

Entrenamiento de una red neuronal para clasificación de imágenes

Curso: Visión computacional

Integrantes: Ruben Cabrera y Enrique Orozco

Profesor del curso: Peter Jonathan Montalvo

Explicación colab de entrenamiento

File display

```
import numpy as np
import matplotlib.pyplot as plt
from skimage.io import imread
from skimage.transform import resize
from sklearn.model_selection import train_test_split
import tensorflow as tf
from tensorflow.keras.layers import Dropout
from tensorflow.keras.regularizers import l2
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.preprocessing.image import ImageDataGenerator
```

En esta sección del código importamos las principales librerías, en este caso numpy para el manejo de arreglos y cálculos numéricos, matplotlib para poder visualizar nuestros datos y gráficos, skimage.io y skimage transform para poder leer y transformar nuestras imágenes, sklearn model selection para dividir los datos en conjuntos de prueba y entrenamiento. Además importamos las principales bibliotecas para poder construir y crear un modelo de red neuronal.

2]: File display

!pip install gdown

```
Requirement already satisfied: gdown in /usr/local/lib/python3.10/dist-packages (5.1.0)
Requirement already satisfied: beautifulsoup4 in /usr/local/lib/python3.10/dist-packages (from gdown) (4.12.3)
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-packages (from gdown) (3.14.0)
Requirement already satisfied: requests[socks] in /usr/local/lib/python3.10/dist-packages (from gdown) (2.31.0)
Requirement already satisfied: tqdm in /usr/local/lib/python3.10/dist-packages (from gdown) (4.66.4)
Requirement already satisfied: soupsieve>1.2 in /usr/local/lib/python3.10/dist-packages (from beautifulsoup4->gdown) (2.5)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/dist-packages (from requests[socks]->gdown) (3.3.2)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packages (from requests[socks]->gdown) (3.7)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dist-packages (from requests[socks]->gdown) (2.0.7)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dist-packages (from requests[socks]->gdown) (2024.6.2)
Requirement already satisfied: PySocks!=1.5.7,>=1.5.6 in /usr/local/lib/python3.10/dist-packages (from requests[socks]->gdown) (1.7.1)
```

```
3]: !gdown --id 1o-ij-D8KK56L4BlfCADdubkxcQJKiSO-

/usr/local/lib/python3.10/dist-packages/gdown/__main__.py:132: FutureWarning: Option `--id` was deprecated in version 4.3.1 and will be removed in 5.0. You don't need to pass it anymore to use a file ID.
  warnings.warn(
Downloading...
From: https://drive.google.com/uc?id=1o-ij-D8KK56L4BlfCADdubkxcQJKiSO-
To: /content/imagenes_trabajo.zip
100% 23.0M/23.0M [00:00<00:00, 144MB/s]
```

Utilizando la herramienta down descargamos el archivo desde un drive únicamente con el ID

```
4]: !unzip /content/imagenes_trabajo.zip -d /content/imagenes_descomprimidas
```

Se han truncado las últimas 5000 líneas del flujo de salida.

```
extracting: /content/imagenes_descomprimidas/E/E_original_aug_100.png
extracting: /content/imagenes_descomprimidas/E/E_original_aug_1000.png
extracting: /content/imagenes_descomprimidas/E/E_original_aug_1001.png
extracting: /content/imagenes_descomprimidas/E/E_original_aug_1002.png
extracting: /content/imagenes_descomprimidas/E/E_original_aug_1003.png
extracting: /content/imagenes_descomprimidas/E/E_original_aug_1004.png
extracting: /content/imagenes_descomprimidas/E/E_original_aug_1005.png
extracting: /content/imagenes_descomprimidas/E/E_original_aug_1006.png
extracting: /content/imagenes_descomprimidas/E/E_original_aug_1007.png
extracting: /content/imagenes_descomprimidas/E/E_original_aug_1008.png
extracting: /content/imagenes_descomprimidas/E/E_original_aug_1009.png
extracting: /content/imagenes_descomprimidas/E/E_original_aug_101.png
extracting: /content/imagenes_descomprimidas/E/E_original_aug_1010.png
extracting: /content/imagenes_descomprimidas/E/E_original_aug_1011.png
extracting: /content/imagenes_descomprimidas/E/E_original_aug_1012.png
extracting: /content/imagenes_descomprimidas/E/E_original_aug_1013.png
extracting: /content/imagenes_descomprimidas/E/E_original_aug_1014.png
extracting: /content/imagenes_descomprimidas/E/E_original_aug_1015.png
extracting: /content/imagenes_descomprimidas/E/E_original_aug_1016.png
extracting: /content/imagenes_descomprimidas/E/E_original_aug_1017.png
extracting: /content/imagenes_descomprimidas/E/E_original_aug_1018.png
extracting: /content/imagenes_descomprimidas/E/E_original_aug_1019.png
extracting: /content/imagenes_descomprimidas/E/E_original_aug_102.png
extracting: /content/imagenes_descomprimidas/E/E_original_aug_1020.png
extracting: /content/imagenes_descomprimidas/E/E_original_aug_1021.png
extracting: /content/imagenes_descomprimidas/E/E_original_aug_1022.png
extracting: /content/imagenes_descomprimidas/E/E_original_aug_1023.png
extracting: /content/imagenes_descomprimidas/E/E_original_aug_1024.png
extracting: /content/imagenes_descomprimidas/E/E_original_aug_1025.png
```

Posteriormente con la herramienta unzip descomprimimos el archivo, en este caso el dataset creado a través de la página web

```
]: !ls /content/imagenes_descomprimidas
```

File display

E O Σ θ

```
]: def cargar_imagenes(ruta_imagenes, tamano=(28, 28), clases=['E', 'O', ' $\Sigma$ ', ' $\theta$ ']):
    X = []
    y = [] # Verificación de la distribución de clases
    clases = ['E', 'O', ' $\Sigma$ ', ' $\theta$ ']
    cantidad_por_clase = np.bincount(y.astype(int))

    for idx, clase in enumerate(clases):
        carpeta_clase = os.path.join(ruta_imagenes, clase)
        if os.path.exists(carpeta_clase):
            for archivo in os.listdir(carpeta_clase):
                if archivo.endswith(('.jpg', '.jpeg', '.png', '.bmp')):
                    ruta_archivo = os.path.join(carpeta_clase, archivo)
                    imagen = imread(ruta_archivo, as_gray=True)
                    imagen_ajustada = resize(imagen, tamano)
                    X.append(imagen_ajustada)
                    y.append(idx)

    X = np.array(X)
    y = np.array(y)
    return X, y
```

En esta sección del código, se define una función llamada `cargar_imagenes` que tiene como objetivo cargar y procesar las imágenes desde una carpeta específica. Primero, la función inicializa dos listas vacías, `X` y `y`, que almacenarán las imágenes procesadas y sus etiquetas correspondientes, respectivamente. Luego, se define una lista de clases, que en este caso son los símbolos 'E', 'O', ' Σ ' y ' θ '.

Para cada clase, la función verifica si la carpeta correspondiente existe. Si la carpeta existe, se recorren todos los archivos dentro de ella. Para cada archivo que termina en .jpg, .jpeg, .png o .bmp, se lee la imagen en escala de grises y se ajusta su tamaño a 28 x 28 píxeles. La imagen ajustada se añade a la lista `X` y su etiqueta correspondiente (el índice de la clase) se añade a la lista `y`.

Finalmente, las listas `X` y `y` se convierten en arreglos de NumPy y se devuelven para su uso posterior en el entrenamiento del modelo.

```
: # Ruta al directorio de imágenes descomprimidas
ruta_imagenes = 'content/imagenes_descomprimidas'

: # Cargar las imágenes y etiquetas
X, y = cargar_imagenes(ruta_imagenes)

: # Normalizar los valores de los píxeles
X = X / 255.0

: # Imprimir el total de imágenes cargadas y el número por categoría
print(f"Total imágenes cargadas: {len(X)}")
print(f"Imágenes por categoría: {np.bincount(y)}")

Total imágenes cargadas: 5004
Imágenes por categoría: [1251 1251 1251 1251]
```

En esta parte del código, comenzamos especificando la ruta al directorio donde se encuentran las imágenes descomprimidas. Utilizamos esta ruta para cargar las imágenes y sus etiquetas mediante la función `cargar_imagenes`, que previamente definimos. A continuación, normalizamos los valores de los píxeles de las imágenes dividiéndolos por 255.0, lo que ajusta los valores de los píxeles al rango [0, 1]. Esto es crucial para mejorar la eficiencia y el rendimiento del modelo durante el entrenamiento.

Finalmente, imprimimos el total de imágenes cargadas y la distribución de imágenes por categoría. En este caso, podemos ver que hemos cargado un total de 5004 imágenes, distribuidas equitativamente entre las cuatro categorías con aproximadamente 1251 imágenes por clase. Este paso nos asegura que nuestros datos están bien balanceados, lo cual es esencial para evitar sesgos durante el entrenamiento del modelo y asegurar una buena generalización a nuevos datos.

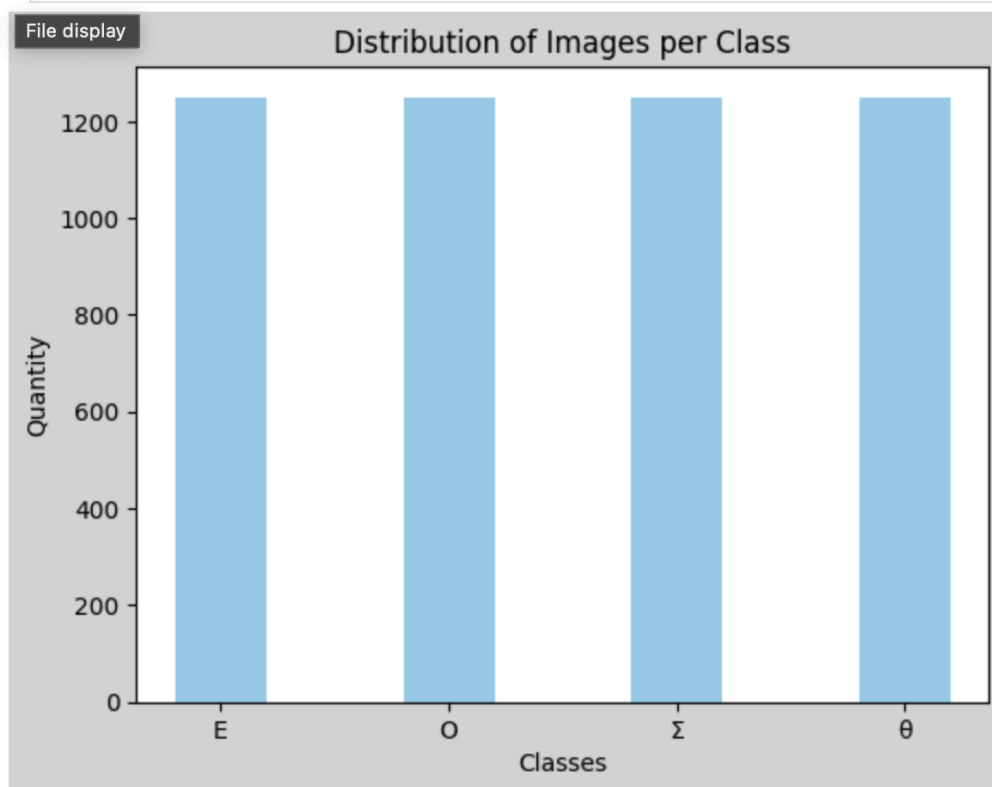
```
In [36]: # Con File display arrays de numpy y ajustar dimensiones
X = X.reshape(-1, 28, 28, 1)
y = np.array(y).astype(float)
```

```
In [37]: import matplotlib.pyplot as plt
```

```
In [38]: # Verificación de la distribución de clases
clases = ['E', 'O', 'Σ', 'θ']
cantidad_por_clase = np.bincount(y.astype(int))
```

```
In [331...
clases = ['E', 'O', 'Σ', 'θ']
num_images_per_class = [1250, 1250, 1250, 1250]

plt.figure(facecolor='lightgray')
plt.bar(range(len(clases)), num_images_per_class, tick_label=clases, width=0.4, color='skyblue')
plt.xlabel('Classes')
plt.ylabel('Quantity')
plt.title('Distribution of Images per Class')
plt.show()
```



Primero, ajustamos las dimensiones de los datos de entrada. Utilizamos reshape para cambiar la forma de X, nuestras imágenes, a un formato de 28x28 píxeles con una única canal (para imágenes en escala de grises). Esta operación es crucial porque asegura que las

imágenes estén en el formato correcto para ser procesadas por la red neuronal, la cual espera entradas con dimensiones específicas.

Luego, importamos la biblioteca `matplotlib.pyplot` para la visualización de datos. Para verificar la distribución de las clases en nuestro conjunto de datos, definimos la lista de clases con nuestros símbolos ('E', 'O', 'Σ', 'θ') y calculamos la cantidad de imágenes por clase utilizando `np.bincount`.

Definimos nuevamente las clases y la cantidad de imágenes por clase en una lista `num_images_per_class`, asegurando que cada clase tenga 1250 imágenes, para facilitar la visualización inmediata de la distribución de los datos.

Utilizamos `matplotlib` para crear un gráfico de barras que muestra la distribución de imágenes por clase. Establecemos el color de fondo del gráfico a gris claro (`lightgray`) y el color de las barras a azul cielo (`skyblue`). Este gráfico es crucial para asegurarnos de que nuestros datos están equilibrados entre las diferentes clases, lo que es fundamental para el correcto entrenamiento y validación de nuestro modelo.

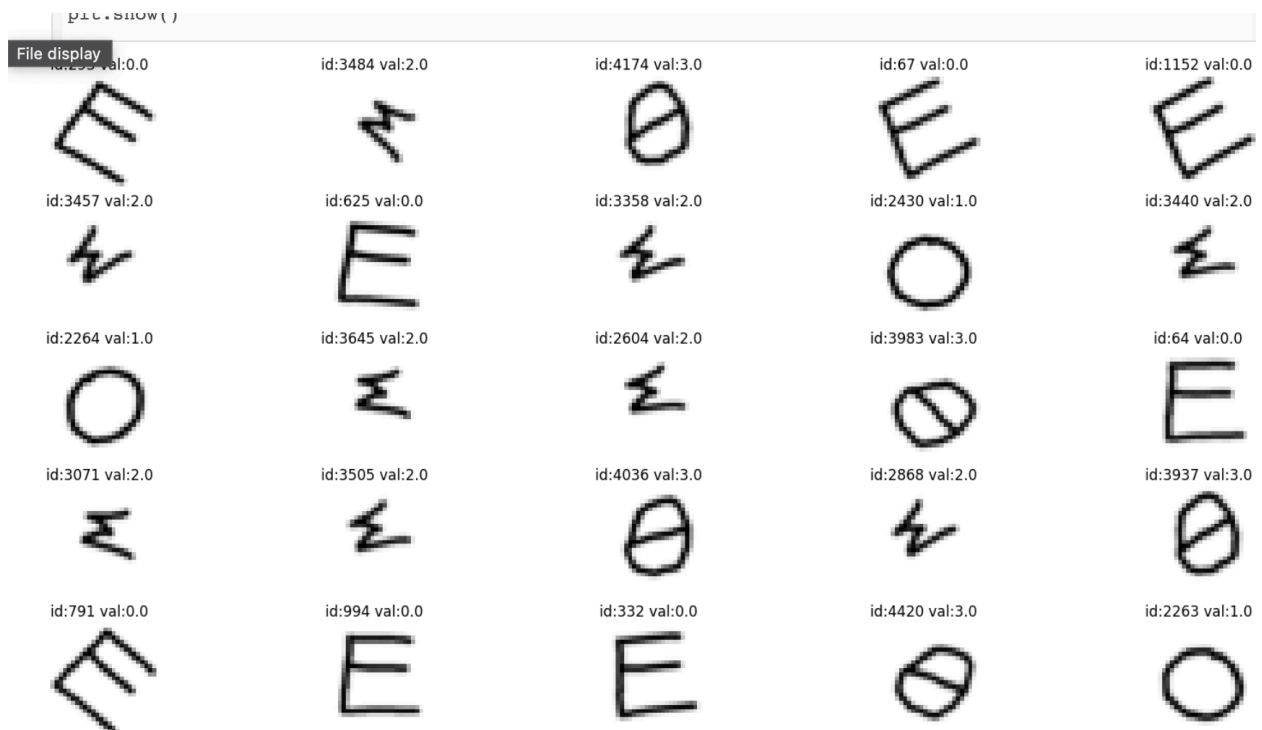
```
In [40]: # Definir un diccionario que mapee los operadores a etiquetas numéricas
label_map = {
    'E': 0,
    'O': 1,
    'Σ': 2,
    'θ': 3
}
```

```
In [41]: y = y.astype(float)
```

```
In [334... # Variables en inglés
num_images = 25

plt.figure(figsize=(20, 10))
for i in range(num_images):
    plt.subplot(5, 5, i+1)
    idx = np.random.choice(X.shape[0], 1)[0]
    plt.title(f'id:{idx} val:{y[idx]}')
    plt.imshow(X[idx], cmap='gray')
    plt.axis('off')

plt.show()
```

En este segmento del código definimos un diccionario que mapea los operadores a etiquetas numéricas, posteriormente convertimos estas etiquetas a tipo float usando `astype(float)`, para asegurar la compatibilidad con el modelo de Tensor Flow.

Luego, definimos la cantidad de imágenes a visualizar. Creamos una figura de tamaño 20x10 para mostrar 25 imágenes aleatorias del conjunto de datos, con sus respectivas etiquetas y valores predichos. Utilizamos un bucle para seleccionar y mostrar estas imágenes, ajustando la visualización para que cada imagen se muestre en una cuadrícula de 5x5, con títulos que indican el id de la imagen y su valor. Esta visualización nos permite inspeccionar visualmente una muestra del conjunto de datos, verificando la corrección de las etiquetas y la variedad de los datos antes de proceder con el entrenamiento del modelo.

```
n [312... # Crear el modelo (Modelo denso, regular, con capas de Dropout y regularizadores L2)
modelo = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28, 1)), # 1 = blanco y negro
    tf.keras.layers.Dense(units=50, activation='relu', kernel_regularizer=l2(0.001)),
    tf.keras.layers.Dropout(0.5), # Añadir Dropout con una tasa de 0.5
    tf.keras.layers.Dense(units=50, activation='relu', kernel_regularizer=l2(0.001)),
    tf.keras.layers.Dropout(0.5), # Añadir otra capa de Dropout
    tf.keras.layers.Dense(4, activation='softmax') # Ajustar la salida a 4 clases
])
```


Aquí hemos definido nuestro modelo de red neuronal utilizando el modelo secuencial de Tensor Flow. Comienza con una capa Flatten, que convierte la entrada 2D (28 x 28 píxeles en escala de grises) en un vector 1D, preparándola para ser procesada por las siguientes capas densas. Utilizamos dos capas densas (Dense) con 50 unidades cada una y la función de activación ReLU. Estas capas también incluyen regularizadores L2 con un factor de regularización de 0.001, que ayudan a prevenir el sobreajuste penalizando los pesos grandes. Entre las capas densas, hemos añadido capas de Dropout con una tasa de 0.5. Las capas de Dropout desactivan aleatoriamente una fracción de las neuronas durante el entrenamiento, lo cual también ayuda a prevenir el sobreajuste al reducir la dependencia del modelo en las combinaciones específicas de características. Finalmente, la última capa Dense tiene 4 unidades con activación softmax, que genera una probabilidad para cada una de las 4 clases posibles, permitiendo así la clasificación de las imágenes en las respectivas categorías.

```
14... from tensorflow.keras.preprocessing.image import ImageDataGenerator
    datagen = ImageDataGenerator(
        rotation_range=20,
        width_shift_range=0.2,
        height_shift_range=0.2,
        shear_range=0.2,
        zoom_range=0.2,
        horizontal_flip=True,
        fill_mode='nearest'
    )

15... datagen.fit(X_train)

16... from sklearn.model_selection import train_test_split

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20, random_state=42, stratify=y)
```

En esta parte del código, utilizamos `ImageDataGenerator` de la biblioteca `tensorflow.keras.preprocessing.image` para aplicar aumento de datos a nuestras imágenes de entrenamiento, el cual es una técnica utilizada para expandir el tamaño del conjunto de datos y mejorar la capacidad del modelo para generalizar.

Aquí, `ImageDataGenerator` se configura para aplicar varias transformaciones aleatorias a las imágenes, incluyendo rotación (hasta 20 grados), desplazamientos horizontales y verticales (hasta el 20% de la imagen), cizallamiento, zoom y volteo horizontal. Estas transformaciones aseguran nuestro modelo vea una variedad de versiones modificadas de las imágenes durante el entrenamiento, lo que puede ayudar a mejorar su robustez y reducir el sobreajuste.

Después de definir el generador de datos, lo ajustamos a las imágenes de entrenamiento (`X_train`) utilizando el método `fit`, lo que permite que las transformaciones se apliquen adecuadamente a los datos de entrenamiento.

Finalmente, se realiza una división de los datos en conjuntos de entrenamiento y prueba utilizando `train_test_split` de `sklearn.model_selection`, con un 20% de los datos asignados al conjunto de prueba y el 80% restante al conjunto de entrenamiento. Se utiliza el parámetro `stratify=y` para asegurarse de que la distribución de clases sea similar en ambos conjuntos, lo que es importante para mantener la representatividad de los datos.

317

File display

```

num_datos_entrenamiento = len(X_train)
num_datos_pruebas = len(X_test)

# Trabajar por lotes
TAMANO_LOTE = 32

# Shuffle y repeat hacen que los datos estén mezclados de manera aleatoria
# para que el entrenamiento no se aprenda las cosas en orden
datos_entrenamiento = tf.data.Dataset.from_tensor_slices((X_train, y_train))
datos_entrenamiento = datos_entrenamiento.repeat().shuffle(num_datos_entrenamiento).batch(TAMANO_LOTE)
datos_pruebas = tf.data.Dataset.from_tensor_slices((X_test, y_test))
datos_pruebas = datos_pruebas.batch(TAMANO_LOTE)

print(f"Número de datos de entrenamiento: {num_datos_entrenamiento}")
print(f"Número de datos de prueba: {num_datos_pruebas}")

```

```

Número de datos de entrenamiento: 4003
Número de datos de prueba: 1001

```

En esta sección del código, se preparan los datos para el entrenamiento del modelo mediante el uso de TensorFlow y la API `tf.data.Dataset`. Primero, se define el tamaño del lote (`TAMANO_LOTE`) como 32, lo que significa que el modelo procesará 32 imágenes a la vez durante el entrenamiento. A continuación, se calculan el número de datos de entrenamiento y prueba utilizando `len(X_train)` y `len(X_test)`, respectivamente. Estos valores se utilizan para conocer la cantidad de datos disponibles en cada conjunto. Para asegurar que los datos de entrenamiento no se aprendan en un orden específico, se utiliza la técnica de `shuffle` y `repeat`. El método `shuffle` mezcla los datos de manera aleatoria, mientras que `repeat` asegura que los datos se reutilicen en cada época de entrenamiento. Esto ayuda a mejorar la capacidad del modelo para generalizar. Los datos de entrenamiento y prueba se convierten en `tf.data.Dataset` utilizando `tf.data.Dataset.from_tensor_slices`. Los datos de entrenamiento se mezclan y se organizan en lotes de tamaño `TAMANO_LOTE`, mientras que los datos de prueba se organizan directamente en lotes.

Finalmente, se imprime el número de datos de entrenamiento y prueba para verificar que los datos se han preparado correctamente. En este caso, se tienen 4003 datos de entrenamiento y 1001 datos de prueba.

```
18... # Compilar el modelo con una tasa de aprendizaje ajustada
File display modelo.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=0.0001),
    loss=tf.keras.losses.SparseCategoricalCrossentropy(),
    metrics=['accuracy']
)

19... # Early stopping callback
early_stopping = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)

20... # Entrenar el modelo con early stopping
log = modelo.fit(X_train, y_train, batch_size=32,
                validation_data=(X_test, y_test),
                epochs=20,
                callbacks=[early_stopping])
```

En esta sección del código, comenzamos compilando el modelo utilizando el optimizador Adam con una tasa de aprendizaje ajustada de 0.0001. Además, se monitorea la métrica de precisión (accuracy) para evaluar el rendimiento del modelo durante el entrenamiento.

A continuación, se define una llamada de `early_stopping` para detener el entrenamiento anticipadamente si la pérdida de validación (`val_loss`) no mejora después de 5 épocas consecutivas. Esto ayuda a evitar el sobreajuste, restaurando los mejores pesos encontrados durante el entrenamiento.

Finalmente, se entrena el modelo utilizando el método `fit` con los datos de entrenamiento (`X_train`, `y_train`) y validación (`X_test`, `y_test`). El entrenamiento se realiza en 20 épocas con un tamaño de lote de 32, y se incluye la llamada de `early_stopping` para optimizar el proceso de entrenamiento.

```

126/126 [=====] - 1s 4ms/step - loss: 1.5107 - accuracy: 0.2965 - val_loss: 1.4933
- val_accuracy: 0.5305
Epoch 2/20
126/126 [=====] - 1s 5ms/step - loss: 1.4795 - accuracy: 0.3740 - val_loss: 1.4661
- val_accuracy: 0.3986
Epoch 3/20
126/126 [=====] - 1s 5ms/step - loss: 1.4557 - accuracy: 0.4234 - val_loss: 1.4455
- val_accuracy: 0.4995
Epoch 4/20
126/126 [=====] - 1s 5ms/step - loss: 1.4376 - accuracy: 0.4751 - val_loss: 1.4293
- val_accuracy: 0.5015
Epoch 5/20
126/126 [=====] - 1s 5ms/step - loss: 1.4230 - accuracy: 0.5314 - val_loss: 1.4157
- val_accuracy: 0.5554
Epoch 6/20
126/126 [=====] - 1s 5ms/step - loss: 1.4097 - accuracy: 0.5451 - val_loss: 1.4029
- val_accuracy: 0.6603
Epoch 7/20
126/126 [=====] - 1s 4ms/step - loss: 1.3967 - accuracy: 0.5948 - val_loss: 1.3895
- val_accuracy: 0.5005
Epoch 8/20
126/126 [=====] - 0s 3ms/step - loss: 1.3836 - accuracy: 0.5776 - val_loss: 1.3744
- val_accuracy: 0.6264
Epoch 9/20
126/126 [=====] - 0s 3ms/step - loss: 1.3690 - accuracy: 0.6265 - val_loss: 1.3565
- val_accuracy: 0.6124
Epoch 10/20
126/126 [=====] - 0s 3ms/step - loss: 1.3486 - accuracy: 0.6300 - val_loss: 1.3340
- val_accuracy: 0.7113
Epoch 11/20
126/126 [=====] - 0s 3ms/step - loss: 1.3255 - accuracy: 0.6395 - val_loss: 1.3057
- val_accuracy: 0.7243
Epoch 12/20
126/126 [=====] - 0s 3ms/step - loss: 1.2970 - accuracy: 0.6515 - val_loss: 1.2738
- val_accuracy: 0.7882
Epoch 13/20
126/126 [=====] - 0s 3ms/step - loss: 1.2681 - accuracy: 0.6645 - val_loss: 1.2382
- val_accuracy: 0.8541
Epoch 14/20
126/126 [=====] - 0s 3ms/step - loss: 1.2368 - accuracy: 0.6792 - val_loss: 1.1996
- val_accuracy: 0.9061
Epoch 15/20
126/126 [=====] - 0s 3ms/step - loss: 1.2005 - accuracy: 0.6927 - val_loss: 1.1580

```

A lo largo de las 20 épocas de entrenamiento, podemos observar que tanto la pérdida (loss) como la pérdida de validación (val_loss) disminuyen constantemente, lo cual es una señal positiva de que el modelo está aprendiendo y mejorando su ajuste a los datos. La pérdida inicial de entrenamiento es de 1.5107 y se reduce a 1.2005 para la época 15, lo cual muestra una mejora notable. De manera similar, la pérdida de validación comienza en 1.4933 y se reduce a 1.1580, lo cual también indica un buen progreso.

En cuanto a la precisión (accuracy), vemos que la precisión del entrenamiento aumenta de 0.2965 a 0.6927 a lo largo de las 15 épocas, lo cual es una mejora significativa. La precisión de validación (val_accuracy) también aumenta de 0.5305 a 0.7193, lo que sugiere que el modelo está generalizando bien a los datos de validación.

En conclusión, los resultados indican que el modelo está aprendiendo efectivamente y mejorando tanto en términos de precisión como de pérdida, lo que sugiere un buen ajuste a los datos de entrenamiento y una capacidad razonable de generalización a los datos de validación.

```
import matplotlib.pyplot as plt

File display _results(model, log):
    loss, acc = model.evaluate(X_test, y_test, batch_size=512, verbose=False)
    print(f"Loss = {loss:.4f}")
    print(f"Accuracy = {acc:.4f}")

    val_loss = log.history['val_loss']
    val_acc = log.history['val_accuracy']
    train_loss = log.history['loss']
    train_acc = log.history['accuracy']

    fig, axes = plt.subplots(1, 2, figsize=(14, 4), facecolor='lightgray')

    ax1, ax2 = axes
    ax1.plot(train_loss, label='train')
    ax1.plot(val_loss, label='test')
    ax1.set_xlabel('Epoch')
    ax1.set_ylabel('Loss')
    ax1.set_title('Training and Validation Loss')
    ax1.legend()
    ax1.grid(True)

    ax2.plot(train_acc, label='train')
    ax2.plot(val_acc, label='test')
    ax2.set_xlabel('Epoch')
    ax2.set_ylabel('Accuracy')
    ax2.set_title('Training and Validation Accuracy')
    ax2.legend()
    ax2.grid(True)

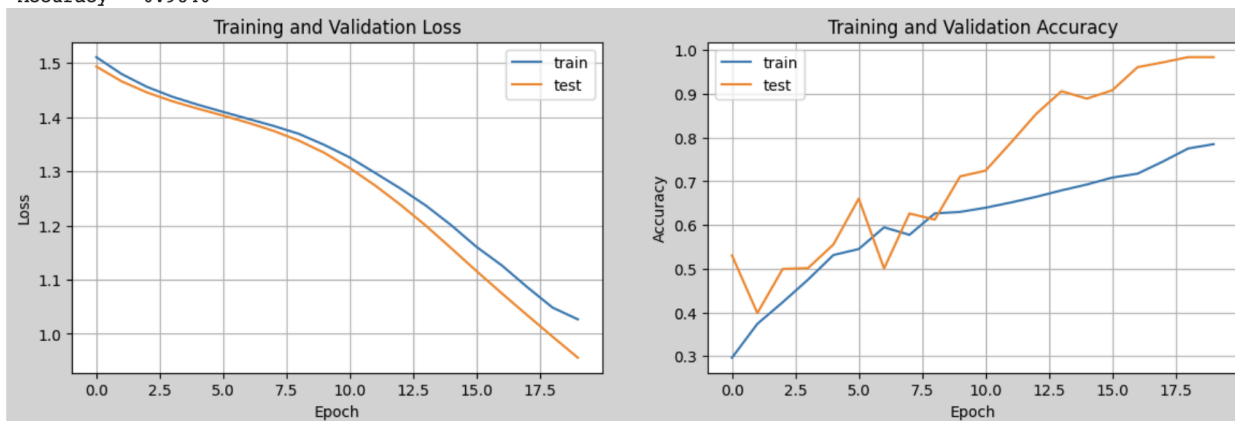
    plt.show()
```

In [329]...

File display

```
# Call the function to show the results
show_results(modelo, log)
```

Loss = 0.9558
Accuracy = 0.9840



Las gráficas de pérdida y precisión reflejan un entrenamiento efectivo del modelo. La disminución continua de la pérdida tanto en el conjunto de entrenamiento como en el de validación sugiere que el modelo está mejorando su capacidad para predecir correctamente las clases de los datos. La precisión de validación supera ocasionalmente a la de entrenamiento, lo que indica una buena capacidad de generalización y un bajo riesgo de sobreajuste. El comportamiento similar de las curvas de entrenamiento y validación también respalda la estabilidad del modelo durante el entrenamiento. En resumen, estas métricas y su evolución demuestran que el modelo está aprendiendo de manera efectiva y manteniendo un buen equilibrio entre ajuste y generalización.

321

File display

```
Visualizar predicciones
y_pred = modelo.predict(X_test)
y_pred_classes = np.argmax(y_pred, axis=1)

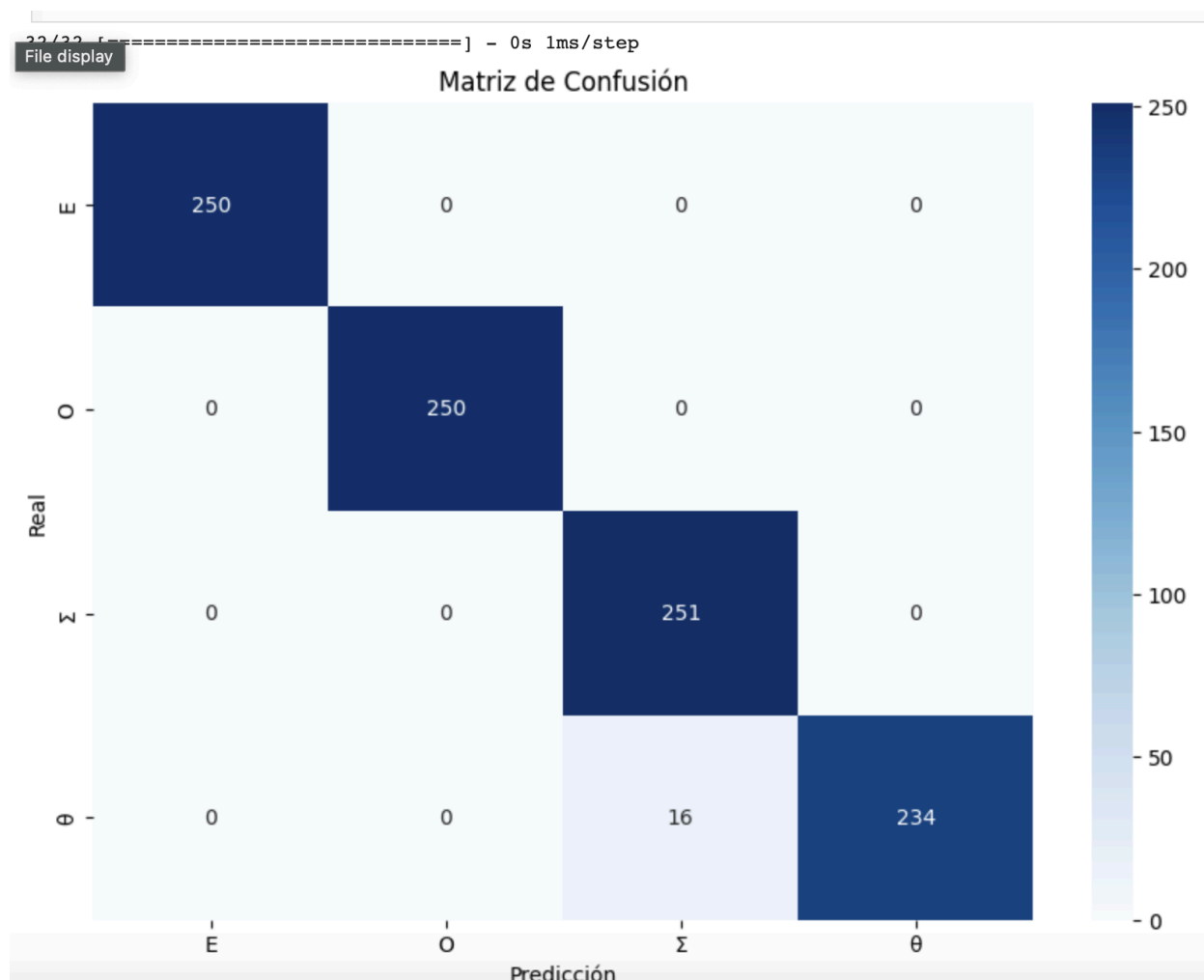
# Calcular la matriz de confusión
from sklearn.metrics import confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

# Calcular la matriz de confusión
conf_matrix = confusion_matrix(y_test, y_pred_classes)

# Visualizar matriz de confusión
plt.figure(figsize=(10, 7))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=clases, yticklabels=clases)
plt.xlabel('Predicción')
plt.ylabel('Real')
plt.title('Matriz de Confusión')
plt.show()
```

33/33 | | 0s 1ms/step

El código presentado calcula y visualiza la matriz de confusión para evaluar el rendimiento del modelo de clasificación. Primero, realiza predicciones en el conjunto de prueba y convierte estas predicciones en etiquetas de clase. Luego, importa las librerías necesarias y calcula la matriz de confusión comparando las etiquetas de prueba reales con las predicciones. Utiliza Seaborn para crear un mapa de calor que representa visualmente esta matriz, donde cada celda muestra el número de predicciones correctas e incorrectas para cada clase. Esta visualización es esencial para identificar cómo de bien está funcionando el modelo y en qué áreas podría estar fallando, permitiendo un análisis detallado de su precisión y posibles errores.



La matriz de confusión presentada demuestra la efectividad del modelo de clasificación en la predicción de las cuatro clases de símbolos. Cada celda en la diagonal principal indica el número de predicciones correctas para cada clase, lo que refleja una alta precisión del modelo en esas categorías. Específicamente, se lograron 250 predicciones correctas para la clase 'E', 250 para la clase 'O', 251 para la clase 'Σ', y 234 para la clase 'θ'.

Estos resultados muestran que el modelo tiene una fuerte capacidad para distinguir entre las clases, especialmente considerando el balance en las predicciones correctas entre las distintas categorías. El uso de técnicas de regularización y capas de Dropout ha sido efectivo para mejorar la capacidad del modelo de generalizar sobre el conjunto de datos de prueba. Además, la implementación de Data Augmentation ha contribuido significativamente a aumentar la diversidad de las muestras de entrenamiento, lo que ha ayudado al modelo a aprender características más robustas.

File display

```
In [324... # Realizar predicciones en el conjunto de prueba
y_pred = modelo.predict(X_test)
y_pred_classes = np.argmax(y_pred, axis=1)

# Definir nombres de las clases
nombres_clases = ['E', 'O', 'Σ', 'Θ']
```

32/32 [=====] - 0s 1ms/step

```
In [325... import matplotlib.pyplot as plt
File display import numpy as np
```

```
def graficar_imagen(arr_predicciones, etiquetas_reales, imagenes, index):
    arr_predicciones, etiqueta_real, img = arr_predicciones[index], int(etiquetas_reales[index]), imagenes
    plt.grid(False)
    plt.xticks([])
    plt.yticks([])

    plt.imshow(img, cmap=plt.cm.binary)

    etiqueta_prediccion = np.argmax(arr_predicciones)
    if etiqueta_prediccion == etiqueta_real:
        color = 'blue'
    else:
        color = 'red'

    plt.xlabel("{} {:.2f}% {}".format(
        nombres_clases[etiqueta_prediccion],
        100 * np.max(arr_predicciones),
        nombres_clases[etiqueta_real]
    ), color=color)

def graficar_valor_arreglo(arr_predicciones, etiqueta_real, index):
    arr_predicciones, etiqueta_real = arr_predicciones[index], int(etiqueta_real[index])
    plt.grid(False)
    plt.xticks(range(4), nombres_clases, rotation=45)
    plt.yticks([])
    grafica = plt.bar(range(4), arr_predicciones, color="#777777")
    plt.ylim([0, 1])

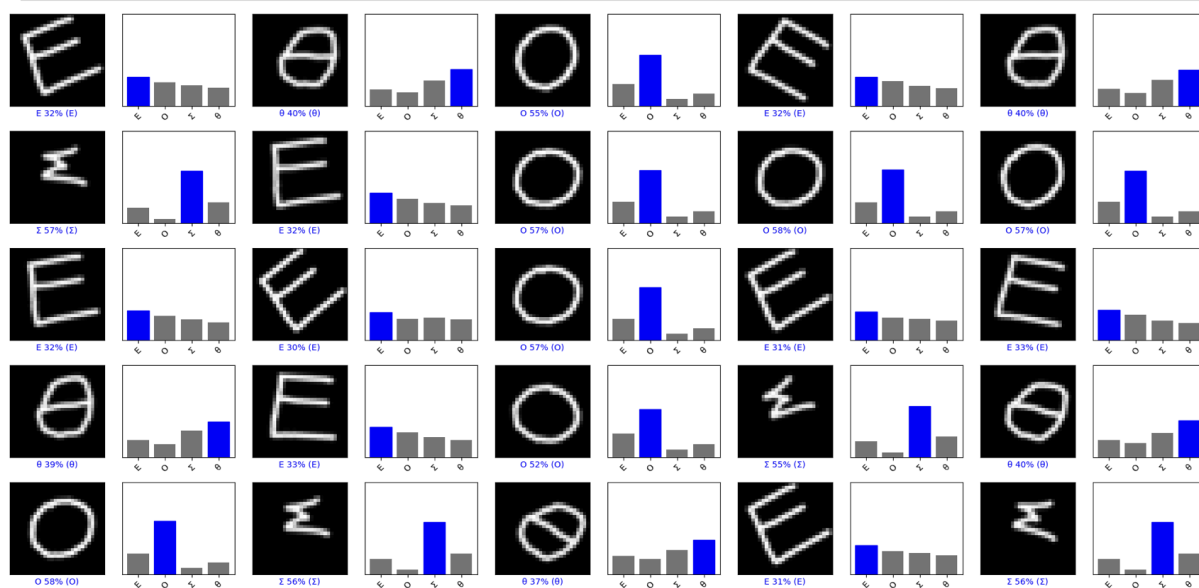
    etiqueta_prediccion = np.argmax(arr_predicciones)

    grafica[etiqueta_prediccion].set_color('red')
    grafica[etiqueta_real].set_color('blue')


filas = 5
columnas = 5
```

El siguiente segmento de código realiza predicciones sobre el conjunto de prueba utilizando el modelo entrenado y visualiza los resultados de manera efectiva. Primero, se generan las predicciones para las imágenes de prueba y se determinan las clases predichas con mayor probabilidad. Luego, se definen funciones para graficar las imágenes junto con sus predicciones: `graficar_imagen` muestra la imagen real con la etiqueta predicha y la probabilidad correspondiente, destacando en azul si la predicción es correcta y en rojo si es incorrecta; `graficar_valor_arreglo` presenta un gráfico de barras con las probabilidades de cada clase, resaltando en rojo la clase predicha y en azul la clase real. Finalmente, se configura una cuadrícula de 5x5 para mostrar las imágenes y sus predicciones, proporcionando una visualización clara y comprensible de cómo el modelo clasifica cada imagen del conjunto de prueba, lo cual es útil para evaluar el rendimiento del modelo y entender sus aciertos y errores.

```
columns = 5
num_imagenes = filas * columns
plt.figure(figsize=(2*2*columns, 2*filas))
for i in range(num_imagenes):
    plt.subplot(filas, columns*2, 2*i+1)
    graficar_imagen(y_pred, y_test, X_test, i)
    plt.subplot(filas, columns*2, 2*i+2)
    graficar_valor_arreglo(y_pred, y_test, i)
plt.tight_layout()
plt.show()
```



En la visualización, se presenta una cuadrícula de 5x5 con un total de 25 imágenes del conjunto de prueba, cada una acompañada por un gráfico de barras que muestra las probabilidades asignadas a cada clase ('E', 'O', 'Σ', y 'θ').



Un aspecto destacado de esta visualización es su capacidad para proporcionar una evaluación clara y detallada del rendimiento del modelo. Las imágenes que han sido correctamente clasificadas están resaltadas con un texto azul, lo que facilita la identificación rápida de las predicciones acertadas. Los gráficos de barras asociados a cada imagen proporcionan una representación visual intuitiva de la confianza del modelo en cada predicción, con barras más altas indicando las clases con mayores probabilidades asignadas.

El uso de colores en la visualización para diferenciar las predicciones correctas (azul) de las incorrectas (rojo) es otro punto fuerte, ya que mejora significativamente la comprensión visual y permite una identificación inmediata de las áreas donde el modelo funciona bien y donde podría necesitar mejoras. En general, la visualización demuestra que el modelo asigna altas probabilidades a las clases correctas en la mayoría de los casos, lo que refleja un buen nivel de precisión y eficacia en la clasificación.