# Atividade_3_Backpropagation

Adriel Martins

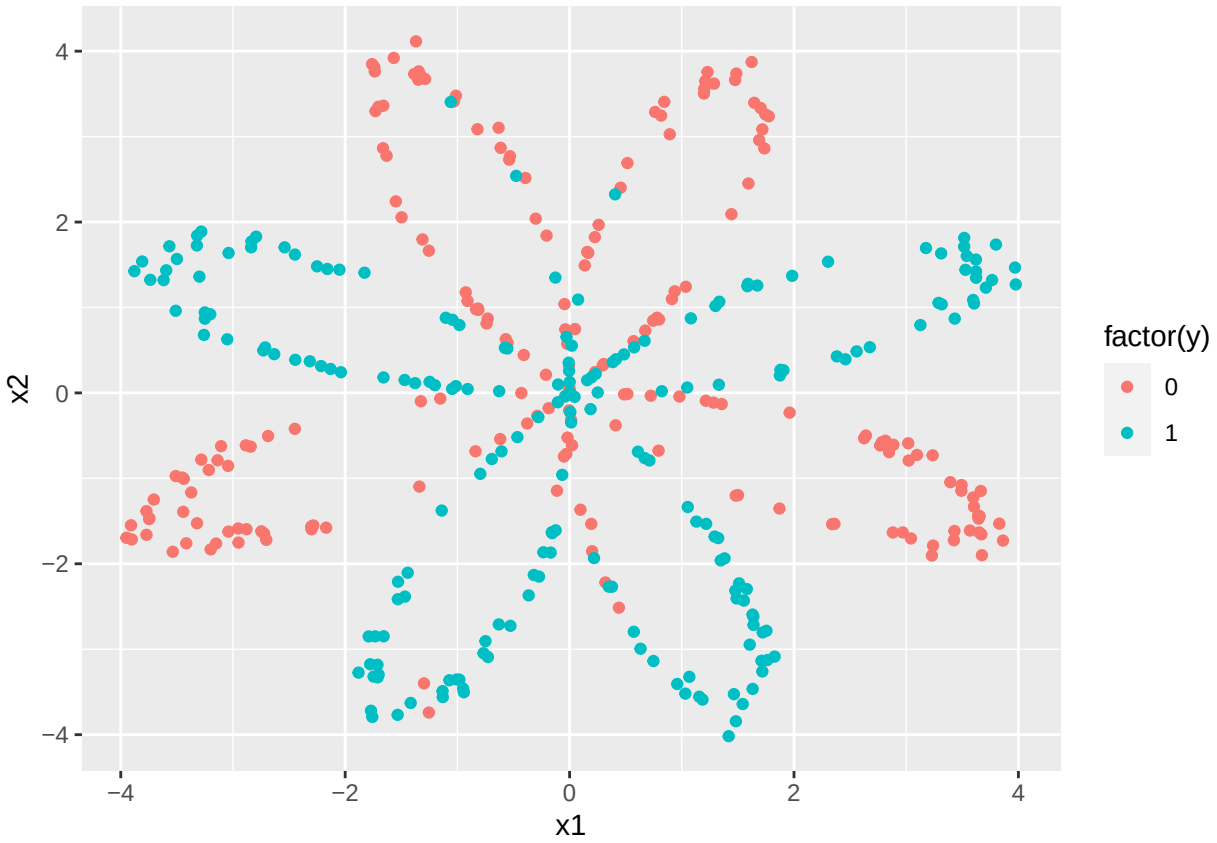
3/8/2021

## ET645.A2: Explicado

### Os dados

Isse trabalho se propõe a explicar os processo de estimação dos parâmetros das Redes Neurais como parte da disciplia ET645 do curso de Estatística da UFPE.

```
set.seed(69)

planar_dataset <- function(){
    m <- 400
    N <- m/2
    D <- 2
    X <- matrix(0, nrow = m, ncol = D)
    Y <- matrix(0, nrow = m, ncol = 1)
    a <- 4

for(j in 0:1){
    ix <- seq((N*j)+1, N*(j+1))
    t <- seq(j*3.12,(j+1)*3.12,length.out = N) + rnorm(N, sd = 0.2)
    r <- a*sin(4*t) + rnorm(N, sd = 0.2)
    X[ix,1] <- r*sin(t)
    X[ix,2] <- r*cos(t)
    Y[ix,] <- j
    }

d <- as.data.frame(cbind(X, Y))
names(d) <- c('x1','x2','y')
return(d)
}


df <- planar_dataset()

ggplot(df, aes(x = x1, y = x2, color = factor(y))) +
geom_point()
```




Temos como objetivo de classificar a variável y em função de duas variáveis explicativas x1 e x2.

```
# Base Artificial ###################################
set.seed(69)
df <- df[sample(nrow(df)), ]
fraction = .8

train_test_split_index <- fraction * nrow(df)

# Train
train <- df[1:train_test_split_index,]
head(train)
```

```
##             x1          x2 y
## 209 -0.1241424 -1.60860366 1
## 347  1.5798432 -2.29534600 1
## 386  1.6394224 -2.71539412 1
## 112  0.4862432 -0.01708582 0
## 104  2.7827754 -0.57753602 0
## 111  2.6408498 -0.49882334 0
```

```
##### Test
test <- df[(train_test_split_index+1): nrow(df),]
head(test)
```

```
##             x1         x2 y
## 210 -0.3186974 -2.1311652 1
## 348 -2.8349787  1.7721735 1
```

```
## 19   1.4865982  3.7380291 0
## 362  1.6058978 -2.9465299 1
## 143  0.4107194 -0.3788482 0
## 4    0.3193160 -2.2186398 0
```

```
####################################

# Normalização e base de treinamento e teste ###################################

X_train <- scale(train[, c(1:2)])

y_train <- train$y
dim(y_train) <- c(length(y_train), 1) # dimensão extra

X_test <- scale(test[, c(1:2)])

y_test <- test$y
dim(y_test) <- c(length(y_test), 1) # dimensão extra
```

Houve primeiramente uma divisão entre dados de treino e de teste. Assim, poderemos testar a acurácia e fazer certos diagnóstico sobre o nosso modelo treinado apenas com os dados de treino.

Depois disso houve a normalização seguindo a seguinte fórmula:

$$X_{new} = \frac{(X - E(X))}{\sqrt{Var(X)}}$$

```
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)
```

Temos nossas matrizes de treino e teste. Focando na de treino: A matriz $X_{2,n}$ temos que cada linha representa uma variável (x1, e x2)

## Arquitetura da Rede Neural

```
getLayerSize <- function(X, y, hidden_neurons, train=TRUE) {
  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)
}
```

Uma simples função de cálculo das dimensões para montar a estrutura da Rede Neural. Onde ela está basicamente feita seguindo a fórmula da Figura 1.

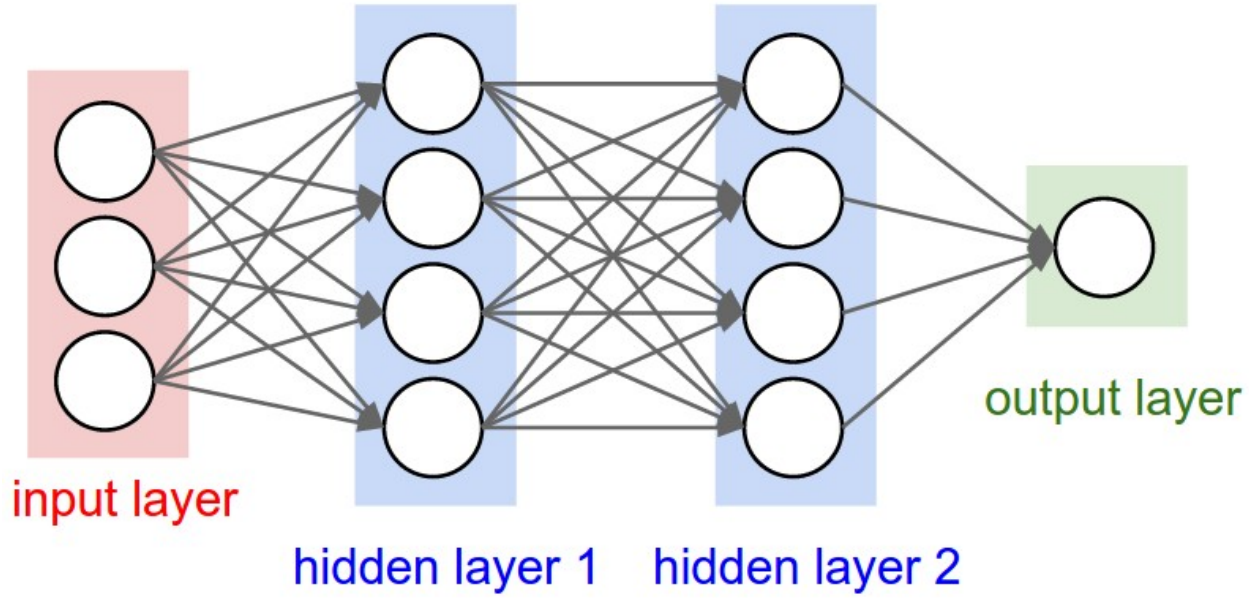



Figure 1: Alt text

Os nódulos se conectam seguindo esta ordem:

n_x nódulos -> n_h nódulos -> n_h nódulos

Diremos que esta rede neural só tem uma camada, porque os dados de $X_{train}$ está tendo contado somente com os nódulos de n_h, sendo que este conjunto é chamado de camada.

```
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, min = -.01, max = .01), nrow = n_h, ncol = n_x, byrow = TRUE)
    b1 <- matrix(rep(0, n_h), nrow = n_h)
    W2 <- matrix(runif(n_y * n_h, min = -.01, max = .01), nrow = n_y, ncol = n_h, byrow = TRUE)
    b2 <- matrix(rep(0, n_y), nrow = n_y)

    params <- list("W1" = W1,
                   "b1" = b1,
                   "W2" = W2,
                   "b2" = b2)

    return (params)
}
```

Como foi dito anteriormente há duas camadas, ou seja conjunto de nódulos antes de chegar a camada final que nos dará as probabilidades de cada categoria. Cada nódulo da camada pode ser visto como um peso que irá ponderar o valor recebido. Pode-se observar também que cada entrada de $X_{train}$ é conectada com cada um dos nódulos. Isto forma o que chamamos de *full-connected layer*. Sendo assim, a camada é na verdade uma matriz que irá pré-multiplicar os valores de entrada. Além disso, iremos adicionar uma variável constante

para cada nódulo. Ou seja, na nossa matriz de multiplicação iremos adicionar um vetor de variáveis.

Ficamos com o seguinte entendimento, Z sendo o resultado das conexões entre a camada de entrada dos dados e a camada, ou seja o vetor resultante das operações lineares:

$$Z = WX + b$$

Inicialmente, colocamos pesos aleatórios, os quais serão otimizados seguindo funções de perda e métodos de otimização. Chamaremos este processo de estimação dos parâmetros.

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

Ao finalizarmos os resultados das operações de lineares que irão definir $Z$, gostaríamos de jogar interpretações inteligenete sobre os resultados. No nosso caso, iremos escolher a função sigmoíde. Ou seja, quanto maior o resultado de cada nódulo mais perto de um o resultado; quanto menor o resultado do nódulo de Z, mais perto de 0 o resultado. O que nos dá uma interpretação interessante sobre os resultados. Essa função que escalaciona os resultados é chamada "função de ativação".

Sendo assim, temos definido a arquitetura da rede neural, um conjunto de matrizes de peso que irão ponderar as entradas de nossa variável nos dando um output que deverá definir as probabilidade de ser cada categoria da variável resposta.

## Foward-Propagation

```
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)
}
```

O processo de estimação dos parâmetros pode ser divido entre Foward-Propagation e Backward Propagation. Comecemos pelo primeiro.

Ele representa aquilo que mencionamos anteriormente sobre as operações lineares entre os nódulos. Só que isto de maneira encadeada, por isto o nome. O método sempre joga para a frente o resultado passado.

No nosso caso de duas camadas, temos:

$$Z_1 = W_1 * X + b_1$$
$$Z_2 = W_2 * Z_1 + b_2$$

Veja como há esta dependência do resultado anterior.

```
layer_size <- getLayerSize(X_train, y_train, hidden_neurons = 4)
init_params <- initializeParameters(X_train, layer_size)

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

```
## $W1
## [1] 4 2
##
## $b1
## [1] 4 1
##
## $W2
## [1] 1 4
##
## $b2
## [1] 1 1
```

```
fwd_prop <- forwardPropagation(X_train, init_params, layer_size)
lapply(fwd_prop, function(x) dim(x))
```

```
## $Z1
## [1]   4 320
##
## $A1
## [1]   4 320
##
## $Z2
## [1]   1 320
##
## $A2
## [1]   1 320
```

## Backward-propagation

```
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)
}
```

Para começarmos o processo de BP, temos que definir nossa função de erro ou de custo. Ou seja, uma comparação quantitativa do quanto o nosso output está distan- te da realidade. No nosso caso, estamos escolhendo a métrica da Entropia Cruzada.

$$H(p,q) = -\sum_{x \in \mathcal{X}} p(x) \log q(x)$$

```
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)
}
```

O processo aqui realizado é intuitivamente entendido como a partir do erro, a partir do final do processo, eu entender como cada nódulo teve seu efeito no erro e atualizar na direção de diminuir este erro.

A maneira como se faz isto é utilizando as funções de ativação de cada nódulo. A contribuição no erro de cada nódulo será dada em relação a função de ativação da última camada, ou seja a sua derivada, o impacto médio na função. Só que, pelo processo de Fowarding-Propagation, eu tenho que todas as funções na frente, são definidas pelos processos anteriores. Ou seja, eu precico calcular as derivadas de cada nódulo em relação a sua camada utilizando resultados dos nódulos da frente.

```
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)
}
```

Uma vez calculado as contribuições no erro, iremos utilizar estes valores para realizar um update nos pesos dos nódulos na direção de minimizar o erro. Porém, iremos ponderar esse impacto através de "learning rate", onde será arbitŕario para nós.

Este método é chamado de "stochastic gradient", ou seja, pelo vetor de gradientes, as derivadas, eu tento diminuir a função de perda para cada nódulo.

## Treinando o modelo

```
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 = 0.9


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

```
## Iteration 10000  | Cost:  0.3064668
```

```
## Iteration 20000  | Cost:  0.2824738
## Iteration 30000  | Cost:  0.2473587
## Iteration 40000  | Cost:  0.2994775
## Iteration 50000  | Cost:  0.2221273
## Iteration 60000  | Cost:  0.6030751
```

Como podemos ver o treinar o modelo é apenas realizar o FP e BP para cada um das colunas de $X_{train}$, que representa as observações das nossas variáveis x1 e x2.

Porém, iremos definir também que não só faremos isto uma vez. Repetiremos o processo completo "EPOCHS" vezes. Onde cada EPOCH significa uma rodada inteira utilizando $X_{train}$ e $y_{train}$.

## Predição e Diagnósticos

```
lr_model <- glm(y ~ x1 + x2, data = train)
lr_model
```

```
##
## Call:  glm(formula = y ~ x1 + x2, data = train)
##
## Coefficients:
## (Intercept)           x1           x2
##     0.49646     -0.01300     -0.05203
##
## Degrees of Freedom: 319 Total (i.e. Null);  317 Residual
## Null Deviance:       80
## Residual Deviance: 76.7  AIC: 459
```

```
lr_pred <- round(as.vector(predict(lr_model, test[, 1:2])))
lr_pred
```

```
##  [1] 1 0 0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 1 0 1 1 0 1 0 0 0 1 0 1 0 1 0 1 0 1 1 0 0
## [39] 1 1 1 0 1 0 1 1 0 0 0 1 0 1 1 1 0 0 0 0 1 0 1 1 0 0 1 0 1 0 1 1 0 1 0 1 0 1
## [77] 1 1 0 1
```

Iremos comparar com um simples GLM.

```
makePrediction <- function(X, y, hidden_neurons){
    layer_size <- getLayerSize(X, y, hidden_neurons)
    params <- train_model$updated_params
    fwd_prop <- forwardPropagation(X, params, layer_size)
    pred <- fwd_prop$A2

    return (pred)
}
```

A predição é exatamente o FP. Onde iremos ponderar as entradas pelos pesos e depois colocar o resultado na função de ativação.

```
y_pred <- makePrediction(X_test, y_test, HIDDEN_NEURONS)
y_pred <- round(y_pred)
```

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

cat("NN Confusion Matrix: \n")
```

```
## NN Confusion Matrix:
```

```
tb_nn
```

```
##       y_pred
## y_test  0  1
##      0 16 25
##      1  6 33
```

```
cat("\nLR Confusion Matrix: \n")
```

```
##
## LR Confusion Matrix:
```

```
tb_lr
```

```
##       lr_pred
## y_test  0  1
##      0 24 17
##      1 15 24
```

Podemos ver as métricas de diagnóstico. A matriz de confusão nos diz quantas observações acertamos nos dados de teste, e quanto erramos. Iremos realizar o diagnóstico em dados nunca antes observado pela nossa rede para não nos preocupar- mos com overfitting ou viés.

```
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, "Redes Neurais")
```

```
## Redes Neurais :
##  Accuracy =  61.25 %.
##  Precision =  84.61538 %.
##  Recall =  56.89655 %.
##  F1 Score =  68.04124 %.
```

```
calculate_stats(tb_lr, "Regressão Logística")
```

```
## Regressão Logística :
##  Accuracy =  60 %.
##  Precision =  61.53846 %.
##  Recall =  58.53659 %.
##  F1 Score =  60 %.
```

Podemos assim ver bons resultados da nossa rede neural, em comparação ao GLM.