Universidad Autónoma Chapingo

Departamento de Mecánica Agrícola

Ingeniería Mecatrónica Agrícola

Informe 1:
Redes Neuronales Convolucionales

Asignatura:

Inteligencia artificial

Nombre del profesor:

Luis Arturo Soriano Avendaño

Alumno:

Cocotle Lara Jym Emmanuel [1710451-3]

GRADO:	GRUPO:
7°	7

Chapingo, Texcoco Edo. México

Fecha de entrega: 28/11/2021

Índice

Introducción 2
Desarrollo 3
Redes Neuronales Convolucionales 3
Dataset 4
Convolución 4
Pooling 5
Softmax 6
Programa 6
Resultados 13
Conclusión 18
Bibliografía 19

Introducción

La inteligencia artificial es una de las herramientas de desarrollo que son ocupadas en gran medida en el mundo moderno ya que el progreso tecnológico ha sido grande en los últimos años. Uno de los elementos de la inteligencia artificial son las redes neuronales artificiales, las cuales son modelos computacionales formados por neuronas.

Las Redes Neuronales Convolucionales (CNN por sus siglas en inglés) comenzaron a ser desarrolladas en 1982 por Kunihiko Fukushima quien en 1982 desarrolló el neocognitron, es decir, una red neuronal de tipo backpropagation que imita el proceso del cortex visual (Fukushima, K., & Miyake, S. 1982)

Las CNN son un tipo de redes neuronales artificiales donde las "neuronas" corresponden a campos receptivos de una manera muy similar a las neuronas en la corteza visual primaria de un cerebro biológico. Este tipo de red es una variación de un perceptrón multicapa, sin embargo, debido a que su aplicación es realizada en matrices bidimensionales, son muy efectivas para tareas de visión artificial, como en la clasificación y segmentación de imágenes, entre otras aplicaciones.

Para ello, la CNN contiene varias capas ocultas especializadas y con una jerarquía: esto quiere decir que las primeras capas pueden detectar líneas, curvas y se van especializando hasta llegar a capas más profundas que reconocen formas complejas como un rostro o la silueta de un animal.

Para observar la diferencia entre las CNN y el resto de redes, es necesario poner atención en sus entradas y sus salidas. Las cuales suelen ser estructuradas, lo cual aprovechan las redes convolucionales para tener arquitecturas más eficientes. Estas redes constan de diversas

capas convolucionales y de reducción (pooling) que aparecen de manera alternadas (Aguado López, I.2020).

Las redes neuronales han conseguido grandes logros, como en el caso de Google, el cual ha conseguido derrotar a su propio reCAPTCHA con redes neuronales o en Stanford, donde se ha conseguido crear una red neuronal que genere pies de fotos automáticamente (Sánchez, E. G, 2019).

En este informe se abordarán los conocimientos básicos para la comprensión de las redes neuronales convolucionales, para posteriormente crear una CNN capaz de identificar rostros, esto con ayuda del software Python.

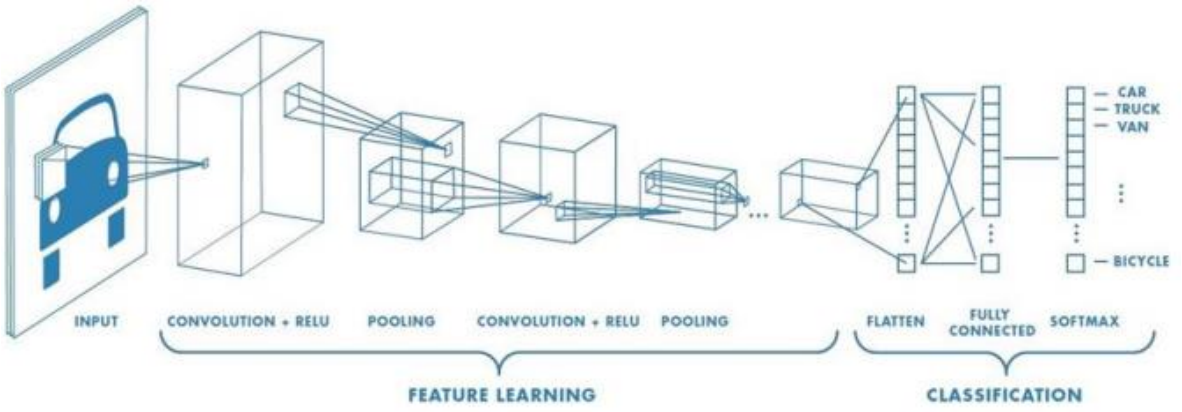
Desarrollo

Se empezará comentando las características de una CNN, para poder detallar la forma en que trabaja una CNN y por ultimo se realizará un programa en Python donde se muestre dicho funcionamiento, con el fin de observar los resultados esperados.

Redes Neuronales Convolucionales

La entrada de una red convolucional suele ser una imagen m x m x r, donde m es la altura y ancho de la imagen y r son los canales. La salida también puede ser una imagen o pueden ser convencionales cuando se trate de un problema de clasificación o regresión.
Como primer paso la red toma como entrada los píxeles de una imagen. En el caso de este informe optaremos a que las imágenes sean en escala de grises y las entradas serán de 80x80 píxeles de alto y ancho, por lo que habrá en total 6400 neuronas. En caso de que dispusiéramos de una imagen a color serían necesarios tres canales distintos, es decir, rojo, verde y azul, utilizando entonces un total de 80 x 80 x 3 = 19200 neuronas de entrada para la capa de entrada de la red.

La arquitectura de una red neuronal convolucional empieza aplicando una capa convolucional y una capa de pooling, siguiendo este proceso repetidamente hasta obtener un conjunto de matrices, tal que al poner todos los elementos como un vector (proceso de complementación), tenga una cantidad computacionalmente viable de elementos como para ser introducido como entrada de una red neuronal.



1.- Arquitectura de una CNN.

Dataset

Las redes neuronales convolucionales están basadas en la idea de que las características se replican: como un objeto se mueve y aparece en diferentes píxeles, un detector de una característica que funciona en una parte de la imagen es probable que también vaya a funcionar en otra parte de la imagen.

El objetivo es construir varias copias del mismo detector de características en diferentes posiciones y para una misma posición tener muchos detectores de características, de manera que cada parte de la imagen pueda ser representada con características de diferentes tipos. Replicar a través de posición ayuda a reducir el número de parámetros a aprender (Pusiol, P. D. 2014).

Para poder entrenar a nuestra red neuronal es necesario tener un conjunto de datos tanto verdaderos como falsos (en este caso los datos son imágenes de rostros), y dichos datos requieren una dimensionalidad de entrada constante. Por lo tanto, redujimos la muestra de las imágenes a una resolución fija de 80 × 80, esta resolución se logró con ayuda de un software, esto para evitar una sobrecarga en el programa de Python.

Las imágenes de los rostros se tomaron de una pagina de internet cuyo enlace es el siguiente: https://www.kaggle.com/ciplab/real-and-fake-face-detection, de las cuales se tomaron 700 reales y 700 no reales para el entrenamiento y 10 reales y 10 no reales para probar la CNN.

Convolución

El nombre de redes neuronales convoluciones tiene su origen en la operación que se realiza en algunas de sus capas. Se sustituye la operación de multiplicación matricial de entradas por pesos por la operación matemática de convolución.

La convolución consiste en filtrar una imagen usando una máscara, diferentes mascaras llevan a distintos resultados. En la convolución, cada píxel de salida es una combinación lineal de los pixeles de entrada por lo que las máscaras representan la conectividad entre las capas sucesivas (Loncomilla, P. 2016).

Si hablamos matemáticamente, una convolución es una operación aplicada a dos funciones con números reales como argumentos. La salida de esta operación es una tercera función que se interpreta como la modificada de una de las funciones de entrada (Aguado López, I. 2020).

Otras de las diferencias que presentan estas capas es que no todas las imágenes de entrada están conectadas con las imágenes de salida, sino que cada neurona de una capa realiza la operación sobre un conjunto reducido de imágenes de entrada.

Para optimizar y mejorar este procedimiento se hace uso principalmente de tres ideas:

Interacciones dispersas: Como el kernel aplicado tiene un tamaño menor que la imagen de entrada, esto hace que se reduzca el número de parámetros.

Parámetros replicados: Se utilizan los mismos parámetros, pesos por ejemplos, en diferentes neuronas donde las entradas están conectadas a otras regiones de la imagen.

Actividad neuronal equivariante, es decir, implica que, si las entradas cambian, las salidas van a cambiar de manera similar.

La primera convolución es capaz de detectar características primitivas como líneas o curvas. A medida que se van realizando más capas con las convoluciones, los mapas de características serán capaces de reconocer formas más complejas.

Pooling

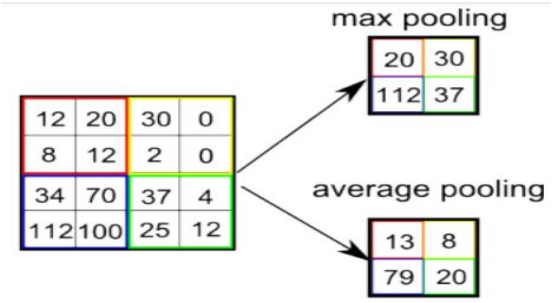
Una vez terminado el proceso anterior, se reduce la cantidad de neuronas antes de realizar una nueva convolución. Como se comentó anteriormente, partiendo de una imagen de 80x80 tenemos una primera capa de entrada de 6400 neuronas. Si se llevara a cabo una nueva convolución a partir de esta capa, el número de neuronas de la próxima capa crecería exponencialmente implicando un mayor procesamiento. Para reducir el tamaño de la próxima capa se realiza el proceso de submuestreo en el que se reduce el tamaño de las imágenes filtradas, pero donde prevalecen las características más importantes que detectó cada filtro.

Como se ha mencionado un aspecto importante es el elevado coste computacional que conlleva procesar imágenes de una red neuronal. Para solventar este problema, se utilizan las llamadas capas de Pooling o reducción de muestreo. La función de estas capas es reducir las dimensiones espaciales (ancho y alto) sin afectar a la profundidad para que las capas posteriores reciban datos de menor tamaño. Esta reducción produce que se pierda información, aunque tiene ventajas:

- Disminución del tamaño para tener una menor sobrecarga de cálculo en las capas posteriores.
- Reducción del sobreajuste.

Para realizar esta reducción de muestreo existen dos tipos de funciones destacadas que se suelen usar:

- Técnica Max - pooling: Esta operación divide la imagen de entrada en un conjunto de rectángulos y va escogiendo el valor máximo. La forma más común de realizarlo es aplicarlo en zonas 2 x 2 lo que reduce la muestra un 75 %.
- Técnica Average - pooling: En esta operación lo que se hace es coger la media aritmética de los valores de la región en lugar del máximo.



2.- Tipos de reducción de muestreo.

Como se ha mencionado existen diversos tipos de submuestreo, pero el más utilizado es el Max-Pooling. Si suponemos que se realiza un Max-pooling de tamaño 2x2 se recorrerán cada una de las imágenes de características obtenidas anteriormente de 80x80 pixeles de izquierda a derecha, arriba-abajo, pero en lugar de a un solo pixel se toman de 2x2 es decir, 2 de alto por 2 de ancho y se preserva el valor más elevado de esos 4 pixeles, de ahí el termino Max.

Softmax

La función softmax (función exponencial normalizada) es una generalización de la función sigmoidea y popularizada por las redes neuronales convolucionales. Se utiliza como función de activación de salida para la clasificación multiclase, ya que escala las entradas precedentes de un rango entre 0 y 1 y normaliza la capa de salida, de modo que la suma de todas las neuronas de salida sea igual a la unidad (Artola Moreno, Á. 2019).

En otras palabras, la función Softmax permite que para cada posible clase se tenga una probabilidad de que la entrada a la red neuronal pertenezca a cada clase, donde la suma de las probabilidades es 1.

Programa

El programa empleado para la clasificación de rostros es el siguiente:

```
1 # Universidad Autónoma Chapingo
2 # Ingeniería Mecatrónica Agrícola
3 # Jym Emmanuel Cocotle Lara 7° 7
4
5 # Importamos las librerías a ocupar
6 import os
7 import numpy as np
8 from numpy import asarray
9 import cv2
```

```
import matplotlib.pyplot as plt


# Directorio donde se encuentran las imágenes que se ocuparán para entrenar y para probar a la CNN
Data_train = "Data/training"  # Imágenes para entrenar
Data_det = "Data/detection"  # Imágenes para probar"


# Convolución
class Convolucion:
    # Filtros requeridos para la convolución
    def __init__(self, num_filt, tam_filt):
        self.num_filt = num_filt
        self.tam_filt = tam_filt
        self.conv_filt = np.random.randn(num_filt, tam_filt, tam_filt)/(tam_filt * tam_filt)

    # Obtenemos los parches de ruta de las imágenes
    def image_region(self,image):
        height, width = image.shape
        self.image = image
        # Extraemos las medidas de los parches
        for j in range(height - self.tam_filt + 1):
            for k in range(width - self.tam_filt +1):
                # Se crea una imagen a partir de los valores de los parches.
                image_patch = image[j:(j+self.tam_filt),k:(k+self.tam_filt)]
                yield image_patch, j, k

    # Propagación hacia adelante
    def forward_prop(self,image):
        height, width = image.shape
        sal_conv = np.zeros((height - self.tam_filt + 1, width - self.tam_filt+1, self.num_filt))
        for image_path, i,j in self.image_region(image):
            # Multiplicación de los filtros con el parche de la imagen
            sal_conv[i,j]= np.sum(image_path * self.conv_filt, axis=(1,2))
        return sal_conv

    # Propagación hacia atrás
    def back_prop(self, dL_dout, learning_rate):
        dL_dF_params = np.zeros(self.conv_filt.shape)
```

```
        for image_patch, i,j in self.image_region(self.image):
            for k in range(self.num_filt):
                dL_dF_params[k] += image_patch*dL_dout[i,j,k]

    # Actualización de los parámetros del filtro
        self.conv_filt -= learning_rate*dL_dF_params
        return dL_dF_params


# Max-Pooling
class Max_Pool:
    # Definición del tamaño del filtro
    def __init__(self, tam_filt):
        self.tam_filt = tam_filt

    # Reducción del tamaño de la imagen
    def image_region(self, image):
        # Nueva altura
        new_height = image.shape[0] // self.tam_filt
        # Nuevo ancho
        new_width = image.shape[1] // self.tam_filt
        self.image = image
        # Extracción de los parches de la imagen nueva que se calcula
en la función de convolución
        for i in range(new_height):
            for j in range(new_width):
                image_patch =
image[(i*self.tam_filt):(i*self.tam_filt + self.tam_filt),
(j*self.tam_filt):(j*self.tam_filt + self.tam_filt)]
                yield image_patch, i, j

    # Propagacion hacia adelante
    def forward_prop(self,image):
        height, width, num_filt = image.shape
        output = np.zeros((height // self.tam_filt, width //
self.tam_filt, num_filt))

        for image_patch, i, j in self.image_region(image):
            output[i,j] = np.amax(image_patch, axis = (0,1))

        return output

    # Propagación hacia atrás
    def back_prop(self, dL_dout):
        dL_dmax_pool = np.zeros(self.image.shape)
        for image_patch, i, j in self.image_region(self.image):
```

```
 91             height, width, num_filt = image_patch.shape
 92             maximum_val = np.amax(image_patch,axis = (0,1))
 93 
 94             for i1 in range(height):
 95                 for j1 in range(width):
 96                     for k1 in range(num_filt):
 97                         if image_patch[i1,j1,k1] == maximum_val[k1]:
 98                             dL_dmax_pool[i*self.tam_filt + i1,
j*self.tam_filt + j1, k1]=dL_dout[i,j,k1]
 99             return dL_dmax_pool
100 
101 # Softmax
102 class Softmax:
103 
104     def __init__(self, input_node, softmax_node):
105         # Se inicializa el peso a partir de valores aleatorios
106         self.weight = np.random.randn(input_node,
softmax_node)/input_node
107         # Se crea un sesgo inicial a partir de una matriz de ceros
108         self.bias = np.zeros(softmax_node)
109 
110     # Se multiplican los pesos por las bases
111     def forward_prop(self, image):
112         self.orig_im_shape = image.shape
113         image_modified = image.flatten()
114         self.modified_input = image_modified
115         output_val = np.dot(image_modified, self.weight) + self.bias
116         self.out = output_val
117         exp_out = np.exp(output_val)
118         # Transformación de la salida en las salidas probables dadas
119         return exp_out/np.sum(exp_out, axis=0)
120 
121     # Propagación hacia atrás
122     def back_prop(self, dL_dout, learning_rate):
123         for i, grad in enumerate(dL_dout):
124             if grad ==0:
125                 continue
126             #
127             transformation_eq = np.exp(self.out)
128             S_total = np.sum(transformation_eq)
129 
130             # Gradiente con especto a la salida Z
131             dy_dz = -transformation_eq[i]*transformation_eq /
(S_total **2)
132             dy_dz[i] = transformation_eq[i]*(S_total -
transformation_eq[i]) / (S_total **2)
```

```
133
134             # Gradiente del total con pesos y entradas
135             dz_dw =  self.modified_input
136             dz_db = 1
137             dz_d_inp = self.weight
138
139             # Gradiente total contra el perdido
140             dL_dz = grad * dy_dz
141
142             # Gradiente de pérdida contra pesos, bases y entradas
143             dL_dw = dz_dw[np.newaxis].T @ dL_dz[np.newaxis]
144             dL_db = dL_dz * dz_db
145             dL_d_inp = dz_d_inp @ dL_dz
146
147       # Actualizamos los pesos y las bases
148         self.weight -= learning_rate *dL_dw
149         self.bias -= learning_rate * dL_db
150
151         return dL_d_inp.reshape(self.orig_im_shape)
152
153
154 def dataset_function(directorio,ancho,alto):
155     dataset = []
156     label = ["real","no_real"]
157     for categoria in label:
158         rostro = label.index(categoria)
159         ruta_imagen = os.path.join(directorio,categoria)
160
161         for file_name in os.listdir(ruta_imagen):
162             img =
cv2.imread(os.path.join(ruta_imagen,file_name),cv2.IMREAD_GRAYSCALE)
163             img = cv2.resize(img, (ancho,alto))
164             dataset.append([img,rostro])
165     X = []
166     Y = []
167     for caracteristicas,label in dataset:
168         X.append(caracteristicas)
169         Y.append(label)
170
171     X = np.array(X).reshape(-1,ancho,alto)
172     Y = np.array(Y)
173
174     # Retornamos los valores X,Y
175     return X,Y
176
177
```

```
178 (X_train,y_train) = dataset_function(Data_train, 80,80) #resize
179
180 (X_test,y_test) = dataset_function(Data_det, 80,80)
181
182 # imás del entrenamiento (número de imágenes que tengamos)
183 train_images = X_train[0:1400]
184 train_labels = y_train[0:1400]
185
186 # imágenes de la prueba (número de imágenes que tengamos)
187 test_images = X_test[0:20]
188 test_labels = y_test[0:20]
189
190
191 num_test = len(test_images)
192 num_train = len(train_images)
193
194 # 8 filtros de 3x3
195 conv = Convolucion(8,3)
196 # Operación de encontrar el número mayor
197 pool = Max_Pool(2)
198 # (80 - tamaño de filtro + 1/maxpool)*, numero de imagenes a
compactar
199 softmax = Softmax(39*39*8,5)
200
201
202
203 def cnn_forward_prop(image, label):
204     #  alimentamos a la imagen con la operación de convolución hacia
adelante
205     out_p = conv.forward_prop((image /255) - 0.5)
206     # le pasamos el parámetro a la operación de max pool
207     out_p = pool.forward_prop(out_p)
208     # nuevamente le pasamos el parámetro a la función de softmax
209     out_p = softmax.forward_prop(out_p)
210
211     # calculamos la pérdida de la entropía y la precisión
212     cross_ent_loss = -np.log(out_p[label])
213     accuracy_eval = 1 if np.argmax(out_p) == label else 0
214
215
216     return out_p, cross_ent_loss, accuracy_eval
217
218 # vamos a entrenar a la CNN a través de la propagación hacia atrás
219 # obtenemos los resultados de la salida y alimentamos hacia atrás el
error a las capas anteriores
220 def training_cnn(image, label, learn_rate = 0.005):
```

```
    # Se calcula la salida, la pérdida y la precisión
    out, loss, acc = cnn_forward_prop(image, label)

    # Se calcula el gradiente inicial
    gradient = np.zeros(10)
    gradient[label] = -1 / out[label]

    # propagación hacia atrás
    # le pasamos el valor del gradiente inicial a las funciones
    grad_back = softmax.back_prop(gradient, learn_rate)
    grad_back = pool.back_prop(grad_back)
    grad_back = conv.back_prop(grad_back, learn_rate)

    return loss, acc

# Se entrena a la CNN de acuerdo con el número de épocas que se
quieran

for epocas in range(5):
    print("Epoca numero: %d "% (epocas +1))

    # 1400 imágenes se dividen en parches y cada parche tiene 100
imágenes
    shuffle_data = np.random.permutation(len(train_images))
    train_images = train_images[shuffle_data]
    train_labels = train_labels[shuffle_data]

    loss = 0.0
    num_correct = 0

    for i, (im, label) in enumerate(zip(train_images, train_labels)):
        #
        if i % 100 == 0:
            # Se muestra en la consola el número de paso en el que se
encuentra el entrenamiento
            # de acuerdo con la época en la que se encuentra
            print('No. pasos: %d/%d Pérdida promedio: %.3f
Precisión: %d%%' %(i+1, num_train,loss/100, num_correct))
            loss = 0
            num_correct = 0

        # Se calcula el entrenamiento de la red llamando a la función
antes creada
        l1, accu = training_cnn(im, label)
        loss += l1
        num_correct +=accu
```

```
262
263 # Puesta a prueba de la CNN
264 print("Puesta a prueba...")
265 loss = 0 # Variable para mostrar la perdida
266 num_correct = 0 # Variable conocer el número de imágenes correctas
267 n = 0 # Identificador del número de imagen
268
269
270
271 for i,(im, label) in enumerate(zip(test_images, test_labels)):
272     __, l1, accu = cnn_forward_prop(im, label)
273     loss += l1
274     num_correct += accu
275
276     if (accu == 1):
277         plt.imshow(X_test[n],cmap="gray")
278         plt.title(f"Sí existe un rostro")
279         plt.show()
280
281     else:
282         plt.imshow(X_test[n],cmap="gray")
283         plt.title(f"No existe un rostro")
284         plt.show()
285
286     n+=1
287
288 # Cálculo de la precisión final
289 precision = (num_correct / num_test)*100
290 # cálculo de la perdida final
291 perdida = loss /num_test
292 # Se visualiza la perdida y la precisión
293 print("Precisión:", precision)
294 print("\nPérdida: ", perdida)
295
```

Teniendo el anterior código, se ejecutó con 5 épocas, y con 1400 imágenes, 700 reales y 700 falsas.

Resultados

Una vez que se terminó de ejecutar el programa se obtuvieron los siguientes resultados:


Figure 1
Sí existe un rostro
0
10
20
30
40
50
60
70
0 10 20 30 40 50 60 70
x=55.7 y=24.7
[37]


Figure 1
Sí existe un rostro
0
10
20
30
40
50
60
70
0 10 20 30 40 50 60 70

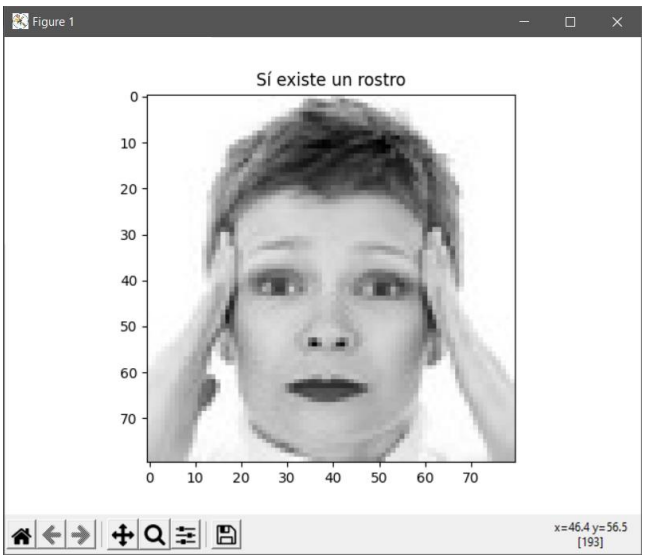


Figure 1
Sí existe un rostro
0
10
20
30
40
50
60
70
0 10 20 30 40 50 60 70
x=46.4 y=56.5
[193]

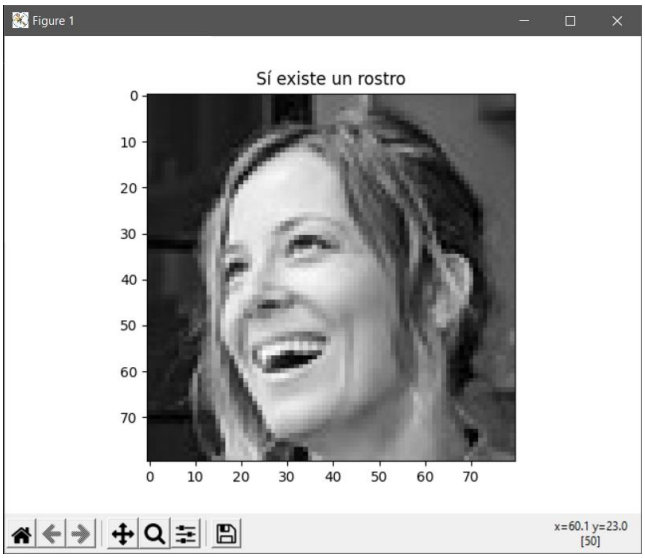


Figure 1
Sí existe un rostro
0
10
20
30
40
50
60
70
x=60.1 y=23.0
[50]


Figure 1
Sí existe un rostro
0
10
20
30
40
50
60
70

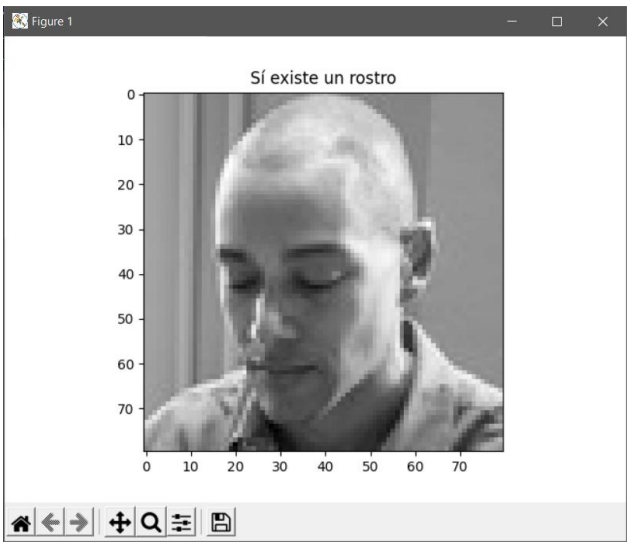


Figure 1
Sí existe un rostro
0
10
20
30
40
50
60
70

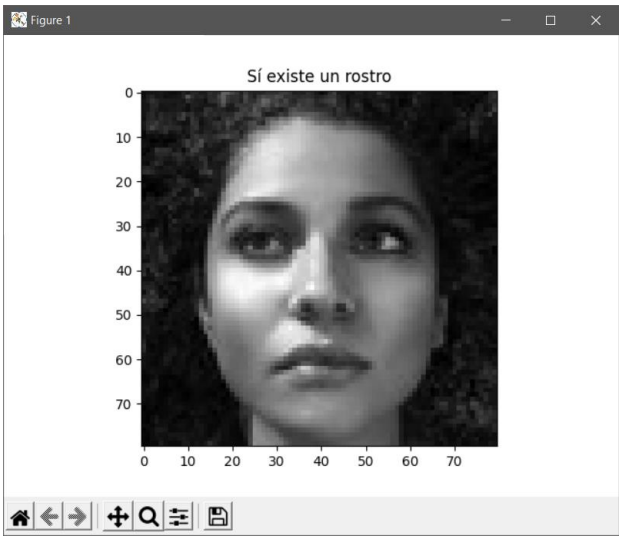


Figure 1
Sí existe un rostro
0
10
20
30
40
50
60
70
0 10 20 30 40 50 60 70


Figure 1
Sí existe un rostro
0
10
20
30
40
50
60
70
0 10 20 30 40 50 60 70
x=68.5 y=53.0
[50]


Figure 1
Sí existe un rostro
0
10
20
30
40
50
60
70
0 10 20 30 40 50 60 70


Figure 1
Sí existe un rostro
0
10
20
30
40
50
60
70
0 10 20 30 40 50 60 70

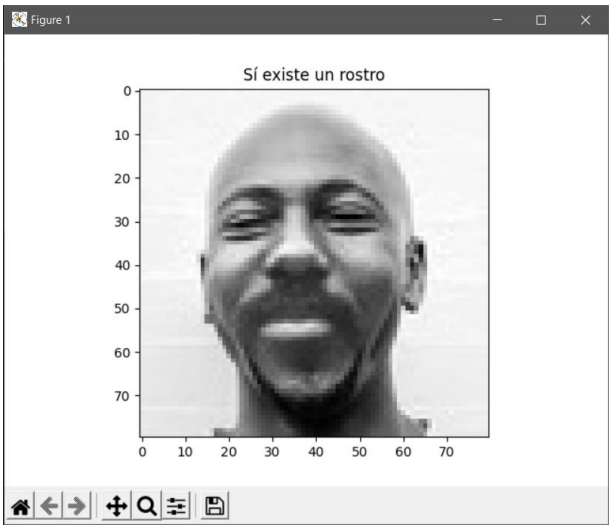


Figure 1
Sí existe un rostro
0
10
20
30
40
50
60
70
0 10 20 30 40 50 60 70

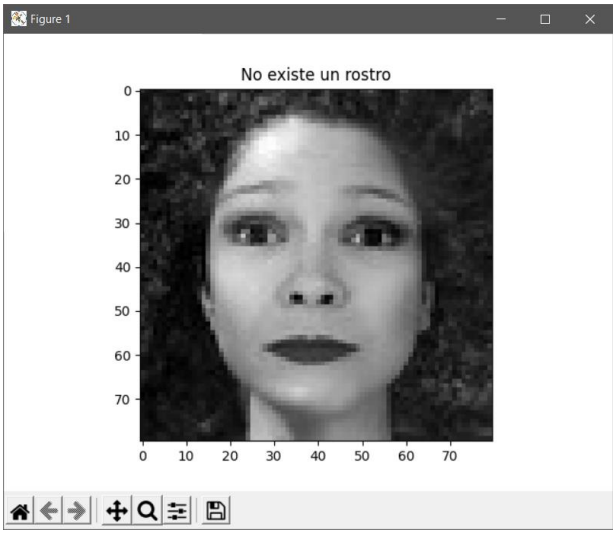
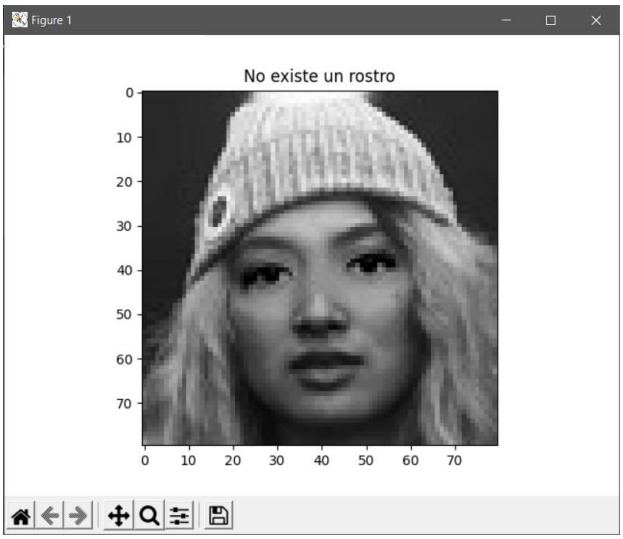
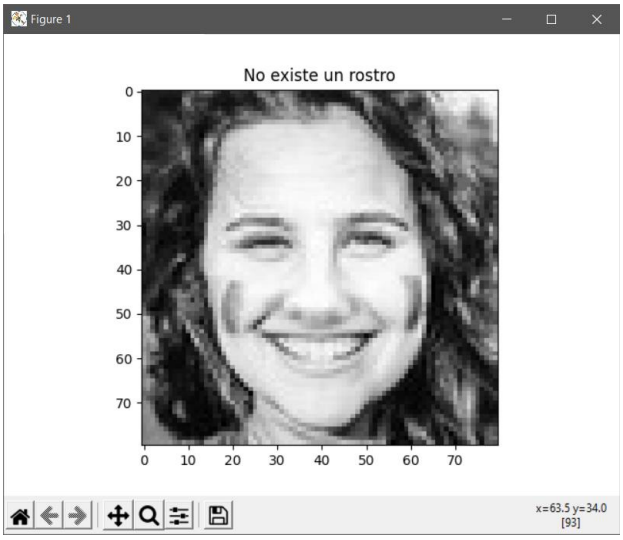


Figure 1
No existe un rostro
0
10
20
30
40
50
60
70
0 10 20 30 40 50 60 70





El entrenamiento para poder identificar los rostros tardo aproximadamente media hora.

Después de que se entrenó a la CNN con 5 épocas, se obtuvo una precisión del 75% y con una pérdida de 0.5923.

Conclusión

Una vez que se entreno a la CNN con las 1400 imágenes se pudo hacer el reconocimiento de rostros, sin embargo, dicho reconocimiento no tiene una precisión tan alta, y esta precisión puede ser mejorada si se aumenta el numero de épocas, sin embargo, se tiene que tener en cuenta que si se realizan muchas épocas el tiempo para que la CNN sea entrenada aumentara en gran medida, y además, podría existir el problema que la red neuronal solo pueda identificar las imágenes con las que se entrenó, por lo que no tendríamos un reconocimiento general si no uno mas especializado.

Aunque es una red neuronal muy potente esta puede ser optimizada ya que la actual tarda demasiado en completar una época por lo que si aumentáramos la resolución de las imágenes,

este tiempo aumentaría aun más, una de las opciones de optimización podría ser tratar a las imágenes como números, es decir, convertir las imágenes de entrenamiento en un array el cual puede ser almacenado en un archivo de Excel, con lo cual solo referenciaríamos al archivo de Excel y podríamos agilizar el entrenamiento.

Bibliografía

Artola Moreno, Á. (2019). Clasificación de imágenes usando redes neuronales convolucionales en Python.

Sánchez, E. G. (2019). Introducción a las redes neuronales de convolución. Aplicación a la visión por ordenador.

Loncomilla, P. (2016). Deep learning: Redes convolucionales. Recuperado de https://ccc. inaoep.mx/pgomez/deep/presentations.

Pusiol, P. D. (2014). Redes convolucionales en comprensión de escenas (Bachelor's thesis).

Matas González, I. (2021). Clasificación de imágenes mediante Redes Neuronales Convolucionales y técnicas de Deep Learning avanzadas: Transformers.

Fukushima, K., & Miyake, S. (1982). Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition. In Competition and cooperation in neural nets (pp. 267-285). Springer, Berlin, Heidelberg.

Aguado López, I. (2020). Deep Learning: Redes Neuronales Convolucionales en R.