

Facultad de Ciencias, UNAM

Redes Neuronales

Tarea 2

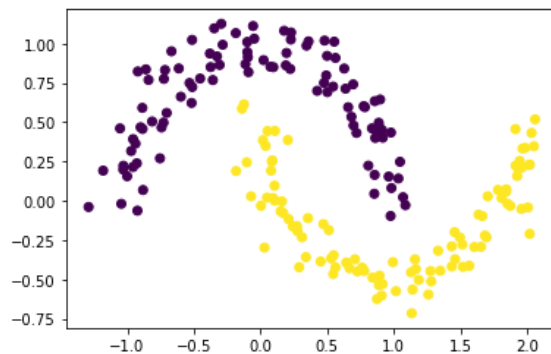
Rubí Rojas Tania Michelle

13 de abril de 202

1. Usando `sklearn.datasets.make_moons` genera un conjunto de datos de la siguiente forma:

```
In [1]: C1, C2 = moons(random state=123, n samples=200, noise=0.1)
```

```
1 from sklearn.datasets import make_moons
2 import matplotlib.pyplot as plt
3
4 C1, C2 = make_moons(random_state = 123, n_samples=200, noise=0.1)
5 plt.scatter(C1[:,0], C1[:,1], c = C2)
6 plt.show()
7
```



- a) Implementa la regresión logística usando el descenso gradiente para clasificar C_1 y C_2 .

SOLUCIÓN:

```
1 import numpy as np
2
3 '''
4 La funcion sigmoide es de la forma  $S(z) = 1/(1 + e^{-z})$ , donde
5  $z = x * w^T + b$ .
6 '''
7 def sigmoide(z):
8     return 1 / (1 + np.exp(-z))
9
10 def de_coste(y, y_hat):
11     j = - (y.dot(np.log(y_hat)) + (1 - y).dot((np.log(1 - y_hat))))
12     return j
13
```

```

14     def descenso_gradiente(x, y, alpha, theta, num_iteraciones):
15         costos = []
16         for each_iter in range (num_iteraciones):
17             z = np.dot(x, theta[1:]) + theta[0]
18             y_hat = sigmoide(z)
19             error = y_hat - y
20             gradiente = x.T.dot(error)
21             theta[0] -= alpha * error.sum()
22             theta[1:] -= alpha * gradiente
23             costos.append((1 / num_iteraciones) * (de_coste(y, y_hat)))
24         return costos
25
26     def prediccion(data):
27         z = np.dot(data, theta[1:]) + theta[0]
28         return np.where(sigmoide(z) >= 0.5, 0, 1)
29

```

Asignando valores a θ , α y al número de iteraciones, obtenemos

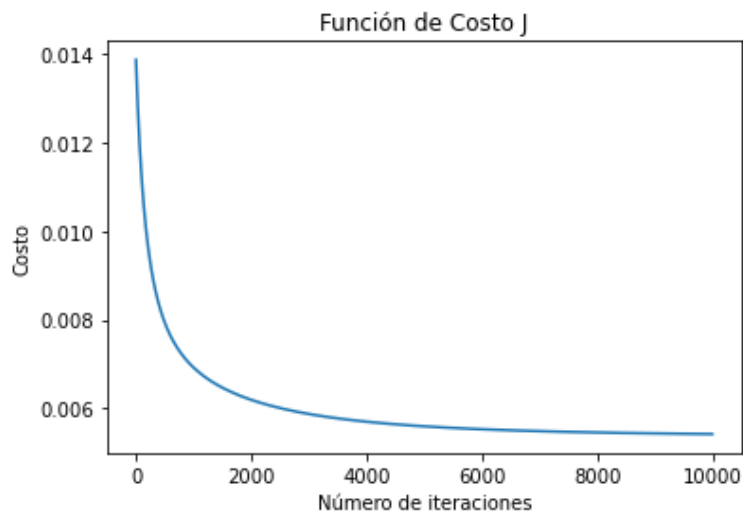
```

1     k, l = C1.shape
2     theta = np.zeros(l+1)
3     costo = descenso_gradiente(C1, C2, 0.0001, theta, 10000)
4
5     print ('Interseccion:', theta[0])
6     print ('Coeficientes estimados:', theta[1:])
7
8     # Visualizamos la funcion de costo.
9     plt.title('Funcion de Costo J')
10    plt.xlabel('Numero de iteraciones')
11    plt.ylabel('Costo')
12    plt.plot(costo)
13    plt.show()
14

```

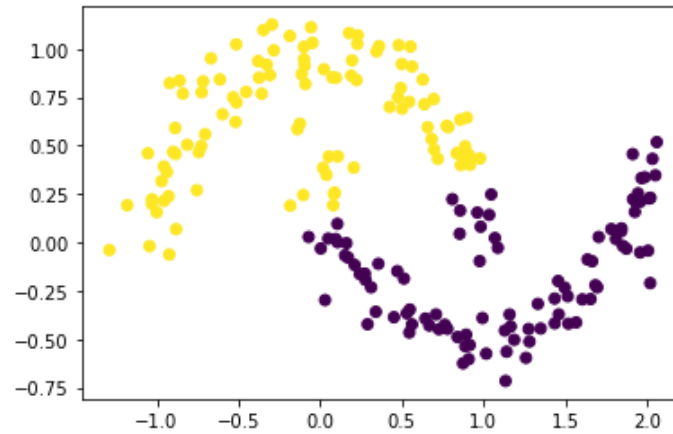
Intersección: 0.5894707477910425

Coeficientes estimados: [1.15656526 -4.85276922]



Finalmente, la clasificación quedaría de la siguiente forma:

```
1 plt.scatter(C1[:,0], C1[:,1], c = prediccion(C1))
2 plt.show()
3
```

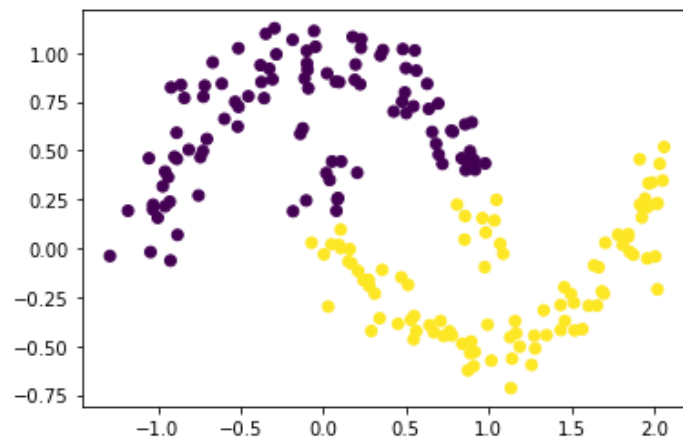


Utilizando **sklearn** para comparar los resultados obtenemos

```
1 from sklearn.linear_model import LogisticRegression
2
3 regresion_logistica = LogisticRegression()
4 modelo = regresion_logistica.fit(C1, C2)
5
6 print ("Interseccion: ", modelo.intercept_)
7 print ("Coeficientes estimados: ", modelo.coef_)
8
9 # Visualizamos la clasificacion.
10 plt.scatter(C1[:,0], C1[:,1], c = regresion_logistica.predict(C1))
11 plt.show()
12
```

Intersección: [0.36023346]

Coeficientes estimados: [[1.09045248 -3.69780292]]

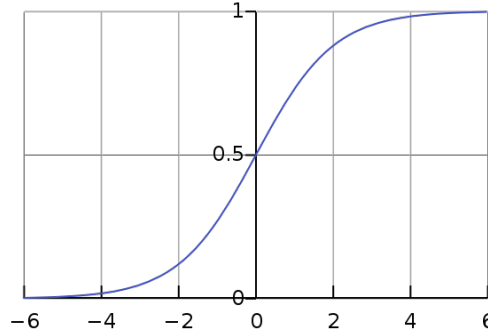


b) ¿Qué transformación de los datos ocupaste para poder hacer la correcta clasificación?

SOLUCIÓN: Sabemos que la función *sigmoide*

$$S(z) = \frac{1}{1 + e^{-z}} \quad z = w^T x + b$$

nunca llega a tocar el eje de las y a 0 o 1, y se mueve entre estos dos valores.



Lo cual la hace perfecta para expresar una probabilidad para nuestra decisión binaria (clasificar).

Ahora bien, debemos encontrar los valores correctos de w y b que nos permitan encontrar el valor correcto de la función S . El vector de pesos w se inicializa de manera arbitraria (aunque en nuestra implementación lo inicializamos como el vector cero).

Para saber qué tan bien o qué tan mal logra clasificar nuestra función, requerimos una función de error o de coste. Y teniendo en cuenta que nuestro problema no es convexo, entonces sería de la forma

$$L(y', y) = -(y \log y' + (1 - y) \log(1 - y'))$$

Esta función únicamente se usará sobre una entrada del conjunto de datos. Ahora, para obtener una imagen general sobre el error de los parámetros con respecto al total de los datos, usaremos la función de costo, que es definida como:

$$\begin{aligned} J(w, b) &= \frac{1}{m} \sum_{i=1}^m L(y'^i, y^i) \\ &= -\frac{1}{m} \sum_{i=1}^m [y^i \log y'^i + (1 - y^i) \log(1 - y'^i)] \end{aligned}$$

El objetivo es minimizar el costo total del modelo, para así acercarnos lo más posible a nuestras clases dadas y . Esta minimización la hacemos utilizando el *descenso gradiente*.

Por lo tanto, la función *sigmoide* usa los datos de entrada y los valores del modelo θ para minimizar la función de coste con ayuda del algoritmo del descenso. Ésta es la transformación de los datos que se va realizando, y se puede ver en la gráfica de función de coste 1a .

2. Calcula la derivada de la tangente hiperbólica \tanh .

SOLUCIÓN:

$$\begin{aligned}
 \frac{d}{dx} \tanh(x) &= \frac{d}{dx} \left(\frac{\sinh(x)}{\cosh(x)} \right) && \text{definición de } \tanh(x) \\
 &= \frac{(\sinh'(x) \cdot \cosh(x)) - (\cosh'(x) \cdot \sinh(x))}{\cosh^2(x)} && \text{derivative quotient rule} \\
 &= \frac{(\cosh(x) \cdot \cosh(x)) - (\sinh(x) \cdot \sinh(x))}{\cosh^2(x)} && \sinh'(x) = \cosh(x) \text{ y } \cosh'(x) = \sinh(x) \\
 &= \frac{\cosh^2(x) - \sinh^2(x)}{\cosh^2(x)} && \text{aritmética} \\
 &= \frac{1}{\cosh^2(x)} && \cosh^2(x) - \sinh^2(x) = 1 \\
 &= \operatorname{sech}^2(x) && \frac{1}{\cosh^2} = \operatorname{sech}^2(x)
 \end{aligned}$$

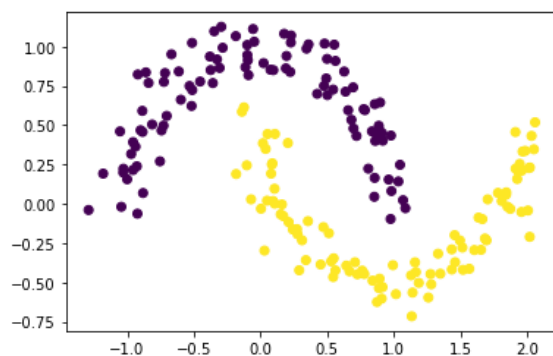
3. Usando el perceptrón multicapa visto en clase, clasifica a C_1 y C_2 . ¿Qué parámetros ocupaste?

SOLUCIÓN: Primero, generamos un conjunto de datos.

```

1  from sklearn.datasets import make_moons
2  import matplotlib.pyplot as plt
3
4  C1, C2 = make_moons(random_state = 123, n_samples=200, noise=0.1)
5  plt.scatter(C1[:,0], C1[:,1], c = C2)
6  plt.show()
7

```



Apoyándonos del **MLPClassifier** para clasificar el conjunto de datos, obtenemos

```

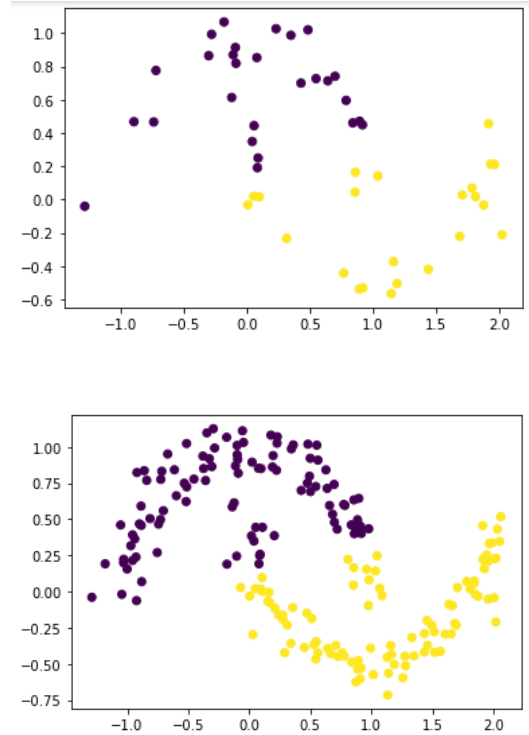
1  from sklearn.neural_network import MLPClassifier
2  from sklearn.model_selection import train_test_split
3  import matplotlib.pyplot as plt
4
5  # Dividimos al conjunto de datos para poder manejarlo.
6  X_train, X_test, y_train, y_test = train_test_split(C1, C2)
7
8  # Aquí podemos observar los parametros que utilizamos.
9  modelo = MLPClassifier(activation = 'tanh', max_iter = 1000,
10                        learning_rate_init = 0.001, hidden_layer_sizes = (10,4))
11
12  modelo.fit(X_train, y_train)
13
14

```

```

15 # Visualizamos la clasificacion con los datos de prueba.
16 plt.scatter(X_test[:,0], X_test[:,1], c = modelo.predict(X_test))
17 plt.show()
18
19 fig = plt.figure()
20 ax1 = fig.add_subplot(111)
21 # Visualizamos la clasificacion con los datos de prueba y entrenamiento.
22 ax1.scatter(X_test[:,0], X_test[:,1], c = modelo.predict(X_test))
23 ax1.scatter(X_train[:,0], X_train[:,1], c = modelo.predict(X_train))
24 plt.show()
25

```



4. Con la red neuronal, vista en clase, que hace la clasificación multiclase usando la función *softmax*, realiza los siguiente ejercicios:

a) Encuentra la mejor arquitectura para el conjunto de Iris. Justifica tu respuesta de por qué es la mejor.

SOLUCIÓN: Dado un vector de entrada, queremos poder clasificarlo en *Setosa*, *Versicolor* o *Virginica*.

Tomando el siguiente esquema como referencia

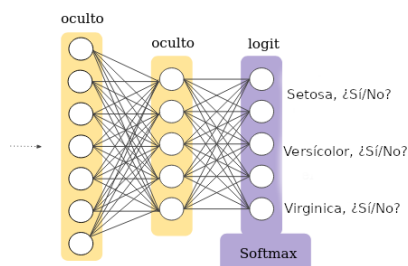


Figura 2. Una capa de softmax en una red neuronal

podemos visualizar un poco mejor la arquitectura elegida:

- Capa de entrada. Tendremos 4 neuronas, una por cada entrada de nuestro vector de inicio. Cada uno de estos vectores se ve de la forma:

$$v = [l \ a \ l \ a]$$

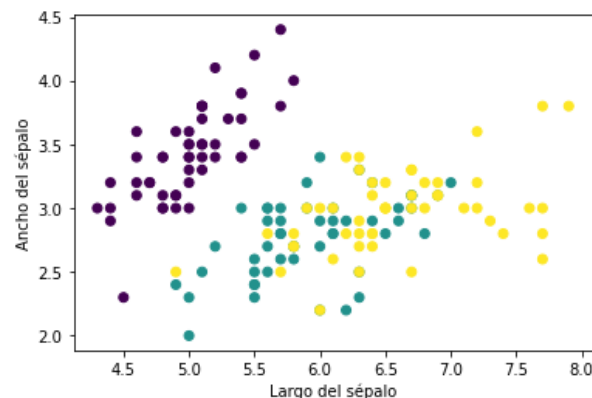
donde l es el largo del sépalo y a es el ancho del sépalo.

- Capa oculta. Tendremos ocho neuronas. Esta elección la hice basándome en una implementación que encontré (donde utilizan softmax) para clasificar la flor Iris. Al principio probé con tres y cuatro neuronas en esta capa, pero no lograba que la tasa de aprendizaje y el número de iteraciones logaran un buen resultado al momento de hacer las clasificaciones con los vectores de entrada (las entradas en el vector que regresa la función *predict* eran muy parecidas entre sí, (casi siempre un 0.33...) y definitivamente esto no es lo que nosotros estamos buscando). Por lo que, esta arquitectura es la mejor para mí (ya que funciona).
- Capa de salida. Tendremos tres neuronas, una por cada especie relacionada a la flor Iris.

b) Usa las funciones \tanh y γ en la capa intermedia. ¿Cuál funciona mejor?

SOLUCIÓN: Primero, generamos el conjunto de datos **Iris**.

```
1 from sklearn.datasets import load_iris
2 import matplotlib.pyplot as plt
3
4 # Cargamos el conjunto de datos.
5 iris = load_iris()
6 # Creamos la matriz de características.
7 X = iris.data
8 # Creamos el vector de características.
9 y = iris.target
10
11 # Visualizamos los puntos de entrenamiento.
12 plt.scatter(X[:,0], X[:,1], c = y)
13 plt.xlabel('Largo del sépalo')
14 plt.ylabel('Ancho del sépalo')
15 plt.show()
16
```



Los estímulos (vectores de entrada) son codificados para mostrar un 1 en entrada correspondiente a su valor, por ejemplo

$$2 = [0, 0, 1]$$

```

1 from sklearn.preprocessing import OneHotEncoder
2
3 y = iris.target.reshape(-1,1)
4 encoder = OneHotEncoder(sparse=False)
5 y = encoder.fit_transform(y)
6

```

Ahora, no sólo queremos un valor entre 0 y 1, sino que queremos un vector en el que la salida deseada marque un 1 y los demás 0 (o algo cercano, en ambos casos). Es decir, no podemos usar *n* *sigmoides* porque el valor de cada entrada está condicionada a ser 1 sólo si los demás dan 0.

Esto se puede lograr con la función *Softmax* que nos da la probabilidad *i*-ésima de cada entrada en el vector aleatorio *x*.

$$p_i = \frac{e^{x_i}}{\sum_j^N e^{x_j}}$$

Pero ahora no podemos usar el *MSE* (ni *MAE*) como nuestra función de error. Debemos usar algo que nos permita ponderar el error de haberle *atinado* a la entrada deseada con un 1, y con un 0 a las demás.

Cross-Entropy es la función que generaliza a la función que usamos para el caso binario.

$$CE = \sum_i^N y_i \log(x_i)$$

El paso siguiente es definir los parámetros, la arquitectura (definida en el inciso anterior) y las funciones que vamos a utilizar:

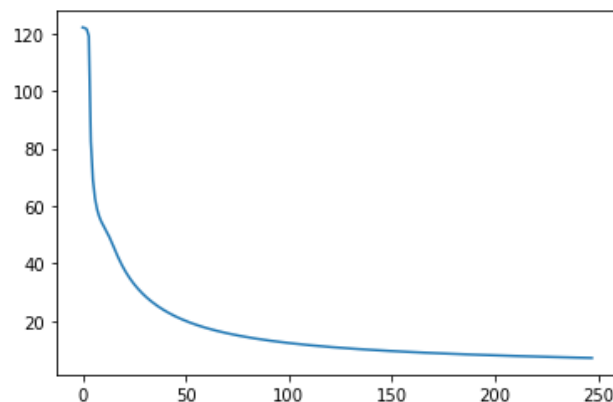
```

1 import numpy as np
2
3 def sigmoide(z):
4     return 1 / (1 + np.exp(-z))
5
6 def sigmoide_prima(x):
7     return sigmoide(x) * (1 - sigmoide(x))
8
9 def tanh(x):
10    return ((np.exp(x) - np.exp(-x)) / (np.exp(x) + np.exp(-x)))
11
12 def tanh_prima(x):
13    return (1 / (np.exp(x) + np.exp(-x)))**2
14
15 def softmax(A):
16    expA = np.exp(A)
17    return expA / expA.sum(axis = 1, keepdims = True)
18
19 N = X.shape[0] # numero de instancias
20 n = X.shape[1] # dimension de instancias
21 nh = 8         # numero de neuronas ocultas
22 m = 3         # dimension salida
23
24 wh = np.random.rand(n,nh) # pesos escondidos
25 bh = np.random.randn(nh)  # nodo sesgo escondidos
26
27 wo = np.random.rand(nh,m) # pesos salida
28 bo = np.random.randn(m)   # nodo salida
29 # lr = 0.0001              # tasa de aprendizaje para tanh
30 # lr = 0.00001             # tasa de aprendizaje para sigmoide
31

```

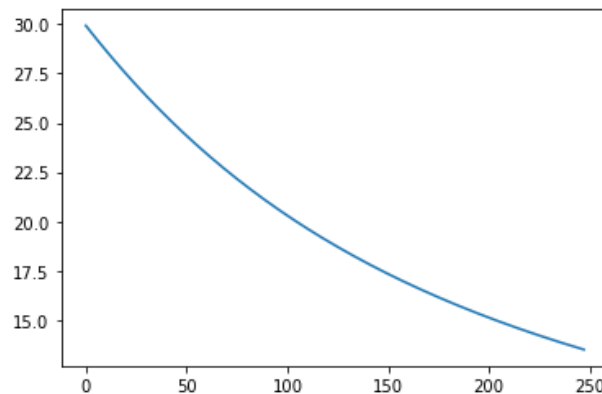

Ahora, realizamos el entrenamiento usando la función de activación tanh

```
1      from sklearn.model_selection import train_test_split
2
3      # Dividimos nuestro conjunto de datos para poder manejarlo.
4      Xt, Xs, yt, ys = train_test_split(X,y)
5
6      J = []
7      for epoca in range (50000):
8          # Retropropagacion
9
10         # Feed forward
11         zh = np.dot(X_train, wh) + bh      # activacion escondidas
12         ah = tanh(zh)                      # disparo escondidas
13
14         zo = np.dot(ah, wo) + bo           # activacion salida
15         ao = softmax(zo)                  # disparo salida (neuronas salida)
16
17         # Envio de error hacia atras
18
19         # Capa salida - escondida
20         error = ao - y_train
21
22         deltas_wo = np.dot(ah.T, error)
23         deltas_bo = error
24
25         # Capa escondida - entrada
26         temp_dah = np.dot(error , wo.T)
27         deriv_h = tanh_prima(zh)
28
29         deltas_wh = np.dot(X_train.T, deriv_h * temp_dah)
30         deltas_bh = temp_dah * deriv_h
31
32         # Actualizacion de pesos
33         wh -= lr * deltas_wh
34         bh -= lr * deltas_bh.sum(axis=0)
35
36         wo -= lr * deltas_wo
37         bo -= lr * deltas_bo.sum(axis=0)
38
39         if epoca % 200 == 0:
40             logL = np.sum(-y_train * np.log(ao))
41             J.append(logL)
42
```



Realizamos el entrenamiento también con la función de activación γ .

```
1  from sklearn.model_selection import train_test_split
2
3  # Dividimos nuestro conjunto de datos para poder manejarlo.
4  X_train, X_test, y_train, y_test = train_test_split(X, y)
5
6  J = []
7  for epoca in range (50000):
8      # Retropropagacion
9
10     # Feed forward
11     zh = np.dot(X_train, wh) + bh      # activacion escondidas
12     ah = sigmoide(zh)                  # disparo escondidas
13     zo = np.dot(ah, wo) + bo           # activacion salida
14     ao = softmax(zo)                   # disparo salida
15
16     # Envio de error hacia atras
17
18     # Capa salida - escondida
19     error = ao - y_train
20
21     deltas_wo = np.dot(ah.T, error)
22     deltas_bo = error
23
24     # Capa escondida - entrada
25     temp_dah = np.dot(error, wo.T)
26     deriv_h = sigmoide_prima(zh)
27
28     deltas_wh = np.dot(X_train.T, deriv_h * temp_dah)
29     deltas_bh = temp_dah * deriv_h
30
31     # Actualizacion de pesos
32     wh -= lr * deltas_wh
33     bh -= lr * deltas_bh.sum(axis=0)
34
35     wo -= lr * deltas_wo
36     bo -= lr * deltas_bo.sum(axis=0)
37
38     if epoca % 200 == 0:
39         logL = np.sum(-y_train * np.log(ao))
40         J.append(logL)
41
42 plt.plot(J[2:])
43
```



Ahora bien, ¿cómo sabemos cuál de las dos funciones de activación es mejor? Esto depende de la arquitectura de la red neuronal, de la función de activación y de la tasa de aprendizaje que utilicemos. La arquitectura de la red fue mencionada en el inciso anterior, y sabemos que para las funciones de activación \tanh y γ , sus tasas de activación son 0.0001 y 0.00001, respectivamente. Experimentando con ambos modelos obtenemos que la configuración que minimiza el error es aquella que posee la función de activación γ , es decir, el sigmoide.

c) Clasifica los siguientes estímulos y reporta a qué clase pertenece cada uno:

- 5.97 4.20 1.23 0.25
- 6.80 5.00 1.25 1.20
- 12.50 9.20 40.32 21.55

SOLUCIÓN: Utilizamos la siguiente función

```

1      def predict(wh, wo, fact1, fact2, x):
2          zh = np.dot(x.T, wh) + bh          # activacion escondidas
3          ah = fact1(zh)                     # disparo escondidas
4
5          zo = np.dot(ah, wo) + bo          # activacion salida
6          ao = fact2(zo)                     # disparo salida
7          return ao
8

```

para clasificar los estímulos solicitados:

- Con la función de activación \tanh .

```

1      v1 = np.array([[5.97],[4.20],[1.23],[0.25]])
2      v2 = np.array([[6.80],[5.00],[1.25],[1.20]])
3      v3 = np.array([[12.50],[9.20],[40.32],[21.55]])
4
5      p1 = predict(wh, wo, tanh, softmax, v1)
6      p2 = predict(wh, wo, tanh, softmax, v2)
7      p3 = predict(wh, wo, tanh, softmax, v3)
8
9      print(p1)
10     print(p2)
11     print(p3)
12

```

```

[[9.80936010e-01 1.90639844e-02 6.05670038e-09]]
[[9.80199084e-01 1.98009097e-02 6.53005296e-09]]
[[3.22145137e-09 6.30947763e-03 9.93690519e-01]]

```

Por lo tanto, los dos primeros vectores corresponden a la especie *setosa* y el último corresponde a la especie *virginica*.

- Con la función de activación γ .

```

1      v1 = np.array([[5.97],[4.20],[1.23],[0.25]])
2      v2 = np.array([[6.80],[5.00],[1.25],[1.20]])
3      v3 = np.array([[12.50],[9.20],[40.32],[21.55]])
4
5      p1 = predict(wh, wo, sigmoide, softmax, v1)
6      p2 = predict(wh, wo, sigmoide, softmax, v2)
7      p3 = predict(wh, wo, sigmoide, softmax, v3)
8
9      print(p1)
10     print(p2)
11     print(p3)
12

```

```
[[0.96800806 0.02926022 0.00273171]]
[[0.96823689 0.02905003 0.00271307]]
[[0.00214928 0.03275745 0.96509327]]
```

Por lo tanto, los dos primeros vectores corresponden a la especie *setosa* y el último corresponde a la especie *virginica*.

- d) ¿Te parecen correctas todas las clasificaciones? En caso de que alguna no, ¿por qué? ¿cómo corregirías este error?

SOLUCIÓN: Me parecen correctas las clasificaciones para los dos primeros vectores de entrada (ambos se clasifican como *setosa*), pero no la clasificación del último vector (ya que por los datos que tiene cada entrada, podemos notar que no se trata de una especie de la flor Iris). Este último vector se clasifica como una especie de la flor Iris por que nuestra red neuronal no tiene la cantidad suficiente de datos para poder entrenarse y poder discernir correctamente (con esta premisa, dudaríamos un poco sobre la veracidad de la clasificación de los dos primeros vectores, pero la consideramos correcta ya que con ambas funciones de activación obtuvimos el mismo resultado).

El error lo podríamos solucionar incrementando (mucho) el conjunto de datos para después entrenar la red neuronal con él y que así sea capaz de detectar el error. Otra opción sería entrenar la red neuronal usando una función de activación que se fije en la densidad de un punto (se utiliza la densidad de puntos como clases válidas) o puntos. Las funciones de activación alternativas que buscamos son las llamadas *Funciones de Base Radial* (podemos ver un poco de ellas en el siguiente inciso).

5. ¿Qué es y cómo funciona la función de activación *Radial Basis Function (RBF)*?

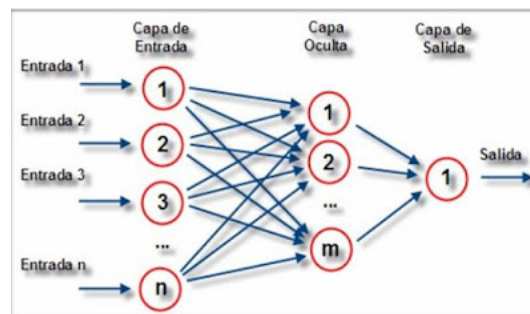
SOLUCIÓN: Una *función de base radial* $\phi(x)$ es una función que calcula la distancia euclideana de un vector de entrada x respecto de un centro c , de tal manera que resulta la siguiente función:

$$f(x) = (\|x - c_i\|)$$

A cada neurona de la capa de entrada le corresponde una función de base radial $\Phi(x)$ y un peso de salida w_i . El patrón de salida ingresa a una neurona de salida que suma las entradas y da como resultado una salida. La función de una red *RBF* final resulta:

$$F(x) = \sum_{i=1}^N w_i \Phi(\|x - c_i\|)$$

Las redes *RBF* tienen una construcción rígida de tres capas:



- **Capa de Entrada.** Transmiten las señales de entrada a las neuronas ocultas sin realizar procesamiento, es decir, las conexiones de la capa de entrada a la capa oculta no llevan pesos asociados.

- **Capa Oculta.** Realizan una transformación local y no lineal de dichas señales.

Cada elemento de procesado i de la capa oculta tiene asociada una función de base radial de tal manera que representa una clase o categoría, donde dicha clase viene dada por (C_i, d_i) . C_i representa un centro de cluster (pesos asociados a cada neurona i) y d_i representa la desviación, anchura o dilatación de la función de base radial asociada a dicho elemento.

La salida de cada elemento de la capa oculta $z_i(n)$ se calcula como la distancia que existe entre el patrón de entrada $X(n)$ al centro del clouster C_i ponderada inversamente por d_i , y aplicando después a ese valor una función de base radial.

$$z_i(n) = \Phi \left(\frac{\left(\sum_{j=1}^p (x_j(n) - c_{ji})^2 \right)^{\frac{1}{2}}}{d_i} \right) \quad i \in \{1, 2, \dots, m\}$$

donde Φ es una función de base radial, dentro de éstas la más utilizada es la función Gaussiana

$$\Phi(r) = e^{-\frac{r^2}{2}}$$

- **Capa de Salida.** Realiza una combinación de las actividades de las neuronas ocultas. Tiene la responsabilidad en la red de activación de patrones aplicados en la capa de entrada.

Cada elemento de procesado calcula su valor neto como una combinación lineal de las salidas de los elementos de procesado de la capa oculta. La función de activación y transferencia es lineal, por lo que para un patrón n , $X(n) = (x_1(n), x_2(n), \dots, x_p(n))$, la salida de la red asociada a cada elemento k de la capa de salida se obtiene de la forma

$$y_k(n) = \sum_{i=1}^m w_{ik} z_i(n) + \mu_k \quad k \in \{1, 2, \dots, r\}$$

donde los w_{ik} son los pesos asociados al elemento k de la capa oculta y el elemento i de la capa oculta, que ponderan cada uno las salidas $z_i(n)$ del elemento de procesado de la capa oculta correspondiente.

El término μ_k es un término denominado umbral y está asociado a cada uno de los elementos de procesado de la capa de salida.

El aprendizaje consiste en la determinación de centros, desviaciones y pesos de la capa oculta a la capa de salida. Como las capas de la red realizan diferentes tareas, se separarán los parámetros de la capa oculta de la capa de salida para optimizar el proceso. De esta forma, los centros y las desviaciones siguen un proceso guiado por una optimización en el espacio de entrada, mientras que los pesos siguen una optimización sobre la base de las salidas que se desean obtener.

Los métodos de aprendizaje más utilizados son el *método híbrido* y el *método totalmente supervisado*.

Bibliografía

- <https://medium.com/@dtellogaete/regresi%C3%B3n-log%C3%ADstica-en-python-y-r-machine-learning->
- <http://www.eenube.com/index.php/ldp/machine-learning/121-programar-una-red-neuronal-para-clas>
- <https://www.interactivechaos.com/manual/tutorial-de-machine-learning/perceptron-multicapa>
- <https://analisisydecision.es/machine-learnig-analisis-grafico-del-funcionamiento-de-algunos-a>
- <https://www.analyticslane.com/2020/04/20/entrenamiento-validacion-y-test-con-scikit-learn/>

- https://chrisalbon.com/machine_learning/basics/loading_scikit-learns_iris_dataset/
- https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_iris.html
- <https://developers.google.com/machine-learning/crash-course/multi-class-neural-networks/softmax?hl=es-419>
- https://es.wikipedia.org/wiki/Funci%C3%B3n_de_activaci%C3%B3n
- <https://blog.crespo.org.ve/2018/02/>
- https://www.wikiwand.com/es/RNA_de_base_radial#/Referencias
- http://ele.aut.ac.ir/~abdollahi/Lec_3_NN11.pdf
- <http://dianainteligenciaartificial.blogspot.com/2015/07/redes-de-neuronas-de-base-radial.html>
- <http://www.varpa.org/~mgpenedo/cursos/scx/archivospdf/Tema5-6.pdf>