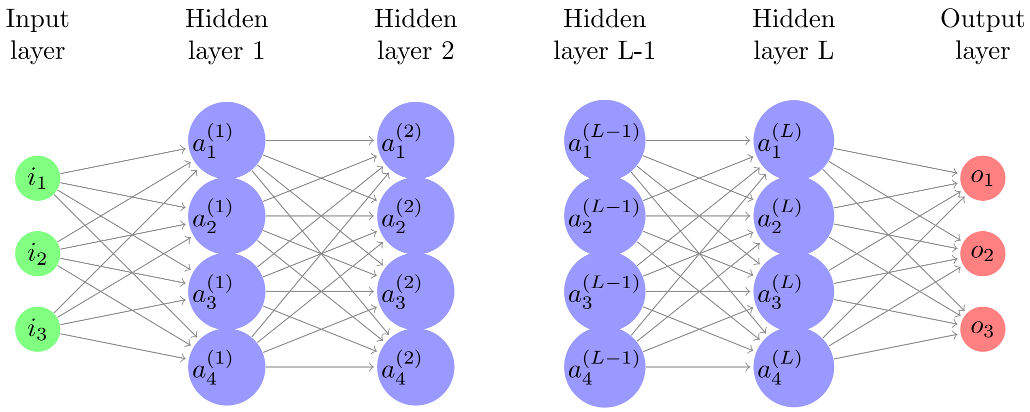
Neural Network

Andoni Latorre Galarraga

June 2020

1 Forward Propagation



Sea $a_i^{(J)} = \sigma(z_i^{(J)})$ el nodo número i de la capa J al cual le corresponden los weights:

$$\begin{pmatrix} w_{i1}^{(J)} \\ w_{i2}^{(J)} \\ \vdots \\ w_{in}^{(J)} \end{pmatrix}^T = \begin{pmatrix} w_{i1}^{(J)} & w_{i2}^{(J)} & \cdots & w_{in}^{(J)} \end{pmatrix}$$

Y el bias $b_i^{(J)}$. Tenemos:

$$z_i^{(J)} = \begin{pmatrix} w_{i1}^{(J)} & w_{i2}^{(J)} & \cdots & w_{in}^{(J)} \end{pmatrix} \begin{pmatrix} a_1^{(J-1)} \\ a_2^{(J-1)} \\ \vdots \\ a_n^{(J-1)} \end{pmatrix} + b_i^{(J)}$$

Si consideramos la capa J entera, de k nodos:

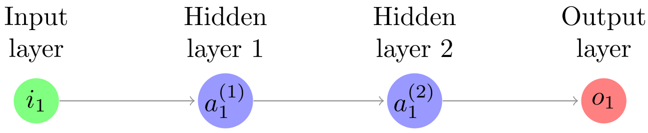
$$\begin{pmatrix} w_{11}^{(J)} & w_{12}^{(J)} & \cdots & w_{1n}^{(J)} \\ w_{21}^{(J)} & w_{22}^{(J)} & \cdots & w_{2n}^{(J)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k1}^{(J)} & w_{k2}^{(J)} & \cdots & w_{kn}^{(J)} \end{pmatrix} \begin{pmatrix} a_1^{(J-1)} \\ a_2^{(J-1)} \\ \vdots \\ a_n^{(J-1)} \end{pmatrix} + \begin{pmatrix} b_1^{(J)} \\ b_2^{(J)} \\ \vdots \\ b_k^{(J)} \end{pmatrix} = \begin{pmatrix} z_1^{(J)} \\ z_2^{(J)} \\ \vdots \\ z_n^{(J)} \end{pmatrix}$$

$$\sigma(\begin{pmatrix} w_{11}^{(J)} & w_{12}^{(J)} & \cdots & w_{1n}^{(J)} \\ w_{21}^{(J)} & w_{22}^{(J)} & \cdots & w_{2n}^{(J)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k1}^{(J)} & w_{k2}^{(J)} & \cdots & w_{kn}^{(J)} \end{pmatrix} \begin{pmatrix} a_1^{(J-1)} \\ a_2^{(J-1)} \\ \vdots \\ a_n^{(J-1)} \end{pmatrix} + \begin{pmatrix} b_1^{(J)} \\ b_2^{(J)} \\ \vdots \\ b_k^{(J)} \end{pmatrix}) = \sigma(\begin{pmatrix} z_1^{(J)} \\ z_2^{(J)} \\ \vdots \\ z_n^{(J)} \end{pmatrix}) = \begin{pmatrix} a_1^{(J)} \\ a_2^{(J)} \\ \vdots \\ a_n^{(J)} \end{pmatrix}$$

2 Back propagation

Tras incializar la red con weights y biases aleatorios la probamos con nuestros datos. Sean los inputs $I = \{I_1, I_2, \cdots, I_g\}$ tales que cada I_h contiene las entradas de la input layer, $I_h = \{i_1^h, i_2^h, \cdots, i_r^h\}$, la input layer tendria r entradas. Si la output layer tiene u salidas, los resultados esperados para cada I_h serian $\mathcal{O}_h = \{\mathcal{O}_1^h, \mathcal{O}_2^h, \cdots, \mathcal{O}_u^h\}$.

En una red simple:



Entonces,

$$o_1 = \sigma(z_i^{(3)}) = \sigma(w_{11}^{(3)} a_{11}^{(2)} + b_1^{(3)}) = \sigma(w_{11}^{(3)} \sigma(w_{11}^{(2)} a_{11}^{(1)} + b_1^{(2)}) + b_1^{(3)}) =$$

$$\sigma(w_{11}^{(3)} \sigma(w_{11}^{(2)} \sigma(w_{11}^{(1)} i_1^h + b_1^{(1)}) + b_1^{(2)}) + b_1^{(3)}) = \sigma(w_3 \sigma(w_2 \sigma(w_1 i_1^h + b_1) + b_2) + b_3)$$

Definimos la funcion coste para un I_h:

$$C_0^h(w_1, b_1, w_2, b_2, w_3, b_3) = (o_1 - \mathcal{O}_1^h)^2$$

L funcion coste:

$$C(w_1, b_1, w_2, b_2, w_3, b_3) = \frac{1}{g} \sum_{p=1}^{g} (o_1 - \mathcal{O}_1^p)^2$$

Como queremos minimizar C tendremos que mover los weights y los biases en la dirección de $-\nabla C$

$$\nabla C = \begin{pmatrix} \frac{\partial C}{\partial w_1} \\ \frac{\partial C}{\partial b_1} \\ \frac{\partial C}{\partial w_2} \\ \frac{\partial C}{\partial b_2} \\ \frac{\partial C}{\partial w_3} \\ \frac{\partial C}{\partial b_3} \end{pmatrix}$$

Los weights y los biases cambian de la siguiente manera:

$$\begin{pmatrix} w_1 \\ b_1 \\ w_2 \\ b_2 \\ w_3 \\ b_3 \end{pmatrix} \to \begin{pmatrix} w_1 \\ b_1 \\ w_2 \\ b_2 \\ w_3 \\ b_3 \end{pmatrix} - \mu \nabla C$$

En general:

$$\frac{\partial C}{\partial b_i^{(J)}} = \frac{\partial}{\partial b_i^{(J)}} \frac{1}{g} \sum_{p=1}^{g} C_o^p = \frac{1}{g} \sum_{p=1}^{g} \frac{\partial C_o^p}{\partial b_i^{(J)}} \quad , \quad \frac{\partial C}{\partial w_{ij}^{(J)}} = \frac{\partial}{\partial w_{ij}^{(J)}} \frac{1}{g} \sum_{p=1}^{g} C_o^p = \frac{1}{g} \sum_{p=1}^{g} \frac{\partial C_o^p}{\partial w_{ij}^{(J)}}$$

$$\frac{\partial C_o^p}{\partial b_j^{(J)p}} = \frac{C_0^p(a_i^{(J)p}(z_i^{(J)p}))}{b_i^{(J)p}} = \frac{\partial C_0^p}{\partial a_i^{(J)p}} \frac{\partial a_i^{(J)p}}{\partial z_i^{(J)p}} \frac{\partial z_i^{(J)p}}{\partial z_i^{(J)p}} \quad , \quad \frac{\partial C_o^p}{\partial w_{ij}^{(J)p}} = \frac{C_0^p(a_i^{(J)p}(z_i^{(J)p}))}{w_{ij}^{(L)p}} = \frac{\partial C_0^p}{\partial a_i^{(L)p}} \frac{\partial a_i^{(J)p}}{\partial z_i^{(J)p}} \frac{\partial z_i^{(L)p}}{\partial w_{ij}^{(J)p}}$$

$$\frac{\partial C_0^p}{\partial a_i^{(J)p}} \sigma'(z_i^{(J)p}) \quad , \quad \frac{\partial C_0^p}{\partial a_i^{(J)p}} \sigma'(z_i^{(J)p}) a_j^{(J-1)p}$$

$$\frac{\partial C_0^p}{\partial a_i^{(Y)p}} = \sum_{k=1}^{n_{Y-1}} \frac{\partial C_0^p}{\partial a_k^{(Y+1)p}} \sigma'(z_k^{(Y+1)p}) w_{jk}^{(Y+1)p} \quad Y \leq L \quad , \quad \frac{\partial C_0}{\partial a_i^{(L+1)}} = 2(a_i^{(L+1)p} - \mathcal{O}_i^p)$$

3 Implementación en Python

La input y output layer tienen indices 0 y L+1 respectivamente.

$$z_i^{(J)} = d[J]["z"][i]$$
$$a_i^{(J)} = d[J]["a"][i]$$
$$b_i^{(J)} = d[J]["b"][i]$$
$$w_{ik}^{(J)} = d[J]["w"][i][k]$$

$$d = [\underbrace{\{"a" : [i_0, \cdots]\}}_{0}, \underbrace{\{\cdots\}}_{1}, \cdots, \underbrace{\{"z" : [z_0^{(J)}, \cdots], "a" : [a_0^{(J)}, \cdots], "b" : [b_0^{(J)}, \cdots], "w" : [\underbrace{[w_{00}^{(J)}, w_{01}^{(J)}]}_{0}, \cdots\}}_{J}, \cdots, \underbrace{\{\cdots\}}_{L}, \underbrace{\{\cdots\}}_{L+1}]$$

```python
import random
import math
import time

#objetos algebra
class Matrix:

    def __init__(self, M):
        #k x n
        #[[11,12,13,...,1n]
        #,[21,22,23,...,2n]
        #,...
        #,[k1,k2,k3,...,kn]]
        self.__m = M
        self.__k = len(M)
        self.__n =  len(M[0])

    def transpose(self):
        M = [[None for k in range(self.__k)] for n in range(self.__n)]
        for i in range(self.__k):
            for j in range(self.__n):
                M[j][i] = self.__m[i][j]
        self.__k, self.__n = self.__n, self.__k
        self.__m = M

    def __mul__(self, other):
        if self.__n != other.__k:
            raise ValueError
        M = []
        for i in range(self.__k):
            M.append([])
            for j in range(other.__n):
                s = 0
                for k in range(self.__n):
                    s += self.__m[i][k]*other.__m[k][j]
                M[i].append(s)
        return Matrix(M)

    def __add__(self, other):
        if self.__n != other.__n or self.__k != other.__k:
            raise ValueError
```

```python
        M = []
        for i in range(self.__k):
            M.append([])
            for j in range(self.__n):
                M[i].append(self.__m[i][j]+other.__m[i][j])
        return Matrix(M)

    def __repr__(self):
        return "Matrix(" + str(self.__m) + ")"

    def __str__(self):
        s = ""
        fila = -1
        for fila in range(self.__k - 1):
            s += str(self.__m[fila]) + "\n"
        return s + str(self.__m[fila+1])

    def __getitem__(self, index):
        if self.__n == 1:
            return self.__m[index][0]
        return self.__m[index]

    def __setitem__(self, index, item):
        if self.__n == 1:
            self.__m[index][0] = item
            return
        self.__m[index] = item

    def __len__(self):
        return self.__k

    def sig(self):
        return Matrix(sigmoid(self.__m))



#funcion sigmoid para convertir de R a (0,1) 1/1+e^-x
def sigmoid(x):
    try:
        if type(x) == type(list()):
            return [sigmoid(xx) for xx in x]
        return 1/(1+math.exp(-x))
    except OverflowError:
        if x < 0:
            return 0
        return 1

def sigmoid_d(x):
    e = math.exp(-x)
    try:
        return e/((1+e)**2)
    except OverflowError:
        return 0

#funcion ReLU
```

```
def ReLU(x):
    return max(0,x)

#Clase de la red
class Neural_Network:

    def __init__(self, capas, mu):
        self.__capas = capas
        self.__mu = mu
        self.__L = len(capas)-2
        self.__d = []
        d = d = {"z":None, "a":None, "b":None, "w":None}
        d["a"]=Matrix([[None for m in range(capas[0])]])
        d["a"].transpose()
        self.__d.append(d)
        for r in range(1, self.__L+2):
            d = {"z":None, "a":None, "b":None, "w":None}
            d["z"]=Matrix([[None for m in range(capas[r])]])
            d["z"].transpose()
            d["a"]=Matrix([[None for m in range(capas[r])]])
            d["a"].transpose()
            d["b"]=Matrix([[0.0 for m in range(capas[r])]])
            d["b"].transpose()
            d["w"]=Matrix([[0.0 for mm in range(capas[r-1])] for m in range(capas[r])])
            self.__d.append(d)

    def reset_d(self):
        D = []
        d = d = {"z":None, "a":None, "b":None, "w":None}
        d["a"]=Matrix([[None for m in range(len(self.__d[0]["a"]))]])
        d["a"].transpose()
        D.append(d)
        for r in range(1, self.__L+2):
            d = d = {"z":None, "a":None, "b":None, "w":None}
            d["z"]=Matrix([[None for m in range(self.__capas[r])]])
            d["z"].transpose()
            d["a"]=Matrix([[None for m in range(self.__capas[r])]])
            d["a"].transpose()
            d["b"]=self.__d[r]["b"]
            d["w"]=self.__d[r]["w"]
            D.append(d)
        self.__d = D

    def forward(self):
        for J in range(1, self.__L+2):
            self.__d[J]["z"] = (self.__d[J]["w"]*self.__d[J-1]["a"])+self.__d[J]["b"]
            self.__d[J]["a"] = self.__d[J]["z"].sig()
        self.__p.append(self.__d.copy())
        self.reset_d()

    def back(self):
        self.__grad = dict()
        for J in range(1, self.__L+2):
            for i in range(len(self.__d[J]["b"])):
                #print(self.__d[J]["b"][i], self.b(J, i))
```

```
                self.__d[J]["b"][i] = self.__d[J]["b"][i] - (self.__mu*self.b(J, i))
            for i in range(len(self.__d[J]["w"])):
                for j in range(len(self.__d[J]["w"][i])):
                    #print(self.__d[J]["w"][i][j], self.w(J, i, j))
                    self.__d[J]["w"][i][j] = self.__d[J]["w"][i][j] - (self.__mu*self.w(

    def b(self, J, i):
        s = 0
        for p in range(self.__g):
            s += self.a(p, J , i)*sigmoid_d(self.__p[p][J]["z"][i])
        return s/self.__g

    def w(self, J, i, j):
        s = 0
        for p in range(self.__g):
            s += self.a(p, J-1 , j)*sigmoid_d(self.__p[p][J]["z"][i])*self.__p[p][J-1]["
        return s/self.__g

    def a(self, p, Y, i):
        if (p, Y, i) in self.__grad:
            return self.__grad[(p, Y, i)]
        if Y <= self.__L:
            s = 0
            for k in range(self.__capas[Y+1]):
                s += sigmoid_d(self.__p[p][Y+1]["z"][k])*self.__p[p][Y+1]["w"][k][i]*sel
        else:
            s = 2*(self.__p[p][Y]["a"][i]- self.__O[p][i])
        self.__grad[(p, Y, i)] = s
        return s

    def generation(self, inputs, outputs):
        self.__d[0]["a"] = Matrix([inputs])
        self.__d[0]["a"].transpose()
        self.__o = outputs
        self.__O.append(self.__o)
        self.forward()


    def train(self, INPUTS, OUTPUTS):
        self.__g = len(INPUTS)
        self.__O = []
        self.__p = list()
        for wea in range(len(INPUTS)):
            self.generation(INPUTS[wea], OUTPUTS[wea])
        self.back()

    def compute(self, inputs):
        self.__d[0]["a"] = Matrix([inputs])
        self.__d[0]["a"].transpose()
        for J in range(1, self.__L+2):
            self.__d[J]["z"] = (self.__d[J]["w"]*self.__d[J-1]["a"])+self.__d[J]["b"]
            self.__d[J]["a"] = self.__d[J]["z"].sig()
        return self.__d[self.__L+1]["a"]

fast = True
```

```python
Precision = False
kkk = 10**5 #precision
kk = 10**2 #entrenamiento
k = 10**15 #generaciones
n = Neural_Network([4, 4, 2], 1) #red

IN, OUT = [], []

for aew in range(kk):
    a, b, c, d = random.random(), random.random(), random.random(), random.random()
    IN.append([a, b, c, d])
    o = [1, 0]
    if a*d-b*c < 0:
        o = [0,1]
    OUT.append(o)

def check(mat):
    if mat[0] < mat[1]:
        return [0, 1]
    return [1, 0]

print("0 %")
t0 = time.process_time()
for gen in range(k):
    if Precision:
        aciertos = 0
        IN, OUT = [], []
        for aew in range(kkk):
            a, b, c, d = random.random(), random.random(), random.random(), random.rando
            IN.append([a, b, c, d])
            o = [1, 0]
            if a*d-b*c < 0:
                o = [0,1]
            OUT.append(o)
        for aew in range(kkk):
            if check(n.compute(IN[aew])) == OUT[aew]:
                aciertos += 1
        n.train(IN, OUT)
        print("precisi n:", 100*aciertos/kkk, "%")
    if not fast:
        print(100*round(((gen+1)/k),len(str(k))),"%", round((((time.process_time()-t0))*


aciertos = 0
IN, OUT = [], []
for aew in range(kkk):
    a, b, c, d = random.random(), random.random(), random.random(), random.random()
    IN.append([a, b, c, d])
    o = [1, 0]
    if a*d-b*c < 0:
        o = [0,1]
    OUT.append(o)

for aew in range(kkk):
    if check(n.compute(IN[aew])) == OUT[aew]:
```

```
        aciertos += 1

print("precisi n:", 100*aciertos/kkk, "%")
```

4 Notas

$\sigma(x) = \frac{1}{1+e^{-x}}$
$ReLU(x) = \max(0, x)$
Número de nodos en la capa l, n_l

5 Bilbliografía

[1] *Serie Neural Networks 3b1b*