such feature samples as follows:

$$\mathbf{h}_f^{(t)}, \mathbf{c}_f^{(t)} = LSTM_{\theta_f}(\mathbf{h}_f^{(t-1)}, \mathbf{c}_f^{(t-1)}, \mathbf{a}^{(t)}) \tag{5.4}$$

$$\mathbf{h}_b^{(t)}, \mathbf{c}_b^{(t)} = LSTM_{\theta_b}(\mathbf{h}_b^{(t+1)}, \mathbf{c}_b^{(t+1)}, \mathbf{a}^{(t)}) \tag{5.5}$$

$$\mathbf{o}^{(t)} = \mathbf{h}_f^{(t)} || \mathbf{h}_b^{(t)} + FC_\theta(\mathbf{a}^{(t)}) \tag{5.6}$$

where $LSTM$ represents a function that processes sequences of features $\mathbf{a}^{(t)}$ using the two-layers LSTM parameterized by θ_f and θ_b for forward and backward directions (here we use t as the time index instead of i to introduce the notion of time sequence); $\mathbf{h}$ and $\mathbf{c}$ are vectors of hidden and cell states of the LSTMs; $\mathbf{h}_f^{(0)}, \mathbf{c}_f^{(0)}, \mathbf{h}_b^{(N+1)}$ and $\mathbf{c}_b^{(N+1)}$ of forward and backward LSTMs are set to zero vectors; FC represents a fully-connected layer parameterized by θ that transform features $\mathbf{a}^{(t)}$ into a vector that can be added (element-wise) with the concatenated output vector $\mathbf{h}_f^{(t)} || \mathbf{h}_b^{(t)}$ from the bidirectional-LSTMs. The outputs from the sequence residual learning part $\{\mathbf{o}^{(1)}, ..., \mathbf{o}^{(N)}\}$ are then forwarded to the softmax layer.

It should be noted that the hidden and cell states $\mathbf{h}_f^{(t)}, \mathbf{h}_b^{(t)}, \mathbf{c}_f^{(t)}$ and $\mathbf{c}_b^{(t)}$ in Equation 5.4 and Equation 5.5 will be re-initialized to zeros at the beginning of each subject data during the training and testing. This is to make sure that the model uses only temporal information from the current subject data for both training and testing.

5.2.3 Sleep Stage Classification

Suppose there are N input samples from the sequence residual learning part $\{\mathbf{o}^{(1)}, ..., \mathbf{o}^{(N)}\}$, the softmax layer (**softmax** in Figure 5.1) determines the output probability of each class as follows:

$$\mathbf{z}^{(i)} = \mathbf{W}^T \mathbf{o}^{(i)} + \mathbf{b} \ , \tag{5.7}$$

$$\hat{y}_j^{(i)} = \frac{\exp(z_j^{(i)})}{\sum_k \exp(z_k^{(i)})} \ , \tag{5.8}$$

where $\theta = \{\mathbf{W}, \mathbf{b}\}$ are parameters of the softmax layer, the subscript j and k indicate the index of a vector, $\hat{y}_j^{(i)}$ is the output probability of class index j of the i-th 30-s EEG epoch. Our model classifies

the i-th 30-s EEG epoch into the sleep stage whose has the highest probability:

$$\text{sleep stage}^{(i)} = \arg\max_j \hat{y}_j^{(i)} \quad . \tag{5.9}$$

where $\text{sleep stage}^{(i)} \in \{0, 1, 2, 3, 4\}$, corresponding to the five sleep stages $\{\text{W}, \text{N1}, \text{N2}, \text{N3}, \text{REM}\}$.

5.2.4 Model Specification

For the representation learning part, the parameters of the CNN-small and CNN-large were selected with the aim to capture temporal and frequency information from the EEG according to the guideline provided by [39]. For instance, in Figure 5.1, the filter size of the **conv-1** layers of the CNN-small was set to Fs/2 (i.e., half of the sampling rate (Fs)), and its stride size was set to Fs/16 to detect when certain of EEG patterns appear. On the other hand, the filter size of the **conv-1** layer of the CNN-large was set to Fs×4 to better capture the frequency components from the EEG. Its stride size was also set to Fs/2, which is larger than the **conv-1** layer of the CNN-small, as it is not necessary to perform a fine-grained convolution to extract frequency components. The filter and stride sizes of the subsequent convolutional layers **conv-[2-4]** were chosen to be small fix sizes. It is believed that the use of multiple convolutional layers with a small filter size instead of a single convolutional layer with a large filter can reduce the number of parameters and the computational cost, and can still achieve the similar level of model expressiveness [144].

For the sequence residual learning part, the parameters of the **bidirect-lstm** and **fc** layers were set to be smaller than the output of the representation learning part, which is 1024 in Figure 5.1. This is to restrict our model to select and combine only the important features to prevent overfitting. The number of the hidden units of the **bidirect-lstm** and the **fc** layer were also chosen to be the same, so that the output of the bidirect-lstm can be added (element-wise) with the features extracted from the CNN-small and CNN-large, facilitating the residual framework.

For the **softmax** layer, the number of hidden units was chosen to be five, equal to the number of possible sleep stages in this study.

5.3 Two-Step Training Algorithm

The two-step training algorithm (see Algorithm 5.1) is a technique we develop to effectively train our model end-to-end via backpropagation, while preventing the model from suffering class imbalance problem (i.e., learning to classify only the majority of sleep stages) present in a large sleep dataset. The algorithm first pre-trains the representation learning part of the model and then fine-tunes the whole model using two different learning rates.

5.3.1 Pre-training

The first step is to perform a supervised pre-training on the representation learning part of the model with a class-balance training set so that the model does not overfit to the majority of sleep stages. This can be seen in Algorithm 5.1, lines 1-8. Specifically, the two CNNs are extracted from the model and then stacked with a softmax layer, $softmax$. It is important to note that this $softmax$ is different from the last layer in the model (see Figure 5.1). This stacked softmax layer is only used in this step to pre-train the two CNNs, in which its parameters are discarded at the end of the pre-training. We denote these two CNNs stacked with $softmax$ as *pre_model*. Then the *pre_model* is trained with a class-balance training set using a mini-batch gradient-based optimizer called Adam [56] with a learning rate, lr. At the end of the pre-training, the softmax layer is discarded. The class-balance training set is obtained from duplicating the minority sleep stages in the original training set such that all sleep stage have the same number of samples (i.e., oversampling).

5.3.2 Fine-tuning

The second step is to perform a supervised fine-tuning on the whole model with a sequential training set. This can be seen in Algorithm 5.1, lines 9-19. This step is to encode the stage transition rules into the model as well as to perform necessary adjustments on the pre-trained CNNs. Specifically, the parameters θ_s and θ_l of the two CNNs of *init_model* are replaced with the ones from the *pre_model*, resulting in *model*. Then the *model* is trained with the sequence training set using a mini-batch Adam optimizer with two different learning rates, lr_1 and lr_2. As the CNNs part has already been pre-trained, we, therefore, use a lower learning rate lr_1 for the CNNs part and a higher learning rate lr_2 for the sequence residual learning part, and a softmax layer. We found that when we used the same learning

Algorithm 5.1 Two-step Training

Input: *init_model*, *data*
Output: *model*

Initialization:
$init_CNN_{\theta_s,\theta_l} \leftarrow extract_cnns(init_model)$
$pre_model \leftarrow stack(init_CNN_{\theta_s,\theta_l}, softmax)$
$data_{over} \leftarrow oversample(data)$
Pre-training Step:
for $i = 1$ **to** $n_pretrain_epochs$ **do**
  for each *batch* **in** $shuffle(data_{over})$ **do**
    $pre_model \leftarrow adam_{lr}(pre_model, batch)$
  end for
end for
Fine-tuning Step:
$pre_CNN_{\theta_s,\theta_l} \leftarrow extract_cnns(pre_model)$
$model \leftarrow replace_cnns(init_model, pre_CNN_{\theta_s,\theta_l})$
for $i = 1$ **to** $n_finetune_epochs$ **do**
  for each *subject* **in** *data* **do**
    $model \leftarrow reset_lstm_cell_state(model)$
    $subject_data_{seq} \leftarrow arrange_sequence(subject)$
    for each *batch* **in** $subject_data_{seq}$ **do**
      $model \leftarrow adam_{lr_1,lr_2}(model, batch)$
    end for
  end for
end for
return *model*

rate to fine-tune the whole network, the pre-trained CNN parameters were excessively adjusted to the sequential data, which were not class-balanced. As a consequence, the model started to overfit to the majority of the sleep stages toward the end of the fine-tuning. Therefore, two different learning rates are used during fine-tuning. Also, we use a heuristic gradient clipping technique to prevent the exploding gradients, which is a well-known problem when training RNNs such as LSTMs [67]. This technique rescales the gradients to smaller values using their global norm whenever they exceed a pre-defined threshold. The sequential training set is obtained by arranging the original training set sequentially according to time across all subjects.

5.3.3 Loss Function

We use the cross-entropy loss to quantify the agreement between the predicted and the target sleep stages in both of these training steps. The combination of the softmax function and the cross-entropy loss are used to train our model to output probabilities for mutually exclusive classes.

Formally, suppose there are N input samples $\{\mathbf{h}^{(1)}, \ldots, \mathbf{h}^{(N)}\}$. These samples can be either from the

representation learning $\{\mathbf{a}^{(1)}, \ldots, \mathbf{a}^{(N)}\}$ or the sequence residual learning $\{\mathbf{o}^{(1)}, \ldots, \mathbf{o}^{(N)}\}$ depending on which training step we are in. We first compute a vector of the probability distribution over five sleep stages $\hat{\mathbf{y}}^{(i)}$ using Equation 5.7 and 5.8. Then we compute the cross-entropy loss as follows:

$$J(\theta) = -\frac{1}{N}\sum_{i=1}^{N} \log(\hat{y}^{(i)}_{y^{(i)}}) \ , \tag{5.10}$$

where $\hat{y}^{(i)}_{y^{(i)}}$ is the output probability of the correct class index, indicated by the ground truth label $y^{(i)}$. For more details about this cross-entropy loss, please go to Section 2.9.1 in Chapter 2.

5.3.4 Regularization

We employed two regularization techniques to help prevent overfitting problems.

- The first technique was Dropout [6,145] that randomly sets the input values to 0 (i.e., dropping units along with their connection) with the specified probability during training. Dropout layers with the probability of 0.5 were used throughout the model (**dropout** in Figure 5.1). It is important to note that these dropout layers were used for training only, and were removed from the model during testing to provide deterministic outputs.

- The second technique was the L2 weight decay $\lambda \|\mathbf{W}\|_2^2$, in which we adds to a loss function in Equation 5.10 as a penalty term to prevent large values of the parameters in the model. We only applied the weight decay on the first layers of the two CNNs because of the two main reasons. Firstly, it is pointed out in [67] that L2 weight decay can limit the model capabilities of learning long-term dependencies. Secondly, we found that, without weight decay, the filters of the first layers of the CNNs overfitted to noises or artifacts in EEG data. This weight decay helped the model learn smoother filters (i.e., containing less high-frequency elements) which resulted in slightly performance gains. The weight decay parameter that defines the degree of penalty, λ, was set to 10^{-3}.

For more details about the Dropout and the L2 weight decay, please go to Section 2.6 in Chapter 2.

5.4 Results

5.4.1 Data

We evaluated our model using different EEG channels from two public datasets: Montreal Archive of Sleep Studies (MASS) [146] and Sleep-EDF [128, 147].

MASS. In MASS cohort 1, there were five subsets of recordings, SS1-SS5, which were organized according to their research and acquisition protocols. We used data from SS3, which contained PSG recordings from 62 healthy subjects (age 42.5±18.9). Each recording contained 20 scalp-EEG, 2 EOG (left and right), 3 EMG and 1 ECG channels. The EEG electrodes were positioned according to the international 10-20 system, and the EOG electrodes were positioned diagonally on the outer edges of the eyes. EEG and EOG recordings were pre-processed with a notch filter of 60 Hz, and band-pass filters of 0.30-100 Hz (EEG) and 0.10-100 Hz (EOG). All EEG and EOG recordings had the same sampling rate of 256 Hz. These recordings were manually classified into one of the five sleep stages (W, N1, N2, N3 and REM) by a sleep expert according to the AASM standard [27]. There were also movement artifacts at the beginning and the end of each subject's recordings that were labeled as UNKNOWN. We evaluated our model using the F4-EOG (Left) channel, which was obtained via montage reformatting [148] without any further pre-processing.

Sleep-EDF. There were two sets of subjects from two studies: age effect in healthy subjects (SC) and Temazepam effects on sleep (ST). We used 20 subjects (age 28.7±2.9) from SC. Each PSG recording contained 2 scalp-EEG signals from Fpz-Cz and Pz-Cz channels, 1 EOG (horizontal), 1 EMG, and 1 oro-nasal respiration signal. All EEG and EOG had the same sampling rate of 100 Hz. These recordings were manually classified into one of the eight classes (W, N1, N2, N3, N4, REM, MOVEMENT, UNKNOWN) by sleep experts according to the R&K standard [26]. We evaluated our model using the Fpz-Cz and Pz-Cz channels without any further pre-processing. We also merged the N3 and N4 stages into a single stage N3 to use the same AASM standard as the MASS dataset. There were long periods of awake or stage W at the start and the end of each recording, in which a subject was not sleeping. We only included 30 minutes of such periods just before and after the sleep periods, as we were interested in sleep periods.

We excluded MOVEMENT and UNKNOWN (which were at the start or the end of the each recording) stages, as they did not belong to the five sleep stages [27]. Table 5.1 summarizes the number of 30-s

Table 5.1: Number of 30-s EEG epochs for each sleep stage from two datasets

Dataset	W	N1	N2	N3 (N4)	REM	Total
MASS	6227	4724	29534	7651	10464	58600
Sleep-EDF	7927	2804	17799	5703	7717	41950

Table 5.2: The mean (standard deviation) of number of training and test samples in each cross-validation from two datasets

Dataset	Training Samples	Test Samples
MASS	48629.0 (153.9)	1890.3 (102.3)
Sleep-EDF	34200.0 (522.0)	4195.0 (492.9)

epochs for each sleep stage from these two datasets.

5.4.2 Experimental Design

We evaluated our model using a k-fold cross-validation scheme, where k was set to 31 and 10 for the MASS and Sleep-EDF datasets respectively. Specifically, in each fold, we used recordings from $N_s - (N_s/k)$ to train the model, and from the remaining N_s/k subjects to test the trained model, where N_s is the number of subjects in the dataset. This process was repeated k times so that all of the recordings were tested. Then we combined the predicted sleep stages from all folds and computed the performance metrics, which will be discussed in Section 5.4.3. We also repeatedly performed this k-fold cross-validation on the randomly shuffled lists of subjects for 15 times on each dataset. This is to investigate whether the use of different sets of subjects to train the model, and the randomly initialized weights and biases of the model affect the sleep scoring performance.

Table 5.2 shows the mean and standard deviation of the number of training and test samples in each cross-validation fold. It can be seen that the sum of the training and the test samples were not equal to the total number of EEG epochs shown in Table 5.1. This was due to the way our two-step training algorithm iterates through mini-batch EEG epochs during training, in which the number of epochs from each subject was not divisible by the mini-batch size (see more in Section 5.4.5). Thus the number of samples we used to train our model was slightly less than the total number of training samples in each cross-validation fold. Even though we did not use all samples from the subjects in each cross-validation fold to train our model, we did use all samples from the withheld subjects to test our model.

5.4.3 Performance Metrics

We evaluated the performance of our model using per-class precision (PR), per-class recall (RE), per-class F1-score (F1), macro-averaging F1-score (MF1), overall accuracy (ACC), and Cohen's Kappa coefficient (κ) [58, 149]. The per-class metrics are computed by considering a single class as a positive class, and all other classes combined as a negative class. The MF1 and ACC are calculated as follows:

$$\text{ACC} = \frac{\sum_{c=1}^{C} \text{TP}_c}{N} \tag{5.11}$$

$$\text{MF1} = \frac{\sum_{c=1}^{C} \text{F1}_c}{C} \tag{5.12}$$

where TP_c is the true positives of class c, F1_c is per-class F1-score of class c, C is the number of sleep stages, and N is the total number of test epochs. We did not use the ROC analysis to report the classification performance as our model does not use any thresholds to classify 30-s EEG epochs. Instead, our model classifies each 30-s EEG epoch into the sleep stage that has the highest probability, obtained from the **softmax** layer.

5.4.4 Tests for Statistical Significance

One of the statistical tests that we can use to compare our proposed method with the state-of-the-art methods is **bootstrap confidence interval** [22]. This approach has already been applied in [22, 140] to compute the 95% confidence intervals (CIs) to compare their sleep scoring performance with other methods. For each dataset, we first trained and evaluated our model using the k-fold cross-validation to obtain confusion matrices of all subjects. These matrices were then added up to compute the performance metrics (**original metrics**). Next, we generated 10,000 bootstrap samples for each dataset using the confusion matrices, previously obtained from the k-fold cross-validation, of the subjects in that dataset. To generate each bootstrap sample, we randomly sampled the confusion matrices of the subjects with replacement, such that we have the same number of confusion matrices as the number of subjects in a dataset. Then we added up these resampled confusion matrices and calculated the performance metrics (**bootstrap metrics**). Once we generated 10,000 bootstrap samples and computed the performance metrics for each bootstrap sample (bootstrap metrics), we computed the *differences* between the original metrics and the bootstrap metrics. In particular, for

each performance metric, we would like to compute:

$$\delta^{(i)} = x_b^{(i)} - x_o \tag{5.13}$$

where x_o is the original metric (e.g., ACC, MF1, etc.), $x_b^{(i)}$ is the bootstrap metric computed from the i-th bootstrap sample, and $\delta^{(i)}$ is the difference between the original and the i-th bootstrap metrics. We computed such differences for all 10,000 bootstrap samples to approximate the distribution of the differences. After that we sorted these differences from smallest to biggest. The 95% CIs (i.e., the significance level is 0.05) can be approximated using the the 2.5th ($\delta_{.975}$) and 97.5th ($\delta_{.025}$) percentiles of the sorted differences (i.e., 251st and the 9750th elements in our case):

$$[x_o - \delta_{.025}, x_o - \delta_{.975}] \tag{5.14}$$

It has been suggested in [150] that the CIs approximated from the distribution of the relative variation is more robust than the distribution of the bootstrap sample statistic, where we instead sorted the bootstrap metrics and directly use the 2.5th and 97.5th percentiles of the sorted metrics to define the CIs. To test for statistical significance, for each metric, if such interval does not contain the metric from other methods, the metric from others are significantly different from ours.

Apart from the bootstrap confidence interval, there are also the other statistical tests that can be used to compare our method with the others.

- **McNemar's test [151] or 5x2 cv paired t test [152].** These two tests have been shown to have low type I error (i.e., incorrectly detecting a difference when no difference exists) when comparing two supervised learning algorithms [152]. The null hypothesis of these tests is that the two learning algorithms will have the same error rate, which is rejected if the calculated p-value is smaller than the significance level $\alpha = 0.05$ (i.e., $p < 0.05$). To test this null hypothesis, we need to construct a contingency table [151], which requires both of our method and others to be trained and tested on the same training and test sets. However, the codes to reproduce the results of the other methods are not publicly available. We believe that it can lead to a bias comparison if we re-implement their code without knowing all of the hyperparameter settings.

- **Permutation test [153].** This is a statistical technique for constructing sampling distribution, which is similar to the bootstrapping technique except that it relies on random resampling

without replacement. To perform this test, we first generate random permutations from the datasets (e.g., shuffle the list of subjects), then carry out the k- fold cross-validation. This process is repeated for, at least, 1,000 times, such that, for each permutation, we have different sets of subjects in the training and test sets to evaluate the scoring performance of our model. The null hypothesis of this test is that our method and others give the similar performance. We can apply the one-tailed test to test whether the scoring performance of our method is significantly better than others. In particular, we compute the p-value by dividing the number of permutations whose performance metrics are less than the metrics from others by the total number of permutations. If the calculated p-value is smaller than the significance level $\alpha = 0.05$ (i.e., $p < 0.05$), the null hypothesis is rejected. However, it would be too computationally expensive and time-consuming to train our deep learning model for 1,000 times. This is because it took approximately three hours (see Section 5.4.6) to train the model in each cross-validation fold and we had to train model for $k \times 1,000$ times for each dataset.

Therefore, according to the limitations of the other statistical tests mentioned above, we can only use the bootstrap confidence interval technique to compare our method with the state-of-the-art ones in this study. This is because we only need to run the k-fold cross-validation one times, and use the bootstrapped confusion matrices to compute the confidence intervals. We leave the comparison with the McNemar's test, the 5x2 cv paired t test and the permutation test as our future work.

5.4.5 Training Parameters

The representation learning part was pre-trained using the oversampled training set with the mini-batch size of 100. The Adam optimizer's parameters lr, $beta1$, and $beta2$ were set to 10^{-4}, 0.9 and 0.999 respectively. Then the whole model was fine-tuned using the sequential training set. Specifically, we equally split the sequences of 30-s EEG epochs from each subject data into 10 sub-sequences (i.e., batch size was 10). Then we fed 25 epochs (i.e., sequence length was 25) from each sub-sequence yielding 250 epochs per one step training. The Adam optimizer's parameters were similar to the pre-training step except that the learning rate of each part of the model, $lr1$ and $lr2$, were set to 10^{-6} and 10^{-4} respectively. The threshold of the gradient clipping was set to 10. The numbers of epochs for the pre-training and the fine-tuning steps were set to 100 and 200 respectively. There was no early stopping as there was no validation set in our evaluation scheme.

For the batch normalization in **conv** and **fc** layers, the ϵ constant of 10^{-5} was added to the mini-batch variance for numerical stability. The mean and variance of the training set, which were used as fixed parameters during testing, were estimated by computing the moving average of with a decay rate of 0.999 from the sampling mean and variance of each mini-batch.

5.4.6 Implementation

We implemented our model using TensorLayer (https://github.com/zsdonghao/tensorlayer), which is a deep learning library extended from Google Tensorflow [154]. This library allows us to deploy numerical computation such as the training and validation tasks to multiple CPUs and GPUs. We ran the k-fold cross-validation using the eTRIKS Analytical Environment (eAE) (https://eae.doc.ic.ac.uk/), which provides a cluster of high-performance computing nodes. Each node was equipped with an NVIDIA GeForce GTX 980. The training time for each validation fold was approximately 3 hours on each node. The testing or prediction time for each batch of 25 EEG epochs (according to the sequence length specified during training in Section 5.4.5) was approximately 50 milliseconds on each node. Our code is publicly available at https://github.com/akaraspt/deepsleepnet.

5.4.7 Initial Experiments

We initially conducted experiments with the first fold of the 31-fold cross-validation with the MASS dataset. This was to design the architecture and the hyperparameters for DeepSleepNet. For model architecture, we tried several configurations such as increasing/decreasing convolutional layers, changing the number of filters and the stride sizes, and changing the number of hidden sizes in the bidirectional-LSTMs and the fully-connected layer. The architecture in Figure 5.1 gave us the best performance. For regularization parameters, we tried several values for the weight decay parameters ranging from 10^{-1} to 10^{-5}. The value of 10^{-3} gave us the best performance. For training parameters, we tried several values of learning rates ranging from 10^{-3} to 10^{-8}. We also experimented with the mini-batch size (from 50 to 200) during the pre-training, the batch size (from 5 to 40) and sequence length (from 5 to 40) during fine-tuning. Other parameters such as $beta1$, $beta2$ and the threshold of the gradient clipping were chosen from the default values reported in the literature. The training parameters mentioned in Section 5.4.5 gave us the best performance. With these settings, the pre-training and fine-tuning steps started to converge after 100 and 200 epochs respectively.

Table 5.3: Confusion matrix obtained from 31-fold cross-validation on F4-EOG (Left) channel from the MASS dataset. PR, RE and F1 are per-class precision, per-class recall and per-class F1-score respectively.

	Predicted					Per-class Metrics		
	W	N1	N2	N3	REM	PR	RE	F1
W	**5433**	572	107	13	102	87.3	87.2	87.3
N1	452	**2802**	827	4	639	60.4	59.3	59.8
N2	185	906	**26786**	1158	499	89.9	90.7	90.3
N3	18	4	1552	**6077**	0	83.8	79.4	81.5
REM	132	356	533	1	**9442**	88.4	90.2	89.3

Table 5.4: Confusion matrix obtained from 10-fold cross-validation on Fpz-Cz channel from the Sleep-EDF dataset. PR, RE and F1 are per-class precision, per-class recall and per-class F1-score respectively.

	Predicted					Per-class Metrics		
	W	N1	N2	N3	REM	PR	RE	F1
W	**6614**	745	181	81	306	86.0	83.4	84.7
N1	295	**1406**	631	30	442	43.5	50.1	46.6
N2	391	618	**14542**	1473	775	90.5	81.7	85.9
N3	29	9	291	**5370**	4	77.1	94.2	84.8
REM	360	457	419	7	**6474**	80.9	83.9	82.4

5.4.8 Sleep Stage Scoring Performance

Table 5.3 and Table 5.4 show confusion matrices obtained from the 31-fold and the 10-fold cross-validation on the F4-EOG (Left) and the Fpz-Cz channels from the MASS and Sleep-EDF datasets respectively. We did not include the confusion matrix obtained from the Pz-Oz channel from the Sleep-EDF dataset as the Fpz-Cz channel gave a better performance. Each row and column represent the number of 30-s EEG epochs of each sleep stage classified by the sleep expert and our model respectively. The numbers in bold indicate the number of epochs that were correctly classified by our model. The last three columns in each row indicate per-class performance metrics computed from the confusion matrix.

It can be seen that the poorest performance was noted for the stage N1, with the F1 less than 60, while the F1 for other stages were significantly better, with the range between 81.5 and 90.3. Most of the misclassified stages were between N2 and N3. It can also be seen that the confusion matrix is almost symmetric via the diagonal line (except the pair of N2-N3). This indicates that the misclassifications

were less likely to be due to the imbalance-class problem.

Figure 5.4 demonstrate examples of hypnograms that were manually scored by a sleep expert, and automatically scored by our DeepSleepNet for Subject-1 from the MASS dataset.

Table 5.5 shows the mean and standard deviation of performance metrics obtained from repeatedly performing the 31-fold and the 10-fold cross-validations on the randomly shuffled lists of subjects for 15 times on the MASS and Sleep-EDF datasets. It can be seen that the variances (i.e., standard deviation) of the metrics were relatively low, suggesting that our model can achieve similar scoring performances regardless of which set of subjects were used to train the model. It should be noted that the confusion matrices in Table 5.3 and Table 5.4 were from the model that gave the median scoring performance on each dataset. This model will also be used to compare with the other state-of-the-art methods in the next section.

5.4.9 Comparison with State-of-the-Art Approaches

Table 5.6 shows a comparison between our method and other sleep stage scoring methods across ACC, MF1, κ and F1. These methods include the ones that utilize hand-engineered features [22,138,139,141], CNNs only [140] or LSTMs only (which is our previous work) [23]. The other methods' metrics were computed using the confusion matrices reported in their papers. We classified the methods into two groups: *non-independent* and *independent* training and test sets. The non-independent ones were the methods that included parts of the test subjects' epochs in the training data, while the independent ones were the methods that excluded all epochs of the test subjects from the training data. We believe that the practical evaluation scheme should not include any epochs from the test subjects. Also, it has been shown that the non-independent scheme resulted in an improvement of the performance [22]. Thus we did not compare the performance of our method with the non-independent group. The numbers in bold indicate the highest performance metrics of all methods in each dataset of each group.

Among the methods in the independent group, it can be seen that our method either achieved a better overall scoring performance or a similar performance compared to the state-of-the-art methods that use the same EEG channel on the Sleep-EDF and MASS dataset, respectively. In the case of the Sleep-EDF dataset, the lower bounds of the 95% CIs of both the ACC and MF1 (the top number in parentheses) was higher than the other methods, which was statistically significant. Such improvement

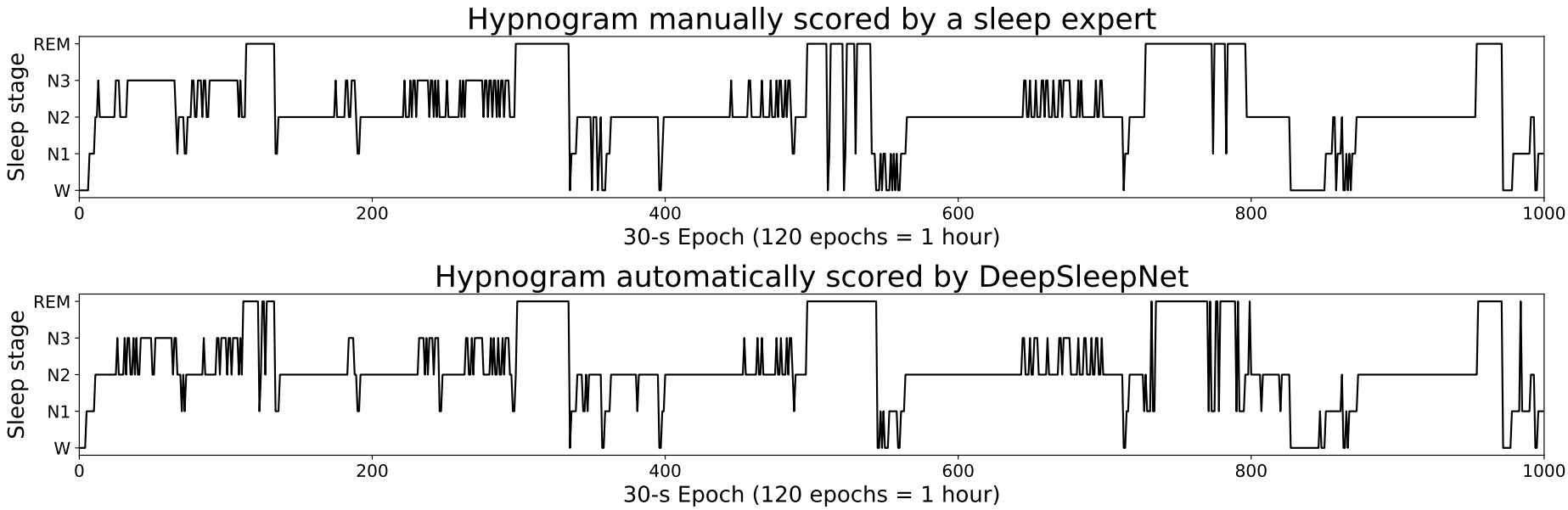


Figure 5.4: Examples of the hypnogram manually scored by a sleep expert (top) and the hypnogram automatically scored by DeepSleepNet (bottom) for Subject-1 from the MASS dataset.

Table 5.5: The mean (standard deviation) of performance metrics obtained from repeatedly performing the 31-fold and the 10-fold cross-validation on the randomly shuffled lists of subjects for 15 times on the MASS and Sleep-EDF dataset. The per-class precision (PR) and recall (RE) are not shown here as the per-class F1-score (F1) is a summary of the precision and recall using a harmonic mean.

Dataset	EEG Channel	Overall Metrics			Per-class F1-Score (F1)				
		ACC	MF1	Cohen	W	N1	N2	N3	REM
Sleep-EDF	Fpz-Cz	82.1 (0.5)	76.8 (0.6)	0.76 (0.0)	85.3 (0.8)	46.2 (1.5)	85.6 (0.4)	83.9 (0.8)	83.1 (0.6)
MASS	F4-EOG (Left)	86.1 (0.2)	81.6 (0.3)	0.80 (0.0)	87.4 (0.3)	59.7 (0.3)	90.2 (0.2)	81.0 (1.0)	89.5 (0.2)

Table 5.6: Comparison between DeepSleepNet and other sleep stage scoring methods that utilizes hand-engineering features across overall accuracy (ACC), macro-F1 score (MF1), Cohen's Kappa (κ), and Per-class F1-Score (F1). We only compare the scoring performance with the methods that evaluated using the independent training and test sets. The bootstrap 95% confidence intervals (CIs) of the performance metrics are calculated for the Fpz-Cz channel from the Sleep-EDF dataset and the F4-EOG (Left) channel from the MASS dataset. The top and bottom numbers in parentheses are the lower and upper bounds of the CIs respectively.

Methods	Dataset	EEG Channel	Test Epochs	Overall Metrics			Per-class F1-Score (F1)				
				ACC	MF1	κ	W	N1	N2	N3	REM
Non-independent Training and Test Sets											
Ref. [141]	Sleep-EDF	Fpz-Cz	960	90.3	76.5	-	77.3	46.5	**94.9**	72.2	**91.8**
Ref. [138]	Sleep-EDF	Pz-Oz	15136	**91.3**	77	**0.86**	**97.8**	30.4	89	**85.5**	82.5
Ref. [139]	Sleep-EDF	Pz-Oz	7596	90.8	**80**	0.85	96.9	**49.1**	89	84.2	81.2
Independent Training and Test Sets											
Ref. [22]	Sleep-EDF	Fpz-Cz	37022	78.9	73.7	-	71.6	**47.0**	84.6	84.0	81.4
Ref. [140]	Sleep-EDF	Fpz-Cz	37022	74.8	69.8	-	65.4	43.7	80.6	**84.9**	74.5
DeepSleepNet	Sleep-EDF	Pz-Oz	41950	79.8	73.1	0.72	**88.1**	37	82.7	77.3	80.3
				(79.3)	(74.2)		(79.8)	(41.5)	(83.3)	(81.0)	(78.6)
DeepSleepNet	Sleep-EDF	Fpz-Cz	41950	**82.0**	**76.9**	**0.76**	84.7	46.6	**85.9**	84.8	**82.4**
				(84.9)	(79.9)		(90.5)	(52.4)	(89.1)	(89.3)	(86.9)
Ref. [23]	MASS	F4-EOG (Left)	59066	85.9	80.5	-	84.6	56.3	**90.7**	**84.8**	86.1
				(85.3)	(80.6)		(84.8)	(56.6)	(89.6)	(79.2)	(87.9)
DeepSleepNet	MASS	F4-EOG (Left)	58600	**86.2**	**81.7**	**0.80**	**87.3**	**59.8**	90.3	81.5	**89.3**
				(87.1)	(82.9)		(89.9)	(61.6)	(91.1)	(86.2)	(90.2)

was mainly due to an increase in the scoring performance on the sleep stage W, without sacrificing the performance on the other stages. It should also be noted that our model performed better when applied to the Fpz-Cz channel compared to the Pz-Oz, which is similar to [22]. Thus we did not calculate the 95% CIs for the comparison. In the case of the MASS dataset, only the lower bound of the 95% CIs of the MF1 was higher than the other method, not ACC. This might be due to a slight decrease in the performance on the stage N3 (not statistically significant), even though the performance on the stage W and N1 were statistically significantly higher than the other. Thus we concluded that our model achieved the similar scoring performance compared to the other on the MASS dataset, as both the ACC and MF1 were not statistically significantly higher than the other.

It should be emphasized that our model achieved such scoring performance without sacrificing the performance on the stage N1, which is the most difficult sleep stage to classify. This indicates that our method was not biased by favoring the majority of the sleep stages than the minority ones. Also, according to the κ coefficient, it showed that the agreement between the sleep experts and our model were substantial (between 0.61 and 0.80) [139].

5.4.10 Sequence Residual Learning

We performed additional experiments to verify the important of the sequence residual learning part with the MASS dataset. Table 5.7 shows a confusion matrix obtained from 31-fold cross-validation on the F4-EOG (Left) channel using DeepSleepNet without the sequence residual learning part (i.e., using the *pre_model* in Algorithm 5.1). It can be seen that the F1 of all sleep stages, except the stage N3, were lower than the ones in Table 5.3. This was because of an increase in the misclassifications between the pairs of N1-N2, N2-N3 and N1-REM. This may well be due to the effects of oversampling the training set to have balanced-class samples. As a consequence the model tended to predict more of stages N1 and N3. These results indicated that the process to stack the pre-trained representation learning part with the sequence residual learning part, and then fine-tune the both parts with sequential training set helped improve the classification performance.

5.4.11 Model Analysis

To better understanding how our model classified a sequence of 30-s EEG epochs, we analyzed and compared: 1) the learned filters at the first convolutional layers of the two CNNs in the representation

Table 5.7: Confusion matrix obtained from 31-fold cross-validation on the F4-EOG (Left) channel from the MASS dataset using DeepSleepNet without Sequence Residual Learning

	Predicted					Per-class Metrics		
	W	N1	N2	N3	REM	PR	RE	F1
W	**5215**	709	94	19	190	84.5	83.7	84.1
N1	468	**2582**	747	11	916	40.8	54.7	46.8
N2	241	1846	**24140**	2435	872	93.4	81.7	87.2
N3	19	3	472	**7156**	1	74.3	93.5	82.8
REM	227	1181	383	5	**8668**	81.4	82.8	82.1

learning part; and 2) the memory cells inside the bidirectional-LSTMs in the sequence residual learning part. This analysis was carried out with the MASS dataset across 31 cross-validation folds.

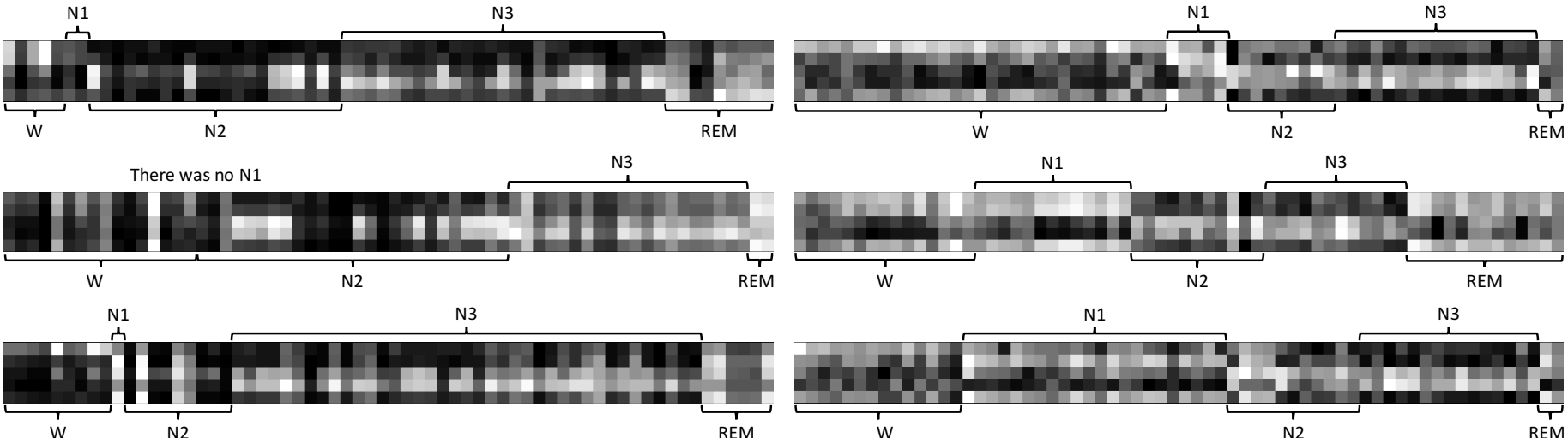
CNN Filters

We analyzed how our model utilized the learned filters of the first **conv** layers of the CNN-small and CNN-large in the representation learning part. In particular, we determined which filters were most active for each sleep stage by computing the average of the sum of the activations of all filters across samples of each sleep stage. These activations were the outputs of the first **conv** layers (i.e., after applying the ReLU activation function).

Formally, suppose there were N 30-s EEG epochs from a subject in a validation fold $\{\mathbf{x}^{(1)}, ..., \mathbf{x}^{(N)}\}$. We fed these epochs to our model to obtain activations from the first **conv** layer of each CNN: $\{\mathbf{z}^{(1)}, ..., \mathbf{z}^{(N)}\}$, where $\mathbf{z}^{(i)} \in \mathbb{R}^{p \times q}$, p and q are the activation output size and the number of filters of the first **conv** layer. The average of the sum of the activations of the filter k for the sleep stage c is computed as follows:

$$u_{c,k} = \frac{\sum_{i=1}^{N_{y_{pred}=c}} \sum_{j=1}^{p} z_{j,k}^{(i)}}{N_{y_{pred}=c}} \tag{5.15}$$

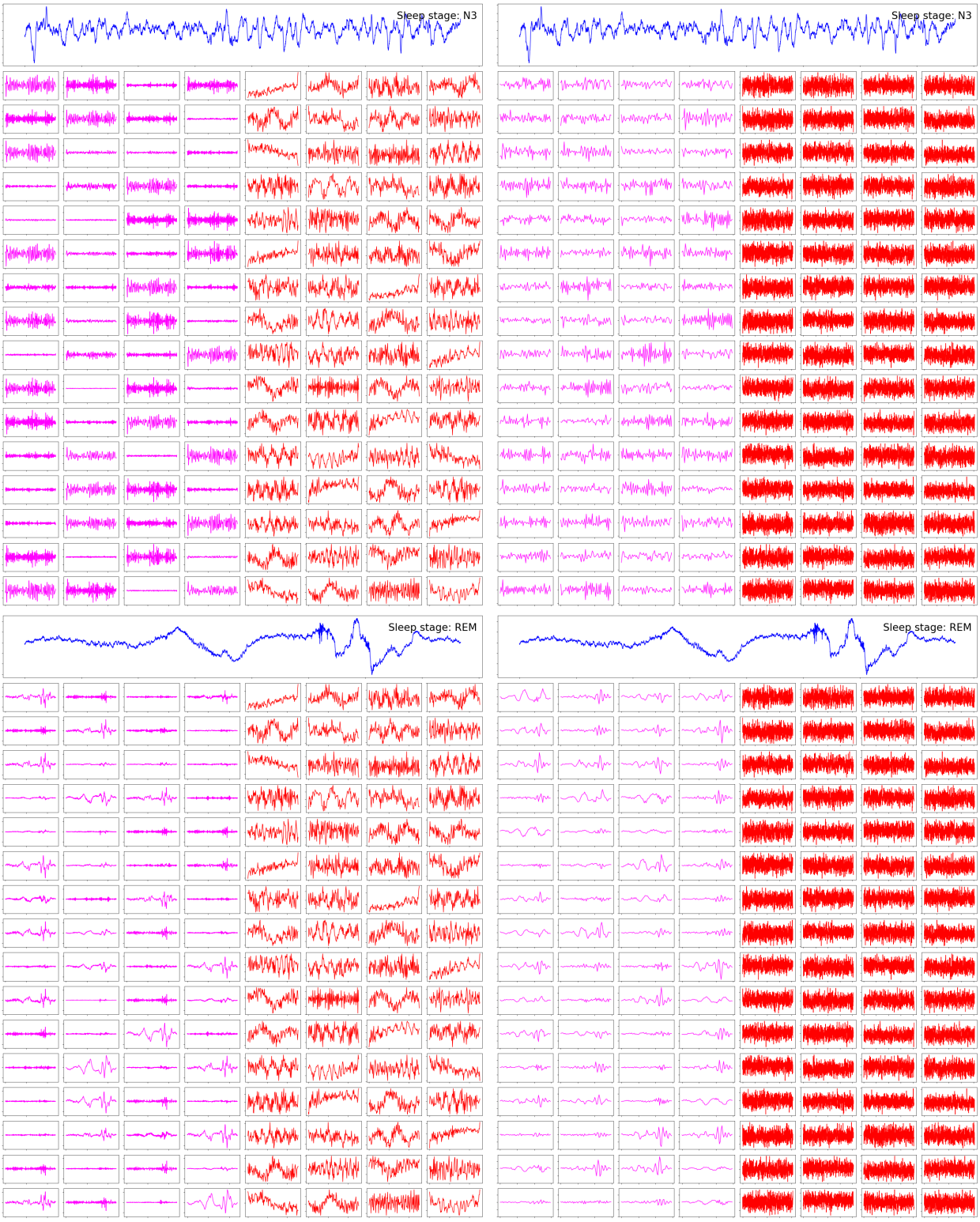
where $u_{c,k}$ is the average of the sum of the activation of the filter k for sleep stage c, $z_{j,k}^{(i)}$ is the j-th index of the activation vector $\mathbf{z}^{(i)}$ of the filter k, and $N_{y_{pred}=c}$ is the number of EEG epochs that our model predicted as stage c. After we computed the $u_{c,k}$ of all filters for sleep stage c, we rescaled them into a range of 0 and 1. We denote this scaled k-dimension vector $\mathbf{u}_c$ as **filter activations** for stage c. This process was repeated for all c sleep stages. Once we got the filter activations from all sleep stages, we stacked them together and rearranged the order of the filters such that the filters that were



(a) Filter activations from the small filters

(b) Filter activations from the large filters

Figure 5.5: Examples of the average of the sum of the filter activations from the first **conv** layers of the CNN-small and CNN-large obtained by feeding our model with data from 3 subjects. The filter activations from the small filters are on the left (a), and the larger filters are on the right (b). Each image has 5 rows and 64 columns, corresponding to 5 sleep stages and 64 filters respectively. Each pixel represents the scaled value of $u_{c,k}$ from Equation 5.15, where 1 (i.e., active) is white and 0 (i.e., inactive) is black. Each row corresponds to the 64-dimension vector (i.e., k is 64) for each sleep stage c. The first row is from stage W, and the last row is from stage REM. Each image also has labels indicating which filters are most active for which sleep stages. It can be seen that, for instance, some filters were most active for stage N2-N3, some were most active for stage W-N1-REM, and some were either most active or inactive at all stages.



(a) Small filters **(b)** Large filters

Figure 5.6: Outputs of 1D-convolution using the filters from the first **conv** layers from the CNN-small and CNN-large. Here, we illustrate the outputs (magenta signals) from convolving the raw input 30-s EEG epochs from sleep stage N3 and REM (blue signals) with the corresponding filters (red signals) (see Figure 5.2 for more details). These EEG epochs from the F4-EOG (Left) channel from one subject in the MASS dataset. We found that most of the small and large filters were mixed frequency signals, and some of them are similar. Also, some of the small filters exhibit either an increasing or decreasing trends, which produce opposite trend of output signals.

most active for each sleep stage were grouped together (see Figure 5.5). Also, we visually inspected the outputs of the 1D-convolution (before batch normalization and ReLU function), and the filters used to convolve with the input 30-s EEG epochs from the first **conv** layers to produce such outputs (see Figure 5.6). This is to understand which patterns (i.e., the frequency components) of the EEG signals the first **conv** layers were trying to detect.

According to our analysis of the learned filters of the first **conv** layers of the CNN-small and CNN-large in the representation learning part, we found that the number of filters that were most active for each sleep stages varied across subjects in both datasets. We also found that the patterns or the frequency components of the filters that were most active different sleep stages also varied across subjects. These preliminary results suggested that these layers did not learn any filters that were dedicated to detecting certain patterns appeared in each sleep stage. They instead used a combination of the filters to detect the features that are useful for sleep stage scoring. The further analysis of how the CNNs utilizes the filters to extract features is beyond the scope of this study. We leave this analysis as our future work.

LSTM Cells

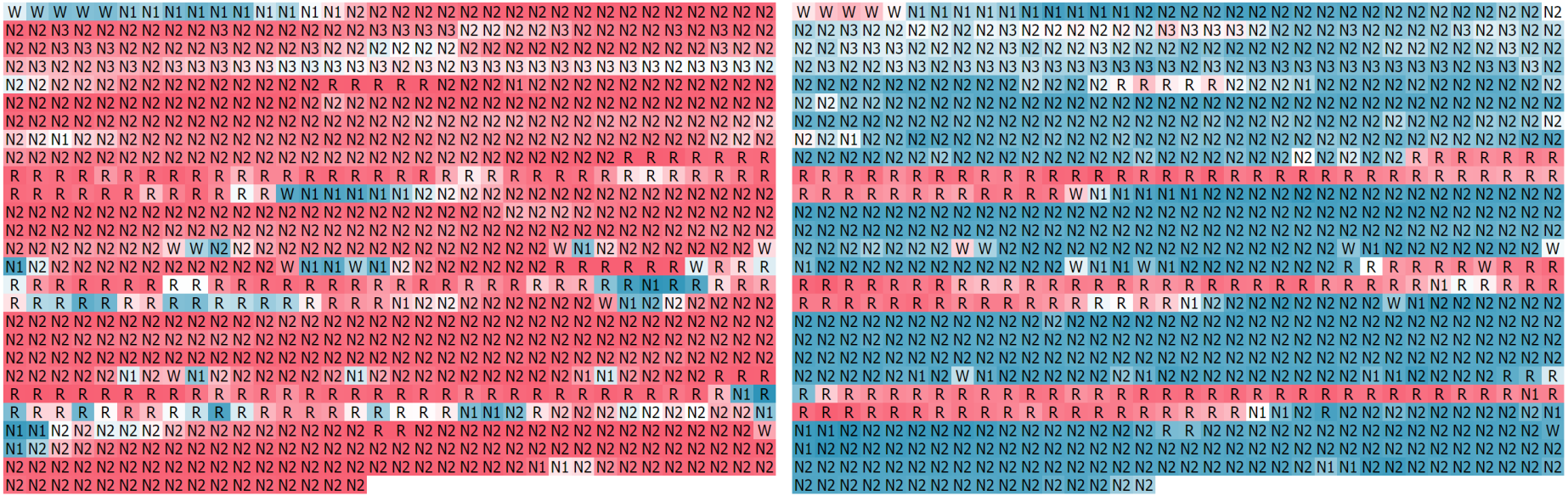
Secondly, we analyzed how our model utilized the bidirectional-LSTMs (i.e., **bidirect-lstm** layers) to learn the temporal information from a sequence of EEG epochs. Specifically, we investigated how the bidirectional-LSTMs managed their **cell states** or what is called **LSTM cells** (i.e., $\mathbf{c}_f$ and $\mathbf{c}_b$ in Equation 5.4 and Equation 5.5 respectively).

Recall that our sequence residual learning part contains two **bidirect-lstm** layers, each consists of two LSTMs that process the input sequence in the forward and backward directions (see Figure 5.3). Each LSTM maintains two states: cell $\mathbf{c}^{(t)}$ and hidden $\mathbf{h}^{(t)}$ states at every time step. At each time step t, the LSTM updates the cell state using the gating mechanism (more details can be found in Section 2.9.5). The cell state is then squashed by tanh function before being selectively exposed to the current hidden state. The hidden state is what the LSTM forwards to the downstream layers. What we are interested in is how the LSTM updates the cell state at each time step, as the cell state is an internal memory that the LSTM uses to remember long-term information of the input sequences it is currently processing. Thus we visually inspected each LSTM cell as our model processes the input sequences.

To visualize the LSTM cells, we apply the technique proposed in [155]. Suppose there were a sequence of N 30-s EEG epochs from a subject in a validation fold $\{\mathbf{x}^{(1)}, ..., \mathbf{x}^{(N)}\}$. We fed these epochs to our model to obtain the sequence of cell states from each **bidirect-lstm** layer $\{\mathbf{c}_{f|b}^{(1)}, ..., \mathbf{c}_{f|b}^{(N)}\}$ (see Equation 5.4 and 5.5), and the sequence of predicted sleep stages $\{\text{sleep stage}^{(1)}, ..., \text{sleep stage}^{(N)}\}$ (see Equation 5.9). Then we squashed the cell states to be in a range of -1 and 1 using tanh function. For each LSTM cell j, we plot the sequence of the predicted sleep stage $\text{sleep stage}^{(t)}$ for all time steps t, colored using the values from the j-th index of the squashed cell state (i.e., $\tanh(\mathbf{c}_{f|b}^{(t)})_j$). The j-th index of the cell state $\mathbf{c}_{f|b}^{(t)}$ can be illustrated as the j-th node in the $\mathbf{c}_{f|b}^{(t)}$ in Figure 5.3.

According to our visual inspection, we found several LSTM cells with interpretable activations in the LSTM-forward (see Figure 5.7). For instance, several cells mostly turned on when the model predicted that a subject was in stage W or N1, resetting their values to positive values (i.e., active) and then decay with each stage to the negative values (i.e., inactive) when the model predicted other stages. These cells can be helpful for keeping track of when the subject is awake or the sleep onset. Some cells became inactive for a long sequence of particular stages, especially N2, N3, and REM. Each of these cells decreased its value each time the model predicted the stage that it was tracking. This cell might be used by the model to specify the certainty of the next stage being the same as the previous one. Apart from these interpretable LSTM cells, we found that a large portion of the cells from both LSTM-forward and LSTM-backward had activations that were not easily interpretable (see Figure 5.8). We also found that these cells did exist across subjects on both sleep datasets.

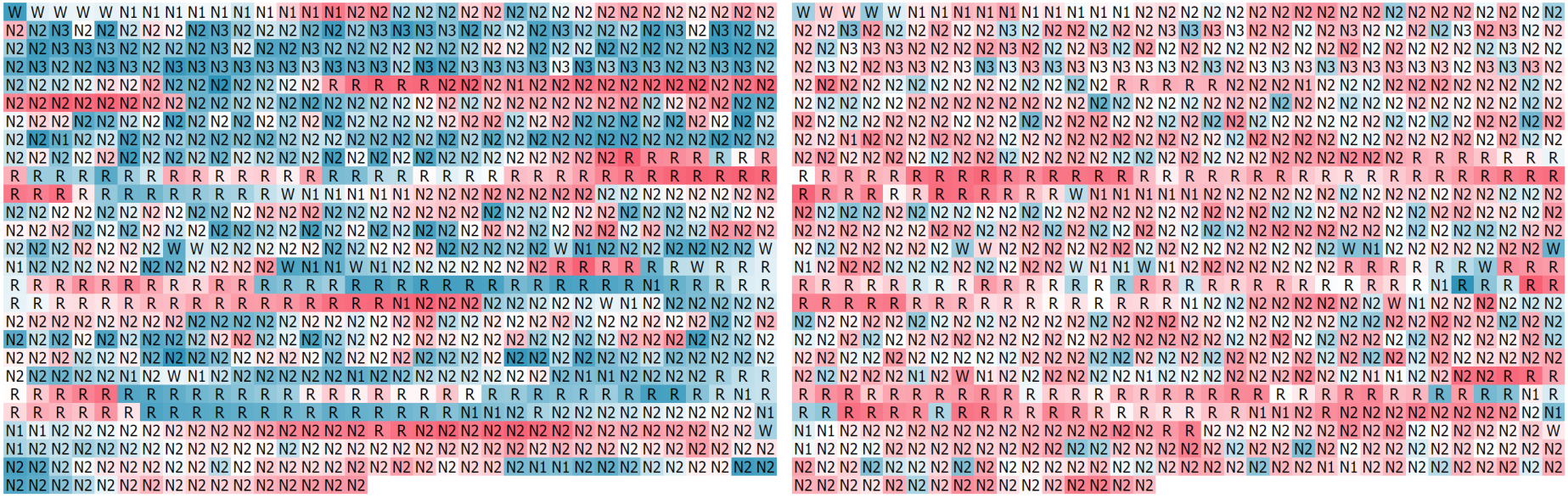
This information from these cells is important for our model to correctly identify the next sleep stages, which is similar to what the sleep experts use for scoring sleep stages according to the transition rules [27]. In particular, the experts consider the previous stages of the PSG epochs they have scored so far when they are classifying the next sleep stage. For instance, suppose the previous epoch is scored as stage REM. The experts have to continue to score the current epoch as stage REM, even in the absence of rapid eye movements (which is the main characteristic of this stage), if the EEG continues to show low amplitude, mixed frequency without K complexes or sleep spindles (see Rule 7.B, page 28 in the AASM manual [27]). In this case, the experts need to know whether they have previously scored the previous epochs as the stage REM when the rapid eye movement is not present in the current PSG epoch so that they can continue to score the current epochs as stage REM. There are also other similar rules for the other stages in the AASM manual that guide the experts to continue to score a particular stage even in the absence of its main feature if certain conditions are met.



(a) Cell that mostly activates for stage W or N1.

(b) Cell that turns off (inactive) for trains of REM stages

Figure 5.7: Interpretable LSTM cells. These two plots show the same sequence of sleep stages (capital characters), which are the the predictions from our model on one subject data, arranged through time from-left-to-right and from-top-to-bottom. Each stage in the sequence is colored in background using the corresponding value of one node (i.e., one LSTM cell) from $tanh(\mathbf{c})$, where +1 is blue and -1 is red. In **(a)**, this LSTM cell mostly activates (blue color) when our model predicts stage W or N1. In **(b)**, this cell turns off or becomes inactive (red color) when our model outputs a long sequence of REM stage (i.e., R in the plot). This suggests that the LSTMs inside the sequence residual learning part learn to keep track of the current status of a subject.



(a) A cell from LSTM-forward

(b) A cell from LSTM-backward

Figure 5.8: A large portion of cells from LSTMs, that process the input sequences in the forward (a) and backward (b) directions, are not easily interpretable. Note that these LSTM cells are from the same subject as in Figure 5.7.

The existence of these interpretable cells suggests that the LSTMs inside the sequence residual learning part learn to keep track of the current status of a subject. This also shows that our model learns to formulates its own transition rules using the information from these cells, which is, to some extent, consistent to the transition rules that the experts use to score PSG epochs.

5.5 Discussion

We propose the DeepSleepNet model that utilizes CNNs and bidirectional-LSTMs to automatically learn features for sleep stage scoring from raw single-channel EEGs without using any hand-engineered features. The results showed that, without changing the model architecture and the training algorithm, the model could be applied in different EEG channels: F4-EOG (Left), Fpz-Cz and Pz-Oz from two sleep datasets, which have different properties such as sampling rate and scoring standards (AASM and R&K). The model can achieved a better sleep scoring performance on the Sleep-EDF dataset and a similar performance on the MASS dataset compared to the state-of-the-art methods. The results also showed that the temporal information learned from the sequence residual learning part helped improve the classification performance. These demonstrated that our model could automatically learn features for sleep stage scoring from different raw single-channel EEGs.

There are two main reasons that we evaluated our model with the F4-EOG (Left) channel from the MASS dataset, which is different from most of the existing methods reported in the literature that rely on the electrodes at the central lobe such as Cz, C4 and C3. The first reason is to compare the scoring performance with our previous hand-engineering approach. The second reason is that it is much easier and more comfortable to collect data either at sleep clinics or from home environment compared to the existing methods. This is because both of the electrodes do not have problems of reading the electrical activity from the hairy scalp. Even though the F4-EOG (Left) channel does not have information from the central and occipital lobes as recommended in the AASM manual [27], our results showed that our model was still able to achieve a similar performance compared to the state-of-the-art methods.

According to the results from our model analysis, we found that our model learned several interesting features that were consistent with the AASM manual, which is one of the sleep manuals the experts followed to score PSG epochs. In the representation learning part, we found that the first convolutional layers of the CNN-small and CNN-large learned mixed-frequency filters that can be used to detect

features in more than one sleep stages. For instance, some filters mostly activated for stage N2-N3 and W-N1-REM. This implies that our model recognized some patterns that are similar among such stages. Our model might learn the filters that can be combined to detect K complexes or sleep spindles that appear in stage N2 and N3, and to detect different features of the eye movements from the EOG (Left) that can be used to distinguish among W, N1 and REM stages. In the sequence residual learning part, we found some LSTM cells with interpretable activations in the bidirectional-LSTM layers. For instance, the cells that were keeping track of the wakefulness or the sleep onset, and the cells that our model might use to specify the certainty of the next stage being the same stage as the previous one. The existence of these cells suggests that our model utilized a combination of these cells to understand the current status of a subject and to formulate its own transition rules, which is, to some extent, consistent with the criteria in the AASM manual. For instance, our model might remember that the subject was now awake (stage W) so that the probability of the next stage being either stage W or N1 is higher than stage N2, N3, and REM. Our model might remember the stage of the previously scored epoch to assist its scoring decision on the current epoch, which is typically the case for stage N2, N3 and REM when their main characteristics (e.g., rapid eye movement for stage REM) are not present in the current epoch. It should be emphasized that our model can learn these features from raw single-channel EEG without utilizing any hand-engineered features. We also observed that the features that our model learned were consistent across subjects from different cross-validation folds in both sleep datasets. Therefore, we believe that our proposed DeepSleepNet model is a better approach to implement automatic sleep stage scoring system compared to the hand-engineering ones that require prior knowledge to design feature extraction algorithms.

Even though our results are encouraging, our model is still subject to several limitations. Firstly, our model requires being trained with a sufficient amount of sleep dataset. This is due to the nature of the deep learning techniques that require a significant amount of training data to learn useful representations from the data. We performed additional experiments with the MASS dataset to estimate the number of epochs required to train our DeepSleepNet. We tried different numbers of folds (i.e., k) for the k-fold cross-validation from 2 to 31. We found that the scoring performance started to drop when k was less than 12, or when the number of training epochs was approximately less than 54000. Secondly, as our model learns features from the training data, it might not perform well when the trained model is applied to the data that have properties different from the training data such as data from different EEG channels. The model might have to be re-trained or fine-tuned before it can be applied to the data with different properties. Thirdly, as our model utilizes

bidirectional-LSTMs, the model has to wait until it has collected enough 30-s EEG epochs (depending on the sequence length of the EEG epochs used during the training process) before it can score these epochs. For instance, when the sequence length is set to 25, the model has to wait for 30×25 seconds (or 12.5 minutes) before it can identify sleep stages for these 25 EEG epochs. Lastly, we acknowledged that we only performed the statistical test using the bootstrap confidence interval approach in this study, which did not compare the scoring performance on the *same* training and test sets as in the McNemar's test or the 5x2 cv t test. This is mainly because the codes to reproduce the results of other methods are not publicly available. We leave this comparison as our future work when the codes of other methods have become publicly available. We also publish our code to verify the results of this study online (see Section 5.4.6), which can also be used to compare with other methods.

5.6 Conclusion

We propose a deep learning model, named DeepSleepNet, for automatic sleep stage scoring based on raw single-channel EEG without utilizing any hand-engineered features. Our model utilizes CNNs to extract time-invariant features, and bidirectional-LSTMs to learn stage transition rules among sleep stages from EEG epochs. We also implement the two-step training algorithm that pre-trains our model with the oversampled dataset to alleviate class-imbalance problems, and fine-tunes the model with the sequences of EEG epochs to encode the temporal information into the model. Our results showed that, without changing the model architecture and the training algorithm, our model was able to automatically learn features for sleep stage scoring from different raw single-channel EEGs from two datasets that have different properties and scoring standards. Our model analysis results also demonstrated that our model learned several features that are, to some extent, consistent with the AASM manual. As our model automatically learn features from raw EEG, we believe that DeepSleepNet is a better approach to realize a remote sleep monitoring compared to the hand-engineering ones.

This work has moved us one step closer to the realization of the remote sleep monitoring from home environment, which is important for sleep analysis in healthy people, elder people and people with sleep disorders. In the future, we plan to improve our DeepSleepNet to be able to apply on the single-channel EEG such as the F4-EOG (Left) and Fp2-EOG (Left) collected from wearable devices.

In this chapter, we show that if the biosignals are properly labeled or annotated with the well-define

and consistent criteria (unlike epileptic seizures in Chapter 4), deep learning can be trained in the supervised setting to automatically extract meaningful features without utilizing any hand-engineering features. In this next chapter, we will discuss our method proposed to detect desynchronization as anomaly in the unsupervised setting inspired from Chapter 4.

6

Towards Desynchronization Detection

Combining the knowledge we have learned from the previous chapters:

1. In Chapter 4, we found that it is difficult for deep learning model trained in the unsupervised setting to utilize information from biosignals from one domain (e.g., EEG) to detect epileptic seizures as anomalies via the signal reconstruction error. It may be better if we utilize biosignals from multiple domains to detect desynchronization as an anomaly as the model can use other signals as references for the expected normal behaviors.

2. In Chapter 5, we found that the convolutional layers are a better choice if we would like to increase the input dimension across time and channels of the model. We demonstrate that our DeepSleepNet model trained in supervised setting can automatically learn useful features for sleep stage scoring. The remaining challenge would be to develop a model that can be trained in the unsupervised setting to provide meaningful information such as anomalies.

In this chapter, we propose a novel data-driven approach that utilizes a deep learning model to detect *desynchronizations* among biosignals from multiple domains as *anomalies* without utilizing any annotations (i.e., *unsupervised* learning). We demonstrate its usefulness with the application of detecting eye-movements using unlabeled EEG and EOG signals.

6.1 Introduction

Estimating degree of synchrony or interdependence between signals is a common step in the analysis of biological data [27, 30, 107]. It can provide new insights into the functioning of the physiological systems that produce such signals.

In addition to the measure of synchrony, detecting *desynchronization* among simultaneous signals can also provide useful information about the status of our physiological systems. For instance, healthy subjects normally show a sign of synchronized behavior across different biosignals such as body temperature, heart rate, respiratory rate, and electrophysiological measure. Being able to detect desynchronization across modalities may be used as an early sign of harmful diseases. Desynchronization in this context can be thought of as *anomalous patterns of simultaneous signals*.

To detect desynchronization, ones may utilize the existing techniques that assess the interdependence between simultaneous signals to estimate their degree of synchrony. For instance, Pearson's R coefficient [104] can measure the linear correlation between two signals. Mutual information [156] can test for the amount of information obtained from one signal through another as a sign of their relationship. Phase synchronization [30, 157] can measure the existence of interdependence between the phases of the signals regardless of whether their amplitudes are correlated. Once the degree of synchrony has been estimated, a threshold can be used to detect desynchronization among simultaneous signals.

However, the main purpose of the techniques above is to discover and quantify the linear and non-linear relationships among signals. Without prior knowledge, it is not straightforward to interpret and justify when the signals are desynchronized. Also, there may be non-trivial mappings between biosignals from multiple modalities that can cause commonly used synchronization methods to miss important relationships among the signals. It would be better if such synchronized patterns or relationships can be learned automatically from the simultaneous signals.

In this chapter, we propose to a novel data-driven approach to detect desynchronization among biosignals from two domains. Our approach is to train a model to learn the relationships between biosignals from two domain by transcribing signal from one domain into their expected, simultaneous or synchronized signal in another domain. This cross-domain transcription model can provide an interpretable way to detect desynchronization. In particular, if the measured simultaneous signals are different from the expected, simultaneous ones that are synthesized from the transcription model, it is very likely that desynchronization is happening. Suppose we would measure signals from the two eyes via

electrooculography (EOG) and the brain activity via electroencephalography (EEG) from a subject, can we train a transcription model to learn the mapping between the two by synthesizing the EEG from EOG? If a good model can be learned, we may perhaps detect desynchronization (i.e., an outlier case or an anomaly) between the real and the synthesized EEG signals rather than between EEG and EOG signals from two different domains. In this way, desynchronization can be detected by computing the degree of synchrony via the traditional similarity measures (e.g., correlation and synchronization) between the synthesized (or transcribed) and the measured (or real) signals that are from the same domain. If the degree of synchrony is greater than a specific threshold, these simultaneous signals are desynchronized.

Our main contributions are as follows:

- To our knowledge, this is the first work to explore the possibility to transcribe between biosignals from two different modalities for biomedical applications. In this case, we managed to show that eye movement signals (or EOG) can be converted into a synthesized brain signal (or EEG).
- With biosignals signals from different modalities being transcribed into the same modality, we present a new approach for cross-domain biosignal analysis. We demonstrate desynchronization detection as one possible application. To our knowledge, this is the first work to transform the problem of detecting desynchronization among biosignals from different modalities into a *simpler* problem of quantifying the degree of synchrony between biosignals from the same modality.

6.2 Related Works

The existing methods to estimate the degree of synchrony among biosignals can be classified into two main groups: linear and non-linear measures.

In linear measures, cross-correlation [28] has been introduced to measure the linear correlation between two signals as a function of their time delay. If the correlation value is positive, it implies a tendency of both signals to have similar values with the same sign. If the correlation value is negative, it implies a tendency of both signals to have similar absolute values but with opposite signs. If the correlation value is zero, it implies a lack of linear correlation. Another widely-used linear correlation measure is Pearson's R coefficient [104], which can be thought of as the zero-normalized version of cross-correlation at zero time delay. This correlation ranges from -1 to +1, in which 0, +1 and -1 mean no correlation,

total positive correlation and total negative correlation respectively. Coherence [29] (or magnitude squared coherence) has also been introduced to measure the linear correlation between two signals in the frequency domain. The coherence ranges from 0 to 1, in which 0 and 1 means no correlation and complete correlation. This measure is useful when one is interested in the synchronization in certain frequency bands.

In non-linear measures, mutual information [156] has been applied to estimate the amount of extra information one can get from one signal through another signal. The mutual information is zero when two signals are independent. Phase synchronization [32, 157] has been applied to quantify the synchronization in phase or oscillation regardless of the signals amplitudes. This measure has been widely used in neurophysiology.

However, these existing methods mainly focus on measuring the degree of synchrony among biosignals. Without prior knowledge, it is not straightforward to interpret and justify when the signals are desynchronized. In particular, different techniques capture different relationships among signals. It typically requires prior knowledge about the synchronized patterns of the simultaneous signals to select one or more techniques or even develop a new technique to properly estimate the degree of synchrony among them. Also, there may be non-trivial mappings between biosignals from multiple modalities that can cause commonly aggregation techniques, such as summation and average, to miss or discard important relationships among the simultaneous signals. In contrast, our approach can learn synchronized patterns or relationships among simultaneous signals via cross-domain signal transcription from a training set, without any prior knowledge regarding the synchronized patterns among them. Specifically, a model is trained to directly synthesize the expected, simultaneous or synchronized signals that can be compared with the measured simultaneous signals. This provides an interpretable way of detecting desynchronization, in which the input simultaneous signals are deemed desynchronized if the degree of synchrony between the real and the synthesized, expected, synchronized signals is less than a threshold.

For instance, Figure 6.1 illustrates the difference between the traditional and our proposed approaches to measure the degree of synchrony among three signals $\{\mathbf{x}_{1,A}, \mathbf{x}_{2,A}, \mathbf{x}_B\}$ from two domains (A and B). $\mathbf{x}_{j,A}$ denotes the signals from the j-th channel from domain A. $\mathbf{x}_B$ and $\hat{\mathbf{x}}_B$ denotes the real and the transcribed (or synthesized) signals from domain B respectively. *sync* function refers to any functions that measure the degree of synchrony. S function is an aggregation function that summarizes all input values into a single value such as summation and average. Without any prior knowledge about

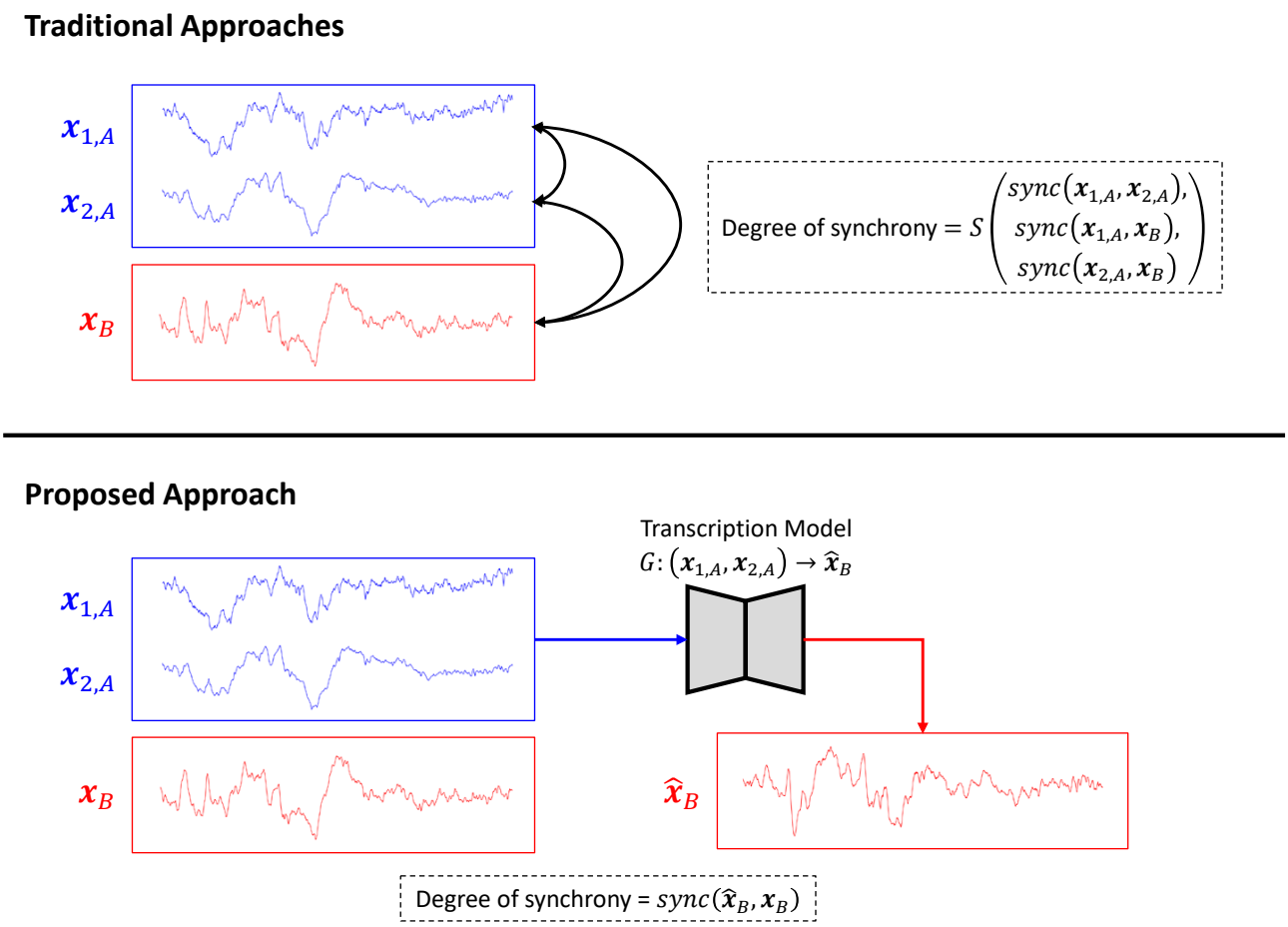


Figure 6.1: Comparison between the traditional and our approaches. The traditional approaches (top) measure the degree of synchrony directly from these three signals. Specifically, they compute the degree of synchrony among all possible pairs of signals. Then degrees of synchrony from all pairs are summarized (e.g., computing the average) into a single value. In contrast, our approach (bottom) first transcribes input signals into their expected, simultaneous signal. The degree of synchrony is then estimated by computing the similarity between the real and the transcribed signals.

how these signals are desynchronized, one of the possible way to measure the degree of synchrony is to compute the degree of synchrony among all possible pairs of the signals. Then the degree of synchrony from all pairs are summarized into a single value. It should be noted that the number of all possible pairs will increase significantly when the number of channels in domain A increases. This would make the process to quantify the degree of synchrony more computational expensive and complicated. In contrast, our approach first transcribes signals from domain A to their expected, simultaneous signals in domain B, $G(\mathbf{x}_{1,A}, \mathbf{x}_{2,A}) \rightarrow \hat{\mathbf{x}}_B$, where G is a transcription model trained to learn the relationships between these two domains. The degree of synchrony can be then quantified by comparing the synthesized and the measured simultaneous signals $\{\hat{\mathbf{x}}_B, \mathbf{x}_B\}$, which are from the same domain, using the traditional approaches.

We believe that our approach is more simple to detect desynchronization compared to the existing approaches because of two main reasons. Firstly, our approach transforms the problem of detecting desynchronization among biosignals from *different* modalities into the problem of quantifying the degree of synchrony between the real and the synthesized biosignals from the *same* modality. Thus, a number of existing techniques for measuring their similarity can be applied. Secondly, our approach automatically learns the relationships between two modalities from the training data (i.e., without prior knowledge) via the cross-domain transcription model. This can ease the process of defining the synchronized features among signals from different modalities.

6.3 Method

We propose to train a model to transcribe input biosignals from one domain into their expected, simultaneous biosignals in another domain. The relationships between two domains are captured by the cross-domain transcription model. Desynchronization is detected by computing the degree of synchrony via the similarity measures between the real and the transcribed simultaneous signals from the same domain. If the computed value is less than a threshold, it is very likely that desynchronization is happening.

In this section, we will discuss our approach to build the transcription model and how the existing techniques for measuring the degree of synchrony between two biosignals can be employed to detect desynchronization.

6.3.1 Cross-domain signal transcription

The key challenge to build a signal transcription model is to learn feature representation that can be transferred across modalities. Generative Adversarial Nets (GAN) [77] has recently shown promising results in modeling complex multimodal data and synthesizing cross-domain, realistic images, such as image-to-image [10, 158] and text-to-image [76, 159]. The GAN framework introduces two players, named discriminator and generator. The discriminator is trying to distinguish between the real and the synthesized signals, while the generator is trying to synthesized signals that can fool the discriminator. If we define a loss function for the discriminator to distinguish between the real and the synthesized simultaneous signals, this discriminator can be used to train the generator to synthesize

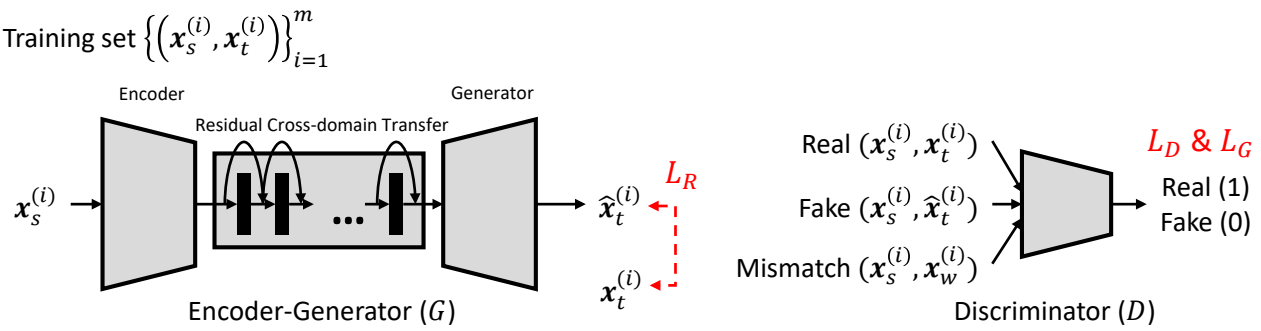


Figure 6.2: An overview of the architecture of our cross-domain signal transcription model and its training framework. The encoder-generator network G transcribes signals from the source domain S to its simultaneous signals in the target domain T, $G : \mathbf{x}_s^{(i)} \rightarrow \hat{\mathbf{x}}_t^{(i)}$. It consists of an encoder, a residual cross-domain transfer, and a generator. The encoder is a CNN that encodes signals from S into a feature representation. The residual cross-domain transfer is a residual network that transfers from the feature representation in the source domain to the representation in the target domain. The generator is a deconvolutional network that generates or synthesizes expected, simultaneous signals in T. The discriminator network D is a CNN that determines whether the input simultaneous signals from both domains are real or fake, $D : (\mathbf{x}_s^{(i)}, \mathbf{x}_t^{(i)} | \hat{\mathbf{x}}_t^{(i)} | \mathbf{x}_w^{(i)}) \rightarrow \{0, 1\}$. L_D, L_G and L_R (highlighted in red) are losses used during the training. Note that the D network is only used during training to provide useful gradients for synthesizing the expected, simultaneous signals, and is not used during the signal transcription.

valid simultaneous signals. Although the GAN has been successfully applied in the field of computer vision, no one has ever employed GAN for cross-domain biosignal transcription.

We, therefore, propose to employ the GAN framework (see Section 2.10) to build our cross-domain signal transcription model. Our model consists of two main components: *encoder-generator* network and *discriminator* network (see Figure 6.2). We first describe our encoder-generator and discriminator networks, and then explain how do we train our signal transcription model using the GAN framework.

Let us introduce notations that will be used throughout this chapter. $\mathbf{x}_s^{(i)}$ denotes the i-th epoch from the source domain S. $\mathbf{x}_t^{(i)}$ denotes the i-th epoch from the target domain T. $\hat{\mathbf{x}}_t^{(i)}$ denotes the synthesized, simultaneous signal in T. $\mathbf{x}_w^{(i)}$ denotes the mismatch epoch randomly selected from T (which is not the same as $\mathbf{x}_t^{(i)}$). $\mathbf{x}_s^{(i)} \in \mathbb{R}^{n_s}$ and $\mathbf{x}_t^{(i)}, \hat{\mathbf{x}}_t^{(i)}, \mathbf{x}_w^{(i)} \in \mathbb{R}^{n_t}$, where n_s and n_t are the number of data points in each epoch from S and T respectively.

Encoder-Generator Network (G). It transcribes signals from S to its simultaneous signals in T, $G : \mathbf{x}_s^{(i)} \rightarrow \hat{\mathbf{x}}_t^{(i)}$. The G network consists of an *encoder*, a *residual cross-domain transfer* and a *generator*. The encoder is a CNN that encodes signals from S into a feature representation. The residual cross-domain transfer is a residual network that transfers from the feature representation in the source domain to the representation in the target domain. The generator is a deconvolutional

network that generates or synthesizes expected, simultaneous signals in T.

Discriminator Network (D). It is a CNN that determines whether the input simultaneous signals from S and T are real or fake, $D : (\mathbf{x}_s^{(i)}, \mathbf{x}_t^{(i)} | \hat{\mathbf{x}}_t^{(i)} | \mathbf{x}_w^{(i)}) \rightarrow \{0, 1\}$.

Training Framework. We train our model using the GAN framework by alternating between one gradient descent step on D, then one step on G. The D network is trained to distinguish between real $(\mathbf{x}_s^{(i)}, \mathbf{x}_t^{(i)})$ and synthesized (or mismatched) $(\mathbf{x}_s^{(i)}, \hat{\mathbf{x}}_t^{(i)} | \mathbf{x}_w^{(i)})$ pairs of simultaneous biosignals from two domains. The G network is trained to transcribe $\mathbf{x}_s^{(i)}$ into $\hat{\mathbf{x}}_t^{(i)}$ such that it can fool the D network. Thus there are two training losses L_D and L_G, in which L_D is minimized over D and L_G is minimized over G. Suppose there are a set of m simultaneous biosignals from S and T, $\{(\mathbf{x}_s^{(i)}, \mathbf{x}_t^{(i)})\}_{i=1}^m$. These losses are defined as follows:

$$L_D = -\sum_{i=1}^{m} \log D(\mathbf{x}_s^{(i)}, \mathbf{x}_t^{(i)}) - \sum_{i=1}^{m} \log(1 - D(\mathbf{x}_s^{(i)}, G(\mathbf{x}_s^{(i)})) - \sum_{i=1}^{m} \log(1 - D(\mathbf{x}_s^{(i)}, \mathbf{x}_w^{(i)})) \ , \tag{6.1}$$

$$L_G = -\sum_{i=1}^{m} \log D(\mathbf{x}_s^{(i)}, G(\mathbf{x}_s^{(i)})) + \alpha L_R \ , \tag{6.2}$$

where L_R is a reconstruction loss, and α is a hyperparameter that control the strength of the L_R.

The first two terms in L_D and the first term in L_G are common adversarial learning that can guide G to synthesize simultaneous biosignals in T. However, it has been shown in [76] that these two terms are not sufficient to train the cross-domain model as the discriminator only observes two kinds of pairs: the real simultaneous signals $(\mathbf{x}_s^{(i)}, \mathbf{x}_t^{(i)})$, and the real and synthesized simultaneous signals, $(\mathbf{x}_s^{(i)}, \hat{\mathbf{x}}_t^{(i)})$. Therefore, the third term in L_D is added to enforce the G network to synthesize $\hat{\mathbf{x}}_t^{(i)}$ that do not only have the same properties as signals in T, but also have synchronized behaviors with the given $\mathbf{x}_s^{(i)}$.

The L_R term in L_G is the reconstruction loss which can also help the G network to synthesize $\hat{\mathbf{x}}_t^{(i)}$ that look similar to $\mathbf{x}_t^{(i)}$. In our work, the mean-squared-error (MSE), $\left\| \mathbf{x}_t^{(i)} - \hat{\mathbf{x}}_t^{(i)} \right\|_2^2$, is used as the reconstruction loss.

Note that, after the training, only the G network is used to synthesize signals. The D network is only used during the training to provide gradients that are useful to synthesize the expected, simultaneous signals in addition to MSE.

6.3.2 Desynchronization Detection

As our transcription model learns synchronized relationships between two modalities from the training set, we hypothesize that the model can synthesize the expected, simultaneous biosignals that are similar to the measured ones when there is a sufficient amount of simultaneous pairs from two domains in the training set. Therefore, when the model observes signals that do not sufficiently appear in the training set, such as desynchronized signals (i.e., outlier cases or anomalies), the synthesized signals can be expected to differ from the measured ones.

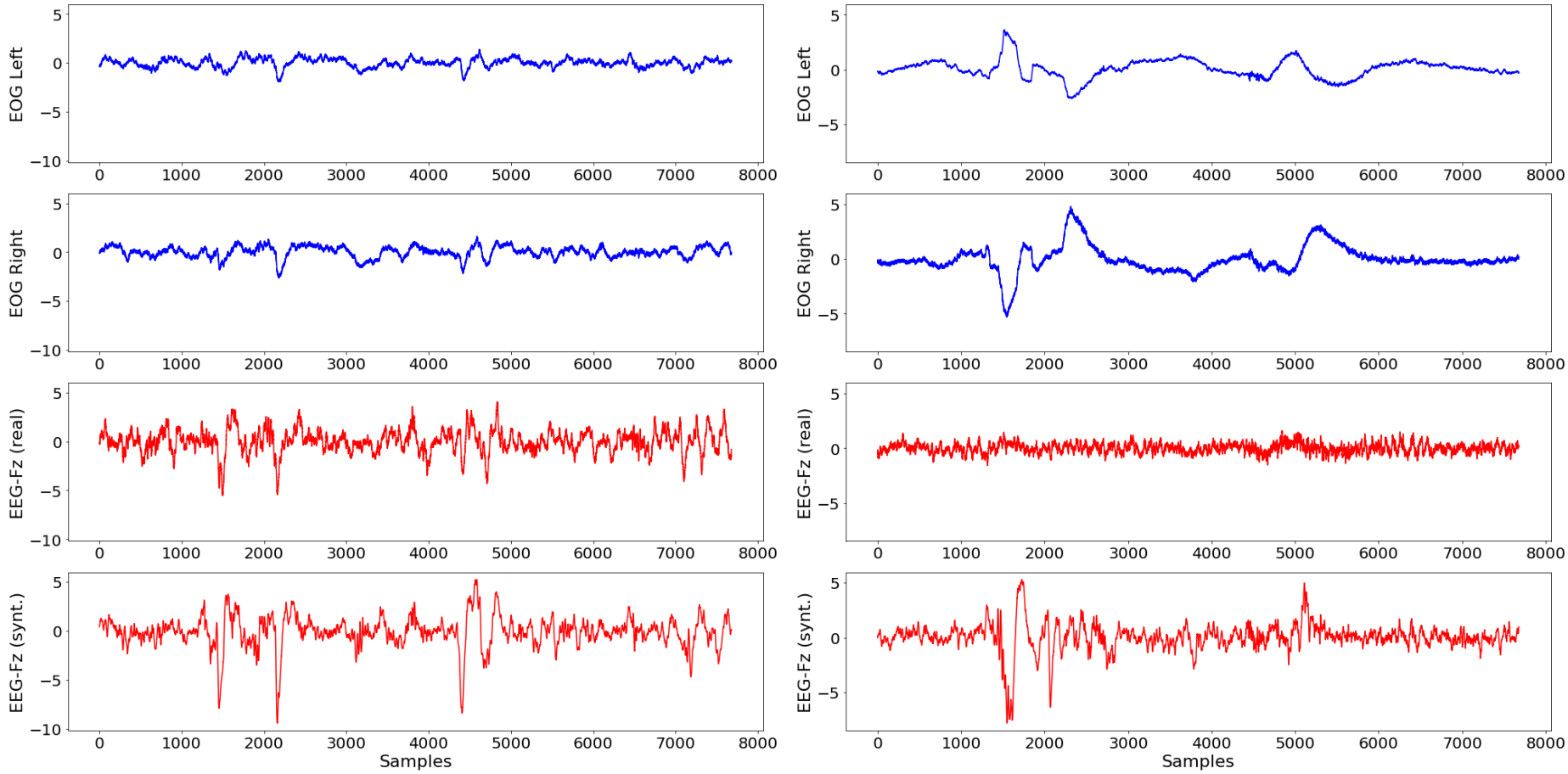
Figure 6.3 demonstrates two examples of synthesized EEG signals from two EOG signals. The blue signals refers to the signals from EOG domain, and the red signals refers to the signals from EEG domain. The first two rows are EOG left and right signals which are the input (i.e., $\mathbf{x}_s^{(i)}$ in Figure 6.2) to the trained transcription model. The third row is the real EEG signal from the Fz-CLE channel (i.e., $\mathbf{x}_t^{(i)}$ in Figure 6.2) The forth row is the synthesized EEG signal (red) which is the output from the model (i.e., $\hat{\mathbf{x}}_t^{(i)}$ in Figure 6.2). In these examples, only the simultaneous recordings of EOG and EEG signals that do not contain eye-movements are used to train the transcription model. It can be seen that the synthesized signal in Figure 6.3a is more similar to the real EEG-Fz signal compared to the synthesized one in Figure 6.3b. This is because the EOG signals in Figure 6.3b contain eye-movements which cannot be explained by the model. Thus the eye-movement (or desynchronization in these examples) can be detected by comparing the similarity between the real and the synthesized signals with a threshold.

Formally, each i-th pair of the real and synthesized biosignals, $(\mathbf{x}_t^{(i)}, \hat{\mathbf{x}}_t^{(i)})$ is classified as the synchronized (0) or the desynchronized (1) class if:

$$d^{(i)} = \begin{cases} 1, & \text{if } \phi(\mathbf{x}_t^{(i)}, \hat{\mathbf{x}}_t^{(i)}) < \varepsilon \\ 0, & \text{otherwise} \end{cases}, \tag{6.3}$$

where $d^{(i)}$ is the desynchronization label of the i-th simultaneous signal, ϕ is a function that estimates the degree of synchrony between $\mathbf{x}_t^{(i)}$ and $\hat{\mathbf{x}}_t^{(i)}$, and ε is the threshold used to distinguish between the synchronized and the desynchronized signals.

As we are trying to measure the similarity between the real and the synthesized biosignals, amplitude correlation is necessary. We propose to use the Pearson's R coefficient [104], r, as the function ϕ in



(a) Synchronized example

(b) Desynchronized example

Figure 6.3: Examples of EEG signals (the last row) synthesized from two EOG signals using the transcription model. The blue signals refers to the signals from EOG domain, while the red signals refers to the signals from EEG domain. As the transcription model is trained with the signals that do not contain eye-movements, the synthesized signal in (a) is more similar to the real EEG-Fz signal (the third row) compared to the synthesized one in (b). This is because the EOG signals from the right contain eye-movements which cannot be explained by the model.

Equation 6.3. The resulting coefficient $r(\mathbf{x}_t^{(i)}, \hat{\mathbf{x}}_t^{(i)}) \in [-1, 1]$ is an estimate for the linear correlation between the real and the synthesized simultaneous signals. If the value approaches +1, $\mathbf{x}_t^{(i)}$ and $\hat{\mathbf{x}}_t^{(i)}$ are considered as being synchronized. If the value approaches -1, they are considered as being desynchronized.

6.4 Results

6.4.1 Dataset

We evaluate the performance of our signal transcription model and desynchronization detection using the subset 3 (SS3) from the cohort 1 from the MASS dataset [146], which is the same dataset we used for sleep stage scoring in the previous chapter. It consists of simultaneous recordings from 62 healthy subjects, including two EOG (left and right) and three EEG signals from Fz-CLE, Cz-CLE and Oz-CLE channels (i.e., EEG-Fz, EEG-Cz and EEG-Oz). The EEG-Fz, EEG-Cz and EEG-Oz are from the electrodes positioned at the front, central and the back of the head respectively. These signals have the sampling rate of 256 Hz. Each 30-s epoch of these recordings is annotated with one of the five sleep stages: W, N1, N2, N3 and REM.

6.4.2 Experimental Design

We assume that eye-movements are desynchronization events. Thus the recordings from the sleep stages that contains eye-movements, which is REM, are annotated as the eye class (i.e., the desynchronized class), while the recordings from the sleep stages that do not contain eye-movements, which are N2 and N3, are annotated as the no-eye class (i.e., the synchronized class). Even though the recordings from stage W and N1 may also contains eye-movements according to the AASM manual [27], we exclude them from the experiment as some of them may not contain eye-movements.

We randomly split the subjects into two sets: 50 subjects for training and validation, and 12 subjects for testing. Only the recordings from the no-eye class are used to train the transcription model. Specifically, our model was trained to transcribe each 5-s epoch of two EOG signals (i.e., $n_s = 256 \cdot 5$) into its expected, simultaneous 5-s epoch of EEG-Fz signal (i.e., $n_t = 256 \cdot 5$). The trained model is then applied to all recordings (both the eye and no-eye classes) from the test set. The synthesized 5-s EEG-Fz epochs are aggregated back into 30-s epochs following how each epoch is initially annotated

in the dataset. The signal transcription and the desynchronization detection performance from all test subjects are reported using the metrics discussed in the later section.

6.4.3 Network Architecture and Training Parameter

The architecture of G and D networks and the training parameters were obtained from the results of our initial experiment with three subjects in the training set. In particular, the signals from 47 training subjects were used to train the transcription model, and evaluated with the three withheld training subjects.

In the G network, the *encoder* consists of four 1-D convolutional layers with 64, 128, 256 and 512 filters respectively, each followed by batch normalization [84] and LeakyReLU function [59] with the leak parameter of 0.2. Each convolutional layer has the filter size of 4 and the stride size of 2. The *residual cross-domain transfer* consists of 4 residual blocks [9]. Each residual block consists of two 1-D convolutional layers with the filter size of 4 and the stride size of 1, each followed by the batch normalization. The LeakyReLU activation with the leak parameter of 0.2 is only applied to output of the first convolutional layer. The *generator* consists of four 1-D deconvolutional layers with 256, 128, 64 and 1 filters respectively, each followed by batch normalization and LeakyReLU with the leak parameter of 0.2. Each deconvolutional layer has the filter size of 4 and the stride size of 2. At the final deconvolutional layer, we found that when we stacked one more 1-D convolutional layer on top of the output from the deconvolution, the G network can synthesize better signals. This convolutional layer has the filter size of 4 and the stride size of 1 without the batch normalization and the LeakyReLU.

The architecture of D network consists of five convolutional layers with 64, 128, 256, 256 and 32 filters respectively, each followed by batch normalization and LeakyReLU with the leak parameter of 0.2. Each convolutional layer has the filter size of 4 and the stride size of 2. The output of the deconvolution is forward to the binary logistic classifier to distinguish between real and fake pairs.

The Adam optimizer [56] was used to train the G and D networks. The parameters of the Adam lr, $beta1$ and $beta2$ were set to $2{\times}10^{-4}$, 0.5 and 0.999 respectively. The learning was decayed for every 50 training epochs by 0.5. The weight of the reconstruction loss α is set to 0.1. We found that the training process converged after 140 training epochs.

6.4.4 Performance Metrics

Cross-domain Signal Transcription

In order to evaluate the quality of the synthesized signals, we suggest the following measures, which are computed for each 30-s epoch:

- The r value between real and synthesized signals, denoted $c1 = r(x_t, \hat{x}_t)$
- The r value between real and input signals, denoted $c2 = r(x_t, x_s)$
- The r value between synthesized and input signals, denoted $c3 = r(\hat{x}_t, x_s)$
- The r value between mismatch and synthesized signals, denoted $c4 = r(x_w, \hat{x}_t)$

If the model is able to synthesize signals that are similar to the real simultaneous signals, the following criteria should be met:

- The value of $c1$ is close to 1
- The value of $c2$ and $c3$ are similar
- The value of $c4$ is lower than $c1$

Desynchronization Detection

In our experiment, the desynchronization detection is a binary classification between eye and no-eye classes. Specifically, for each 30-s epoch, if the r value between the real and synthesized signals (i.e., $c1$) is less than a given threshold ε, this epoch is classified as the eye class. Otherwise, the epoch is classified as the no-eye class.

We evaluate the detection performance using detection accuracy (ACC), F1-score (F1), sensitivity (also called recall or true positive rate (TPR)), false positive rate (FPR), and area under the ROC curve (AUC). These metrics are used to compare the detection performance with baseline approaches, which will be discussed in the next section.

6.4.5 Baseline Methods

We compare the performance of desynchronization detection with two baseline methods: Pearson's R coefficient (BASE-1) and intrinsic phase synchrony [32] (BASE-2). For each method, we directly compute the degree of synchrony among 30-s epochs of simultaneous EOG and EEG signals without the synthesized signals. Specifically, we estimate the degree of synchrony of three pairs: EOG (left) - EOG (right), EOG (left) - EEG Fz, and EOG (right) - EEG Fz.

- **BASE-1**: we compute linear correlations using Pearson's R coefficient. The degree of synchrony of each 30-s epoch is the average of the absolute values of the correlations from the three pairs.
- **BASE-2**: we quantified phase relationship between the signals using the so-called intrinsic phase synchrony [32]. In particular, all the three signals considered were first decomposed into a set of intrinsic mode functions (IMFs), i.e., intrinsic oscillations, using the noise-assisted multivariate empirical mode decomposition (NA-MEMD) algorithm. Next, we removed IMFs that contained noise (i.e., frequency components approximately $>$ 60 Hz), DC components and trends. We then computed instantaneous phases of the sum of the remaining IMFs of all the three signals using the Hilbert transform. Phase synchrony index (PSI) was subsequently computed for all the three pairs of the instantaneous phases. The PSI ranges between 0 and 1, with 1 indicating perfect synchrony and 0 no synchrony. The degree of phase synchrony of each 30-s epoch was then obtained by averaging the PSIs from all the pairs.

Similar to our approach, these degrees of synchrony are then compared with the specific threshold to detect desynchronization.

6.4.6 Cross-domain Signal Transcription Performance

Table 6.1 shows the mean and standard deviation of the c1, c2, c3 and c4 from each test subject. These values are computed only for signals from the no-eye class as they were used to train the transcription model or the G network. "R-{ch}" denotes the real EEG signals from the {ch} channel, such as R-Fz refers to the real EEG-Fz signals. "R-Right" denotes the real EOG-Right channel. "S-Fz" denotes the synthesized EEG-Fz channel.

It can be seen that most of the subjects had the average of $c1$ more than 0.5, and the value of $c2$ and $c3$ were similar with the difference between their overall mean around 0.3. The $c4$ from both

R-Cz/S-Fz and R-Oz/S-Fz were lower than the values of $c1$ across all subjects. These results showed that our model can synthesized the expected, simultaneous signals that are not random, and only linearly correlated with the target channel in the target domain.

However, there were four subjects: B, F, H and L that had the average of $c1$ less than 0.5. Figure 6.4 illustrates two examples of the EEG-Fz signals synthesized from the EOG signals from Subject B. Even though the input EOG signals are from the no-eye class, the synthesized EEG-Fz signals are not highly correlated with the real EEG-Fz signals. This can be due to two main problems. Firstly, the synchronization patterns of the EOG and EEG signals of such patients were different from the ones in the training set. This problem is common in the time-series analysis, especially for brain signals, as different subjects can have different synchronized patterns [27]. If the number of simultaneous recordings of each subject are sufficient to train the transcription model, we believe that the subject-specific training can alleviate this problem. Secondly, the synchronized pattern of the EOG and EEG signals are highly multimodal. This means that there are many plausible EEG signals that are synchronized with one pattern of EOG signals. When the model is trained with the MSE loss (see Section 6.3.1), it would try to synthesize signals, which have low MSE loss in average, that are not similar to any of the real EEG-Fz signals. This problem might be address by introducing a random element into the input of the G network (e.g., a noise vector sampled from Gaussian distribution) to allow the G network to synthesizes more than one expected, simultaneous signals. These synthesized signals are then compared with the real one. However, this can lead another problem of quantifying the degree of synchrony between multiple synthesized signals and the real one. We leave this investigation as an open research problem.

6.4.7 Desynchronization Detection Performance

Table 6.2 shows a comparison between our proposed and the two baseline methods across degree of synchrony of each class, ACC, F1 and AUC. The numbers in bold indicate the highest performance metrics of all methods. Figure 6.5 shows the histograms of the degree of synchrony from all test subjects across all methods. The ε thresholds in Equation 6.3 used to distinguish between the no-eye and the eye classes in Equation 6.3 were selected from the ones that gave the best performance in term of TPR and FPR. In particular, we plotted ROC curves of all methods obtained by varying ε from -1 to 1 for BASE-1 and the propose method, and from 0 to 1 for BASE-2 (see Figure 6.6). The bold circle markers indicate the selected thresholds that are closest to the perfect classification point

Table 6.1: Summary of the cross-domain signal transcription performance in terms of $c1$, $c2$, $c3$ and $c4$. Each row shows the average ± standard deviation of each metric for each subject. The last row shows the overall performance from all subjects.

Subject	c1 R-Fz/S-Fz	c2 R-Fz/R-Right	c3 S-Fz/R-Right	c4 R-Cz/S-Fz	c4 R-Oz/S-Fz
A	0.54 ± 0.11	0.60 ± 0.10	0.55 ± 0.10	0.36 ± 0.11	-0.02 ± 0.13
B	0.33 ± 0.11	0.41 ± 0.11	0.42 ± 0.13	0.15 ± 0.09	-0.11 ± 0.09
C	0.56 ± 0.12	0.63 ± 0.09	0.56 ± 0.11	0.31 ± 0.11	-0.07 ± 0.11
D	0.56 ± 0.09	0.63 ± 0.07	0.59 ± 0.09	0.29 ± 0.09	-0.02 ± 0.09
E	0.55 ± 0.15	0.56 ± 0.17	0.61 ± 0.15	0.25 ± 0.11	-0.18 ± 0.10
F	0.37 ± 0.09	0.43 ± 0.08	0.50 ± 0.09	0.21 ± 0.09	0.01 ± 0.13
G	0.53 ± 0.10	0.49 ± 0.14	0.51 ± 0.13	0.32 ± 0.11	0.03 ± 0.14
H	0.47 ± 0.09	0.46 ± 0.10	0.55 ± 0.09	0.27 ± 0.11	-0.13 ± 0.14
I	0.51 ± 0.14	0.40 ± 0.23	0.53 ± 0.13	0.33 ± 0.13	-0.04 ± 0.11
J	0.52 ± 0.11	0.46 ± 0.13	0.49 ± 0.10	0.29 ± 0.12	-0.01 ± 0.13
K	0.50 ± 0.10	0.46 ± 0.12	0.49 ± 0.11	0.33 ± 0.10	-0.09 ± 0.10
L	0.46 ± 0.13	0.37 ± 0.15	0.47 ± 0.12	0.26 ± 0.13	-0.08 ± 0.11
Overall	**0.49 ± 0.13**	**0.49 ± 0.16**	**0.52 ± 0.12**	**0.28 ± 0.12**	**-0.06 ± 0.13**

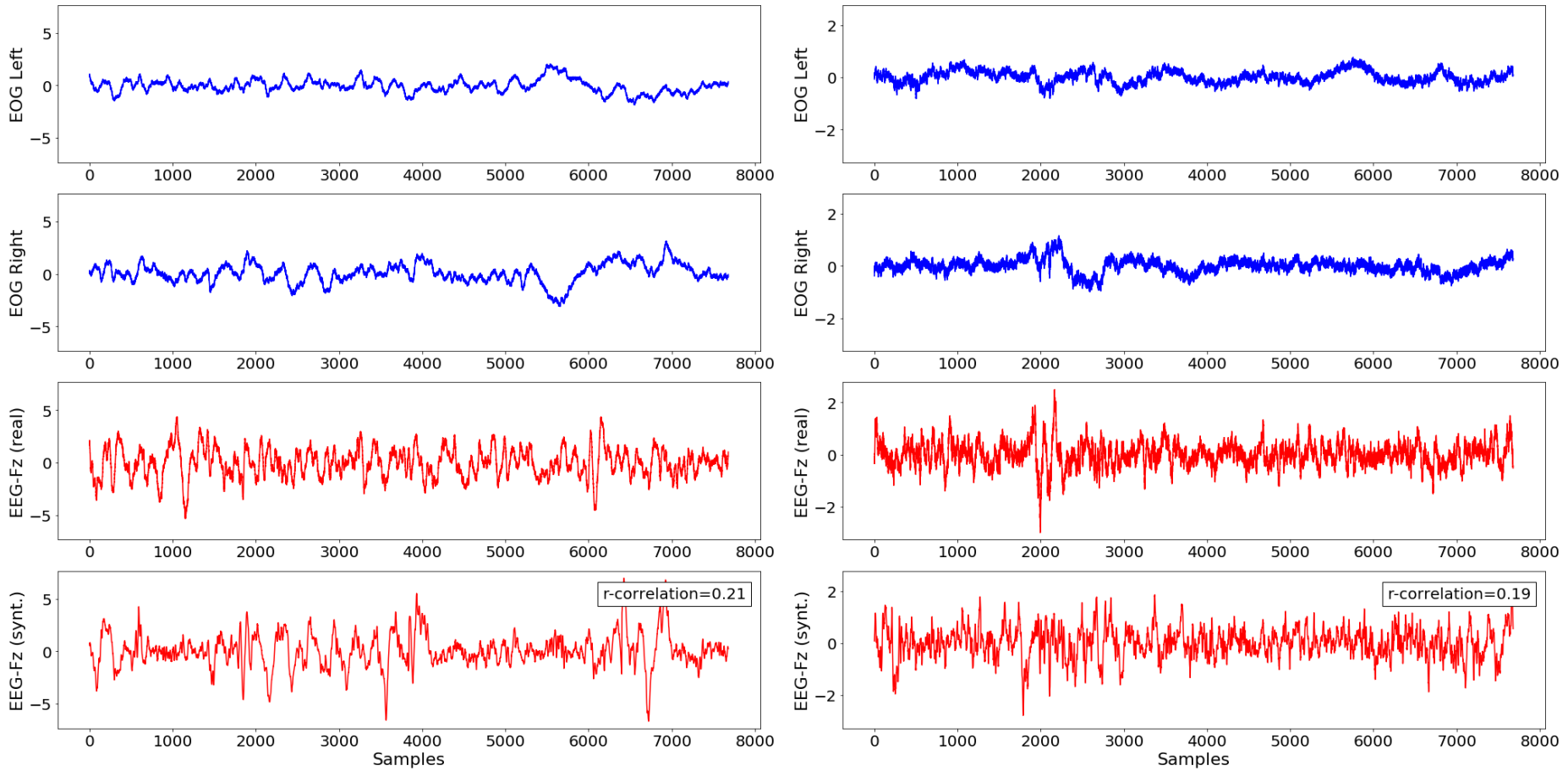


Figure 6.4: Examples of the EEG-Fz signals synthesized from the EOG signals from the no-eye class from Subject B that have low r-correlations. The r-correlation annotated at the top-right of the last row refers to the Pearson's R coefficient between the real (the third row) and the synthesized EEG-Fz (the last row).

Table 6.2: Comparison between our proposed and the baseline methods across degree of synchrony of each class, detection accuracy (ACC), F1-score (F1), and area under ROC curves (AUC)

Method	Test Epochs		Degree of Synchrony		ε	Performance Metrics		
	No-eye	Eye	No-eye	Eye		ACC	F1	AUC
BASE-1	7579	1904	0.45 ± 0.13	0.34 ± 0.09	0.39	67.87	49.31	0.73
BASE-2	7579	1904	0.04 ± 0.01	0.05 ± 0.03	0.04	47.89	23.62	0.40
Proposed	7579	1904	0.49 ± 0.13	0.33 ± 0.13	0.42	**70.64**	**51.16**	**0.80**

Table 6.3: Confusion matrices obtained by applying BASE-1, BASE-2 and our proposed method on simultaneous recordings from all test subjects. Each row and column represent the number of 30-s epochs of the no-eye and the eye classes, classified by the criteria mentioned in Section 6.4.2 and by each method respectively. The numbers in bold indicate the number of epochs that were correctly classified.

	Predicted	
	No-eye	Eye
No-eye	**4954**	2625
Eye	422	**1482**

(a) BASE-1

	Predicted	
	No-eye	Eye
No-eye	**3777**	3802
Eye	1140	**764**

(b) BASE-2

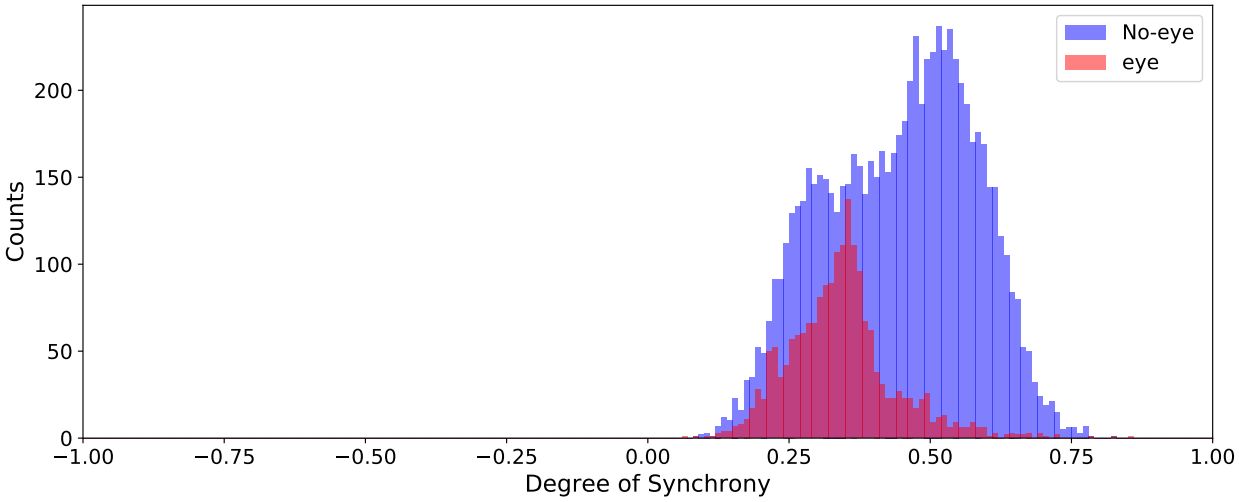
	Predicted	
	No-eye	Eye
No-eye	**5241**	2338
Eye	446	**1458**

(c) Proposed

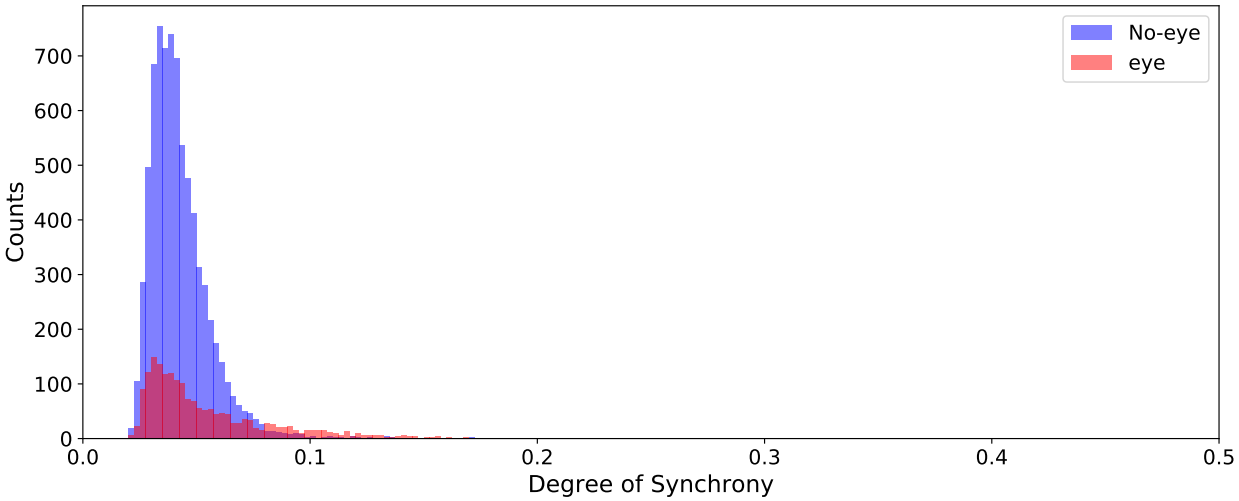
(i.e., TPR=1 and FPR=0) in terms of the Euclidean distance. The AUC of each method in Table 6.2 was also obtained from these ROC curves. Table 6.3 shows the confusion matrices used to compute the ACC and F1 in Table 6.2. Each matrix was obtained by applying each method on simultaneous recordings from all test subjects. Each row and column represent the number of 30-s epochs of the no-eye and the eye classes, classified by the criteria mentioned in Section 6.4.2 and by each method respectively. The numbers in bold indicate the number of epochs that were correctly classified.

It can be seen that our method achieved a better performance compared to the baseline methods. Figure 6.7 and 6.8 show examples of the EEG-Fz signals synthesized from the EOG signals from the no-eye and the eye epochs of Subject A respectively. It can be seen that the r correlation between the real and the synthesize signals is high for the no-eye epochs, and low for the eye epochs.

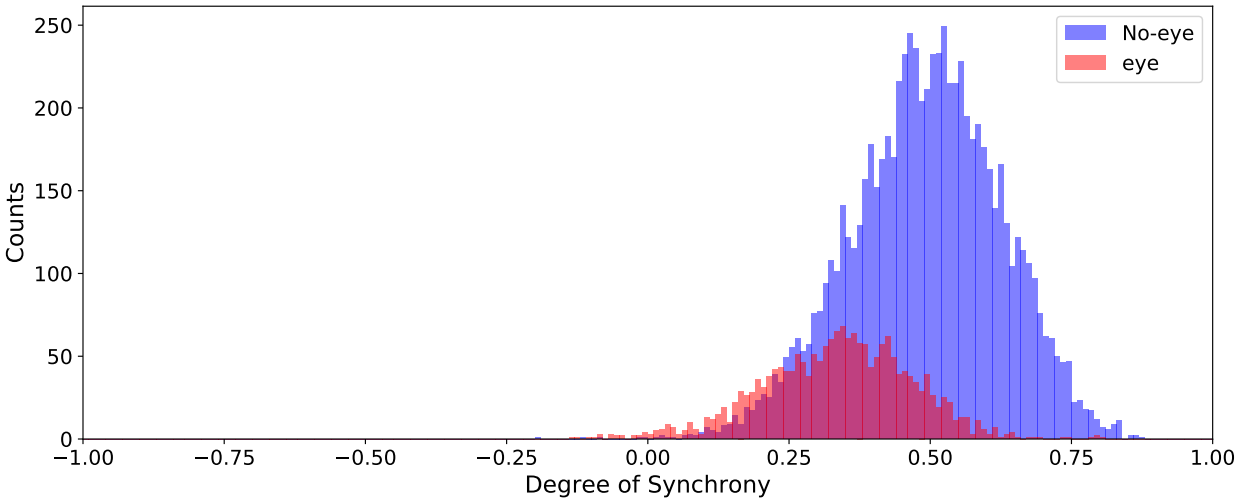
Comparing with BASE-1, it can be seen that our method achieved a better performance on all performance metrics. This showed that the use of the linear correlation alone was not sufficient to capture the relationships among multiple signals from different modalities. Some important information may be lost because the degree of synchrony was estimated by averaging the correlations from all pairs. These results demonstrated that the transcription model can help capturing such relationships and preventing the lost of information from averaging the degree of synchrony from all pairs.



(a) Base-1



(b) BASE-2



(c) Proposed Method

Figure 6.5: Histograms of the degree of synchrony from the no-eye (red) and the eye (blue) epochs quantified using BASE-1 (top), BASE-2 (middle) and our method (bottom)

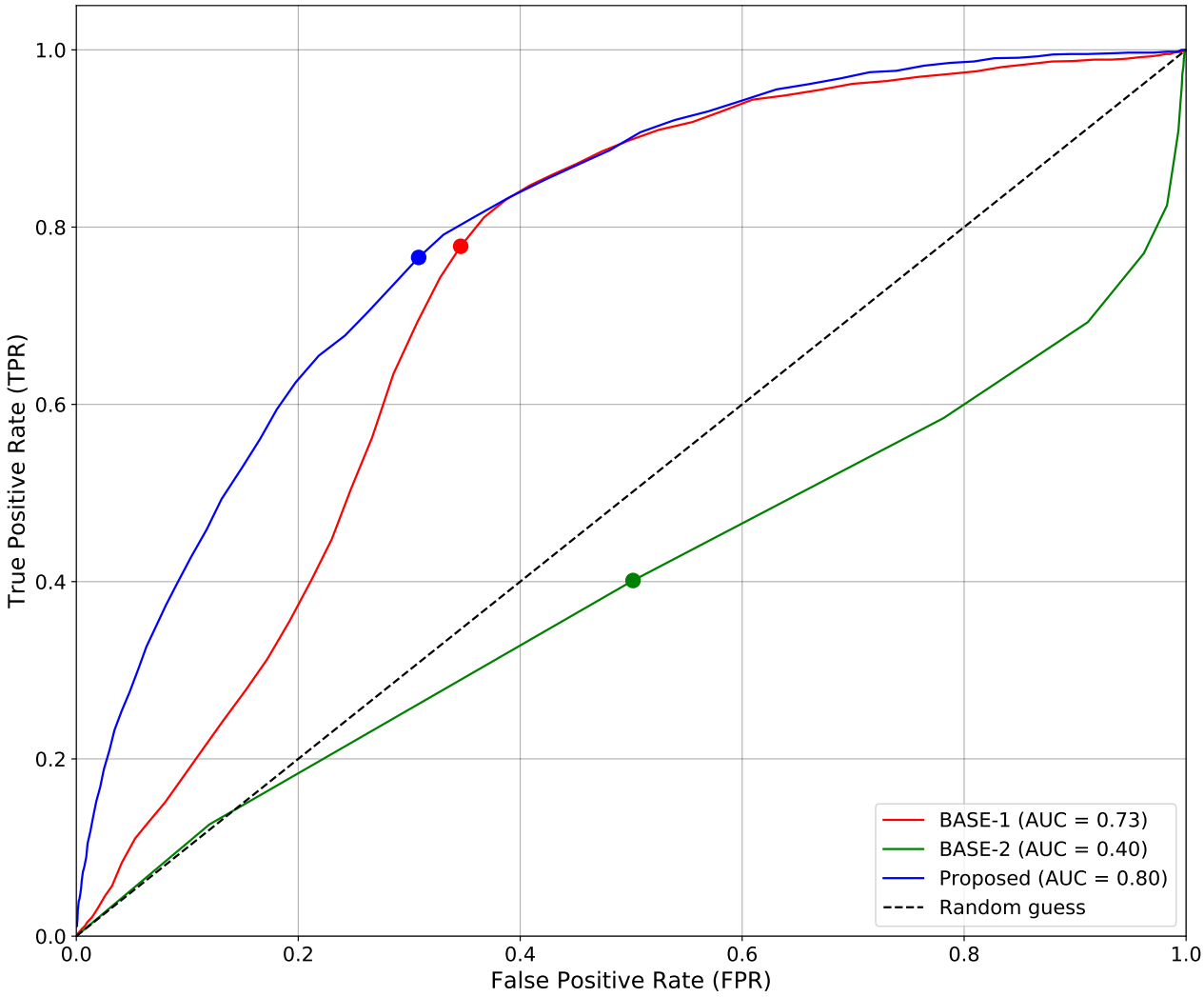


Figure 6.6: ROC curves of BASE-1, BASE-2 and our proposed method obtained by varying ε from -1 to 1 for BASE-1 and the propose method, and from 0 to 1 for BASE-2. Each bold circle marker indicates the threshold that gives TPR and FPR closest to the perfect classification point (i.e., TPR=1 and FPR=0) in terms of the Euclidean distance.

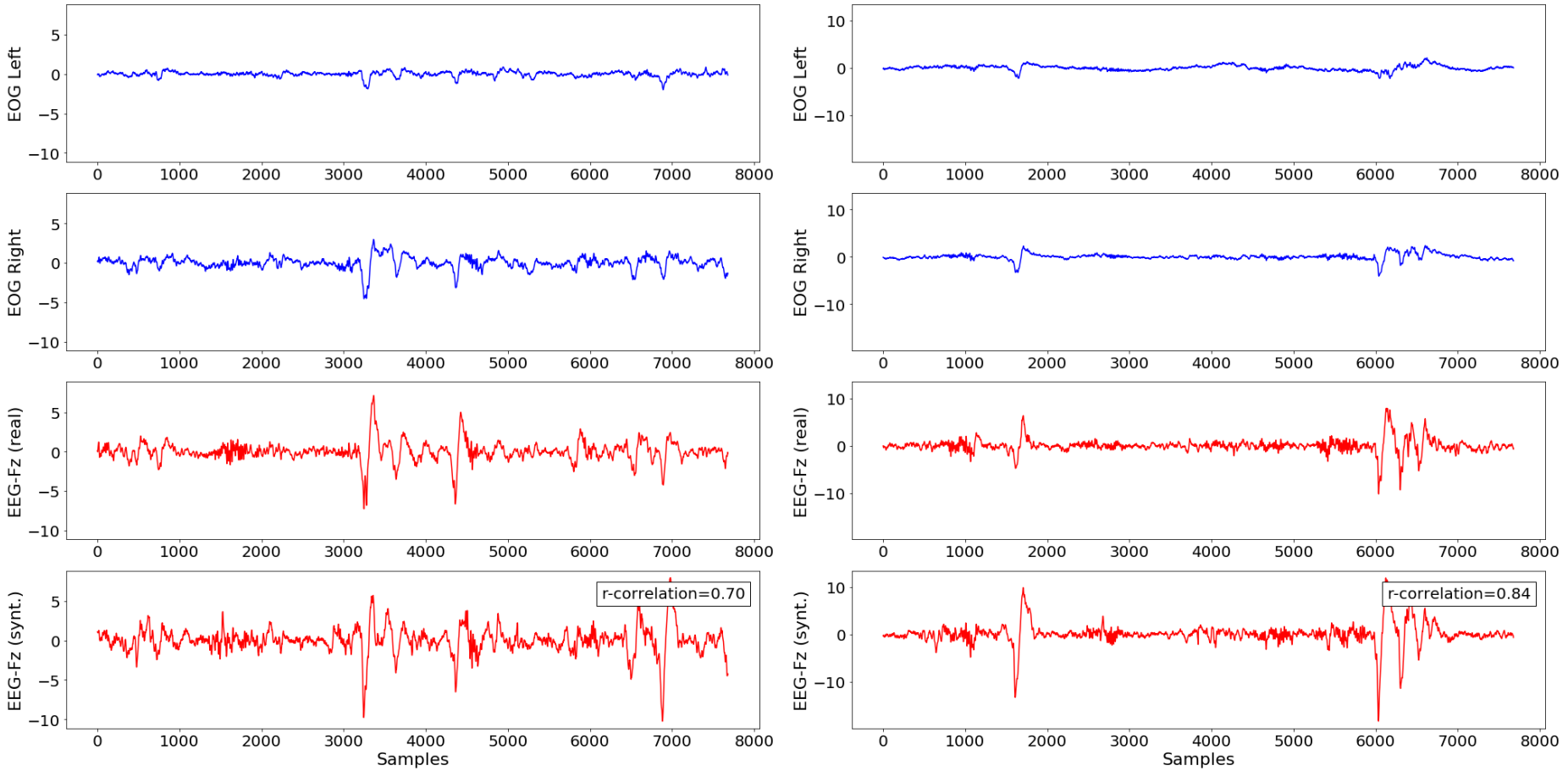


Figure 6.7: Examples of the EEG-Fz signals synthesized from the EOG signals from the no-eye class from Subject A.

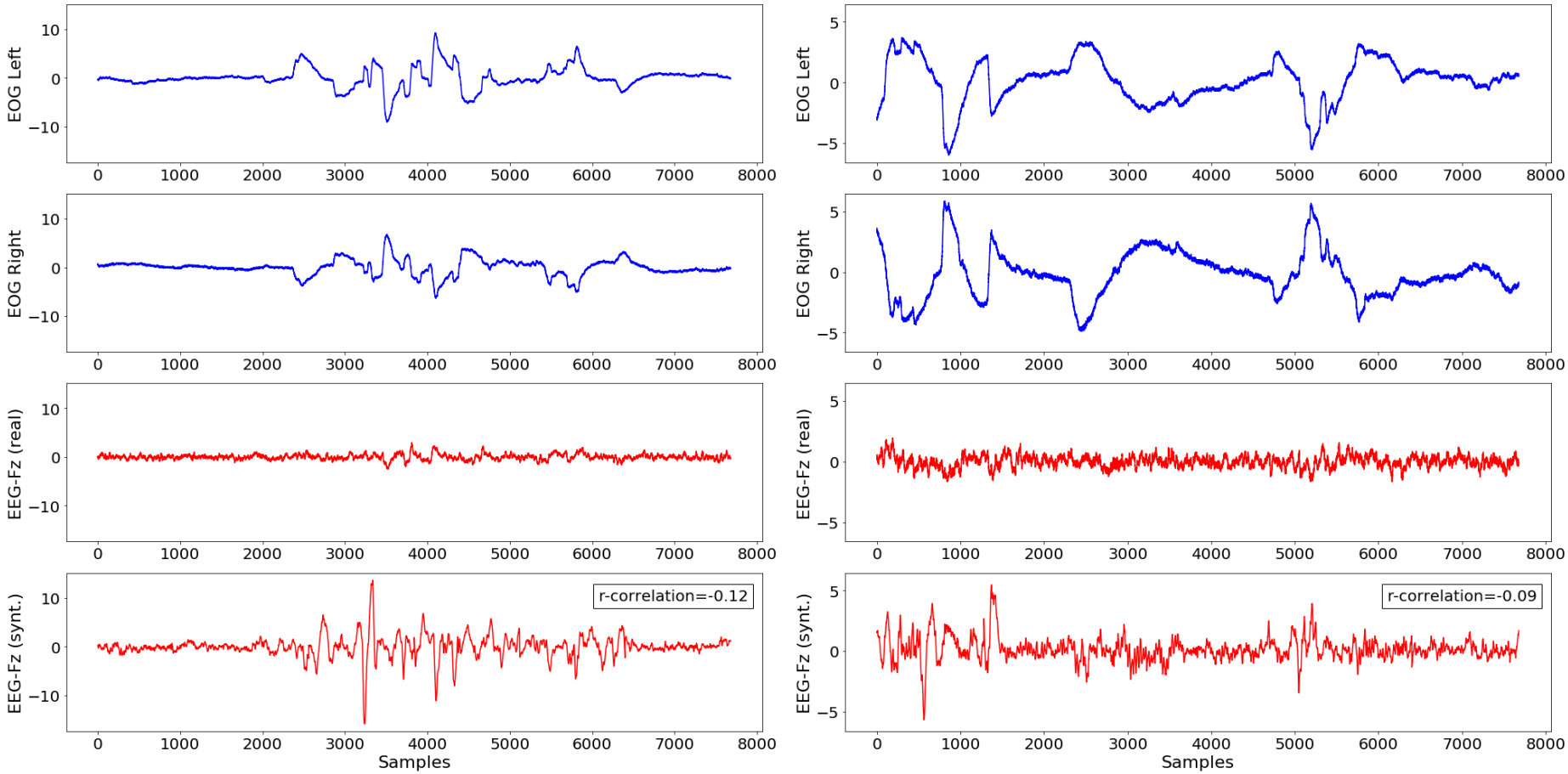


Figure 6.8: Examples of the EEG-Fz signals synthesized from the EOG signals from the eye class from Subject A.

According to the detection performance of BASE-2, it can be seen the phase synchrony performed poorly compared to BASE-1 and our method. Figure 6.5b also shows that the degree of synchrony of some epochs from the eye class are even higher than the ones from the no-eye class. These showed that the simple thresholding is not sufficient for phase synchronization to distinguish between the no-eye and the eye epochs.

6.5 Discussion

In the evaluation of the cross-domain signal transcription, the results demonstrated that our transcription model can transcribe two EOG signals into the expected, simultaneous EEG signals that are not random, and only linearly correlated with the target channel in the target domain. In the evaluation of the desynchronization detection, the results showed that the transcription model can help capturing the relationships between EOG and EEG signals that were missed by averaging of the linear correlations from all pairs of signals (BASE-1). The results also showed that we cannot simply measure the degree of phase synchrony (i.e., BASE-2) from all pairs, compute the average, and compare with a threshold to detect eye-movements as desynchronized events. Even though a more sophisticated algorithm may improve the performance of the phase synchronization, but we do not know in advance, for instance, which frequency bands should be compared to detect eye-movements. This process would require application-specific prior knowledge about the dynamics between EEG and EOG during eye-movements to design such algorithm. This process can be time-consuming, and the algorithm may not generalize well to detect other desynchronization patterns. In contrast, our approach automates this hand-engineering features by training the cross-domain transcription model to automatically learn synchronized patterns between EOG and EEG domains from the training data. This can help simplify the process of manually hand-engineering synchronized features. Our approach also provides an interpretable way of detecting desynchronization, in which the input simultaneous signals are deemed desynchronized if the degree of synchrony between the real and the synthesized, expected, synchronized signals is less than a threshold.

Even though we demonstrated that the r correlation between the real and the synthesized signals can be used to detect desynchronization, our approach is not restricted to this correlation metric. Other existing techniques can also be applied to quantify the degree of synchrony as well. For instance, the frequency domain analysis can be employed to extract a spectral-based metric that would allow an

additional quantiative comparison between the real and synthesized signals in the frequency domain. Figure 6.9 show the spectrogram obtained by STFT (window-size of 1 sec and overlapping of 0.5 sec) to compare the change of the frequency components over time between the real and synthesized signals. It can be seen that the changing trend of the synchronized signals coincides, while the trend of the desynchronized signals is different. We leave the investigation to include such spectral-based metrics as our future work.

In this study, we investigate the transcription model that transforms from multi-channel signals in one domain into single-channel signals in another domain. When there are more than one channel in the target domain, the use of only one bivaraite synchrony measure such as Pearson's R coefficient demonstrated in this study might not be sufficient to effectively quantify the degree of synchrony between the real and synthesized multi-channel signals. A combination of bivariate measures or the multivariate synchrony measures such as multivariate phase synchronization might be more appropriate. However, we believe that our approach is still simpler compared to the conventional approaches that quantify the degree of synchrony among the multi-channel signals from multiple domains. With our approach, we only compute the degree of synchorny between the real and synthesized multi-channel signals from the same domain. We leave this further investigation as our future work.

6.6 Conclusions

We propose a novel data-driven approach to detect desynchronization between biosignals from two modalities. Our contribution is that we transform the problem of detecting desynchronization among biosignals from multiple modalities into the simpler problem of quantifying the degree of synchrony between biosignals of the same modality. In particular, our approach is to train a model to learn non-trivial mappings or relationships between biosignals from multiple modalities by transcribing biosignals from one domain to their expected, simultaneous or synchronized biosignals in another domain. The transformation is achieved via the cross-domain signal transcription. The desynchronization detection is achieved by computing the degree of synchrony via the similarity between the measured (or real) and the transcribed (or synthesized) signals. If the degree of synchrony is less than a threshold, it is very likely that the desynchronization (i.e., the anomaly) is happening.

We demonstrate the capability of our approach to detect desynchronization by assuming that eye-movements in simultaneous recordings of EOG and EEG signals from sleep dataset are desynchronized

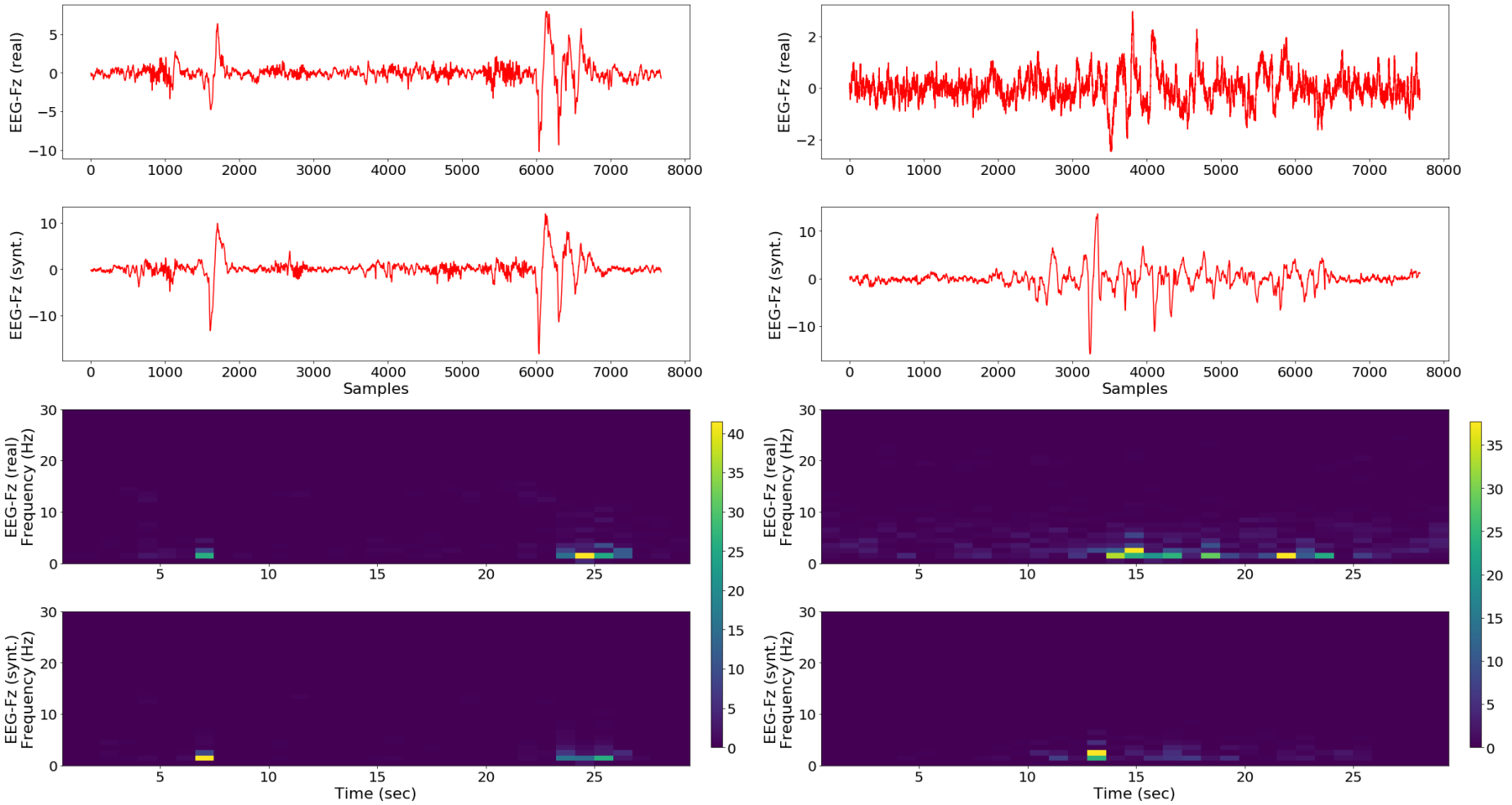


Figure 6.9: Examples of the spectrograms from the real and the synthesized EEG-Fz signals from the no-eye epoch in Figure 6.7a (a) and from the eye epoch in Figure 6.8a (b). It can be seen that the changing trends of the synchronized signals coincide, while the trends of the desynchronized signals are different.

events. For cross-domain signal transcription, the results showed that our model can synthesize the expected, simultaneous or synchronized signals which can be used to detect eye-movements. For the desynchronization detection, comparing to the two baseline methods, we found that our transcription model can help capturing the relationships between EOG and EEG signals that are missed by the traditional Pearson's R coefficient approaches. We also found that the simple threshold is not sufficient for the traditional phase synchronization to detect eye-movements, as it requires prior knowledge to define features that are useful for eye-movement detection. This shows that our approach simplifies the problem of detecting desynchronization into the problem of comparing the similarity between the real and the synthesized signals with a specific threshold.

We believe that our approach is simpler compared to the conventional approaches. Our approach automates the process of hand-engineering desynchronized features by training the cross-domain transcription model to learn relationships between two domains automatically. Our approach also provides an interpretable way of detecting desynchronization, in which the input simultaneous signals are deemed desynchronized if the degree of synchrony between the real and the synthesized, expected, synchronized signals is less than a threshold.

In the future, we plan to incorporate the spectral-based metric that allows additional comparison between the real and the synthesized signals in the frequency domain. We will also extend our work to synthesize multi-channel signals, and investigate a technique to quantify the degree of synchrony among the real and the synthesized multi-channel signals.

7

Conclusions

Recent developments of new technology in both clinical and home environment allows us to continuously collect biosignals for a better understanding of the current status of our well-being. Despite the availability of an enormous amount of biosignals collected from either clinical or wearable device, it is still challenging to learn useful information from them. One of the main reasons is because we do not have enough domain experts to explain and annotate them with meaningful information. For instance, to understand how well people sleep, we need sleep experts to manually go through all recorded EEG signals and annotate them with different sleep stages, which will be used to determine the quality of our sleep.

The first challenge is to automate the feature-engineering process that is typically time-consuming, labor-intensive and application-specific. The second challenge is to develop methods that can extract meaningful information even when the annotations of biosignals (which very expensive to obtain) are not available. In this thesis, we propose methods that can automatically learn features from biosignals without utilizing any hand-engineered features and can extract meaningful information from biosignals even when the annotations of the signals are not available.

The concrete contributions of our thesis to address such challenges are as follows:

- In Chapter 3, we propose a method for subject-specific gait calibration using the actigraphy data from wearable, tri-axial accelerometers for automated estimation of walking speed for people with MS. We show that our approach can remotely and accurately estimate walking speed independent of MS subject's disability level. We also show that the standard approach that calibrates a

generic model for walking speed estimation lead to inaccuracies in PwMS, and such inaccuracies increase with disability level. Our results also demonstrated that walking speeds measured by the standard procedure in the clinic strongly correlated with the maximum sustained walking speed in the home environment.

- In Chapter 4, we propose a subject-specific system that employs a deep learning model to automatically learn features that are useful epileptic seizure detection from raw scalp EEG. We show that, without utilizing any seizure annotations, our system achieved a comparable detection performance compared to the state-of-the-art hand-engineering approaches that were trained in the supervised setting. We also demonstrate that our system can utilize the seizure annotations to reduce the false detection rate.

- In Chapter 5, we propose a deep learning model, named DeepSleepNet, for automatic sleep stage scoring based on raw single-channel scalp EEG. The results demonstrated that, without any changes to the model architecture, our DeepSleepNet can automatically learn features for sleep stage scoring from different raw single-channel EEGs from two datasets that have different properties and scoring standards. This has shown, for the first time, that deep learning applied to automate the feature extraction from raw single-channel EEG can achieve a better and the similar scoring performance compared to the state-of-the-art hand-engineering approaches on the Sleep-EDF and MASS datasets respectively.

- In Chapter 6, we propose a novel data-driven approach to detect desynchronization among biosignals from different domains without utilizing any annotations in the dataset. We demonstrate its usefulness via an example application of eye-movement detection. We show that our cross-domain signal transcription model can convert from two EOG signals into the expected, simultaneous EEG signals that are not random, and only linearly correlated with the target channel. We also demonstrate that, without prior knowledge, the transcription model can help capturing the relationship between EOG and EEG signals that are missed by the conventional approaches, which can be utilized to detect eye-movements as desynchronization events. Our approach also provides an interpretable way of detecting desynchronization, in which the input simultaneous signals are deemed desynchronized if the degree of synchrony between the real and the synthesized, expected, synchronized signals is less than a threshold.

As most of our proposed methods are based deep learning, it also offers practical benefits in addition to

automating the process of hand-engineering features. Firstly, our models rely on relatively simple and homogeneous computations represented in layers, which can reduce the coding complexity. Secondly, they also can utilize computation power of Graphical Processing Units (GPUs) to run the algorithm efficiently. We also develop an opensource library, named TensorLayer [160], that provides rich data processing, model training, and serving modules aimed at helping researchers and engineers efficiently develop deep learning systems. We hope that many of the architectural elements featured in our proposed models can be reused to solve problems that have similar properties and help inspire future works.

8

Future Works

Even though we have developed methods that can automatically extract features that are useful for particular tasks and can detect desynchronization as anomaly without utilizing any annotations, it is clear that many challenges remain before we can realize our methods in the real-world clinical applications. Here, we discuss the remaining challenges and suggest potential future works to address them:

- We have demonstrated that our DeepSleepNet model can generalize to two datasets that have different properties without altering the model architecture. The remaining challenge would be to apply DeepSleepNet on the single-channel EEG recorded from wearable devices. This may include the additional experiments to compress the network size such that it can be executed on the wearable devices that have low computation power, and clinical trials to verify the model in the clinical uses.

- With the promising performance of deep learning in automatic feature learning in Chapter 5, we can apply the similar architecture to detect or predict epileptic seizures in the supervised setting. We might be able to learn new interesting features extracted by the network, which can help us to have a better understanding of the characteristic of epileptic seizures.

- We have demonstrated that our cross-domain signal transcription model can transcribe two EOG signals into its, expected simultaneous single-channel EEG signals. We can further introduce a random noise vector as an additional input to the model such that the model can synthesize more than one possible expected, simultaneous EEG signals to be compared with the measured ones.

Also, we can further extend the model to synthesize more than one signals, which is typically the case for the analysis of brain signals. This would require future work to develop a way to compare many possible expected signals to the measured ones.

- We have demonstrated one possible application of the transcribed signals via desynchronization detection. Currently, we only apply Pearson's R coefficient to compute the similarity between the transcribed and the measured signals. We can further incorporate additional metrics to quantify the degree of synchrony by also measuring the synchrony in the frequency-domain, such as frequency-spectral-based metrics. This can provide additional information about the differences between two signals, which may help improve the desynchronization detection performance.

- Another possible application of the proposed approach to detect desynchronization is to detect epileptic seizures. We can further extend our transcription model to transcribe, probably, from multiple-channel EEG signals into single-channel ECG to detect abnormal dynamics between EEG and ECG during seizures.

To make further progress, we believe that it is these kind of research areas that we should turn to next.

Bibliography

[1] American Clinical Neurophysiology Society, "Guideline 5: Guidelines for Standard Electrode Position Nomenclature," pp. 1–3, 2006. [Online]. Available: https://www.acns.org/pdf/guidelines/Guideline-5.pdf

[2] A. Atkielski, "Schematic diagram of normal sinus rhythm for a human heart as seen on ECG," 2007. [Online]. Available: https://en.wikipedia.org/wiki/QRS{_}complex{#}/media/File:SinusRhythmLabels.svg

[3] J. Malmivuo and R. Plonsey, *Bioelectromagnetism: principles and applications of bioelectric and biomagnetic fields*. Oxford University Press, 1995.

[4] W. Zijlstra and A. L. Hof, "Assessment of spatio-temporal gait parameters from trunk accelerations during human walking," *Gait and Posture*, vol. 18, no. 2, pp. 1–10, 2003.

[5] A. Karpathy, "Connecting Images and Natural Language," Stanford University, Tech. Rep. August, 2016.

[6] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout : A Simple Way to Prevent Neural Networks from Overfitting," *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014.

[7] T. Fawcett, "An introduction to ROC analysis," *Pattern Recognition Letters*, vol. 27, no. 8, pp. 861–874, 2006.

[8] Stanford University, "CS231n: Convolutional Neural Networks for Visual Recognition," 2017. [Online]. Available: http://cs231n.stanford.edu/

[9] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *Cvpr*, 2016, pp. 770–778.

[10] P. Isola, J.-Y. Zhu, T. Zhou, and A. A. Efros, "Image-to-Image Translation with Conditional Adversarial Networks," *arXiv preprint*, p. 16, 2016.

[11] R. Moe-Nilssen, "A new method for evaluating motor control in gait under real-life environmental conditions . Part 1 : The instrument," *Clinical Biomechanics*, vol. 13, no. 4-5, pp. 320–327, 1998.

[12] E. Kaniusas, "Fundamentals of Biosignals," in *Biomedical Signals and Sensors I. Linking Physiological Phenomena and Biosignals.*, E. Kaniusas, Ed. Springer, Berlin, Heidelberg, 2012, ch. 1, pp. 1–20.

[13] G. Bianchi and R. Sorrentino, *Electronic filter simulation & design.* New York: McGraw-Hill, 2007.

[14] R. W. Motl, M. Weikert, Y. Suh, J. J. Sosnoff, J. Pula, C. Soaz, M. Schimpl, C. Lederer, and M. Daumer, "Accuracy of the actibelt ® accelerometer for measuring walking speed in a controlled environment among persons with multiple sclerosis," *Gait and Posture*, vol. 35, no. 2, pp. 192–196, 2012.

[15] L. Shammas, T. Zentek, B. von Haaren, S. Schlesinger, S. Hey, and A. Rashid, "Home-based system for physical activity monitoring in patients with multiple sclerosis (Pilot study)," *BioMedical Engineering OnLine*, vol. 13, 2014.

[16] V. Bajaj and R. B. Pachori, "Automatic classification of sleep stages based on the time-frequency image of EEG signals," *Computer Methods and Programs in Biomedicine*, vol. 112, no. 3, pp. 320–328, 2013.

[17] T. Lajnef, S. Chaibi, P. Ruby, P.-E. Aguera, J.-B. Eichenlaub, M. Samet, A. Kachouri, and K. Jerbi, "Learning machines and sleeping brains: Automatic sleep stage classification using decision-tree multi-class support vector machines," *Journal of Neuroscience Methods*, vol. 250, pp. 94–105, 2015.

[18] A. H. Shoeb and J. V. Guttag, "Application of machine learning to epileptic seizure detection," *Icml*, pp. 975–982, 2010.

[19] Y. Liu, W. Zhou, Q. Yuan, and S. Chen, "Automatic seizure detection using wavelet transform and SVM in long-term intracranial EEG," *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, vol. 20, no. 6, pp. 749–755, 2012.

[20] P. Verdonck, *Advances in biomedical engineering.* Oxford: Elsevier, 2008.

[21] N. Ahammad, T. Fathima, and P. Joseph, "Detection of Epileptic Seizure Event and Onset Using EEG," *BioMed Research International*, vol. 2014, p. 7, 2014.

[22] O. Tsinalis, P. M. Matthews, and Y. Guo, "Automatic Sleep Stage Scoring Using Time-Frequency Analysis and Stacked Sparse Autoencoders," *Annals of Biomedical Engineering*, vol. 44, no. 5, pp. 1587–1597, 2016.

[23] H. Dong, A. Supratak, W. Pan, C. Wu, P. M. Matthews, and Y. Guo, "Mixed Neural Network Approach for Temporal Sleep Stage Classification," *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, 2017.

[24] M. Längkvist, L. Karlsson, and A. Loutfi, "Sleep Stage Classification Using Unsupervised Feature Learning," *Advances in Artificial Neural Systems*, vol. 2012, pp. 1–9, 2012.

[25] L. Fraiwan, K. Lweesy, N. Khasawneh, H. Wenz, and H. Dickhaus, "Automated sleep stage identification system based on time-frequency analysis of a single EEG channel and random forest classifier," *Computer Methods and Programs in Biomedicine*, vol. 108, no. 1, pp. 10–19, 2012.

[26] J. Allan Hobson, "A manual of standardized terminology, techniques and scoring system for sleep stages of human subjects," *Electroencephalography and Clinical Neurophysiology*, vol. 26, no. 6, p. 644, 1969.

[27] C. Iber, S. Ancoli-Israel, A. L. Chesson Jr., and S. F. Quan, *The AASM manual for the scoring of sleep and associated events.* American Academy of Sleep Medicine, Westchester, Illinois, 2007.

[28] R. N. Bracewell, *The Fourier transform and its applications.* McGraw-Hill, New York, 1965.

[29] L. B. White and B. Boashash, "Cross Spectral Analysis of Nonstationary Processes," *IEEE Transactions on Information Theory*, vol. 36, no. 4, pp. 830–835, 1990.

[30] E. Pereda, R. Q. Quiroga, and J. Bhattacharya, "Nonlinear multivariate analysis of neurophysiological signals," *Progress in Neurobiology*, vol. 77, no. 1-2, pp. 1–37, 2005.

[31] K. Edakawa, T. Yanagisawa, H. Kishima, R. Fukuma, S. Oshino, H. M. Khoo, M. Kobayashi, M. Tanaka, and T. Yoshimine, "Detection of Epileptic Seizures Using PhaseAmplitude Coupling in Intracranial Electroencephalography," *Scientific Reports*, vol. 6, no. 1, pp. 1–8, 2016.

[32] D. Looney, A. Hemakom, and D. P. Mandic, "Intrinsic multi-scale analysis: a multi-variate empirical mode decomposition framework," *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 471, no. 2173, 2014.

[33] Y. Zheng, G. Wang, K. Li, G. Bao, and J. Wang, "Epileptic seizure prediction using phase synchronization based on bivariate empirical mode decomposition," *Clinical Neurophysiology*, vol. 125, no. 6, pp. 1104–1111, 2014.

[34] A. Supratak, G. Datta, C. Wu, S. Yu, C. D'Arcy, R. Nicholas, Y. Guo, and P. Matthews, "Remote Actigraphy for Quantitative Assessment of Walking Speed in People with MS," *Neurology*, vol. 86, no. 16 Supplement, 2016.

[35] A. Supratak, L. Li, and Y. Guo, "Feature extraction with stacked autoencoders for epileptic seizure detection," in *36th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, 2014, pp. 4184–4187.

[36] A. Supratak, H. Dong, C. Wu, and Y. Guo, "DeepSleepNet: a Model for Automatic Sleep Stage Scoring based on Raw Single-Channel EEG," *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, pp. 1–10, 2017.

[37] A. Supratak, S. Schneider, H. Dong, L. Li, and Y. Guo, "Towards Desynchronization Detection in Biosignals," in *NIPS Time Series Workshop*, no. Nips, 2017.

[38] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[39] M. X. Cohen, *Analyzing Neural Time Series Data: Theory and Practice*. MIT Press, 2014.

[40] H. Gothwal, S. Kedawat, and R. Kumar, "Cardiac arrhythmias detection in an ECG beat signal using fast fourier transform and artificial neural network," *Journal of Biomedical Science and Engineering*, vol. 4, no. 04, pp. 289–296, 2011.

[41] G. Robertson, G. Caldwell, J. Hamill, G. Kamen, and S. Whittlesey, *Research methods in biomechanics*. Human Kinetics, 2013.

[42] Mayo Foundation for Medical Education and Research (MFMER), "Electromyography (EMG) Why it's done - Mayo Clinic." [Online]. Available: http://www.mayoclinic.org/tests-procedures/emg/basics/why-its-done/prc-20014183

[43] B. M. Sandroff, R. W. Motl, L. A. Pilutti, Y. C. Learmonth, I. Ensari, D. Dlugonski, R. E. Klaren, S. Balantrapu, and B. J. Riskin, "Accuracy of StepWatch and ActiGraph accelerometers for measuring steps taken among persons with multiple sclerosis," *PLoS ONE*, vol. 9, no. 4, 2014.

[44] R. W. Motl, J. A. Cohen, R. Benedict, G. Phillips, N. LaRocca, L. D. Hudson, and R. Rudick, "Validity of the timed 25-foot walk as an ambulatory performance outcome measure for multiple sclerosis," *Multiple Sclerosis Journal*, 2016.

[45] C. Cortes and V. Vapnik, "Support-Vector Networks," *Machine Learning*, vol. 20, no. 3, pp. 273–297, 1995.

[46] C.-c. Chang and C.-j. Lin, "LIBSVM : A Library for Support Vector Machines," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 2, pp. 1–39, 2013.

[47] H. Drucker, C. J. C. Burges, L. Kaufman, A. Smola, and V. Vapnik, "Support vector regression machines," *Advances in Neural Information Processing Systems (NIPS)*, vol. 1, pp. 155–161, 1997.

[48] Y. Bengio, "Learning Deep Architectures for AI," *Foundations and Trends® in Machine Learning*, vol. 2, no. 1, pp. 1–127, 2009.

[49] Y. Bengio, A. Courville, and P. Vincent, "Representation Learning: A Review and New Perspectives," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 35, no. 8, pp. 1798–1828, 2013.

[50] S. Wold, K. Esbensen, and P. Geladi, "Principal component analysis," *Chemometrics and Intelligent Laboratory Systems*, vol. 2, no. 1-3, pp. 37–52, aug 1987.

[51] Stanford University, "PCA - Ufldl." [Online]. Available: http://ufldl.stanford.edu/wiki/index.php/PCA

[52] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly Detection: A Survey," *ACM Computing Surveys*, vol. 41, no. 3, pp. 1–58, 2009.

[53] A. Ng, "Machine Learning (Week 9) - Anomaly Detection using the Multivariate Gaussian Distribution," 2017. [Online]. Available: https://www.coursera.org/learn/machine-learning/home/week/9

[54] J. Duchi, E. Hazan, and Y. Singer, "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization," *Journal of Machine Learning Research*, vol. 12, pp. 2121–2159, 2011.

[55] T. Tieleman and G. E. Hinton, "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude," 2012.

[56] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[57] A. P. Bradley, "The use of the area under the ROC curve in the evaluation of machine learning algorithms," *Pattern Recognition*, vol. 30, no. 7, pp. 1145–1159, 1997.

[58] M. Sokolova and G. Lapalme, "A systematic analysis of performance measures for classification tasks," *Information Processing and Management*, vol. 45, no. 4, pp. 427–437, jul 2009.

[59] A. L. Maas, A. Y. Hannun, and A. Y. Ng, "Rectifier Nonlinearities Improve Neural Network Acoustic Models," in *Proceedings of the 30 th International Conference on Machine Learning*, vol. 28, 2013, p. 6.

[60] K. He, X. Zhang, S. Ren, and J. Sun, "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification," *arXiv preprint*, pp. 1–11, 2015.

[61] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[62] Y. Zhou and R. Chellappa, "Computation of optical flow using a neural network," *IEEE International Conference on Neural Networks*, vol. 2, no. 86, pp. 71–78, 1988.

[63] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, p. 533, 1986.

[64] S. Hochreiter, "Untersuchungen zu dynamischen neuronalen Netzen," *Diploma Thesis, Technische Universität München*, 1991.

[65] Y. Bengio, P. Simard, and P. Frasconi, "Learning Long-Term Dependencies with Gradient Descent is Difficult," *IEEE Transactions on Neural Networks*, vol. 5, no. 2, pp. 157–166, 1994.

[66] I. Sutskever, J. Martens, and G. E. Hinton, "Generating Text with Recurrent Neural Networks," in *Proceedings of the 28th International Conference on Machine Learning*, no. 1, 2011, pp. 1017–1024.

[67] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training Recurrent Neural Networks," *arXiv preprint arXiv:1211.5063*, 2012.

[68] R. J. Williams and J. Peng, "An Efficient Gradient-Based Algorithm for On-Line Training of Recurrent Network Trajectories," *Neural Computation*, vol. 2, no. 4, pp. 490–501, 1990.

[69] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[70] M. Schuster and K. Paliwal, "Bidirectional recurrent neural networks," *IEEE Transactions on Signal Processing*, vol. 45, no. 11, pp. 2673–2681, 1997.

[71] A. Graves, A.-R. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, no. 6, 2013, pp. 6645–6649.

[72] P. Baldi, S. Brunak, P. Frasconi, G. Soda, and G. Pollastri, "Exploiting the past and the future in protein secondary structure prediction." *Bioinformatics*, vol. 15, no. 11, pp. 937–946, 1999.

[73] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle, "Greedy Layer-Wise Training of Deep Networks," in *Advances in neural information processing systems*, 2007, pp. 153–160.

[74] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics (AISTATS)*, vol. 9, 2010, pp. 249–256.

[75] C. Ledig, L. Theis, F. Huszar, J. Caballero, A. Aitken, A. Tejani, J. Totz, Z. Wang, and W. Shi, "Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network," *arXiv preprint arXiv:1609.04802*, 2016.

[76] S. Reed, Z. Akata, X. Yan, L. Logeswaran, B. Schiele, and H. Lee, "Generative Adversarial Text to Image Synthesis," *Icml*, pp. 1060–1069, 2016.

[77] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative Adversarial Nets," *Advances in Neural Information Processing Systems (NIPS)*, pp. 2672–2680, 2014.

[78] I. Goodfellow, "NIPS 2016 Tutorial: Generative Adversarial Networks," *arXiv preprint arXiv:1701.00160*, 2016.

[79] M. Arjovsky, S. Chintala, and L. Bottou, "Wasserstein GAN," *arXiv preprint arXiv:1701.07875*, 2017.

[80] G.-J. Qi, "Loss-Sensitive Generative Adversarial Networks on Lipschitz Densities," *arXiv preprint*, 2017.

[81] S. Chintala, E. Denton, M. Arjovsky, and M. Mathieu, "How to Train a GAN? Tips and tricks to make GANs work," 2016. [Online]. Available: https://github.com/soumith/ganhacks

[82] M. Mirza and S. Osindero, "Conditional Generative Adversarial Nets," *arXiv preprint arXiv:1411.1784*, pp. 1–7, 2014.

[83] E. Niedermeyer and F. L. da Silva, *Electroencephalography: basic principles, clinical applications, and related fields*. Lippincott Williams & Wilkins, 2005.

[84] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," *arXiv preprint arXiv:1502.03167*, 2015.

[85] R. J. Fox, A. Thompson, D. Baker, P. Baneke, D. Brown, P. Browne, D. Chandraratna, O. Ciccarelli, T. Coetzee, G. Comi, A. Feinstein, R. Kapoor, K. Lee, M. Salvetti, K. Sharrock, A. Toosy, P. Zaratin, and K. Zuidwijk, "Setting a research agenda for progressive multiple sclerosis: the International Collaborative on Progressive MS," *Multiple Sclerosis Journal*, vol. 18, no. 11, pp. 1534–1540, 2012.

[86] J. F. Kurtzke, "Rating neurologic impairment in multiple sclerosis: An expanded disability status scale (EDSS)," *Neurology*, vol. 33, no. 11, pp. 1444–52, 1983.

[87] O. R. Pearson, M. E. Busse, R. W. M. Van Deursen, and C. M. Wiles, "Quantification of walking mobility in neurological disorders," *QJM: An International Journal of Medicine*, vol. 97, no. 8, pp. 463–475, 2004.

[88] M. D. Goldman, R. A. Marrie, and J. A. Cohen, "Evaluation of the six-minute walk in multiple sclerosis subjects and healthy controls," *Multiple Sclerosis Journal*, vol. 14, no. 3, pp. 383–390, 2008.

[89] U. Givon, G. Zeilig, and A. Achiron, "Gait analysis in multiple sclerosis: Characterization of temporal-spatial parameters using GAITRite functional ambulation system," *Gait and Posture*, vol. 29, no. 1, pp. 138–142, 2009.

[90] World Health Organization (WHO), *International Classification of Functioning, Disability and Health: ICF*. Geneva, World Health Organization, 2001.

[91] N. G. LaRocca, "Impact of Walking Impairment in Multiple Sclerosis," *The Patient: Patient-Centered Outcomes Research*, vol. 4, no. 3, pp. 189–201, sep 2011. [Online]. Available: http://link.springer.com/10.2165/11591150-000000000-00000

[92] M. D. Goldman, R. W. Motl, and R. A. Rudick, "Possible clinical outcome measures for clinical trials in patients with multiple sclerosis," *Therapeutic Advances in Neurological Disorders*, vol. 3, no. 4, pp. 229–239, 2010.

[93] J. M. Huisinga, M. L. Filipi, K. K. Schmid, and N. Stergiou, "Is there a relationship between fatigue questionnaires and gait mechanics in persons with multiple sclerosis?" *Archives of Physical Medicine and Rehabilitation*, vol. 92, no. 10, pp. 1594–1601, 2011.

[94] E. Kim, J. Lovera, L. Schaben, J. Melara, D. Bourdette, and R. Whitham, "Novel method for measurement of fatigue in multiple sclerosis: Real-Time Digital Fatigue Score." *Journal of Rehabilitation Research and Development*, vol. 47, no. 5, pp. 477–484, 2010.

[95] H. Albrecht, C. Wötzel, L. P. Erasmus, M. Kleinpeter, N. König, and W. Pöllmann, "Day-to-day variability of maximum walking distance in MS patients can mislead to relevant changes in the Expanded Disability Status Scale (EDSS): average walking speed is a more constant parameter." *Multiple sclerosis (Houndmills, Basingstoke, England)*, vol. 7, no. 2, pp. 105–109, 2001.

[96] Y. C. Learmonth, D. D. Dlugonski, L. A. Pilutti, B. M. Sandroff, and R. W. Motl, "The reliability, precision and clinically meaningful change of walking assessments in multiple sclerosis," *Multiple Sclerosis Journal*, vol. 19, no. 13, pp. 1784–1791, 2013.

[97] R. W. Motl, L. Pilutti, B. M. Sandroff, D. Dlugonski, J. J. Sosnoff, and J. H. Pula, "Accelerometry as a measure of walking behavior in multiple sclerosis," *Acta Neurologica Scandinavica*, vol. 127, no. 6, pp. 384–390, 2013.

[98] M. Schimpl, C. Lederer, and M. Daumer, "Development and Validation of a New Method to Measure Walking Speed in Free-Living Environments Using the Actibelt® Platform," *PLOS ONE*, vol. 6, no. 8, pp. 1–12, 2011.

[99] R. S. Mcginnis, N. Mahadevan, Y. Moon, K. Seagers, N. Sheth, J. A. J. Wright, S. Dicristofaro, I. Silva, E. Jortberg, M. Ceruolo, J. A. Pindado, J. Sosnoff, R. Ghaffari, and S. Patel, "A machine learning approach for gait speed estimation using skin-mounted wearable sensors : From healthy controls to individuals with multiple sclerosis," *Plos One*, vol. 12, no. 6, pp. 1–16, 2017.

[100] C. Tudor-Locke, T. V. Barreira, and J. M. J. Schuna, "Comparison of Step Outputs for Waist and Wrist Accelerometer Attachment Sites," *Medicine & Science in Sports & Exercise*, vol. 47, no. 4, pp. 839–842, 2015.

[101] J. M. Balto, D. L. Kinnett-Hopkins, and R. W. Motl, "Accuracy and precision of smartphone applications and commercially available motion sensors in multiple sclerosis," *Multiple Sclerosis Journal - Experimental, Translational and Clinical*, vol. 2, pp. 1–8, 2016.

[102] J. M. Huisinga, M. Mancini, R. J. St. George, and F. B. Horak, "Accelerometry reveals differences in gait variability between patients with multiple sclerosis and healthy controls," *Annals of Biomedical Engineering*, vol. 41, no. 8, pp. 1670–1679, 2013.

[103] J. Bland and D. Altman, "Measuring agreement in method comparison studies," *Statistical Methods in Medical Research*, vol. 8, no. 2, pp. 135–160, 1999.

[104] K. A. Bollen and K. H. Barb, "Pearson's R and Coarsely Categorized Measures," *American Sociological Review*, vol. 46, no. 2, pp. 232–239, 1981.

[105] J. C. Hobart, A. Riazi, D. L. Lamping, R. Fitzpatrick, and A. J. Thompson, "Measuring the impact of MS on walking ability: The 12-Item MS Walking Scale (MSWS-12)," *Neurology*, vol. 60, no. 1, pp. 31–36, 2003.

[106] World Health Organization, *Atlas: epilepsy care in the world.* Geneva: World Health Organization, 2005.

[107] R. S. Fisher, W. van Emde Boas, W. Blume, C. Elger, P. Genton, P. Lee, and J. Engel Jr., "Epileptic Seizures and Epilepsy: Definitions Proposed by the International League Against Epilepsy (ILAE) and the International Bureau for Epilepsy (IBE)," *Epilepsia*, vol. 46, pp. 470–472, 2005.

[108] H. Witte, L. Iasemidis, and B. Litt, "Special issue on epileptic seizure prediction," *IEEE Transactions on Biomedical Engineering*, vol. 50, no. 5, pp. 537–539, 2003.

[109] M. E. Saab and J. Gotman, "A system to detect the onset of epileptic seizures in scalp EEG," *Clinical Neurophysiology*, vol. 116, pp. 427–442, 2005.

[110] P. Thodoroff, J. Pineau, and A. Lim, "Learning Robust Features using Deep Learning for Automatic Seizure Detection," in *Proceedings of the 1st Machine Learning for Healthcare Conference (PMLR)*, vol. 56, 2016, pp. 178–190.

[111] K. Polat and S. Gunes, "Classification of epileptiform EEG using a hybrid system based on decision tree classifier and fast Fourier transform," *Applied Mathematics and Computation*, vol. 187, no. 2, pp. 1017–1026, 2007.

[112] H. Adeli, S. Ghosh-Dastidar, and N. Dadmehr, "A wavelet-chaos methodology for analysis of EEGs and EEG subbands to detect seizure and epilepsy," *IEEE Transactions on Biomedical Engineering*, vol. 54, pp. 205–211, 2007.

[113] S. Ghosh-Dastidar, H. Adeli, and N. Dadmehr, "Mixed-Band Wavelet-Chaos-Neural Network Methodology for Epilepsy and Epileptic Seizure Detection," *IEEE Transactions on Biomedical Engineering*, vol. 54, pp. 1545–1551, 2007.

[114] ——, "Principal Component Analysis-Enhanced Cosine Radial Basis Function Neural Network for Robust Epilepsy and Seizure Detection," *IEEE Transactions on Biomedical Engineering*, vol. 55, pp. 512–518, 2008.

[115] S. Ghosh-Dastidar and H. Adeli, "A new supervised learning algorithm for multiple spiking neural networks with application in epilepsy and seizure detection," *Neural Networks*, vol. 22, pp. 1419–1431, 2009.

[116] L. Kuhlmann, A. N. Burkitt, M. J. Cook, K. Fuller, D. B. Grayden, L. Seiderer, and I. M. Y. Mareels, "Seizure detection using seizure probability estimation: Comparison of features used to detect seizures," *Annals of Biomedical Engineering*, vol. 37, no. 10, pp. 2129–2145, 2009.

[117] A. S. Zandi, M. Javidan, G. A. Dumont, and R. Tafreshi, "Automated Real-Time Epileptic Seizure Detection in Scalp EEG Recordings Using an Algorithm Based on Wavelet Packet Transform," *IEEE Transactions on Biomedical Engineering*, vol. 57, pp. 1639–1651, 2010.

[118] T. Gandhi, B. K. Panigrahi, M. Bhatia, and S. Anand, "Expert model for detection of epileptic activity in EEG signature," *Expert Systems with Applications*, vol. 37, no. 4, pp. 3513–3520, 2010.

[119] W. Zhou, Y. Liu, Q. Yuan, and X. Li, "Epileptic seizure detection using lacunarity and bayesian linear discriminant analysis in intracranial EEG," *IEEE Transactions on Biomedical Engineering*, vol. 60, no. 12, pp. 3375–3381, 2013.

[120] R. Yadav, M. N. S. Swamy, and R. Agarwal, "Model-based seizure detection for intracranial EEG recordings," *IEEE Transactions on Biomedical Engineering*, vol. 59, no. 5, pp. 1419–1428, 2012.

[121] M. Längkvist, L. Karlsson, and A. Loutfi, "A review of unsupervised feature learning and deep learning for time-series modeling," *Pattern Recognition Letters*, vol. 42, no. 1, pp. 11–24, 2014.

[122] D. F. Wulsin, J. R. Gupta, R. Mani, J. A. Blanco, and B. Litt, "Modeling electroencephalography waveforms with semi-supervised deep belief nets: fast classification and anomaly measurement," *Journal of Neural Engineering*, vol. 8, no. 3, p. 13, 2011.

[123] D. Wulsin, J. Blanco, R. Mani, and B. Litt, "Semi-supervised anomaly detection for EEG waveforms using deep belief nets," in *Icmla*, 2010, pp. 436–441.

[124] P. Bashivan, I. Rish, M. Yeasin, and N. Codella, "Learning Representations from EEG with Deep Recurrent-Convolutional Neural Networks," *Iclr*, pp. 1–15, 2016.

[125] M. K. Kiymik, I. Güler, A. Dizibüyük, and M. Akin, "Comparison of STFT and wavelet transform methods in determining epileptic seizure activity in EEG signals for real-time application," *Computers in Biology and Medicine*, vol. 35, no. 7, pp. 603–616, 2005.

[126] L. Logesparan, A. J. Casson, S. A. Imtiaz, and E. Rodriguez-Villegas, "Discriminating between best performing features for seizure detection and data selection," *Embc*, pp. 1692–1695, 2013.

[127] G. E. Hinton and R. R. Salakhutdinov, "Reducing the Dimensionality of Data with Neural Networks," *Science*, vol. 313, no. 5786, pp. 504–507, 2006.

[128] A. L. Goldberger, L. Amaral, L. Glass, J. Hausdorff, P. Ivanov, R. Mark, J. Mietus, G. Moody, C.-K. Peng, and H. Stanley, "PhysioBank, PhysioToolkit, and PhysioNet: Components of a New Research Resource for Complex Physiologic Signals," *Circulation*, vol. 101, no. 23, pp. 215–220, 2000.

[129] R. Raina, A. Battle, H. Lee, B. Packer, and A. Y. Ng, "Self-taught learning: transfer learning from unlabeled data," in *Icml*, 2007, pp. 759–766.

[130] J. M. Lucas, "A Modified "V" Mask Control Scheme," *Technometrics*, vol. 15, no. 4, pp. 833–847, 1973.

[131] M. Basseville, "Detecting changes in signals and systems-A survey," *Automatica*, vol. 24, no. 3, pp. 309–326, 1988.

[132] M. Zabihi, S. Kiranyaz, A. B. Rad, A. K. Katsaggelos, M. Gabbouj, and T. Ince, "Analysis of High-Dimensional Phase Space via Poincaré Section for Patient-Specific Seizure Detection," *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, vol. 24, no. 3, pp. 386–398, 2016.

[133] K. S. Eggleston, B. D. Olin, and R. S. Fisher, "Ictal tachycardia: The head-heart connection," *Seizure*, vol. 23, no. 7, pp. 496–505, 2014.

[134] D. Piper, K. Schiecke, L. Leistritz, B. Pester, F. Benninger, M. Feucht, M. Ungureanu, R. Strungaru, and H. Witte, "Synchronization analysis between heart rate variability and EEG activity before, during, and after epileptic seizure," *Biomedical Engineering / Biomedizinische Technik*, vol. 59, no. 4, pp. 343–355, 2014.

[135] K. Wulff, S. Gatti, J. G. Wettstein, and R. G. Foster, "Sleep and circadian rhythm disruption in psychiatric and neurodegenerative disease," *Nature Reviews Neuroscience*, vol. 11, no. 8, pp. 589–599, 2010.

[136] C.-S. Huang, C.-L. Lin, L.-W. Ko, S.-Y. Liu, T.-P. Su, and C.-T. Lin, "Knowledge-based identification of sleep stages based on two forehead electroencephalogram channels," *Frontiers in Neuroscience*, vol. 8, p. 263, 2014.

[137] S. Güne, K. Polat, and e. Yosunkaya, "Efficient sleep stage recognition system based on EEG signal using k-means clustering based feature weighting," *Expert Systems with Applications*, vol. 37, no. 12, pp. 7922–7928, 2010.

[138] R. Sharma, R. B. Pachori, and A. Upadhyay, "Automatic sleep stages classification based on iterative filtering of electroencephalogram signals," *Neural Computing and Applications*, pp. 1–20, 2017.

[139] A. R. Hassan and A. Subasi, "A decision support system for automated identification of sleep stages from single-channel EEG signals," *Knowledge-Based Systems*, vol. 0, pp. 1–10, 2017.

[140] O. Tsinalis, P. M. Matthews, Y. Guo, and S. Zafeiriou, "Automatic Sleep Stage Scoring with Single-Channel EEG Using Convolutional Neural Networks," *arXiv preprint arXiv:1610.01683*, 2016.

[141] Y.-L. Hsu, Y.-T. Yang, J.-S. Wang, and C.-Y. Hsu, "Automatic sleep stage recurrent neural classifier using energy features of EEG signals," *Neurocomputing*, vol. 104, pp. 105–114, 2013.

[142] F. Gers and J. Schmidhuber, "Recurrent nets that time and count," in *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks*, vol. 3, 2000, pp. 189–194 vol.3.

[143] H. Sak, A. Senior, and F. Beaufays, "Long Short-Term Memory Based Recurrent Neural Network Architectures for Large Vocabulary Speech Recognition," *arXiv preprint arXiv:1402.1128*, 2014.

[144] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the Inception Architecture for Computer Vision," *arXiv preprint arXiv:1512.00567*, 2015.

[145] W. Zaremba, I. Sutskever, and O. Vinyals, "Recurrent Neural Network Regularization," *arXiv preprint arXiv:1409.2329*, 2014.

[146] C. O'Reilly, N. Gosselin, J. Carrier, and T. Nielsen, "Montreal Archive of Sleep Studies: an open-access resource for instrument benchmarking and exploratory research," *Journal of Sleep Research*, vol. 23, no. 6, pp. 628–635, 2014.

[147] B. Kemp, A. H. Zwinderman, B. Tuk, H. A. C. Kamphuisen, and J. J. L. Oberyé, "Analysis of a sleep-dependent neuronal feedback loop: The slow-wave microcontinuity of the EEG," *IEEE Transactions on Biomedical Engineering*, vol. 47, no. 9, pp. 1185–1194, 2000.

[148] T. D. Lagerlund, "Manipulating the magic of digital EEG: Montage reformatting and filtering," *American Journal of Electroneurodiagnostic Technology*, vol. 40, no. 2, pp. 121–136, 2000.

[149] J. Cohen, "A Coefficient of Agreement for Nominal Scales," *Educational and Psychological Measurement*, vol. 20, no. 1, pp. 37–46, 1960.

[150] J. Orloff and J. Bloom, "Bootstrap confidence intervals," Massachusetts Institute of Technology, Tech. Rep., 2014. [Online]. Available: https://ocw.mit.edu/courses/mathematics/18-05-introduction-to-probability-and-statistics-spring-2014/readings/MIT18{_}05S14{_}Reading24.pdf

[151] B. S. Everitt, *The analysis of contingency tables*. CRC Press, 1992.

[152] T. G. Dietterich, "Approximate Statistical Tests for Comparing Supervised Classification Learning Algorithms," *Neural Comput.*, vol. 10, no. 7, pp. 1895–1923, oct 1998.

[153] T. J. Leeper, "Permutation test." [Online]. Available: http://thomasleeper.com/Rcourse/Tutorials/permutationtests.html

[154] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, L. Kaiser, M. Kudlur, J. Levenberg, D. Man, R. Monga, S. Moore, D. Murray, J. Shlens, B. Steiner, I. Sutskever, P. Tucker, V. Vanhoucke, V. Vasudevan, O. Vinyals, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems," *arXiv preprint arXiv:1603.04467*, 2015.

[155] A. Karpathy, J. Johnson, and L. Fei-Fei, "Visualizing and Understanding Recurrent Networks," *arXiv preprint arXiv:1506.02078*, 2015.

[156] T. M. Cover and J. A. Thomas, *Elements of Information Theory 2nd Edition*. Wiley, 2006.

[157] F. Varela, J.-p. Lachaux, E. Rodriguez, and J. Martinerie, "The brainweb: phase synchronization and large-scaleintegration," *Nature Reviews Neuroscience*, vol. 2, no. 4, pp. 229–239, 2001.

[158] Y. Taigman, A. Polyak, and L. Wolf, "Unsupervised Cross-Domain Image Generation," *Iclr*, 2017.

[159] H. Zhang, T. Xu, H. Li, S. Zhang, X. Huang, X. Wang, and D. Metaxas, "StackGAN: Text to Photo-realistic Image Synthesis with Stacked Generative Adversarial Networks," *arXiv preprint*, 2016.

[160] H. Dong, A. Supratak, L. Mai, F. Liu, A. Oehmichen, S. Yu, and Y. Guo, "TensorLayer: A Versatile Library for Efficient Deep Learning Development," in *ACM Multimedia*, no. 1, 2017, pp. 1–4.