GUIÓN DE PRÁCTICAS DE REDES NEURONALES ARTIFICIALES (TSCAO): Parte III

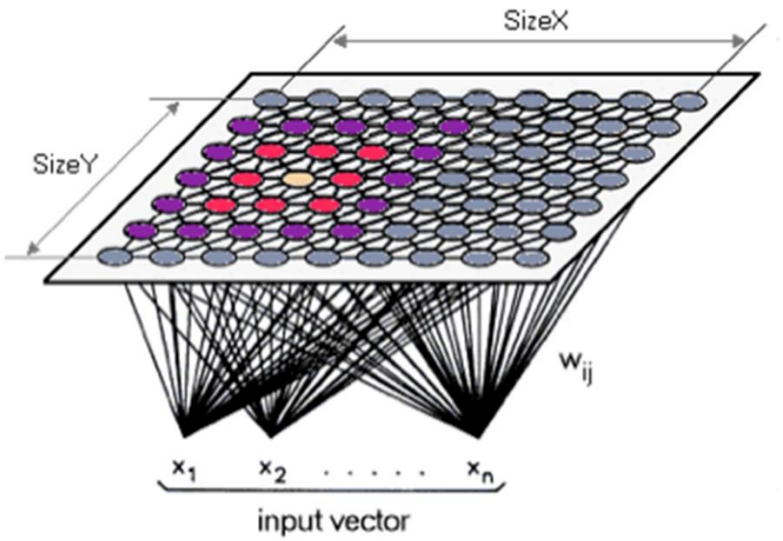
MªCarmen Pegalajar Jiménez

15/01/2024

Dpto Ciencias de la Computación e Inteligencia Artificial
Universidad de Granada

4. MAPAS AUTOORGANIZATIVOS (SOM)

Como hemos visto en la presentación de teoría, el Mapa Autoorganizativo de Kohonen (SOM) es un tipo de red neuronal artificial (ANN) que se entrena utilizando aprendizaje no supervisado produciendo una representación discreta de baja dimensión (típicamente bidimensional) del espacio de entrada de las muestras de entrenamiento, denominada mapa. Por lo tanto, esté método puede servir incluso para realizar reducción de dimensionalidad. Los Mapas Autoorganizativos se distinguen de otras redes neuronales artificiales en que aplican el aprendizaje competitivo en oposición al aprendizaje del ajuste del error (como la propagación hacia atrás con el descenso de gradiente), y usan una función de vecindad para preservar las propiedades topológicas del espacio de entrada.



El SOM fue inicialmente propuesto por el profesor finlandés Teuvo Kohonen en la década de 1980, por esta razón también es llamado Mapa de Kohonen.

Como podemos ver en la figura consta de dos capas, una de entrada y otra capa competitiva. Cada ejemplo de datos competirá por una neurona de la red. Los pasos del "mapeo" del SOM comienzan inicializando aleatoriamente los vectores asociados a los pesos. A partir de aquí, se busca en el mapa de vectores de pesos qué vector de pesos representa mejor ese ejemplo. La neurona correspondiente a estos pesos, tiene neuronas vecinas y, por tanto, pesos vecinos que están cerca de ésta. El peso que se elige se recompensa al ser el más parecido a este vector de ejemplo. Los

vecinos de este peso también son recompensados por parecerse algo al vector de muestra elegido. Esto permite que el mapa crezca y forme diferentes formas. En general, tiene formas cuadradas / rectangulares / hexagonales en el espacio de entidades 2D.

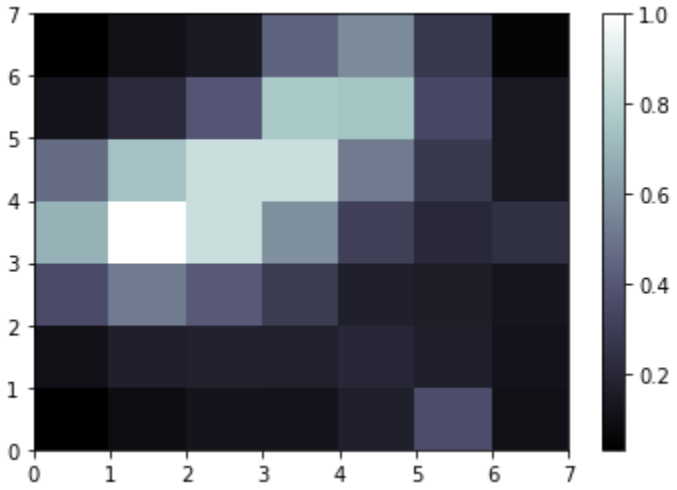
El algoritmo

1. Los pesos de cada nodo se inicializan aleatoriamente.
2. Se elige un vector del conjunto de datos de entrenamiento.
3. Cada nodo se examina para calcular cuáles son los pesos que más se parecen al vector de entrada. El nodo ganador se conoce comúnmente como BMU (Best Matching Unit).
4. Luego se calcula la vecindad de la BMU. El radio de vecindad (cantidad de vecinos) disminuye con el tiempo.
5. El vector de pesos ganador se recompensa al ser el más parecido al vector de ejemplo. También los vecinos se parecen algo al vector de muestra. Cuanto más cerca esté un nodo de la BMU, más se alterarán sus pesos y cuanto más lejos esté el vecino de la BMU, menos se modificará.
6. Repetir el paso 2 para N iteraciones.

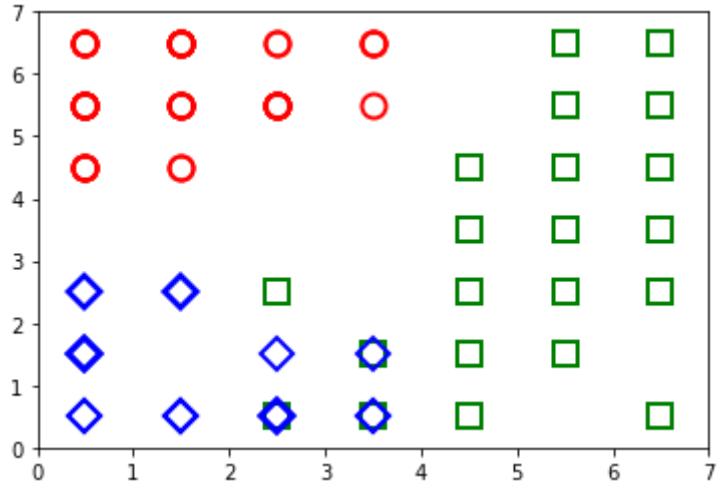
Best Matching Unit es una técnica que calcula la distancia desde cada peso al vector de ejemplo, pasando por todos los vectores de pesos. El vector de pesos con la distancia más pequeña es el ganador. Existen numerosas formas de determinar la distancia, sin embargo, el método más utilizado es la Distancia Euclidea.

En cuanto a la parte de implementación, hay varias bibliotecas de Python (minisom, sompy) que puedes usar directamente para implementar SOM.

Visualizaremos los datos en el SOM mediante una gráfica parecida a la siguiente.



En términos más simples, las partes más oscuras representan grupos, mientras que las partes más claras representan la división de los grupos. Si la distancia promedio es alta, entonces los pesos circundantes son muy diferentes y se asigna un color claro a la ubicación del peso (el color blanco indica máxima distancia). Si la distancia promedio es baja, se asigna un color más oscuro. El mapa anterior por ejemplo muestra que la concentración de diferentes grupos es más predominante en tres zonas. Esta figura solo nos dice dónde la densidad de ejemplos es mayor (regiones más oscuras) o menos (regiones más claras). La segunda visualización nos dice cómo están agrupados los ejemplos, específicamente en un problema que trabaja con 3 clases.



A continuación, vamos a implementar un SOM trabajando con el conjunto de datos: Credit Card Applications de Kaggle

Para esto además de las librerías anteriores tenemos que instalar la librería *Minisom*. Esta librería contiene una implementación del SOM, para instalarla tenemos que usar la siguiente línea de código: ***pip install minisom***.

Nuestro primer paso, será importar las librerías que son necesarias para nuestro estudio:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from minisom import MiniSom
```

A continuación, cargaremos nuestro archivo de datos:

```
dataset=pd.read_csv('D:\mcarmen2019-2010\PYTHON\CODIGO\Credit_Card_Applications.csv')
dataset.head()
```

Como podemos ver, en la tabla que se nos muestra con la sentencia "head", tenemos valores categóricos y continuos, y en estos datos tenemos "*fraudes*" que tenemos que detectar, las columnas son información de los clientes y cada fila representa un cliente. Cuando hablamos de que las SOMS van a buscar patrones, en este caso serán tipos de clientes. **Lo que haremos se llama segmentación de clientes, y uno de esos segmentos calculados va a contener clientes que son potencialmente fraudulentos.**

Todos los Clientes serán el valor de entrada, y todos estos puntos de entrada van a ser organizados en un nuevo espacio vectorial de entrada, y entre este espacio y el de salida, cada neurona va a ser inicializada con una cantidad de pesos igual a la cantidad de atributos, y la salida va a ser la neurona más cercana al tipo de cliente, esta neurona se llama BMU. BMU es la neurona que tiene un valor más similar al cliente que le estamos pasando, y así el mapa se irá ajustando a nuestros datos y por cada repetición (Época) nuestro espacio de entrada pierde dimensiones, y cuando el espacio vectorial ya no se hace más pequeño, es el momento en el cual obtenemos nuestro SOM.

```
In [3]:
```

```
X=dataset.iloc[:, :-1].values
Y=dataset.iloc[:,-1].values
```

Vamos a escalar nuestros valores de X entre 0 y 1 para que no haya grandes diferencias entre las variables usadas.

```
sc=MinMaxScaler(feature_range=(0,1))
X=sc.fit_transform(X)
```

Ahora vamos a construir nuestro SOM:

```
som = MiniSom(x = 10, y = 10, input_len = 15, sigma = 1.0, learning_rate = 0.5)
```

Los valores *x* e *y* van a ser el tamaño de nuestro espacio vectorial de salida. En este caso como no tenemos tantos valores de entrada, quiero que mi mapa sea de 10 x 10 (2 Dimensiones) en caso de tener una entrada mucho más grande podría ocurrir que nuestro mapa tuviera que ser más grande.

El siguiente parámetro es *input_len*, que corresponde a la dimensión de nuestros datos o el número de columnas que le estamos pasando, X contiene la ID del cliente, en otros casos no serviría de nada, pero aquí nos servirá para identificar a los potenciales clientes fraudulentos. *sigma*, es el valor del radio que tendrán las BMU para actualizar las neuronas más cercanas. *lerning_rate*, este parámetro define cuánto vamos a actualizar las neuronas más cercanas a la BMU por cada época (razón de aprendizaje). Ahora vamos a entrenarlo sobre nuestros datos para ello vamos a inicializar los pesos de manera aleatoria y pasaremos a entrenar.

```
# inicializa aleatoriamente los vectores de peso a números pequeños cercanos a 0

som.random_weights_init(X)
```

```
# entrenar SOM con X, matriz de características y patrones reconocidos

# som.train_random(data = X, num_iteration = 5000)
som.train_random(data = X, num_iteration = 100, verbose=1)
```

Visualizando los datos:

Una vez entrenado nuestro SOM tenemos que visualizar los resultados obtenidos, para ello vamos a representar gráficamente el SOM como una malla 2D que va a contener a todas las neuronas BMU. De cada una de estas BMU vamos a obtener lo que nos interesa, que es la distancia intermedia promedio entre estas neuronas. Mientras más bajo sea este número, más cerca estará este nodo de nuestro vecindario, y mientras más grande sea el número más lejos estará y si la mayoría de nuestras BMU representan las reglas que identificamos, la neurona que estaría lejos de esta mayoría seria nuestro cliente fraudulento.

Para visualizarlo usaremos colores, donde los BMU estarán coloreadas y mientras más grande sea la distancia intermedia promedio, más blanco será el color. No vamos a usar Matplotlib para esto porque el gráfico que necesitamos hacer es más específico, como estamos haciendo un SOM, necesitamos montar la visualización desde 0. Vamos a importar las siguientes funciones de *pylab*.

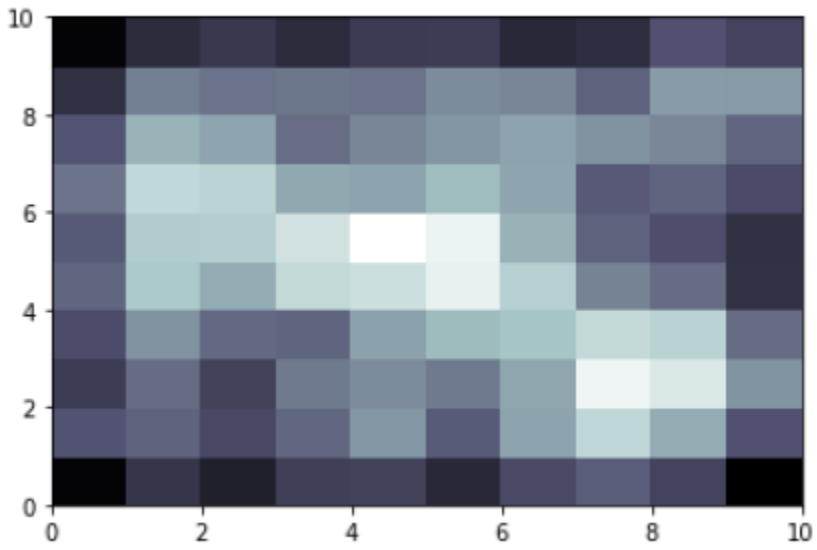
```
from pylab import bone, pcolor, colorbar, plot, show
```

Primero necesitamos inicializar el gráfico, que va a ser una ventana donde lo construiremos, para ello usaremos la función **bone**, que nos proporciona una ventana en blanco.

Lo siguiente que vamos a hacer es poner las diferentes **BMU** en el mapa, para ello pondremos en nuestro mapa la información de nuestra *Distancia Promedio* para todos los **BMU** que nuestro **SOM** identificó, para ello usaremos colores, que van a representar los valores promedios de todas las **Distancias Intermedias Promedio.**

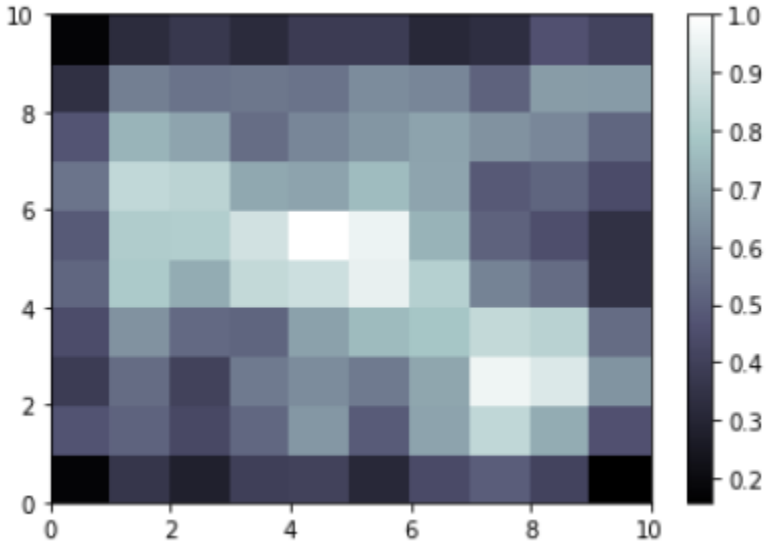
Para ello usaremos la función **pcolor** y dentro de ella pondré todas estas distancias intermedias de todos nuestros **BMU** en nuestro **SOM**, para obtener las distancias intermedias, tenemos un método que se llama *distance_map*, y esta función devuelve todas las distancias promedio en una matriz, y para pesar a la función *pcolor* los valores en el orden correcto, vamos a transponer la matriz con el método T, que nos devuelve la transpuesta de la misma matriz.

```
bone()
pcolor(som.distance_map().T)
```



Todos estos colores corresponden a la Distancia intermedia promedio de todas las neuronas, pero ahora quiero agregar más información, para saber si el blanco corresponde a una distancia grande o pequeña. Para agregar esta leyenda haremos lo siguiente:

```
bone()
pcolor(som.distance_map().T)
colorbar()
show
```



Esta barra de colores es el rango de distancias intermedias entre las neuronas, estos valores están normalizados lo que significa que van del 0 al 1 y claramente podemos ver que los valores más altos es decir los que están más distantes son los blancos y las más cercanos son más oscuros, y como explicamos antes ahora podemos saber dónde están los fraudes, estos con mucha probabilidad podrán ser los valores más claros.

Ahora haremos marcadores para ver qué clientes son fraudulentos y los que no, porque los clientes que son fraudulentos y fueron aprobados son más relevantes que los que no, sería interesante saber dónde están los clientes dentro del SOM.

```
bone()
pcolor(som.distance_map().T)
colorbar()
show
```

Ahora vamos a crear dos tipos de marcadores, círculos rojos y cuadrados verdes.

```
bone()
pcolor(som.distance_map().T)
colorbar()

markers= ['o','s']
colors = ['r','g']

for i,x in enumerate(X):
    w= som.winner(x)
    plot(w[0] + 0.5,
         w[1] + 0.5,
         markers [Y[i]],
         markeredgecolor = colors[Y[i]],
         markerfacecolor = 'None',
         markersize = 10,
         markeredgewidth = 2)
    show ()
```

Vamos a explicar línea por línea este código, las 3 primeras líneas ya están explicadas.

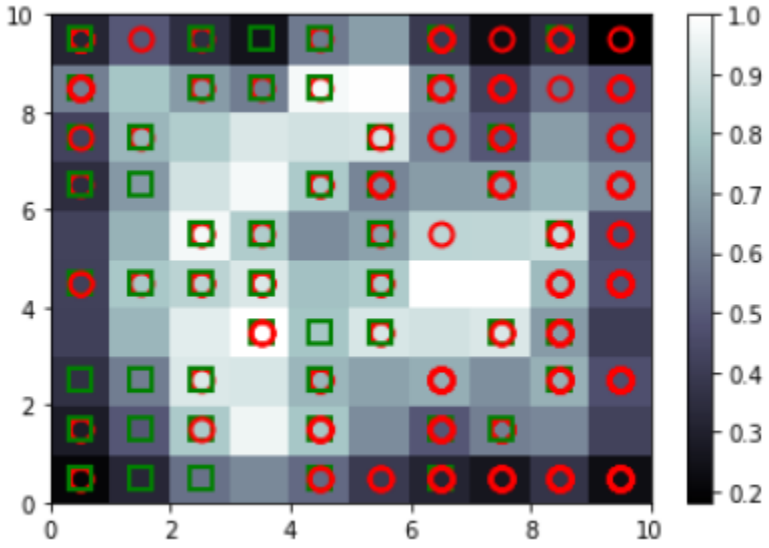
Necesitamos definir los marcadores y los colores, en este caso los marcadores son un círculo rojo para clientes no aprobados y un cuadrado verde para los aprobados.

Necesitamos un índice i y un valor de *x*, el índice es para buscar en la matriz Y si fueron o no aprobados, y las x son los datos individualmente.

w va a ser el valor que representa si nuestro nodo es una BMU, y para obtenerlo necesitamos usar la función *winner* de nuestro mapa, que devuelve el BMU para nuestro cliente específico.

Esto nos devolverá un valor que representa las coordenadas de nuestro nodo ganador al que pertenece nuestro dato pero estos w[0] y w[1] representan las esquinas de este cuadrado así que para llevarlos al centro necesitamos sumarle 0.5

Lo siguiente es buscar si el cliente fue aprobado o no en nuestra matriz Y, por lo que vamos a tomar el marcador y el color correspondiente al valor de Y que obtuvimos (0 o 1), y lo siguiente es configurar un poco la manera en que se mostraran estos valores y el resultado es el siguiente.



El marcador con el círculo nos dice que no se aprobó el cliente, que fue detectado como fraude, el cuadrado verde nos dice que sí fue aprobado, y cuando salen ambos significa que se aprobaron tanto como se rechazaron, por lo que vemos que algunos fraudes si fueron aprobados como clientes legítimos y vice-versa. Por último, vamos a extraer los valores de los clientes que son fraudulentos. Para hacer esto, desafortunadamente, no tenemos una función de mapeo inversa tenemos que tener la lista de clientes directamente de nuestros nodos ganadores, pero podemos usar una función de la librería que contiene todos los valores de los nodos ganadores que nosotros le pasemos, eso significa que primero debemos obtener esos nodos ganadores (que sean Blancos) y con esa información nos va a dar la lista de clientes fraudulentos. Por ejemplo, identifiquemos un nodo blanco que contenga información: "nodo (4,4)", este nodo contendrá clientes cuya distancia media entre neuronas está próxima o igual a 1 (muy alta), por tanto este nodo está lejos de los valores de las neuronas vecinas y los valores del conjunto de ejemplo que haya "agrupado" serán valores considerados como atípicos en el conjunto de ejemplos y por tanto posibles candidatos a clientes fraudulentos.

Primero vamos a obtener los valores de cada nodo con la siguiente línea de código

```
mappings = som.win_map(X)
```

con esto obtenemos los valores para todos los nodos, y ahora para obtener el valor de nuestro nodo hacemos lo siguiente:

```
frauds=np.array(mappings[(4,4)])
```

con esta línea obtendremos todos los valores de los posibles fraudes y para revisar qué valores son debemos transformarlos inversamente a nuestros valores originales

```
frauds =  sc.inverse_transform(frauds)
```

y así obtenemos todos los valores de los parámetros de los clientes fraudulentos

```
array([[1.5776156e+07, 1.0000000e+00, 2.2080000e+01, 1.1460000e+01,
        2.0000000e+00, 4.0000000e+00, 4.0000000e+00, 1.5850000e+00,
        0.0000000e+00, 0.0000000e+00, 0.0000000e+00, 1.0000000e+00,
        2.0000000e+00, 1.0000000e+02, 1.2130000e+03],
       [1.5768295e+07, 1.0000000e+00, 2.5580000e+01, 3.3500000e-01,
        2.0000000e+00, 4.0000000e+00, 8.0000000e+00, 3.5000000e+00,
        0.0000000e+00, 0.0000000e+00, 0.0000000e+00, 1.0000000e+00,
        2.0000000e+00, 3.4000000e+02, 1.0000000e+00],
       [1.5690772e+07, 1.0000000e+00, 2.7750000e+01, 1.2900000e+00,
        2.0000000e+00, 4.0000000e+00, 8.0000000e+00, 2.5000000e-01,
        0.0000000e+00, 0.0000000e+00, 0.0000000e+00, 1.0000000e+00,
        1.0000000e+00, 1.4000000e+02, 1.0000000e+00],
       [1.5783859e+07, 1.0000000e+00, 3.3580000e+01, 2.5000000e-01,
        2.0000000e+00, 3.0000000e+00, 5.0000000e+00, 4.0000000e+00,
        0.0000000e+00, 0.0000000e+00, 0.0000000e+00, 1.0000000e+00,
        1.0000000e+00, 4.2000000e+02, 1.0000000e+00],
       [1.5753550e+07, 1.0000000e+00, 3.5580000e+01, 7.5000000e-01,
        2.0000000e+00, 4.0000000e+00, 4.0000000e+00, 1.5000000e+00,
        0.0000000e+00, 0.0000000e+00, 0.0000000e+00, 1.0000000e+00,
        2.0000000e+00, 2.3100000e+02, 1.0000000e+00],
       [1.5747757e+07, 1.0000000e+00, 3.5000000e+01, 3.3750000e+00,
        2.0000000e+00, 8.0000000e+00, 8.0000000e+00, 8.2900000e+00,
        0.0000000e+00, 0.0000000e+00, 0.0000000e+00, 1.0000000e+00,
        2.0000000e+00, 0.0000000e+00, 1.0000000e+00],
       [1.5810485e+07, 1.0000000e+00, 3.4170000e+01, 2.7500000e+00,
        2.0000000e+00, 3.0000000e+00, 5.0000000e+00, 2.5000000e+00,
        0.0000000e+00, 0.0000000e+00, 0.0000000e+00, 1.0000000e+00,
        2.0000000e+00, 2.3200000e+02, 2.0100000e+02],
       [1.5702149e+07, 1.0000000e+00, 3.2750000e+01, 2.3350000e+00,
        2.0000000e+00, 2.0000000e+00, 8.0000000e+00, 5.7500000e+00,
        0.0000000e+00, 0.0000000e+00, 0.0000000e+00, 1.0000000e+00,
        2.0000000e+00, 2.9200000e+02, 1.0000000e+00],
       [1.5787229e+07, 1.0000000e+00, 2.1830000e+01, 1.5400000e+00,
        2.0000000e+00, 4.0000000e+00, 4.0000000e+00, 8.5000000e-02,
        0.0000000e+00, 0.0000000e+00, 0.0000000e+00, 1.0000000e+00,
        2.0000000e+00, 3.5600000e+02, 1.0000000e+00],
       [1.5759133e+07, 1.0000000e+00, 2.3580000e+01, 8.3500000e-01,
        2.0000000e+00, 3.0000000e+00, 8.0000000e+00, 8.5000000e-02,
        0.0000000e+00, 0.0000000e+00, 0.0000000e+00, 1.0000000e+00,
        2.0000000e+00, 2.2000000e+02, 6.0000000e+00],
       [1.5713160e+07, 1.0000000e+00, 2.7420000e+01, 1.2500000e+01,
        2.0000000e+00, 6.0000000e+00, 5.0000000e+00, 2.5000000e-01,
        0.0000000e+00, 0.0000000e+00, 0.0000000e+00, 1.0000000e+00,
        2.0000000e+00, 7.2000000e+02, 1.0000000e+00],
       [1.5790113e+07, 1.0000000e+00, 1.7500000e+01, 2.2000000e+01,
        3.0000000e+00, 1.0000000e+00, 7.0000000e+00, 0.0000000e+00,
        0.0000000e+00, 0.0000000e+00, 0.0000000e+00, 1.0000000e+00,
        3.0000000e+00, 4.5000000e+02, 1.0000100e+05],
       [1.5735330e+07, 1.0000000e+00, 2.6670000e+01, 1.4585000e+01,
        2.0000000e+00, 3.0000000e+00, 5.0000000e+00, 0.0000000e+00,
        0.0000000e+00, 0.0000000e+00, 0.0000000e+00, 1.0000000e+00,
        2.0000000e+00, 1.7800000e+02, 1.0000000e+00],
       [1.5758477e+07, 1.0000000e+00, 2.6920000e+01, 2.2500000e+00,
        2.0000000e+00, 3.0000000e+00, 5.0000000e+00, 5.0000000e-01,
        0.0000000e+00, 0.0000000e+00, 0.0000000e+00, 1.0000000e+00,
        2.0000000e+00, 6.4000000e+02, 4.0010000e+03]])
```

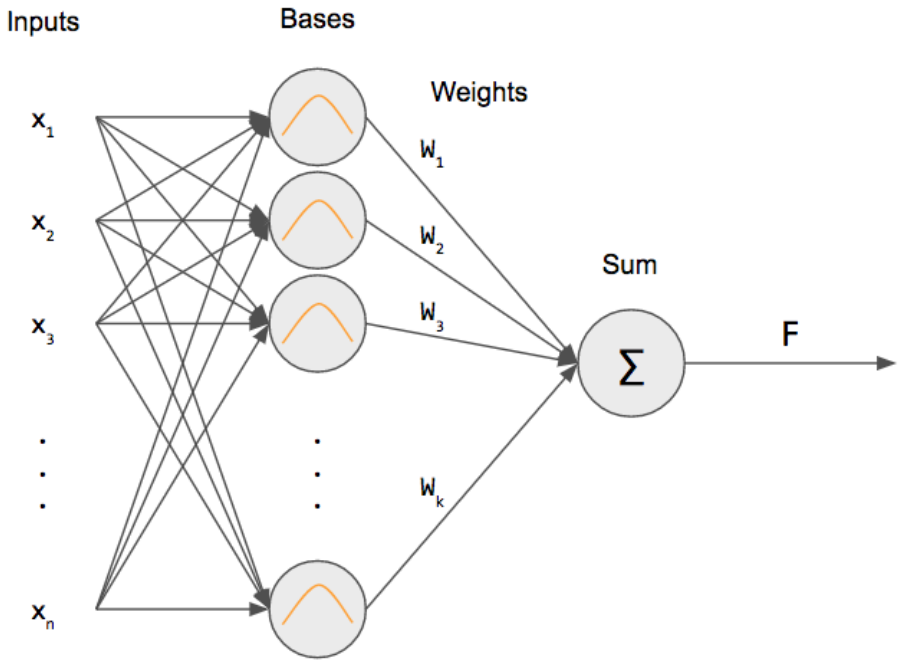
4. REDES NEURONALES DE FUNCION BASE RADIAL

Las redes neuronales de Función Base Radial son modelos de aprendizaje automático que actualmente no son especialmente muy utilizados, pero en cambio son rápidos, efectivos y se puede decir que incluso hasta intuitivos.

Como hemos visto en teoría, esta red neuronal cuenta de 3 capas y podrá utilizarse tanto en problemas de clasificación como regresión. Veremos un ejemplo de regresión. La regresión como sabéis es muy importante ya que tiene muchas aplicaciones en finanzas, física, biología y muchos otros campos.

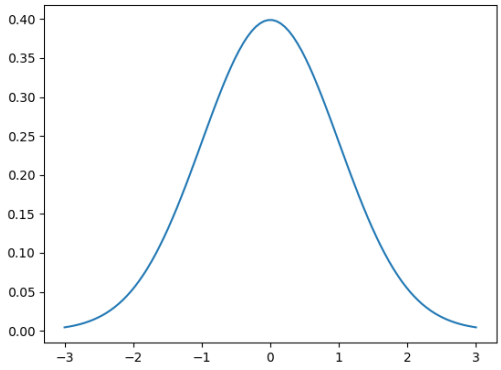
Supongamos que tenemos un conjunto de puntos de datos y queremos proyectar esta tendencia en el futuro para hacer predicciones. Las redes de funciones de base radial (redes RBF) se utilizan exactamente para este escenario: regresión o aproximación de funciones. Tenemos algunos datos que representan una tendencia o función subyacente y queremos modelarla. Las redes RBF pueden aprender a aproximar la tendencia subyacente utilizando muchas curvas gaussianas / campanas.

Recordemos que una red RBF es similar a una red de 2 capas. Tenemos una entrada que está completamente conectada a una capa oculta. Luego, tomamos la salida de la capa oculta y realizamos una suma ponderada para obtener nuestra salida.



A diferencia de una red neuronal “normal” como las que hemos visto hasta el momento, en la capa oculta nuestras neuronas tienen funciones gaussianas en la función de activación. La siguiente

figura muestra una función gaussiana, centrada en 0 y con una desviación standard de 1



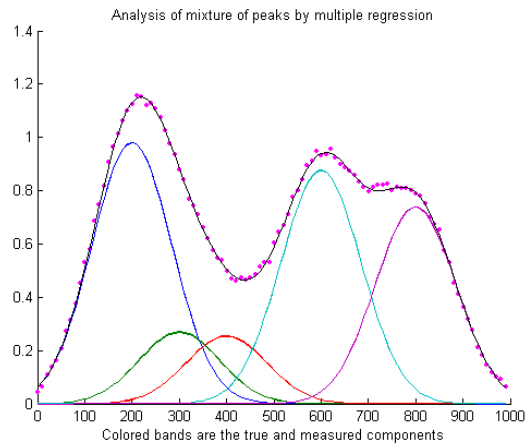
La función que describe la distribución normal es la siguiente:

$$\mathcal{N}(x;\mu,\sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Si observamos la ecuación vemos que hay una entrada y dos parámetros. Los dos parámetros son la media y la desviación estándar. En algunos casos la desviación estándar se sustituye por la varianza que es su cuadrado. La media cambia el centro gaussiano, es decir, la parte superior de la campana. En la imagen la media es 0, por lo que el mayor valor está en x=0.

La desviación estándar afecta la "amplitud" de la campana. El uso de una desviación estándar mayor significa que los datos están más dispersos, en lugar de estar más cerca de la media.

¿Por qué nos importan los gaussianos en las redes RBF? Nos interesan por que podemos aproximar una función como combinación lineal de gaussianas.



En la figura anterior, los gaussianas tienen diferentes colores y tienen un peso diferente. Cuando tomamos la suma obtenemos una función continua. Para hacer esto, necesitamos saber dónde colocar los centros de las funciones gaussianas y las desviaciones estándar de cada una de ellas

Podemos usar la agrupación k-means en nuestros datos de entrada para descubrir dónde colocar las gaussianas. El razonamiento detrás de esto es que queremos que nuestras gaussianas "abarquen" los grupos de datos más grandes ya que tienen esa forma de curva de campana.

El siguiente paso es descubrir cuáles deberían ser las desviaciones estándar. Podemos tomar dos enfoques como hemos visto en la presentación de teoría: establecer la desviación estándar para que sea la de los puntos asignados a un grupo particular c_j o podemos usar una única desviación estándar para todos los grupos.

¿Cuántos gaussianos usamos? Este es un hiperparámetro *k* llamado número de bases o núcleos.

Backpropagation para redes RBF

La agrupación K-means se usa para determinar los centros c_j para cada una de las funciones de base radial φ_j. Dada una entrada x, una red RBF produce una salida de suma ponderada.

$$F(x) = \sum_{j=1}^{k} w_j \varphi_j(x, c_j) + b$$

dónde w_j son los pesos, b es el sesgo , k es el número de bases o núcleos, y $\varphi_j(\cdot)$ es la función base radial gaussiana:

$$\varphi_j(x, c_j) = \exp\left(\frac{-||x - c_j||^2}{2\sigma_j^2}\right)$$

Usando estas definiciones, podemos derivar las reglas de actualización para w_j y b para el descenso del gradiente. Utilizamos la función de costo cuadrático para minimizar.

$$C = \sum_{i=1}^{N} (y^{(i)} - F(x^{(i)}))^2$$

Las reglas de retropropagación hacia atrás, en la capa de salida, serían igual que las que vimos en una Red Feedforward.

Keras no tiene implementación propia de estas redes por tanto hay que generar el código para que

podamos utilizarlo.

Código para implementar una Red Neuronal RBF

Vamos a implementar la función base radial:

```
def rbf(x, c, s):
    return np.exp(-1 / (2 * s**2) * (x-c)**2)
```

Ahora necesitaremos usar el algoritmo de agrupación k-means para determinar los centros de los cluster. El siguiente código nos da los centros de clúster y las desviaciones estándar de los clústeres.

```
def kmeans(X, k):
    """Performs k-means clustering for 1D input
        Arguments:
                X {ndarray} -- A Mx1 array of inputs
                k {int} -- Number of clusters

        Returns:
                ndarray -- A kx1 array of final cluster centers
    """

    # randomly select initial clusters from input data

    clusters = np.random.choice(np.squeeze(X), size=k)
    prevClusters = clusters.copy()
    stds = np.zeros(k)
    converged = False

    while not converged:
        """
        compute distances for each cluster center to each point
        where (distances[i, j] represents the distance between the ith point and jth
         cluster)
        """
        distances = np.squeeze(np.abs(X[:, np.newaxis] - clusters[np.newaxis, :]))

        # find the cluster that's closest to each point

        closestCluster = np.argmin(distances, axis=1)

        # update clusters by taking the mean of all of the points assigned
         # to that cluster

        for i in range(k):
            pointsForCluster = X[closestCluster == i]
            if len(pointsForCluster) > 0:
                clusters[i] = np.mean(pointsForCluster, axis=0)

        # converge if clusters haven't moved

        converged = np.linalg.norm(clusters - prevClusters) < 1e-6
        prevClusters = clusters.copy()

    distances = np.squeeze(np.abs(X[:, np.newaxis] - clusters[np.newaxis, :]))
    closestCluster = np.argmin(distances, axis=1)

    clustersWithNoPoints = []
```

```
    for i in range(k):
        pointsForCluster = X[closestCluster == i]
        if len(pointsForCluster) < 2:

            # keep track of clusters with no points or 1 point

            clustersWithNoPoints.append(i)
            continue
        else:
            stds[i] = np.std(X[closestCluster == i])

    # if there are clusters with 0 or 1 points, take the mean std
    # of the other clusters

    if len(clustersWithNoPoints) > 0:
        pointsToAverage = []
        for i in range(k):
            if i not in clustersWithNoPoints:
                pointsToAverage.append(X[closestCluster == i])
        pointsToAverage = np.concatenate(pointsToAverage).ravel()
        stds[clustersWithNoPoints] = np.mean(np.std(pointsToAverage))

    return clusters, stds
```

Este código implementa el algoritmo de clustering k-means y calcula las desviaciones estándar. Si hay un cluster con ninguno o uno de los puntos asignados, simplemente promediamos la desviación estándar de los otros grupos. (No podemos calcular la desviación estándar sin puntos de datos, y la desviación estándar de un solo punto de datos es 0).

Ahora implementaremos la clase RBF:

```
class RBFNet(object):

    """Implementation of a Radial Basis Function Network"""

    def __init__(self, k=2, lr=0.01, epochs=100, rbf=rbf, inferStds=True):
        self.k = k
        self.lr = lr
        self.epochs = epochs
        self.rbf = rbf
        self.inferStds = inferStds

        self.w = np.random.randn(k)
        self.b = np.random.randn(1)
```

Tenemos varios parámetros que elegir para la RBF: el número de bases, la tasa de aprendizaje, la cantidad de épocas, y si queremos usar las desviaciones estándar de k-means. También

inicializamos los pesos y el sesgo. Recuerda que una red RBF es una red de 2 capas modificada, por lo que solo hay un vector de pesos y un solo sesgo en el nodo de salida, ya que estamos aproximando una función 1D (específicamente, una salida). Si tuviéramos una función con múltiples salidas (una función con una salida con valor vectorial), usaríamos múltiples neuronas de salida y nuestros pesos serían una matriz y nuestro sesgo un vector.

Luego, tenemos que escribir nuestra función de ajuste para calcular nuestros pesos y sesgos. En las primeras líneas, usamos las desviaciones estándar del algoritmo k-medias modificado, o forzamos a todas las bases a usar la misma desviación estándar calculada a partir de la fórmula. El resto es similar a la propagación hacia atrás, donde propagamos nuestra entrada en el futuro y actualizamos nuestros pesos hacia atrás.

```
    def fit(self, X, y):

        if self.inferStds:

            # compute stds from data

            self.centers, self.stds = kmeans(X, self.k)

        else:

            # use a fixed std

            self.centers, _ = kmeans(X, self.k)
            dMax = max([np.abs(c1 - c2) for c1 in self.centers for c2 in s
elf.centers])
            self.stds = np.repeat(dMax / np.sqrt(2*self.k), self.k)

        # training

        for epoch in range(self.epochs):
            for i in range(X.shape[0]):
                # forward pass
                a = np.array([self.rbf(X[i], c, s) for c, s, in zip(self.c
enters, self.stds)])
                F = a.T.dot(self.w) + self.b

                loss = (y[i] - F).flatten() ** 2
                print('Loss: {0:.2f}'.format(loss[0]))

                # backward pass
                error = -(y[i] - F).flatten()

                # online update
                self.w = self.w - self.lr * a * error
                self.b = self.b - self.lr * error
```

Tened en cuenta que estamos imprimiendo el error en cada paso. También estamos realizando una actualización en línea, lo que significa que actualizamos nuestros pesos y sesgos para cada entrada. Alternativamente, podríamos haber realizado una actualización por lotes, donde actualizamos nuestros parámetros después de ver todos los datos de entrenamiento, o una actualización de minibatch, donde actualizamos nuestros parámetros después de ver un subconjunto de los datos de entrenamiento.

Hacer una predicción será tan simple como propagar nuestra entrada hacia adelante:

```
def predict(self, X):
        y_pred = []
        for i in range(X.shape[0]):
            a = np.array([self.rbf(X[i], c, s) for c, s, in zip(self.cente
rs, self.stds)])
            F = a.T.dot(self.w) + self.b
            y_pred.append(F)
        return np.array(y_pred)
```

Hay que tener en cuenta que estamos permitiendo una matriz de entradas, donde cada fila es un ejemplo.

Ya podemos utilizar nuestro código para un ejemplo. Para nuestros datos de entrenamiento, generaremos 100 muestras de la función seno. Luego, agregaremos un poco de ruido uniforme a nuestros datos.

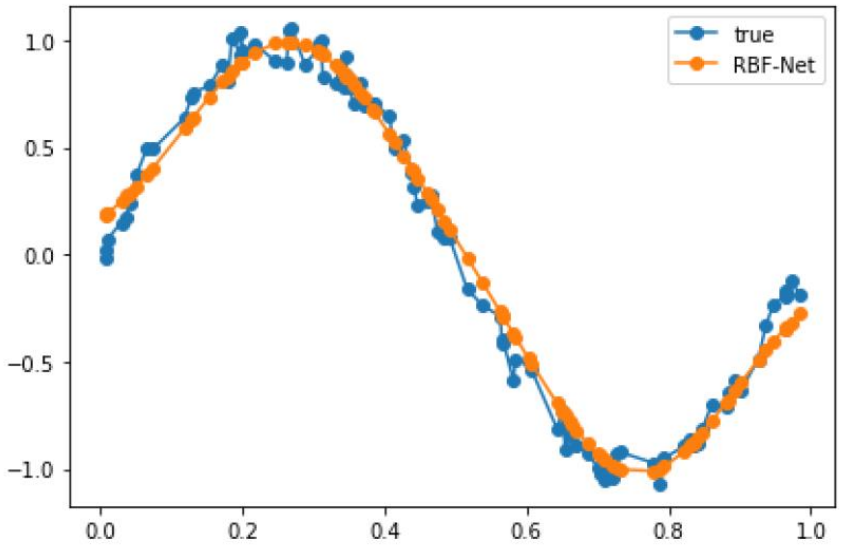
```
import numpy as np
import matplotlib.pyplot as plt

# sample inputs and add noise
NUM_SAMPLES = 100
X = np.random.uniform(0., 1., NUM_SAMPLES)
X = np.sort(X, axis=0)
noise = np.random.uniform(-0.1, 0.1, NUM_SAMPLES)
y = np.sin(2 * np.pi * X)  + noise

rbfnet = RBFNet(lr=1e-2, k=2)
rbfnet.fit(X, y)

y_pred = rbfnet.predict(X)
```

Dibujamos nuestra función aproximada para compararla con la real:



Podemos intentar jugar con algunos parámetros clave, como el número de bases. ¿Qué pasa si aumentamos el número de bases a 4? ¿Es mejor el resultado? ¿Qué está ocurriendo?

Otro parámetro que podemos cambiar es la desviación estándar. ¿Qué tal si usamos una única desviación estándar para todas nuestras bases en lugar de que cada una tenga su propia?

Lanzad varias ejecuciones con diferentes parámetros buscando la mejor configuración para la Red Neuronal RBF. Recogedlos en una tabla y sacad conclusiones.