



Universidad Nacional Autónoma de México
Facultad de Estudios Superiores Aragón



Proyecto Tercer Parcial

Equipo 2

Integrantes:

Monzón Lucero Miguel Ángel

Robles Leon Cristopher Ruben

Grupo: 1509

Materia:

Diseño y Análisis de Algoritmos

Profesor:

Miguel Angel Sanchez Hernandez

Índice

Índice	1
Introducción	3
Objetivos	3
Análisis Algoritmo “Perceptrones Keras”	4
Objetivo	4
Restricciones	4
Modelo	4
Descripción del Algoritmo	4
Números Impares	4
Entrada	5
Salida	6
Código	7
Números Pares	8
Entrada	8
Salida	10
Código	11
Números Mayores o Iguales a 5	12
Entrada	12
Salida	13
Código	15
Imágenes Aceptadas	15
Entrada	16
Salida	18
Código	19
Compuerta OR	20
Entrada	20
Salida	21
Código	21
Compuerta XOR	22
Entrada	22
Salida	23
Código	24
Pseudocódigo	25
Bloque para leer los datos de un archivo de texto/csv.	25
Bloque para definir y entrenar nuestra red.	25
Conclusiones	25
Reconocimiento de números hechos a mano con k-nn	26
Objetivo	26
Restricciones	26
Modelo	26
Descripción del algoritmo.	26

Código	26
Referencias	29

Introducción

En este trabajo se presenta la documentación de los algoritmos vistos para este parcial, y una investigación e implementación de una variante de los algoritmos.

La primera parte de esta documentación se centra en el análisis de un solo algoritmo que va ir variando en unos pocos detalles para obtener los resultados deseados. Se hará un análisis general del algoritmo “principal” y posteriormente se especificará en qué cambia el algoritmo en cada caso solicitado.

Para la segunda parte de la documentación se hará una investigación sobre cómo podemos reconocer números escritos a mano con ayuda de un algoritmo e identificar cuando se trata de un número u otro.

Objetivos

- Como objetivo principal queremos que nuestras implementaciones, den los resultados deseados y logremos transmitir de manera correcta lo que entendemos por redes neuronales; y no solo se quede en que nosotros entendemos de qué se trata sin poder transmitir ese conocimiento.
- Nuestro segundo objetivo está marcado por llegar a resolver de manera satisfactoria la red neuronal que reconoce imágenes de números escritos a mano.
- Como último objetivo, pero no por eso el menos importante, marcamos como meta el poder comprender de mejor manera este tema de las redes neuronales y poder aprender aplicarlas en más campos y para diferentes objetivos.

Análisis Algoritmo “Perceptrones Keras”

Objetivo

El objetivo de este algoritmo es poder entrenar una red neuronal para que pueda reconocer datos, imágenes, números y pueda clasificarlos de manera eficiente. Con esto queremos decir que nos indique si un dato, una imagen o un número entra en un grupo o no.

Restricciones

Teóricamente este algoritmo no tiene restricciones en los datos que puede procesar, pero para que esto se cumpla se tiene que analizar bien cómo es que se van a interpretar los datos que le daremos y cómo interpretaremos las salidas que nos devuelva. En particular en nuestro algoritmo se le ingresarán archivos de datos ordenados.

Modelo

Entradas:

Para el caso concreto de este algoritmo, recibe archivos de txt o csv donde nosotros organizaremos los datos de acuerdo a una interpretación concreta.

Proceso:

1. Primero el algoritmo lee el archivo con los datos y estos los asignamos a variables de entrada y salida.
2. Después aplicamos el algoritmo para hacer nuestra red neuronal.
3. Posteriormente entrenamos nuestra red neuronal con los datos de entrada y salida que obtuvimos de archivo.
4. Comprobamos que nuestra red neuronal esté bien entrenada.

Descripción del Algoritmo

El algoritmo se basa principalmente en el entrenamiento de una red neuronal.

La primera parte del algoritmo lo que hace es entrenar nuestra red neuronal con los datos que le agregamos. Esto lo que hace es simplificar la labor de calcular los pesos y un bias, que nos servirá para poder sacar la frontera de decisión y poder clasificar los datos. Concretamente nos calcula el peso de cada entrada, y el bias que debemos ocupar para obtener esta frontera de decisión. Estos cálculos los va haciendo por épocas; ya que va dando números aleatorios para calcular los pesos y el bias y nos va mostrando cuánta precisión tiene durante cada época. Por lo que al final cuando se obtiene un 100% significa que nuestra red está funcionando de manera correcta de acuerdo a los datos que le proporcionamos.

Este cálculo va depender de cómo es que diseñamos nuestra red neuronal, ya que si solo usamos 1 neurona, únicamente nos dará un peso por entrada y un bias correspondiente a esos pesos, pero si los datos nos piden que usemos más neuronas, el algoritmo nos dará más pesos por entrada y un bias por cada par de pesos.

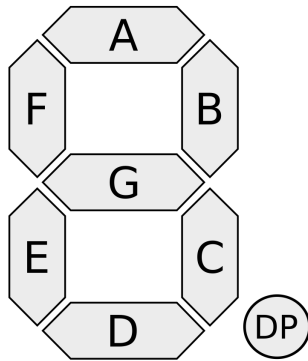
La segunda parte del algoritmo lo único que hace es mostrarnos los pesos obtenidos, el bias obtenido, los resultados que deseamos obtener por cada entrada, y la matriz de confusión; esto con el único propósito de comprobar que este funcionando correctamente el algoritmo.

Números Impares

Para este caso se hizo el diseño de la red neuronal tomando en cuenta que los números que tiene que comparar provienen de un display de 7 segmentos; por lo que la cantidad de entradas que recibió nuestro algoritmo fueron 7, y únicamente nos da una salida, que es si nos dice si es o no es impar.

Entrada

Este es el diseño básico del display de 7 segmentos en la siguiente tabla indicamos cómo es que tomamos en cuenta nuestras entradas.



A	B	C	D	E	F	G	Número Decimal	1 = Impar
1	1	1	1	1	1	0	0	0
0	1	1	0	0	0	0	1	1
1	1	0	1	1	0	1	2	0
1	1	1	1	0	0	1	3	1
0	1	1	0	0	1	1	4	0
1	0	1	1	0	1	1	5	1
1	0	1	1	1	1	1	6	0
1	1	1	0	0	0	1	7	1
1	1	1	1	1	1	1	8	0
1	1	1	0	0	1	1	9	1

Este es el contenido del archivo que lee el algoritmo para obtener los datos, los primeros 7 elementos de cada fila corresponden a las entradas y el último corresponde a si es o no impar, 0 cuando no es impar y 1 cuando es impar.

```

1 1,1,1,1,1,1,0,0
2 0,1,1,0,0,0,0,1
3 1,1,0,1,1,0,1,0
4 1,1,1,1,0,0,1,1
5 0,1,1,0,0,1,1,0
6 1,0,1,1,0,1,1,1
7 1,0,1,1,1,1,1,0
8 1,1,1,0,0,0,0,1
9 1,1,1,1,1,1,1,0
10 1,1,1,0,0,1,1,1

```

Las siguientes líneas de código son las encargadas de delimitar cómo es que tomará en cuenta los datos del archivo. Del primer elemento al penúltimo serán las entradas y el último será la salida deseada:

```
datos_entrada = conjunto_datos.iloc[:,0:7].values
```

```
datos_salida = conjunto_datos.iloc[:,7:8].values
```

También le indicaremos cómo será el diseño de nuestra red neuronal que en este caso será una red de 16 neuronas y le indicamos que realice 400 épocas

Salida

La salida que esperamos es la siguiente, de acuerdo con nuestra matriz de confusión:

	0 (Cantidad de 0 esperados)	1 (Cantidad de 1 esperados)
0 (Obtenidos)	5	0
1 (Obtenidos)	0	5

Los bias podrían corroborarse manualmente con excel, pero el propósito de este algoritmo es ahorrarnos esa parte además que sería demasiado trabajo manual. Por lo que nos basaremos únicamente en la matriz de confusión.

Este fue el resultado de nuestro algoritmo al ser entrenada como podemos observar en la época 95/400 ya había obtenido una precisión del 100%:

```
Epoch 89/400
1/1 [=====] - 0s 2ms/step - loss: 0.1763 - binary_accuracy: 0.9000
Epoch 90/400
1/1 [=====] - 0s 3ms/step - loss: 0.1754 - binary_accuracy: 0.9000
Epoch 91/400
1/1 [=====] - 0s 3ms/step - loss: 0.1745 - binary_accuracy: 0.9000
Epoch 92/400
1/1 [=====] - 0s 2ms/step - loss: 0.1736 - binary_accuracy: 0.9000
Epoch 93/400
1/1 [=====] - 0s 999us/step - loss: 0.1728 - binary_accuracy: 0.9000
Epoch 94/400
1/1 [=====] - 0s 2ms/step - loss: 0.1719 - binary_accuracy: 0.9000
Epoch 95/400
1/1 [=====] - 0s 3ms/step - loss: 0.1710 - binary_accuracy: 1.0000
Epoch 96/400
1/1 [=====] - 0s 999us/step - loss: 0.1702 - binary_accuracy: 1.0000
Epoch 97/400
1/1 [=====] - 0s 2ms/step - loss: 0.1693 - binary_accuracy: 1.0000
```

Estos son los pesos y el bias obtenidos por el algoritmo:

```
[[-0.2929388 -0.9094409  0.0881843 -0.47624546 -0.11409567  0.07973713
  0.2372018  0.43191615 -0.40232888  0.26625898  0.561709   -0.25102645
 -0.16188118 -0.4624534  0.3410067  -0.24974996]
 [ 0.00486906 -0.14398292  0.32441467 -0.34402674  0.7283     -0.22958395
 -0.23503011  0.27138674  0.687547   -0.20501661 -0.42781952 -0.11092588
 -0.30735624  0.10411626 -0.42115697  0.32585955]
 [ 0.14413114  0.0870093  0.31824338  0.33112225  0.5393587  -0.06320775
  0.64860183  0.0665775  -0.18379892 -0.12840545  0.11357755 -0.01489518
 -0.06038088  0.2612808  -0.47480622 -0.06600833]
 [ 0.22074118  0.26145697 -0.53331995 -0.416844   -0.5540149  -0.5057303
  0.24684057 -0.42948785 -0.24376786  0.1889679  0.3147274  0.28573427
  0.23479313 -0.09619365 -0.1406275  0.18654746]
 [ 0.72880197 -0.05454657  0.13397428 -0.02008516 -0.5795839  -0.41600937
 -0.7522751  -0.7153861  0.39467573  0.11032611 -0.72683084  0.20223732
 -0.489971    0.31254596 -0.38850018  0.35623673]
 [ 0.43434793  0.32399723 -0.41325212  0.4530412  -0.27004173 -0.02792394
 -0.43758962  0.20498379  0.4673102  0.03791936  0.0358902  -0.48226508
 -0.13877824 -0.22039846  0.41736007  0.73674047]
 [ 0.55209184  0.5856278  0.15172046 -0.4796456  0.07946837  0.05572659
  0.23340896  0.21692862  0.5136553  0.38360855  0.339668  -0.1151716
 -0.4673129  -0.24585931 -0.4360821  0.581792  ]
 [ 0.14371619  0.05730788  0.08127212 -0.01842623  0.24956387  0.
  0.26990467  0.22464275  0.07247771 -0.08807009 -0.06119128 -0.07232014
  0.          0.10032042  0.          0.08804935]
```

Este es el resultado de la precisión con los pesos y bias obtenidos, los resultados que se esperaban, y la matriz de confusión:

```
1/1 [=====] - 0s 120ms/step - loss: 0.0419 - binary_accuracy: 1.0000

<keras.metrics.MeanMetricWrapper object at 0x000002C62F131A90>: 100.00%
[[0.]
 [1.]
 [0.]
 [1.]
 [0.]
 [1.]
 [0.]
 [1.]
 [0.]
 [1.]]
[[5 0]
 [0 5]]
```

Código

Este código nos sirve para importar las librerías necesarias para implementar el algoritmo para entrenar nuestra red neuronal.

```
from keras.models import Sequential
import numpy as np
from keras.layers.core import Dense
import tensorflow as tf
from sklearn.metrics import confusion_matrix
import pandas as pd
tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.ERROR)
```


Esta parte del código nos sirve para poder asignar los datos de entrada y salida, esto dependiendo del archivo que le demos a leer.

```
conjunto_datos = pd.read_csv("datos_entrada_impares.txt", header=None)
datos_entrada = conjunto_datos.iloc[:,0:7].values
datos_salida = conjunto_datos.iloc[:,7:8].values
```

Esta sección de código es la que se encarga de diseñar y entrenar nuestra red neuronal, aquí definimos cuantas neuronas necesitamos, cuantos datos de entrada vamos a utilizar, las funciones de activación que ocuparemos y la cantidad de épocas que queremos que haga para calcular los pesos y el bias.

```
modelo = Sequential()
modelo.add(Dense(16, input_dim = 7, activation='relu'))
modelo.add(Dense(1, activation='sigmoid'))
modelo.compile(loss='mean_squared_error', optimizer='adam', metrics=['binary_accuracy'])
modelo.fit(datos_entrada, datos_salida, epochs=400)
```

En esta sección indicamos que queremos observar cuales son los pesos y el bias que calculo, así como una muestra de que realmente está funcionando correctamente solicitando que nos de los resultados deseados, y que nos muestre la matriz de confusión.

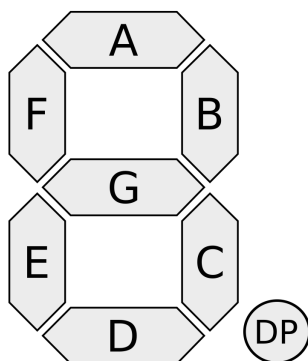
```
pesos,bias=modelo.layers[0].get_weights()
print(pesos)
print(bias)
datos=modelo.evaluate(datos_entrada,datos_salida)
print("\n%s: %.2f%%" %(modelo.metrics[1],datos[1]*100))
y_pred=modelo.predict(datos_entrada).round()
print(y_pred)
y_pred=(y_pred>=.5)
cm=confusion_matrix(datos_salida,y_pred)
print(cm)
```

Números Pares

Para este caso se hizo el diseño de la red neuronal tomando en cuenta que los números que tiene que comparar provienen de un display de 7 segmentos; por lo que la cantidad de entradas que recibió nuestro algoritmo fueron 7, y únicamente nos da una salida, que nos dice si es o no es par.

Entrada

Este es el diseño básico del display de 7 segmentos en la siguiente tabla indicamos cómo es que tomamos en cuenta nuestras entradas.



A	B	C	D	E	F	G	Número Decimal	1 = Par
1	1	1	1	1	1	0	0	1
0	1	1	0	0	0	0	1	0
1	1	0	1	1	0	1	2	1
1	1	1	1	0	0	1	3	0
0	1	1	0	0	1	1	4	1
1	0	1	1	0	1	1	5	0
1	0	1	1	1	1	1	6	1
1	1	1	0	0	0	1	7	0
1	1	1	1	1	1	1	8	1
1	1	1	0	0	1	1	9	0

Este es el contenido del archivo que lee el algoritmo para obtener los datos, los primeros 7 elementos de cada fila corresponden a las entradas y el último corresponde a si es o no par, 0 cuando no es par y 1 cuando es par.

```

1 1,1,1,1,1,1,0,1
2 0,1,1,0,0,0,0,0
3 1,1,0,1,1,0,1,1
4 1,1,1,1,0,0,1,0
5 0,1,1,0,0,1,1,1
6 1,0,1,1,0,1,1,0
7 1,0,1,1,1,1,1,1
8 1,1,1,0,0,0,0,0
9 1,1,1,1,1,1,1,1
10 1,1,1,0,0,1,1,0

```

Las siguientes líneas de código son las encargadas de delimitar cómo es que tomará en cuenta los datos del archivo. Del primer elemento al penúltimo serán las entradas y el último será la salida deseada:

```
datos_entrada = conjunto_datos.iloc[:,0:7].values
```

```
datos_salida = conjunto_datos.iloc[:,7:8].values
```

También le indicaremos cómo será el diseño de nuestra red neuronal que en este caso será una red de 16 neuronas y le indicamos que realice 400 épocas

Salida

La salida que esperamos es la siguiente, de acuerdo con nuestra matriz de confusión:

	0 (Cantidad de 0 esperados)	1 (Cantidad de 1 esperados)
0 (Obtenidos)	5	0
1 (Obtenidos)	0	5

Los bias podrían corroborarse manualmente con excel, pero el propósito de este algoritmo es ahorrarnos esa parte además que sería demasiado trabajo manual. Por lo que nos basaremos únicamente en la matriz de confusión.

Este fue el resultado de nuestro algoritmo al ser entrenada como podemos observar para este caso en la época 304/400 ya había obtenido una precisión del 100%:

```
Epoch 302/400
1/1 [=====] - 0s 2ms/step - loss: 0.0749 - binary_accuracy: 0.9000
Epoch 303/400
1/1 [=====] - 0s 2ms/step - loss: 0.0744 - binary_accuracy: 0.9000
Epoch 304/400
1/1 [=====] - 0s 3ms/step - loss: 0.0740 - binary_accuracy: 1.0000
Epoch 305/400
1/1 [=====] - 0s 2ms/step - loss: 0.0735 - binary_accuracy: 1.0000
Epoch 306/400
1/1 [=====] - 0s 3ms/step - loss: 0.0730 - binary_accuracy: 1.0000
```

Estos son los pesos y el bias obtenidos por el algoritmo:

```
[[-0.05516442 -0.17024218 -0.46158847 -0.5704839 -0.21999654 1.0035936
-0.41814414 -0.23139846 0.3938061 -0.21095946 0.5411111 -0.21589649
-0.4508167 0.84445584 0.23609893 -0.16162619]
[ 0.48962596 0.3639907 -0.03607094 0.24104849 0.43581235 -0.3706207
0.42267352 -0.4792595 -0.28111276 -0.2807175 -0.40318307 -0.29899576
0.07197559 -0.13657446 0.34875548 -0.15276864]
[-0.79437804 -0.01936706 -0.41392192 -0.16050605 0.22306243 0.69501823
-0.569316 -0.4401403 -0.16278729 -0.2454707 0.29022914 -0.27179736
-0.22754683 0.14899321 0.36412448 -0.4803021 ]
[ 0.5057244 0.07189185 0.10831547 0.7454308 0.46704623 -0.73501843
0.21474993 -0.01889822 -0.39229 -0.4793915 0.00923878 0.21023488
-0.3778163 -0.49235123 -0.21978359 -0.02405217]
[ 0.24172598 0.04493431 0.430713 0.4499093 -0.68732315 -0.262167
0.06524294 -0.30339265 -0.4598532 0.20783538 -0.09891266 0.18910104
-0.22745216 -0.2648641 -0.5918561 0.05398172]
[ 0.47478852 0.67641515 -0.33067313 0.5833295 -0.6647299 -0.6322636
0.05407487 -0.5040725 0.15590835 0.09084475 -0.27956542 -0.1449228
0.18276887 -0.28085148 -0.20800592 0.02454996]
[ 0.11211039 0.20721102 -0.3086099 0.31046388 -0.1434934 0.38599578
0.6210783 -0.21535704 -0.44123566 0.20029348 0.35256037 -0.05205441
0.4538563 0.30376446 0.12769485 0.11900312]]
[ 0.3266893 -0.07988042 0. 0.3450266 0.14917822 0.30066732
0.11530871 0. 0. 0. 0.03924047 0.
-0.02864571 0.18036714 0.29079458 0. ]
```

Este es el resultado de la precisión con los pesos y bias obtenidos, los resultados que se esperaban, y la matriz de confusión:

```
1/1 [=====] - 0s 112ms/step - loss: 0.0402 - binary_accuracy: 1.0000
```

```
<keras.metrics.MeanMetricWrapper object at 0x000002C62F422AF0>: 100.00%
```

```
[[1.]  
 [0.]  
 [1.]  
 [0.]  
 [1.]  
 [0.]  
 [1.]  
 [0.]  
 [1.]  
 [0.]]  
[[5 0]  
 [0 5]]
```

Código

Este código nos sirve para importar las librerías necesarias para implementar el algoritmo para entrenar nuestra red neuronal.

```
from keras.models import Sequential  
import numpy as np  
from keras.layers.core import Dense  
import tensorflow as tf  
from sklearn.metrics import confusion_matrix  
import pandas as pd  
tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.ERROR)
```

Esta parte del código nos sirve para poder asignar los datos de entrada y salida, esto dependiendo del archivo que le demos a leer.

```
conjunto_datos = pd.read_csv("datos_entrada_pares.txt", header=None)  
datos_entrada = conjunto_datos.iloc[:,0:7].values  
datos_salida = conjunto_datos.iloc[:,7:8].values
```

Esta sección de código es la que se encarga de diseñar y entrenar nuestra red neuronal, aquí definimos cuantas neuronas necesitamos, cuantos datos de entrada vamos a utilizar, las funciones de activación que ocuparemos y la cantidad de épocas que queremos que haga para calcular los pesos y el bias.

```
modelo = Sequential()  
modelo.add(Dense(16, input_dim = 7, activation='relu'))  
modelo.add(Dense(1, activation='sigmoid'))  
modelo.compile(loss='mean_squared_error', optimizer='adam', metrics=['binary_accuracy'])  
modelo.fit(datos_entrada, datos_salida, epochs=400)
```

En esta sección indicamos que queremos observar cuales son los pesos y el bias que calculo, así como una muestra de que realmente está funcionando correctamente solicitando que nos de los resultados deseados, y que nos muestre la matriz de confusión.

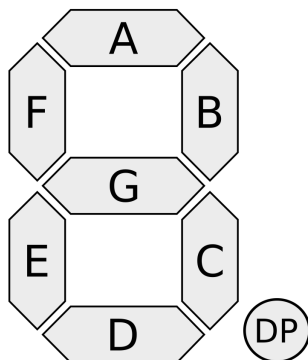
```
pesos,bias=modelo.layers[0].get_weights()
print(pesos)
print(bias)
datos=modelo.evaluate(datos_entrada,datos_salida)
print("\n%s: %.2f%%" %(modelo.metrics[1],datos[1]*100))
y_pred=modelo.predict(datos_entrada).round()
print(y_pred)
y_pred=(y_pred>=.5)
cm=confusion_matrix(datos_salida,y_pred)
print(cm)
```

Números Mayores o Iguales a 5

Para este caso se hizo el diseño de la red neuronal tomando en cuenta que los números que tiene que comparar provienen de un display de 7 segmentos; por lo que la cantidad de entradas que recibió nuestro algoritmo fueron 7, y únicamente nos da una salida, que nos dice si es mayor o igual que 5.

Entrada

Este es el diseño básico del display de 7 segmentos en la siguiente tabla indicamos cómo es que tomamos en cuenta nuestras entradas.



A	B	C	D	E	F	G	Número Decimal	1 = Par
1	1	1	1	1	1	0	0	0
0	1	1	0	0	0	0	1	0
1	1	0	1	1	0	1	2	0
1	1	1	1	0	0	1	3	0
0	1	1	0	0	1	1	4	0
1	0	1	1	0	1	1	5	1
1	0	1	1	1	1	1	6	1

1	1	1	0	0	0	1	7	1
1	1	1	1	1	1	1	8	1
1	1	1	0	0	1	1	9	1

Este es el contenido del archivo que lee el algoritmo para obtener los datos, los primeros 7 elementos de cada fila corresponden a las entradas y el último corresponde a si es mayor o igual a 5, 0 cuando no es mayor o igual que 5 y 1 cuando es mayor o igual que 5.

```

1 1,1,1,1,1,1,0,0
2 0,1,1,0,0,0,0,0
3 1,1,0,1,1,0,1,0
4 1,1,1,1,0,0,1,0
5 0,1,1,0,0,1,1,0
6 1,0,1,1,0,1,1,1
7 1,0,1,1,1,1,1,1
8 1,1,1,0,0,0,0,1
9 1,1,1,1,1,1,1,1
10 1,1,1,0,0,1,1,1

```

Las siguientes líneas de código son las encargadas de delimitar cómo es que tomará en cuenta los datos del archivo. Del primer elemento al penúltimo serán las entradas y el último será la salida deseada:

```
datos_entrada = conjunto_datos.iloc[:,0:7].values
```

```
datos_salida = conjunto_datos.iloc[:,7:8].values
```

También le indicaremos cómo será el diseño de nuestra red neuronal que en este caso será una red de 16 neuronas y le indicamos que realice 400 épocas.

Salida

La salida que esperamos es la siguiente, de acuerdo con nuestra matriz de confusión:

	0 (Cantidad de 0 esperados)	1 (Cantidad de 1 esperados)
0 (Obtenidos)	5	0
1 (Obtenidos)	0	5

Los bias podrían corroborarse manualmente con excel, pero el propósito de este algoritmo es ahorrarnos esa parte además que sería demasiado trabajo manual. Por lo que nos basaremos únicamente en la matriz de confusión.

Este fue el resultado de nuestro algoritmo al ser entrenada como podemos observar para este caso en la época 304/400 ya había obtenido una precisión del 100%:

```
Epoch 303/400
1/1 [=====] - 0s 2ms/step - loss: 0.1226 - binary_accuracy: 0.9000
Epoch 304/400
1/1 [=====] - 0s 2ms/step - loss: 0.1222 - binary_accuracy: 1.0000
Epoch 305/400
1/1 [=====] - 0s 3ms/step - loss: 0.1219 - binary_accuracy: 1.0000
Epoch 306/400
1/1 [=====] - 0s 2ms/step - loss: 0.1215 - binary_accuracy: 1.0000
```

Estos son los pesos y el bias obtenidos por el algoritmo:

```
[[-0.6828777 -0.02538499 -0.8968325 0.7978532 0.69404435 0.48356918
-0.23150274 -0.51502836 0.04931236 -0.00172408 -0.17485401 0.11674035
-0.45738488 -0.35568303 -0.4589237 0.17913347]
[ 0.56018007 0.1202463 0.543484 -0.52132154 -0.2093588 0.35558808
0.5174897 0.87419367 -0.31891474 0.08569816 -0.40294427 0.05894983
0.3664958 -0.0301597 -0.29679972 -0.12777019]
[-0.14343035 0.22988245 0.0866321 0.5760075 0.42818183 -0.29561788
0.17509799 0.27633384 0.28179273 0.1562325 0.23223154 -0.32390702
0.19020925 0.20618396 -0.3666107 -0.46966442]
[-0.1887091 0.34717607 0.26105303 -0.09805737 0.15050738 0.08551712
0.7035316 -0.04872632 -0.07031687 0.24763522 -0.05354159 0.03528488
0.9030504 0.14621224 0.21099198 0.18291664]
[ 0.7480937 0.35627687 0.2482443 -0.25354356 -0.23621008 -0.29079825
-0.18560681 0.14456515 -0.452497 -0.24406001 0.24143945 -0.4091158
-0.12041999 -0.30374008 0.24249119 0.41255468]
[ 0.25646558 -0.64248186 -0.24278896 0.05794941 0.3605186 0.52345717
0.18968217 -0.6503038 0.17285661 -0.28389794 0.56352174 -0.19306135
-0.42110503 0.13512558 -0.1681183 -0.03121926]
[-0.25684297 0.6530037 0.434147 0.12975256 0.3769201 -0.5233149
-0.3456582 0.7205814 -0.00100779 -0.33163676 0.57910866 -0.31934005
0.05126555 -0.14599428 -0.44007736 0.38098198]]
[ 0.26693848 0.17413075 0.35751748 -0.05427707 -0.13067627 -0.06007092
0.1784961 0.26542118 -0.13156389 -0.11400893 -0.17964964 0.
0.27031794 -0.17108047 0. -0.14502634]
```

Este es el resultado de la precisión con los pesos y bias obtenidos, los resultados que se esperaban, y la matriz de confusión:

```
1/1 [=====] - 0s 120ms/step - loss: 0.0934 - binary_accuracy: 1.0000

<keras.metrics.MeanMetricWrapper object at 0x000002C6307EF610>: 100.00%
[[0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]]
[[5 0]
 [0 5]]
```


Código

Este código nos sirve para importar las librerías necesarias para implementar el algoritmo para entrenar nuestra red neuronal.

```
from keras.models import Sequential
import numpy as np
from keras.layers.core import Dense
import tensorflow as tf
from sklearn.metrics import confusion_matrix
import pandas as pd
tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.ERROR)
```

Esta parte del código nos sirve para poder asignar los datos de entrada y salida, esto dependiendo del archivo que le demos a leer.

```
conjunto_datos = pd.read_csv("datos_entrada_mayores.txt", header=None)
datos_entrada = conjunto_datos.iloc[:,0:7].values
datos_salida = conjunto_datos.iloc[:,7:8].values
```

Esta sección de código es la que se encarga de diseñar y entrenar nuestra red neuronal, aquí definimos cuantas neuronas necesitamos, cuantos datos de entrada vamos a utilizar, las funciones de activación que ocuparemos y la cantidad de épocas que queremos que haga para calcular los pesos y el bias.

```
modelo = Sequential()
modelo.add(Dense(16, input_dim = 7, activation='relu'))
modelo.add(Dense(1, activation='sigmoid'))
modelo.compile(loss='mean_squared_error', optimizer='adam', metrics=['binary_accuracy'])
modelo.fit(datos_entrada, datos_salida, epochs=400)
```

En esta sección indicamos que queremos observar cuales son los pesos y el bias que calculo, así como una muestra de que realmente está funcionando correctamente solicitando que nos de los resultados deseados, y que nos muestre la matriz de confusión.

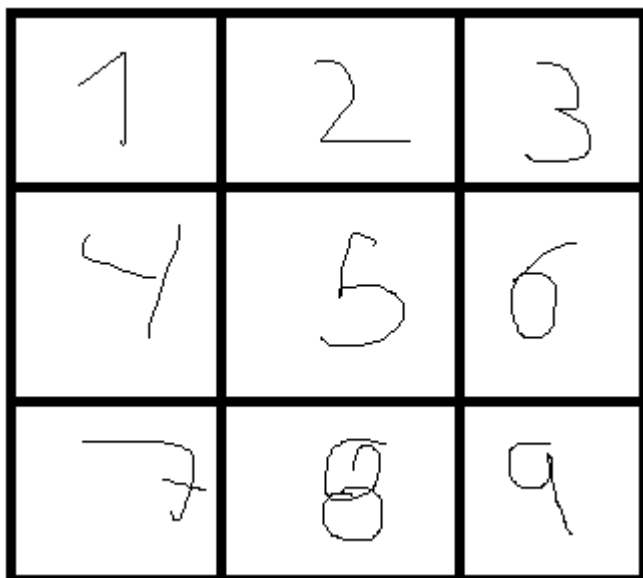
```
pesos,bias=modelo.layers[0].get_weights()
print(pesos)
print(bias)
datos=modelo.evaluate(datos_entrada,datos_salida)
print("\n%s: %.2f%%" %(modelo.metrics[1],datos[1]*100))
y_pred=modelo.predict(datos_entrada).round()
print(y_pred)
y_pred=(y_pred>=.5)
cm=confusion_matrix(datos_salida,y_pred)
print(cm)
```

Imágenes Aceptadas

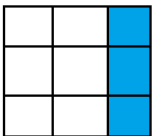


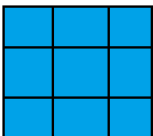
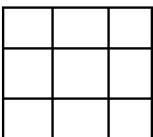
Al igual que en los casos anteriores se utilizó más de una entrada para poder entrenar a nuestro perceptrón sólo que, para la interpretación de los datos, en este caso, se toma en cuenta como si fueran paneles led que se encienden y apagan, estos al estar encendidos en un orden u otro nos muestran una imagen como si de un una pantalla se tratase. En este caso buscamos ingresar estas “imágenes”, y que el perceptrón reconozca cuales son las únicas que si tomamos como válidas.

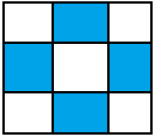
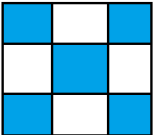

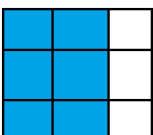
Entrada

Para esta entrada consideramos nuestros paneles de la siguiente manera.



Y la tabla que usaremos para guiarnos, al momento de ingresar los datos es la siguiente.

1	2	3	4	5	6	7	8	9	Figura Mostrada	Salida esperada
0	0	1	0	0	1	0	0	1		1
1	1	1	0	0	1	1	1	1		1
1	1	1	0	1	0	0	1	0		1
1	1	1	1	1	1	1	1	1		0
0	0	0	0	0	0	0	0	0		0

0	1	0	1	0	1	0	1	0		0
1	0	1	0	1	0	1	0	1		0
1	1	0	0	1	1	0	0	1		0
1	1	0	1	1	0	1	1	0		0

Este es el contenido del archivo que lee el algoritmo para obtener los datos, los primeros 9 elementos de cada fila corresponden a las entradas y el último corresponde a si es una imagen aceptada o no, 0 cuando no es una imagen válida y 1 cuando es una imagen aceptada.

```

1 0,0,1,0,0,1,0,0,1,1
2 1,1,1,0,0,1,1,1,1,1
3 1,1,1,0,1,0,0,1,0,1
4 1,1,1,1,1,1,1,1,1,0
5 0,0,0,0,0,0,0,0,0,0
6 0,1,0,1,0,1,0,1,0,0
7 1,0,1,0,1,0,1,0,1,0
8 1,1,0,0,1,1,0,0,1,0
9 1,1,0,1,1,0,1,1,0,0
10
```

Las siguientes líneas de código son las encargadas de delimitar cómo es que tomará en cuenta los datos del archivo. Del primer elemento al penúltimo serán las entradas y el último será la salida deseada:

```
datos_entrada = conjunto_datos.iloc[:,0:9].values
```

```
datos_salida = conjunto_datos.iloc[:,9:10].values
```

También le indicaremos cómo será el diseño de nuestra red neuronal que en este caso será una red de 16 neuronas y le indicamos que realice 400 épocas.

NOTA: Hay que tener en cuenta que esto se puede hacer con imágenes reales mediante una cámara o un sensor, pero por cuestiones prácticas se hizo con números o señales de encendido o apagado, al igual que los ejemplos de impares, pares y mayor o igual.

NOTA 2: La tabla de los datos que ingresamos a nuestra red es muy pequeña, ya que tomando en consideración que podemos obtener hasta 512 imágenes sería inhumano meter

tantos datos a mano al algoritmo o al archivo, creemos que como demostración para darnos a entender cómo podría funcionar el algoritmo para este caso es más que suficiente, pero es sin problemas se podría obtener todas estas combinaciones e ingresarlas al algoritmo.

Salida

La salida que esperamos es la siguiente, de acuerdo con nuestra matriz de confusión:

	0 (Cantidad de 0 esperados)	1 (Cantidad de 1 esperados)
0 (Obtenidos)	6	0
1 (Obtenidos)	0	3

Los bias podrían corroborarse manualmente con excel, pero el propósito de este algoritmo es ahorrarnos esa parte además que sería demasiado trabajo manual. Por lo que nos basaremos únicamente en la matriz de confusión.

Este fue el resultado de nuestro algoritmo al ser entrenada como podemos observar para este caso en la época 83/400 ya había obtenido una precisión del 100%:

```
Epoch 81/400
1/1 [=====] - 0s 2ms/step - loss: 0.1516 - binary_accuracy: 0.8889
Epoch 82/400
1/1 [=====] - 0s 2ms/step - loss: 0.1507 - binary_accuracy: 0.8889
Epoch 83/400
1/1 [=====] - 0s 2ms/step - loss: 0.1498 - binary_accuracy: 1.0000
Epoch 84/400
1/1 [=====] - 0s 3ms/step - loss: 0.1490 - binary_accuracy: 1.0000
```

Estos son los pesos y el bias obtenidos por el algoritmo:

```
[ [ 0.3651257  0.40539688  0.45349365  0.00254668  0.27529797 -0.30163783
  0.48122942  0.14164734  0.5033166  0.28359282 -0.29346377 -0.5003204
 -0.44846332  0.32886684  0.10071284 -0.16199565]
 [ -0.27575123  0.42942047  0.03598161  0.37006447 -0.14933486 -0.32844
  0.10957901  0.32006207  0.14693171 -0.5667582  0.31427222  0.5587507
 -0.23704225  0.7137861 -0.2105518 -0.46533066]
 [ -0.15273929  0.07632852  0.09375568 -0.26477015 -0.6554619  0.4786308
  0.35926893 -0.70201814  0.6672341  0.07039759  0.48231536 -0.00528763
  0.17935166  0.5322994  0.6637232 -0.47581878]
 [ -0.39376414  0.71704537  0.38099885 -0.46870637  0.09350969 -0.0561244
 -0.84650904  0.42688665 -0.64689106  0.65695673 -0.406478  0.36055693
  0.318971 -0.17030513 -0.6262195 -0.29346392]
 [ 0.20814355  0.0913975  0.5662685 -0.5076316  0.2851011 -0.2725197
 -0.3173242 -0.07181085 -0.02494715  0.5839732  0.21189523  0.0567598
 -0.10745215 -0.06636663  0.19752862 -0.35360348]
 [ -0.6391933 -0.285714 -0.1324859  0.44055793  0.42336002  0.5816028
  0.36025086  0.4805096 -0.01700749 -0.35250762  0.17154822  0.19181368
 -0.13543212 -0.7231077  0.30042687 -0.35775852]
 [ -0.12213834  0.25476205  0.1103707  0.16294083  0.44889385  0.14346652
 -0.44296858 -0.43999028 -0.02284476 -0.05371934  0.20037192  0.09198868
  0.44950214 -0.52492464  0.21474186 -0.36878443]
 [ -0.3951696  0.06967551 -0.6026529  0.4082329  0.04074078 -0.13036178
  0.3772436 -0.07306917 -0.16004266 -0.349648 -0.07879117 -0.15339027
 -0.4067337  0.5718232  0.5364853 -0.07279924]
 [ 0.5427442  0.40312964 -0.17208768 -0.27276146  0.02838582  0.34004775
  0.24808255 -0.0348105  0.27386573  0.2843303  0.05589746 -0.5299682
 -0.10965419 -0.25930658 -0.2777091 -0.05238551]]
[ 2.5039804e-01  2.4700086e-01  2.1360183e-01 -3.2952879e-02
 2.0371658e-01 -6.6836551e-02 -6.4079004e-04  3.8504058e-01
-1.2895187e-04  4.2653555e-01 -1.2982859e-04  3.4607691e-01
 0.0000000e+00 -1.0597087e-02  8.0136160e-05  0.0000000e+00]
```

Este es el resultado de la precisión con los pesos y bias obtenidos, los resultados que se esperaban, y la matriz de confusión:

```
1/1 [=====] - 0s 285ms/step - loss: 0.0126 - binary_accuracy: 1.0000
```

```
<keras.metrics.MeanMetricWrapper object at 0x000002AD838AF7F0>: 100.00%
```

```
[[1.]  
 [1.]  
 [1.]  
 [0.]  
 [0.]  
 [0.]  
 [0.]  
 [0.]  
 [0.]]  
[[6 0]  
 [0 3]]
```

Código

Este código nos sirve para importar las librerías necesarias para implementar el algoritmo para entrenar nuestra red neuronal.

```
from keras.models import Sequential  
import numpy as np  
from keras.layers.core import Dense  
import tensorflow as tf  
from sklearn.metrics import confusion_matrix  
import pandas as pd  
tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.ERROR)
```

Esta parte del código nos sirve para poder asignar los datos de entrada y salida, esto dependiendo del archivo que le demos a leer.

```
conjunto_datos = pd.read_csv("datos_entrada_imagenes.txt", header=None)  
datos_entrada = conjunto_datos.iloc[:,0:9].values  
datos_salida = conjunto_datos.iloc[:,9:10].values
```

Esta sección de código es la que se encarga de diseñar y entrenar nuestra red neuronal, aquí definimos cuantas neuronas necesitamos, cuantos datos de entrada vamos a utilizar, las funciones de activación que ocuparemos y la cantidad de épocas que queremos que haga para calcular los pesos y el bias.

```
modelo = Sequential()  
modelo.add(Dense(16, input_dim = 9, activation='relu'))  
modelo.add(Dense(1, activation='sigmoid'))  
modelo.compile(loss='mean_squared_error', optimizer='adam', metrics=['binary_accuracy'])  
modelo.fit(datos_entrada, datos_salida, epochs=400)
```

En esta sección indicamos que queremos observar cuales son los pesos y el bias que calculo, así como una muestra de que realmente está funcionando correctamente solicitando que nos de los resultados deseados, y que nos muestre la matriz de confusión.

```
pesos,bias=modelo.layers[0].get_weights()
print(pesos)
print(bias)
datos=modelo.evaluate(datos_entrada,datos_salida)
print("\n%s: %.2f%%" %(modelo.metrics[1],datos[1]*100))
y_pred=modelo.predict(datos_entrada).round()
print(y_pred)
y_pred=(y_pred>=.5)
cm=confusion_matrix(datos_salida,y_pred)
print(cm)
```

Compuerta OR

Para el caso de este ejemplo se le ingresaron los datos necesarios de una compuerta lógica OR, al ser una cantidad menor de datos nuestra red neuronal fue diseñada de manera diferente ya que no necesitaba demasiadas neuronas ni épocas para poder llegar al 100% de precisión mientras es entrenada.

Entrada

Esta es la tabla de verdad de una compuerta OR y los datos que ingresamos están en ese mismo orden.

A	B	Salida
0	0	0
0	1	1
1	0	1
1	1	1

Este es el contenido del archivo que lee el algoritmo para obtener los datos, los primeros 2 elementos de cada fila corresponden a las entradas y el último corresponde al valor de la salida de la compuerta OR.

```
1 0,0,0
2 0,1,1
3 1,0,1
4 1,1,1
```

Las siguientes líneas de código son las encargadas de delimitar cómo es que tomará en cuenta los datos del archivo. Del primer elemento al penúltimo serán las entradas y el último será la salida deseada:

```
datos_entrada = conjunto_datos.iloc[:,0:2].values
```

```
datos_salida = conjunto_datos.iloc[:,2:3].values
```

También le indicaremos cómo será el diseño de nuestra red neuronal que en este caso será una red de 1 neurona y le indicamos que realice 130 épocas.

Salida

La salida que esperamos es la siguiente, de acuerdo con nuestra matriz de confusión:

	0 (Cantidad de 0 esperados)	1 (Cantidad de 1 esperados)
0 (Obtenidos)	1	0
1 (Obtenidos)	0	3

Los bias podrían corroborarse manualmente con excel, pero el propósito de este algoritmo es ahorrarnos esa parte además que sería demasiado trabajo manual. Por lo que nos basaremos únicamente en la matriz de confusión.

Este fue el resultado de nuestro algoritmo al ser entrenada como podemos observar para este caso en la época 1/130 ya había obtenido una precisión del 100%:

```
Epoch 1/130
1/1 [=====] - 0s 311ms/step - loss: 0.0940 - binary_accuracy: 1.0000
Epoch 2/130
1/1 [=====] - 0s 2ms/step - loss: 0.0934 - binary_accuracy: 1.0000
Epoch 3/130
1/1 [=====] - 0s 3ms/step - loss: 0.0929 - binary_accuracy: 1.0000
Epoch 4/130
1/1 [=====] - 0s 3ms/step - loss: 0.0924 - binary_accuracy: 1.0000
- - - - -
```

Estos son los pesos y el bias obtenidos por el algoritmo:

```
[[0.589789 ]
 [0.6315052]
 [0.09504319]
```

Este es el resultado de la precisión con los pesos y bias obtenidos, los resultados que se esperaban, y la matriz de confusión:

```
1/1 [=====] - 0s 101ms/step - loss: 0.0708 - binary_accuracy: 1.0000

<keras.metrics.MeanMetricWrapper object at 0x000002AD88E9E340>: 100.00%
[[0.]
 [1.]
 [1.]
 [1.]]
[[1 0]
 [0 3]]
```

Código

Este código nos sirve para importar las librerías necesarias para implementar el algoritmo para entrenar nuestra red neuronal.

```
from keras.models import Sequential
import numpy as np
from keras.layers.core import Dense
import tensorflow as tf
from sklearn.metrics import confusion_matrix
import pandas as pd
tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.ERROR)
```

Esta parte del código nos sirve para poder asignar los datos de entrada y salida, esto dependiendo del archivo que le demos a leer.

```
conjunto_datos = pd.read_csv("datos_entrada_or.txt", header=None)
datos_entrada = conjunto_datos.iloc[:,0:2].values
datos_salida = conjunto_datos.iloc[:,2:3].values
```

Esta sección de código es la que se encarga de diseñar y entrenar nuestra red neuronal, aquí definimos cuantas neuronas necesitamos, cuantos datos de entrada vamos a utilizar, las funciones de activación que ocuparemos y la cantidad de épocas que queremos que haga para calcular los pesos y el bias.

```
modelo = Sequential()
modelo.add(Dense(1, input_dim = 2, activation='relu'))
#modelo.add(Dense(1, activation='sigmoid'))
modelo.compile(loss='mean_squared_error', optimizer='adam', metrics=['binary_accuracy'])
modelo.fit(datos_entrada, datos_salida, epochs=130)
```

En esta sección indicamos que queremos observar cuales son los pesos y el bias que calculo, así como una muestra de que realmente está funcionando correctamente solicitando que nos de los resultados deseados, y que nos muestre la matriz de confusión.

```
pesos,bias=modelo.layers[0].get_weights()
print(pesos)
print(bias)
datos=modelo.evaluate(datos_entrada,datos_salida)
print("\ns: %.2f%%" %(modelo.metrics[1],datos[1]*100))
y_pred=modelo.predict(datos_entrada).round()
print(y_pred)
y_pred=(y_pred>=.5)
cm=confusion_matrix(datos_salida,y_pred)
print(cm)
```

Compuerta XOR

Para el caso de este ejemplo se le ingresaron los datos necesarios de una compuerta lógica XOR, a diferencia de la compuerta OR, nuestra red neuronal se parece más a la de los primeros ejemplos ya que al tener que tener 2 fronteras de decisión fue necesario usar más neuronas, capas y épocas para poder llegar al 100% de precisión.

Entrada

Esta es la tabla de verdad de una compuerta OR y los datos que ingresamos están en ese mismo orden.

A	B	Salida
0	0	0
0	1	1
1	0	1
1	1	0

Este es el contenido del archivo que lee el algoritmo para obtener los datos, los primeros 2 elementos de cada fila corresponden a las entradas y el último corresponde al valor de la salida de la compuerta OR.

1	0,0,0
2	0,1,1
3	1,0,1
4	1,1,0

Las siguientes líneas de código son las encargadas de delimitar cómo es que tomará en cuenta los datos del archivo. Del primer elemento al penúltimo serán las entradas y el último será la salida deseada:

```
datos_entrada = conjunto_datos.iloc[:,0:2].values
```

```
datos_salida = conjunto_datos.iloc[:,2:3].values
```

También le indicaremos cómo será el diseño de nuestra red neuronal que en este caso será una red de 16 neuronas, le agregaremos una capa de neuronas más con la función sigmoid y le indicamos que realice 150 épocas.

Salida

La salida que esperamos es la siguiente, de acuerdo con nuestra matriz de confusión:

	0 (Cantidad de 0 esperados)	1 (Cantidad de 1 esperados)
0 (Obtenidos)	2	0
1 (Obtenidos)	0	2

Los bias podrían corroborarse manualmente con excel, pero el propósito de este algoritmo es ahorrarnos esa parte además que sería demasiado trabajo manual. Por lo que nos basaremos únicamente en la matriz de confusión.

Este fue el resultado de nuestro algoritmo al ser entrenada como podemos observar para este caso en la época 71/150 ya había obtenido una precisión del 100%:

```
Epoch 70/150
1/1 [=====] - 0s 2ms/step - loss: 0.2442 - binary_accuracy: 0.5000
Epoch 71/150
1/1 [=====] - 0s 2ms/step - loss: 0.2440 - binary_accuracy: 1.0000
Epoch 72/150
1/1 [=====] - 0s 2ms/step - loss: 0.2437 - binary_accuracy: 1.0000
Epoch 73/150
1/1 [=====] - 0s 1000us/step - loss: 0.2435 - binary_accuracy: 1.0000
```


Estos son los pesos y el bias obtenidos por el algoritmo:

```
[[ 0.0619751 -0.18557693 -0.2848908  0.42988628  0.18999828 -0.1892685
  0.17282407 -0.29961962 -0.5650112  0.44575223 -0.1971452  0.14088896
 -0.39419454 -0.45705578 -0.10021217 -0.39274716]
 [ 0.1051755  0.18478532  0.4256713  0.39930674 -0.22989056  0.189027
 -0.5240083 -0.04687899  0.37076962 -0.02070928 -0.13937938  0.1884796
 -0.41553265 -0.49758342  0.34757683 -0.33768934]]
 [ 8.2428521e-03  2.5087275e-04  7.6396406e-02  3.0419787e-03
  2.4449510e-02  8.7866865e-05  2.3621190e-02  0.0000000e+00
 -1.2758164e-01  4.6036620e-02  0.0000000e+00 -1.4062200e-01
  0.0000000e+00  0.0000000e+00  1.1565997e-01  0.0000000e+00]
```

Este es el resultado de la precisión con los pesos y bias obtenidos, los resultados que se esperaban, y la matriz de confusión:

```
1/1 [=====] - 0s 117ms/step - loss: 0.2244 - binary_accuracy: 1.0000

<keras.metrics.MeanMetricWrapper object at 0x000002AD8A4C3FA0>: 100.00%
[[0.]
 [1.]
 [1.]
 [0.]]
[[2 0]
 [0 2]]
```

Código

Este código nos sirve para importar las librerías necesarias para implementar el algoritmo para entrenar nuestra red neuronal.

```
from keras.models import Sequential
import numpy as np
from keras.layers.core import Dense
import tensorflow as tf
from sklearn.metrics import confusion_matrix
import pandas as pd
tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.ERROR)
```

Esta parte del código nos sirve para poder asignar los datos de entrada y salida, esto dependiendo del archivo que le demos a leer.

```
conjunto_datos = pd.read_csv("datos_entrada_xor.txt", header=None)
datos_entrada = conjunto_datos.iloc[:, 0:2].values
datos_salida = conjunto_datos.iloc[:, 2:3].values
```

Esta sección de código es la que se encarga de diseñar y entrenar nuestra red neuronal, aquí definimos cuantas neuronas necesitamos, cuantos datos de entrada vamos a utilizar, las funciones de activación que ocuparemos y la cantidad de épocas que queremos que haga para calcular los pesos y el bias.

```
modelo = Sequential()
modelo.add(Dense(16, input_dim = 2, activation='relu'))
modelo.add(Dense(1, activation='sigmoid'))
modelo.compile(loss='mean_squared_error', optimizer='adam', metrics=['binary_accuracy'])
modelo.fit(datos_entrada, datos_salida, epochs=150)
```

En esta sección indicamos que queremos observar cuales son los pesos y el bias que calculo, así como una muestra de que realmente está funcionando correctamente solicitando que nos de los resultados deseados, y que nos muestre la matriz de confusión.

```
pesos,bias=modelo.layers[0].get_weights()
print(pesos)
print(bias)
datos=modelo.evaluate(datos_entrada,datos_salida)
print("\n%s: %.2f%%" %(modelo.metrics[1],datos[1]*100))
y_pred=modelo.predict(datos_entrada).round()
print(y_pred)
y_pred=(y_pred>=.5)
cm=confusion_matrix(datos_salida,y_pred)
print(cm)
```

Pseudocódigo

Esta parte del pseudocódigo aplica para todos los casos anteriores, ya que únicamente cambian algunas cosas dependiendo lo que se requiera hacer, estos cambios se especifican a detalle en las partes correspondientes.

Bloque para leer los datos de un archivo de texto/csv.

```
conjuntos_datos = pd.read_csv("nombre del archivo donde se encuentran tus datos")
datos_entrada = conjunto_datos.iloc[rango de valores para los datos de entrada]
datos_salida = conjunto_datos.iloc[rango de valores para los datos de salida]
```

Bloque para definir y entrenar nuestra red.

```
modelo = Sequential()
modelo.add(se agregan las neuronas que ocuparemos, la cantidad de datos de entrada que recibirá y la funcion de activacion que se utilizara)
modelo.add(se agrega una segunda capa de neuronas solo si es necesario)
modelo.compile(se agregan parametros como las metrcias, el optimizador y el error que puede haber)
modelo.fit(indicamos cuales son los datos de entrada y salida y cuantas epocas queremos que entrene)
```

Conclusiones

Creo que hemos logrado explicar de manera satisfactoria como es que funciona este algoritmo, que únicamente adaptamos a las necesidades que teníamos, como lo mencionamos en alguno de los ejemplos anteriores, este algoritmo se puede combinar como periféricos que reconozcan caras por ejemplo, o imágenes, figuras e incluso número manuscritos como lo ejemplificamos más adelante. Pensamos que aunque pudimos resolver los problemas solicitados aún tenemos mucho que aprender sobre este algoritmo y las redes neuronales en general, ya que hay mucha documentación y desarrollo detrás del algoritmo que nos permite hacer cosas aún más complejas.

Reconocimiento de números hechos a mano con k-nn

Objetivo

El objetivo de este algoritmo como su nombre lo dice es que sea capaz de detectar numero hechos a mano por medio de dibujos ya preestablecidos (problema conocido como **OCR** o reconocimiento óptico de caracteres), se utilizará un dataset que es **MNIST** el cual contiene imágenes de entrenamiento, etiquetas, etc.

Restricciones

Este dataset que se mencionó contiene miles de imágenes con dígitos escritos a mano, sin embargo el algoritmo sólo detectará los números individuales por lo cual solo podrá ir del 0 al 9.

Modelo

Entradas:

- se tendrá un dataset con miles de imágenes de números dibujados a mano.
- se le dara un numero de vecinos, los cuales son la cantidad de datos se le da para el análisis del dígito,
- se tomará en cuenta un dato totalmente random de entre las miles de imágenes del dataset

Proceso:

1. se cargan las imágenes del dataset y se introducen a matrices de **NumPy**
2. se agarra un dígito al azar de grupo de test del dataset
3. Si se pone toda la matriz en una sola fila quedaría una matriz de 1x784 y se guarda su etiqueta para compararla con la clasificación que haga el algoritmo.
4. Se busca entre los vecinos cercanos y se clasifica un dato que fue el elegido al azar y se compara con la etiqueta de los vecinos y se decide cual el el número seleccionado con la imagen hecha a mano.

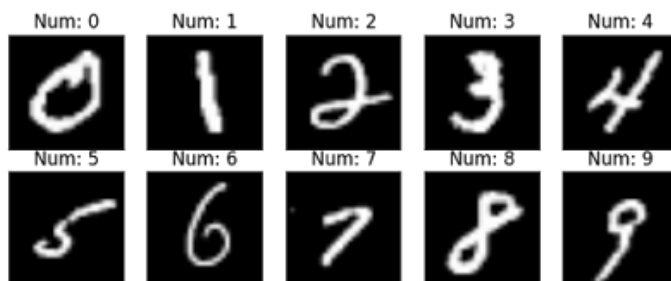
Descripción del algoritmo.

Para este algoritmos se utilizó **knn(k-nearest neighbors)** que es un algoritmo muy usado en machine learning, el cual es un algoritmo conceptualmente sencillo y fácil de implementar, el cual pertenece a la familia de algoritmos clasificados supervisada, es decir, que necesitan la intervención humana para realizar su aprendizaje.

el dataset utilizado es **MNIST** el cual consiste de los siguientes archivos:

- train-images-idx3-ubyte.gz: imágenes de entrenamiento
- train-labels-idx1-ubyte.gz: etiquetas de las imágenes de entrenamiento
- t10k-images-idx3-ubyte.gz: conjunto de imágenes para test
- t10k-labels-idx1-ubyte.gz: conjunto de etiquetas de las imágenes de test

Este *dataset* contiene miles de imágenes con dígitos escritos a mano.



Código

Este código nos sirve para importar las librerías necesarias para implementar el algoritmo

```
In [1]: import numpy as np
import random
from IPython.display import display
import matplotlib.pyplot as plt
```

aquí estamos cargando las imágenes del dataset y se introducen en matrices de NumPy. Para esto se necesitan los archivos de dataset por lo que para que funciones se debe verificar que la dirección de la carpeta sea la correcta donde se tienen los archivos.

```
In [5]: def loadMNIST( prefix, folder ):
    intType = np.dtype( 'int32' ).newbyteorder( '>' )
    nMetaDataBytes = 4 * intType.itemsize

    data = np.fromfile( folder + "/" + prefix + '-images.idx3-ubyte', dtype = 'ubyte' )
    magicBytes, nImages, width, height = np.frombuffer( data[:nMetaDataBytes].tobytes(), intType )
    data = data[nMetaDataBytes:].astype( dtype = 'float32' ).reshape( [ nImages, width, height ] )

    labels = np.fromfile( folder + "/" + prefix + '-labels.idx1-ubyte',
        dtype = 'ubyte' )[2 * intType.itemsize:]

    return data, labels

images_tr, labels_tr = loadMNIST( "train", "C:\\Users\\crist\\alg" )
images_te, labels_te = loadMNIST( "t10k", "C:\\Users\\crist\\alg" )
# imágenes en array de 60000 x 28 x 28 -> 60000 imágenes de 28x28
```

aquí seleccionamos un dígito aleatorio del grupo de test

```
In [28]: # coger una aleatoria del grupo de test
i = random.randint(0, images_te.shape[0])
img_test = images_te[i].flatten()
label_test = labels_te[i]
```

Ahora buscamos a los k vecinos más cercanos (en este caso k=5).

```
In [29]: # buscamos los vecinos más cercanos (KNN)
k = 5 # número de vecinos

distances = []
for i in range(images_tr.shape[0]):
    dist = np.sqrt(np.sum(np.square(images_tr[i].flatten() - img_test)))
    distances.append((dist, labels_tr[i])) # guardamos las etiquetas y la distancia

#ordenamos por distancia y nos quedamos con los k vecinos más cercanos
distances.sort(key=lambda x: x[0])
neighbors = distances[:k]
```

Aquí se empiezan a contar las etiquetas de los 5 vecinos cercanos y el ganador es nuestra predicción.

```
In [30]: # contamos los votos para ver qué etiqueta gana
votes = [0,0,0,0,0,0,0,0,0,0]
for neighbor in neighbors:
    votes[neighbor[1]] = votes[neighbor[1]] + 1
# obtenemos la etiqueta ganadora
pred_label = votes.index(max(votes))
```

Por último la siguiente imagen muestra una captura de la ejecución del código en la cual nos muestra el funcionamiento del algoritmo y nos acierta la predicción.

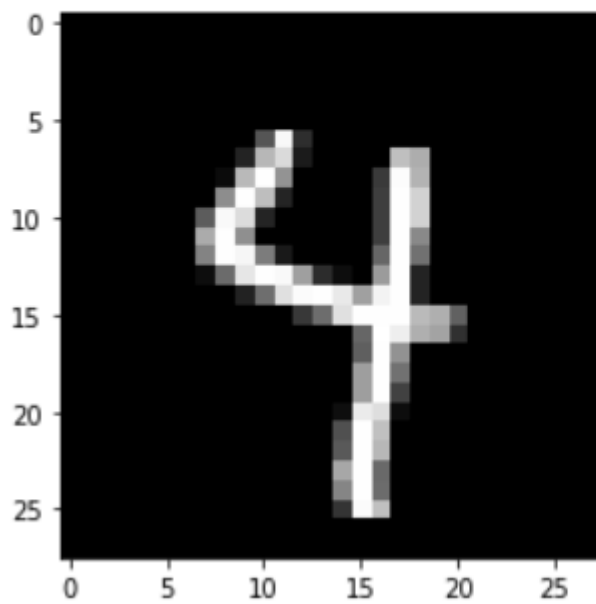
```
In [31]: print ("Predicted: " + str(pred_label))
print ("Real: " + str(label_test))
img = plt.imshow(img_test.reshape(28,28), cmap="gray")
display(img)
```

Resultado

Predicted: 4

Real: 4

<matplotlib.image.AxesImage at 0x20eb1d95220>



Referencias

- Alberto Garcia. (febrero de 2021). Reconocimiento de caracteres manuscritos con k-nn. 8 de diciembre de 2021, de El laberinto de Falken Sitio web: <https://www.ellaberintodefalken.com/2019/02/knn-mnist.html>
- Alberto Garcia. (febrero 2019). Clasificación con k-nearest neighbors. 8 de diciembre de 2021, de El laberinto de Falken Sitio web: <https://www.ellaberintodefalken.com/2019/02/clasificacion-con-k-nearest-neighbors.html>
- [LeCun et al., 1998a] Y. LeCun, L. Bottou, Y. Bengio y P. Haffner. "Aprendizaje basado en gradientes aplicado al reconocimiento de documentos". Proceedings of the IEEE , 86 (11): 2278-2324, noviembre de 1998
- Yann LeCun. (1998). LA BASE DE DATOS MNIST. 8 de diciembre de 2021, de yann.lecun Sitio web: <http://yann.lecun.com/exdb/mnist/>