Learning Algorithm

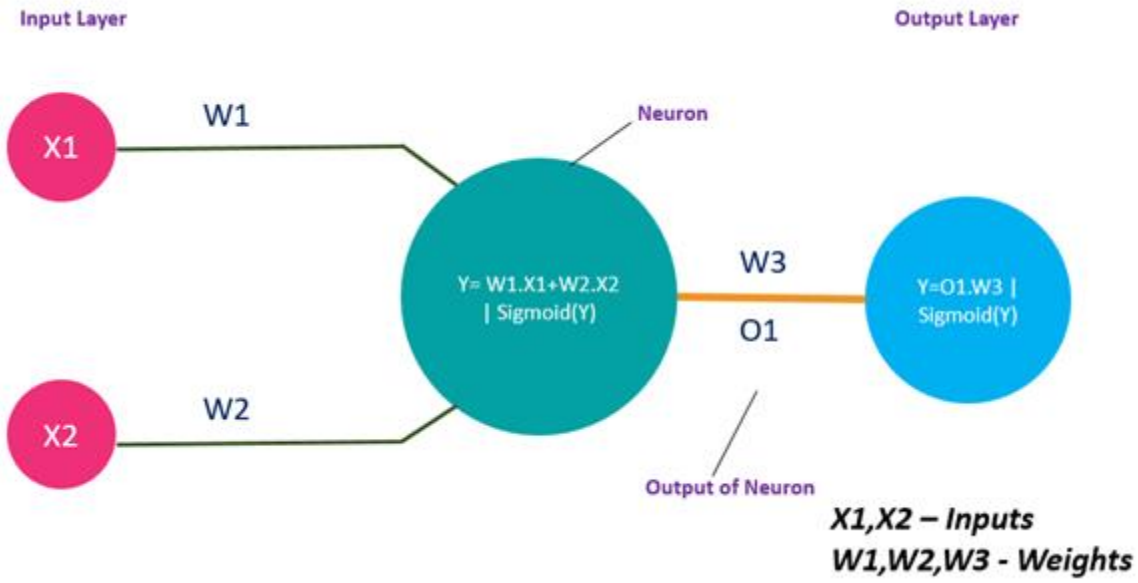
Working of Neural Network

A neural network works based on two principles

1. Forward Propagation
2. Backward Propagation

Let's understand these building blocks with the help of an example. Here I am considering a single input layer, hidden layer, output layer to make the understanding clear.

Forward Propagation



1. Considering we have data and would like to apply binary classification to get the desired output.
2. Take a sample having features as X1, X2, and these features will be operated over a set of processes to predict the outcome.
3. Each feature is associated with a weight, where X1, X2 as features and W1, W2 as weights. These are served as input to a neuron.
4. *A neuron performs both functions.*

    a) Summation

    b)Activation.

5. In the summation, all features are multiplied by their weights and bias are summed up. (Y=W1X1+W2X2+b).
6. This summed function is applied over an Activation function. The output from this neuron is multiplied with the weight W3 and supplied as input to the output layer.

7. The same process happens in each neuron, but we vary the activation functions in hidden layer neurons, not in the output layer.

Backpropagation:

Backpropagation, short for *backward propagation of errors*, refers to the algorithm for computing the gradient of the loss function with respect to the weights. However, the term is often used to refer to the entire learning algorithm. The backpropagation carried out in a perceptron is explained in the following two steps.

- **Step 1**: To know an estimation of how far are we from our desired solution a *loss function* is used. Generally, *mean squared error* is chosen as the loss function for regression problems and *cross entropy* for classification problems. Let's take a regression problem and its loss function be mean squared error, which squares the difference between *actual* (y_i) and *predicted value* ($\hat{y}_i$).

$$MSE_i = (y_i - \hat{y}_i)^2$$

Loss function is calculated for the entire training dataset and their average is called the *Cost function* **C**.

$$C = MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

- **Step 2**: In order to find the best weights and bias for our Perceptron, we need to know how the cost function changes in relation to weights and bias. This is done with the help of the *gradients* (*rate of change)* — how one quantity changes in relation to another quantity. In our case, we need to find the gradient of the cost function with respect to the weights and bias.

Let's calculate the gradient of cost function **C** with respect to the weight $\mathbf{w_i}$ using *partial derivation.* Since the cost function is not directly related to the weight w_i, let's use the chain rule.

$$\frac{\partial C}{\partial w_i} = \frac{\partial C}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial z} \times \frac{\partial z}{\partial w_i}$$

Now we need to find the following three gradients

$$\frac{\partial C}{\partial \hat{y}} = ? \qquad \frac{\partial \hat{y}}{\partial z} = ? \qquad \frac{\partial z}{\partial w_1} = ?$$

Let's start with the gradient of the c*ost function (C)* with respect to the *predicted value* ($\hat{y}$)

Learning Algorithm

$$\frac{\partial C}{\partial \hat{y}} = \frac{\partial}{\partial \hat{y}} \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 = 2 \times \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)$$

Let $y = [y_1, y_2, \ldots y_n]$ and $\hat{y} = [\hat{y}_1, \hat{y}_2, \ldots \hat{y}_n]$ be the row vectors of actual and predicted values. Hence the above equation is simplified as

$$\frac{\partial C}{\partial \hat{y}} = \frac{2}{n} \times sum(y - \hat{y})$$

Now let's find the gradient of the *predicted value* with respect to the *z*. This will be a bit lengthy.

$$\begin{aligned}
\frac{\partial \hat{y}}{\partial z} &= \frac{\partial}{\partial z} \sigma(z) \\
&= \frac{\partial}{\partial z} \left(\frac{1}{1 + e^{-z}} \right) \\
&= \frac{e^{-z}}{(1 + e^{-z})^2} \\
&= \frac{1}{(1 + e^{-z})} \times \frac{e^{-z}}{(1 + e^{-z})} \\
&= \frac{1}{(1 + e^{-z})} \times \left(1 - \frac{1}{(1 + e^{-z})} \right) \\
&= \sigma(z) \times (1 - \sigma(z))
\end{aligned}$$

The gradient of **z** with respect to the weight $\mathbf{w_i}$ is

$$\begin{aligned}\frac{\partial z}{\partial w_i} &= \frac{\partial}{\partial w_i}(z) \\ &= \frac{\partial}{\partial w_i}\sum_{i=1}^{n}(x_i.w_i + b) \\ &= x_i\end{aligned}$$

Therefore we get,

$$\frac{\partial C}{\partial w_i} = \frac{2}{n} \times sum(y - \hat{y}) \times \sigma(z) \times (1 - \sigma(z)) \times x_i$$

What about Bias? — Bias is theoretically considered to have an input of constant value *1*. Hence,

$$\frac{\partial C}{\partial b} = \frac{2}{n} \times sum(y - \hat{y}) \times \sigma(z) \times (1 - \sigma(z))$$

- **Optimization:** Optimization is the selection of the best element from some set of available alternatives, which in our case, is the selection of best weights and bias of the perceptron. Let's choose *gradient descent* as our optimization algorithm, which changes the *weights* and *bias*, proportional to the negative of the gradient of the cost function with respect to the corresponding weight or bias. *Learning rate* (α) is a hyperparameter which is used to control how much the weights and bias are changed.

The weights and bias are updated as follows and the backpropagation and gradient descent is repeated until convergence.

$$\begin{aligned}w_i &= w_i - \left(\alpha \times \frac{\partial C}{\partial w_i}\right) \\ b &= b - \left(\alpha \times \frac{\partial C}{\partial b}\right)\end{aligned}$$

Pseudocode

Pseudocode for a stochastic gradient descent algorithm for training a three-layer network (one hidden layer):

```
initialize network weights (often small random values)
do
    for each training example named ex do
        prediction = neural-net-output(network, ex)  // forward pass
```

Learning Algorithm

```
        actual = teacher-output(ex)
        compute error (prediction - actual) at the output units

        compute      for all weights from hidden layer to output layer
// backward pass

        compute      for all weights from input layer to hidden layer
// backward pass continued
        update network weights // input layer not modified by error
estimate
until error rate becomes acceptably low
return the network
```