Clasificador de Dígitos Propios con Python

Electiva II

Docente a cargo

Cristhian Alejandro Cañar Muñoz

Presentado por

Ulises ortega revelo

Universidad autónoma del cauca-2025

Resumen
Este informe presenta el desarrollo de una red neuronal recurrente (RNN) simple en TensorFlow/Keras, diseñada para predecir la siguiente letra en una secuencia de texto usando la palabra "hola" como ejemplo. Se implementaron técnicas básicas de preprocesamiento, entrenamiento con one-hot encoding y se optimizó el modelo mediante el uso de una capa de entrada explícita y la técnica de EarlyStopping. El modelo demostró su capacidad para aprender relaciones secuenciales simples entre caracteres.

I. INTRODUCCIÓN

En este proyecto se desarrolla una red neuronal recurrente (RNN) básica utilizando TensorFlow y Keras, con el objetivo de predecir la siguiente letra en una secuencia de texto. Se emplea la palabra "hola" como conjunto de datos para que el modelo aprenda las relaciones secuenciales entre sus caracteres. Este ejercicio permite explorar los fundamentos del procesamiento secuencial en redes neuronales, aplicando técnicas como la codificación one-hot, activaciones tanh y softmax, y estrategias de entrenamiento como EarlyStopping para optimizar el rendimiento del modelo. La implementación sirve como una introducción práctica al uso de RNNs para tareas de predicción de texto.

II. Marco teórico

Las redes neuronales recurrentes (RNN, por sus siglas en inglés) son un tipo de arquitectura de red neuronal especializada en procesar datos secuenciales, como texto, audio o series temporales. A diferencia de las redes neuronales tradicionales, las RNN poseen conexiones recurrentes que permiten mantener información de estados anteriores, lo que las hace adecuadas para modelar dependencias temporales.

En el procesamiento de texto, las RNN pueden aprender patrones de secuencia entre caracteres o palabras. Para facilitar este aprendizaje, los datos de texto suelen ser transformados mediante one-hot encoding**,** una técnica que convierte cada carácter en un vector binario con una única posición activa. Esto permite al modelo interpretar los símbolos como entradas numéricas sin imponer un orden implícito.

Dentro de las RNN, una de las implementaciones más sencillas es la capa SimpleRNN, que procesa secuencias paso a paso, acumulando información de manera temporal. Comúnmente, se utilizan funciones de activación como tanh para las neuronas recurrentes y softmax en la capa de salida, especialmente en tareas de clasificación.

El proceso de entrenamiento ajusta los pesos del modelo mediante algoritmos como el descenso por gradiente, optimizado con técnicas como Adam. Para evitar el sobreentrenamiento y mejorar la eficiencia, se puede aplicar EarlyStopping, una técnica que detiene el entrenamiento cuando la métrica de desempeño deja de mejorar después de varias iteraciones.

Este proyecto se enfoca en la aplicación de estos conceptos mediante el uso de TensorFlow y Keras, entrenando una red RNN para predecir la siguiente letra en una secuencia simple, lo cual representa una introducción efectiva al modelado de secuencias en inteligencia artificial.

TensorFlow

Es una biblioteca de código abierto desarrollada por Google para la creación y entrenamiento de modelos de machine

learning y deep learning. TensorFlow permite definir redes neuronales de forma flexible, ejecutarlas en CPU o GPU, y gestionar eficientemente operaciones matemáticas complejas como derivadas, matrices y tensores. Se usa tanto en investigación como en producción.

- **Keras**

Keras es una API de alto nivel incluida dentro de TensorFlow (a partir de TensorFlow 2.x) que facilita la construcción de modelos de redes neuronales de forma más intuitiva y rápida. Permite definir modelos secuenciales o con arquitecturas más complejas con pocas líneas de código. Es ideal para principiantes y también útil para prototipado rápido.

- **Codificación One-Hot**

Es una técnica usada para representar datos categóricos, como letras o palabras, en vectores binarios. Cada elemento único se convierte en un vector donde una sola posición es 1 (indicando su presencia) y el resto son 0.

Ejemplo:

Si el alfabeto es ['a', 'b', 'c'], entonces:

'a' → [1, 0, 0]

'b' → [0, 1, 0]

'c' → [0, 0, 1]

Se usa para que las redes neuronales no interpreten relaciones numéricas inexistentes entre categorías (por ejemplo, que 'b' es mayor que 'a').

- **tanh (Tangente Hiperbólica)**

Es una función de activación usada en redes neuronales, especialmente en capas ocultas. Su salida está en el rango de -1 a 1, lo que permite que la red capture relaciones tanto positivas como negativas. Es útil en RNNs porque ayuda a mantener un equilibrio en los valores propagados entre pasos temporales.

Ventaja sobre ReLU o Sigmoid: No tiene un sesgo positivo, y su centro es 0, lo que mejora la estabilidad del entrenamiento.

- **softmax**

Es una función de activación usada típicamente en la capa de salida de modelos de clasificación multiclase. Transforma un vector de valores (logits) en una distribución de probabilidades, es decir, cada salida estará entre 0 y 1 y la suma será 1.

Ejemplo:

si el modelo predice [2.0, 1.0, 0.1], softmax los transforma en [0.65, 0.24, 0.11], indicando la probabilidad de cada clase.

- **EarlyStopping**

Es una técnica de regularización que detiene automáticamente el entrenamiento del modelo cuando la métrica (como la pérdida) deja de mejorar durante cierto número de épocas consecutivas. Esto previene el sobre entrenamiento y ahorra tiempo de cómputo.

Ejemplo: Si después de 10 épocas la pérdida no mejora, el entrenamiento se detiene y se conservan los mejores pesos obtenidos.

III. Arquitectura

La red neuronal recurrente (RNN) utilizada en este proyecto fue diseñada para predecir la siguiente letra en una secuencia de caracteres, utilizando la palabra "hola" como conjunto de datos de entrada. La arquitectura de la red es relativamente simple y se organiza en tres capas principales: una capa de entrada, una capa recurrente y una capa de salida.

1. Capa de Entrada

La capa de entrada es un componente fundamental que define la forma de los datos que se alimentarán al modelo. En este caso, la entrada consiste en una secuencia de longitud 1 (un solo carácter), que se representa mediante one-hot encoding. El one-hot encoding transforma cada carácter de la

secuencia en un vector binario, donde cada posible carácter se representa como una única posición activa en el vector. La dimensión de la entrada corresponde a la longitud del conjunto de caracteres posibles (en este caso, 4: 'h', 'o', 'l', 'a').

2. Capa Recurrente: SimpleRNN

La capa recurrente utilizada en este modelo es una SimpleRNN, que es una forma básica de red neuronal recurrente. Esta capa consta de 8 neuronas y utiliza la función de activación tanh (tangente hiperbólica). La función tanh se utiliza para transformar las salidas de las neuronas a valores entre -1 y 1, lo que ayuda a regular las activaciones y evitar problemas de explosión o desaparición de gradientes durante el entrenamiento.

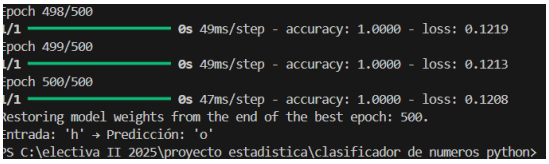
La principal función de esta capa es procesar la secuencia de entrada de manera temporal, es decir, la capa mantiene una memoria del estado anterior de la secuencia mientras procesa cada nuevo carácter. Esto permite a la red aprender dependencias temporales entre las letras y predecir de manera más precisa cuál es el siguiente carácter en la secuencia.

3. Capa de Salida: Dense

La capa de salida está compuesta por una capa Dense, que es completamente conectada. Esta capa tiene una cantidad de neuronas igual al número de caracteres posibles en la secuencia, en este caso, 4 (correspondientes a 'h', 'o', 'l', 'a'). La función de activación utilizada en esta capa es softmax, que convierte las salidas de la capa en una distribución de probabilidad, con valores entre 0 y 1, cuya suma total es igual a 1.

La capa de salida produce una probabilidad para cada uno de los caracteres posibles, y la predicción final del modelo será el carácter con la probabilidad más alta. De esta forma, el modelo es capaz de predecir cuál será el siguiente carácter en la secuencia, basándose en el contexto aprendido de las letras anteriores.

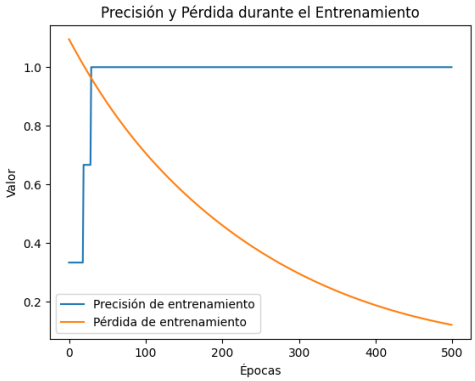
IV. PRUEBAS



1prueba con la letra h

Se realiza la primera prueba con la letra “h” con 500 epochs y como se puede observar la predicción es correcta arrojando la letra “o”

Tenemos la siguiente grafica



2grafica de precisión y perdida de letra h

La siguiente gráfica ilustra la dinámica de la precisión y la pérdida del modelo durante las 500 épocas de entrenamiento. Se observa una clara tendencia en ambas métricas a medida que el modelo itera sobre los datos de entrenamiento.

La precisión de entrenamiento (línea azul) exhibe un incremento significativo en las etapas iniciales del entrenamiento. Partiendo de un valor bajo, experimenta un aumento abrupto, estabilizándose posteriormente en un valor cercano a 1.0. Este comportamiento sugiere una rápida convergencia del modelo hacia la correcta clasificación de los datos de entrenamiento, indicando una eficiente identificación de los patrones relevantes en el conjunto de datos utilizado para el aprendizaje. La posterior estabilidad en un

nivel alto de precisión señala que el modelo ha alcanzado un punto donde las mejoras en la clasificación de los datos de entrenamiento son mínimas.

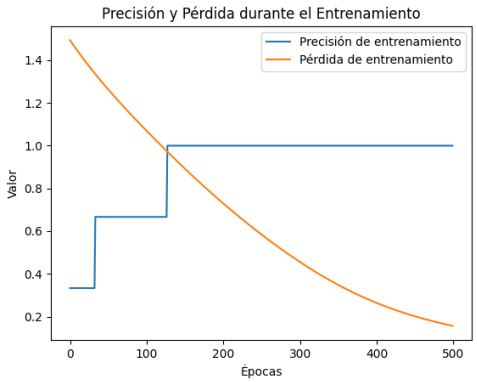
En contraste, la pérdida de entrenamiento (línea naranja) muestra una disminución gradual y continua a lo largo de las épocas. Iniciando en un valor elevado, la pérdida decrece de manera suave, lo que refleja la mejora progresiva del modelo en la reducción del error entre sus predicciones y los valores reales en los datos de entrenamiento. La tendencia descendente de la pérdida es un indicador positivo del aprendizaje del modelo, sugiriendo que está ajustando sus parámetros internos para minimizar las discrepancias en sus salidas. Al final del periodo de entrenamiento, la pérdida se ha reducido considerablemente, aunque sin alcanzar un valor de cero.

```
Epoch 498/500
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 50ms/step - accuracy: 1.0000 - loss: 0.1583
Epoch 499/500
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 49ms/step - accuracy: 1.0000 - loss: 0.1575
Epoch 500/500
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 46ms/step - accuracy: 1.0000 - loss: 0.1568
Restoring model weights from the end of the best epoch: 500.
Entrada: 'o' → Predicción: 'l'
PS C:\electiva II 2025\proyecto estadistica\clasificador de numeros python>
```

3prueba con la letra o

Se realiza con la siguiente letra correspondiente que es la “o” y nos debería arrojar la letra “l” lo cual podemos observar que el resultado la predicción es correcta

Obtenemos la siguiente grafica



4 precisión y perdida letra o

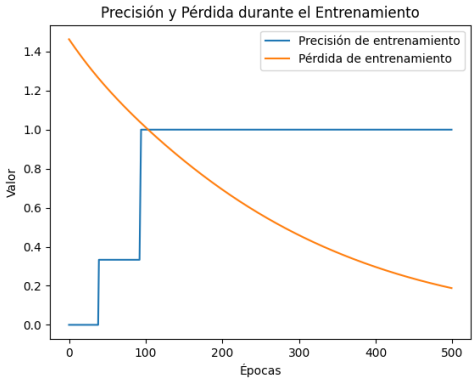
La gráfica muestra la precisión y la pérdida del modelo durante 500 épocas. La precisión de entrenamiento (azul) aumenta en etapas hasta estabilizarse cerca de 1.0 tras unas 150 épocas, indicando un aprendizaje efectivo por fases. La pérdida de entrenamiento (naranja) disminuye continuamente, reflejando una reducción constante del error. En conjunto, las tendencias sugieren un aprendizaje exitoso sobre los datos de entrenamiento.

```
Epoch 498/500
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 49ms/step - accuracy: 1.0000 - loss: 0.1900
Epoch 499/500
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 50ms/step - accuracy: 1.0000 - loss: 0.1891
Epoch 500/500
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 46ms/step - accuracy: 1.0000 - loss: 0.1882
Restoring model weights from the end of the best epoch: 500.
Entrada: 'l' → Predicción: 'a'
PS C:\electiva_II_2025\proyecto_estadistica\clasificador_de_numeros_python>
```

5 prueba con la letra L

Se realiza con la siguiente letra correspondiente que es la “L” y nos debería arrojar la letra “a” lo cual podemos observar que el resultado la predicción es correcta

Obtenemos la siguiente grafica

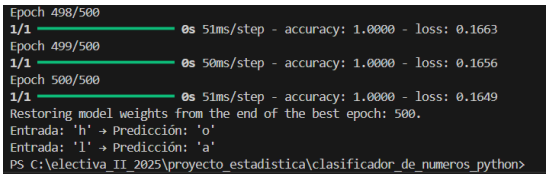


6 precisión y perdida letra L

La gráfica ilustra la precisión y la pérdida durante el entrenamiento del modelo a lo largo de 500 épocas. La precisión de entrenamiento (azul) muestra aumentos por etapas hasta estabilizarse en un valor alto cerca de la época 100, indicando un aprendizaje efectivo en fases. La pérdida de entrenamiento (naranja) disminuye continuamente, reflejando una reducción constante del error. Estas tendencias

sugieren un aprendizaje exitoso sobre los datos de entrenamiento.

Otras pruebas



V. Conclusiones

Los patrones observados en las gráficas sugieren que los modelos lograron aprender los patrones presentes en los datos de entrenamiento, evidenciado por la alta precisión final y la baja pérdida. Sin embargo, es crucial recordar que el rendimiento sobre los datos de entrenamiento no garantiza un rendimiento igual de bueno en datos nuevos o no vistos. Para una evaluación completa de la eficacia de estos modelos, sería imprescindible analizar su desempeño en conjuntos de datos de validación o prueba independientes, lo que permitiría determinar su capacidad de generalización y evitar el sobreajuste.

Este proyecto ha demostrado la efectividad de las Redes Neuronales Recurrentes (RNN), específicamente mediante una arquitectura SimpleRNN, para el procesamiento y aprendizaje de datos secuenciales como el texto. La capacidad inherente de las RNN para mantener un estado interno y modelar dependencias temporales se reveló crucial para la correcta predicción de la secuencia de letras en la palabra "hola".

A pesar de la simplicidad de la arquitectura, compuesta por una capa SimpleRNN con función de activación tanh y una capa de salida Dense con softmax, el modelo logró aprender la secuencia objetivo con éxito. Este resultado subraya el potencial de las RNN, incluso con recursos computacionales y datos limitados, para capturar relaciones secuenciales fundamentales.

La implementación de la codificación one-hot se erigió como un paso esencial para la representación de los caracteres de entrada. Esta técnica permitió transformar los elementos categóricos del vocabulario en un formato numérico procesable por la red, al tiempo que evitó la introducción de sesgos numéricos espurios entre las distintas letras.

La estrategia de EarlyStopping demostró ser una herramienta valiosa para la optimización del proceso de entrenamiento. Al monitorear el rendimiento del modelo y detener el entrenamiento ante la ausencia de mejoras significativas, se previno el sobreentrenamiento, se optimizó el tiempo de cómputo y se aseguró la retención de los pesos del modelo correspondientes a su mejor desempeño.

Este ejercicio práctico resalta la accesibilidad y el poder de las bibliotecas TensorFlow y Keras para el diseño, implementación y entrenamiento de modelos de aprendizaje profundo, incluso en tareas introductorias de predicción secuencial. La claridad y la eficiencia de estas herramientas facilitan la experimentación y el desarrollo de soluciones basadas en redes neuronales.

En última instancia, el aprendizaje exitoso de una secuencia corta como "hola" se establece como un bloque de construcción fundamental para abordar desafíos más complejos dentro del campo del Procesamiento del Lenguaje Natural (NLP) y el análisis de series temporales. Este proyecto sienta las bases para la comprensión y aplicación de RNN en tareas más sofisticadas que involucran secuencias de mayor longitud y dependencias temporales más intrincadas.

VI. CÓDIGO

```
import numpy as np

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import SimpleRNN, Dense, Input

from tensorflow.keras.callbacks import EarlyStopping

from tensorflow.keras.utils import to_categorical
```

```
import matplotlib.pyplot as plt
import tensorflow as tf

# Verificación de GPU
print("GPU disponible:", tf.config.list_physical_devices('GPU'))

# === Datos ===
texto = "hola"
caracteres = sorted(list(set(texto)))  # ['a', 'h', 'l', 'o']
char_to_index = {c: i for i, c in enumerate(caracteres)}
index_to_char = {i: c for c, i in char_to_index.items()}

# One-hot encoding de entrada y salida
X = []
y = []

for i in range(len(texto) - 1):
    entrada = texto[i]
    salida = texto[i + 1]

    # One-hot para entrada y salida
                X.append(to_categorical(char_to_index[entrada], num_classes=len(caracteres)))
                y.append(to_categorical(char_to_index[salida], num_classes=len(caracteres)))

# Convertir a arrays y ajustar forma
X = np.array(X)
y = np.array(y)

# RNN espera [batch, timesteps, features], usamos timesteps=1
X = X.reshape((X.shape[0], 1, X.shape[1]))

# === Modelo ===
model = Sequential([
    Input(shape=(1, len(caracteres))),  # Primera capa con tamaño de entrada explícito
    SimpleRNN(8, activation='tanh'),
    Dense(len(caracteres), activation='softmax')
])

model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

#  EarlyStopping
early_stopping = EarlyStopping(monitor='loss', patience=10, verbose=1, restore_best_weights=True)

#  Entrenamiento
history = model.fit(X, y, epochs=500, verbose=1, callbacks=[early_stopping])

#  Graficar precisión y pérdida
plt.plot(history.history['accuracy'], label='Precisión de entrenamiento')
plt.plot(history.history['loss'], label='Pérdida de entrenamiento')
plt.title('Precisión y Pérdida durante el Entrenamiento')
plt.xlabel('Épocas')
plt.ylabel('Valor')
plt.legend()
plt.show()

# Prueba del modelo
def predecir_siguiente(letra):
    if letra not in char_to_index:
        return "Letra desconocida"

    x_input = to_categorical(char_to_index[letra], num_classes=len(caracteres)).reshape((1, 1, len(caracteres)))
    pred = model.predict(x_input, verbose=0)
    pred_index = np.argmax(pred)
    return index_to_char[pred_index]

# Ejemplo:
for letra in "hl":
    print(f"Entrada: '{letra}' → Predicción: '{predecir_siguiente(letra)}'")
```