Reconocimiento de caracteres con redes neuronales

Julián Garzón, Alejandro Martínez, Yery Pedraza, Oscar Rodríguez, y Santiago Téllez
est.{julian.garzon2, nicolasa.marti1, yery.pedraza, oscar.rodriguez9 y santiago.téllez}@unimilitar.edu.co
Profesor: Camilo Hurtado

***Resumen*—Se implementaron dos modelos de redes neuronales par el reconocimiento de caracteres en Python por medio de una interfaz de usuario y visión de máquina. El primer modelo, consta de una serie de funciones desarrolladas matemáticamente para una red neuronal de dos capas ocultas, capaz de leer imágenes del dataset de MNIST. El segundo modelo, fue desarrollado con ayuda de la librería Keras con la misma arquitectura del modelo anterior. Para la visión de máquina utilizamos la librería OpenCV y para la interfaz gráfica usamos TKinter.**

***Palabras clave*—Reconocimiento de caracteres, redes neuronales, Python, Keras, GUI, OpenCV.**

I. Objetivos

- Diseñar y construir un sistema de reconocimiento de caracteres basado en redes neuronales
- Implementar un software para la creación, entrenamiento (algoritmo de Back-Propagation) y uso de redes neuronales

II. Desarrollo de la práctica

Para el desarrollo de este conjunto de objetivos propuestos de optó por crear dos modelos: manual y Keras, los cuales fueron comparados para tener mejores resultados. El modelo que fue escogido, se le agregó visión de máquina para que pudiera leer imágenes desde un directorio y así poder reconocer el patrón al que pertenece. Todo esto a través de una interfaz de usuario que permita mostrar la imagen, un vector con el porcentaje de pertenencia a cada carácter y su predicción final.

II-A. Descripción de patrones a reconocer

El reconocimiento de caracteres fue implementado por medio del complemento del dataset MNIST, el cual nos permite entrenar un conjunto de imágenes de dígitos con un total de más de 60.000 imágenes de entrenamiento y 10.000 de muestras o test.

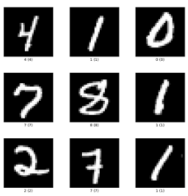

Figura 1. MNIST

El presente documento corresponde a un informe de desarrollo práctica de laboratorio de Inteligencia Artificial presentado en la Universidad Militar Nueva Granada durante el periodo 2022-1.

El paquete de imágenes es normalizado por medio de un grid de 20x20 pixels donde se mantiene las proporciones de la imagen original, el procesamiento de las imágenes se desarrollan por medio de la herramienta TensorFlow Keras donde se llevan a cabo los siguientes pasos

• Importación de conjunto de datos: En la importación y adquisición de los datos no es necesario descargar y almacenar lo datos de forma externa ya que por medio de la librería MNIST permite la importación directa del conjunto de datos, seguido de esto se procede a almacenar los datos en variables como mnist.loaddata().

• Dividir el conjunto en bloques de entrenamiento y prueba: Teniendo en cuenta el dataset implementado en el código se realiza cuatro variables Xtrain, ytrain, Xtest y y test donde se realiza la respectiva independencia de los datos para su pronto procesamiento.

• Construcción del modelo: Con la implementación de las librerías para las funciones requeridas en la red neuronal por medio de las biblioteca de Keras, se procede a desarrollar el modelo secuencial con diferentes capas densas con funciones de activación, además de desarrollar los valores de los hiperparametros del modelo.

• Entrenamiento del modelo y resultado: Por ultimo se realiza el entrenamiento de la red neuronal teniendo en cuenta los hiperparametros asignados y la validación con respecto al loss y al accurracy del sistema.

II-B. Implementación del modelo manual

Para el modelo manual se usó Python y Numpy. El primer paso es analizar la arquitectura de la red neuronal, que es definida por medio de un vector así:

```
model = [784, 16, 8, 10]
```

donde primero definimos la entrada de la red neuronal que en éste caso, dado que las imágenes son de tamaño 28x28 al vectorizarlas queda un vector de 784 columnas. Posteriormente se van definiendo las capas ocultas con su respectivo numero de neuronas; en éste caso, tenemos dos capas ocultas con 16 y 8 neuronas respectivamente y una capa de salida con 10 neuronas que representan los diez caracteres a reconocer.

El siguiente paso es la inicialización de los parámetros (los pesos y el bias), que lo hacemos de manera aleatoria teniendo en cuenta el numero de capas que se usaron por medio de un diccionario donde cada llave representa los pesos (W) y los bias (b). Una breve representación:

```
def Initialize_params(model):
    parameters = {}
    L = len(model)
    # Initial hyperparams creation:
    for l in range(0, L-1):
        parameters['W' + str(l+1)] =  (np.random.rand(model[l],model[l+1]) * 2 ) - 1 # Weights (W)
        parameters['b' + str(l+1)] =  (np.random.rand(1,model[l+1]) * 2 ) - 1 # Bias (b)
    return parameters
```

Figura 2. Función de inicialización de parámetros

Luego definimos algunas variables como lo son el *Learning rate* para el gradiente y el número de *épocas de entrenamiento* con un valor de 0.1 y 50 respectivamente. Después empezamos el ciclo de entrenamiento que está alojado en un ciclo que se completa hasta llegar al número de épocas definidas. El primer paso para entrenar es el *Forward Propagation*; en este paso agregamos al diccionario de parámetros los números asociados (Z) a la multiplicación matricial entre los pesos (W) y los datos de la capa anterior (A) sumándoles el bias (b) de la respectiva capa (Figura 3). Posteriormente se hace el proceso de *Back Propagation*; donde se hallan los diferenciales en los pesos de cada neurona (Figura 4) comparándolos con la respuesta real por medio de la función de error MSE (Figura 5) y usando las diversas funciones de activación (Figura 6) guardando los valores en el diccionario de parámetros. Por último, se realiza el *Ajuste de pesos* por medio del gradiente (Figura 7).

```
def Forward(params, x_data):

    params['A0'] = x_data

    params['Z1'] = (params['A0']@params['W1']) + params['b1']
    params['A1'] = relu( params['Z1'])

    params['Z2'] = (params['A1']@params['W2']) + params['b2']
    params['A2'] = relu(params['Z2'])

    params['Z3'] = (params['A2']@params['W3']) + params['b3']
    params['A3'] = sigmoid(params['Z3'])

    output = params['A3']

    return params, output
```

Figura 3. Función Forward

```
def BackPropagation(params, y_train, d):
    params['dZ3'] = mse(y_train, params['A3'], d) * sigmoid(params['A3'], d)
    params['dW3'] = params['A2'].T@params['dZ3']

    params['dZ2'] = params['dZ3']@params['W3'].T * relu(params['A2'], d)
    params['dW2'] = params['A1'].T@params['dZ2']

    params['dZ1'] = params['dZ2']@params['W2'].T * relu(params['A1'], d)
    params['dW1'] = params['A0'].T@params['dZ1']

    return params
```

Figura 4. Función BackPropagation

```
def mse(y, y_hat, d = False):
    if d:
        return y_hat-y
    else:
        return np.mean((y_hat - y)**2)
```

Figura 5. Función de error

```
def relu(x, derivate = False):
    if derivate:
        x[x<=0] = 0
        x[x>0] = 1
        return x
    else:
        return np.maximum(0,x)

def sigmoid(x, derivate = False):
    if derivate:
        return np.exp(-x)/((np.exp(-x)+1)**2)
    else:
        return (1/(1+np.exp(-x)))
```

Figura 6. Funciones de activación

```
def WeightAdjust(params, lr):
    params['W3'] = params['W3'] - params['dW3'] *  lr
    params['b3'] = params['b3'] - (np.mean(params['dZ3'], axis=0, keepdims=True)) * lr

    params['W2'] = params['W2'] - params['dW2'] *  lr
    params['b2'] = params['b2'] - (np.mean(params['dZ2'], axis=0, keepdims=True)) * lr

    params['W1'] = params['W1'] - params['dW1'] *  lr
    params['b1'] = params['b1'] - (np.mean(params['dZ1'], axis=0, keepdims=True)) * lr

    return params
```

Figura 7. Función ajuste de pesos

Los resultados del error entre la respuesta correcta y la respuesta dada entrenamiento fueron los siguientes:

II-C. Implementación del modelo con Keras

Para el segundo modelo se usó la librería Keras para redes neuronales. Lo primero que haremos es construir el modelo de la misma forma que el anterior para poder compararlos. Definimos el modelo de tal manera:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (32, 784)                 0
_________________________________________________________________
dense (Dense)                (32, 16)                  12560
_________________________________________________________________
dense_1 (Dense)              (32, 8)                   136
_________________________________________________________________
dense_2 (Dense)              (32, 10)                  90
=================================================================
Total params: 12,786
Trainable params: 12,786
Non-trainable params: 0
_________________________________________________________________
```

Figura 8. Modelo en Keras

Posteriormente se hace el entrenamiento y se obtiene el siguiente resultado:

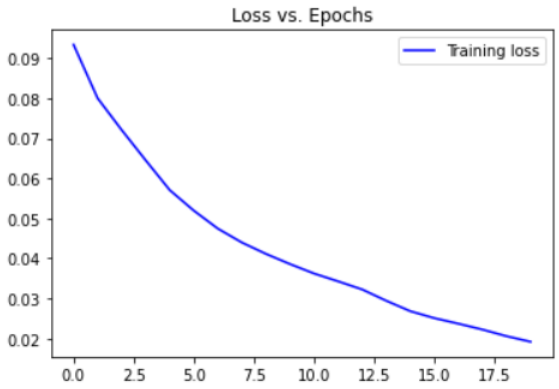

Figura 9. MSE en entrenamiento

II-D. Ajuste del modelo

De acuerdo a la comparación de los resultados obtenidos se decidió usar el método con Keras para el desarrollo de la práctica. Se realizaron siete pruebas con distintos modelos para mejorar el rendimiento de la predicción de la red neuronal.

El primer modelo está construido de la siguiente manera:

Layer (type)	Output Shape	Param #
flatten (Flatten)	(20, 784)	0
dense (Dense)	(20, 32)	25120
dense_1 (Dense)	(20, 10)	330

Total params: 25,450
Trainable params: 25,450
Non-trainable params: 0

Figura 10. Modelo 1

Y entregó los siguientes resultados:

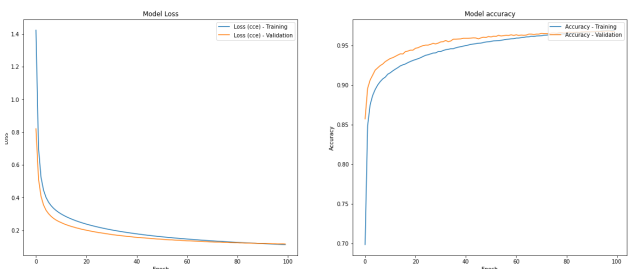

Figura 11. Resultados modelo 1

El segundo modelo está construido de la siguiente manera:

Layer (type)	Output Shape	Param #
flatten_1 (Flatten)	(20, 784)	0
dense_2 (Dense)	(20, 32)	25120
dense_3 (Dense)	(20, 10)	330

Total params: 25,450
Trainable params: 25,450
Non-trainable params: 0

Figura 12. Modelo 2

Y entregó los siguientes resultados:

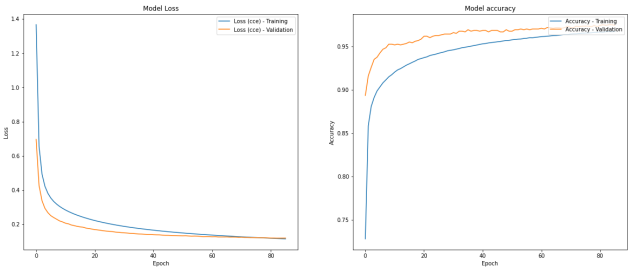

Figura 13. Resultados modelo 2

El tercer modelo está construido de la siguiente manera:

Layer (type)	Output Shape	Param #
flatten_2 (Flatten)	(20, 784)	0
dense_4 (Dense)	(20, 64)	50240
dense_5 (Dense)	(20, 10)	650

Total params: 50,890
Trainable params: 50,890
Non-trainable params: 0

Figura 14. Modelo 3

Y entregó los siguientes resultados:

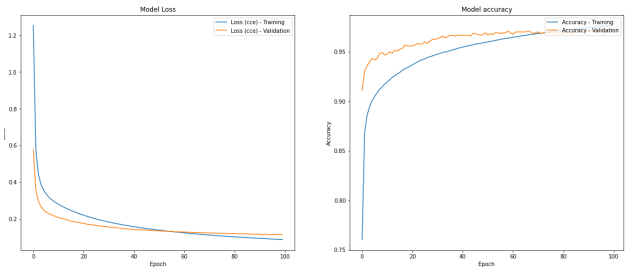

Figura 15. Resultados modelo 3

El cuarto modelo está construido de la siguiente manera:

Layer (type)	Output Shape	Param #
flatten_4 (Flatten)	(20, 784)	0
dense_8 (Dense)	(20, 64)	50240
dense_9 (Dense)	(20, 10)	650

Total params: 50,890
Trainable params: 50,890
Non-trainable params: 0

Figura 16. Modelo 4

Y entregó los siguientes resultados:

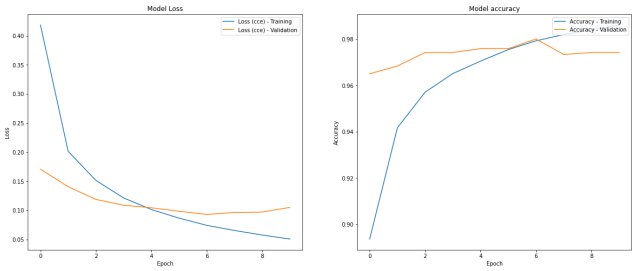

Figura 17. Resultados modelo 4

El quinto modelo está construido de la siguiente manera:

Layer (type)	Output Shape	Param #
flatten_5 (Flatten)	(20, 784)	0
dense_10 (Dense)	(20, 64)	50240
dense_11 (Dense)	(20, 10)	650

Total params: 50,890
Trainable params: 50,890
Non-trainable params: 0

Figura 18. Modelo 5

Y entregó los siguientes resultados:

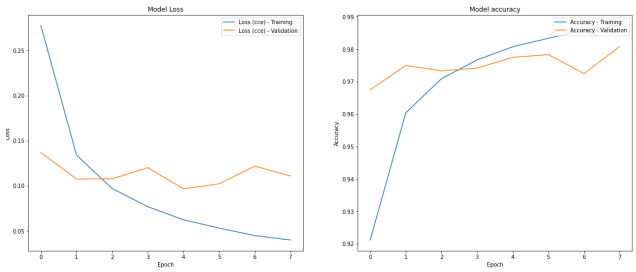

Figura 19. Resultados modelo 5

El sexto modelo está construido de la siguiente manera:

Model: "sequential_6"

Layer (type)	Output Shape	Param #
flatten_6 (Flatten)	(20, 784)	0
dense_12 (Dense)	(20, 32)	25120
dense_13 (Dense)	(20, 64)	2112
dense_14 (Dense)	(20, 10)	650

Total params: 27,882
Trainable params: 27,882
Non-trainable params: 0

Figura 20. Modelo 6

Y entregó los siguientes resultados:

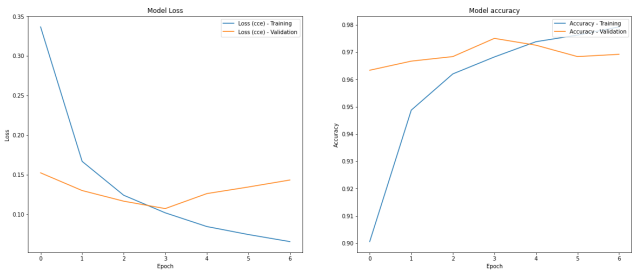

Figura 21. Resultados modelo 6

El séptimo modelo está construido de la siguiente manera:

Layer (type)	Output Shape	Param #
flatten_7 (Flatten)	(20, 784)	0
dense_15 (Dense)	(20, 64)	50240
dense_16 (Dense)	(20, 96)	6240
dense_17 (Dense)	(20, 10)	970

Total params: 57,450
Trainable params: 57,450
Non-trainable params: 0

Figura 22. Modelo 7

Y entregó los siguientes resultados:

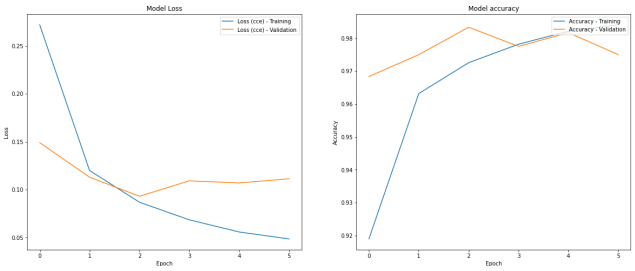

Figura 23. Resultados modelo 7

Posteriormente todos los modelos fueron puesto a prueba con datos de testeo (es decir, que no conocía antes) para evaluar su desempeño y elegir el mejor. La siguiente gráfica muestra los resultados:

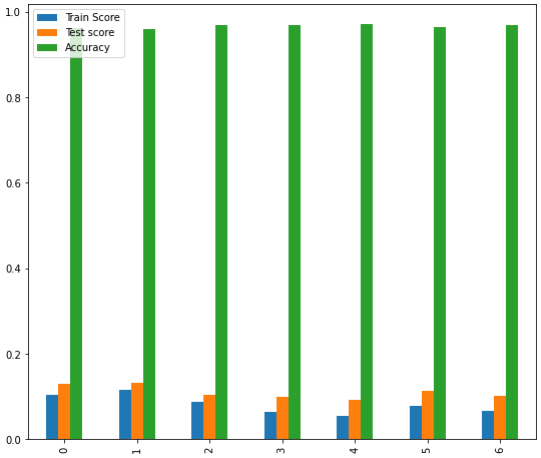

Figura 24. Resultados en entrenamiento y test (Azul y narnaja). Accuracy (Verde)

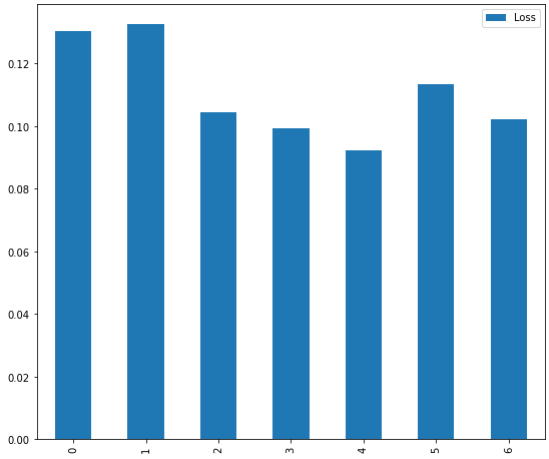

Figura 25. Pérdida de cada modelo

Se optó por el quinto modelo (llamado model_4) ya que es el que tiene menos error con respecto a los demás, aunque la diferencia entre un modelo y otro no es mucha. Aún así demostramos que existen diferencias entre cada implementación.

II-E. Implementación de visión de máquina

Para la identificación de caracteres, se emplea la librería Opencv, numpy y tensorflow. Adicionalmente, y para la obtención de la imagen se hace uso del programa Droidcam, que permite usar la cámara del dispositivo móvil como cámara web.

Por otro lado, con el objetivo de probar el código, se toma una fotografía que servirá para definir el procesamiento de la imagen y los métodos útiles para identificar los caracteres en las imágenes.

Con el fin de identificar los bordes a leer, y de predecir los valores que hay en las imágenes, se pasa la imagen a escala de grises, escala en la que resulta más sencilla para realizar contrastes en posteriores etapas del procesamiento.

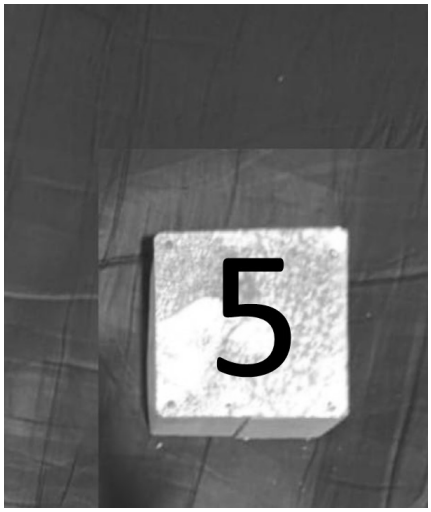

Figura 26. Imagen a escala de grises

Debido al tipo de superficie en la que se encuentra dibujado el número, y la posibilidad de brillos fuertes en la imagen, se decide realizar una reducción de ruido en la imagen. Obteniendo la siguiente imagen de salida.

Figura 27. Imagen con reducción de ruido

Una vez filtrado el ruido de la imagen, se realiza la umbralización de la imagen (donde todo lo que esté en cierto rango de valores se asignará el valor máximo (255) y lo que no, el valor mínimo (0), esto con el fin de generar un alto contraste entre el fondo y la superficie de interés, para que resulte más sencillo identificar los bordes de la imagen.

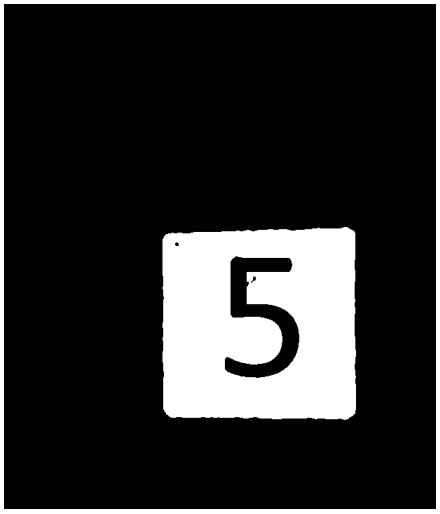

Figura 28. Umbralización

La librería open-cv cuenta con métodos con los que se puede realizar la identificación de bordes de una imagen a partir de los contrastes existentes en las mismas. Este método se conoce como findcontours, función a la que se le debe ingresar el tipo de aproximación a realizar, como en el caso particular se desea aproximar a una geometría simple, se ingresa "chain_aprox_simpe" obteniendo el siguiente resultado.

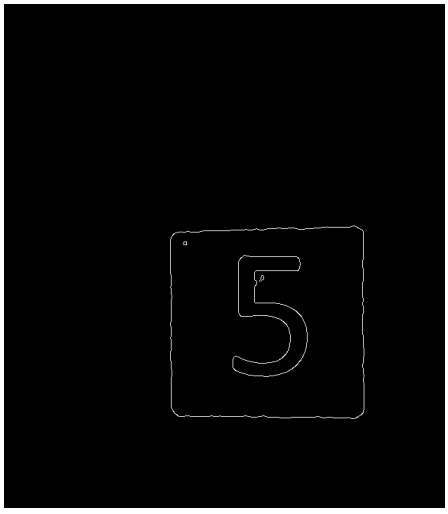

Figura 29. Detección de bordes

Si bien el resultado obtenido es similar a un cuadrado, este presenta demasiados vértices como para ingresarlos de una manera regular. Por lo anterior, se debe realizar una aproximación a figura geométrica que más se acerque a la forma obtenida. A partir de lo anterior, se toman los puntos de esta figura geométrica y se dibuja un cuadrado con las mismas dimensiones aproximadas buscando delimitar el área en la que se quiere detectar el símbolo.

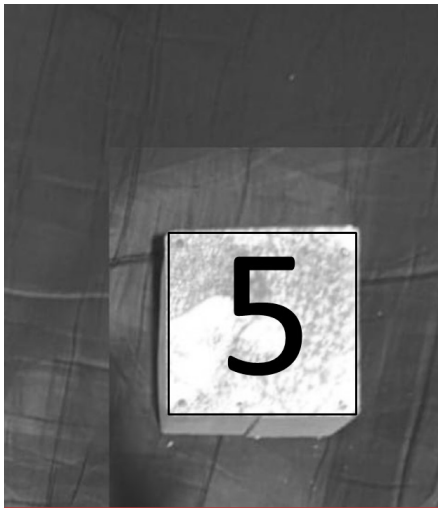

Figura 30. Área de detección de caractér

Finalmente, se toman los valores que se encuentran dentro del cuadro enmarcado anteriormente, se aplica moralización nuevamente, aunque, a diferencia de la anterior vez que se realizó este procedimiento, y con el fin de hacer que los datos sean lo más parecido posible al dataset de entrenamiento (mnist) se invierten los colores de la imagen que se ingresará al modelo entrenado.

Figura 31. Imagen entrante a la red neuronal

Este dato 31, es ingresado al modelo previamente entrenado (importado con la librería tensorflow), que por medio del método predict realizará una predicción del valor en la imagen ingresada.

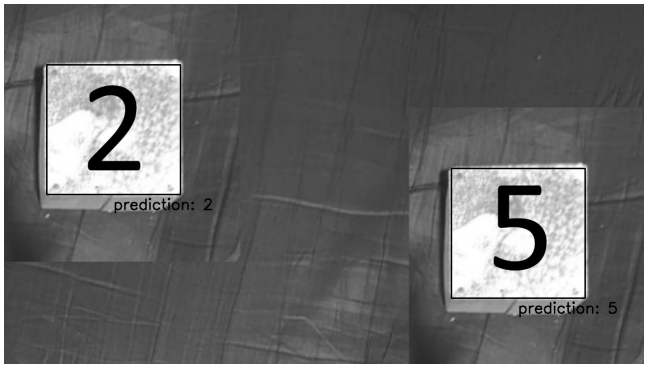

Figura 32. Predicción de las imágenes

II-F. Implementación de interfaz gráfica de usuario

La implementación de la interfaz gráfica se lleva a cabo por medio de la herramienta tkinter en el ambiente de programación tkinter, donde como se muestra en la siguiente figura consta de un panel donde se presenta la imagen del número a estimar, seguido de dos botones que realizan los eventos del modelo 1 y 2 de la red neuronal, además se plotean en la parte derecha de la interfaz los valores del vector y la predicción según el número y la imagen implementada.

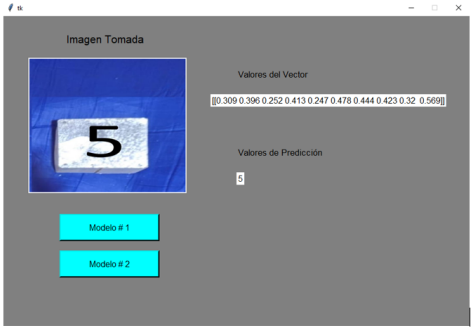

Figura 33. Interfaz grafica implementada.

Con respecto a la programación en el ambiente de Python se realiza en primer lugar las dimensiones y estilo de la interfaz, seguido de esto se realizan dos funciones las cuales contienen el procedimiento de predicción de caracteres con respecto a cada modelo implementado, como se ilustra en la siguiente figura, teniendo en cuenta los eventos asignados en las funciones se realiza el llamado de los eventos en la configuración de cada botón asignado en la interfaz gráfica de tkinter.

```python
def modelo1():
    print("Modelo1")

    model = tf.keras.models.load_model('model_1.h5')
    image = cv2.imread('imagen1.jpg')

    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    No_Noise = cv2.fastNlMeansDenoising(gray, None,25,7,21)
    _, th = cv2.threshold(No_Noise, 170, 255, cv2.THRESH_BINARY) #Modificar a conveniencia
    border = cv2.Canny(th, 170, 200) #Modificar a conveniencia

    contours, hierarchy = cv2.findContours(border, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
    idGray = gray.copy()
    points = []
    points.append([0, np.array(image).shape[0]//2])
    for cont in contours:

        epsilon = 0.01*cv2.arcLength(cont, True) #Modificar a conveniencia
        aprox =  cv2.approxPolyDP(cont, epsilon, True)
        aprox = aprox.reshape(aprox.shape[0], aprox.shape[-1])

        cv2.rectangle(idGray, (aprox[1, 0], aprox[1, 1]), (aprox[-1, 0], aprox[-1, 1]), (0,0,255), 2) #x, y
        content = idGray[ aprox[1, 1]:aprox[-1, 1], aprox[1, 0]:aprox[-1, 0] ]
        content = cv2.resize(content, (28, 28), interpolation = cv2.INTER_CUBIC) #Tamaño de la imágen re hecha
        # POSIBLES ARREGLOS RED NEURONAL
        _, th2 = cv2.threshold(content, 170, 255, cv2.THRESH_BINARY)
        # print(th2)
        th2 = 255 - th2

        toPredictContent = (th2.reshape((1, 28, 28, 1))).astype('float32') / 255.0
        # prediction = model.predict(toPredictContent)
        prediction = np.argmax(model.predict(toPredictContent)[0], axis=-1)
        print(prediction)
```

Figura 34. Modelo 1 interfaz grafica

```python
def modelo2():
    print("Modelo2")

    params = np.load('handmade_model.npy', allow_pickle=True)
    params2 = params.tolist()
    print(type(params))
    image = cv2.imread('imagen1.jpg')

    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    No_Noise = cv2.fastNlMeansDenoising(gray, None,25,7,21)
    _, th = cv2.threshold(No_Noise, 170, 255, cv2.THRESH_BINARY) #Modificar a conveniencia
    border = cv2.Canny(th, 170, 200) #Modificar a conveniencia

    contours, hierarchy = cv2.findContours(border, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
    idGray = gray.copy()
    points = []
    points.append([0, np.array(image).shape[0]//2])
    for cont in contours:

        epsilon = 0.01*cv2.arcLength(cont, True) #Modificar a conveniencia
        aprox =  cv2.approxPolyDP(cont, epsilon, True)
        aprox = aprox.reshape(aprox.shape[0], aprox.shape[-1])

        cv2.rectangle(idGray, (aprox[1, 0], aprox[1, 1]), (aprox[-1, 0], aprox[-1, 1]), (0,0,255), 2) #x, y
        content = idGray[ aprox[1, 1]:aprox[-1, 1], aprox[1, 0]:aprox[-1, 0] ]
        content = cv2.resize(content, (28, 28), interpolation = cv2.INTER_CUBIC) #Tamaño de la imágen re hecha
        # POSIBLES ARREGLOS RED NEURONAL
        _, th2 = cv2.threshold(content, 170, 255, cv2.THRESH_BINARY)
        # print(th2)
        th2 = 255 - th2

        toPredictContent = (th2.reshape((1, 28, 28, 1))).astype('float32') / 255.0
        toPredictContent = toPredictContent.reshape(1, 784)
        params2, prediction_v = Forward(params2, toPredictContent)
        prediction = np.argmax(prediction_v)
        print("\nPrediction: ", prediction)
        np.set_printoptions(precision = 3, suppress = True)
        print("Ownership Percentage:", prediction_v*100)#5, prediction_v*100)
```

Figura 35. Modelo 2 interfaz grafica

```python
BotonModelo1 = Button(window, text = "Modelo # 1", font=("Helvetica, 12"),command=modelo1, width="20", height="2",bd=3, bg="cyan" )
BotonModelo1.place(x=110, y=380)

BotonModelo2 = Button(window, text = "Modelo # 2", font=("Helvetica, 12"),command = modelo2, width="20", height="2",bd=3, bg="cyan"  )
BotonModelo2.place(x=110, y=450)

label1 = Label(text = "Valores del Vector", font=("Arial, 13"), bg="grey")
label1.place(x=450,y=100)

labelvector = Label(text = modelo_2, font=("Arial, 12"), bg="white")
labelvector.place(x=400,y=150)

label2 = Label(text = "Valores de Predicción", font=("Arial, 13"), bg="grey")
label2.place(x=450,y=250)

labelprediccion = Label(text = modelo_1, font=("Arial, 12"), bg="white")
labelprediccion.place(x=450,y=300)
window.mainloop()
```

Figura 36. Programación de botones

II-G. Generación de trayectorias

Una vez se tienen las predicciones realizadas por el modelo, se deben obtener los puntos en los cuales se ubican los cubos de los que se leerán las imágenes. Gracias a lo realizado en el punto anterior (figura 30), cuando se realiza la aproximación a figuras geométricas simples, se deben ingresar los vértices presentes en la imagen. Por lo que los mismos son guardados en un vector con el fin de tener las coordenadas en las que se espera que el robot tenga que ubicarse para tomar los cubos.

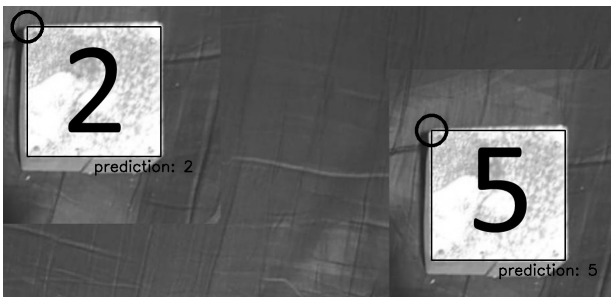

Figura 37. Vértices empleados para trayectoria

Los vértices tomados se encuentran en la esquina superior izquierda de los cubos identificados. Como puntos de inicio y final del recorrido son tomados los puntos medios en la izquierda y la derecha de la imagen, por lo que finalmente, con el fin de graficar la trayectoria que debe seguir el robot, se emplea la función “draw_line” nativa de open-cv, obteniendo:

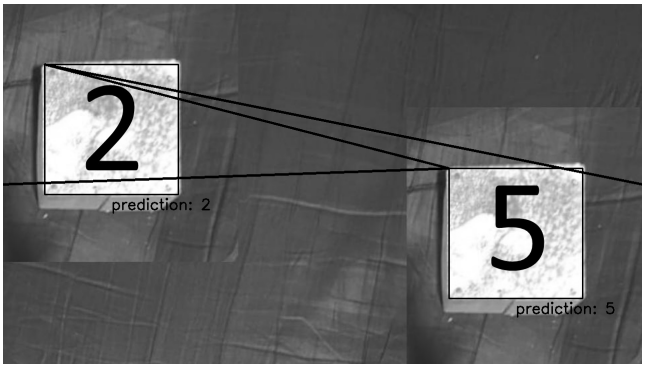

Figura 38. Trayectoria generada

Estos puntos serán posteriormente pasados por un algoritmo de optimización de trayectoria con el fin de obtener un recorrido óptimo.

III. CONCLUSIONES

Al implementar los dos modelos propuesto se pudo concluir que a pesar de que es una buena práctica saber como funciona internamente un red neuronal, es mucho más efectivo usar las herramientas que nos proporciona el avance tecnológico de hoy en día. En este caso utilizamos Keras, pero hay mas herramientas que para la construcción de redes neuronales son más útiles y precisas como lo pudimos notar. El reconocimiento de patrones funciona de manera precisa por medio de visión de máquina y es capaz de realizar las trayectorias para una aplicación de inteligencia artificial.

REFERENCIAS

[1] https://keras.io/guides/sequential_model/
[2] hhttp://yann.lecun.com/exdb/mnist/