Vuelven las minas y las rocas

Haga memoria: en el capítulo Clasificar bien no es una opción, tratamos el caso de la clasificación de un objeto como una mina o una roca en función de las mediciones efectuadas por un sonar.

Este caso de estudio también se trató en un artículo titulado «Analysis of Hidden Units in a Layered Network Trained to Classify Sonar Targets» («Análisis de unidades ocultas en una red multicapa entrenada para clasificar objetivos de sonar»), escrito por Terry Sejnowski y R. Paul Gorman en 1988. Su objetivo era **demostrar la importancia del papel de la cantidad de neuronas en la capa oculta sobre la precisión de clasificación**.

Por supuesto, este estudio se puede descargar en la web, pero hemos puesto una copia a su disposición dentro de la carpeta destinada al código de este capítulo, que puede descargarse desde el sitio del editor.

Ahora vamos a ver los distintos parámetros utilizados por estos dos científicos para realizar su demostración:

* 192 datos de aprendizaje.
* 16 datos de pruebas.
* 60 neuronas de entrada (correspondientes a cada una de las frecuencias).
* 2 neuronas de salida (una por la probabilidad correspondiente a una mina y otra por la probabilidad correspondiente a una roca).
* 300 epochs.
* Una tasa de aprendizaje de 2,0.
* Pesos inicializados en el intervalo de valores comprendidos entre -0,3 y 0,3.
* Una actualización de los pesos solo en caso de que el valor predicho difiera más o menos 0,2 con el esperado.
* Uso de una función de activación de tipo sigmoide.
* Una capa de neuronas compuesta de 0, 2, 3, 12 o 24 neuronas.

En este capítulo vamos a intentar conseguir los resultados que obtuvieron durante el estudio, pero nosotros también trataremos de demostrar que la cantidad de neuronas en una capa oculta influye sobre la precisión de la clasificación.

1. ¿Más neuronas en la capa oculta dan lugar a mejores resultados?

Antes de seguir, le invitamos a crear un proyecto de Python nuevo con una carpeta datas donde colocaremos los datos de las observaciones que ya utilizamos en el capítulo Clasificar bien no es una opción.

**a. Carga de datos de aprendizaje**

Una vez hecho esto, podemos cargar las observaciones de la siguiente manera:

#---------------------------------------------

# CARGA DE LAS OBSERVACIONES

#---------------------------------------------

import pandas as pnd

observaciones = pnd.read\_csv("datas/sonar.all-data.csv")

Se extraen y a continuación se transforman los datos necesarios para el aprendizaje y las pruebas. Hay que destacar la creación de clases que corresponderán a los valores esperados por las dos neuronas de salida. Si es una mina, el valor esperado en la neurona dedicada a la predicción del objeto Mina será 1, y 0 en la dedicada a la predicción del objeto Roca.

#Para el aprendizaje solo se toman los datos procedentes del sonar

X = observaciones[observaciones.columns[0:60]].values

#Solo se toman los etiquetados

y = observaciones[observaciones.columns[60]]

#Se codifica: Las minas son iguales a 0 y las rocas son iguales a 1

from sklearn.preprocessing import LabelEncoder

encoder = LabelEncoder()

encoder.fit(y)

y = encoder.transform(y)

#Se añade un cifrado para crear clases:

# Si es una mina [1,0]

# Si es una roca [0,1]

import numpy as np

n\_labels = len(y)

n\_unique\_labels = len(np.unique(y))

one\_hot\_encode = np.zeros((n\_labels,n\_unique\_labels))

one\_hot\_encode[np.arange(n\_labels),y] = 1

Y=one\_hot\_encode

#Verificación tomando los registros 0 y 97

print("Clase Roca:",int(Y[0][1]))

print("Clase Mina:",int(Y[97][1]))

**b. Creación de los conjuntos de aprendizaje y de pruebas**

A continuación creamos los conjuntos de aprendizaje y de pruebas conforme a los recomendados por Terry Sejnowski y R. Paul Gorman, es decir, 192 observaciones de aprendizaje y 16 observaciones de pruebas (0,07 %).

#Mezclamos las observaciones

from sklearn.utils import shuffle

X, Y = shuffle(X, Y, random\_state=1)

#Creación de los conjuntos de aprendizaje y de las pruebas

from sklearn.model\_selection import train\_test\_split

train\_x, test\_x, train\_y, test\_y = train\_test\_split

(X, Y, **test\_size=0.07**, random\_state=42)

**c. Parametrización de la red neuronal con una capa oculta de 24 neuronas**

A continuación viene la fase de parametrización de la red neuronal con la creación de tuplas pesos y peso\_sesgo para la parametrización de los distintos pesos y sesgos de las capas de entrada, de la capa oculta y de la capa de salida.

Aquí podemos ver la cantidad de neuronas utilizadas para cada capa:

* Cantidad de neuronas presentes en la capa de entrada: 60.
* Cantidad de neuronas presentes en la capa de oculta: 24.
* Cantidad de neuronas en salida: 2

Los valores de los pesos se generan en el interior del intervalo [-0.3, 0.3] como recomienda el estudio. Esta generación se hace utilizando la función tf.random.uniform. Esta función también toma como parámetro una lista que define la cantidad de neuronas de la capa de entrada y la cantidad de neuronas de la capa de salida. Así [60,24] significa que hay que generar pesos para 60 neuronas conectadas a 24 neuronas, es decir, un total de 1440 pesos (60\*24).

import tensorflow as tf

epochs = 300

cantidad\_neuronas\_entrada = 60

cantidad\_neuronas\_salida = 2

cantidad\_neuronas\_capa\_oculta = **24**

tasa\_aprendizaje = 0.01

#Variable TensorFLow correspondiente a los 60 valores de las

#neuronas de entrada

tf\_neuronas\_entradas\_X = tf.placeholder(tf.float32,[None, 60])

#Variable TensorFlow correspondiente a las 2 neuronas de salida

tf\_valores\_reales\_Y = tf.placeholder(tf.float32,[None, 2])

pesos = {

   # 60 neuronas de entradas hacia 24 Neuronas de la capa oculta

   'capa\_entrada\_hacia\_oculta':

tf.Variable(tf.random\_uniform([60, **24**], **minval=-0.3, maxval=0.3**),

tf.float32),

   #24 neuronas de la capa oculta hacia 2 de la capa

   #de salida

   'capa\_oculta\_hacia\_salida':

tf.Variable(tf.random\_uniform([**24**, 2], **minval=-0.3, maxval=0.3**),

tf.float32),

}

peso\_sesgo = {

    #1 sesgo de la capa de entrada hacia las 24 neuronas

    #de la capa oculta

   'peso\_sesgo\_capa\_entrada\_hacia\_oculta':

tf.Variable(tf.zeros([**24**]), tf.float32),

   #1 sesgo de la capa oculta hacia las 2 neuronas de la capa

   #de salida

   'peso\_sesgo\_capa\_oculta\_hacia\_salida':

tf.Variable(tf.zeros([2]), tf.float32),

}

A continuación definimos una función que se encarga de crear la red neuronal. Recuperamos la definición de las distintas funciones de activación de tipo sigmoide de cada capa prestando atención a utilizar correctamente los pesos adecuados en relación con la capa afectada. Esta función nos devuelve el modelo que podremos utilizar a continuación en la fase de pruebas para verificar el aprendizaje correcto de la red neuronal.

def red\_neuronas\_multicapa(observaciones\_en\_entradas, pesos,

peso\_sesgo):

   #Cálculo de la activación de la primera capa

   primera\_activacion =

**tf.sigmoid**(tf.matmul(tf\_neuronas\_entradas\_X,

pesos['capa\_entrada\_hacia\_oculta']) +

peso\_sesgo['peso\_sesgo\_capa\_entrada\_hacia\_oculta'])

   #Cálculo de la activación de la segunda capa

   activacion\_capa\_oculta = 

**tf.sigmoid**(tf.matmul(primera\_activacion,

pesos['capa\_oculta\_hacia\_salida']) +

peso\_sesgo['peso\_sesgo\_capa\_oculta\_hacia\_salida'])

   return activacion\_capa\_oculta

A continuación, la creación de la red neuronal propiamente dicha:

red = red\_neuronas\_multicapa(tf\_neuronas\_entradas\_X,

pesos, peso\_sesgo)

Después es el turno de la especificación de la función de error y de la función de optimización:

#Función de error de media cuadrática MSE

funcion\_error = tf.reduce\_sum(tf.pow(tf\_valores\_reales\_Y-

red,2))

#Descenso de gradiente con una tasa de aprendizaje fijada a 0,01

optimizador =

tf.train.GradientDescentOptimizer(learning\_rate=tasa\_aprendizaje)

.minimize(funcion\_error)

**d. Realización del aprendizaje**

#Inicialización de las variables

init = tf.global\_variables\_initializer()

#Inicio de una sesión de aprendizaje

session = tf.Session()

session.run(init)

#Para la realización del gráfico para la MSE

Grafica\_MSE=[]

#Para cada epoch

for i in range(epochs):

  #Realización del aprendizaje con actualización de los pesos

  session.run(optimizador, feed\_dict = {tf\_neuronas\_entradas\_X:

train\_x, tf\_valores\_reales\_Y:train\_y})

  #Calcular el error

  MSE = session.run(funcion\_error, feed\_dict =

{tf\_neuronas\_entradas\_X: train\_x, tf\_valores\_reales\_Y:train\_y})

  #Visualización de la información

  Grafica\_MSE.append(MSE)

  print("EPOCH (" + str(i) + "/" + str(epochs) + ") -  MSE: "+

str(MSE))

#Visualización gráfico de la MSE

import matplotlib.pyplot as plt

plt.plot(Grafica\_MSE)

plt.ylabel('MSE')

plt.show()

**e. Cálculo de la precisión del aprendizaje**

Antes de sumergirnos en los detalles del programa, vamos a ver cómo se evalúan las distintas clasificaciones realizadas por la red neuronal:

Una vez realizado el aprendizaje, las probabilidades de cada una de las clasificaciones efectuadas se guardan en el modelo:

Clasificación 1 [0.56, 0.89].

Clasificación 2 [0.90, 0.34].

Para cada clasificación, vamos a recuperar el índice de la probabilidad con un valor mayor usando la función tf.argmax.

Clasificación 1 [0.56, **0.89**] el índice es igual a 1.

Clasificación 2 [**0.90**, 0.34] el índice es igual a 0.

Recordamos que las minas se definen según los valores de probabilidades [1,0] y las rocas según los valores [0,1], lo que en términos del índice se traduce con los siguientes valores:

Mina [**1**,0] el índice es igual a 0.

Roca [0,**1**] el índice es igual a 1.

En el marco de una clasificación, esperábamos que la red clasificara los valores de las frecuencias suministradas en entrada como una mina:

* Índice esperado 0.
* Probabilidades de clasificaciones realizadas: [0.56, **0.89**].
* Índice de la probabilidad más elevada: 1.

Como el valor 1 (índice de la clasificación realizada) es distinto de 0 (índice de la clasificación esperada), entonces se trata de una clasificación errónea.

Ahora imaginemos una clasificación nueva:

* Índice esperado 0 (caso de una mina).
* Probabilidades de clasificaciones realizadas: [**0.90**, 0.34].
* Índice de la probabilidad más elevada: 0.

Como el valor 0 (índice de la clasificación realizada) es igual a 0 (índice de la clasificación esperada), entonces se trata de una clasificación correcta.

A fin de calcular la precisión del algoritmo sobre un conjunto de datos específico (aprendizaje, pruebas o el conjunto de los datos), para cada clasificación contenida en este conjunto de datos vamos a verificar si corresponde a las clasificaciones esperadas para poder deducir la cantidad de errores y calcular la precisión global.

Aquí podemos ver ahora este principio transformado con ayuda de algunas líneas de código:

#Recuperación de los índices de las clasificaciones realizadas

**clasificaciones = tf.argmax(red, 1)**

#Comparamos los índices procedentes de las clasificaciones con los

#esperados para conocer la cantidad de clasificaciones correctas

**formula\_calculo\_clasificaciones\_correctas =**

**tf.equal(clasificaciones,**

**tf.argmax(tf\_valores\_reales\_Y,1))**

#A continuación calculamos la precisión haciendo la media (tf.mean)

#de las clasificaciones correctas (después de haberlas convertido en

decimales tf.cast, tf.float32)

**formula\_precision =**

**tf.reduce\_mean(tf.cast(formula\_calculo\_clasificaciones\_correctas,**

**tf.float32))**

#----------------------------------------------------------------

# PRECISIÓN EN LOS DATOS DE PRUEBAS

#----------------------------------------------------------------

num\_clasificaciones = 0;

num\_clasificaciones\_correctas = 0

#Miramos todo el conjunto de los datos de pruebas (text\_x)

for i in range(0,test\_x.shape[0]):

   #Recuperamos la información de los datos de aprendizaje

   # test\_x a los que cambiamos el formato a una línea compuesta de

60 columnas para los datos procedentes del sonar y en una línea y

dos columnas para la clasificación esperada

**datosSonar = test\_x[i].reshape(1,60)**

**clasificacionEsperada = test\_y[i].reshape(1,2)**

   # Hacemos la clasificación

**prediction\_run = session.run(clasificaciones,**

**feed\_dict={tf\_neuronas\_entradas\_X:datosSonar})**

   #Se calcula la precisión de la clasificación con ayuda de la

fórmula antes establecida

**accuracy\_run = session.run(formula\_precision,**

**feed\_dict={tf\_neuronas\_entradas\_X:datosSonar,**

**tf\_valores\_reales\_Y:clasificacionEsperada})**

   #Se muestra para observación la clase original y la

**clasificación realizada**

**print(i,"Clase esperada: ",**

**int(session.run(tf\_valores\_reales\_Y [i][1],feed\_dict={tf\_valores\_**

**reales\_Y:test\_y})), "Clasificación: ", prediction\_run[0] )**

**num\_clasificaciones = num\_clasificaciones+1**

**if(accuracy\_run\*100 ==100):**

**num\_clasificaciones\_correctas =**

**num\_clasificaciones\_correctas+1**

**print("-------------")**

**print("Precisión en los datos de pruebas =**

**"+str((num\_clasificaciones\_correctas/num\_clasificaciones)\*100)+"%")**

Si ejecutamos el script ahora, podemos ver que la precisión del algoritmo es del 86 % en el conjunto de datos.

Clase esperada:  1  Clasificación:  1

Clase esperada:  0  Clasificación:  0

Clase esperada:  1  Clasificación:  1

**Clase esperada:  1  Clasificación:  0 - Error**

Clase esperada:  0  Clasificación:  0

Clase esperada:  1  Clasificación:  1

Clase esperada:  0  Clasificación:  0

**Clase esperada:  1  Clasificación:  0 - Error**

**Clase esperada:  1  Clasificación:  0 - Error**

Clase esperada:  0  Clasificación:  0

Clase esperada:  0  Clasificación:  0

Clase esperada:  0  Clasificación:  0

Clase esperada:  0  Clasificación:  0

Clase esperada:  0  Clasificación:  0

Clase esperada:  0  Clasificación:  0

-------------

Precisión en los datos de pruebas = 80.0%

Dentro del programa dedicado a este capítulo que puede descargar en el sitio del editor, también hemos programado los cálculos de precisión con los datos de aprendizaje y el conjunto de los datos, para verificar que no estamos en un caso de sobreajuste.

Precisión en los datos de pruebas = 80.0%

Precisión en los datos de aprendizaje = 79.6875%

Precisión en el conjunto de los datos = 79.71014492753623%

Como no hay una desviación importante entre la precisión obtenida durante el aprendizaje y la validación, no estamos en un caso de sobreajuste.

**f. ¿Una capa oculta de 24 neuronas da lugar a mejores resultados?**

Veamos si obtenemos mejores resultados con 12 lugar en vez de 24. Para ello, vamos a modificar un poco nuestro programa:

pesos = {

   #60 neuronas de entrada hacia las 12 neuronas de la capa

   #oculta

   'capa\_entrada\_hacia\_oculta':

tf.Variable(tf.random\_uniform([60, **12**], minval=-0.3,

maxval=0.3),

tf.float32),

   #12 neuronas de la capa oculta hacia las 2 neuronas de la

   #capa de salida

   'capa\_oculta\_hacia\_salida':

tf.Variable(tf.random\_uniform([**12**, 2], minval=-0.3, maxval=0.3),

tf.float32),

}

peso\_sesgo = {

    #1 sesgo de la capa de entrada hacia las 12 neuronas de la

    #capa oculta

   'peso\_sesgo\_capa\_entrada\_hacia\_oculta':

tf.Variable(tf.zeros([**12**]), tf.float32),

   #1 sesgo de la capa oculta hacia las 2 neuronas de la capa

   #de salida

   'peso\_sesgo\_capa\_oculta\_hacia\_salida':

tf.Variable(tf.zeros([2]), tf.float32),

}

Y esta vez, después de haber ejecutado nuestro programa, obtendremos una precisión de clasificación del 81 % en el conjunto de los datos, lo que es superior a la que hemos obtenido antes.

Precisión en los datos de pruebas = 93.33333333333333%

Precisión en los datos de aprendizaje = 80.72916666666666%

Precisión en el conjunto de los datos = 81.64251207729468%

Para cada uno de los dos casos (uso de 12 o de 24 neuronas), le invitamos a realizar varios aprendizajes que, como podrá comprobar, darán valores diferentes. Esto es debido a los distintos factores de inicialización y de aprendizaje elegidos de manera aleatoria (pesos, datos de aprendizaje…).

A la vista de estos resultados podemos afirmar que estamos de acuerdo con la conclusión del estudio, es decir, que la cantidad de neuronas utilizadas en una o varias capas ocultas influye en los resultados.

The results of the network classification experiments,

as well as the analysis of variance, **demonstrate that the**

**hidden layer contributed significantly** to the performance

of the network classifier

**g. ¿Podemos obtener mejores resultados?**

Ahora vamos a ver si podemos obtener mejores resultados modificando la configuración de nuestra red neuronal.

La pregunta que nos hacemos a menudo es: ¿cuántas capas de neuronas necesitamos y cuántas neuronas deben contener?

La respuesta más habitual a esta pregunta es que en muchos casos una sola capa oculta es suficiente para resolver la mayoría de los problemas de clasificación de los datos no lineales. Sobre la cantidad de neuronas contenidas en esta capa, esa cifra corresponde a la media de la cantidad de neuronas en la entrada y la cantidad de neuronas en la salida. Sin embargo, podemos afirmar que no existe un método de cálculo que permita identificar claramente la cantidad de capas ocultas necesarias y su cantidad de neuronas.

Todo depende del problema de clasificación que tengamos que resolver.

Para nuestro caso vamos a probar varias mejoras.

La primera consistirá en modificar la cantidad de epochs, aumentándola de 300 a 600.

epochs = 600

Después utilizaremos 26 neuronas en la capa oculta y modificaremos la función de generación de los pesos. Esta vez no tomaremos un intervalo de valor, sino que se generarán en forma de distribución de la ley normal (la famosa curva en forma de campana):

pesos =

   #60 neuronas de entrada hacia 24 neuronas de la capa oculta

   'capa\_entrada\_hacia\_oculta':

tf.Variable(**tf.random\_normal**([60, 26]), tf.float32),

   #26 neuronas de la capa oculta hacia 2 de la capa de

salida

   'capa\_oculta\_hacia\_salida' :

tf.Variable(tf.random\_normal([26, 2]), tf.float32),

}

peso\_sesgo = {

    #1 sesgo de la capa de entrada hacia las 24 neuronas de la

    #capa oculta

   'peso\_sesgo\_capa\_entrada\_hacia\_oculta':

tf.Variable(tf.zeros([26]), tf.float32),

   #1 sesgo de la capa oculta hacia las 2 neuronas de la capa

   #de salida

   'peso\_sesgo\_capa\_oculta\_hacia\_salida':

tf.Variable(tf.zeros([2]), tf.float32),

}

Atención: aumentar la cantidad de epochs también implica el aumento del riesgo de sobreajuste.

Esto nos permite obtener una precisión del 94 % en el conjunto de los datos. Una precisión superior a la que hemos obtenido hasta ahora.

Precisión en los datos de pruebas = 80.95238095238095%

Precisión en los datos de aprendizaje = 97.57575757575758%

Precisión en el conjunto de los datos = 94.20289855072464%

Es difícil saber exactamente la cantidad de neuronas que debe haber en cada una de las capas ocultas. Sin embargo, parece que ha surgido una «norma» indicando que la cantidad de neuronas de la capa oculta debe ser igual a la media entre la cantidad de neuronas de entrada y la cantidad de neuronas en la salida. Es decir, en nuestro caso 31 ((60+2)/2).

Si aplicamos esta norma, obtenemos un resultado de precisión en el conjunto de los datos ligeramente inferior. Destacamos el hecho de que la precisión de los datos de aprendizaje es superior alrededor de un 12 % a la obtenida en los datos de pruebas. Es posible que el modelo encuentre dificultades para generalizarse (sobreajuste).

Precisión en los datos de pruebas = 83.33333333333334%

Precisión en los datos de aprendizaje = 95.15151515151516%

Precisión en el conjunto de los datos = 92.7536231884058%