Handmade Feedforward Neural Network

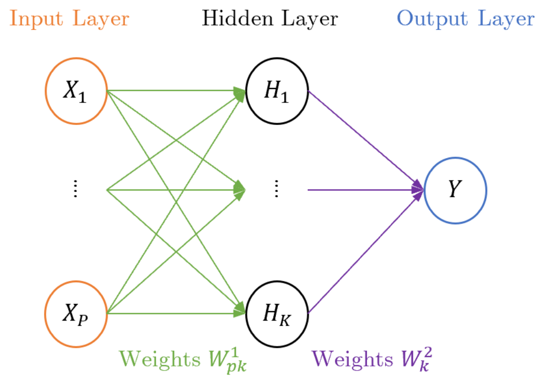
Álvaro Orgaz Expósito

THEORY

The aim of the *feedforward neural network* is to predict a variable y (categorical for *classification* problem or continuous for *regression problem*) knowing the explanatory variables $x = \{x_1, \ldots, x_p\}$. However, in this model it is possible to predict multiple target variables in the output layer but this paper will cover only one.

Firstly, you need to understand the *model structure* which consists of multiple layers with nodes or neurons fully connected with respective weights to the next layer in the network. This model is called feedforward because information flows only in one direction, from the input layer to the output layer.

Note: This paper covers the *feedforward neural network* with P neurons in the input layer, one hidden layer with K neurons, and one neuron in the output layer.



Activity of neuron H_{ik} → $Z^1_{ik} = X_{i1}W^1_{1k} + \cdots + X_{ip}W^1_{Pk}$

Activation of neuron H_{ik} → $A^1_{ik} = f(Z^1_{ik})$

Activity of neuron Y_i → $Z^2_i = A^1_{i1}W^2_1 + \cdots + A^1_{iK}W^2_k$

Activation of neuron Y_i → $A^2_i = f(Z^2_i)$

with $1 \leq k \leq K,\ 1 \leq p \leq P,\ 1 \leq i \leq N$

- *Input layer*

  Corresponds to the input data $X_{[NxP]}$ with N observations and P explanatory variables. Then, the input layer has P nodes.

$$X_{[NxP]} = \begin{bmatrix} x_{11} & \cdots & x_{1P} \\ \vdots & \ddots & \vdots \\ x_{N1} & \cdots & x_{NP} \end{bmatrix}$$

- *Hidden layer*

  The number of hidden layers ranges from one to many and the number of neurons is a tuning parameter with no optimal value for all cases. Using a lot of hidden layers gives name to the concept of *deep learning*.

- *Output layer*

  Corresponds to the output data $Y_{[Nx1]}$ to predict. For predicting a continuous variable, it has only 1 neuron but for a categorical variable, it has as nodes as the number of classes in the variable.

$$Y_{[Nx1]} = \begin{bmatrix} y_1 \\ \vdots \\ y_N \end{bmatrix}$$

- *Weights*

  The weights connect each neuron in the neural network and they are the parameters to optimize. When the *learning* or *training* process is finished, the weights are constant values that connect layers by multiplying them with the layers inputs (or outputs of previous layers).

  This is the weights matrix that connects the input layer with the hidden layer 1.

$$W^1_{[PxK]} = \begin{bmatrix} W^1_{11} & \cdots & W^1_{1K} \\ \vdots & \ddots & \vdots \\ W^1_{P1} & \cdots & W^1_{PK} \end{bmatrix}$$

  This is the weights matrix that connects the hidden layer 1 with the output layer.

$$W^2_{[Kx1]} = \begin{bmatrix} W^2_1 \\ \vdots \\ W^2_K \end{bmatrix}$$

- *Neuron activity*

  Except in the input layer, each node is a neuron with an activity and it is calculated as the linear combination of the outputs in the previous layer (neurons activation) and the weights that connect the previous layer with the actual node.

  This is the formula for the activity in the layer 1 or hidden layer 1.

$$Z^1_{[NxK]} = X_{[NxP]} W^1_{[PxK]}$$

  This is the formula for the activity in the layer 2 or the output layer.

$$Z^2_{[Nx1]} = A^1_{[NxK]} W^2_{[Kx1]}$$

- *Neuron activation*

  Except in the input layer, each node is a neuron that uses a nonlinear activation function and it means that in every node, its activity value is applied to an activation function. In short, it is the same idea than the *link function* of the *logistic regression* (explained in the paper *Handmade_Logistic_Regression.pdf*) but applied to every neuron in hidden and output layers.

  This paper will use the *sigmoid activation function.*

$$f(z) = \frac{1}{1+e^{-z}} = sigmoid(z)$$

  This is the formula for the activation in the layer 1 or hidden layer 1.

$$A^1_{[NxK]} = f\left(Z^1_{[NxK]}\right)$$

  This is the formula for the activation in the layer 2 or the output layer.

$$\hat{Y}_{[Nx1]} = A^2_{[Nx1]} = f\left(Z^2_{[Nx1]}\right)$$

Secondly, you need to understand the *learning* or *training* process. Basically, it consists of optimizing the weights that connect layers using the *cost function* which compares the prediction $\hat{Y}_{[Nx1]}$ with the expected output $Y_{[Nx1]}$.

The *learning* or *training* process is an optimization algorithm that repeats a two-phase cycle for each weights combination: *forward propagation* to make the prediction, and *backpropagation* to update the weights in the optimal direction. After repeating this process for a sufficiently large number of training iterations, the network will usually converge to a weights combination with a small prediction error.

Note: Although it seems more complex, it is the same process than in the paper *Handmade_Logistic_Regression.pdf* or *Handmade_Linear_Regression.pdf* where in every iteration of the weights optimization we firstly predicted the output and secondly updated the weights using the gradient of J .

- *Cost function*

  The prediction of the network is compared with the expected output using an *error function* or *cost function* or *cost function*. This paper will use as *cost function* the *Sum Square Error* divided by 2 (just to simplify the gradient calculation).

$$J = \frac{1}{2}\sum_{i=1}^{N}(y_i - \hat{y_i})^2$$

- *Forward propagation*

  When an input data is presented to the network, it is propagated forward through the network, layer by layer, until it reaches the output layer and then the final prediction. In the figure of page 1, you can see all the steps for predicting a new observation from X_i until $\hat{Y_i} = A_i^2$.

- *Backpropagation*

  Once the prediction is done through *forward propagation*, it is time to optimize the weights in order to reduce the value of the error predictions. To optimize or adjust weights properly in *backpropagation*, it is common to use the *gradient descent* algorithm which calculates the derivative of the *cost function* J with respect to the weights and updates the weights in the opposite direction of the gradient. Let's see the mathematical formula for the *gradient descent* used in the optimization process.

  Derivative of J by W_k^2:

$$\frac{\partial J}{\partial W_k^2} = \sum_{i=1}^{N}\frac{2}{2}(y_i - \hat{y_i})\frac{\partial(y_i - \hat{y_i})}{\partial W_k^2}$$

  where

$$\frac{\partial(y_i - \hat{y_i})}{\partial W_k^2} = -\frac{\partial \hat{y_i}}{\partial W_k^2} = -\frac{\partial sigmoid(Z_i^2)}{\partial Z_i^2}\frac{\partial Z_i^2}{\partial W_k^2} = -\frac{e^{-Z_i^2}}{(1+e^{-Z_i^2})^2}A_{ik}^1$$

  then

$$\frac{\partial J}{\partial W_k^2} = \sum_{i=1}^{N} -(y_i - \hat{y_i})\frac{e^{-Z_i^2}}{(1+e^{-Z_i^2})^2}A_{ik}^1$$

  Derivative of J by W_{pk}^1:

$$\frac{\partial J}{\partial W_{pk}^1} = \sum_{i=1}^{N}\frac{2}{2}(y_i - \hat{y_i})\frac{\partial(y_i - \hat{y_i})}{\partial W_{pk}^1}$$

  where

$$\frac{\partial(y_i - \hat{y_i})}{\partial W_{pk}^1} = -\frac{\partial \hat{y_i}}{\partial W_{pk}^1} = -\frac{\partial sigmoid(Z_i^2)}{\partial Z_i^2}\frac{\partial Z_i^2}{\partial A_{ik}^1}\frac{\partial A_{ik}^1}{\partial Z_{ik}^1}\frac{\partial Z_{ik}^1}{\partial W_{pk}^1} = -\frac{e^{-Z_i^2}}{(1+e^{-Z_i^2})^2}W_k^2\frac{e^{-Z_{ik}^1}}{(1+e^{-Z_{ik}^1})^2}x_ip$$

then

$$\frac{\partial J}{\partial W^1_{pk}} = \sum_{i=1}^{N} -(y_i - \hat{y}_i)\frac{e^{-Z^2_i}}{(1+e^{-Z^2_i})^2} W^2_k \frac{e^{-Z^1_{ik}}}{(1+e^{-Z^1_{ik}})^2} x_ip$$

Finally, the gradient of J is used to update the weights in each iteration of the *training* process

$$W^{1^{new}}_{[PxK]} = W^1_{[PxK]} - \eta \frac{\partial J}{\partial W^1_{[PxK]}}$$

$$W^{2^{new}}_{[Kx1]} = W^2_{[Kx1]} - \eta \frac{\partial J}{\partial W^2_{[Kx1]}}$$

where η is the learning rate parameter.

CODE

The data used is the popular dataset *iris*. Let's predict the variable *Petal.Length* with the explanatory features: *Sepal.Length* and *Sepal.Width*. It is necessary to normalize (minus the minimum and divided by maximum value) the target continuous variable because we are using the *sigmoid activation function* which makes sense to predict $0 \leq Y \leq 1$.

```
x <- as.matrix(iris[,c("Sepal.Length","Sepal.Width")])
y <- iris[,c("Petal.Length")]
y <- as.matrix((y-min(y))/max(y))
colnames(x) <- c("x1","x2")
colnames(y) <- c("y")
```

Set the neural network structure or number of neurons in each layer.

```
inputLayerSize <- ncol(x)
outputLayerSize <- ncol(y)
hiddenLayerSize <- 3
```

Set initial values for the weights randomly.

```
set.seed(1)
w1 <- matrix(rnorm(inputLayerSize*hiddenLayerSize),nrow=inputLayerSize,ncol=hiddenLayerSize)
w2 <- matrix(rnorm(hiddenLayerSize*outputLayerSize),nrow=hiddenLayerSize,ncol=outputLayerSize)
```

Start the weights optimization (*training* or *learning*) with *gradient descent*.

```
sigmoid <- function(z){
  1/(1+exp(-z))
}
derivativeSigmoid <- function(z){
  exp(-z)/((1+exp(-z))^2)
}
costs <- c()
rounds <- 5000
learning <- 0.01
for(i in 1:rounds){
  # Forward propagation: make prediction
  z1 <- x%*%w1         # Activity of layer 1
  a1 <- sigmoid(z1)    # Activation of layer 1
  z2 <- a1%*%w2        # Activity of layer 2
  yHat <- sigmoid(z2) # Activation of layer 2 or final prediction
  # Forward propagation: compute the actual cost
  costs <- c(costs,0.5*sum((y-yHat)^2))
  # Backpropagation: calculate the J gradient by weights
  dJdW1 <- t(x)%*%((((y-yHat)*(-1)*derivativeSigmoid(z2))%*%t(w2))*derivativeSigmoid(z1))
  dJdW2 <- t(a1)%*%((y-yHat)*(-1)*derivativeSigmoid(z2))
  # Backpropagation: update the weights
  w1 <- w1-learning*dJdW1
  w2 <- w2-learning*dJdW2
}
```

Let's see the optimal estimated weights.

```
w1
```

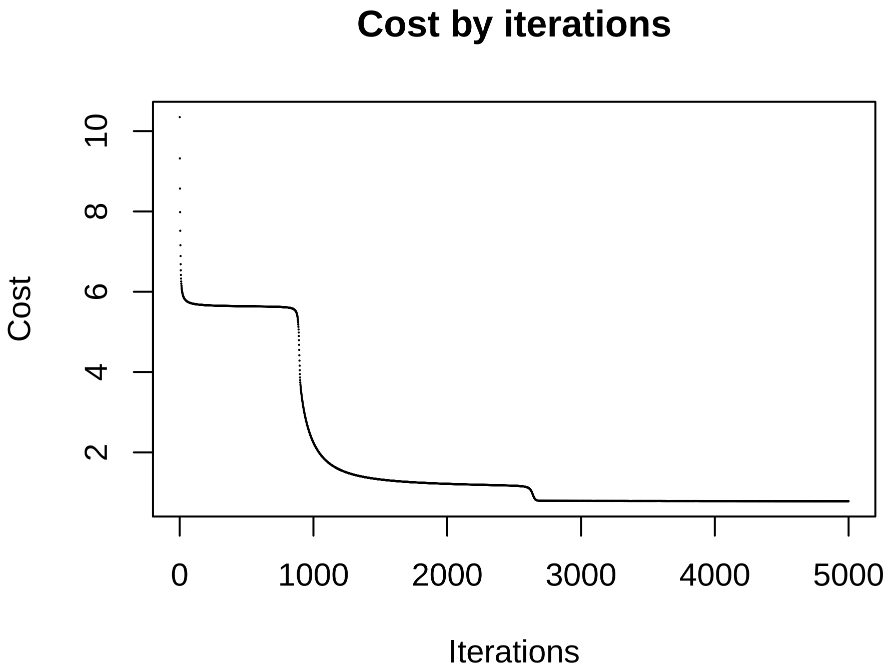
```
##          [,1]      [,2]       [,3]
## x1 -1.148844 -2.445544  0.5820264
## x2 -0.172851  3.986818 -0.7708414
```

```
w2
```

```
##                y
## [1,]  0.1179990
## [2,] -3.9337464
## [3,]  0.5082662
```

Let's plot the cost reduction in every iteration.

```
plot(costs,xlab="Iterations",ylab="Cost",main="Cost by iterations",cex=0.01)
```


Cost by iterations
Cost
Iterations