Imagen que contiene Interfaz de usuario gráfica

Descripción generada automáticamente

**VICEPRESIDENCIA DE AUDITORÍA INTERNA BANCOLOMBIA**

**Modelo OCR** (**Optical Character Recognition**)

**2023-06-10**

# Modelo OCR(Optical Character Recognition)

1. **IMPORTAR LIBRERIAS:**

import os

import numpy as np

import matplotlib.pyplot as plt

from pathlib import Path

from collections import Counter

import tensorflow as tf

from tensorflow import keras

from tensorflow.keras import layers

1. **CARGAR DATOS:** El dataset contiene 1498 archivos de captcha en formato .png ubicados en la ruta /ImagenesPrueba/Pruebas/. Que serán usados para el entrenamiento del modelo.

## LEER CARPETA DE IMAGNES

data\_dir = Path("C:/Users/jchinome/Documents/ImagenesPrueba/Pruebas/")

## TOMAR LA LISTA DE PAD DE IMAGENES

images = sorted(list(map(str, list(data\_dir.glob("\*.png")))))

## TOMAR LA LISTA DE ETIQUETAS

labels = [img.split(os.path.sep)[-1].split(".png")[0] for img in images]

## TOMAMOS LA LISTA DE CARACTERES CONTENIDAS EN EL TOTAL DE ETIQUETEAS

characters = set(char for label in labels for char in label)

characters = sorted(list(characters))

print("Number of images found: ", len(images))

print("Number of labels found: ", len(labels))

print("Number of unique characters: ", len(characters))

print("Characters present: ", characters)

## TAMAÑO DE LOTE(DIVICION DEL LOTE TOTAL ) ENTRENAMIENTO Y VALIDACION

## División del lote para un entrenamiento en paralelo.

batch\_size = 16

## DEFINIR DIMENCIONES DE LA IMAGEN

img\_width = 204

img\_height = 53

# Factor por el cual se reducirá la resolución de la imagen

# por los bloques convolucionales. estaremos usando dos

# bloques de convolución y cada bloque tendrá

# una capa de agrupación que reduce la muestra de las características por un factor de 2.

# Por lo tanto, el factor de reducción de muestreo total sería 4.

downsample\_factor = 4

## Longitud máxima de cualquier captcha en el conjunto de datos

max\_length = max([len(label) for label in labels])

1. **PREPROCESAMIENTO :**

# Mapear caracteres a números enteros

char\_to\_num = layers.StringLookup(

    vocabulary=list(characters), mask\_token=None

)

# Mapear enteros a los caracteres originales

num\_to\_char = layers.StringLookup(

    vocabulary=char\_to\_num.get\_vocabulary(), mask\_token=None, invert=True

)

La función **StringLookup** se utiliza comúnmente en el contexto de procesamiento de lenguaje natural (NLP) y tiene como objetivo mapear cada carácter de una secuencia de texto a un número único. Esto es útil para convertir texto en datos numéricos que pueden ser procesados por modelos de aprendizaje automático.

a**. vocabulary**: Es una lista que contiene los caracteres únicos que se encuentran en el texto de entrada. La función **list(characters)** crea esta lista a partir de la variable **characters**, que probablemente contiene todos los caracteres únicos presentes en el conjunto de datos de entrenamiento.

b. **mask\_token**: Es un token especial opcional que se utiliza para representar valores no conocidos o ausentes. En este caso, se establece en **None**, lo que significa que no se utiliza ningún token de máscara. Cualquier valor que no se encuentre dentro de la lista de carácter se tomara como cero, a menos que se aplique un token mascara.

c. **invert=True:** Esto indica que se desea realizar una conversión inversa, es decir, convertir números a caracteres en lugar de caracteres a números. Al establecer **invert=True**, la instancia **num\_to\_char** estará configurada para realizar esta conversión inversa.

def encode\_single\_sample(img\_path, label):

    # 1. Leer imagen

    img = tf.io.read\_file(img\_path)

    # 2. Comvertir a escala de grises

    img = tf.io.decode\_png(img, channels=1) #[Altura , Ancho , Canales ] channels = 1 es escala de grises

    # 3. Convertir a float32 in rango [0, 1].

    img = tf.image.convert\_image\_dtype(img, tf.float32)

    # 4. Mantener la forma de la imagen siempre.

    img = tf.image.resize(img, [img\_height, img\_width]) #[Altura= 53 , Ancho=204 , Canales=1 ]

    # 5. Transponemos la imagen, queremos que la dimensión de

# Tiempo corresponda a el ancho de la imagen

    img = tf.transpose(img, perm=[1, 0, 2]) #[Altura= 204 , Ancho=53 , Canales=1 ]

    # 6. Mapear los caracteres de las etiquetas a números

    label = char\_to\_num(tf.strings.unicode\_split(label, input\_encoding="UTF-8"))

    # 7. Retornar un diccionario con dos llaves, de esta forma entrara al modelo

    return {"image": img, "label": label}

El código que has proporcionado es una función llamada encode\_single\_sample que realiza la codificación de una única muestra de datos de imagen y etiqueta. A continuación, se explica cada paso de la función:

a**. img = tf.io.read\_file(img\_path):** Lee el archivo de imagen en **img\_path** utilizando la función **tf.io.read\_file** de TensorFlow y guarda la imagen en la variable **img.**

b**. img = tf.io.decode\_png(img, channels=1):** Decodifica la imagen en formato PNG utilizando **tf.io.decode\_png.** Se especifica **channels=1** para convertir la imagen a escala de grises. La imagen resultante se guarda en **img.**

c**. img = tf.image.convert\_image\_dtype(img, tf.float32):** Convierte la imagen a tipo de datos **tf.float32** y normaliza los valores en el rango [0, 1] utilizando **tf.image.convert\_image\_dtype**. Esto es importante para asegurarse de que los valores de píxeles estén en un rango adecuado para el procesamiento posterior.

d. **img = tf.image.resize(img, [img\_height, img\_width]):** Cambia el tamaño de la imagen a las dimensiones deseadas **[img\_height, img\_width]** utilizando **tf.image.resize**. En este caso, se redimensiona la imagen a una altura de **img\_height** y un ancho de **img\_width.**

e**. img = tf.transpose(img, perm=[1, 0, 2]):** Transpone la imagen utilizando **tf.transpose** para cambiar las dimensiones de la imagen. En este caso, se intercambian las dimensiones 0 y 1, de modo que la dimensión del tiempo corresponda al ancho de la imagen. El argumento **perm=[1, 0, 2]** especifica el nuevo orden de las dimensiones.

f. **label = char\_to\_num(tf.strings.unicode\_split(label, input\_encoding="UTF-8")):** Mapea los caracteres de la etiqueta a números utilizando el objeto **char\_to\_num.** Se utiliza **tf.strings.unicode\_split** para dividir la etiqueta en caracteres Unicode y luego se aplica la función **char\_to\_num** para obtener la representación numérica de los caracteres.

g. **return {"image": img, "label": label}:** Retorna un diccionario que contiene la imagen y la etiqueta codificadas. Esto se hace para que los datos de entrada puedan ser proporcionados al modelo de aprendizaje automático en el formato esperado. La clave "image" se utiliza para acceder a la imagen codificada y la clave "label" para acceder a la etiqueta codificada.

La razón detrás de esta transposición es que algunos modelos están diseñados para procesar datos secuenciales o de series temporales a lo largo del tiempo (ancho). Al cambiar la dimensión del tiempo al eje horizontal de la imagen, se asegura que el modelo pueda capturar las relaciones temporales adecuadas al procesar la imagen.

def split\_data(images, labels, train\_size=0.8, shuffle=True):

    # 1. Tomamos el tamaño total de los datos

    size = len(images)

    # 2. Tomamos los indices con el valor de tamaño de los datos.

    indices = np.arange(size)

    if shuffle:

        np.random.shuffle(indices)

    # 3. Tomamos el tamaño de los datos de entrenamiento

    train\_samples = int(size \* train\_size)

    # 4. Separamos los datos entre los datos de entrenamiento y validación

    x\_train, y\_train = images[indices[:train\_samples]], labels[indices[:train\_samples]]

    x\_valid, y\_valid = images[indices[train\_samples:]], labels[indices[train\_samples:]]

    return x\_train, x\_valid, y\_train, y\_valid

# Aplicamos la función para tomar nuestros datos de validación y entrenamiento

x\_train, x\_valid, y\_train, y\_valid = split\_data(np.array(images), np.array(labels))

La función **split\_data** toma un conjunto de imágenes y etiquetas y los divide en conjuntos de entrenamiento y validación.

a**. size= len(images):** Calcula el tamaño total del conjunto de imágenes. Esto se hace tomando la longitud de la lista **images**, que representa el número de imágenes en el conjunto de datos.

b. **indices = np.arange(size):** Crea un arreglo de índices que va desde **0** hasta **size - 1.** Estos índices se utilizan para realizar un muestreo aleatorio si se habilita la opción **shuffle.**

c**. train\_samples = int(size \* train\_size):** Calcula el número de muestras de entrenamiento utilizando el tamaño total del conjunto de datos (**size**) y el parámetro **train\_size**, que representa la proporción de datos que se utilizarán para entrenamiento. El valor de **train\_samples** se calcula multiplicando size por **train\_size** y luego redondeando hacia abajo al entero más cercano utilizando **int()**.

d. **x\_train, y\_train = images[indices[:train\_samples]], labels[indices[:train\_samples]]:** Divide las imágenes y las etiquetas en conjuntos de entrenamiento. Los primeros **train\_samples** elementos de los arreglos **images** y **labels**, seleccionados mediante **indices[:train\_samples],** se asignan a **x\_train** y **y\_train** respectivamente. Estos serán los conjuntos de datos utilizados para el entrenamiento del modelo.

e. **x\_valid, y\_valid = images[indices[train\_samples:]], labels[indices[train\_samples:]]:** Divide las imágenes y las etiquetas en conjuntos de validación. Los elementos restantes de los arreglos **images** y **labels**, seleccionados mediante **indices[train\_samples:],** se asignan a **x\_valid** y **y\_valid** respectivamente. Estos serán los conjuntos de datos utilizados para la validación del modelo.

f. Retorna los conjuntos de entrenamiento y validación: La función devuelve cuatro arreglos: **x\_train**, **x\_valid**, **y\_train** y **y\_valid**, que contienen las imágenes y las etiquetas divididas en conjuntos de entrenamiento y validación, respectivamente.

g. En el código posterior, se utiliza la función **split\_data** para dividir las imágenes y etiquetas almacenadas en los arreglos **images** y **labels**. Los conjuntos de entrenamiento y validación se asignan a **x\_train, x\_valid**, **y\_train** y **y\_valid**, respectivamente.

1. **CREAR OBJETO DE BASE DE DATOS:**

# DATOS DE ENTRENAMIENTO

# Convertimos los datos de entrenamiento a un objeto de tipo tf.data.dataset

# Mapeamos cada uno de nuestros datos pasándola por la función encode\_simple\_sample ademar de tomar los lotes de tamaño batch\_size = 16

train\_dataset = tf.data.Dataset.from\_tensor\_slices((x\_train, y\_train))

train\_dataset = (

    train\_dataset.map(

        encode\_single\_sample, num\_parallel\_calls=tf.data.AUTOTUNE

    )

    .batch(batch\_size)

    .prefetch(buffer\_size=tf.data.AUTOTUNE)

)

# DATOS DE VALIDACION

validation\_dataset = tf.data.Dataset.from\_tensor\_slices((x\_valid, y\_valid))

validation\_dataset = (

    validation\_dataset.map(

        encode\_single\_sample, num\_parallel\_calls=tf.data.AUTOTUNE

    )

    .batch(batch\_size)

    .prefetch(buffer\_size=tf.data.AUTOTUNE)

)

El código se encarga de crear un conjunto de datos (Dataset) en TensorFlow a partir de los datos de entrenamiento **(x\_train y y\_train).** A continuación, se explica cada paso del código:

a. **train\_dataset = tf.data.Dataset.from\_tensor\_slices((x\_train, y\_train)):** Crea un conjunto de datos a partir de tensores utilizando la función **from\_tensor\_slices().** Se pasa como argumento una tupla **(x\_train, y\_train)** que contiene los datos de entrada **x\_train** (imágenes) y los datos de salida **y\_train** (etiquetas).

b. **train\_dataset.map(encode\_single\_sample, num\_parallel\_calls=tf.data.AUTOTUNE):** Aplica una transformación a cada elemento del conjunto de datos utilizando la función **map().** En este caso, se utiliza la función **encode\_single\_sample** para codificar cada muestra individualmente. **encode\_single\_sample** es una función definida previamente que procesa y prepara una muestra de imagen y su etiqueta para el entrenamiento del modelo. El argumento **num\_parallel\_calls=tf.data.AUTOTUNE** permite que TensorFlow utilice automáticamente múltiples hilos para acelerar el procesamiento en paralelo.

c. **batch(batch\_size):** Agrupa las muestras en lotes utilizando el método **batch().** Se especifica el tamaño de lote deseado mediante el argumento **batch\_size**. Esto es útil para entrenar el modelo en mini lotes, lo que puede mejorar la eficiencia computacional y el rendimiento del entrenamiento.

d. **prefetch(buffer\_size=tf.data.AUTOTUNE): Prefetch (precarga)** los datos en el buffer de memoria para acelerar el rendimiento del entrenamiento. El argumento **buffer\_size=tf.data.AUTOTUNE** permite que TensorFlow ajuste automáticamente el tamaño del búfer según las capacidades del sistema.

1. **VISUALIZACION DE LOS DATOS**

# VISUALIZAMOS UN LOTE DE LOS DATOS DE ENTRENAMIENTO.

\_, ax = plt.subplots(4, 4, figsize=(10, 5))

for batch in train\_dataset.take(1):

    images = batch["image"]

    labels = batch["label"]

    for i in range(16):

        img = (images[i] \* 255).numpy().astype("uint8")

        label = tf.strings.reduce\_join(num\_to\_char(labels[i])).numpy().decode("utf-8")

        ax[i // 4, i % 4].imshow(img[:, :, 0].T, cmap="gray")

        ax[i // 4, i % 4].set\_title(label)

        ax[i // 4, i % 4].axis("off")

plt.show()



1. **MODELO:**

class CTCLayer(layers.Layer):

    def \_\_init\_\_(self, name=None):

        super().\_\_init\_\_(name=name)

        self.loss\_fn = keras.backend.ctc\_batch\_cost

    def call(self, y\_true, y\_pred):

        # Computa el valor de perdida y se agrega a

        # la capa usando `self.add\_loss()`.

        batch\_len = tf.cast(tf.shape(y\_true)[0], dtype="int64")

        input\_length = tf.cast(tf.shape(y\_pred)[1], dtype="int64")

        label\_length = tf.cast(tf.shape(y\_true)[1], dtype="int64")

        input\_length = input\_length \* tf.ones(shape=(batch\_len, 1), dtype="int64")

        label\_length = label\_length \* tf.ones(shape=(batch\_len, 1), dtype="int64")

        loss = self.loss\_fn(y\_true, y\_pred, input\_length, label\_length)

        self.add\_loss(loss)

        # Retorna la predicción

        return y\_pred

La función **CTCLayer** es una subclase de **layers.Layer** en TensorFlow. Esta clase se utiliza para calcular y agregar la pérdida CTC (Connectionist Temporal Classification) a un modelo de aprendizaje automático. A continuación, se explica cada parte de la función:

a**. \_\_init\_\_(self, name=None):** El método **\_\_init\_\_** es el constructor de la clase. Recibe un parámetro opcional **name** que se utiliza para asignar un nombre a la capa. En el constructor, se llama al constructor de la superclase **(layers.Layer)** utilizando **super().\_\_init\_\_(name=name).**

b. **self.loss\_fn = keras.backend.ctc\_batch\_cost:** Se inicializa el atributo **loss\_fn** con la función **keras.backend.ctc\_batch\_cost.** Esta función es proporcionada por Keras y se utiliza para calcular la pérdida CTC en el entrenamiento.

c. **call(self, y\_true, y\_pred):** El método **call** es donde se define la lógica de la capa. Recibe dos argumentos: **y\_true**, que representa las etiquetas verdaderas, y **y\_pred**, que representa las predicciones del modelo.

d. **batch\_len = tf.cast(tf.shape(y\_true)[0], dtype="int64"):** Se calcula la longitud del lote (batch length) como el tamaño del primer eje de **y\_true**. Se utiliza **tf.shape(y\_true)** para obtener las dimensiones de **y\_true**, y luego **tf.shape(y\_true)[0]** obtiene el tamaño del primer eje. **tf.cast()** se utiliza para convertir el resultado en un tipo de datos **int64**.

e. **input\_length = tf.cast(tf.shape(y\_pred)[1], dtype="int64"):** Se calcula la longitud de la entrada (input length) como el tamaño del segundo eje de **y\_pred**. Se utiliza **tf.shape(y\_pred)** para obtener las dimensiones de **y\_pred**, y luego **tf.shape(y\_pred)[1**] obtiene el tamaño del segundo eje.

f. **label\_length = tf.cast(tf.shape(y\_true)[1], dtype="int64"):** Se calcula la longitud de la etiqueta (label length) como el tamaño del segundo eje de **y\_true**.

g. **input\_length = input\_length \* tf.ones(shape=(batch\_len, 1), dtype="int64"):** Se crea un tensor con la misma forma que **input\_length,** pero con todos los elementos iguales a **input\_length.** Esto se hace multiplicando **input\_length** por un tensor de unos con forma **(batch\_len, 1).**

h. **label\_length = label\_length \* tf.ones(shape=(batch\_len, 1), dtype="int64"):** Similar al paso anterior, se crea un tensor con la misma forma que **label\_length**, pero con todos los elementos iguales a **label\_length.**

i. **loss = self.loss\_fn(y\_true, y\_pred, input\_length, label\_length):** Se calcula la pérdida CTC utilizando la función **loss\_fn.** Esta función toma como argumentos las etiquetas verdaderas **(y\_true),** las predicciones del modelo **(y\_pred),** las longitudes de entrada **(input\_length)** y las longitudes de etiqueta **(label\_length).**

j. **self.add\_loss(loss):** Se agrega la pérdida calculada a la capa utilizando **self.add\_loss().** Esto asegura que la pérdida se tenga en cuenta durante el entrenamiento del modelo.

Luego retorna el valor que se predijo.

**NOTA:**

¿Qué es perdida (loss) en un modelo?

En el contexto de un modelo de aprendizaje automático, la "pérdida" (también conocida como función de pérdida o función objetivo) es una medida que se utiliza para evaluar cuán bien está realizando el modelo en una tarea específica. Representa una medida de la discrepancia entre las predicciones del modelo y los valores reales (etiquetas) de los datos de entrenamiento.

**EJEMPLO:**

.Supongamos que **y\_true** es un tensor de forma **(100, 10),** lo que significa que tienes **100** muestras de datos verdaderos, cada una con una longitud de **10**. Entonces, **batch\_len** será igual a **100**.

.Supongamos que **y\_pred** es un tensor de forma **(50, 20, 30),** lo que significa que tienes **50** predicciones, cada una con una longitud de **20.** Entonces, **input\_length** será igual a **20.**

.Supongamos que **y\_true** es un tensor de forma **(100, 10),** lo que significa que tienes **100** muestras de datos verdaderos, cada una con una longitud de **10**. Entonces, **label\_length** será igual a **10**.

.Supongamos que batch\_len es igual a **100**, **input\_length** es igual a **20** y **label\_length** es igual a **10**. Después de ejecutar estas líneas de código, obtendrás tensores **input\_length** y **label\_length** de forma **(100, 1),** donde cada elemento será igual a **20** y **10** respectivamente.

def build\_model():

    # ENTRADA DE IMAGEN AL MODELO

    input\_img = layers.Input(

        shape=(img\_width, img\_height, 1), name="image",dtype="float32"

    )

    # ENTRADA DE ETIQUETA AL MODELO

    labels = layers.Input(name="label", shape=(None,),dtype="float32")

    # PRIMER BLOQUE CONVOLUCIONAL

    x = layers.Conv2D(

        32,

        (3, 3),

        activation = "relu",

        kernel\_initializer="he\_normal",

        padding="same",

        name="Conv1",

    )(input\_img)

    # REDUCE LA IMAGEN

    # Tomando los valores maximos de cada uno de los valores en el mapa de caracteristicas

    # (2,2) es el tamaño de la matriz que pasara por cada uno.

    x = layers.MaxPooling2D((2, 2), name="pool1")(x)

    # SEGUNDO BLOQUE CONVOLUCIONAL

    # 64 , la dimensionalidad del espacio de salida (es decir, el número de filtros de salida en la convolución) tambien podriamos llamarlo los canales .

    # (3,3) tamaño de la matriz de convolucion que pasa por la imagen.

    # kernel\_initializer="he\_normal" , inicia con pesos aleatorios números positivos o negativos que serán partes de la matriz de convolucion.

    x = layers.Conv2D(

        64,

        (3, 3),

        activation="relu",

        kernel\_initializer="he\_normal",

        padding="same",

        name="Conv2",

    )(x)

    # REDUCE LA IMAGEN

    # Tomando los valores maximos de cada uno de los valores en el mapa de caracteristicas

    # (2,2) es el tamaño de la matriz que pasara por cada uno.

    x = layers.MaxPooling2D((2, 2), name="pool2")(x)

    # Hemos usado dos max pool con tamaño de pool y strides 2.

    # Por lo tanto, los mapas de características re muestreados son 4 veces más pequeños. El número de

    # filtros en la última capa es 64. Remodelar en consecuencia antes de

    # pasar la salida a la parte RNN del modelo

    # FLATTEN

    # Reducir una red de 2 dimensiones(matriz) a una sola dimensión(vector) de tamaño 64.

    new\_shape = ((img\_width // 4), (img\_height // 4) \* 64)

    x = layers.Reshape(target\_shape=new\_shape, name="reshape")(x)

    x = layers.Dense(64, activation="relu", name="dense1")(x)

    x = layers.Dropout(0.2)(x)

    # RNNs

    # Entrenamiento bidireccional, la predicción resultante de la predicción en el tiempo t , será tomado en cuenta

    # pueden depender de la entrada en el tiempo t+1 , usado habitualmente para aplicaciones de traducción automática.

    # Dimensionalidad de la red aumenta aun vector lineal de tamaño (128)

    # Dimensionalidad de la red disminuye a un vector lineal de tamaño (64)

    x = layers.Bidirectional(layers.LSTM(128, return\_sequences=True, dropout=0.25))(x)

    x = layers.Bidirectional(layers.LSTM(64, return\_sequences=True, dropout=0.25))(x)

    # SALIDA DE LA RED

    # Se reduce la red a solo 5 salida que es la longitud de nuestro captcha

    x = layers.Dense(

        len(char\_to\_num.get\_vocabulary()) + 1, activation="softmax", name="dense2"

    )(x)

    # Agregar CTC layer para calcular la perdida CTC loss por cada paso.

    output = CTCLayer(name="ctc\_loss")(labels, x) # labels = entrenamiento y x = prediccion final

    # Define the model

    model = keras.models.Model(

        inputs=[input\_img, labels], outputs=output, name="ocr\_model\_v1"

    )

    # Optimizer

    opt = keras.optimizers.Adam()

    # Compile the model and return

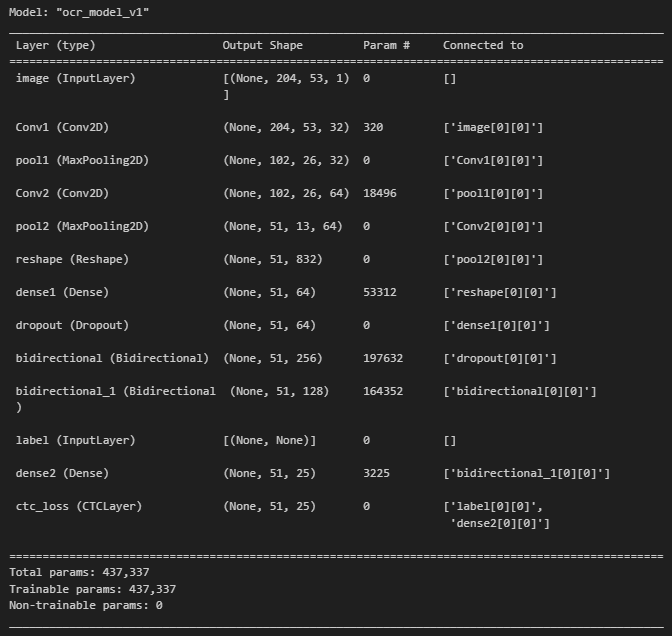
    model.compile(optimizer=opt)

    return model

# Get the model

model = build\_model()

model.summary()



La función **build\_model()** en Python define y construye un modelo de red neuronal para el reconocimiento óptico de caracteres **(OCR).** A continuación, se explica línea por línea el funcionamiento de esta función:

a. Se define la entrada de la imagen al modelo utilizando la **capa layers.Input().** La forma de la imagen se especifica como **(img\_width, img\_height, 1)** para indicar el ancho, alto y el canal de la imagen (escala de grises). Se le asigna el nombre **"image"** y el tipo de datos **"float32".**

b.Se define la entrada de la etiqueta al modelo utilizando la capa **layers.Input().** La forma de la etiqueta se especifica como **(None,),** lo que indica que la longitud de la etiqueta puede variar. Se le asigna el nombre "label" y el tipo de datos **"float32".**

c.Se define el primer bloque convolucional utilizando la capa **layers.Conv2D().** Se especifica el número de filtros como **32**, el tamaño de la matriz de convolución como **(3, 3)**, la función de activación como **"relu",** el inicializador de pesos como **"he\_normal"** y el tipo de relleno como **"same".** El nombre de la capa se establece como **"Conv1".** Esta capa realiza la convolución de la imagen de entrada.

d. Se reduce la imagen utilizando la capa **layers.MaxPooling2D().** Se especifica el tamaño de agrupación como **(2, 2),** lo que reduce la dimensión espacial de la imagen tomando el valor máximo en cada región de agrupación. El nombre de la capa se establece como **"pool1".**

e. Se define el segundo bloque convolucional de manera similar al primer bloque. Se utiliza una capa **layers.Conv2D()** con **64** filtros, tamaño de matriz de convolución **(3, 3),** función de activación **"relu",** inicializador de pesos **"he\_normal"** y tipo de relleno **"same".** El nombre de la capa se establece como **"Conv2".**

f. Se reduce la imagen nuevamente utilizando otra capa **layers.MaxPooling2D()** con tamaño de agrupación **(2, 2).** Esto reduce aún más la dimensión espacial de la imagen. El nombre de la capa se establece como **"pool2".**

g. Se realiza una operación de aplanamiento de la salida de la capa anterior utilizando la capa **layers.Reshape().** Se especifica el nuevo tamaño de la forma utilizando **target\_shape** calculado en base al tamaño original de la imagen. El nombre de la capa se establece como **"reshape".**

h. Se agrega una capa densa **(layers.Dense())** con 64 unidades de neuronas y función de activación **"relu".** Esto proporciona una transformación lineal a los datos a una dimensión más baja. También se aplica una capa de dropout **(layers.Dropout())** con una tasa de dropout del **20%** para regularizar el modelo y evitar el sobreajuste.

i. Se añaden dos capas bidireccionales de **LSTM (layers.Bidirectional(layers.LSTM())).** Estas capas permiten que las predicciones en el tiempo t dependan de la entrada en el tiempo **t+1**, lo cual es útil en aplicaciones como la traducción automática. La primera capa tiene **128** unidades y la segunda capa tiene **64** unidades. Estas capas **RNN** permiten capturar patrones secuenciales en los datos.

j. Se aplica una capa densa final con una función de activación **softmax** para generar la salida final del modelo. El número de unidades en esta capa es igual a la longitud del vocabulario de caracteres más 1 (debido a la clase de espacio en blanco utilizada en el CTC).

k. Se agrega una capa **CTCLayer** personalizada para calcular y agregar la pérdida CTC al modelo. Esta capa toma como entrada las etiquetas y las predicciones y calcula la pérdida CTC.

l. **Compilación del modelo:** Se utiliza el optimizador Adam y se compila el modelo.

m. **Resumen del modelo:** Se muestra un resumen del modelo que muestra la estructura y el número de parámetros del modelo.

1. **ENTRENAMIENTO:**

epochs = 100

early\_stopping\_patience = 10

# Añadir parada anticipada

early\_stopping = keras.callbacks.EarlyStopping(

    monitor="val\_loss", patience=early\_stopping\_patience, restore\_best\_weights=True

)

# Entrenar el modelo

history = model.fit(

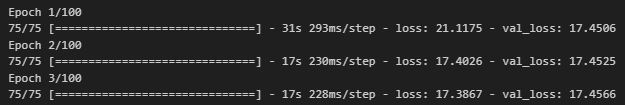
    train\_dataset,

    validation\_data=validation\_dataset,

    epochs=epochs,

    callbacks=[early\_stopping],

)



Se encarga de entrenar el modelo de red neuronal utilizando los conjuntos de datos de entrenamiento y validación. A continuación, se explica cada parte del código:

a. **epochs:** Se define el número de epochs, es decir, la cantidad de veces que el modelo recorrerá todo el conjunto de datos de entrenamiento durante el entrenamiento.

b**. early\_stopping\_patience:** Se define la paciencia del criterio de detención temprana. Es el número de epochs que el modelo esperará sin mejorar en la métrica de pérdida de validación antes de detener el entrenamiento.

c. **early\_stopping:** Se crea una instancia de la clase EarlyStopping de Keras, que se utilizará como callback durante el entrenamiento del modelo. Esta callback monitoriza la métrica de pérdida de validación y detiene el entrenamiento si no mejora después de un número determinado de epochs definido por la patience.

d. **model.fit():** Esta función se encarga de entrenar el modelo. Recibe los siguientes argumentos:

* **train\_dataset:** El conjunto de datos de entrenamiento.
* **validation\_data:** El conjunto de datos de validación.
* **epochs:** El número de epochs definido anteriormente.
* **callbacks:** Una lista de callbacks, en este caso, solo se utiliza el callback de detención temprana definido anteriormente.

El entrenamiento del modelo se realiza a través de la llamada **a model.fit(**), donde el modelo se ajusta a los datos de entrenamiento y se evalúa en los datos de validación en cada **epoch**. El entrenamiento se detendrá si la pérdida de validación no mejora después de la cantidad de epocas definida por **early\_stopping\_patience**, y se restaurarán los pesos del modelo correspondientes a la mejor métrica de validación obtenida. El historial del entrenamiento se guarda en la **variable history**, que contiene información sobre la pérdida y las métricas en cada **epoch.**

En resumen, este código configura el entrenamiento del modelo con un criterio de detención temprana y luego entrena el modelo utilizando los conjuntos de datos de entrenamiento y validación.

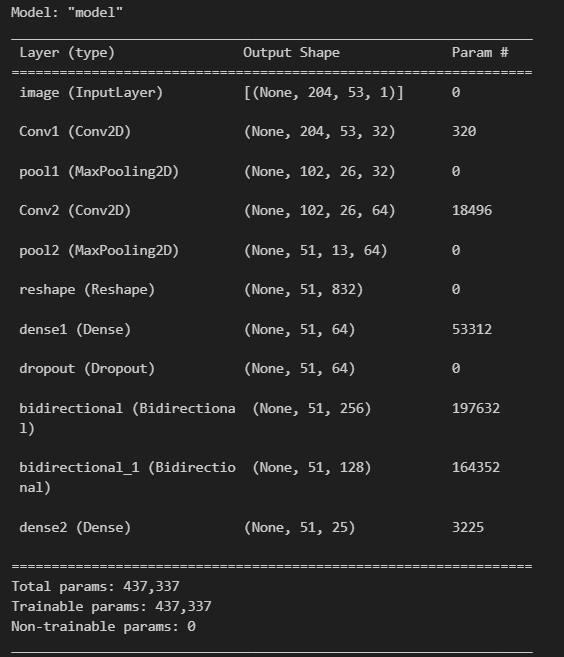
1. **PREDICCION:**

prediction\_model = keras.models.Model(

    model.get\_layer(name="image").input, model.get\_layer(name="dense2").output

)

prediction\_model.summary()



Se crea una nueva instancia de modelo llamada **prediction\_model** utilizando la clase **keras.models.Model.** Este nuevo modelo se construye utilizando las capas del modelo original **(model)** que se seleccionan mediante sus nombres.

a. **model.get\_layer(name="image").input:** Aquí se obtiene la capa de entrada del modelo original **(model**) con el nombre "**image**". Esta capa de entrada representa la imagen de entrada al modelo.

b. **model.get\_layer(name="dense2").output:** Aquí se obtiene la capa de salida del modelo original (model) con el nombre "dense2". Esta capa de salida representa las predicciones finales del modelo, que es una salida de clasificación con activación **softmax.**

Luego, se utiliza **keras.models.Model** para crear un nuevo modelo llamado **prediction\_model** que toma la capa de entrada de model y la capa de salida de model. Este nuevo modelo se utiliza para hacer predicciones sobre nuevas imágenes.

c. Finalmente, la línea **prediction\_model.summary()** muestra un resumen del nuevo modelo prediction\_model, que proporciona información sobre la arquitectura de las capas, la forma de entrada y salida, y el número de parámetros entrenables del modelo.

En resumen, la línea de código crea un nuevo modelo **(prediction\_model)** que toma una imagen de entrada y produce las predicciones finales utilizando las capas relevantes del modelo original (model). Esto permite utilizar prediction\_model para hacer predicciones en imágenes sin necesidad de entrenar todo el modelo original.

def decode\_batch\_predictions(pred):

    input\_len = np.ones(pred.shape[0]) \* pred.shape[1]

    results = keras.backend.ctc\_decode(pred, input\_length=input\_len, greedy=True)[0][0][

        :, :max\_length

    ]

    output\_text = []

    for res in results:

        res = tf.strings.reduce\_join(num\_to\_char(res)).numpy().decode("utf-8")

        output\_text.append(res)

    return output\_text

Esta función se utiliza para decodificar las predicciones realizadas por un modelo de reconocimiento óptico de caracteres (OCR) en forma de lotes.

a. **pred:** Es el tensor de predicciones realizado por el modelo de **OCR**. Es un tensor de forma **(batch\_size, time\_steps, num\_classes),** donde **batch\_size** es el tamaño del lote, **time\_steps** es el número de pasos de tiempo en la secuencia y **num\_classes** es el número de clases de caracteres.

b. **input\_len:** Es un arreglo **NumPy** de forma **(batch\_size)** que representa la longitud de entrada para cada secuencia en el lote. En este caso, se crea un arreglo con todos los elementos iguales a la longitud de las secuencias de entrada **(pred.shape[1]).**

c. **results:** Utilizando la función **keras.backend.ctc\_decode**, se decodifican las predicciones realizadas por el modelo. Se utiliza el algoritmo de búsqueda ávida **(greedy search)** para decodificar las secuencias de caracteres. La variable **max\_length** representa la longitud máxima permitida para las secuencias decodificadas.

d. **output\_text:** Es una lista que almacenará las secuencias de caracteres decodificadas.

e. En el **bucle for**, se itera sobre cada secuencia decodificada en results. Se utiliza la función **tf.strings.reduce\_join** para concatenar los números de caracteres (representados por el tensor res) en una única cadena de texto. Luego, se utiliza **numpy().decode("utf-8")** para convertir el tensor a una cadena de texto en formato **UTF-8**.

f. Finalmente, se agrega la secuencia de texto decodificada a la lista **output\_text.**

En resumen, la función **decode\_batch\_predictions** toma las predicciones realizadas por un modelo **OCR** y las decodifica en secuencias de texto legibles. Utiliza la decodificación **CTC (Connectionist Temporal Classification)** y la búsqueda ávida para obtener las secuencias más probables. La función devuelve una lista de secuencias de texto decodificadas correspondientes a cada predicción en el lote de entrada.

1. **VISUALIZACION DE LOS DATOS DE VALIDACION(PREDICCION) :**

#  Mostrar resultados

for batch in validation\_dataset.take(2):

    batch\_images = batch["image"]

    batch\_labels = batch["label"]

    preds = prediction\_model.predict(batch\_images)

    pred\_textsv = decode\_batch\_predictions(preds)

    orig\_texts\_v = []

    for label in batch\_labels:

        label = tf.strings.reduce\_join(num\_to\_char(label)).numpy().decode("utf-8")

        orig\_texts\_v.append(label)

    \_, ax = plt.subplots(4, 4, figsize=(15, 5))

    for i in range(len(pred\_textsv)):

        img = (batch\_images[i, :, :, 0] \* 255).numpy().astype(np.uint8)

        img = img.T

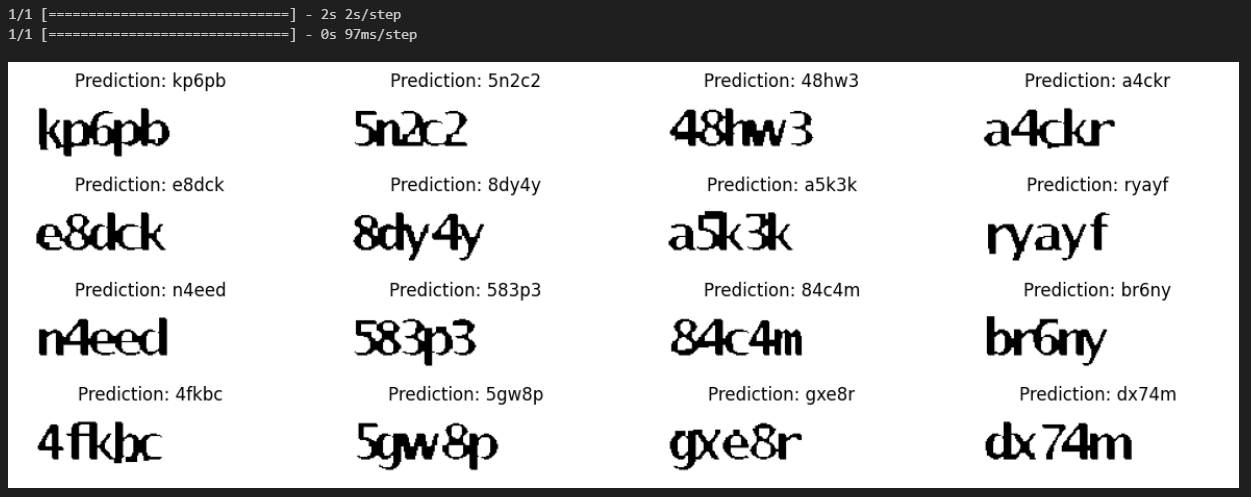
        title = f"Prediction: {pred\_textsv[i]}"

        ax[i // 4, i % 4].imshow(img, cmap="gray")

        ax[i // 4, i % 4].set\_title(title)

        ax[i // 4, i % 4].axis("off")

plt.show()





1. **ACCURACY :**

# Accuracy

from sklearn.metrics import accuracy\_score

accuracy\_score(orig\_texts\_v, pred\_textsv)



Calcula la precisión (accuracy) de las predicciones realizadas por un modelo en comparación con las etiquetas originales.

1. **GUARDAR MODELO:**

model.save(r'C:\Users\jchinome\OneDrive - Grupo Bancolombia\Auditoria\Automatizacion Querys\Capcha\Modelo\path\_to\_my\_model2.h5')