Identificación de caracteres con redes neuronales convolucionales F3027.1 Física Computacional II

Roberto Vázquez Menchaca - A01196927, Alejandro Salinas de León - A01282503,
Carla López Zurita - A00822301, Rubén Casso de León - A01196975, and Hugo Moreno Rodríguez - A01282646

Se desarrolló un modelo de reconocimiento de caracteres alfanuméricos con redes neuronales convolucionales. El modelo utiliza la base de datos EMNIST y es capaz de reconocer los diez dígitos (0-9) y 26 letras (A-Z, a-z), en mayúsculas y minúsculas. Se analizaron distintos parámetros del modelo, tales como funciones de activación, y número y tipos de capas neuronales, evaluando su efectividad con base en pérdidas y precisión. El modelo escogido utiliza la función ReLU y cuenta con 3 capas convolucionales dando una precisión del 87.15 %.

I. INTRODUCCIÓN

Las redes neuronales son una herramienta computacional ampliamente utilizada en la actualidad, gracias a su capacidad de adaptación y resolución de problemas. Dentro del área de Machine Learning, esta yace como el corazón de los algoritmos de aprendizaje profundo. Su éxito se basa en la estructura del modelo con el cual se trabaja y la cantidad de datos disponible para el problema.

El problema EMNIST es uno de visión computacional artificial y procesamiento de imagen. La solución consiste en diseñar, entrenar y probar un algoritmo que sea capaz de colocar una etiqueta a una imagen que contiene un caracter alfanumérico. Originalmente, la base de datos MNIST, la cual toma su nombre de "Modified National Institute of Standards and Technology", contiene puramente caracteres del 0 al 9; por su parte, EMNIST es la versión extendida para incluir las 26 letras del alfabeto moderno inglés, tanto en mayúsculas como en minúsculas[1].

En este proyecto, se ha propuesto una red neuronal convolucional (CNN) secuencial original alimentada con bases de datos existentes, específicamente con la librería **emnist**. Las librerías sobre las cuales se trabajará han sido creadas y utilizadas previamente por usuarios de Python y se encuentran disponibles libremente en la red.

El objetivo de este trabajo es entonces presentar una red neuronal funcional que reconozca caracteres alfanuméricos de la base de datos mencionada, y posteriormente caracteres de nuestra propia escritura, habiendo comparado diferentes estructuras neuronales, parámetros y funciones de activación para obtener un modelo con la mayor precisión posible.

II. DESARROLLO

Para la elaboración del algoritmo se utilizó el lenguaje de programación Python, principalmente la librería de **tensorflow**, con algunas funciones extras de **sklearn**.

Como primer paso, se definió la función **load_and_preprocess_dataset** para acceder a los datos que se utilizarán y asegurarse de que estén en un formato útil y congruente con los parámetros de los modelos. Al igual que MNIST, EMNIST no es un conjunto único de datos, sino que consta de 6 posibles conjuntos con diferentes pesos por clase. En este caso, utilizaremos una de las más comunes, "EMNIST-By_Class", que contiene 62 clases no balanceadas y 814,255 imágenes en escala de grises[2]. La distribución de de clases de esta base de datos puede verse en la Fig. 1.

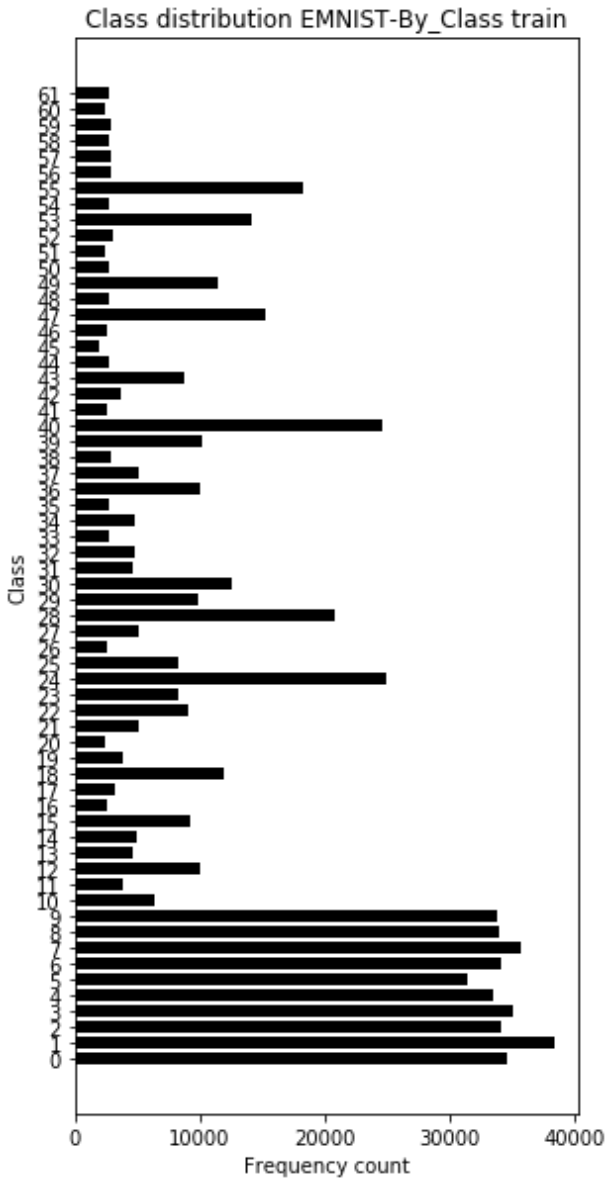

Figura 1. Distribución de clases de EMNIST-By_Class para muestras de entrenamiento. Recuperado de Simon Wenkel[2].

Se asignan después las muestras de prueba y entrenamiento correspondientes y luego se preprocesan para que se ajusten a los requisitos de una CNN, modificando sus dimensiones y normalizándolas. El tamaño de las imágenes, incluyendo las que se añadieron de nuestra propia escritura, es de 28 por 28 píxeles.

A. Capas del modelo

Una vez cargados correctamente los datos, se puede proceder con la construcción del modelo. Esta se realizó de manera secuencial; es decir, compuesta de tensores, de ahí que la librería utilizada sea **tensorflow**.

En el extracto de código sig. se muestra la implementación de todas las capas neuronales que resultan en el modelo con mayor precisión. Cada línea del código es una capa distinta, las cuales serán explicadas a continuación.

```
model = Sequential([
    Conv2D(filters, kernel_size,
        input_shape=input_shape),
    BatchNormalization(),
    layers.Activation(activations.
        relu),
    MaxPool2D(pool_size=(2, 2),
        strides=None, padding='valid
        ', data_format=None),
    Conv2D(64, kernel_size,
        activation = act_function),
    Conv2D(64, kernel_size,
        activation = act_function),
    MaxPool2D(pool_size=(2, 2),
        strides=None, padding='valid
        ', data_format=None),
    Flatten(data_format=None),
    Dense(units1),
    BatchNormalization(),
    layers.Activation(activations.
        relu),
    Dropout(rate, noise_shape=None,
         seed=None),
    Dense(units2, activation=
        act_function2)
         ])
```

Principalmente, la inclusión de la capa **Conv2D** es lo que hace que esta red neuronal se denomine como convolucional. La justificación de usar este tipo de capa, a diferencia de una "fully connected", es debido al número de elementos por imagen (28x28), en el que cada neurona es la encargada de procesar cada subdivisión. Redes únicamente de propagación directa resultarían en un modelo más complejo computacionalmente y poco eficiente[3]. Así, la ventaja de emplear redes convolucionales es que, en conjunto con distintas capas, logran reducir el tamaño de las imágenes para que sean más fácilmente manejables. De ahí que estas se suelan utilizar en problemas de clasificación de imágenes, como en este caso[4].

Una característica fundamental en las redes neuronales en general es la capa de normalización por paquetes, o en inglés, "batch normalization". La función **BatchNormalization** busca estandarizar las entradas a una capa para cada paquete o batch. El objetivo es estabilizar el proceso de aprendizaje y reducir el número de épocas en el entrenamiento de la red. Estas capas son agregadas en lugares que han sido corroborados como más óptimos por la literatura y estudios académicos[5].

Por su parte, la capa **MaxPool2D** aplica un "filtro" que es en realidad una convolución entre la imagen original y una función, **kernel_size**, que extrae en cuadros de dimensión especificada por nosotros, en este caso 3x3, el valor máximo de esa sección de la imagen. Esto es con el propósito de obtener el valor más significativo y reducir al mismo tiempo la imagen. Se ha decidido en esta ocasión no usar un borde sobre la imagen, denominado "padding", ya que al ser imágenes de números escritos a mano, a diferencia de lo que podría ser una imagen fotográfica cualquiera, naturalmente incluyen un borde o espacio en blanco alrededor de la sección de interés. En pocas palabras, consideramos que agregar un padding no ayudaría a mejorar el rendimiento del modelo.

Otra función que ha sido agregada es una que logra modificar o reestablecer las dimensiones de los datos a un tamaño indicado para ser procesado por la red. Esta es **Flatten**.

Por otro lado, **Dense** crea una capa conectada de manera directa o "densa" en el lenguaje de redes neuronales. Esto quiere decir que cada neurona está conectada con todas las otras neuronas linealmente. Este proceso es muy costoso computacionalmente al tener muchas neuronas; sin embargo, al haber pasado ya por las capas previamente descritas, se sabe que el número de neuronas se ha reducido significativamente.

La penúltima capa **Dropout** consiste en eliminar neuronas de manera aleatoria con una probabilidad especificada por el usuario. Su función es evitar que el modelo consiga un sobreajuste a los datos de entrenamiento y sea capaz de predecir correctamente elementos de entrada que sean nuevos para el programa. Normalmente se recomienda colocar esta capa en esta posición para mejores resultados.

Por último, en la capa final, se ha escogido la conexión **Dense** con una función de activación denominada SoftMax. A grandes rasgos, esta garantiza la distribución categórica de la probabilidad, la cual es deseable dada la naturaleza del problema. Esto quiere decir que cada letra será asignada solamente una etiqueta, mutuamente excluyente. Una más detallada explicación y comparación de las funciones de activación utilizadas en el modelo, se encuentra en la siguiente sección.

Como nota adicional, cabe recalcar que durante la compilación del modelo se utilizó el optimizador *adam*, con una métrica de *accuracy* y usando la función de pérdida *sparse_categorical_crossentropy*, la cual evita usar un esquema de codificación "one-hot encoded" para ahorro de memoria y tiempo computacional debido a la gran cantidad de clases. El optimizador seleccionado es la opción que mejores resultados brinda para este problema en particular[6].

B. Funciones de activación

La decisión de utilizar una red neuronal convolucional conlleva sus propias limitaciones, ya que la función de activación que garantiza su funcionamiento correcto se reduce a las ReLU, por sus siglas en inglés "Rectified Linear Unit"(incluida en la línea 5 del código antepasado), o sus derivadas como Leaky ReLU.

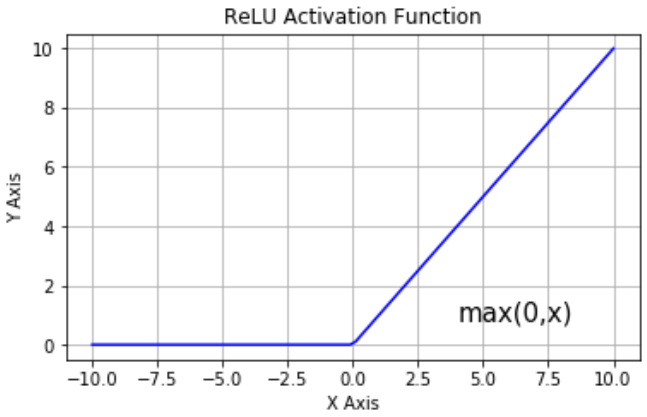

Figura 2. Función de activación ReLU. Recuperado de Aditya Sharma[7].

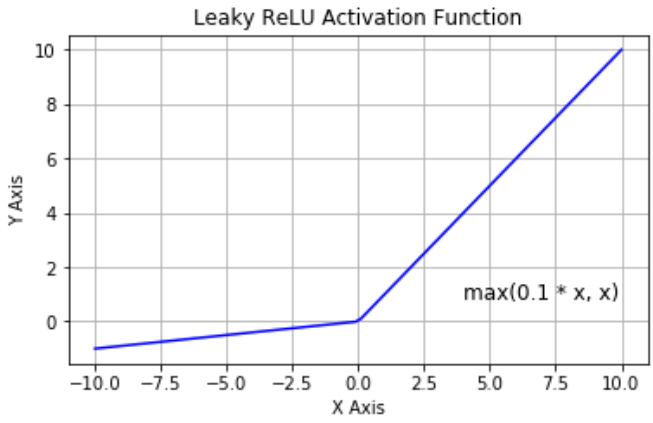

Figura 3. Función de activación LeakyReLU. Recuperado de i2 Tutorials[8].

La diferencia entre estas dos funciones de activación es sencilla, mientras que ReLU se activa de manera lineal a partir de 0, LeakyReLU le da valores negativos pequeños al primer intervalo de la función. Esta última función apareció debido a que las funciones ReLU pueden ser frágiles durante el entrenamiento de la red neuronal, por un problema conocido como *dying ReLU*. El gradiente a través de una neurona ReLU puede provocar que los pesos se actualicen de tal manera que la neurona no vuelva a activarse en ningún punto, truncando el aprendizaje de la red[8]. De todas maneras, se probaron ambas funciones, en caso de que este modelo pudiese ser afectado por *dying ReLU*.

Adicionalmente, a lo largo del curso de Física Computacional II, se han estudiado otros ejemplos de funciones de activación, como la sigmoide, o tangente hiperbólica. Para el caso a resolver en el presente proyecto, se llegó a la conclusión de que esta función no es recomendada, ya que no cumple con los requisitos necesarios para el funcionamiento de feed-forward y backpropagation. Aunque las funciones son diferenciales, corren el riesgo de caer en el problema de la desaparición de gradiente. De igual manera, se prefieren las funciones de activación no lineales, como es ReLU, ya que permiten que los nodos aprendan estructuras más complejas en los datos. Se sabe que el resultado de una función lineal está dado como una combinación lineal con el input. Es decir, no importa la cantidad de capas que se utilicen, al final se podrá reducir a una sola. Por lo tanto, no se puede mejorar el desempeño utilizando más capas. Asimismo, la mayoría de los problemas en la vida real pueden ser modelados con funciones de comportamientos no lineales, por lo que es preferible e incluso a veces necesario usar funciones no lineales.

Una comparación cuantitativa del rendimiento de las funciones de activación se encuentra en la siguiente sección.

C. Pre-evaluación de modelos

Con el objetivo de evaluar la mejor alternativa entre los diferentes modelos que podemos utilizar, se ha diseñado la función **pre_evaluation**, mostrada a continuación.

```
1  def   pre_evaluation(data_X,
     data_Y, k_folds = 6):
2
3      scores, histories = list(),
           list() #Initializing result
           lists
4      # Call cross validation
           function, shuffling outside
           the loop to maintain
           consistency when testing k-
           fold and seeding
5      kfold = KFold(k_folds, shuffle=
           True, random_state=1)
6      for train_ix, test_ix in kfold.
           split(data_X):
7          model = define_model()
8          train_X, train_Y, test_X,
               test_Y = data_X[train_ix
               ], data_Y[train_ix],
               data_X[test_ix], data_Y[
               test_ix]
9          history = model.fit(train_X
               , train_Y, epochs=10,
               batch_size=36,
               validation_data=(test_X,
                test_Y), verbose=0)
10         loss, accuracy = model.
               evaluate(test_X, test_Y,
                verbose=0)
```

```
11          print('> %.3f' % (accuracy
                * 100.0))
12          scores.append(accuracy)
13          histories.append(history)
14      return scores, histories
```

La función se basa en usar el método de k_folds, para partir los datos originales en un número dado k de subdivisiones y así preevaluar y comparar entre sí las precisiones de diferentes modificaciones al modelo de manera más eficiente. En este caso se escogieron 6 divisiones. En la siguiente sección se reportarán los promedios de las seis medidas de precisión en porcentaje.

III. RESULTADOS

A. Comparación de funciones de activación

Cada capa del modelo secuencial utiliza una función de activación, lo cual se presta a muchas alternativas que pueden llevar a distintas precisiones y eficiencias. Para escoger la mejor opción, se decidió basarse únicamente en el parámetro de precisión. Una vez descartando algunas opciones (léase II B), nos quedamos con dos funciones de activación distintas: ReLU y LeakyReLU, a las cuales se le aplicarán diferentes modificaciones. A continuación se presentan los resultados de cada opción, y posteriormente se describirán a detalle.

Tabla I. Precisiones de distintas funciones de activación

Precisión (%)			
ReLU	LeakyReLU	ReLU (batch)	**ReLU (batch+3 CNN)**
85.494	85.1965	85.8945	**87.1535**

A partir de estos resultados, se observa que la función ReLU obtuvo un porcentaje de precisión ligeramente mayor al de LeakyReLU, lo cual parece indicar que el problema de *dying ReLU* no se presentó para este conjunto de datos. Sin embargo, la modificación que arrojó mayor precisión fue utilizar la función ReLU con una capa de **BatchNormalization** y 3 capas convolucionales. Esto concuerda con lo esperado, ya que el ranking de mejores modelos de clasificación de imágenes tiene muchos ejemplos de alta precisión mediante el uso de múltiples capas convolucionales.

Por último, se decidió incluir una gráfica que evalúe el aprendizaje del modelo. Lo importante es ver una convergencia en las curvas de aprendizaje y que las mismas decaigan conforme se avanza en las épocas.

Lo que nos dice la Fig. 4 es que a mayor suministro de datos y la combinación de feed-forward con backpropagation, más se minimiza la función de costo. Cada línea representa un k_fold distinto para los datos de entrenamiento (verdes) y de prueba (rojas) a través de las épocas. A raíz de esto, una posible mejora del modelo sería agregar una mayor cantidad de épocas, para observar qué tanto más las curvas se aproximan a un 0 en pérdidas, que sería el caso ideal. Además, es importante ver que

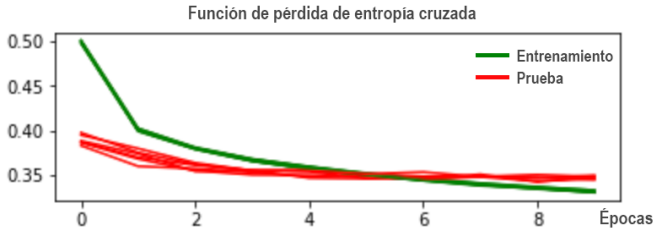

Figura 4. Curva de aprendizaje del modelo seleccionado.

no hay overfitting ni underfitting, ya que una gráfica con underfitting mostraría una línea recta, y una con overfitting tendría mucho ruido sin convergencia[9]. De ahora en adelante, cuando nos refiramos al modelo, estaremos hablando de este último que obtuvo la mayor precisión.

B. Matriz de confusión

Una vez escogido el mejor modelo con base en su precisión, lo utilizaremos sobre el conjunto de datos de prueba de la base de datos EMNIST, y observaremos los aciertos y fallas en una matriz de confusión. Debido a su tamaño (62x62), la matriz se encuentra en la Fig. 7, que ha sido incluida en el apéndice V A para una mejor visualización.

En la gráfica, observamos la cuenta de aciertos con un gradiente hacia el amarillo. El eje vertical corresponde a la etiqueta del input (el número o letra real) y el eje horizontal representa la predicción del modelo (output).

Lo primero a observar es que los caracteres con mayor coincidencia corresponden a los primeros 10 elementos de la diagonal, que son los numéricos. Si nos referimos a la Fig. 1, se observa que son precisamente los números los que aparecen con mayor frecuencia que las letras en la base de datos seleccionada, por lo que la matriz resultante tienen sentido. Es decir, nuestro modelo acertó muy frecuentemente en el reconocimiento de números, probablemente a causa de que fue entrenado con mayor cantidad de números que letras.

Posteriormente, observamos que el resto de la diagonal refleja verdaderos positivos, es decir, predicciones correctas del modelo, lo cual de entrada nos habla de un buen comportamiento. Sin embargo, para balancear la diagonal completa, es posible utilizar otra base de datos, como la EMNIST-Balanced, que contiene mismo número de muestras numéricas y alfabéticas.

Fuera de la diagonal, podemos ver en su mayoría valores cercanos a 0 (azules), como se esperaba. Aunque hay algunas cuentas de aciertos en otras regiones (falsos positivos), estos se deben a errores comunes y esperados que se discutirán más a detalle en la siguiente subsección.

C. Predicción de caracteres escritos a mano

La prueba final del modelo escogido consiste en utilizar de input imágenes de caracteres escritos a mano por nosotros. Para esto, cargamos al código imágenes de 28x28 píxeles de todos los caracteres alfanúmericos, las cuales se encuentran en la sig. liga: https://drive.google.com/drive/folders/1CeofxdKezJYbWrWLplv3O3cdg5zf1qjh?usp=sharing.

En general, todos los caracteres fueron predichos exitosamente, salvo algunos ejemplos particulares. En la Fig. 5, observamos el primer tipo de error, que son letras minúsculas que son erróneamente identificadas como su contraparte mayúsculas. Esto es de esperarse, ya que sólo sucede con letras muy similares, como la "s" y la "w". El segundo tipo de error en la Fig. 6 es de distinta naturaleza, en donde una "O" mayúscula se identificó como un dígito "0", y una "l" minúscula como un dígito "1". Este tipo de errores es común en soluciones de EMNIST, posiblemente resueltos por el uso de otras bases de datos más balanceadas.

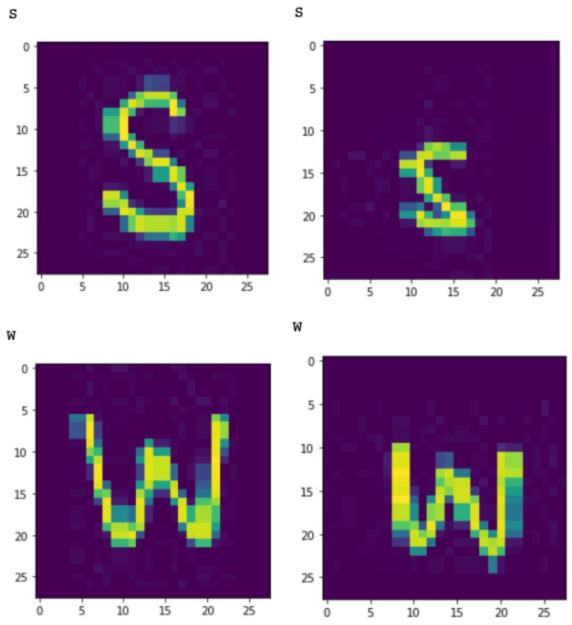

Figura 5. Falsos positivos de distinción entre mayúsculas y minúsculas.

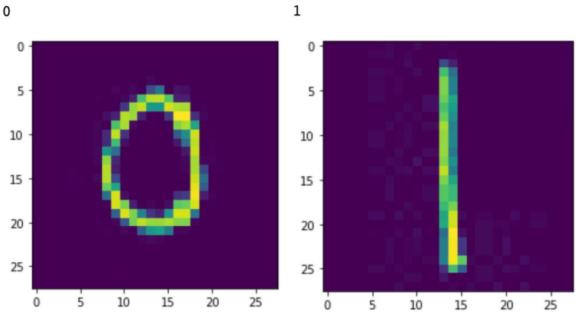

Figura 6. Falsos positivos de distinción entre letras y dígitos.

IV. CONCLUSIÓN

El modelo final, aquel con mejor resultado en precisión, fue aquel con la mayor cantidad de capas convolucionales. Esto se relaciona con el funcionamiento de las mismas, ya que en cada capa se realiza un análisis de la relevancia y la correlación entre las secciones de la imagen más significativas y la relevancia para asignar una etiqueta. Sin embargo, debe haber un límite en el número de capas, que si es sobrepasado, el resultado podría reducir tanto la imagen a tal grado que ya no se distinga el contenido original. Otra consecuencia de un número excedente de capas puede llevar a un modelo sobreajustado a los datos de entrenamiento. Este límite podría ser obtenido como trabajo futuro de manera experimental, analizando la variación entre la precisión y el número de capas como hiperparámetro.

Por último, algunas de las posibles mejoras que se podrían hacer al modelo es variar algunas características de la alimentación de datos, construcción y evaluación del modelo. Uno de estos podría ser utilizar otro tipo de base de datos de EMNIST, como una completamente balanceada o "EMNIST-By_Merge", que contiene solo 47 clases, pues ha eliminado algunas letras parecidas. Se podrían incluir también en las mismas imágenes de entrenamiento algunas de creación propia. Asimismo, se podría evaluar el número de épocas adecuado para obtener una eficacia óptima tomando en cuenta el costo computacional.

REFERENCIAS

[1] G. Cohen, S. Afshar, J. Tapson, and A. Van Schaik, in *2017 International Joint Conference on Neural Networks (IJCNN)* (IEEE, 2017) pp. 2921–2926.
[2] S. Wenkel, "Exploring emnist - another mnist-like dataset," https://www.simonwenkel.com/2019/07/16/exploring-EMNIST.html#emnist-by-classy, accessed: 2021-06-13.
[3] G. Bebis and M. Georgiopoulos, IEEE Potentials **13**, 27 (1994).
[4] T. Guo, J. Dong, H. Li, and Y. Gao, in *2017 IEEE 2nd International Conference on Big Data Analysis (ICBDA)* (IEEE, 2017) pp. 721–724.
[5] S. Ioffe and C. Szegedy, 32nd International Conference on Machine Learning, ICML 2015 **1**, 448 (2015), arXiv:1502.03167.
[6] D. Giordano, "7 tips to choose the best optimizer," https://towardsdatascience.com/7-tips-to-choose-the-best-optimizer-47bb9c1219e, accessed: 2021-06-10.
[7] A. Sharma, "Relu activation function," https://learnopencv.com/understanding-activation-functions-in-deep-learning/relu-activation-function-2/, accessed: 2021-06-13.
[8] i2 Tutorials, "Threshold and leaky relu activation functions," https://www.i2tutorials.com/explain-step-threshold-and-leaky-relu-activation-functions/, accessed: 2021-06-13.
[9] J. Brownlee, "How to use learning curves to diagnose machine learning model performance," https://machinelearningmastery.com/learning-curves-for-diagnosing-machine-learning-model-performance/, accessed: 2021-06-13.
[10] G. Cohen, S. Afshar, J. Tapson, and A. van Schaik, CoRR **abs/1702.05373** (2017), arXiv:1702.05373.

V. APÉNDICES

A. Matriz de confusión

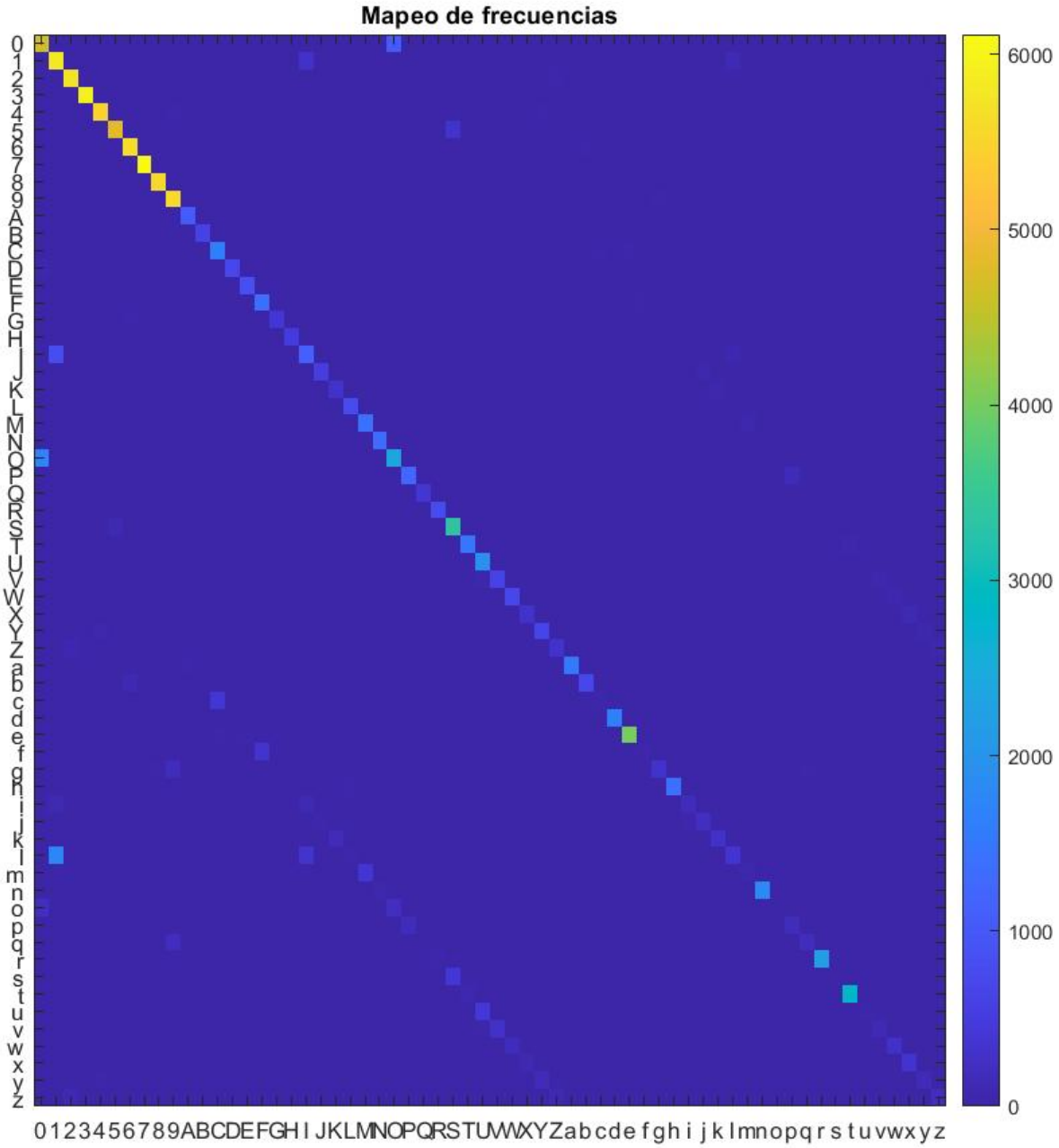

Figura 7. Matriz de confusión, resultado de secuencia de aciertos del modelo seleccionado.

B. Código completo

```
1
2  from emnist import list_datasets, extract_training_samples,
       extract_test_samples
3  import h5py
4  import tensorflow as tf
5  from tensorflow.keras import layers
6  from tensorflow.keras.models import Sequential
7  from tensorflow.keras.layers import Dense, Conv2D, Dropout, Flatten,
       MaxPool2D, LeakyReLU, BatchNormalization
8  from tensorflow.keras import activations
9  import matplotlib.pyplot as plt
10 from pandas import DataFrame
11 import seaborn as sn
12 from sklearn.metrics import confusion_matrix
13 import numpy as np
14 from PIL import Image
15 import os
16 from sklearn.model_selection import KFold
17 from numpy import mean, std
18 from tensorflow.keras.preprocessing.image import load_img, img_to_array
19
20 ###
21
22 def load_and_preprocess_dataset():
23     '''
24     This function loads the EMNIST byclass dataset that contains 62
           unbalanced classes and 814,255 characters.
25     Assigns the corresponding training and test samples and then preprocess
           them to fit the requisites to pass them
26     into a CNN by reshaping and normalizing them.
27     '''
28     #Extract training and test samples from EMNIST byclass dataset and
           assigning to corresponding variables
29     x_train, y_train = extract_training_samples('byclass')
30     x_test, y_test = extract_test_samples('byclass')
31     # Reshape the array to 4 dimensions adding a single color channel
           parameter for it to work with the Keras API.
32     x_train = x_train.reshape(x_train.shape[0], 28, 28, 1) #x_train.shape[0]
           =
33     x_test = x_test.reshape(x_test.shape[0], 28, 28, 1)
34     # Convert values to float for division to normalize pixels
35     x_train = x_train.astype('float32')
36     x_test = x_test.astype('float32')
37     # Normalizing the RGB codes by dividing by the max RGB value to rescale
           our sampling sets between 0 and 1
38     x_train /= 255
39     x_test /= 255
40     return x_train, x_test, y_train, y_test
41
42 ###
43
44 def define_model():
45     '''
46     Construct a CNN model with hyperparameters flexible to be tuned and
           modified.
47     '''
48     filters = 32 #Number of feature maps
```

```
49      kernel_size = (3,3) #Size of the convolution filter
50      input_shape = (28, 28, 1) # Specific and requisite for the CNN
51      units1 = 128 #Outputs of that layer
52      units2 = 62 #Number of labels, since it is the output layer.
53      act_function = tf.nn.relu #Relu necessary for convolutional layers,
            implemented for baseline model
54      #act_function = LeakyReLU(alpha=0.2) # Non-zero gradient ReLU, candidate
            for improvement
55      #act_function1 = tf.nn.relu #Maintain consistency in activation functions
             across hidden layers
56      #act_function1 = LeakyReLU(alpha=0.2)
57      act_function2 = tf.nn.softmax #Scaling 'arbitrary' numbers to mutually
            exclussive probabilities since it is a multiclass problem
58      rate = 0.2 #20% of neurons dropped to avoid overfitting
59      model = Sequential(
60          [
61              Conv2D(filters, kernel_size, input_shape=input_shape),
62              BatchNormalization(),
63              layers.Activation(activations.relu),
64              MaxPool2D(pool_size=(2, 2), strides=None, padding='valid',
                    data_format=None),
65              Conv2D(64, kernel_size, activation = act_function),
66              Conv2D(64, kernel_size, activation = act_function),
67              MaxPool2D(pool_size=(2, 2), strides=None, padding='valid',
                    data_format=None),
68              Flatten(data_format=None),
69              Dense(units1),
70              BatchNormalization(),
71              layers.Activation(activations.relu),
72              Dropout(rate, noise_shape=None, seed=None),
73              Dense(units2, activation=act_function2)
74          ])
75      optimizer1 = 'adam' #Hyperparameter that could change to non-adaptive
            optimizers such as SGD
76      #Specific loss function used to reduce memory usage. The other option was
             categorical crossentropy, but it required for us to one-hot encoded
77      #our target arrays, having many zeros in them since this problem has a
            big amount of classes.
78      loss1 = 'sparse_categorical_crossentropy'
79      metrics1 = ['accuracy']
80      model.compile(optimizer = optimizer1,
81                    loss = loss1,
82                    metrics = ['accuracy'])
83      return model
84
85  ###
86
87  def pre_evaluation(data_X, data_Y, k_folds = 6):
88      '''
89      Pre-evaluating the model using 6-folds cross-validation (to divide
            exactly the 697,932 data)
90      '''
91      scores, histories = list(), list() #Initializing result lists
92      # Call cross validation function, shuffling outside the loop to maintain
            consistency when testing k-fold and seeding
93      kfold = KFold(k_folds, shuffle=True, random_state=1)
94      # Initialize loop by alternating and evaluating on k-folds
95      for train_ix, test_ix in kfold.split(data_X):
96          # Call model
97          model = define_model()
```

```
98          # Slicing the data according to how the split of the k-fold was made
99          train_X, train_Y, test_X, test_Y = data_X[train_ix], data_Y[train_ix
                ], data_X[test_ix], data_Y[test_ix]
100         # Fit model (will be discarded after this preevaluation, since the
                real fitting will be afterwards)
101         #Hyperparameters of 10 epochs subject to change, batch size of 36 so
                each loop of each epoch has the same number of samples
102         history = model.fit(train_X, train_Y, epochs=10, batch_size=36,
                validation_data=(test_X, test_Y), verbose=0)
103         # Obtain loss and accuracy of the model
104         loss, accuracy = model.evaluate(test_X, test_Y, verbose=0)
105         print('> %.3f' % (accuracy * 100.0))
106         # Store information on scores and histories arrays
107         scores.append(accuracy)
108         histories.append(history)
109     return scores, histories
110
111
112 ###
113
114 def diagnosis(histories):
115     '''
116     Plot learning curves to evaluate how the model performed per fold of the
            6-folds cross-validation
117     '''
118     for i in range(len(histories)):
119         # Evaluate graphically the loss
120         plt.subplot(2, 1, 1)
121         plt.title('Sparse Cross Entropy Loss')
122         plt.plot(histories[i].history['loss'], color='green', label='Train')
123         plt.plot(histories[i].history['val_loss'], color='red', label='Test')
124         # Evaluate graphically the accuracy
125 #         plt.subplot(2, 1, 2)
126 #         plt.title('Multi-Classification Accuracy')
127 #         plt.plot(histories[i].history['accuracy'], color='green', label='
        Train')
128 #         plt.plot(histories[i].history['val_accuracy'], color='red', label='
        Test')
129     plt.show()
130
131 ###
132
133 def performance(scores):
134     '''
135     Quantitatively evaluate how the model performed by using mean, standard
            deviation and k-folds results.
136     Plotting the accuracies to graphically evaluate the model's performance.
137     '''
138     # Quantitatively determine how well the model performed considering mean
            of accuracies, standard deviation and k-folds
139     print('Accuracy: mean=%.3f std=%.3f, n=%d' % (mean(scores)*100, std(
            scores)*100, len(scores)))
140     # Graphically evaluate how well the model performed by using a boxplot to
             catch the max, min and average accuracies
141     plt.boxplot(scores)
142     plt.show()
143
144 ###
145
146 def run_pre_test():
```

```
    '''
    Run the whole pre-test by calling this function.
    '''
    # Load and preprocess the dataset
    train_X, test_X, train_Y, test_Y = load_and_preprocess_dataset()
    # Evaluate model
    scores, histories = pre_evaluation(train_X, train_Y)
    # Plot performance of model learning
    diagnosis(histories)
    # Quantify how the model performed
    performance(scores)

###

def save_model():
    '''
    Run the test with the hold out test set by calling this function.
    '''
    # Load and preprocess the dataset
    train_X, test_X, train_Y, test_Y = load_and_preprocess_dataset()
    # Define model
    model = define_model()
    # Fit model
    model.fit(train_X, train_Y, epochs=10, batch_size=36, verbose=0)
    # save model
    model.save('final_EMNISTmodel.h5')

###

save_model()

###

def confusion(y_test, y_pred):
    matrix = metrics.confusion_matrix(y_test.argmax(axis=1), y_pred.argmax(
        axis=1))
    return matrix


###

def test():
    # Load and preprocess the dataset
    train_X, test_X, train_Y, test_Y = load_and_preprocess_dataset()
    # Load model
    model = tf.keras.models.load_model('final_EMNISTmodel.h5')
    # Evaluate model on test dataset
    loss, acc = model.evaluate(test_X, test_Y, verbose=0)
    y_pred = model.predict(test_X)
    confm = confusion_matrix(test_Y, y_pred.argmax(axis=1))
    df_cm = DataFrame(confm)
    #df_cm.to_csv(r'D:\Downloads\export_dataframe.csv', index = False, header
        =True)
    fig, ax = plt.subplots(figsize=(75,75))
    ax = sn.heatmap(df_cm, cmap='Oranges', annot=True)
    print('> %.3f' % (acc * 100.0))


###

```

```
205
206 def load_image(filename):
207     # Load the image
208     img = load_img(filename, color_mode = "grayscale", target_size=(28, 28))
209     # Convert to array
210     img = img_to_array(img)
211     # Reshape with 1 channel
212     img = img.reshape(1, 28, 28, 1)
213     # Preprocess array by converting type to float and normalizing
214     img = img.astype('float32')
215     img = 255-img
216     img = img / 255
217     return img
218
219 ###
220
221 def run_example():
222     # Load the image
223     img = load_image('char_a_mayus.png')
224     # Load model
225     model = tf.keras.models.load_model('final_EMNISTmodel.h5')
226     # Predict the class
227     prediction = model.predict_classes(img)
228     img_fig = np.resize(img, (28,28,1))
229     plt.imshow(img_fig)
230     d_num_to_real = {0:'0', 1:'1', 2: '2', 3: '3', 4:'4', 5:'5', 6:'6', 7:'7'
        , 8: '8', 9: '9', 10: 'A', 11: 'B', 12: 'C',
231                      13: 'D', 14: 'E', 15: 'F', 16: 'G', 17: 'H', 18: 'I', 19:
                             'J', 20: 'K', 21: 'L', 22: 'M', 23: 'N',
232                      24: 'O', 25: 'P', 26: 'Q', 27: 'R', 28: 'S', 29: 'T', 30:
                             'U', 31: 'V', 32: 'W', 33: 'X', 34: 'Y',
233                      35: 'Z', 36: 'a', 37: 'b', 38: 'c', 39: 'd', 40: 'e', 41:
                             'f', 42: 'g', 43: 'h', 44: 'i', 45: 'j',
234                      46: 'k', 47: 'l', 48: 'm', 49: 'n', 50: 'o', 51: 'p', 52:
                             'q', 53: 'r', 54: 's', 55: 't', 56: 'u',
235                      57: 'v', 58: 'w', 59: 'x', 60: 'y', 61: 'z'
236                      }
237     print(d_num_to_real[prediction[0]])
238     #print(prediction[0])
239
240 ###
241
242 run_example()
```