Feed Forward Neural Networks
10/31/2025

Simulation of a complex system is analogous to solving for a function that yields outputs (y_j) in response to a set of inputs (x_i). For a simple system, this function can be approximated using a mathematical expression and the parameters of this function determined by experiments. For example, the Ohm's Law is a linear function:

$$I = f(V) = \frac{V}{R}$$

where V is the applied voltage to a conductor rod as the input and I the current through the rod as the output. R is the parameter also called the resistance determined by measuring the voltage and current pairs over many rods of the same dimension and chemistry. A complex system with multiple inputs and multiple outputs may be approximated by a multivariate function:

$$Y = f(X)$$

where Y is the output matrix of $k \times m$ dimensions, X input matrix of $k \times n$ dimensions, and k number of rows in the dataset. A linear function can be expanded as follows:

$$y_{k,j} = \sum_{i=1}^{n} x_{k,i} \cdot w_{i,j} + b_j \qquad j = 1, 2, \ldots, m \qquad k = 1, 2, \ldots, l$$

where $w_{i,j}$ are the elements of parameter or weight matrix, and b_j the biases.

Alternatively, one may use a neural network to approximate a linear or non-linear function.

Feed Forward Neural Network

A feed forward network is made of multiple layers. Each layer has several nodes or neurons. The first layer accepts the input matrix X with the same number of nodes as the column dimension of the input matrix. The input matrix is multiplied by the weights (as a weight matrix) and forms the output matrix (Z). If one takes the zeroth column of X and all the elements equal unity, the bias (b_j) vector can be accounted as part of the weight matrix.

$$Z^{(1)} = X \cdot W^{(1)}$$

where W is the weight matrix and the superscript (1) denotes the first layer. The input matrix (A) to the second layer takes the output from the first layer and "maps" it through an "activation function" (σ), often the $sigmoid()$ function. Sometimes, $ReLU()$ and $tanh()$ are used as the activation function. Therefore, the input matrix (A) of a layer l is the mapping of the output matrix (Z) of the previous layer ($l-1$) which in turn is a weighted linear combination (or dot product) of the input matrix to the layer.

$$A^{(l)} = \sigma\left(A^{(l-1)} \cdot W^{(l-1)}\right)$$

For a feed forward network with L layers, the input vector to the virtual $L+1$ layer becomes the output matrix of the network.

$$\hat{Y} = \sigma\left(A^{(L)} \cdot W^{(L)}\right)$$

Therefore the approximation of the system function is a nested $sigmoid$ functions:

$$\hat{f}(X) = \sigma\left(\sigma\left(\sigma\left(\cdots\sigma\left(X \cdot W^{(1)}\right)\cdots\right) \cdot W^{(L-1)}\right) \cdot W^{(L)}\right)$$

Construction of a Feedforward Network

A feed froward network is made of several matrices, particularly, the weight matrices. Figure 1 shows the R code for initializing these matrices and the connections between the layers described in the forward propagation function.

The size of the first layer weight matrix is determined by the number of input variables (number of columns of the input matrix) and the number of nodes on the hidden layer immediately behind the input layer. Likewise, the size of the output layer weight matrix is determined by the number of nodes on the last hidden layer and the number of output variables. The forward propagation function performs the matrix multiplication followed by the mapping of the output by the activation ($sigmoid$) function.

```
initialize_parameters = function(num_input, num_hidden, num_output) {
  W1 = matrix(runif(num_input * num_hidden), nrow = num_input)
  ...
  W2 = matrix(runif(num_hidden * num_output), nrow = num_hidden)
  return(list(W1 = W1, ..., W2 = W2))
}

params = initialize_parameters(num_input, num_hidden, num_output)

forward_propagation = function(X, params) {
  A1 = sigmoid(X %*% params$W1)
  A2 = sigmoid(A1 %*% params$W2)
  ...
  return(list(A1 = A1, A2 = A2, ...))
}
```

Figure 1, Construction of a Feed Forward Neural Network.

Training Feed Forward Neural Network

Gradient descent is used to train the neural network. This method seeks the optimal parameter matrix W to minimize the error between the neural network prediction and the true values in the training set. The correction direction in the parameter W space is opposite to the gradient of the loss function with respect to the parameter W (in fact, each component in the W matrix). The loss function (mean error square, or MES) is defined as below:

$$\Gamma\left(\sigma(X, W), Y\right) = \frac{1}{k}\sum_{i=1}^{k}\left(\hat{Y} - Y\right)_k^2 = \frac{1}{k}\sum_{i=1}^{k}\left(\sigma(X, W) - Y\right)_k^2$$

The gradient of the loss function with respect to W is:

$$\nabla\Gamma\left(\sigma(X,W),Y\right)=\begin{bmatrix}\frac{\partial}{\partial w_1}\Gamma\left(\sigma(X,W),Y\right)\\ \frac{\partial}{\partial w_2}\Gamma\left(\sigma(X,W),Y\right)\\ \vdots\\ \frac{\partial}{\partial w_{n\times m}}\Gamma\left(\sigma(X,W),Y\right)\end{bmatrix}$$

The correction to W is therefore:

$$W^{t+1}=W^{t}-\eta\nabla_{W}\Gamma\left(\sigma(X^{(i)},W),Y\right)$$

where t is the epoch time for each learning step and η the learning rate (or step size). The correction to W is implemented using the back propagation function (see Figure 2).

```
calculate_loss = function(y_pred, y_true) {return(mean((y_pred -
y_true)^2))}

backpropagation = function(X, y_true, forward_output, params) {
    A1 = forward_output$A1
    A2 = forward_output$A2
    dZ2 = (A2 - y_true) * sigmoid_derivative(A2)
    dW2 = t(A1) %*% dZ2
    dW1 = t(X) %*% (dZ2 %*% t(params$W2) * sigmoid_derivative(A1))
    return(list(dW1 = dW1, dW2 = dW2))
}

update_parameters = function(params, gradients, learning_rate) {
    params$W1 = params$W1 - learning_rate * gradients$dW1
    params$W2 = params$W2 - learning_rate * gradients$dW2
    return(params)
}
```

Figure 2, R functions for loss calculation, back propagation, and parameter (weight) updates.

An Online Advertising Example

The same training dataset used for the previous logistic regression (See Figure 4) is used again for the feed forward neural network classification. First, regularization ($scale()$) was performed on the dataset, so that the all the data are distributed with zero mean and unity standard deviation:

$$z_i=\frac{x_i-\mu}{\sigma}$$

This operation turns out to be very important for the success of this example. The dataset is then formulated as an input matrix. Further, in order to eliminate the bias term, a column of 400 ones are appended to the input matrix. This operation is important for the input layer but becomes less useful for the hidden layers. Therefore, a bias node is not used for the hidden layer even though an extra node is added to the hidden layer.

Figure 3 shows the prediction results of a feed froward network with one hidden layer of 5 nodes after 200 training cycles (epoch time) with a learning rate of 0.1. The final classification is based on a decision boundary at probability 0.5 and the prediction accuracy is 92.25%. Unlike the data shown in Figure 4 from logistic regression, the feed forward network predictions exhibit a greater separation among the data and away from the decision boundary (0.5).

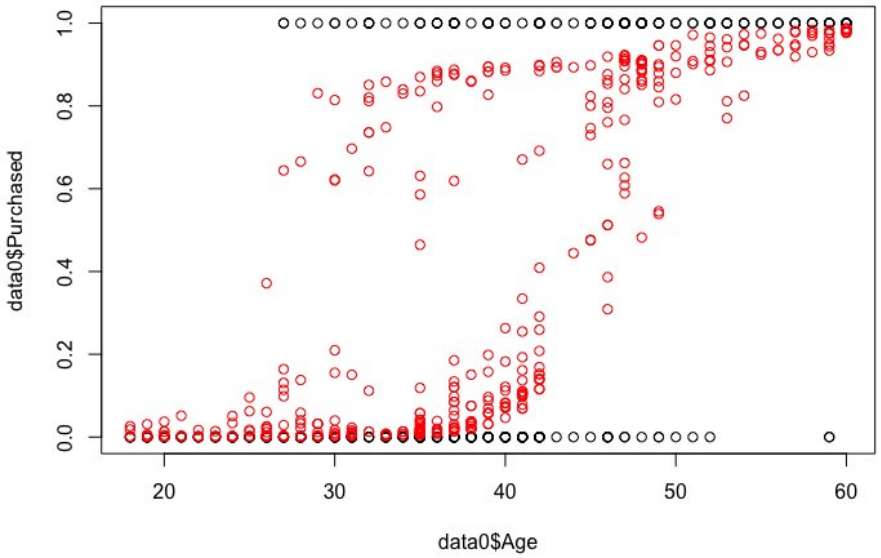

Figure 3, Training results from a feed forward neural network showing the probability of Purchased (black circles) and the predicted values prior to binary decision (red circles).

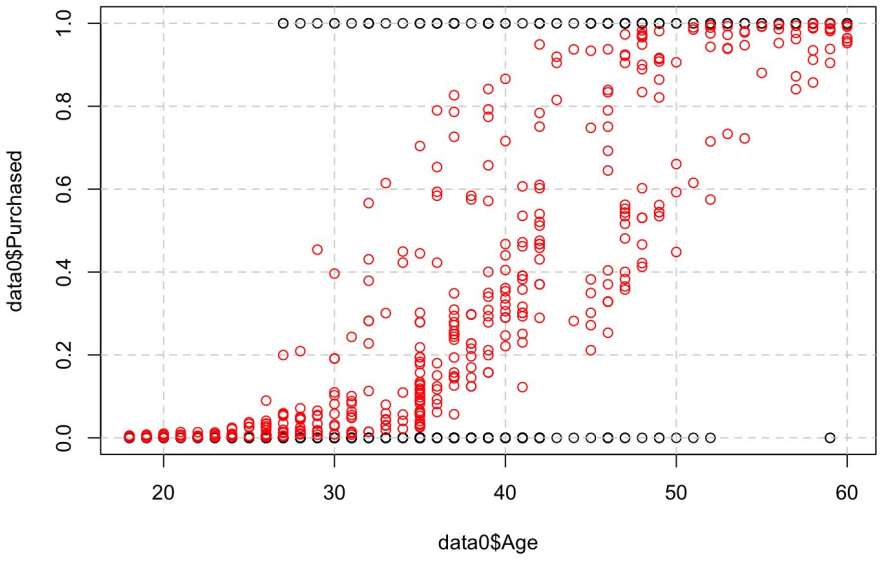

Figure 4, Regression with the multinom() function from the nnet library showing the probability of Purchased (black circles) and the fitted values (red circles).

Appendix: R Script of FFNN

```
## A single layer feed forward neural network ###
sigmoid = function(x) {return(1 / (1 + exp(-x)))}
sigmoid_derivative = function(x) {return(x * (1 - x))}

forward_propagation = function(X, params) {
  A1 = sigmoid(X %*% params$W1)
  A2 = sigmoid(A1 %*% params$W2)
  return(list(A1 = A1, A2 = A2))
}

train_neural_network = function(X, y_true, num_hidden, num_epochs, learning_rate) {
  num_input = ncol(X)
  num_hidden = num_hidden + 1    # add the bias column
  num_output = ncol(y_true)

  initialize_parameters = function(num_input, num_hidden, num_output) {
    W1 = matrix(runif(num_input * num_hidden), nrow = num_input)
    W2 = matrix(runif(num_hidden * num_output), nrow = num_hidden)
    return(list(W1 = W1, W2 = W2))
  }

  calculate_loss = function(y_pred, y_true) {return(mean((y_pred - y_true)^2))}

  backpropagation = function(X, y_true, forward_output, params) {
    A1 = forward_output$A1
    A2 = forward_output$A2
    dZ2 = (A2 - y_true) * sigmoid_derivative(A2)
    dW2 = t(A1) %*% dZ2
    dW1 = t(X) %*% (dZ2 %*% t(params$W2) * sigmoid_derivative(A1))
    return(list(dW1 = dW1, dW2 = dW2))
  }

  update_parameters=function(params, gradients, learning_rate) {
    params$W1=params$W1 - learning_rate * gradients$dW1
    params$W2=params$W2 - learning_rate * gradients$dW2
    return(params)
  }

  params=initialize_parameters(num_input, num_hidden, num_output)

  for (epoch in 1:num_epochs) {
    forward_output = forward_propagation(X, params)
    loss = calculate_loss(forward_output$A2, y_true)
    gradients = backpropagation(X, y_true, forward_output, params)
    params = update_parameters(params, gradients, learning_rate)
  }
  return(params)
}

### read training data set ###
fn="advert.csv"; data0=read.csv(fn, stringsAsFactors=TRUE)
X=as.matrix(data.frame(data0$Female, scale(data0$Age), scale(data0$Salary)))
y_true=as.matrix(data0$Purchased)
X = cbind(X, matrix(1, nrow = nrow(X), ncol = 1))   # add the bias column

trained_params=train_neural_network(X, y_true, num_hidden=5, num_epochs=200, learning_rate = 0.1)

predictions=forward_propagation(X, trained_params)$A2
classification=ifelse(predictions > 0.5, 1, 0)

cat("Accuracy:", mean(classification == y_true), "\n")

plot(data0$Age, data0$Purchased)
points(data0$Age,predictions, col="red")
```