Classification algorithm - Neural Networks

Eric Roseren

5/9/2019

Part 1 - Logistic Regression

Introduction

In this section we will implement an L-layer neural network from scratch and use it for classification tasks.

We will first start with a 1 layer Neural Network with a sigmoid activation function so that it is equivalent to a standart logistic regression function.

We will then procede to increase the number of layer and unit cell to obtain a L- layer Neural Network with appropriate activation function and regularization method (L-2 norm, dropout) to reduce overfitting.

To load a dataset that is stored on an H5 file we can use the rhdf5 package from Bioconductor as follow:

```
#source("http://bioconductor.org/biocLite.R") # Install package from bioconductor sou
rce
#biocLite("rhdf5")
library(rhdf5) # load package
library(countcolors)  # plot (rgb array)
```

```
## Warning: package 'countcolors' was built under R version 3.5.2
```

Guidelines: *List the objects within the file to find the data group you want to read:

```
#h5ls("train_catvnoncat.h5")
f <- "train_catvnoncat.h5"
# view the structure of the H5 file
h5ls(f, all = TRUE)
```

gr…	name	ltype	corder_valid	corder	c…	otype	num_attrs	dc
<chr>	<chr>	<fctr>	<lgl>	<int>	<int>	<fctr>	<int>	<cl
0 /	list_classes	H5L_TYPE_HARD	FALSE	0	0	H5I_DATASET	0	ST
1 /	train_set_x	H5L_TYPE_HARD	FALSE	0	0	H5I_DATASET	0	INT
2 /	train_set_y	H5L_TYPE_HARD	FALSE	0	0	H5I_DATASET	0	INT

3 rows | 1-10 of 15 columns

*Read the HDF5 data:

```
load.dataset <- function(){
#Standardised training set
train.x <- aperm(h5read("train_catvnoncat.h5","train_set_x"))/255
train.y <- t(h5read("train_catvnoncat.h5","train_set_y"))


# Standardised test set
test.x <- aperm(h5read("test_catvnoncat.h5","test_set_x"))/255
test.y <- t(h5read("test_catvnoncat.h5","test_set_y"))

# classes

classes = h5read("test_catvnoncat.h5","list_classes")
output <- list(train.x=train.x,train.y=train.y,
               test.x=test.x,test.y=test.y,classes=classes)
return(output)
}
```

```
original.data <- load.dataset()
```

The data

To represent color images, the red, green and blue channels (RGB) must be specified for each pixel, and so the pixel value is actually a vector of three numbers ranging from 0 to 255.

One common preprocessing step in machine learning is to center and standardize your dataset, meaning that you substract the mean of the whole numpy array from each example, and then divide each example by the standard deviation of the whole numpy array. But for picture datasets, it is simpler and more convenient and works almost as well to just divide every row of the dataset by 255 (the maximum value of a pixel channel).

Let's visualise the 25th image:

```
index <- 25
m <- original.data$train.x[index,,,]
result <-as.numeric( t(original.data$train.y[index]))
cat.noncat <- original.data$classes[result+1]
plotArrayAsImage(m, main = paste("y= ",result,", it is a ",cat.noncat," image ",sep =
"" ))
```

y= 1, it is a cat image

We will be performing a multitude of matrix multiplication so having in mind the dimensions of the data training and test set will be helpful.

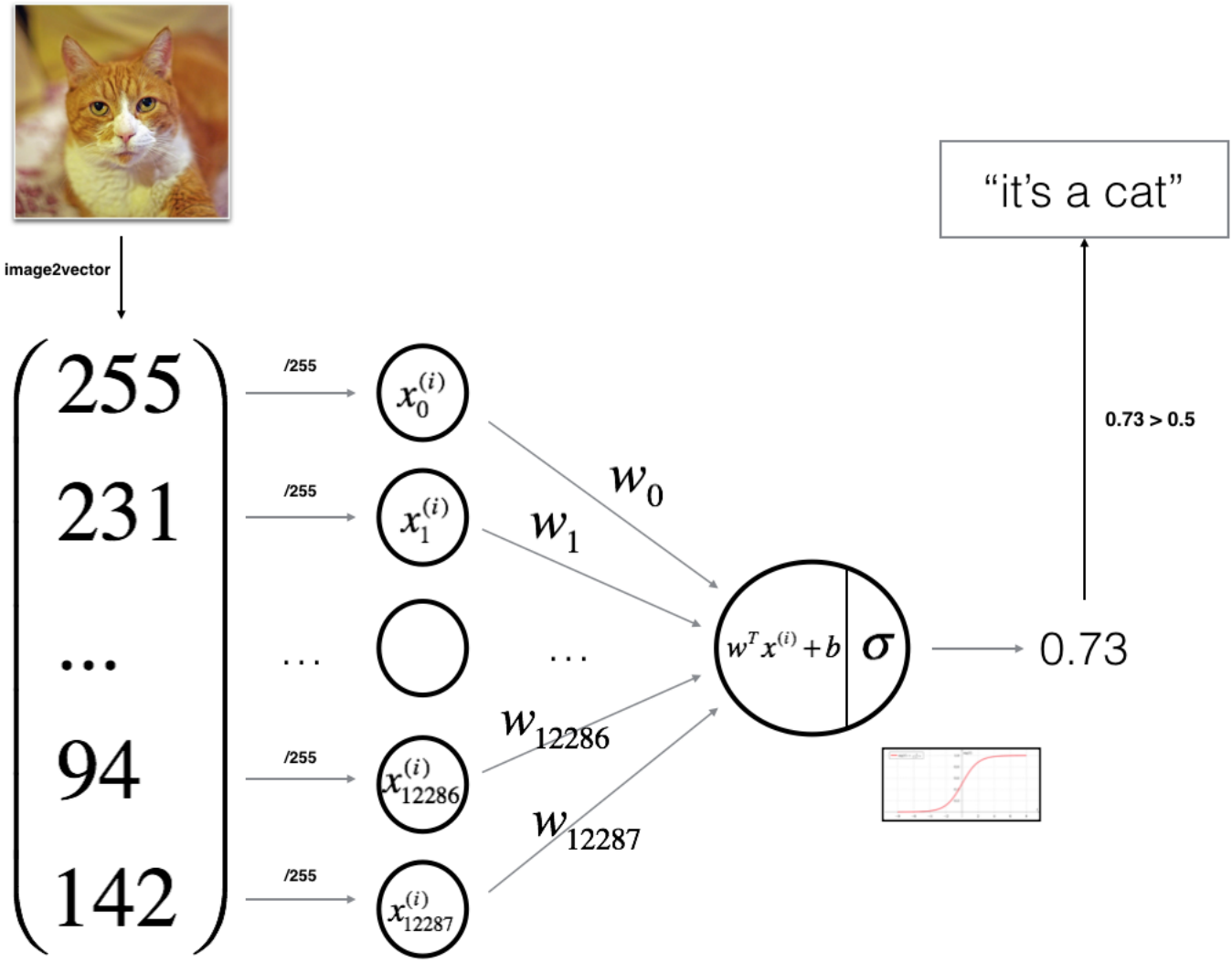
```
m.train <- dim(original.data$train.x)[1]
m.test <- dim(original.data$test.x)[1]
num.px <- dim(original.data$train.x)[2]
```

Remember that train.set.x is an array of shape (m.train, num.px, num.px, 3). To feed the images to the neural network, the dimension of the array needs to be flatten. The dimension of the new array will have the following shape: (num.px ∗ num.px ∗ 3, 1).

```
train.x.flatten <- matrix(data = aperm(original.data$train.x),num.px*num.px*3,m.trai
n)
test.x.flatten <- matrix(data = aperm(original.data$test.x),num.px*num.px*3,m.test)
```

General Architecture of the learning algorithm

The following Figure explains why Logistic Regression is actually a very simple Neural Network!



Mathematical expression of the algorithm:

For one example $x^{(i)}$:

$$z^{(i)} = w^T x^{(i)} + b \tag{1}$$

$$\hat{y}^{(i)} = a^{(i)} = sigmoid(z^{(i)}) \tag{2}$$

$$\mathcal{L}(a^{(i)}, y^{(i)}) = -y^{(i)} \log(a^{(i)}) - (1 - y^{(i)}) \log(1 - a^{(i)}) \tag{3}$$

The cost is then computed by summing over all training examples:

$$J = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(a^{(i)}, y^{(i)}) \tag{6}$$

4 - Building the parts of our algorithm

The main steps for building a Neural Network are: 1. Define the model structure (such as number of input features) 2. Initialize the model's parameters 3. Loop: - Calculate current loss (forward propagation) - Calculate current gradient (backward propagation) - Update parameters (gradient descent)

```
# Sigmoid function

sigmoid <- function(z){
  "
    Compute the sigmoid of z

    Arguments:
    z -- A scalar or array of any size.

    Return:
    s -- sigmoid(z)
    "
  s <- 1/(1+exp(-z))
  return(s)
}
```

```
print (paste("sigmoid([0, 2]) = ", as.numeric(sigmoid(array(data=c(0,2)))),sep = ""))
```

```
## [1] "sigmoid([0, 2]) = 0.5"
## [2] "sigmoid([0, 2]) = 0.880797077977882"
```

4.2 - Initializing parameters

We need to initialise the parameters w and b to zero:

```
initialise.to.zero <- function(shape){
  "
    This function creates a vector of zeros of dimension (shape, 1) for w and initial
izes b to 0.

    Argument:
    dim -- size of the w vector we want (or number of parameters in this case)

    Returns:
    w -- initialized vector of shape (dim, 1)
    b -- initialized scalar (corresponds to the bias)
    "
 w <- matrix(data = rep(0,shape),nrow = shape,ncol = 1)
 b <- 0
 result <- list(w=w,b=b)
 return(result)
}
```

4.3 - Forward and Backward propagation

Forward Propagation: We get X:

$$A = \sigma(w^T X + b) = (a^{(1)}, a^{(2)}, \ldots, a^{(m-1)}, a^{(m)})$$

We calculate the cost function:

$$J = \frac{1}{m} \sum_{i=1}^{m} y^{(i)} \log(a^{(i)}) + (1 - y^{(i)}) \log(1 - a^{(i)})$$

Here are the two formulas :

$$\frac{\partial J}{\partial w} = \frac{1}{m} X(A - Y)^T \tag{7}$$

$$\frac{\partial J}{\partial b} = \frac{1}{m} \sum_{i=1}^{m} (a^{(i)} - y^{(i)}) \tag{8}$$

```
propagate <- function(w, b, X, Y){
    "
    Implement the cost function and its gradient for the propagation explained above

    Arguments:
    w -- weights, a numpy array of size (num_px * num_px * 3, 1)
    b -- bias, a scalar
    X -- data of size (num_px * num_px * 3, number of examples)
    Y -- true label vector (containing 0 if non-cat, 1 if cat) of size (1, number of
 examples)

    Return:
    cost -- negative log-likelihood cost for logistic regression
    dw -- gradient of the loss with respect to w, thus same shape as w
    db -- gradient of the loss with respect to b, thus same shape as b


    "

    m <-  dim(X)[2]

    # FORWARD PROPAGATION (FROM X TO COST)
    A <- sigmoid((t(w) %*% X)+b)                                    # compute activat
ion
    cost <- (-1/m)*sum(Y*log(A)+(1-Y)*log(1-A))                                 # com
pute cost

    # BACKWARD PROPAGATION (TO FIND GRAD)
    dw <-  (1/m)*X %*% t((A-Y))
    db <-  (1/m)*sum(A-Y)

    stopifnot(dim(dw) == dim(w))

    #cost <-  np.squeeze(cost)

    grads = list(dw=dw,db=db)

    results <- list(grads=grads,cost=cost)

    return (results)
}
```

```
w <- array(data = c(1,2),dim = c(2,1))
b <- 2
X <- array(data =c(1,3,2,4,-1,-3.2),dim = c(2,3))
Y <- array(data =c(1,0,1),dim = c(1,3))

propagate(w,b,X,Y)
```

```
## $grads
## $grads$dw
##           [,1]
## [1,] 0.998456
## [2,] 2.395072
##
## $grads$db
## [1] 0.001455578
##
##
## $cost
## [1] 5.801545
```

4.4 - Optimization

You have initialized your parameters.You are also able to compute a cost function and its gradient. Now, you want to update the parameters using gradient descent.

```
optimize <- function(w, b, X, Y, num.iterations, learning.rate, print.cost = F){
    "
    This function optimizes w and b by running a gradient descent algorithm

    Arguments:
    w -- weights, a numpy array of size (num_px * num_px * 3, 1)
    b -- bias, a scalar
    X -- data of shape (num_px * num_px * 3, number of examples)
    Y -- true label vector (containing 0 if non-cat, 1 if cat), of shape (1, number o
f examples)
    num.iterations -- number of iterations of the optimization loop
    learning.rate -- learning rate of the gradient descent update rule
    print.cost -- True to print the loss every 100 steps

    Returns:
    params -- dictionary containing the weights w and bias b
    grads -- dictionary containing the gradients of the weights and bias with respect
to the cost function
    costs -- list of all the costs computed during the optimization, this will be use
d to plot the learning curve.

    Tips:
    You basically need to write down two steps and iterate through them:
        1) Calculate the cost and the gradient for the current parameters. Use propag
ate().
        2) Update the parameters using gradient descent rule for w and b.
    "

    costs = NULL

    for (i in 1:num.iterations){


        # Cost and gradient calculation (≈ 1-4 lines of code)
        temp.val <- propagate(w, b, X, Y)
        grads <-  temp.val$grads
        cost <-  temp.val$cost

        # Retrieve derivatives from grads
        dw <-  grads$dw
        db <-  grads$db

        # update rule (≈ 2 lines of code)

        w <-  w-learning.rate*dw
        b <-  b-learning.rate*db

        # Record the costs
        if (i %% 100 == 0){
            costs[i] <- cost
        }
        # Print the cost every 100 training iterations
        if (print.cost & i %% 100 == 0){
            print (paste("Cost after iteration ", i,": ",cost,sep = ""))
        }
    }
```

```
    params <-  list(w= w,b=b)
    grads <-  list(dw= dw,db= db)

    out <- list(params=params,grads=grads,cost=cost)

    return (out)
}
```

```
optimize(w, b, X, Y, num.iterations= 100, learning.rate = 0.009, print.cost = F)
```

```
## $params
## $params$w
##           [,1]
## [1,] 0.1903359
## [2,] 0.1225916
##
## $params$b
## [1] 1.92536
##
##
## $grads
## $grads$dw
##           [,1]
## [1,] 0.6775204
## [2,] 1.4162550
##
## $grads$db
## [1] 0.2191945
##
##
## $cost
## [1] 1.078431
```

The previous function will output the learned w and b. We are able to use w and b to predict the labels for a dataset X. To implement the `predict()` function. There are two steps to computing predictions:

1. Calculate

$$\hat{Y} = A = \sigma(w^T X + b)$$

2. We convert the entries of a into 0 (if activation <= 0.5) or 1 (if activation > 0.5), stores the predictions in a vector `Y_prediction` .We can use an `if` / `else` statement in a `for` loop (though there is also a way to vectorize this).

```
predict <- function(w, b, X){
    "
    Predict whether the label is 0 or 1 using learned logistic regression parameters
 (w, b)

    Arguments:
    w -- weights, a numpy array of size (num_px * num_px * 3, 1)
    b -- bias, a scalar
    X -- data of size (num_px * num_px * 3, number of examples)

    Returns:
    Y.prediction -- a numpy array (vector) containing all predictions (0/1) for the e
xamples in X
    "

    m <-  dim(X)[2]
    Y.prediction <-  matrix(data=rep(0,m),ncol = m)


    # Compute vector "A" predicting the probabilities of a cat being present in the p
icture

    A <- sigmoid((t(w) %*% X)+b)                                    # compute activat
ion

    for (i in 1:m){

        # Convert probabilities A[1,i] to actual predictions p[1,i]
        if (A[1,i]> 0.5){
            Y.prediction[1,i] <- 1
        }
        else{
            Y.prediction[1,i] <- 0
        }
    }

    return (Y.prediction)
}
```

```
w <- array(data = c(0.1124579,0.23106775),dim = c(2,1))
b <- -0.3
X <- array(data =c(1,1.2,-1.1,2,-3.2,0.1),dim = c(2,3))

predict(w,b,X)
```

```
##      [,1] [,2] [,3]
## [1,]    1    1    0
```

5 - Merge all functions into a model

We will now see how the overall model is structured by putting together all the building blocks (functions implemented in the previous parts) together, in the right order.

```
model <- function(X.train, Y.train, X.test, Y.test, num.iterations = 2000, learning.r
ate = 0.5, print.cost = F){
    "
    Builds the logistic regression model by calling the function you've implemented p
reviously

    Arguments:
    X.train -- training set represented by a numpy array of shape (num_px * num_px *
 3, m_train)
    Y.train -- training labels represented by a numpy array (vector) of shape (1, m_t
rain)
    X.test -- test set represented by a numpy array of shape (num_px * num_px * 3, m_
test)
    Y.test -- test labels represented by a numpy array (vector) of shape (1, m_test)
    num.iterations -- hyperparameter representing the number of iterations to optimiz
e the parameters
    learning.rate -- hyperparameter representing the learning rate used in the update
rule of optimize()
    print.cost -- Set to true to print the cost every 100 iterations

    Returns:
    d -- dictionary containing information about the model.
    "

    # initialize parameters with zeros (≈ 1 line of code)
    m <- dim(X.train)[1]
    w <- initialise.to.zero(m)$w
    b <- initialise.to.zero(m)$b

    # Gradient descent (≈ 1 line of code)
    temp <- optimize(w, b, X.train, Y.train, num.iterations, learning.rate,print.cost
= T)
    parameters <- temp$params
    grads <-  temp$grads
    costs <-  temp$costs


    # Retrieve parameters w and b from dictionary "parameters"
    w <-  parameters$w
    b <-  parameters$b

    # Predict test/train set examples (≈ 2 lines of code)
    Y.prediction.test <-  predict(w,b,X.test)
    Y.prediction.train <-  predict(w,b,X.train)

    # Print train/test Errors
    print(paste("train accuracy: ",(100 - mean(abs(Y.prediction.train - Y.train)) * 1
00),sep = ""))
    print(paste("test accuracy:  ",(100 - mean(abs(Y.prediction.test - Y.test)) * 100
),sep = ""))


    out <-  list(cost=costs,
         Y.prediction.test= Y.prediction.test,
         Y.prediction.train = Y.prediction.train,
         w = w, b = b,
         learning.rate = learning.rate,
```

```
        num.iterations= num.iterations)

    return (out)
}
```

```
train.set.y <- original.data$train.y
test.set.y <- original.data$test.y

d <-  model(train.x.flatten, train.set.y, test.x.flatten, test.set.y, num.iterations
= 3000, learning.rate = 0.005, print.cost = T)
```

```
## [1] "Cost after iteration 100: 0.64489788295317"
## [1] "Cost after iteration 200: 0.484893614148485"
## [1] "Cost after iteration 300: 0.377761495216381"
## [1] "Cost after iteration 400: 0.331775405552359"
## [1] "Cost after iteration 500: 0.303528672026055"
## [1] "Cost after iteration 600: 0.280094277579675"
## [1] "Cost after iteration 700: 0.260225847562819"
## [1] "Cost after iteration 800: 0.243100183998389"
## [1] "Cost after iteration 900: 0.228144327694654"
## [1] "Cost after iteration 1000: 0.214943770696825"
## [1] "Cost after iteration 1100: 0.203189282191046"
## [1] "Cost after iteration 1200: 0.192644280203659"
## [1] "Cost after iteration 1300: 0.183123891531329"
## [1] "Cost after iteration 1400: 0.17448101386887"
## [1] "Cost after iteration 1500: 0.166596753540429"
## [1] "Cost after iteration 1600: 0.159373695098585"
## [1] "Cost after iteration 1700: 0.152731058499392"
## [1] "Cost after iteration 1800: 0.146601146298989"
## [1] "Cost after iteration 1900: 0.140926691669652"
## [1] "Cost after iteration 2000: 0.13565884743937"
## [1] "Cost after iteration 2100: 0.130755639055269"
## [1] "Cost after iteration 2200: 0.126180758519891"
## [1] "Cost after iteration 2300: 0.121902612548046"
## [1] "Cost after iteration 2400: 0.117893562824919"
## [1] "Cost after iteration 2500: 0.114129313270849"
## [1] "Cost after iteration 2600: 0.110588411149568"
## [1] "Cost after iteration 2700: 0.107251837326786"
## [1] "Cost after iteration 2800: 0.104102667072715"
## [1] "Cost after iteration 2900: 0.101125787227981"
## [1] "Cost after iteration 3000: 0.0983076588075479"
## [1] "train accuracy: 99.5215311004785"
## [1] "test accuracy:  68"
```

```
# Example of a picture that was wrongly classified.
index = 6
m <- original.data$test.x[index,,,]
result <-as.numeric(d$Y.prediction.test[1,index])
cat.noncat <- original.data$classes[result+1]
plotArrayAsImage(m, main = paste("y = ",result, ", the model predicted that it is a "
, cat.noncat," image",sep = ""))
```

y = 1, the model predicted that it is a cat image

Part 2 - L- layer Neural Network