Signal Processing and Analytics

of Multimodal Biosignals

Koh Bee Hock David

Work submitted to Newcastle University
for the degree of Doctor of Philosophy in the Faculty of Science,
Agriculture and Engineering

October 2019

Abstract

Biosignals have been extensively studied by researchers for applications in diagnosis, therapy, and monitoring. As these signals are complex, they have to be crafted as features for machine learning to work. This begs the question of how to extract features that are relevant and yet invariant to uncontrolled extraneous factors.

In the last decade or so, deep learning has been used to extract features from the raw signals automatically. Furthermore, with the proliferation of sensors, more raw signals are now available, making it possible to use multi-view learning to improve on the predictive performance of deep learning.

The purpose of this work is to develop an effective deep learning model of the biosignals and make use of the multi-view information in the sequential data. This thesis describes two proposed methods, namely:

(1) The use of a deep temporal convolution network to provide the temporal context of the signals to the deeper layers of a deep belief net.
(2) The use of multi-view spectral embedding to blend the complementary data in an ensemble.

This work uses several annotated biosignal data sets that are available in the open domain. They are non-stationary, noisy and non-linear signals. Using these signals in their raw form without feature engineering will yield poor results with the traditional machine learning techniques. By passing abstractions that are more useful through the deep belief net and blending the complementary data in an ensemble, there will be improvement in performance in terms of accuracy and variance, as shown by the results of 10-fold validations.

Author's Declaration

This thesis is submitted to Newcastle University for the degree of Doctor of Philosophy. The research detailed within was performed between the years 2012-2018 and was supervised by Dr W. L. Woo and Prof. Satnam Dlay. I certify that none of the material offered in this thesis has been previously submitted by me for a degree or any other qualification at this or any other university.

Acknowledgements

Acknowledgement is hereby given to my supervisor Dr W. L. Woo for advising and supporting my work in this PhD program for the many years that it has taken.

I am also grateful to the sponsorship provided by Nanyang Polytechnic and the grant of leave for the study.

Thanks are due to my colleagues at Embedded Technology Centre for providing me with the computer, software and technical support during the course of this work.

Lastly, I would like to thank my family for their patience and tolerance during this time.

Table of Contents

List of Tables

List of Figures

Chapter 1. Introduction

1.1 Purpose

Computers can learn patterns from data, such as analysing brain waves to predict if the eyes are open or not. Once trained, the models can work well with future and yet-to-be-seen data. With the support of technologies in sensors, networks and cloud computing, the users can benefit from the analytics quickly. Incorporating signal processing and analytics into a solution can thus create compelling value proposition.

In the biomedical domain, numerous researchers have studied biosignals extensively for applications in diagnosis, therapy and monitoring. As these signals are complex, the signals have to be characterised and crafted as features for machine learning to work. In the absence of domain knowledge, this begs the question of how to extract relevant features that are invariant to uncontrolled extraneous factors in time and scale.

In the last decade or so, deep learning has been used to extract features from the raw signals automatically. With the proliferation of sensors, more raw signals are now available, making it possible to improve generalization performance by making use of the complementarity in the measurements.

The purpose of this work is to apply deep learning and multi-view learning to the predictive modelling of biosignals that are time series data. This thesis proposes two novel methods. They are first set as hypotheses and then validated with data experiments. They are, namely:

(1) The use of a deep temporal convolution network to provide the temporal context of the signals to the deeper layers of a deep belief net.
(2) The use of multi-view spectral embedding to blend the complementary data in an ensemble.

This work uses several annotated biosignal data sets that are available in the open domain. They are time series data, measured in discrete time steps, numeric in values, and non-stationary in distribution. The generative models of the phenomenon and the factors of variation are assumed unknown. With shallow static networks, these signals, in their raw form (without feature

engineering), will yield poor results. By passing abstractions that are more useful through the deep belief net and blending the complementary data in an ensemble, there will be improvement in performance in terms of accuracy and variance, as shown by the results of 10-fold validations and the significance level of the Student's paired t-test.

1.2 Background

This section provides some concepts of signal processing and analytics, particularly on machine learning, so that the hypotheses that assume this background knowledge can be better understood when they are described in the next section.

A signal, such as the brain wave, is a time-varying quantity that represents some phenomenon in the real world. Nowadays, sensors collect signals which are stored digitally, and so the term "signal" is often used interchangeably with the term "time series".

Biosignals are signals generated by living organisms. Living organisms can generate the signals either by themselves, or by interaction with their environment. They provide valuable information about the subjects. Examples of biosignals include physioelectrical signals, motion signals, sounds, etc.

The notion of analysing biosignals is daunting. The waveforms look random with no obvious patterns. However, they are predictable (from the point of view of machine leaning), unlike the white noise. There is a relationship between the neighbouring observations in a time series. They can be hard to see, but time-varying, i.e. temporal, patterns do exist in the biosignals.

To complicate the matter, biosignals often come as a set of signals rather than a single signal. For example, in brain waves, multiple signals are measured concurrently at different spatial locations, forming multivariate signals (Figure 1.1). It is also possible that, for the same context, sensors of different types collect signals that are related, forming heterogeneous signals, such as the accelerometer and gyroscope signals of a moving person. Furthermore, different latent modes may exist in a data set, such as the signals from different subjects. The content between the signals in a data set can supplement and/or complement each other. With data fusion, it is possible to increase the discriminatory information and reduce the noise in the signals in the data set.

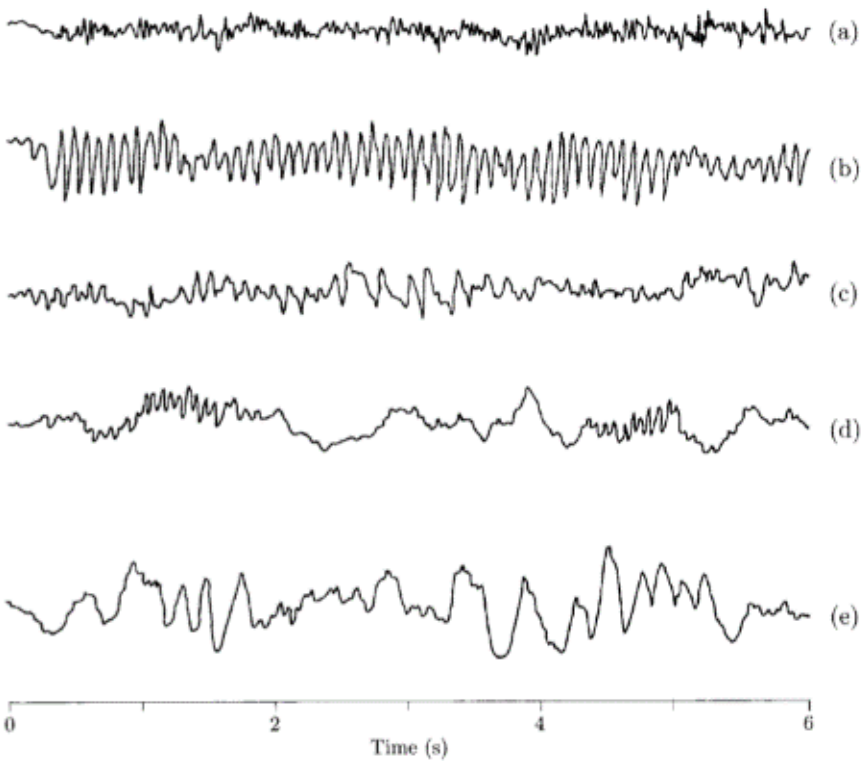

Figure 1.1. Five types of brain waves: (a) excited, (b) relaxed, (c) drowsy, (d) asleep, (e) deep sleep. Reproduced from [1].

However, without the proper context, domain knowledge, and a good dose of practical skills in statistics, the biosignals will be too complex to be analysed and run with algorithms. For example, in a study [2], a set of brain waves were collected using 14 electrodes. 42 different machine learning algorithms in the WEKA toolkit [3] were used to train and test the data set. The purpose was to predict whether the eyes of the subject were open or not. The average classification error was 44.9%. This is a surprisingly high figure. Even classifiers with a proven track record in classification, such as support vector machine and neural network, produced rather poor results of over 30% classification error (Figure 1.2).

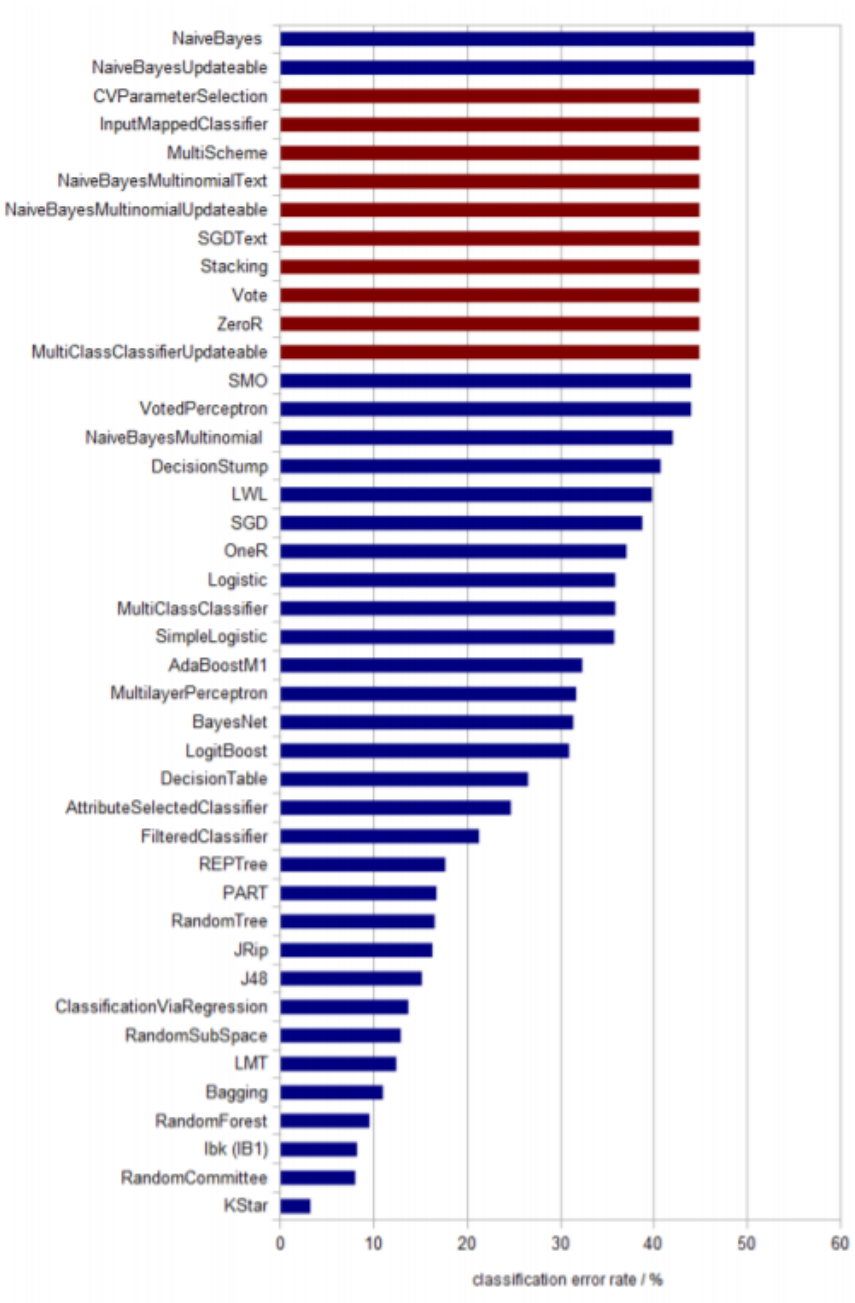

Figure 1.2. Classification error rate of 14-channel EEG. Reproduced from [2].

The poor performance is because biosignals are highly varying in both time and scale. The number of variations in a biosignal is much higher than the limited number of training instances (i.e. the input vectors). The high-dimensionality of the signal, even if it is short, exacerbates the problem. For example, the total number of possible combinations for a short time sequence of 10 samples, each sample represented by an 8-bit number, is $2^{8\times10}$. In contrast, a typical data set will only have a few thousand instances.

A highly varying signal needs a complex function to fit it. However, this often leads to the overfitting of the function to the noise in the data. The following methods may overcome the problem: (1) signal pre-processing to account for the variations, for example, de-trending, curve-fitting, etc., (2) extract features that are invariant in time and scale, based on what is important in the signal, (3) arrange the signal in short time segments, so that the distribution is quasi-stationary, and (4) regularize the model to reduce the effective complexity of the model. These methods can be demanding on the data analyst, as they require a fair amount of domain knowledge and statistical skills.

This is where deep learning, or as some would have said, re-labelled neural network, proves to be helpful. An early evidence of the efficient representation of a highly varying function by a deep network was provided by the depth-breadth trade-off in the design of Boolean circuits [4]. Not only can a deep network approximate the function with an exponentially lower number of training parameters compared to a shallow network with an equivalent approximation accuracy, it seems to be immune to overfitting too [5].

The key idea in deep learning is the *composition* of functions that distributes the features over multiple layers. At each successive layer, the true factors are disentangled from the raw data better.

Compositional functions are a common phenomenon in many natural signals such as image, text and speech [6]. They have the property of locality, which means that the features formed by the neighbouring points are related to one another at different time and scales. The success of deep network is due to the good fit of these types of signals to the deep learning model. It works well with images, and with some variations, will work well with natural time series such as biosignals too. This understanding motivates the use of deep learning in this thesis for the biosignals.

1.2.1 Bias versus Variance in Machine Learning

Machine learning, including deep learning, consists of two parts – (1) prediction/classification (also known as inference), and (2) statistical learning of the model parameters. The aim of statistical learning, given a training set (a sample from the population), is not to attain perfect

classification result with the training data, but to approximate the new and unseen data coming from the underlying population.

The approximate mapping from the input vector $\boldsymbol{x}$ to the output y, where y is the target class label, is represented by the function $f(\boldsymbol{x};\boldsymbol{\theta})$, where it is called a model. The model represents the population, even though it is trained by the sample. If the values of the model parameters $\boldsymbol{\theta}$ are known, the model is a trained model. A trained model can be used to predict/classify/infer y, given $\boldsymbol{x}$.

$$\boldsymbol{x} \mapsto f(\boldsymbol{x};\boldsymbol{\theta}) \approx y \tag{1.1}$$

Different algorithms make different assumptions about the functional form of $f(\boldsymbol{x};\boldsymbol{\theta})$ and how it can be learnt. Variations in the form and learning of $f(\boldsymbol{x};\boldsymbol{\theta})$ in the algorithms may result in better or worse approximation of the underlying function.

The adjustable model parameters $\boldsymbol{\theta}$ are learnt by minimizing an error function over the N data pairs $(\boldsymbol{x}_i, y_i), i \in \{1, \dots, N\}$, in the training set. As shown in Equation (1.2), the error function typically consists of two parts, the training loss and the regularization.

$$E\left((\boldsymbol{x}_1, y_1), \dots, (\boldsymbol{x}_N, y_N); \boldsymbol{\theta}\right) = \left\{\frac{1}{N}\sum_{i=1}^{N} loss\big(f(\boldsymbol{x}_i;\boldsymbol{\theta}), y_i\big)\right\} + reg(\|f(\boldsymbol{x};\boldsymbol{\theta})\|) \tag{1.2}$$

Three types of errors exist in the approximation - bias, variance, and irreducible error, i.e. the error that is intrinsic in the problem and cannot be generalized. Equation (1.3) below shows the bias variance decomposition of the error.

$$error^2 = bias^2 + variance + irreducible\ error^2 \tag{1.3}$$

The cause of bias is underfitting. In underfitting, the algorithm assumes an inaccurate functional form, such as the use of a model that is too simple. When there is underfitting, the training data (as well as the test data) will have a high error rate.

The cause of variance is overfitting. In overfitting, the model fits to the noise in the training data. This is often due to the use of a model that is too complex for the data. If there is overfitting, the out-of-sample problem will occur - the performance on the out-of-sample test data will be poor despite the good performance on the in-sample training data. Another symptom of overfitting is the fluctuating error rates over repeated trials (epochs and validation folds) during testing.

The regularization term in Equation (1.2) controls the effective complexity of the model. More formally, the complexity can be stated as the Vapnik-Chervonenkis (VC) dimension, where for a neural network with V nodes and E weights, the VC dimension is at least $\mathcal{O}(|E|^2)$ and at most $\mathcal{O}(|E|^2 \cdot |V|^2)$ [7]. There are many regularization methods in the literature, such as parameter norm penalties, data set augmentation, early stopping, dropout, multi-task learning etc. [8].

In theory, it is possible to reduce both the bias and the variance by using a larger data set, together with a more complex model. This is because a complex model will have a large number of adjustable parameters and so has an expressive hypothesis space to accommodate the complex patterns in the signals.

In practice, due to the limited number of training instances, it is not possible to train a complex model without overfitting, so a trade-off in the model complexity between the bias and the variance is often necessary.

The approach based on the Occam's razor principle is to use a simple model to avoid overfitting the problem, and then check it against the possibility of underfitting. In reality, the opposite approach is taken - a complex model is used to fit the data, and a penalty term (i.e. the regularization term in Equation (1.2)) is then added to the error function to regularize (reduce) its effective complexity.

As mentioned earlier, the adjustable model parameters $\boldsymbol{\theta}$ are learnt by minimizing an error function. Optimization minimizes the error function at the right $\boldsymbol{\theta}$ values. There are three broad categories of optimization methods, namely (1) direct analytical calculation, (2) alternate

optimization, and (3) gradient descend method (and the various versions based on it, such as the momentum, line search, and conjugate gradient method). The latter two methods are applicable to non-convex optimization where there exists multiple local minima with respect to the parameters $\boldsymbol{\theta}$ of the associated cost function.

The simplest algorithm for statistical learning as mentioned in the literature is density estimation. In density estimation, a simple functional form, such as the multivariate Gaussian distribution or a mixture of Gaussian distributions [9], is used to fit the data. Fitting the probability density function to the data means determining the values of the parameters $\boldsymbol{\theta}$ of the distribution. It is an unsupervised learning process for each of the classes. Optimization such as expectation maximization can determine the parameters $\boldsymbol{\theta}$. Once the class-conditioned probability density distribution is fitted, the probability of a test data instance will be able to be obtained from it. The class that has the highest posterior probability will become the predicted class.

However, density estimation based on a finite data set with a large number of attributes (where large can be as low as $d = 10$) is prone to underfitting. Therefore, density estimation is hardly useful for machine learning. This is because real world data used in machine learning are scattered very sparsely in a high dimensional space. As a result, the data distribution and the underlying function are far more complex than the shape of a Gaussian distribution, or even a mixture of Gaussian distributions. In general, a global model with a single formula holding over the entire data space is not likely to fit the underlying function well.

To avoid underfitting the underlying function, structures that are more powerful, such as generalized linear models, decision trees, nearest neighbours and neural networks, should be used whenever the number of attributes is large and the interaction between them is complicated. The function $f(\boldsymbol{x}; \boldsymbol{\theta})$ represented by these machine learning structures are universal approximation functions. They can approximate, with differing performances, the underlying function of any dimension by increasing the number of parameters linearly with the number of attributes in the data set. These are used in a wide range of applications, from optical character recognition to product recommendation.

1.2.2 Deep Learning

Deep network is a form of machine learning. It can overcome the problem of generalizing a highly varying function in a noisy data set. Many different kinds of structures have been used for deep learning, but they all share a common "stack" architecture where many different functions are composed together. In general, a model with three or more hidden layers is considered deep, although there exists some quantitative measures of "deepness" in the literature, such as the concept of Credit Assignment Paths (CAP) [10].

In composition, the output of a layer is used as the input of the next layer. For example, if there are three functions f, g, and h, their composition will be $h\left(g\left(f(\boldsymbol{x})\right)\right)$, where $\boldsymbol{x}$ is the input layer, f is the first layer, g is the second layer, and h is the third layer. Note that this is not the same as the multiplication of the outputs of f, g and h.

Each function learns a set of adjustable parameters that represents the features. The composition of functions distributes these features across the layers. Simple features at the lower layers are recombined into higher order features at the deeper layers. This allows the underlying function to be represented compactly and yet provides a large number of possible combinations for a rich form of generalization.

The most basic argument on why deep learning works is that 'when a function can be compactly represented by a deep architecture, it will need a very large architecture to be represented by an insufficiently deep one' [5]. This means that a deep architecture can compactly represent a highly varying function that would otherwise requires a very large shallow machine learning network.

In practice, deep network may encounter computational issues that prevent it from being effective. Standard gradient-based training from the random initialization of weights of a neural network often yields poor result when three or more layers are used. The first successful attempt to train deep multi-layer neural networks was the deep belief net (DBN) [11] in 2006. It makes use of greedy layer-wise unsupervised pre-training of weights where each layer is deemed to be a restricted Boltzmann machine (RBM). The proper initiation of weights places the network

parameters in a flat and smooth region of the cost function, alleviating the problem of exploding or vanishing (go to zero or overflow) gradient.

Other common architectures used in deep learning, besides DBN, are convolutional neural network (CNN) and long short-term memory (LSTM) recurrent neural network. They can be used as modules in a combination of networks to improve the system performance, either in series to form a sequence of layers, or in parallel to form an ensemble of sub-models.

1.2.3 Data Preparation and Signal Processing

With the "just show me the data" approach espoused by deep learning, less or even no feature engineering may be required by machine learning. Will deep learning really dispense with signal processing and cause it to die? This is up for debate. While it is true that an interesting and important characteristic of deep learning is its ability to extract features directly from the raw signals, how data are represented is still intricately related to the function $f(\boldsymbol{x};\boldsymbol{\theta})$ assumed by the model. For example, if the two-dimensional CNN model is used, then it would be helpful if the signal is represented in the two-dimensional time-frequency representation rather than the one-dimensional time series representation. As such, to make the data more suitable for the algorithm and give the performance a lift, some signal processing should still be applied to the raw data at the pre-processing stage of deep learning.

Figure 1.3 below shows the machine learning pipeline used in this work. The first stage involves understanding the data characteristics and setting the target performance based on the baseline performance. The intermediate stages (data preparation and signal processing) pre-process the data and put them into the expected structured data format of the machine learning algorithm. The final stage develops and validates the machine learning algorithm with the data set.

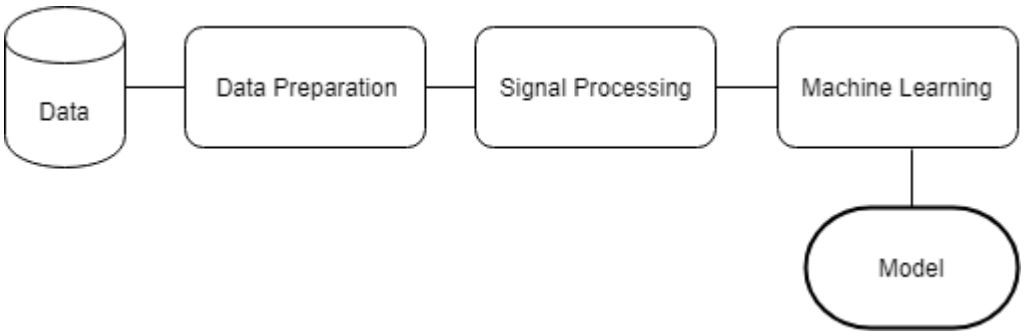

Figure 1.3. Machine learning pipeline

The distinction between machine learning and signal processing is that the former contains adjustable parameters $\boldsymbol{\theta}$ whose values are set as part of the training process while the latter is some fixed transformation of the data. Signal processing are usually complex linear operations, such as Fourier transform. Simple non-linear operations, such as the grouping of data by sliding window, or the early fusion of data by concatenation, are too simple to be called signal processing and are referred to as data preparation.

The objective of data preparation and signal processing is to change the signal $\boldsymbol{x}$ to a format $\boldsymbol{\phi}(\boldsymbol{x})$ that is easier for learning to take place. When the change $\boldsymbol{\phi}$ is specific to a particular type of data, it is known as feature engineering. Feature engineering includes feature extraction and feature selection. While feature engineering can yield better performance, it is tedious and is often impossible due to the lack of domain knowledge.

Even without specific feature engineering, generic pre-processing should still be carried out, such as the tasks listed below:

(1) Impute missing values and remove outliers.
(2) Standardize or normalize the attributes so that the scale of the data will not affect the algorithm's performance.
(3) Convert the categorical variables and/or the target class labels to one-hot encoding, so that the algorithm can classify the data.
(4) Transform the signal to its spectrogram so that the patterns in both the time and frequency domain can be exposed to the algorithm.

(5) Select a subset of relevant features and discard the rest, either by univariate statistical test, recursive feature selection or model-based feature selection.
(6) Group the data into overlapping segments in the time delay representation to allow the learning of the temporal patterns in the segments.

The end result of pre-processing is the placement of the natural signals in the structured data format as expected by the algorithm. This is usually a two-dimensional data table. Each row in the data table is an input vector, corresponding to a data instance as seen by the algorithm. The input vectors are assumed independent and identically distributed according to the underlying distribution.

One last step in pre-processing, before passing the input vectors to the algorithm, is to shuffle the input vectors. This is an indispensable step. Without shuffling, the algorithm will learn the simple output class patterns instead of the actual input patterns. Very poor performance will result when the wrongly trained model is used to predict the new and unseen test data.

1.3 Hypotheses

Connecting the abovementioned background concepts can help to build new ideas on how to analyse the biosignals. In the scientific method, these ideas form the hypothesis that will need to be validated and tested later. It is in this vein (hypothesis followed by validation and testing) that this thesis proposes two variations in the form and learning of the deep learning function, as described in the following sub-sections.

1.3.1 Deep Temporal Convolution Network

The first hypothesis in this thesis is that the generalization performance will improve with the use of a deep non-recurrent network to provide more of the temporal context of the signals for abstraction by the deeper layers of a deep belief net (DBN).

The proposed deep temporal convolutional network exploits the compositional locality of the biosignal at each level of the architecture. The purpose is to 1) improve generalization performance by modelling the temporal context at each of the layers of the network, 2) classify the patterns with shift-invariance, and (3) reduce the time resolution in time series classification.

In term of architecture, the deep temporal convolutional network extends from the DBN by inserting a concatenation sublayer in each of the deeper layers of the DBN. The learning of the network parameters is by backpropagation, with gradient routing introduced between the concatenation sublayer and its parent layer.

The motivation for inserting the concatenation sublayer is for it to provide more temporal context to the next hidden layer in the network. Figure 1.4 below illustrates this concept. The left hand side shows a mini-batch of eight output vectors #1 to #8 of a hidden layer. The right hand side shows the new mini-batch, consisting of six concatenated vectors, i.e. vectors formed by the concatenation operation. The concatenated vectors will subsequently become the input vectors of the next hidden layer. By doing so, the receptive field of the next hidden layer will become larger and contain more temporal context, making a richer form of generalization possible. (As an aside, this thesis uses the term "mini-batch" to differentiate it from the term "batch" as used in batch gradient descend, where it refers to the entire data set.)

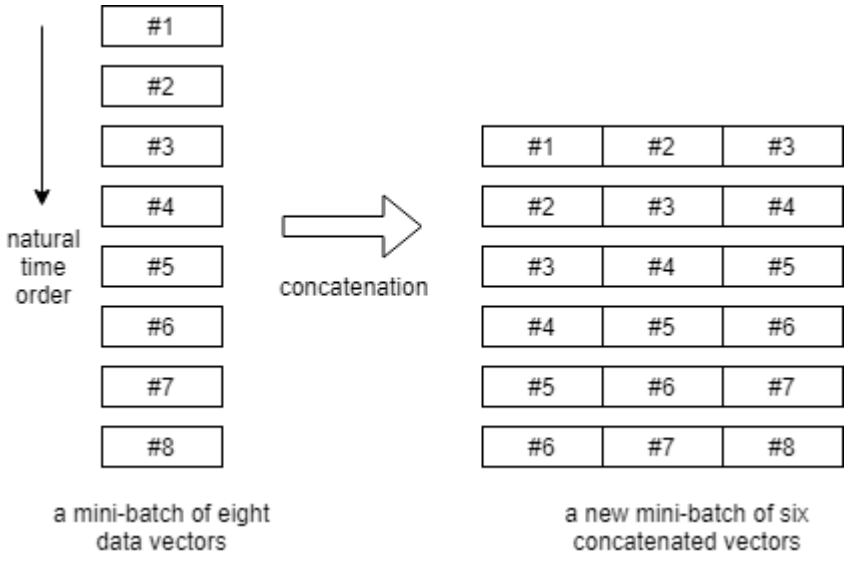

Figure 1.4. Concatenation of output vectors for the deeper layer

Each of the data vectors #1 to #8 in Figure 1.4 above are short time sequences. They are the product of a window sliding over a time series, with some overlap in time depending on the stride of the window. This arrangement, or time delay representation, results in shift invariance within the mini-batch. Due to the overlapping of the short time sequences within the mini-batch, the temporally-ordered data vectors may contain the same temporal pattern at slightly different

time points. The error gradients of these data vectors will be averaged out during the backpropagation of the mini-batch, constraining them to be equal, thus effectively ignoring the difference in time between them.

The data vectors in the mini-batch have natural time order. They are not shuffled. There is no point to concatenate data vectors that have no natural time order, as the resulting concatenated vector will not have the temporal context. In fact, shuffling the data vectors will cause the performance of the proposed model to drop drastically, as randomness, i.e. noise, is now injected into the concatenated vectors used for model training.

On the other hand, if the data instances are not shuffled, the neural network will unwittingly learn the simple output class patterns. This will lead to an extremely poor performance. In fact, this is the reason why it is a usual practice to shuffle the input vectors of the neural network.

The proposed solution to solve this dilemma (the need to maintain natural time order, and yet having to shuffle the input) is to maintain short-term temporal order within the mini-batches and then randomize the order of the mini-batches. In other words, in each mini-batch, the data instances will be kept in their natural temporal order. The order of the mini-batches is then randomized before feeding them to the training algorithm.

Another issue of the proposed model is that the first few and the last few of the data instances in the mini-batch do not contribute to the training as much as the rest of the data instances. This can be seen from the right hand side of Figure 1.4 above, where the data vectors #1 and #8 appear just once in the new mini-batch of concatenated vectors, and the data vectors #2 and #7 appear just twice, while the rest of the data vectors in the mini-batch appear three times in the new mini-batch.

Overlapping the mini-batches when they are made from the original data set can alleviate the problem of unequal contribution to training. Epoch wise, there will be an almost equal contribution from each of the data instances when the mini-batches overlap with each other.

More importantly, the overlapping of the mini-batches ensures that the short-term temporal order in the mini-batches is learnt in a shift-invariant manner. With shift-invariance, there is no need to segment the beginning and end of the mini-batches precisely.

The proposed model also has an advantage in term of the time resolution in time series classification over an equivalent network that does not make use of the temporal context in the deeper layers. In both cases, the time resolution is the length of the short time sequence at the input layer. However, the proposed model provides more temporal context to the network in the form of short-term temporal order in the mini-batches. Thus, the proposed model can make use of short time sequences that are shorter than the short time sequences of an equivalent network to achieve the same performance.

Notation wise, if the t-th output vector at layer l is denoted as $\boldsymbol{x}_t^{(l)}$, then concatenating it with the next two output vectors $\boldsymbol{x}_{t+1}^{(l)}$ and $\boldsymbol{x}_{t+2}^{(l)}$ at layer l will form a new data instance $\boldsymbol{x}_{t+2}^{(lc)}$ at the concatenation sublayer lc.

$$\boldsymbol{x}_{t+2}^{(lc)} = \left(\boldsymbol{x}_t^{(l)} \frown \boldsymbol{x}_{t+1}^{(l)} \frown \boldsymbol{x}_{t+2}^{(l)}\right) \tag{1.4}$$

The concatenation of the three consecutive data vectors at layer l, $\boldsymbol{x}_t^{(l)}$, $\boldsymbol{x}_{t+1}^{(l)}$, and $\boldsymbol{x}_{t+2}^{(l)}$, is equivalent to sliding a window of length 3 with a stride of 1 over the data vectors in the mini-batch. The result is a new mini-batch of concatenated vectors for the next layer.

The above arrangement results in weight sharing. This is because the ingredients of the concatenated vector $\boldsymbol{x}_{t+2}^{(lc)}$, which are $\boldsymbol{x}_t^{(l)}$, $\boldsymbol{x}_{t+1}^{(l)}$, and $\boldsymbol{x}_{t+2}^{(l)}$, are generated from the same set of weights. Due to weight sharing, the number of weights is smaller than what it would have been if the longer concatenated vector $\boldsymbol{x}_{t+2}^{(lc)}$ is obtained directly as $\boldsymbol{x}_t^{(l)} \frown \boldsymbol{x}_{t+1}^{(l)} \frown \boldsymbol{x}_{t+2}^{(l)}$. Weight sharing enables the proposed network to learn features that are invariant across the time dimension.

The concatenation operation added to the proposed model, as described above, will not affect the use of unsupervised DBN pre-training in the proposed model. This is because unsupervised DBN pre-training by contrastive divergence [12] is greedy layer-wise. In contrastive divergence,

no other layer in the stack of layers in the network will be involved, other than the pair of layers in the restricted Boltzmann machine. In a DBN, the pair would be layer l and layer $l+1$. In the proposed model, it would be the concatenated sublayer lc and the next hidden layer $l+1$. What about the weights between layer l and the concatenation sublayer lc? Well, there is no weight there, and so there is no need to train the non-existent weights. The concatenation sublayer lc is obtained from layer l by the concatenation operation, a simple non-linear operation that does not involve the use of weights.

However, the existence of the concatenation sublayer does cause some difficulty to the backpropagation procedure during fine-tuning (as opposed to pre-training). The concatenation operation, being a non-continuous process, is not amenable to differentiation. It stands in the backward path of backpropagation and obstructs the chain rule of differentiation used by backpropagation. Therefore, a method will have to be devised to solve the problem of backpropagation in the proposed model. The general idea of the solution is that, with the concatenation sublayer added to the proposed architecture, the error at the concatenation sublayer will have to be attributed to the parent layer, i.e. the pre-concatenation layer. This thesis will describe the proposed gradient routing method, termed "split-slide-add", in Chapter 3.

1.3.2 Multi-view Temporal Ensemble

The second hypothesis in this thesis is that the generalization performance will improve with the use of multi-view spectral embedding to blend the complementary data of the ensemble's sub-models.

In an ensemble, multiple sub-models combine to produce the final output. In such a system, the output of each sub-model is a view. Each of the views is somewhat different from the other views. The difference could be due to: (1) the inputs of the sub-models are different (i.e. the natural views are different), or (2) the configurations of the sub-models are different (i.e. the artificial views of the same data are different). Figure 1.5 below shows the architectures for these two types of views:

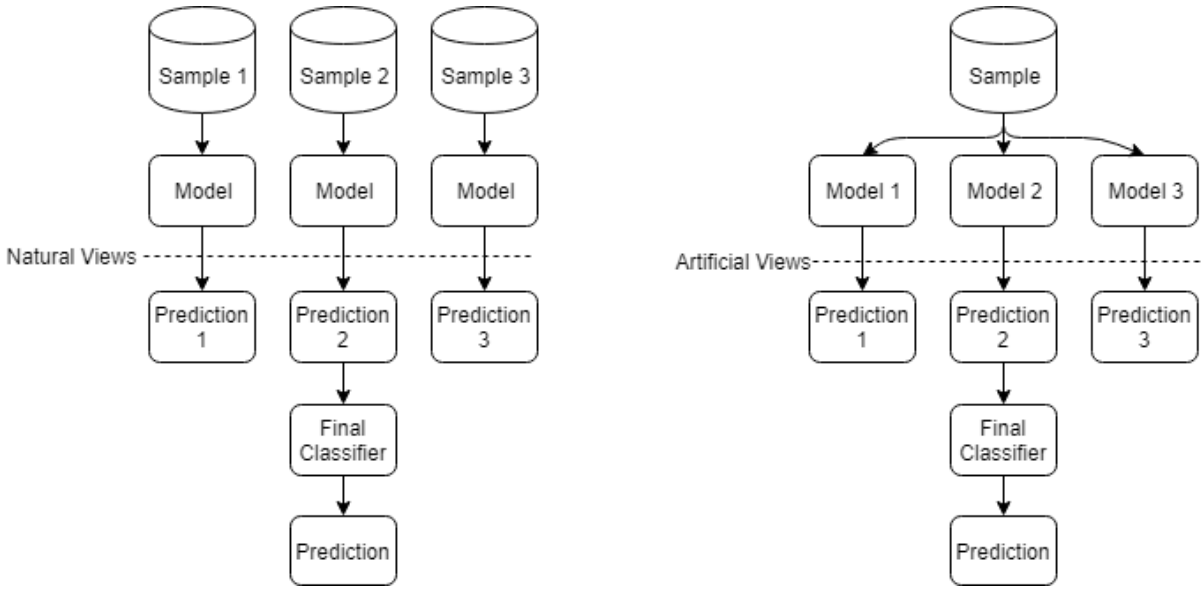

Figure 1.5. Natural (left) and artificial (right) views of multi-view ensemble

According to the Representer theorem, the approximate function $f(\boldsymbol{x};\boldsymbol{\theta})$ is a linear combination of the basis functions [13]. Thus, assuming that each view is the result of a basis function, the views can be combined linearly, with appropriate weights assigned to each of the views. Equation (1.5) shows the linear combination of M views, where each of the M views is denoted as $\boldsymbol{V}^{(i)}, i \in \{1, \dots, M\}$. The weights $\alpha_i, i \in \{1, \dots, M\}$ are the mixing coefficients of the ensemble.

$$\boldsymbol{V}_{combined} = \sum_{i=1}^{M} \alpha_i \boldsymbol{V}^{(i)} \tag{1.5}$$

In many cases, the views $\boldsymbol{V}^{(i)}, i \in \{1, \dots, M\}$ are simply deemed as independent and supplemental, in which case the logical way to blend them is to average them out with the average weight $1/M$. The supplemental nature of the independent views results in a less noisy combined output. When such a combined output is used as the input of the final classifier, the overall system performance will have a lower variance.

Instead of the average weight $1/M$, it is proposed in this work that the complementarity of the views be used as the weights of the ensemble. Complementary views are non-independent. They share some similarities (i.e. consensus), as well as some differences, with each other. By

combining the complementary views, a more complete view of the underlying phenomenon is available to the final classifier. The consensus between the views will reduce the variance. The increase in the discriminatory information will reduce the bias.

The measure of complementarity is the contribution of each of the views to a global view. Larger contribution implies that more of the complementary information is available in the view. Chapter 4 of this thesis will explain the procedure for computing complementarity. The following provides a summary of it.

To obtain the measure of complementarity, first compute the adjacency matrix of the data instances in the individual view, which is the local proximity of the data instances in the data manifold. Then compute the Laplacian matrix from the adjacency matrix. After that, merge the individual Laplacian matrices by linear combination and apply eigen-decomposition to the global Laplacian matrix. The eigenvectors are the spectral embedding of the global view. Lastly, update the weights of the linear combination by computing the cost values of preserving the local proximity of the individual views in the spectral embedding of the global view. This two-stage alternate optimization process (getting the spectral embedding of the global view, and then updating the weight values of the linear combination) will continue until there is a convergence in the weight values. The weights that converged are the measure of complementarity.

The above description makes use of a spectral embedding method called Laplacian eigenmap (LE). Laplacian eigenmap preserves the local proximity of the original data instances in the spectral embedding. This preservation is achieved by the minimization of the cost function as shown in Equation (1.6) below.

$$J(\boldsymbol{Y}) = \sum_{i,j\in\{1,\ldots,N\}} \left\|\boldsymbol{y}_i - \boldsymbol{y}_j\right\|^2 [\boldsymbol{W}]_{i,j} \tag{1.6}$$

As seen from Equation (1.6) above, the cost function $J(\boldsymbol{Y})$ is the total amount of differences between any two embedded vectors ($\boldsymbol{y}_i$ and $\boldsymbol{y}_j$) that are modulated by the distance $[\boldsymbol{W}]_{i,j}$ between their corresponding data instances ($\boldsymbol{x}_i$ and $\boldsymbol{x}_j$). When the pair ($\boldsymbol{x}_i$ and $\boldsymbol{x}_j$) is in close

proximity (i.e. the distance is small), the value of the adjacency matrix $[\boldsymbol{W}]_{i,j}$ will be large, thus contributing more to the cost function.

It was shown [14] that the solution of the above minimization problem can be reduced to

$$\boldsymbol{Y}^* = arg \min_{\boldsymbol{Y}^T\boldsymbol{D}\boldsymbol{Y}=1, \boldsymbol{Y}^T\boldsymbol{D}\boldsymbol{1}=0} \boldsymbol{Y}^T\boldsymbol{L}\boldsymbol{Y} \tag{1.7}$$

In Equation (1.7) above, the newly introduced variable $\boldsymbol{L}$ is the Laplacian matrix, and $\boldsymbol{Y} = [\boldsymbol{y}_1, \dots, \boldsymbol{y}_N]^T$ is the data matrix made up by the N embedded vectors. The Laplacian matrix is defined as $\boldsymbol{L} = \boldsymbol{D} - \boldsymbol{W}$, where $\boldsymbol{W}$ is the adjacency matrix and the diagonal element of $\boldsymbol{D}$ is the sum of the corresponding column in $\boldsymbol{W}$, i.e. $\boldsymbol{D}_{i,i} = \sum_{j=1}^{N} \boldsymbol{W}_{j,i}$.

Importantly, finding $\boldsymbol{Y}^* = arg \min_{\boldsymbol{Y}^T\boldsymbol{D}\boldsymbol{Y}=1, \boldsymbol{Y}^T\boldsymbol{D}\boldsymbol{1}=0} \boldsymbol{Y}^T\boldsymbol{L}\boldsymbol{Y}$ is equivalent to finding the eigenvectors $\boldsymbol{Y}$ corresponding to the smallest eigenvalues of the generalized eigenvalue problem $\boldsymbol{L}\boldsymbol{Y} = \boldsymbol{\lambda}\boldsymbol{D}\boldsymbol{Y}$. In other words, the eigenvectors $\boldsymbol{Y}^*$ are the solutions to the minimization of the cost function as shown in Equation (1.6) above. Thus, given the Laplacian matrix $\boldsymbol{L}$, the spectral embedding $\boldsymbol{Y}^*$ can be found. The resulting cost value is $\boldsymbol{Y}^{*T}\boldsymbol{L}\boldsymbol{Y}^*$. The lower the cost value, the easier it is to attain the objective of preserving the local proximity in the spectral embedding.

When there are more than one view, the minimization as shown in Equation (1.7) above cannot be carried out. This is because there are now two unknowns in the solution. Not only is the spectral embedding $\boldsymbol{Y}^*$ unknown, the global Laplacian matrix $\boldsymbol{L}$ is also unknown, as only the individual Laplacian matrix $\boldsymbol{L}^{(i)}$ of the i-th view is available.

Fortunately, the global Laplacian matrix $\boldsymbol{L}$ can be represented by the linear combination of the Laplacian matrix of the individual views, as shown in Equation (1.8).

$$\boldsymbol{L} = \sum_{i=1}^{M} \alpha^{(i)} \boldsymbol{L}^{(i)} \tag{1.8}$$

Still, what should be the weight values $\alpha^{(i)}$ in Equation (1.8)? It is proposed that the reciprocal of the cost value $\boldsymbol{Y}^{*T}\boldsymbol{L}^{(i)}\boldsymbol{Y}^{*}$ of the i-th view be used as the weight values $\alpha^{(i)}$. These values should be normalized so that the L_1 norm of $\boldsymbol{\alpha} = \{\alpha^{(1)}, \ldots, \alpha^{(M)}\}$ is 1.

The weight values of $\alpha^{(i)}$ are initialised to $1/M$ to kick-start the computation of the global Laplacian matrix $\boldsymbol{L}$. Once the global Laplacian matrix $\boldsymbol{L}$ is computed, the global embedding $\boldsymbol{Y}^{*}$ can be obtained by eigen-decomposition. This enables the subsequent values of $\alpha^{(i)}$ to be computed. The computation process for $\alpha^{(i)}, i \in \{1, \ldots, M\}$ with $M = 3$ is summarised in Figure 1.6 below.

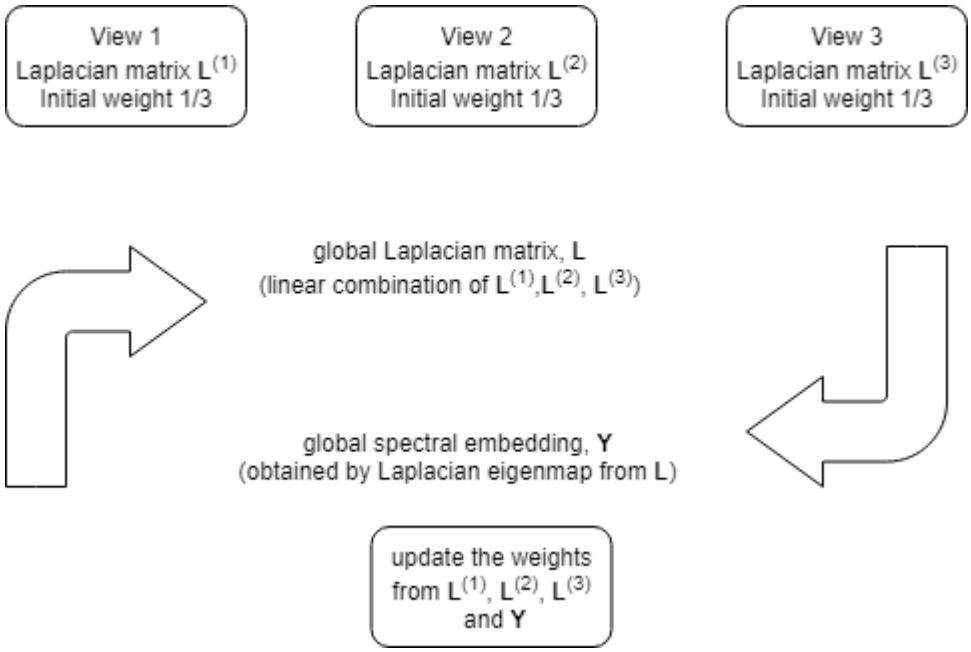

Figure 1.6. Alternate optimization of $\boldsymbol{L}$ and $\boldsymbol{Y}$

The above procedure is a form of alternate optimization. Since the cost function $J(\boldsymbol{Y})$ involves not one but two unknowns (namely, the global embedding $\boldsymbol{Y}^{*}$ and the global Laplacian matrix $\boldsymbol{L}$), they will have to be computed alternately by assuming that the value of one of the two unknowns is fixed. This leads to the eventual minimization of the cost function $J(\boldsymbol{Y})$, at which point the local proximity of the original data instances will be preserved in the global embedding.

The weight values $\alpha^{(i)}, i \in \{1, \ldots, M\}$ can be viewed as the contribution of the individual Laplacian matrix to the global embedding. The contribution is relative to the other Laplacian matrices. Thus, the larger the value of $\alpha^{(i)}$, the more complementary the view.

In order for the computed complementarity to be meaningful, the data instances in the views must be time-aligned to each other, i.e. co-occurring. The data instances need not be in their original time order, but they will have to be at the same time point relative to the other views. To ensure this, the same data set (in whichever random order) will have to be used as input by all the sub-models. Both training and testing will have to enforce the rule on co-occurrence.

The multi-view ensemble should use sub-models based on deep learning, such as the proposed deep temporal convolution network or the CNN-LSTM sub-models. This is because the output of deep learning sub-models are smooth in the data manifold, and smoothness is a requirement for spectral embedding.

The topology proposed in this thesis is to first make use of CNN in two dimensions, followed by CNN in one dimension, and eventually LSTM. To facilitate the use of CNN-LSTM as the sub-models, the input data should be decomposed into the two-dimensional time-frequency representation. This will allow both the temporal and spectral patterns to be extracted by the CNN-LSTM sub-model, thus providing good discriminatory data for multi-view learning in the multi-view temporal ensemble.

1.4 Evaluation Methods

Algorithms in machine learning, including deep learning, are black boxes. The relationship between the input and the output are not well described by any kind of theory. As such, the validity of the algorithm cannot be proved by mathematical deduction. In place of proof, the repeatability of the algorithm is validated with data sets.

The proposed deep temporal convolution network and multi-view temporal ensemble are algorithmic in nature. The validity of these algorithmic methods will have to be validated with data sets through data experiments. This section names the data sets used in the data

experiments, the performance metrics used in analysing the results of the data experiments, and the cross-validation and model comparison method.

1.4.1 Data Sets

Data analysis starts with an annotated data set and the benchmark for its performance. This work uses open source data sets from the biomedical domain because data acquisition is not in the scope of this work. The data sets used in this thesis are:

(1) EEG Eye State [2]
(2) EEG Epileptic Seizure [15]
(3) Human Activity Recognition based on Smart Phone Sensors [16]
(4) Freezing of Gait during Walking [17]
(5) EMG Lower Limb Analysis [18]
(6) Environmental Sounds [19]
(7) Heart Sounds [20]

1.4.2 Performance Metrics

In classification, the confusion matrix is the source of all the classification performance metrics, be they classification accuracy, sensitivity/specificity, recall/precision, F_1 score, etc.

Table 1.1 below shows the confusion matrix of a 3-class test set as an example of the confusion matrix of a multi-class classification problem. The terms “true” and “false” as used in Table 1.1 below refer to whether that prediction corresponds to the actual target class labels or not.

Table 1.1. Confusion matrix

		Predicted Class		
		Class 0, $p^{(0)}$	Class 1, $p^{(1)}$	Class 2, $p^{(2)}$
Actual Class	Class 0, $a^{(0)}$	$true_{class\ 0}$ $(a^{(0)}, p^{(0)})$	$false_{class\ 0}$ $(a^{(0)}, p^{(1)})$	$false_{class\ 0}$ $(a^{(0)}, p^{(2)})$
	Class 1, $a^{(1)}$	$false_{class\ 1}$ $(a^{(1)}, p^{(0)})$	$true_{class\ 1}$ $(a^{(1)}, p^{(1)})$	$false_{class\ 1}$ $(a^{(1)}, p^{(2)})$
	Class 2, $a^{(2)}$	$false_{class\ 2}$ $(a^{(2)}, p^{(0)})$	$false_{class\ 2}$ $(a^{(2)}, p^{(1)})$	$true_{class\ 2}$ $(a^{(2)}, p^{(2)})$

The performance of a classifier can be gleaned from the diagonal elements of the confusion matrix. The diagonal elements are the counts of the true predictions. The rest are the false predictions. Obviously, the larger the counts in the diagonal, the better is the classification performance.

Metrics that are quantitative in nature are needed for the evaluation of the results shown in the confusion matrix. Classification accuracy, as the most common type of classification performance metric, is defined as the count of the total number of true predictions over the total number of instances in the test set.

$$accuracy = \frac{true}{total} = \frac{\left(a^{(0)}, p^{(0)}\right) + \left(a^{(1)}, p^{(1)}\right) + \left(a^{(2)}, p^{(2)}\right)}{a^{(0)} + a^{(1)} + a^{(2)}} \tag{1.9}$$

Class-specific classification accuracy is useful in specifying the performance of the individual classes. Equation (1.10) below shows the class 0 accuracy.

$$accuracy_{class\ 0} = \frac{true_{class\ 0}}{total_{class\ 0}} = \frac{\left(a^{(0)}, p^{(0)}\right)}{a^{(0)}} \tag{1.10}$$

In the biomedical domain, a common scenario is the 2-class problem, where class 0 is normal and class 1 is abnormal. The class-specific classification accuracies for class 0 (normal) and class 1 (abnormal) are termed as specificity (Sp) and sensitivity (Se).

Equation (1.11) below shows specificity, which is class 0 accuracy. From the Bayesian perspective, it is the conditional probability of the prediction being class 0 when the actual class is class 0.

$$Sp = accuracy_{class\ 0} = \frac{\left(a^{(0)}, p^{(0)}\right)}{a^{(0)}} = P(predict = 0 | actual = 0) \tag{1.11}$$

Equation (1.12) below shows sensitivity, which is class 1 accuracy. From the Bayesian perspective, it is the conditional probability of the prediction being class 1 when the actual class is class 1.

$$Se = accuracy_{class\ 1} = \frac{\left(a^{(1)}, p^{(1)}\right)}{a^{(1)}} = P(predict = 1 | actual = 1) \tag{1.12}$$

In a multi-class problem, the class-specific performance metrics are often macro-averaged by assuming that all the classes are equally important. So, for a 2-class problem,

$$accuracy_{macro} = \frac{Sp + Se}{2} \tag{1.13}$$

There are two kinds of false predictions, namely "false positive" (also known as Type I error, or false alarm) and "false negative" (also known as Type II error, or miss). It will be more meaningful for the classification performance metric to take into consideration both these errors, instead of just the "false negative" as is the case in classification accuracy. The diagnostic testing of rare disease illustrates the importance of this point [21]. Table 1.2 shows the confusion matrix of a diagnostic testing indicating high sensitivity (99%) and high specificity (99%). The precision is poor (10%), however, as only 99 subjects tested positive are truly positive, out of 1099 subjects tested positive. This is the famous false positive paradox in Bayes theorem.

Table 1.2. An example of low precision despite high sensitivity and high specificity

		Tested		
		Normal	Has Disease	Total
Actual	Normal	99,000	1000	100,000
	Has Disease	1	99	100
	Total	99,001	1099	100,100

The macro-average of sensitivity and specificity, as shown in Equation (1.13), does not make sense with an imbalanced data set such as the rare disease case. In this case, the F_1 score should be used instead.

The F_1 score is the harmonic mean of its sub-metrics, precision and recall. Recall is the same as class-specific classification accuracy. It is defined as the true positive over all the *actual* instances. For example, for Class 1, it is:

$$recall_{class\ 1} = \frac{(a^{(1)}, p^{(1)})}{a^{(1)}} = P(predict = 1 | actual = 1) \tag{1.14}$$

Precision is defined as the true positive over all the *predicted* instances. From the Bayesian perspective, it is the posterior probability given a prediction. For example, for Class 1, it is:

$$precision_{class\ 1} = \frac{(a^{(1)}, p^{(1)})}{p^{(1)}} = P(actual = 1 | predict = 1) \tag{1.15}$$

The F_1 score of class 1 is the reciprocal of the average parallel sum of the recall and precision of class 1.

$$\frac{1}{F_{1,class\ 1}} = \frac{1}{2}\left(\frac{1}{recall_{class\ 1}} + \frac{1}{precision_{class\ 1}}\right) \tag{1.16}$$

In a multi-class problem, the macro-average of the F_1 scores of all the classes should be used, as shown in Equation (1.7) below for a 3-class problem, even though there are other schemes such as the weighted average of the F_1 scores based on the class sizes.

$$F_{1,macro} = \frac{F_{1,class\ 0} + F_{1,class\ 1} + F_{1,class\ 2}}{3} \tag{1.17}$$

1.4.3 Cross-Validation and Model Comparison

In a data experiment, multiple identical trials are run with randomly-partitioned sets of data in what is called the k-fold cross-validation. This validation process produces a set of performance scores, which is a sample from the population (of the performance scores). Based on the law of large numbers, the mean of this sample is a good approximation of the true performance of the model. It is the figure of merit for the model performance.

Since the mean of the sample is only an approximation of the true performance, it will contain some error. It is desirable to specify the confidence interval of the approximation. According to the central limit theorem, the approximation (the sample mean) is Gaussian distributed. So m repetitions of k-fold cross-validations (with new random splits of the k folds in each of the m repetitions) will produce m sample means $\bar{x}_i, i \in \{1, \dots, m\}$ that are Gaussian distributed. The standard error (i.e. standard deviation of the sample means) is given as:

$$standard\ error = \frac{\tilde{\sigma}}{\sqrt{m}} = \frac{1}{\sqrt{m}} \sqrt{\frac{1}{m} \sum_{i=1}^{m} (\bar{x}_i - \tilde{\mu})^2} \tag{1.18}$$

Note that in Equation (1.18) above, the standard error is estimated with $\tilde{\mu}$ and $\tilde{\sigma}^2$, which are the mean and variance of the m sample means. This is because the population mean μ and variance σ^2 are unknown. Note also that the standard deviation of the m sample means is divided by $\sqrt{m}$, where m is the repetitions. Thus, an approximation with 95% confidence is the mean of the m sample means ± 1.96 of the estimated standard error. This process is rather long and tedious, as it involves $m \times k$ trials. As such, it will only be used expeditiously, as deep learning can be time consuming and make it less than practical when time is of concern.

To compare the performance of one model against another, two samples of performance scores obtained by k-fold validation of the same data set can be used in a Student's paired t-test. The null hypothesis is that the two samples of performance scores are drawn from the same distribution, and any differences are due simply to statistical noise.

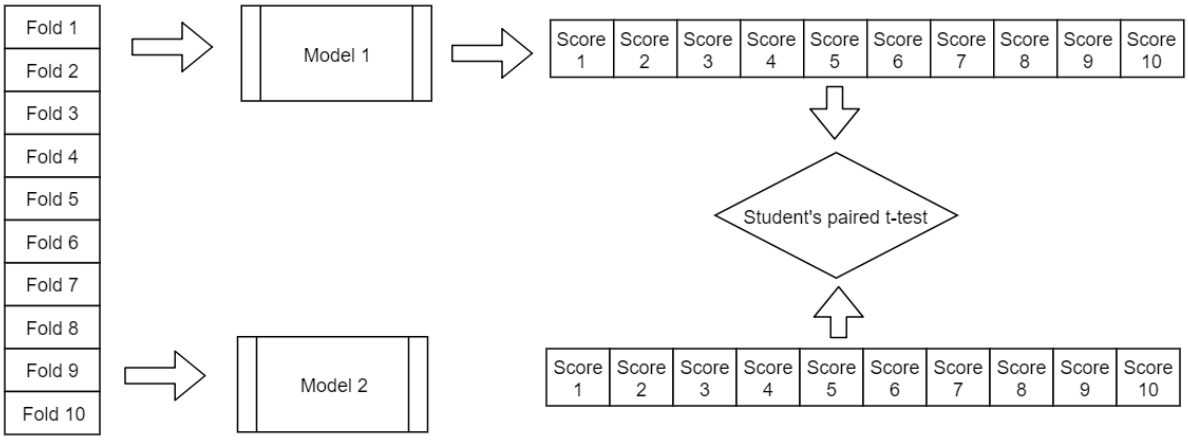

Figure 1.7. Comparing models

1.5 Organisation

This chapter covers the purpose and background of the study, followed by the two main hypotheses in this thesis. It then describes the evaluation methods used in the validation.

The next chapter will provide a literature review on biosignals, deep learning and multi-view learning. Following that, two dedicated chapters will describe the two proposed methods, first on Deep Temporal Convolution Network, and then on Multi-view Temporal Ensemble.

The fourth chapter, which is the second last chapter, describes the data and results of the data experiments that were conducted to test the effects of the hypotheses. The concluding chapter follows with a summary of the main points, and the plan for future works.

The references quoted in this thesis are available at the end of the thesis.

Chapter 2. Review

This chapter provides the systematic foundation in understanding the research topics in this thesis. It contains the background information necessary for this work, and describes some of the latest methods and techniques used to address the classification problem of time series data. It also serves as a general guide to the terms and notations used in this thesis.

This chapter starts with the explanation of what the object of interest in this thesis is all about – biosignals that are time series. The specific examples mentioned are electroencephalogram, electromyogram, inertial sensor signals and heart sounds. It highlights the characteristics of the signals to facilitate the understanding of the object of interest. This will form the basis for further discussion in the later part of the thesis on selecting and developing models for the data. This is because the model for the data is intrinsically linked to the characteristics of the data - the reason why deep learning is so effective when used on certain types of data.

Following that, the section on the deep learning of signals will describe the motivations and the principles of deep leaning. It does so in a progressive manner, starting with machine learning topics that are generally well known, such as static linear model, logistic regression, linear Gaussian model, and hidden Markov model. Along the way, it will describe a useful model called the extreme learning machine. It will then move on to the time delay neural network (TDNN) and its unrolled expansion. This is followed by a brief description of deep belief net (DBN) and the two current workhorses in deep learning – CNN and LSTM. The overarching idea espoused here is that a deep architecture is a hierarchical structure with many successive layers of non-linear operations. The front layers form the feature extractor, and the final layer is the classifier. It can be used end-to-end, with raw data fed to the network at the front end, and the prediction obtained from the network at the final layer.

The next section focusses on the latest methods and techniques used to address the problem of time series classification. It divides the methods into three broad categories, namely feature-based, distance-based and neural network-based methods. It will briefly describe some specific techniques, such as COTE (Collective of Transformation-based Ensembles).

The section on multi-view learning, or data fusion of multiple feature sets, follows the general outline of dividing data fusion techniques into early data fusion, intermediate data fusion and late data fusion techniques. It uses the concatenation of feature sets as an example of early data fusion, and ensemble learning by stacking as an example of late data fusion. It explains why an ensemble that exploits the complementarity of multi-view data will outperform algorithms based on single-view. The intuitive sense that data complementarity is useful is explained as a drop in performance when the number of attributes in a feature set is reduced, implying that the reduction has removed some of the complementary information in the features. Then, to facilitate the discussion on the complementarity of multi-view data, the concept of spectral embedding, in particular Laplacian eigenmap, will be described.

2.1 Biosignals

A biosignal is the result of interaction between the physiological process, the body volume, and the sensors. In general, it is non-stationary**,** non-linear, and noisy. However, it is not white noise. There is temporal dependence (i.e. correlation) between the values at different points in time. Because of this, they cannot be swopped in order. Due to the presence of temproal patterns in the biosignal, they can be classified by machine learning algorithms, even though they cannot be easily distinguished by humans.

As there are many biosignals, it is important that they can all be formed in a structured data format that can be used by machine learning algortithms. The time delay representation exposes the temporal patterns to the machine learning algorithms and will be discussed first. This will then be followed by other sub-sections on electroencephalogram, electromyogram, inertial sensor signals and heart sounds.

2.1.1 Time Delay Representation

The biosignals considered in this thesis have numeric data type and discrete time steps. They are time series data. Many biosignals collected by sensors fall into this category, such as physiological signals, motion signals, and sounds.

A time series is a sequence of N sampling points (i.e. observations) sampled at uniform time intervals. It is different from the time-stamped transactional data collected at no particular frequency. Notation wise, a time series is denoted as $\boldsymbol{x} = (\boldsymbol{x}_1, \dots, \boldsymbol{x}_t, \dots, \boldsymbol{x}_N)$, with the time steps being $1, \dots, t, \dots, N$. For a multivariate time series with d input variables or attributes, each sampling point, i.e. the column vector $\boldsymbol{x}_t$, will have d rows in it.

The experimenter will annotate the time series data by comparing the time series data with other information, such as the video recoding of the subjects. Annotation will result in four kinds of relationships between the sampling points and the annotated classes: (1) one-to-one, (2) one-to-many, (3) many-to-one, and (4) many-to-many. Out of these four kinds of relationship, the most natural annotation for a biosignal is the one-to-one association. It assigns each sampling point a particular state (output class). This results in an output class sequence $\boldsymbol{y} = (y_1, \dots, y_t, \dots, y_N)$ for a sequence of N sampling points, $\boldsymbol{x} = (\boldsymbol{x}_1, \dots, \boldsymbol{x}_t, \dots, \boldsymbol{x}_N)$. The process of assigning the output class to the sampling points, despite being ono-to-one for the sampling points, is actually quite efficient. This is because it can be done in relatively large chunks according to the easily observable subject's state.

In general, given a biosignal data set in time series format, the initial steps are to understand the biosignal, set the performance benchmark for the data set, choose the algorithm for the data set, and change the time series data to the structured data format expected by the algorithm. The most basic structured data format for time series data is the time delay representation.

The time delay representation reveals the temporal dependence of the sampling points along the time axis to the algorithm. So, for a multivariate signal $\boldsymbol{x} = (\boldsymbol{x}_1, \dots \boldsymbol{x}_N)$ having N sampling points, multiple short time sequences will be made from it, with each short time sequence having w sampling points, $w < N$. The way to make these short time sequences is by sliding a window of fixed length w across the biosignal. Usually, the sliding is done with a stride of length s, where the stride length s is a fraction of the window length w, i.e. $s < w$.

Sliding the window across the biosignal will result in a set of fixed-length short time sequences that overlap with their immediate neighbours. Each of the short time sequences is placed in a row in a table, forming a table of short time seuqneces. Each row in the table is a data instance

vectorized from the window of $d \times w$-dimension, where d is the number of input variables in the multivariate signal and w is the window length. The data instance has $d \times w$ features or attributes. When the data are arranged in this manner, they form a new data set and are said to be in the time delay representation, as shown in Figure 2.1 below.

$\boldsymbol{x}_1{}^T$	$\boldsymbol{x}_2{}^T$	...	$\boldsymbol{x}_w{}^T$
$\boldsymbol{x}_{1+s}{}^T$	$\boldsymbol{x}_{2+s}{}^T$	...	$\boldsymbol{x}_{w+s}{}^T$
⋮	⋮	⋮	⋮
$\boldsymbol{x}_t{}^T$	$\boldsymbol{x}_{t+1}{}^T$	...	$\boldsymbol{x}_{t+w-1}{}^T$
⋮	⋮	⋮	⋮

Figure 2.1. Time delay representation in table form

The annotation of time delay representation is obtained from the one-to-one annotation of the time series data by majority voting, where the most frequently occurring output class in the short time sequence becomes the output class of that short time sequence. The output class is represented by one-hot encoding in a vector. For example, if there are in total $k = 5$ output classes, then class 2 will be encoded as $\boldsymbol{y}_t = [0,1,0,0,0]^T$, instead of $y_t = 2$.

The time delay representation is a structured data format amenable for use by standard machine learning algorithms. The algorithm can learn the relationship between the short time sequence $(\boldsymbol{x}_t, \boldsymbol{x}_{t+1}, \dots, \boldsymbol{x}_{t+w-1})$ and the output $\boldsymbol{y}_t$. The function learnt from the data set is the classifier $g(\cdot)$. Equation (2.1) below shows the generalized linear model as an example of $g(\cdot)$.

$$\boldsymbol{y}_t = g\left(b_0 + \boldsymbol{b}_1{}^T\boldsymbol{x}_t + \boldsymbol{b}_2{}^T\boldsymbol{x}_{t+1} + \cdots + \boldsymbol{b}_w{}^T\boldsymbol{x}_{t+w-1}\right) \tag{2.1}$$

The window length w and the stride length s are the hyper-parameters of the classifier. Suitable values for w and s are chosen based on prior knowledge, or determined by sensitivity analysis methods such as the grid search. In a grid search, a suite of w and s values are used to see which one yields a better performance. At some point, there will be diminishing return. In general, the window length has to be large enough to enclose most of the possible temporal features. At the same time, it should also be small enough to provide good temporal resolution. As for the stride length, it has to be large enough to capture a portion of the temporal features in the previous window to mimic the time-shift of the temporal features. Typically, s is 50% of w. Using data arranged in this manner will result in a classifier having time-invariance in feature detection.

This means that the short time sequences need not have clear-cut start and end time with respect to the temporal features.

Sometimes, summary statistics such as minimum, mean, median, maximum etc. are extracted from the rows in the table as statistical features. In general, this should not done in deep learning, as the deep network is able to extract the features automatically. There is a loss of information with summary statistics, so the performance will deteriorate with the use of summary statistics.

The biosignal can be associated with co-variates such as the gender and age of the subject. This is known as panel data in epidermiology (the use of statistics to explain the phenomenon in the population) [22]. The co-variates are often categorical in data type rather than numeric. In such cases, the categorical data will have to be represented as indicative variables, i.e. one-hot encoding of the categorical data. The indicative variables can be placed alongside the short time sequences of the biosignal, as shown in Figure 2.2 below.

M	W	Temporal Signals	Outcome
1	0	$[\boldsymbol{X}]_{1,\cdot} = \{\boldsymbol{x}_1{}^T, , \dots, \boldsymbol{x}_w{}^T\}$	2
1	0	$[\boldsymbol{X}]_{2,\cdot} = \{\boldsymbol{x}_{1+s}{}^T, , \dots, \boldsymbol{x}_{w+s}{}^T\}$	1
0	1	$[\boldsymbol{X}]_{3,\cdot} = \{\boldsymbol{x}_{1+2s}{}^T, , \dots, \boldsymbol{x}_{w+2s}{}^T\}$	2
0	1	$[\boldsymbol{X}]_{4,\cdot} = \{\boldsymbol{x}_{1+3s}{}^T, , \dots, \boldsymbol{x}_{w+3s}{}^T\}$	1

Figure 2.2. Structured format of panel data

Figure 2.2 above shows the indicative variables for the gender (M: men, W: women) placed alongside the short time sequences generated by the sliding window. The first indicative variable M (indicated as greyed-out) is optional, as the rest of the indicative variables (in this case W) are enough to represent all the possible categories.

There is no temporal information in the co-variates, unlike the biosignal data. It is therefore a common practice to distinguish the effect of the time series data on the outcome as the random effect, and the effect of the co-variates on the outcome as the fixed effect.

With the conversion of the time series data to the time delay representation as mentioned above, most algorithms, including neural network, will be able to use them for supervised training. For applications such as the predictive modelling of biosignals, this kind of data-based approach, i.e. supervised training followed by the tesing of new and unseen data on the trained model, is more practical than the knowledge-based approach of building models from first principles and non-linear differentiation equations. A data-based approach does not require complicated mathematical modelling of the generaticve process that is specific to the problem. As long as the data set is large enough to cover the range of variation in the underlying phenomenon, the input-output mappings will form a function that can be used to predict the actual class of the new and unseen data.

2.1.2 Electroencephalogram

Besides the structure, the content of the data is also important. It is necessary to understand the information that is carried in the biosignal because this will ensure that the classes are understood correctly and the results are interpreted in line with the base facts.

The surface electroencephalogram (EEG), or the brain waves, is the "poster boy" of physiological signals. It is often used in graphics related to neurology because of its mystifying randomness. It is in fact the electrical activity of the brain as recorded by the electrodes on the scalp. There are two kinds of surface EEG, spontaneous and evoked, according to the cause of the signal. The spontaneous brain signal is the brain activity in the background, while the evoked brain signal is the brain activity due to external stimulus, such as touch or sound.

Spontaneous brain activity can be analyzed and used in many applications, such as in brain-computer interface for neuroprosthetics and the diagnosis of diseases like epilepsy, sleep disorders and dementia.

The clinical surface EEG is commonly recorded using 21 electrodes attached to the surface of the scalp, as shown in Figure 2.3. The signals are normally very weak. A typical adult human EEG signal is about 10 μV to 100 μV in amplitude when measured from the scalp.

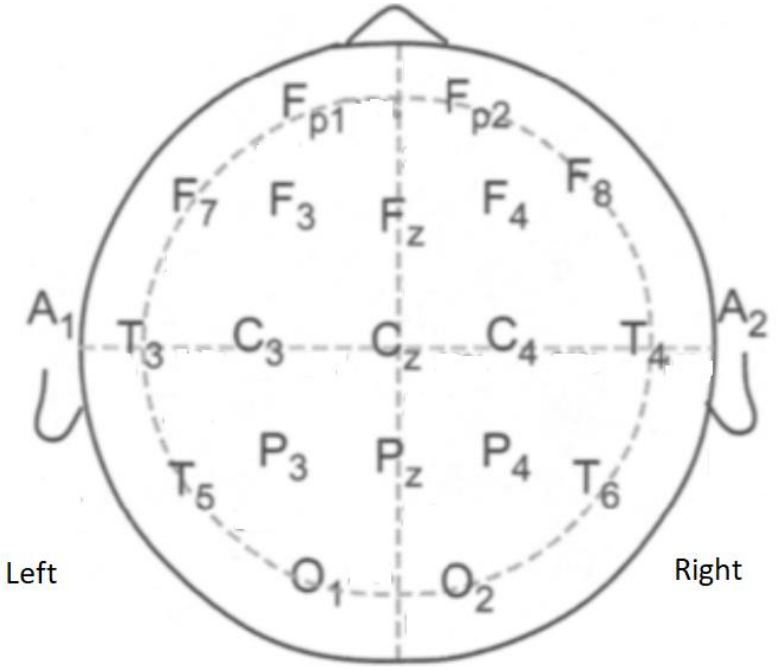

Figure 2.3. Symbols for the placements of electrodes on the scalp. Reproduced from [1].

The letters F, P, C, T, O, and A in Figure 2.3 denote the frontal, parietal, central, temporal, occipital, and auricle region of the cerebral cortex in the brain. The odd-numbered electrodes are on the left side, even-numbered electrodes are on the right side, and z (zero) is along the midline [23].

There is no single mathematical model that can fully explain the EEG rhythms. In general, the EEG rhythms depends on the mental state of the subject. When the subject is attentive, the neuron activity will be unsynchronized, resulting in low amplitude. When the subject is asleep, the neurons will be synchronized, resulting in high amplitude.

Conventionally, the EEG rhythms are classified according to the following five frequency bands:

- α alpha rhythm, 8-13 Hz. This is the most prominent EEG rhythm in a relaxed subject with the eyes closed. It is reduced when the eyes are open. The amplitude of the alpha rhythm is largest in the occipital regions.

- β beta rhythm, 14-30 Hz. This is a fast rhythm with low amplitude associated with

dreaming (rapid eye movement). It is mainly observed in the frontal and central regions of the scalp.

- γ gamma rhythm, >30 Hz. This is the EEG rhythm of an excited subject. It occurs for only a few seconds each time.

- δ delta rhythm, <4 Hz. This is a large amplitude EEG rhythm usually observed in deep sleep. It may indicate dementia if it is observed in subjects who are awake.

- θ theta rhythm, 4-7 Hz. The theta rhythm occurs during drowsiness.

Traditionally, the Rechtschaffen & Kales (R&K) rule [24] is used to analysis the sleep stages, i.e. wake, rapid eye movement sleep, S1 (light sleep), S2, S3 and S4 (deep sleep). It requires the classification all the 30-second subsequences of an eight-hour recording, which is a time-consuming task that could benefit from machine learning.

2.1.3 Electroencephalogram in Epilepsy

Besides the EEG rhythms (α, β, γ, δ, and θ), there may also be an increase in entropy and other abnormal electrical activities in epileptic patients [25].

Epilepsy is a chronic neurological disorder affecting approximately 70 million people in the world. It is characterised by unpredictable seizures. Most epileptic patients have an unknown etiology, although there are some who may be due to brain injury and genetic factors.

Most epileptic patients will only have a few seizures during a lifetime, although some may have a few dozen seizures during a single day. The duration of each seizure ranges from a few seconds to a few minutes. For most epileptic patients, the interictal state (period between seizures) corresponds to more than 99% of their life.

During the interictal period, clinicians may diagnose epilepsy by analysing the EEG trace, since there may be small spikes and sharp waves (SSWs) in the EEG when there is no observable seizure. SSWs are transient waveforms that stand out from the EEG rhythm with an irregular

temporal pattern. A spike has a duration in the range of $20 - 70\,ms$, while a sharp wave is $70 - 200\,ms$ long. Spike-wave complexes occur repeatedly, from less than 3 Hz to 6 Hz.

For subjects with suspected epilepsy, an EEG is recorded for half an hour in a relatively dark and quiet room. During this period, the subject is asked to open and close his eyes to study the changes in the EEG. At the end of that, the subject is asked to breathe rapidly and deeply and to look at a strobe light flashing at a rate of 1-25 Hz to trigger the SSWs.

2.1.4 Electromyogram

Motion signals refer to the speed and direction of a body in a scene. They represent the locomotion and other movements caused by the muscles and joints of an organism. An example of motion signal is the surface electromyogram (EMG).

EMG is the electrical activity of the skeletal muscles. Skeletal muscles are muscles attached to the skeleton, as opposed to the heart muscles and the smooth muscles. These muscles facilitate body movement and facial expression. As such, EMG can be used in many areas concerning muscle movement in the limbs and face, such as motion analysis for rehabilitation purpose [26].

A motor unit comprises a motor neuron and the fibers to which it connects (see Figure 2.4 below). The summation of the action potentials in the muscle fibres of a motor unit is termed the motor unit action potential (MUAP) [23].

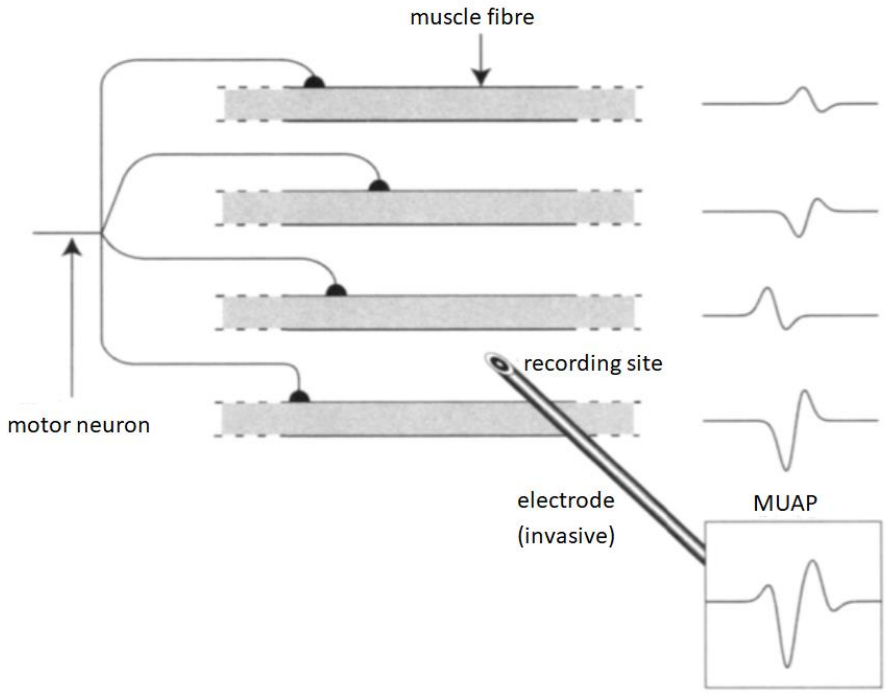

Figure 2.4. A motor unit action potential (MUAP). Reproduced from [23].

The surface EMG is the gross activity due to a large number of MUAPs near to the recording electrode. Although it is the gross activity and not the individual MUAP, the surface EMG does contain the necessary timing and amplitude information required for the study of motion. Most of the spectral power of the surface EMG is below 400-500 Hz, so a sampling rate of I kHz or higher is often used to sample the surface EMG.

There are two forms of muscular contraction. They are (1) spatial muscular contraction (more motor units are used), and (2) temporal muscular contraction (higher firing rate of the action potentials). The spatial form dominates at lower muscular intensity, while the temporal form dominates at higher muscular intensity.

Artefacts in EMG include (1) motion artefacts (usually less than 20 Hz), (2) powerline interference, and (3) superposition of the electrocardiogram on the EMG signal.

From the description above, it is clear that the EMG, like the EEG, has no single mathematical model to describe it. It is, however, suitable for time series classification, as it contains discriminatory motion information detectable by algorithms.

2.1.5 Inertial Sensor Signals

Besides the surface EMG, another common mean to read body motion signals is the use of inertial sensors such as accelerometer and gyroscope. Although they are noisy and not so accurate, they have the advantage of being less intrusive than the surface EMG. Together, they provide a rich set of body motion data that are sufficient for artificial motion perception for use in human activity recognition (HAR).

The smartphone is often used to collect the motion signals for HAR because it is ubiquitous and is packed with sensors - video camera, microphone, GPS, and inertial sensors (accelerometer and gyroscope). It has built-in communication capability to send data to other devices. A low-pass filter can separate the acceleration data into two parts, (1) the low-frequency (almost constant) acceleration due to the gravity, and (2) the high-frequency acceleration of the body due to the movement of the body.

HAR is a human-centred field of study with applications in ambient intelligence and assistive technology. Potential benefits include eco-friendly facilities and the provision of elderly care with less supervision.

In HAR, sensors are used to collect data from the ambience or the body, followed by machine learning to identify the physical activity. Theses activities can be classified as (1) basic activities, (2) transitional activities, and (3) complex activities. Basic activities are simple actions such as standing or walking. Transitional activities are the postural change, such as from sitting to walking. Complex activities are a sequence of activities that are executed for a common purpose, such as playing a game.

2.1.6 Heart Sound

The heart sound is another biosignal that can be measured on the body surface. During the cardiac cycle, the heart generates electrical impulses that cause the atria and the ventricles to contract. This forces the blood round the body. The opening and closure of the heart valves causes the entire cardiac structure to vibrate. These vibrations are audible at the chest wall.

Auscultation (listening for arrhythmia and murmurs with a stethoscope) can give an indication of the health of the heart. For data analysis, the heart sound is recorded as a phonocardiogram (PCG). Figure 2.5 below illustrates a short section of a PCG recording.

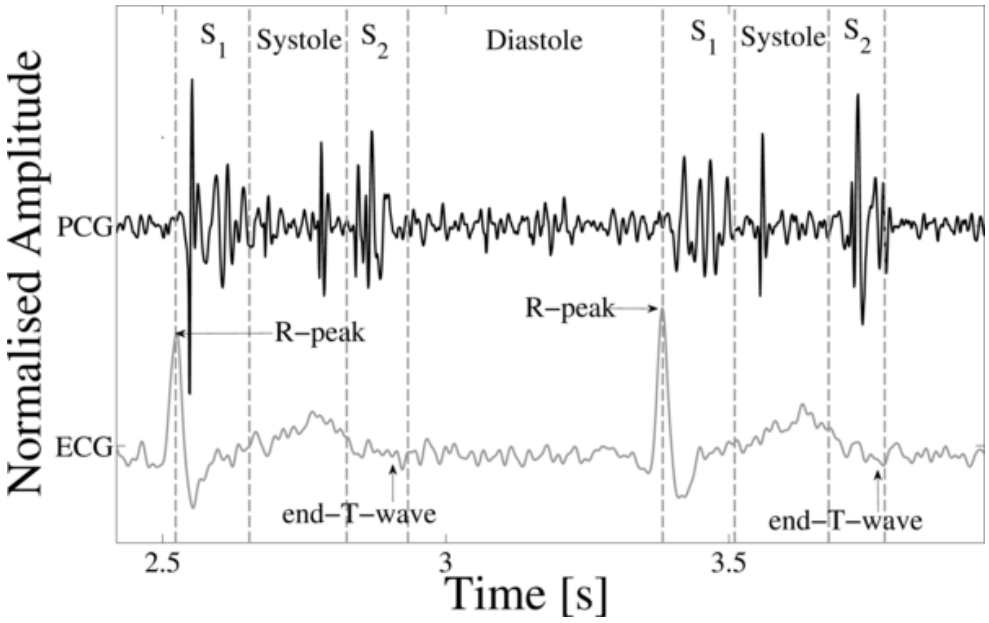

Figure 2.5. (Top) Heart sound and its four states: S1, systole, S2 and diastole. (Bottom) Electrocardiogram (ECG). Reproduced from [23].

Fundamental heart sounds include the first (S1) and second (S2) heart sounds. S1 occurs at the beginning of isovolumetric ventricular contraction, when the mitral and tricuspid valves close due to the rapid increase in pressure within the ventricles. S2 occurs at the beginning of diastole with the closure of the aortic and pulmonic valves.

While the fundamental heart sounds (S1 and S2) are the most recognizable sounds of the heart cycle, the mechanical activity of the heart may also cause other sounds. These include the third heart sound (S3), the fourth heart sound (S4), the systolic ejection click (EC), the mid-systolic click (MC), the diastolic sound or opening snap (OS), as well as the heart murmurs caused by the turbulent, high-velocity flow of blood [23].

Patients with abnormal heart sounds suffer from a variety of illnesses, such as heart valve defects and coronary artery disease. Heart valve defects include mitral valve prolapse, mitral regurgitation, and aortic stenosis.

It is possible to differentiate normal heart sound from abnormal heart sound based on the heart sound signal. Subjects with abnormal heart sounds will go to the specialist for further examination.

2.2 Deep Learning of Signals

With the proliferation of sensors, time series data are now widely available. They are encountered in many real-world applications, such as the identification of epileptic condition [15], human activity recognition [16], diagnostic of heart diseases [20], and many others. Due to the non-stationary, non-linear, and noisy nature of real-world time series data, it is daunting for the human cognitive process to classify them. This is, however, not a problem for machine learning, and in particular deep learning.

Before describing the workhorses of deep learning (deep belief net (DBN), convolutional neural network (CNN) and long short-term memory (LSTM) recurrent network), some descriptions of the linear model and its derivatives (logistic regression, extreme learning machine, support vector machine, hidden Markov model etc.) will be provided. This will allow deep learning to be understood in the context of machine learning, where deep learning comes from.

2.2.1 Linear Model

A linear model is the basis of many classifiers, such as the neural network and the support vector machine. It is a function that depends linearly on the adjustable parameters, i.e. the weights. For a data instance $\boldsymbol{x}$ with d input variables, i.e. $\boldsymbol{x} = [1, x_1, \dots, x_d]^T$ and weights $\boldsymbol{w} = [w_0, w_1, \dots, w_d]^T$, the output $y(\boldsymbol{x})$ is a scalar value and is the convolution sum of the inputs $\boldsymbol{x}$ and the weights $\boldsymbol{w}$.

$$y(\boldsymbol{x}) = w_0 1 + w_1 x_1 + \cdots + w_d x_d = \sum_{i=0}^{d} w_i x_i = \boldsymbol{x}^T \boldsymbol{w} \tag{2.2}$$

Several linear functions can form a linear network with multiple outputs. In Figure 2.6 below, the linear network has c linear functions forming c output variables $(y_1(\boldsymbol{x}), \dots, y_c(\boldsymbol{x}))$.

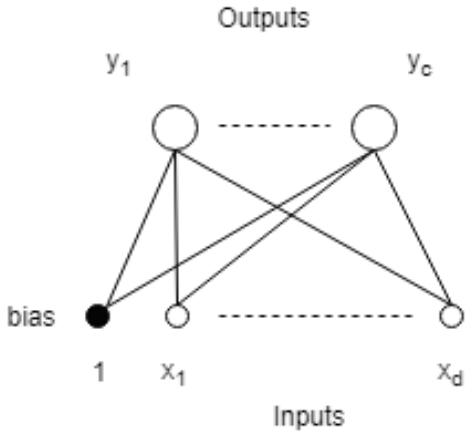

Figure 2.6. A linear model represented as a network

A $d \times c$ matrix, as shown in Equation (2.3) below, stores the weights of the linear network compactly. Each column in $\boldsymbol{W}$ is a vector of weights associated with a particular output variable.

$$\boldsymbol{W} = \begin{bmatrix} w_{1,1} & \cdots & w_{1,c} \\ \vdots & \ddots & \vdots \\ w_{d,1} & \cdots & w_{d,c} \end{bmatrix} \tag{2.3}$$

The $d \times c$ matrix is sometimes used in its $c \times d$ transposed form, due to convenience. This thesis will use the $d \times c$ matrix form for consistency sake. Figure 2.7 below shows the shape of the $d \times c$ matrix in relation to the input and the output variables of the linear network.

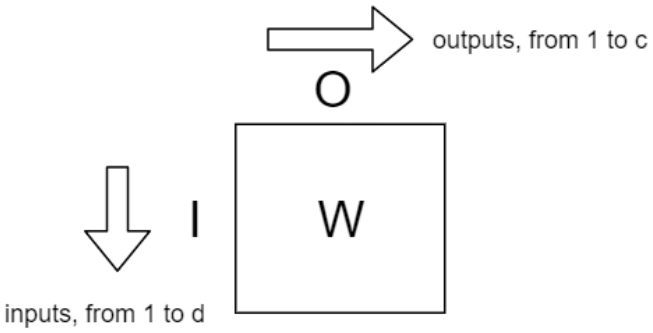

Figure 2.7. Matrix shape of the weights $\boldsymbol{W}$ of a linear network

Suppose there is a data matrix $\boldsymbol{X}$ with N data instances $(\boldsymbol{x}_i{}^T, \boldsymbol{y}_i{}^T), i \in \{1, \ldots, N\}$, where each data instance $\boldsymbol{x}_i{}^T$ is a d-dimensional row vector in the i-th row of $\boldsymbol{X}$, and each output vector

$\boldsymbol{y}_i{}^T$ is a c-dimensional row vector in the i-th row of $\boldsymbol{Y}$. The linear network will then take the linear algebra form as shown in Equation (2.4) below.

$$\boldsymbol{Y} = \boldsymbol{X}\boldsymbol{W} \tag{2.4}$$

There is no exact solution to the system shown in Equation (2.4) above when the data matrix $\boldsymbol{X}$ has more data instances than input variables, i.e. $N > d$. The system is overdetermined, as there are more equations than there are unknowns in the system.

An approximate solution for the weights $\boldsymbol{W}$ is possible when the system is overdetermined. This is often done in linear regression by framing the problem as an ordinary least square problem. In the ordinary least square problem, the cost function is the mean squared error $J(\boldsymbol{W}) = \|\boldsymbol{Y} - \boldsymbol{X}\boldsymbol{W}\|^2$ with respect to the weights $\boldsymbol{W}$. The estimated weights $\widehat{\boldsymbol{W}}$ is the pseudoinverse $\boldsymbol{X}^\dagger$ post-multiplied by $\boldsymbol{Y}$, as shown in Equation (2.5) below.

$$\widehat{\boldsymbol{W}} = arg\min_{\boldsymbol{W}} \|\boldsymbol{Y} - \boldsymbol{X}\boldsymbol{W}\|^2 = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{Y} = \boldsymbol{X}^\dagger\boldsymbol{Y} \tag{2.5}$$

There is a geometrical interpretation to the linear algebra $\boldsymbol{Y} = \boldsymbol{X}\boldsymbol{W}$. The column space of $\boldsymbol{X}$, i.e. $C(\boldsymbol{X})$, is spanned by d column vectors, and so $C(\boldsymbol{X})$ is a d-dimensional hyperplane passing through the origin. When the output $\boldsymbol{Y}$ is orthogonally projected to this hyperplane, it results in the projection $\widehat{\boldsymbol{Y}} = \boldsymbol{X}\widehat{\boldsymbol{W}}$. The projection $\widehat{\boldsymbol{Y}}$ is the linear combination of $\boldsymbol{X}$ by $\widehat{\boldsymbol{W}}$ and is the approximation of $\boldsymbol{Y}$ on the hyperplane with the error $\boldsymbol{Y} - \widehat{\boldsymbol{Y}}$. This is shown in Figure 2.8 below.

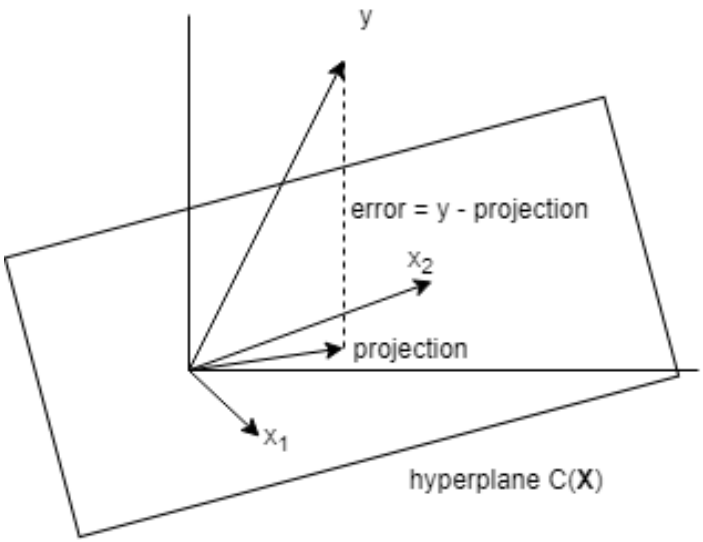

Figure 2.8. Geometric interpretation of the linear model

2.2.2 Logistic Regression

The linear model as described in the previous sub-section is useful for linear regression but not for classification. This is because the outputs of the linear model are numeric with unlimited range $(-\infty, +\infty)$. There is no meaningful threshold value to separate the output values into classes. For the linear model to be useful for classification, it will have to be generalised, either by transforming the outputs with a non-linear activation function to keep the numeric values to the range $[0,1]$, or by transforming the inputs with a non-linear kernel function so that the output can be separated by a linear hyperplane.

Logistic regression, despite its name, is a method not for regression but binary classification. It is a generalised linear model derived from the linear model used for linear regression. It transforms the outputs of the linear model with a non-linear activation function. The non-linear activation function $g(\cdot)$ that acts on the linear output $\boldsymbol{x}^T\boldsymbol{w}$ is the sigmoid function $\sigma(\cdot)$. The sigmoid function $\sigma(\cdot)$ is an element-by-element function that takes any real-valued vector and maps the vector elements to a value between 0 and 1. The values of the sigmoid function, or the activation a, can thus be interpreted as the probability $p(\boldsymbol{x})$. Equation (2.6) below shows the sigmoid function acting on the linear output $\boldsymbol{x}^T\boldsymbol{w}$.

$$a = \sigma(\boldsymbol{x}^T\boldsymbol{w}) = \frac{1}{1+\exp(-\boldsymbol{x}^T\boldsymbol{w})} = p(\boldsymbol{x}) \tag{2.6}$$

The log-transform of the odds, where the odds is defined as $\frac{p(\boldsymbol{x})}{1-p(\boldsymbol{x})}$, is known as "log-odds" or "logit". By re-arranging Equation (2.6), it can be shown that the log-transform links the probability $p(\boldsymbol{x})$ to the linear output $\boldsymbol{x}^T\boldsymbol{w}$ in the following form:

$$ln\left(\frac{p(\boldsymbol{x})}{1-p(\boldsymbol{x})}\right) = \boldsymbol{x}^T\boldsymbol{w} \tag{2.7}$$

For a data set with *N* data instances $\left(\boldsymbol{x}^{(i)}, y^{(i)}\right)$, the cost function for logistic regression is the negative of the average amount of correlation between the actual outcome $y^{(i)}$ and the log of the sigmoid output, for both possibilities of $y^{(i)}$ (i.e. 1 and 0), as shown in Equation (2.8) below. The cost function uses the negative of the log of the sigmoid output because it increases the cost to a very high value when the actual outcome $y^{(i)}$ and the predicted outcome $\sigma\left(\boldsymbol{x}^{(i)^T}\boldsymbol{w}\right)$ are not the same.

$$J(\boldsymbol{W}) = -\frac{1}{m}\sum_{i=1}^{N}\left(y^{(i)}log\left(\sigma\left(\boldsymbol{x}^{(i)^T}\boldsymbol{w}\right)\right) + \left(1-y^{(i)}\right)log\left(1-\sigma\left(\boldsymbol{x}^{(i)^T}\boldsymbol{w}\right)\right)\right) \tag{2.8}$$

The cost function $J(\boldsymbol{W})$ in Equation (2.8) above can be minimised with the gradient descend method. The general form of the gradient descend method is $\boldsymbol{W}_{k+1} = \boldsymbol{W}_k - \alpha\frac{\partial}{\partial \boldsymbol{W}}J(\boldsymbol{W})$, where α is the step size and $\frac{\partial}{\partial \boldsymbol{W}}J(\boldsymbol{W})$ is the differentiation of the cost function $J(\boldsymbol{W})$ with respect to the weights $\boldsymbol{W}$. Based on the gradient descend method, the update equation for the weights in logistic regression is as follows:

$$\boldsymbol{W}_{k+1} = \boldsymbol{W}_k - \frac{\alpha}{N}\sum_{i=1}^{N}\left(\sigma\left(\boldsymbol{x}^{(i)^T}\boldsymbol{w}\right) - \boldsymbol{y}^{(i)}\right)\boldsymbol{x}^{(i)} \tag{2.9}$$

The logistic regression model as described above is a static model, i.e. a model with no memory. The elements in the data instance $\boldsymbol{x}$ are input variables not related to each other in time. The

logistic regression model, with some changes in the data instance $\boldsymbol{x}$, will become a dynamic model. With a tapped delay line at the input, the data instance $\boldsymbol{x}$ will contain the temporal context of the input. It will then be possible to use the logistic regression on time series classification. Figure 2.9 below shows the logistic regression method applied to both the static network and the dynamic network.

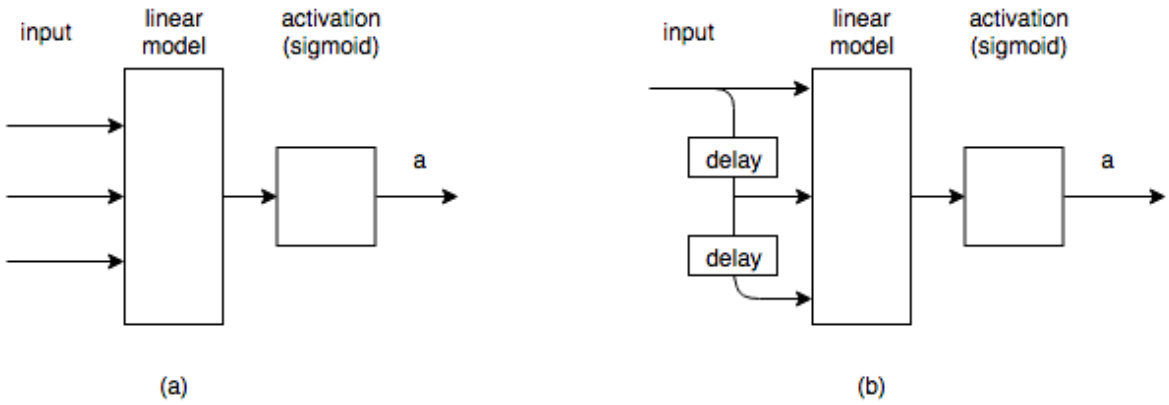

Figure 2.9: Logistic regression used as (a) static network, (b) dynamic network

2.2.3 Extreme Learning Machine

It is interesting that the linear network, despite the well-known argument against it (which says that a data set based on the XOR function is linearly inseparable), can in fact be used effectively for classification in what is called the extreme learning machine (ELM) [27].

This is because the *exact* representation of a given data set is irrelevant when the objective is to generalize the model to new and unseen data. An approximation, rather than an exact representation, of the input-output relationship is more robust on new and unseen data. The approximation is facilitated further by the fact that most practical problems have data that are smooth (small changes in the inputs lead to small changes in the outputs). Since the output of deep learning has the smooth property, it can use the ELM as the final classifier. The advantage of using ELM is its simplicity and its surprisingly good performance despite its simplicity.

The architecture of an ELM, shown in Figure 2.10 below, consists of an input layer, a non-linear hidden layer, and a linear output layer. Each of the output neurons represents an output class. The target output value is “1” if the input instance belongs to that class, and “-1” if the

input instance does not belong to that class. During inference, the output neuron with the largest value will become the predicted class. This is the one vs. rest method.

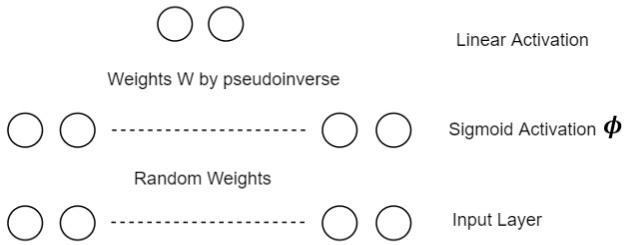

Figure 2.10. The two-layer arrangement of an extreme learning machine

The adjustable parameters in the ELM consist of two sets of weights, namely (1) the input weights (the weights of the non-linear hidden layer), and (2) the output weights (the weights of the linear output layer).

The input weights are set to random values. The output weights are obtained from the pseudo-inverse $\boldsymbol{X}^{\dagger}$ as follows: $\widehat{\boldsymbol{W}} = \boldsymbol{X}^{\dagger}\boldsymbol{Y}$ (refer to Equation (2.5)), where $\boldsymbol{X}$ refers to the activation of the non-linear hidden layer and $\boldsymbol{X}^{\dagger}$ is its pseudo-inverse.

The weights $\widehat{\boldsymbol{W}}$ obtained by the pseudoinverse $\boldsymbol{X}^{\dagger}$ will only be meaningful in the least square sense if the data are linearly separable. The well-known problem of the data not being linearly separable is resolved by using a non-linear hidden layer before the linear output layer. This gives the non-linearly separable data a chance to become linearly separable.

The weights of the non-linear hidden layer are random in values. As long as there are enough hidden units in the non-linear hidden layer, it will preserve much of the information at the input data for classification at the output layer.

Once the output weights $\widehat{\boldsymbol{W}}$ are obtained, they can be used to predict the target value of the test data $\boldsymbol{x}_{test}$.

$$\hat{\boldsymbol{y}}_{test}^{\boldsymbol{T}} = \boldsymbol{x}_{test}^{T}\widehat{\boldsymbol{W}} \tag{2.10}$$

As the above description shows, the weights of the ELM can be obtained in just one run, unlike logistic regression where multiple epochs are needed. The fast training time makes it suitable for incremental learning. When a trained model needs to work on a new data set, training at the final ELM classifier will be enough to adapt the trained but old model to the new data set.

2.2.4 Support Vector Machine

Another way to increase the computational power of a linear machine is to use a non-linear kernel function to map the input variables to a higher dimensional space, and then use a linear hyperplane to separate the output variables into classes.

A kernel is a similarity function $K(\cdot)$ with a scalar output value, such that for all pairs of data instances $(\boldsymbol{x}, \boldsymbol{z}) \in \boldsymbol{X}$,

$$K(\boldsymbol{x}, \boldsymbol{z}) = \langle \phi(\boldsymbol{x}) \cdot \phi(\boldsymbol{z}) \rangle \tag{2.11}$$

where $\phi(\cdot)$ is the mapping from $\boldsymbol{X}$ to a higher dimensional space, sometimes called the feature space or Hilbert space, and $\langle \cdot \rangle$ is the dot product [28]. There is no need to specify $\phi(\cdot)$ in a valid kernel. This is known as the kernel trick. A popular kernel is the Gaussian kernel, also known as the radial basis function (RBF), where $\frac{1}{\sigma^2}$ (called "gamma") controls the size of the region around $\boldsymbol{x}_i$. This is shown in Equation (2.12).

$$K(\boldsymbol{x}, \boldsymbol{x}_i) = e^{-\|\boldsymbol{x}-\boldsymbol{x}_i\|^2/\sigma^2} \tag{2.12}$$

The transformation of data instances to a higher dimensional space is called the kernel trick. With the kernel trick, the SVM is able to classify many non-linear data very well. Besides the kernel shown in Equation (2.11) above, a popular kernel used by SVM is the Gaussian kernel $K(\boldsymbol{x}, \boldsymbol{x}_i) = e^{-\|\boldsymbol{x}-\boldsymbol{x}_i\|^2/\sigma^2}$, also known as the radial basis function (RBF), where $\frac{1}{\sigma^2}$ (called "gamma") controls the size of the region around $\boldsymbol{x}_i$.

The support vector machine (SVM) is a kernel-based binary classifier that combines the kernels of a number of support vectors linearly, as shown in Equation (2.13) below. For any data instance $\boldsymbol{x}$, the output $f(\boldsymbol{x})$ of the SVM model is either 1 or -1

$$f(\boldsymbol{x}) = \begin{cases} 1 \; if \; sign\left(w_0 + \sum_{i \in \mathcal{S}} \alpha_i K(\boldsymbol{x}, \boldsymbol{x}_i)\right) = 1 \\ -1 \qquad\qquad\qquad otherwise \end{cases} \tag{2.13}$$

In Equation (2.13) above, $\mathcal{S}$ is a set of p support vectors $\boldsymbol{x}_i, i \in \{1, \dots, p\}$, and the alpha vector $\boldsymbol{\alpha}$ is the set of weights $\{\alpha_0, \dots, \alpha_i, \dots, \alpha_p\}$ that in essence defines how much influence the i-th support vector has on the final decision.

The support vectors $\boldsymbol{x}_i, i \in \{1, \dots, p\}$, are determined by maximising the margin between the classes. This means reducing the norm of $\boldsymbol{\alpha}$, which is the perpendicular of the hyperplane. The hyperplane separates the data instances into two sides. Those data instances within the margins of the hyperplane are the support vectors. By increasing the amount of slack allowed in the margin, more support vectors p will result.

The alpha vector $\boldsymbol{\alpha}$ is determined by the gradient descend method. The objective function of the gradient descend method consists of a loss function, called the hinge loss, and the L_2 norm regularizer. Equation (2.14) below shows the loss function for a training set with N data instances.

$$J(\boldsymbol{\alpha}) = \left\{\sum_{n=1}^{N} \left(1 - y_n \times f(\boldsymbol{x}_n)\right)\right\} + \left\{\min_{w} \lambda \|\boldsymbol{\alpha}\|^2\right\} \tag{2.14}$$

In Equation (2.14) above, if $f(\boldsymbol{x}_n)$ is on the correct side, then the term $y_n \times f(\boldsymbol{x}_n)$ will be 1, otherwise it will be -1. It is costly for $f(\boldsymbol{x}_n)$ to be at the wrong side of the hyperplane. Conversely, the cost is zero if it is at the correct side.

Weight update, based on the gradient descend method, is the sum of the partial derivative of the regularizer and the loss function with respect to $\boldsymbol{\alpha}$, as shown in Equation (2.15) and (2.16) below.

$$\frac{\partial}{\partial \boldsymbol{\alpha}} \lambda \|\boldsymbol{\alpha}\|^2 = 2\lambda \boldsymbol{\alpha} \tag{2.15}$$

$$\frac{\partial}{\partial \boldsymbol{\alpha}} \left(1 - y_n \times f(\boldsymbol{x}_n)\right) = \begin{cases} 0 \; if \; correctly \; classified \\ -y_n K(\boldsymbol{x}_n, \boldsymbol{x}_i) \; for \; i \in \{1, \dots, p\} \quad otherwise \end{cases} \tag{2.16}$$

If a sample is misclassified, the alpha vector $\boldsymbol{\alpha}$ is updated using the gradients of both terms. Otherwise, $\boldsymbol{\alpha}$ is updated by using just the gradient of the regularizer.

Once the support vectors and the alpha weights are determined, they will be stored in memory, ready for the prediction of new and unseen data.

All the kernels used by SVM are local kernels. This means that $K(\boldsymbol{x}, \boldsymbol{x}_i) > \rho$ if $\boldsymbol{x}$ is in a connected region around the support vector $\boldsymbol{x}_i$. A local kernel provides the discrimination criteria when the target function is smooth. For a highly varying function, the target function may not be smooth and may scatter over a large section of the input space. Hence, for complex data, the trained SVM model may not be able to generalize to new variations outside of the limited section of the input space not covered by the training set [29].

2.2.5 Linear Gaussian Model

The linear Gaussian model extends the linear model by adding noise to the linear model. It associates the signal data, called the *observation*, to the *state* of the process. At any point in time, the observation $\boldsymbol{x}$ (a column vector with *d* input variables) is visible, while the hidden state $\boldsymbol{y}$ (a column vector with *c* state variables) is latent. Both $\boldsymbol{x}$ and $\boldsymbol{y}$ are stochastic.

From the generative point of view, it is the hidden state $\boldsymbol{y}$ that emits the observation $\boldsymbol{x}$. From the point of view of machine learning, it is the output $\boldsymbol{y}$ that ought to be predicted, given the input $\boldsymbol{x}$.

In the linear Gaussian model, the hidden state $\boldsymbol{y}$ is a process that evolves according to the first order Markov chain. This means that the current state $\boldsymbol{y}_t$ depends only on the previous state $\boldsymbol{y}_{t-1}$ and not earlier.

Equations (2.17) and (2.18) below form the linear Gaussian model. The $c \times c$ transition matrix $\boldsymbol{A}$ and the $d \times c$ measurement matrix $\boldsymbol{C}$, together with the zero-mean process noise $\boldsymbol{w}$ and the observation noise $\boldsymbol{v}$, are the model parameters.

$$\boldsymbol{y}_t = \boldsymbol{A}\boldsymbol{y}_{t-1} + \boldsymbol{w} \tag{2.17}$$

$$\boldsymbol{x}_t = \boldsymbol{C}\boldsymbol{y}_t + \boldsymbol{v} \tag{2.18}$$

Figure 2.11 below shows two common linear Gaussian models used for time series, namely the Kalman filter and the hidden Markov model.

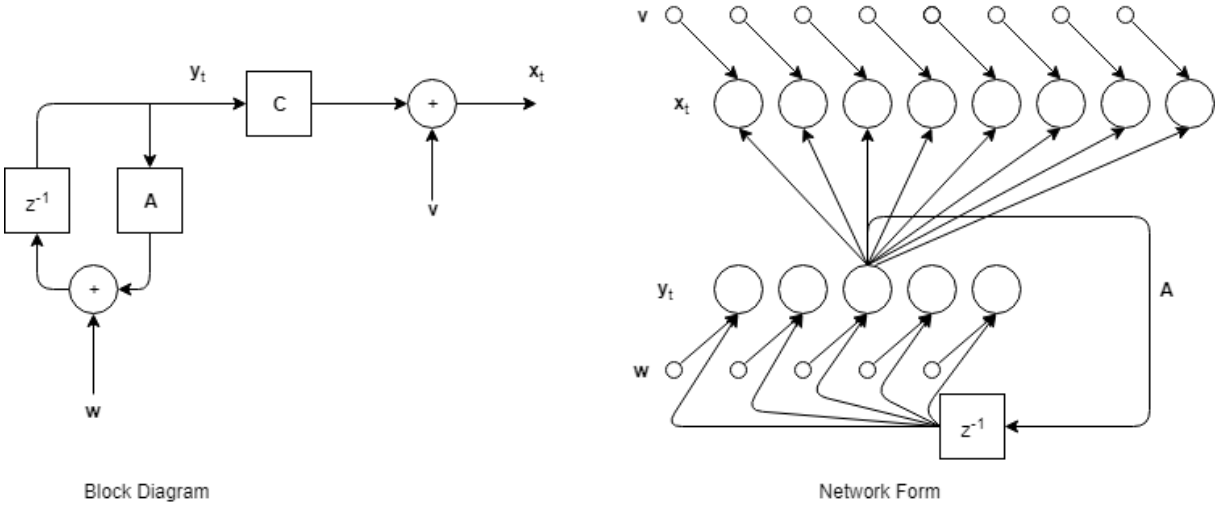

Figure 2.11. Linear Gaussian model: Kalman filter (left) and hidden Markov model (right)

The Kalman filter is a linear Gaussian model with continuous state. The vector elements in $\boldsymbol{y}_t$ are numeric in data type. Like linear regression that predicts a continuous output for every input, Kalman filter predicts the latent state values from the observation $\boldsymbol{x}_t$ continuously, so it is suitable for use in tracking,.

For classification, the hidden Markov model (HMM) should be used instead. It adopts the winner-take-all strategy for the state $\boldsymbol{y}_t$, where the most probable vector element in $\boldsymbol{y}_t$ will be set to 1 and the rest set to 0.

The HMM is widely used in automatic speech recognition [30], since it can handle inputs of variable-length (for example, different versions of "hello") and still predict the state accordingly. The biggest limitation on the use of the HMM model is the assumption of conditional independence (i.e. the observation depends on the current state only). This is not really true for complex signals.

The HMM is often used with the probabilistic Gaussian mixture model (GMM) instead of the deterministic measurement matrix $\boldsymbol{C}$. It is then known as HMM-GMM [9]. In HMM-GMM, it is assumed that (1) the state $\boldsymbol{y}_t$ is a process that changes according to the Markov chain defined by the initial state distribution $\boldsymbol{\pi}$ and the transition matrix $\boldsymbol{A}$, with $[\boldsymbol{A}]_{t-1,t} = p(\boldsymbol{y}_t|\boldsymbol{y}_{t-1})$, and (2) the emissive probability $b_k(\boldsymbol{x}_t)$ of observing an input vector $\boldsymbol{x}_t$ at a particular state k is a mixture of M multivariate Gaussian distribution, denoted as

$$b_k(\boldsymbol{x}_t) = \sum_{l=1}^{M} [\boldsymbol{a}]_{l,k} \mathcal{N}\left(\boldsymbol{x}_t|\boldsymbol{\theta}_{l,k}\right) \tag{2.19}$$

where $[\boldsymbol{a}]_{l,k}$ is the value of the membership distribution of the l-th component in state k, and $\boldsymbol{\theta}_{l,k}$ is the parameters (mean and covariance) of the l-th component in state k.

In the Baum-Welch algorithm, which is a form of EM algorithm, the GMM parameters $\boldsymbol{\theta}$ and $\boldsymbol{a}$ are used to compute the emissive probability $b_k(\boldsymbol{x}_t)$ and the other model parameters (the transition matrix $\boldsymbol{A}$ and the state distribution $\boldsymbol{\pi}$), which are then in turn used to update the GMM parameters $\boldsymbol{\theta}$ and $\boldsymbol{a}$.

The Baum-Welch algorithm has two parts, the forward and the backward algorithm. The forward algorithm determines the probability of observing the time sequence $(\boldsymbol{x}_1, \dots, \boldsymbol{x}_t)$ and the hidden state $\boldsymbol{y}_t$ at t. Equation (2.20) below shows the forward probability.

$$\boldsymbol{\alpha}_t = p(\boldsymbol{y}_t, \boldsymbol{x}_1, \dots, \boldsymbol{x}_t) \tag{2.20}$$

The backward algorithm determines the probability of observing the time sequence $(\boldsymbol{x}_{t+1}, \dots, \boldsymbol{x}_T)$, i.e. the time sequence from $t+1$ to the end of the signal, given the hidden state $\boldsymbol{y}_t$ at t. Equation (2.21) shows the backward probability.

$$\boldsymbol{\beta}_t = p(\boldsymbol{x}_{t+1}, \dots, \boldsymbol{x}_T | \boldsymbol{y}_t) \tag{2.21}$$

In the above definitions, $\boldsymbol{\alpha}_t$ and $\boldsymbol{\beta}_t$ are vectors of the same length as the hidden state $\boldsymbol{y}_t$. Notation wise, the k-th element in $\boldsymbol{\alpha}_t$ is $[\boldsymbol{\alpha}_t]_k = p(\boldsymbol{y}_t = k, \boldsymbol{x}_1, \dots, \boldsymbol{x}_t)$. Likewise, the k-th element in $\boldsymbol{\beta}_t$ is $[\boldsymbol{\beta}_t]_k = p(\boldsymbol{x}_{t+1}, \dots, \boldsymbol{x}_T | \boldsymbol{y}_t = k)$.

In the forward algorithm, at $t = 1$, the probability of the time sequence up to $\boldsymbol{x}_1$ and at state k is the product of two probability values: (1) $b_k(\boldsymbol{x}_1)$, the emissive probability of $\boldsymbol{x}_1$ at state k, and (2) $[\boldsymbol{\pi}]_k$, the initial probability of being in state k.

$$[\boldsymbol{\alpha}_1]_k = [\boldsymbol{\pi}]_k b_k(\boldsymbol{x}_1), k \in \{1, \dots, c\} \tag{2.22}$$

Once the initial value $[\boldsymbol{\alpha}_1]_k$ is available, subsequent values of $[\boldsymbol{\alpha}_t]_k$ at $t = 2, \dots, T$ can be computed from the transition matrix $\boldsymbol{A}$, as shown in Equation (2.23) below.

$$[\boldsymbol{\alpha}_t]_k = \left[\sum_{i=1}^{c} [\boldsymbol{\alpha}_{t-1}]_i [\boldsymbol{A}]_{i,k} \right] b_k(\boldsymbol{x}_t) \tag{2.23}$$

In the backward algorithm, the probability of the time sequence $(\boldsymbol{x}_{t+1}, \dots, \boldsymbol{x}_T)$, given state $\boldsymbol{y}_t$, is computed at t (one time-step before $t+1$). When $t = T$, the backward probability will be fixed at 1.

$$[\boldsymbol{\beta}_T]_k = 1 \tag{2.24}$$

Once the final value $[\boldsymbol{\beta}_T]_k$ is fixed, subsequent values of $[\boldsymbol{\beta}_t]_k$ at $t = T-1, , \dots, 1$ can be computed from the transition matrix $\boldsymbol{A}$, as shown in Equation (2.25) below:

$$[\boldsymbol{\beta}_t]_k = \left[\sum_{i=1}^{c} [\boldsymbol{\beta}_{t+1}]_i [\boldsymbol{A}]_{k,i} b_i(\boldsymbol{x}_{t+1})\right] \tag{2.25}$$

Once the forward and backward probabilities $\boldsymbol{\alpha}_t$ and $\boldsymbol{\beta}_t$ are computed *progressively* for all the sampling points in the $\boldsymbol{x}$ sequence, two other variables, $\boldsymbol{\gamma}(t)$ and $\boldsymbol{\xi}(t)$, will have to be computed over the *entire* $\boldsymbol{x}$ sequence for the update of the adjustable parameters in HMM-GMM.

$\boldsymbol{\gamma}_t$ is the probability of the entire sequence $\boldsymbol{x} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T)$ being in state $k \in \{1, \ldots, c\}$ at time step t.

$$[\boldsymbol{\gamma}_t]_k = P(\boldsymbol{y}_t = k|\boldsymbol{x}) \tag{2.26}$$

It is a vector defined in terms of the forward probability $[\boldsymbol{\alpha}_t]_k$ and the backward probability $[\boldsymbol{\beta}_t]_k$, normalized across all the state values.

$$[\boldsymbol{\gamma}_t]_k = \frac{[\boldsymbol{\alpha}_t]_k [\boldsymbol{\beta}_t]_k}{\sum_{i=1}^{c} [\boldsymbol{\alpha}_t]_i [\boldsymbol{\beta}_t]_i} \tag{2.27}$$

$\boldsymbol{\xi}_t$ is the joint probability of the entire sequence $\boldsymbol{x} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T)$ being in state $k \in \{1, \ldots, c\}$ at time t *and* state $j \in \{1, \ldots, c\}$ at time $t + 1$.

$$[\boldsymbol{\xi}_t]_{k,j} = p(\boldsymbol{y}_t = k, \boldsymbol{y}_{t+1} = j|\boldsymbol{x}) \tag{2.28}$$

It is a matrix defined in terms of the forward probability $[\boldsymbol{\alpha}_t]_k$ and the backward probability $[\boldsymbol{\beta}_{t+1}]_j$ and then normalized to make the sum of the resultant matrix as 1, as shown in Equation (2.29) below.

$$[\boldsymbol{\xi}_t]_{k,j} = \frac{[\boldsymbol{\alpha}_t)]_k [\boldsymbol{\beta}_{t+1}]_j b_j(\boldsymbol{x}_{t+1})}{\cdot} \tag{2.29}$$

The sum of $[\boldsymbol{\gamma}_t]_k$ over the entire length of the signal, $\sum_{t=1}^{T} [\boldsymbol{\gamma}_t]_k$, is the expected number of times the signal is in state k. Similarly, the sum of $[\boldsymbol{\xi}_t]_{k,j}$ over the entire length of the signal, $\sum_{t=1}^{T-1} [\boldsymbol{\xi}_t]_{k,j}$, is the expected number of transitions from state k to state j. Therefore, the

proportion of transitioning from state k to state j when it is at state k is the transition matrix $[\boldsymbol{A}]_{k,j}$, shown in Equation (2.30) below.

$$[\boldsymbol{A}]_{k,j} = \frac{\sum_{t=1}^{T-1}[\boldsymbol{\xi}_t]_{k,j}}{\sum_{t=1}^{T-1}[\boldsymbol{\gamma}_t]_k} \tag{2.30}$$

A GMM has M components. The probability that $\boldsymbol{x}$, at t, is in the k-th state ($k \in \{1, \dots, c\}$) and belong to the l-th component ($l \in \{1, \dots, M\}$ is a matrix with $[\boldsymbol{\gamma}_t]_{l,k}$ as its elements. This matrix can be obtained from the multiplication of $[\boldsymbol{\gamma}_t]_k$ (see Equation (2.26) above) with the member distribution $[\boldsymbol{a}]_{l,k}$ (initialised as uniformly distributed), as shown in Equation (2.31) below.

$$[\boldsymbol{\gamma}_t]_{l,k} = [\boldsymbol{\gamma}_t]_k \frac{[\boldsymbol{a}]_{l,k} b_k(\boldsymbol{x}_t)}{\sum_{i=1}^{c} b_i(\boldsymbol{x}_t)} \tag{2.31}$$

The member distribution $\boldsymbol{a}$ and the GMM parameters $\boldsymbol{\theta}$ at state variable k are updated with the use of $[\boldsymbol{\gamma}_t]_{l,k}$ in Equation (2.31) above. The GMM parameters $\boldsymbol{\theta}$ consist of the mean $\boldsymbol{\mu}$ and the covariance $\boldsymbol{\Sigma}$ of the l-th component in the GMM. They are computed as shown in Equations (2.32), (2.33), and (2.34) below:

$$[\boldsymbol{a}]_{l,k} = \frac{\sum_{t=1}^{T}[\boldsymbol{\gamma}_t]_{l,k}}{\sum_{t=1}^{T}[\boldsymbol{\gamma}_t]_k} \tag{2.32}$$

$$[\boldsymbol{\mu}]_{l,k} = \frac{\sum_{t=1}^{T}([\boldsymbol{\gamma}_t]_{l.k}\boldsymbol{x}_t)}{\sum_{t=1}^{T}[\boldsymbol{\gamma}_t]_{l.k}} \tag{2.33}$$

$$[\boldsymbol{\Sigma}]_{l,k} = \frac{\sum_{t=1}^{T}\left([\boldsymbol{\gamma}_t]_{l.k}(\boldsymbol{x}_t - \boldsymbol{\mu}_{l,k})(\boldsymbol{x}_t - \boldsymbol{\mu}_{l,k})^T\right)}{\sum_{t=1}^{T}[\boldsymbol{\gamma}_t]_{l.k}} \tag{2.34}$$

The Baum-Welch process gives consistent results with well-segmented sequences. However, the accuracy drops with sequences that are not well-segmented.

2.2.6 Neural Network

Historically, neural network is the beginning of deep learning. In a neural network, the output of a layer becomes the input of the next layer. This is composition, which is the foundation of deep learning.

Just two layers (one hidden layer and one output layer) are all that is required for a neural network to be a universal approximator [31]. A neural network with one or two hidden layers is a shallow network.

The count of the number of layers in a neural network often omits the input layer. Figure 2.12 below shows a three-layer multilayer perceptron (MLP) network with one input layer l_0, two hidden layers, l_1 and l_2, and an output layer l_3.

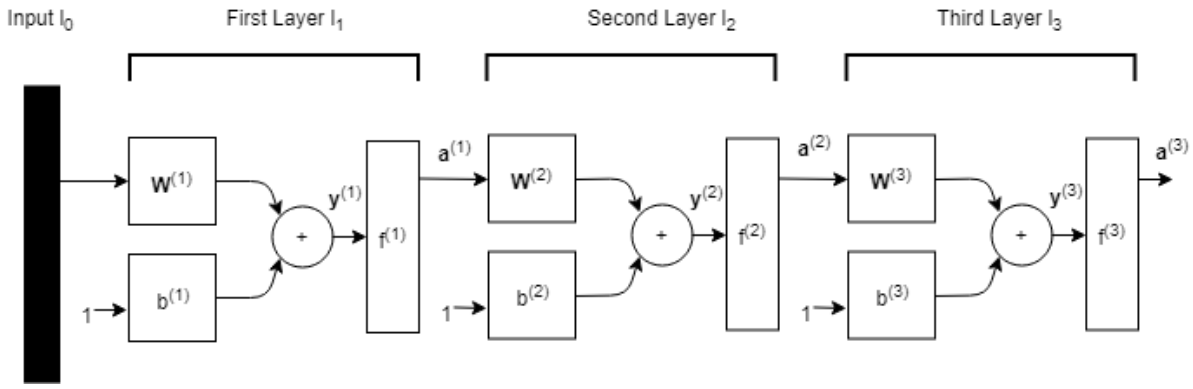

Figure 2.12. A 3-layer MLP (two hidden layers and an output layer)

In a neural network, learning of the adjustable parameters, i.e. the weights, is by gradient descend and backpropagation. According to Schmidhuber (2015), the minimisation of errors through gradient descent in the parameter space of nonlinear multi-stage systems has been discussed at least since the early 1960s. Rumelhart et al (1986) demonstrated that the outputs of the hidden layers represent useful abstractions, and this caused the neural network to be widely used. The general form of gradient descend is as follows:

$$\boldsymbol{W}_{k+1} = \boldsymbol{W}_k - \alpha \frac{\partial}{\partial \boldsymbol{W}} J(\boldsymbol{W}) \tag{2.35}$$

In Equation (2.35) above, α is the step size, and $J(\boldsymbol{W})$ is the cost function (often abbreviated as E). What Equation (*2.35*) above shows is that the weights have to be changed in the direction of the negative gradient of the cost function.

As the weights are distributed across multiple layers, the gradient at all the layers will have to be computed layer by layer. This is made possible by the chain rule of differentiation. For each layer l, two feedforward terms, computed during forward propagation, are stored in memory, namely the linear output $\boldsymbol{y}^{(l)}$ and the non-linear activation $\boldsymbol{a}^{(l)}$. Using these feedforward terms, the backward term, delta $\boldsymbol{\delta}^{(l)} \triangleq \frac{\partial E}{\partial \boldsymbol{y}^{(l)}}$, can be computed in backward manner from $\boldsymbol{a}^{(l)}$ to $\boldsymbol{y}^{(l)}$ to $\boldsymbol{a}^{(l-1)}$ to $\boldsymbol{y}^{(l-1)}$ and so on. The operation based on the chain rule of differentiation is summarised in Equations (2.36) and (2.37) as shown below:

$$\frac{\partial E}{\partial \boldsymbol{W}^{(l)}} = \frac{\partial E}{\partial \boldsymbol{y}^{(l)}} \frac{\partial \boldsymbol{y}^{(l)}}{\partial \boldsymbol{W}^{(l)}} = \boldsymbol{a}^{(l-1)} \left(\boldsymbol{\delta}^{(l)}\right)^T \tag{2.36}$$

$$\boldsymbol{\delta}^{(l-1)} \triangleq \frac{\partial E}{\partial \boldsymbol{y}^{(l-1)}} = \frac{\partial E}{\partial \boldsymbol{y}^{(l)}} \frac{\partial \boldsymbol{y}^{(l)}}{\partial \boldsymbol{a}^{(l-1)}} \frac{\partial \boldsymbol{a}^{(l-1)}}{\partial \boldsymbol{y}^{(l-1)}} = \boldsymbol{W}^{(l)} \boldsymbol{\delta}^{(l)} \odot f'\left(\boldsymbol{y}^{(l-1)}\right) \tag{2.37}$$

As a result of the relationship $\boldsymbol{y}^{(l)} = {\boldsymbol{W}^{(l)}}^T \boldsymbol{a}^{(l-1)}$ that links $\boldsymbol{y}^{(l)}$ to $\boldsymbol{a}^{(l-1)}$, the $\frac{\partial \boldsymbol{y}^{(l)}}{\partial \boldsymbol{W}^{(l)}}$ term in Equation (2.36) is $\boldsymbol{a}^{(l-1)}$, and the $\frac{\partial \boldsymbol{y}^{(l)}}{\partial \boldsymbol{a}^{(l-1)}}$ term in Equation (2.37) is $\boldsymbol{W}^{(l)}$.

The delta term $\boldsymbol{\delta}^{(l)}$ in Equation (2.36) above is backpropagated from layer l to layer $l-1$ in Equation (2.37). This can be seen by the presence of $\boldsymbol{\delta}^{(l)}$ on the right hand side of Equation (2.37) and $\boldsymbol{\delta}^{(l-1)}$ on the left hand side of the same equation.

The backpropagation process starts at the final output layer with the computation of the delta term there, i.e. $\boldsymbol{\delta}^{(L)}$. Its computation is shown in Equation (2.38) below.

$$\boldsymbol{\delta}^{(L)} \triangleq \frac{\partial E}{\partial \boldsymbol{y}^{(L)}} = \frac{\partial E}{\partial \boldsymbol{a}^{(L)}} \frac{\partial \boldsymbol{a}^{(L)}}{\partial \boldsymbol{y}^{(L)}} = \left(\boldsymbol{a}^{(L)} - \boldsymbol{t}\right) \odot \left[\boldsymbol{y}^{(L)}(\boldsymbol{1} - \boldsymbol{y}^{(L)})\right] \tag{2.38}$$

Here, $\boldsymbol{a}^{(L)}$ is the softmax output vector as shown in Equation (2.39) below. It is the vector $e^{-\boldsymbol{y}^{(L)}}$ normalized by the sum of the vector elements.

$$softmax = \frac{e^{-\boldsymbol{y}^{(L)}}}{\text{sum}\left(e^{-\boldsymbol{y}^{(L)}}\right)} \tag{2.39}$$

Since $J(\boldsymbol{W}) = \left(\boldsymbol{a}^{(L)} - \boldsymbol{t}\right)^2$, its derivative $\frac{\partial E}{\partial \boldsymbol{a}^{(L)}}$ is proportional to $\boldsymbol{a}^{(L)} - \boldsymbol{t}$. As for the term $\frac{\partial \boldsymbol{a}^{(L)}}{\partial \boldsymbol{y}^{(L)}}$ in Equation (2.38) above, it is simply the derivative of the sigmoid function, which has the well-known result of $\boldsymbol{y}^{(L)}(\boldsymbol{1} - \boldsymbol{y}^{(L)})$.

The neural network accepts tabular data as its input. Other types of data, for example, the lag observations of a time series, have to be rearranged into the vector form expected by the input layer of the neural network. The vector at the input layer is actually a tapped delay line. This forms what is called a time-delay neural network (TDNN). It was introduced by Waibel et al (1989) and has been used in many time series applications, such as human sound location [34] and the detection of Parkinson disease [35].

When the hidden layers (in addition to the input layer) have tapped delay lines also, the network becomes a distributed TDNN. This can be visualized in two ways, as shown by the diagrams in Figure 2.13 below.

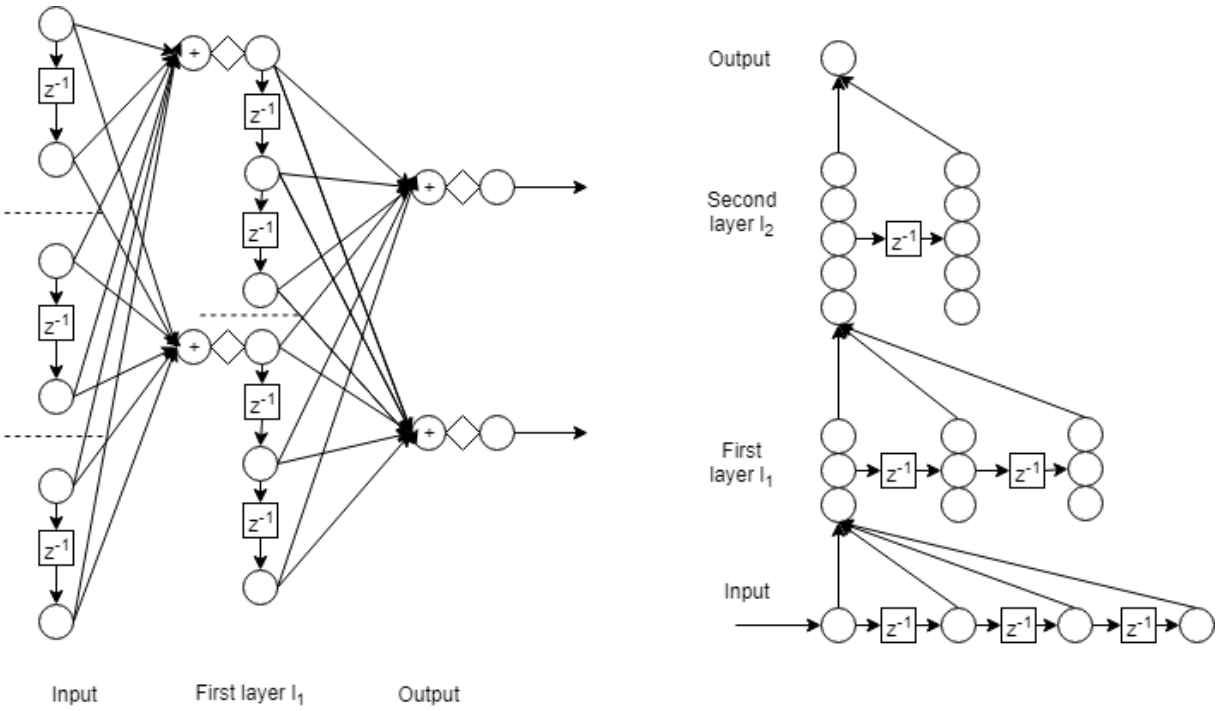

Figure 2.13. Visualization of a TDNN

The left hand side of Figure 2.13 shows a neural network with (1) an input layer with 3 units (each with a 2-tap delay line), (2) a hidden layer with 2 units (each with a 3-tap delay line), and (3) a final output layer with 2 units. The right hand side of Figure 2.13 shows a neural network with (1) an input layer with one unit (with a 4-tap delay line), (2) a hidden layer with 3 units (sharing a 3-tap delay line), (3) another hidden layer with 5 units (sharing a 2-tap delay line), and (4) an output layer with 1 unit. Over here, the term "unit" refers to a neuron with a tapped delay line attached to it.

The distributed TDNN can be unrolled into its equivalent static network. The distributed TDNN of Figure 2.13 above is shown on the right hand side of Figure 2.14 below. The left hand side of the same figure shows the equivalent static network.

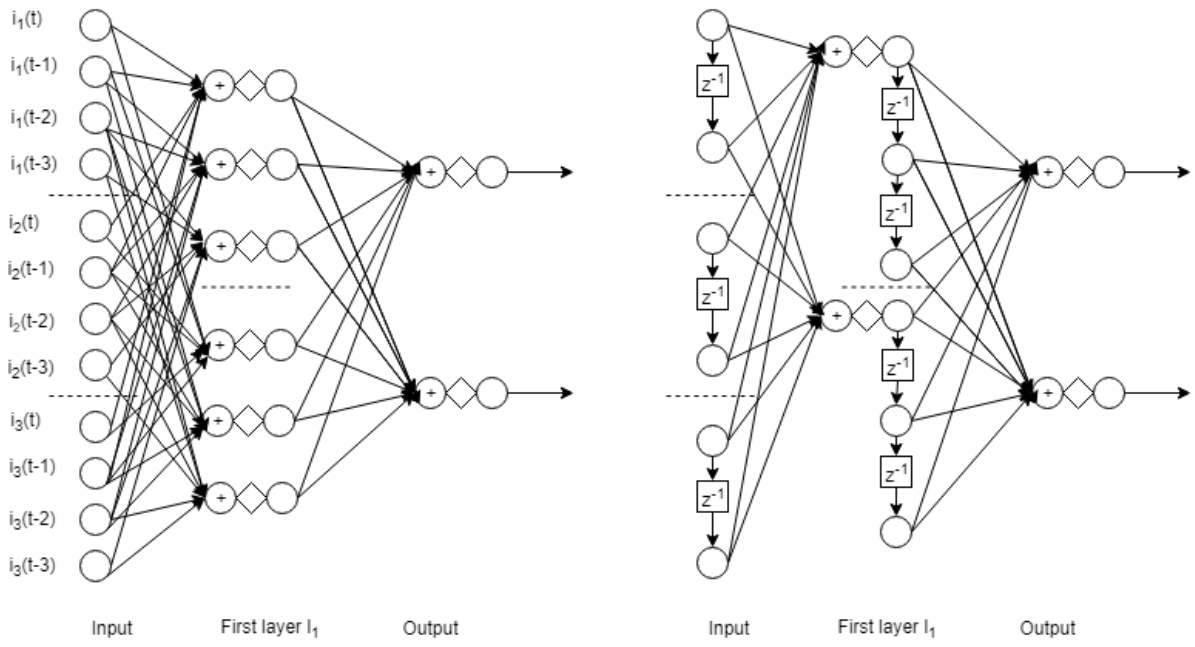

Figure 2.14. Equivalent static network and the actual distributed TDNN

Although the equivalent static network has no tapped delay line, it can be envisaged as having nodes arranged according to the tapped delay lines. Note the dotted lines in Figure 2.14 above. There are more nodes in the equivalent static network than the number of taps in the actual distributed TDNN. In other words, for each of the layers, the equivalent tap length is either the same or longer than the actual tap length. Notation wise, for a particular layer l, $T_{equi}^{(l)} \geq T_{tdnn}^{(l)}$. In Figure 2.14 above, for the output layer, $T_{equi}^{(L)} = 1$ and $T_{tdnn}^{(L)} = 1$, for the hidden layer, $T_{equi}^{(1)} = 3$ and $T_{tdnn}^{(1)}$=3, and for the input layer, $T_{equi}^{(0)} = 4$ and $T_{tdnn}^{(0)} = 2$.

The equivalent tap length at layer $l-1$, i.e. $T_{equi}^{(l-1)}$, is equal to that of layer l (i.e. $T_{equi}^{(l)}$) plus the actual tap length at layer $l-1$ minus 1. This is shown in Equation (2.40) below.

$$T_{equi}^{(l-1)} = T_{equi}^{(l)} + T_{tdnn}^{(l-1)} - 1 \tag{2.40}$$

For example, referring to Figure 2.14 where the final output layer has 1 unit ($T_{equi}^{(L)} = 1$). The equivalent tap length in the hidden layer is $T_{equi}^{(1)} = 1 + 3 - 1 = 3$. Similarly, the equivalent tap length in the input layer is $T_{equi}^{(0)} = 3 + 2 - 1 = 4$.

The receptive field of the equivalent static network is longer than the number of actual taps at the input layer. Not only is the equivalent tap length longer, the equivalent activation vector $\boldsymbol{a}_{equi}^{(l)}$ and the equivalent linear output vector $\boldsymbol{y}_{equi}^{(l)}$ are also longer. Notation wise, for a particular layer l, $\boldsymbol{a}_{equi}^{(l)} > \boldsymbol{a}_{tdnn}^{(l)}$ and $\boldsymbol{y}_{equi}^{(l)} > \boldsymbol{y}_{tdnn}^{(l)}$. They are decomposed into $T_{equi}^{(l)}$ sets of vectors as shown in Equations (2.41) and (2.42) below, where the subscripts $0, \dots, T_{equi}^{(l)} - 1$ act as the selectors of the elements in $\boldsymbol{a}_{equi}^{(l)}$ and $\boldsymbol{y}_{equi}^{(l)}$.

$$\boldsymbol{a}_{equi}^{(l)} = \left[\boldsymbol{a}_{equi}^{(l)}\right]_0 + \left[\boldsymbol{a}_{equi}^{(l)}\right]_1 + \cdots + \left[\boldsymbol{a}_{equi}^{(l)}\right]_{T_{equi}^{(l)}-1} \tag{2.41}$$

$$\boldsymbol{y}_{equi}^{(l)} = \left[\boldsymbol{y}_{equi}^{(l)}\right]_0 + \left[\boldsymbol{y}_{equi}^{(l)}\right]_1 + \cdots + \left[\boldsymbol{y}_{equi}^{(l)}\right]_{T_{equi}^{(l)}-1} \tag{2.42}$$

At each layer l, the unique weights $\boldsymbol{W}^{(l)}$ are reused and shared by sliding them between the input side (i.e. layer $l-1$) and the output side (i.e. layer l). The number of slides is $T_{equi}^{(l)}$. As a result, there are $T_{equi}^{(l)}$ exact copies of $\boldsymbol{W}^{(l)}$ in the equivalent static network.

For example, on the right hand side of Figure 2.14 above, the hidden layer l_1 of the actual TDNN has 12 weights in $\boldsymbol{W}^{(1)}$ (3 input neurons $\times$ 2 hidden neurons, with 2 such sets corresponding to the 2-tap delay line at the input layer). These weights are shared 3 times in the equivalent static network on the left hand side of Figure 2.14 above. There are thus 36 weights in layer $l_1$of the equivalent static network. This is less than the 72 weights if the network is fully connected. Out of the 36 weights, only 12 of them are unique, and the rest are reused and shared.

In summary, there are 3 points to note about the equivalent static network:

(1) The input side is not fully interconnected to the output side.

(2) For the units that are connected, the weights $\boldsymbol{W}^{(l)}$ are reused and shared $T_{equi}^{(l)}$ times.

(3) The number of unique weights in the equivalent static network is the same as the actual distributed TDNN.

To update $\boldsymbol{W}^{(l)}$, weight sharing has to be taken into account during backpropagation. Now, consider the activation $\left[\boldsymbol{a}_{equi}^{(l-1)}\right]_t$ at an arbitrary node position $t \in \left\{0, \dots, T_{equi}^{(l-1)} - 1\right\}$ at the input side of the equivalent static network. It connects to the output node at position $m \in \left\{0, \dots, T_{equi}^{(l)} - 1\right\}$ in a fan-out that lies in the range of $\left(t - T_{tdnn}^{(l-1)} + 1\right) \leq m \leq t$. Due to the two exceptions at the beginning and end of layer $l-1$, this range can be re-expressed as Equation (2.43) below.

$$max\left(0, t - T_{equi}^{(l)} + 1\right) \leq m \leq min\left(t, T_{tdnn}^{(l-1)}\right) \tag{2.43}$$

The activation at layer l is made up of $T_{equi}^{(l)}$ sets of activations, as shown in Equation (2.44).

$$\boldsymbol{a}_{equi}^{(l)} = \sum_{m=0}^{T_{equi}^{(l)}-1} \left[\boldsymbol{a}_{equi}^{(l)}\right]_m \tag{2.44}$$

Similarly, the backward term delta $\boldsymbol{\delta}_{equi}^{(l)} \triangleq \frac{\partial E}{\partial \boldsymbol{y}_{equi}^{(l)}}$ at the output side (layer l) is shown in Equation (2.45) below.

$$\boldsymbol{\delta}_{equi}^{(l)} = \sum_{m=0}^{T_{equi}^{(l)}-1} \left[\boldsymbol{\delta}_{equi}^{(l)}\right]_m \tag{2.45}$$

Let the unique weights that connect to the input node at positions $t \in \left\{0, \dots, T_{equi}^{(l-1)} - 1\right\}$ and the output node at position $m \in \left\{0, \dots, T_{equi}^{(l)} - 1\right\}$ be denoted as $\left[\boldsymbol{W}^{(l)}\right]_{t,m}$. To update the unique weights and propagate the delta, simply decompose the activation and delta of the equivalent static network according to Equation (2.43), (2.44) and (2.45), and then substitute them in the standard backpropagation equations, as shown in Equation (2.46) and (2.47) below.

$$\frac{\partial E}{\partial \boldsymbol{W}^{(l)}} = \sum_{m=0}^{T_{equi}^{(l)}-1} \left[\left(\sum_{t=m}^{m+T_{tdnn}^{(l-1)}-1} \left[\boldsymbol{a}_{equi}^{(l-1)} \right]_t \right) \left(\left[\boldsymbol{\delta}_{equi}^{(l)} \right]_m \right)^T \right] \tag{2.46}$$

$$\left[\boldsymbol{\delta}_{equi}^{(l-1)} \right]_t = \left(\sum_{t=0}^{T_{tdnn}^{(l-1)}-1} \sum_{m=max\left(0,t-T_{equi}^{(l)}+1\right)}^{min\left(t,T_{tdnn}^{(l-1)}\right)} \left[\boldsymbol{W}^{(l)} \right]_{t,m} \left[\boldsymbol{\delta}_{equi}^{(l)} \right]_m \right) \odot f' \left(\left[\boldsymbol{a}_{equi}^{(l-1)} \right]_t \right) \tag{2.47}$$

In implementing the distributed TDNN, the activation $\boldsymbol{a}_{equi}^{(l)}$ and the delta values $\boldsymbol{\delta}_{equi}^{(l)}$ of the equivalent delay lines will have to be stored in memory. They will then be available for use by the above equations for backpropagation.

2.2.7 Deep Belief Net

A complex function can be represented compactly by factorization and composition. Figure 2.15 below illustrates this, where a function with eight variables $f(x_1, x_2, \cdots, x_8)$ is distributed to three separate layers in a binary tree network.

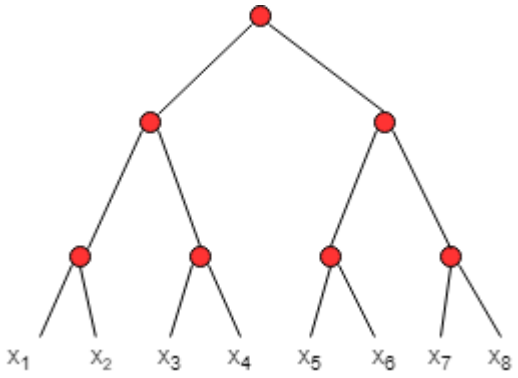

Figure 2.15. Binary tree network

$$f(x_1 \cdots x_8) = h_3 \left(h_{2,1} \left(h_{1,1}(x_1, x_2), h_{1,2}(x_3, x_4) \right), h_{2,2} \left(h_{1,3}(x_5, x_6), h_{1,4}(x_7, x_8) \right) \right) \tag{2.48}$$

The distribution in Figure 2.15 above can be learnt by an MLP network. However, the performance of an MLP with more than 2 hidden layers can be quite poor, due to the computational issue of exploding and/or diminishing gradient. This causes the update direction

to become unreliable during gradient descend. When this happens, more data will not be able to provide more information. [36].

The solution to this problem is to initialize the weights to some "good" values instead of random values. "Good" values can be found by using ReLU (rectified linear unit) as the activation function instead of the sigmoid, and a number of other methods. This will place the deep network in a smooth region of the error surface. One particular method, pre-training by restricted Boltzmann machine (RBM), was described in the paper "A Fast Learning Algorithm for Deep Belief Nets" [11]. This breakthrough started the deep learning movement in 2006.

When done properly, deep learning is relatively immune to overfitting compared to the shallow network. According to Mhaskar et. al. (2017), for a specified test error ϵ, the size of a d-dimensional training set needed by a deep network N_{deep} is smaller than that of a shallow network $N_{shallow}$ by the following ratio:

$$\frac{N_{deep}}{N_{shallow}} \approx \epsilon^{d} \tag{2.49}$$

Suppose the length of the instance is $d \approx 10^2$ and the test error is $\epsilon \approx 0.1$. In this case, the size of the training set required by the shallow network is $N_{shallow} \approx 10^{10^2} N_{deep}$. The size of the training set required by the shallow network is much larger than that required by by the deep network, causing the shallow network to have a much higher chance of overfitting than the deep network.

A DBN is a stack of restricted Boltzmann machines (RBMs). The RBMs are pre-trained in a greedy layer-wise unsupervised manner using contrastive divergence. A final output classifier (for example, a softmax layer) is then added on top of the DBN. This forms a network known as the deep belief net – deep neural network (DBN-DNN). The weights in the DBN-DNN are then fine-tuned by backpropagation. This is shown in Figure 2.16 below.

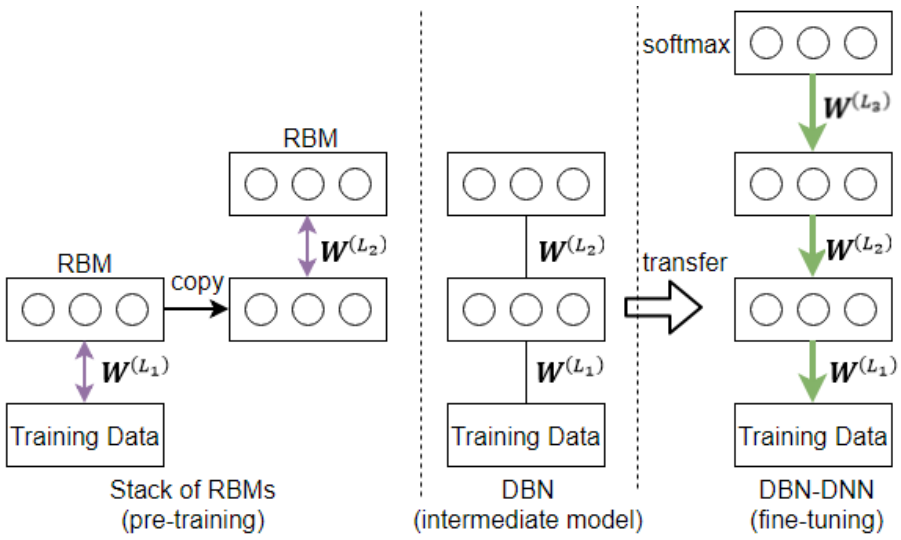

Figure 2.16. Training process of a DBN-DNN

The RBM is the main component in the DBN. The term "restricted" as used in RBM means that the only connections in an RBM are those between the input $\boldsymbol{v}$ and the output $\boldsymbol{h}$. There is no connection between units of the same layer.

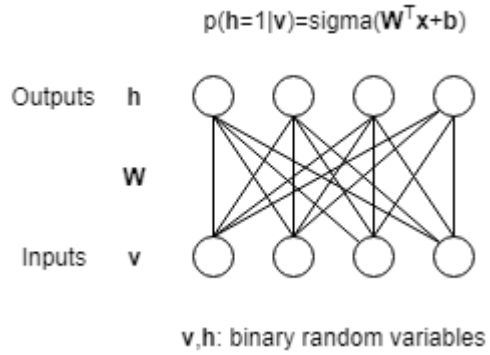

Figure 2.17. The probabilistic model of an RBM

The RBM is a so-called energy-based probabilistic model. In an energy-based model, a function, called the energy function $E(\boldsymbol{v}, \boldsymbol{h})$, assigns an energy value to the pair of binary vectors $\boldsymbol{v}$ and $\boldsymbol{h}$ according to Equation (2.50) below.

$$E(\boldsymbol{v}, \boldsymbol{h}) = -\boldsymbol{b}^T\boldsymbol{v} - \boldsymbol{c}^T\boldsymbol{h} - \boldsymbol{v}^T\boldsymbol{W}\boldsymbol{h} \tag{2.50}$$

where $\boldsymbol{b}$ is the bias of the input, $\boldsymbol{c}$ is the bias of the output, and $\boldsymbol{W}$ is the weight between the input and output. These parameters ($\boldsymbol{b}$, $\boldsymbol{c}$, and $\boldsymbol{W}$) may be collectively denoted as $\boldsymbol{\theta}$.

The energy function $E(\boldsymbol{v}, \boldsymbol{h})$ is related to the joint probability $P(\boldsymbol{v}, \boldsymbol{h})$ as shown below:

$$P(\boldsymbol{v}, \boldsymbol{h}) = \frac{e^{-E(\boldsymbol{v},\boldsymbol{h})}}{Z} \tag{2.51}$$

where Z is the normalizing term. Thus, energy minimization is equivalent to the minimization of the negative log of the probability. The objective is to increase the joint probability of having the input $\boldsymbol{v}$ and the output $\boldsymbol{h}$.

Contrastive divergence is a sequential two-phase approximation of the minimization of the energy. It generates the binary output $\boldsymbol{h}$ from the visible data $\boldsymbol{v}$ and the weights and bias $\boldsymbol{\theta}$ by binary stochastic sampling. This allows the reconstruction of the input by the transposed weight. The negative phase uses the exact copy of RBM as the positive phase except for the use of the reconstructed data $\tilde{\boldsymbol{v}}$ as its input. The negative phase is thus the "dream" version of the positive phase.

In the positive phase, the linear combination of the visible data $\boldsymbol{v}$ by the weights and bias for the i-th hidden unit is the energy as shown below:

$$c_i + \boldsymbol{v}^T \boldsymbol{W}_{.i} \tag{2.52}$$

The probability $P(h_i = 1|\boldsymbol{v})$ is the sigmoid activation of the energy, as shown in Equation (2.53) below.

$$P(h_i = 1|\boldsymbol{v}) = \frac{1}{1 + e^{-(c_i + \boldsymbol{v}^T \boldsymbol{W}_{.i})}} = sigmoid(c_i + \boldsymbol{v}^T \boldsymbol{W}_{.i}) \tag{2.53}$$

where $\boldsymbol{W}_{.i}$ is the i-th column of $\boldsymbol{W}$, a matrix arranged in the $input \times output$ form, and c_i is the bias of the i-th hidden unit.

The binary output of the positive phase, i.e. $\boldsymbol{h}$, is obtained from the probability $P(h_i = 1|\boldsymbol{x})$ by binary stochastic sampling, as shown in Equation (2.54) below.

$$\boldsymbol{h} = P(h_i = 1|\boldsymbol{v}) > uniform(0,1)\ \forall i \in \{0, \dots, |\boldsymbol{h}| - 1\} \tag{2.54}$$

The reconstructed data is obtained from the transposed weight as shown in Equation (2.55) below.

$$\tilde{\boldsymbol{v}} = sigmoid\left(b_j + \boldsymbol{h}^T \boldsymbol{W}^T_{.j}\right)\ \forall j \in \{0, \dots, |\boldsymbol{v}| - 1\} \tag{2.55}$$

where $\boldsymbol{W}^T{}_{.j}$ is the jth column of $\boldsymbol{W}^T$, and b_i is the bias of the i-th reconstructed unit of the negative phase.

$\boldsymbol{v}$ and $\tilde{\boldsymbol{v}}$ play a symmetric role in the energy function. Thus, $P\left(\tilde{h}_j = 1|\tilde{\boldsymbol{v}}\right)$ is the probability of the negative phase.

$$P\left(\tilde{h}_j = 1|\tilde{\boldsymbol{v}}\right) = sigmoid(c_i + \tilde{\boldsymbol{v}}^T \boldsymbol{W}_{.i})\ \forall i \in \left\{1, \dots, \left|\tilde{\boldsymbol{h}}\right|\right\} \tag{2.56}$$

The gradients $\frac{\partial E}{\partial \boldsymbol{W}}$ of both the positive and the negative phase can be computed quite easily as the outer product of the visible vector, $\boldsymbol{v}$ or $\tilde{\boldsymbol{v}}$, and the conditional probability vector of the hidden layer. For the positive phase, this is represented by the symbol $\langle \boldsymbol{v}, p(\boldsymbol{h}|\boldsymbol{v})\rangle$. For the negative phase, this is represented by the symbol $\langle \tilde{\boldsymbol{v}}, p\left(\tilde{\boldsymbol{h}}|\tilde{\boldsymbol{v}}\right)\rangle$.

The gradient values of the positive and negative phase are different, due to binary stochastic sampling applied at the hidden layer in the positive phase. The difference is used to update the weights.

$$\Delta \boldsymbol{W} \propto \langle \boldsymbol{v}, P(\boldsymbol{h}|\boldsymbol{v})\rangle - \langle \tilde{\boldsymbol{v}}, P(\tilde{\boldsymbol{h}}|\tilde{\boldsymbol{v}})\rangle \tag{2.57}$$

The biases ($\boldsymbol{b}$ and $\boldsymbol{c}$) are updated in a similar vein. In Equation (2.58) below, the difference between the probability vector of the hidden layer of the positive and negative phase is used to

update the hidden bias $\boldsymbol{c}$. In Equation (2.59) below, the difference between the input and the reconstructed data is used to update the visible bias $\boldsymbol{b}$.

$$\Delta \boldsymbol{c} \propto P(\boldsymbol{h}|\boldsymbol{v}) - P\left(\widetilde{\boldsymbol{h}}|\widetilde{\boldsymbol{v}}\right) \tag{2.58}$$

$$\Delta \boldsymbol{b} \propto \boldsymbol{v} - \widetilde{\boldsymbol{v}} \tag{2.59}$$

When the inputs are real rather than probabilistic (i.e. not in the range of $(0,1)$), the following changes are needed for contrastive divergence to work: (1) use the real values for the visible layer, and (2) simply use $b_j + \boldsymbol{h}^T \boldsymbol{W}_{.j}^T$ as the reconstructed data instead of its sigmoid (refer to Equation (2.55)).

2.2.8 Convolution Neural Network

CNN has been the workhorse of image processing for a long time. Its initial form was introduced by Fukushima in 1979 as the neocognitron based on the neurophysiological findings on the visual systems of mammals [37]. The standard CNN as known today is credited to the architecture used by LeCun et. al (1998) for the optical character recognition of handwritten digits. It is a stack of layers consisting of groups formed by a convolution layer, a pooling layer and a dropout layer.

The benefit of the CNN is its ability to represent two-dimensional features that are position and scale invariant. This is important when working with images, as the images of the same object are often captured at different distances and orientations.

In the convolution layer of the CNN, a rectangular kernel is slide across the entire input matrix. The patch covered by the kernel at any point in time is the local receptive field of the hidden neuron. The receptive field may overlap with other receptive fields during sliding. This is shown in Figure 2.18 below.

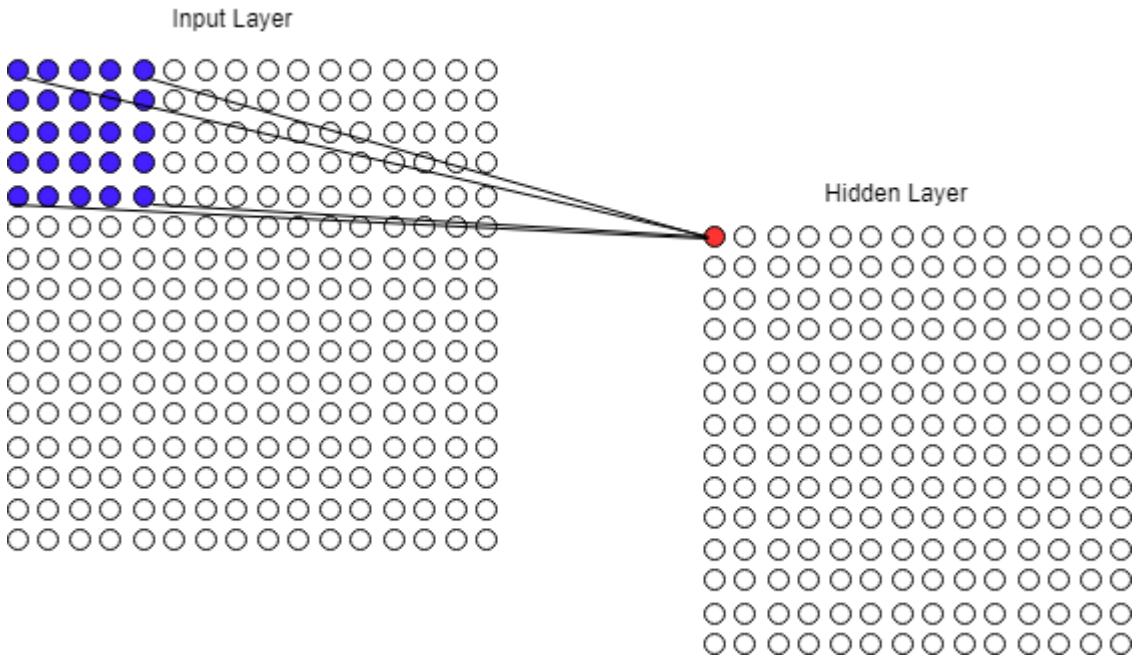

Figure 2.18. Local receptive field in CNN

Kernel sliding is a form of weight sharing with a small local fan-in. This operation will form a feature map at the hidden layer. More than one kernel will be needed if more than one feature map is required. There are usually multiple feature maps at the hidden layer.

The pooling layer comes after the convolutional layer. It is here that pooling is applied to the feature maps. It regularizes the model and prevents the CNN from being overfitted. It does this by summarizing and reducing the data in the feature map. One common kind of pooling is max-pooling. In max-pooling, a pooling unit, which is rectangular patch, will output the maximum value within the feature region covered by it and discard the rest of the values.

Another form of regularization that can be applied to the CNN is dropout [8]. During training, the individual nodes are dropped out of the network with a probability of $1 - p$ (or kept with a probability of p). This will reduce the learning of interdependent set of feature weights among the nodes. During testing, all the nodes are used, but their values will have to be multiplied by the factor p, a fractional number, to take into account the fact that some of the nodes were ignored during training.

The CNN, with kernel sliding, pooling and dropout as described above, works well with images, even when the weights in the first layers are initially random in nature. Thus, unlike the deep belief net, there is no need to pre-train the weights to avoid overfitting.

The massive weight sharing is a particularly strong prior used by the CNN. It is particularly suitable for image processing, and by decomposing the time series into two-dimensional time-frequency representation, can be applied to time series data also.

2.2.9 Long Short-Term Memory Recurrent Neural Network

Unlike the CNN, which was first used on images, the recurrent neural network (RNN) is designed for time series in mind. It passes its internal state to the next input so that both the internal state and the next input can be used to generate the next output. This allows it to make use of the past memory of the time series to predict the output.

The general form of an unrolled RNN is shown in Figure 2.19 below:

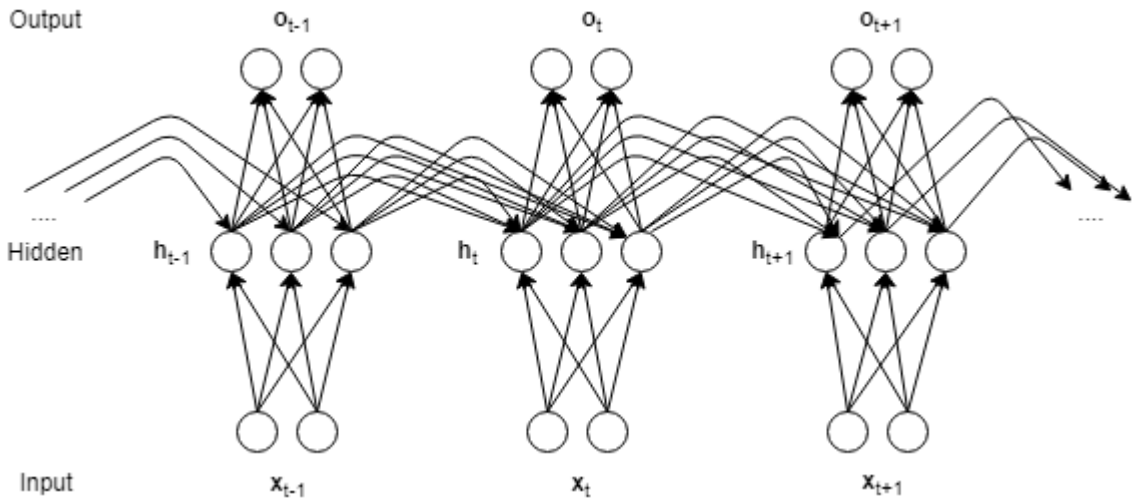

Figure 2.19. General form of an unrolled recurrent network

The application of the backpropagation training algorithm to a recurrent neural network requires that the network be unrolled. This is to enable the error gradient to be calculated and the weights to be updated. When unrolled, at each time step $t \in \{1, \dots, T\}$, each unrolled neural network will take an input $\boldsymbol{x}_t$ and give an output $\boldsymbol{o}_t$, and at the same time pass the hidden state

$\boldsymbol{h}_t$ to the next time step. Notation wise, the hidden state $\boldsymbol{h}_t$ and the output $\boldsymbol{o}_t$ can be expressed as shown in Equation (2.60) and (2.61) below.

$$\boldsymbol{h}_t = \sigma(\boldsymbol{z}_t) = \sigma(\boldsymbol{W}_x \cdot \boldsymbol{x}_t + \boldsymbol{W}_h \cdot \boldsymbol{h}_{t-1} + c) \tag{2.60}$$

$$\boldsymbol{o}_t = softmax(\boldsymbol{W}_o \cdot \boldsymbol{h}_t) \tag{2.61}$$

Backpropagation through time can be computationally expensive when the number of time steps is increased. More importantly, this will cause the weights to vanish or explode which can make the learning slow and the generalization performance poor [39]. This can be understood by using the chain rule of derivative along the path of the backpropagation of the unrolled network, as shown in Figure 2.20 below.

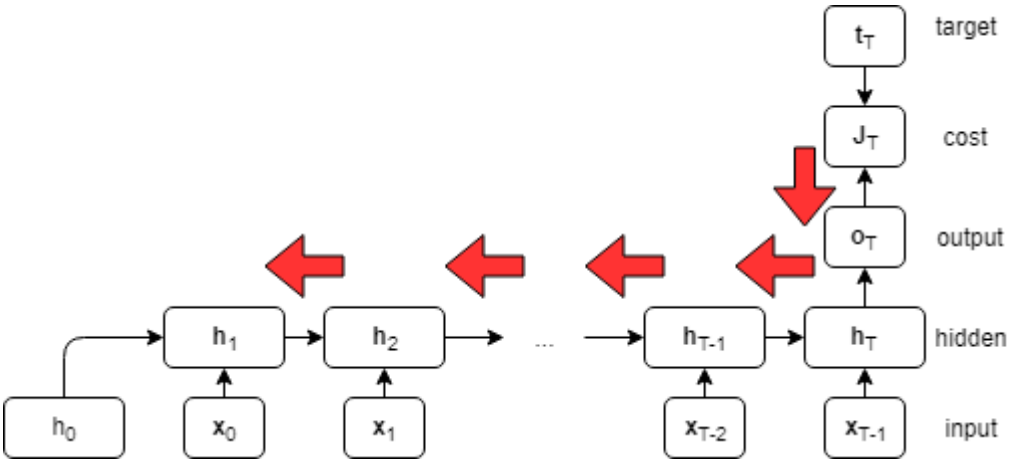

Figure 2.20. Unfolding of a general RNN

As can be seen from Figure 2.20, the error gradient $\frac{\partial J}{\partial \boldsymbol{W}}$ depends on the chain of hidden states $\boldsymbol{h}_t$ all the way from $t = T$ to the current time step. This will cause the weights $\boldsymbol{W}_h$ to be multiplied repeatedly, leading to the problem of exploding or vanishing gradient. This is shown in the next few equations, starting from Equation (2.62) below.

Equation (2.62) shows the error gradient based on the chain rule.

$$\frac{\partial J}{\partial \boldsymbol{W}} = \frac{\partial J}{\partial \boldsymbol{o}_T}\frac{\partial \boldsymbol{o}_T}{\partial \boldsymbol{h}_T}\left(\prod_{t\in\{T,\ldots,2\}}\frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_{t-1}}\right)\frac{\partial \boldsymbol{h}_1}{\partial \boldsymbol{W}} \tag{2.62}$$

Equation (2.63) below rewrites the term $\frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_{t-1}}$ as the chain of terms over the linear value $\boldsymbol{z}_t$ of the hidden layer $\boldsymbol{h}_t$, where $g(\boldsymbol{z}_t)$ is the activation function and $g'(\boldsymbol{z}_t)$ its derivative.

$$\frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_{t-1}} = \frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{z}_t}\frac{\partial \boldsymbol{z}_t}{\partial \boldsymbol{h}_{t-1}} = g'(\boldsymbol{z}_t)\boldsymbol{W}_h \tag{2.63}$$

This leads to a re-expression of the error gradient in Equation (2.62) above to the form as shown in Equation (2.64) below. Here, it can be seen that the weights $\boldsymbol{W}_h$ are being multiplied repeatedly.

$$\frac{\partial J}{\partial \boldsymbol{W}} = \frac{\partial J}{\partial \boldsymbol{o}_T}\frac{\partial \boldsymbol{o}_T}{\partial \boldsymbol{h}_T}\left(\prod_{t\in\{T,\ldots,2\}} g'(\boldsymbol{z}_t)\boldsymbol{W}_h\right)\frac{\partial \boldsymbol{h}_1}{\partial \boldsymbol{W}} \tag{2.64}$$

Repeated multiplication of the weights $\boldsymbol{W}_h$ is problematic, as it will lead to exploding or vanishing gradient. If the largest eigenvalue of $\boldsymbol{W}_h$ is 1, the gradient will propagate along the backward path, which is fine. However, if it is < 1, the product will vanish along the backward path, and if it is >1, the product will explode along the backward path.

LSTM was introduced by Hochreiter & Schmidhuber (1997). It is effective in solving the problem of the vanishing/exploding gradient problem in recurrent neural network. In LSTM, the state information is carried through the time in an additive (rather than multiplicative) manner. Instead of using $\boldsymbol{h}$ as both the state information and the carrier of that information through time, it is now used solely as the state information at a particular time step. The carrier of the state information is stored in a new variable, called the cell state $\boldsymbol{c}_t$, which is the state of a new kind of unit called the cell.

An LSTM cell has three gates (the input gate, the forget gate, and the output gate). All the gates have a sigmoid activation function and a set of weights for $\boldsymbol{W}_x$ and $\boldsymbol{W}_h$. The weights for the three gates are denoted as $\boldsymbol{W}_i$, $\boldsymbol{W}_f$, $\boldsymbol{W}_o$ respectively. Their outputs (serving as the intermediate

values for the subsequent element-wise addition or multiplication) are denoted as $\boldsymbol{i}_t$, $\boldsymbol{f}_t$, and $\boldsymbol{o}_t$. In addition, there is another set of weights $\boldsymbol{W}_c$ for the cell state.

The following three equations (Equation (2.65), (2.66), and (2.67)) show how the gate outputs for the input gate, the forget gate and the output gate are determined.

The output of the input gate $\boldsymbol{i}_t$ is computed as follows:

$$\boldsymbol{i}_t = \sigma(\boldsymbol{W}_i \cdot [\boldsymbol{h}_{t-1}; \boldsymbol{x}_t]) \tag{2.65}$$

The output of the forget gate $\boldsymbol{f}_t$ is computed as follows:

$$\boldsymbol{f}_t = \sigma\big(\boldsymbol{W}_f \cdot [\boldsymbol{h}_{t-1}; \boldsymbol{x}_t]\big) \tag{2.66}$$

The output of the output gate $\boldsymbol{o}_t$ is computed as follows:

$$\boldsymbol{o}_t = \sigma(\boldsymbol{W}_o \cdot [\boldsymbol{h}_{t-1}; \boldsymbol{x}_t]) \tag{2.67}$$

The operations of an LSTM cell is summarized in Figure 2.21 below.

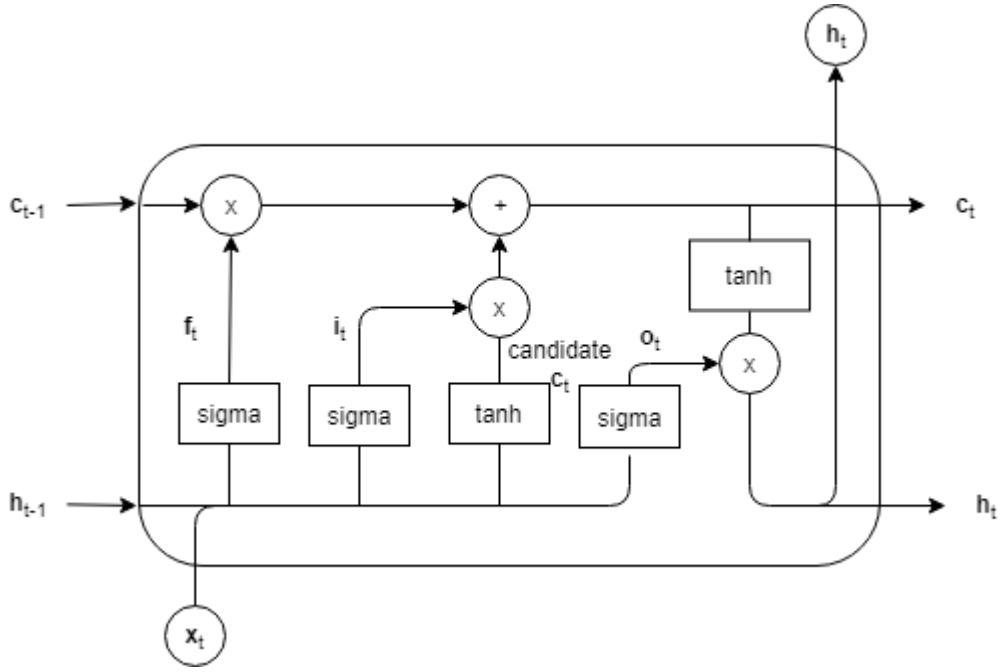

Figure 2.21. An LSTM cell

The candidate information of what goes into the cell state $\boldsymbol{c}_t$ is determined as shown in Equation (2.68) below. The tanh function will push the values to be between -1 and 1.

$$candidate\ of\ \boldsymbol{c}_t \triangleq \tilde{\boldsymbol{c}}_t = tanh(\boldsymbol{W}_c \cdot [\boldsymbol{h}_{t-1}; \boldsymbol{x}_t]) \tag{2.68}$$

The cell state $\boldsymbol{c}_t$ is computed from the candidate $\tilde{\boldsymbol{c}}_t$ and the gate outputs ($\boldsymbol{i}_t$ and $\boldsymbol{f}_t$). Note that it is related to the previous cell state by addition, not multiplication.

$$\boldsymbol{c}_t = \boldsymbol{i}_t \odot \tilde{\boldsymbol{c}}_t + \boldsymbol{f}_t \odot \boldsymbol{c}_{t-1}$$

The hidden state $\boldsymbol{h}_t$ is computed from the output gate $\boldsymbol{o}_t$ and the cell state $\boldsymbol{c}_t$, as follows:

$$\boldsymbol{h}_t = \boldsymbol{o}_t \odot tanh(\boldsymbol{c}_t) \tag{2.69}$$

Beside the LSTM, there are many other variants of the recurrent neural network, such as the gated recurrent unit (GRU) [41]. In general, these recurrent networks are designed for time series forecasting rather than time series classification [42]. However, they can be stacked with other classifiers, such as CNN, to provide time series classification.

2.3 Time Series Classification

Time series problems are either classification or forecasting. Since the classification of time series is more pertinent in biosignals than forecasting, this section will focus on time series classification rather than time series forecasting.

Some words on time series forecasting first, to make clear how it is different from time series classification. In forecasting, the future values are predicted a few time steps away from the previous values. Accuracy measures such as symmetric mean absolute percentage error (sMAPE) and mean absolute scaled error (MASE) [43] are used as the performance metrics. The state of the art in time series forecasting still lies in statistical models such as exponential smoothing (ETS), autoregressive integrated moving average (ARIMA) [44] and multivariate generalized autoregressive conditional heteroskedasticity (GARCH) [45] models. It was found [46] that these statistical models outperform many machine learning methods,

including multilayer perceptron network (MLP) and LSTM recurrent neural network. How deep learning can be superior to the statistical models in time series forecasting remains an open research question. For example, a stack of LSTM could outperform the ARIMA model [47].

In time series classification, a short sequence of a time series is associated with a class label [48]. For example, in electrocardiography (ECG), several electrodes are used to pick up the electrical activity of the heart on the body surface, and the signals are collectively labelled as either normal or abnormal based on the pre-defined set of classes. Researchers have devised many methods to solve the problem of time series classification. These methods can generally be categorized as (1) feature-based, (2) distance-based, (3) neural network-based, i.e. deep learning, and (4) ensemble-based.

2.3.1 Feature-Based Methods

The traditional approach in time series classification is feature-based. Feature extraction ignores the redundant and irrelevant information and maps the time series to a lower dimensional space. It can be done by morphological or statistical feature extraction, time series analysis, and decomposition techniques. Morphological features are the measures or ratios of pertinent points in the signal, such as the electrocardiogram's peak-to-peak interval, i.e. RR value, while statistical features are the frequency counts or the aggregated summary of the sub-sequences in the signals. In time series analysis, the generative process of the time series [42] is represented by a time series statistical model, such as the autoregressive model [49], the linear dynamic system [50], and the hidden Markov model (HMM) [51]. In decomposition, the components of the time series are obtained using non-parametric techniques such as principal component analysis, non-negative matrix factorization and factor analysis. The model parameters or the components are then used as features in machine learning [52]. The following are some works in the literature based on these techniques.

Kampouraki et al. [53] classified electrocardiogram by extracting features with multiple techniques, including morphological and statistical feature extraction, time series analysis, and decomposition technique. The feature extraction started with the segmentation of the signals into the QRS complex, which is an easily identifiable shapelet in electrocardiogram. After this, the RR intervals were measured from the QRS complexes. Statistical features often used in

heart rate variability, such as the standard deviation, root mean square of successive differences, autocorrelation, Shannon's entropy etc. were computed from the series of RR intervals. They also used the local linear prediction (LLP), a simple autoregressive prediction method [54], to derive the features based on the mean values of the absolute differences between the predicted and the actual values. Furthermore, they decomposed the electrocardiogram by discrete wavelet transform with the Haar wavelet [55]. The standard deviation of the detail coefficients, representing the high-frequency content of the signal, are used as the features [56]. All these features were then used in a support vector machine (SVM).

Nigam et al. [57] extracted two time-domain attributes of EEG, namely, relative spike amplitude and spike rhythmicity, and used them in neural network to detect epilepsy. They also used frequency-domain features such as dominant frequency, average power in the main energy zone, and normalized spectral entropy.

Zhou et al. [58] determined the relationships within the same variable (intra-temporal patterns) and between different variables (inter-temporal patterns) at different time points based on data statistics and frequency counts, and then combine the degree of these relationships as feature vector. Thereafter, classifiers such as SVM and neural network were applied to the statistical features.

Andrzejak et al. [15] analysed EEG with correlation dimension, a measure in chaos theory of the dimensionality of the space occupied by a set of random points. They concluded that there are significant differences in the features derived from correlation dimension in subjects that are normal and those with seizure activity.

Flexer et al. [59] developed a sleep feature extractor based on the Gaussian Observation HMM (GOHMM) model using only a single EEG signal. HMM exploits the probabilistic dependence of successive sleep stages. They used the coefficients of the AR process as the reflection coefficients in the GOHMM model, and Viterbi decoding to identify the most likely sequence corresponding to a time series and compute the probabilities of being in any one of the states at any point in time.

J. Grabocka et al. [60] used non-negative matrix factorization to project time series data, all thirty seven data sets, into a latent space through stochastic learning. The projected data was then used to train a logistic regression model.

Li et al. [61] applied singular value decomposition (SVD) on the data, where eigenvalue analysis is carried out in order to find an optimal set of features, and then apply SVM on the feature vectors to classify the time series data.

Weng et al. [62] projected the data into the PCA subspace by throwing away the smallest principal components, and then project the PCA subspace further with supervised Locality Preserving Projection (LPP) [63]. LPP finds the optimal linear approximations to the eigenfunctions of the Laplace Beltrami operator [14] on the data manifold. The LPP is a linear method, although it shares many of the data representation properties of nonlinear techniques such as Laplacian eigenmaps.

The problem with the feature-based approach is that the important correlation among the variables may be lost during feature extraction, and some of the features that have different lengths cannot be extracted easily. Also, the data analyst will not be able to extract the features if he does not know the important aspects of the complex signal.

2.3.2 Distance-Based Methods

There are two main approaches in distance-based methods. The first approach works directly on the time series data by comparing the sequences as a whole and use the similarity between them as the features, such as dynamic time warping (DTW) [64], while the second approach is based on subsequence discovery techniques that searches for recurrent patterns in the time series, such as motif analysis or shapelet discovery [65].

Working directly with the time series data is a global feature approach, while subsequence discovery technique is a local feature approach. The global feature approach is relatively simple and is the reason why the 1-nearest neighbour (1-NN) and other forms of k-nearest neighbour (k-NN) classifiers [66] are widely used. They are simple methods that produce good performance, even though they are sensitive to noise in the training set and are high in

classification time due to the need to scan the entire training set before taking a decision at test time in an instance-based classifier. Their effectiveness is due to the distance measures, as the shapes and structures of the sequences, such as temporal dependency, high dimensionality, and variable lengths, have to be well represented by the distance measures. These distance measures are either lock-step distance measures or elastic distance measures [66]. Lock-step distance measures, such as the Euclidean distance, compare the i-th point of one sequence to the i-th point of another sequence, whereas elastic distance measures, such as dynamic time warping (DTW), creates a non-linear mapping in order to align and synchronize the sequences. It calculates the distance between each possible pair of points out of two signals and then builds a cumulative distance matrix to find the least expensive path through the matrix. Usually, the signals are normalised and smoothed before the distances between the points are calculated. Experimental comparison of distance measures of time series data suggests that the DTW distance measure is difficult to beat [67][68].

The complexity of DWT is quadratic in the length of the sequences. Holt et al. [69] used a form of DTW called derivative DTW on multi-dimensional time series for gesture recognition. In derivative DTW, the distances are calculated not between the sequences but their associated first order derivatives. In this way, synchronisation is based on slopes and peaks rather than the raw values. They found that derivative DTW performed well under noisy conditions.

For complex classifiers other than k-NN classifiers, distance measures are used to represent the time series as order-free feature vectors in $\mathbb{R}^N$, bridging the gap between the time series and the general purpose classifiers. A global distance matrix [70] is built by calculating the distances between each pair of sequences, and then each row of the distance matrix is used as the feature vector for the sequence. The drawback of using the global distance matrix is its computational intensity, as an $n \times n$ matrix needs to be computed for a data set with n instances or sequences.

Gudmundsson et al. [71] used DTW to build the global distance matrix and then applied it to an SVM classifier. They concluded that the new representation in conjunction with SVM is competitive with the benchmark 1-NN with DTW. Jain et al. [72] used PCA to reduce the dimensionality of the $n \times n$ global distance matrix by keeping only those dimensions that are important. They found that PCA can have a consistent positive effect on the performance of the

classifier but this effect seems to be dependent of the choice of the kernel function applied in the SVM.

The distance measures in the global distance matrix can be embedded as a vector representation in a new feature space while preserving the distances [73]. Embedding techniques include multidimensional scaling, pseudo-Euclidean space embedding and Euclidean space embedding by the Laplacian eigenmap technique [74]. Mizuhara et al. [75] experimented with various embedded techniques and found that the Laplacian eigenmap-based embedded method achieved a better performance than the 1-NN classifier with DTW.

The aforementioned methods work on transforming distances to feature vectors. For some machine learning algorithms, for example SVM, the feature vector form is not required. Only the similarity of the input objects are required. If two inputs are similar, their kernel output will be similar too. The kernel approach can handle any kind of data, including time series and images, as long as there is a suitable kernel that can capture the similarity between pairs of inputs. The kernel method has worked successfully with biosignals, although there are not many "benchmark" kernels for time series in the literature [76]. Chaovalitwongse et al. [77] made use of the Gaussian kernel and the Fourier kernel [78] of the DTW distance to classify normal and pre-seizure electroencephalograms. The Fourier kernel is useful when the time series has spectral patterns due to certain events, such as the spikes in the electroencephalogram.

In summary, distance-based methods, in particularly DTW and its variants, are widely used in time series classification. Nevertheless, learning with the distance features can often become cumbersome, depending on the length and size of the time series data. Modified DTW and other dimensionality reduction technique are often required to lower the otherwise intractable computational cost.

2.3.3 Neural-Network-Based Methods

The neural network-based method, also known as deep learning, is an end-to-end method. It is exciting, as it is able to extract features from the raw signals without the need to perform feature engineering or specify distance measure.

The foundation of all deep learning networks is composition. In composition, the output of a layer becomes the input of the next layer. This kind of compositional structure matches with the compositional function of many natural signals such as image, text and speech [6]. These signals have what is called the property of locality [79], which means that the features formed by neighbouring points are related to one another at different scales and time.

There are many network architectures for deep learning. Most of them are for image classification, and only some are for the one-dimensional time series data. Although there exist many types of deep neural networks, the three main deep learning networks in time series classification are the multilayer perceptron (MLP), the convolutional neural network (CNN) and the echo state network (ESN) [80].

The baseline network for time series classification is the MLP [81]. It consists of hidden layers that are fully connected. The MLP is often preceded by an unsupervised pre-training phase, such as stacked denoising auto-encoders (SDAEs) [82]. The auto-encoder learns the latent features from the unlabelled data in an unsupervised manner, which are then leveraged to classify the time series by training the MLP with the labels. This kind of MLP with auto-encoders is called deep belief net – deep neural network (DBN-DNN).

Längkvist et al. [83] applied DBN-DNN to sleep stage classification. They showed that an automatic sleep stager, i.e. auto-encoder, can be applied to multimodal sleep data without using any hand crafted features.

Wang et al. proposed a form of DBN called the Cycle DBN where they performed unsupervised learning to discover the structure hidden in the data. In essence, the Cycle DBN predicts the label at time t not only based on the current input but also the previous label at $t-1$.

The convolutional neural network is widely used in time series classification with two-dimensional time-frequency representation [84]. Variants of CNN proposed for the time series classification include the fully convolutional networks (FCN) [85], multi-channel deep convolution neural network (MC-DCNN) [86] and residual network (ResNet) [87]. They make use of layers such as the batch normalization (BN) layer [88] and the global average pooling

(GAP) layer to reduce the number of parameters to avoid overfitting, and shortcut links to reduce the vanishing gradient effect.

Karim et al. [89] augmented the FCN with LSTM to significantly enhanced the FCN with a nominal increase in model size. In particular, they used the attention mechanism of LSTM to visualize the decision process of the LSTM cell.

An ESN consists of an input layer, a non-linear reservoir layer and a linear readout layer. The reservoir layer is a large, sparsely connected, tanh-activated, recurrent layer used for the contrastive encoding of the history of driving input signals into a state space. There is no need to compute the gradient for the reservoir layer. The inter-connected weights inside the reservoir and the input weights are not learnt by gradient descent; only the output weights are tuned using a learning algorithm such as the logistic regression or softmax classifier, which in a way is similar to the extreme learning machine (ELM). This reduces the training time of ESN and avoid the vanishing gradient problem.

D. Bacciu et al. [90] focused on a particular variant of ESN [91] called the leaky integrator ESN (LI-ESN) [92]. In LI-ESN, the standard tanh reservoir units are replaced by the leaky integrator units, which apply an exponential moving average to the reservoir state values for better handling of slow-changing input sequences. Using the LI-ESN, they were able to predict the change of location of a subject as he moved room-to-room, based on the received signal strength of the wireless sensors placed around the rooms.

2.3.4 Ensemble Methods

Single algorithm, whether feature-based, distance-based or neural network-based, may perform poorly on the test set due to overfitting to the training data set. An ensemble is a set of base classifiers, or sub-models, that are fused together. It makes use of weak base classifiers to inject diversity into the system and then constructs a strong learner from the combination of the base classifiers. Broadly speaking, the diversity can be achieved by employing different algorithms as the sub-models, changing the training data of the sub-models by resampling the data, training the sub-models on different attributes, or re-weighting the training data. The diversity reduces the need to optimize the parameters of the sub-models and thus improves on the generalization

performance [93], at the expense of more training time. It can be used for time series classification. For example, Deng et al. proposed a random forest built on summary statistics of intervals [94], and Buza et al. [95] proposed an ensemble that combines alternative elastic and inelastic distance measures.

Lines et al. [66] found that ensembling individual nearest neighbour classifiers with different distance measures, e.g. weighted and derivative DTW, edit distance-based measures, time warp with edit, etc., results in significantly better accuracy than the ensemble's individual components. Many other classifiers can be ensembled as well, such as decision trees (random forest) [94] and SVM [96].

COTE (Collective Of Transformation-based Ensembles) [97] is an ensemble of 35 classifiers. Not only does it ensemble different classifiers over the same transformation, it also ensembles different classifiers over different time series representations.

HIVE-COTE [98] extended COTE with a Hierarchical Vote system. It achieved improvement over COTE by leveraging a new hierarchical structure with probabilistic voting. It is considered the state-of-the-art algorithm for time series classification [68] when evaluated over the 85 datasets from the UCR archive [99].

COTE and HIVE-COTE are hugely computationally intensive and impractical to run on a real big data mining problem [68]. It requires training 37 classifiers, as well as cross-validating their hyper-parameters. Some of the classifiers are time-consuming to run, such as shapelet transform and nearest neighbour classifier. For example, the time complexity of shapelet transform [100] is $\mathcal{O}(n^2 \cdot l^4)$ where n is the number of time series in the data set and l is the length of each time series [100]. The nearest neighbour classifier needs to scan the training set before taking a decision at test time, which is a problem when the training set is large.

Besides ensembling traditional machine learning models, it is also possible to ensemble deep neural networks too [85]. This approach has the desirable characteristics of end-to-end processing, with the added bonus of a lift in performance.

Deng et al. [101] went beyond basic linear stacking to the log-linear stacking of sub-models (DNN, CNN, and RNN) in an ensemble.. In linear stacking, the outputs of the sub-models are linearly combined, whereas in log-linear stacking, the outputs of the sub-models are subjected to the logarithmic function. The stacking parameters are learned using both the training and validation data, with the cost function based on the total square error. When used on raw acoustic signals in speech, it resulted in a significant increase in phoneme recognition accuracy.

Lin et al. [102] developed a DNN-based ensemble based on the "local and distorted" view transformation. First, different filtering methods were used to pre-process the time series, and then downsampled selectively. These views were then used to train two separate DNNs. In the testing phase, the "subview prediction" of the two DNNs were averaged out for use by the final classifier. Experiment results on the CCDD database [103] demonstrate that the method is effective in electrocardiogram time series classification.

2.4 Multi-view Learning

Multi-view learning, or data fusion from multiple feature sets, is an emerging direction in machine learning [104]. In multi-view learning, more than one feature set is used for learning. The features may be redundant, but they are not entirely similar. As such, besides learning the patterns in the features, the relationship among the feature sets can be used for learning too.

Figure 2.22 below illustrates the concept of multi-view data. It shows two views with some overlap between them, highlighting that: (1) Part A and Part C exist as unique views, i.e., the two views are complementary, (2) Part B is shared by both views, i.e., the two views are supplementary.

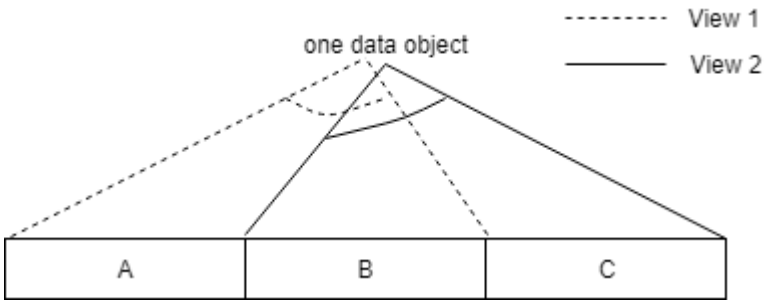

Figure 2.22. Two Views

The objective of multi-view learning is to improve the generalization performance by exploiting the discriminatory information in the views to complement each other, and the similarity among the views to supplement each other.

An empirical way to check for the existence of complementarity in a data set is to first use all the features in the data set for classification and then reduce the size of the feature set progressively. The removal of the features will reduce the discriminatory information available for training, which will likely cause the performance to drop, providing the evidence that these features contribute to the training of the model.

2.4.1 Review of Multi-View Learning

This sub-section provides a general survey of the methods and techniques used to address the multi-view learning problem.

Multi-view learning was first introduced as a framework by Blum and Mitchell [105] for the semi-supervised learning of web page classification. The text of the web pages and the anchor text in the hyperlinks of the web pages were used as the feature sets in the two-view setting. Using co-training, two separate models built on the two disjoint views were used to predict the unlabeled data. This was used to decide on which unlabeled data to add to the training set. In this way, the training set can be enlarged for further training.

A survey on multi-view learning by Sun [106] reviews the theories, properties and behaviours of multi-view learning. It shows that multi-view learning, as an emerging and rapidly growing field in machine learning, has been used in all branches of machine learning, from unsupervised learning [107], semi-supervised learning, active learning [108], supervised learning, transfer learning [109], and ensemble training [110]. Some examples of applications include the sentiment analysis of the attitude or opinion of a user [111], and speech analysis for phonetic recognition [112]. As for the use of deep learning in multi-view learning, the first such application was on audio-visual speech recognition [113].

At the moment, there are generally three ways to do multi-view learning: (1) early fusion, where the data from different sources are concatenated and then fed to a single learner, (2) intermediate

fusion, where the data from different sources are combined as the common features for the final classifier, and (3) late fusion, where the decisions by a number of classifiers are combined in a fixed or trained combiner. By this categorization, the aforementioned framework by Blum and Mitchell [105] is a late fusion technique applied to semi-supervised learning.

Yan Zheng [114] proposed selection strategies (multi-view simple disagreement sampling, and multi-view entropy priority sampling) for the active learning, i.e. semi-supervised learning, of environmental sounds. He used two views, namely (1) the CELP features in 10 dimensions, and (2) the MFCC features in 13 dimensions. One of the two views is selected for use by the final classifier based on the outcome of the selection strategies.

Teng Niu [111] showed that the human perception in image-text pairs (collected from Twitter) can be learnt using bag-of-words (BoW) as textual features and SIFT, GIFT etc. as visual features. In early fusion, the textual features and the visual features were concatenated as a single feature vector. In late fusion, the output of the base classifiers are weighted to produce a final score.

An important part of multi-view learning is the construction of the views. The views may be naturally distinct, as in the text of the web page and the anchor text in the hyperlink of the web page, or the video and audio signals of a multimedia content [115]. They may be distinct due to the feature extraction methods used on the raw data, such as the CELP features and the MFCC features of an audio signal [114]. They may be subsets that are split from a single feature set, based on the ordered importance of the features in the feature set.

When multi-view features are not available, random feature split of a single view can be used to construct artificial views. This can be done by, for example, resampling the data after data augmentation, or generate the outputs from several networks that are configured differently. The use of the artificial views in multi-view learning can still improve the generalization performance, even though they are not natural views. This is because multi-view learning is robust to the violated assumptions of its underlying classifiers [116].

The architectures for the two types of multi-view data, namely natural and artificial data, are shown in Figure 2.23 below:

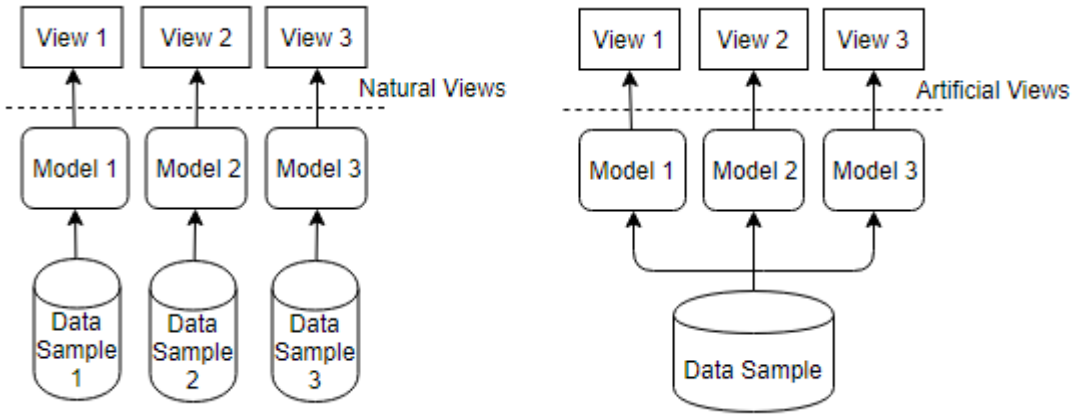

Figure 2.23. Natural (left) and artificial (right) views for multi-view learning.

2.4.2 Ensemble Learning

From the point of view of multi-view learning, the ensemble is a technique to blend multiple views together. It can be used for either intermediate or late data fusion, depending on whether the output of the ensemble is the final output of the multi-view learning, or whether the output serves as the intermediate feature used by the final classifier. This sub-section will discuss ensemble learning based on the way the outputs from the base learners are combined, namely (1) ensemble by bagging, (2) ensemble by boosting, and (3) ensemble by stacking [117].

In ensemble by bagging, multiple random subsets are created, with replacement, from the original training data set [118]. The final prediction is by averaging, if it is regression, and by voting, if it is classification. An example of ensemble by bagging is random forest.

Bagging can be applied to imbalanced data set by applying the stratified resampling method. It can be summarized as follows: (1) resample the minor class with replacement, using a size that is the same as the minor class, (2) resample the major class with replacement, using a size that is smaller than the minor class.

To prove that bagging can reduce overfitting, let's say there are k random subsets, and that each subset $i \in \{1 \cdots k\}$ results in an error $\boldsymbol{e}^{(i)}$ for a particular input vector. For k subsets, the average error for that input vector will be $\frac{1}{k}\sum_{i=1}^{k} \boldsymbol{e}^{(i)}$. Suppose the variance of the error is

$\mathbb{E}\left[\left(\boldsymbol{e}^{(i)}\right)^2\right] = v$, and that the covariance between the errors made by two such subsets is $\mathbb{E}\left[\boldsymbol{e}^{(i)}\boldsymbol{e}^{(j)}\right] = c$. In this case, the expected squared error made by the ensemble will be:

$$\mathbb{E}\left[\left(\frac{1}{k}\sum_{i=1}^{k}\boldsymbol{e}^{(i)}\right)^2\right] = \frac{1}{k^2}\mathbb{E}\left[\sum_{i=1}^{k}\left(\left(\boldsymbol{e}^{(i)}\right)^2 + \sum_{j\neq i}\boldsymbol{e}^{(i)}\boldsymbol{e}^{(j)}\right)\right] \tag{2.70}$$

$$= \frac{1}{k}v + \frac{k-1}{k}c$$

If the errors are perfectly correlated ($c = v$), the mean squared error will be reduced to v, so ensemble averaging will not help with reducing the error. However, if the errors are perfectly uncorrelated ($c = 0$), the expected squared error will be reduced to $\frac{1}{k}v$. This ideal situation will occur if the outputs of the subsets are independent of each other.

Another way of looking at it is to denote the expectation of the squared error for the i-th sub-model as $\mathbb{E}\left[\left(\boldsymbol{e}^{(i)}\right)^2\right]$. This will result in the average performance of the k subsets to be

$$E_{ave} = \frac{1}{k}\sum_{i=1}^{k}\mathbb{E}\left[\left(\boldsymbol{e}^{(i)}\right)^2\right] \tag{2.71}$$

By considering the Cauchy's inequality,

$$E_{ensemble} = \mathbb{E}\left[\left(\frac{1}{k}\sum_{i=1}^{k}\boldsymbol{e}^{(i)}\right)^2\right] \leq \frac{1}{k}\sum_{i=1}^{k}\mathbb{E}\left[\left(\boldsymbol{e}^{(i)}\right)^2\right] = E_{ave} \tag{2.72}$$

It is clear that $E_{ensemble} \leq E_{ave}$, which indicates that the ensemble can give more accurate and reliable estimations than the average performance of the individual models.

The second type of ensemble, ensemble by boosting, is based on the theory of probability approximately correct (PAC) learning [119]. It uses many weak learners in cascade, where the

instances deemed difficult by the previous weak learner will have a higher chance to be trained by the next weak learner. The final prediction is the linear combination of the outputs of these weak learners.

The first practical boosting method in the literature is Adaptive Boosting, or AdaBoost for short [120]. It is a method developed for binary classification. The steps in AdaBoost are as follows: (1) sample a subset (with replacement) from the training data, according to the weights of the data instances, (2) train with the subset, (3) test with the total training data set, and adjust the strength of the weak learner and the weights of the data instances accordingly, and (4) repeat the procedure for the next weak learner.

The strength $\alpha^{(i)}$ of the i-th weak learner in the aforementioned step (3) can be computed as follows:

$$\alpha^{(i)} = \ln \frac{(1-\varepsilon)}{\varepsilon} \tag{2.73}$$

ε is the weak learner's weighted error rate for the total training data set. It is computed as shown in Equation (2.74) below.

$$\varepsilon = \frac{sum\left(\boldsymbol{w}^{(i)} \odot I\left(\boldsymbol{t} \odot \boldsymbol{y}^{(i)} \in -1\right)\right)}{sum(\boldsymbol{w}^{(i)})} \tag{2.74}$$

In Equation (2.74) above, $\boldsymbol{t}$ (the target) and $\boldsymbol{y}^{(i)}$ (the prediction of the i-th weak learner) have values of either 1 or -1 for each of the instances, and $I(\cdot)$ is an indicator function that produces a 1 if the prediction is wrong and 0 otherwise.

A weak learner with $\varepsilon = 50\%$ will have a strength of 0. The strength will be exponentially positive if ε is lower than 50%, and exponentially negative if it is higher than 50%.

The weights $\boldsymbol{w}^{(i+1)}$ for the next weak learner, i.e. the $i+1$-th weak learner, can be computed after the strength $\alpha^{(i)}$ is computed. This is shown in Equation (2.75) below:

$$\boldsymbol{w}^{(i+1)} = \boldsymbol{w}^{(i)} \odot \frac{e^{\alpha^{(i)} I(\boldsymbol{t} \odot \boldsymbol{y}^{(i)} \in -1)}}{Z} \tag{2.75}$$

The weights $\boldsymbol{w}^{(i+1)}$ is a vector where the vector element $w_j^{(i+1)}$ represents the probability for the j-th data instance to be selected for the $(i+1)$-th weak learner. To make it a distribution, the weights $\boldsymbol{w}^{(i+1)}$ will have to be divided by Z, the sum of all the vector elements in $\boldsymbol{w}^{(i+1)}$. As seen in Equation (2.75) above, a wrong prediction will result in a larger weight value for the data instance. The larger weight value will give the data instance a bigger chance of being used by the next weak classifier.

The final prediction is the weighted combination of the outputs of the weak learners.

$$y(\boldsymbol{x}) = sign\left(\sum_{i=1}^{k} \alpha^{(i)} \boldsymbol{y}^{(i)}\right) \tag{2.76}$$

The number of weak learners k is a user-defined parameter in AdaBoost. The training error will eventually reach zero if more and more weak learners are trained. The optimal generalization performance will likely be reached before that, as overfitting will kick in with more weak learners are added.

The gradient boosting machines (GBM) is a powerful ensemble technique that is built upon AdaBoost. It minimizes the cost function by sequentially adding a weak learner (usually a regression tree) in a gradient descent like procedure to reduce the cost [121].

A regression tree is a decision tree with a linear regression model at the terminal nodes (i.e. leaves). A leaf acts as a small partition of the data space. A data instance $\boldsymbol{x}$, starting from the root node, will follow the splits along the tree and reach the leaf that it belongs. Each split is dichotomous and is usually based on a single attribute.

The steps to train a GBM are: (1) train the first weak classifier with the original target class labels, (2) choose the errors between the target class labels and the predicted values as the labels

for the next classifier, and (3) keep adding weak learners until the function reaches the convergence.

The errors mentioned in the aforementioned step (2) are computed as shown in Equation (2.77) below, where $\boldsymbol{x}$ is the input, $\boldsymbol{y}(\boldsymbol{x})$ is the predicted value, $\boldsymbol{t}$ is the target class labels, and $\boldsymbol{e}$ is the error made by the weak learner.

$$\boldsymbol{e} = \boldsymbol{t} - \boldsymbol{y}(\boldsymbol{x}) \tag{2.77}$$

If $\boldsymbol{e}^{(i)}$, the error produced by the i-th classifier, is used as the target label for the next weak learner, then

$$\boldsymbol{e}^{(i+1)} = \boldsymbol{e}^{(i)} - \boldsymbol{y}^{(i+1)}(\boldsymbol{x}) \tag{2.78}$$

In Equation (2.78) above, the error of the $(i+1)$-th classifier $\boldsymbol{e}^{(i+1)}$ is the difference between the error learned by i-th weak learner and the prediction of the current classifier.

As GBM is a greedy method, it can overfit the training data set quickly. To avoid this, regularization methods, such as tree constraints and training with a random subset, should be used for the training of the regression tree.

The third type of ensemble, ensemble by stacking, is also known as committee machine [122]. It is similar in concept to bagging, in that multiple trainings are done in parallel before their outputs are combined. The main differences are (1) the data it combines are obtained from multiple types of sub-models (the meta learners) instead of just a single type, and (2) the outputs from the sub-models are subjected to a higher level classifier.

Stacking works best when the sub-models are skilful in classifying the input but in different ways, such as algorithms that use very different internal representations. If the sub-models are able to provide complementary views, then giving more weight to those views that are different will enhance the informational content of the blended data. The sum of M mixing coefficients, where M is the number of sub-models, must be 1, as shown in Equation (2.79) below.

$$\sum_{i=1}^{M} \alpha_i = 1 \tag{2.79}$$

The difficulty in determining the mixing coefficients α_i for the ensemble will be addressed by the proposed method in this thesis.

2.4.3 Multiple Kernel Learning

Multiple kernel learning (MKL) [123] is class of algorithms that combines a set of kernels as a unified kernel by either linear or non-linear combination. This is useful for multi-view learning, as the kernels that are established for the data sources can be used without having to create a common kernel for all of them (which can be difficult when there are different notions of similarity in the multi-view data). The bias of the kernels can be reduced by mixing the kernels optimally. A number of algorithms exists for doing so [124]. Figure 2.24 shows the topology for multiple kernel learning.

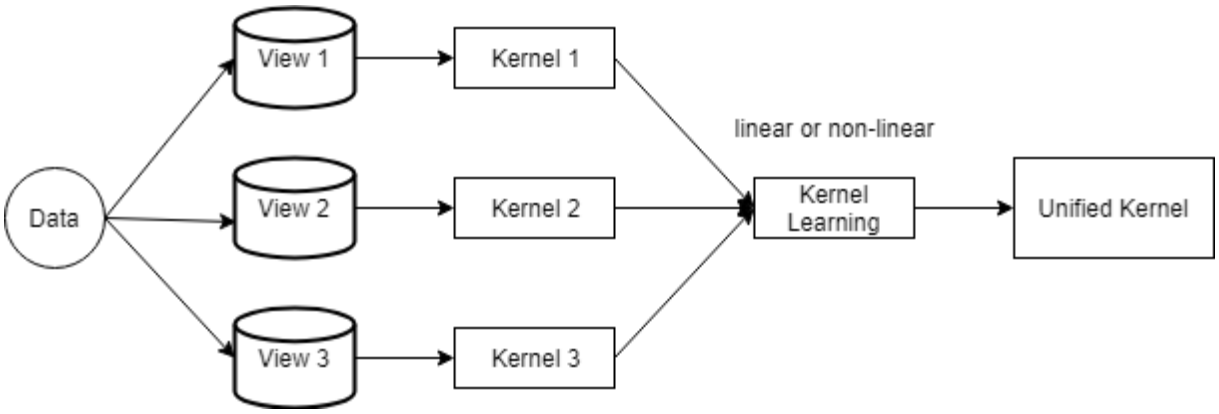

Figure 2.24. Multi-kernel training

Multiple kernel learning is possible because the kernels are additive (by the theory of reproducing kernels [125]). Thus, for a set of kernels $\boldsymbol{K} = \{\boldsymbol{K_1}, \dots, \boldsymbol{K_M}\}$, the unified kernel can be obtained by Equation (2.80) below if they are combined linearly.

$$K' = \sum_{i=1}^{M} \alpha_i K_i = 1 \tag{2.80}$$

The kernel weights $\boldsymbol{\alpha_i} \in \{\mathbf{1}, \ldots, \boldsymbol{M}\}$ are learned by minimizing the cost function and the regularizer shown in Equation (2.81) below, where $\boldsymbol{Y}$ is the training data.

$$\min_{\boldsymbol{\alpha}} cost(\boldsymbol{Y}, K') + reg(\boldsymbol{K}) \tag{2.81}$$

2.4.4 Spectral Embedding

In multi-view learning, the individual views can be used as they are, or embedded in a new low-dimensional feature space that preserves the pertinent information in the original view. The dimensional reduction will reduce the time and space required for machine learning, and also provide an alternative view of the data for processing, such as computing the complementarity of the views. This is the idea that will be exploited by the proposed multi-view temporal ensemble which will be described later in this thesis.

There are many types of dimensional reduction techniques. Those under the category of spectral dimensionality reduction techniques may be categorized as: 1) linear dimensionality reduction methods, for example, principle component analysis (PCA), and 2) manifold learning-based algorithms, for example, Laplacian eigenmap (LE).

Manifold learning-based algorithms are often termed as spectral embedding, where the term "embedding" simply means some sort of encoding. It is used for the dimensional reduction of data points that are clustered in a non-linear manifold. In this method, data points of the same cluster are mapped close to each other in the embedding.

In a non-linear manifold, the global distance between the data points is not a good indication of their cluster-ness. This is because the data are distributed in a "curvy" style in the data space (refer to Figure 2.25 below). Data points of different clusters, despite being associated with different categories, may not be far away from each other in terms of distance. Therefore, only the local data points that are very near to each other are significant in determining whether they are of the same cluster or not.

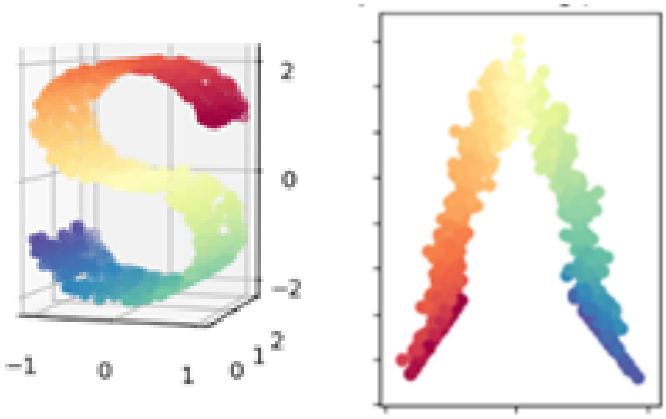

Figure 2.25. Reduction of a non-linear manifold preserves the local proximity of the data points. Reproduced from internet sources.

In Figure 2.25 above, the colours represent the target class labels. On the left hand side of the figure, the change of colours in the manifold is smooth. On the right hand side of the figure, the change of colours in the embedding is also smooth. This feat is achieved by preserving the local proximity of the data points.

Notation wise, the problem can be stated as: given a set of N data points $\boldsymbol{x} = \{\boldsymbol{x}_1 \dots \boldsymbol{x}_N\}$ in some manifold $\mathcal{M} \in \mathcal{R}^d$, find a set of N points $\boldsymbol{y} = \{\boldsymbol{y}_1 \dots \boldsymbol{y}_N\}$ in $\mathcal{R}^m$ $(m \ll d)$ such that $\boldsymbol{y}_i$ represents $\boldsymbol{x}_i$.

In the problem above, the requirement is that the embedded vectors $\boldsymbol{y}_i$ and $\boldsymbol{y}_j$ must preserve the local proximity of the data points $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$. The objective to satisfy the requirement is as follows: $distance(\boldsymbol{y}_i, \boldsymbol{y}_j) \approx distance(\boldsymbol{x}_i, \boldsymbol{x}_j)$ if and only if $distance(\boldsymbol{x}_i, \boldsymbol{x}_j)$ is small. This objective can be expressed as the minimization of the cost function in Equation (2.82) below:

$$J(\boldsymbol{y}) = \min_{i,j \in \{1,\dots,N\}} \sum_{i,j} \left(\|\boldsymbol{y}_i - \boldsymbol{y}_j\|^2 [\boldsymbol{W}]_{i,j} \right) \tag{2.82}$$

In Equation (2.82) above, i and j are the general indices of a pair of data instances within the set of N data points. The weight $[\boldsymbol{W}]_{i,j}$ is large only if the distance between $(\boldsymbol{x}_i, \boldsymbol{x}_j)$ is small. This property of "large weight value at small local distance" can be achieved with the use of a

local kernel such as the radial basis function. It uses pairwise measurements to provide information about the closeness of the data points.

In spectral embedding, the weight value is assigned to the radial basis function only if the node pair is connected, and 0 otherwise. This is shown in Equation (2.83) below.

$$[\boldsymbol{W}]_{i,j} = \begin{cases} e^{-\|\boldsymbol{x}_i - \boldsymbol{x}_j\|^2/\sigma^2} & if\ \boldsymbol{x}_i\ and\ \boldsymbol{x}_j\ are\ connected \\ 0 & otherwise \end{cases} \tag{2.83}$$

The radius of the local kernel is represented by the hyper-parameter $\gamma = \frac{1}{\sigma^2}$. When $\gamma = 0$, i.e. $\sigma = \infty$, the weights will be simplified as 1 (connected) or 0 (not connected), as shown in Equation (2.84) below.

$$\boldsymbol{W}_{i,j} = \begin{cases} 1 & if\ \boldsymbol{x}_i\ and\ \boldsymbol{x}_j\ are\ connected \\ 0 & otherwise \end{cases} \tag{2.84}$$

To illustrate the representation of a set of n data points with an undirected weighted graph and the weight matrix, a set of edges connecting six nodes together is shown in Figure 2.26 below.

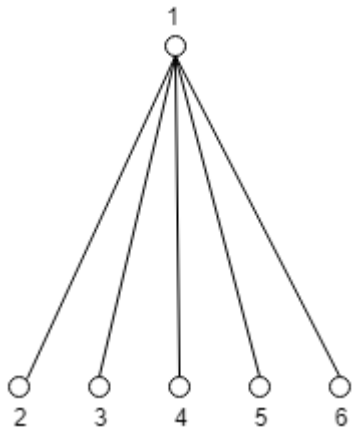

Figure 2.26: Simple graph, with node 1 connected to node 2, 3, 4, 5, and 6

In this case, using the simplified local kernel ($\sigma = \infty$), the weight matrix of the graph in Figure 2.26 above will become what is also called the adjacency matrix $\boldsymbol{A}$ as shown in Equation (2.85) below.

$$\boldsymbol{A} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{2.85}$$

2.4.5 Laplacian Eigenmap

In subsection 2.3.2 on Distance-Based Methods, it was mentioned that Mizuhara et al. [75] experimented with various embedded techniques and found that the Laplacian eigenmap-based embedded method achieved a better performance than the 1-NN classifier with DTW. This subsection will describe the detail of this spectral embedding technique, as it will be exploited by the proposed multi-view temporal ensemble which will be described later in this thesis.

In Laplacian eigenmap (LE) [14], a Laplacian matrix $\boldsymbol{L}$ is formed from the weight matrix $\boldsymbol{W}$. $\boldsymbol{L}$ is then subjected to eigen-decomposition. The collection of the eigenvectors $\boldsymbol{Y}$ forms the eigenmap of the data. It represents the data in the reduced space and is regarded as the spectral embedding of the data.

To illustrate the idea of the Laplacian $\boldsymbol{L}$, consider the graph in Figure 2.26 again. Let $\boldsymbol{D}$ be the diagonal weight matrix of $\boldsymbol{W}$, defined as $[\boldsymbol{D}]_{i,i} = \sum_{j=1}^{N} [\boldsymbol{W}]_{i,j}$ and let the Laplacian matrix $\boldsymbol{L}$ be $\boldsymbol{D} - \boldsymbol{W}$. In this case, the Laplacian matrix $\boldsymbol{L}$ will be as shown in Equation (2.86) below. Note that the diagonal entry $[\boldsymbol{L}]_{i,i}$ is given by the degree of connection of node i. The off-diagonal entries represent the edges such that $[\boldsymbol{L}]_{i,j} = [\boldsymbol{L}]_{j,i} = -1$ if there is an edge between nodes i and j, and $[\boldsymbol{L}]_{i,j} = 0$ otherwise.

$$\boldsymbol{L} = \begin{bmatrix} 5 & -1 & -1 & -1 & -1 & -1 \\ -1 & 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 & 1 & 0 \\ -1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.86}$$

The outline of the LE method is as follows:

1. Construct the adjacency matrix $\boldsymbol{A}$ for a set of n data points. Whether the node pair (i, j) is connected or not is based on any one of the following criteria:
   - ϵ-neighbourhoods: i and j are connected if $\left\|\boldsymbol{x}_i - \boldsymbol{x}_j\right\|_2^2 < \epsilon$
   - k-nearest neighbours: i and j are connected if $\boldsymbol{x}_j$ is among the k nearest neighbours of $\boldsymbol{x}_i$
   - combination of the above: : i and j are connected if $\boldsymbol{x}_j$ is among the k nearest neighbours of $\boldsymbol{x}_i$, and $\left\|\boldsymbol{x}_i - \boldsymbol{x}_j\right\|_2^2 < \epsilon$
2. Based on $\boldsymbol{A}$, set up the weight matrix $\boldsymbol{W} \in \mathcal{M}_{N\times N}(\mathbb{R})$ according to the local kernel function, where higher value implies closeness. There are two common kernels that can be used to generate the weights:
   - Radial basis kernel (with σ^2 as a parameter):

$$[\boldsymbol{W}]_{i,j} = \begin{cases} exp\left(-\frac{\left\|\boldsymbol{x}_i - \boldsymbol{x}_j\right\|_2^2}{\sigma^2}\right) & if\ \boldsymbol{x}_i\ and\ \boldsymbol{x}_j\ are\ connected \\ 0 & otherwise \end{cases} \tag{2.87}$$

   - Simple-minded kernel (radial basis kernel with $\sigma = \infty$):

$$[\boldsymbol{W}]_{i,j} = \begin{cases} 1 & if\ x_i\ and\ x_j\ are\ connected \\ 0 & otherwise \end{cases} \tag{2.88}$$

3. Let $\boldsymbol{D}$ be the diagonal weight matrix of $\boldsymbol{W}$, and form the Laplacian matrix $\boldsymbol{L} = \boldsymbol{D} - \boldsymbol{W}$.
4. Solve the generalized eigenvector problem $\boldsymbol{Ly} = \boldsymbol{\lambda Dy}$. This will yield the eigenvectors $\boldsymbol{y}_0, \boldsymbol{y}_1, \dots, \boldsymbol{y}_{n-1}$, and the eigenvalues $\lambda_0, \lambda_1, \dots, \lambda_{n-1}$.
5. Arrange the eigenvectors $\boldsymbol{y}_0, \boldsymbol{y}_1, \dots, \boldsymbol{y}_{n-1}$ in increasing eigenvalue order, as shown in Equation (2.89) below.

$$0 = \lambda_0 \leq \lambda_1 \leq \cdots \leq \lambda_{n-1} \tag{2.89}$$

6. Map $\boldsymbol{x}_i$ to $\boldsymbol{y}_i$ using the smallest m eigenvectors other than $\boldsymbol{y}_0$ (since $\boldsymbol{y}_0$ may represent noise) to embed $\boldsymbol{x}_i$ in the most compact m-dimensional Euclidean space.

The above procedure describes the solution of LE as the eigendecomposition of $\boldsymbol{L}$. It results in the cost function $\min_{i,j\in\{1,\dots,N\}} \sum_{i,j} \|\boldsymbol{y}_i - \boldsymbol{y}_j\|^2 [\boldsymbol{W}]_{i,j}$ as shown in Equation (2.82) previously to be minimized.

The cost value is the real value of the cost function. It will decrease in value if the cost function is being minimized. The lower the cost value, the better the preservation of the local proximity in the spectral embedding.

In LE, it can be shown that the cost value is equivalent to the value of $\boldsymbol{y^T L y}$. To derive that, it is easier to first assume a 1-dimensional embedding, i.e. $\boldsymbol{y}_i \in \mathcal{R}^1$ for $i \in \{1, \dots, N\}$, with $\boldsymbol{y}$ being an N-row column vector and i is the row index of the column vector. Noting that the weight matrix $\boldsymbol{W}$ is non-negative and symmetric (i.e. $[\boldsymbol{W}]_{i,j} = [\boldsymbol{W}]_{j,i}$), the cost function can be expanded and simplified as shown in Equation (2.90) below.

$$J(\boldsymbol{y}) = \sum_{i,j\in\{1,\dots,N\}} \|\boldsymbol{y}_i - \boldsymbol{y}_j\|^2 [\boldsymbol{W}]_{i,j} = \sum_{i,j\in\{1,\dots,N\}} \left(\boldsymbol{y}_i^2 + \boldsymbol{y}_j^2 - 2\boldsymbol{y}_i\boldsymbol{y}_j\right)[\boldsymbol{W}]_{i,j} \tag{2.90}$$

$$= 2 \sum_{i\in\{1,\dots,N\}} \boldsymbol{y}_i^2 \sum_{j\in\{1,\dots,n\}} [\boldsymbol{W}]_{i,j} - 2 \sum_{i,j\in\{1,\dots,N\}} \boldsymbol{y}_i\boldsymbol{y}_j [\boldsymbol{W}]_{i,j}$$

$$= 2 \sum_{i\in\{1,\dots,N\}} \boldsymbol{y}_i^2 [\boldsymbol{D}]_{i,i} - 2 \sum_{i,j\in\{1,\dots,N\}} \boldsymbol{y}_i\boldsymbol{y}_j [\boldsymbol{W}]_{i,j}$$

$$= 2\boldsymbol{y}^T\boldsymbol{D}\boldsymbol{y} - 2\boldsymbol{y}^T\boldsymbol{W}\boldsymbol{y}$$

$$= 2\boldsymbol{y}^T\boldsymbol{L}\boldsymbol{y}$$

Therefore, the minimization problem is reduced to

$$\boldsymbol{y}^* = arg \min_{\boldsymbol{y}^T\boldsymbol{D}\boldsymbol{y}=1, \boldsymbol{y}^T\boldsymbol{D}1=0} \boldsymbol{y}^T\boldsymbol{L}\boldsymbol{y} \tag{2.91}$$

The additional condition $\boldsymbol{y}^T\boldsymbol{D}\boldsymbol{y} = 1$ is for normalization as it removes an arbitrary scaling factor. The condition $\boldsymbol{y}^T\boldsymbol{D}\boldsymbol{1} = 0$ is to avoid trivial solution.

Importantly, finding $\boldsymbol{y}^* = arg \min_{\boldsymbol{y}^T\boldsymbol{D}\boldsymbol{y}=1, \boldsymbol{y}^T\boldsymbol{D}\boldsymbol{1}=0} \boldsymbol{y}^T\boldsymbol{L}\boldsymbol{y}$ is equivalent to finding the eigenvector corresponding to the smallest eigenvalue for the generalized eigenvalue problem $\boldsymbol{L}\boldsymbol{y} = \boldsymbol{\lambda}\boldsymbol{D}\boldsymbol{y}$.

Now, instead of a 1-dimensional embedding as used above, consider an m-dimensional embedding, i.e. $\boldsymbol{y}_i = ([\boldsymbol{y}_i]_1, \dots, [\boldsymbol{y}_i]_m) \in \mathcal{R}^m$. The size of the matrix $\boldsymbol{Y}$, which is the collection of all $\boldsymbol{y}_i$'s, will be $N \times m$, where N is the number of data points and m is the reduced dimension of the encoding space. In this case, the cost function is the sum across the m dimensions, as shown in Equation (2.92) below.

$$J(\boldsymbol{Y}) = \sum_{i,j\in\{1,\dots,N\}} \left\|\boldsymbol{y}_i - \boldsymbol{y}_j\right\|^2 [\boldsymbol{W}]_{i,j} = \sum_{k\in\{1,\dots,m\}} \sum_{i,j\in\{1,\dots,N\}} \left([\boldsymbol{y}_i]_k - [\boldsymbol{y}_j]_k\right)^2 [\boldsymbol{W}]_{i,j} \tag{2.92}$$

$$= \sum_{k\in\{1,\dots,m\}} [\boldsymbol{Y}]_{\cdot k}{}^T \boldsymbol{L}\, [\boldsymbol{Y}]_{\cdot k}$$

$$= tr(\boldsymbol{Y}^T\boldsymbol{L}\boldsymbol{Y})$$

Therefore, the minimization problem reduces to

$$\boldsymbol{Y}^* = arg \min_{\boldsymbol{Y}^T\boldsymbol{Y}=\boldsymbol{1}} tr(\boldsymbol{Y}^T\boldsymbol{L}\boldsymbol{Y}) \tag{2.93}$$

The solution $\boldsymbol{Y}^*$ of the above minimization problem is the set of eigenvectors that correspond to the smallest m eigenvalues of the generalized eigenvalue problem $\boldsymbol{L}\boldsymbol{y} = \boldsymbol{\lambda}\boldsymbol{D}\boldsymbol{y}$.

This concludes the derivation of $tr(\boldsymbol{Y}^T\boldsymbol{L}\boldsymbol{Y})$ as the cost value of m-dimensional spectral embedding. The significance of $tr(\boldsymbol{Y}^T\boldsymbol{L}\boldsymbol{Y})$ is that it represents how much the local proximity is preserved in the spectral embedding $\boldsymbol{Y}^*$, or conversely, how much "spreading" there is in the spectral embedding. This shows that not only can a view be encoded, how well it can be done can be quantified too.

This concludes the review in this thesis on biosignals, deep learning, time series classification, and multi-view learning. It has covered the topics necessary to understand the methods proposed in this thesis. The next two chapters will describe the proposed methods, namely the deep temporal convolution network and the multi-view temporal ensemble.

Chapter 3. Deep Temporal Convolution Network

This chapter describes the proposed deep temporal convolution network in detail.

A network that matches with a complex data function is likely to boost the classification performance, as it is able to learn the useful aspect of the highly varying data. In this work, the temporal context of the time series data is chosen as the useful aspect of the data that should be passed through the network for learning.

The proposed deep learning network, called the deep temporal convolution network, exploits the compositional locality of the time series data at each level of the network. The aim is to match the network with the complex data function of a highly varying time series, so that shift-invariant features can be extracted layer by layer at different time scales, thus boosting the classification performance.

The proposed network makes use of data processing and the concatenation operation to introduce the temporal context to the deeper layers. A matching learning algorithm for the revised network, based on the idea of gradient routing in the backpropagation path, learns the weight values from the training set.

The temporal context passes through the deeper layers of the network with temporally constant abstraction at the higher levels. The non-stationary characteristics of the input will thus have less effect on the output. As a result, the performance of the trained model of the proposed deep temporal convolution network will be higher than a time delay neural network of the same complexity (in terms of the number of adjustable parameters).

The framework proposed in this work comprises the following components: (1) the structure of the network, (2) the necessary data preparation of the input data, and (3) the learning algorithm. The proposed network is able to handle the prediction of multivariate signals with high temporal resolution and is suitable for highly varying signals that require deep learning. Data experiment with electroencephalogram signals shows that the proposed method can improve the classification performance. It is able to attain better generalization without overfitting the network to the data, as the weights can be pre-trained appropriately. It can be used end-to-end

with multivariate time series data in their raw form, without manual feature crafting or data transformation.

The remainder of this chapter is organised in order of the framework components. The first section shows the architecture of the proposed network and builds up the case for the proposed architecture. The second section explains how to represent and distribute the temporal context in multiple layers. The third section describes the proposed methodology on concatenating the temporal context, preparing the data, and learning by backpropagation with gradient routing. The last section concludes the chapter by providing a quick summary of the results of a data experiment done on a multi-channel electroencephalogram data set.

3.1 Network with Temporal Context

The network structure of the deep temporal convolution network is a hybrid of the DBN-DNN architecture and the additional concatenation sublayers, as shown in Figure 3.1 below. The figure shows that the concatenation operation is repeated in the hidden layers. Starting from the bottom of the figure, the time series data are first arranged in the time delay representation in mini-batches, each consisting of a small number of data instances, for example 8 or 32. The output of the hidden layer is re-arranged by the concatenation operation, resulting in a new input for the next hidden layer. The network weights are located between the new input and the next hidden layer. These weights are trained by pre-training [126] in the forward path, and then by backpropagation [127] with gradient routing in the backward path.

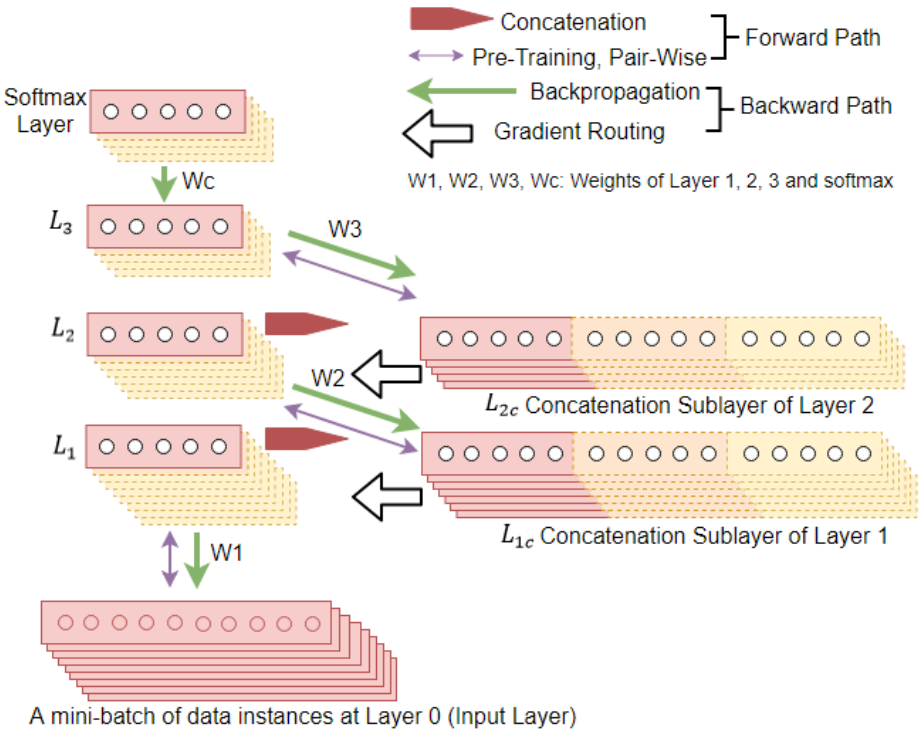

Figure 3.1. Architecture of the proposed deep temporal convolution network

Overall, the structure comprises a feature detector at the lower layers and a classifier at the upper layers. The arrowheads in Figure 3.1 above indicate the directions of the forward and backward paths in learning. As can be seen from the diagram, the forward propagation passes through the concatenation sublayers of the first two hidden layers before reaching the softmax layer (the final classifier) at the top of the stack.

A concatenation sublayer is governed by the hyper-parameter, TS. It is the number of time steps used for the concatenation sublayer. Figure 3.2 below shows the use of $TS = 3$ for the concatenation sublayer L_{1c}. Note that the time steps on the left hand side of Figure 3.2 must be in their natural time order.

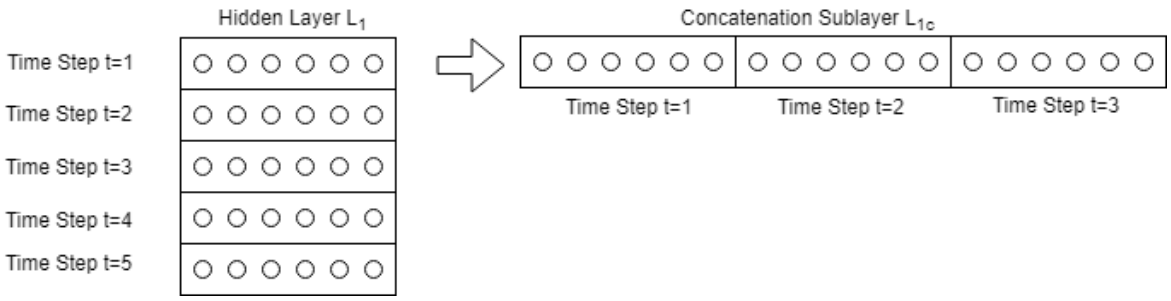

Figure 3.2. Time Steps and Concatenation

The above structure fits into a deep belief net – deep neural network (DBN-DNN) architecture as described in Section 2.2.7 earlier on. This allows it to make use of the auto pre-learning mechanism of DBN-DNN to initialize the network weights.

To pre-train the weights, each pair of layers (nominally called the visible and hidden layer) in the deep temporal convolution network will come together to form a restricted Boltzmann machine (RBM). This will result in a stack of RBMs. In the RBM stack, the hidden layer of the prior RBM will be concatenated in the manner as shown in Figure 3.2 above. The concatenation sublayer will then be used as the visible layer of the next RBM.

For the example shown in Figure 3.1 earlier on, the RBM stack consists of the following three RBMs: (L_{input}, L_1), (L_{1c}, L_2), and (L_{2c}, L_3). Notice that even though the hidden layer for the first RBM is layer L_1, the visible layer for the second RBM is layer L_{1c} and not layer L_1.

The pre-training process, being greedy layer-wise, will leave the weights of the first RBM as they are once they are trained and then proceed to the next RBM in the stack. This forward process will continue until all the weights in the hidden layers are initialized. Thereafter, the softmax layer will be trained by supervised learning. Finally, the weights of the entire deep temporal convolution network will be fine-tuned by the proposed learning method that will be described later.

The combination of weight initialization, classifier's training and fine-tuning will ensure that the proposed network structure will work well even if there are many hidden layers with their own concatenation sublayers.

The proposed network addresses the following problems: (1) representation of temporal context, (2) distribution of temporal context, and (3) learning with many layers. These are explained in the next few sub-sections.

3.1.1 Representation of Temporal Context

For a signal to be classified by a neural network, it will have to be represented in what is called the time delay representation [128]. This can be done easily for discrete time series with N time series elements, i.e. sample points, at constant sampling rates, $\boldsymbol{x} = (\boldsymbol{x}_1, \dots, \boldsymbol{x}_N)$. Simply slide a window of fixed length w across the signal with stride s, $s < w$. The result is a set of overlapping segments. Each segment is a data vector containing w samples.

The data vector, used at the input of the neural network, can be viewed as a tapped delay line used for convolution, as shown in Figure 3.3 below. A neural network that treats its input in this way is called the time-delay neural network (TDNN).

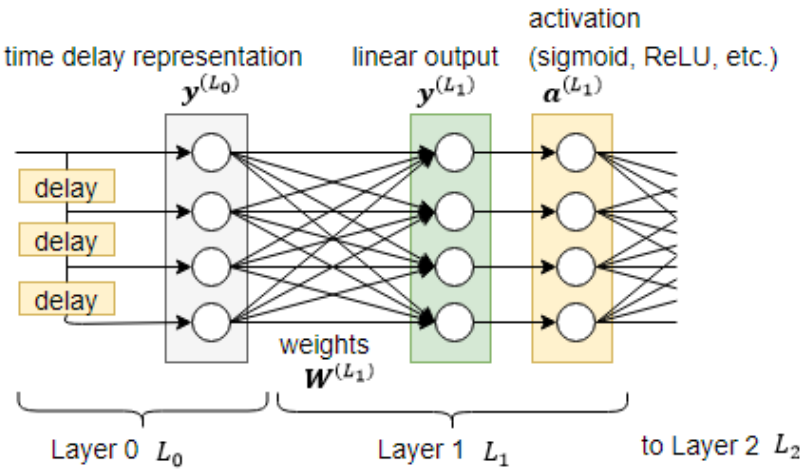

Figure 3.3. A tapped delay line at the input of a TDNN, $\boldsymbol{w} = \boldsymbol{4}, \boldsymbol{s} = \boldsymbol{1}$.

The sample points in the data vector are the lag observations of the signal. They contain the time-dependent patterns that the algorithm can learn.

The amount of overlap between any two neighboring segments is shown in Equation (3.1) below.

$$overlap\ \% = \frac{w - s}{\mathrm{w}} \times 100\% \tag{3.1}$$

The overlapping of the segments is important. It ensures that the non-stationary features are represented at different time points. This makes the training of a shift-invariant model possible, so that there is no need to provide the exact starting and ending points of the temporal features.

The sliding window method, used at the input to create the time delay representation, is sufficient for good performance in time series classification. The problem with this approach is the loss of temporal context in the hidden layers, and so the features learned are no longer time-invariant in the hidden layers.

3.1.2 Distribution of Temporal Context

In Figure 3.3 shown earlier on, the data vector at the input of the neural network was a tapped delay line. Likewise, to distribute the temporal context to the hidden layers, the data at the hidden layers can be stored in tapped delay lines too. This is shown in Figure 3.4 below.

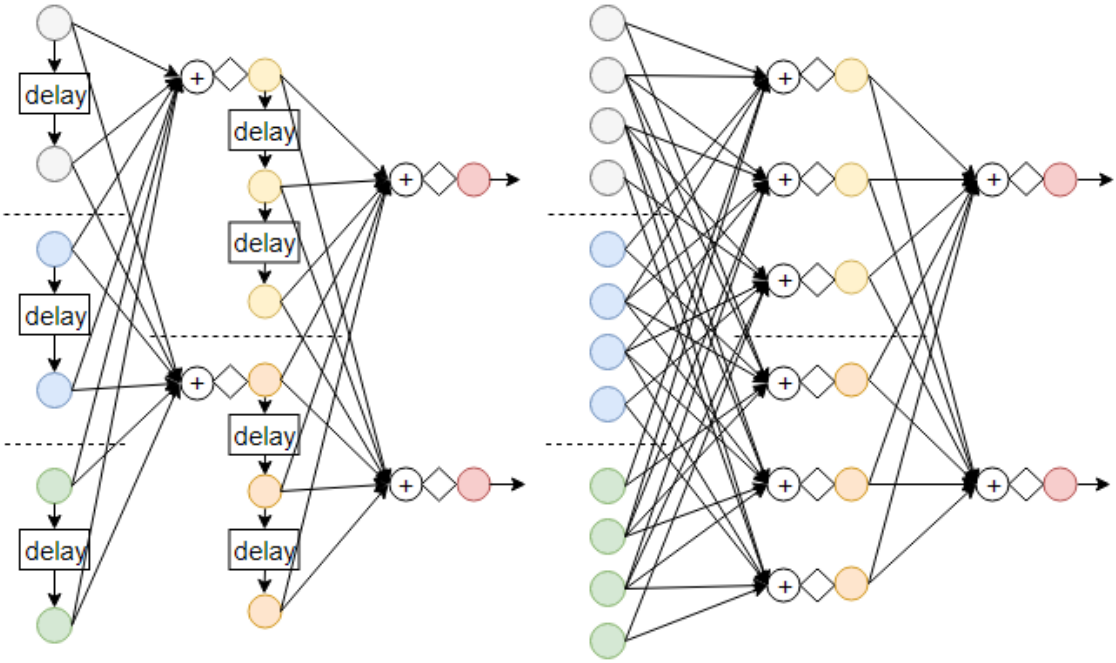

Figure 3.4. A distributed TDNN (left) and its equivalent network (right)

The left hand side of Figure 3.4 above shows the distributed TDNN with an input layer of 3 units (each unit with a 2-tap delay line), a hidden layer with 2 units (each unit with a 3-tap delay line), and a final output layer with 2 units. The right hand side of Figure 3.4 shows the equivalent network, which is a plain static neural network with no tapped delay line.

The number of unique weights for the distributed TDNN on the left hand side of Figure 3.4 and the equivalent network on the right hand side of Figure 3.4 are the same, as can be seen by comparing the two sides carefully. This is despite the equivalent network having more nodes than the distributed TDNN. The reason for this is that the nodes in the equivalent network are not fully connected. For nodes connected, many share their weights by downward shifting along the tapped delay line.

The distributed TDNN is not amendable to pre-training because the tap-delay line architecture does not fit into the stack form required for pre-training. It will suffer from the computational issue of exploding and/or diminishing gradient when the number of hidden layers is increased [36].

The distribution of temporal context and weight sharing manifest as concatenation in the deeper layers of the proposed network. Unlike the distributed TDNN, the proposed network can be pre-trained, thus alleviating the computational issue of exploding and/or diminishing gradient.

3.1.3 Learning with Many Layers

To overcome the computation problem, it is necessary to initialize the network weights to some "good" values [129]. This is possible with pre-training in the Deep Belief Net - Deep Neural Network (DBN-DNN) [126].

The DBN-DNN is a static network that comprises two parts: a stack of restricted Boltzmann machines (RBMs) [130], collectively known as the DBN, and a final output classier (for example, a softmax layer) on top of it.

As shown in Figure 2.16 earlier on, the training process of the DBN-DNN is divided into two stages, comprising the pre-training stage and the fine-tuning stage. The pre-training stage

applies only to the DBN. It does not involve the softmax layer or the target labels. It is thus an unsupervised training process. This is in contrast with the fine-tuning stage, which is a supervised training process.

The pre-training is pair-wise and operates in the forward direction [131]. It starts at the bottom of the DBN, where a pair of layers, nominally the visible layer and the hidden layer, forms the RBM. The RBM runs unsupervised training by contrastive divergence [132]. Upon convergence, the weights between the two layers will become fixed, and the same process (unsupervised training by contrastive divergence) will be brought forward to the next pair of layers. In moving forward, the output (hidden layer) of the previous RBM will become the input (visible layer) of the current RBM.

After pre-training, the weights in the DBN are transferred to the DBN-DNN, where together with the weights of the softmax layer, they are fine-tuned by backpropagation.

A DBN-DNN trained in this manner (pre-training in the forward path, followed by fine-tuning in the backward path) will make the network relatively immune to overfitting.

The limitation of the DBN-DNN is that the temporal context is not distributed to the deeper layers of the network. To do so, we propose using data processing based on the concatenation operation in the DBN-DNN, and we provide the matching learning algorithm for the revised network.

3.2 Data Preparation

As mentioned in the introductory chapter, a time series contains temporal patterns (i.e. time-dependent features). An intuitive way of understanding the idea of temporal patterns is to use the analogy of reading a text, where the word before and after the current word provide the temporal context of the current word, serving the purpose of making the meaning of the current word clear. This is illustrated in Figure 3.5 below.

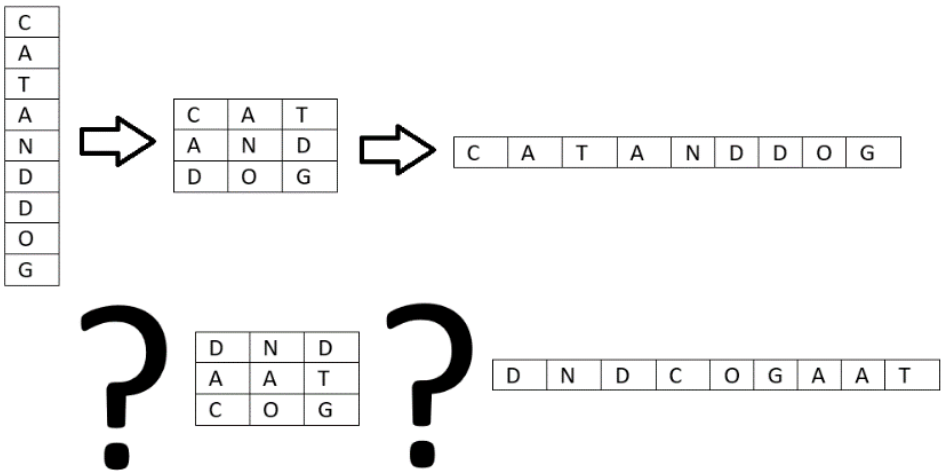

Figure 3.5. Time-dependent feature

A neural network will not have any notion of temporal patterns if the time series data are used singly, i.e. one time step at a time. This is because the neural network will treat each of the data instances as independent. Some rearrangement or regrouping is necessary before the neural network can extract the temporal patterns from the data.

In this work, the temporal context of the time series data is chosen as the useful aspect of the data that is passed through the network [133]. The temporal context consist of neighbours that are next to each other in time. To pass in the temporal context, the proposed deep temporal convolution network will make use of the common data representation known as the time delay representation. Several modifications to this representation will be described later. These modifications are necessary for the data to be suitable for use with the proposed deep temporal convolution network.

3.2.1 Concatenation of Temporal Context

In the time delay representation, multiple data instances in consecutive sampling time are placed together, i.e. concatenated, as a single data instance. As an example, the time delay representation of a multivariate data set with four attributes is shown in Figure 3.6 below.

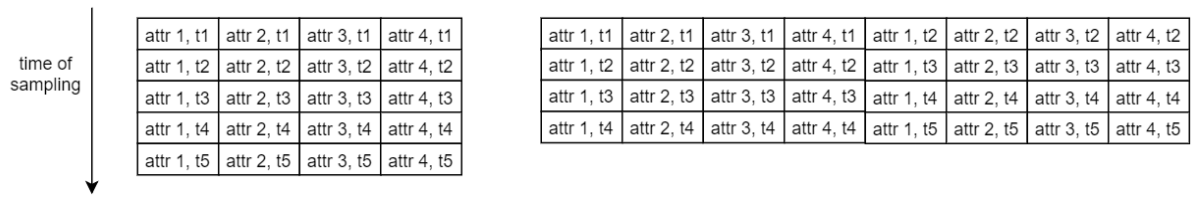

Figure 3.6. Time delay representation of a multivariate data set

On the left hand side of Figure 3.6 above, each row vector is a sample in time. The first row is the sample at the time step $t1$, the second row is the sample at the time step $t2$, and so on. Each sample has four attribute values, $attr$ 1, 2, 3, and 4.

To rearrange the data, a fixed-size sliding window is moved vertically downward. After each slide, the content of the window is vectorised and placed in a new row in the table on the right hand side of Figure 3.6 above. The window used for the operation in this example has a width (data dimension) of 4, a height (window length) of 2, and a stride (the amount of movement per window slide) of 1.

Notice that the content of consecutive rows on the right hand side of Figure 3.6 above will overlap with each other. The idea of overlap is critical for the deep temporal convolution network. It ensures that the non-stationary features are represented by at least two different versions. These versions are located at different time positions, making the training of a shift-invariant model possible. With a shift-invariant model, there is no need to provide the exact beginning and end points of the temporal features. As long as the window length is long enough to cover the temporal features, the model will be able to learn them.

A few other observations can be made about the table content in Figure 3.6 above:

(1) On the left hand side, the rows are arranged in natural time order, i.e. not randomized.

(2) On the right hand side, the attributes of the sample are arranged horizontally with their time delay versions.

(3) After concatenation, the number of rows N will decrease, as shown in Equation (3.2) below.

$$N = \frac{length\ of\ time\ series - window\ size}{slide} + 1 \tag{3.2}$$

A necessary step in neural network before training is to shuffle the data vectors on the right hand side of Figure 3.6 above. Without shuffling, the target class labels in the data set will have a simple output pattern, for example, 1,1,1,1,1,0,0,0,0,0,…. This pattern is incidental in the training data. It is unlikely to recur in the test data. If the neural network overfits to the output pattern in the training data set, the test result will be very bad despite the good training result. Shuffling the data will break up the output pattern. This will make the data vectors independent and identically distributed. This is what the machine learning algorithm expects of the training data for good test result.

The structured format as described above is called the time-delay representation because each row in the data table is similar to the tapped delay line of a finite impulse filter. In a finite impulse filter, the multiplication of the row vector with the weights is the convolution. When the row vector is time series data, this will be the temporal convolution.

Figure 3.7 below shows an example of the concatenation operation used in the proposed deep temporal convolution network. It shows 5 data instances in the layer L_i at time t_1, t_2, t_3, t_4, and t_5. They are combined to become the new data instances in the layer L_{ic}, which is the concatenation sublayer of the input in L_i.

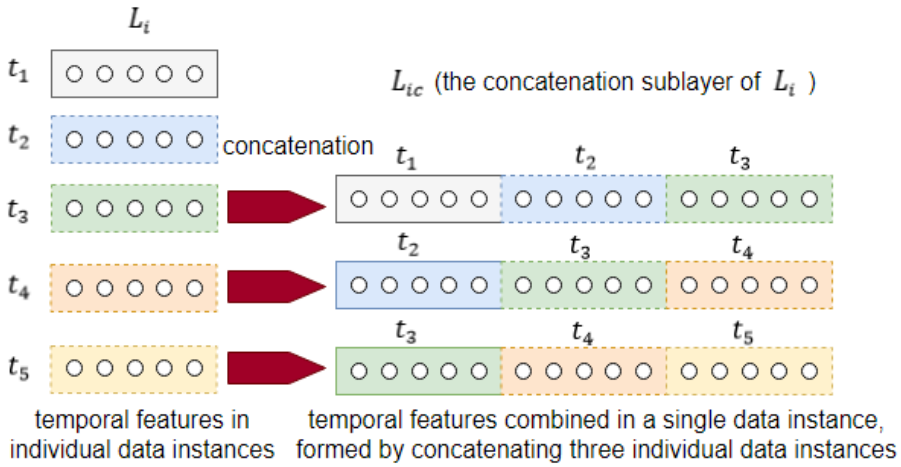

Figure 3.7. Formation of concatenation sublayer, $TS = 3$.

The combination of the data instances is according to their natural time order. For example, in Figure 3.7 above, the data instances at t_1, t_2 and t_3 form a new data instance, while the data instances at t_2, t_3 and t_4 form another new data instance. As such, the new data instances in L_{ic} formed by the concatenation operation will have more temporal context than the individual data instances in L_i. They will act as the new input for the next hidden layer at $i + 1$.

In this work, the amount of concatenation will be described by a variable known as the time steps, TS. It is a hyper-parameter of the proposed network. In the example in Figure 3.7 above, the value of TS is 3. This is because each concatenation consists of 3 data instances.

The 5 individual data instances in the above example form what is known as a mini-batch. It is a term used to differentiate from the term "batch" as used in "batch gradient descend" where it refers to the entire data set. All operations in the proposed network, including data preparation and network learning, will be done in mini-batches rather than individual data instances.

3.2.2 Concatenation in the Deeper Layers

The proposed extension of the sliding window method to the deeper layers will extend the benefits of shift-invariance and temporal context learning to the deeper layers. However, this is not possible without making some modifications to the time delay representation.

The following reformatting will have to be done to the data in the proposed network so that the shift-invariant temporal context can be learnt:

(1) Maintain short-term temporal order within a mini-batch
(2) Create mini-batches that overlap with the neighbouring mini-batches
(3) Randomize the order of the mini-batches
(4) Pool the count of the target labels through the deeper layers

The first two steps are used to prepare the time series data for use as the input of the network. The third step is used to associate the training data with target labels that are valid.

Some of the considerations in making the modifications are:

(1) The temporal order of the data vectors must be maintained for the concatenation to be meaningful, i.e. contain the temporal patterns.

(2) Duplicated features at different time positions must be available in the training data set for the model to be time-invariant.

(3) The input data vectors must be independent and identifically distributed.

(4) Concatenation through the deeper layers will affect not just the data vectors but the target class label also.

(5) Side effects may arise when handling the above issues, such as the progressive reduction in size of the mini-batches and the unequal contribution of the input vectors to the training.

3.2.3 Short-Term Temporal Order

The maintenance of temporal order in the data set is a necessary requirement for the deep temporal convolution network. However, this cannot be met by a shuffled data set, as the randomized data instances in a shuffled data set are no longer time-dependent on each other. The concatenation of such data instances will increase the irreducible error in the trained model. This will lead to performance that is very poor for both the training and the test set. Thus, without a method to resolve the dilemma of (1) the need to shuffle, and (2) the need to maintain temporal order, the performance will degrade instead of being boosted.

To overcome the abovementioned dilemma, it is proposed that the short-term temporal order be maintained within a mini-batch, with shuffling done for the mini-batches but not the data within the mini-batches. In other words, the data instances in the mini-batches will keep their natural time order, but the mini-batches will be shuffled to shatter the long-term time order.

Maintaining short-term temporal order clears up the following dilemma faced by the proposed network.

On one hand, the concatenation of the data instances is only meaningful if the data instances are in their natural time order, otherwise randomness will be injected into the concatenated data and worsen the network performance.

On the other hand, each of the data instances must be a sample that is independent and identically distributed, otherwise the simple output pattern in the time series data set will be learnt by the network. As this pattern is incidental to the training data and unlikely to recur in the test data, overfitting the network to it will produce poor test result in spite of the good training result.

The use of short-term temporal order solves the aforementioned dilemma. In addition, it fits into the practice of using mini-batches for computational efficiency. As the size of a mini-batch is typically a small number from 2 to 32, it provides improved generalization performance with a small memory footprint [134].

The proposed use of mini-batches to maintain short-term temporal order fits into the common practice of using mini-batches over several training epochs in deep learning [135]. This practice arises from the following benefits of doing so: (1) higher throughput due to computational parallelism, (2) faster convergence as the noise is reduced, (3) a higher usable learning rate due to better quality gradient.

The mini-batch size is a user-defined hyper-parameter. It is usually quite small (between 2 and 32). A small size provides improved generalization performance and allows a smaller memory footprint. Thus, only short-term temporal order (instead of a longer temporal order) is proposed for use in the deep temporal convolution network.

In deciding on the mini-batch size, consideration must also be given to the progressive decrement in size when the concatenation is extended to the deeper layers. For example, for a mini-batch of 8 data instances, only 6 data instances will be available after concatenation with $TS = 3$. When this is extended to the third layer, the mini-batch will be left with just 4 data instances. Figure 3.8 below illustrates this effect.

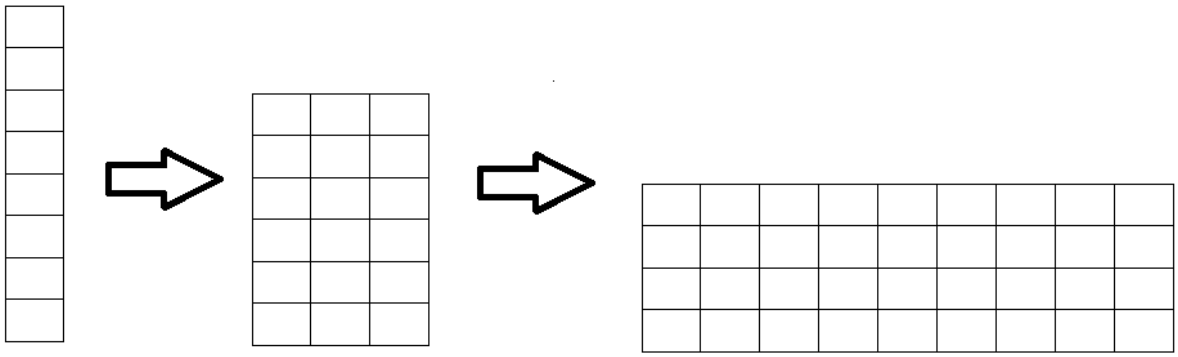

Figure 3.8. Size reduction in a mini-batch

The mini-batch size can be determined by the formula as shown in Equation (3.3) below:

$$B^{(l-1)} = B^{(l)} + \left(TS^{(l-1)c} - 1\right) \tag{3.3}$$

Here, $B^{(l-1)}$ is the mini-batch size at layer L_{l-1}, $B^{(l)}$ is the mini-batch size at layer L_l, and $TS^{(l-1)c}$ is the hyper-parameter for the time step of the concatenation sublayer $L_{(l-1)c}$.

In order not to have too few instances in the mini-batch at the final output layer, it would be expedient to increase the size of the mini-batch at the input layer l_0. This can be done according to Equation (3.4) below.

$$B^{(L_0)} = B^{(L_L)} + L \times (TS - 1) \tag{3.4}$$

In Equation (3.4) above, $B^{(L_0)}$ is the mini-batch size at layer L_0, and $B^{(L_L)}$ is the mini-batch size at layer L_L. Here, it is assumed that there are L hidden layers and that the same TS (time steps) value is used for all the concatenation sublayers in the hidden layers.

Once there is short-term temporal order in the mini-batches at the input layer, the temporal order will be maintained in the deeper layers. This is because the concatenation operation at the deeper layers will not affect the temporal order. Thus, with short-term temporal order in the mini-batches, the temporal context in the mini-batches can be passed to the deeper layers, and at the same time ensure computational efficiency in training.

3.2.4 Mini-Batches that Overlap

The mini-batches should overlap with each other so that the network can be shift-invariant and less dependent on the precise location of the temporal feature within the mini-batch. This is a necessary step. It is a step in addition to the overlap of the samples in the time delay representation. This is because there is temporal context within a mini-batch, and the temporal context maintained by short-term temporal order in the mini-batch has to be learnt in a shift-invariant manner.

It is proposed that instead of simply dividing the concatenated data set into blocks of mini-batches, the sliding window method should be used to make the mini-batches. Figure 3.9 shows the proposed two-stage sliding window method to create mini-batches that overlap with their neighbours. First, slide a fixed-size window along the time series to create the time delay representation. On the time delay representation thus created, slide another fixed-size window to create the mini-batches that overlap with each other.

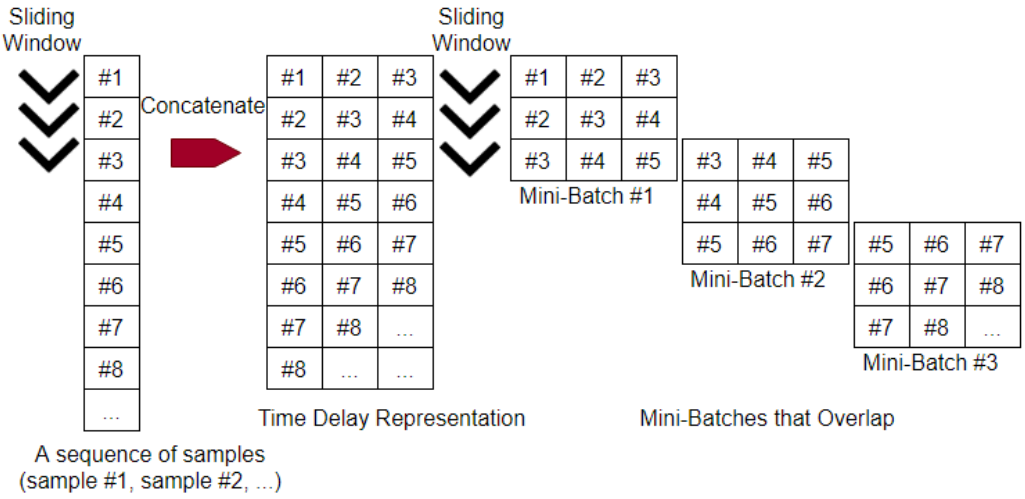

Figure 3.9. A two-stage sliding window to create mini-batches that overlap.

The aforementioned make process will result in mini-batches that overlap with their precedent and consequent mini-batches. The first and last few data instances of a mini-batch re-appear in the neighbouring mini-batches. Overlapping mini-batches will enhance the shift-invariant property of the trained model. During training, it will coax the model to be time-invariant to the short-term temporal patterns in the mini-batches.

Within each of the mini-batches, there is an unequal contribution of the data instances. For example, in the first mini-batch in Figure 3.9 above, the data instance #1 appears once, whereas the data instance #3 appears three times.

There is unequal contribution of data at the concatenation sublayers also. Figure 3.10 below shows the concatenation with $TS = 3$ at the hidden layer. The data vector $\#5$ contributes 3 times in mini-batch #1, while the data vector #3 and #4 contribute 5 times.

#1, #2, #3	#2, #3, #4	#3, #4, #5
#2, #3, #4	#3, #4, #5	#4, #5, #6

mini-batch #1 at the deeper layer

#3, #4, #5	#4, #5, #6	#5, #6, #7
#4, #5, #6	#5, #6, #7	#6, #7, #8

mini-batch #2 at the deeper layer

#5, #6, #7	#6, #7, #8	#7, #8, #9
#6, #7, #8	#7, #8, #9	#8, #9, #10

mini-batch #3 at the deeper layer

Figure 3.10. Unequal contribution at the deeper layer

The unequal contribution of data within the mini-batches will be largely eradicated when all the overlapping mini-batches are considered as a whole. It will not affect the effective training of the network, as it is well known that a model can be trained by a random sampling of the data set rather than the whole data set. Nevertheless, to minimise the unequal contribution, the amount of overlap of the mini-batches $O_{mini-batch}$ can be set as shown in Equation (3.5) below, where TS is the amount of overlap in the time delay representation.

$$O_{mini-batch} = TS - 1 \tag{3.5}$$

3.2.5 Randomization of the Mini-Batches

The mini-batches have to be shuffled in order before being used as input for the proposed network. The independent and identically distributed order of the mini-batches ensures that the deep temporal convolution network does not learn the simple output pattern. If the order of the mini-batches is not shuffled, the deep temporal convolution network will fail miserably with

new and unseen data, as they are unlikely to have the same output pattern as the training data set.

To enhance computation efficiency, a combo-batch consisting of more than 1 mini-batch may be used during fine-tuning of the deep temporal convolution network. A combo-batch is formed from a number of mini-batches that have been randomized in order. The catch here is that in a combo-batch, there is no temporal continuity between the mini-batches. However, this is not a practical issue. Not only is it not detrimental to the training, it is in fact desirable, as the artificial arrangements constitute a form of regularization that will improve the generalization of the trained model. Empirically, the rule of thumb is that about 25% random arrangement of the total data is acceptable.

3.2.6 Pooling of Target Labels through Deeper Layers

As there are many samples in a single data vector, the target label in common for the samples has to be decided by majority voting. Although this can be done at the input layer of the proposed network, it is more expedient to delay it until the final classifier. This is because the concatenation operation adds more data to the deeper layers, and so it is wise for the majority voting to take into consideration the additional count of the target labels due to the concatenation operation. An overly simplistic scheme that ignores the effect of concatentaion will likely distort the actual class distribution at the final output layer.

For example, if there are three data vectors in a concatenation, two of them class 1 and one of them class 2, then class 1, being the majority class, will be deemed by a simplistic scheme as the target label. The distortion occurs because the target labels of the data vectors are themselves the result of majority voting and have lost some of the information due to the summarization.

Figure 3.11 below shows a one-to-one annotated time series. For every observation, there is an output class, which is either 1 or 0.

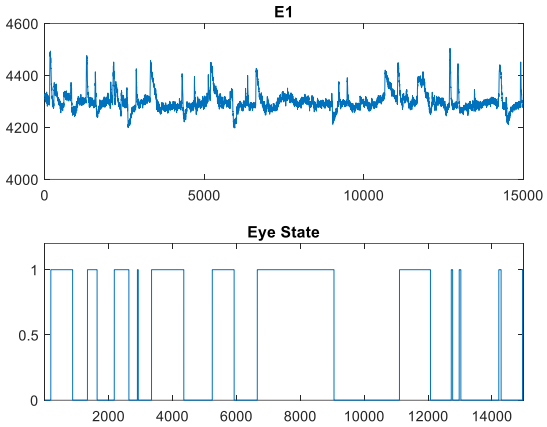

Figure 3.11. A time series (top) and its target class labels of 1 and 0 (bottom)

Consider the target class label $\boldsymbol{y}(t) \in \{0,1\}$ of the time series shown in Figure 3.11 above. In table form, $\boldsymbol{y}(t)$ will look like Figure 3.12 below, where each row with 16 target class labels corresponds to the 16 samples of a data vector in the time delay representation.

0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1
0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1
0	1	0	1	1	1	0	0	0	0	1	1	0	0	0	1
0	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	0	1	0	1	0	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Figure 3.12. Target label class in time delay representation

Majority voting can be done row-wise on Figure 3.12 above with the elaborate use of one-hot encoding format. After one-hot encoding, there will be one category per class, for example, one category for Class 0 and another category for Class 1. Figure 3.12 above will then appear as Figure 3.13 below.

Class 0	Class 1
1	0
1	0
1	0
1	0
1	0
0	1

Class 0	Class 1
1	0
1	0
1	0
0	1
0	1
0	1

…

Class 0	Class 1
1	0
0	1
0	1
1	0
0	1
1	0

Class 0	Class 1
1	0
0	1
0	1
0	1
0	1
1	0

Figure 3.13. Target class labels in one-hot encoding format

The class-wise summation of the 16 one-hot encodings, or pooling of the target labels, is as shown in Figure 3.14 below.

Class 0	Class 1
14	2
9	7
3	13
9	7
2	14
4	12

Figure 3.14. Result of class-wise summation

Majority voting is used to decide on the target class label based on the pooled target labels. The end result is shown in Figure 3.15 below.

Class 0	Class 1
1	0
1	0
0	1
1	0
0	1
0	1

Figure 3.15. Majority voting, input layer

The above simplistic scheme presents two problems when it is applied to the deep temporal convolution network: (1) it is based on the input layer and has not considered the concatenation effect at the deeper layers, (2) it will distort the actual target class if it is duplicated in the deeper layers.

In the deep temporal convolution network, the concatenation does not stop at just the input layer. It will carry on at the hidden layers. As such, there is no hurry to perform the majority voting at the input layer. It should be done only at the final output stage where the error needs to be computed.

Therefore, instead of majority voting at each concatenation sublayer, it is proposed that the target class label of each observation in the many-to-one time series be accumulated through the deeper layers and be subjected to majority voting only at the final output layer. This will track the target class label not just at the input layer but all the way through the deeper layers until the final output layer where majority vote need to be applied.

This can be done easily by first (before any concatenation) expressing the target class labels $\boldsymbol{y}(t)$ in the one-hot encoding format, and then add up the values whenever their associated data are involved in concatenation. This will result in the modified one-hot encoding format, where the integer represents the pooled class count of a particular class.

Pooling the target labels through the deeper layers of the deep temporal convolution network avoids the loss of information, thus enhancing the validity of the target labels. The proposed pooling of the target labels through the deeper layers is shown in Figure 3.16 below.

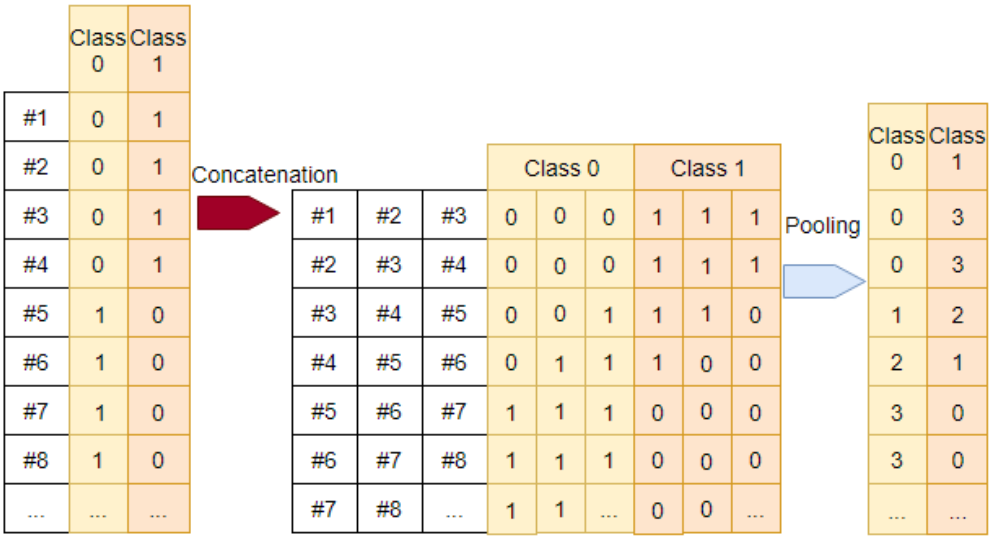

Figure 3.16. Pooling of the class counts of newly concatenated data.

As an example, consider the modified one-hot encoding format shown in Figure 3.14 earlier on. It is reproduced on the left hand side of Figure 3.17 below for ease of reference. Subjecting it to a concatenation of $TS = 3$, the class counts of the concatenated data set is shown on the right hand side of Figure 3.17 below.

Class 0	Class 1
14	2
9	7
3	13
9	7
2	14
4	12

Class 0	Class 1	Class 0	Class 1	Class 0	Class 1
14	2	9	7	3	13
9	7	3	13	9	7
3	13	9	7	1	15
9	7	2	14	4	12
2	14	4	12	⋮	⋮
4	12	⋮	⋮	⋮	⋮

Figure 3.17. Class counts of newly concatenated data set

Pooling is applied to the concatenation sublayer here. The pooled class count of the concatenated data set is shown in Figure 3.18 below.

Class 0	Class 1
26	22
21	27
13	35
15	33
⋮	⋮
⋮	⋮

Figure 3.18. Pooled class count of target class labels, concatenation sublayer

If the next layer is the final output layer, majority voting will have to be carried out there. The result is as shown in Figure 3.19 below.

Class 0	Class 1
1	0
0	1
0	1
0	1
⋮	⋮
⋮	⋮

Figure 3.19. Majority voting, final output layer

A minor problem encountered when implementing majority voting is the occurrence of ties, for example, a data vector having a pooled class count of 24 for class 0 and 24 for class 1. To break the tie, jittering both counts with a positive random value smaller than 1 will do.

3.3 Learning Algorithm

It is proposed that the deep temporal convolution network be trained in three phases: (1) pre-training as a stack of RBMs, (2) training of the final classifier, and (3) fine-tuning of the entire network by backpropagation with the proposed gradient routing method.

3.3.1 Pre-training

Pre-training of the MLP as a stack of RBMs was the breakthrough in 2006 by Hinton that started the deep learning movement. It is proposed that the deep temporal convolution network be likewise arranged as a stack of RBMs. The pre-training is by the same pair-wise unsupervised training with contrastive divergence as the DBN-DNN. The difference is that now, the visible layer of the RBM is the concatenation sublayer, not the hidden layer.

Figure 3.20 below illustrates the above idea with a pair of hidden layers, layer L_1 and layer L_2. In a standard DBN-DNN, the RBM will be between L_1 and L_2. However, in the proposed network, the weights are located between the concatenated sublayer L_{1c} and the next hidden layer L_2. Therefore, in the deep temporal convolution network, the RBM is formed between L_{1c} and L_2 instead.

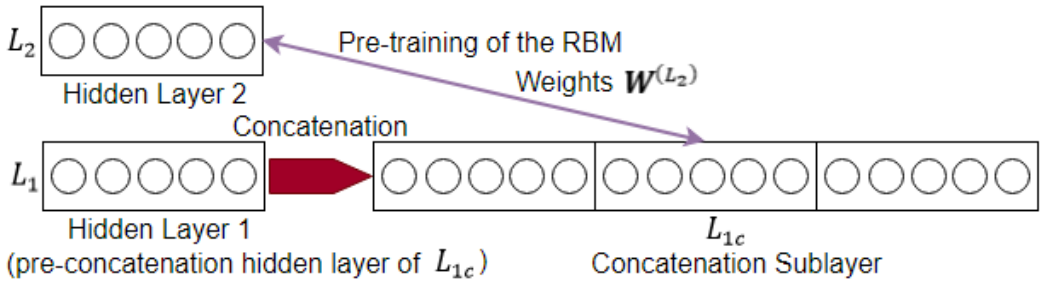

Figure 3.20. RBM in the deep temporal convolution network

With this arrangement, the output of the current RBM can be used as the input of the next RBM, albeit through the concatenation sublayer. By repeating this process, a stack of RBM can be created. There is no backward computation down the stack during pre-training, since RBM training by contrastive divergence is a greedy layer-wise process. Once an RBM is trained, it is considered "fitted" and its weights are deemed as "good" enough for the deep temporal convolution network.

3.3.2 Training of the Final Classifier

At the input side of the final classifier, deep learning by the hidden layers of the deep temporal convolution network would have already extracted the temporal patterns as a set of stationary features suitable for classification. As such, the final classifier need not be too complex. The consideration in selecting the kind of classifier will be based on the classifier's capability (e.g. binary classifier or multi-class classifier) and its computational efficiency during deployment.

The softmax classifier is the simplest kind of multi-class classifier and is suggested for the deep temporal convolution network. It was described in the review in Chapter 2 and will not be repeated here.

3.3.3 Backpropagation

In general, the weights in a network can be updated by gradient descend, as shown in Equation (3.6) below.

$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \alpha \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}} \tag{3.6}$$

In Equation (3.6) above, $J(\boldsymbol{W})$ is the cost function. It is abbreviated hereafter as the error E.

To update the weights of a layer in a network with many layers, say the i-th layer, the contribution of the i-th layer to the error E should be determined precisely. That contribution, sometimes referred to as the delta or the sensitivity, is denoted as $\boldsymbol{\delta}^{(L_i)}$, where L_i is the i-th layer. It is by definition the derivative of the cost function E with respect to the linear output $\boldsymbol{y}^{(L_i)}$, and is shown in Equation (3.7) below.

$$\boldsymbol{\delta}^{(L_i)} \triangleq \frac{\partial E}{\partial \boldsymbol{y}^{(L_i)}} \tag{3.7}$$

The determination of $\boldsymbol{\delta}^{(L_i)}$ should proceed layer by layer in the backward direction. It is a well-known procedure, described in the review in Chapter 2. There is one catch, though. All the sections in the backward path must be able to be linked together by the chain rule of derivative.

The proposed network does not satisfy the above condition. Thus, the standard backpropagation procedure will not work for the temporal deep convolution network. This is because the concatenation operation is not a smooth function, and so the backward path from the concatenation sublayer to the pre-concatenation hidden layer is not differentiable. In Figure 3.21 below, the non-differentiable section is from L_{1c} to L_1.

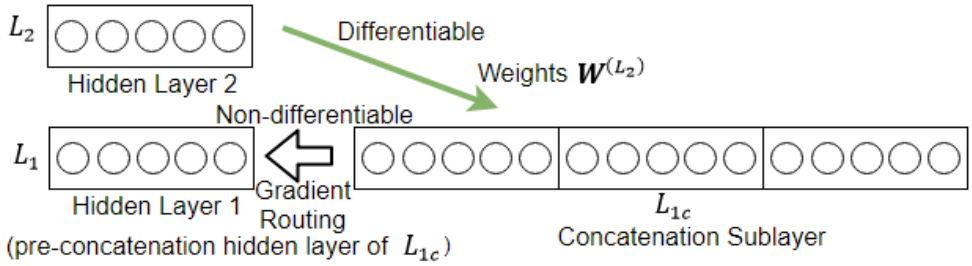

Figure 3.21. Backward path from concatenation sublayer to the pre-concatenation hidden layer

By the chain rule of derivative, the contribution of the i-th layer $\boldsymbol{\delta}^{(L_i)}$ can be factorized as the product of four terms, as shown in Equation (3.8) below.

$$\boldsymbol{\delta}^{(L_i)} \triangleq \frac{\partial E}{\partial \boldsymbol{y}^{(L_i)}} = \frac{\partial E}{\partial \boldsymbol{y}^{(L_{i+1})}} \frac{\partial \boldsymbol{y}^{(L_{i+1})}}{\partial \boldsymbol{a}^{(L_{ic})}} \frac{\partial \boldsymbol{a}^{(L_{ic})}}{\partial \boldsymbol{a}^{(L_i)}} \frac{\partial \boldsymbol{a}^{(L_i)}}{\partial \boldsymbol{y}^{(L_i)}} \tag{3.8}$$

By tracing through the four terms in Equation (3.8) above, it can be seen that the delta passes through the following parts of the network:

(1) $\boldsymbol{y}^{(L_{i+1})}$, the linear output of the upper hidden layer L_{i+1}

(2) $\boldsymbol{a}^{(L_{ic})}$, the activation of the layer L_{ic}, which is a concatenation sublayer

(3) $\boldsymbol{a}^{(L_i)}$, the activation of the layer L_i, which is the pre-concatenation hidden layer

(4) $\boldsymbol{y}^{(L_i)}$, the linear output of the layer L_i

The first two terms in Equation (3.8) above pose no problem for computation. The first term is, by definition, the delta of the upper layer $\boldsymbol{\delta}^{(L_{i+1})}$, and so is available from the previous

calculation in backpropagation. The second term is, by the application of differentiation, the weight $\boldsymbol{W}^{(L_{i+1})}$ of the upper layer L_{i+1}, since $\boldsymbol{y}^{(L_{i+1})} = \boldsymbol{W}^{(L_{i+1})}\boldsymbol{a}^{(L_{ic})}$. With these two terms available, their product, denoted as $\frac{\partial E}{\partial \boldsymbol{a}^{(L_{ic})}}$ in Equation (3.9) below, can be computed directly by multiplication according to the chain rule.

$$\frac{\partial E}{\partial \boldsymbol{a}^{(L_{ic})}} = \frac{\partial E}{\partial \boldsymbol{y}^{(L_{i+1})}} \frac{\partial \boldsymbol{y}^{(L_{i+1})}}{\partial \boldsymbol{a}^{(L_{ic})}} = \boldsymbol{W}^{(L_{i+1})}\boldsymbol{\delta}^{(L_{i+1})} \tag{3.9}$$

The third term in Equation (3.8) above is problematic. It is over the concatenation operation, which is non-differentiable. As a result, the product of the first three terms, denoted as $\frac{\partial E}{\partial \boldsymbol{a}^{(L_i)}}$ in Equation (3.10) below, cannot be computed directly by multiplication according to the chain rule.

$$\frac{\partial E}{\partial \boldsymbol{a}^{(L_i)}} = \frac{\partial E}{\partial \boldsymbol{y}^{(L_{i+1})}} \frac{\partial \boldsymbol{y}^{(L_{i+1})}}{\partial \boldsymbol{a}^{(L_{ic})}} \frac{\partial \boldsymbol{a}^{(L_{ic})}}{\partial \boldsymbol{a}^{(L_i)}} \tag{3.10}$$

Although non-differentiable, concatenation is an invertible operation. The proposed solution is to make use of gradient routing to unstack the concatenation, so as to change $\frac{\partial E}{\partial \boldsymbol{a}^{(L_{ic})}}$ in Equation (3.9) above to $\frac{\partial E}{\partial \boldsymbol{a}^{(L_i)}}$ in Equation (3.10) above. The transformation that the gradient routing intends to achieve is shown in Equation (3.11) below.

$$\frac{\partial E}{\partial \boldsymbol{a}^{(L_{ic})}} \rightarrow \frac{\partial E}{\partial \boldsymbol{a}^{(L_i)}} \tag{3.11}$$

The very last term in Equation (3.8) above, i.e. $\frac{\partial \boldsymbol{a}^{(L_i)}}{\partial \boldsymbol{y}^{(L_i)}}$, is the derivative of the activation function. The derivative is well known for activation functions that are common, such as sigmoid or ReLU [136]. It can be computed and then multiplied with the result of gradient routing in an element-wise manner. The result is the delta $\boldsymbol{\delta}^{(L_i)}$ that was shown earlier on in Equation (3.8) above.

This completes the backpropagation with gradient routing for the i-th layer. With delta $\boldsymbol{\delta}^{(L_i)}$ available, it can used to compute the error gradient, which is then used to update the weights.

3.3.4 Gradient Routing

What gradient routing does is to redistribute the error contribution from the concatenation sublayer to the actual hidden layer. In other words, the amount of "delta" that the actual hidden layer receives is exactly the same as the "delta" passed to it from the concatenation sublayer.

Another important point to note is that there is no learning involved in gradient routing, as there is no adjustable parameters to be learnt.

The proposed "split-slide-add" method implements the gradient routing in the deep temporal convolution network. First, the error attributed to the concatenation sublayer is split into its pre-concatenation parts. Then, the pre-concatenation parts are aligned in time by sliding. Finally, the aligned parts are summed together.

To illustrate the "split-slide-add" method, consider the error contribution from the concatenation sublayer L_{ic}, i.e. $\frac{\partial E}{\partial \boldsymbol{a}^{(L_{ic})}}$ in Equation (3.9) above. Figure 3.22 below shows a table with 16 rows. Each of the rows in the table is the contribution of a particular concatenated vector in L_{ic} to the error. There are 16 concatenated vectors in L_{ic} in this example, as it is assumed here that the mini-batch size is 18 and that the concatenation is done with the time steps TS set to 3.

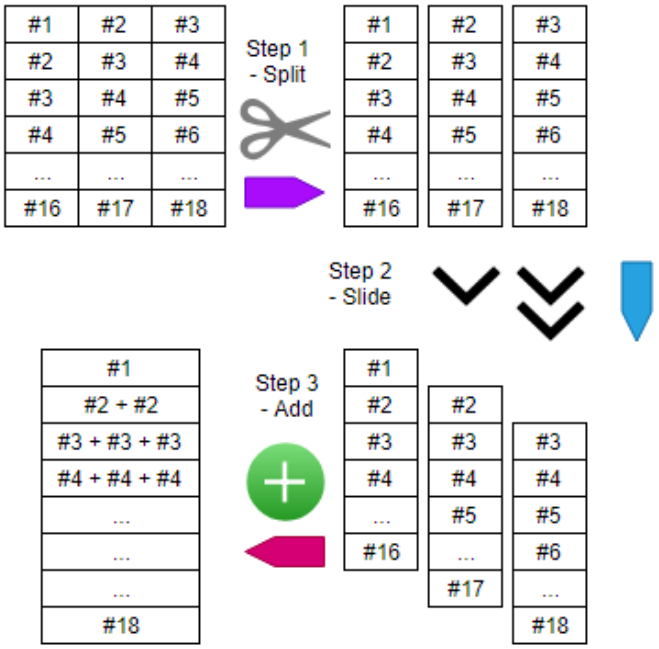

Figure 3.22. The three steps of the "split-slide-add" method for gradient routing

The first operation is to split the table into separate columns. Each of the columns is the contribution of the data before concatenation. The second operation, that of sliding the columns, aligns the pre-concatenation parts according to their natural temporal order. This enables the summation in the third step to be meaningful. In summation, the values in the columns, now aligned in time, are added together. In consequence, $\frac{\partial E}{\partial \boldsymbol{a}^{(L_{ic})}}$ is transformed to $\frac{\partial E}{\partial \boldsymbol{a}^{(L_i)}}$.

Gradient routing redistributes the error contribution from the concatenation sublayer L_{ic} to the pre-concatenation hidden layer L_i. It does not involve any learning of the weight values. In other words, the amount of delta that the pre-concatenation hidden layer receives is exactly the same as the delta passed to it from the concatenation sublayer.

With gradient routing done with the "split-slide-add" method, the proposed network will be able to learn about the temporal context passed through the network by the concatenation operation, even though it is a non-differentiable operation.

Once the error at the concatenation sublayer is re-distributed properly by gradient routing, the backward distribution of error to the other layers can proceed as usual through the use of backpropagation. Any gradient descend method can be used in the backpropagation procedure,

for example the conjugate gradient method. The entire backpropagation with gradient routing procedure should be repeated over multiple epochs.

The trained model thus obtained is now ready for test. An optional step could be taken before that. The softmax layer could be replaced by the extreme learning machine (ELM). The output of the penultimate layer of the deep temporal convolution network is passed to the ELM for training. The use of ELM as the final classifier (instead of the softmax layer) could give the system performance a lift.

The proposed methodology is different from the other convolutional neural networks used for time series classification. In those networks, the up-sampling function used in the pooling layer for backward propagation of error is done based on the individual data instances. In contrast, in the deep temporal convolution network, the gradient routing in the backward path are done in mini-batches. These mini-batches keep the short-term temporal order of the data instances within them, and so the learning algorithm is able to learn the temporal context in the deeper layers of the network.

3.4 Summary of Deep Temporal Convolution Network

A number of issues in implementing the deep temporal convolution network are addressed in this chapter, covering the network structure, the data preparation and the learning algorithm.

As an example of the performance improvement by the deep temporal convolution network, Table 3.1 below shows the classification accuracies (in percentage) of the 10-fold validation on the 14-channel EEG Eye State data set. Different values of TS (time steps) are used for the deep temporal convoltuion network.

Table 3.1. 10-fold cross-validation of DTCN, eye state

TS	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	Fold 6	Fold 7	Fold 8	Fold 9	Fold 10
1	93.11	95.54	95.00	94.86	91.62	86.49	96.22	94.19	91.49	82.57
2	93.51	97.97	99.46	99.32	98.92	98.78	92.97	97.97	97.97	98.11
5	99.01	98.92	100	98.92	99.73	100	99.59	99.01	99.59	99.73

The mean of the 10-fold cross-validation is the figure of merit used for comparing the effect of TS (time steps). This is shown in Table 3.2 below.

Table 3.2. Mean of 10-fold cross-validation of DTCN, eye state

TS	Mean	Std Dev
1	92.11	10.27
2	97.5	5.42
5	99.46	1.01

As can be seen from Table 3.2 above, the accuracy improves from 92.11% when there is no concatenation to 97.50% when $TS = 2$. It further improves to 99.46% when $TS = 5$.

In general, for the deep temporal convolution network, the number of nodes required for the input and hidden layers is reduced, as more temporal context is available in the deeper layers. On the other hand, more epochs are required for training, as there is more temporal context to be learnt. Overall, there is an improvement in generalization performance over the equivalent time delay neural network. This validates the assumption that it is useful to pass the temporal context of the time series data through a network for deep learning.

.

Chapter 4. Multi-view Temporal Ensemble

This chapter describes the novel method of data fusion by multi-view temporal ensemble. It is applicable to the classification of non-stationary time series data such as sounds, where it is often tedious and expensive to get a training set that is representative of the target concept. This method alleviates the problem by treating the outputs of deep learning sub-models as the views of the same target concept, which are then linearly combined according to their complementarity. The complementarity of the data mixes the outputs of the deep learning sub-models optimally. This results in the final classifier having a better performance than the individual sub-models.

The view's complementarity is the contribution of the view to the global view. In this work, the Laplacian eigenmap of the combined data is the global view. The method uses alternate optimization to compute the complementarity. It involves the cost function of the Laplacian eigenmap and the weights of the linear combination of the views. By blending the views in this way, a more complete view of the underlying phenomenon can be made available to the final classifier. Better generalization is obtained, as the consensus between the views reduces the variance while the increase in the discriminatory information reduces the bias.

This method, which is an intermediate data fusion technique, consists of the following steps: (1) train with deep learning sub-models such as the deep temporal convolution network or the CNN-LSTM model, (2) determine the complementarity of the outputs of the sub-models by multi-view spectral embedding, and (3) make use of the complementarity as the mixing coefficients of the linear mixture.

Although there have been previous work on multi-view spectral embedding (MSE) [137] for clustering, classification, and dimensionality reduction, there is no work on it for the ensemble mixing of temporal signals. The proposed method is also different from multi-kernel training as the sub-models used are deep learning models, not the SVM kernels. The interpretation of the mixing coefficient (obtained by the two-step method that will be described later) is new and provides the basis for the weighted combination of the sub-models. The multi-view temporal ensemble is thus a novel framework for the classification of temporal signals, whether they are multivariate, heterogeneous or multimodal. Data experiment with the artificial views of

environment sounds formed by deep learning structures of different configurations shows that the proposed method can improve the classification performance.

4.1 Overview

In this work, deep learning creates the artificial views of the time series data, which are then used in multi-view learning for classification. The framework makes use of (1) the linear relationship of the deep learning sub-models, where the output of each sub-model is seen as a view, (2) the computation of the complementarity of the views, and (3) the formation of the time-frequency view by the sub-models.

Figure 4.1 shows the architecture of the proposed network. The time series data are first decomposed in the time-frequency domain to expose the spectral aspect of the time series to the deep learning sub-models. The features extracted by the sub-models form the views. These views are redundant in information, but they are not quite similar to each other as they are produced by deep learners of different configurations. These views can be combined according to their complementarity. The combined data, being more representative of the target concept, will result in better performance in the final classifier.

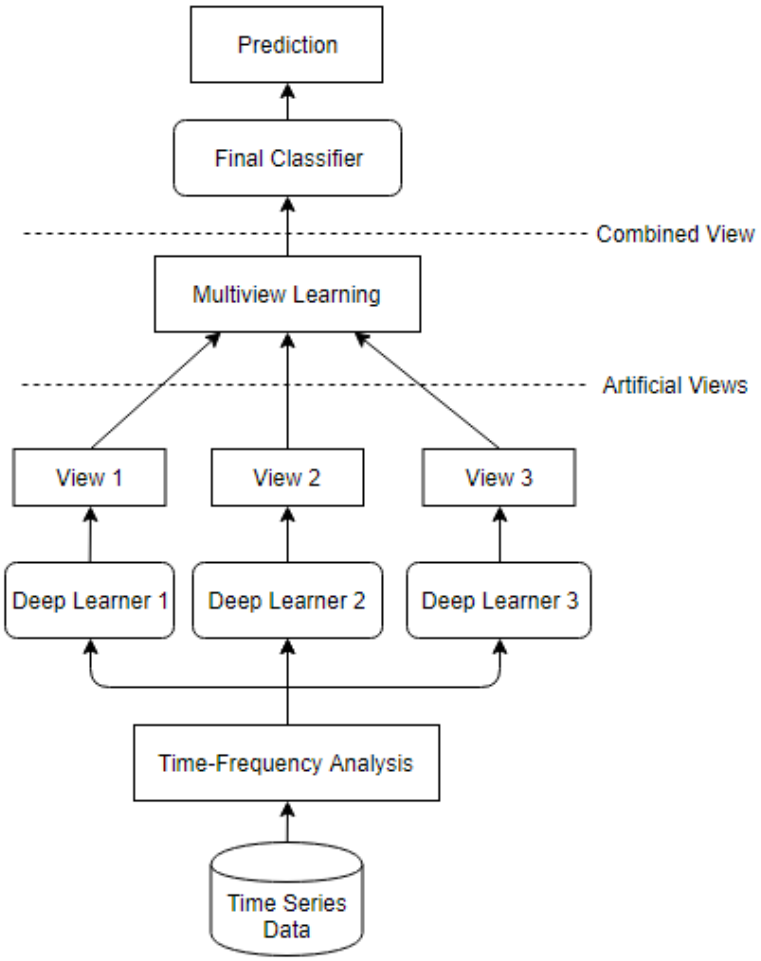

Figure 4.1. Architecture of the proposed multi-view temporal ensemble.

The proposed framework addresses the problem of the strong dependency of the performance of a trained model on the representativeness of the data. As is well known, it is tedious and expensive to construct a representative training set, due to the extensive manual curation and annotation that are needed. This is particularly true for time series data, as clear segmentation is not readily available. By treating the outputs of the deep learning sub-models as the views of the same target concept, the dependency on any one of the views could be weakened through the appropriate use of the views' complementarity. This helps reduce the need for clear segmentation and improve the generalization performance of the classifier.

4.1.1 Construction of Views

In a traditional single-view classifier, the training set consists of a single feature set $\boldsymbol{V}$. By contrast, in the multi-view setting, there are $\boldsymbol{M}$ views, denoted as $\boldsymbol{V^{(i)}}, \boldsymbol{i} \in \{\mathbf{1}, \ldots, \boldsymbol{M}\}$. Each of these views is sufficient for the learning of the target concept. However, in this work, the alternative approach, that of fusing the $\boldsymbol{M}$ views according to their complementarity, is proposed. This results in a common feature set that is more representative of the target concept compared to the individual views.

The proposed method to construct the views is to subject each of the input data segments, $\boldsymbol{x} \in \mathbb{R}^{\boldsymbol{d}}$ of length $\boldsymbol{d}$, to a number of deep learning sub-models that are configured differently in terms of the number of hidden nodes. With different configurations, it is tantamount to a random split of the input data that results in views that are distinct from each other.

The view to be retrieved from the sub-model is the penultimate layer of the sub-model, rather than the final softmax layer. The penultimate layer can be thought of as the feature set that is extracted by deep learning from the input data. Notation wise, it can be represented as the approximate function $\boldsymbol{f}(\boldsymbol{x})$ of the input data $\boldsymbol{x}$. As there are $\boldsymbol{M}$ different sub-models, represented as $\boldsymbol{f^{(1)}}(\cdot)$, $\boldsymbol{f^{(2)}}(\cdot)$, …, $\boldsymbol{f^{(M)}}(\cdot)$, so $\boldsymbol{M}$ views will be available, i.e. $\boldsymbol{f^{(1)}}(\boldsymbol{x})$, $\boldsymbol{f^{(2)}}(\boldsymbol{x})$, …, $\boldsymbol{f^{(M)}}(\boldsymbol{x})$.

According to the Representer theorem [138], the approximate function of a machine learning model is the linear combination of the basis functions. Thus, assuming that each of the views is a basis function, the views can be combined linearly, with appropriate weights assigned to the linear combination, as shown in Equation (4.1) below.

$$\boldsymbol{V}_{combined} = \sum_{i=1}^{M} \alpha^{(i)} \boldsymbol{V}^{(i)} \tag{4.1}$$

The weights $\boldsymbol{\alpha^{(i)}}, \boldsymbol{i} \in \{\mathbf{1}, \ldots, \boldsymbol{M}\}$, are the mixing coefficients of the ensemble. The restriction on $\boldsymbol{\alpha^{(i)}}$ is according to the linear sum as shown in Equation (4.2) below.

$$\sum_{i=1}^{M} \alpha^{(i)} = 1, \alpha^{(i)} > 0 \tag{4.2}$$

The value of $\boldsymbol{\alpha^{(i)}}$ is the complementarity of the $\boldsymbol{i}$-th view. It is the probability of the $\boldsymbol{i}$-th view being compatible with the common target concept. An example of the construction of three different views by deep learning is shown in Fig. 3 below.

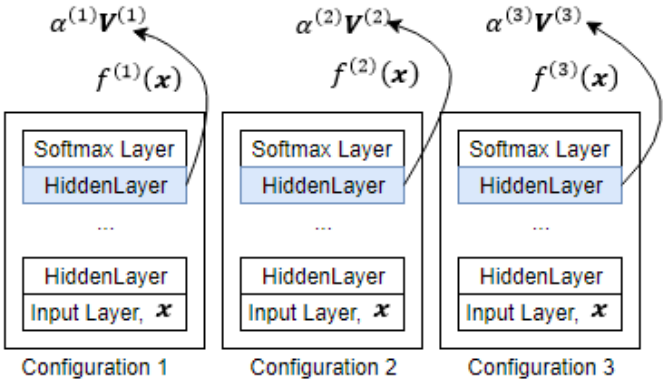

Figure 4.2. Architecture of the proposed multi-view temporal ensemble.

The view $f^{(i)}(\boldsymbol{x})$ is probabilistic in nature. It can be expressed as the density function $p\left(f^{(i)}(\boldsymbol{x})|\boldsymbol{x}, \boldsymbol{\theta}^{(i)}\right)$, where $\boldsymbol{\theta}^{(i)}$ is the parameters of the i-th model $f^{(i)}(\cdot)$. The linear mixture of the M views is also probabilistic, as shown in Equation (4.3) below.

$$p(f(\boldsymbol{x})|\boldsymbol{\Theta}) = \sum_{i=1}^{M} \alpha^{(i)} p^{(i)}(f^{(i)}(\boldsymbol{x})|\boldsymbol{x}, \boldsymbol{\theta}^{(i)})) \tag{4.3}$$

In Equation (4.3) above, $\boldsymbol{\Theta} = \left(\alpha^{(1)}, \ldots, \alpha^{(M)}, \boldsymbol{\theta}^{(1)}, \ldots, \boldsymbol{\theta}^{(M)}\right)$ is the parameter set of the multi-view temporal ensemble.

These views, i.e. $f^{(1)}(\boldsymbol{x})$, $f^{(2)}(\boldsymbol{x})$, …, $f^{(M)}(\boldsymbol{x})$, are not independent. They bear some similarity with the rest of the views, and at the same time, something unique to themselves. Ideally, the weighted sum of these views, i.e. the global view, should contain the most discriminative information. It should retain features that are similar across the views and enhance features that

are unique in the individual views. The key to achieve this is the determination of the optimal mixing coefficient $\left(\alpha^{(1)}, \ldots, \alpha^{(M)}\right)$.

4.1.2 Complementarity

Intuitively, views that are independent and supplemental will contribute equally to the global view of the combined data. The weight of each of these views is the average weight $1/M$. This will result in a less noisy combined output. When the combined output is used as the input of the final classifier, the overall system performance will have a lower variance [139].

On the other hand, if a view contains complementary information, it will contribute more to the global view, and its weight will be higher than $1/M$. This is at the expense of the view that contains less complementary information.

So, instead of using the average weight $1/M$ for $\alpha^{(i)}$, it is proposed in this work that the complementarity of the views be used as the weights. The purpose is for the combined output to have a higher probability of a lower generalization error. Thus, the larger the contribution of the view to the global view, the more complementary it is, and the higher the weight should be.

However, the global view of a linear mixture is actually latent, given the individual views. In other words, although the global view can be obtained from the weighted sum of the individual views, it begs the question of what the weight values should be for the linear combination.

The candidate method to solve the minimization problem with two unknowns (the weights and the global view) is alternate optimization. An example of alternate optimization is the expectation maximization (EM) method used in the Gaussian mixture [140].

The multi-view temporal ensemble uses a similar approach. The cost function, which has to be defined in alternate optimization, is based on that of Laplacian eigenmap [74], a non-linear data reduction technique. This work modifies it for use in the multi-view setting. This will be described later when the computation method for complementarity is explained.

4.1.3 Time-Frequency Features

Time-frequency decomposition exposes the spectral changes in the time series data to the sub-model $f(\cdot)$ and is useful for the analysis of signals that are non-stationary. While there are many time-frequency analysis techniques (for example, Wigner-Ville decomposition [141], empirical mode decomposition [142], wavelet transform etc.), the most common practice, particularly for multivariate signals, is still the short time analysis method, such as the spectrogram and the Mel-frequency cepstrum [143], where the signal is split into overlapping segments and transformed to their time-frequency representation.

The sub-models in the ensemble can be a generalized linear model, decision tree, k nearest neighbour, or neural network. In the past decade, deep learning, which is the composition of layers of models, has been found to be effective in the classification of raw signals.

Deep learning, as a feature extractor, has a smooth output in the feature space that can be classified easily by the final classifier. Not only can it approximate the function with an exponentially lower number of training parameters compared to a shallow network, it is also more immune to overfitting [5].

The workhorses of deep learning are deep belief net [11], convolutional neural network (CNN) [37] and long short-term memory (LSTM) recurrent neural network [40]. These models form practical models for signal classification in various combinations.

In this work, the CNN-LSTM model is proposed for use in the multi-view temporal ensemble. The reason for using the CNN-LSTM model is to extract the temporal and spectral patterns from the time-frequency aspect of the time series. The lower CNN layer receives the input data in two-dimension, while the LSTM works on the subsequently flattened layer in one-dimension.

In the CNN-LSTM model, the fully-connected layer just before the final softmax layer is the penultimate layer. It contains the features that form the view of the sub-model. By linearly combining the penultimate layers of the CNN-LSTM sub-models, a new input can be formed for the final classifier. This qualifies the proposed multi-view temporal ensemble as an intermediate data fusion technique, rather than a late data fusion technique. This is because the

penultimate layer represents the feature extracted by the sub-model, not the decision made by the sub-model.

4.2 Equality of Target Concept

The purpose of computing complementarity is to produce data that are more representative of the target concept. For the computation to be valid, the data points across the views must be descriptive of the same target concept. Careless implementation, such as ad hoc randomization of the data instances, will destroy the equality of target concept across the views.

For time series data, the equality of target concept across the views translates to the following rules:

(1) The data points across the views must be aligned in time, i.e. co-occurring.
(2) The data points must belong to the same class, i.e. class-specific.

The proposed solutions to satisfy the two requirements (co-occurrence and class-specificity) are:

(1) Use the same data vectors for the inputs of all the sub-models. With this, the data vectors will be co-occurring at the outputs of the sub-models. It will be so as long as there is no randomization of the data vectors within the sub-models.
(2) Re-arrange the outputs of the sub-models by class without disrupting the time order. Then compute the complementariness based on the class-specific data. After that, recombine the data from the classes and shuffle the data for the final classifier.

4.2.1 Co-occurrence

Co-occurrence does not preclude the shuffling of data points in the individual views, which is often a necessary operation to achieve independent and identical distribution of the input data for model training. The rule of co-occurrence merely states that the same shuffled order must be used across the views so that the data points across the views will occur at the same time point and thus describe the same target concept.

To ensure co-occurrence at the penultimate layers of the deep learning sub-models, the same data set (shuffled and so random in order) will have to be used as the inputs for all the sub-models. As long as there is no randomization of the data vectors in the sub-models, the outputs at the penultimate layers of the sub-models will be co-occurring. These outputs, which are co-occurring, can then be used as the views for multi-view learning.

Figure 4.3 below illustrates this idea by showing a set of $N = 6$ data vectors for each of the three views. As can be seen from the figure, the time order in any single view is random, but it is co-occurring across the views.

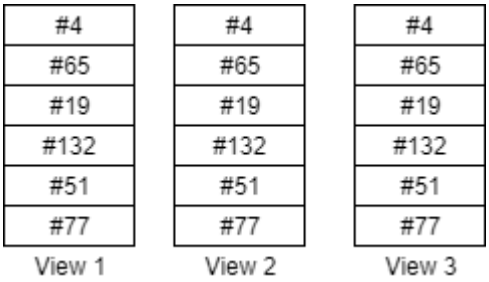

Figure 4.3. Co-occurrence

The rule of co-occurrence has to be enforced during both the training and the testing process. It means that the algorithm must not randomize the order of data vectors within itself.

4.2.2 Class-specificity

The rule of class-specificity applies to multi-view learning because complementarity can only be determined among data of the same class. It cannot be used for classes that are different.

The cats and dogs analogy illustrates this idea. For a set of dog images and a set of cat images, it is meaningful to define the complementarity of the images within the sets (either the cats or the dogs) but not across the sets. This is because complementarity is ill defined for a combined data set that has different concepts.

The proposed solution to satisfy the requirement of class-specificity is to re-arrange the outputs of the sub-models by class, yet without disturbing the time order necessary for co-occurrence.

Complementarity is then computed on the class-specific data, which is then combined across the views.

This process, when carried out separately for the classes, will result in linearly combined data that are class-specific. The data of these classes will have to be stacked together as one single feature set and then shuffled so that they can be used as the input by the final classifier.

The rule of class-specificity seems to contradict the testing requirement in machine learning, where the class in the test set is assumed unknown. Class-specific data seems impossible when the class information is not available in the test set.

Actually, this is not a problem in the proposed multi-view temporal ensemble. This is because the sub-models can predict the class during testing. The predicted class, instead of the actual class, can be used to re-arrange the outputs of the sub-models. The linearly combined data, based on the predicted classes, are then used by the final classifier for the final prediction.

4.3 Implementation

This section first provides the overview of how to compute complementarity and is followed by the details in the sub-sections.

The input, output, and initial weight values for the computation are as shown below:

Input: A set of M data matrices, each with N data points of length d, $\boldsymbol{X} = \left\{\boldsymbol{X}^{(i)} \in \mathbb{R}^{N \times d}\right\}_{i=1}^{M}$

Output: A set of M mixing coefficients, $\boldsymbol{\alpha} = \left\{\alpha^{(i)}\right\}_{i=1}^{M}$

Initialize $\boldsymbol{\alpha} = \left[\frac{1}{M}, \ldots, \frac{1}{M}\right]$

$\boldsymbol{X}^{(i)}$ represents a group of N co-occurring data vectors of the same class in the i-th view. N is typically a small number less than 32. For a given data set, the complementarity will be computed in many such mini-batches across the views.

A summary of the terms used in this section are shown below:

$\boldsymbol{W}$ – Weighted adjacency matrix of a view, $\boldsymbol{W} \in \mathbb{R}^{N \times N}$

$\boldsymbol{L}$ – Laplacian matrix of a view, $\boldsymbol{L} \in \mathbb{R}^{N \times N}$

$\boldsymbol{Y}$ – Spectral embedding of a view, $\boldsymbol{Y} \in \mathbb{R}^{N \times m}$, $m < N$

$\boldsymbol{W}^{(i)}$ – Weighted adjacency matrix of the i-th view

$\boldsymbol{L}^{(i)}$ – Laplacian matrix of the i-th view

$\boldsymbol{Y}^{(i)}$ – Spectral embedding of the i-th view

$\boldsymbol{L}^{(G)}$ – Laplacian matrix of the global view

$\boldsymbol{Y}^{(G)}$ – Spectral embedding of the global view

$\alpha^{(i)}$ – Complementarity (i.e. the mixing coefficient, or weight) of the i-th view

To compute complementarity, a set of N data points in the i-th view are first represented as an adjacency matrix $\boldsymbol{W}^{(i)}$ [144]. This matrix describes the distances between pairs of data points. It can be seen as the local proximity information of the data points in the data manifold.

From the adjacency matrix $\boldsymbol{W}^{(i)}$ of the i-th view, the Laplacian matrix $\boldsymbol{L}^{(i)}$ of the i-th view can be computed quite easily. The individual views $\boldsymbol{L}^{(i)}, i \in \{1, \dots, M\}$ are then linearly combined to form the global view $\boldsymbol{L}^{(G)}$ with the initial weight values. Once the global view $\boldsymbol{L}^{(G)}$ is obtained by linear combination, its spectral encoding $\boldsymbol{Y}^{(G)}$ can be computed directly by eigen-decomposition [14].

The global spectral embedding $\boldsymbol{Y}^{(G)}$, in turn, can be used with the Laplacian matrices $\boldsymbol{L}^{(i)}$, $i \in \{1, \dots, M\}$, to compute the weights $\alpha^{(i)}$. These weights are used to update the global view $\boldsymbol{L}^{(G)}$.

In this way, through the alternate updates of the global view $\boldsymbol{L}^{(G)}$ and the weights $\alpha^{(i)}$, the global spectral embedding $\boldsymbol{Y}^{(G)}$ will converge to $\boldsymbol{Y}^{(G)*}$. At this point in time, the weights $\alpha^{(i)}$ will represent the complementarity of the i-th view, relative to the other views. Figure 4.4 below illustrates this process.

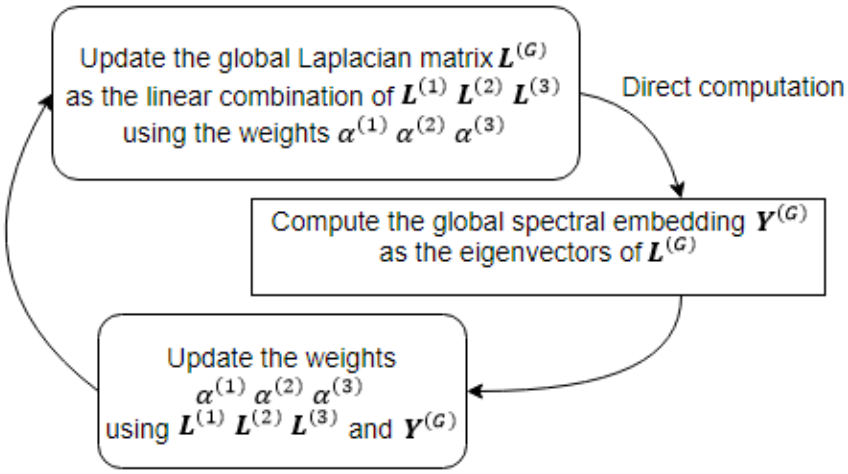

Figure 4.4. Alternate optimization of $\boldsymbol{L}^{(G)}$ and $\alpha^{(i)}$

The iterative process in Figure 4.4 can be summarized as follows:

(1) Obtain $\boldsymbol{L}^{(i)}$ from a set of N co-occurring data vectors of the same class from the i-th view

(2) Align the individual $\boldsymbol{L}^{(i)}$ to the global spectral embedding in 2 steps:

   a. obtain $\boldsymbol{L}^{(G)}$ from $\boldsymbol{L}^{(i)}$ by linear combination, according to the weights $\alpha^{(i)}$

   b. obtain $\boldsymbol{Y}^{(G)}$ from $\boldsymbol{L}^{(G)}$ by eigen-decomposition, formed by the m eigen-vectors that correspond to the m smallest eigenvalues other than λ_0, where $0 = \lambda_0 \leq \lambda_1 \leq \cdots \lambda_1 \leq \cdots \leq \lambda_{N-1}$, and $m < N$

(3) Update the values of $\alpha^{(i)}$, which is the inverse of the trace of $\boldsymbol{Y}^{(G)^T}\boldsymbol{L}^{(i)}\boldsymbol{Y}^{(G)}$

Iterate through (2) if the norm of the change in $\boldsymbol{\alpha}$ is bigger than a user-defined threshold.

After the complementarity $\alpha^{(i)}$ of the mini-batches across the views are determined, they can be blended in a linear mixture as shown in Figure 4.5 below. Then shuffle the blended data for use as the input of the final classifier.

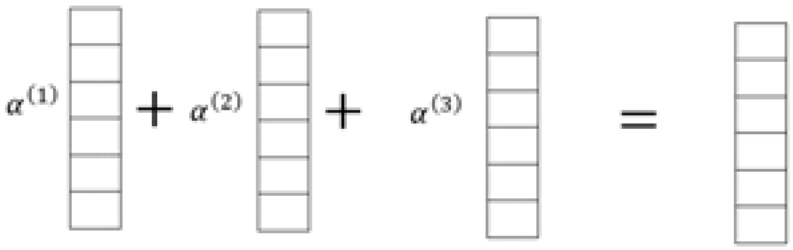

Figure 4.5. Linear combination in small batches of N co-occurring vectors of the same class

4.3.1 Laplacian Matrix of Individual View

For a set of N data points $\{\boldsymbol{x}_i \in \mathbb{R}^{\boldsymbol{d}}\}_{i=1}^{N}$ of an individual view, the weighted adjacency matrix $\boldsymbol{W}$ is a square symmetric matrix of size $N \times N$. The (i,j)-th entry of $\boldsymbol{W}$ can be computed according to Equation (4.4) below.

$$[\boldsymbol{W}]_{i,j} \tag{4.4}$$
$$= \begin{cases} exp\left(-\frac{\|\boldsymbol{x}_i - \boldsymbol{x}_j\|_2^2}{\sigma^2}\right) & if\ x_i\ and\ x_j\ are\ connected\ as\ k-nearest\ neighbours \\ 0 & otherwise \end{cases}$$

According to Equation (4.4) above, the entry $[\boldsymbol{W}]_{i,j}$ is cleared to 0 if the data points $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$, $i,j \in \{1,\dots,N\}$, are not connected. Whether $\boldsymbol{x}_i$ is connected to $\boldsymbol{x}_j$ depends on whether $\boldsymbol{x}_j$ is in the k-nearest neighbourhood of $\boldsymbol{x}_i$, where $k < N$ is a user-defined hyper-parameter. The value of $[\boldsymbol{W}]_{i,j}$ represents the proximity between $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ in the data manifold formed by the set of N data points. The closer the points, the higher the value of the proximity.

4.3.2 Spectral Embedding of Data Manifold

The spectral embedding $\boldsymbol{Y}^*$ of N data points in a single view can be obtained by a method called Laplacian eigenmap [13]. Laplacian embedding is a data reduction technique that projects the data points onto the alternative spectral view while preserving the local proximity of the data points in the new view. Conceptually, this preservation is achieved by the minimization of the cost function $J(\boldsymbol{Y})$ as shown in Equation (4.5) below:

$$J(\boldsymbol{Y}) = \sum_{i,j \in \{1,\ldots,N\}} \left\| \boldsymbol{y}_i - \boldsymbol{y}_j \right\|^2 [\boldsymbol{W}]_{i,j} \tag{4.5}$$

As seen from Equation (4.5) above, the cost function $J(\boldsymbol{Y})$ is the total amount of differences between two embedded vectors ($\boldsymbol{y}_i$ and $\boldsymbol{y}_j$, $i, j \in \{1, \ldots, N\}$), modulated by $[\boldsymbol{W}]_{i,j}$. When the data points $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ are in close proximity in the data manifold, the value of the adjacency matrix $[\boldsymbol{W}]_{i,j}$ will be large, thus contributing more to the cost function. This helps to promote the preservation of the local proximity in the resultant spectral embedding.

The solution $\boldsymbol{Y}^*$ of the above minimization problem [14] can be shown to reduce to

$$\boldsymbol{Y}^* = arg \min_{\boldsymbol{Y}^T \boldsymbol{D} \boldsymbol{Y} = 1, \boldsymbol{Y}^T \boldsymbol{D} \boldsymbol{1} = 0} tr(\boldsymbol{Y}^T \boldsymbol{L} \boldsymbol{Y}) \tag{4.6}$$

In Equation (4.6) above, $\boldsymbol{L}$ is the Laplacian matrix that can be computed as $\boldsymbol{L} = \boldsymbol{D} - \boldsymbol{W}$, where the diagonal matrix $\boldsymbol{D}$ is the degree of connectedness in the data manifold, i.e. $[\boldsymbol{D}]_{i,i} = \sum_{j=1}^{N} [\boldsymbol{W}]_{i,j}$.

Importantly, finding $\boldsymbol{Y}^* = arg \min_{\boldsymbol{Y}^T \boldsymbol{D} \boldsymbol{Y} = 1, \boldsymbol{Y}^T \boldsymbol{D} \boldsymbol{1} = 0} tr(\boldsymbol{Y}^T \boldsymbol{L} \boldsymbol{Y})$ is equivalent to finding the eigenvectors $\boldsymbol{Y}^*$ of the generalized eigenvalue problem $\boldsymbol{L}\boldsymbol{Y}^* = \boldsymbol{\lambda}\boldsymbol{D}\boldsymbol{Y}^*$. Thus, given the Laplacian matrix $\boldsymbol{L}$, the spectral embedding $\boldsymbol{Y}^*$ can be found easily. The spectral embedding $\boldsymbol{Y}^*$ can be computed directly by the eigen-decomposition of $\boldsymbol{L}$, even though it is a minimization problem that comes with a cost function.

With $\boldsymbol{L}$ and $\boldsymbol{Y}^*$ known, the cost value $tr(\boldsymbol{Y}^{*T} \boldsymbol{L} \boldsymbol{Y}^*)$ can be computed. The lower the cost value, the closer it is to reach the objective of preserving the local proximity of the data points in the spectral embedding.

The Laplacian eigenmap as described above is for single view only. It will have to be modified for use in the multi-view setting. This will be described in the next subsection.

4.3.3 Global View Problem

With multiple views, say M views, the solution of the minimization problem of a single view in Equation (4.6) becomes not useful at all. Not only is the global view $\boldsymbol{L}^{(G)}$ unknown, the weights that are needed to form $\boldsymbol{L}^{(G)}$ are also unknown.

The global view problem mentioned above can be framed as one with two sets of unknowns, the global view itself, and the weights that are needed to form the global view. The candidate method to solve a minimization problem with two sets of unknowns is alternate optimization. An example of alternate optimization is the expectation maximization (EM) method used in the Gaussian mixture [140]. Over there, the Gaussian parameters and the class distribution are both unknown initially. With one set of unknowns initialized and fixed, the other set of unknowns can be determined. This will continue iteratively by switching between the two sets of unknown, one fixed and the other one to be determined. The two-step process will continue until the unknown values converge, or until the pre-determined number of iterations is reached. Theoretically, this process is equivalent to the maximum likelihood estimation of the unknowns.

Following the method of patch alignment with multi-view spectral embedding for image and video [137], it is proposed that the global view $\boldsymbol{L}^{(G)}$ be formed as the linearly combination of the individual views $\boldsymbol{L}^{(i)}, i \in \{1, \dots, M\}$, based on the weights $\alpha^{(i)}$ (initialized as $1/M$).

$$\boldsymbol{L}^{(G)} = \sum_{i=1}^{M} \left(\alpha^{(i)}\right)^{r} \boldsymbol{L}^{(i)}, r > 1 \tag{4.7}$$

The minimization problem of a single view in Equation (4.6) will then become Equation (4.8) below:

$$\boldsymbol{Y}^{(G)*} = arg \min_{\boldsymbol{Y}^{(G)^T}\boldsymbol{Y}^{(G)}=\boldsymbol{1}} \sum_{i=1}^{M} \left(\alpha^{(i)}\right)^{r} tr(\boldsymbol{Y}^{(G)^T}\boldsymbol{L}^{(i)}\boldsymbol{Y}^{(G)}) \tag{4.8}$$

In Equation (4.8) above, the Laplacian matrix $\boldsymbol{L}^{(i)}$ is the i-th individual view. The Laplacian eigenmap $\boldsymbol{Y}^{(G)}$ is the spectral embedding of the global view $\boldsymbol{L}^{(G)}$.

The hyper-parameter r has been introduced in Equation (4.7) above. Its value should be $r > 1$. It is a trick to induce each view to contribute unequally to the global spectral embedding $\boldsymbol{Y}^{(G)*}$. If $r = 1$, the alternate optimization will end up with only the best view instead of the complementary views [145].

$\boldsymbol{Y}^{(G)*}$ in Equation (4.8) above can be computed directly from the global Laplacian matrix $\boldsymbol{L}^{(\boldsymbol{G})}$. It is the set of eigenvectors of $\boldsymbol{L}^{(\boldsymbol{G})}$ that correspond to the m smallest eigenvalues other than $\lambda_0 = 0$.

The eigenvectors are arranged in order of the eigenvalue, from the smallest eigenvalue to the largest value, up to the specified dimension $m < N$, where N is the dimension of the Laplacian matrix.

The eigenvectors with the smallest eigenvalues are selected because a compact representation in the projection space is desired. However, since the eigenvector associated with the smallest eigenvalue is likely to represent the noise, it will have to be discarded. Thus, only column vectors $\boldsymbol{Y}_{.j}, j \in \{2, \cdots m+1\}$ are used. The shape of $\boldsymbol{Y}$ is $(N \times m)$, where N is the number of data points in the mini-batch and m is the user-defined hyper-parameter value for the number of selected eigenvectors.

The initial weights in the cell array $\boldsymbol{\alpha}$ is based on non-informative assumption. For example, for a 3-view problem, the initial weights are $1/3$, $1/3$, and $1/3$. With these initial weight values, it is possible to obtain $\boldsymbol{L}^{(G)}$ from $\boldsymbol{L}^{(i)}$ and thus kick start the alternate optimization process.

4.3.4 Complementarity

The minimization problem shown in Equation (4.8) in the previous subsection has two sets of unknowns. The first set of unknowns are the weights $\boldsymbol{\alpha}$. Although they have some initial values, they cannot stay put at those values and must be updated to the values that represent complementarity. The second set of unknowns are the global Laplacian $\boldsymbol{L}^{(G)}$, without which the spectral embedding $\boldsymbol{Y}^{(G)}$ required for the update of the weights $\boldsymbol{\alpha}$ will not be available.

In this work, it is proposed that given $\boldsymbol{L}^{(G)}$ and thus $\boldsymbol{Y}^{(G)}$, the weight $\alpha^{(i)}$ in $\boldsymbol{\alpha}$ can be computed as shown in Equation (4.9) below. It is the inverse of the cost value of the i-th view, normalized across the M views.

$$\alpha^{(i)} = \left(1/tr\left(\boldsymbol{Y}^{(G)^T}\boldsymbol{L}^{(i)}\boldsymbol{Y}^{(G)*}\right)\right)^{\frac{1}{r-1}} / \sum_{i=1}^{M}\left(1/tr\left(\boldsymbol{Y}^{(G)^T}\boldsymbol{L}^{(i)}\boldsymbol{Y}^{(G)*}\right)\right)^{\frac{1}{r-1}} \tag{4.9}$$

With the values of $\boldsymbol{\alpha}$ computed as shown in Equation (4.9) above, it will then be possible to compute $\boldsymbol{L}^{(G)}$ from the linear combination of the individual views $\boldsymbol{L}^{(i)}$ according to Equation (4.7).

The aforementioned two-step iteration will continue until a stopping criterion is met. The L_2 norm in Equation (4.10) below can be used as the criterion for the convergence of the alternate optimization.

$$\sqrt{\sum_{i=1}^{M}\left(\alpha_k^{(i)} - \alpha_{k-1}^{(i)}\right)^2} < \varepsilon \tag{4.10}$$

In Equation (4.10) above, $\alpha_k^{(i)}$ is the weight at the k-th iteration and $\alpha_{k-1}^{(i)}$ is the weight at the $(k-1)$-th iteration. ε is a user-defined threshold that is set to a value much smaller than 1. The iteration continues until the change in the norm of $\boldsymbol{\alpha}$ in successive iterations is smaller than ε.

At convergence, $\alpha^{(i)}$ will have a value that is different from $1/M$. A value larger than $1/M$ means that the view is more complementary and contributes more to the global spectral embedding (compared to the other views). Conversely, a value smaller than $1/M$ means that the view is less complementary and contributes less to the global spectral embedding compared to the other views.

4.4 CNN-LSTM Sub-Model

The multi-view temporal ensemble, when applied to time series data, entails some considerations as shown below:

(1) In general, it is a good idea to decompose the time series into the time-frequency representation to expose the spectral features to the learner.
(2) A sub-model based on a good learner will be able to produce data that are smooth with respect to their target class labels, thus making the criterion of local proximity in spectral embedding achievable.
(3) Sub-models of the same type can be configured differently to generate artificial views from the same data set.

The use of CNN as the front end has been verified to be an effective method for time series data in previous works [146]. Thus, instead of using the raw data of a time series as the input, it is proposed that the data be transformed into its time-frequency domain for the multi-view temporal ensemble. The CNN will accept the data in the time-frequency domain in the tensor format of channel × height × width, where height is the time steps and width is the frequency bins.

With the two-dimensional CNN as the front end, a one-dimensional CNN can be added on top of it to extract the temporal features across the feature maps. This is then followed by an LSTM to extract the remaining high-level temporal features. Different configurations of such CNN-LSTM models can be used as the sub-models of the multi-view temporal ensemble to produce the views that are needed by multi-view learning.

4.4.1 Time-Frequency Representation

Time-frequency decomposition exposes the spectral changes in the signals and is useful for the analysis of signals that are non-stationary. Instead of using the raw data of a time series segment as input, it is first transformed into a set of time-frequency features by one of the many time frequency analysis techniques. For biosignals such as heart sounds, the mel scaled spectrogram will work better than the linear one because it provides higher frequency resolution in the lower frequency regions where the audible information for auscultation lies.

Figure 4.6 below illustrates this idea for a particular time series segment. The time series segment is split into 21 overlapping frames. Each frame is then transformed to a 60-bin Mel cepstrum. The resulting time-frequency features are arranged in a 2-D table as shown in Figure

4.6 below, where the time steps are in the vertical axis and the frequency bins are in the horizontal axis.

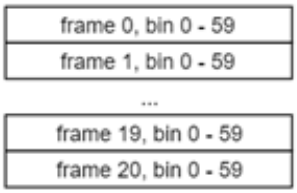

Figure 4.6. Time-frequency data format of a segment

A CNN will accept as its input the tensor format of $channel \times height \times width$. The height and width of the tensor format corresponds to the time steps and the frequency bins of the time-frequency representation. As for the channel of the tensor format, it corresponds to the number of input variables of a multi-channel signal. For a univariate temporal signal such as heart sound, that will be 1.

4.4.2 CNN-LSTM

It is proposed that the two-dimensional CNN model be used as the feature extractor of the two-dimensional time-frequency data objects. The data format allows both the time-invariant and the frequency-invariant features to be extracted. The use of the two-dimensional CNN as the front end has been verified to be an effective method for time series in previous works [146].

With the two-dimensional CNN at the front end, it is proposed that a one-dimensional CNN be added on top of it to extract the temporal features across the feature maps. This is then followed by an LSTM to extract the remaining high-level temporal features. The proposed CNN-LSTM sub-model for the multi-view temporal ensemble is shown in Figure 4.7 below.

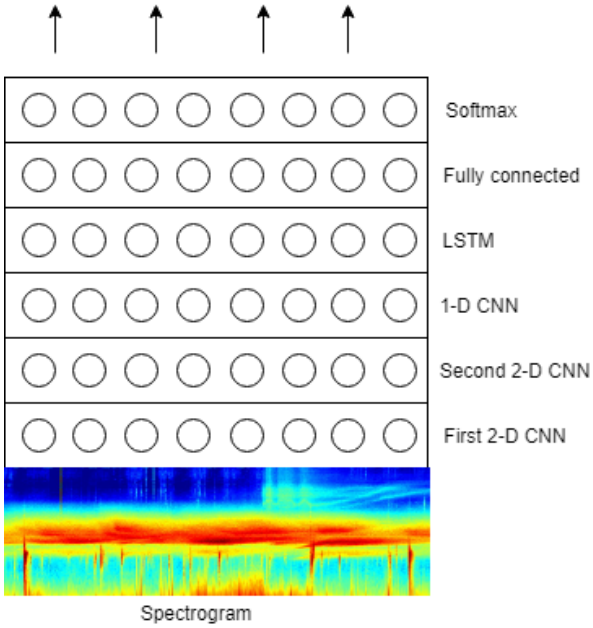

Figure 4.7. Architecture of the CNN-LSTM sub-model for the multi-view temporal ensemble

To extract more complex features, more than one layer should be used in the network. A good starting point is to have two groups of layers for the two-dimensional CNN front end. Each group of two-dimensional CNN consists of the following: (1) a two-dimensional convolutional layer (for detecting features), (3) a pooling layer (for generalization across the channels), and (3) a dropout layer (for further regularization).

Due to the pooling, the number of time steps and features will drop in size at the deeper layers. For example, say the shape of a time series segment is $(21\ time\ steps, 60\ frequency\ bins)$. If a pooling region of $(2,2)$ is used at the pooling layer, then the shape of the feature map will be reduced by the pooling layer to $(10\ time\ steps, 30\ features)$. The smaller data size at the deeper layers is often accompanied by higher number of channels. For example, suppose 32 kernels are used. This will increase the number of channels from 1 to 32, resulting in 32 feature maps, each of $(10\ time\ steps, 30\ features)$ in shape.

The tensor output of the two-dimensional CNN front end can be reshaped to the shape required by one-dimensional CNN and LSTM. In reshaping for these layers, the time-step (i.e. height) dimension in the tensor is retained, but the channel and the feature (i.e. width) are combined into a single dimension.

There are many hyper-parameters that can be tuned in the CNN-LSTM sub-model. Some of the hyper-parameters are: (1) the number and types of layers, (2) the size and stride of the kernels, (3) the pooling region size and the pooling stride, and (3) the number of units in the fully-connected layer.

The final classifier in the LSTM-CNN sub-model consists of one or more fully connected layer and a softmax layer. The penultimate layer of the final classifier is the view of the sub-model. The complementarity of multiple such views can be computed in the multi-view temporal ensemble.

4.5 Summary of Multi-view Temporal Ensemble

The proposed multi-view temporal ensemble provides a way to combine multiple views. To do so, the views are mapped to a global spectral encoding. The contribution of each view to the global spectral encoding is computed. The score, or complementarity, is used as the weight in the linear combination of the views.

The views are the penultimate output of deep learning models, such as the CNN-LSTM model or the deep temporal convolution network.

As an example of the performance improvement by the multi-view temporal ensemble, Table 4.1 below shows the classification accuracies (in percentage) of the 10-fold validation on the 14-channel EEG Eye State data set. Different configurations of the deep temporal convolution network are used for the sub-models in View 1, 2, and 3.

Table 4.1. 10-fold cross-validation of individual views and MTE, eye state

	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	Fold 6	Fold 7	Fold 8	Fold 9	Fold 10
View 1	93.11	93.24	92.03	95.68	91.35	92.57	89.19	82.84	89.59	90.95
View 2	95.41	96.76	89.46	94.59	98.78	88.38	91.76	97.03	91.08	92.70
View 3	89.32	93.11	91.89	99.19	92.43	90.00	94.05	89.86	93.92	92.03
MTE	95.75	95.89	95.75	98.49	98.63	93.42	97.12	96.71	93.42	94.52

The mean of the 10-fold cross-validation is the figure of merit used for comparing the effect of the models. This is shown in Table 4.2 below.

Table 4.2. Mean of 10-fold cross-validation of individual views and MTE, eye state

	Mean	Std Dev
View 1	91.05%	3.44%
View 2	93.59%	3.46%
View 3	92.58%	2.85%
MTE	95.97%	1.84%

As can be seen from Table 4.2 above, the accuracy for the individual models (91.05% for View 1, 93.59% for View 2, and 92.58% for View 3) improves to 95.97% when the views are combined in the multi-view temporal ensemble.

The multi-view temporal ensemble lifts the performance of time series classification at the expense of more training time. This is justified when a more robust performance is required, especially when the time series data set are not well segmented and high variation is expected of the new and unseen test data.

Chapter 5. Data Experiments and Results

This chapter will first describe the data analysis of a typical biosignal data set, the EEG Eye State data set. The objective is to provide the data understanding that is necessary for the data experiments that follow. After that, the following data experiments and their results will be presented.

(1) Deep temporal convolution network (with multi-view temporal ensemble)
    a. EEG Eye State [2]
    b. EEG Epileptic Seizure [15]
    c. Human Activity Recognition [16]
    d. Freezing of Gait during Walking [17]
    e. EMG Lower Limb Analysis [18]
(2) Multi-view temporal ensemble with CNN-LSTM sub-models
    a. Environmental Sound [19]
    b. Heart Sound [20]

The above data experiments will each consist of three parts: (1) spot-checking to provide the benchmark against the proposed method, (2) 10-fold validation of the proposed method, and (3) comparison with an equivalent method and other existing works.

All the codes were run on Matlab 2018a and Python 3.6.5 (with scikit-learn version 0.19.1 and Keras version 2.2.2). The machine used was a HP ProBook 440 G3 laptop with Intel i7-6500U CPU @ 2.5 GHz and Windows 10 Pro.

5.1 Data Analysis of the EEG Eye State Data Set

The EEG Eye State data set is a multivariate numeric time series data set collected by Oliver & Suendermann (2013). It was downloaded from the UCI Machine Learning Repository [147]. It is a relatively simple data set, with a single time series recorded from a single subject. The time series has 14,980 samples in it. Each sample has 14 readings, corresponding to the fourteen electrodes (AF3, F7, F3, FC5, T7, P7, O1, O2, P8, T8, FC6, F4, F8, AF4) of a commercial EEG headset. The data was acquired at 128 sample per second and is 117-second long.

The subject was relaxed but alert during data collection, and so the EEG rhythm is likely to be the alpha waves. He was asked to deliberately open and close his eyes for the data collection. As a result, the EEG signal has two eye states, namely eye-open (class 0) and eye-closed (class 1). The annotation was done manually by comparing the EEG against the video recording of the subject.

5.1.1 Data Exploration

From the line plot of the eye state in Figure 5.1 below, it can be seen that the eye state stays for a few hundred samples (a few seconds) in a state before changing.

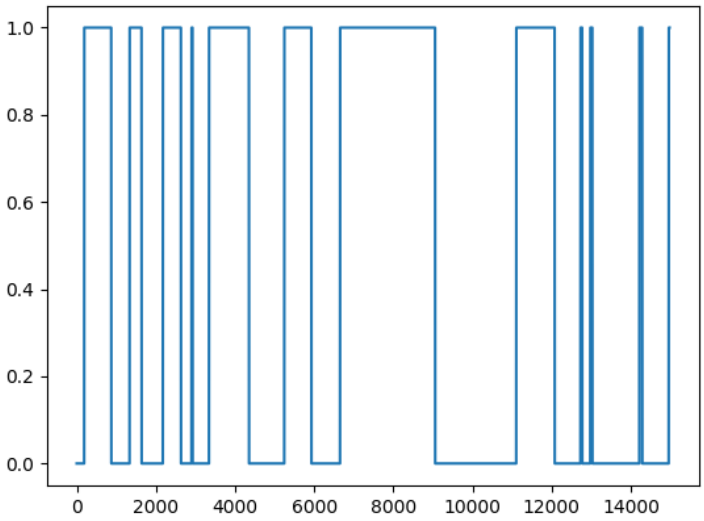

Figure 5.1. Target class labels of an electrode - 0 for eye-open and 1 for eye-closed

At the start of the data experiment, some data cleaning was done to remove the obvious measurement artefacts that may affect model training. This is done by viewing the line plots of the 14 electrodes individually. From the individal line plots, it was found that 4 of the samples contain amplitude that are not in line with the expected alpha wave. They are either too large ($> 5000\ nV$) or too small ($< 3500\ nV$).

After the removal of the 4 outliers, the data set is left with 14,976 samples. It is a balanced data set, with 8,254 eye-open samples (55.12%) and 6,722 eye-closed samples (44.88%). A zero-rule algorithm that always predicts the majority class will yield an accuracy of 55.12% for this data set.

The summary statistics of the electrode signals are shown in Table 5.1 below. Not much can be gleaned about the eye state from the data, other than that the outliers have been removed.

Table 5.1. Summary statistic of the 14 electrodes by class

Eye State	open (class 0)			closed (class 1)		
	min	mean	max	min	mean	max
AF3	4198	4298	4504	4199	4305	4445
F7	3924	4013	4157	3906	4005	4139
F3	4197	4263	4386	4212	4266	4367
FC5	4073	4123	4250	4058	4121	4214
T7	4304	4342	4464	4310	4342	4435
P7	4566	4621	4757	4575	4619	4709
O1	4027	4072	4178	4026	4074	4167
O2	4567	4615	4732	4568	4617	4696
P8	4152	4200	4464	4148	4203	4435
T8	4153	4229	4363	4174	4233	4323
FC6	4027	4200	4178	4026	4205	4167
F4	4201	4277	4398	4226	4282	4369
F8	4443	4602	4834	4510	4611	4811
AF4	4206	4357	4573	4246	4367	4553

The line plots of the cleaned-up signals are shown in Figure 5.2 below. The line plots exhibit the typcial non-stationary, non-linear and noisy look of a biosignal.

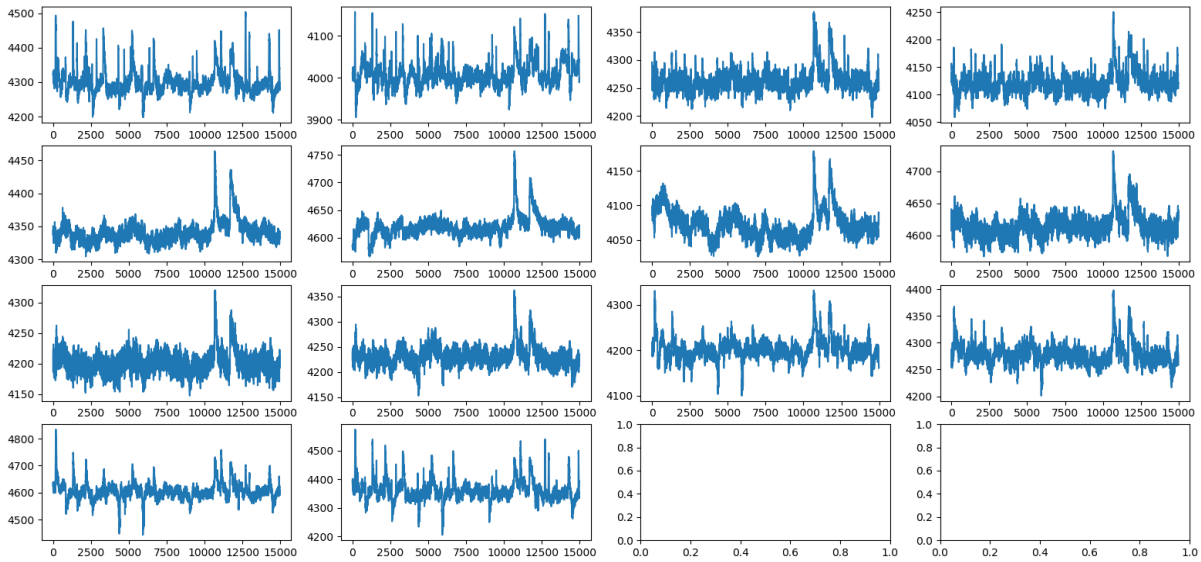

Figure 5.2. Top left to bottom right: AF3, F7, F3, FC5, T7, P7, O1, O2, P8, T8, FC6, F4, F8, AF4

It is known that the most prominent EEG rhythm in a relaxed subject with the eyes closed is the alpha wave. It is also known that the alpha wave will be suppressed when the eyes are open. However, Figure 5.3 below shows that the expected effect is quite illegible even when the electrode signals are placed on top of the annotated eye state for easier cross-reference with the annotation.

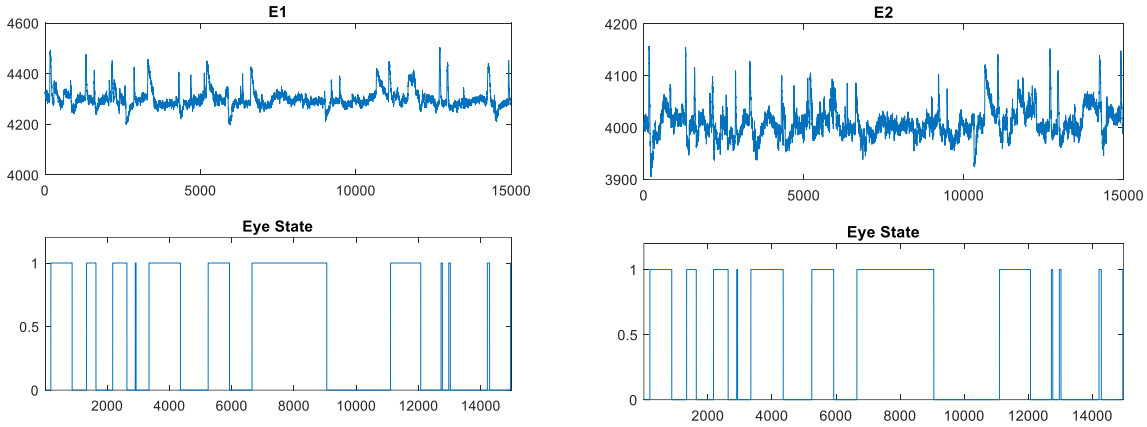

Figure 5.3. AF3 (top left) and F7 (top right), with their corresponding eye state (bottom)

When the electrode signals are arranged in a stack, as shown in Figure 5.4 below, the characteristic of the alpha waves becomes more discernible. It can be seen from Figure 5.4 that

whenever there is a state change from eye-open to eye-closed, the electrode signals will have a spike. Conversely, whenever there is a state change from eye-closed to eye-open, the electrode signals will dip. The general agreement among the electrode signals implies that the electrodes are complementary and will reinforce each other in the classification of the eye state.

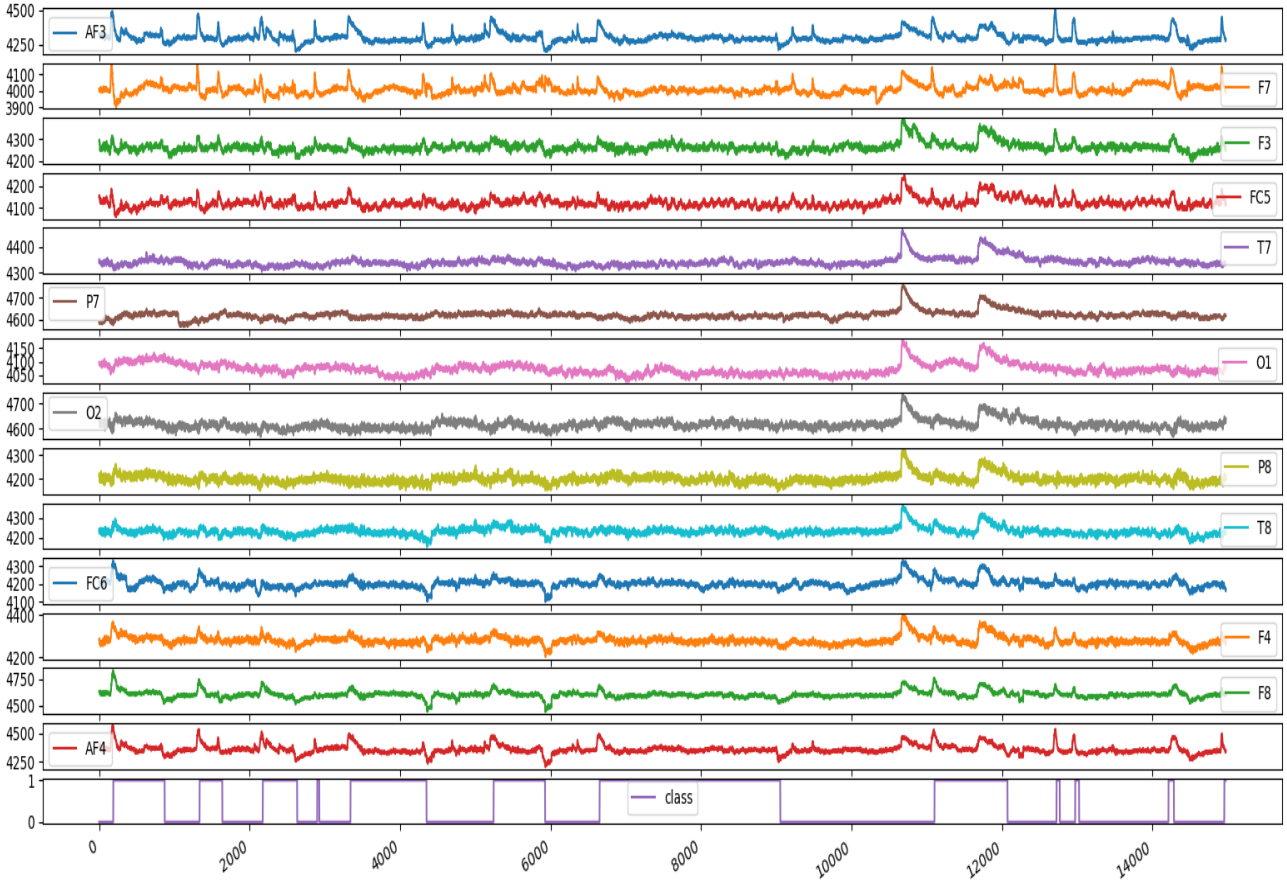

Figure 5.4. Stack of line plots of all the 14 electrodes on top of the eye state

5.1.2 Non-Stationarity

It is known that the EEG signal is non-stationary. This means that the mean and the variance of the signals will change with time. This can be seen in Figure 5.5 below, where the mean and the variance change randomly with time. They do not seem to provide any indication of the eye state.

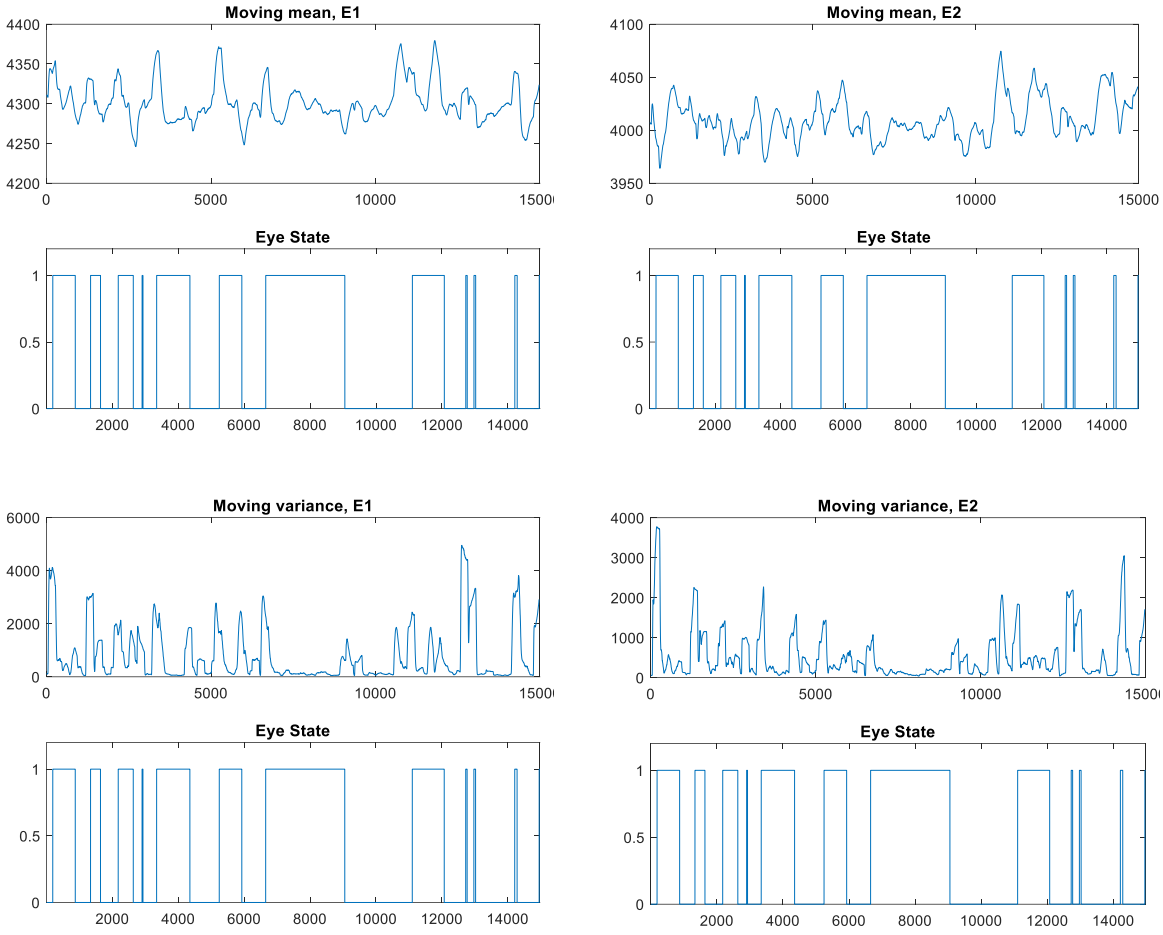

Figure 5.5. AF3 (left) and F7 (right): moving average (top) and moving variance (bottom)

Since the signals are non-stationary, they cannot be Gaussian in distribution. Figure 5.6 below shows the distributions of the 14 electrodes by their amplitude values.

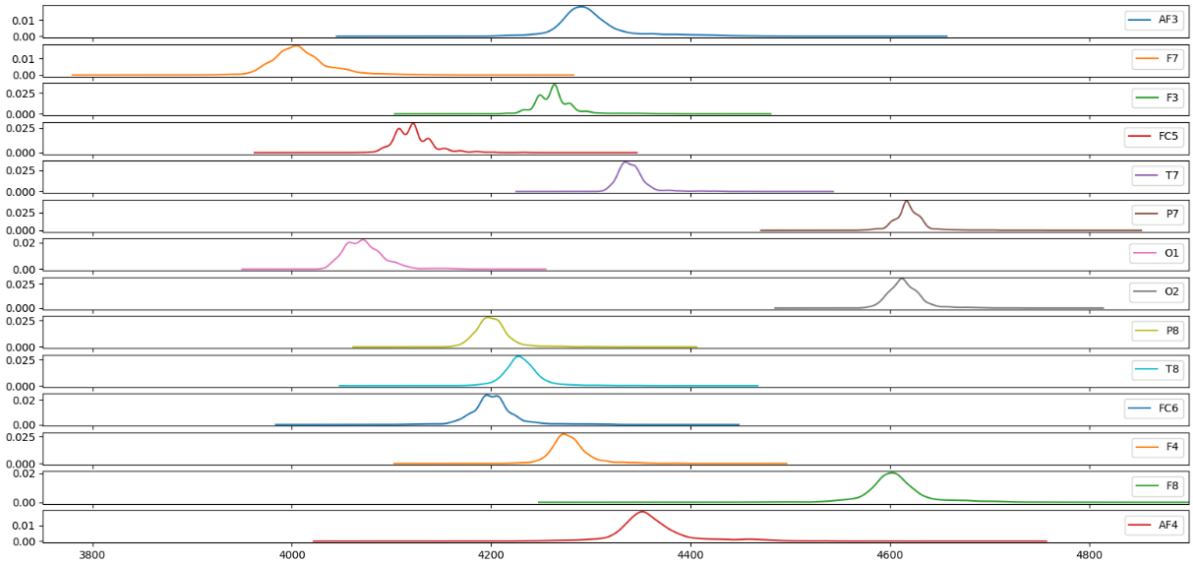

Figure 5.6. Distribution of the 14 electrode signals by amplitude values

The distributions in Figure 5.6 may, at first look, appear Gaussian-like. However, from their QQ plots (graph of the quantiles of the signal against the quantile of the standard normal distribution), it is obvious that there are deviations from the normal distribution. In Figure 5.7, the QQ plots of the signals for class 0 (eye-open) are not Gaussian, as they all have a large positive skew. Similarly, in Figure 5.8, the QQ plots of the signals for class 1 (eye-closed) have a large positive skew too.

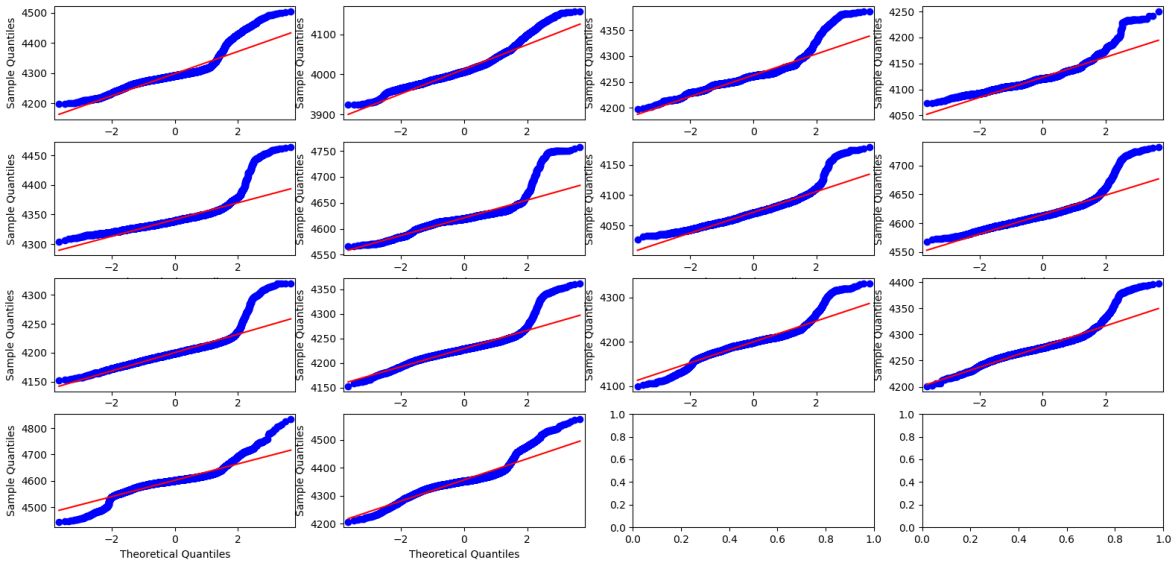

Figure 5.7. QQ plots of the 14 electrode signals, class 0 (eye open)

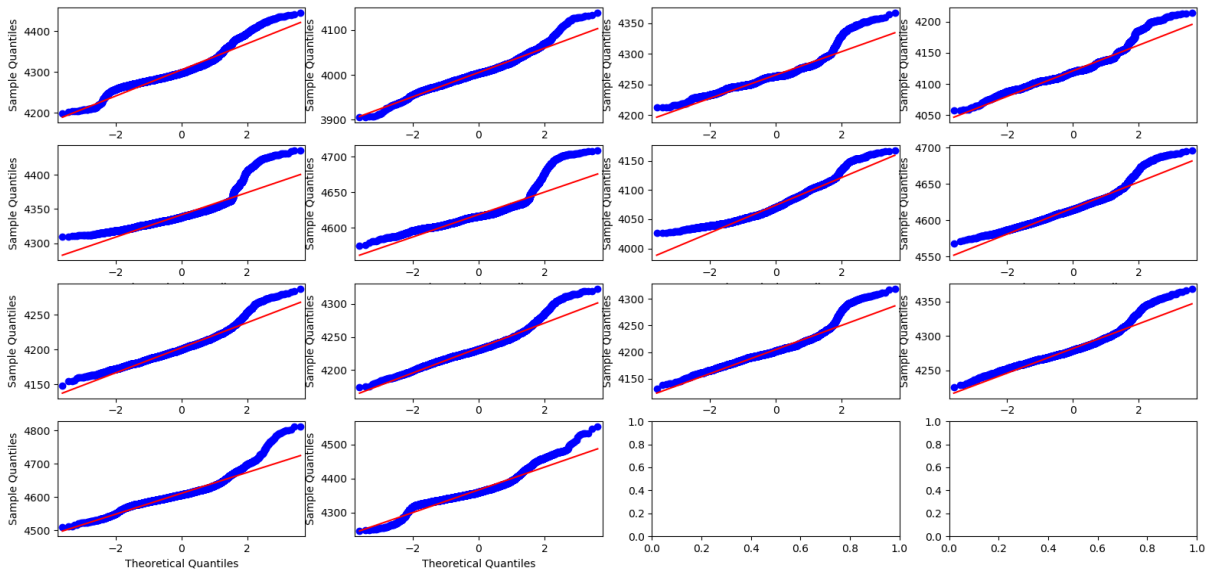

Figure 5.8. QQ plots of the 14 electrode signals, class 1 (eye closed)

The non-stationarity of the electrode signals is confirmed by dividing each of them into a number of short time segments and then perform the Augmented Dickey-Fuller test (a type of statistical test called the unit root test). The null hypothesis of the test is that the signal is non-stationary. A small p-value (typically ≤ 0.05) indicates strong evidence against the null hypothesis.

After conducting the Augmented Dickey-Fuller test on the electrode signals, it was found that the p-values range from 0.5988 to 0.6579. There is thus no evidence to reject the null hypothesis that the signal is non-stationary.

5.1.3 Time Dependency

Although the electrode signals are non-stationary, they do contain temporal patterns that can be learnt by machine learning algorithm. Figure 5.9 below shows the typical autocorrelation and cross-correlation of the electrode signals. As they are not zero in value, the signals cannot be white noise.

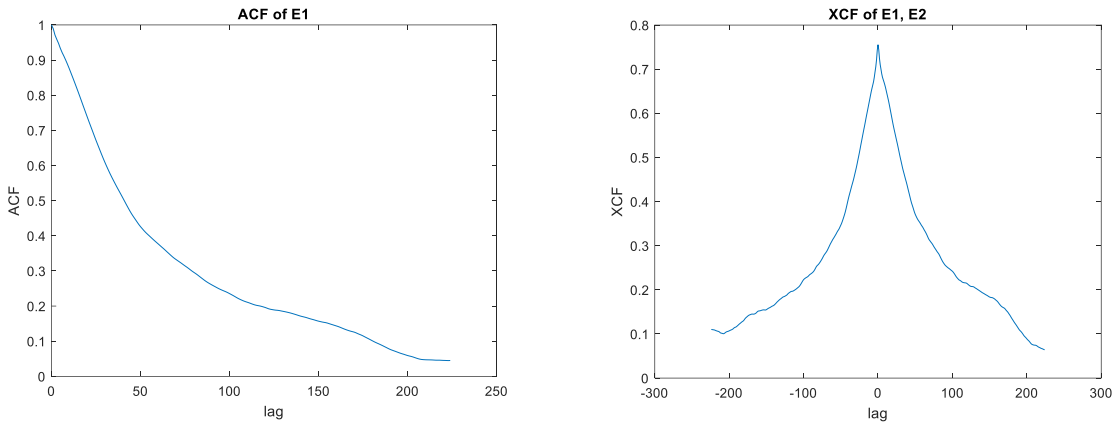

Figure 5.9. Autocorrelation of AF3 (left), and cross-correlation of AF3 and F7 (right)

Another observation of Figure 5.9 above is that the autocorrelation and cross-correlation do not decay exponentially. Instead, they decay in a somewhat linear manner. This is the evidence that the electrode signals have long-term memory to keep the temporal patterns.

From Table 5.2 below, it can be seen that many of the electrodes are positively correlated (>0.3) to one another. They are not independent of each other. This suggests that there is complementary information in the electrode signals. This is not surprising, since they are brain waves (largely alpha waves) that are collected from the same scalp at different locataions.

Table 5.2. Correlation between the electrode signals

	AF3	F7	F3	FC5	T7	P7	O1	O2	P8	T8	FC6	F4	F8	AF4
AF3	1	0.59	0.76	0.61	0.35	0.23	0.32	0.22	0.32	0.50	0.65	0.80	0.76	0.94
F7	0.59	1	0.57	0.75	0.49	0.33	0.26	0.11	0.14	0.18	0.25	0.38	0.25	0.41
F3	0.76	0.57	1	0.77	0.64	0.60	0.49	0.55	0.56	0.64	0.67	0.83	0.62	0.71
FC5	0.61	0.75	0.77	1	0.68	0.54	0.40	0.36	0.37	0.40	0.41	0.56	0.36	0.47
T7	0.35	0.49	0.64	0.68	1	0.83	0.66	0.66	0.64	0.63	0.53	0.53	0.36	0.30
P7	0.23	0.33	0.60	0.54	0.83	1	0.66	0.72	0.71	0.65	0.50	0.51	0.31	0.21
O1	0.32	0.26	0.49	0.40	0.66	0.66	1	0.64	0.67	0.57	0.51	0.58	0.39	0.33
O2	0.22	0.11	0.56	0.36	0.66	0.72	0.64	1	0.87	0.72	0.59	0.59	0.41	0.29
P8	0.32	0.14	0.56	0.37	0.64	0.71	0.67	0.87	1	0.83	0.68	0.67	0.54	0.39
T8	0.50	0.18	0.64	0.40	0.63	0.65	0.57	0.72	0.83	1	0.80	0.76	0.71	0.59
FC6	0.65	0.25	0.67	0.41	0.53	0.50	0.51	0.59	0.68	0.80	1	0.84	0.84	0.74
F4	0.80	0.38	0.83	0.56	0.53	0.51	0.58	0.59	0.67	0.76	0.84	1	0.82	0.84
F8	0.76	0.25	0.62	0.36	0.36	0.31	0.39	0.41	0.54	0.71	0.84	0.82	1	0.86
AF4	0.94	0.41	0.71	0.47	0.30	0.21	0.33	0.29	0.39	0.59	0.74	0.84	0.86	1

The heat map of the correlation is shown in Figure 5.10 below:

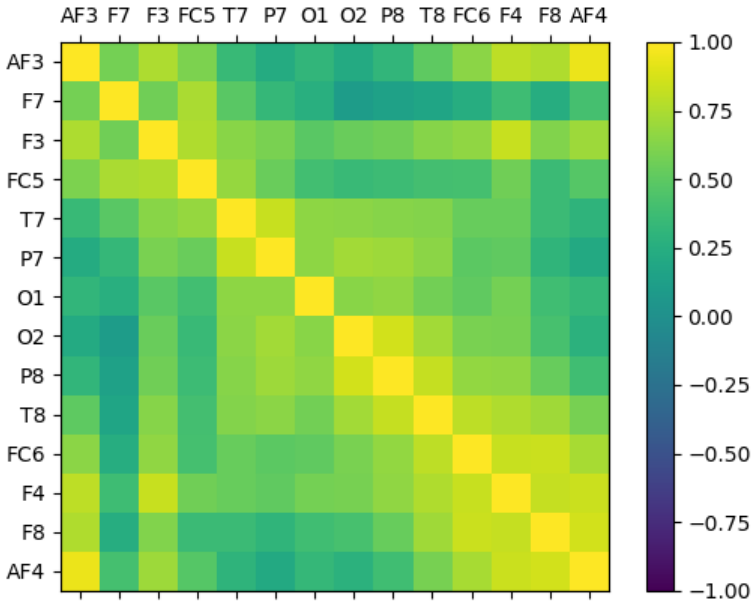

Figure 5.10. Heat map of the correlation of the electrode signals

The cross-correlation shows that the values of the electrodes tend to move together. They are co-integrated, meaning that they share a common stochastic drift in the long run. The test for co-integration using the Engle-Granger Test produces a p-value of 0.001 with the full set of 14,976 samples, thus rejecting the null hypothesis that there is no co-integration.

From the autocorrelation plot in Figure 5.11 below, it can be seen that there is significant autocorrelation over a large range of lags (more than a few thousands samples). The alternation between positive and negative correlation reflects the changing pattern of the eye state.

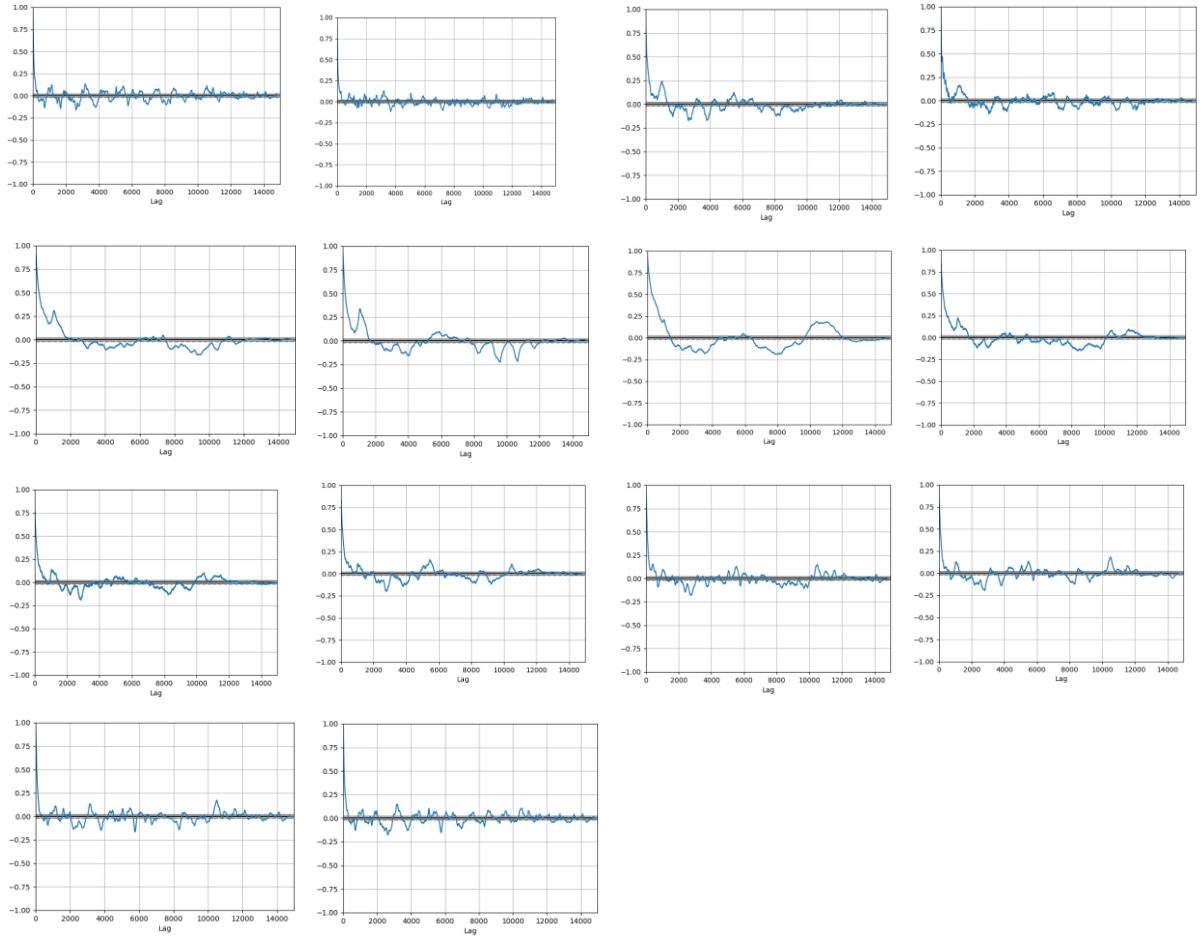

Figure 5.11. Top-left to bottom-right: auto-correlation plots of 14 electrodes (AF3, F7, F3, FC5, T7, P7, O1, O2, P8, T8, FC6, F4, F8, AF4)

Despite the autocorrelation, the signals look random. Each of the values are built from the previous value with an aditional amount of white noise. This suggests a Markov process (random walk) where the values are autocorrelated, but the direction of change are random. For

such a signal, the direction cannot be reasonably predicted. The best prediction is persistence, which is to say that the value at the previous time step is the best predictor for the next time step.

A short run of a random walk signal can be converted to a stationary signal by taking the first difference of the samples. Figure 5.12 shows the autocorrelation and cross-correlation of the first differences, showing clearly that the signal is now stationary. The p-value of the Augmented Dickey-Fuller test is now 0.001.

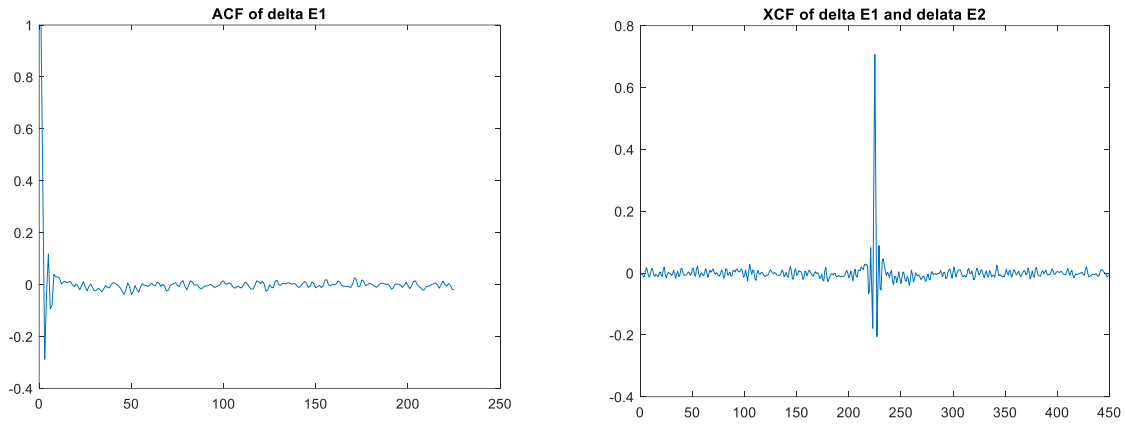

Figure 5.12. (Left) Autocorrelation of first difference of AF3. (Right) Cross-correlation of the first difference of AF3 and F3

When the first difference is split according to the eye states (eye-open or eye-closed), the QQ plots appears nonlinear for eye-open but linear for eye-closed (except for some odd samples).

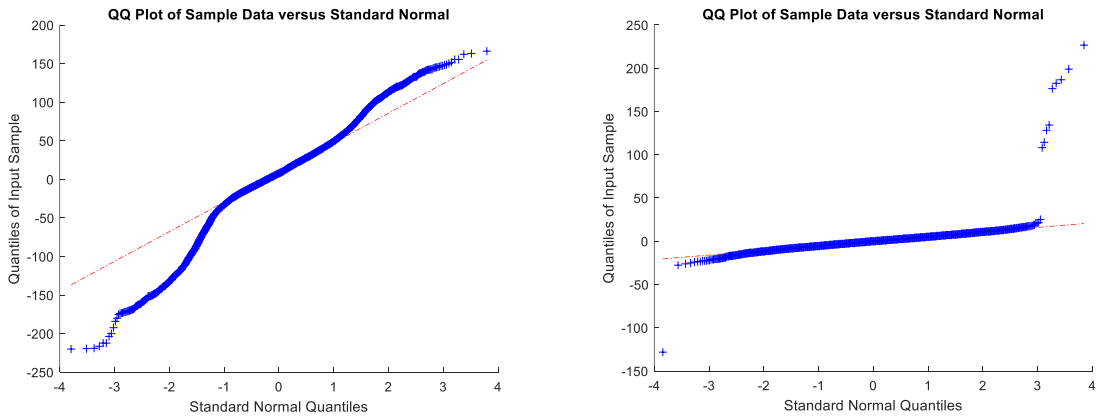

Figure 5.13. Quantile-to-quantile plot of AF3's first difference, class 0 (left) & class 1 (right)

The difference in the distributions by eye state is also evident from the kernel density estimation of the first difference by class, as shown in Figure 5.14 below.

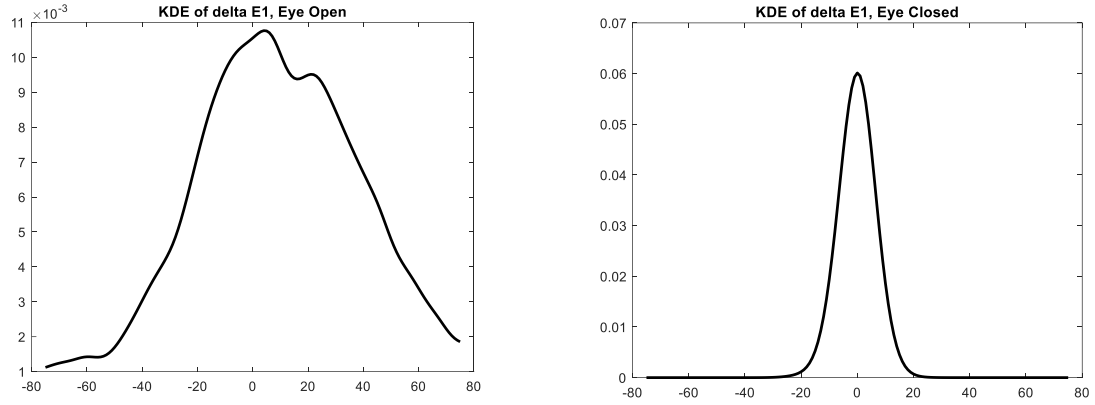

Figure 5.14. Kernel density estimation of AF's first difference, class 0 (left) & class 1 (right)

The difference in the EEG signal by eye states provides the discriminative information for the signals to be classified. However, as the difference is too complex to be described by a general form, deep learning should be used to fit the data to an approximate function that represents the underlying function.

5.1.4 Effect of the Sliding Window

Short runs of a random walk signal can be created from the time series by the sliding window method. Each of these segments is a row in the resulting table. Due to the sliding window that breaks the time series into jagged segments, the columnar data in the resulting table will not correlate to each other and so there is no issue of multi-collinearity there.

As shown in Figure 5.15 below, due to the sliding window, the data of a single column in the table do not autocorrelate anymore. The data of two different columns in the table do not cross-correlate also.

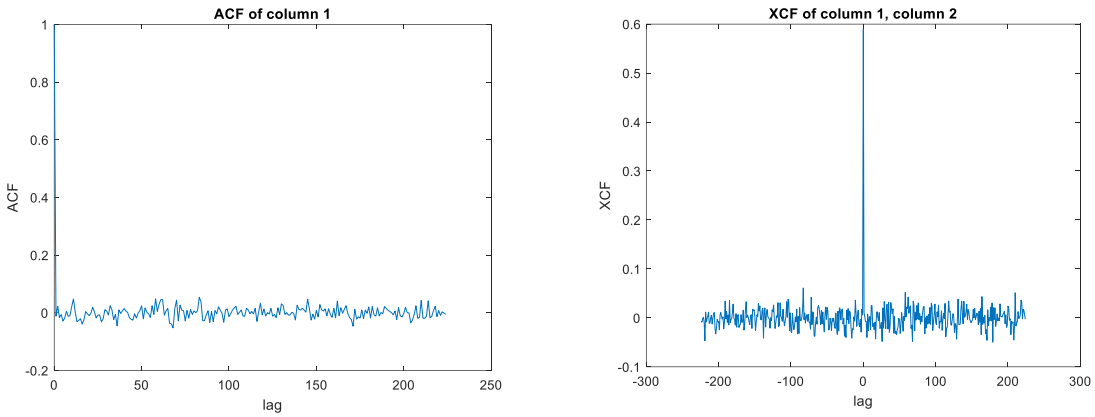

Figure 5.15. Autocorrelation of column 1 (left), and cross-correlation of column 1 & 2 (right)

With row shuffling, the data instances in the table will become independent and identically distributed. This is the kind of distribution expected by a typical machine learning algorithm. Machine learning may now proceed, now that the time series is put in the structured format as expected by the algorithm.

5.2 EEG Eye State

This section will describe the data experiment done on the EEG Eye State data set [2] from the UCI Machine Learning Repository [147]. The purpose is to validate the performance of the deep temporal convolution network and the multi-view temporal ensemble on the data set.

The work is presented in five parts: (1) spot-checking to get the general benchmark, (2) the 10-fold validation of the deep temporal convolution network with TS (time steps) value of 1, 2, and 5, (3) the comparison with a DBN-DNN of equal complexity, (4) the comparison of the individual views with the multi-view temporal ensemble, and (5) comparison with exsiting works.

5.2.1 Spot Checking

With the EEG Eye State data set, 10-fold cross-validation was done in Python with five different algorithms, namely (1) logistic regression, (2) K nearest neighbour, (3) CART decision tree, (4) MLP neural network, and (5) ensemble by voting. All the algorithms were run in their default configurations. The data attributes were standardized in each of the folds during cross-validation.

Two different data formats were used for the 10-fold validations. This is shown in Table 5.3 below as “without windowing” and “with windowing” (window length as 16, slide as 8). In both cases, the data were in their natural time order, i.e. no shuffling was done. Both data formats yield rather poor classification accuracy scores and standard deviation (in brackets), as shown in Table 5.3 below.

Table 5.3. Classification accuracy without shuffling (eye state)

	without windowing	with windowing	p-value
LR	37.76% (19.38%)	36.44% (15.87%)	0.563
KNN	51.03% (13.33%)	53.81% (16.29%)	0.204
CART	50.67% (8.90%)	52.05% (8.48%)	0.516
MLP	50.38% (14.36%)	53.78% (17.48%)	0.453
Ensemble by Voting	50.84% (18.39%)	51.24% (19.89%)	0.817

As seen from Table 5.3 above, the accuracy is close to the random chance of 55.12%. There was no effective learning. The high p-value of the Student’s paired t-test (between the 10-fold validation results of the windowed and non-windowed data) shows that there is no benefit in using the time delay representation if the data was not shuffled before training.

The reason for the poor performance is not the lack of temporal patterns in the input. It is the overfitting of the classifier to the output pattern (eye state) in the training set. This results in poor generalization in the test set. Figure 5.16 shows the box plot of the test results of the five algorithms for the windowed data.

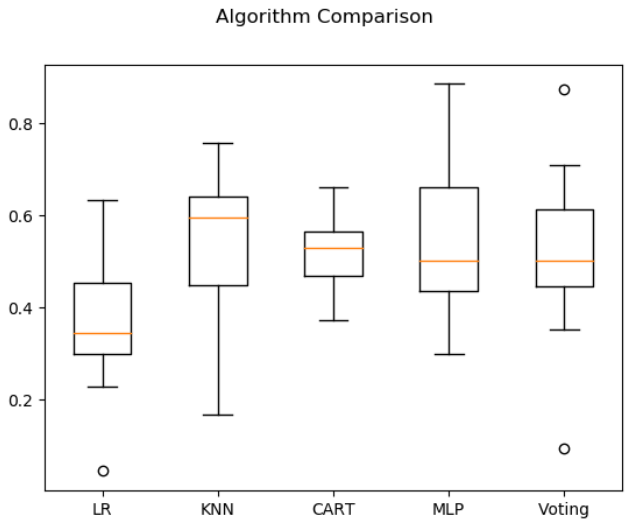

Figure 5.16. Boxplot of algorithm comparison without shuffling (eye state)

With the importance of shuffling confirmed, the spot checking is done again, this time with shuffling. The performance, as shown in Table 5.4 below, is now significantly better. In particular, the MLP in the default configuration (115 nodes in the hidden layer) achieved a classification accuracy of 95.2% without windowing and 97.4% with windowing.

Table 5.4. Classification accuracy with shuffling (eye state)

	Without Windowing	With Windowing	p-value
LR	64.102% (0.771)	62.202% (2.141)	0.058
KNN	96.347% (0.413)	95.803% (1.023)	0.148
CART	83.927% (1.034)	75.675% (2.597)	0.000
MLP	95.199% (0.503)	97.433% (1.117)	0.001
Ensemble by Voting	92.208% (0.707)	90.189% (2.108)	0.022

The low p-values in Table 5.4 above shows that the time delay representation does affect the performance of the classifiers. The box plot in Figure 5.17 provides a visual comparison of the performances of the classifiers.

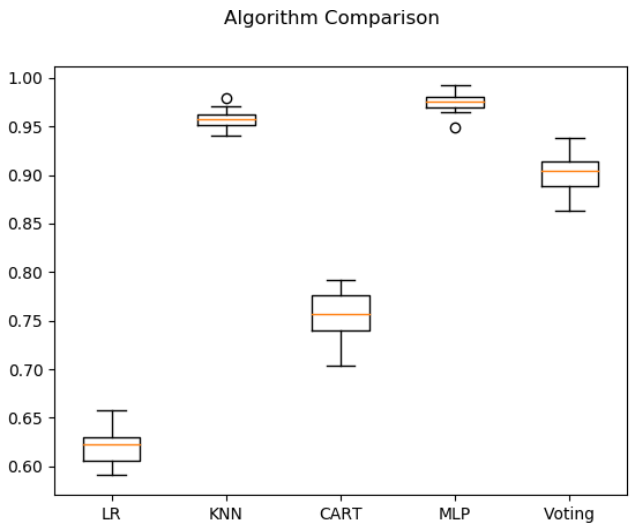

Figure 5.17. Boxplot of algorithm comparison with shuffling (eye state)

To confirm the results obtained so far, the data set (with and without windowing, both with shuffling) was used with the WEKA toolkit based on the default setting for the classifiers. The 10-fold cross-validation results are shown in Table 5.5 below.

Table 5.5. WEKA classification accuracy with shuffling (eye state)

	Without Windowing	With Windowing
LR	64.1%	61.1%
KNN	97.3%	96.0%
CART	83.2%	74.4%
MLP	85.1%	96.3%
KStar	96.7%	55.2%

The difference in the performance score in Table 5.4 and Table 5.5 is attributed to the differences in (1) the default setting and (2) the implementation of the algorithms in Python and WEKA. By and large, it confirms that shuffling is necessary. In the case of MLP, windowing will improve the performance score.

5.2.2 10-Fold Validation, TS = 1, 2, 5

In this data experiment, three different configurations of the deep temporal convolution network were used. They are shown in Table 5.6 below as View 1, View 2, and View 3. These are the sub-models of the multi-view temporal ensemble.

Table 5.6. Configurations of the deep temporal convolution network

	View 1	View 2	View 3
Input layer	224	224	224
First hidden layer	20	20	20
Second hidden layer	20	50	35
Third hidden layer	20	20	20
Softmax layer	2	2	2

The three configurations were based on a network structure that consists of one input layer, three hidden layers and the final softmax layer. Their differences lie in the number of nodes in the second hidden layer (20 for View 1, 50 for View 2, and 35 for View 3).

The data was rearranged in the time delay representation by the sliding window method. The window length was 16-sample long (125 millisecond), and the slide was 8-sample long.

The hyper-parameter TS (time steps) was set initially to 1. $TS = 1$ implies that there is no concatenation in the concatenation sublayer. It is equivalent to the usual DBN-DNN. The performance based on this value is thus used as the benchmark for the other time step values.

Table 5.7 below shows the classification accuracy for the three configurations of the deep temporal convolution network at $TS = 1$, as well as the classficiation accuracy of the multi-view temporal ensemble based on the three views. The results are arranged in 10 folds so that the fluctuation across the folds can be seen.

Table 5.7. Cross-validation results (accuracies in percentage) of DTCN at TS=1 (eye state)

	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	Fold 6	Fold 7	Fold 8	Fold 9	Fold 10
View 1	93.11	93.24	92.03	95.68	91.35	92.57	89.19	82.84	89.59	90.95
View 2	95.41	96.76	89.46	94.59	98.78	88.38	91.76	97.03	91.08	92.70
View 3	89.32	93.11	91.89	99.19	92.43	90.00	94.05	89.86	93.92	92.03
MTE	95.75	95.89	95.75	98.49	98.63	93.42	97.12	96.71	93.42	94.52

The 10-fold results in Table 5.7 above is summarized in Table 5.8 below. It shows the means and standard deviations of the 10-fold results for each of the three views, as well as the combined view provided by the multi-view temporal ensemble.

Table 5.8. Means and standard deviations of the 10-fold results of DTCN at TS=1 (eye state)

	Mean	Std Dev
View1	91.05%	3.44%
View2	93.59%	3.46%
View3	92.58%	2.85%
MTE	95.97%	1.84%

To show the improvement due to the deep temporal convolution network, the hyper-parameter value was then set to $TS = 2$. The cross-validation results are shown in Table 5.9 below, and the summarized results in Table 5.10 further below that.

Table 5.9. Cross-validation results (accuracies in percentage) for DTCN at TS=2 (eye state)

	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	Fold 6	Fold 7	Fold8	Fold 9	Fold 10
View 1	93.51	95.81	99.46	99.86	98.92	91.08	94.73	94.32	90.54	92.03
View 2	98.78	98.24	99.86	97.97	98.51	99.46	99.46	97.84	95.41	97.97
View 3	91.89	98.11	96.76	100.0	98.78	89.73	100.0	97.97	97.57	98.2
MTE	96.99	97.81	99.32	99.45	99.04	97.53	99.73	98.22	96.30	97.26

Table 5.10. Means and standard deviations of the 10-fold results of DTCN at TS=2 (eye state)

	Mean	Std Dev
View 1	95.03%	3.44%
View 2	98.35%	1.26%
View 3	96.91%	3.40%
MTE	98.16%	1.17%

As Table 5.10 above shows, there is a lift in the classification accuracies for all the three configurations of the deep temporal convolution network, as well as the multi-view temporal ensemble. For example, the classification accuracy of View 1 has improved from 91.05% at $TS = 1$ to 95.03% at $TS = 2$. This imporvement is attributed to the use of more temporal context in the deep temporal convolution network.

Since there is an imporvement in performance by increasing the TS value from 1 to 2, why not take it further to $TS = 5$? The cross-validation results for this case is shown in Table 5.11 below.

Table 5.11. Cross-validation results (accuracies in percentage) of DTCN at TS=5 (eye state)

	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	Fold 6	Fold 7	Fold 8	Fold 9	Fold 10
View 1	99.05	98.92	100.0	99.86	99.46	98.92	98.51	99.86	99.19	98.92
View 2	98.92	98.51	100.0	99.86	100.0	100.0	97.16	98.24	98.78	99.73
View 3	100.0	99.46	100.0	98.51	99.46	99.73	99.73	99.46	99.73	100.0
MTE	99.04	98.36	99.86	99.04	99.73	99.86	99.59	99.59	99.04	99.59

The 10-fold results in Table 5.11 above is summarized in Table 5.12 below.

Table 5.12. Means and standard deviations of 10-fold results of DTCN at TS=5 (eye state)

	Mean	Std Dev
View 1	99.27%	0.50%
View 2	99.12%	0.96%
View 3	99.61%	0.44%
MTE	99.37%	0.48%

Table 5.12 above shows that there is indeed a lift in performance when the TS value is increased from 2 to 5. For example, the classification accuracy of View 1 was 91.05% at $TS = 1$, 95.03% at $TS = 2$, and 99.27% at $TS = 5$.

The results in Table 5.10 and Table 5.12 confirm the hypothesis that the concatenation of features at the deeper layers will provide the temporal context that helps with the extraction of time-invariant features for better discrimination by the final classifier. The implementation of the necessary modification (for example, short-term temporal order, shuffled mini-batches, gradient routing, etc.) in the deep temporal convolution network is proven effective with the EEG Eye State data set. These results are visualized in Figure 5.18 below.

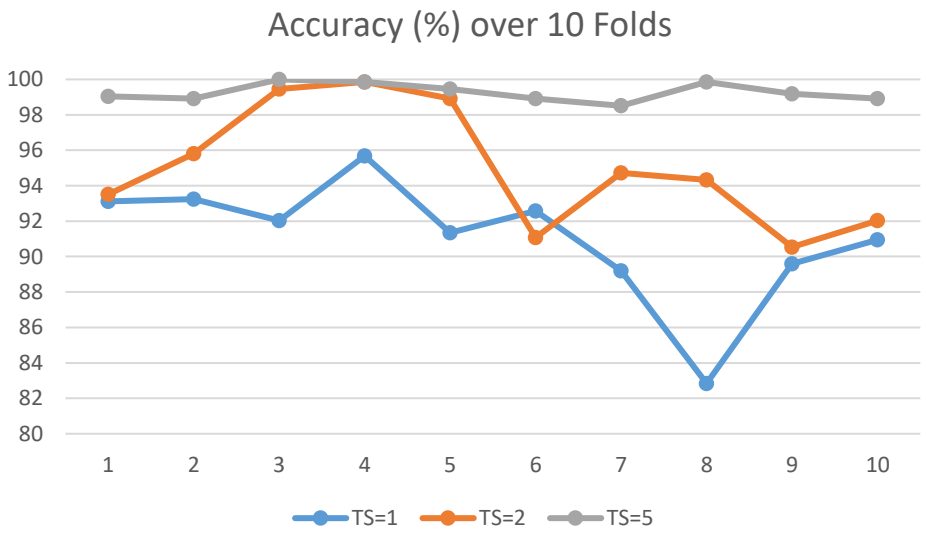

Figure 5.18. Classification accuracy over 10 folds, DTCN, View 1, $TS = 1,2$ and 5 (eye state)

Two observations can be made about Figure 5.18 above: (1) the curve for $TS = 5$ is higher, compared to the other two curves, and (2) there is less fluctuation in the curve for $TS = 5$, compared to the other two curves. This means that the bias is reduced (i.e. the classification accuracy has improved) and the variance is reduced (less overfitting to the noise in the data). It can thus be concluded that there is an improvement in the generalization performance of the deep temporal convolution network.

5.2.3 Comparing with Equivalent DBN-DNN

To confirm the improvement of the proposed deep temporal convolution network over a DBN-DNN, the equivalent DBN-DNN (in terms of complexity) will be used for the comparison. Here, complexity refers to the number of adjustable parameters in the network.

Two sets of comparisons are made, one at $TS = 2$ and another one at $TS = 5$:

(1) Compare the deep temporal convolution network at $TS = 2$ in three different configurations, View 1, View 2, and View 3, with the equivalent DBN-DNNs.
(2) Compare the deep temporal convolution network at $TS = 5$ in three different configurations, View 1, View 2, and View 3, with the equivalent DBN-DNNs.

The three configurations, View 1, View 2, and View 3, are the same as those shown in Table 5.6 earlier on. The number of adjustable parameters for these three configurations, at $TS = 1$, 2, and 5, are as shown in Table 5.13 below.

Table 5.13. No. of adjustable parameters, DTCN, View 1, 2, and 3, at TS=1, 2, and 3

	View 1	View 2	View 3
$TS = 1$	5,320	6,520	5,920
$TS = 2$	6,120	8,520	7,320
$TS = 5$	8,520	14,520	11,520

Many DBN-DNNs would have about the same number of adjustable parameters as Table 5.13 above. For the sake of comparison, the configurations in Table 5.14 below were used as the equivalent of View 1, View 2, and View 3 at $TS = 2$.

Table 5.14. Equivalent DBN-DNN of View 1, View 2, and View 3 at TS=2 (eye state)

	View 1	View 2	View 3
Input layer	224	224	224
First hidden layer	23	31	27
Second hidden layer	23	31	27
Third hidden layer	20	20	20
Softmax layer	2	2	2
Total No. of Parameters	6,181	8,565	7,357

In a similar vein, the configurations in Table 5.15 below were used as the equivalent of View 1, View 2, and View 3 at $TS = 5$.

Table 5.15. Equivalent DBN-DNN of View 1, View 2, and View 3 at TS=5 (eye state)

	View 1	View 2	View 3
Input layer	224	224	224
First hidden layer	31	50	40
Second hidden layer	31	50	40
Third hidden layer	20	20	20
Softmax layer	2	2	2
Total No. of Parameters	8,565	14,740	11,400

Using the equivalent configurations of $TS = 2$ in Table 5.14 shown earlier on, the 10-fold validation results were obtained and are shown in Table 5.16 below. The statistics of the 10-fold validation results are shown in Table 5.17 further below that.

Table 5.16. Cross-validation results (accuracies in percentage) of equivalent DBN-DNN at TS=2 (eye state)

Equi TS=2	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	Fold 6	Fold 7	Fold 8	Fold 9	Fold 10
View 1	94.46	92.84	96.49	92.70	88.78	88.24	95.95	88.65	94.32	83.92
View 2	99.32	97.84	89.59	94.19	95.0	98.11	93.51	90.95	90.54	72.84
View 3	89.73	96.35	96.77	93.78	91.35	88.65	89.86	88.24	95.14	95.14
MTE	98.49	97.81	98.22	95.07	95.62	97.81	96.71	92.60	95.62	95.34

Table 5.17. Means and standard deviations of 10-fold results of equivalent DBN-DNN at TS=2 (eye state)

	Mean	Std Dev
View 1	91.64%	4.06%
View 2	92.19%	7.58%
View 3	92.50%	3.29%
MTE	96.33%	1.83%

By comparing Table 5.16 (equivalent DBN-DNN at $TS = 2$) with Table 5.9 (deep temporal convolution network at $TS = 2$) and then plotting the mean results as a bar chart in Figure 5.19 below, it can be seen that the deep temporal convolution network exhibits higher classification accuracies than its DBN-DNN equivalent across all the three views. The fluctuation across the folds (the variance) is smaller in the case of the deep temporal convolution network also.

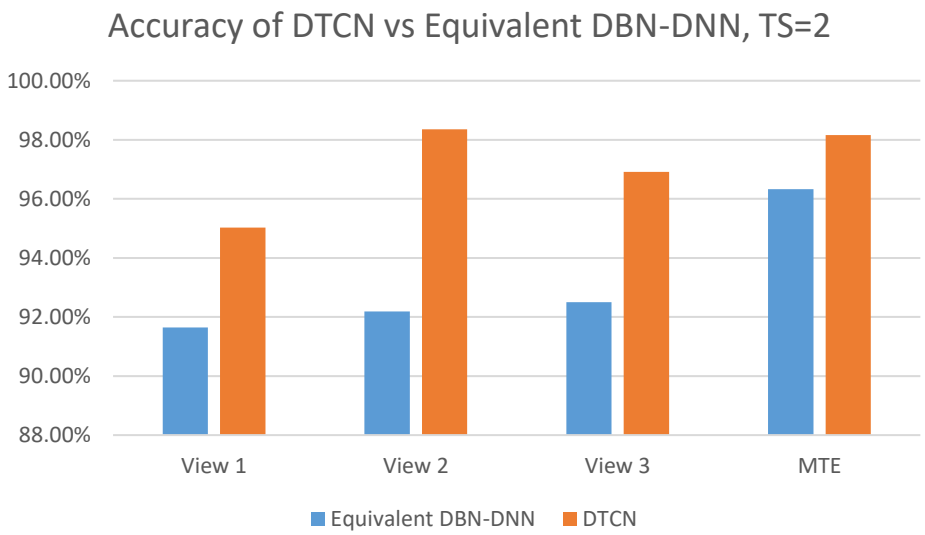

Figure 5.19. Performance of DTCN vs equivalent DBN-DNN, $TS = 2$ (eye state)

Using the equivalent configurations of $TS = 5$ in Table 5.15 shown earlier on, the 10-fold validation results were obtained and are shown in Table 5.18 below. The statistics of the 10-fold validation results are shown in Table 5.19 further below that.

Table 5.18. Cross-validation results (accuracies in percentage) of equivalent DBN-DNN at TS=5 (eye state)

Equi TS=5	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	Fold 6	Fold 7	Fold 8	Fold 9	Fold 10
View 1	99.32	88.65	92.84	99.46	94.32	83.65	73.78	95.68	98.78	97.30
View 2	97.02	97.16	99.73	95.54	98.65	89.19	99.46	98.51	98.92	96.49
View 3	90	97.03	95.95	98.65	94.73	98.11	99.0	94.86	97.30	97.84
MTE	97.67	96.44	99.59	99.04	98.36	94.93	98.90	96.58	98.22	97.26

Table 5.19. Means and standard deviations of 10-fold results of equivalent DBN-DNN at TS=5 (eye state)

	Mean	Std Dev
View 1	92.38%	8.26%
View 2	97.07%	3.08%
View 3	96.35%	2.68%
MTE	97.70%	1.42%

By comparing Table 5.18 (equivalent DBN-DNN at $TS = 5$) with Table 5.11 (deep temporal convolution network at $TS = 5$) and then plotting the mean results as a bar chart as shown in Figure 5.20 below, it can be seen that deep temporal convolution exhibits higher classification accuracies than its DBN-DNN equivalent across all the three views. The fluctuation across the folds (the variance) is smaller in the case of the deep temporal convolution network also.

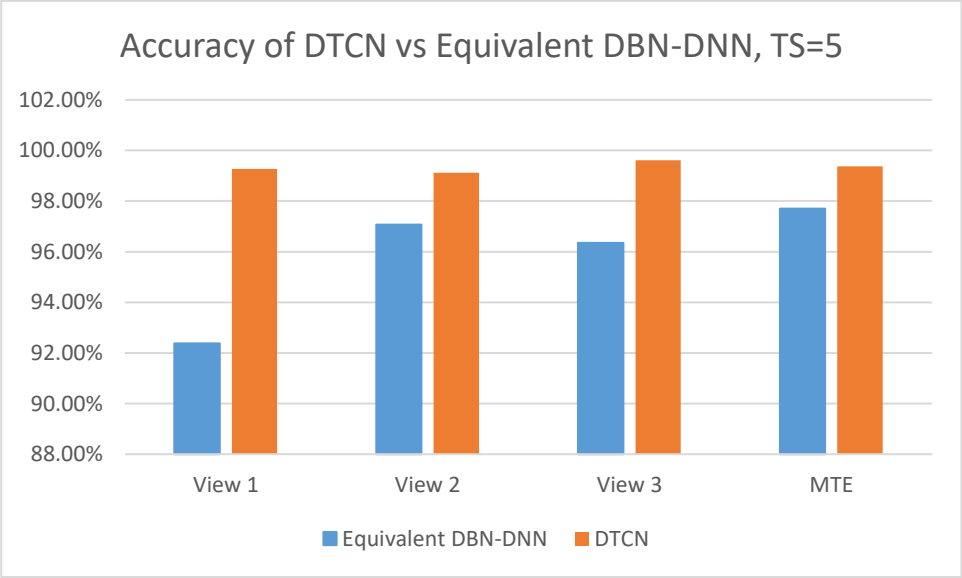

Figure 5.20. Performance of DTCN vs equivalent DBN-DNN, $TS = 5$ (eye state)

5.2.4 Performance Improvement with Ensemble

Two ensembles are used for the comparison here, namely the proposed multi-view temporal ensemble and the ensemble average of the individual views. Figure 5.21 below shows the the classification accuracies over 10 folds for the three different views (all at $TS = 1$). It is found

that not only is the ensemble's accuracy higher than the individual views, it is higher than the ensemble by averging also. Also, the variance is reduced by the multi-view temporal ensemble.

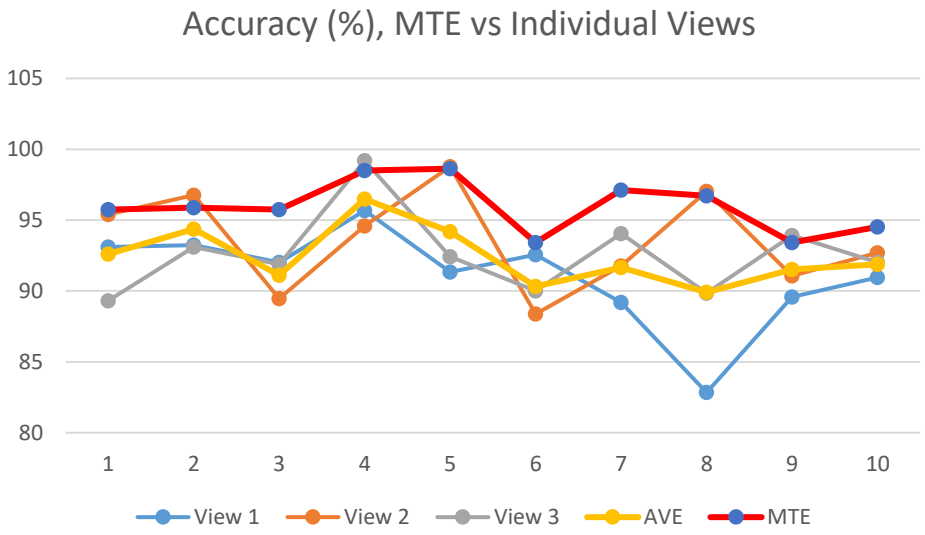

Figure 5.21. Validation result for View 1, View 2, View 3 and their ensembles at $TS = 1$ (eye state)

The same improvement in bias and variance can be seen in the multi-view temporal ensemble at $TS = 2$ and 5 also. These were shown in the bar charts in Figure 5.19 and Figure 5.20 earlier on.

In summary, this data experiment has shown that the proposed deep temporal convolution network is able to produce better generalization performance than its equivalent DBN-DNN. It has also shown that the multi-view temporal ensemble is able to give a lift to the individual views and is better than ensembling by averaging.

5.2.5 Comparison with Existing Works

Oliver et al. [2] achieved an accuracy of 97.0% on the ECG Eye State data set using the instance-based learner KStar [148]. As an instance-based classifier, KStar classifies an instance by comparing the distance of the instance from all the instances in the training set. As a result, it has a long runtime during actual deployment. It utilised only the spatial information in the multichannel EEG data and not the temporal context in the signals.

Wang et al. [149] used incremental attribute learning (IAL) on time series data and then train them using neural network. Feature extraction and feature ordering are carried out before training. The features are based on twelve EEG samples, or 93.75 milliseconds. The features are then imported, one by one, into the network for training. A classification accuracy of 72.61% was reported. This is an improvement over conventional batch-training method using the neural network, which produces a classification accuracy of 69.37%.

For the EEG signals, given that temporal context can provide much more information, the deep learning approach proposed in this thesis is a better approach. Each of the data instances used by the deep temporal convolution is 16 EEG samples, or 125 milliseconds. This is only slightly longer than the 12 samples used by Wang et al. in the aforementioned work. At $TS = 2$, the deep temporal convolution network has a classification accuracy of 96.76%, which is quite comparable to the classification accuracy of 97.0% achieved by the KStar method. The classification accuracy is generally above 99.0% when at $TS = 5$. This performance is achieved without the drawback of long runtime during deployment of the KStar method.

5.3 EEG Epileptic

This section will describe the data experiment done on the EEG Epileptic data set [15] from University Hospital Bonn, Germany and made available at the UCI Machine Learning Repository [147]. The purpose is to validate the performance of the deep temporal convoltion network and multi-view temporal ensemble on the data set.

The work is presented in six parts: (1) description of the data set, (2) spot-checking to get the general benchmark, (3) 10-fold validation of the deep temporal convolution network with TS (time steps) value of 1, 2, and 5, (4) comparison with a DBN-DNN of equal complexity, (5) comparison of the individual views with the multi-view temporal ensemble, and (6) comparison with existing works.

5.3.1 Data Set

The EEG Epileptic data set is a univariate numeric time series data set. There are five sets of EEG files in the data set. Each set corresponds to a target class and has 100 files in it. Each file belongs to a particular subject. Each subject has five files, one for each target class. In total, there are 500 files from 100 subjects.

In each file, there are 4,097 samples (23.6 seconds). They are divided into 23 chunks, arranged in natural time order one after another. Each chunk has 178 samples (1 second) in it.

In total, the data set has $23 \times 500 = 11{,}500$ chunks. They are arranged in a tensor with the following shape: $(readings = 178, chunks = 23, subjects = 100, sets = 5)$. The first $178 \times 23 \times 100$ readings belong to class 5, the next $178 \times 23 \times 100$ readings belongs to class 4, and so on. Each of the file is associated with a target class $y \in \{1,2,\cdots,5\}$. The description of these classes is in Table 5.20 below.

Table 5.20. Description of the 5 classes, EEG epileptic seizure

Class Category	Description
1	Epilepsy patient during seizure
2	Epilepsy patient during seizure free interval from the tumour region
3	Epilepsy patient during seizure free interval from the healthy brain area
4	Healthy subject with eyes closed
5	Healthy subject with eyes open

Unlike the EEG Eye State data set, which is the signal of a single subject, multiple subjects exist in the EEG Epileptic data set. In this data experiment, they exist as latent modes and are not distinguished in the target class labels.

All subjects in classes 2, 3, 4, and 5 did not have epileptic seizure. Only subjects in class 1 have epileptic seizure. Although there are 5 classes in the data set, most researchers did only binary classification, namely class 1 (epileptic seizure) against the rest.

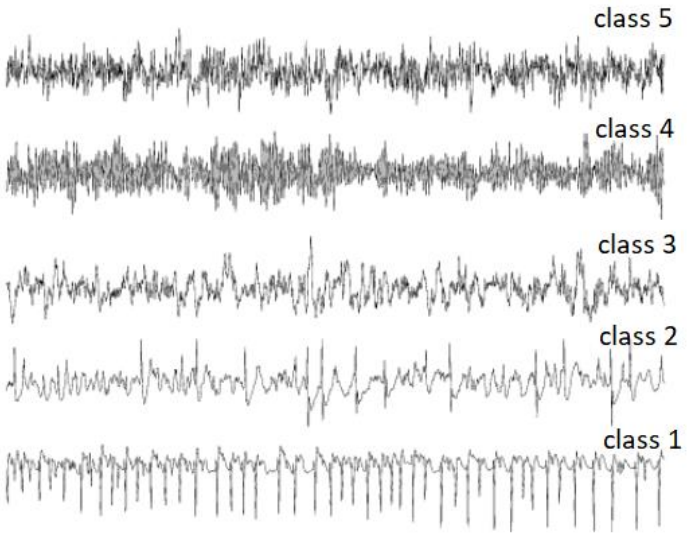

Figure 5.22. An example of EEG signals of all the 5 classes, 1 second long

Previous detection systems had achieved classification accuracy from 84% to 96% for the data set [15].

5.3.2 Spot Checking

For spot checking, the EEG Epileptic data set was pre-processed with the very common method of rearranging every 4,097 data points into 23 chunks and then shuffling the chunks, each with 178 samples and a target class label. This resulted in a single table with 11,500 rows in random order. The purpose is to test the performance of existing algorithms with the time series in time delay representation without the additional measures in the proposed deep temporal convolution network, such as the maintenance of short term tempoal order within the mini-batches, the overlap of the mini-batches, and the shuffling of the order of the mini-batches.

10-fold cross-validation was done in Python with five different algorithms, namely (1) logistic regression, (2) K nearest neighbour, (3) CART decision tree, (4) MLP neural network, and (5) ensemble by voting. All the algorithms were run in their default configurations. The data attributes were standardized in each of the folds during cross-validation. The data vectors in the data table are shuffled before they are used for training.

Two different data formats were used for the 10-fold validations. This is shown in Table 5.21 below as “without windowing” and “with windowing” (window length is 3 chunks, which is

534 samples, and slide length is 1 chunk, which is 178 samples). Both data formats yield rather poor classification accuracy scores (in percentage) and standard deviations (in brackets), as shown in Table 5.21 below.

Table 5.21. Classification accuracy with shuffling (epileptic)

	without windowing	with windowing	p-value
LR	25.16% (1.40%)	26.14% (1.45%)	0.128
KNN	47.56% (1.74%)	42.43% (1.47%)	0.000
CART	47.50% (1.18%)	47.20% (1.13%)	0.536
MLP	67.26% (1.09%)	69.77% (0.83%)	0.000
Ensemble by Voting	57.89% (1.55%)	57.85% (1.07%)	0.924

From Table 5.21 above, it can be seen that with the MLP in the default configuration, a classification accuracy of 67.26% is achieved without windowing and 69.77% with windowing. The box plot of the performances of the classifiers are as shown in Figure 5.23 below.

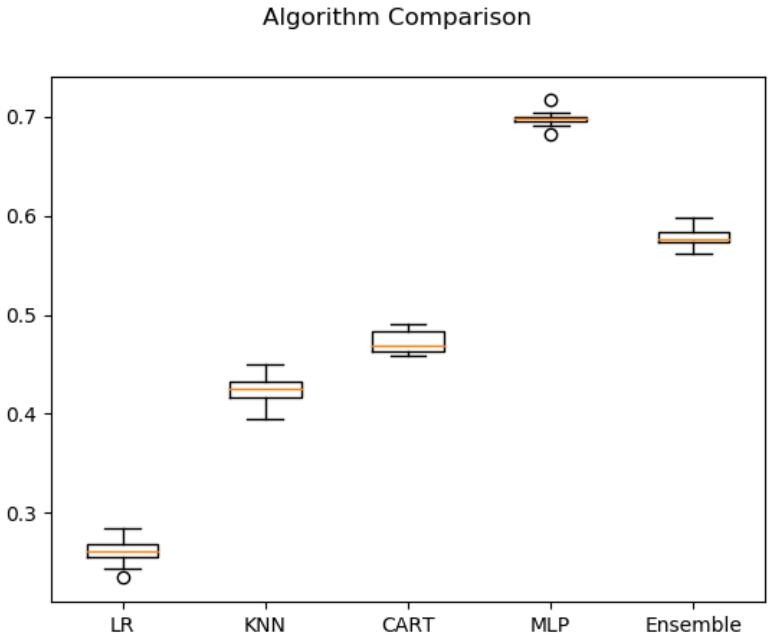

Figure 5.23. Boxplot of algorithm comparison (epileptic)

The poor performance, despite the time delay representation and shuffling before training, could be attributed to the lack of some of the features in the proposed deep temporal convolution network, such as the maintenance of short-term temporal order within the mini-batches, the overlap of mini-batches, and their randomization before training. As a result, the shift-invariant capability of the trained model is poor.

5.3.3 10-Fold Validation, TS = 1, 2, 5

The classification accuracy improves markedly from the dismal 69.77% by MLP to above 96% in the deep temporal network convolution. 10-fold cross-validations yields classification accuracies of 97.65%, 97.46%, and 97.45% for the three configurations, View 1, View 2, and View 3, as shown in Table 5.22 below.

Table 5.22. Configuration of DTCN, View 1, View 2 and View 3 (epileptic)

	View 1	View 2	View 3
Input layer	534	534	534
First hidden layer	200	200	200
Second hidden layer	200	500	350
Third hidden layer	200	200	200
Softmax layer	5	5	5

Table 5.23 and Table 5.24 below shows that classification accuracies of the deep temporal convolution network at $TS = 1$.

Table 5.23. Cross-validation result for DTCN at TS=1 (epileptic)

	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	Fold 6	Fold 7	Fold 8	Fold 9	Fold 10
View 1	97.83	97.97	97.5	98.96	97.45	97.59	96.79	97.59	97.97	96.84
View 2	97.92	97.41	97.64	98.25	97.17	97.17	96.51	97.59	97.74	97.22
View 3	97.78	97.64	97.88	98.16	97.50	97.59	96.98	96.51	97.36	97.08
MTE	98.02	98.16	97.36	99.06	97.78	97.64	97.22	97.39	98.02	96.99

Table 5.24. Means and standard deviations of cross-validation results for DTCN at TS=1 (epileptic)

	Mean	Std Dev
View 1	97.65%	0.62%
View 2	97.46%	0.48%
View 3	97.45%	0.48%
MTE	97.36%	1.31%

The results in Table 5.23 and Table 5.24 above are based on deep temporal convolution network at $TS = 1$, which means that there is still the potential to lift the performance by passing the temporal context to the deeper layers. The accuracies can be lifted further by increasing the concatenation in the deeper layers from $TS = 1$ to $TS = 2$. Table 5.25 and Table 5.26 below show the results of the 10-fold validations with $TS = 2$.

Table 5.25. Cross-validation result for DTCN at TS=2 (epileptic)

	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	Fold 6	Fold 7	Fold 8	Fold 9	Fold 10
View 1	98.76	99.00	98.61	99.43	99.04	98.95	97.75	98.56	97.89	97.89
View 2	98.80	98.80	99.33	98.56	98.66	97.99	98.37	98.76	98.76	97.99
View 3	98.80	98.90	98.04	98.33	98.37	98.28	98.42	98.18	97.99	98.47
MTE	99.23	99.19	98.66	99.28	98.66	98.23	98.23	98.66	98.76	98.23

It can be seen from Table 5.26 below that the accuracies for the three views have improved to 98.59%, 98.60%, and 98.38%, as compared to 97.65%, 97.46%, and 97.45% for $TS = 1$.

Table 5.26. Mean of cross-validation result for DTCN at TS=2 (epileptic)

	Mean	Std Dev
View 1	98.59%	0.57%
View 2	98.60%	0.40%
View 3	98.38%	0.29%
MTE	98.71%	0.41%

If the TS value is increased to 5, the classification accuracies will further improve to 99.89%, 99.87%, and 99.90%. The 10-folds results are shown in Table 5.27 below. The means and standard deviations of the 10-fold results are shown in Table 5.28 below it.

Table 5.27. Cross-validation result for DTCN at TS=5 (epileptic)

	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	Fold 6	Fold 7	Fold 8	Fold 9	Fold 10
View 1	99.62	100.0	99.95	100.0	99.90	99.90	100.0	100.0	99.90	99.62
View 2	99.71	99.71	99.90	99.90	99.62	100.0	100.0	100.0	100.0	99.81
View 3	100.0	99.71	100.0	99.86	99.90	100.0	99.90	100.0	99.81	99.81
MTE	99.95	98.95	99.71	100.0	99.71	100.0	99.95	99.81	100.0	99.90

Table 5.28. Mean and std dev of cross-validation result for DTCN at TS=5 (epileptic)

	Mean	Std Dev
View 1	99.89%	0.15%
View 2	99.87%	0.14%
View 3	99.90%	0.10%
MTE	99.80%	0.32%

Figure 5.24 below shows the performance of the deep temporal convolution network at $TS = 1, 2,$ and 5. It can be seen the classification accuracies increase while the fluctuation decrease when there is more temporal context passed into the deeper layers.

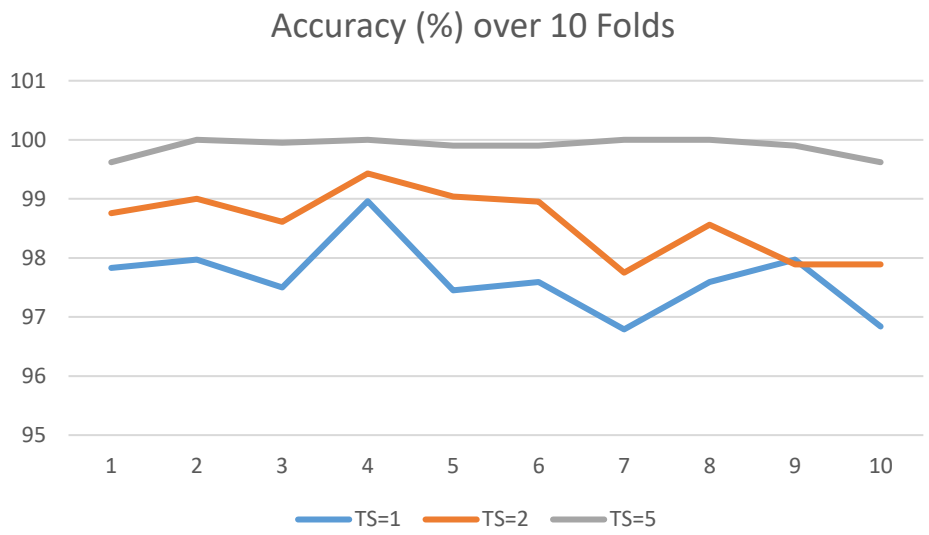

Figure 5.24. Classification accuracy over 10 folds, DTCN, View 1, $TS = 1,2$ and 5 (epileptic)

From the data experiment on the EEG Epileptic data set, it is concluded that deep temporal convolution network leads to better generalization when the temporal context is passed into the deeper layers. The data preparation methods, such as the maintenance of short term temporal order within the mini-batches, the overlapping of the mini-batches, and the shuffling of the overlapping mini-batches, ensure that the time-invariant features are learnt by the network.

5.3.4 Comparing with Equivalent DBN-DNN

To confirm the improvement of the proposed deep temporal convolution network over a DBN-DNN, the equivalent DBN-DNN (in terms of complexity) will be used for the comparison. Here, complexity refers to the number of adjustable parameters in the network.

Two sets of comparisons are made, one at $TS = 2$ and another one at $TS = 5$:

(1) Compare the deep temporal convolution network at $TS = 2$ in three different configurations, View 1, View 2, and View 3, with the equivalent DBN-DNNs.
(2) Compare the deep temporal convolution network at $TS = 5$ in three different configurations, View 1, View 2, and View 3, with the equivalent DBN-DNNs.

The three configurations, View 1, View 2, and View 3, are the same as those shown in Table 5.22 earlier on. The number of adjustable parameters for these configurations at $TS = 1$, 2, and

5 are shown in Table 5.29 below. As can be seen, the number of adjustable parameters for this data experiment is tens of times larger than the data experiment on the EEG Eye State data set.

Table 5.29. No. of adjustable parameters, DTCN, at TS=1, 2, 5

	View 1	View 2	View 3
$TS = 1$	187,800	307,800	247,800
$TS = 2$	267,800	507,800	387,800
$TS = 5$	507,800	1,107,800	807,800

Many DBN-DNNs have about the same number of adjustable parameters as Table 5.29. For the sake of comparison, the configurations in Table 5.30 below shall be used as the equivalent of View 1, View 2, and View 3 at $TS = 2$.

Table 5.30. Equivalent DBN-DNN for TS=2, epileptic

	View 1	View 2	View 3
Input layer	534	534	534
First hidden layer	266	434	355
Second hidden layer	266	434	355
Third hidden layer	200	200	200
Softmax layer	5	5	5
Total No. of Parameters	267,000	507,912	387,595

Using the equivalent configurations of $TS = 2$ in Table 5.30 above, the 10-fold validation results as shown in Table 5.31 and Table 5.32 below were obtained.

Table 5.31. Cross-validation result for equivalent DBN-DNN at TS=2 (epileptic)

Equi TD=2	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	Fold 6	Fold 7	Fold 8	Fold 9	Fold 10
View 1	94.46	92.84	96.49	92.70	88.78	88.24	95.95	88.65	94.32	83.92
View 2	99.32	97.84	89.59	94.18	95.0	98.11	93.51	90.95	90.54	72.84
View 3	89.73	96.35	96.77	93.78	91.35	88.65	89.86	88.24	95.14	95.14
MTE	98.49	97.81	98.22	95.07	95.62	97.81	96.71	92.60	95.62	95.34

Table 5.32. Mean and standard deviation cross-validation result for equivalent DBN-DNN at TS=2 (epileptic)

	Mean	Std Dev
View 1	91.64%	4.06%
View 2	92.19%	7.58%
View 3	92.50%	3.29%
MTE	96.33%	1.83%

By comparing Table 5.31 (equivalent DBN-DNN at $TS = 2$) with Table 5.25 (deep temporal convolution network at $TS = 2$) and plotting the mean results as a bar chart as shown in Figure 5.25 below, it can be seen that the deep temporal convolution network has higher classification accuracy than its DBN-DNN equivalent. The fluctuation across the folds (the variance) is smaller in the case of the deep temporal convolution network also.

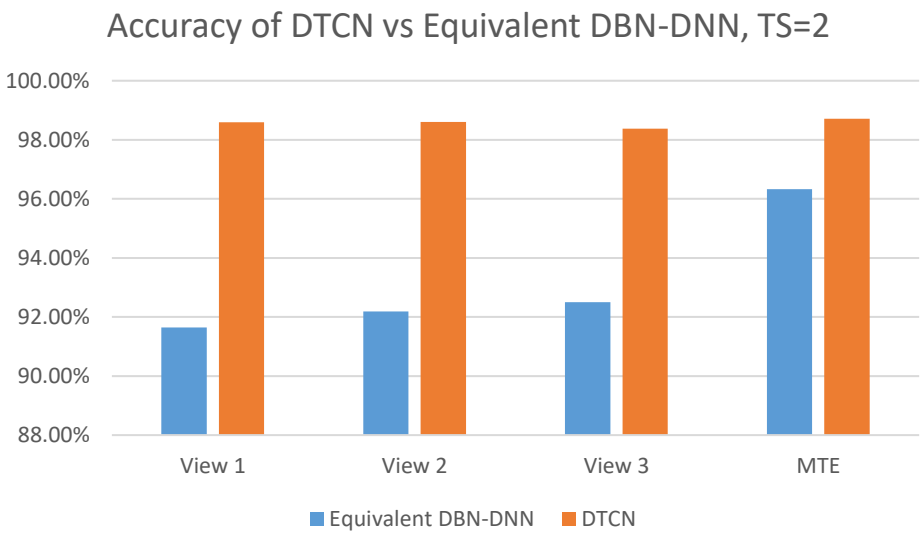

Figure 5.25. Performance of DTCN vs equivalent DBN, $TS = 2$ (epileptic)

5.3.5 Performance Improvement with Ensemble

Two ensembles are used for the comparison here, namely the proposed multi-view temporal ensemble and the ensemble by averaging. Figure 5.26 shows the the classification accuracies over 10 fold over three different views (all at $TS = 1$). It is found that not only is the ensemble's accuracy is generally higher than the individual views, it is higher than the ensemble by averging also.

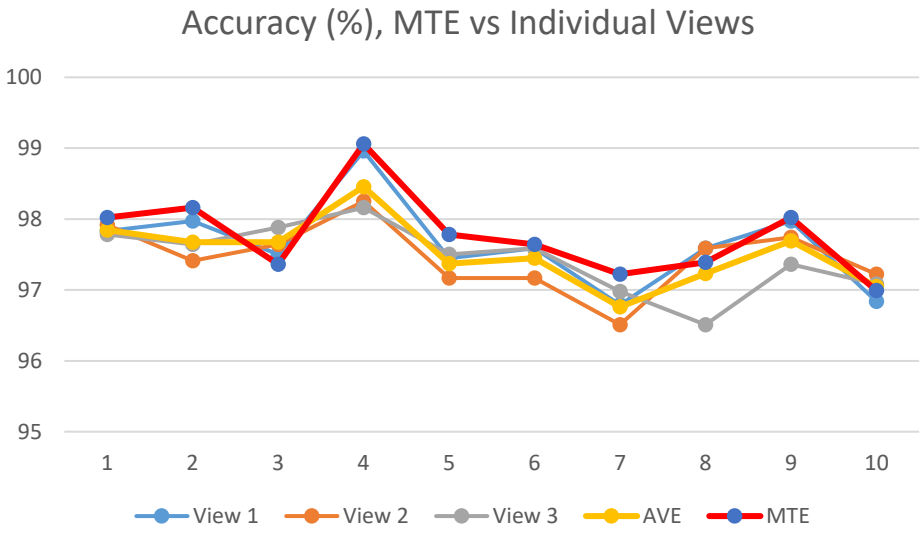

Figure 5.26. Validation result for View 1, 2, 3 and their ensembles at $TS = 1$ (epileptic)

5.3.6 Comparison with Existing Works

The EEG Epileptic data set is a common data set used by many researchers for the study of epileptic. The following are some of the works done with this data set.

Nigam et al. [57] used a multistage nonlinear pre-processing filter in combination with a neural network to achieved an overall accuracy of 97.2% for the EEG Epileptic data set. Both time and frequency features were used in the work.

Güler [150] used entropy measures [25] and adaptive neuro-fuzzy inference system (ANFIS) classifier [151] to distinguish electrical status epilepticus and normal EEG signals, achieving a classification accuracy of about 89%.

Srinivasan et al. [152] used a special type of recurrent neural network known as Elman network. It is a two-layered recurrent neural network with a feedback connection from the output of the hidden layer to its input. The data experiments were carried out with five different attributes in the time and frequency domain, resulting in a classification accuracy that is as high as 99.6%.

Wang et al. [153] used a combination of multi-domain feature extraction (time domain, frequency domain, and time-frequency domain) and nonlinear analysis of EEG signals to extract the features. The dimension of the original feature space is then reduced by using principal component analysis. These features are used with classical classifiers such as k-NN, naïve Bayes, logistic regression and SVM, resulting in accuracies that range from 94.03% to 96.58%.

The performance of the proposed deep temporal convolution network, at 98.53% with $TS = 5$, is comparable to the abovementioned state-of-the-art performance in epileptic detection. In addition, it has the advantage of not needing feature engineering.

5.4 Human Activity Recognition

This section will describe the data experiment done on the Human Activity Recognition (HAR) data set [16] from the UCI Machine Learning Repository [147]. The purpose is to validate the performance of the deep temporal convolution network on the data set.

Hereogenous signals (signals from the accelerometer and the gyroscope) are present in this data set. The views from the acceleraometer and the gyroscope are combined at the data level and so can be considered to be a kind of early data fusion.

The work is presented in four parts: (1) the description of the data set, (2) the spot-checking to get the general benchmark of the data set, (3) the 10-fold validation of the deep temporal convolution network with TS (time steps) value of 1, 2, and 5, and (4) comparison with existing works.

5.4.1 Data Set

The HAR data set is a multivariate numeric time series data set. It is a motion sensor data set based on the recordings of 30 subjects performing activities of daily living. The signals were annotated by comparing the signals with the video recording of the subjects.

The six activities are: (1) walking, (2) walking upstairs, (3) walking downstairs, (4) sitting, (5) standing, and (6) laying down. An additional six transitional activities are also described in the data set, as shown in Table 5.33 below.

Table 5.33. Description of the 12 classes in the HAR data set

	Class	Description
Basic Activity	1	Walking
	2	Walking Upstairs
	3	Walking Downstairs
	4	Sitting
	5	Standing
	6	Laying
Transitional Activity	7	Stand to Sit
	8	Sit to Stand
	9	Sit to Lie
	10	Lie to Sit
	11	Stand to Lie
	12	Lie to Stand

The subject was made to carry a waist-mounted Samsung Galaxy smartphone and go through a set routine twice. The sensors in the smartphone acquired the linear acceleration and the angular velocity of the subject in three dimensions at a constant sampling rate of 50 Hz. As a result, there are six channels in the data set, namely:

- acceleration, x-axis
- acceleration, y-axis
- acceleration, z-axis
- gyroscope, x-axis
- gyroscope, y-axis
- gyroscope, z-axis

The unit for acceleration is in g and that for gyroscope is rad/sec.

The sensor signals were pre-processed by applying a low-pass filter with a cut-off frequency of 20 Hz. This is because 99% of motion signal energy is below 15 Hz [154].

The sensor acceleration signal has a low frequency component (gravitation) and a high frequency component (body motion). They could be separated by a Butterworth low-pass filter with a cut-off frequency of 0.3 Hz.

A data table was created from the data set by the sliding window method. The length of the fixed-width sliding window was 128 samples (2.56 second). The slide was 64 samples (50% overlap). The data table thus created has 10,299 rows of data vectors in it. As all the six attributes of the sample were placed in a row, each data vector contains 768 values.

In each fold of the 10-fold validation, 3 subjects were picked for testing and the remaining 27 subjects were used for training. The data vectors of each subject are in their natural time order. The number of data vectors in the folds are unequal, as shown in Table 5.34 below.

Table 5.34. Number of data vectors in each of the 10 folds

Fold Number	No. of Instances
1	1303
2	1170
3	1355
4	1252
5	1293
6	1378
7	1168
8	1317
9	1195
10	1222

For convenience in storing the data as a tensor, the number of data vectors will have to be the same for all the 10 folds. Thus, only the first 1,168 data vectors of each subject were used in training and/or test, with the rest discarded.

5.4.2 Spot Checking

With the HAR data set, 10-fold cross-validation was done in Python with five different algorithms, namely (1) logistic regression, (2) K nearest neighbour, (3) CART decision tree, (4) MLP neural network, and (5) ensemble by voting. All the algorithms were run in their default configurations. The data attributes were standardized in each of the folds during cross-validation. The data vectors in the data table are shuffled before they are used for training.

The data were rearranged (using the sliding window method, with window length as 128 sample, which has 768 values, and the slide as 64 samples) as overlapping segments.

The 10-fold validated results are as shown in Table 5.35 below.

Table 5.35. Classification accuracy with shuffling (HAR)

Algorithm	Accuracy (in percentage), 10-fold validated
Logistic Regression	64.822 (1.309)
K Nearest Neighbour	77.009 (1.035)
CART	72.836 (1.465)
MLP	87.868 (0.870)
Ensemble	85.29 (0.95)

From Table 5.35 above, it can be seen that with the MLP in the default configuration, a classification accuracy of 87.87% was achieved. The box plot of the performances of the classifiers are as shown in Figure 5.27 below.

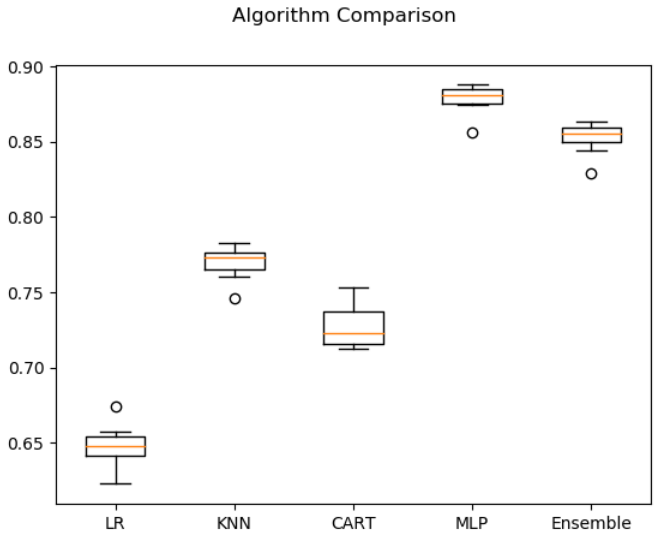

Figure 5.27. Boxplot of algorithm comparison (HAR)

There is certainly room for improvement in the performance, as it is likely that the temporal context in related data vectors was not utilised for learning. That could be done with the deep temporal convolution network.

5.4.3 10-Fold Validation, TS = 1, 2, 5

The three configurations, View 1, View 2, and View 3, used for the cross-validations are as shown in Table 5.36 below. All the three views are based on deep temporal convolution network at $TS = 1$.

Table 5.36. Configuration of DTCN, View 1, 2 and 3 (HAR)

	View 1	View 2	View 3
Input layer	768	768	768
First hidden layer	200	200	200
Second hidden layer	200	500	350
Third hidden layer	200	200	200
Softmax layer	12	12	12

Table 5.37 and Table 5.38 below shows that classification accuracies of the deep temporal convolution network at $TS = 1$.

Table 5.37. Cross-validation accuracies (%) for DTCN at TS=1 (HAR)

	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	Fold 6	Fold 7	Fold 8	Fold 9	Fold 10
View 1	99.87	99.87	99.87	98.96	97.45	97.59	96.79	97.59	97.97	96.84
View 2	97.92	97.41	97.64	98.25	97.17	97.17	96.51	97.59	97.74	97.22
View 3	97.78	97.64	97.88	98.16	97.50	97.59	96.98	96.51	97.36	97.08
MTE	98.02	98.16	97.36	99.06	97.78	97.64	97.22	94.39	98.02	95.99

Table 5.38. Means and standard deviation deviations of cross-validation result for DTCN at TS=1 (HAR)

Mean	Std Dev
97.65%	0.62%
97.46%	0.48%
97.45%	0.48%
97.36%	1.31%

The accuracies can be lifted further by increasing the concatenation in the deeper layers from $TS = 1$ to $TS = 2$. Table 5.39 and Table 5.40 below shows the results of the 10-fold validations with $TS = 2$.

Table 5.39. Cross-validation accuracies (%) for DTCN at TS=2 (HAR)

	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	Fold 6	Fold 7	Fold 8	Fold 9	Fold 10
View 1	99.83	99.78	99.87	99.74	99.82	99.91	99.87	99.87	99.87	99.83
View 2	99.83	99.87	99.87	99.87	99.96	99.83	99.87	99.78	99.78	99.91
View 3	99.83	99.96	99.91	99.91	99.96	99.83	99.91	99.83	99.87	99.87
MTE	99.70	99.96	99.48	99.78	99.91	99.66	99.87	99.87	99.87	99.91

Table 5.40. Means and standard deviations of cross-validation result for DTCN at TS=2, HAR

Mean	Std Dev
99.84%	0.05%
98.84%	0.04%
99.89%	0.05%
99.80%	0.15%

The accuracies for the three views are 99.84%, 99.84%, and 99.89% at $TS = 2$ (Table 5.39), as compared to 97.65%, 97.46%, and 97.45% at $TS = 1$ (Table 5.38).

However, there is a limit to how much lift there is to the performance of deep temporal convolution network. When $TS = 5$ was used in the cross-validations, the classification accuracies actually decreased slightly to 98.95%, 99.32%, and 99.40%.

Table 5.41. Cross-validation accuracies (%) for DTCN at TS=5, HAR

	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	Fold 6	Fold 7	Fold 8	Fold 9	Fold 10
View 1	99.66	98.88	99.66	99.57	98.53	99.48	94.61	99.83	99.91	99.40
View 2	99.66	99.48	99.66	99.91	100.0	100.0	99.57	97.84	99.74	97.41
View 3	99.40	97.84	99.74	99.82	98.97	99.90	99.97	99.55	99.37	99.48
MTE	99.87	99.71	99.69	99.90	99.55	99.24	99.72	99.70	99.89	99.86

Table 5.42. Means and standard deviations of cross-validation result for DTCN at TS=5, HAR

	Mean	Std Dev
View 1	98.95%	1.58%
View 2	99.32%	0.92%
View 3	99.40%	0.62%
MTE	99.71%	0.20%

Since $TS = 5$ incurs longer computation time without improving the performance, its use is a sign of diminishing return of passing temporal context to the deeper layers.

5.4.4 Comparison with Existing Works

Anguita et al. [155] adapted the standard multi-class SVM with computational cost reduction in mind. This was done by using fixed-point arithmetic in the computation. A classification accuracy of 88.97% was obtained for the human activity recognition data set.

Romera-Paredes [156] used an ensemble of linear SVMs each trained to discriminate a single motion activity against another single motion activity. A majority voting rule is used to

determine the final outcome. The one-versus-one ensemble achieved a classification accuracy of 96.4%.

Almaslukh et al. [157] used the stacked autoencoder (SAE) in deep learning to improve on the classification and at the same time reduce the time resolution for recognition. The classification accuracy was 97.5%, which is significantly better than the traditional SVM method.

Based on the above, it can be seen the performance of the deep temporal convolution with $TS = 1$, at 97.51%, is able to match the state of the art performance for the data set. When it is used with $TS = 2$, the performance increased to about 99.80%. This shows that it is useful to pass the temporal context to the deeper layers of the proposed network.

5.5 Freezing of Gait during Walking

Freezing of Gait (FOG) is a sudden and transient inability to walk. It is a common symtom in Parkinson's Disease patients, particularly in those severely affected by the disease [158]. It often casues falls, interferes with daily activities, and impairs the quality of life [159]. Subtypes of FoG include start hesitation, turn hesitation, hesitation in tight quarters, destination hesitation, and open space hesitation [160]. Although Parkinson's Disease is neurological in etiology and is treated with Levodopa [161], it is often resistent to pharmacologic treatement [162]. Non-pharmacologic treatment, such as rhythmic auditory stimulation [163], physiotherapy and assisted technology, can help improve the quality of life of the patients.

The work on the FoG data set is presented in three parts: (1) description of the data set, (2) the 10-fold validation of the deep temporal convolution network with TS (time steps) value of 1, 2, and 5, and (3) comparison with existing works.

5.5.1 Data Set

The FoG dataset [17] is obtained from the UCI Machine Learning Repository [147]. It consists of the annotated readings of three accelerometers placed at the ankle, upper leg, and trunk of Parkinson's disease patients. The dataset was recorded in the lab from 10 subjects. Each subject performed movements such as walking in a staright line or with turns, and also other activities of daily living, e.g. going to different rooms, fetching items, opening doors, etc. A professional

physiotherapist reviewed the video recording of the subjects and determined the start and end time of FoG. The whole time series were classified as irrelevant, normal, and FoG, as shown below:

Class 0: irrelevant part of the experiment, e.g. preparation or debriefing (discarded during data experiment)

Class 1: no freeze while standing, walking or turning

Class 2: freezing of gait

There were 237 FoG events as determined by the professional physiotherapist over a total of 8 hours and 20 minutes of recorded data. They came from eight out of the ten subjects, as two subjects did not experience any FoG. The length of the FoG events range from 0.5 second to 40.5 second.

Each file has multiple lines, with each line corresponding to a sample at 64 Hz, i.e. the time interval is 15.625 millisecond. The first column of each line has a time stamp in millisecond. The next nine columns correspond to nine channels of accelerometer readings. The last column is the target class label (0, 1 or 2). The nine channels are as follows:

Ankle (shank) acceleration - horizontal forward acceleration [mg]

Ankle (shank) acceleration - vertical [mg]

Ankle (shank) acceleration – horizontal lateral [mg]

Upper leg (thigh) acceleration - horizontal forward acceleration [mg]

Upper leg (thigh) acceleration - vertical [mg]

Upper leg (thigh) acceleration - horizontal lateral [mg]

Trunk acceleration - horizontal forward acceleration [mg]

Trunk acceleration - vertical [mg]

Trunk acceleration - horizontal lateral [mg]

5.5.2 10-Fold Validation, TS = 1, 2, 5

A window length of 16, corresponding to 250 ms, is used. Since this is a 9-channel data set, each window has 144 numeric values. For validation purpose, each of the folds gets equal chunks of contiguous data from all the subjects.

The three configurations used for the cross-validations, namely View 1, View 2, and View 3, are shown in Table 5.43 below. All the three views are based on deep temporal convolution network at $TS = 1$.

Table 5.43. Configuration of DTCN, View 1, 2 and 3 (FoG)

	View 1	View 2	View 3
Input layer	144	144	144
First hidden layer	100	100	100
Second hidden layer	100	250	175
Third hidden layer	100	100	100
Softmax layer	2	2	2

In this data experiment, the class specific accuracies, i.e. sensitivity and specificity, are computed, instead of the overall classification accuracy. This is for convenient comparison with the reported results of other works in the literature. Sensitivity is the classification accuracy of Class 2 (FoG), and specificity is the classification accuracy of Class 1 (non-FoG).

Table 5.44 and Table 5.45 below show the sensitivities and specificities of the deep temporal convolution network at $TS = 1$.

Table 5.44. Cross-validation accuracies (%) for DTCN at TS=1 (FoG)

		Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	Fold 6	Fold 7	Fold 8	Fold 9	Fold 10
View 1	Se	96.37	97.51	96.84	97.24	91.59	96.97	97.38	96.57	95.49	97.58
	Sp	99.77	99.71	99.66	99.70	99.82	99.85	99.62	99.72	99.74	99.74

View 2	Se	97.92	94.55	97.04	96.97	97.71	97.65	97.85	97.98	95.49	96.37
	Sp	99.72	99.80	99.78	99.75	99.62	99.70	99.65	99.62	99.85	99.77
View 3	Se	97.65	98.18	95.76	96.23	97.24	97.58	96.91	96.91	97.31	96.37
	Sp	99.68	99.76	99.75	99.78	99.73	99.72	99.80	99.74	99.78	99.77
MTE	Se	98.59	98.72	98.82	98.82	99.79	99.79	98.22	98.05	98.62	98.71
	Sp	99.82	99.86	99.78	99.80	99.75	99.79	99.80	99.86	99.81	99.81

Table 5.45. Means and standard deviation deviations of cross-validation result for DTCN at TS=1 (FoG)

	Se		Sp	
	Mean	Std Dev	Mean	Std Dev
View 1	96.36	1.79	99.73	0.07
View 2	96.95	1.16	99.72	0.08
View 3	97.01	0.73	99.75	0.04
MTE	98.71	0.46	99.81	0.03

The accuracies shown in Table 5.45 seem good enough, but they can be lifted further by increasing the concatenation in the deeper layers from $TS = 1$ to $TS = 2$. Table 5.46 and Table 5.47 below shows the results of the 10-fold validations with $TS = 2$.

Table 5.46. Cross-validation accuracies (%) for DTCN at TS=2 (FoG)

		Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	Fold 6	Fold 7	Fold 8	Fold 9	Fold 10
View 1	Se	99.33	95.83	98.65	98.86	98.39	98.39	99.26	98.79	98.79	98.45
	Sp	99.71	99.94	99.92	99.91	99.86	99.86	99.69	99.54	99.85	99.85
View 2	Se	99.06	98.72	98.92	99.19	99.19	94.95	98.92	99.19	97.24	99.13
	Sp	99.81	99.84	99.82	99.91	99.75	99.91	99.91	99.89	99.94	99.66
View 3	Se	99.78	98.39	98.92	99.53	98.86	99.13	99.39	99.13	98.39	99.06
	SP	99.94	99.97	99.93	99.94	99.91	99.90	99.94	99.80	99.89	99.84
MTE	Se	99.46	99.50	99.76	99.36	98.86	99.50	99.56	99.70	99.50	99.96
	Sp	99.97	99.90	99.91	99.96	99.92	99.97	99.91	99.92	99.96	99.93

Table 5.47. Means and standard deviations of cross-validation result for DTCN at TS=2, FoG

	Se		Sp	
	Mean	Std Dev	Mean	Std Dev
View 1	98.47	0.98	99.81	0.13
View 2	98.45	1.36	99.84	0.09
View 3	98.86	0.53	99.90	0.05
MTE	99.41	0.29	99.94	0.03

It can be seen that the sensitivities (accuracies of detecting FoG) for the three views have improved to 98.47%, 98.45% and 98.86%, as compared to 96.36%, 96.95% and 97.01% at $TS = 1$ (in Table 5.45 above). These results are further improved to 99.41% when they are fused together by multi-view temporal ensemble.

However, there is a limit to how much lift there is to the performance by deep temporal convolution network. When $TS = 5$ was used in the cross-validations, the sensitivities actually decreased slightly to 97.89%, 95.40%, and 97.20%. Since $TS = 5$ incurs longer computation

time without improving the performance, its use is a sign of diminishing return. Nevertheless, there is still an improvement in performance when the three views are blended by the multi-view temporal ensemble, as shown by the 99.29% sensitivity when this was done.

Table 5.48. Cross-validation accuracies (%) for DTCN at TS=5, FoG

		Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	Fold 6	Fold 7	Fold 8	Fold 9	Fold 10
View 1	Se	99.06	99.39	98.52	99.19	98.79	95.49	99.60	89.85	99.06	99.95
	Sp	99.74	99.56	99.83	99.86	99.87	99.93	99.95	99.69	99.93	99.95
View 2	Se	99.73	70.21	98.18	94.35	96.30	99.39	99.39	97.65	98.79	100
	Sp	99.97	99.48	99.92	99.50	99.34	99.98	99.97	99.77	99.39	99.39
View 3	Se	98.99	97.78	93.14	92.87	98.72	99.73	93.88	99.53	98.45	98.85
	SP	99.98	99.95	99.70	99.93	99.65	99.95	99.82	97.93	99.58	99.66
MTE	Se	99.46	99.02	99.60	99.70	98.62	99.43	99.09	99.66	99.56	98.72
	Sp	99.92	99.90	99.94	99.90	99.74	99.91	99.91	99.94	99.93	99.76

Table 5.49. Means and standard deviations of cross-validation result for DTCN at TS=5, FoG

	Se		Sp	
	Mean	Std Dev	Mean	Std Dev
View 1	97.89	3.08	99.83	0.13
View 2	95.40	9.02	99.67	0.27
View 3	97.20	2.75	99.61	0.61
MTE	99.29	0.39	99.89	0.07

5.5.3 Comparison with Existing Works

The original paper on the FoG data set by Bächlin et al. [17] mentioned about an algorithm [164] that ran on an wearable device that had a sensitivity of 73.1% and a specificity of 81.6%, based on 0.5-second frames. They extracted frequency components from the signals and computed a freeze index (FI) [165], defined as the power in the “freeze” band (0.5 to 3 Hz)

divided by the power in the "locomotor" band (3 to 8 Hz). Two threshold values were then used to classify the FoGs. The distribution in sensitivities and specificities among the 10 subjects was substantial. This shows that the algorithm is subject specific and prone to uncontrolled extraneous factors, such as the walking styles of the subjects. After using subject-specific thresholds, the sensitivity and specificity improved to 85.9% sensitivity and 90.9% specificity.

Mazilu et al [166] made use of multiple machine learning techniques on the FoG data set. The test was done with 1-second frames. The results are as shown in Table 5.50 below.

Table 5.50. Classification accuracy with shuffling (FoG)

Algorithm	Sensitivity (%)	Specificity (%)
Random Forest	97.76	99.75%
C4.5	93.47%	99.38%
Naïve Bayes	48.06%	98.66%
MLP	77.46%	97.29%
Ensemble (AdaBoost)	98.35%	99.72%

Due to the use of more powerful machine learning techniques and the increased window length (from 0.5-second to 1-second), the performances in Table 5.35 above are generally better than the algorithm used in the original experiment. The classification of the non-FOG signals seems reliable, as most of the specificities in Table 5.50 above are near or above 99%.

Nevertheless, it can be seen from Table 5.50 above that the MLP is quite weak in sensitivity, even though is specificity is close to that of the other methods. By using the proposed deep temporal convolution network, the sensitivity is significantly higher. When $TS = 2$, the sensitivity is 98.45%, which is almost the same as the best score of 98.35% based on the adaboost ensemble, even though the window length is only 0.25-second long. This shows that the temporal context in the mini-batch is able to provide useful information to the model, while keeping the instances in the mini-batch short. With the lift provided by the multi-view temporal ensemble, the sensitivity is further improved to 99.41%.

5.6 EMG Lower Limb Analysis

Knee dystonia is an involuntary muscle contraction that results in abnormal posture or twisting of the body. In this data experiment, five channels of time series comprising four EMG electrode signals and one goniometric reading are used with the deep temporal convlution network to classify abonrmal knee movements.

The work is presented in three parts: (1) description of the data set, (2) 10-fold validation of the deep temporal convolution network with TS (time steps) value of 1, 2, and 5, and (3) comparison with existing works.

5.6.1 Data Set

The EMG Lower Limb dataset [167] is from the UCI Machine Learning Repository [147]. This database contains samples from 11 subjects with knee disorder and 11 normal subjects. All the subjects were asked to undergo three movements to analyze the behavior associated with the knee muscle, namely walking gait, leg extension from a sitting position, and flexion of the leg up. The dataset contains six classes, three corresponding to normal leg movements, and three corresponding to abnormal leg movements. The use of this dataset is to predict if a person has abnormal knee or not, and in which of the three movements.

These data were collected with an instrument for electromyography and goniometry. Four EMG electrodes were placed on the leg of the subject (rectus femoris RF, biceps femoris BF, vastus medialis VM, semitendinosus ST), while the goniometer was placed on the flexion of the knee (FX). Each reading thus has five attributes. The sampling rate was 1,000 samples per second. The EMG values are in millivolt, while the FX values are in degrees. Each movement of a subject is about 15 seconds.

5.6.2 10-Fold Validation, TS = 1, 2, 5

The three configurations, View 1, View 2, and View 3, used for the cross-validations are as shown in Table 5.51 below. All the three views are based on the proposed network at $TS = 1$.

Table 5.51. Configuration of DTCN, View 1, View 2 and View 3 (EMG Lower Limb)

	View 1	View 2	View 3
Input layer	80	80	80
First hidden layer	100	100	100
Second hidden layer	100	250	175
Third hidden layer	100	100	100
Softmax layer	6	6	6

Table 5.52 and Table 5.53 below shows that classification accuracies of the deep temporal convolution network at $TS = 1$.

Table 5.52. Cross-validation accuracies (%) for DTCN at TS=1 (EMG Lower Limb)

	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	Fold 6	Fold 7	Fold 8	Fold 9	Fold 10
View 1	87.11	94.31	93.27	90.65	96.04	92.27	90.19	91.46	96.46	93.54
View 2	89.04	93.81	93.58	91.19	93.65	93.65	91.50	91.65	96.85	94.69
View 3	90.04	94.23	95.00	92.23	95.23	92.12	91.04	92.27	96.19	94.69
MTE	94.77	96.42	97.38	98.27	98.65	97.23	96.69	97.15	98.85	96.12

Table 5.53. Means and standard deviation deviations of cross-validation result for DTCN at TS=1 (EMG Lower Limb)

	Mean	Std Dev
View 1	92.53%	2.83%
View	92.96%	2.17%
View 3	93.30%	2.03%
MTE	97.15%	1.24%

The accuracies can be lifted further by increasing the concatenation in the deeper layers from $TS = 1$ to $TS = 2$. Table 5.54 and Table 5.55 below show the results of the 10-fold validations with $TS = 2$.

Table 5.54. Cross-validation accuracies (%) for DTCN at TS=2 (EMG Lower Limb)

	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	Fold 6	Fold 7	Fold 8	Fold 9	Fold 10
View 1	92.92	96.15	95.31	94.30	97.54	95.81	94.27	94.19	98.00	95.96
View 2	94.58	97.11	96.12	97.77	96.58	96.58	95.27	95.38	97.73	97.04
View 3	92.65	97.38	96.85	96.38	98.23	97.15	95.00	95.53	98.62	97.08
MTE	96.88	97.62	98.65	98.69	99.58	98.08	96.88	96.38	99.12	96.92

Table 5.55. Means and standard deviations of cross-validation result for DTCN at TS=2 (EMG Lower Limb)

	Mean	Std Dev
View 1	95.45%	1.58%
View 2	96.42%	1.07%
View 3	96.49%	1.74%
MTE	97.88%	1.10%

It can be seen that the accuracies for the three views have improved to 95.45%, 96.42%, 96.49% and 97.88% from 92.53%, 92.96%, 93.30% and 97.15% at $TS = 1$ (Table 5.53).

However, there is a limit to how much lift there is to the performance by deep temporal convolution network. When $TS = 5$ was used in the cross-validations, the classification accuracies actually decreased slightly to 97.65%, 97.46%, and 97.45%.

Table 5.56. Cross-validation accuracies (%) for DTCN at TS=5 (EMG Lower Limb)

	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	Fold 6	Fold 7	Fold 8	Fold 9	Fold 10
View 1	87.12	96.15	95.27	96.04	97.77	95.58	93.77	96.38	94.31	96.69
View 2	93.73	97.04	93.31	95.23	93.69	93.69	95.81	86.96	95.31	96.69
View 3	91.27	96.77	94.35	94.00	97.46	96.15	96.88	95.69	92.50	95.08
MTE	96.69	97.88	95.31	97.62	97.15	95.96	97.62	93.58	95.88	97.42

Table 5.57. Means and standard deviations of cross-validation result for DTCN at TS=5 (EMG Lower Limb)

	Mean	Std Dev
View 1	94.91%	2.97%
View 2	94.15%	2.85%
View 3	95.02%	2.00%
MTE	96.51%	1.35%

Since $TS = 5$ incurs longer computation time without improving the performance, its use is a sign of diminishing return. Sensitivity analysis such as this is used to determine the value of the TS hyper-parameter.

5.6.3 Comparison with Existing Works

Table 5.58 below shows the performances of some previous works. C.D. Joshi et al. [168] made use of Bayesian Information Criteria (BIC), feature extraction and linear discriminant analysis to classify the knee patterns, achieving a classification accuracy of 92.72%. Zhang et al. [169] extracted time and frequency domain features and subject them to five-level wavelet decomposition, which are then classified with the support vector machine. The scheme achieved 91.85% of classification accuracy. Balasubramanyam et al. [167] used wavelet transformation to convert the time series to the time-scale domain and then extract a multitude of features such as Renyi entropy, peak-magnitude to root mean square ratio (PMRS) etc. The features are then classified by a k-nearest neighbour classifier. This scheme achieved 93.5% accuracy.

Table 5.58. Comparative analysis

	Classification Method	Accuracy (%)
C.D. Joshi, U. Lahiri, and N.V. Thakor [168]	Linear discriminant analysis	92.72
P. Li, X. Zhu, S.W. Su, Q. Guo, P. Xu, and D. Yao [169]	Support vector machine	91.85
V. Balasubramanyam et al. [167]	K Nearest Neighbour	93.5%

The proposed deep temporal convolution network was able to achieve a classification accuracy of 96.45%. With the lift provided by multi-view temporal ensemble, it can be further improved to 97.88%. This is higher in performance then the feature extraction methods listed in Table 5.58 above.

5.7 Environment Sound

This section will describe the data experiment done on the Environment Sound (ESC-50) data set [19]. The purpose is to validate the performance of the multi-view temporal ensemble with the CNN-LSTM sub-models on the data set.

The work is presented in five parts: (1) description of the data set, (2) spot-checking to get the general benchmark of the data set, (3) performance evaluation of the individual views, each of which is a CNN-LSTM model configured in a particular way, (4) performance evaluation of the fusion of these CNN-LSTM sub-models using multi-view temporal ensemble, and (5) comparison with existing works.

5.7.1 Data Set

The ESC-50 data set is a univariate numeric time series data set with 2,000 audio recordings constructed from the sound clips in the Freesound project. There are 50 classes in the ESC-50 data set, as shown in Table 5.59 below. Out of these 50 classes, 22 are sounds of animals and humans, and the rest are natural or mechanical sounds.

Table 5.59. Target class labels of the environment sound data set

Animals	Human	Natural sounds	Interior sounds	Exterior sounds
Dog	Crying baby	Rain	Door knock	Helicopter
Rooster	Sneezing	Sea waves	Mouse click	Chainsaw
Pig	Clapping	Crackling fire	Keyboard typing	Siren
Cow	Breathing	Crickets	Door, wood creaks	Car horn
Frog	Coughing	Chirping birds	Can opening	Engine
Cat	Footsteps	Water drops	Washing machine	Train
Hen	Laughing	Wind	Vacuum cleaner	Church bells
Insects (Flying)	Brushing teeth	Pouring water	Clock alarm	Airplane
Sheep	Snoring	Toilet flush	Clock tick	Fireworks
Crow	Drinking, sipping	Thunderstorm	Glass breaking	Hand saw

Each of the 50 classes has 40 recordings. Each recording is a 5 second long .wav file (110,250 samples at 22,050 Hz). They can be decoded and processed with Python packages, namely avconv and LibROSA.

The ESC-50 data set is chosen for this work because the signals are non-stationary and have no obvious time-dependent structure. In this data experiment, the following techniques are used in combination: (1) use the time-frequency representation of the signal as the input, and (2) use deep learning to extract the features from the time-frequency representation and then do classification.

For this data set, there is hardly enough training instances per class for deep learning. This is because there are only 40 recordings per class. To overcome this problem, each of the 5-second audio clips is split into 9 overlapping segments, with 20,992 samples per segment (0.952 seconds).

Within each segment, there are 41 time-consecutive frames. Each frame has 512 samples. Each frame is subjected to Fourier transform and converted to the energy values of a 60-bin Mel cepstrum.

As a result, each segment is a two-dimensional matrix with 41 time steps and 60 coefficients. The two-dimensional matrix has a total of 2,460 coefficients in it. Each of these matrices is associated with the class. There are 360 sets of such matrices for each of classes.

5.7.2 Spot Checking

Before evaluating the performance of the CNN-LSTM model, it is interesting to note that not all deep learning will yield good result on the ESC-50 data set. To show this, two LSTM layers with a softmax layer, as shown in Figure 5.28 below, was used on the time-frequency representation of the ESC-50 data set.

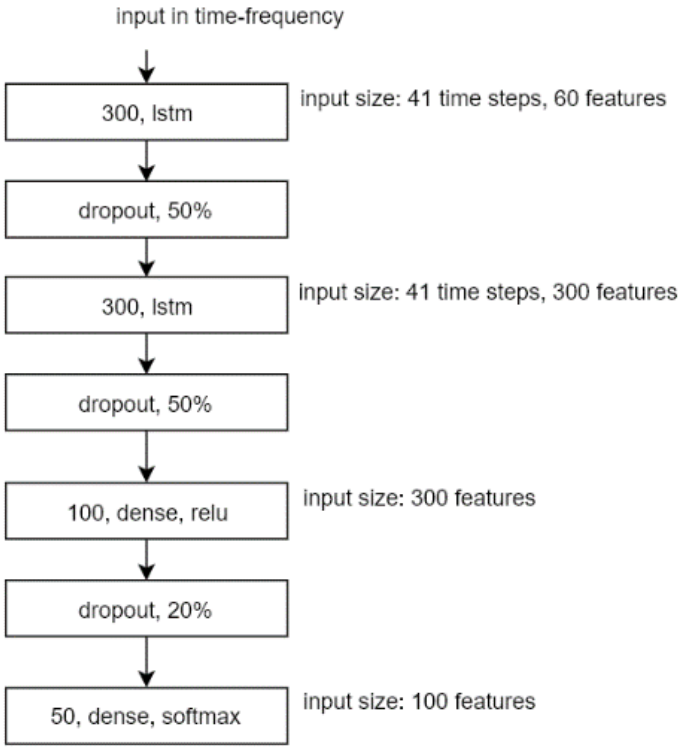

Figure 5.28. Deep learning with two layers of LSTM, ESC-50

The result in Table 5.60 shows that the two LSTM layer configuration is no better than the two CNN layer configuration as reported by Piczak [19]. The result (60.9% accuracy) is less than

appealing despite the use of dropout as regularization. This is likely due to the spectral features not being extracted by the LSTMs as well as the CNNs.

Table 5.60. Classification accuracy of ESC-50 based on different topologies

Topology	Accuracy
2 layers of 2D-CNNs [19]	64.5%
2 layers of LSTMs	60.9%

5.7.3 Performance of the Individual Views

To evaluate the performance of the multi-view temporal ensemble, three configurations of the CNN-LSTM sub-model will first be created. The multi-view temporal ensemble will then blend the penultimate output of the views to give a lift to the system performance.

10-fold validation in deep learning is often too time-consuming. Thus, in this data experiment, the validation is done by 66/33 training/test splitting instead of 10-fold validation.

As the data set is a balanced data set, the classification accuracy is used as the performance metrics in this data experiment.

The CNN-LSTM model is a sequence of layers, consisting of two groups of two-dimensional convolution layers, one group of one-dimensional convolution layer, one group of LSTM layer, a fully connected dense layer, and a softmax layer. For View 1, it is configured as shown in Table 5.61 below.

Table 5.61. CNN-LSTM Topology, View 1 (ESC-50)

Type of Layer	Output Shape	Number of Parameters
Two-dimensional convolution	(32, 41, 60)	832
Max pooling	(32, 20, 30)	0
Dropout	(32, 20, 30)	0
Two-dimensional convolution	(64, 20, 30)	51,264
Max pooling	(64, 10, 15)	0
Dropout	(64, 10, 15)	0
Permute	(10, 64, 15)	0
Reshape	(10, 960)	0
One-dimensional convolution	(10, 192)	553,152
Max pooling	(5, 192)	0
Dropout	(5, 192)	0
LSTM	(360)	796,320
Dropout	(360)	0
Dense	(128)	46,208
Activation, ReLU	(128)	0
Dropout	(128)	0
Dense, Softmax	(50)	6,450

The input to the CNN-LSTM model is a tensor of size $(1,41,60)$. The number of channels is 1, the number of time steps is 41, and the number of attributes is 60.

As can be seen from Table 5.63 above, the tensor output of the first CNN layer is of size $(32,41,60)$. There are 32 feature maps in the tensor output. After max pooling by a 2×2 region, the tensor size is reduced to $(32,20,30)$. The CNN layer and the max pooling layer, together with the dropout layer, form a group of layers.

The group of layers is repeated one more time. Together, these two groups of layers serve to capture the invariant features across the time-frequency structure of the audio segment.

The output of the two convolution groups is a tensor of size $(64,10,15)$. It will be re-organized as a matrix of 10 time-steps of 960 features. This will be used as the input for the next layer, which is the one-dimensional convolution layer. For the one-dimensional convolution layer, the kernel size is 3 time steps by 960 features. It will slide over the time steps, giving off 192 outputs at each of the slide positions.

The output from the one-dimensional convolution layer is fed to an LSTM layer to extract the remaining high-level features. Thereafter, a fully connected layer with ReLU activation is used with a softmax layer to implement the multi-class classification.

Table 5.62 below shows the View 1 result (classification accuracies) over 20 epochs when the above topology is used with the ESC-50 data set. The result, at 83.94%, is close to the reported top scores for this data set.

Table 5.62. Classification accuracies (%) over 20 epochs, View 1 (ESC-50)

19.27	35.90	45.18	51.73	60.45	64.44	67.61	70.65	72.32	74.16
75.09	77.62	78.99	79.89	81.19	82.17	82.23	82.98	83.94	83.94

The topology of View 2 of the CNN-LSTM model is shown in Table 5.63 below. The difference between View 2 and View 1 is that there are fewer feature maps (8 and 16, compared to 32 and 64) at the two-dimensional convolution layers in View 2.

Table 5.63. CNN-LSTM Topology, View 2 (ESC-50)

Type of Layer	Output Shape	Number of Parameters
Two-dimensional convolution	(8, 41, 60)	208
Max pooling	(8, 20, 30)	0
Dropout	(8, 20, 30)	0
Two-dimensional convolution	(16, 20, 30)	3,216
Max pooling	(16, 10, 15)	0
Dropout	(16, 10, 15)	0
Permute	(10, 16, 15)	0
Reshape	(10, 240)	0
One-dimensional convolution	(10, 192)	138,432
Max pooling	(5, 192)	0
Dropout	(5, 192)	0
LSTM	(360)	796,320
Dropout	(360)	0
Dense	(128)	46,208
Activation, ReLU	(128)	0
Dropout	(128)	0
Dense, Softmax	(50)	6,450

The View 2 results produced by the topology are shown in Table 5.64 below. At 82.64%, it is close to the reported top scores for this data set.

Table 5.64. Classification accuracies (%) over 20 epochs, View 2 (ESC-50)

21.00	35.48	47.56	54.30	60.91	64.94	67.84	69.27	72.03	72.55
75.49	75.36	76.85	78.45	79.50	79.62	79.72	81.64	81.58	82.64

The topology of View 3 of the CNN-LSTM model is shown in Table 5.65 below. The difference between View 3 with the earlier two views is again in the number of feature maps at the two-dimensional convolution layers (16 and 32, compared to 32, 16 and 8, 16 in View 1 and 2).

Table 5.65. CNN-LSTM Topology, View 3 (ESC-50)

Type of Layer	Output Shape	Number of Parameters
Two-dimensional convolution	(16, 41, 60)	416
Max pooling	(16, 20, 30)	0
Dropout	(16, 20, 30)	0
Two-dimensional convolution	(32, 20, 30)	12,832
Max pooling	(32, 10, 15)	0
Dropout	(32, 10, 15)	0
Permute	(10, 32, 15)	0
Reshape	(10, 480)	0
One-dimensional convolution	(10, 192)	276,672
Max pooling	(5, 192)	0
Dropout	(5, 192)	0
LSTM	(360)	796,320
Dropout	(360)	0
Dense	(128)	46,208
Activation, ReLU	(128)	0
Dropout	(128)	0
Dense, Softmax	(50)	6,450

The View 3 results produced by the topology are as shown in Table 5.66 below. At 83.06%, it is close to the reported top scores for this data set.

Table 5.66. Classification accuracies (%) over 20 epochs, View 3 (ESC-50)

18.21	35.21	43.62	52.57	57.24	62.24	65.69	68.48	70.98	73.68
73.51	76.82	78.37	78.39	79.47	79.39	80.64	81.71	82.31	83.06

The results in Table 5.62, Table 5.64 and Table 5.66 show that the time-frequency representation of the ESC-50 data set can be classified with state-of-the-art performance. This is because invariant features in both the time and frequency domains were extracted by the two-

dimensional convolution layers, and the remaining high-level temporal features were extracted the one-dimensional convolution layer and the LSTM layer.

5.7.4 Performance Improvement with Multi-view Temporal Ensemble

The penultimate output of View 1, View 2 and View 3 were sent to the multi-view temporal ensemble for blending. It was found that the accuracy of the blended result improved to 85.5%.

Figure 5.29 below shows the performance of the multi-view temporal ensemble versus those of the individual views on the ESC-50 data set. It shows that the complementary data boosts the system performance.

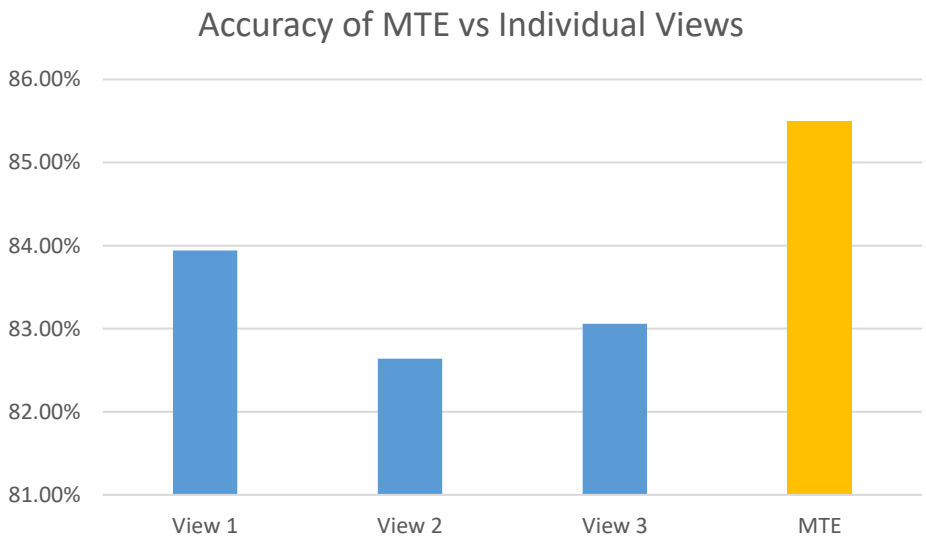

Figure 5.29. Comparison of MTE vs individual view (ESC-50)

5.7.5 Comparison with Existing Works

According to the research done by Piczak [19], the human capabilities in recognizing the sounds in the data set is estimated to be 81.3%. The performance varies across the sounds, with a low of 34.1% for the washing machine noise and almost 100% for crying babies. He postulated that trained and attentive listeners could reach 90% accuracy for the data set.

Previous work by Piczak shows that using a deep learning approach with two convolutional layers with max-pooling and followed by two fully connected layers can produce a classification accuracy of 64.5%.

Since then, better results have been reported in the published papers. Some of the top scores are listed in Table 5.67 below.

Table 5.67. State of the art performance on the ESC-50 data set

Title	Technique	Accuracy
Unsupervised Filterbank Learning Using Convolutional Restricted Boltzmann Machine for Environmental Sound Classification [170]	CNN with filterbanks learned using convolutional RBM + fusion with GTSC and mel energies	86.50%
Learning from Between-class Examples for Deep Sound Recognition [171]	EnvNet-v2 + data augmentation + Between-Class Learning	84.90%
Novel Phase Encoded Mel Filterbank Energies for Environmental Sound Classification [172]	CNN working with phase encoded mel filterbank energies (PEFBEs), fusion with Mel energies	84.15%
Knowledge Transfer from Weakly Labeled Audio using Convolutional Neural Network for Sound Events and Scenes [173]	CNN pre-trained on AudioSet	83.50%

The performance of the individual views, at 83.94%, 82.64% and 83.06% are close to the values in Table 5.67 above. The use of multi-view temporal ensemble was shown to be able to lift the performance to 85.5% and bring the performance even nearer to the best result (86.50%) in Table 5.67 above.

5.8 Heart Sounds

This section will describe the data experiment done on the Heart Sounds data set [20] The purpose is to validate the performance of the multi-view temporal ensemble with the CNN-LSTM sub-models on the data set.

The work is presented in five parts: (1) description of the data set, (2) data preparation done on the data set, (3) performance evaluation of the individual views, each of which is a CNN-LSTM model configured in a certain way, (4) performance evaluation of the multi-view temporal ensemble based on the penultimate outputs of the CNN-LSTM sub-models, and (5) comparison with existing works.

5.8.1 Data Set

The Heart Sounds data set is a univariate numeric time series data set. It is the training and validation set used in the 2016 PhysioNet/CinC Challenge [174]. There are two classes in this data set, namely normal and abnormal. Each of these two classes are further labelled as clean or noisy. In this data experiment, only the two classes, normal and abnormal, are used for classification. The signal quality is not taken into consideration.

There are altogether 5 collections (A through E) in the data set, as shown in Table 5.68 below.

Table 5.68. Collection A, B, C, D, and E of normal and abnormal heart sound recordings

	Normal	Abnormal
Collection A	117	292
Collection B	386	104
Collection C	7	24
Collection D	27	28
Collection E	1,958	183
Total	2,495	631

As can be seen from Table 5.68 above, the data set is unbalanced. There are more normal recordings (2,495) than the abnormal recordings (631).

The recordings were done in clinical as well as non-clinical (for example, in-home visits) settings. Most of the recordings are clean, but some are corrupted by noise such as talking, stethoscope motion, breathing and intestinal sounds etc. Of all these collections, Collection E is a particularly clean set of recordings.

Each subject contributed between one and six heart sound recordings. Each recording contains only one lead. The length of each recording is 5 to 120 seconds. Sampling was done at 2,000 Hz. The data are stored in the .wav format.

5.8.2 Data Preparation

As the recordings are unequal in length (5 seconds to 120 seconds), it is pertinent that only the actual heart sounds (and not the end-of-file silence) be used for the data experiment. After processing the data, those segments that fall below the average decibel of -70dB are removed.

The data processing will normalize the data by the maximum value of each recording. Then, overlapping segments will be created from the normalized data. Each of the overlapping segments will have 5,120 samples (2.56 seconds).

A 512-point frame is slide across the segment, from which 60-bin Mel cepstral features will be computed. Thus, each segment will have 1,260 Mel coefficients (i.e. 21 frames $\times$ 60 bins). The segment has a tensor structure of the shape $(channel = 1, time\ steps = 21, features = 60)$, meeting the expected format of the CNN-LSTM model.

5.8.3 Performance of the Individual Views

To evaluate the performance of the multi-view temporal ensemble, three configurations of the CNN-LSTM model are created. In addition, a different subset of the data set is used for each of the configurations. This produced three different views, with three sets of performances.

Performing 10-fold validation in deep learning is very time-consuming, and so in this data experiment, the validation is done by 66/33 training/test splitting instead of 10-fold validation.

As the data set is unbalanced, sensitivity and specificity will be used as the performance metrics in this data experiment.

The CNN-LSTM model is a sequence of layers, consisting of two groups of two-dimensional convolution layers, one group of one-dimensional convolution layers, one group of LSTM layers, a fully connected dense layer, and a softmax layer. For View 1, it is configured as shown in Table 5.69 below.

Table 5.69. CNN-LSTM Topology, View 1 (heart sounds)

Type of Layer	Output Shape	Number of Parameters
Two-dimensional convolution	(32, 21, 60)	832
Max pooling	(32, 10, 30)	0
Dropout	(32, 10, 30)	0
Two-dimensional convolution	(64, 10, 30)	51,264
Max pooling	(64, 5, 15)	0
Dropout	(64, 5, 15)	0
Permute	(5, 64, 15)	0
Reshape	(5, 960)	0
One-dimensional convolution	(5, 192)	553,152
Max pooling	(2, 192)	0
Dropout	(2, 192)	0
LSTM	(360)	796,320
Dropout	(360)	0
Dense	(128)	46,208
Activation, ReLU	(128)	0
Dropout	(128)	0
Dense, Softmax	(2)	258

The input to the CNN-LSTM model is a tensor of size $(1,21,60)$. The number of channels is 1, the number of time steps is 20, and the number of attributes is 60.

As can be seen from Table 5.69 above, the tensor output of the first two-dimensional convolutional layer is of the size $(32,21,60)$. There are 32 feature maps in the tensor output. After max pooling by a 2×2 region, the tensor size is reduced to $(32,10,30)$. The two-dimensional convolution layer and the max pooling layer, together with the dropout layer, form a group of layers. This group of layers is repeated one more time, and together, the two groups of layers serve to capture the invariant features across the time-frequency structure of the audio segment.

The output of the two convolution groups is a tensor of size $(64,5,15)$. It is re-organized as a matrix of 5 time-steps of 960 features. This is used as the input of the next layer, which is the one-dimensional convolution layer. For the one-dimensional convolution layer, the kernel size is 3 time steps by 960 features. It is slide over the time steps, giving 192 outputs at each of the slide positions.

The output from the one-dimensional convolution layer is fed to an LSTM layer to extract the remaining high-level features. Thereafter, a fully connected layer with ReLU activation is used with a softmax layer to implement the multi-class classification.

Table 5.70 shows the View 1 result (classification accuracies) over 20 epochs when the above topology is used with the heart sound data set. The result, at 85.98%, is very good.

Table 5.70. Performance over 20 epochs, View 1 (heart sounds)

75.18	75.42	79.54	79.99	81.15	81.57	81.77	80.75	82.94	82.59
83.28	83.58	83.61	84.18	84.43	84.72	86.03	85.96	85.27	85.98

As the test set is unbalanced, the sensitivity and specificity (instead of accuracy) will be computed. To do that, the confusion matrix for the test set is obtained and shown in Table 5.71 below.

Table 5.71. Confusion matrix, View 1 (heart sounds)

	Predicted, Normal	Predicted, Abnormal
Actual, Normal	1,879	499
Actual, Abnormal	338	3,252

For View 1, the sensitivity is 79.02% and the specificity is 90.58%. The arithmetic mean of the two scores is 84.80%.

The same process is repeated for View 2 and View 3, each with a different number of feature maps in the CNN layers in the CNN-LSTM topology.

The results produced by the topology for View 2 are shown in Table 5.72 below. At 98.84%, it is much better than the results for View 1. This is due to the subset used for View 2, which is different from the subset used for View 1. The results suggest that the subset used for View 2 has a much higher signal quality (i.e. cleaner).

Table 5.72. Performance over 20 epochs, View 2 (heart sounds)

95.53	96.14	97.03	97.26	97.76	98.07	97.83	97.80	97.93	97.39
97.69	98.40	98.42	98.42	98.58	98.48	98.47	98.75	98.58	98.84

Table 5.73 below shows the confusion matrix for the test set of View 2.

Table 5.73. Confusion matrix, View 2 (heart sounds)

	Predicted, Normal	Predicted, Abnormal
Actual, Normal	3,338	43
Actual, Abnormal	39	3,583

For View 2, the sensitivity is 98.73% and the specificity is 98.92%. The arithmetic mean of the two scores is 98.83%. This is better than View 1 (84.80%).

The results produced by the topology for View 3 are as shown in Table 5.74 below. At 98.40%, it is comparable to the results for View 2 and much better than the results for View 1.

Table 5.74. Performance over 20 epochs, View 3 (heart sounds)

94.43	96.03	96.97	97.21	97.52	97.49	97.77	97.84	97.70	97.65
97.53	98.00	97.83	98.19	98.21	98.36	98.48	98.50	98.34	98.40

Table 5.75 below shows the confusion matrix for the test set of View 3.

Table 5.75. Confusion matrix, View 3 (heart sounds)

	Predicted, Normal	Predicted, Abnormal
Actual, Normal	3,344	64
Actual, Abnormal	49	3,596

For View 3, the sensitivity is 98.12% and the specificity is 98.66%. The arithmetic mean of the two scores is 98.39%.

5.8.4 Performance Improvement with Ensemble

To run the subsets through the multi-view temporal ensemble, the rule of co-occurrence will apply. This means that the same data should be used as the input by all the sub-models. To achieve this, the three subsets are combined as one and then run through the three CNN-LSTM configurations. This produces three separate views. The penultimate outputs of the three views are then blended by the multi-view temporal ensemble to give the system performance a lift.

The confusion matrix of the multi-view temporal ensemble is as shown below.

Table 5.76. Confusion matrix, multi-view temporal ensemble (heart sounds)

	Predicted, Normal	Predicted, Abnormal
Actual, Normal	8,745	705
Actual, Abnormal	244	10,356

For the multi-view temporal ensemble in Table 5.76 above, the sensitivity is 92.54% and the specificity is 97.70%. The arithmetic mean of the two scores is 95.12%. This is higher than the average of the individual views (94.01%), as shown in Figure 5.30 below.

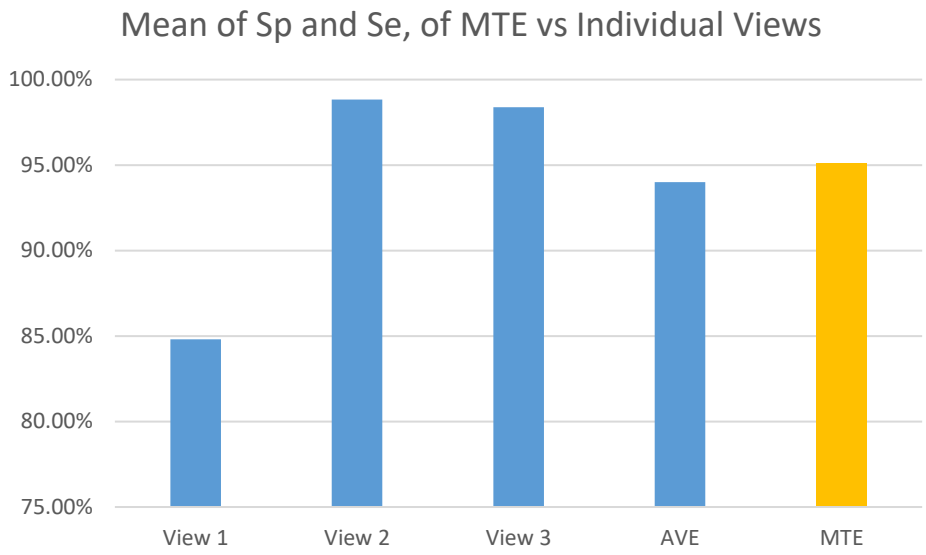

Figure 5.30. Comparison of MTE vs individual view (heart sound)

The shows that the multi-view temporal ensemble is able to reduce the effect of the noisy data and improve the generalization performance over the average performance of the individual sub-models.

5.8.5 Comparison with Existing Works

Physionet Challenge 2016 [20] provides a sample entry to serve as the benchmark of the heart sound data set. It uses a state-of-the-art segmentation technique to identify the S1, S2, systole and diastole states of the heart sounds, from which features are extracted by a form of Hidden Markov Model (HMM). Logistic regression is then used for classification. The validation result (based on subsets from Collections A through E) gives a sensitivity of 71.52% and a specificity of 70.67%.

The same data set, based on Collections A through E, are used in this data experiment. The best score is obtained when multi-view temporal ensemble is used with deep temporal convolution networks at $TS = 2$. The sensitivity is 92.54% and the specificity is 97.70%. They are much

better than the baseline performance in the sample entry in the Physionet Challenge. This shows that the proposed methods can achieve very good performance for the heart sound data set.

The scores of the top entries in the 2016 Physionet Challenge are substantially higher than those of the sample entry [20]. The test set used in the actual challenge is a new and unseen data set different from the training set (Collections A through E). Three classes are defined, namely normal, abnormal and unsure. The scores for sensitivities and specificities are weighted by the amount of noisy signal in the data set. The scores of the top five entries, based on the modified computation for sensitivity and specificity, are shown in Table 5.77 below.

Table 5.77. Final scores for the 2016 Physionet Challenge

	Classification Method	Sensitivity	Specificity
Potes et al. [175]	AdaBoost & CNN	94.24	77.81
Zabihi et al. [176]	Ensemble of SVMs	86.91	84.90
Kay, Agarwal et al. [177]	Regularized Neural Network	87.43	82.97
Bobillo [178]	MFCCs, Wavelets, Tensors & KNN	86.39	82.69
Plesinger et al. [179]	Probability-distribution based	76.96	91.25

The scores in Table 5.77 above are based on a different test set with a slightly different criterion, but they show that the performance of the multi-view temporal ensemble with CNN-LSTM sub-models is able to match up with the top scores in the Physionet Challenge. This complements the conclusion of the other data experiments that the proposed methods provide higher generalization performance on time series data.

Chapter 6. Conclusion

The goal of this thesis is to develop new deep temporal model for the classification of high-dimensional data made from biosignals. Two hypotheses were made towards this goal, namely the deep temporal convolution network and the multi-view temporal ensemble. A number of new ideas were incorporated into their implementations. They were validated using biosignal data sets in the public domain. The results show that the proposed methods could improve the accuracy and variance of the classification of these signals.

This chapter concludes the thesis by highlighting the main points that have been made in the thesis, as well as the work to be done in the future.

6.1 Main Points

The proposed deep temporal convolution network addresses the need in deep learning to match the data function with the appropriate network structure. This takes the form of the introduction of a concatenation sublayer to the deeper layers of a DBN-DNN. This enables the learning of the compositional temporal context within a deep network. For non-stationary time series data such as physiological signals, the data function is highly varying, and so the composition of functions, as used in the proposed network, can be helpful in achieving better performance. To expose the temporal context and encourage the model to be shift-invariant, data processing is used, including (1) short term temporal order, (2) mini-batches that overlap, and (3) pooling of target labels through deeper layers. A matching learning algorithm by backpropagation with gradient routing is also proposed, with the “split-slide-add” operation being used for gradient routing.

The proposed network was tested with the EEG Eye State data set, among others. The result shows that it can generalize better than the equivalent DBN-DNN that makes use of just the time delay representation at the input layer. Figure 6.1 below shows the improvement in accuracy and the reduction in variance with the proposed network.

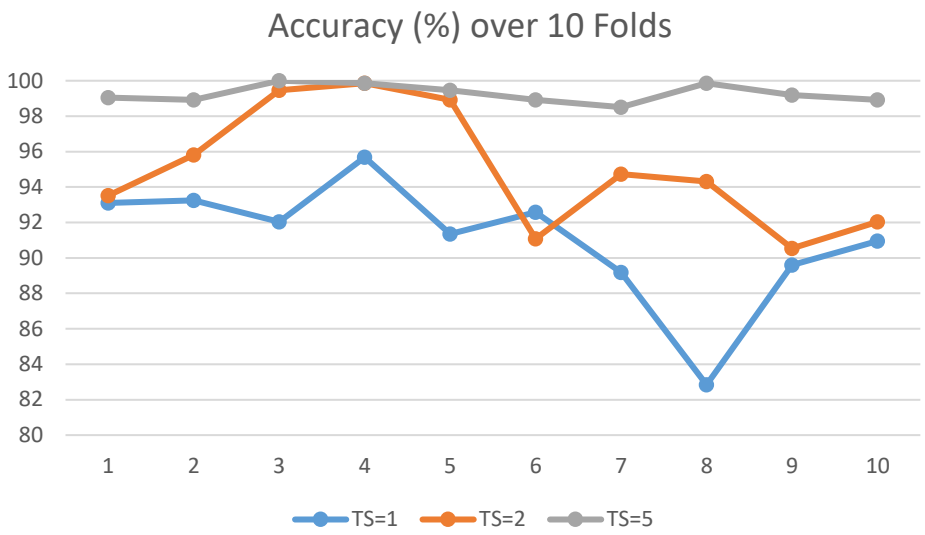

Figure 6.1. Classification accuracies of DTCN with $TS = 1,2,5$, eye state

In Figure 6.1 above, TS (time steps), which is the amount of concatenation, is increased from 1 (no concatenation) to 2 and then to 5. It can be seen from that there is a lift in the curve for $TS = 5$, implying higher accuracy with larger amount of concatenation. At the same time, the fluctuation is smaller (lower variance) in the curve.

The exploration of deep learning was extended to ensemble technique and multi-view learning in this work. An intermediate data fusion technique, called the multi-view temporal ensemble, is proposed for use with time series data such as sound to boost the generalization performance of classification. In the proposed method, the outputs of the sub-models in the ensemble are linearly combined with mixing coefficients so that the features, used as the input by the final classifier, can be more representative of the target concept. The mixing coefficients are based on the complementarity of the views.

In this work, the cost function of the Laplacian eigenmap is adopted for alternate optimization to solve the following two-fold problem: (1) the mixing coefficients are unknown, and (2) the global view (i.e. the weighted sum of the individual views) is also unknown. The alternate update of the two unknowns will result in the minimization of the cost function, resulting in the convergence of the mixing coefficients. This technique can be used with time series data with two rules: (1) co-occurrence, and (2) class-specificity.

A CNN-LSTM ensemble framework was described and tested with a time series data set. The result shows that without manual segmentation and curation, the time series data can be classified with greater generalization performance in the multi-view setting, compared to deep learning based on single view alone.

Figure 6.2 below shows the improvement in accuracy and the reduction in variance for the ECG data set when the multi-view temporal ensemble was used.

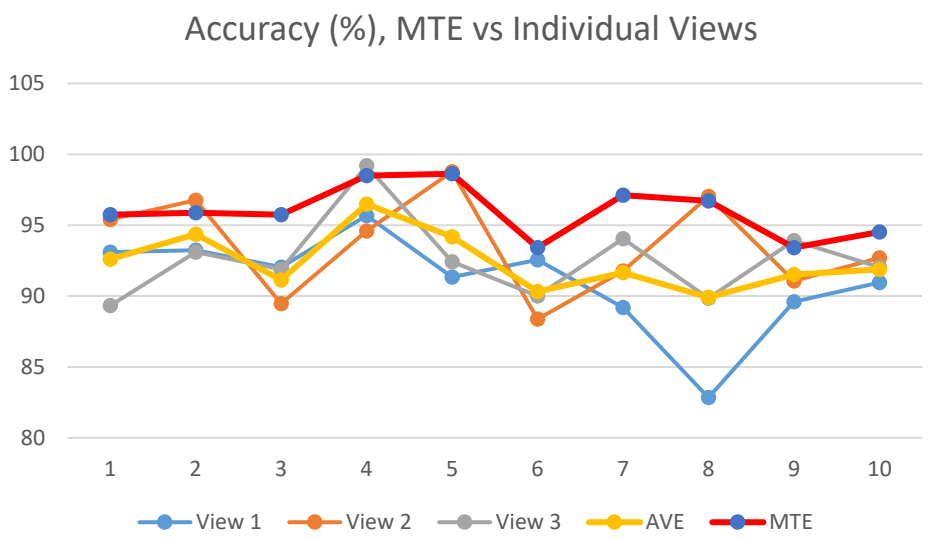

Figure 6.2. Classification accuracies for View 1, 2, 3, their average, and MTE (eye state)

As shown in Figure 6.2 above, the performance of the ensemble is better the simple average of the three individual views. This is a sign that complementarity is a helpful property for learning in ensemble learning.

6.2 Future Works

The current work provides a sound foundation for time series classification, but it does not address the why and how in explanatory data analytics. In future, the time series captured by sensor nodes should be combined with the other dimensions in the process. In such an intelligent sensor network, the deep temporal convolution network and the multi-view temporal ensemble can be used as the artificial intelligence engines within multidimensional data analytics. This

will infuse time series data into the data analytics for better insight into the causes of the phenomenon.

6.2.1 Dimensions as Co-variates

The work in this thesis focuses on multivariate numeric data. It is proposed that the work be extended to a richer kind of biomedical data that contains not just biosignals but also categorical co-variates. These co-variates can be (1) control factors used in the design of experiments, (2) confounding factors that need to be taken into account in observational study, or (3) the identity or attributes of the subjects. These co-variates are likely to be available in a database rather than from the sensors.

An example of such a data set is the data set of patients, where categorical attributes such as gender, social economic status, activity mode, etc. are available. When these factors are not taken into consideration, the results could be biased. For example, the heart rhythm may be biased by the gender, the age, or the activity mode. Therefore, incorporating categorical co-variates to the time series will adjust for the bias.

6.2.2 Intelligent Sensor Network

The advances in sensor, network and cloud computing provide ample opportunities to deploy the solution in the real world setting. With signals from sensors and user data from databases, real world data with time series and categorical covariates can be collected for epidermiological study on social and preventive healthcare issues. Empathy of the ground and understanding of the data are necessary for this effort to add real value to the solution. The evidence-based approach, aided by signals collected by sensors, will provide reliable interpretation of the underlying complex phenomenon.

6.2.3 More Robust to Noise

More strategies can be incorporated into multi-view temporal ensemble, such as wavelet-transformation views and compressive sensing views, instead of just the distance views based on the Gaussian kernel. These strategies make use of other aspects of the signals and may result

in complementarity that is more robust to noise, as the distance measure is known to be sensitive to noise.

6.2.4 Incremental Learning

Incremental learning [180] can be incorporated into the deep learning mode. The deep model trained with the general population can be adapted for use in new situations by training just the final classifier with the new data set. It allows rapid deployment without the time-consuming training of a brand new deep model. There are a number of choices here, but the ELM classifier is a ready-to-use top layer classifier for this purpose.

6.3 Final Words

This work has contributed to the research community by showing that random-looking biosignals can be classified with high accuracies with the proposed deep temporal convolution network and multi-view temporal ensemble. It shows that end-to-end learning of the biosignals can be achieved with little or no feature engineering. Going forward, these proposed networks can be incorporated as engines in an explanatory data analytics framework of an intelligent sensor network to create value in applications that are yet to be explored.

References

[1] L. Sörnmo and P. Laguna, *Bioelectric Signal Processing in Cardiac and Neurological Applications*, First. Elsevier Academic Press, 2005.

[2] R. Oliver and D. Suendermann, "A First Step towards Eye State Prediction Using EEG," *Proc. of the AIHLS*, 2013.

[3] E. Frank, M. A. Hall, and I. H. Witten, "The WEKA Workbench. Online Appendix for 'Data Mining: Practical Machine Learning Tools and Techniques,'" 2016. .

[4] S. A. Kurtz and J. Hastad, "Computational Limitations of Small-Depth Circuits.," *J. Symb. Log.*, 1988.

[5] Y. Bengio, "Learning Deep Architectures for AI," *Found. Trends® Mach. Learn.*, 2009.

[6] H. Mhaskar, Q. Liao, and T. Poggio, "When and Why Are Deep Networks Better than Shallow Ones ?," *Proc. 31th Conf. Artif. Intell. (AAAI 2017)*, pp. 2343–2349, 2017.

[7] S. Ben-David and S. Shalev-Shwartz, *Understanding Machine Learning: From Theory to Algorithms*. 2014.

[8] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A Simple Way to Prevent Neural Networks from Overfitting," *J. Mach. Learn. Res.*, 2014.

[9] S. Roweis and Z. Ghahramani, "A unifying review of linear gaussian models," *Neural Computation*. 1999.

[10] J. Schmidhuber, "Deep Learning in neural networks: An overview," *Neural Networks*, 2015.

[11] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A Fast Learning Algorithm for Deep Belief Nets," *Neural Comput.*, 2006.

[12] G. E. Hinton, "Training products of experts by minimizing contrastive divergence," *Neural Comput.*, 2002.

[13] F. Cucker and S. Smale, "On the mathematical foundations of learning," *Bull. Am.*

Math. Soc., 2002.

[14] M. Belkin and P. Niyogi, “Laplacian eigenmaps for dimensionality reduction and data representation,” *Neural Comput.*, 2003.

[15] R. G. Andrzejak, K. Lehnertz, F. Mormann, C. Rieke, P. David, and C. E. Elger, “Indications of nonlinear deterministic and finite-dimensional structures in time series of brain electrical activity: Dependence on recording region and brain state,” *Phys. Rev. E - Stat. Physics, Plasmas, Fluids, Relat. Interdiscip. Top.*, vol. 64, no. 6, p. 8, 2001.

[16] D. Anguita, A. Ghio, L. Oneto, X. Parra, and J. L. Reyes-Ortiz, “A Public Domain Dataset for Human Activity Recognition Using Smartphones,” *Eur. Symp. Artif. Neural Networks, Comput. Intell. Mach. Learn.*, no. April, pp. 24–26, 2013.

[17] M. Bächlin *et al.*, “Wearable assistant for Parkinsons disease patients with the freezing of gait symptom,” *IEEE Trans. Inf. Technol. Biomed.*, 2010.

[18] J. Kohlschuetter, J. Peters, and E. Rueckert, “Learning probabilistic features from EMG data for predicting knee abnormalities,” in *IFMBE Proceedings*, 2016.

[19] K. J. Piczak, “ESC: Dataset for Environmental Sound Classification,” *Proc. 23rd ACM Int. Conf. Multimedia, MM 2015*, pp. 1015–1018, 2015.

[20] G. D. Clifford *et al.*, “Classification of Normal / Abnormal Heart Sound Recordings : the PhysioNet / Computing in Cardiology Challenge 2016,” *Comput. Cardiol. (2010).*, pp. 3–6, 2016.

[21] G. A. Diamond, “The wizard of odds: Bayes’ theorem and diagnostic testing,” *Mayo Clinic Proceedings*, 1999.

[22] L. Gordis, *Epidemiology*, 5th ed. Saunders, 2013.

[23] L. Sörnmo and P. Laguna, “Bioelectrical Signal Processing in Cardiac and Neurological Applications,” 2005.

[24] P. Anderer, G. Gruber, S. Parapatics, and G. Dorffner, “Automatic sleep classification according to Rechtschaffen and Kales,” in *Annual International Conference of the IEEE Engineering in Medicine and Biology - Proceedings*, 2007.

[25] N. Kannathal, M. L. Choo, U. R. Acharya, and P. K. Sadasivan, “Entropies for detection of epilepsy in EEG,” *Comput. Methods Programs Biomed.*, 2005.

[26] A. Norali and M. Som, “Surface Electromyography Signal Processing and Application: A Review,” *Int. Conf. Man-Machine Syst.*, no. October, pp. 11–13, 2009.

[27] L. Cao, W. Huang, and F. Sun, “A Deep and Stable Extreme Learning Approach for Classification and Regression,” *Proc. ELM*, vol. 2, pp. 141–150, 2015.

[28] V. N. Vapnik, “An overview of statistical learning theory,” *IEEE Transactions on Neural Networks*. 1999.

[29] K. J. McGarry, S. Wermter, and J. MacIntyre, “Knowledge extraction from radial basis function networks and multilayer perceptrons,” 2003.

[30] M. Gales and S. Young, “The Application of Hidden Markov Models in Speech Recognition,” *Found. Trends® Signal Process.*, vol. 1, no. 3, pp. 195–304, 2007.

[31] K. Hornik, M. Stinchcombe, and H. White, “Multilayer feedforward networks are universal approximators,” *Neural Networks*, 1989.

[32] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning representations by back-propagating errors,” *Nature*, vol. 323, no. 6088, pp. 533–536, Oct. 1986.

[33] A. Waibel, T. Hanazawa, G. E. Hinton, K. Shikano, and K. J. Lang, “Phoneme recognition using time-delay neural networks,” *IEEE Transactions on Acoustics, Speech, and Signal Processing*. 1989.

[34] C. Jin, M. Schenkel, and S. Carlile, “Neural system identification model of human sound localization.,” *J. Acoust. Soc. Am.*, vol. 108, no. 3 Pt 1, pp. 1215–1235, 2000.

[35] Y. Nancy Jane, H. Khanna Nehemiah, and K. Arputharaj, “A Q-backpropagated time delay neural network for diagnosing severity of gait disturbances in Parkinson’s disease,” *J. Biomed. Inform.*, vol. 60, pp. 169–176, 2016.

[36] Y. Bengio, P. Simard, and P. Frasconi, “Learning Long-Term Dependencies with Gradient Descent is Difficult,” *IEEE Trans. Neural Networks*, 1994.

[37] K. Fukushima, “Artificial vision by multi-layered neural networks: Neocognitron and its advances,” *Neural Networks*, 2013.

[38] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proc. IEEE*, 1998.

[39] R. Pascanu, T. Mikolov, and Y. Bengio, “Understanding the exploding gradient problem,” *Proc. 30th Int. Conf. Mach. Learn.*, 2012.

[40] S. Hochreiter and J. Schmidhuber, “Long Short-Term Memory,” *Neural Comput.*, 1997.

[41] K. Cho *et al.*, “Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation,” 2014.

[42] M. Längkvist, L. Karlsson, and A. Loutfi, “A review of unsupervised feature learning and deep learning for time-series modeling,” *Pattern Recognit. Lett.*, 2014.

[43] R. J. Hyndman and A. B. Koehler, “Another look at measures of forecast accuracy,” *Int. J. Forecast.*, 2006.

[44] M. Geurts, G. E. P. Box, and G. M. Jenkins, “Time Series Analysis: Forecasting and Control,” *J. Mark. Res.*, 2006.

[45] L. Bauwens, S. Laurent, and J. V. K. Rombouts, “Multivariate GARCH models: A survey,” *Journal of Applied Econometrics*. 2006.

[46] S. Makridakis, E. Spiliotis, and V. Assimakopoulos, “Statistical and Machine Learning forecasting methods: Concerns and ways forward,” *PLoS One*, 2018.

[47] S. Krstanovic and H. Paulheim, “Ensembles of recurrent neural networks for robust time series forecasting,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2017.

[48] Z. Xing, J. Pei, and E. Keogh, “A brief survey on sequence classification,” *ACM SIGKDD Explor. Newsl.*, 2010.

[49] H. Lütkepohl, *New introduction to multiple time series analysis*. 2005.

[50] F. W. Fairman, “Introduction to dynamic systems: Theory, models and applications,” *Proc. IEEE*, 2008.

[51] L. R. Rabiner and B. H. Juang, “An Introduction to Hidden Markov Models,” *IEEE*

ASSP Mag., 1986.

[52] S. Ding, H. Zhu, W. Jia, and C. Su, "A survey on feature extraction for pattern recognition," *Artificial Intelligence Review*. 2012.

[53] A. Kampouraki, G. Manis, and C. Nikou, "Heartbeat time series classification with support vector machines," in *IEEE Transactions on Information Technology in Biomedicine*, 2009.

[54] A. Boardman, F. S. Schlindwein, A. P. Rocha, and A. Leite, "A study on the optimum order of autoregressive models for heart rate variability," *Physiol. Meas.*, 2002.

[55] I. Popivanov and R. J. Miller, "Similarity search over time-series data using wavelets," *Proc. - Int. Conf. Data Eng.*, 2002.

[56] V. Pichot *et al.*, "Wavelet transform to quantify heart rate variability and to assess its instantaneous changes," *J. Appl. Physiol.*, 2017.

[57] V. P. Nigam and D. Graupe, "A neural-network-based detection of epilepsy," *Neurological Research*. 2004.

[58] P. Y. Zhou and K. C. C. Chan, "A feature extraction method for multivariate time series classification using temporal patterns," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2015.

[59] A. Flexer, G. Gruber, and G. Dorffner, "A reliable probabilistic sleep stager based on a single EEG signal," *Artif. Intell. Med.*, 2005.

[60] J. Grabocka, A. Nanopoulos, and L. Schmidt-thieme, "Classification of Sparse Time Series via Supervised Matrix Factorization," *Proc. Twenty-Sixth AAAI Conf. Artif. Intell.*, 2010.

[61] C. Li, L. Khan, and B. Prabhakaran, "Feature selection for classification of variable length multiattribute motions," in *Multimedia Data Mining and Knowledge Discovery*, 2007.

[62] X. Weng and J. Shen, "Classification of multivariate time series using locality preserving projections," *Knowledge-Based Syst.*, 2008.

[63] X. He and P. Niyogi, “Locality Preserving Projections,” in *Proceedings of Neural Information Processing Systems*, 2003.

[64] E. Keogh and C. A. Ratanamahatana, “Exact indexing of dynamic time warping,” *Knowl. Inf. Syst.*, 2005.

[65] L. Ye and E. Keogh, “Time Series Shapelets: A New Primative for Data Mining,” *Proc. 15th ACM SIGKDD Int. Conf. Knowl. Discov. data Min. - KDD ’09*, 2009.

[66] J. Lines and A. Bagnall, “Time series classification with ensembles of elastic distance measures,” *Data Min. Knowl. Discov.*, 2015.

[67] X. Wang, A. Mueen, H. Ding, G. Trajcevski, P. Scheuermann, and E. Keogh, “Experimental comparison of representation methods and distance measures for time series data,” *Data Min. Knowl. Discov.*, 2013.

[68] A. Bagnall, J. Lines, A. Bostrom, J. Large, and E. Keogh, “The great time series classification bake off: a review and experimental evaluation of recent algorithmic advances,” *Data Min. Knowl. Discov.*, 2017.

[69] G. a Ten Holt, M. J. T. Reinders, and E. a Hendriks, “Multi-Dimensional Dynamic Time Warping for Gesture Recognition,” *Time*, 2007.

[70] H. Kaya and Ş. Gündüz-Öğüdücü, “A distance based time series classification framework,” *Inf. Syst.*, 2015.

[71] S. Gudmundsson, T. P. Runarsson, and S. Sigurdsson, “Support vector machines and dynamic time warping for time series,” in *Proceedings of the International Joint Conference on Neural Networks*, 2008.

[72] B. Jain and S. Spiegel, “Dimension reduction in dissimilarity spaces for time series classification,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2016.

[73] A. Hayashi, Y. Mizuhara, and N. Suematsu, “Embedding Time Series Data for Classification,” 2005.

[74] M. Belkin and P. Niyogi, “Laplacian Eigenmaps and Spectral Techniques for Embedding and Clustering,” *NIPS*, 2001.

[75] Y. Mizuhara, A. Hayashi, and N. Suematsu, “Embedding of time series data by using Dynamic Time Warping distances,” *Syst. Comput. Japan*, 2006.

[76] J. Breneman, “Kernel Methods for Pattern Analysis,” *Technometrics*, 2009.

[77] W. A. Chaovalitwongse and P. M. Pardalos, “On the time series support vector machine using dynamic time warping kernel for brain activity classification,” *Cybern. Syst. Anal.*, 2008.

[78] E. G. Băzăvan, F. Li, and C. Sminchisescu, “Fourier kernel learning,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2012.

[79] H. W. Lin, M. Tegmark, and D. Rolnick, “Why Does Deep and Cheap Learning Work So Well?,” *J. Stat. Phys.*, 2017.

[80] Y. Lecun, Y. Bengio, and G. Hinton, “Deep learning,” *Nature*. 2015.

[81] P. Marius-Constantin, V. E. Balas, L. Perescu-Popescu, and N. Mastorakis, “Multilayer perceptron and neural networks,” *WSEAS Trans. Circuits Syst.*, 2009.

[82] G. Alain *et al.*, “GSNs: generative stochastic networks,” *Inf. Inference*, 2016.

[83] M. Längkvist, L. Karlsson, and A. Loutfi, “Sleep Stage Classification Using Unsupervised Feature Learning,” *Adv. Artif. Neural Syst.*, 2012.

[84] M. Zębik, M. Korytkowski, R. Angryk, and R. Scherer, “Convolutional neural networks for time series classification,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2017.

[85] Z. Wang, W. Yan, and T. Oates, “Time series classification from scratch with deep neural networks: A strong baseline,” in *Proceedings of the International Joint Conference on Neural Networks*, 2017.

[86] Y. Zheng, Q. Liu, E. Chen, Y. Ge, and J. L. Zhao, “Exploiting multi-channels deep convolutional neural networks for multivariate time series classification,” *Front. Comput. Sci.*, 2016.

[87] S. Zagoruyko and N. Komodakis, “Wide Residual Networks,” in *Procedings of the*

British Machine Vision Conference 2016, 2016.

[88] E. D. Munz, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift Sergey," *Nervenheilkunde*, 2017.

[89] F. Karim, S. Majumdar, H. Darabi, and S. Chen, "LSTM Fully Convolutional Networks for Time Series Classification," *IEEE Access*, 2017.

[90] D. Bacciu, P. Barsocchi, S. Chessa, C. Gallicchio, and A. Micheli, "An experimental characterization of reservoir computing in ambient assisted living applications," *Neural Comput. Appl.*, 2014.

[91] M. Lukoševičius, "A practical guide to applying echo state networks," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, 2012.

[92] H. Jaeger, M. Lukoševičius, D. Popovici, and U. Siewert, "Optimization and applications of echo state networks with leaky- integrator neurons," *Neural Networks*, 2007.

[93] M. G. Baydogan, G. Runger, and E. Tuv, "A bag-of-features framework to classify time series," *IEEE Trans. Pattern Anal. Mach. Intell.*, 2013.

[94] H. Deng, G. Runger, E. Tuv, and M. Vladimir, "A time series forest for classification and feature extraction," *Inf. Sci. (Ny).*, 2013.

[95] K. Buza, "Fusion methods for time-series classification," *Update*, 2011.

[96] R. J. Kate, "Using dynamic time warping distances as features for improved time series classification," *Data Min. Knowl. Discov.*, 2016.

[97] A. Bagnall, J. Lines, J. Hills, and A. Bostrom, "Time-series classification with COTE: The collective of transformation-based ensembles," in *2016 IEEE 32nd International Conference on Data Engineering, ICDE 2016*, 2016.

[98] J. Lines, S. Taylor, and A. Bagnall, "HIVE-COTE: The hierarchical vote collective of transformation-based ensembles for time series classification," in *Proceedings - IEEE International Conference on Data Mining, ICDM*, 2017.

[99] C. Yanping *et al.*, "UCR Time Series Classification Archive,"

URL:www.cs.ucr.edu/~eamonn/time_series_data/, 2015.

[100] J. Hills, J. Lines, E. Baranauskas, J. Mapp, and A. Bagnall, "Classification of time series by shapelet transformation," *Data Min. Knowl. Discov.*, 2014.

[101] L. Deng and J. C. Platt, "Ensemble deep learning for speech recognition," in *Proceedings of the Annual Conference of the International Speech Communication Association, INTERSPEECH*, 2014.

[102] L. P. Jin and J. Dong, "Ensemble Deep Learning for Biomedical Time Series Classification," *Comput. Intell. Neurosci.*, 2016.

[103] J.-W. ZHANG, X. LIU, and J. DONG, "CCDD: AN ENHANCED STANDARD ECG DATABASE WITH ITS MANAGEMENT AND ANNOTATION TOOLS," *Int. J. Artif. Intell. Tools*, 2012.

[104] J. Zhao, X. Xie, X. Xu, and S. Sun, *Multi-view Learning Overview: Recent Progress and New Challenges*, vol. 38. 2017.

[105] A. Blum and T. Mitchell, "Combining labeled and unlabeled data with co-training," in *Proceedings of the eleventh annual conference on Computational learning theory - COLT' 98*, 1998.

[106] S. Sun, "A survey of multi-view machine learning," *Neural Computing and Applications*. 2013.

[107] J. Liu, C. Wang, J. Gao, and J. Han, "Multi-View Clustering via Joint Nonnegative Matrix Factorization," in *Proceedings of the 2013 SIAM International Conference on Data Mining*, 2013.

[108] I. Muslea, S. Minton, and C. A. Knoblock, "Active learning with multiple views," *J. Artif. Intell. Res.*, 2006.

[109] B. Tan, E. Zhong, E. W. Xiang, and Q. Yang, "Multi-transfer: Transfer learning with multiple views and multiple sources," *Stat. Anal. Data Min.*, 2014.

[110] C. O. Sakar, O. Kursun, and F. Gurgen, "Ensemble canonical correlation analysis," *Appl. Intell.*, 2014.

[111] T. Niu, S. Zhu, L. Pang, and A. Elsaddik, "Sentiment analysis on multi-view social

data," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2016.

[112] W. Wang, R. Arora, K. Livescu, and J. A. Bilmes, "Unsupervised learning of acoustic features via deep canonical correlation analysis," in *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, 2015.

[113] J. Ngiam, A. Khosla, M. Kim, J. Nam, H. Lee, and A. Y. Ng, "Multimodal Deep Learning," *Proc. 28th Int. Conf. Mach. Learn.*, pp. 689–696, 2011.

[114] zydyr@163. co. Yan Zhang1, lvdanjv@hotmail. co. Danjv Lv1, and ylzhao@vip. sina. co. Yili Zhao1, "Multiple-View Active Learning for Environmental Sound Classification.," *International Journal of Online Engineering*. 2016.

[115] T. Bänziger, D. Grandjean, and K. R. Scherer, "Emotion Recognition From Expressions in Face, Voice, and Body: The Multimodal Emotion Recognition Test (MERT)," *Emotion*, 2009.

[116] K. Nigam and R. Ghani, "Analyzing the effectiveness and applicability of co-training," in *Proceedings of the ninth international conference on Information and knowledge management - CIKM '00*, 2000.

[117] A. Dixit, *Ensemble Machine Learning*. Packt Publishing Ltd., 2017.

[118] L. Breiman, "Bagging predictors," *Mach. Learn.*, vol. 24, no. 2, pp. 123–140, 1996.

[119] L. G. Valiant, "A theory of the learnable," *Commun. ACM*, vol. 27, no. 11, pp. 1134–1142, 1984.

[120] Y. Freund and R. E. Schapire, "A desicion-theoretic generalization of on-line learning and an application to boosting," vol. 139, pp. 23–37, 1995.

[121] J. H. Friedman and J. H., "Stochastic gradient boosting," *Comput. Stat. Data Anal.*, vol. 38, no. 4, pp. 367–378, 2002.

[122] D. H. Wolpert, "Stacked Generalization," *Neural Networks*, vol. 5, no. 2, 1992.

[123] A. Zien and C. S. Ong, "Multiclass Multiple Kernel Learning," *Proc. ICML*, vol. 1, no. 2, 2007.

[124] M. Gönen and E. Alpaydın, “Multiple Kernel Learning Algorithms,” *J. Mach. Learn. Res.*, vol. 12, pp. 2211–2268, 2011.

[125] N. Aronszajn, “Theory of Reproducing Kernels,” *Trans. Am. Math. Soc.*, vol. 68, no. 3, pp. 337–404, 1950.

[126] D. Erhan, Y. Bengio, A. Courville, Pierre-Antoine Manzagol, Pascal Vincent, and S. Bengio, “Why Does Unsupervised Pre-training Help Deep Learning?,” *J. Mach. Learn. Res.*, 2010.

[127] Y. Chauvin and D. E. Rumelhart, *Backpropagation : theory, architectures, and applications*. 1995.

[128] E. Keogh, S. Chu, D. Hart, and M. Pazzani, “An online algorithm for segmenting time series,” 2002.

[129] I. Sutskever, J. Martens, G. E. Dahl, and G. E. Hinton, “On the importance of initialization and momentum in deep learning,” *Proc. 30th Int. Conf. Mach. Learn. Jmlr W&Cp*, 2013.

[130] A. Fischer and C. Igel, “An introduction to restricted Boltzmann machines,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2012.

[131] “Greedy Layer-Wise Training of Deep Networks,” in *Advances in Neural Information Processing Systems 19*, 2018.

[132] Y. Bengio and O. Delalleau, “Justifying and generalizing contrastive divergence,” *Neural Computation*. 2009.

[133] P. A. Cariani, “Neural timing nets,” *Neural Networks*, 2001.

[134] M. Li, T. Zhang, Y. Chen, and A. J. Smola, “Efficient mini-batch training for stochastic optimization,” 2014.

[135] D. Masters and C. Luschi, “Revisiting Small Batch Training for Deep Neural Networks,” pp. 1–18, 2018.

[136] X. Glorot, A. Bordes, and Y. Bengio, “ReLU,” *AISTATS ’11 Proc. 14th Int. Conf. Artif. Intell. Stat.*, 2011.

[137] T. Xia, D. Tao, T. Mei, and Y. Zhang, “Multiview spectral embedding,” *IEEE Trans. Syst. Man, Cybern. Part B Cybern.*, vol. 40, no. 6, pp. 1438–1446, 2010.

[138] F. Cucker and S. Smale, “On the mathematical foundations of learning,” *Bull. Am. Math. Soc.*, vol. 39, no. 1, pp. 1–49, 2002.

[139] U. Naftaly†, N. Intrator‡, and D. Horn§, “Optimal ensemble averaging of neural networks,” *Netw. Comput. Neural Syst.*, 1997.

[140] A. P. Dempster, N. M. Laird, and D. B. Rubin, “Maximum likelihood from incomplete data via the EM algorithm,” *J. R. Stat. Soc. Ser. B Methodol.*, vol. 39, no. 1, pp. 1–38, 1977.

[141] W. Martin and P. Flandrin, “Wigner-Ville Spectral Analysis of Nonstationary Processes,” *IEEE Trans. Acoust.*, 1985.

[142] N. Rehman and D. P. Mandic, “Multivariate empirical mode decomposition,” *Proc. R. Soc. A Math. Phys. Eng. Sci.*, 2010.

[143] V. Tiwari, “MFCC and its applications in speaker recognition,” *Int. J. Emerg. Technol.*, 2010.

[144] L. Hogben, “Spectral graph theory and the inverse eigenvalue problem of a graph,” in *Electronic Journal of Linear Algebra*, 2005.

[145] M. Wang, X. S. Hua, X. Yuan, Y. Song, and L. R. Dai, “Optimizing multi-graph learning: towards a unified video annotation scheme,” *Proc. 15th Int. Conf. Multimed.*, pp. 862–871, 2007.

[146] N. Hatami, Y. Gavet, and J. Debayle, “Classification of Time-Series Images Using Deep Convolutional Neural Networks,” 2017.

[147] E. Dua, D. and Karra Taniskidou, “UCI Machine Learning Repository [http://archive.ics.uci.edu/ml],” *Irvine, CA: University of California, School of Information and Computer Science.*, 2017. .

[148] J. G. Cleary and L. E. Trigg, “K*: An Instance-based Learner Using an Entropic Distance Measure,” in *Machine Learning Proceedings 1995*, 1995.

[149] T. Wang, S. U. Guan, K. L. Man, and T. O. Ting, “EEG eye state identification using

incremental attribute learning with time-series classification," *Math. Probl. Eng.*, 2014.

[150] I. Güler and E. D. Übeyli, "Adaptive neuro-fuzzy inference system for classification of EEG signals using wavelet coefficients," *J. Neurosci. Methods*, 2005.

[151] J. S. R. Jang, "ANFIS: Adaptive-Network-Based Fuzzy Inference System," *IEEE Trans. Syst. Man Cybern.*, 1993.

[152] V. Srinivasan, C. Eswaran, and A. N. Sriraam, "Artificial neural network based epileptic detection using time-domain and frequency-domain features," *J. Med. Syst.*, 2005.

[153] L. Wang *et al.*, "Automatic epileptic seizure detection in EEG signals using multi-domain feature extraction and nonlinear analysis," *Entropy*, 2017.

[154] D. M. Karantonis, M. R. Narayanan, M. Mathie, N. H. Lovell, and B. G. Celler, "Implementation of a Real-Time Human Movement Classifier Using a Triaxial Accelerometer for Ambulatory Monitoring," *IEEE Trans. Inf. Technol. Biomed.*, vol. 10, no. 1, pp. 156–167, 2006.

[155] D. Anguita, A. Ghio, L. Oneto, X. Parra, and J. L. Reyes-Ortiz, "Human activity recognition on smartphones using a multiclass hardware-friendly support vector machine," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2012.

[156] B. Romera-Paredes, M. S. H. Aung, and N. Bianchi-Berthouze, "A One-Vs-One classifier ensemble with majority voting for activity recognition," in *ESANN 2013 proceedings, 21st European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning*, 2013.

[157] B. Almaslukh, A. Jalal, and A. Abdelmonim, "An Effective Deep Autoencoder Approach for Online Smartphone-Based Human Activity Recognition," *Int. J. Comput. Sci. Netw. Secur.*, 2017.

[158] N. Giladi, R. Kao, and S. Fahn, "Freezing phenomenon in patients with parkinsonian syndromes," *Mov. Disord.*, 1997.

[159] L. Dauwerse, A. Hendrikx, K. Schipper, C. Struiksma, and T. A. Abma, "Quality-of-

life of patients with Parkinson's disease," *Brain Inj.*, 2014.

[160] J. D. Schaafsma, Y. Balash, T. Gurevich, A. L. Bartels, J. M. Hausdorff, and N. Giladi, "Characterization of freezing of gait subtypes and the response of each to levodopa in Parkinson's disease," *Eur. J. Neurol.*, 2003.

[161] H. Braak, E. Ghebremedhin, U. Rüb, H. Bratzke, and K. Del Tredici, "Stages in the development of Parkinson's disease-related pathology," *Cell and Tissue Research*. 2004.

[162] B. R. Bloem, J. M. Hausdorff, J. E. Visser, and N. Giladi, "Falls and freezing of Gait in Parkinson's disease: A review of two interconnected, episodic phenomena," *Movement Disorders*. 2004.

[163] I. Lim *et al.*, "Effects of external rhythmical cueing on gait in patients with Parkinson's disease: A systematic review," *Clinical Rehabilitation*. 2005.

[164] M. Bächlin, D. Roggen, M. Plotnik, J. M. Hausdorff, N. Giladi, and G. Tröster, "Online detection of freezing of gait in Parkinson's disease patients: A performance characterization," in *BODYNETS 2009 - 4th International ICST Conference on Body Area Networks*, 2011.

[165] S. T. Moore, H. G. MacDougall, and W. G. Ondo, "Ambulatory monitoring of freezing of gait in Parkinson's disease," *J. Neurosci. Methods*, 2008.

[166] S. Mazilu *et al.*, "Online Detection of Freezing of Gait with Smartphones and Machine Learning Techniques," in *Proceedings of the 6th International Conference on Pervasive Computing Technologies for Healthcare*, 2012.

[167] V. Balasubramanyam and K. Balachander, "Evaluation of knee activities using EMG signals for pre-predicting lower limb dystonia diseases," *Int. J. Intell. Eng. Syst.*, 2018.

[168] C. D. Joshi, U. Lahiri, and N. V. Thakor, "Classification of gait phases from lower limb EMG: Application to exoskeleton orthosis," in *IEEE EMBS Special Topic Conference on Point-of-Care (POC) Healthcare Technologies: Synergy Towards Better Global Healthcare, PHT 2013*, 2013.

[169] Z. Yi *et al.*, "Extracting time-frequency feature of single-channel vastus medialis EMG

signals for knee exercise pattern recognition," *PLoS One*, 2017.

[170] H. B. Sailor, D. M. Agrawal, and H. A. Patil, "Unsupervised filterbank learning using Convolutional Restricted Boltzmann Machine for environmental sound classification," in *Proceedings of the Annual Conference of the International Speech Communication Association, INTERSPEECH*, 2017.

[171] Y. Tokozume, Y. Ushiku, and T. Harada, "Between-Class Learning for Image Classification," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2018.

[172] R. N. Tak, D. M. Agrawal, and H. A. Patil, "Novel Phase Encoded Mel Filterbank Energies for Environmental Sound Classification," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2017.

[173] A. Kumar, M. Khadkevich, and C. Fugen, "Knowledge Transfer from Weakly Labeled Audio Using Convolutional Neural Network for Sound Events and Scenes," in *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, 2018.

[174] A. L. Goldberger *et al.*, "PhysioBank, PhysioToolkit, and PhysioNet: Components of a New Research Resource for Complex Physiologic Signals," *Circulation*, vol. 101, no. 23, pp. e215–e220, 2000.

[175] C. Potes, S. Parvaneh, A. Rahman, and B. Conroy, "Ensemble of feature-based and deep learning-based classifiers for detection of abnormal heart sounds," in *Computing in Cardiology*, 2016.

[176] M. Zabihi, A. B. Rad, S. Kiranyaz, M. Gabbouj, and A. K. Katsaggelos, "Heart sound anomaly and quality detection using ensemble of neural networks without segmentation," in *Computing in Cardiology*, 2016.

[177] E. Kay and A. Agarwal, "DropConnected neural networks trained on time-frequency and inter-beat features for classifying heart sounds," *Physiol. Meas.*, 2017.

[178] I. J. Diaz Bobillo, "A tensor approach to heart sound classification," in *Computing in Cardiology*, 2016.

[179] F. Plesinger, J. Jurco, P. Jurak, and J. Halamek, “Discrimination of normal and abnormal heart sounds using probability assessment,” in *Computing in Cardiology*, 2016.

[180] S. J. Pan and Q. Yang, “A survey on transfer learning,” *IEEE Transactions on Knowledge and Data Engineering*. 2010.