

Neuronas artificiales

MANADINE – Análisis de Datos 1

18 de febrero de 2026

Introducción

- ▶ inspirado en las conexiones (sinapsis) entre neuronas cerebrales
- ▶ permiten clasificación o regresión
- ▶ son modelos de regresión no lineal con muchos parámetros (caja negra: el ajuste no es constructivo)
- ▶ permiten ajustar información no estructurada (textos, fotos...) pero a veces alucinan
- ▶ tipos
 - ▶ perceptrón
 - ▶ base radial
 - ▶ mapas autoorganizados
 - ▶ ...

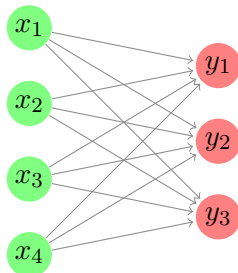
Índice

- ▶ Clasificación
 - ▶ Perceptrón simple
 - ▶ Perceptrón multicapa (1 oculta)
- ▶ Regresión
- ▶ Decaimiento de pesos
- ▶ Recomendaciones

Perceptrón simple

Capa de
entrada

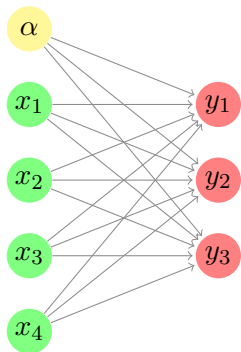
Capa de
salida



Perceptrón simple

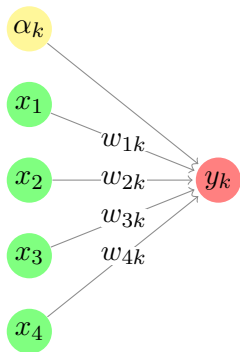
Capa de
entrada

Capa de
salida



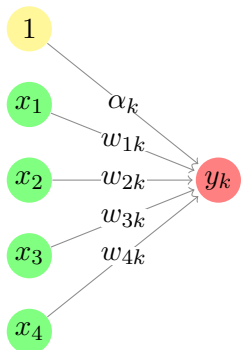
Perceptrón simple

Capa de entrada Capa de salida



Perceptrón simple

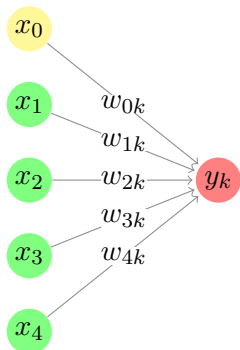
Capa de entrada Capa de salida



Perceptrón simple

Capa de
entrada

Capa de
salida



Perceptrón simple

- ▶ i = entrada (*input*)
- ▶ o = salida (*output*)
- ▶ ϕ = función de activación
- ▶ α = constante, sesgo (*bias*)
- ▶ w = peso (*weight*), coeficiente de sinapsis

$$y_k = \phi_o \left(\alpha_k + \sum_{i=1}^I w_{ik} x_i \right) = \phi_o \left(\sum_{i=0}^I w_{ik} x_i \right)$$

Perceptrón simple

- ▶ funciones de activación ϕ habituales

- ▶ lineal

$$\phi_{\sigma}(x) = x$$

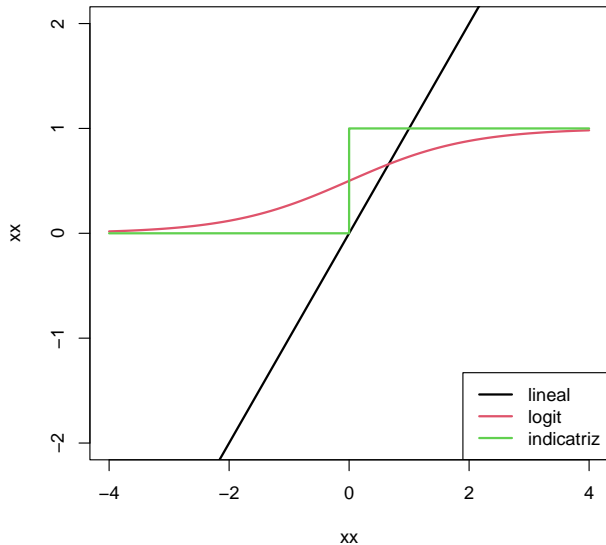
- ▶ logística

$$\phi_{\sigma}(x) = \ell(x) = \frac{\exp(x)}{1 + \exp(x)}$$

- ▶ indicatriz, umbral, característica, Heaviside

$$\phi_{\sigma}(x) = \mathbb{1}_{[0, \infty)}(x) = \begin{cases} 0 & \text{si } x < 0 \\ 1 & \text{si } x \geq 0 \end{cases}$$

Perceptrón simple



Perceptrón simple: ajuste

- ▶ función de activación:

- ▶ para respuesta dicótoma, logística: $\phi_o(x) = \frac{\exp(x)}{1+\exp(x)}$

- ▶ para tres o más categorías, lineal: $\phi_o(x) = x$

- ▶ criterios de ajuste

$p =$ patrón $t =$ respuesta $\in \{0; 1\}$ $y =$ predicción

- ▶ para respuesta dicótoma, entropía ($0 \leq y \leq 1$)

$$E = \sum_p \sum_k \left[t_k^{(p)} \ln \frac{t_k^{(p)}}{y_k^{(p)}} + (1 - t_k^{(p)}) \ln \frac{1 - t_k^{(p)}}{1 - y_k^{(p)}} \right]$$

- ▶ para respuesta múltiple, *softmax* ($y \in \mathbb{R}$)

$$E = \sum_p \sum_k -t_k^{(p)} \log \widehat{\Pr}[p \in k] \quad \widehat{\Pr}[p \in k] = \frac{\exp(y_k^{(p)})}{\sum_{c=1}^K \exp(y_c^{(p)})}$$

Perceptrón simple

```
> ## para ahorrar espacio en esta presentación:
> options (width = 58)
> names(iris)[1:4] <- c("Lsep", "Asep", "Lpet", "Apet")
> library (nnet) # biblioteca distribuida con R básico
> red <- nnet (Species ~ ., iris, size = 0, skip = TRUE)
# weights: 15
initial value 285.325363
iter 10 value 8.096373
iter 20 value 5.960549
iter 30 value 5.952579
iter 40 value 5.949318
iter 50 value 5.949290
final value 5.949277
converged
```

Perceptrón simple

```
> red
```

```
a 4-0-3 network with 15 weights
```

```
inputs: Lsep Asep Lpet Apet
```

```
output(s): Species
```

```
options were - skip-layer connections softmax modelling
```

Perceptrón simple

```
> summary (red)
```

```
a 4-0-3 network with 15 weights
```

```
options were - skip-layer connections  softmax modelling
```

```
b->o1 i1->o1 i2->o1 i3->o1 i4->o1
```

```
-2.82  1.96  17.93 -14.07 -11.96
```

```
b->o2 i1->o2 i2->o2 i3->o2 i4->o2
```

```
22.60  0.21  -5.82  2.29  -2.75
```

```
b->o3 i1->o3 i2->o3 i3->o3 i4->o3
```

```
-20.08  -2.26 -12.51  11.73  15.55
```

Perceptrón simple

```
> names (red)
 [1] "n"                "nunits"          "nconn"
 [4] "conn"            "nsunits"         "decay"
 [7] "entropy"         "softmax"         "censored"
[10] "value"           "wts"             "convergence"
[13] "fitted.values"  "residuals"      "lev"
[16] "call"            "terms"          "coefnames"
[19] "xlevels"
```

Perceptrón simple

```
> red $ wts
[1] -2.8239735  1.9633937  17.9281992 -14.0713310
[5] -11.9612859 22.6031112  0.2066642  -5.8225870
[9]  2.2915637  -2.7476730 -20.0809056  -2.2580146
[13] -12.5066021 11.7273984  15.5503582

> head (red $ fitted.values)
  setosa  versicolor  virginica
1      1 6.295937e-19 9.314251e-46
2      1 1.285481e-13 8.803349e-39
3      1 3.076484e-16 3.526606e-42
4      1 1.040148e-13 1.964745e-38
5      1 6.980244e-20 6.771966e-47
6      1 2.379159e-20 7.638111e-46
```

Perceptrón simple

```
> head (red $ fitted.values)
  setosa  versicolor  virginica
1      1 6.295937e-19 9.314251e-46
2      1 1.285481e-13 8.803349e-39
3      1 3.076484e-16 3.526606e-42
4      1 1.040148e-13 1.964745e-38
5      1 6.980244e-20 6.771966e-47
6      1 2.379159e-20 7.638111e-46

> summary (apply (red $ fitted.values, 1, sum))
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
     1      1      1      1      1      1
```

Perceptrón simple

```
> red $ value
[1] 5.949277
> indices.fila    <- 1 : nrow (iris)
> indices.columna <- match (iris$Species,
+                           levels(iris$Species))
> indices <- cbind (indices.fila, indices.columna)
> - sum (log (red$fitted.values [indices]))
[1] 5.949277
```

Perceptrón simple

```
> aggregate (iris[,1:4], list(iris$Species), median)
      Group.1 Lsep Asep Lpet Apet
1      setosa  5.0  3.4 1.50  0.2
2 versicolor  5.9  2.8 4.35  1.3
3  virginica  6.5  3.0 5.55  2.0
> flor <- data.frame(Lsep=6,Asep=2.9,Lpet=5,Apet=1.7)
> predict (red, flor)
      setosa versicolor virginica
1 2.509364e-20  0.1930841 0.8069159
> predict (red, flor, type="class")
[1] "virginica"
```

Perceptrón simple

```
> predict (red, flor)
      setosa versicolor virginica
1 2.509364e-20  0.1930841 0.8069159
> summary (red)
a 4-0-3 network with 15 weights
options were - skip-layer connections  softmax modelling
b->o1 i1->o1 i2->o1 i3->o1 i4->o1
-2.82  1.96  17.93 -14.07 -11.96
b->o2 i1->o2 i2->o2 i3->o2 i4->o2
22.60  0.21  -5.82  2.29  -2.75
b->o3 i1->o3 i2->o3 i3->o3 i4->o3
-20.08 -2.26 -12.51  11.73  15.55
```

Perceptrón simple

```
> predict (red, flor)
      setosa versicolor virginica
1 2.509364e-20 0.1930841 0.8069159
> flor1 <- c (1, as.numeric (flor))
> e1 <- exp (as.numeric (flor1 %*% red$wts[1:5]))
> e2 <- exp (as.numeric (flor1 %*% red$wts[6:10]))
> e3 <- exp (as.numeric (flor1 %*% red$wts[11:15]))
> c(e1,e2,e3) / (e1+e2+e3)
[1] 2.509364e-20 1.930841e-01 8.069159e-01
```

Respuesta dicótoma

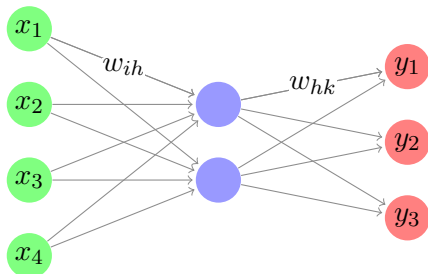
```
> red2 <- nnet (factor(am)~mpg, mtcars,
+             size=0, skip=TRUE, trace=FALSE)
> predict (red2, data.frame (mpg = 20))
      [,1]
1 0.3862832
> 1 / (1 + 1/exp (red2$wts[1] + red2$wts[2] * 20)) #logit
[1] 0.3862832
> red2 $ value
[1] 14.83758
> p <- red2 $ fitted.values           #una sola columna
> - sum (p * log(p) + (1-p) * log(1-p)) #entropía
[1] 14.83758
```

Respuesta dicótoma

```
> p <- red2 $ fitted.values           #una sola columna
> - sum (p * log(p) + (1-p) * log(1-p)) #entropía
[1] 14.83758
> ## equivale a
> t <- +(mtcars$am == mtcars$am[1])
> n0 <- function (x) ifelse (is.na(x), 0, x)
> sum (n0(t*log(t/p)) + n0((1-t)*log((1-t)/(1-p))))
[1] 14.83758
```

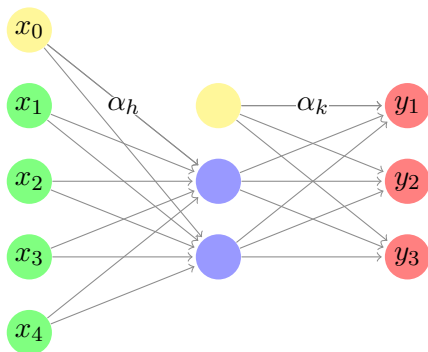
Perceptrón multicapa (1 oculta)

Entrada Capa oculta Salida



Perceptrón multicapa (1 oculta)

Entrada Oculta Salida



Perceptrón multicapa (1 oculta)

- ▶ h = índice de neuronas en capa oculta (*hidden*)

$$y_k = \phi_o \left(\alpha_k + \sum_{h=1}^H w_{hk} \phi_h \left(\alpha_h + \sum_{i=1}^I w_{ih} x_i \right) \right)$$

$$y_k = \phi_o \left(\sum_{h=0}^H w_{hk} \phi_h \left(\sum_{i=0}^I w_{ih} x_i \right) \right)$$

- ▶ ϕ casi siempre logística en la oculta

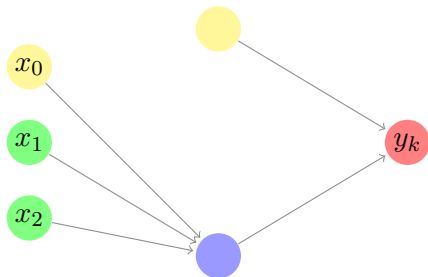
$$\phi_h(x) = \ell(x) = \frac{\exp(x)}{1 + \exp(x)}$$

Perceptrón multicapa (1 oculta, skip=FALSE)

Entrada

Oculto

Salida

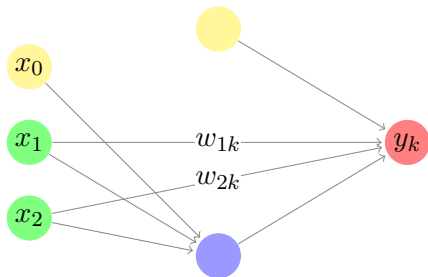


Perceptrón multicapa (1 oculta, skip=TRUE)

Entrada

Oculta

Salida



Perceptrón multicapa (1 oculta)

- ▶ skip=FALSE

$$y_k = \phi_o \left(\sum_h w_{hk} \phi_h \left(\sum_i w_{ih} x_i \right) \right)$$

- ▶ skip=TRUE

$$y_k = \phi_o \left(\sum_h w_{hk} \phi_h \left(\sum_i w_{ih} x_i \right) + \sum_i w_{ik} x_i \right)$$

- ▶ las conexiones *skip* pueden facilitar la interpretación de la red neuronal

Perceptrón multicapa (1 oculta)

```
> red0 <- nnet (Species ~ ., iris, size = 2,  
+             skip = FALSE, trace = FALSE)
```

```
> red0
```

a 4-2-3 network with 19 weights

inputs: Lsep Asep Lpet Apet

output(s): Species

options were - softmax modelling

```
> red1 <- nnet (Species ~ ., iris, size = 2,  
+             skip = TRUE, trace = FALSE)
```

```
> red1
```

a 4-2-3 network with 31 weights

inputs: Lsep Asep Lpet Apet

output(s): Species

options were - skip-layer connections softmax modelling

Perceptrón multicapa (1 oculta)

```
> summary (red0)
```

```
a 4-2-3 network with 19 weights
```

```
options were - softmax modelling
```

b->h1	i1->h1	i2->h1	i3->h1	i4->h1
1.03	0.13	0.34	-0.50	-0.92
b->h2	i1->h2	i2->h2	i3->h2	i4->h2
-1.62	-7.56	-3.33	-5.56	-1.92
b->o1	h1->o1	h2->o1		
-48.26	104.01	-1.35		
b->o2	h1->o2	h2->o2		
11.49	1.00	0.28		
b->o3	h1->o3	h2->o3		
37.11	-105.44	0.74		

Perceptrón multicapa (1 oculta)

```
> summary (red1)
a 4-2-3 network with 31 weights
options were - skip-layer connections  softmax modelling
b->h1 i1->h1 i2->h1 i3->h1 i4->h1
-4.93 -29.72 -17.20 -14.72 -4.16
b->h2 i1->h2 i2->h2 i3->h2 i4->h2
 1.31  0.32  4.72 -8.49 -4.16
b->o1 h1->o1 h2->o1 i1->o1 i2->o1 i3->o1 i4->o1
 3.02 -5.74  8.73  3.14  4.90 -8.63 -8.23
b->o2 h1->o2 h2->o2 i1->o2 i2->o2 i3->o2 i4->o2
20.05 -4.76 -21.75 -0.42  0.69 -1.00 -4.73
b->o3 h1->o3 h2->o3 i1->o3 i2->o3 i3->o3 i4->o3
-22.57  9.81 13.76 -2.89 -5.99  8.43 13.55
```

Regresión

- ▶ función de salida lineal: $\phi_o(x) = x$
- ▶ teorema
 - ▶ sea f cualquier función continua sobre un compacto
 - ▶ se puede aproximar f uniformemente
 - ▶ basta incrementar el número de neuronas en la capa oculta
- ▶ la aproximación es “no constructiva”
- ▶ criterios de ajuste (p =patrón, t =objetivo, y =predicción)
 - ▶ mínimos cuadrados: $E = \sum_p \|t^{(p)} - y^{(p)}\|^2$

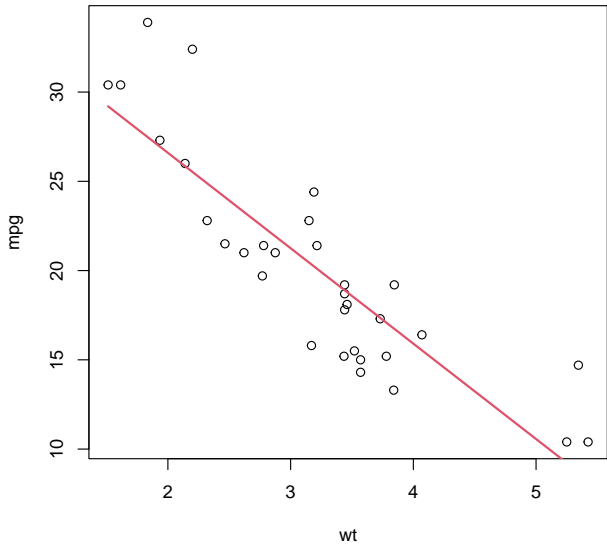
Regresión

```
> reg <- lm (mpg ~ wt, mtcars)
> print (summary (reg))
[...]
```

Coefficients:

	Estimate	Std. Error	t value	Pr(> t)
(Intercept)	37.2851	1.8776	19.858	< 2e-16
wt	-5.3445	0.5591	-9.559	1.29e-10

Residual standard error: 3.046 on 30 degrees of freedom
Multiple R-squared: 0.7528, Adjusted R-squared: 0.7446
F-statistic: 91.38 on 1 and 30 DF, p-value: 1.294e-10



Regresión

```
> set.seed(1) # para reproducir ejemplo malo
> red <- nnet (mpg ~ wt, mtcars, size = 2, linout = TRUE)
# weights:  7
initial  value 13690.144505
iter  10 value 1126.051149
final  value 1126.047233
converged
> summary (red)
a 1-2-1 network with 7 weights
options were - linear output units
b->h1 i1->h1
  3.94   7.04
b->h2 i1->h2
  5.78   9.94
b->o  h1->o h2->o
 7.57  4.78  7.75
```

Regresión

```
> red $ value
```

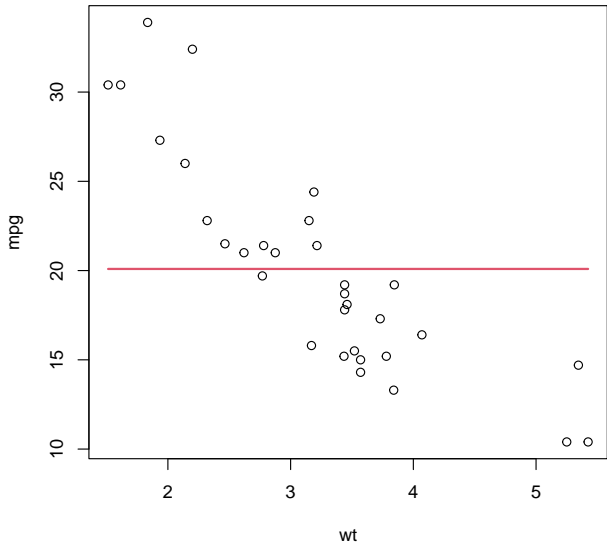
```
[1] 1126.047
```

```
> sum (red $ residuals ^ 2)
```

```
[1] 1126.047
```

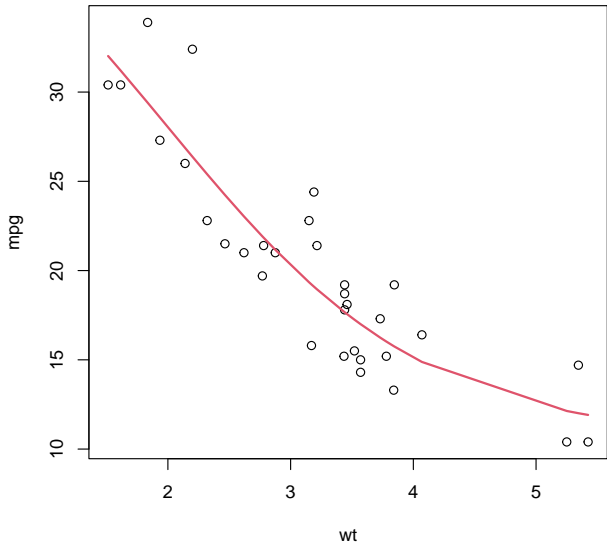
```
> sum (reg $ residuals ^ 2)
```

```
[1] 278.3219
```



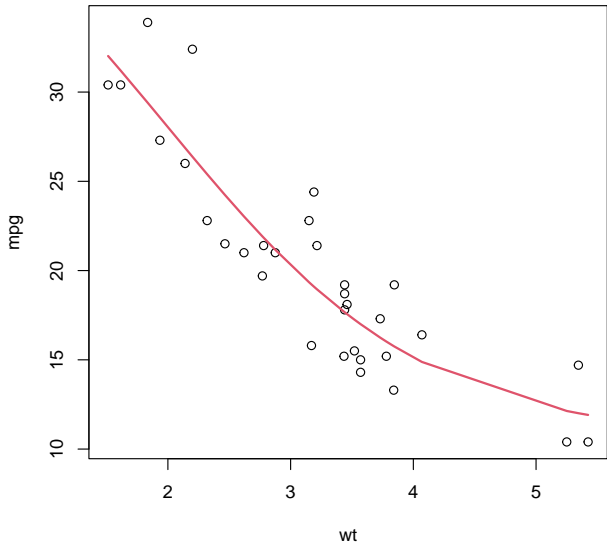
Regresión

```
> set.seed(2) # mucho mejor cambiando la semilla
> red <- nnet (mpg ~ wt, mtcars, size = 2, linout = TRUE)
# weights:  7
initial  value 12943.660208
iter   10 value  962.385509
iter   20 value  251.143680
iter   30 value  202.430605
iter   40 value  202.330422
final   value  202.291329
converged
```



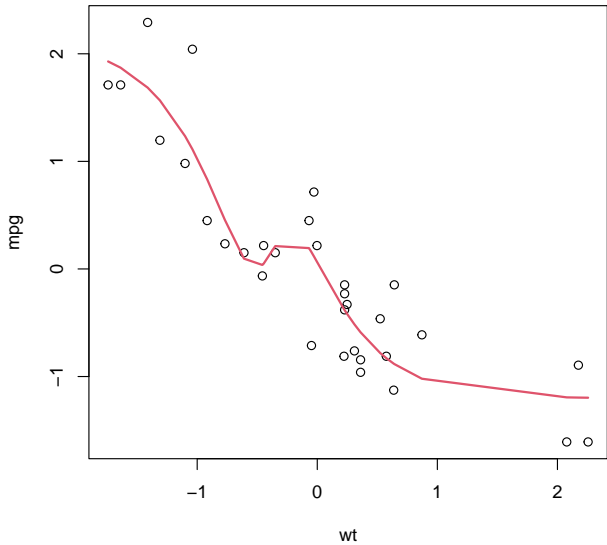
Regresión

```
> set.seed(1) # caso malo; pesos com mayor rango inicial
> red <- nnet (mpg ~ wt, mtcars, size = 2, linout = TRUE,
+             rang = 5)
# weights:  7
initial  value 12211.292513
iter   10 value  744.853983
iter   20 value  205.109500
iter   30 value  202.359319
final   value  202.291137
converged
```



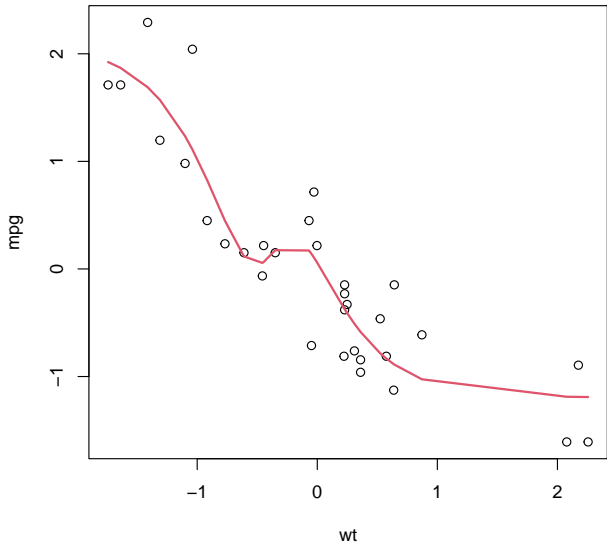
Regresión

```
> set.seed(1) # caso malo; tipificando; sobreajuste
> red <- nnet (mpg ~ wt, scale(mtcars), size = 2,
+             linout = TRUE)
# weights:  7
initial  value 34.908961
iter   10 value 6.037538
iter   20 value 5.526884
iter   30 value 5.265161
iter   40 value 4.800987
iter   50 value 4.798314
iter   60 value 4.792750
iter   70 value 4.792613
iter   80 value 4.792603
final   value 4.792594
converged
```



Regresión

```
> set.seed(1) # decay para evitar sobreajuste
> red <- nnet (mpg ~ wt, scale(mtcars), size = 2,
+             linout = TRUE, decay=.001)
# weights:  7
initial  value 34.910311
iter   10 value 6.087401
iter   20 value 5.574842
iter   30 value 5.178789
iter   40 value 4.927741
iter   50 value 4.924397
iter   60 value 4.924349
iter   60 value 4.924349
iter   60 value 4.924349
final   value 4.924349
converged
```

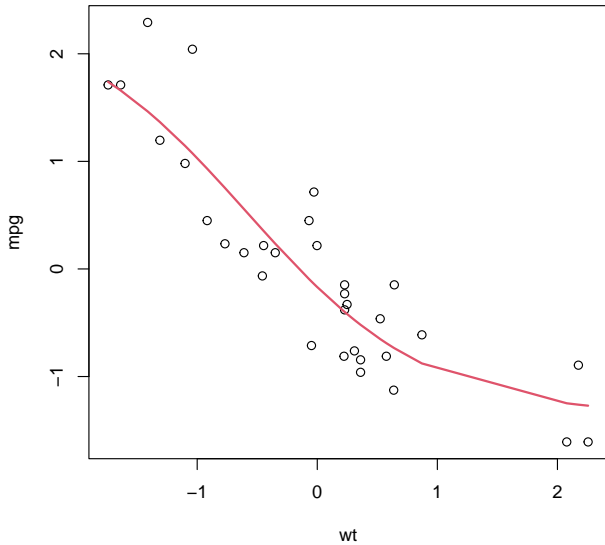


Regresión

```
> set.seed(1) # decay para evitar sobreajuste
> red <- nnet (mpg ~ wt, scale(mtcars), size = 2,
+             linout = TRUE, decay=.01)
# weights:  7
initial  value 34.922460
iter   10 value 6.544019
iter   20 value 5.954441
iter   30 value 5.895883
iter   40 value 5.893031
iter   50 value 5.791990
iter   60 value 5.779464
iter   60 value 5.779464
iter   60 value 5.779464
final   value 5.779464
converged
```

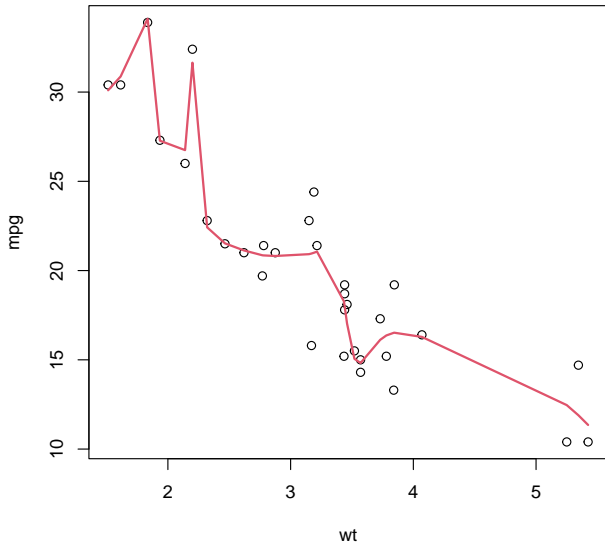

Regresión

```
> set.seed(1) # decay para evitar sobreajuste
> red <- nnet (mpg ~ wt, scale(mtcars), size = 2,
+             linout = TRUE, decay=.1)
# weights:  7
initial  value 35.043951
iter   10 value  8.545438
iter   20 value  7.131476
iter   30 value  7.103682
final   value  7.103680
converged
```



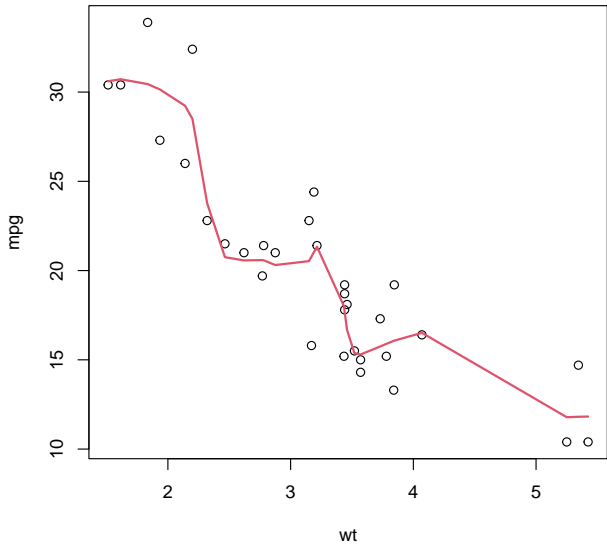
Regresión

```
> set.seed(1) # muchas neuronas ocultas; sobreajuste
> red <- nnet (mpg~wt, mtcars, size = 100, linout = TRUE)
# weights: 301
initial value 21248.821010
iter 10 value 229.802676
iter 20 value 203.348311
iter 30 value 200.484030
iter 40 value 191.191581
iter 50 value 169.455915
iter 60 value 144.765317
iter 70 value 134.589994
iter 80 value 116.245707
iter 90 value 101.921759
iter 100 value 91.396961
final value 91.396961
stopped after 100 iterations
```



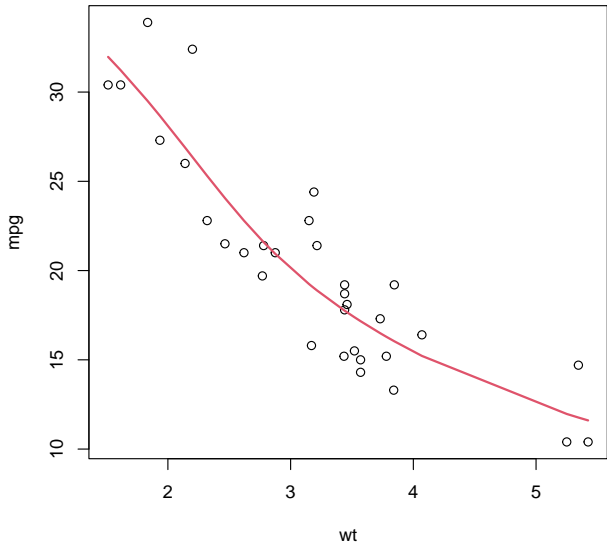
Regresión

```
> set.seed(1) # rang no evita sobreajuste
> red <- nnet (mpg~wt, mtcars, size = 100, linout = TRUE,
+             rang=5)
# weights: 301
initial value 113453.230339
iter 10 value 268.817455
iter 20 value 204.853381
iter 30 value 189.632583
iter 40 value 180.592357
iter 50 value 155.232946
iter 60 value 146.266433
iter 70 value 143.899068
iter 80 value 141.962872
iter 90 value 140.209864
iter 100 value 137.646885
final value 137.646885
stopped after 100 iterations
```



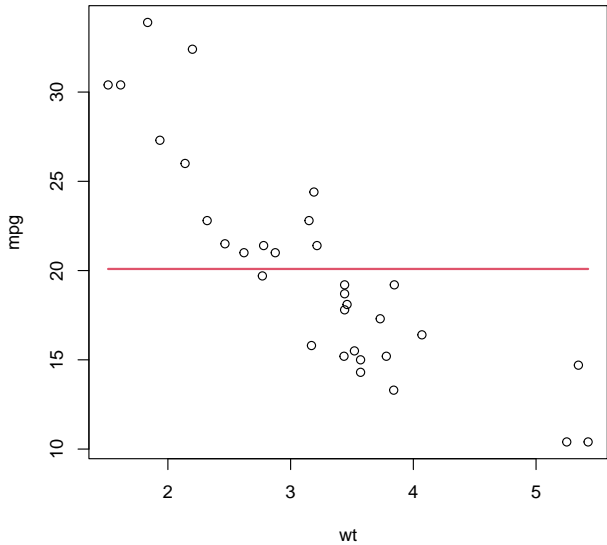
Regresión

```
> set.seed(1) # decay evita sobreajuste
> red <- nnet (mpg~wt, mtcars, size = 100, linout = TRUE,
+             decay=.1)
# weights: 301
initial value 21253.252588
iter 10 value 289.968862
iter 20 value 238.766078
iter 30 value 226.206939
iter 40 value 219.981624
iter 50 value 218.469071
iter 60 value 217.911102
iter 70 value 217.510822
iter 80 value 216.974718
iter 90 value 216.531471
iter 100 value 216.264122
final value 216.264122
stopped after 100 iterations
```



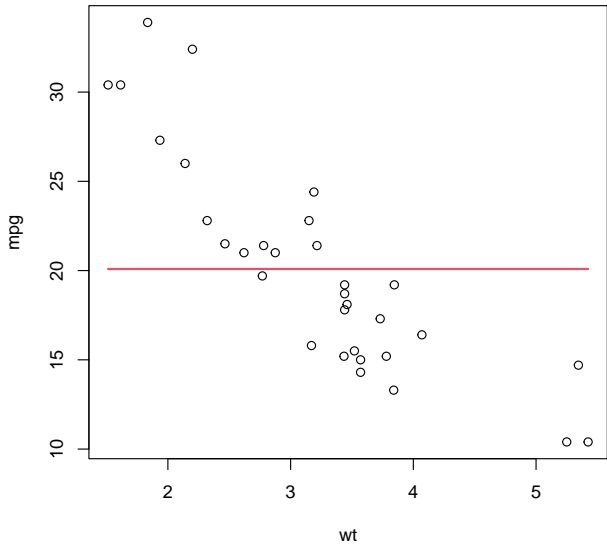
Regresión

```
> set.seed(1) # decay no funciona siempre
> red <- nnet (mpg ~ wt, mtcars, size = 2, linout = TRUE,
+             decay = 0.001)
# weights:  7
initial  value 13690.145855
iter 10 value 1126.294236
final   value 1126.291748
converged
```



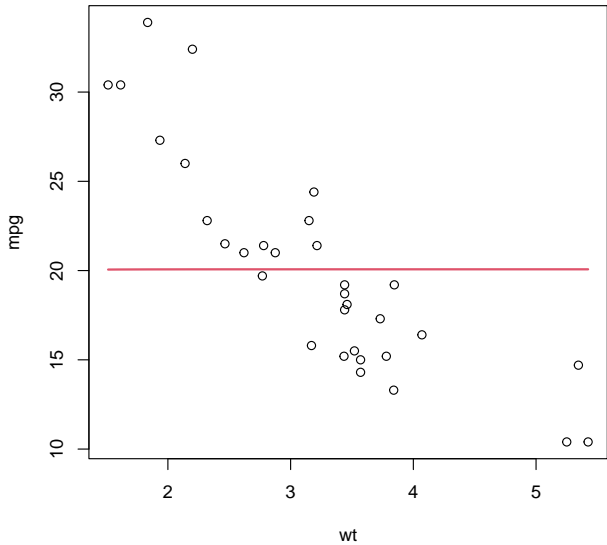
Regresión

```
> set.seed(1) # decay no funciona siempre
> red <- nnet (mpg ~ wt, mtcars, size = 2, linout = TRUE,
+            decay = 0.01)
# weights:  7
initial  value 13690.158004
iter   10 value 1128.325971
iter   20 value 1128.069333
final   value 1127.956753
converged
```



Regresión

```
> set.seed(1) # decay no funciona siempre
> red <- nnet (mpg ~ wt, mtcars, size = 2, linout = TRUE,
+             decay = 0.1)
# weights: 7
initial value 13690.279495
iter 10 value 1145.255981
final value 1143.101582
converged
```



Parámetros

- ▶ **maxit**: número de iteraciones
- ▶ **rang**: pesos inicializados según $\mathcal{U}(-\text{rang}, +\text{rang})$
- ▶ **decay**: decaimiento de pesos
 - ▶ pretende evitar óptimos locales al ajustar los w_{ij}
 - ▶ ajustar $E + \lambda \sum_{i,j} w_{ij}^2$
 - ▶ tiene sentido si $0 \lesssim x, y \lesssim 1$
 - ▶ se aconseja $0,001 \lesssim \lambda \lesssim 0,1$

Recomendaciones

- ▶ tipificar / normalizar / rescalar las variables
- ▶ ejecutar varias veces (probar distintas semillas)
- ▶ validación cruzada

```

> valcruz <- function (numneur, decai, partes=10)
+ {
+   iparte <- sample (rep (1:partes,
+                           length.out = nrow(mtcars)))
+   mean (sapply (1:partes,
+                 function (i)
+                 {
+                   red <- nnet (mpg ~ wt,
+                                mtcars[iparte!=i,],
+                                linout = TRUE,
+                                size = numneur,
+                                decay = decai,
+                                trace = FALSE)
+                   mean ((predict (red,
+                                   mtcars[iparte==i,]) -
+                                   mtcars$mpg[iparte==i]) ^ 2)
+                 })))
+ }

```

```
> valcruz ( 2, 0.001)
[1] 13.09448
> valcruz ( 2, 0.001)
[1] 28.91391
> valcruz ( 10, 0.001)
[1] 11.78668
> valcruz ( 10, 0.001)
[1] 11.84809
> valcruz (100, 0.001)
[1] 11.53392
> valcruz (100, 0.001)
[1] 11.5865
> mean (reg $ residuals ^ 2)
[1] 8.697561
```

```
> valcruzreg <- function (partes=10)
+ {
+   iparte <- sample (rep (1:partes,
+                           length.out = nrow(mtcars)))
+   mean (sapply (1:partes,
+                 function (i)
+                 {
+                   reg <- lm (mpg ~ wt,
+                               mtcars[iparte!=i,])
+                   mean ((predict (reg,
+                                   mtcars[iparte==i,]) -
+                                   mtcars$mpg[iparte==i]) ^ 2)
+                 })))
+ }
```

```
> valcruzreg ()  
[1] 9.345148  
> mean (reg $ residuals ^ 2)  
[1] 8.697561
```

```

> valcruz01 <- function (numneur, decai, partes=10)
+ {
+   iparte <- sample (rep (1:partes,
+                           length.out=nrow(mtcars01)))
+   mean (sapply (1:partes,
+                 function (i)
+                 {
+                   red <- nnet (mpg ~ wt,
+                                mtcars01[iparte!=i,],
+                                linout = TRUE,
+                                size = numneur,
+                                decay = decai,
+                                trace = FALSE)
+                   mean ((predict (red,
+                                    mtcars01[iparte==i,])
+                            - mtcars01$mpg[iparte==i]) ^ 2)
+                 })))
+ }

```

```
> mtcars01 <- data.frame (scale (mtcars))
> mean (lm(mpg~wt,mtcars01) $ residuals ^ 2)
[1] 0.2394432
> valcruz01 (2, 0.001)
[1] 0.2467679
> valcruz01 (2, 0.001)
[1] 0.2425711
> valcruz01 (2, 0.001)
[1] 0.3180561
> valcruz01 (2, 0.001)
[1] 0.2629049
```

Bibliografía

- ▶ <https://cran.r-project.org/view=MachineLearning>
- ▶ Ripley B.; 1996; Pattern recognition and neural networks; Cambridge University Press
- ▶ Venables W., Ripley B.; 2002; Modern applied statistics with S; Springer

Anexo: Retropropagación

Perceptrón simple: retropropagación

Para un patrón p :

$$y_k^{(p)} = \phi_\sigma \left(\sum_{i=0}^I w_{ik} x_i^{(p)} \right)$$

Error (entropía, respuesta dicótoma):

$$E = \sum_p \sum_k \left[t_k^{(p)} \ln \frac{t_k^{(p)}}{y_k^{(p)}} + (1 - t_k^{(p)}) \ln \frac{1 - t_k^{(p)}}{1 - y_k^{(p)}} \right]$$

Objetivo: minimizar E respecto a w_{ik} .

Perceptrón simple: gradiente

Para activación logística:

$$\phi'_o(z) = y_k(1 - y_k)$$

Derivada del error:

$$\frac{\partial E}{\partial w_{ik}} = \sum_p \left(y_k^{(p)} - t_k^{(p)} \right) x_i^{(p)}$$

Regla de actualización (descenso por gradiente):

$$w_{ik} \leftarrow w_{ik} - \eta \sum_p \left(y_k^{(p)} - t_k^{(p)} \right) x_i^{(p)}$$

$\eta > 0$: tasa de aprendizaje.

Perceptrón multicapa (1 oculta)

Capa oculta:

$$h_j^{(p)} = \phi \left(\sum_{i=0}^I w_{ij} x_i^{(p)} \right)$$

Capa de salida:

$$y_k^{(p)} = \phi_{\sigma} \left(\sum_{j=0}^J v_{jk} h_j^{(p)} \right)$$

El error E es el mismo que antes.

Multicapa: retropropagación

Definimos el *error local* en salida:

$$\delta_k^{(p)} = y_k^{(p)} - t_k^{(p)}$$

Gradiente en la capa de salida:

$$\frac{\partial E}{\partial v_{jk}} = \sum_p \delta_k^{(p)} h_j^{(p)}$$

Actualización:

$$v_{jk} \leftarrow v_{jk} - \eta \sum_p \delta_k^{(p)} h_j^{(p)}$$

Multicapa: error en capa oculta

Error propagado a la neurona oculta:

$$\delta_j^{(p)} = \phi' \left(\sum_i w_{ij} x_i^{(p)} \right) \sum_k \delta_k^{(p)} v_{jk}$$

Gradiente en pesos de entrada:

$$\frac{\partial E}{\partial w_{ij}} = \sum_p \delta_j^{(p)} x_i^{(p)}$$

Actualización:

$$w_{ij} \leftarrow w_{ij} - \eta \sum_p \delta_j^{(p)} x_i^{(p)}$$

Resumen: algoritmo de retropropagación

Para cada patrón p :

1. Propagación hacia delante: calcular $h_j^{(p)}$, luego $y_k^{(p)}$.
2. Calcular errores en salida: $\delta_k^{(p)}$.
3. Propagar errores hacia atrás: $\delta_j^{(p)}$.
4. Actualizar pesos: v_{jk} , luego w_{ij} .

Es un descenso por gradiente aplicado mediante regla de la cadena.

Descenso por gradiente vs BFGS (lo que usa `nnet`)

Sea $E(\mathbf{w})$ la función de error y $\nabla E(\mathbf{w})$ su gradiente.

1. Descenso por gradiente clásico

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \nabla E(\mathbf{w}^{(t)})$$

- ▶ $\eta > 0$: tasa de aprendizaje explícita.
- ▶ Dirección: gradiente negativo.
- ▶ Convergencia lineal.
- ▶ Sensible a la elección de η .

Descenso por gradiente vs BFGS (lo que usa `nnet`)

2. BFGS (cuasi-Newton)

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - H_t^{-1} \nabla E(\mathbf{w}^{(t)})$$

donde H_t^{-1} aproxima la inversa del Hessiano.

- ▶ No hay parámetro η explícito.
- ▶ Dirección adaptativa usando curvatura.
- ▶ Búsqueda en línea interna para el tamaño de paso.
- ▶ Convergencia superlineal (en condiciones regulares).

¿Qué implementa `nnet` en R?

Paquete: `nnet`

- ▶ Optimización mediante algoritmo BFGS.
- ▶ Minimiza directamente la entropía (o SSE).
- ▶ El tamaño del paso se determina por búsqueda en línea.
- ▶ No existe parámetro de tasa de aprendizaje η .
- ▶ El argumento `decay` implementa regularización:

$$E_{\text{reg}} = E + \lambda \sum w^2$$

Por tanto:

`nnet` \neq descenso por gradiente con tasa fija

Es un método cuasi-Newton determinista, adecuado para redes pequeñas y medianas.

Búsqueda en línea: condición de Armijo

Sea $E(\mathbf{w})$ diferenciable y \mathbf{d}_t una dirección de descenso:

$$\nabla E(\mathbf{w}^{(t)})^\top \mathbf{d}_t < 0$$

Se actualiza:

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} + \alpha_t \mathbf{d}_t$$

Condición de Armijo (suficiente descenso)

Dado $c \in (0, 1)$, se busca $\alpha_t > 0$ tal que:

$$E(\mathbf{w}^{(t)} + \alpha_t \mathbf{d}_t) \leq E(\mathbf{w}^{(t)}) + c\alpha_t \nabla E(\mathbf{w}^{(t)})^\top \mathbf{d}_t$$

Interpretación: el descenso real debe ser proporcional al descenso lineal predicho por el gradiente.

Búsqueda en línea: *backtracking*

Procedimiento típico:

1. Fijar $\alpha = \alpha_0$ (p.ej. 1).
2. Mientras no se cumpla Armijo:

$$\alpha \leftarrow \rho\alpha, \quad \rho \in (0, 1)$$

3. Tomar $\alpha_t = \alpha$.

Propiedades:

- ▶ Garantiza descenso.
- ▶ Evita pasos excesivos.
- ▶ Compatible con descenso por gradiente y BFGS.