Neural Networks practical

Marc Sabaté

08/03/2019

Deep Learning practical in R

In this practical we will build and train a feedforward network with sigmoid activated neurons from scratch.

Load the necessary libraries

We load ggplot2 for plotting purposes. If `ggplot2` is not installed, run first `install.packages('ggplot2')`.

```
library(ggplot2)
```

Data loading and visualisation

Create a directory *DL_practical* in your home directory and save the file *data.txt*. We read the data:

```
HOME = Sys.getenv(x='HOME')
setwd(paste(HOME,"DL_practical",sep="/"))
data = read.table("data.txt", sep=",", header = TRUE)
```

We explore the data. As you can see, each observation of the dataset is two-dimensional with a binary output. The goal of this practical is to fit a binary classifier on the data.

```
summary(data)
```

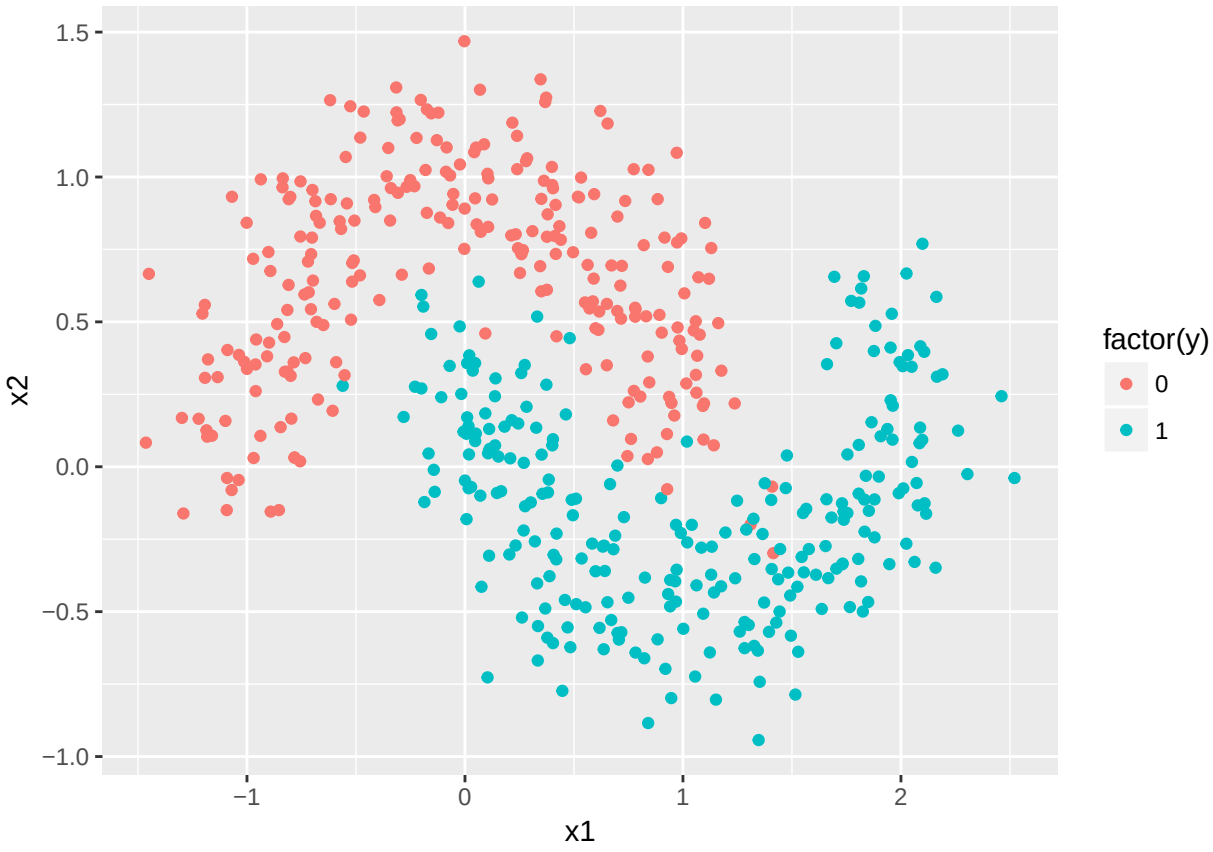
```
##        x1                x2                y
##  Min.   :-1.4637   Min.   :-0.9431   Min.   :0.0
##  1st Qu.:-0.1236   1st Qu.:-0.1601   1st Qu.:0.0
##  Median : 0.4607   Median : 0.2435   Median :0.5
##  Mean   : 0.5003   Mean   : 0.2561   Mean   :0.5
##  3rd Qu.: 1.1205   3rd Qu.: 0.6904   3rd Qu.:1.0
##  Max.   : 2.5208   Max.   : 1.4685   Max.   :1.0
```

```
table(data$y)
```

```
##
##   0   1
## 250 250
```

Execute the following lines to visualise it.

```
p = ggplot(data=data, aes(x1,x2, color=factor(y))) + geom_point()
print(p)
```



Logistic Regression

As a first choice, we try to fit a Logistic regression on the data. For this, we use the R function `glm()` taking into account that its parameter `family` providing the error distribution should be `'binomial'`.

```
model = glm(formula = y~x1+x2, data = data, family = "binomial")
```

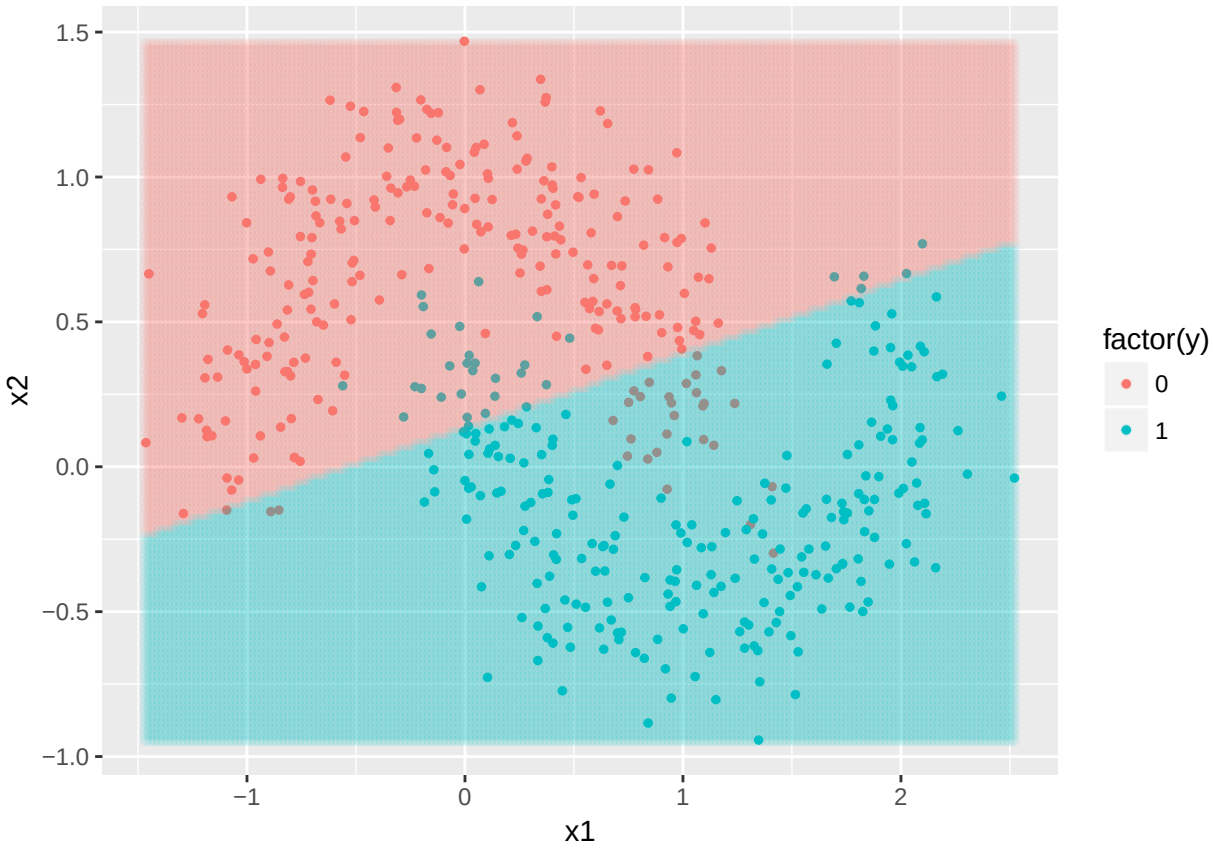
We create a grid in order to test our model. The output of the Logistic regression is a numeric value between 0 and 1, providing the probability of the observation being 1. In other words, $f_\theta(x_1, x_2) = P(Y = 1|x_1, x_2; \theta)$. Therefore, we create the binary variable `y_grid_binary`, that outputs the class `1` when the probability is greater than 0.5.

```
grid = expand.grid(x1 = seq(min(data$x1), max(data$x1), 0.02),
                   x2 = seq(min(data$x2), max(data$x2), 0.02))

y_grid = predict(model, grid, type="response")
y_grid_binary = as.numeric(y_grid>0.5)
grid$y = y_grid_binary
```

We plot the prediction of the model, along with the training set, in order to see how well the logistic regression is doing.

```
p = ggplot(data=data, aes(x1,x2, color=factor(y))) + geom_point(size=1)
p = p + geom_point(data=grid, aes(x1, x2, color=factor(y)), alpha=0.1)
print(p)
```



As you can see, the Logistic Regression is trying to linearly separate the training set, and therefore it doesn't work well in those datasets that are not linearly separable.

Neural Networks

Ingredients

- Definition of the model to do a forward pass: $y = \sigma(W^{[L-1]}(\sigma(W^{[L-2]}(...\sigma(W^{[1]}x + b^{[1]})...) + b^{[L-2]})) + b^{[L-1]})$ where
  - σ is the activation function
  - $Wx + b$ is a linear function that goes from $\mathbb{R}^n$ to $\mathbb{R}^m$.
  - In our example, the output of the model is a value between 0 and 1 that tells the probability of a point being blue or red. The input of the model are the coordinates of the point.
- Definition of a cost function that tells how good is the model in terms of its parameters:

$$J(\theta), \text{where } \theta := \{W^{[1]}, b^{[1]}, \ldots, W^{[L-1]}, b^{[L-1]}\}$$

- Optimisation of the cost function using gradient descent.
  - We need to apply the chain rule (backpropagation) in order to obtain $\partial_\theta J$ for each optimisation step.

Activation function and model definition

We will use the `sigmoid` activation function. Please create a function that returns the output of the sigmoid, $\sigma(z)$ where $\sigma(z) = \frac{1}{1+\exp(-z)}$

```
# activation function
sigmoid <- function(z){
  # TODO
  #output =
  #return(output)
}
```

We will create a model with a two-dimensional input layer, one hidden layer, and a one-dimensional output layer.

$$y = \sigma(W^{[2]}(\sigma(W^{[1]}x + b^{[1]})) + b^{[2]})$$

We define the next function that goes forward through the model. It takes the input `x`, and return the output of the model, as well as the intermediate layers that we will use in backpropagation.

Please fill in the gaps below taking into account that: $z_2 = W^{[1]}x + b^{[1]}$, $a_2 = \sigma(z_2)$, $z_3 = W^{[2]}a_2 + b^{[1]}$, $a_3 = \sigma(z_3)$. For this, take into account that in R matrix multiplication is performed using `%*%`, and the transposed of a matrix `x` is performed using `t(x)`. You will only need to tranpose the input `x` of the model, as in our data the observations are given per row, and we want to put them into columns.

```
# forward model
forward_model <- function(model, x){
  # model is a list with the following
  # x is the input
  W1 = model[['W1']]
  b1 = model[['b1']]
  W2 = model[['W2']]
  b2 = model[['b2']]

  # forward pass
  # hidden layer
  #z2 = #TODO
  #a2 = #TODO

  # output layer
  #z3 = #TODO
  #a3 = #TODO
  output = list(z2 = t(z2), a2=t(a2), z3=t(z2), a3=t(a3))

  return(output)
}
```

Loss function

Let $\mathcal{D} = \{(x^1, y^1), (x^2, y^2), \ldots, (x^N, y^N)\}$ be our training set, where $x^i \in \mathbb{R}^n$. We define the loss function as

$$J(\theta) = -\sum_{i=1}^{N} y^i \log(f_\theta(x^i)) + (1 - y^i) \log(1 - f_\theta(x^i))$$

, which corresponds to the negative log-likelihood of of our dataset assuming it was sampled from a Bernouilli distribution.

```
loss_fn <- function(model, x, y){
  layers_model = forward_model(model, x)
  y_pred = layers_model[['a3']]
  loss = y * log(y_pred) + (1-y) * log(1-y_pred)
```

```
  return(sum(-loss))
}
```

Gradient descent step.

See for example (https://arxiv.org/pdf/1801.05894.pdf) for a general form of the gradient of the cost function for the parameters in each layer of the network. The following function performs a gradient descent step using the whole training set

$$\theta := \theta - \alpha \cdot \partial_\theta J(\theta)$$

```
GD_step <- function(model, x, y, lr=0.001){
  W1 = model[['W1']]
  b1 = model[['b1']]
  W2 = model[['W2']]
  b2 = model[['b2']]

  pred_model = forward_model(model, x)
  z2 = pred_model[['z2']]
  a2 = pred_model[['a2']]
  z3 = pred_model[['z3']]
  a3 = pred_model[['a3']]

  delta3 = a3 - y
  dW2 = t(a2) %*% delta3
  db2 = apply(FUN=sum, X=delta3, MARGIN=2)

  delta2 = sigmoid(z2) * (1-sigmoid(z2)) * (delta3%*%W2)
  dW1 = t(x) %*% delta2
  db1 = apply(FUN=sum, X=delta2, MARGIN=2)

  W2 = W2 - lr * t(dW2)
  b2 = b2 - lr * db2
  W1 = W1 - lr * t(dW1)
  b1 = b1 - lr * db1

  model[['W1']] = W1
  model[['b1']] = b1
  model[['W2']] = W2
  model[['b2']] = b2

  return(model)
}
```

We put everything together for the training:

- Initialise $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}$.
- While Not convergence:
  - Calculate $J(\theta)$
  - Update $W^{[i]} := W^{[i]} - \alpha \cdot \partial_{W^{[i]}} J$
  - Update $b^{[i]} := b^{[i]} - \alpha \cdot \partial_{b^{[i]}} J$

```
train <- function(model, n_epochs, x, y){
  for(epoch in 1:n_epochs){
```

```
    model = GD_step(model, x=x, y=y)
    loss = loss_fn(model, x=x, y=y)

    if(epoch %% 10 == 0){
      log = paste("Epoch: ", epoch, "/", n_epochs,", loss: ", loss, sep="")
      print(log)
    }
  }
  return(model)
}

n_hidden = 30
matrix
W1 = matrix(data=rnorm(2*n_hidden), nrow=n_hidden, ncol=2)
b1 = rnorm(n_hidden)
W2 = matrix(data=rnorm(n_hidden), nrow=1, ncol=n_hidden)
b2 = rnorm(1)
model = list(W1 = W1, b1=b1, W2=W2, b2=b2)

model = train(model=model, n_epochs=10000, x=as.matrix(x=data[,c("x1", "x2")]), y=c(x=data$y))
```

We plot the results with the new model.

```
grid = expand.grid(x1 = seq(min(data$x1), max(data$x1), 0.02),
                   x2 = seq(min(data$x2), max(data$x2), 0.02))
y_grid = forward_model(model, x=as.matrix(grid))[['a3']]
y_grid_binary = y_grid>0.5
grid$y = as.numeric(y_grid_binary)

p = ggplot(data=data, aes(x1,x2, color=factor(y))) + geom_point(size=1)
p = p + geom_point(data=grid, aes(x1, x2, color=factor(y)), alpha=0.1)
print(p)
```