ЗВІТ З ЛАБОРАТОРНОЇ РОБОТИ №2
З ДИСЦИПЛІНИ «СТАТИСТИЧНІ АЛГОРИТМИ НАВЧАННЯ»
НЕЙРОННІ МЕРЕЖІ

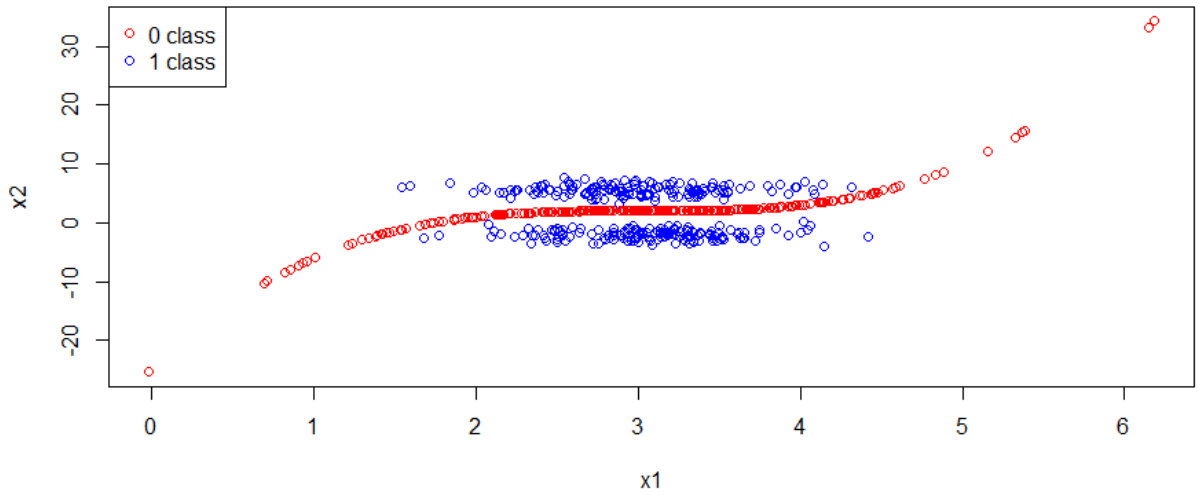
Виконав студент 2к. маг. Ломако Олександр

Варіант 9 => 3

На цей раз маємо датасет, в якому присутні 600 спостережень, пов'язаних змінними x_1, x_2, y, що розбиваються на два класи. Метою роботи є реалізація алгоритму backward-propagation.

Спершу, традиційно, зчитаємо дані, і зобразимо діаграму розсіювання x_1, x_2 із відповідним розмалюванням належності певному класу.

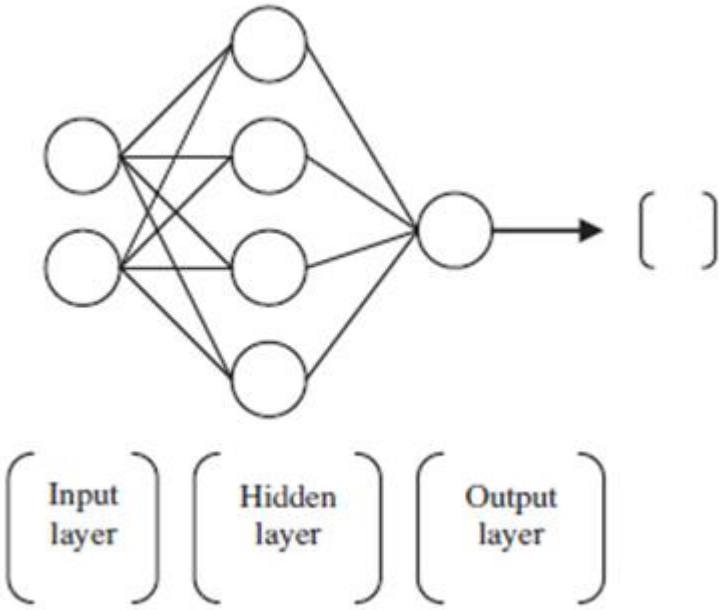
```
> # зчитуємо дані
> data <- read.table('C:\\Users\\Razor\\Desktop\\дистанційне навчання\\статис
тичні алгоритми навчання\\lab2\\data3.csv',
+                    header = T, sep = ',')
>
> col <- c('red', 'blue') # задамо палітру кольорів
> # виведемо діграму розсіювання x1 і x2 із розфарбуванням
> plot(data[,1],data[,2], col = col[as.factor(data$y)], xlab = 'x1', ylab = '
x2')
> legend('topleft', legend = c('0 class', '1 class'), col = col, pch = 1) # в
иведемо легенду на діаграму
```



Тут бачимо, що спостереження першого (нульового за датасетом) класу чітко утворюють собою певну криву, в той час як спостереження другого (відповідно, першого) розташовані по обидва боки (зверху і знизу) від основної маси спостережень першого класу.

Можна було би припустити, що спостереження першого класу являють собою кубічну параболу, або, на мою думку, більш доцільніше, tg, бо змінна x_1 якраз змінюється від 0 до трішки більше 6 ($\approx 2\pi$).

Задамо архітектуру нейронної мережі наступним чином:



- **Input layer** – вхідний шар, що складатиметься з двох нейронів (змінні x_1, x_2);
- **Hidden layer** – прихований шар, що складатиметься з чотирьох нейронів;
- **Output layer** – вихідний шар, що буде складатись з одного нейрону – результату класифікації.

У нас кожен нейрон матиме зовнішнє і внутрішнє значення. Введемо позначення:

$Z_i^{[l](k)}$ *– внутрішнє значення* k*-го нейрона* l*-го шару на* i*-му спостереженні*

$A_i^{[l](k)} = a_l\left(Z_i^{[l](k)}\right)$ *– зовнішнє значення* k*-го нейрона* l*-го шару на* i*-му спостереженні, де* a_l *– активаційна функція.*

$W^{[l]}$ *– матриця вагів для* l*-го шару*

$b^{[l]}$ *– вектор зміщень (вільний член)*

При цьому всьому маємо слідкувати за розмірностями, а саме зазначимо, що $Z^{[l]} \in R^{N \times m_l}, A^{[l-1]} \in R^{N \times m_{l-1}}, W^{[l]} \in R^{m_{l-1} \times m_l}, b^{[l]} \in R^{1, m_l}$, де m_l – кількість нейронів у l-му шарі, N – кількість спостережень.

В якості активаційних функцій будемо використовувати на першому шарі гіперболічний тангенс, а на другому шарі – сигмоїду, оскільки маємо справу з задачею бінарної класифікації.

Спершу застосуємо алгоритм **forward-propagation.**

Для цього маємо спершу згенерувати $W^{[1]}$ – як випадкові величини в районі 0 (можна, наприклад, взяти нормальний розподіл із середнім 0 та якоюсь достатньо малою дисперсією) та $b^{[1]}$ – вектор з нулів. Не забуватимемо при цьому про розмірності!

```
> # задамо параметри кількості нейронів на кожному шарі, а також к-сть спосте
режень
> N <- length(data[,1])
> neurons <- c(2,4,1)
> # задамо функцію, що генеруватиме W^1, W^2, b^1 та b^2
> generator <- function(n){
+   w1 <- rnorm(n[1]*n[2], mean = 0, sd = 0.0001)
+   W1 <- matrix(data = w1, nrow = n[1], ncol = n[2])
+   w2 <- rnorm(n[2]*n[3], mean = 0, sd = 0.0001)
+   W2 <- matrix(data = w2, nrow = n[2], ncol = n[3])
+   b1 <- matrix(data = rep(0, n[2]), nrow = n[2])
+   b2 <- matrix(data = rep(0, n[3]), nrow = n[3])
+   res <- list('W1' = W1, 'W2' = W2, 'b1' = b1, 'b2' = b2)
+   return(res)
+ }
> generator(neurons)
$W1
              [,1]          [,2]         [,3]          [,4]
[1,] -1.719036e-04 -6.149072e-06 5.456507e-05 -9.965095e-05
[2,] -2.418775e-05 -2.943469e-05 7.694662e-05 -8.820195e-05

$W2
              [,1]
[1,]  1.868902e-04
[2,] -2.230611e-05
[3,] -9.626503e-05
[4,] -3.197345e-05

$b1
     [,1]
[1,]    0
[2,]    0
[3,]    0
[4,]    0

$b2
     [,1]
[1,]    0
```

Ініціалізувавши функцію один раз, переконаємося в правильності розмірностей. Тут мають виконуватись наступні належності:

$$W^{[1]} \in R^{2\times4}, W^{[2]} \in R^{4\times1}, b^{[1]} \in R^{1\times4}, b^{[2]} \in R^{1\times1}$$

Як бачимо, це має місце бути.

Тепер реалізуємо функцію, що обчислює значення $A^{[l]}, Z^{[l]}$.

$A^{[1]} = a_1\left(Z^{[1]}\right) = \tanh(XW^{[1]} + B^{[1]})$, де тут і в подальшому $B^{[l]}$ – матриця з N однакових рядків $b^{[l]}$. $A^{[2]} = a_2\left(Z^{[2]}\right) = sigmoid(A^{[1]}W^{[2]} + B^{[2]})$, де функція сигмоїда задана наступним чином: $\sigma(u) = \frac{1}{1+e^{-u}}$. Причому $A^{[2]}$ – і буде прогнозом даної моделі.

```
> # реалізуємо прогноз за допомогою forward propagation
> library(e1071)
> frw <- function(d, n, N, param){
+   c <- param
```

```
+   B1 <- matrix(data = rep(c$b1, N), nrow = N)
+   Z1 <- as.matrix(d[,1:2]) %*% c$W1 + B1
+   A1 <- tanh(Z1)
+   B2 <- matrix(data = rep(c$b2, N), nrow = N)
+   Z2 <- as.matrix(A1) %*% c$W2 + B2
+   A2 <- sigmoid(Z2)
+   res <- list('A2' = A2, "Z2" = Z2, "A1" = A1, 'Z1' = Z1)
+   return(res)
+ }
```

Визначимо цільову функцію наступним чином:

$$\mathcal{L} = -\frac{1}{N}\sum_{j=1}^{N}\left(Y_j \cdot \ln\left(\widehat{Y}_j\right) + \left(1 - Y_j\right) \cdot \ln\left(1 - \widehat{Y}_j\right)\right) \text{ , де } \widehat{Y}_j = A^{[2]}.$$

Це так звана функція кросс-ентропії у випадку двох класів (*binary cross-entropy loss function*).

Її обчислення також реалізуємо окремою функцією в середовищі R.

```
> # визначимо цільову функцію
> cross_entopy_loss <- function(d, n, N, param){
+   c <- frw(d, n, N, param)
+   c <- c$A2
+   res <- 0
+   for(j in 1:N){
+     r <- d[j,3]*log(c[j]) + (1-d[j,3])*log(1-c[j])
+     res <- res + r
+   }
+   return(-res/N)
+ }> cross_entopy_loss(data, neurons, N)
[1] 0.6931637
```

Наприклад, для випадкової ітерації функція втрат даватиме порядку 0.6931 (щоправда, такі ж самі значення з точністю до десятитисячних виходитимуть і при наступних ітераціях).

Далі застосуємо алгоритм **backward-propagation.**

Для мінімізації цільової функції застосуємо метод градієнтного спуску, який коригуватиме параметри нашої моделі.

$$\delta^{[2]} = \frac{\partial \mathcal{L}}{\partial Z^{[2]}} = \widehat{Y} - Y$$

$$\frac{\partial \mathcal{L}}{\partial W^{[2]}} = \frac{1}{N}\left(A^{[1]}\right)^T \delta^{[2]}$$

$$\frac{\partial \mathcal{L}}{\partial b^{[2]}} = \frac{1}{N}\sum_{i=1}^{N}\delta_i^{[2]}$$

$$\delta^{[1]} = \frac{\partial \mathcal{L}}{\partial Z^{[1]}} = \left(\delta^{[2]}\left(W^{[2]}\right)^T\right) * \left(A^{[1]}\right)' =$$

$$= \left|A^{[1]} = a_1\left(XW^{[1]} + B^{[1]}\right), a_1 = \tanh(x) => a_1' = (1 - a_1^2)\right| =$$

$$= \left(\delta^{[2]}\left(W^{[2]}\right)^T\right) * \left(1 - \left(A^{[1]}\right)^2\right)$$

$$\frac{\partial \mathcal{L}}{\partial W^{[1]}} = \frac{1}{N}(X)^T \delta^{[1]}$$

$$\frac{\partial \mathcal{L}}{\partial b^{[1]}} = \frac{1}{N}\sum_{i=1}^{N} \delta_i^{[1]}$$

```
> # реалізуємо алгоритм backward-propagation
> bcw <- function(d, n, N, frw_res, gen_res){
+   c <- frw_res
+   g <- gen_res
+   d2 <- c$A2 - as.matrix(d[,3])
+   w2 <- (1/N)*t(d2) %*% c$A1
+   b2 <- sum(d2)/N
+   d1 <- d2 %*% t(g$W2) * (1-(c$A1)^2)
+   w1 <- (1/N) * t(d1) %*% as.matrix(d[,1:2])
+   b1 <- matrix(sum(d1)/N, nrow = n[2])
+   res <- list('dW2' = w2, 'dW1' = w1, 'db2' = b2, 'db1' = b1)
+   return(res)
+ }
```

Далі реалізуємо функцію, що покращуватиме параметри нашої моделі. Її параметром виступамиме α, і в залежності від нього параметри коригуватимуться наступним чином:

$$W^{[l]} = W^{[l]} - \alpha * dW^{[l]}$$

$$b^{[l]} = b^{[l]} - \alpha * db^{[l]}$$

```
> # покращимо параметри моделі
> impr_par <- function(alpha, param, bcw_res){
+   a <- param
+   b <- bcw_res
+   w1 <- t(a$W1) - alpha * b$dW1
+   w2 <- t(a$W2) - alpha * b$dW2
+   b1 <- a$b1 - alpha * b$db1
+   b2 <- a$b2 - alpha * b$db2
+   res <- list('W2' = w2, "W1" = w1, 'b2' = b2, 'b1' = b1)
+   return(res)
+ }
```

І нарешті можемо приступити до безпосереднього тренування моделі. Для цього також напишемо окремо функцію, всередині якої викликатимуться функції, згадані вище.

```
> # реалізуємо функцію що тренуватиме модель і підбиратиме оптимальний параме
тр
> final <- function(d, n, n_iter, alpha){
+   p <- generator(n)
+   loss <- c()
+   for(j in 1:n_iter){
+     frw_res <- frw(d, n, N = 600, param = p)
+     loss <- c(loss, cross_entopy_loss(d, n, N = length(d[,1]), p))
+     bcw_res <- bcw(d, n, N = length(d[,1]), frw_res, p)
+     p1 <- impr_par(alpha, p, bcw_res)
+     p1$W1 <- t(p1$W1)
+     p1$W2 <- t(p1$W2)
+     p <- p1
+     if(j %% 500 == 0)cat('Iteration',j,'|CE_loss:',loss[j],'\n')
+   }
+   res <- list('new_parameters' = p1, 'loss_values' = loss)
+   return(res)
```

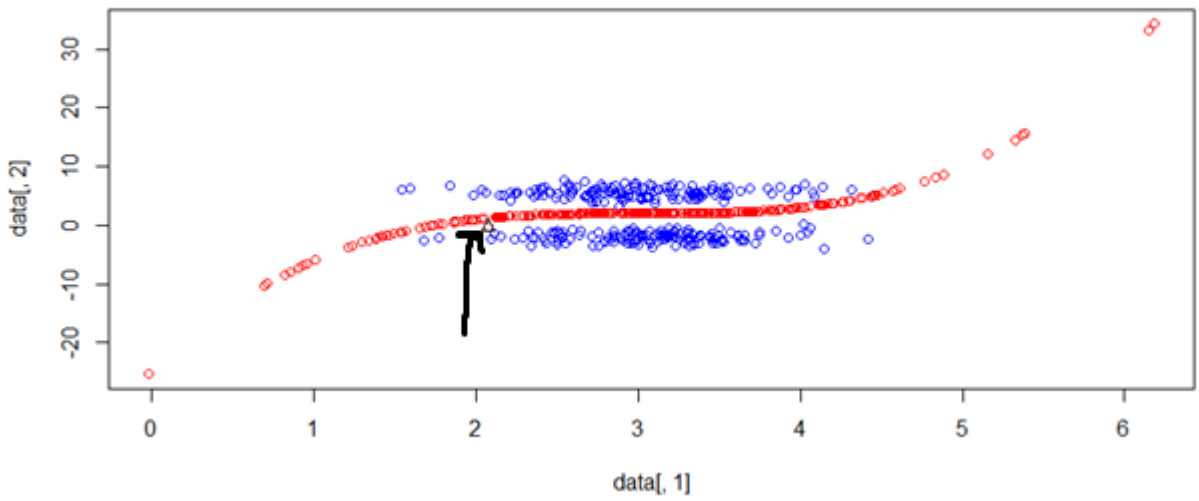
```
+ }
```

Було прорoблено декілька експериментів, і емпіричним шляхом встановлено, що достатньо непоганим є параметр $\alpha = 0.1$. Проженемо даний алгоритм 30 000 ітерацій, і поглянемо на результат.

```
> res2 <- final(data, neurons, 30000, 0.1)
Iteration 500 |CE_loss: 0.5482331
Iteration 1000 |CE_loss: 0.1880941
Iteration 1500 |CE_loss: 0.1619818
Iteration 2000 |CE_loss: 0.152697
Iteration 2500 |CE_loss: 0.146656
Iteration 3000 |CE_loss: 0.1421324
Iteration 3500 |CE_loss: 0.1363209
Iteration 4000 |CE_loss: 0.1222062
Iteration 4500 |CE_loss: 0.1094644
Iteration 5000 |CE_loss: 0.1019598
Iteration 5500 |CE_loss: 0.09743603
Iteration 6000 |CE_loss: 0.09444929
Iteration 6500 |CE_loss: 0.09231884
Iteration 7000 |CE_loss: 0.09071047
Iteration 7500 |CE_loss: 0.08944515
Iteration 8000 |CE_loss: 0.08841836
Iteration 8500 |CE_loss: 0.08756458
Iteration 9000 |CE_loss: 0.08684038
Iteration 9500 |CE_loss: 0.08621568
Iteration 10000 |CE_loss: 0.08566889
Iteration 10500 |CE_loss: 0.08518402
Iteration 11000 |CE_loss: 0.0847489
Iteration 11500 |CE_loss: 0.08435395
Iteration 12000 |CE_loss: 0.08399146
Iteration 12500 |CE_loss: 0.0836549
Iteration 13000 |CE_loss: 0.08333851
Iteration 13500 |CE_loss: 0.08303687
Iteration 14000 |CE_loss: 0.08274438
Iteration 14500 |CE_loss: 0.08245467
Iteration 15000 |CE_loss: 0.08215969
Iteration 15500 |CE_loss: 0.08184811
Iteration 16000 |CE_loss: 0.08150307
Iteration 16500 |CE_loss: 0.08109963
Iteration 17000 |CE_loss: 0.08060872
Iteration 17500 |CE_loss: 0.08002479
Iteration 18000 |CE_loss: 0.07939495
Iteration 18500 |CE_loss: 0.0787565
Iteration 19000 |CE_loss: 0.078065
Iteration 19500 |CE_loss: 0.07691711
Iteration 20000 |CE_loss: 0.07276441
Iteration 20500 |CE_loss: 0.06656038
Iteration 21000 |CE_loss: 0.03876472
Iteration 21500 |CE_loss: 0.03420633
Iteration 22000 |CE_loss: 0.03110896
Iteration 22500 |CE_loss: 0.02879941
Iteration 23000 |CE_loss: 0.02700613
Iteration 23500 |CE_loss: 0.02557219
Iteration 24000 |CE_loss: 0.02439844
Iteration 24500 |CE_loss: 0.02341914
Iteration 25000 |CE_loss: 0.02258903
Iteration 25500 |CE_loss: 0.02187593
Iteration 26000 |CE_loss: 0.02125634
Iteration 26500 |CE_loss: 0.02071265
Iteration 27000 |CE_loss: 0.02023144
Iteration 27500 |CE_loss: 0.01980227
Iteration 28000 |CE_loss: 0.01941687
Iteration 28500 |CE_loss: 0.01906867
Iteration 29000 |CE_loss: 0.01875231
Iteration 29500 |CE_loss: 0.01846344
Iteration 30000 |CE_loss: 0.01819846
> res_frw2 <- frw(data, neurons, N, param = res2$new_parameters)
> final_pred2 <- res_frw2$A2
> final_pred2 <- round(final_pred2)
```

```
> table(final_pred2, data[,3])

final_pred2   0   1
          0 300   1
          1   0 299
> plot(data[,1], data[,2], col = col[final_pred+1])
> plot(data[,1], data[,2], col = col[final_pred2+1])
> which(data[,3] != final_pred2)
[1] 288
> points(data[288,1], data[288,2], pch = 2)
```




```
> print(sum(final_pred2 == data[,3])/N)
[1] 0.9983333
```

Отже, маємо майже стовідсоткову точність роботи класифікатора: лише одне спостереження з 600 було класифіковано неправильно. Можливо, це можна пов’язати з тим, що наші класи не розділені лінійно – їх розділяє нелінійна функція (як я вже припускав, тангенс чи кубічна парабола).