

Informe de trabajo Práctico N°3: Machine Learning

Olguin Nahuel

Molina Matías

Maccapa Luis

Ejercicios obligatorios

1. Familiarizarse con el código de la red neuronal feedforward fully connected de 1 capa oculta explicada en clase

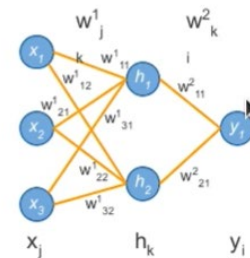
Solo procedemos a explicar las partes más importantes del código. Recordamos la implementación matricial:

● Implementación Matricial

- m ejemplos
- n entradas
- p neuronas ocultas
- q salidas

$$H = f(XW^1 + B^1) = f(Z)$$

$$XW^1 + B^1 = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1n} \\ x_{21} & x_{22} & \dots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m1} & x_{m2} & \dots & x_{mn} \end{bmatrix} \begin{bmatrix} w_{11}^1 & w_{12}^1 & \dots & w_{1p}^1 \\ w_{21}^1 & w_{22}^1 & \dots & w_{2p}^1 \\ \vdots & \vdots & \ddots & \vdots \\ w_{n1}^1 & w_{n2}^1 & \dots & w_{np}^1 \end{bmatrix} + \begin{bmatrix} b_1^1 & b_2^1 & \dots & b_p^1 \\ b_1^1 & b_2^1 & \dots & b_p^1 \\ \vdots & \vdots & \ddots & \vdots \\ b_1^1 & b_2^1 & \dots & b_p^1 \end{bmatrix}$$



Primero asignamos el número de clases, el número de ejemplos y generamos la matriz de entradas "X" y la matriz "t" de salidas deseadas:

```

iniciar(numero_clases=3, numero_ejemplos=300, graficar_datos=True)

# Entradas: 2 columnas (x1 y x2)
x = np.zeros((cantidad_ejemplos, 2))
# Salida deseada ("target"): 1 columna que contendra la clase correspondiente (codificada como un entero)
t = np.zeros(cantidad_ejemplos, dtype="uint8") # 1 columna: la clase correspondiente (t -> "target")

# Generamos las "entradas", los valores de las variables independientes. Las variables:
# radios, angulos e indices tienen n elementos cada una, por lo que le estamos agregando
# tambien n elementos a la variable x (que incorpora ambas entradas, x1 y x2)
x1 = radios * np.sin(angulos)
x2 = radios * np.cos(angulos)
x[indices] = np.c_[x1, x2]

# Guardamos el valor de la clase que le vamos a asociar a las entradas x1 y x2 que acabamos
# de generar
t[indices] = clase
  
```

Hasta ahora sabemos el número de ejemplos y entradas son:

$$m = 100, \quad n = 2$$

Quedando una matriz de entradas como la siguiente:

$$X_{100 \times 2} = \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \\ \vdots & \vdots \\ x_{100\ 1} & x_{100\ 2} \end{bmatrix}$$

Ahora procedemos a inicializar los pesos y sesgos de la red, asignando el número de neuronas de la capa oculta

```
# Inicializa pesos de la red
NEURONAS_CAPA_OCULTA = 100
NEURONAS_ENTRADA = 2
pesos = inicializar_pesos(n_entrada=NEURONAS_ENTRADA, n_capa_2=NEURONAS_CAPA_OCULTA, n_capa_3=numero_clases)
```

```
def inicializar_pesos(n_entrada, n_capa_2, n_capa_3):
    randomgen = np.random.default_rng()

    w1 = 0.1 * randomgen.standard_normal((n_entrada, n_capa_2))
    b1 = 0.1 * randomgen.standard_normal((1, n_capa_2))

    w2 = 0.1 * randomgen.standard_normal((n_capa_2, n_capa_3))
    b2 = 0.1 * randomgen.standard_normal((1, n_capa_3))

    return {"w1": w1, "b1": b1, "w2": w2, "b2": b2}
```

Por lo que el número de neuronas ocultas es:

$$p = 100$$

Ahora que hemos inicializado todas las matrices comenzamos con el entrenamiento de la red asignando las Epochs y el ritmo de aprendizaje “ ϵ ”

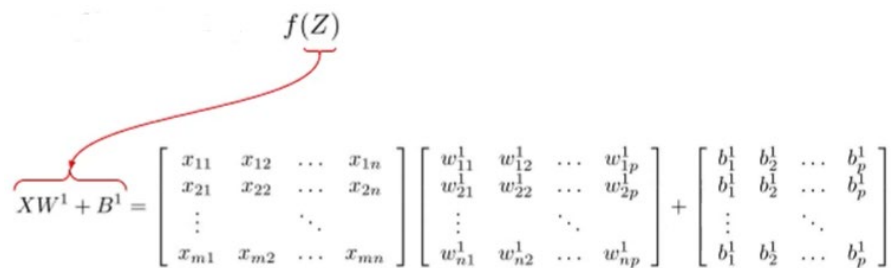
```
# Entrena
LEARNING_RATE=1
EPOCHS=10000
train(x, t, pesos, LEARNING_RATE, EPOCHS)
```

Para cada uno de los Epochs realizamos las operaciones dadas por la función “ejecutar_adelante”

```
def train(x, t, pesos, learning_rate, epochs):
    # Cantidad de filas (i.e. cantidad de ejemplos)
    m = np.size(x, 0)

    for i in range(epochs):
        # Ejecucion de la red hacia adelante
        resultados_feed_forward = ejecutar_adelante(x, pesos)
```

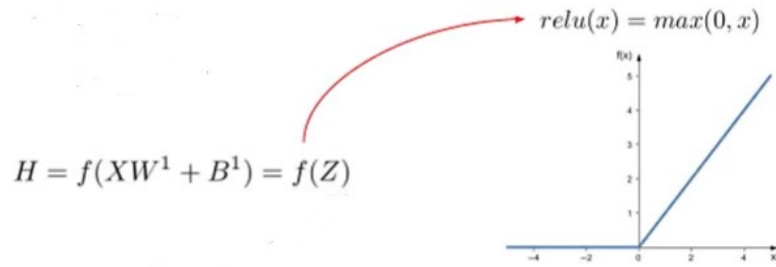
Primero se realiza el cálculo de “Z”



$$XW^1 + B^1 = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1n} \\ x_{21} & x_{22} & \dots & x_{2n} \\ \vdots & & \ddots & \\ x_{m1} & x_{m2} & \dots & x_{mn} \end{bmatrix} \begin{bmatrix} w_{11}^1 & w_{12}^1 & \dots & w_{1p}^1 \\ w_{21}^1 & w_{22}^1 & \dots & w_{2p}^1 \\ \vdots & & \ddots & \\ w_{n1}^1 & w_{n2}^1 & \dots & w_{np}^1 \end{bmatrix} + \begin{bmatrix} b_1^1 & b_2^1 & \dots & b_p^1 \\ b_1^1 & b_2^1 & \dots & b_p^1 \\ \vdots & & \ddots & \\ b_1^1 & b_2^1 & \dots & b_p^1 \end{bmatrix}$$

```
def ejecutar_adelante(x, pesos):
    # Funcion de entrada (a.k.a. "regla de propagacion") para la primera capa oculta
    z = x.dot(pesos["w1"]) + pesos["b1"]
```

Luego se obtiene “H” aplicando como función la “relu(x)”:



```
# Funcion de activacion ReLU para la capa oculta (h -> "hidden")
h = np.maximum(0, z)
```

Por último, obtenemos la matriz de salida de la red “Y” y retornamos las 3 matrices obtenidas

$$Y = g(HW^2 + B^2) \rightarrow g(x) = x \quad (\text{función identidad})$$

```
# Salida de la red (funcion de activacion lineal). Esto incluye la salida de todas
# las neuronas y para todos los ejemplos proporcionados
y = h.dot(pesos["w2"]) + pesos["b2"]

return {"z": z, "h": h, "y": y}
```

Ahora procedemos a calcular la “funcion de pérdida”:

$$L = \frac{1}{m} \sum_m -\log \left(\frac{e^{y_c^m}}{\sum_j e^{y_j^m}} \right)$$

```
# LOSS
# a. Exponencial de todos los scores
exp_scores = np.exp(y)

# b. Suma de todos los exponenciales de los scores, fila por fila (ejemplo por ejemplo).
# Mantenemos las dimensiones (indicamos a NumPy que mantenga la segunda dimension del
# arreglo, aunque sea una sola columna, para permitir el broadcast correcto en operaciones
# subsiguientes)
sum_exp_scores = np.sum(exp_scores, axis=1, keepdims=True)

# c. "Probabilidades": normalizacion de las exponenciales del score de cada clase (dividiendo por
# la suma de exponenciales de todos los scores), fila por fila
p = exp_scores / sum_exp_scores

# d. Calculo de la funcion de perdida global. Solo se usa la probabilidad de la clase correcta,
# que tomamos del array t ("target")
loss = (1 / m) * np.sum( -np.log( p[range(m), t] ))
```

Procedemos a hacer el aprendizaje mediante “Backpropagation”:

$$\frac{\partial L}{\partial W^2} = \frac{\partial L}{\partial Y} \frac{\partial Y}{\partial W^2}$$

$$\frac{\partial L}{\partial Y} = \begin{cases} \frac{1}{m}(P_i - 1) & \text{(clase correcta)} \\ \frac{1}{m}(P_i) & \text{(otras clases)} \end{cases}$$

$$\frac{\partial Y}{\partial W^2} = H$$

$$L = \frac{1}{m} \sum_m -\log \left(\frac{e^{y_c^m}}{\sum_j e^{y_j^m}} \right)$$

$$P_i = \frac{e^{y_i}}{\sum_j e^{y_j}}$$

```
# Ajustamos los pesos: Backpropagation
dL_dy = p # Para todas las salidas, L' = p (la probabilidad)...
dL_dy[range(m), t] -= 1 # ... excepto para la clase correcta
dL_dy /= m

dL_dw2 = h.T.dot(dL_dy) # Ajuste para w2
```

$$\frac{\partial L}{\partial B^2} = \frac{\partial L}{\partial Y} \frac{\partial Y}{\partial B^2}$$

$$\frac{\partial L}{\partial Y} = \begin{cases} \frac{1}{m}(P_i - 1) & \text{(clase correcta)} \\ \frac{1}{m}(P_i) & \text{(otras clases)} \end{cases}$$

$$\frac{\partial Y}{\partial B^2} = 1$$

(se suma verticalmente a lo largo de m)

$$L = \frac{1}{m} \sum_m -\log \left(\frac{e^{y_c^m}}{\sum_j e^{y_j^m}} \right)$$

$$P_i = \frac{e^{y_i}}{\sum_j e^{y_j}}$$

```
dL_db2 = np.sum(dL_dy, axis=0, keepdims=True) # Ajuste para b2
```

$$\frac{\partial L}{\partial W^1} = \frac{\partial L}{\partial Y} \frac{\partial Y}{\partial H} \frac{\partial H}{\partial Z} \frac{\partial Z}{\partial W^1}$$

$$\frac{\partial Y}{\partial H} = W^2$$

$$Y = HW^2 + B^2$$

$$\frac{\partial Z}{\partial W^1} = X$$

$$Z = XW^1 + B^1$$

$$\frac{\partial H}{\partial Z} = 1(z > 0)$$

```
dL_dh = dL_dy.dot(w2.T)
```

```
dL_dz = dL_dh      # El calculo dL/dz = dL/dh * dh/dz. La funcion "h" es la funcion de activacion de la capa
dL_dz[z <= 0] = 0    # para la que usamos ReLU. La derivada de la funcion ReLU: 1(z > 0) (0 en otro caso)
```

```
dL_dw1 = x.T.dot(dL_dz)
```

```
# Ajuste para w1
```

$$\frac{\partial L}{\partial B^1} = \frac{\partial L}{\partial Y} \frac{\partial Y}{\partial H} \frac{\partial H}{\partial Z} \frac{\partial Z}{\partial B^1}$$

$$\frac{\partial Y}{\partial H} = W^2$$

$$Y = HW^2 + B^2$$

$$\frac{\partial Z}{\partial B^1} = 1$$

$$Z = XW^1 + B^1$$

(se suma verticalmente a lo largo de m)

$$\frac{\partial H}{\partial Z} = 1(z > 0)$$

```
dL_db1 = np.sum(dL_dz, axis=0, keepdims=True) # Ajuste para b1
```

Ahora aplicamos los ajustes calculados para ajustar los pesos y sesgos

$$W_{t+1} = W_t - \varepsilon \nabla L_W$$

$$\left\{ \begin{array}{l} W^2(t+1) = W^2(t) - \varepsilon \frac{\partial L}{\partial W^2} \\ B^2(t+1) = B^2(t) - \varepsilon \frac{\partial L}{\partial B^2} \\ W^1(t+1) = W^1(t) - \varepsilon \frac{\partial L}{\partial W^1} \\ B^1(t+1) = B^1(t) - \varepsilon \frac{\partial L}{\partial B^1} \end{array} \right.$$

```
# Aplicamos el ajuste a los pesos
w1 += -learning_rate * dL_dw1
b1 += -learning_rate * dL_db1
w2 += -learning_rate * dL_dw2
b2 += -learning_rate * dL_db2
```

2. Modificar el programa para que:

a. Mida la precisión de clasificación (accuracy) además del valor de Loss

```
#Calculo de la presicion
Vect_clas=clasificar(x,pesos)
aciertos=0
for j in range(300):
    if t[j]==Vect_clas[j]:
        aciertos=aciertos+1
precision=aciertos/len(Vect_clas)

# Mostramos solo cada 1000 epochs
if i %1000 == 0:
    print("Epoch", i, ":", "Loss->", loss, "Precision->", precision)
```

```
Epoch 0 : Loss-> 1.1147846407187567 Precision-> 0.25666666666666665
Epoch 1000 : Loss-> 0.16112415360306517 Precision-> 0.93
Epoch 2000 : Loss-> 0.13231137461826392 Precision-> 0.95
Epoch 3000 : Loss-> 0.11021675066710522 Precision-> 0.9666666666666667
Epoch 4000 : Loss-> 0.09814796379291921 Precision-> 0.9766666666666667
Epoch 5000 : Loss-> 0.08956393165387605 Precision-> 0.97
Epoch 6000 : Loss-> 0.08620040016288279 Precision-> 0.9766666666666667
Epoch 7000 : Loss-> 0.0856217481868031 Precision-> 0.98
Epoch 8000 : Loss-> 0.07975100576958083 Precision-> 0.9833333333333333
Epoch 9000 : Loss-> 0.07711929568827858 Precision-> 0.9833333333333333
```

b. Utilice un conjunto de test independiente para realizar dicha medición (en lugar de utilizar los mismos datos de entrenamiento). Este punto requiere generar más ejemplos.

```
# Test
print("Comienza el test")
x_test, t_test = generar_datos_clasificacion(numero_ejemplos, numero_clases)
Vect_clas=clasificar(x_test, pesos)
aciertos=0
for j in range(len(Vect_clas)):
    if t_test[j]==Vect_clas[j]:
        aciertos=aciertos+1
precision=aciertos/len(Vect_clas)
print("Precision del test:", precision)
```

```
Epoch 0 : Loss-> 1.0852740774232397 Precision-> 0.5133333333333333
Epoch 1000 : Loss-> 0.09835515961092292 Precision-> 0.97
Epoch 2000 : Loss-> 0.08476865494485125 Precision-> 0.97
Epoch 3000 : Loss-> 0.0809571341429591 Precision-> 0.97
Epoch 4000 : Loss-> 0.07822010774189661 Precision-> 0.9733333333333334
Epoch 5000 : Loss-> 0.07562281874595224 Precision-> 0.9733333333333334
Epoch 6000 : Loss-> 0.07375553619425415 Precision-> 0.9733333333333334
Epoch 7000 : Loss-> 0.07217347424342044 Precision-> 0.9733333333333334
Epoch 8000 : Loss-> 0.07141994479966823 Precision-> 0.9766666666666667
Epoch 9000 : Loss-> 0.07037754486515489 Precision-> 0.9733333333333334
Comienza el test
Precision del test: 0.97
```

3. Agregar parada temprana, utilizando un conjunto de validación, distinto del conjunto de entrenamiento y de test (este punto requiere generar más ejemplos). Esto es: verificar el valor de loss o de accuracy cada N epochs (donde N es un parámetro de configuración) utilizando el conjunto de validación, y detener el entrenamiento en caso de que estos valores hayan empeorado (puede incluirse una tolerancia para evitar cortar el entrenamiento por alguna oscilación propia del proceso de entrenamiento).

```
#Generamos el conjunto de validacion para aplicar una parada temprana
x_validacion, t_validacion = generar_datos_clasificacion(300, 3)
precision_validacion=0
```

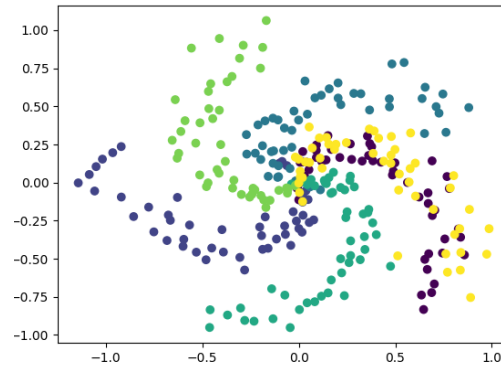
```
# Mostramos solo cada 1000 epochs y verificamos el conjunto de validacion para parada temprana
if i %1000 == 0:
    print("Epoch", i,":","Loss->",loss,"Precision->",precision)
    vect_valid=clasificar(x_validacion,pesos)
    aciertos=0
    for j in range(len(vect_valid)):
        if t_validacion[j]==vect_valid[j]:
            aciertos=aciertos+1
    precision_validacion_ant=precision_validacion
    precision_validacion=aciertos/len(Vect_clas)
    print("Validacion en Epoch", i,":","Precision->",precision_validacion)
    #si la precision anterior supera en 0,004 a la precision actual detenemos el entrenamiento
    if precision_validacion_ant-precision_validacion>0.004:
        print("Se hace una parada temprana del entrenamiento debido a que la precision ha empeorado")
        break
```

```
Epoch 0 : Loss-> 1.09967596970153 Precision-> 0.33
Validacion en Epoch 0 : Precision-> 0.31666666666666665
Epoch 1000 : Loss-> 0.18791285087958942 Precision-> 0.94
Validacion en Epoch 1000 : Precision-> 0.9533333333333334
Epoch 2000 : Loss-> 0.17097524865371688 Precision-> 0.9466666666666667
Validacion en Epoch 2000 : Precision-> 0.95
Epoch 3000 : Loss-> 0.16239705928949635 Precision-> 0.9466666666666667
Validacion en Epoch 3000 : Precision-> 0.9433333333333334
Se hace una parada temprana del entrenamiento debido a que la precision ha empeorado
Comienza el test
Precision del test: 0.92
```

4. Experimentar con distintos parámetros de configuración del generador de datos para generar sets de datos más complejos (con clases más solapadas, o con más clases). Alternativamente experimentar con otro generador de datos distinto (desarrollado por usted). Evaluar el comportamiento de la red ante estos cambios

Si agregamos más clases (clases más solapadas) la precisión disminuye considerablemente como es de esperar. A continuación mostramos que pasa cuando duplicamos el número de clases

```
iniciar(numero_clases=6, numero_ejemplos=300, graficar_datos=True)
```



```
Epoch 1000 : Loss-> 0.5776355101658556 Precision-> 0.75
Validacion en Epoch 1000 : Precision-> 0.6266666666666667
Epoch 2000 : Loss-> 0.5222571039260979 Precision-> 0.79
Validacion en Epoch 2000 : Precision-> 0.6833333333333333
Epoch 3000 : Loss-> 0.5047631455321996 Precision-> 0.7833333333333333
Validacion en Epoch 3000 : Precision-> 0.6966666666666667
Epoch 4000 : Loss-> 0.48562783021922196 Precision-> 0.79
Validacion en Epoch 4000 : Precision-> 0.6666666666666666
Se hace una parada temprana del entrenamiento debido a que la precision ha empeorado
Comienza el test
Precision del test: 0.8066666666666666
```

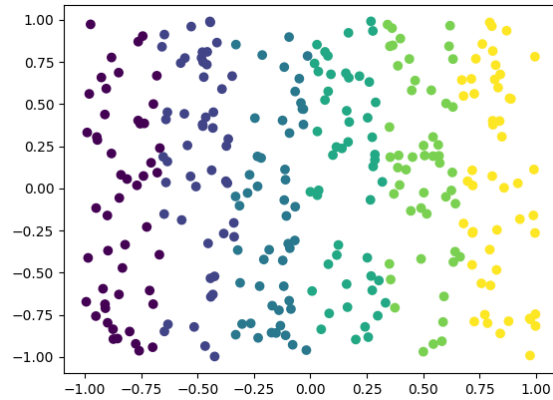
Vemos que ahora la precisión se aproxima a 80%, cuando antes superaba fácilmente el 90%

Ahora veamos qué pasa si usamos otro generador de datos, optamos por uno más sencillo donde las clases son separadas verticalmente

```
for clase in range(cantidad_clases):
    # Generamos un rango con los subíndices de cada punto de esta clase. Este rango se va
    # desplazando para cada clase: para la primera clase los índices están en [0, n-1], para
    # la segunda clase están en [n, (2 * n) - 1], etc.
    indices = range(clase * n, (clase + 1) * n)

    # Generamos las "entradas", los valores de las variables independientes. Dividimos las clases verticalmente
    paso=2/cantidad_clases
    x1 = ((-1+paso*(clase+1))-(-1+paso*clase))*np.random.random(n)+(-1+paso*clase)
    x2 = (1-(-1))*np.random.random(n)+(-1)
    x[indices] = np.c_[x1, x2]

    # Guardamos el valor de la clase que le vamos a asociar a las entradas x1 y x2 que acabamos
    # de generar
    t[indices] = clase
return x, t
```

Como era de esperar, al tratarse de un análisis más fácil de los datos, la precisión aumenta considerablemente llegando a 100% en el entrenamiento y a 99% en el test

```
Epoch 1000 : Loss-> 0.1193122824590505 Precision-> 0.95
Validation en Epoch 1000 : Precision-> 0.9566666666666667
Epoch 2000 : Loss-> 0.06755106315409495 Precision-> 0.98
Validation en Epoch 2000 : Precision-> 0.97
Epoch 3000 : Loss-> 0.047447168856112346 Precision-> 0.9866666666666667
Validation en Epoch 3000 : Precision-> 0.97
Epoch 4000 : Loss-> 0.029579321124361888 Precision-> 0.9966666666666667
Validation en Epoch 4000 : Precision-> 0.9833333333333333
Epoch 5000 : Loss-> 0.024179856248799262 Precision-> 1.0
Validation en Epoch 5000 : Precision-> 0.9866666666666667
Epoch 6000 : Loss-> 0.02040573001070096 Precision-> 1.0
Validation en Epoch 6000 : Precision-> 0.9866666666666667
Epoch 7000 : Loss-> 0.017579070577267694 Precision-> 1.0
Validation en Epoch 7000 : Precision-> 0.9866666666666667
Epoch 8000 : Loss-> 0.015403630227015271 Precision-> 1.0
Validation en Epoch 8000 : Precision-> 0.9866666666666667
Epoch 9000 : Loss-> 0.013676186782575216 Precision-> 1.0
Validation en Epoch 9000 : Precision-> 0.9866666666666667
Comienza el test
Precision del test: 0.9933333333333333
```

5. Modificar el programa para que funcione para resolver problemas de regresión

- Debe modificarse la función de pérdida y sus derivadas, utilizando por ejemplo MSE

Regresión: Mean Squared Error (MSE)

$$L_i(W) = (t_i - y_i)^2$$

Salida real

$$L(W) = \frac{1}{m} \sum_i L_i(W)$$


Salida target

Ejemplos

$$\frac{dL}{dy} = \frac{1}{m} \cdot 2 \cdot (t - y) \cdot (-1) = -\frac{2}{m} (t - y)$$

```
# LOSS
Li=np.power(t-y,2)
loss=1/m*np.sum(Li)
```

$$W_{t+1} = W_t - \varepsilon \nabla L_W$$



$$\left\{ \begin{array}{l} W^2(t+1) = W^2(t) - \varepsilon \frac{\partial L}{\partial W^2} \\ B^2(t+1) = B^2(t) - \varepsilon \frac{\partial L}{\partial B^2} \\ W^1(t+1) = W^1(t) - \varepsilon \frac{\partial L}{\partial W^1} \\ B^1(t+1) = B^1(t) - \varepsilon \frac{\partial L}{\partial B^1} \end{array} \right.$$

- Ecuaciones finales de los ajustes de los parámetros
(Loss MSE y función de activación Sigmoide)

$$\frac{\partial L}{\partial W^2} = H^T \left(\frac{2(Y - T)}{m} \right) \quad (p \times q)$$

$$\frac{\partial L}{\partial B^2} = \frac{2(Y - T)}{m}$$

(1 x q, al sumar verticalmente a lo largo de m)

$$\frac{\partial L}{\partial H} = \frac{\partial L}{\partial Y} \frac{\partial Y}{\partial H} = \frac{\partial L}{\partial Y} (W^2)^T \quad (m \times p)$$

$$\frac{\partial L}{\partial Z} = \frac{\partial L}{\partial H} \frac{\partial H}{\partial Z} = \frac{\partial L}{\partial H} \sigma(x)(1 - \sigma(x)) \quad (m \times p)$$

$$\frac{\partial L}{\partial W^1} = \frac{\partial L}{\partial Z} \frac{\partial Z}{\partial W^1} = X^T \frac{\partial L}{\partial Z} \quad (n \times p)$$

$$\frac{\partial L}{\partial B^1} = \frac{\partial L}{\partial Z} \quad (1 \times p)$$

(se suma verticalmente a lo largo de m)

```
# Ajustamos los pesos: Backpropagation
dL_dw2 = h.T.dot(2*(y-t)/m)          # Ajuste para w2
dL_db2 = np.sum((2*(y-t)/m), axis=0, keepdims=True) # Ajuste para b2

dL_dy = -2/m*(t-y)
dL_dh = dL_dy.dot(w2.T)
dL_dz = dL_dh.dot(sigmoid(z).T).dot(1-sigmoid(z)) # El calculo dL/dz

dL_dw1 = x.T.dot(dL_dz)              # Ajuste para w1
dL_db1 = np.sum(dL_dz, axis=0, keepdims=True) # Ajuste para b1
```

- b. Debe crearse un generador de datos nuevo para que genere datos continuos (pueden mantenerse igualmente 2 entradas; en caso de usar más entradas puede requerirse más capas en la red neuronal)

```
#Datos continuos
def generar_datos_clasificacion(cantidad_ejemplos):

    x = np.zeros((cantidad_ejemplos,1))
    t = np.zeros((cantidad_ejemplos,1))

    for i in range(cantidad_ejemplos):
        t[i] = i
        x[i] = t[i] + np.random.uniform(-1,1)
    return x, t
```

Nota: Cuando ejecutamos el programa la pérdida va en aumento en lugar de disminuir, todavía no logramos encontrar la causa de este problema

6. Realizar un barrido de parámetros (learning rate, cantidad de neuronas en la capa oculta, comparación de ReLU con Sigmoide)