Neuronas artificiales

Análisis de Datos

27 de abril de 2026

Introducción

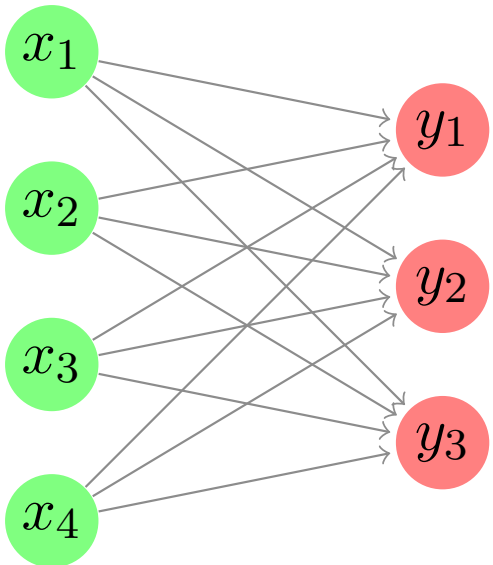
- inspirado en las conexiones (sinapsis) entre neuronas cerebrales
- aprendizaje supervisado: clasificación o regresión
- son modelos de regresión no lineal con muchos parámetros (caja negra: el ajuste no es constructivo)
- permiten extraer patrones de información no estructurada (textos, fotos...) pero a veces alucinan
- tipos
  - perceptrón multicapa (MLP)
  - base radial (RBF)
  - mapas autoorganizados (SOM; no supervisado)
  - convolucionales (CNN)
  - recurrentes (RNN)
  - adversativas (GAN)
  - trasformadores (p.ej. GPT)
  - ...

Índice

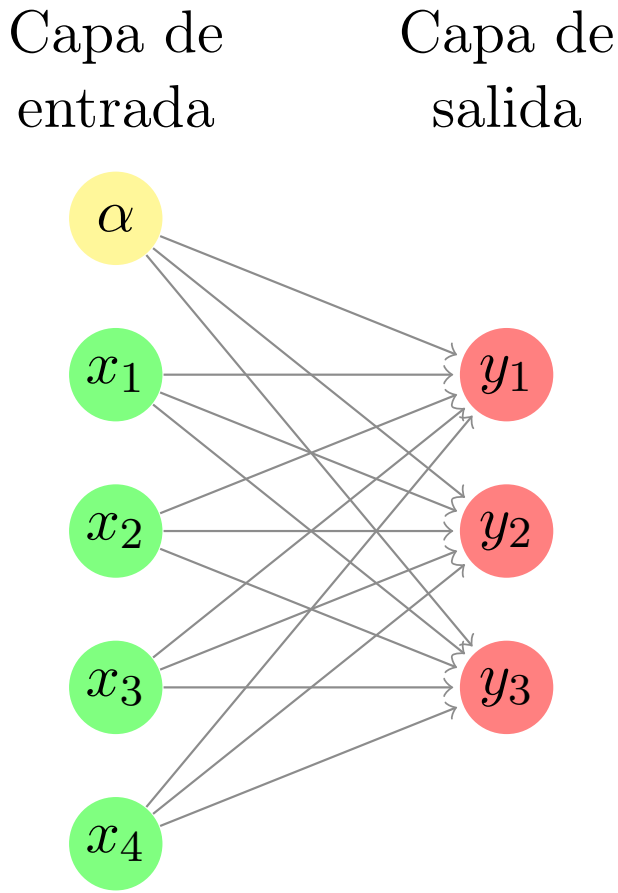
Perceptrón simple

Capa de entrada

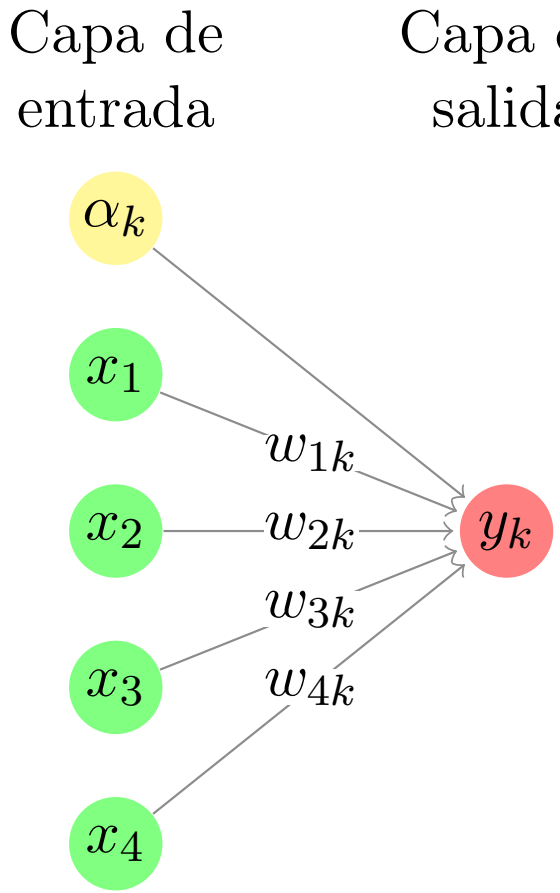
Capa de salida



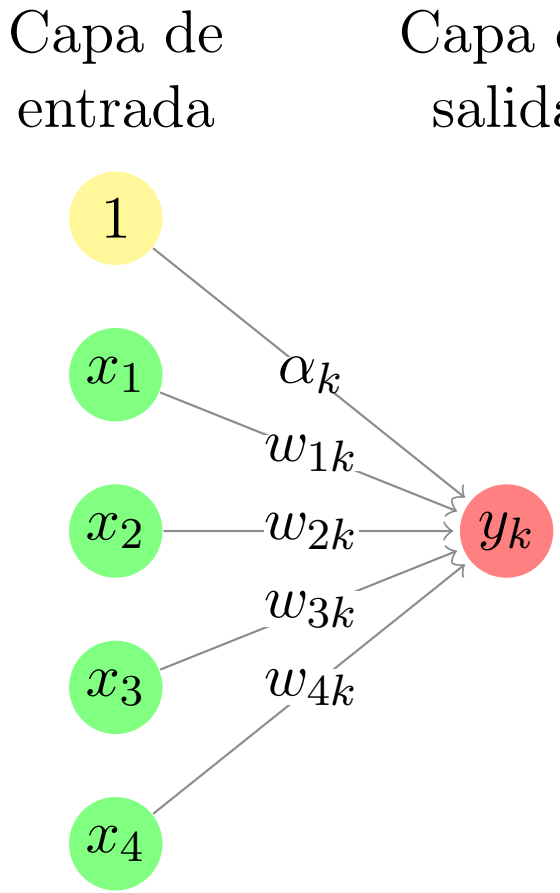
Perceptrón simple



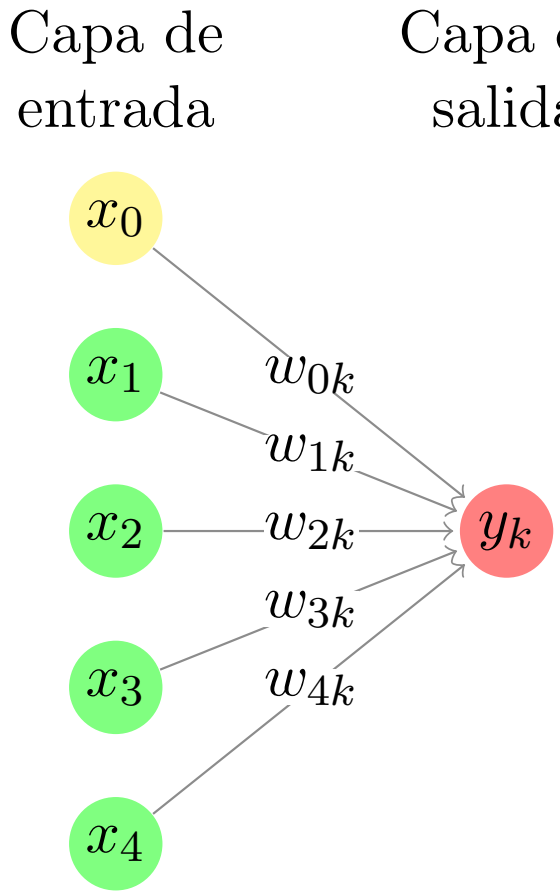
Perceptrón simple



Perceptrón simple



Perceptrón simple



Perceptrón simple

- x_i = variable de entrada $(i = 1, \ldots, I)$
- y_k = variable de salida $(k = 1, \ldots, K)$
- ϕ_o = función de activación en la capa de salida $(output)$
- α_k = constante, sesgo $(bias)$ para la salida k
- w_{ik} = peso $(weight)$, coeficiente de sinapsis entre x_i y y_k

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

Perceptrón simple

Sean $\mathbf{x} = (x_1, \ldots, x_I)^\top \in \mathbb{R}^I$ y $\mathbf{x}_1 = (1, x_1, \ldots, x_I)^\top$.
Definimos el modelo como una familia de aplicaciones:

$$f_{\mathbf{w}} : \mathbb{R}^I \to \mathbb{R}^K, \quad \mathbf{y} = f_{\mathbf{w}}(\mathbf{x}) = g\left[\phi_o\left(\mathbf{w}_1^\top \mathbf{x}_1\right), \ldots, \phi_o\left(\mathbf{w}_K^\top \mathbf{x}_1\right)\right]$$

donde $g \in \{\text{identidad}, \text{softmax}\}$

Sean $\mathbf{t}$=*target* (respuesta) $\mathbf{y}$=predicción (salida)

Error o pérdida $E(\mathbf{w}) = E[\mathbf{y}, \mathbf{t}] = E[f_{\mathbf{w}}(\mathbf{x}), \mathbf{t}]$

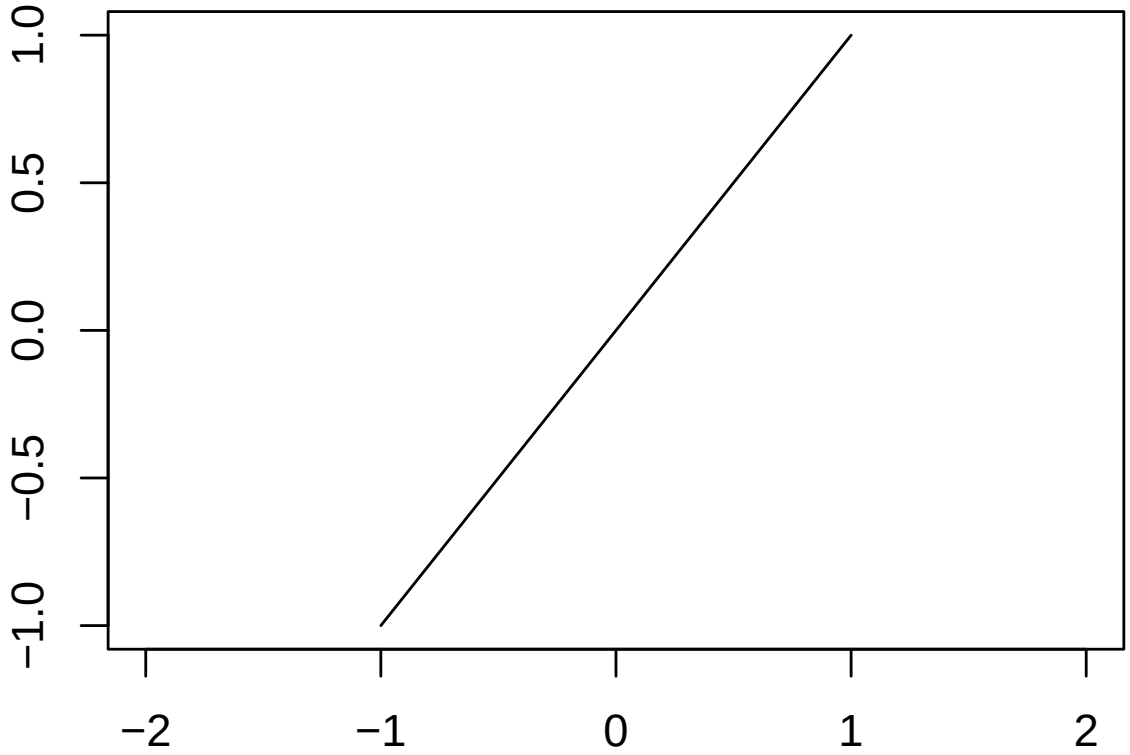
- E es diferenciable si ϕ lo es.
  - esto permite calcular gradientes respecto a $\mathbf{x}$ y a $\mathbf{w}$
  - esencial para optimización mediante retropropagación
- El entrenamiento consiste en resolver: $\min_{\mathbf{w} \in \mathbb{R}^{I+1}} E(\mathbf{w})$

Interpretación: modelo no lineal en $\mathbf{x}$, pero lineal en los parámetros antes de la activación.

Perceptrón simple

- lineal

$$\phi_o(x) = x$$



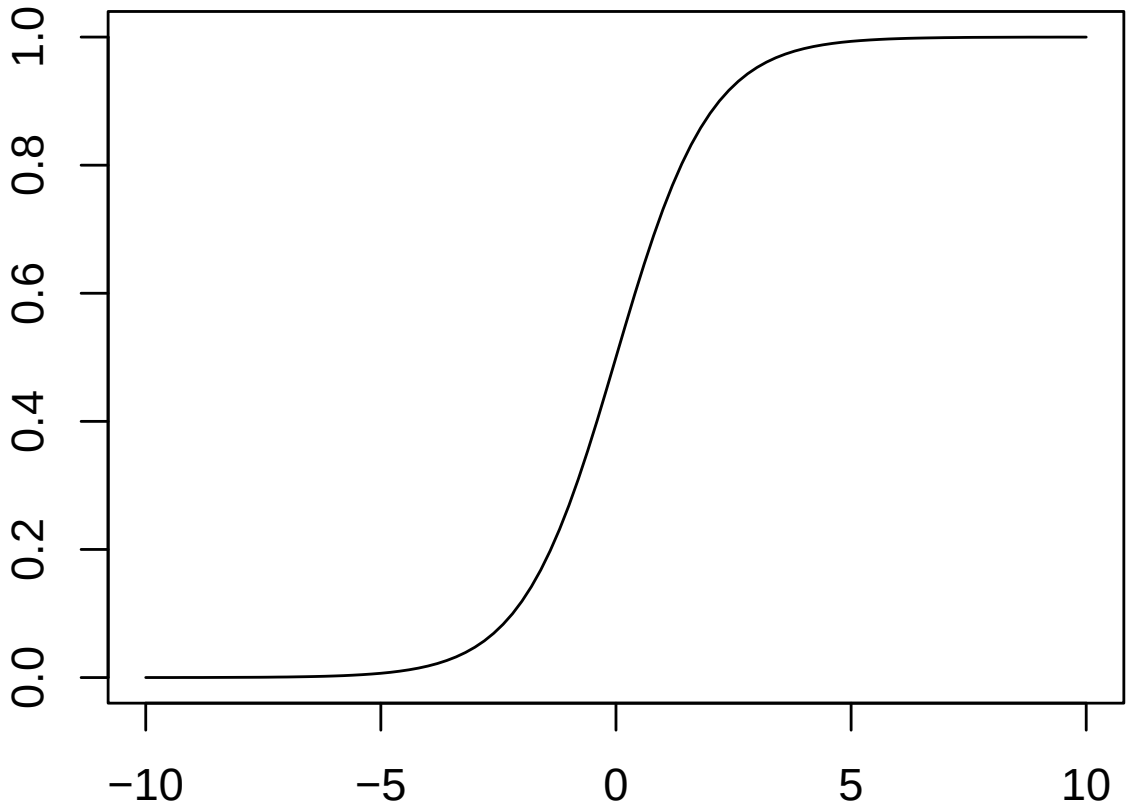
Perceptrón simple

- lineal

$$\phi_o(x) = x$$

- logística

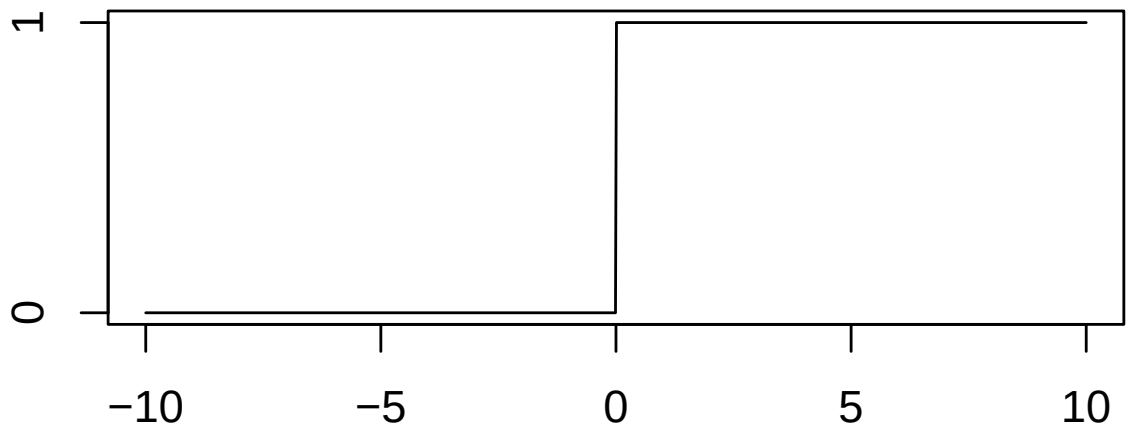
$$\phi_o(x) = \ell(x) = \frac{\exp(x)}{1+\exp(x)} = \frac{1}{1+\exp(-x)}$$



Perceptrón simple

- indicatriz, umbral, característica, Heaviside

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



Perceptrón simple

- indicatriz, umbral, característica, Heaviside

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

  - no es derivable
  - no es adecuada para optimización por gradiente
  - se aproxima en la práctica por funciones suaves ; p.ej. mediante la logística :

$$\ell_\beta(x) = \frac{1}{1+\exp(-\beta x)}, \quad \ell_\beta(x) \to \mathbb{1}_{[0,\infty)}(x) \text{ cuando } \beta \to \infty$$

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
  $n =$ instancia $\quad t =$ respuesta $\in \{0; 1\} \quad y =$ predicción
  - para respuesta dicótoma: una neurona de salida, $y \in [0; 1]$

$$E = \sum_{n=1}^{N} \left[t^{(n)} \ln \frac{t^{(n)}}{y^{(n)}} + (1 - t^{(n)}) \ln \frac{1 - t^{(n)}}{1 - y^{(n)}} \right]$$

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

$$E = \sum_n \sum_k -t_k^{(n)} \ln \widehat{\Pr}[t_n = k] \qquad \widehat{\Pr}[t_n = k] = \frac{\exp\left(y_k^{(n)}\right)}{\sum_{c=1}^{K} \exp\left(y_c^{(n)}\right)}$$

Entropía, entropía cruzada y divergencia KL

Entropía de Shannon

- Distribución discreta sobre K clases: $H(\mathbf{p}) = -\sum_{k=1}^{K} p_k \ln p_k$
- Mide la incertidumbre o la cantidad media de información.

H **cruzada** entre $\mathbf{p}$ (real) y $\mathbf{q}$ (modelo):

$$H(\mathbf{p}, \mathbf{q}) = -\sum_k p_k \ln q_k$$

- No es simétrica: $H(\mathbf{p}, \mathbf{q}) \neq H(\mathbf{q}, \mathbf{p})$.
- En aprendizaje supervisado, $\mathbf{p}$ es la distribución empírica (degenerada o *one-hot*: un 1 y el resto 0).

Divergencia de Kullback–Leibler (o entropía relativa):

$$D_{\mathrm{KL}}(\mathbf{p}\|\mathbf{q}) = \sum_k p_k \ln \frac{p_k}{q_k} = H(\mathbf{p}, \mathbf{q}) - H(\mathbf{p})$$

- Mide la "distancia" (no simétrica) entre dos distribuciones.
- $D_{\mathrm{KL}}(\mathbf{p}\|\mathbf{q}) \geq 0$ y es cero si y solo si $\mathbf{p} = \mathbf{q}$ (casi seguro).

Aplicación a los criterios de ajuste

- El término $H(\mathbf{t})$ es constante (depende sólo de los datos).
- Minimizar D_{KL} equivale a minimizar $H(\mathbf{t}, \mathbf{y})$.
- $H(\mathbf{t}) = 0$ si $\mathbf{t}$ es degenerada.

Caso binario (una salida logística y, objetivo $t \in \{0, 1\}$) :

$$D_{\mathrm{KL}}\big((t, 1-t) \,\|\, (y, 1-y)\big) = t \ln \frac{t}{y} + (1-t) \ln \frac{1-t}{1-y} =$$
$$= \underbrace{\big[-t \ln y - (1-t) \ln(1-y)\big]}_{H(\mathbf{t},\mathbf{y})} - \underbrace{\big[-t \ln t - (1-t) \ln(1-t)\big]}_{H(\mathbf{t})}$$

Caso multiclase ($\mathbf{p} = \mathrm{softmax}(\mathbf{y})$, objetivo $\mathbf{t}$):

$$D_{\mathrm{KL}}(\mathbf{t} \| \mathbf{p}) = \sum_k t_k \ln \frac{t_k}{p_k} = H(\mathbf{t}, \mathbf{p}) - H(\mathbf{t})$$

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 733.700647
iter  10 value 31.866291
iter  20 value 6.864117
iter  30 value 6.052776
iter  40 value 5.957074
iter  50 value 5.951153
iter  60 value 5.949332
final  value 5.949330
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
  6.16   6.23   4.19 -12.85  -9.84
 b->o2 i1->o2 i2->o2 i3->o2 i4->o2
 17.84  -2.50   1.26   2.05  -4.09
 b->o3 i1->o3 i2->o3 i3->o3 i4->o3
-24.80  -4.97  -5.42  11.48  14.20
```

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]   6.163855   6.230980   4.193952 -12.853916
 [5]  -9.835551  17.836360  -2.502295   1.261397
 [9]   2.047186  -4.088112 -24.798713  -4.967447
[13]  -5.418728  11.476033  14.196443
> head (red $ fitted.values)
     setosa   versicolor    virginica
1 1.0000000 6.724377e-10 1.043378e-36
2 1.0000000 1.671133e-08 1.198096e-33
3 1.0000000 1.201458e-08 1.444082e-34
4 0.9999992 7.596998e-07 1.502031e-31
5 1.0000000 1.201086e-09 1.222672e-36
6 1.0000000 4.178565e-09 1.402248e-34
```

Perceptrón simple

```
> tail (red $ fitted.values)
          setosa   versicolor virginica
145 3.073995e-27 1.174533e-08 1.0000000
146 4.035095e-21 6.841385e-06 0.9999932
147 7.257596e-19 8.938795e-04 0.9991061
148 5.805305e-19 1.006592e-03 0.9989934
149 5.379827e-24 4.378460e-06 0.9999956
150 9.559532e-19 2.232494e-02 0.9776751
> 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] 5.94933
> 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] 5.94933
```

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 1.164748e-16  0.1931563 0.8068437
> predict (red, flor, type="class")
[1] "virginica"
```

Perceptrón simple

```
> predict (red, flor)
        setosa versicolor virginica
1 1.164748e-16  0.1931563 0.8068437
> 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
  6.16   6.23   4.19 -12.85  -9.84
 b->o2 i1->o2 i2->o2 i3->o2 i4->o2
 17.84  -2.50   1.26   2.05  -4.09
 b->o3 i1->o3 i2->o3 i3->o3 i4->o3
-24.80  -4.97  -5.42  11.48  14.20
```

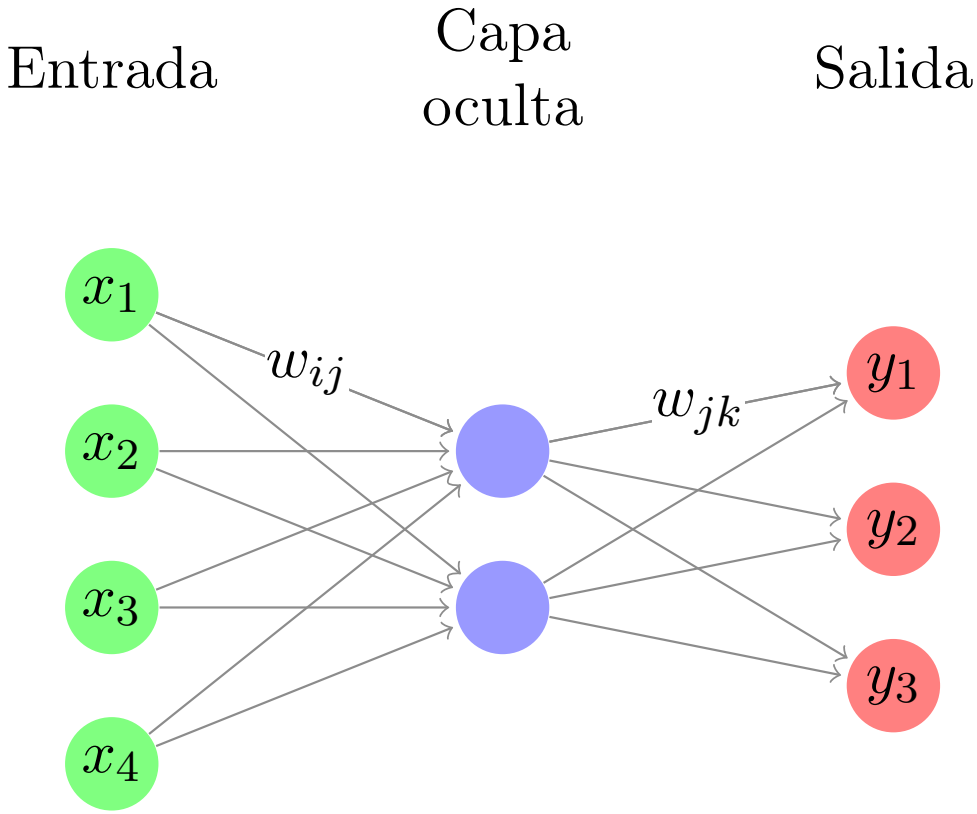
Perceptrón simple

```
> predict (red, flor)
        setosa versicolor virginica
1 1.164748e-16  0.1931563 0.8068437
> 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] 1.164748e-16 1.931563e-01 8.068437e-01
```

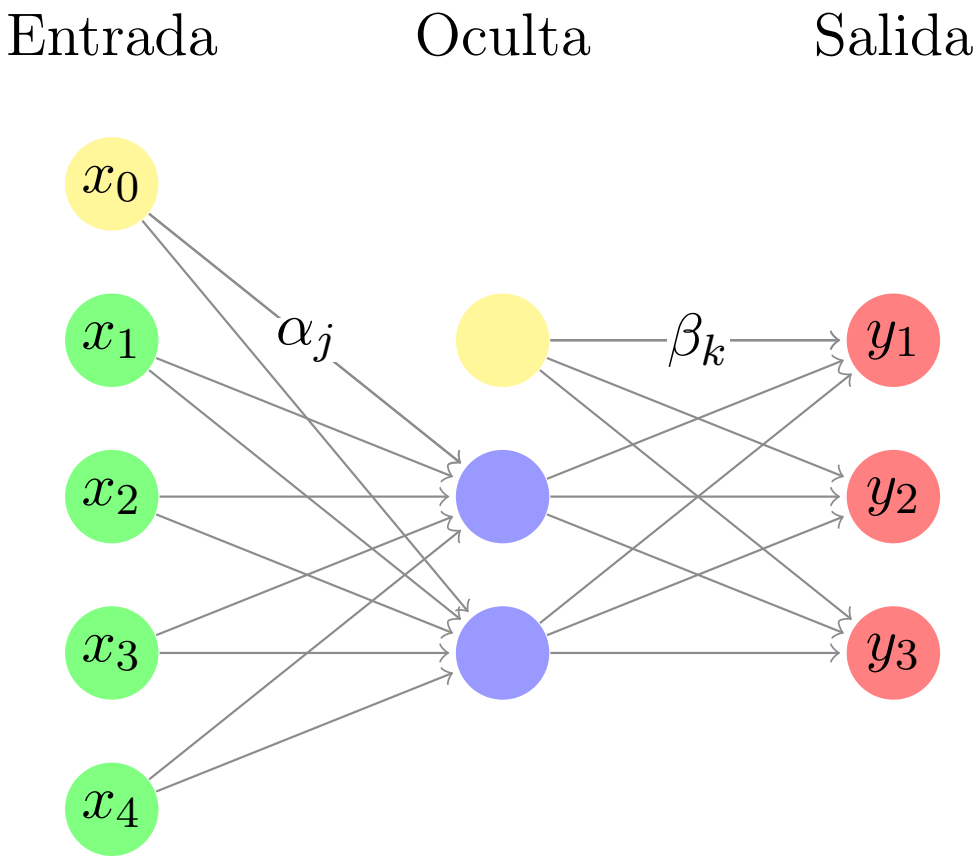
Respuesta dicótoma

```
> red2 <- nnet (factor(am)~mpg, mtcars,
+                 size=0, skip=TRUE, trace=FALSE)
> predict (red2, data.frame (mpg = 20))
        [,1]
1 0.3862902
> 1 / (1 + 1/exp (red2$wts[1] + red2$wts[2] * 20)) #logit
[1] 0.3862902
> red2 $ value
[1] 14.83758
> p <- red2 $ fitted.values                    #una sola columna
> 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)



Perceptrón multicapa (1 oculta)



Perceptrón multicapa (1 oculta)

- $j =$ índice de neuronas en capa oculta (*hidden*)

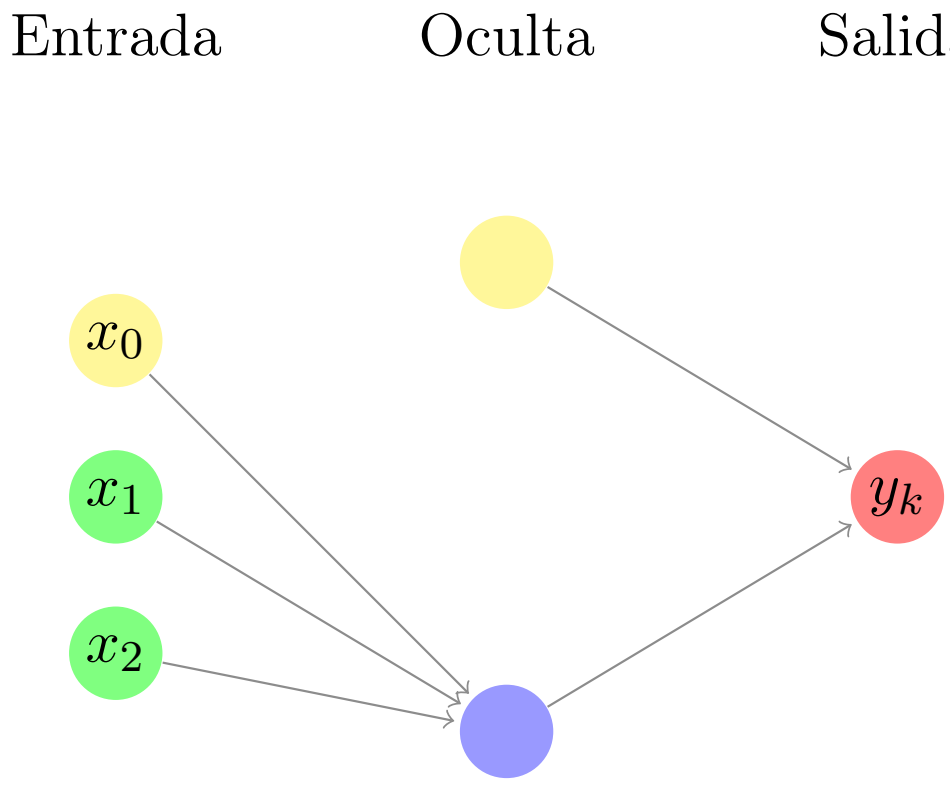
$$y_k = \phi_o \left(\beta_k + \sum_{j=1}^{J} w_{jk}\, \phi_h \left(\alpha_j + \sum_{i=1}^{I} w_{ij}\, x_i \right) \right)$$

$$y_k = \phi_o \left(\sum_{j=0}^{J} w_{jk}\, \phi_h \left(\sum_{i=0}^{I} w_{ij}\, x_i \right) \right)$$

- ϕ casi siempre logística en la oculta

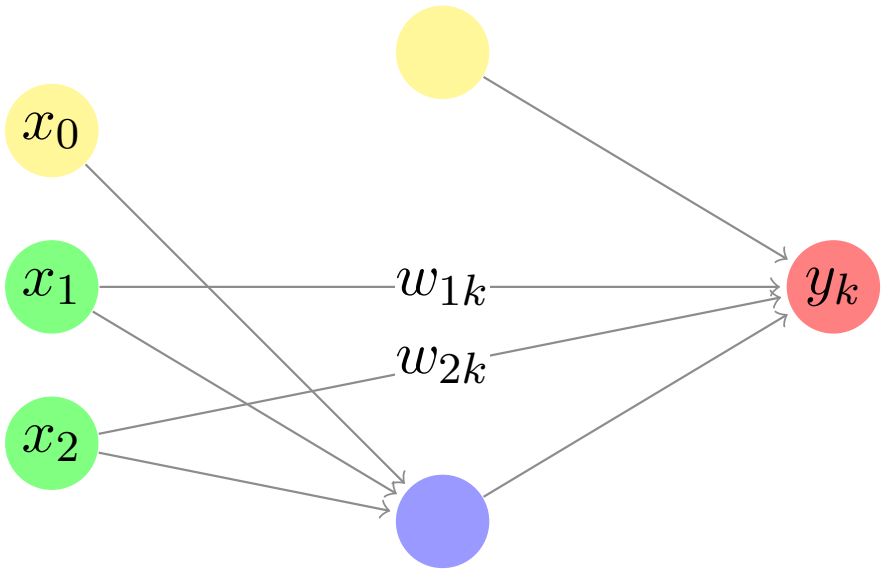
$$\phi_h(x) = \ell(x) = \frac{\exp(x)}{1 + \exp(x)}$$

Perceptrón multicapa (1 oculta, `skip=FALSE`)



Perceptrón multicapa (1 oculta, `skip=TRUE`)

Entrada Oculta Salida



Perceptrón multicapa (1 oculta)

- `skip=FALSE`

$$y_k = \phi_o \left(\sum_j w_{jk} \, \phi_h \left(\sum_i w_{ij} \, x_i \right) \right)$$

- `skip=TRUE`

$$y_k = \phi_o \left(\sum_j w_{jk} \, \phi_h \left(\sum_i w_{ij} \, x_i \right) + \sum_i w_{ik} \, x_i \right)$$

- los atajos o conexiones soslayantes (*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
  79.27  111.02   83.46 -242.90 -114.89
  b->h2  i1->h2  i2->h2  i3->h2  i4->h2
   2.88  158.40 -112.52  470.73  185.39
  b->o1  h1->o1  h2->o1
 -99.26  204.70   38.22
  b->o2  h1->o2  h2->o2
 417.44  -84.87 -386.84
  b->o3  h1->o3  h2->o3
-318.89 -119.03  349.52
```

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
 -0.79  -1.30  -0.01  -2.28  -0.57
 b->h2 i1->h2 i2->h2 i3->h2 i4->h2
  5.37  25.28  15.20  11.53   2.21
 b->o1 h1->o1 h2->o1 i1->o1 i2->o1 i3->o1 i4->o1
  1.62   0.33   0.47   4.47  15.48 -17.22 -10.69
 b->o2 h1->o2 h2->o2 i1->o2 i2->o2 i3->o2 i4->o2
  7.40  -3.89  13.09  -1.39  -4.69   3.91  -4.10
 b->o3 h1->o3 h2->o3 i1->o3 i2->o3 i3->o3 i4->o3
 -8.75   4.03 -13.41  -3.85 -11.37  13.35  14.19
```

Regresión

- función de activación de salida lineal: $\phi_o(x) = x$
- teorema de aproximación universal (Cybenko, Hornik, etc.)
  - sea $f : (C \text{ compacto}) \subset \mathbb{R}^I \to \mathbb{R}$ continua
  - existe perceptrón que aproxima f uniformemente en C
  - basta
    - elegir funciones de activación adecuadas
    - incrementar el número de neuronas en la capa oculta
- la aproximación es "no constructiva"
  - el número de neuronas se decide por validación
- criterios de ajuste (n=instancia, t=objetivo, y=predicción)
  - mínimos cuadrados: $E = \sum_n \|t^{(n)} - y^{(n)}\|^2$

Parámetros

- `maxit`: límite del número de iteraciones antes de alcanzar convergencia
- `rang`: pesos inicializados según $\mathcal{U}(-\text{rang}, +\text{rang})$
- `decay`: coeficiente λ de decaimiento de pesos
  - pretende evitar óptimos locales al ajustar los pesos w
  - minimizar $E + \lambda \sum_{i,j,k} w^2$
  - se aconseja $0{,}001 \lessapprox \lambda \lessapprox 0{,}1$
  - recuerda: conviene tipificar las instancias

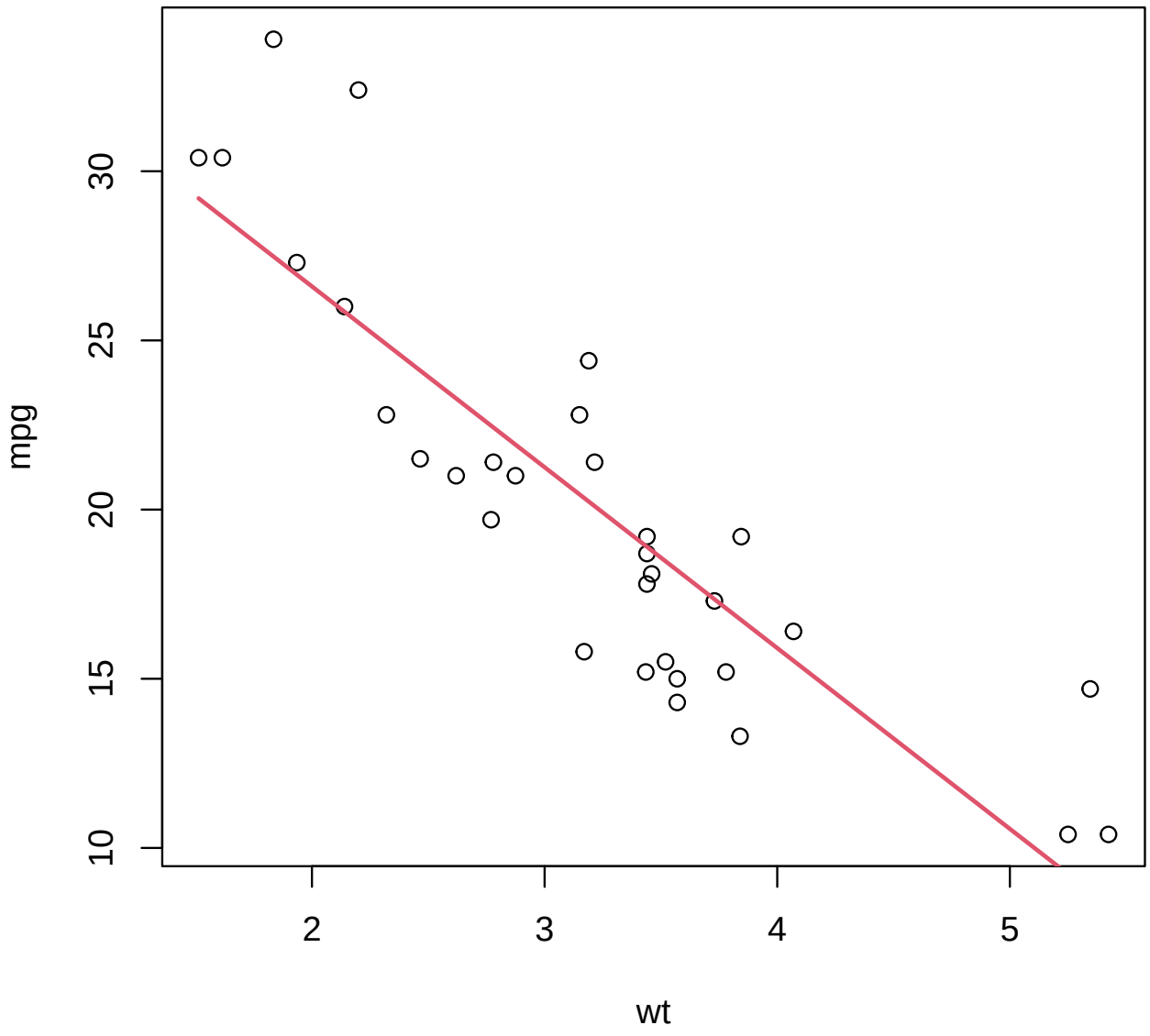
```
> E <- function (l) {
+  red <- nnet(Species~., iris,, 2, decay=l, trace=FALSE)
+  c(red$value,
+   -sum(log(red$fitted[cbind(1:150,rep(1:3,each=50))]))+
+   + l * sum(red$wts^2)) }
> E (0) ; E (.1)
[1] 5.9724 5.9724
[1] 52.72997 52.72997
```

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
```

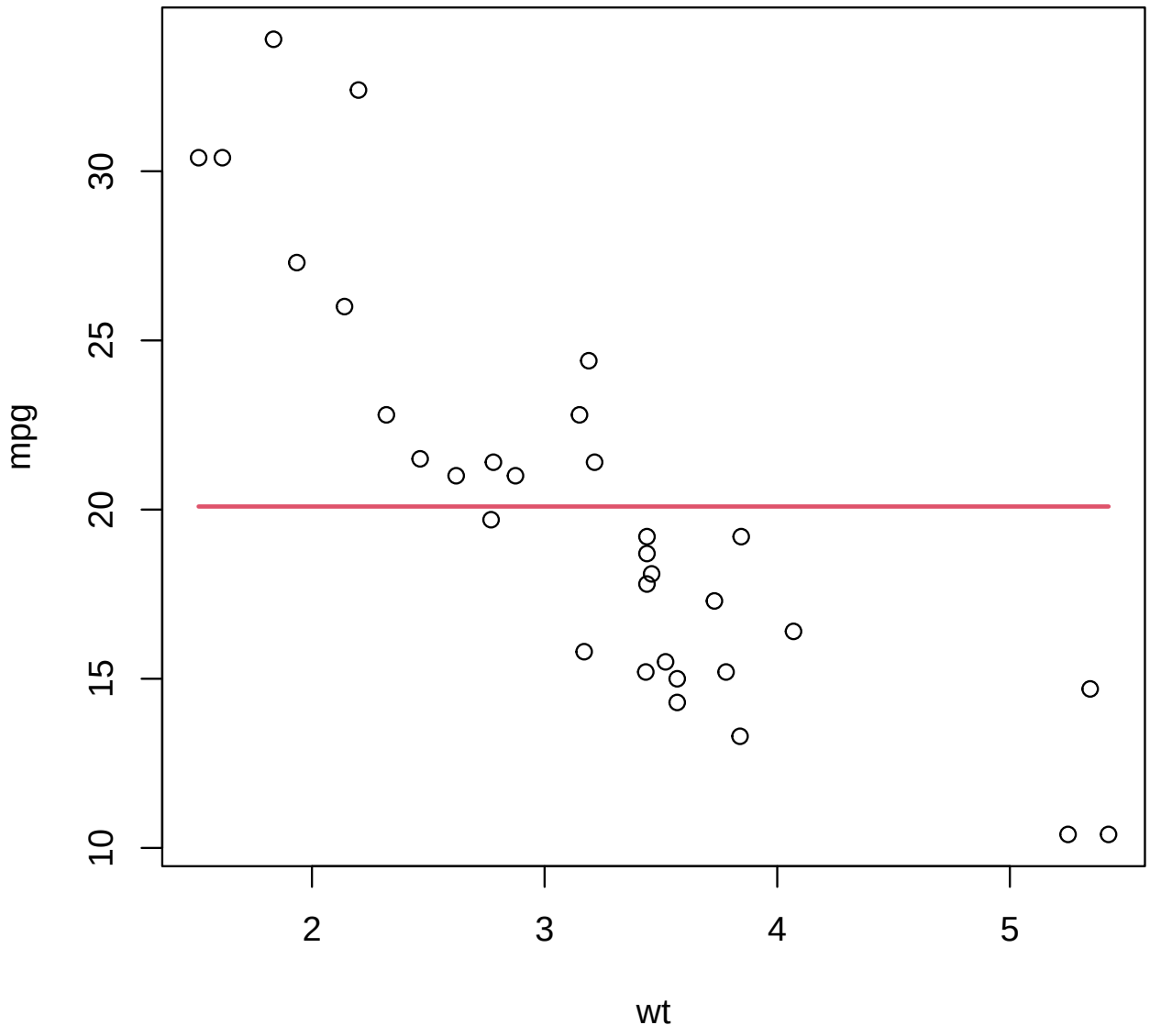

mpg
10
15
20
25
30
2
3
4
5
wt

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
```

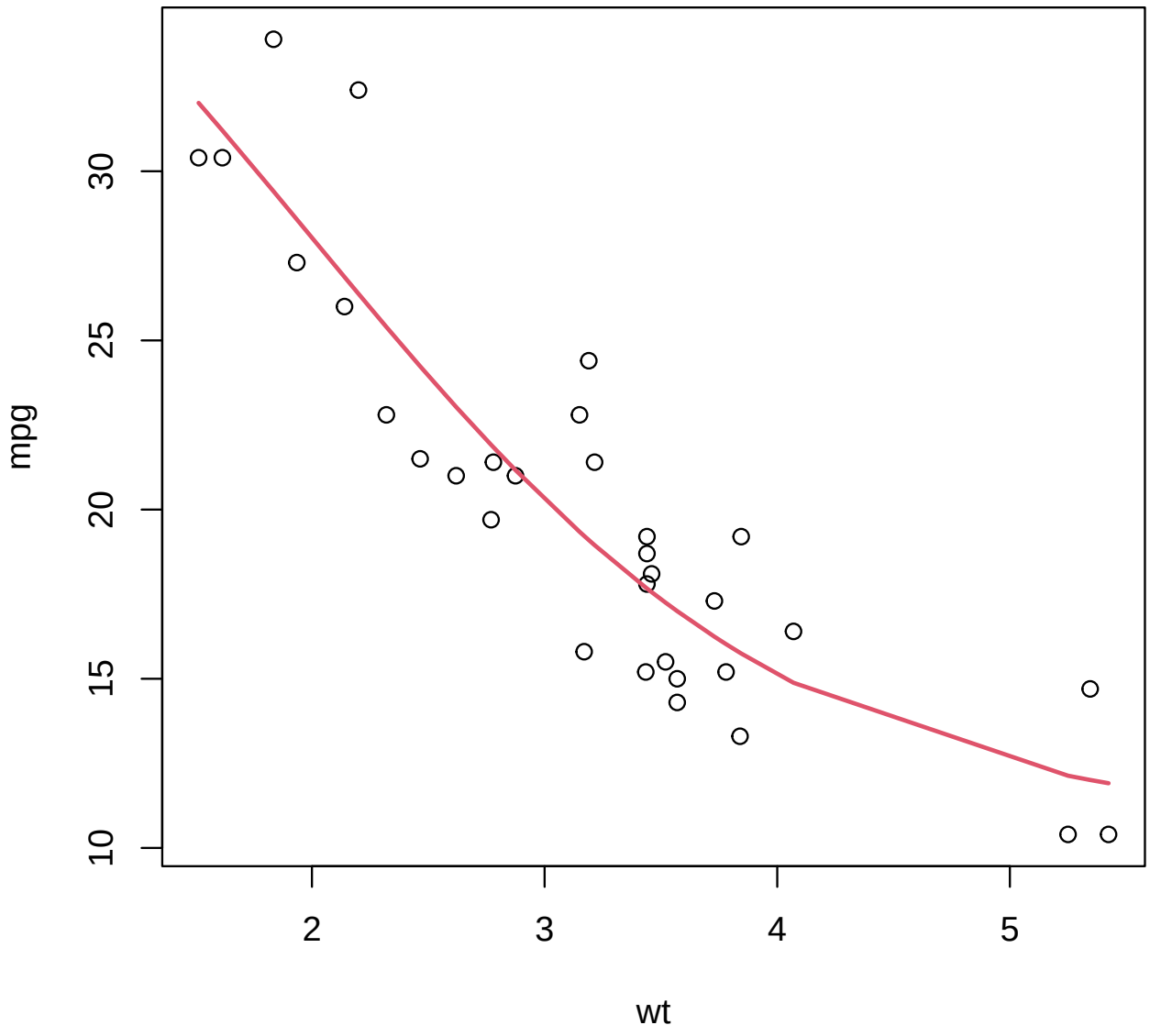
Regresión

```
> red $ value
[1] 1126.047
> sum (red $ residuals ^ 2)
[1] 1126.047
> sum (reg $ residuals ^ 2)
[1] 278.3219
```


mpg
10
15
20
25
30
2
3
4
5
wt

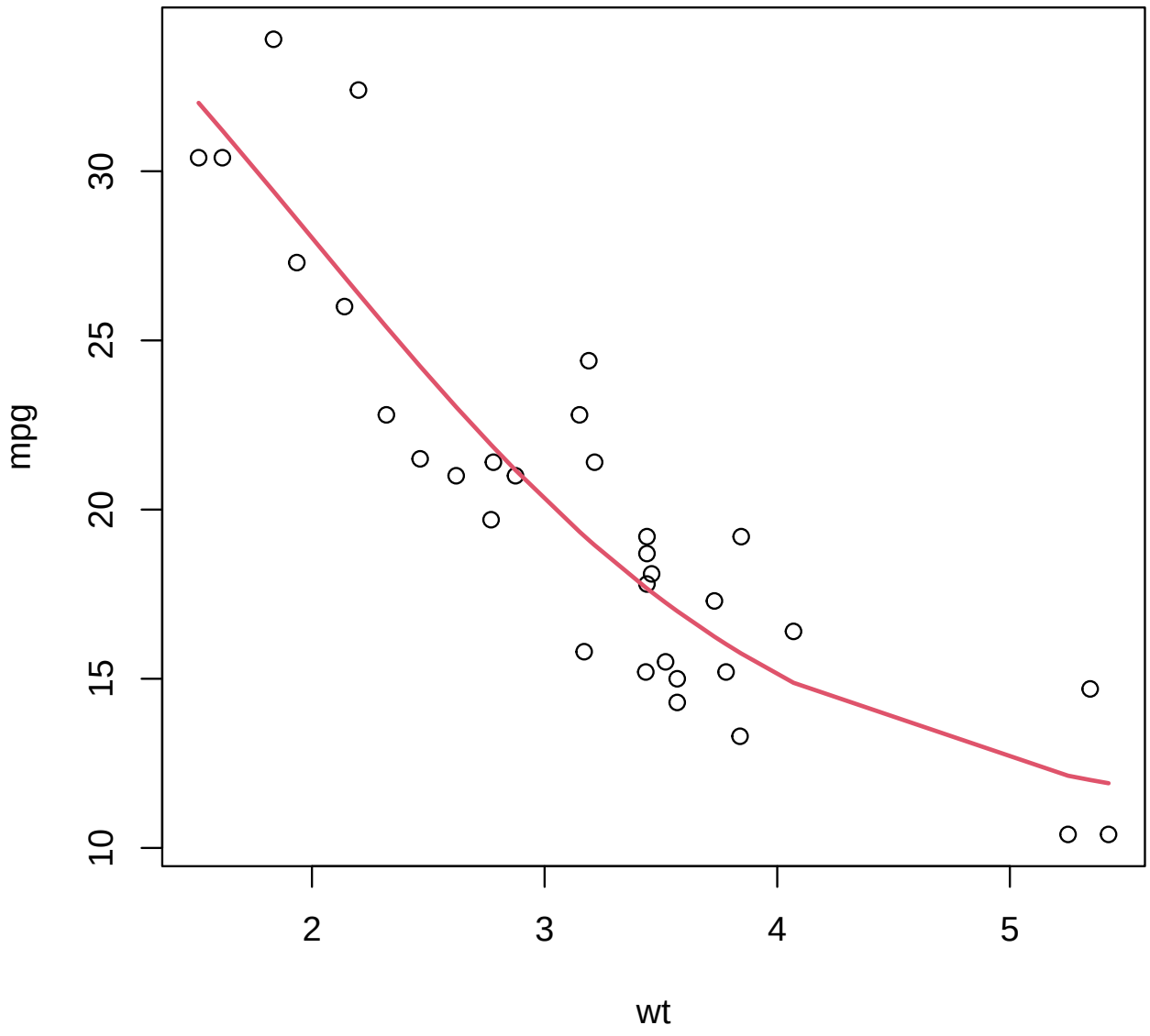
Regresión

```
> 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
```


mpg
10
15
20
25
30
2
3
4
5
wt

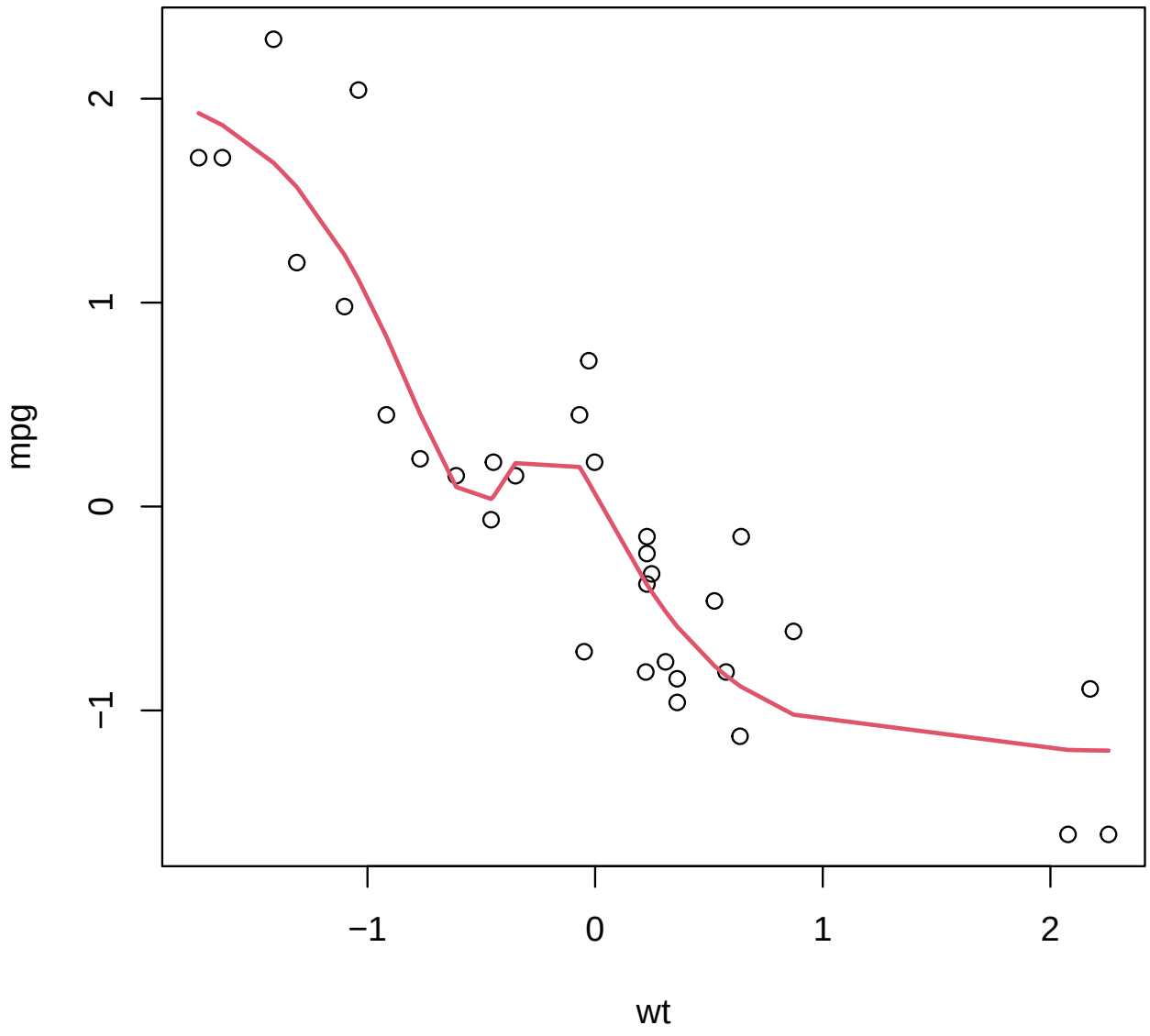
Regresión

```
> 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
```


mpg
10
15
20
25
30
2
3
4
5
wt

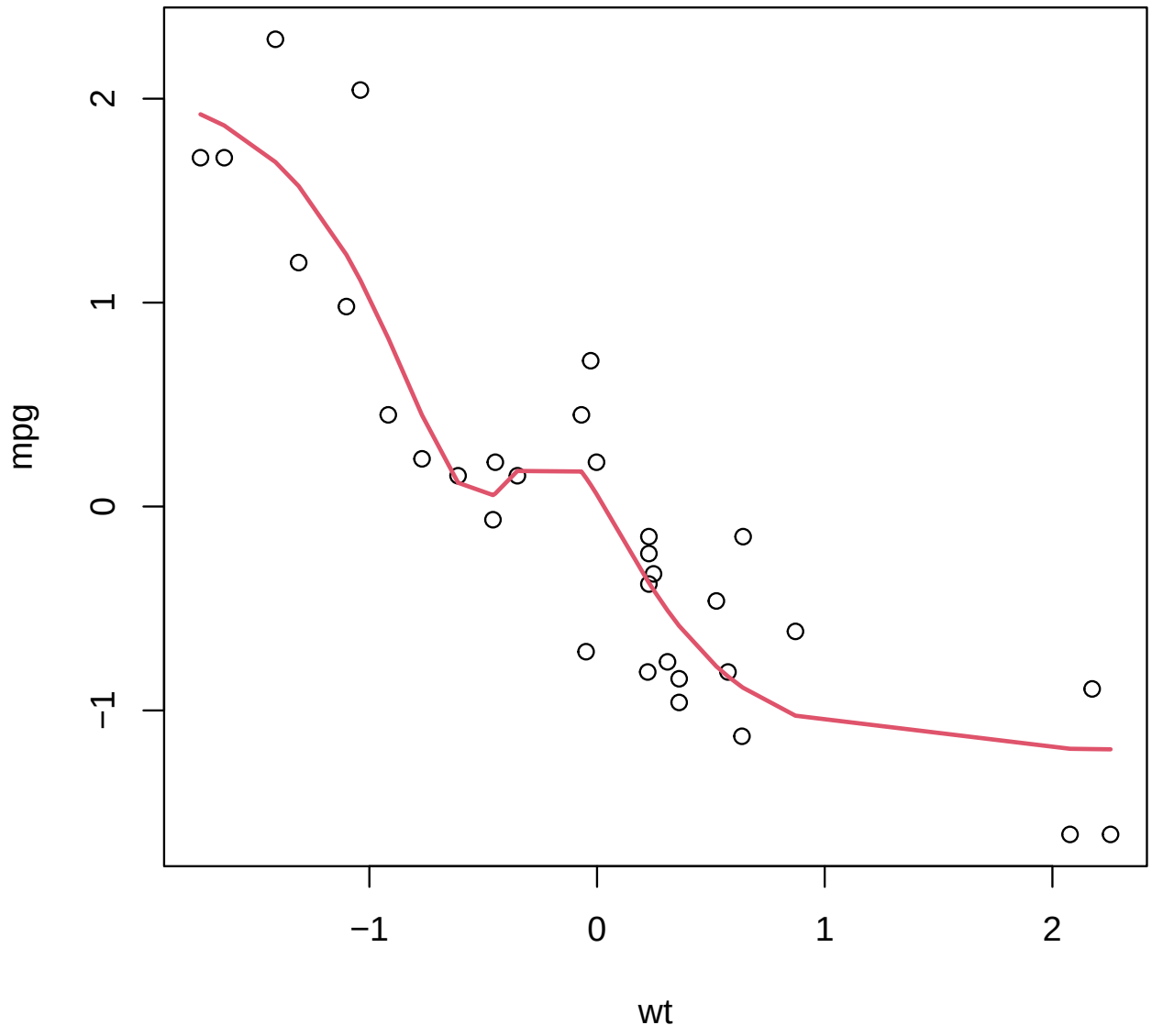
Regresión

```
> 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
```


mpg
2
1
0
−1
−1
0
1
2
wt

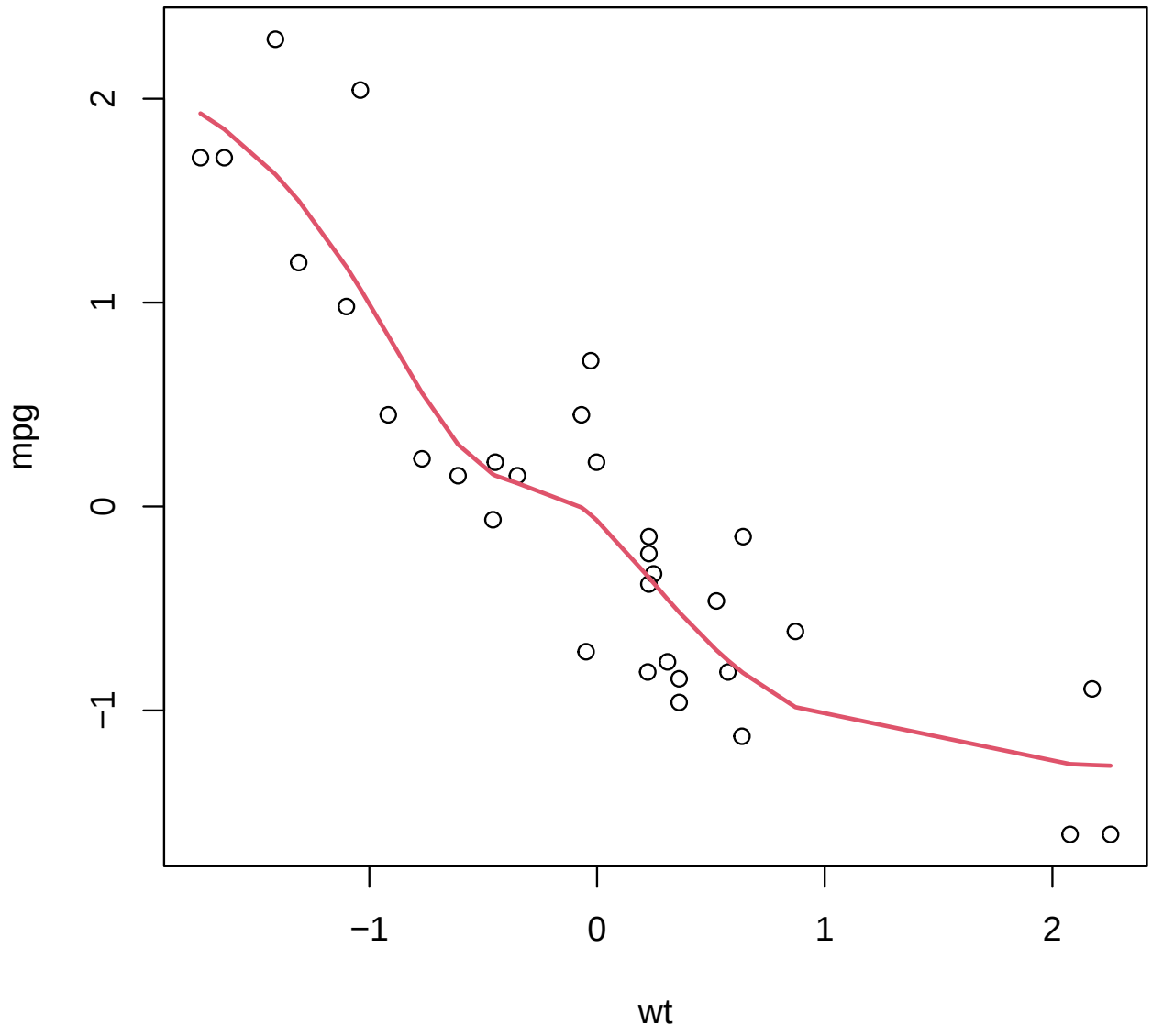
Regresión

```
> 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
```


mpg
2
1
0
−1
−1
0
1
2
wt

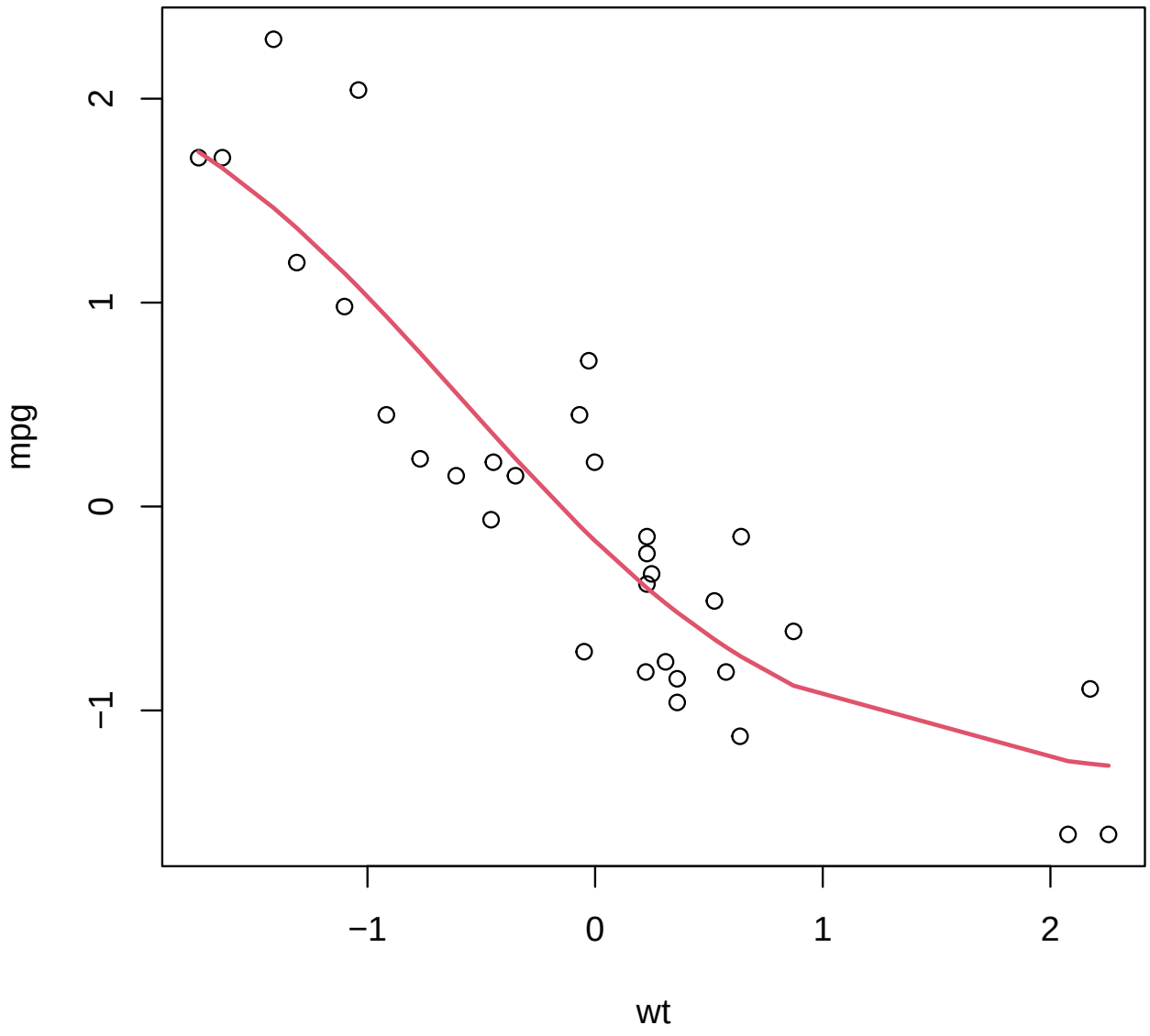
Regresión

```
> 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
```


mpg
2
1
0
−1
−1
0
1
2
wt

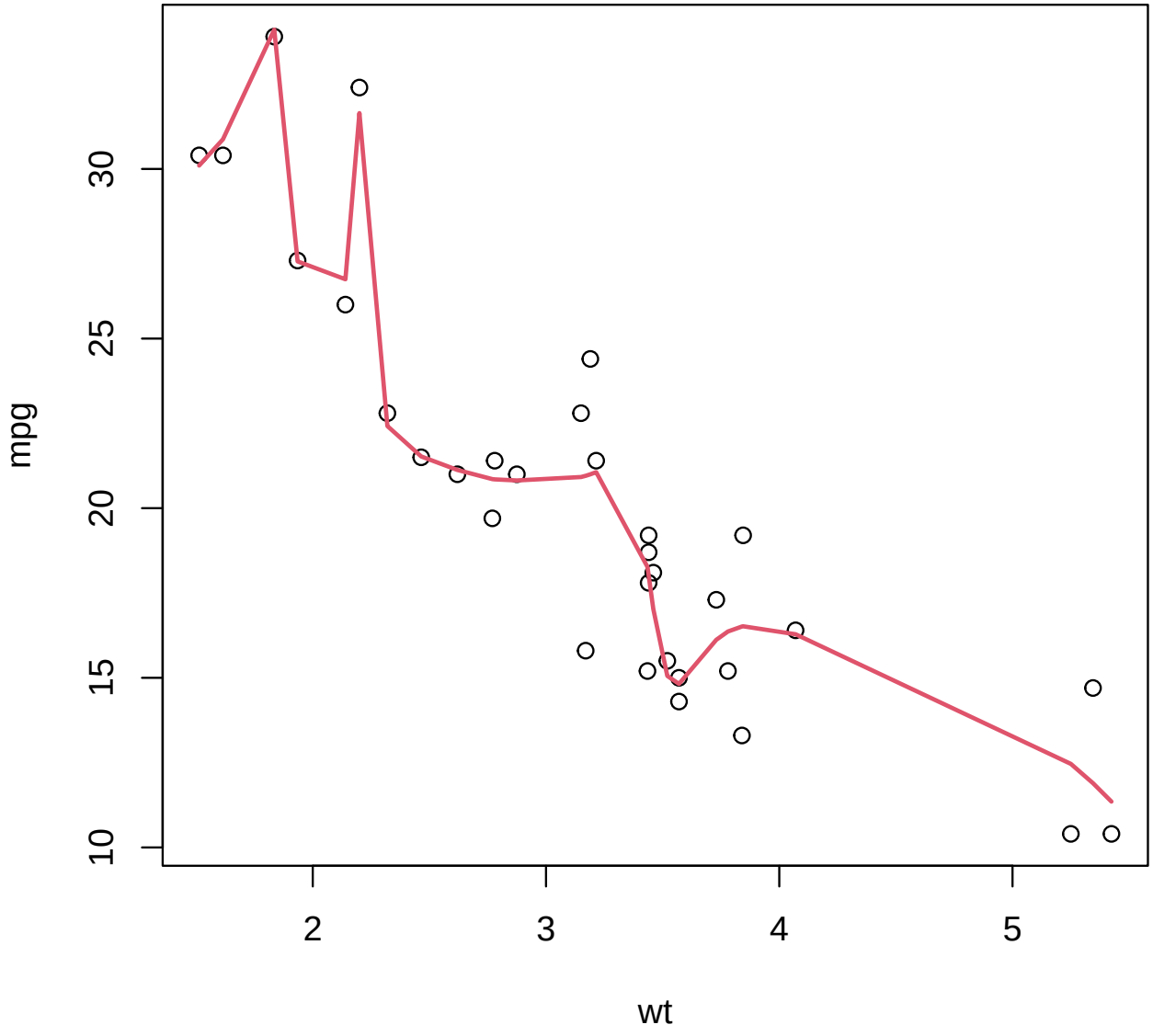
Regresión

```
> 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
```


mpg
2
1
0
−1
−1
0
1
2
wt

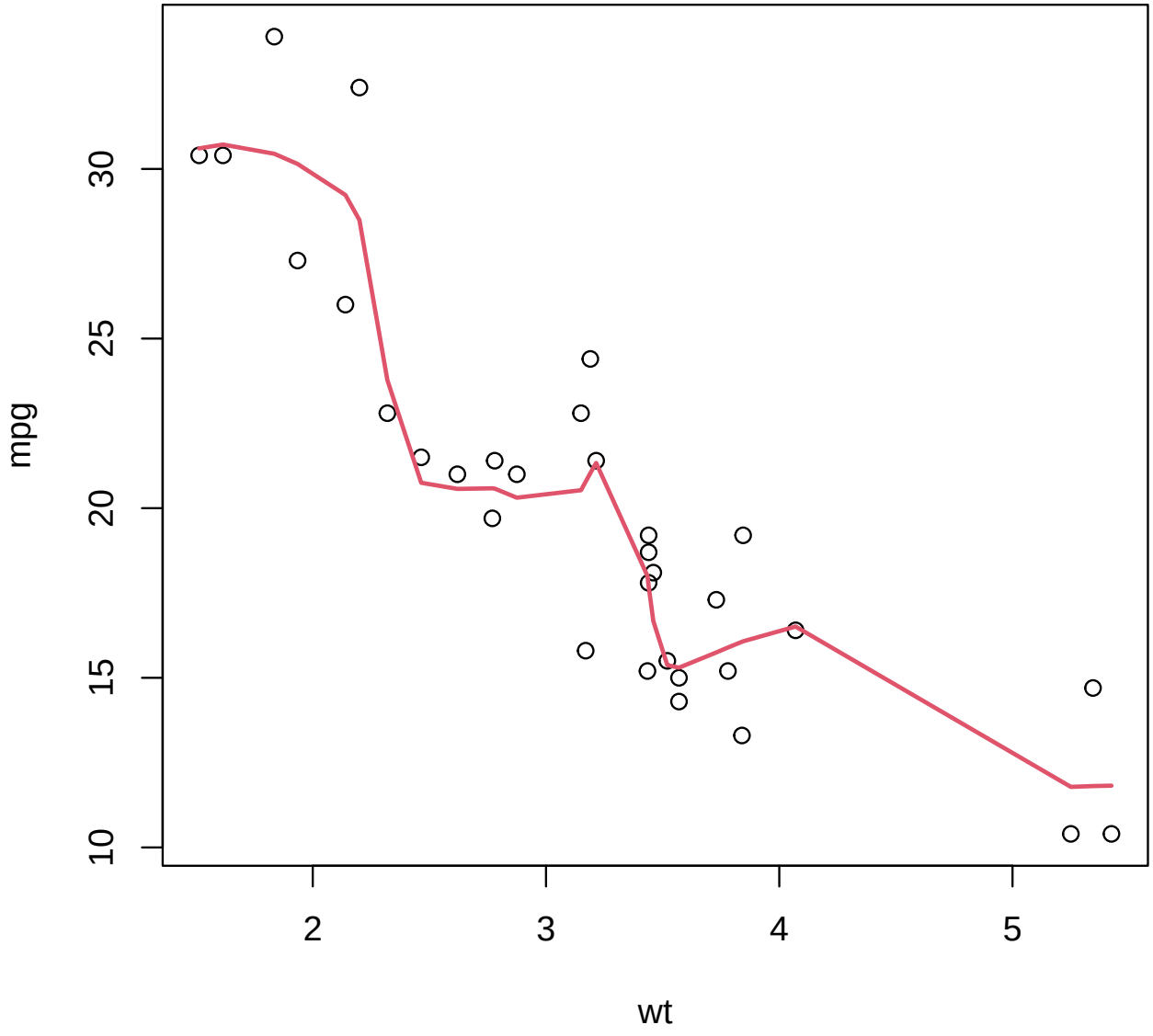
Regresión

```
> 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
```


mpg
10
15
20
25
30
2
3
4
5
wt

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
```

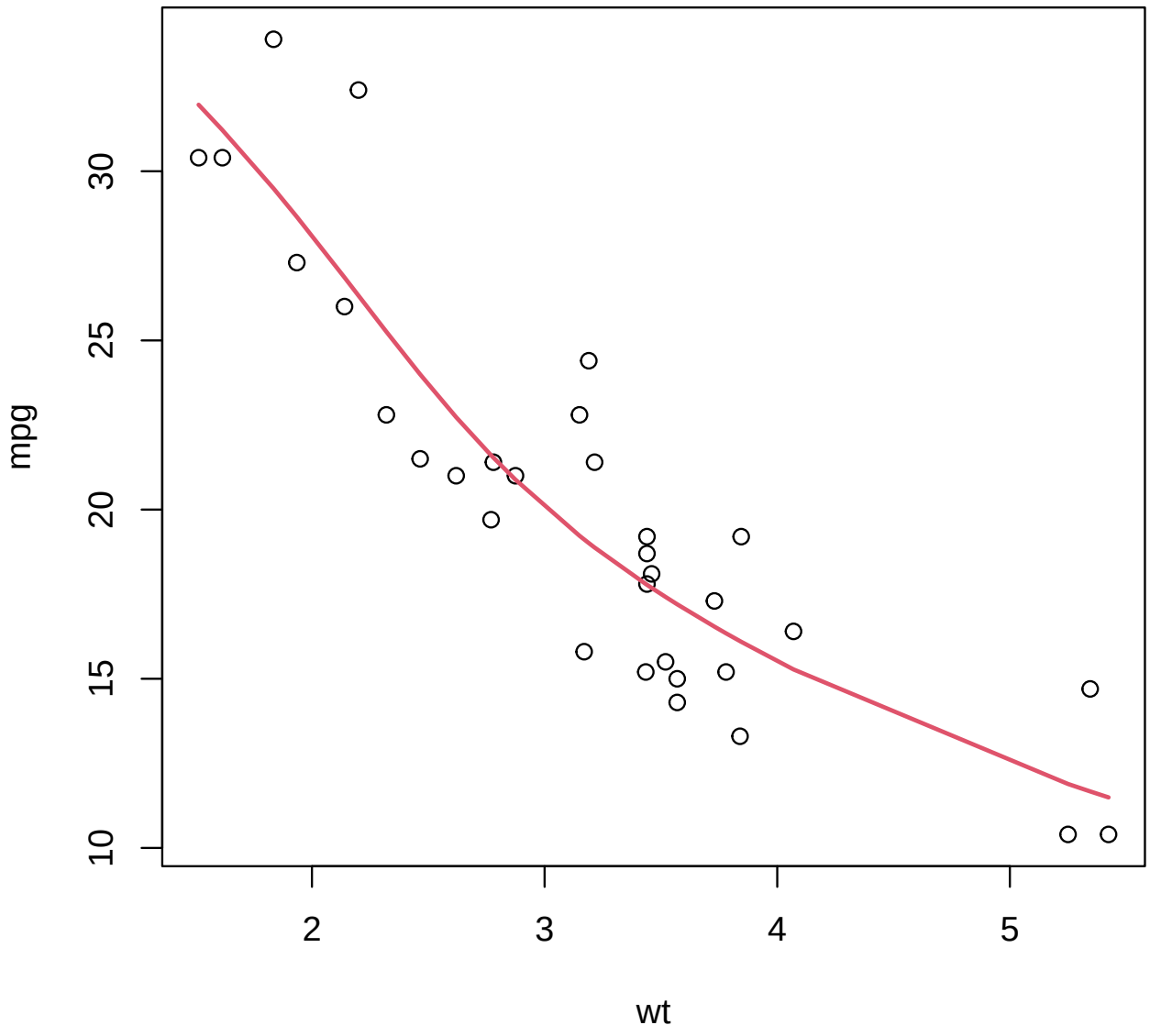

mpg
10
15
20
25
30
2
3
4
5
wt

Regresión

```
> set.seed(1) # decay evita sobreajuste pero no converge
> 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
```

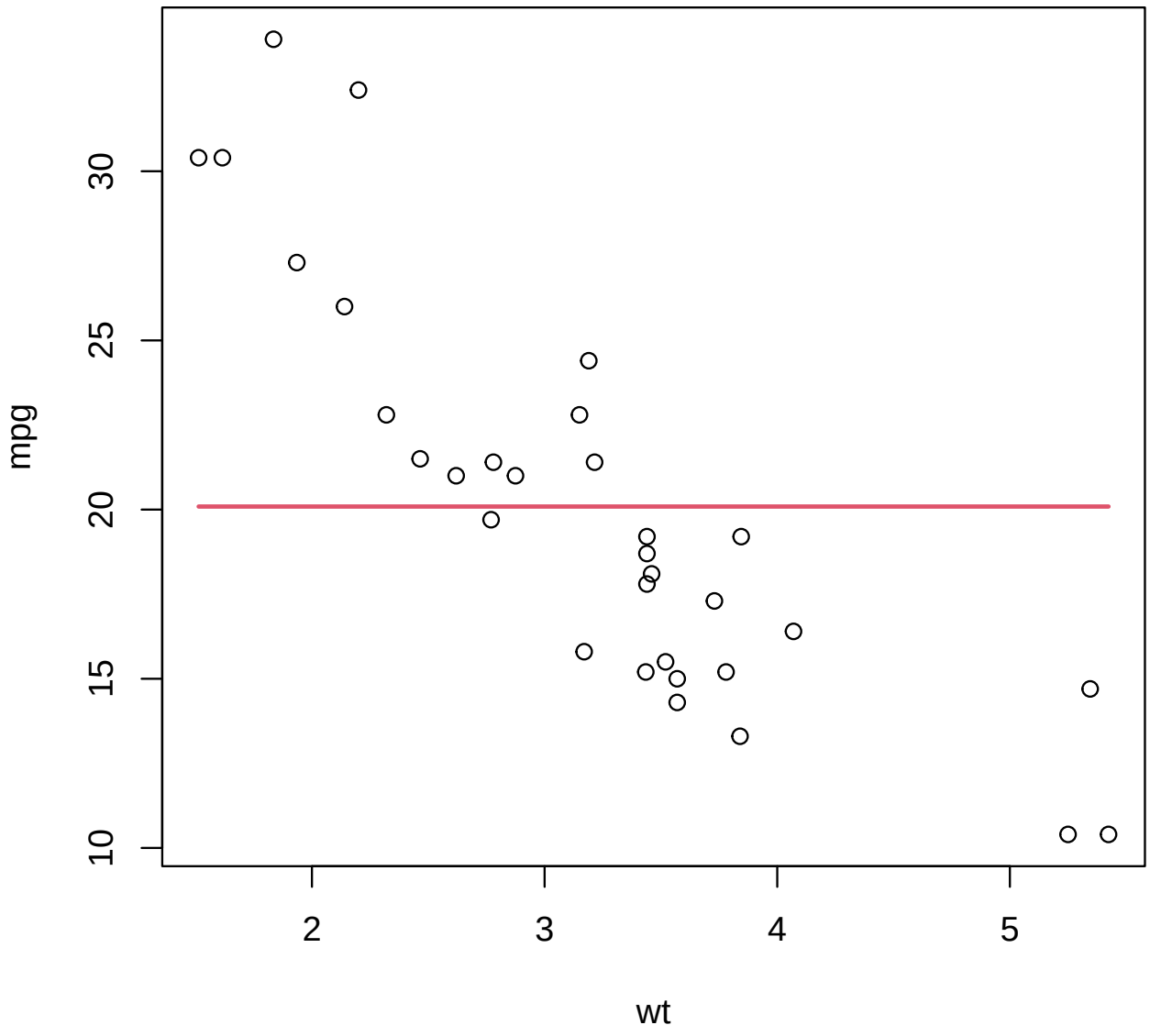
Regresión

```
> set.seed(1) # decay evita sobreajuste si aumentamos maxit
> salida <- capture.output (
+   red <- nnet (mpg~wt, mtcars, size = 100, linout = TRUE,
+                decay=.1, maxit=1000))
> tail (salida)
[1] "iter 660 value 213.940852"
[2] "iter 670 value 213.935819"
[3] "iter 680 value 213.932913"
[4] "iter 690 value 213.932841"
[5] "final  value 213.932819 "
[6] "converged"
```


mpg
10
15
20
25
30
2
3
4
5
wt

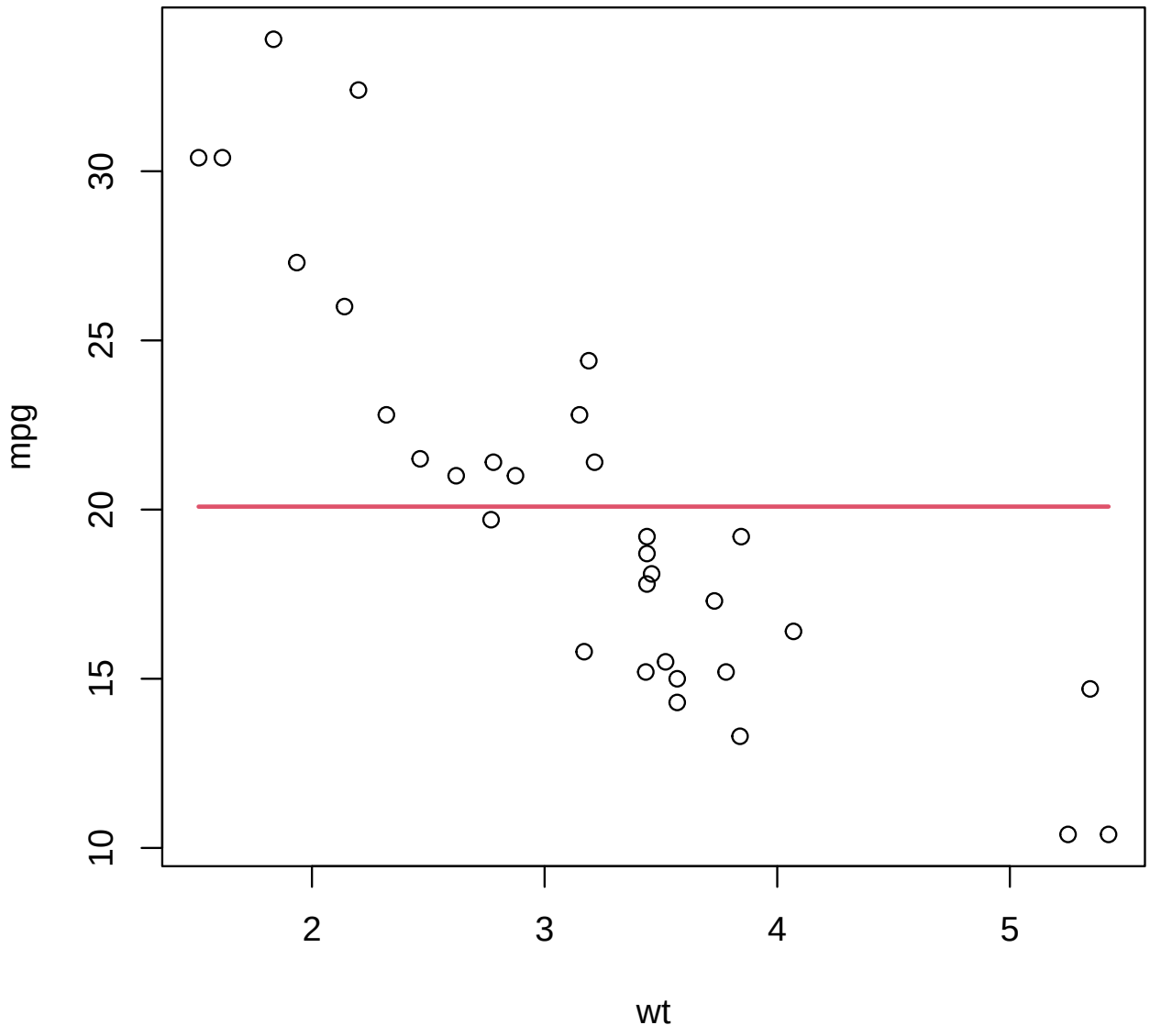
Regresión

```
> 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
```


mpg
10
15
20
25
30
2
3
4
5
wt

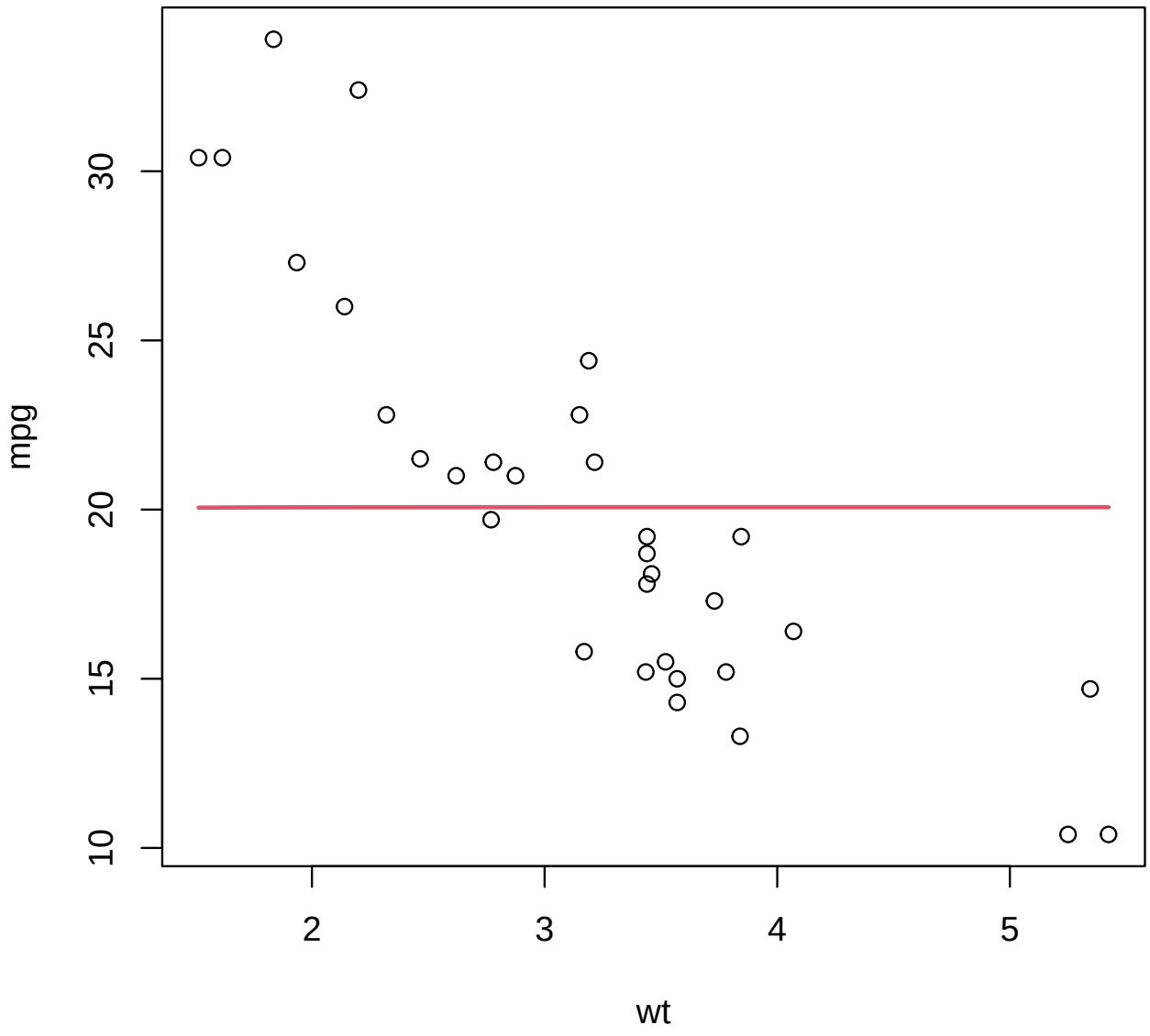
Regresión

```
> 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
```


mpg
10
15
20
25
30
2
3
4
5
wt

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
```


mpg
10
15
20
25
30
2
3
4
5
wt

Recomendaciones

- tipificar / normalizar / rescalar las variables
- ejecutar varias veces (probar distintas semillas)
- validación cruzada
- promediar las predicciones de varias redes (ensamblar) para mejorar la generalización

```
> 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 una instancia $n \in \{1, \ldots, N\}$:

$$y_k^{(n)} = \phi_o \left(\sum_{i=0}^{I} w_{ik}\, x_i^{(n)} \right)$$

Error (p.ej. divergencia en k respuestas dicótomas):

$$\begin{aligned} E &= \sum_n \sum_k \left[t_k^{(n)} \ln \frac{t_k^{(n)}}{y_k^{(n)}} + (1 - t_k^{(n)}) \ln \frac{1 - t_k^{(n)}}{1 - y_k^{(n)}} \right] \\ &= \sum_n \sum_k \left[-t_k^{(n)} \ln y_k^{(n)} - (1 - t_k^{(n)}) \ln(1 - y_k^{(n)}) \right] + \text{cte.} \end{aligned}$$

Objetivo: minimizar E respecto a w_{ik}

Perceptrón simple: derivada de E respecto a w_{ik}

$$y_k^{(n)} = \phi_o\left(\sum_{i=0}^{I} w_{ik} x_i^{(n)}\right) \qquad \phi_o(z) = \frac{1}{1+e^{-z}} \quad \text{(logística)}$$

$$E = \sum_n \sum_k \left[-t_k^{(n)} \ln y_k^{(n)} - (1-t_k^{(n)}) \ln(1-y_k^{(n)})\right]$$

$$\frac{\partial E}{\partial w_{ik}} = \sum_n \frac{\partial E^{(n)}}{\partial y_k^{(n)}} \frac{\partial y_k^{(n)}}{\partial a_k^{(n)}} \frac{\partial a_k^{(n)}}{\partial w_{ik}} \qquad a_k^{(n)} = \sum_i w_{ik} x_i^{(n)}$$

$$\frac{\partial E^{(n)}}{\partial y_k^{(n)}} = -\frac{t_k^{(n)}}{y_k^{(n)}} + \frac{1-t_k^{(n)}}{1-y_k^{(n)}} = \frac{y_k^{(n)} - t_k^{(n)}}{y_k^{(n)}(1-y_k^{(n)})}$$

$$\frac{\partial y_k^{(n)}}{\partial a_k^{(n)}} = \phi_o'(a_k^{(n)}) = y_k^{(n)}\left(1-y_k^{(n)}\right) \qquad \text{(derivada de la logística)}$$

$$\frac{\partial a_k^{(n)}}{\partial w_{ik}} = x_i^{(n)}$$

Perceptrón simple: derivada de E respecto a w_{ik}

Multiplicando los tres factores:

$$\frac{\partial E^{(n)}}{\partial w_{ik}} = \frac{y_k^{(n)} - t_k^{(n)}}{y_k^{(n)}(1 - y_k^{(n)})} \cdot y_k^{(n)}\left(1 - y_k^{(n)}\right) \cdot x_i^{(n)} = \left(y_k^{(n)} - t_k^{(n)}\right) x_i^{(n)}$$

Sumando para todas las instancias:

$$\frac{\partial E}{\partial w_{ik}} = \sum_{n=1}^{N} \left(y_k^{(n)} - t_k^{(n)}\right) x_i^{(n)}$$

Perceptrón simple: gradiente

Para activación logística:

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

Derivada del error:

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

Regla de actualización (descenso por gradiente):

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

$\eta > 0$ es la tasa de aprendizaje.

Perceptrón multicapa (1 oculta)

Capa oculta:

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

Capa de salida:

$$y_k^{(n)} = \phi_o\left(\sum_{j=0}^{J} v_{jk}\, h_j^{(n)}\right)$$

El error E es el mismo que antes.

Multicapa: retropropagación

Definimos el *error local* en salida:

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

Gradiente en la capa de salida:

$$\frac{\partial E}{\partial v_{jk}} = \sum_n \delta_k^{(n)} \, h_j^{(n)}$$

Actualización:

$$v_{jk} \leftarrow v_{jk} - \eta \sum_n \delta_k^{(n)} \, h_j^{(n)}$$

Multicapa: error en capa oculta

Error propagado a la neurona oculta:

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

Gradiente en pesos de entrada:

$$\frac{\partial E}{\partial w_{ij}} = \sum_n \delta_j^{(n)}\, x_i^{(n)}$$

Actualización:

$$w_{ij} \leftarrow w_{ij} - \eta \sum_n \delta_j^{(n)}\, x_i^{(n)}$$

Resumen: algoritmo de retropropagación

Para cada instancia n:

1. Propagación hacia delante: calcular $h_j^{(n)}$, luego $y_k^{(n)}$.
2. Calcular errores en salida: $\delta_k^{(n)}$.
3. Propagar errores hacia atrás: $\delta_j^{(n)}$.
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`)

Sean $E(\mathbf{w})$ la función de error y $\nabla E(\mathbf{w})$ su gradiente.
Sea t el índice de iteración.

1. Descenso por gradiente clásico

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

- $\eta > 0$ 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)

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

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 la SCE.
- 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:

$$\texttt{nnet} \quad \neq \quad \text{descenso por gradiente con tasa fija}$$

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:

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

3. Tomar $\alpha_t = \alpha$.

Propiedades:

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

Anexo: Hiperparámetros

```
> library(e1071) # para tune
> tune(lm, mpg~., data=mtcars, # CV para lm
+      tunecontrol=tune.control(cross=32))
Error estimation of 'lm' using leave-one-out: 12.18156
> library(rpart) # para rpart
> tune(rpart, mpg~., data=mtcars,
+      ranges=list(minsplit=c(5,10)))
Parameter tuning of 'rpart':

- sampling method: 10-fold cross validation

- best parameters:
 minsplit
        5

- best performance: 12.96652
```

```
> library(nnet) # para nnet
> tune(nnet, mpg~., data=mtcars,
+      linout=TRUE, trace=FALSE,
+      ranges=list(size=c(2,10), decay=.1^(1:2)))
Parameter tuning of ‘nnet’:

- sampling method: 10-fold cross validation

- best parameters:
 size decay
   10   0.1

- best performance: 9.599455
```

```
> library(caret, quietly=TRUE)
> salida <- capture.output(
+  res <- train(mpg ~ ., data=mtcars, method="nnet",
+           trControl = trainControl("repeatedcv",32),
+           tuneGrid = expand.grid(size=c(2,10),
+                                  decay=c(.1,.01))))
> names(res)
 [1] "method"       "modelInfo"    "modelType"
 [4] "results"      "pred"         "bestTune"
 [7] "call"         "dots"         "metric"
[10] "control"      "finalModel"   "preProcess"
[13] "trainingData" "ptype"        "resample"
[16] "resampledCM"  "perfNames"    "maximize"
[19] "yLimits"      "times"        "levels"
[22] "terms"        "coefnames"    "xlevels"
```

```
> res
Neural Network

32 samples
10 predictors

No pre-processing
Resampling: Cross-Validated (32 fold, repeated 1 times)
Summary of sample sizes: 31, 31, 31, 31, 31, 31, ...
Resampling results across tuning parameters:

  size  decay  RMSE      Rsquared  MAE
   2    0.01   19.09069  NaN       19.09069
   2    0.10   19.09106  NaN       19.09106
  10    0.01   19.09064  NaN       19.09064
  10    0.10   19.09076  NaN       19.09076

RMSE was used to select the optimal model using
 the smallest value.
The final values used for the model were size = 10
 and decay = 0.01.
```