1. **Perceptron**

Despite the fantastic name - Deep Learning, the field of neural networks is not new at all. Perceptron can be the foundations for neural networks in 1980s. It was developed to solve a binary classification problem.

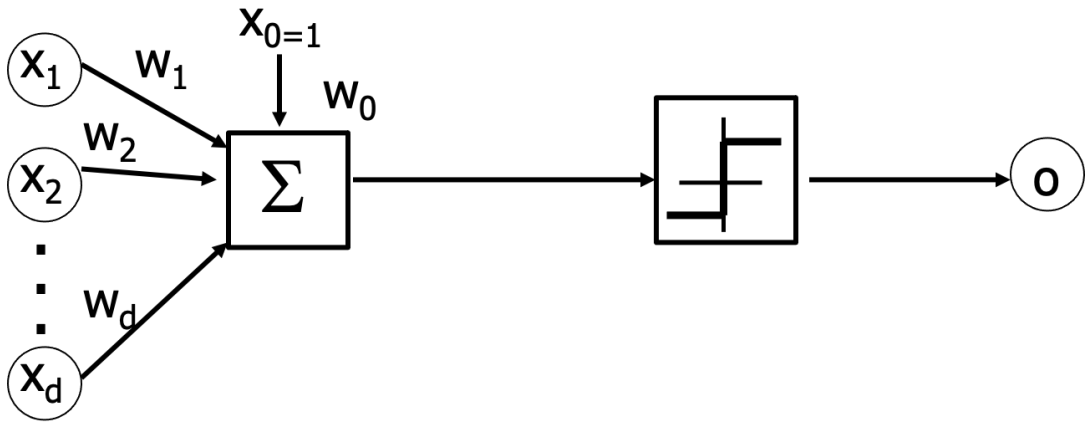

Figure 1. A Perceptron module.

We consider the input signal to be represented by a d-dimensional feature vector $[x_1, x_2, \cdots, x_d]$. In a Perceptron (see Fig. 1), we plan to learn a linear function parameterized by $\mathbf{w}$ to accomplish the classification task,

$$\hat{y} = \sum_{i=1}^{d} w_i x_i + b, \tag{1}$$

where w_i is the weight on the i-th dimension, b is the bias and $\hat{y}$ is the prediction score of the example x. The binary prediction can then be easily with the help of the *sign* function,

$$o = sign(\hat{y}) = \begin{cases} +1, & \hat{y} \geq 0 \\ -1, & \hat{y} < 0 \end{cases} \tag{2}$$

Perceptron simply uses target values $y = 1 + 1$ for the positive class and $y = -1$ for the negative class. According to the above analysis, we find that if an example can be correctly classified by the Perceptron, we have

$$y(\mathbf{w}^T\mathbf{x} + b) > 0, \tag{3}$$

otherwise

$$y(\mathbf{w}^T\mathbf{x} + b) < 0. \tag{4}$$

Therefore, to maximize the prediction accuracy (i.e., to minimize the cost function for Perceptron), the objective function of Perceptron can be written as,

$$\min_{\mathbf{w},b} \quad L(\mathbf{w}, b) = -\sum_{x_i \in \mathcal{M}} y_i(\mathbf{w}^T\mathbf{x}_i + b), \tag{5}$$

where $\mathcal{M}$ stands for the set of mis-classified examples.
Gradient descent can be applied to optimize the Problem (5). By taking the partial derivative, we can calculate the gradient of the objective function,

$$\nabla_{\mathbf{w}} L(\mathbf{w}, b) = -\sum_{x_i \in \mathcal{M}} y_i \mathbf{x}_i, \tag{6}$$

$$\nabla_b L(\mathbf{w}, b) = -\sum_{x_i \in \mathcal{M}} y_i. \tag{7}$$

Gradient descent methods can then be taken to update the parameters $\mathbf{w}$ and b until the convergence.

2. Multilayer Neural Networks

The Perceptron is only capable of separating data points with a linear classifier, and cannot even handle the simple XOr problem. The solution to this problem is to include an additional layer - known as a hidden layer. Then this kind of feed-forward network is a multilayer perceptron, as shown in Fig. 2.

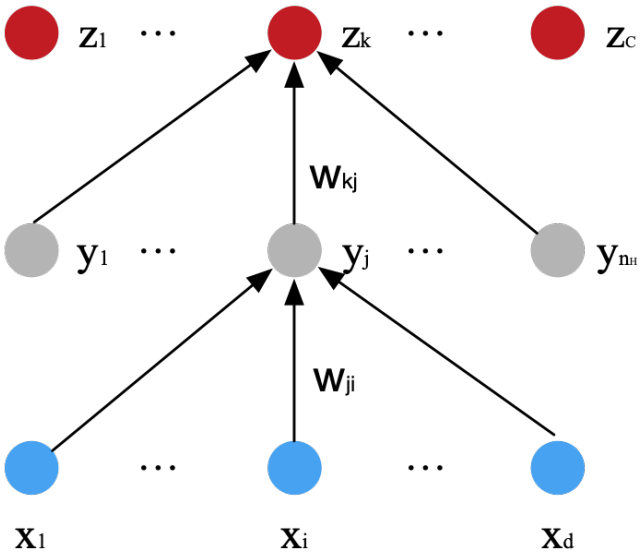

FIGURE 2. A three-layer neural network and the notation used.

Figure 2 shows a simple three-layer neural network, which consists of an input layer, a hidden layer, and an output layer, interconnected by modifiable weights, represented by links between layers. Each hidden unit computes the weighted sum of its inputs to form its scalar net activation, which is denoted simply as *net*. That is, the net activation is the inner product of the inputs with the weights at the hidden unit. Thus, it can be written

$$net_j = \sum_{i=1}^{d} x_i w_{ji} = w_j^T x, \tag{8}$$

where the subscript i indexes units in the input layer; w_{ji} denotes the input-to-hidden layer weights at the hidden unit j. Each hidden unit emits an output that is a nonlinear function of its activation, $f(net)$, that is,

$$y_j = f(net_j). \tag{9}$$

This $f(\cdot)$ is called the activation function or nonlinearity of a unit. Each output unit computes its net activation based on the hidden unit signals as

$$net_k = \sum_{j=0}^{n_H} y_j w_{kj} = w_k^T y, \tag{10}$$

where the subscript k indexes units in the output layer and n_H denotes the number of hidden units. An output unit computes the nonlinear function of its net, emitting

$$z_k = f(net_k) \tag{11}$$

3. **Activation Functions**

Activation functions are functions used in neural networks to decide if a neuron can be fired or not.

3.1. **Sigmoid Function.** The Sigmoid is a non-linear activation function used mostly in feedforward neural networks. It is a bounded differentiable real function, defined for real input values, with positive derivatives everywhere and some degree of smoothness. The Sigmoid function is given by

$$f(x) = \frac{1}{1 + e^{-x}}. \tag{12}$$

However, the Sigmoid activation function suffers major drawbacks:

- Sigmoids saturate and kill gradients. A very undesirable property of the sigmoid neuron is that when the neuron's activation saturates at either tail of 0 or 1, the gradient at these regions is almost zero. Recall that during backpropagation, this (local) gradient will be multiplied to the gradient of this gate's output for the whole objective. Therefore, if the local gradient is very small, it will effectively "kill" the gradient and almost no signal will flow through the neuron to its weights and recursively to its data. Additionally, one must pay extra caution when initializing the weights of sigmoid neurons to prevent saturation. For example, if the initial weights are too large then most neurons would become saturated and the network will barely learn.
- Sigmoid outputs are not zero-centered. This is undesirable since neurons in later layers of processing in a Neural Network (more on this soon) would be receiving data that is not zero-centered. This has implications on the dynamics during gradient descent, because if the data coming into a neuron is always positive (e.g. $x > 0$ elementwise in $f = w^T x + b$), then the gradient on the weights w will during backpropagation become either all be positive, or all negative (depending on the gradient of the whole expression f). This could introduce undesirable zig-zagging dynamics in the gradient updates for the weights. However, notice that once these gradients are added up across a batch of data the final update for the weights can have variable signs, somewhat mitigating this issue. Therefore, this is an inconvenience but it has less severe consequences compared to the saturated activation problem above.

3.2. **Hyperbolic Tangent Function (Tanh).** The hyperbolic tangent function known as tanh function, is a smoother zero-centred function whose range lies between -1 to 1, thus the output of the tanh function is given by,

$$f(x) = \frac{e^{-x} - e^{-x}}{e^{-x} + e^{-x}}. \tag{13}$$

The tanh function became the preferred function compared to the sigmoid function in that it gives better training performance for multi-layer neural networks. However, the tanh function could not solve the vanishing gradient problem suffered by the sigmoid functions as well. The main advantage provided by the function is that it produces zero centred output thereby aiding the back-propagation process. The tanh functions have been used mostly in recurrent neural networks for natural language processing.

3.3. **Rectified Linear Unit (ReLU) Function.** The rectified linear unit (ReLU) activation function was proposed by Nair and Hinton 2010, and ever since, has been the most widely used activation function for deep learning applications with state-of-the-art results to date. It offers the better performance and generalization in deep learning compared to the Sigmoid and tanh activation functions. The ReLU activation function performs a threshold operation to each input element where values less than zero are set to zero thus the ReLU is given by

$$f(x) = max(0, x). \tag{14}$$

The main advantage of using the rectified linear units in computation is that, they guarantee faster computation since it does not compute exponentials and divisions, with overall speed of computation enhanced. Another property of the ReLU is that it introduces sparsity in the hidden units as it squishes the values between zero to maximum.
The ReLU has a significant limitation that it is sometimes fragile during training thereby causing some of the gradients to die. This leads to some neurons being dead as well, thereby causing the weight updates not to activate in future data points, thereby hindering learning as dead neurons gives zero activation. To resolve the dead neuron issues, the leaky ReLU was proposed.
The leaky ReLU introduces some small negative slope to the ReLU to sustain and keep the weight updates alive during the entire propagation process. The alpha parameter was introduced as a solution to the ReLUs dead neuron problems such that the gradients will not be zero at any time during training. The LReLU computes the gradient with a very small constant value for the negative gradient α in the range of 0.01 thus the LReLU is computed as

$$f(x) = \begin{cases} x, & x > 0 \\ \alpha x, & x \leq 0 \end{cases} \tag{15}$$

The LReLU has an identical result when compared to the standard ReLU with an exception that it has non-zero gradients over the entire duration thereby suggesting that there no significant result improvement except in sparsity and dispersion when compared to the standard ReLU and tanh function.

4. **Backpropagation**

The backpropagation is one of the simplest and most general methods for supervised training of multilayer neural networks. Networks have two primary modes of operation: feedforward and learning. Feed-forward operation consists of presenting a pattern to the input units and passing the signals through the network in order to yield outputs from the output units. Supervised learning consists of presenting an input pattern and changing the network parameters to bring the actual outputs closer to the desired teaching or target values.
We consider the training error on a pattern to be the sum over output units of the squared difference between the desired output t_k and the actual output z_k:

$$J(w) = \frac{1}{2}\|t - z\|^2 \tag{16}$$

where t and z are the target and the network output vectors and w represents all the weights in the network.
The backpropagation learning rule is based on gradient descent. The weights are initialized with random values, and then they are changed in a direction that will reduce the error:

$$\Delta w = -\eta \frac{\partial J}{\partial w}, \tag{17}$$

where η is the learning rate, and indicates the relative size of the change in weights. Eq. (17) demands that we take a step in weight space that lowers the criterion function. It is clear from Eq. (16) that the criterion function can never be negative; the learning rule guarantees that

learning will stop. This iterative algorithm requires taking a weight vector at the iteration m and updating it as

$$w(m+1) = w(m) + \Delta w. \tag{18}$$

We now turn to the problem of evaluating Eq. (17) for a three-layer net. Consider first the hidden-to-output weights, w_{kj}. Because the error is not explicitly dependent upon w_{kj}, we must use the chain rule for differentiation:

$$\frac{\partial J}{\partial w_{kj}} = \frac{\partial J}{\partial net_k}\frac{\partial net_k}{\partial w_{kj}} = -\delta_k \frac{\partial net_k}{\partial w_{kj}}, \tag{19}$$

where the sensitivity of unit k is defined to be

$$\delta_k = -\frac{\partial J}{\partial net_k} \tag{20}$$

and describes how the overall error changes with the unit's net activation and determines the direction of search in weight space for the weights . Assuming that the activation function $f(\cdot)$ is differentiable, we differentiate Eq. (16) and find that for such an output unit, δ_k is simply

$$\delta_k = -\frac{\partial J}{\partial net_k} = -\frac{\partial J}{\partial z_k}\frac{\partial z_k}{\partial net_k} = (t_k - z_k) f'(net_k). \tag{21}$$

Taken together, these results give the weight update or learning rule for the hidden-to-output weights:

$$\Delta w_{kj} = \eta \delta_k y_j = \eta (t_k - z_k) f'(net_k) y_j. \tag{22}$$

The learning rule for the input-to-hidden units is subtle. From Eq. (17), and again using the chain rule, we calculate

$$\frac{\partial J}{\partial w_{ji}} = \frac{\partial J}{\partial y_j}\frac{\partial y_j}{\partial net_j}\frac{\partial net_j}{\partial w_{ji}} \tag{23}$$

The first term can be calculated as

$$\begin{aligned} \frac{\partial J}{\partial y_j} &= \sum_{k=1}^{c} \frac{\partial J}{\partial z_k}\frac{\partial z_k}{\partial y_j} = -\sum_{k=1}^{c} (t_k - z_k)\frac{\partial z_k}{\partial net_k}\frac{\partial net_k}{\partial y_j} \\ &= -\sum_{k=1}^{c} (t_k - z_k) f'(net_k) w_{kj} = -\sum_{k=1}^{c} \delta_k w_{kj}. \end{aligned} \tag{24}$$

For the step above, we had to use the chain rule again. The final sum over output units in Eq. (24) expresses how the hidden unit output y_j, affects the error at each output unit. This will allow us to compute an effective target activation for each hidden unit. We use Eq. (24) to define the sensitivity for a hidden unit as

$$\delta_j = f'(net_j) \sum_{k=1}^{c} w_{kj}\delta_k \tag{25}$$

The sensitivity at a hidden unit is simply the sum of the individual sensitivities at the output units weighted by the hidden-to-output weights w_{kj}, all multiplied by $f'(net_j)$. Thus the learning rule for the input-to-hidden weights is

$$\Delta w = \eta x_i \delta_j = \eta f'(net_j) x_i \sum_{k=1}^{c} w_{kj}\delta_k \tag{26}$$

Hence, we conclude the backpropagation algorithm, or more specifically the "backpropagation of errors" algorithm. Backpropagation is just a gradient descent in layered models where application of the chain rule through continuous functions allows the computation of derivatives of the criterion function with respect to all model weights.