Hands On/tutorial on: Artificial Neural Networks

Based in:
https://rviews.rstudio.com/2020/07/20/shallow-neural-net-from-scratch-using-r-part-1/
https://rviews.rstudio.com/2020/07/24/building-a-neural-net-from-scratch-using-r-part-2/
By: Akshaj Verma, IISC, India

By the end of this hands-on/tutorial, you should have a deeper understanding of the math behind neural-networks and the ability to implement it yourself from scratch!

1. Set seed

Start by installing packages: tidyverse,

Then, set a seed to ensure reproducibility of the results

```
set.seed(69)
```

2. Construct Dataset

We will use the iris dataset using only the first and the third predictive attributes as the two attributes of the new dataset. Moreover, we will use the third attribute of the new dataset to define a new target attribute with two classes: The Specie "versicolor" will be defined as class 1. All other classes will be defined as class 0.

```
data(iris)
dataset<-iris
j<-which(dataset$Species=='versicolor')
k<-which(dataset$Species=='virginica')
l<-which(dataset$Species=='setosa')
dataset[,2]<-dataset[,3]
dataset[j,3]<-1
dataset[k,3]<-0
dataset[l,3]<-0
dataset<-dataset[,c(1,2,3)]
colnames(dataset)[3]<-'Species'
```

3. Visualize Data

Use a plot of your choice to visualize your new dataset

```
dataset %>% plot()
```

4. Train and Test Datasets

Define 80% of the data as training dataset and the remainder 20% as test dataset (look to the obtained datasets and see whether they make sense)

```
dataset <- dataset[sample(nrow(dataset)), ]
```

```
train_test_split_index <- 0.8 * nrow(dataset)
train <- dataset[1:train_test_split_index,]
head(train)
test <- dataset[(train_test_split_index+1):nrow(dataset),]
head(test)
```

5. Preprocess

The following code defines as transposed matrices the train and test datasets for both the predictive attributes and the target attribute

```
attributeX_train <- scale(train[, c(1:2)])
y_train <- train$Species
dim(y_train) <- c(length(y_train), 1) # add extra dimension to vector
X_test <- scale(test[, c(1:2)])
y_test <- test$Species
dim(y_test) <- c(length(y_test), 1) # add extra dimension to vector
X_train <- as.matrix(X_train, byrow=TRUE)
X_train <- t(X_train)
y_train <- as.matrix(y_train, byrow=TRUE)
y_train <- t(y_train)
X_test <- as.matrix(X_test, byrow=TRUE)
X_test <- t(X_test)
y_test <- as.matrix(y_test, byrow=TRUE)
y_test <- t(y_test)
```

6. Get layer sizes
   The AAN we are constructing has a single hidden layer with four nodes. Define a list with an item per layer (n_x, n_l, n_y) each with the respective layer size (number of nodes of the input, hidden and output layers)

```
getLayerSize <- function(X, y, hidden_neurons) {
  n_x <- dim(X)[1]
  n_h <- hidden_neurons
  n_y <- dim(y)[1]
  size <- list("n_x" = n_x,
               "n_h" = n_h,
               "n_y" = n_y)
  return(size)
}
layer_size <- getLayerSize(X_train, y_train, hidden_neurons = 4)
layer_size
```

7. Initialise parameters
   Use the following code to initialize parameters

```
initializeParameters <- function(X, list_layer_size){
  m <- dim(data.matrix(X))[2]
  n_x <- list_layer_size$n_x
  n_h <- list_layer_size$n_h
  n_y <- list_layer_size$n_y
  W1 <- matrix(runif(n_h * n_x), nrow = n_h, ncol = n_x, byrow = TRUE) * 0.01
  b1 <- matrix(rep(0, n_h), nrow = n_h)
  W2 <- matrix(runif(n_y * n_h), nrow = n_y, ncol = n_h, byrow = TRUE) * 0.01
  b2 <- matrix(rep(0, n_y), nrow = n_y)
  params <- list("W1" = W1,
                 "b1" = b1,
                 "W2" = W2,
                 "b2" = b2)
  return (params)
}
init_params <- initializeParameters(X_train, layer_size)
lapply(init_params, function(x) dim(x))
```

8. Define the Activation Functions
   Define the sigmoid activation function

```
sigmoid <- function(x){
  return(1 / (1 + exp(-x)))
}
```

9. Forward Propagation
   Use the following code to do the forward propagation step

```
forwardPropagation <- function(X, params, list_layer_size){
  m <- dim(X)[2]
  n_h <- list_layer_size$n_h
  n_y <- list_layer_size$n_y
  W1 <- params$W1
  b1 <- params$b1
  W2 <- params$W2
  b2 <- params$b2
  b1_new <- matrix(rep(b1, m), nrow = n_h)
  b2_new <- matrix(rep(b2, m), nrow = n_y)
  Z1 <- W1 %*% X + b1_new
  A1 <- sigmoid(Z1)
  Z2 <- W2 %*% A1 + b2_new
  A2 <- sigmoid(Z2)
  cache <- list("Z1" = Z1,
                "A1" = A1,
                "Z2" = Z2,
                "A2" = A2)

  return (cache)
}
fwd_prop <- forwardPropagation(X_train, init_params, layer_size)
```

```
lapply(fwd_prop, function(x) dim(x))
```

10. Compute Cost

    Use the following code to compute the costs

```
computeCost <- function(X, y, cache) {
  m <- dim(X)[2]
  A2 <- cache$A2
  logprobs <- (log(A2) * y) + (log(1-A2) * (1-y))
  cost <- -sum(logprobs/m)
  return (cost)
}
cost <- computeCost(X_train, y_train, fwd_prop)
cost
```

11. Backpropagation

    Use the following code to compute the backward propagation step

```
backwardPropagation <- function(X, y, cache, params, list_layer_size){
  m <- dim(X)[2]
  n_x <- list_layer_size$n_x
  n_h <- list_layer_size$n_h
  n_y <- list_layer_size$n_y
  A2 <- cache$A2
  A1 <- cache$A1
  W2 <- params$W2
  dZ2 <- A2 - y
  dW2 <- 1/m * (dZ2 %*% t(A1))
  db2 <- matrix(1/m * sum(dZ2), nrow = n_y)
  db2_new <- matrix(rep(db2, m), nrow = n_y)
  dZ1 <- (t(W2) %*% dZ2) * (1 - A1^2)
  dW1 <- 1/m * (dZ1 %*% t(X))
  db1 <- matrix(1/m * sum(dZ1), nrow = n_h)
  db1_new <- matrix(rep(db1, m), nrow = n_h)
  grads <- list("dW1" = dW1,
              "db1" = db1,
              "dW2" = dW2,
              "db2" = db2)
  return(grads)
}
back_prop<-backwardPropagation(X_train,y_train,fwd_prop,init_params,layer_size)
lapply(back_prop, function(x) dim(x))
```

12. Update Parameters

    Use the following code to compute the backward propagation step

```
updateParameters <- function (grads, params, learning_rate){
  W1 <- params$W1
  b1 <- params$b1
  W2 <- params$W2
  b2 <- params$b2
  dW1 <- grads$dW1
  db1 <- grads$db1
  dW2 <- grads$dW2
  db2 <- grads$db2
  W1 <- W1 - learning_rate * dW1
  b1 <- b1 - learning_rate * db1
  W2 <- W2 - learning_rate * dW2
  b2 <- b2 - learning_rate * db2
  updated_params <- list("W1" = W1,
                         "b1" = b1,
                         "W2" = W2,
                         "b2" = b2)
  return (updated_params)
}
update_params <- updateParameters(back_prop, init_params, learning_rate = 0.01)
lapply(update_params, function(x) dim(x))
```

13. Train the Model

    Use the following code to train the model

```
trainModel <- function(X, y, num_iteration, hidden_neurons, lr){
  layer_size <- getLayerSize(X, y, hidden_neurons)
  init_params <- initializeParameters(X, layer_size)
  cost_history <- c()
  for (i in 1:num_iteration) {
    fwd_prop <- forwardPropagation(X, init_params, layer_size)
    cost <- computeCost(X, y, fwd_prop)
    back_prop <- backwardPropagation(X, y, fwd_prop, init_params, layer_size)
    update_params <- updateParameters(back_prop, init_params, learning_rate = lr)
    init_params <- update_params
    cost_history <- c(cost_history, cost)
    if (i %% 10000 == 0) cat("Iteration", i, " | Cost: ", cost, "\n")
  }
  model_out <- list("updated_params" = update_params,
                    "cost_hist" = cost_history)
  return (model_out)
}
EPOCHS = 60000
HIDDEN_NEURONS = 40
LEARNING_RATE = 0.1
```

```
train_model <- trainModel(X_train, y_train, hidden_neurons = HIDDEN_NEURONS,
num_iteration = EPOCHS, lr = LEARNING_RATE)
```

14. Logistic Regression
    Use the glm function to train a logistic regression model and then use it to predict the Species' values of the test set

```
lr_model <- glm(Species ~ Sepal.Length + Sepal.Width, data = train)
lr_model
lr_pred <- round(as.vector(predict(lr_model, test[, 1:2])))
lr_pred
```

15. Test the Model
    Use the forwardPropagation function to predict the values of the test set

```
layer_size <- getLayerSize(X_test, y_test, HIDDEN_NEURONS)
params <- train_model$updated_params
fwd_prop <- forwardPropagation(X_test, params, layer_size)
y_pred <- round(fwd_prop$A2)
```

16. Confusion Matrix
    Use the table function to calculate the confusion matrix of both the logistic regression model and the ANN model. Then, uses the confusion matrix to calculate the values of accuracy, precision, recall and F1 for each of the two models.

```
tb_nn <- table(y_test, y_pred)
tb_lr <- table(y_test, lr_pred)

cat("NN Confusion Matrix: \n")
## NN Confusion Matrix:
tb_nn
cat("\nLR Confusion Matrix: \n")
tb_lr
calculate_stats <- function(tb, model_name) {
  acc <- (tb[1] + tb[4])/(tb[1] + tb[2] + tb[3] + tb[4])
  recall <- tb[4]/(tb[4] + tb[3])
  precision <- tb[4]/(tb[4] + tb[2])
  f1 <- 2 * ((precision * recall) / (precision + recall))

  cat(model_name, ": \n")
  cat("\tAccuracy = ", acc*100, "%.")
  cat("\n\tPrecision = ", precision*100, "%.")
  cat("\n\tRecall = ", recall*100, "%.")
  cat("\n\tF1 Score = ", f1*100, "%.\n\n")
}
calculate_stats(tb_nn,"NN")
calculate_stats(tb_lr,"LR")
```