1 Neural networks

Neural networks are models based on the weighted sums of a network of j hidden nodes, divided into k layers. Each node has an activation function that determines whether the specific node should output a value, when the input to the node is above or below a given value. The activation functions are inspired by the chemical processes of the neurons in the human brain (Haykin 2009), and can be both linear (Rectifier) as well as non-linear (Sigmoid, Tanh). A neural network can output both a weighted sum for regression, or a probability of a given class, for classification tasks. The difference between the two, resides in the output function of the network, where regression models has an identity function, and classification tasks has sigmoid, tanh in the case of a binary output (Haykin, 2009).

Illustrated below, is a 2 layer neural network, with inputs X_n, and one regression output O_i, for the sake of convenience, each hidden unit is illustrated with the tanh activation function, which will be used in this particular study. The W denotes the weights in each layer, where sub index *i* denotes the particular weight, and sub index *j* denotes the particular layer.

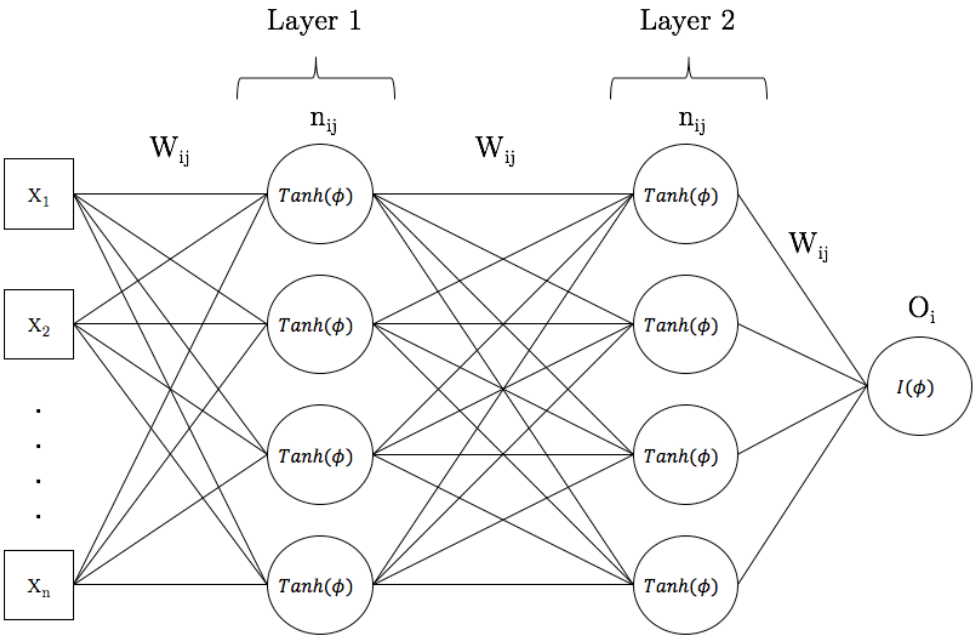

Fig 1.1. A 2-layer neural network.

1.1 Activation functions

The activation functions take in an input (ϕ) from the weighted sum of outputs from subsequent units leading to the particular hidden unit, shown by formula (1) below:

$$\phi = \sum_{i=1}^{k} W_{ij} X \; I_j + bias_i \tag{1}$$

As stated earlier, the activation functions are in different variants. In the following, the focus will be on the nonlinear variants: Sigmoid & Tanh, which are also very similar. The sigmoid activation function, also known as the logistic function (2), produces an output ranging from 0 to 1, illustrated by equation (3):

$$Sigmoid(\phi) = \frac{1}{1 + e^{-\phi}} \tag{2}$$

$$Output(\phi) = \begin{cases} 1 & if \; \phi \; \geq \; 0 \\ 0 & if \; \phi \; < \; 0 \end{cases} \tag{3}$$

The Tanh function (3), which is a modified version of the sigmoid, instead produces an output ranging from -1 to +1 illustrated by equation (5). This provides a benefit in the sense that it centers the output around 0, effectively yielding higher gradients, and thus speeding up the time of training the neural network (Lecun et.al. 1998).

$$Tanh(\phi) = \frac{2}{1 + e^{-2\phi}} - 1 \tag{4}$$

$$Output(\phi) = \begin{cases} 1 & if \; \phi > 0 \\ 0 & if \; \phi = 0 \\ -1 & if \; \phi < 0 \end{cases} \tag{5}$$

1.2 Back propagation

A neural network is fitted to the training data by passing through the data and updating each of the weights based on the gradient of the error of a prediction on the validation set. This process is called back propagation, since it is based on a backward pass through the network where the gradient of each hidden unit is computed, before the weight is adjusted. The goal of the backpropagation algorithm is to minimize the total empirical risk which is essentially the error of the output from the particular node, using a given updating rule. In this case we shall consider the process of Gradient descent (Haykin, 2009). The formula for the empirical risk is described in Eq. 6 below, where C denotes the full set of neurons in the network, N is the full batch of training samples, and subscript n denotes a hidden unit. The error $e_j(n)$ for a specific hidden unit in layer j, is given by equation 7.

$$Empirical\ risk = \varepsilon(N) = \frac{1}{2N}\sum_{n=1}^{N}\sum_{j\in C}(Y_n - \hat{Y}_n)^2 \tag{6}$$

$$e_j(n) = Y_j(n) - \widehat{Y}_j(n) \tag{7}$$

The local gradient for each hidden unit n is found by using back propagation formula in Eq. 8, and Eq. 9 (Haykin, 2009) for the hidden, and output layer, respectively, where $\delta_k(n)$ is the gradient of the prior k outputs to the right of the particular hidden unit n, $w_{kj}(n)$ is the weighted sum of the inputs to the prior k outputs, and φ_j' is used to denote the particular activation function, which is illustrated for hyperbolic tangent in equation 10.

$$\delta_j(n)_{hidden} = \varphi_j'\left(\phi_j(n)\right)\sum_k \delta_k(n)\, w_{kj}(n) \tag{8}$$

$$\delta_j(n)_{output} = e_j(n)\ \varphi_j'\left(\phi_j(n)\right) \tag{9}$$

Activation function: To find the gradient of the activation function (tanh in this case) which is used in Eq. 7 & 8, the following equation below (Haykin, 2009) can be used, where a and b are positive constants (suggested as a = 1.7159, b = $\frac{2}{3}$ by Lecun et.al, 1998), and ϕ_j is the weighted sum of the outputs from the prior layer, of the particular hidden unit *n*:

$$\varphi_j'\left(\phi_j(n)\right) = \frac{a}{b}\left[a - y_j(n)\right]\left[a + y_j(n)\right] \tag{10}$$

Weight update: For updating a given weight from one hidden unit to another, the prior equations are combined in the final weight correction equation (10), where η is a regularization parameter called the learning rate, effectively penalizing the weight update, y_i is the output of the succeeding neuron, and δ_j is the :

$$\Delta w_{ji} = \eta \delta_j(n) y_i(n) \tag{11}$$

Stopping criterion: Since there is no formal way of proving convergence (Haykin, 2009), an early stopping criterion is often used in order to halt the training prior to overfitting. This is often used as the relative decrease in the error function from one epoch to another. A specific threshold Ψ is set, and the training is stopped, if the improvement of the model is beyond Ψ. As suggested by Haykin, Ψ is typically small enough if it lies within 1-0.1 percent of improvement between two epochs. This procedure is of course optional, and can be altered to custom convergence criterions instead: Stopping after k hours or k epochs.

Procedure: for the backpropagation algorithm using gradient descent optimization (adapted from Haykin 2009):

1. Initialize the weights by random e.g. U(-0.5,+0.5)
2. For every sample in the training set:
    a. Forward computation:
    Compute the output of the full neural network going from left to right (see figure 1.1.) in order to obtain the residuals $e_j(n)$.
    b. Backward computation:
    Computed the local gradients $\delta_j(n)$ with respect to the error for every hidden unit from the output layer until the first hidden layer, going from right to left.
    c. Evaluation (optional):
    If the change in empirical risk: $\Delta\varepsilon < \Psi$, the stopping criterion has been reached, and the training is stopped.

1.3 Robustness of neural networks

Neural networks are probably known best for their good performance on nonlinear relationships, since it has been theoretically proven that a neural network can approximate any possible function, within an allowed tolerance of error (Touretzky & Pomerlau 1989), however the theorem assumes that there is an infinite amount of hidden units available to get within the desired error tolerance, but in many cases the problem might be solved more efficiently by adding more than one layer (Haykin, 2009). In the following, some of the factors that can influence the performance of the neural network are briefly described.

Input transformation: As argued by Lecun et.al. 1998, it is important to standardize the input varibles of the neural network ($\mu = 0$, $\sigma = 1$), in order for the model to converge faster, since an extreme case of all positive input values would cause the searching to zigzag which is very inefficient.

Layers: It is argued that by using two layers, the first layer of a neural network will separate the function to be approximated into regions of interest, and in the second layer global features are learned by combining the regions in the first layer (Funahashi, 1989).

Optimization: The efficiency of the optimization algorithm, be it Gradient descend, Stochastic gradient descent or an adaptive optimization

method like Adam (Diederik et.al. 2015), is by all means significant in terms of the overall performance of the neural network, since Stochastic methods might help prevent the model being stuck in a local minima (Lecun et.al. 1998).

Regularization: adding regularization in terms of dropout (Srivastava et.al. 2014) might help the neural network in terms of avoiding the model to *overfit* the training data, which implies high generalization error. Another form of regularization is by adding the L1 and/or L2 norm of the weights to the error function to be optimized, essentially smoothing the learning of the network which would as well help prevent overfitting (Haykin, 2009).