Instituto Tecnológico y de Estudios Superiores de Monterrey Campus Querétaro

CONCENTRACIÓN DE CIENCIA DE DATOS E INTELIGENCIA ARTIFICIAL AVANZADA

DEEP LEARNING IMPLEMENTATION

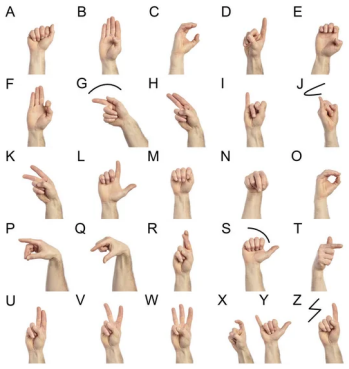
(Sign Language Alphabet)

Carlos Eduardo Ortega Clement A01707480

1. INTRODUCCIÓN

Este problema busca poder interpretar a texto el alfabeto del lenguaje de señas, para poder realizar esto se encontró un dataset con varios ejemplos de cada una de las palabras del abecedario en español. Se creará un código final que se pueda ejecutar desde la consola que sea capaz de buscar alguna imagen en internet de alguna mano interpretando una letra del abecedario y realizar la predicción del modelo final de dicha imagen buscada.

Aquí se encuentra el abecedario que se espera predecir:



2. DATASET

Para el manejo de los datos utilicé el ImageDataGenerator, esto para poder hacer más eficiente la manipulación de los datos y sus respectivas transformaciones. En este caso creé dos ImageDataGenerator, uno para train data y el otro para test data.

En ambos casos solamente apliqué una normalización de las imágenes. En train_datagen asigné un validation_split de 0.2 para tener una sección de validación del 20% del total de train data. Decidí no hacer data augmentation ya que en este caso específico, el realizar variaciones a las imágenes como rotaciones o volteos puede afectar más al resultado del modelo ya que las manos deben estar en una posición específica para entender las palabras.

La función flow_from_directory me permite obtener los datos de mis carpetas de una manera más sencilla ya que tengo una carpeta para cada una de las clases. De igual forma aquí se determina el tamaño de los batches, esto para no saturar la memoria. También se ajusta el tamaño de las imágenes a 150 x 150 ya que no es conveniente trabajar con imágenes grandes debido a que vuelve los arrays más pesados.

Más adelante mencionaré el proceso que llevé a cabo para llegar a mi modelo final.

```
train_datagen = ImageDataGenerator(
                    rescale = 1./255,
                    horizontal_flip = False,
                    validation_split = 0.2

)

train_generator = train_datagen.flow_from_directory(
                    train_dir,
                    target_size = (150, 150),
                    batch_size = 20,
                    class_mode ='categorical')
```

```python
val_generator = train_datagen.flow_from_directory(
    train_dir,
    target_size=(150, 150),
    batch_size=20,
    class_mode='categorical',
    subset='validation'  # Usar parte de los datos para validación
)

# Generador de flujo de datos para el conjunto de prueba
test_datagen = ImageDataGenerator(rescale=1./255)
test_generator = test_datagen.flow_from_directory(
    test_dir,
    target_size=(150, 150),
    batch_size=20,
    class_mode='categorical'
)
```

3. PRIMER MODELO

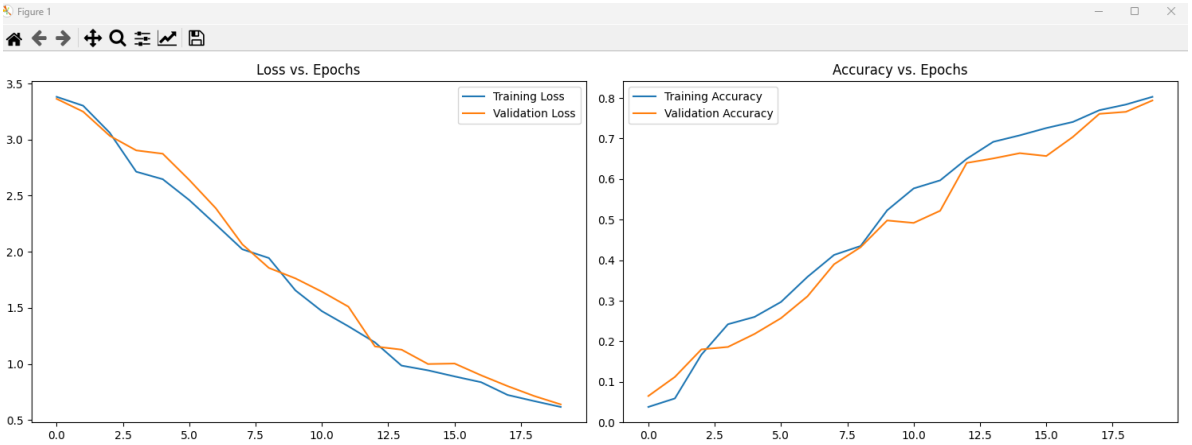
Para mi primera versión del modelo realicé una red neuronal convolutiva no tan compleja, mantuve algunas capas conv2D con el mismo padding y un stride de 1, a todas las capas les mantuve una activación relu. Utilicé como optimizador ”Adam” y para la pérdida “categorical_crossentropy”, esto porque es el más común en el entrenamiento de modelos multiclase.

```python
# Construccion del modelo
def get_model(input_shape):

    model  =  Sequential([
                    Conv2D(32, kernel_size=3, padding='same', activation="relu",
                           input_shape=input_shape),
                    MaxPooling2D((2,2)),
                    Conv2D(32, kernel_size=3, padding='same', strides = 1, activation="relu"),
                    Conv2D(16, kernel_size=3, padding='same', strides = 1, activation="relu"),
                    Conv2D(16, kernel_size=3, padding='same', strides = 1, activation="relu"),
                    MaxPooling2D((2,2)),
                    Flatten(),
                    Dense(29, activation='softmax')
    ])
    return model
```

En cuanto a los hiper parámetros para el entrenamiento, mantuve el learning rate por default, entrené 20 epochs a un paso de 50 batches tanto en train como validation. Después de estos ajustes, puse a entrenar el modelo y guardé los pesos como “pesos_del_modelo1.h5”. Este primer código se puede observar en el github como “alphabet.py”.

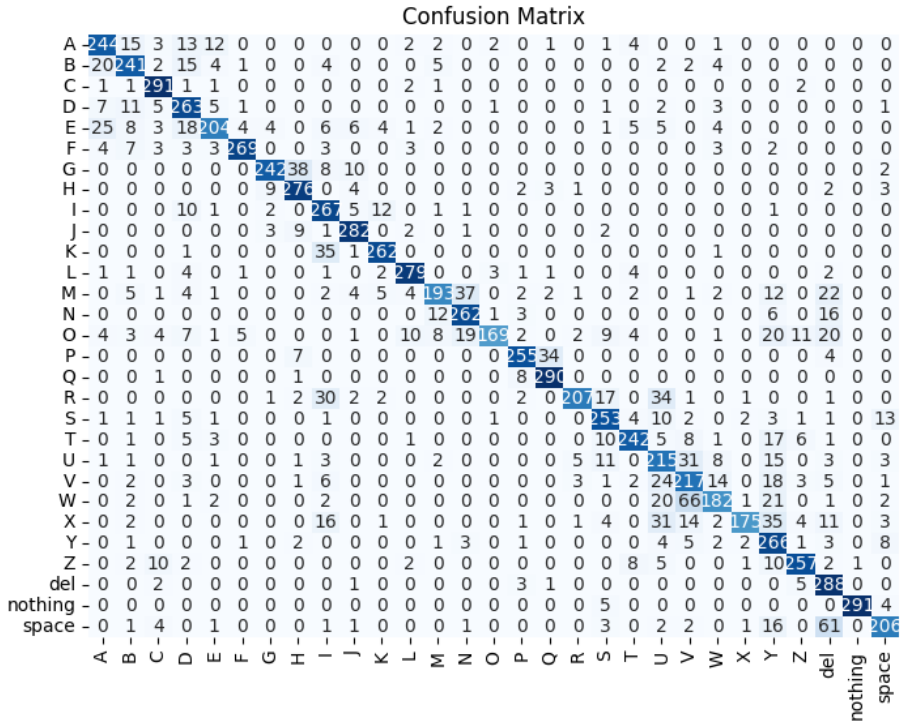
```python
def train_model(model,train_generator,val_generator):

        # Entrenar el modelo usando el generador de datos
    history = model.fit(
        train_generator,
        steps_per_epoch =50,
        epochs=20,  # Número de épocas de entrenamiento
        validation_data=val_generator,
        validation_steps = 50
    )

    return history
```

3.1 RESULTADOS DEL PRIMER MODELO



Analizando las gráficas de pérdida y de accuracy se puede notar que el modelo sí está mejorando a lo largo de las epochs. Sin embargo, el accuracy en validation no está siendo el más adecuado ya que a lo largo de todas las epochs se mantiene desfasado del train accuracy, esto puede indicar un poco de overfitting. Además, el accuracy en general no pasa del 85%, al igual que la pérdida no baja de 0.5.

Se puede notar que no existe un cambio brusco en cuanto a la pérdida a lo largo de las epochs, esto puede decirnos que el learning rate y el optimizador no requieren necesariamente de un cambio. Por último, se puede ver que no existe una curvatura normal en cuanto a las dos gráficas, esto puede ser un indicador de que faltan más epochs o más datos de entrenamiento en cada epoch.



Los resultados de la matriz de confusión indican que casi todas las letras están prediciendo casi la misma cantidad de imágenes de forma correcta, con más de 200 predicciones

correctas. Existen letras que tienen una mayor cantidad de predicciones erróneas como la A, U, V, Y, del, D y la E. Se buscará mejorar estos resultados en el siguiente modelo.

	precision	recall	f1-score	support
A	0.79	0.81	0.80	300
B	0.79	0.80	0.80	300
C	0.88	0.97	0.92	300
D	0.74	0.88	0.80	300
E	0.85	0.68	0.76	300
F	0.95	0.90	0.92	300
G	0.93	0.81	0.86	300
H	0.82	0.92	0.87	300
I	0.69	0.89	0.78	300
J	0.89	0.94	0.91	300
K	0.91	0.87	0.89	300
L	0.91	0.93	0.92	300
M	0.85	0.64	0.73	300
N	0.81	0.87	0.84	300
O	0.95	0.56	0.71	300
P	0.91	0.85	0.88	300
Q	0.87	0.97	0.92	300
R	0.94	0.69	0.80	300
S	0.80	0.84	0.82	300
T	0.88	0.81	0.84	300
U	0.60	0.72	0.65	300
V	0.62	0.72	0.67	300
W	0.80	0.61	0.69	300
X	0.96	0.58	0.72	300
Y	0.60	0.89	0.72	300
Z	0.89	0.86	0.87	300
del	0.65	0.96	0.78	300
nothing	1.00	0.97	0.98	300
space	0.84	0.69	0.75	300
accuracy			0.81	8700
macro avg	0.83	0.81	0.81	8700
weighted avg	0.83	0.81	0.81	8700

Finalmente se puede observar la precisión que obtuvo cada clase. En general, el modelo no está prediciendo de manera correcta, esto debido a que existen clases donde el accuracy es muy alto (como en la clase nothing) pero en un considerado número de clases es muy bajo (como en la clase Y).

Con estos resultados se puede concluir que el modelo obtenido no es óptimo para dejarlo así como está. A pesar de obtener un accurracy de 81%, está muy desbalanceado en muchas clases. Se realizarán los cambios en el segundo modelo.

4. SEGUNDO MODELO

Después de analizar los resultados del primer modelo, lo primero que probé en cambiar fue el **steps per epoch a 100**, el **batch_size en train data a 128** y el **batch_size en validation en 64**, esto debido a que cada lote era muy pequeño, las gráficas obtenidas anteriormente se veían muy cortadas y el tiempo que se tardó en entrenar fue muy pequeño para la cantidad de datos que hay.

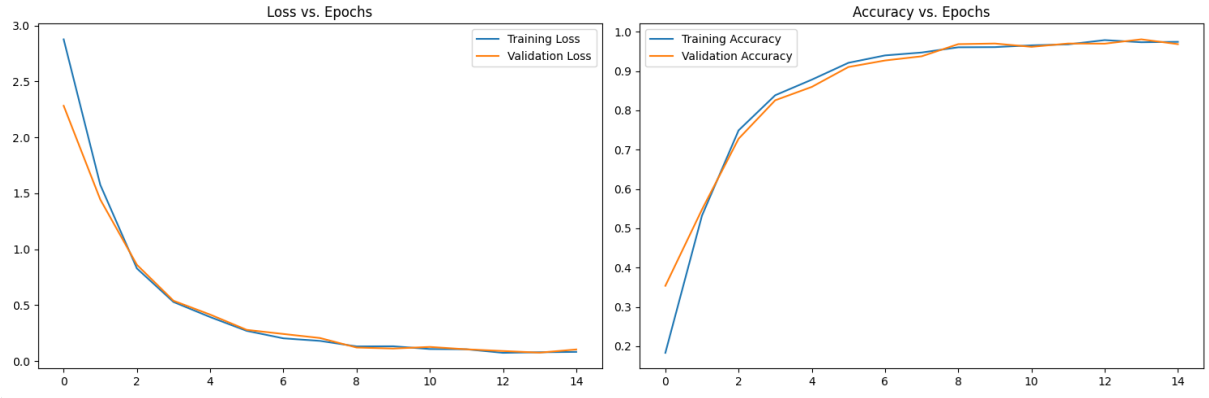
```
train_generator = train_datagen.flow_from_directory(
                  train_dir,
                  target_size = (150, 150),
                  batch_size = 128,
                  class_mode ='categorical')
```

```
val_generator = train_datagen.flow_from_directory(
    train_dir,
    target_size=(150, 150),
    batch_size=64,
    class_mode='categorical',
    subset='validation'  # Usar parte de los datos
)
```

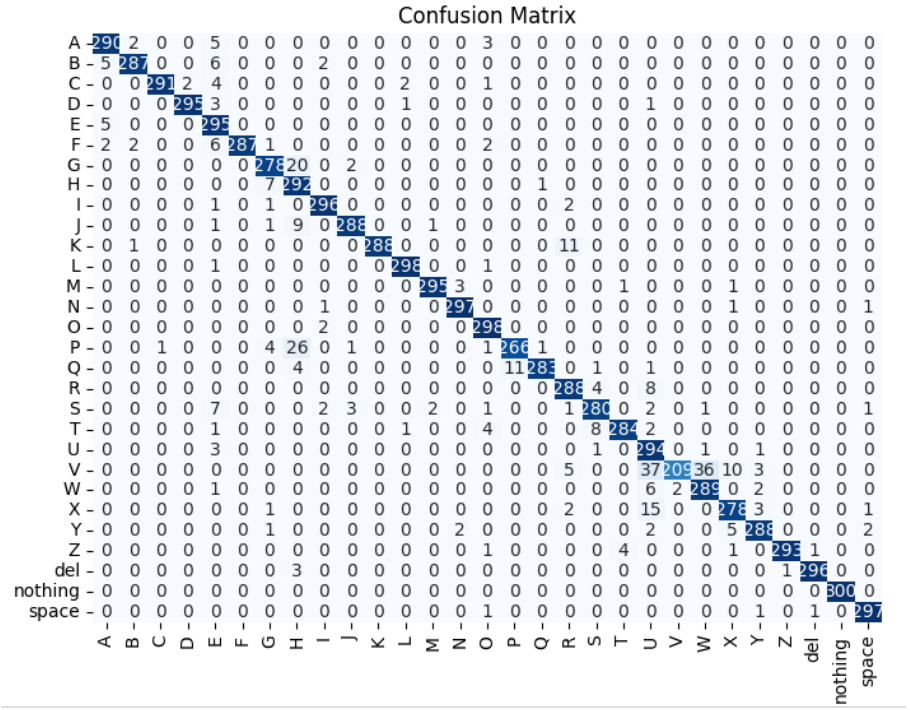
En cuanto al modelo y sus métricas lo dejé igual, solamente cambié el **número de epochs a 15** ya que el procesamiento será más pesado. Con estos ajustes, se espera que el modelo entrene mejor ya que aumentamos la cantidad de datos que se procesan en cada época. Si los resultados no son óptimos o si no se quita el overfitting se probará a cambiar la arquitectura de la red neuronal y agregar algunas técnicas de regularización. El código del segundo modelo se encuentra en el archivo “alphabet_2.py” en el repositorio de Github.

```
# Entrenar el modelo usando el
history = model.fit(
    train_generator,
    steps_per_epoch =100,
    epochs=15,  # Número de épocas
    validation_data=val_generator,
    validation_steps = 50
)
```

4.1 RESULTADOS DEL SEGUNDO MODELO



A diferencia del primer modelo podemos notar que, tanto el training como el validation se mantienen muy cercanas en ambas gráficas. Esto es un indicador que el modelo aprendió de manera adecuada y que ya no existe un posible overfitting. De igual manera, existe un accuracy mayor al anterior, arriba de 85%. Y la pérdida bajó hasta casi 0. En cuanto a las gráficas, se ve que el modelo entrenó de manera esperada.



En la matriz de confusión podemos notar que se redujo mucho la cantidad de predicciones erróneas en varias clases, incluso algunas tienen 0 predicciones incorrectas. Se verá con más detalle en la tabla de abajo:

	precision	recall	f1-score	support
A	0.96	0.97	0.96	300
B	0.98	0.96	0.97	300
C	1.00	0.97	0.98	300
D	0.99	0.98	0.99	300
E	0.88	0.98	0.93	300
F	1.00	0.96	0.98	300
G	0.95	0.93	0.94	300
H	0.82	0.97	0.89	300
I	0.98	0.99	0.98	300
J	0.98	0.96	0.97	300
K	1.00	0.96	0.98	300
L	0.99	0.99	0.99	300
M	0.99	0.98	0.99	300
N	0.98	0.99	0.99	300
O	0.95	0.99	0.97	300
P	0.96	0.89	0.92	300
Q	0.99	0.94	0.97	300
R	0.93	0.96	0.95	300
S	0.95	0.93	0.94	300
T	0.98	0.95	0.96	300
U	0.80	0.98	0.88	300
V	0.99	0.70	0.82	300
W	0.88	0.96	0.92	300
X	0.94	0.93	0.93	300
Y	0.97	0.96	0.96	300
Z	1.00	0.98	0.99	300
del	0.99	0.99	0.99	300
nothing	1.00	1.00	1.00	300
space	0.98	0.99	0.99	300
accuracy			0.96	8700
macro avg	0.96	0.96	0.96	8700
weighted avg	0.96	0.96	0.96	8700

Podemos observar que ahora ya no existe ese desbalance de accuracy entre clases que habíamos visto en el modelo anterior. En general, todas las clases tienen un accuracy mayor a 90%. Solamente la clase H fue la que menos accuracy tuvo con un valor de 82% , que incluso es mayor que el accuracy global que obtuvo el modelo anterior.

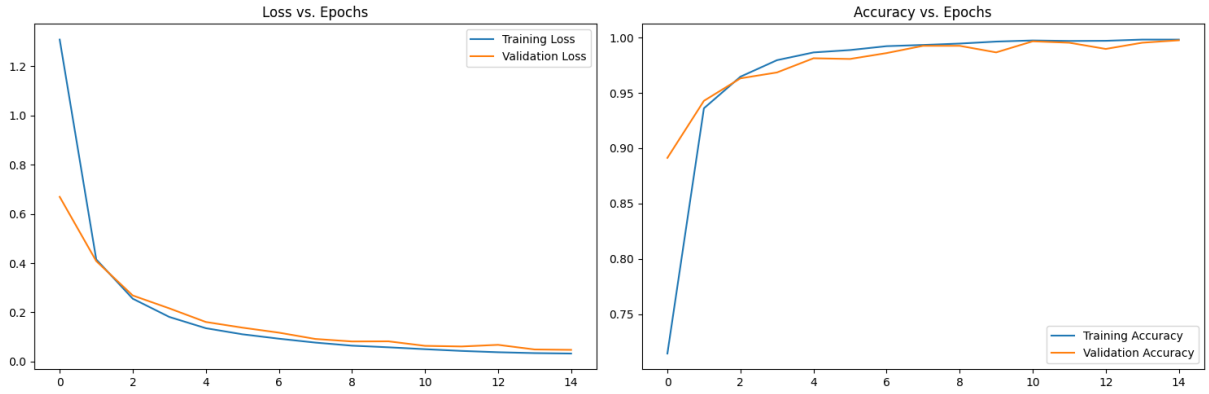
En conclusión, este modelo tuvo un resultado muy bueno y no se requiere de un ajuste muy grande ya que obtuve un accuracy de 96% y en casi todas las clases predice de manera correcta. Sin embargo, quiero ver una manera de reducir el error en dichas clases y subir el porcentaje de accuracy aunque sea 1%. Esas decisiones se tomarán en el modelo final.

5. MODELO FINAL

Antes de presentar el modelo final, cabe mencionar que probé modificar la arquitectura de mi modelo agregando más capas y quitando capas pero el accuracy no logró subir de ese 96% obtenido en el modelo anterior, incluso bajó la precisión en la mayoría de los casos. Debido a esto, probé implementar transfer learning con la arquitectura VGG16 porque tiene una arquitectura que está pensada para modelos de imágenes. Esto permitirá hacer mi modelo más robusto ya que tiene capas más complejas, con esto se espera que el modelo pueda aumentar el accuracy de 96%.

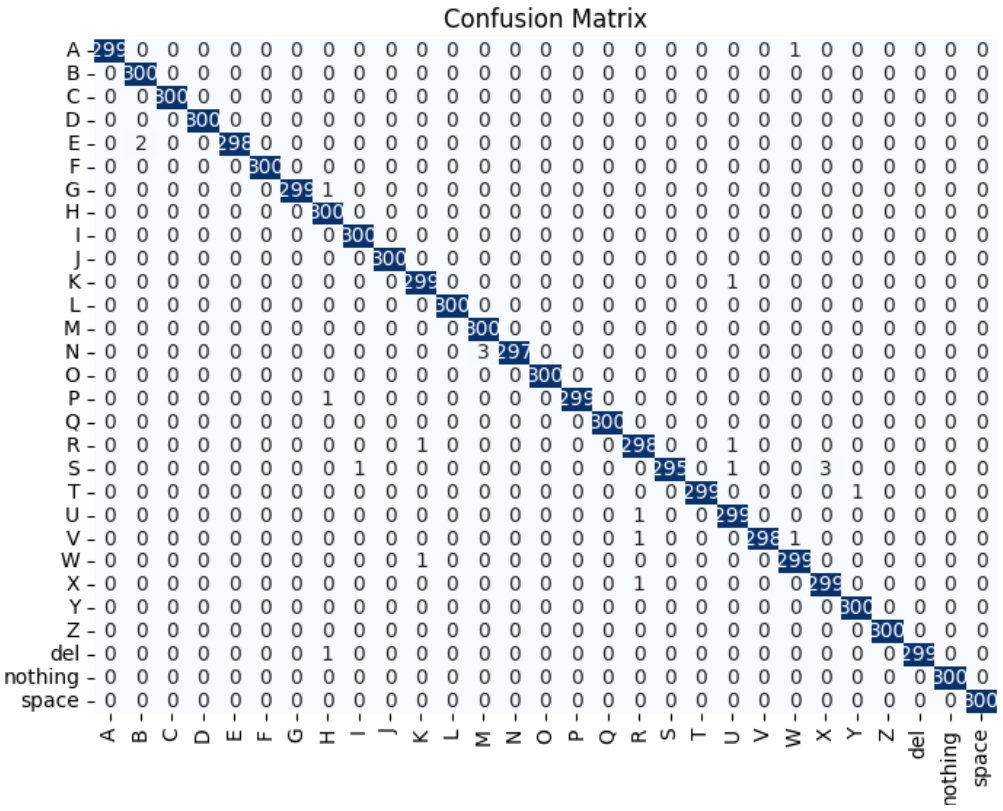
Para esto, congelé las capas de VGG16 para no entrenar toda la arquitectura y conservar los pesos anteriores. Agregué una capa densa con 29 neuronas que representan las 29 clases y puse a entrenar el modelo. Este código se puede observar en el archivo **“**alphabet_final.py**”** en el repositorio de Github.

5.1 RESULTADOS DEL TERCER MODELO



Aquí se pueden observar las gráficas de pérdida y accuracy del tercer modelo. Lo primero que se puede notar al agregar la arquitectura VGG16 es que el aprendizaje subió mucho desde la segunda epoch ya que pasó de 0.70 a 0.93, lo cual ya es un buen accuracy.

A diferencia del segundo modelo, aquí existen más ondulaciones en el accuracy de train con respecto al de validation, sin embargo, no son tan extensas y se mantienen en un accuracy arriba de 0.96, lo cual ya es mejor accuracy que el segundo modelo.



En cuanto a la matriz de confusión podemos notar que prácticamente no existen predicciones erróneas en la mayoría de las clases, y donde existen errores son en una muy pequeña cantidad, por lo que podemos decir que nuestro modelo está bien entrenado y no existen fallas en cuanto a overfitting, lo cual era un poco la preocupación en los resultados de las gráficas de arriba.

	precision	recall	f1-score	support
A	1.00	1.00	1.00	300
B	0.99	1.00	1.00	300
C	1.00	1.00	1.00	300
D	1.00	1.00	1.00	300
E	1.00	0.99	1.00	300
F	1.00	1.00	1.00	300
G	1.00	1.00	1.00	300
H	0.99	1.00	1.00	300
I	1.00	1.00	1.00	300
J	1.00	1.00	1.00	300
K	0.99	1.00	1.00	300
L	1.00	1.00	1.00	300
M	0.99	1.00	1.00	300
N	1.00	0.99	0.99	300
O	1.00	1.00	1.00	300
P	1.00	1.00	1.00	300
Q	1.00	1.00	1.00	300
R	0.99	0.99	0.99	300
S	1.00	0.98	0.99	300
T	1.00	1.00	1.00	300
U	0.99	1.00	0.99	300
V	1.00	0.99	1.00	300
W	0.99	1.00	1.00	300
X	0.99	1.00	0.99	300
Y	1.00	1.00	1.00	300
Z	1.00	1.00	1.00	300
del	1.00	1.00	1.00	300
nothing	1.00	1.00	1.00	300
space	1.00	1.00	1.00	300
accuracy			1.00	8700
macro avg	1.00	1.00	1.00	8700
weighted avg	1.00	1.00	1.00	8700

Finalmente, podemos confirmar en el reporte que el accuracy de todas las clases es incluso igual al 100% a excepción de algunas clases que solo bajan a 99%. Esto es un resultado mucho mejor al segundo modelo, el cual no era un modelo malo, pero se logró reducir el error en las predicciones de las clases. Este fue un modelo mucho más pesado que los primeros dos ya que tardó 2 horas en terminar de entrenar, sin embargo, se logró el resultado esperado, por lo que este será el modelo final para la implementación.

6. CONCLUSIÓN Y NOTAS ADICIONALES

A pesar de no haber realizado gran cantidad de ajustes a lo largo de la implementación, el modelo final obtuvo excelentes resultados. Esto es debido a que en general se buscó hacer buenas prácticas desde el primer modelo y no tener que cambiar de manera drástica todo el código para mejorar el resultado final. Se evaluaron los resultados de cada modelo y con base a eso se buscó mejorar algo específico de lo que fallaba.

El modelo final se puede evaluar ejecutando el código “alphabet_pruebas.py” que se encuentra en el repositorio de Github. Ese código permite poner a prueba el modelo con los

pesos cargados “pesos_del_modelofinal.h5”. El usuario puede ingresar el link de alguna imagen en tiempo real y puede ver las predicciones obtenidas de dicha imagen desde la consola.