Suponga que tenemos información sobre la obesidad, los hábitos de fumar y los hábitos de ejercicio de cinco personas. También sabemos si estas personas son diabéticas o no. Nuestro conjunto de datos se ve así:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Persona 1 | 0 | 1 | 0 | 1 |
| Persona 2 | 0 | 0 | 1 | 0 |
| Persona 3 | 1 | 0 | 0 | 0 |
| Persona 4 | 1 | 1 | 0 | 1 |
| Persona 5 | 1 | 1 | 1 | 1 |

En la tabla anterior, tenemos cinco columnas:

* Persona, Tabaquismo, Obesidad, Ejercicio y Diabético.
* Aquí 1 se refiere a verdadero y 0 a falso.
* Por ejemplo, la primera persona tiene valores de 0, 1, 0,
* Lo que significa que la persona no fuma, es obesa y no hace ejercicio.
* La persona también es diabética. (TARGET)

Nuestra tarea es crear una red neuronal que sea capaz de predecir si una persona es diabética o no dados datos sobre sus hábitos de ejercicio, obesidad y hábitos de tabaquismo.

Este es un tipo de problema de aprendizaje supervisado donde se nos dan entradas y las correspondientes salidas correctas y nuestra tarea es encontrar el mapeo entre las entradas y las salidas.

import numpy as np

feature\_set = np.array([[0,1,0],[0,0,1],[1,0,0],[1,1,0],[1,1,1]])

#SI TIENE O NO DIABETES

labels = np.array([[1,0,0,1,1]])

labels = labels.reshape(5,1)

#Los mismos valores aleatorios al ejecutar el script

np.random.seed(42)

#Inicializamos pesos con valores aleatorios

#Como son 3 caracteristicas (tabaquismo, obesidad, ejercicio) es vector de 3 pesos

weights = np.random.rand(3,1)

#Valor de sesgo con otro aleatorio

bias = np.random.rand(1)

#Tasa de aprendizaje

lr = 0.05

Definimos función de activación y su derivada, la sigmoidea

def sigmoid(x):

return 1/(1+np.exp(-x))

def sigmoid\_der(x):

return sigmoid(x)\*(1-sigmoid(x))

Ahora estamos listos para entrenar nuestra red neuronal que podrá predecir si una persona es obesa o no.

#EPOCA, CANTIDAD DE ITERACIONES, CON 20,000 SE REDUJO ERROR

for epoch in range(20000):

inputs = feature\_set #Set original de features a entrenar

# feedforward step1 PRODUCTO ESCALAR DE DATOS X PESOS Y LE AGREGAMOS BIAS

XW = np.dot(feature\_set, weights) + bias

#feedforward step2, PASAMOS PRODUCTO ESCALAR POR LA FUNCION DE ACTIVACION

**z** = **sigmoid**(XW)

#------PROPAGACION HACIA ATRAS ----

# backpropagation step 1 (SE TOMA EL TARGET Y SE IMPRIME FX COSTE

error = **z** - labels

print(error.sum())

# backpropagation step 2 (ERROR A DERIVADA)

dcost\_dpred = error

dpred\_dz = **sigmoid\_der**(**z**)

z\_delta = dcost\_dpred \* dpred\_dz

inputs = feature\_set.T

weights -= lr \* np.dot(inputs, z\_delta)

for num in z\_delta:

bias -= lr \* num

Ahora es el momento de comenzar la propagación hacia atrás. La variable zcontiene las salidas previstas. El primer paso de la propagación hacia atrás es encontrar el error. Lo hacemos en la siguiente línea:

error = z - labels

Sabemos que nuestra función de costos es:

MSE = frac {mathrm {1}} {mathrm {n}} sumnolimits\_ {i = 1} ^ {n} (predicho – observado) ^ {2}

Necesitamos diferenciar esta función con respecto a cada peso. Usaremos la [regla de diferenciación de](https://en.wikipedia.org/wiki/Chain_rule" \t "_blank) la [cadena](https://en.wikipedia.org/wiki/Chain_rule" \t "_blank) para este propósito. Supongamos que «d\_cost» es la derivada de nuestra función de costo con respecto al peso «w», podemos usar la regla de la cadena para encontrar esta derivada, como se muestra a continuación:

frac {d\_cost} {dw} = frac {d\_cost} {d\_pred}, frac {d\_pred} {dz}, frac {dz} {dw}

Aquí, frac {d\_cost} {d\_pred} se puede calcular como: 2 (previsto – observado)

Aquí, 2 es constante y, por tanto, puede ignorarse. Este es básicamente el error que ya calculamos. En el código, puede ver la línea:

dcost\_dpred = error # ........ (2)

A continuación tenemos que encontrar:

frac {d\_pred}{dz}

Aquí «d\_pred» es simplemente la función sigmoidea y la hemos diferenciado con respecto al producto escalar de entrada «z». En el script, esto se define como:

dpred\_dz = sigmoid\_der(z) # ......... (3)

Finalmente, tenemos que encontrar:

frac {d\_z} {dw}

Lo sabemos:

z = x1w1 + x2w2 + x3w3 + b

Por lo tanto, la derivada con respecto a cualquier peso es simplemente la entrada correspondiente. Por lo tanto, nuestra derivada final de la función de costo con respecto a cualquier peso es:

slope = input x dcost\_dpred x dpred\_dz

Eche un vistazo a las siguientes tres líneas:

z\_delta = dcost\_dpred \* dpred\_dz

inputs = feature\_set.T

weights -= lr \* np.dot(inputs, z\_delta)

Aquí tenemos la z\_deltavariable, que contiene el producto de dcost\_dpredy dpred\_dz. En lugar de recorrer cada registro y multiplicar la entrada por la correspondiente z\_delta, tomamos la transposición de la matriz de características de entrada y la multiplicamos por z\_delta. Finalmente, multiplicamos la variable de la tasa de aprendizaje lrpor la derivada para aumentar la velocidad de convergencia.

Luego recorrimos cada valor derivado y actualizamos nuestros valores de sesgo, como se muestra en este script:

Una vez que comience el ciclo, verá que el error total comienza a disminuir como se muestra a continuación:

0.001700995120272485

0.001700910187124885

0.0017008252625468727

. . . .

0.0016987049293782815

El error es extremadamente pequeño al final del entrenamiento de nuestra red neuronal.

En este momento, nuestros pesos y sesgos tendrán valores que pueden usarse para detectar si una persona es diabética o no, en función de sus hábitos de tabaquismo, obesidad y hábitos de ejercicio.

Ahora puede intentar predecir el valor de una sola instancia. Supongamos que tenemos un registro de un paciente que ingresa, fuma, no es obeso y no hace ejercicio. Veamos si es probable que sea diabético o no. La función de entrada se verá así: [1,0,0].

single\_point = np.array([1,0,0])

result = sigmoid(np.dot(single\_point, weights) + bias)

print(result)

En la salida verá:

[0.00707584]

Puede ver que es probable que la persona no sea diabética, ya que el valor está mucho más cerca de 0 que de 1.

¿Quiere aprender más sobre la creación de redes neuronales para resolver problemas complejos?

[Deep Learning AZ: redes neuronales artificiales prácticas](https://pharos.sh/deep-learning-hands-on-ann)

Cubre las redes neuronales con mucho más detalle, incluidas las redes neuronales convolucionales, las redes neuronales recurrentes y mucho más.

**Conclusión**

En este artículo creamos una red neuronal muy simple con una entrada y una capa de salida desde cero en Python. Esta red neuronal se llama simplemente [perceptrón](https://en.wikipedia.org/wiki/Perceptron" \t "_blank) . Un perceptrón puede clasificar datos separables linealmente. Los datos linealmente separables son el tipo de datos que pueden separarse mediante un hiperplano en un espacio n-dimensional.

Las redes neuronales artificiales de palabras reales son mucho más complejas, poderosas y consisten en múltiples capas ocultas y múltiples nodes en la capa oculta. Estas redes neuronales pueden identificar límites de decisión reales no lineales. Explicaré cómo crear una red neuronal multicapa desde cero en Python en un próximo artículo.

import numpy as np

feature\_set = np.array([[0,1,0],[0,0,1],[1,0,0],[1,1,0],[1,1,1]])

#TARGET, SI TIENE O NO DIABETES

labels = np.array([[1,0,0,1,1]])

labels = labels.reshape(5,1)

#Los mismos valores aleatorios al ejecutar el script

np.random.seed(42)

#Inicializamos pesos con valores aleatorios

#Como son 3 caracteristicas (tabaquismo, obesidad, ejercicio) es vector de 3 pesos

weights = np.random.rand(3,1) #3 numeros de 0-1

#Valor de sesgo con otro aleatorio

bias = np.random.rand(1) **# 0-1**

#Tasa de aprendizaje

lr = 0.05

def sigmoid(x):

    return 1/(1+np.exp(-x))

def sigmoid\_der(x):

    return sigmoid(x)\*(1-sigmoid(x))

#EPOCA, CANTIDAD DE ITERACIONES, CON 20,000 SE REDUJO ERROR

for epoch in range(20000):

    inputs = feature\_set  #Set original de features a entrenar

    # feedforward step1 PRODUCTO ESCALAR DE DATOS X PESOS Y LE AGREGAMOS BIAS

    XW = np.dot(feature\_set, weights) + bias

    #feedforward step2, PASAMOS PRODUCTO ESCALAR POR LA FUNCION DE ACTIVACION

    z = sigmoid(XW)

    #------PROPAGACION HACIA ATRAS ----

    # backpropagation step 1 (SE TOMA EL TARGET Y SE IMPRIME FX COSTE

    error = z - labels

    print(error.sum())

    # backpropagation step 2 (ERROR A DERIVADA)

    dcost\_dpred = error

    dpred\_dz = sigmoid\_der(z)

    z\_delta = dcost\_dpred \* dpred\_dz

    inputs = feature\_set.T

    weights -= lr \* np.dot(inputs, z\_delta)

    for num in z\_delta:

        bias -= lr \* num

single\_point = np.array([1,0,0])

result = sigmoid(np.dot(single\_point, weights) + bias)

print(result)