Aldo Hernández
Abraham López
Damián García
Cristian Antonio

ENTREGABLE 4 - PIA

Abstract *Este documento presenta un análisis de las mejoras realizadas a un modelo de Reconocimiento Óptico de Caracteres (OCR) diseñado para transcribir texto de imágenes. El modelo inicial mostró un rendimiento deficiente con valores de pérdida estancados, lo que motivó una revisión integral. Las modificaciones clave incluyeron: mejora en la carga de datos usando pandas y TensorFlow para un mejor control; vectorización de etiquetas mejorada con tokens de máscara y padding adecuados; creación de una clase personalizada CTCOCRModel con implementación optimizada de pérdida CTC; ajustes arquitectónicos para prevenir problemas de longitud de secuencia; refinamientos a la métrica CWERMetric usando tf.nn.ctc_greedy_decoder; y un mejor cálculo del número de clases utilizando el tamaño del vocabulario de caracteres. Estos cambios resultaron en mejoras significativas del rendimiento, reduciendo la pérdida de validación de 727.3027 a 5.26555, aunque la Tasa de Error de Caracteres (CER) permanece en 1.2292, indicando margen para mejoras adicionales. El documento incluye métricas de rendimiento, gráficos de pérdida y ejemplos de predicciones que demuestran el progreso, reconociendo a la vez la necesidad de mejoras adicionales en trabajos subsecuentes.*

Keywords python, OCR, CNN, TensorFlow, CER

1. Introducción

En este entregable se analizarán los diferentes cambios realizados al modelo de reconocimiento óptico de caracteres (OCR, por sus siglas en inglés), cuyo propósito principal es transcribir texto presente en imágenes de manera precisa y eficiente. Este tipo de modelo es fundamental en aplicaciones como la digitalización de documentos, lectura automatizada de matrículas vehiculares, o extracción de datos de formularios escaneados. Debido al muy bajo rendimiento que ofreció nuestro primer modelo a como esperábamos que funcionara, nos motivó a la revisión y mejora de distintos componentes del mismo.

El objetivo principal de los cambios implementados ha sido reducir la pérdida total del modelo durante el entrenamiento, ya que anteriormente se mantenía en un valor fijo, esto se traduce en una mayor precisión en el reconocimiento de caracteres. Para lograr esto, se abordaron distintos aspectos tanto del preprocesamiento de los datos como de la arquitectura del modelo y el tratamiento de las etiquetas. El análisis presentado en este reporte se enfocará en evaluar el impacto de cada uno de estos cambios en el desempeño global del sistema.

Cada uno de estos cambios será analizado en detalle en las siguientes secciones del entregable, donde se presentarán métricas comparativas entre el modelo antiguo y la versión actualizada. Se incluirán gráficos de pérdida y precisión, además, a partir de estos resultados se discutirá el impacto de las modificaciones implementadas y se propondrán posibles mejoras futuras.

2. Ajustes principales

Para comenzar, cambiamos la manera en cómo se cargan y procesan los datos. Anteriormente se usaba directamente la función *make_csv_dataset* del framework TensorFlow para cargar el archivo csv, pero ahora se optó por hacerlo de una forma más controlada. Primero se leen los metadatos del archivo csv (como nombres de archivo y etiquetas) con *pandas.read_csv* y posteriormente se crea nuestro dataset *tf.data.Dataset* con la función *from_tensor_slices* con la finalidad de tener una mayor flexibilidad y control sobre los datos antes de introducirlos al flujo de TensorFlow, esto además permitió el cálculo de cosas como la longitud máxima de etiquetas y el vocabulario de etiquetas de manera más directa sin hacer que TensorFlow se encargue de todo, lo que alentaba todo el proceso.

El siguiente cambio importante fue el cómo se vectorizaban las etiquetas y cómo se les hacía el *padding*. En el modelo anterior el vectorizador se creaba sin un *mask_token* y se usaba el padding con 99, esto causaba que este número podría entrar en conflicto con los índices reales de los caracteres y por consecuencia se afectaba a la función de pérdida **CTC** que es muy sensible a este tipo de detalles. Ahora el vectorizador *char_to_num_global* se inicializa con un *mask_token* igual a una cadena vacía y un *oov_token* con valor *"[UNK]"* para hacerlo más robusto frente a vacíos o caracteres no conocidos. Para el *padding* se utilizó un valor de -1 ya que es más seguro al estar

fuera del rango válido de caracteres. Aun más, nos cercioramos de que las etiquetas y el *padding* sean del tipo *tf.int32* para que la función *ctc_loss* funcione como se desea.

Al momento de entrenar, antes se utilizaba *keras.Sequential* junto con *model.fit* y la **pérdida CTC** de Keras, esto no dejaba mucho margen para configurar otros detalles importantes; es por esto que se creó la clase **CTCOCRModel** que hereda de *keras.Model* y se personalizaron los métodos *train_step* y *test_step*. Se hizo este cambio de clase porque buscamos llamar directamente a *tf.nn.ctc_loss* para definir parámetros concretos como *blank_index* con un valor de -1 porque este número indica qué índice representa el carácter en blanco que necesita la función CTC y si no está bien configurado, el modelo simplemente no aprenderá.

En cuanto a la arquitectura, un ajuste esencial fue en la parte final de la red neuronal, concretamente en cómo se calcula la longitud de la salida de la CNN para la CTC. Antes, la capa de *Reshape* usaba un *img_width // 4* pero esto a veces hacía que la secuencia de predicción fuera más corta que la etiqueta real, lo cual rompía la función de **pérdida CTC**. Para corregir este problema, cambiamos una de las capas de *MaxPooling2D* para que hiciera un menor *downsampling* en la dimensión del ancho junto con *img_width // 2* para la longitud de salida. Además, se colocó una verificación explícita que lanza un error si alguna etiqueta es más larga que dicha salida para asegurarnos de que todo esté bien antes de entrenar.

Sobre las métricas, se hicieron ajustes a la métrica **CWERMetric** ya que antes usaba *keras.ops.nn.ctc_decode* y se hacía un slicing de las predicciones. Esto generaba que a veces se truncaban los resultados si no estaban bien alineados con la longitud real, en cambio, ahora se usa directamente *tf.nn.ctc_greedy_decoder* para una mayor estabilidad. A su vez se mejoró el manejo de SparseTensor y sus tipos de datos para que todo sea compatible con las etiquetas reales.

Finalmente, se cambió cómo se calcula el número total de clases representado por *num_classes* para la salida del modelo. Antes solamente se usaba la longitud del *set* de caracteres mas uno para el caracter en blanco, pero ahora se utiliza *char_to_num_global.vocabulary_size()* $+1$. Esto ha demostrado ser mejor porque ahora *StringLookup* puede añadir tokens especiales como lo son el *mask_token* o el *oov_token* y sin ellos el modelo podría tener un desajuste entre las etiquetas, la salida de la red y la pérdida.

3. Evaluación del modelo

Se obtuvieron resultados significativamente superiores al utilizar el modelo actual en comparación con el modelo anterior, como se muestra en la tabla 1. Esta mejora en el rendimiento puede atribuirse a varios factores, entre ellos una arquitectura optimizada, un mejor ajuste de los hiperparámetros y un entrenamiento más prolongado. Tal como era de esperarse, al incrementar tanto el número de épocas como el tamaño del *batch*, el modelo fue capaz de aprender de manera más eficiente a partir de los datos, lo que se tradujo en un desempeño sustancialmente mejor.

El incremento en las épocas permitió al modelo realizar un mayor número de iteraciones sobre el conjunto de datos, facilitando una convergencia más estable y precisa. Estos diferentes factores contribuyeron a una mejora global en los resultados obtenidos.

No obstante, estos hallazgos se desarrollarán con mayor profundidad en la siguiente sección, donde se presentarán y analizarán en detalle las métricas específicas y los gráficos que evidencian la evolución del rendimiento del modelo a lo largo del entrenamiento.

Table 1
Comparación entre modelos

Modelo	Anterior	Actual
Epochs	100	100
Learning rate	0.0010	0.0010
Batch size	1024	1024
val_loss	727.3027	5.26555
CER	N/A	1.2292

4. Visualización

Recordando los resultados del primer modelo, a pesar de un entrenamiento con ??? épocas se obtuvo una pérdida (*val_loss*) casi constante de 727 a lo largo de todo el proceso. Podemos ver el impacto que generaron los ajustes que realizamos durante esta semana en las figuras 1 y 2. Si bien hemos reducido notablemente la pérdida (de 727 a 7), el *Character Error Rate* **CER** sigue siendo bastante alto y es por eso que el valor no se muestra en la figura 1.

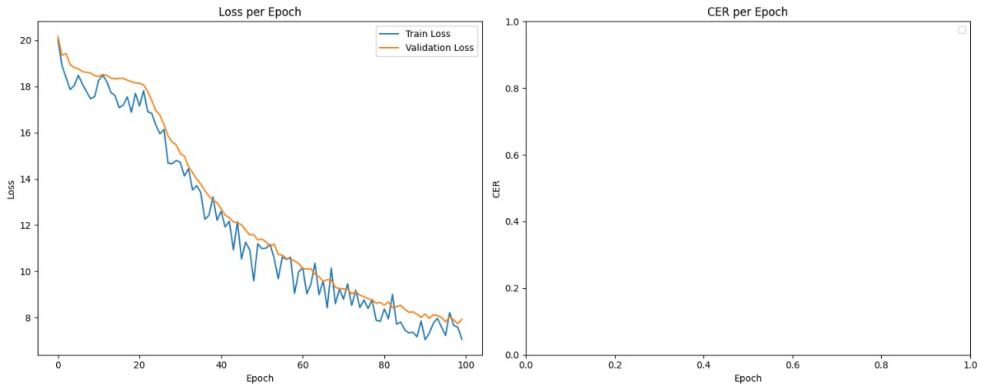

Figure 1. Pérdida por época y CER por época.

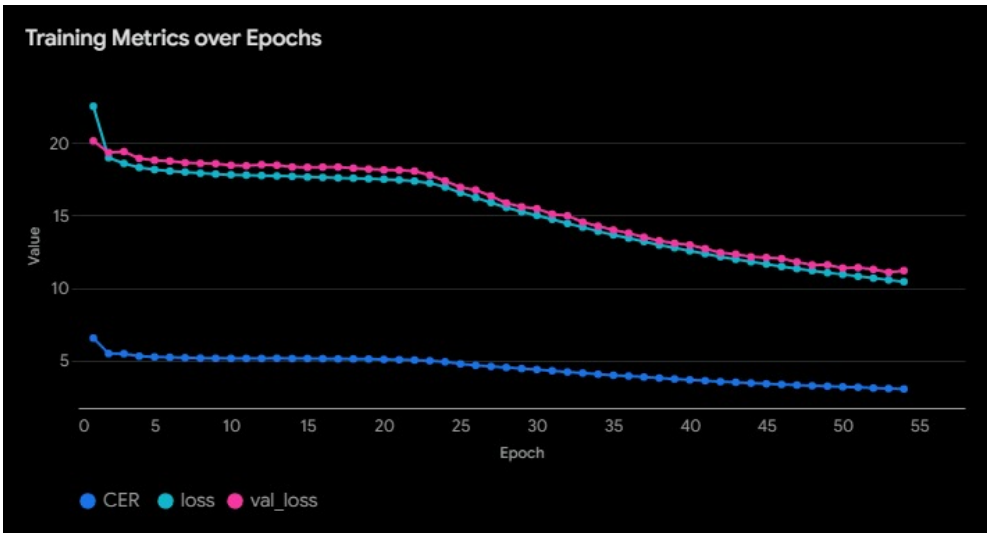

Figure 2. Valor de métricas sobre épocas.

Ya por último podemos ver algunas de las predicciones que hizo nuestro modelo en la figura 3. Observamos que el modelo todavía está lejos de tener predicciones acertadas pero con los cambios que realizaremos para la siguiente semana esperamos mejorar eso.

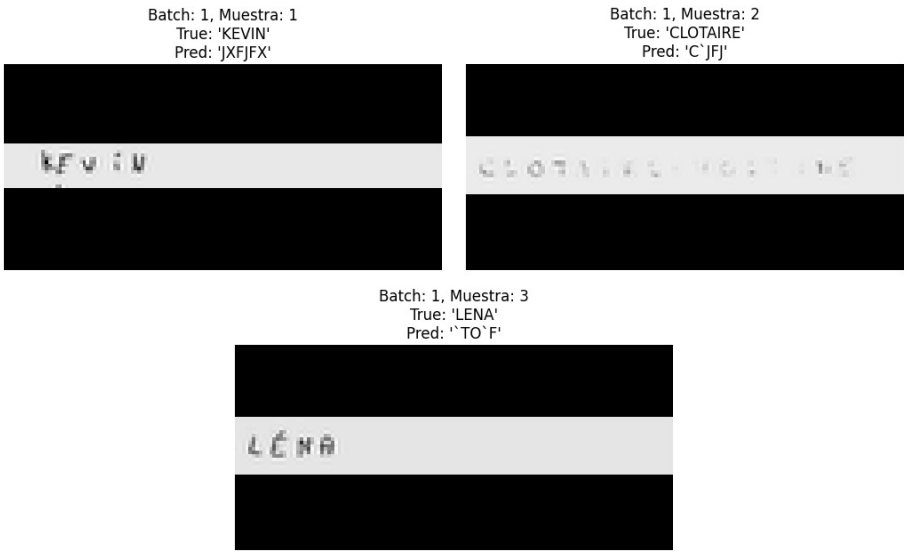

Figure 3. Predicción de KEVIN, CLOTAIRE y LENA, respectivamente.

Affiliations

Aldo Hernández
Universidad Autónoma de Nuevo León, San Nicolás de los Garza, aldo.hernandezt@uanl.edu.mx

Abraham López
Universidad Autónoma de Nuevo León, San Nicolás de los Garza, abraham.lopezg@uanl.edu.mx

Damián García
Universidad Autónoma de Nuevo León, San Nicolás de los Garza, gilberto.garciam@uanl.edu.mx

Cristian Antonio
Universidad Autónoma de Nuevo León, San Nicolás de los Garza, cristian.antoniosnt@uanl.edu.mx

Received: ???
Revised: ???
Accepted: ???