3. DEEP LEARNING

This chapter presents theoretical background about deep learning which is adopted in this research work for building muscular dystrophy disease identification model through self-taught learning. Introduction to deep learning, motivations to construct models using deep learning are explained in this chapter. The concepts of individual neurons and arranging them into basic layers to form feedforward neural networks are presented in section 3.1. Deep neural network architecture is explained in section 3.2. The types of layers used to construct networks for processing spatial data such as Deep Neural Network classifier, Autoencoder and Stacked autoencoder are explained in section 3.3, 3.4 and 3.5.

Deep Learning, also known as deep structured learning or hierarchical learning which is a division of machine learning based on a collection of algorithms that endeavor to model high level of abstractions in data [85]. Deep Learning is an emerging area in research based on machine learning, which has an aim to move machine learning closer like artificial intelligence. Machine learning works on human-designed representations and input features. Notable attributes or representations are learned automatically with the advent of representation learning. Multiple levels of representations are acquired through deep learning algorithms by increasing the level of abstraction. Handcrafting features is time-consuming and the features that are extracted are either over-specified or lacking [86]. To move beyond handcrafted features, simple machine learning and depth of learning is required. Deep learning provides a way of training computers to develop representations for learning and reasoning like human beings. The basic five reasons to explore deep learning are,

- Learning representations
- The need for distributed representations
- Unsupervised feature and weight learning
- Learning multiple levels of representation
- Deep Learn Models have interesting performance characteristics

The motivation to construct models using deep learning networks is to make computers resemble like human brains as the human brains have a deep architecture. Humans categorize their ideas in hierarchical order, through composition of ideas learning from simpler concepts to more abstract ones. The same concepts are achieved through computer from distributed

representations to accomplish non-local simplification, intermediate representations of data to allow sharing by breaking up the solutions into several levels of abstraction and processing [87].

3.1 Feed Forward Neural Networks

As portrayed in Chapter 2, Artificial neural networks (ANNs) are numeric models that are modeled based on biological neural networks. ANNs receive input signals on multiple connections and only produce an output signal if a weighted sum of the inputs reaches a certain threshold. A parallel architecture is attained using ANN by providing the inputs and outputs in parallel way. Artificial neural networks are eminent along with the paths between neurons that are tuned by a learning algorithm that learns from past data to improvise the model. The appropriate cost function is chosen to learn the optimal solution to the problem.

Mathematically, a single artificial neuron with K input values represents a nonlinear function g: $R^K \rightarrow R$, parameterized by a weight vector w, a bias b, and a non-linear activation function σ.

$$g(x) = \sigma\left(\sum_{K=0}^{K}(\mathrm{w}_k \mathrm{x}_k + \mathrm{b})\right) = \sigma(w^T x + b) \tag{3.1}$$

A set of neurons with a common activation function σ can be arranged into a layer. Each neuron of a layer l feeds its outputs only into neurons of layer l +1, constituting the feedforward property. A network of multiple consecutive layers connected in this way is called a feedforward neural network. The number of neurons in each layer determines the width of each layer, while the number of layers determines the depth of the network. Due to the feedforward property, the outputs of all neurons of a layer l are computed in parallel. A layer l with $K^{(l)}$ neurons operating on an input vector $\boldsymbol{x}^{(l-1)}$ thus represents a non-linear function $f^{(l)}$**:** $R^{K(l-1)} \rightarrow R^{K(l)}$, producing an output vector $\boldsymbol{x}^{(l)}$:

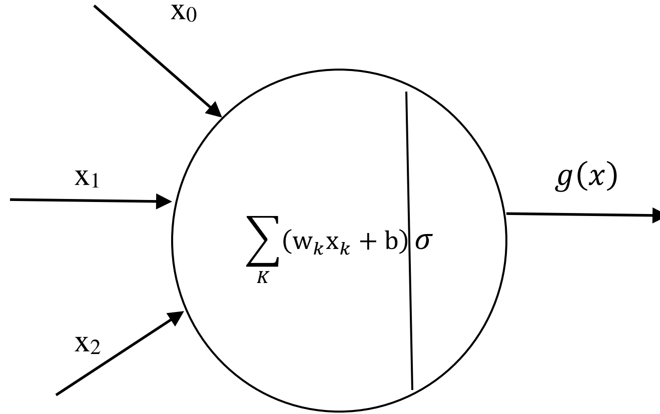

Fig.3.1 Artificial Neuron

As described in eq. (3.1), the neuron computes the weighted sum of an input vector x =(x_0, x_1, x_2) T, adds a bias value, applies an activation function σ, and outputs the resulting scalar value.

$$x^{(l)} = f^{l}(x^{l-1})$$

(3.2)

The layer function is defined as:

$$f^{l}(x) = \sigma^{(l)}(W^{(l)}x + b^{(l)})$$

(3.3)

The weight matrix W(l) and the bias vector $b^{(l)}$ of a layer are constructed from the weight vectors w and bias values b of the individual neurons comprising the layer. For a network of L consecutive layers, the overall network parameters θ are given by the individual layer weights $W^{(l)}$ and biases $b^{(l)}$. Explicitly specifying the parameters θ, the network thus represents a function $y = f(x; \theta)$ and is written as a composition of its layers as f: $R^{K(0)} \rightarrow R^{K(L)}$.

The output of the last layer of a network, $\boldsymbol{x}^{(L)} = f^{(L)}(\boldsymbol{x}^{(L-1)})$, is equivalent to the output of the whole network $\boldsymbol{y} = f(\boldsymbol{x}; \boldsymbol{\theta})$. This layer is thus called the output layer. These output values are used for regression or as class scores for classification, with no activation function σ applied in the output layer. The network input $\boldsymbol{x}$is equivalent to the first layer's input $\boldsymbol{x}^{(0)}$, that is represented as a separate input layer. The intermediate neural layers are called hidden layers.

To define the parameter values of the models such as neuron path, the weights that are to be assigned to achieve target values, tuning parameters such as the learning rate are determined. Defining the parameter values are done with the optimization techniques such as gradient descent or stochastic gradient descent. High performance is achieved with ANN by means of

these optimization techniques which aids in making the ANN solution be as close to the ideal solution [88]. Layers of artificial neurons, computational input units and an activation function with a threshold are required for building models using artificial neural networks.

In a simple ANN model, the input layer is the first layer, next followed by one hidden layer, then an output layer and each layer contains one or more neurons. The models become complex when the number of hidden layers is increased, which also increases the number of neurons in any given layer and the number of paths between neurons and hence the abstraction is also increased.

A deep neural network (DNN) is an artificial neural network (ANN) with multiple hidden layers of units between the input and output layers [89, 90] Similar to shallow ANNs, DNNs can model complex non-linear relationships. Compositional models are generated by DNN architectures by expressing the object as a arrangement of image primitives [91]. Normally, DNNs are designed as feedforward networks, but the research endeavors has applied LSTM [92, 93] recurrent neural networks in applications such as language modeling and attained success [94, 95]. Applications of Convolutional deep neural networks (CNNs) in the area of computer vision is familiar in the research frontier [96]. CNNs have shown their success by building acoustic modeling for automatic speech recognition (ASR) [97].

3.2 Deep Neural Network Architecture

Neural networks are modeled similar to human brains in order to identify patterns. Some of the patterns are numerical, images, sound and time series, which can be identified, while clustering, and classification of data through neural networks. Based on similarities among the inputs the unlabeled data are clustered using neural networks, and classification of data is done when it accepts labeled dataset. Deep learning maps input into output by seeking the association, also it projects to estimate the function $f(x) = y$ between input x and output y.

Deep learning, also known as the stacked neural networks are arranged into several layers that are made of nodes. The algorithm can be processed by integrating the weights to the input of the node with their weights to boost or lessen that input. Summations of the input-weights are passed into the activation function through nodes to determine the progresses though the network to achieve an eventual outcome, an act of classification.

Deep-learning networks are prominent by its deepness that is the number of hidden layers through which data passes in the network while recognizing patterns. Shallow nets possess one input, one output and one hidden layer where the deep networks consists of more than one hidden layer other than the input and output layers. In accordance with the output of the previous layer, distinct features are trained out in every layer to build a model in the deep networks. Addition of more hidden nodes increases the levels of abstraction that learn more intricate features differentiated by the nodes, which are summed up and combines the features from the previous layer. Handling very large, high-dimensional data sets are done by increasing the level of abstraction in deep-learning networks with numerous set of parameters that pass through nonlinear functions [98].

Self-extraction of features is done in deep learning networks where in the traditional machine learning algorithms there is a necessity of handcrafted features to build models [99]. Self-extracted features are vital when higher-level abstractions are involved, since it is tedious to specify the raw inputs in explicit format. Self-learning of dominant features is very important, as there is a tremendous growth in amount of data and range of applications in machine learning methods. The number of levels of composition is the depth of architecture that refers to non-linear operations in the learning function [100].

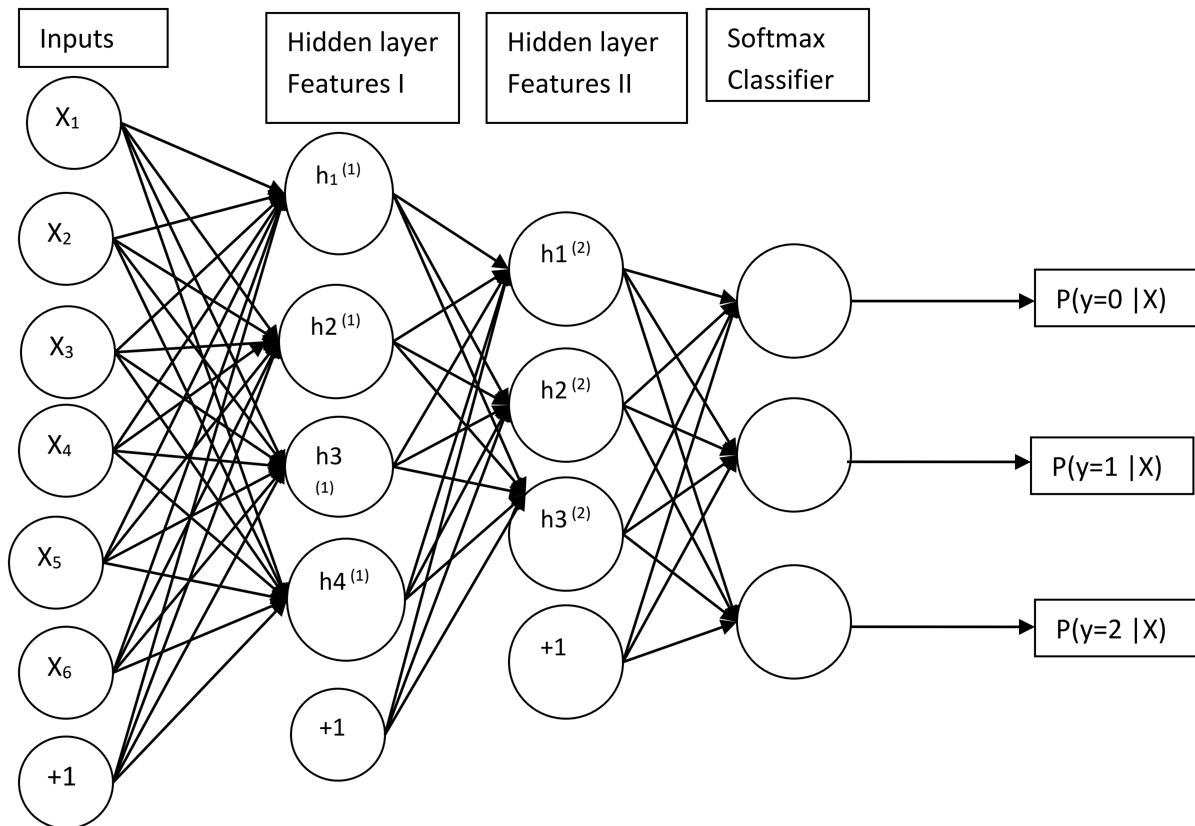

Fig.3.2 Architecture of Deep Neural Network Par diagram

3.3 Deep Neural Network Classifier

Deep neural network involves unsupervised learning between two adjacent layers, to minimize the requirement for the supervised learning by backward propagation. The properties of feed forward neural network are summarized below.

- An input, output, and one or more hidden layers.
- Each unit is a single perceptron .
- The units of the input layer serve as inputs for the hidden layer, which are then, become the inputs to the output layer.
- Each connection between two neurons is implied by a weight w, similar to the perceptron weights.
- Each unit of layer t is connected to every unit of the previous layer t - 1.
- To process input data, clinch the input vector to the input layer by setting the values of the vector as outputs for each of the input units. In a three layered input unit case, 3 input units

are involved and hence, the network can process a 3-dimensional input vectors. For example, consider an input vector is [7, 1, 2], then the input layer is set to 7, the middle layer to 1, and so on. These values are then propagated forward to the hidden units using the weighted sum transfer function for each hidden unit.

- The output layer calculates it's outputs in the same way as the hidden layer. The result of the output layer is the output of the network.

Training

Back propagation is the most common deep learning algorithm to train the multilayer perceptrons that is used for supervised training. The basic procedure is as follows:

1. The network is propagated forward by presenting a training sample
2. The mean squared error is the output error and is calculated as:

$$E = \frac{1}{2}(t - y)^2 \tag{3.4}$$

where t is the target value and y is the actual network output. MSE is a best method for calculating errors than other measures.

3. Stochastic gradient descent method is used to minimize the network error

Stochastic Gradient Descent

The full training set is used in the batch methods for computing parameters and to calculate the next update to parameters in each iteration and tend to converge very well to local optima. As there are only few hyper-parameters to tune, it is a straight forward method to get implemented. The computing cost and gradient for the entire training set is slow and intractable for a single machine to process a dataset with large size [101]. One more issue with batch optimization methods is that they do not incorporate new data in an online setting.

Both of these issues can be addressed using Stochastic Gradient Descent (SGD) addresses. The use of SGD in the neural network setting aids in setting down the high cost of running back propagation over the whole training set. This cost can be overcome and lead to fast convergence using SGD.

The standard gradient descent algorithm with the parameters θ of the objective J (θ) as,

$$\emptyset = \emptyset - \alpha \nabla_{\emptyset} E[J(\emptyset)]$$

(3.5)

The new update of the SGD is given by the following equation, where the expectation in the update used few training examples.

$$\emptyset = \emptyset - \alpha \nabla_{\emptyset} J\left(\emptyset; x^{(i)}, y^{(i)}\right)$$

(3.6)

With a pair ($x^{(i)}$, $y^{(i)}$) from the training set.

Generally each parameter updated in SGD is computed with a few training examples. There are two reasons for this purpose such as, first this reduces the variance in the parameter update and that leads to more stable convergence, second allows the computation with highly optimized matrix operations which are used in a well vectorized computation of the cost and gradient [102].

In SGD the learning rate α is typically much smaller than a corresponding learning rate in batch gradient descent because there is much more variance in the update. Choosing the proper learning rate and schedule is difficult. Using a constant learning rate in the initial epoch or two of training gives a stable convergence and then halve the value of the learning rate slows down the convergence. When there is a change in objective between epochs and if it is below a small threshold, then evaluate a held out set after each epoch and anneal the learning rate. This tends to give good convergence to a local optima. Another commonly used schedule is to anneal the learning rate at each iteration t as a/b+t where a and b dictate the initial learning rate and when the annealing begins respectively. The more vital point regarding SGD is the order in which the data presented to the algorithm. The gradient are biased and lead to poor convergence when the data is provided in meaningful order. Shuffling the data randomly before each epoch on training will avoid convergence.

Issues in Large Networks

A neural network can have more than one hidden layer, new abstractions are built on top of previous layers that makes the learning better with larger networks. Increasing the number of hidden layers leads to two issues such as vanishing gradients and overfitting [103].

Vanishing gradients

Adding more and more hidden layers makes the backpropagation less useful while sending information to the lower level layers. The gradients begin to vanish and become small relative to the weights of the networks, when the information is passed back to the network. The vanishing gradient problem is one of the difficulty faced while training an artificial neural network with gradient-based learning methods and backpropagation [104]. With respect to the current weight in each iteration of training, the neural network's weights receives an update proportional to the gradient of the error function. Gradients are computed by using the chain rule with the activation functions such as the hyperbolic tangent function, which have gradients in the range $(-1, 1)$.

During neural network training with backpropagation, the local minimum of the error function is found by iteratively taking small steps in the direction of the negative error derivative with respect to networks weights i.e. gradients. With each subsequent layer the magnitude of the gradients gets exponentially smaller and vanished by making the steps very small which results in very slow learning of the weights in the lower layers of a deep network. One of the factors causing the gradients to shrink when going through layers are the activation function derivatives. For example the magnitude of sigmoid derivative is well below 1.0 in the whole range of the function and makes the gradients vanish quite fast.

Overfitting: Overfitting describes the experience of fitting the training data very close, with the hypotheses that are too complex. Neural networks that are too complex to memorize the data, therefore overfitting arises and it is overcome with better cost by memorizing instead of generalizing through dropout and minimize model complexity [105].

Dropout: Dropout randomly receives some data by adding null noise to the samples. Therefore, in this dropout method the model will be able to memorize only random sections of each sample therefore it will be forced to generalize in order to get a good prediction accuracy. A 0.5 to 0.75 dropout rate between fully connected layers is going to work quite well and a 0.1–0.3 dropout rate would typically do better for the input layer [106].

- ***Minimize model complexity***

It is proven that too complex models can have better accuracy but it comes with a higher chance to overfit. If the cross-validation set is not large enough and not perfectly sampled, then smaller, less complex model can be chosen [107].

Training error vs. Test error

If the training error is close to 0 and much larger test error, it typically means that the model is overfitting. If problems such as dropout and model complexity problem is already solved and still the problem arises then more samples and features should be used to overcome the error.

3.4 Auto Encoder

An autoencoder, autoassociator or Diabolo network is an artificial neural network used for unsupervised learning [108,109]. The aim of an autoencoder is to learn features for a dataset for the principle of dimensionality reduction. The generative models of data are done using the concept of autoencoder [110]. Hinton and the PDP group [111,112] address the problem of backpropagation, by using the input data, who are the first to introduce autoencoders in the 1980s. To produce global learning and intelligent behavior autoencoders provide one of the fundamental paradigms for unsupervised learning in a self-organized manner Together with Hebbian learning rules [113, 114].

The simplest form of an autoencoder is a feedforward, which is a non-recurrent neural network that is similar to the multilayer perceptron (MLP) having an input layer, an output layer and one or more hidden layers. Instead of predicting the target value Y for the given inputs X, the output layer will have the same number of nodes as the input layer with the purpose of reconstructing its own inputs. For transfer learning and other tasks in deep architectures, autoencoders play a fundamental role in unsupervised learning.

Autoencoders are simple learning circuits that translate inputs into outputs with the least possible amount of alteration. Restricted Boltzmann Machines (RBMS) is one of the autoencoders, that are stacked and trained bottom up in unsupervised fashion, followed by a supervised learning phase to train the top layer and fine-tune the entire architecture [115].

The main aim of autoencoder is to diminish the dimensionality of the represented data. As the gene sequential data is multidimensional data the autoencoder is considered here to decrease the dimensionality. Training of the autoencoder is done by diminishing the reconstruction error between the input data output layer, and the activation values of the hidden layer are measured as a feature vector corresponding to the input data. Unsupervised greedy layer-wise training is

applied on the raw features though autoencoder to learn more features. An autoencoder always consists of two parts, the encoder and the decoder. Auto-encoder is a neural network trained to compute a representation of the input from which it can be reconstructed with as much accuracy as possible [116]. Encoding of an input vector x $\in R^N$ in the autoencoder is done by applying a linear transformation and a nonlinear activation function to x:

$$h = \sigma(w_1 + b_1) \tag{3.7}$$

Where $w_1 \in R^{H \times N}$ is a weight matrix, $b_1 \in R^H$ is a bias, σ (t) = $(1 + \exp(xt))^{-1}$ is a logistic sigmoid function, and h $\in R^H$ represents activation values of the hidden layer. Decoding of the obtained vector h is done by another transformation with a separate weight matrix W_2 and bias b_2:

$$X' = \sigma(W_2^T h + b_2) \tag{3.8}$$

Where X' is a reconstruction of the vector x. The reconstruction error can be written a

$$L(X, X') = \frac{1}{2}\sum_{i=1}^{m} \|X_i - X_i'\|_2^2, \tag{3.9}$$

Where X = $\{X_i\}_{i=1}^{m}$ and $X' = \{X'_i\}_{i=1}^{m}$ are the training and reconstructed data respectively.

The summarization of the training algorithm for an autoencoder is given as,
For each input x,

- A feed-forward pass to be computed by passing activations at all hidden layers, in order to obtain an output x' at the output layer
- The deviation of x' is measured from the input x using squared error function

- The measured error are backpropagated through the net and weights of the nodes are updated

Conjugate gradient method, Steepest descent are the backpropagation variants that are used to train an autoencoder. Usage of autoencoder is effective to reduce the dimensions, but there are some basic issues while using backpropagation to train networks with several hidden layers. When the errors are backpropagated to the first few layers, they become minute and insignificant, which means that the network almost always learn to reconstruct the average of all the training data. This problem can be solved with advanced backpropagation methods such as the conjugate gradient method, but there are issues like very slow learning process and poor solutions.

The problems faced by conjugate gradient method, is solved by using initial weights that approximate the final solution. Pretraining is the process of finding the initial weights. A pretraining technique for training many-layered deep autoencoders involves each neighbouring set of two layers as a restricted Boltzmann machine so that the pretraining approximates a good solution and then the results are fine-tuned using a backpropagation technique.

3.5 Stacked Auto Encoder

Training each layer is a greedy layer wise approach for pretraining a deep network. A stacked autoencoder is a neural network consisting of multiple layers of sparse autoencoders in which the outputs of each layer is connected to the inputs of the consecutive layer. Consider a stacked autoencoder with n layers, where the encoding step for the stacked autoencoder is given by running the encoding step of each layer in forward order:

$$a^{l} = f(Z^{(l)})$$

$$(3.10) Z^{(l+1)} = W^{(l,1)} a^{(l)} + b^{(l,1)}$$

(3.11)

The features from the stacked autoencoder is used for classification problems by feeding *a* (*n*) to a classifier.

Training

A good way to obtain good parameters for a stacked autoencoder is to use greedy layer-wise training. The first layer is first trained using raw input to obtain the parameters $W^{(1, 1)}$,$W^{(1, 2)}$,$b^{(1, 1)}$, $b^{(1, 2)}$. The raw input is transformed into a vector consisting of activation of the hidden units, A in the first layer.

The second layer is trained on this vector to obtain parameters $W^{(2, 1)}$,$W^{(2, 2)}$,$b^{(2, 1)}$, $b^{(2, 2)}$. The steps are repeated for subsequent layers with the output of each layer as input for the subsequent layer. The parameters of each layer are trained individually while freezing for the rest of the model. The effective results are obtained after training and the model is fine-tuned using backpropagation, by tuning the parameters of all layers in the same time. The primary features on the first hidden layer $h^{(1)(k)}$are learned by training a sparse autoencoder on the raw inputs $x^{(k)}$ on the raw input. Phase I of the stacked encoder is shown in Fig.3.3.

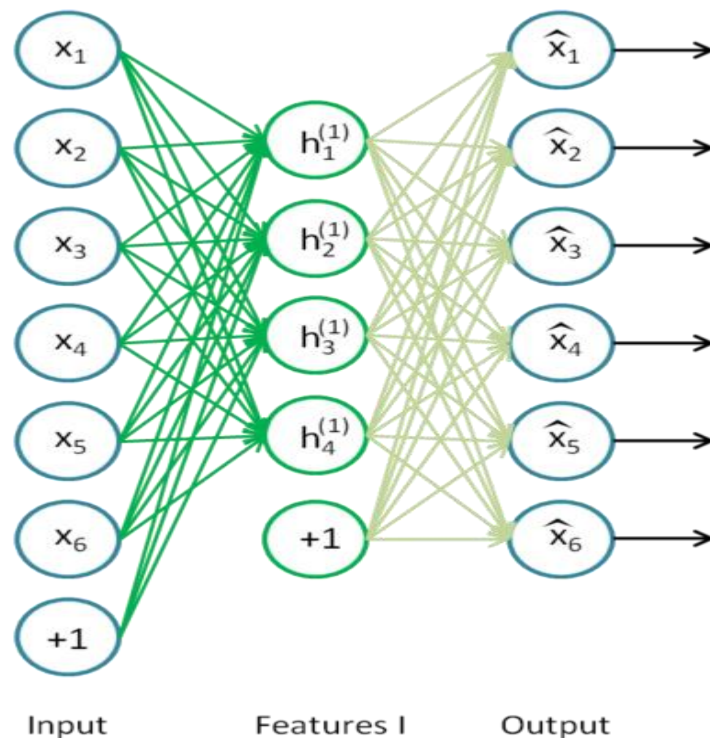

Fig.3.3 Implementing Stacked Autoencoder Phase -I

Next, the raw input is feed into this trained sparse autoencoder, obtaining the primary feature activations $h^{(1)(k)}$ for each of the inputs $x^{(k)}$. Then these primary features are used as the raw input to another sparse autoencoder to learn secondary features $h^{(2)(k)}$ on these primary features as shown in Fig.3.4.

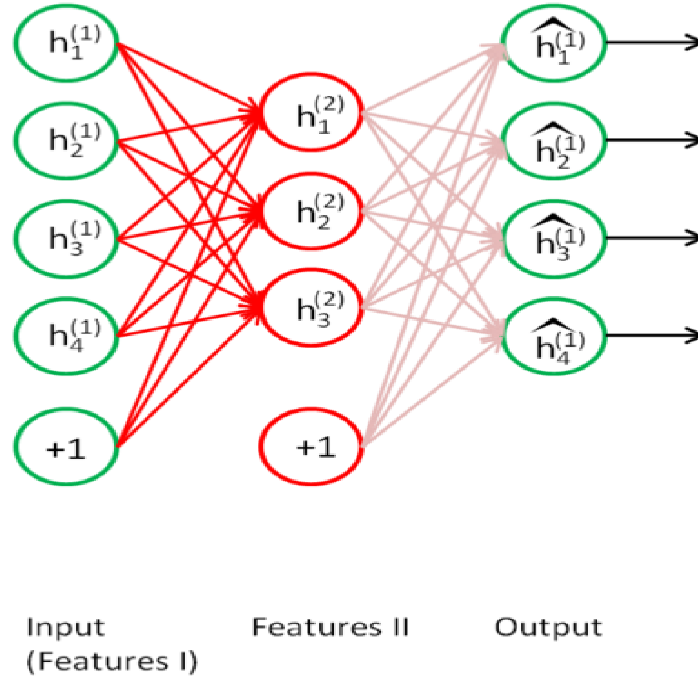

Fig.3.4 Implementing Stacked Autoencoder Phase -II

Following this, the primary features are feed into the second sparse autoencoder to obtain the secondary feature activations $h^{(2)(k)}$ for each of the primary features $h^{(1)(k)}$. The secondary features that is obtained at this layer is passed as the raw input of the softmax classifier by mapping the secondary features into its corresponding labels as shown in Fig.3.5.

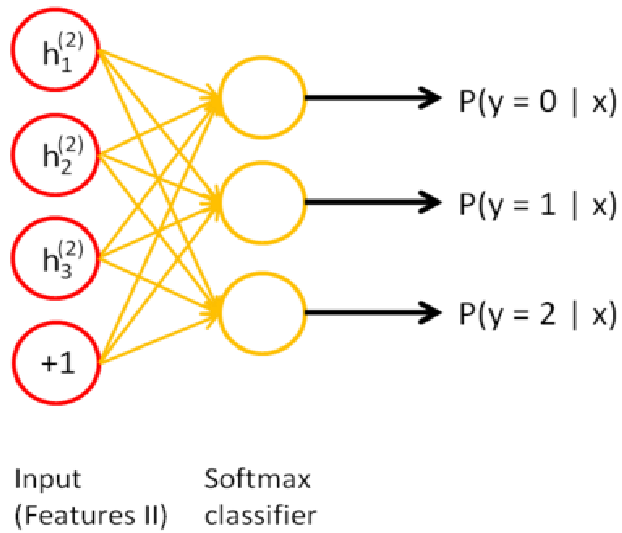

Fig.3.5 Implementing Stacked Autoencoder Phase -III

A stacked autoencoder with 2 hidden layers is formed by combining all the three layers and the data is classified with the final softmax classifier layer as shown in Fig.3.6.

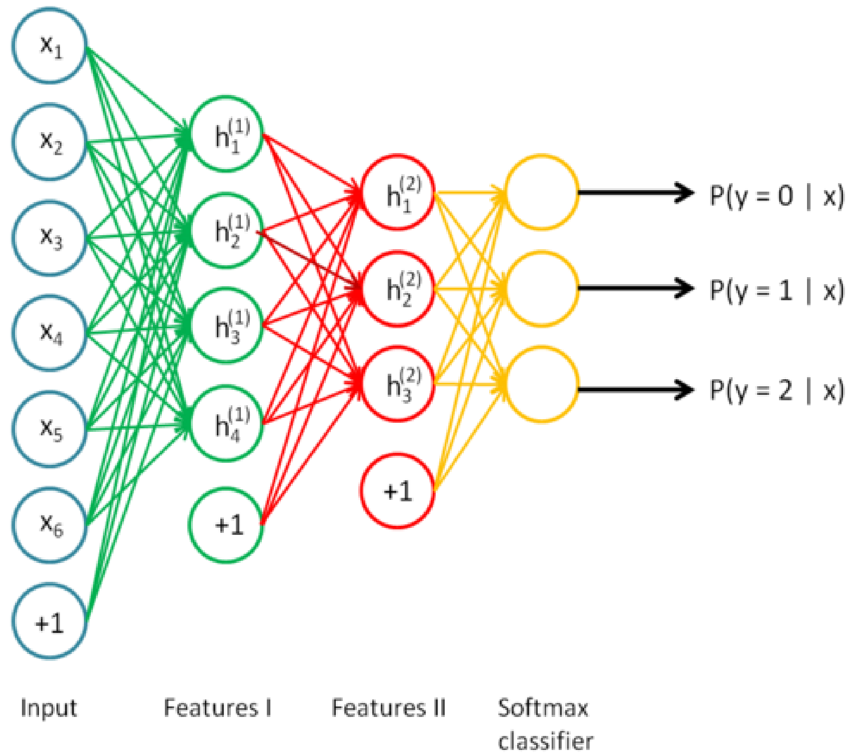

Fig.3.6 Implementing Stacked Autoencoder Phase -IV

Fig.3.6 depicts the whole network. The hidden layer of autoencoder t acts as an input layer to autoencoder t + 1. The input layer of the first autoencoder is the input layer for the whole network. The working of greedy layer-wise training procedure is given below:

1. The first autoencoder (t=1) is trained separately using the backpropagation method with the training data that is available.
2. The second autoencoder is trained with t=2. Since the input layer for t=2 is the hidden layer of t=1, the output layer of t=1 is removed from the network. Training begins by holding an input sample to the input layer of t=1, which is propagated forward to the output layer of t=2. Next, the weights (input-hidden and hidden-output) of t=2 are updated using backpropagation.
3. The previous procedure is repeated for the remaining layers. Remove the output layer of the previous autoencoder and replace it with another autoencoder, and train with back propagation.
4. Steps 1-3 are called pre-training, leave the weights which are properly initialized. For example, if the network is trained for identifying the images of handwritten digits it is not possible to map the units from the final feature detector to the digit type of the image. In such case, one or more fully connected layer can be linked to the last layer. The entire network can be viewed as a multilayer perceptron and is trained using backpropagation through fine-tuning.

The stacked auto encoders to initialize the weights of a network with a complex multi-layer perceptron provide an effective pre-training method.

Pretraining DNN with Autoencoder

The first step is to train one by one each auto encoder to encode and decode their input. After pre training is done, the weights of DNN can be set with the weights of all encoder.

Deep Neural Network Algorithm

Input: Sequence Data = {x}, n-desired layers and L-number of nodes for each layer

Steps:

1. Train an autoencoder to achieve the higher level of code n.
2. Put DNN over the stacked autoencoders as the output layer

Output: Fine-tuned Neural network model

3.6 Summary

This chapter describes the basic concepts of Deep Learning, its architecture, Stacked and auto encoder and deep neural network classifier. Since deep learning technique automatically extracts the features from the training data and predicts the output more accurately, deep neural network classifier is considered in this research work to build a muscular dystrophy disease classification model in a distributed environment.