CORPORACION UNIVERSITARIA AUTÒNOMA DEL CAUCA

RNN SECUENCIAS DE CARACTERES

ENTREGABLE Nº7 TALLER

CRISTIAN CAMILO MARTINEZ CORDOBA

ELECTIVA II

INGENIERÌA ELECTRONICA

MAYO 2025

DESCRIPCION

Con el objetivo de ilustrar el funcionamiento básico de una red neuronal recurrente (RNN), se desarrolló un modelo simple en TensorFlow/Keras para predecir la siguiente letra en una secuencia de texto. En este caso, se utilizó la palabra "hola" como conjunto de datos de entrenamiento, estableciendo relaciones directas entre letras consecutivas. Cada letra fue representada mediante codificación one-hot, y la red se entrenó para aprender estas asociaciones utilizando una arquitectura sencilla compuesta por una capa SimpleRNN con 8 neuronas y una capa de salida Dense con activación softmax. El modelo fue entrenado durante 500 épocas, alcanzando una alta precisión en la tarea, demostrando así la capacidad de una RNN para modelar secuencias cortas de texto.

ARQUITECTURA DE LA RNN.

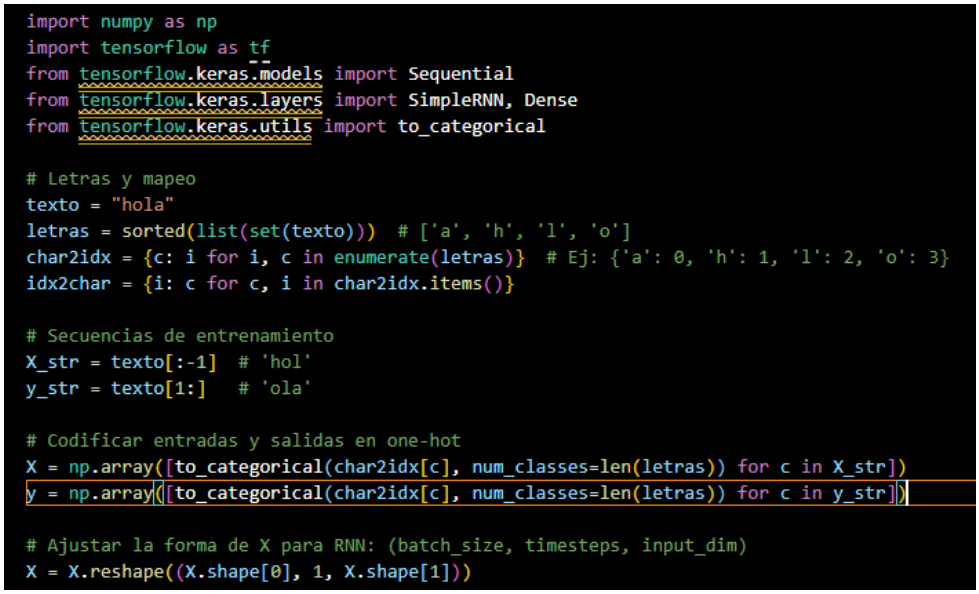
La red neuronal recurrente (RNN) diseñada para este ejercicio tiene como objetivo predecir la siguiente letra en una secuencia corta, utilizando la palabra "hola" como conjunto de datos. El modelo recibe como entrada letras individuales codificadas mediante one-hot encoding, lo que permite representar cada carácter como un vector binario de tamaño fijo. En este caso, como el vocabulario consta de cuatro letras únicas (h, o, l, a), cada vector de entrada tiene una dimensión de 4. La red está configurada para procesar una letra a la vez, es decir, una secuencia con un único paso de tiempo (timesteps = 1).

La arquitectura del modelo está compuesta por dos capas principales. La primera es una capa SimpleRNN con 8 unidades y función de activación tanh. Esta capa es responsable de capturar la relación secuencial entre los caracteres, almacenando un estado oculto que permite aprender dependencias temporales. Aunque el modelo procesa una letra a la vez, la estructura recurrente permite que esta capa mantenga un pequeño historial de contexto que influye en la predicción.

La salida de la capa recurrente se conecta a una capa Dense (totalmente conectada) con 4 neuronas, correspondientes a las posibles letras de salida. Esta capa utiliza una función de activación softmax, que convierte el vector de salida en una distribución de probabilidad sobre las letras posibles, facilitando así la predicción del siguiente carácter en la secuencia. En total, el modelo contiene 140 parámetros entrenables: 104 pertenecientes a la capa recurrente y 36 a la capa densa de salida.

IMPLEMENTACION DEL CODIGO.

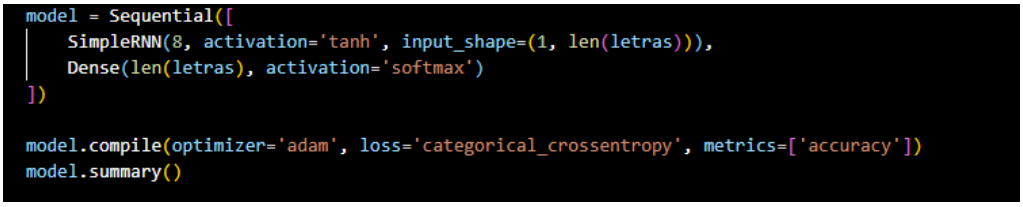
Las siguientes líneas de código se encargan de preparar los datos de entrada y salida para el entrenamiento de la red neuronal recurrente. Primero, se importan las librerías necesarias: numpy para operaciones numéricas, tensorflow y los módulos de Keras que permiten construir y entrenar el modelo. Luego, se define el texto base, que en este caso es la palabra "hola", y se extraen las letras únicas del texto, ordenándolas alfabéticamente para formar un vocabulario. A partir de este vocabulario, se crean dos diccionarios: uno que asigna a cada letra un índice numérico y otro que permite hacer la conversión inversa. Posteriormente, se generan las secuencias de entrada y de salida esperada tomando pares consecutivos de letras en el texto. Estas letras se convierten en vectores one-hot utilizando la función to_categorical, de modo que cada letra se represente como un vector binario del tamaño del vocabulario. Finalmente, se ajusta la forma del array X para que cumpla con el formato requerido por la RNN, que espera entradas tridimensionales con las dimensiones: tamaño del lote, número de pasos de tiempo, y tamaño del vector de entrada.



```python
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import SimpleRNN, Dense
from tensorflow.keras.utils import to_categorical

# Letras y mapeo
texto = "hola"
letras = sorted(list(set(texto)))  # ['a', 'h', 'l', 'o']
char2idx = {c: i for i, c in enumerate(letras)}  # Ej: {'a': 0, 'h': 1, 'l': 2, 'o': 3}
idx2char = {i: c for c, i in char2idx.items()}

# Secuencias de entrenamiento
X_str = texto[:-1]  # 'hol'
y_str = texto[1:]   # 'ola'

# Codificar entradas y salidas en one-hot
X = np.array([to_categorical(char2idx[c], num_classes=len(letras)) for c in X_str])
y = np.array([to_categorical(char2idx[c], num_classes=len(letras)) for c in y_str])

# Ajustar la forma de X para RNN: (batch_size, timesteps, input_dim)
X = X.reshape((X.shape[0], 1, X.shape[1]))
```

Fig 1. Código Implementado Preparación de Datos.

En el siguiente fragmento de código se construye y compila el modelo de red neuronal recurrente utilizando la API Sequential de Keras. El modelo está compuesto por dos capas principales. La primera es una capa SimpleRNN con 8 neuronas, que recibe como entrada secuencias de un solo paso temporal, donde cada letra está representada mediante un vector de dimensión igual al tamaño del vocabulario. Esta capa es responsable de procesar la información secuencial y generar un vector de estado oculto. La segunda capa es una capa Dense totalmente conectada con una cantidad de neuronas igual al número de letras distintas, y utiliza una función de activación, que permite obtener una distribución de probabilidad sobre las posibles letras siguientes. Una vez definido el modelo, se compila utilizando el optimizador. Además, se incluye la métrica de precisión para evaluar el rendimiento durante el entrenamiento. Finalmente, imprime un resumen de la arquitectura del modelo, incluyendo la cantidad de parámetros entrenables.



```python
model = Sequential([
    SimpleRNN(8, activation='tanh', input_shape=(1, len(letras))),
    Dense(len(letras), activation='softmax')
])

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

Fig 2. Código Implementado Compilación del Modelo.

La siguiente línea de código se encarga de entrenar el modelo previamente definido utilizando los datos de entrada y las salidas esperadas. El entrenamiento se realiza durante 500 épocas, lo que significa que el modelo verá todo el conjunto de datos 500 veces, lo cual es adecuado para que aprenda correctamente la relación secuencial entre las letras, dado que el conjunto de datos es muy pequeño. Durante este proceso, el modelo ajusta sus pesos internos minimizando la funciónde pérdida mediante el optimizador adam, con el objetivo de mejorar la precisión en la predicción de la siguiente letra de la secuencia.

```
model.fit(X, y, epochs=500, verbose=0)
```

Fig 3. Código Implementado Entrenamiento del Modelo.

El último bloque de código define y utiliza una función para probar el modelo entrenado, prediciendo la siguiente letra en la secuencia a partir de una letra de entrada. La función toma como argumento una letra inicial, la convierte a su representación one-hot y luego ajusta su forma para que sea compatible con la entrada que espera el modelo (es decir, un array tridimensional con dimensiones correspondientes a batch size, timestep y dimensión del vector de entrada). A continuación, el modelo realiza la predicción y devuelve un vector de probabilidades, del cual se extrae el índice con mayor valor mediante, lo que indica la letra más probable. Este índice se convierte nuevamente en una letra usando el diccionario. Finalmente, se utiliza un bucle for para probar la función con las letras h, o y l, imprimiendo la predicción del modelo para la siguiente letra correspondiente a cada una. Esta parte del código permite verificar si la red ha aprendido correctamente la secuencia de la palabra "hola".

```
def predecir_siguiente_letra(letra_inicial):
    x = to_categorical(char2idx[letra_inicial], num_classes=len(letras))
    x = x.reshape((1, 1, len(letras)))
    pred = model.predict(x, verbose=0)
    idx_pred = np.argmax(pred)
    return idx2char[idx_pred]

# Probar con cada letra de la secuencia
for letra in 'hol':
    siguiente = predecir_siguiente_letra(letra)
    print(f"{letra} → {siguiente}")
```

Fig 4. Código Implementado Prueba del Modelo.

RESULTADOS.

```
def predecir_siguiente_letra(letra_inicial):
    x = to_categorical(char2idx[letra_inicial], num_classes=len(letras))
    x = x.reshape((1, 1, len(letras)))
    pred = model.predict(x, verbose=0)
    idx_pred = np.argmax(pred)
    return idx2char[idx_pred]

# Probar con cada letra de la secuencia
for letra in 'hol':
    siguiente = predecir_siguiente_letra(letra)
    print(f"{letra} → {siguiente}")
```

✓ 0.6s

```
h → o
o → l
l → a
```

Fig 5. Resultado del Codigo.

El resultado obtenido de la imagen anterior muestra que, al proporcionar las letras h, o y l, como entrada, la red neuronal predice correctamente las letras siguientes respectivamente. Esto indica que el modelo ha aprendido de forma efectiva la relación secuencial entre los caracteres de la palabra "hola". A pesar de tratarse de un conjunto de datos extremadamente pequeño y de una red muy simple, la arquitectura basada en una capa SimpleRNN con pocas neuronas fue suficiente para capturar la lógica de la secuencia. Este resultado valida el funcionamiento del modelo, demostrando que las redes neuronales recurrentes, incluso con configuraciones básicas, son capaces de modelar dependencias temporales en secuencias cortas, lo cual es fundamental en tareas de procesamiento de lenguaje natural.

REFERENCIAS.

https://gamma.app/docs/Ele-II-Cap-3-Topicos-AvanzadosCNN-43ngl1jf9kinxns?mode=doc