Aplicaciones de la Arquitectura Transformer al Pronóstico y Procesamiento de Lenguaje Natural

Diego Iván Perea Montealegre, Carlos Ivan Osorio Moreno, Gabriel Jeannot Viaña, Samir Hassan Ordóñez

Facultad de Ingeniería, Universidad Autónoma de Occidente

Cali, Colombia

—Resumen : En este taller, se lleva a cabo la creación y evaluación de modelos auto-regresivos utilizando arquitecturas de redes recurrentes y transformers. Durante el taller, se realizarán diversas actividades relacionadas con el Procesamiento del Lenguaje Natural (PLN), tales como la predicción de series temporales, la clasificación de textos, la transcripción y la traducción. El enfoque principal será explorar y aplicar estos modelos en tareas específicas de procesamiento de lenguaje natural, con el objetivo de comprender su desempeño y eficacia en diferentes contextos.

—Summary: In this workshop, the creation and evaluation of auto-regressive models is carried out using recurrent network architectures and transformers. During the workshop, various activities related to Natural Language Processing (NLP) will be carried out, such as time series prediction, text classification, transcription and translation. The main focus will be to explore and apply these models in specific natural language processing tasks, with the aim of understanding their performance and effectiveness in different contexts.

INTRODUCCIÓN

Procesamiento del Lenguaje Natural (PLN): El Procesamiento del Lenguaje Natural es un campo de la inteligencia artificial que se centra en la interacción entre las computadoras y el lenguaje humano. Su objetivo es permitir que las máquinas comprendan, interpreten y generen texto de manera similar a como lo hacen los humanos. Las aplicaciones de PLN son diversas e incluyen traducción automática, análisis de sentimientos, resumen de texto, chatbots, entre otros.

MARCO TEÓRICO

Redes Neuronales Recurrentes (RNN): Las Redes Neuronales Recurrentes son un tipo de arquitectura de redes neuronales diseñada para procesar secuencias de datos, como texto o series temporales. A diferencia de las redes neuronales tradicionales, las RNN tienen conexiones que forman ciclos, permitiendo a la red retener información sobre eventos anteriores. Esto las hace efectivas en tareas donde la secuencia y el contexto son importantes, como en el procesamiento de lenguaje natural. Sin embargo, las RNN tienen limitaciones, como la dificultad para manejar dependencias a largo plazo. Esto llevó al desarrollo de arquitecturas más avanzadas, como los Transformers. Transformers: Los Transformers son una arquitectura de red neuronal introducida en el campo del PLN por su capacidad para manejar eficientemente secuencias largas de datos y capturar relaciones a largo plazo. A diferencia de las RNN, los Transformers no dependen de conexiones recurrentes, sino que utilizan mecanismos de atención para procesar simultáneamente todas las partes de la secuencia. Esto los hace altamente paralelizables y eficaces en una amplia gama de tareas de PLN. Los Transformers han revolucionado el campo del PLN y son la base de muchos modelos de vanguardia, como BERT, GPT y otros, que han demostrado un rendimiento excepcional en tareas como la comprensión del lenguaje, generación de texto y traducción automática.

I. APLICACIÓN DE CLASIFICACIÓN DE MOVIMIENTO CON DEEP LEARNING (RNN Y TRANSFORMERS)

Este apartado se realiza como una continuación a un primer trabajo presentado en la materia, dentro del cual se pretende estudiar y analizar la misma situación pero con diferentes técnicas de Deep learning, por lo tanto, algunas de las

explicaciones y procedimientos se extraerán del anterior trabajo y se replicarán en este.

Seguidamente, se documentará todo el procedimiento, objetivos y resultados del entrenamiento de tres modelos de deep learning con el objetivo de realizar una clasificación de movimiento.

La finalidad académica de esta sección se enfoca en realizar una comparación entre tres diferentes modelos de deep learning con el objetivo de realizar una clasificación con datos secuenciales de movimiento obtenidos del giroscopio de un smartphone.

Los modelos específicos a evaluar dentro de este contexto son: Modelo RNN, modelo Transformers con capa de normalización y modelo Transformers sin normalización. Todos con el mismo preprocesamiento, diferenciándose únicamente en la modificación de su arquitectura de entrenamiento.

Como propósito específico del contexto, el proyecto se realiza con el fin de solventar una situación muy común entre las personas, la cual es el deficiente lavado de dientes [1] [2] [3].

La solución propuesta se basa en un clasificador de movimiento que mediante reconocimiento de patrones determinara de manera general que tan adecuado es el lavado de dientes de las personas. Al ser un proyecto netamente académico se centra en la realización del modelo clasificador con DL, pero posteriormente se puede aplicar a la práctica mediante la incorporación de un sensor específico en instrumentos como el cepillo dental a través de tecnologías como internet de las cosas (IoT).

La clasificación se realizará mediante la detección de 4 categorías, las cuales detectarán si la persona está cepillando correctamente las caras laterales de sus dientes, la parte superior de los molares, o si por el contrario está cepillando su dentadura de manera errónea, así mismo detectara cuando no esté lavando sus dientes. Por tanto, la clasificación se reduce al movimiento que tiene el cepillo en función de la localización y parte del diente. Para mayor comprensión de la anatomía del diente se presenta una imagen con sus partes exteriores principales [4].

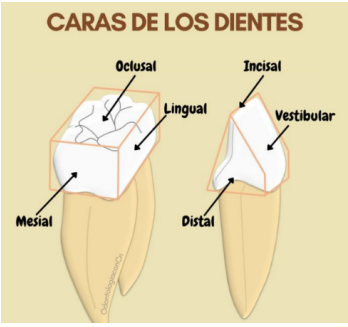


Fig. 22. Partes exteriores del diente.

Como antecedentes de solución, no se encontró alguna que coincida de manera exacta con el mismo enfoque, pero dentro de muchos proyectos de DL en el sector de la salud, se encontró una solución de detección de placa dental en las personas mediante visión computacional (CV) [5].

Para el desarrollo del proyecto, fueron de gran importancia los códigos Python del profesor Jesús A. López, compartidos como parte de la materia 'Procesamiento para datos secuenciales' [6].

A continuación, se presenta de manera general el funcionamiento del proyecto mediante un diagrama de bloques.

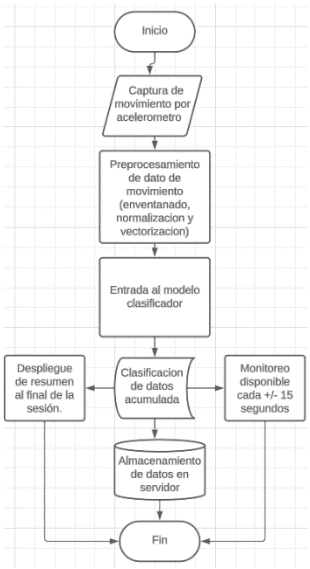


Fig. 23. Diagrama de bloques de clasificación de movimiento.

A nivel técnico, para la realización del proyecto y el entrenamiento del mismo fue requerido un giroscopio el cual se usó desde un celular mediante la plataforma de Edge Impulse [7], la cual sirvió como base para la recopilación de los datos de movimiento capturados de manera manual mediante la repetición de las diferentes acciones a clasificar (lavado frontal, superior, incorrecto e inactividad). Así mismo un computador con conexión a internet para ingresar a la plataforma Google Colab [8] fue requerido.

Cabe resaltar que los datos tomados para entrenar el modelo se basaron en la recopilación de datos en un solo contexto, por lo tanto, el reto presentado en este apartado se basa en el caso que se desee llevar el modelo a otros contextos y usuarios, este deberá tener la posibilidad de ajustarse mediante fine tuning [9] para su óptimo funcionamiento y realizar una clasificación precisa en base a los sonidos captados.

Los tres modelos entrenados difieren, como se mencionó anteriormente, en la composición de la arquitectura característica de cada modelo según su enfoque (RNN o Transformers), sin embargo, el procesamiento de los datos

secuenciales es igual en todos, por tanto, se generalizará esta parte del proceso.

A Proceso general [6] para realizar el entrenamiento de los modelos

1. *Conexión e importación de librerías en Google Colab*

```
from google.colab import drive
drive.mount('/content/gdrive')
```

```
Mounted at /content/gdrive
```

```
# Se cargan las dependencias necesarias
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.utils import to_categorical

from keras.models import Sequential
from keras.layers import Dense, Conv1D, Flatten, MaxPooling1D

import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import matplotlib as mpl
```

Fig. 24. Importación de librerías y conexión a Drive.

2. *Unión de valores de todos los archivos según su canal y graficación*

```
    Datos[i*600:(i+1)*600] = DatosOriNP[0:600]
    i+=1
print(Datos.shape)
```

```
(54000, 3)
```

Fig. 25. Agrupación de datos en 3 arreglos.

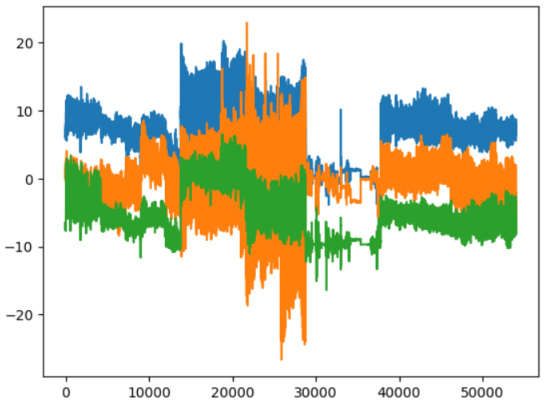


Fig. 26. Visualización de datos en 3 canales.

3. *Se generan los datos de etiquetado (matrices de unos y ceros según categoría)*

```
YtrainIni=np.zeros((90,1))
for i in range(23):
    YtrainIni[i]=0 #front correct

for i in range(23,48):
    YtrainIni[i]=1 # incorrect

for i in range(48,63):
    YtrainIni[i]=2 # no wash

for i in range(63,90):
    YtrainIni[i]=3 # top correct
print(YtrainIni)
```

```
[1.]
[1.]
[1.]
[1.]
[1.]
```

Fig. 27. Etiquetado de datos.

En adelante, el proceso de entrada de los datos y entrenamiento se independizan según el tipo de modelo.

Como dato importante, y aplicable a cada uno de los siguientes modelos es la normalización la cual, si bien, en muchos casos y contextos es aplicable y de utilidad para obtener mejores resultados debido a la estandarización de sus datos, en este caso dado que una de las principales características diferenciales entre las categorías recolectadas es la variación de magnitud (en este caso de aceleración en cada uno de sus ejes), la decisión de aplicar normalización a los datos fue perjudicial y afecto en gran escala la precisión resultante del modelo final.

Como muestra de ello se presenta un ejemplo de resultado:

- Con normalización:

0	0.27	0.50	0.35	8
1	0.00	0.00	0.00	8
2	0.62	1.00	0.77	5
3	0.00	0.00	0.00	10
accuracy			0.29	31
macro avg	0.22	0.38	0.28	31
weighted avg	0.17	0.29	0.21	31

Fig. 28. Matriz de confusión con datos normalizados.

- Sin normalización:

	precision	recall	f1-score	support
0	0.55	0.75	0.63	8
1	0.86	0.75	0.80	8
2	1.00	1.00	1.00	5
3	0.75	0.60	0.67	10
accuracy			0.74	31
macro avg	0.79	0.78	0.77	31
weighted avg	0.77	0.74	0.75	31

Fig. 29. Matriz de confusión con datos sin normalizar.

Como actualización a este apartado cabe resaltar que, en este caso al tratarse de un análisis con arquitectura Transformers, esta cuenta con una capa de normalización intrínseca en su arquitectura, y que de acuerdo a diversas pruebas realizadas incide de la misma manera que la normalización de los datos por fuera del modelo, por tanto a modo comparativo, se utilizaran dos modelos de Transformers (con y sin normalización) para analizar respectivamente esta situación y sus consecuencias en el entrenamiento y resultados.

Como primer modelo se escogió una RNN ya trabajada anteriormente con el fin de dar un mejor contexto.

A. Modelo 1: Red con unidades recurrentes

1. *Se divide los canales del archivo en 3 vectores de igual longitud*

```
for i in range(90):
  Xtrain[i,:,0]=Datos[i*600:(i+1)*600,0].T
  Xtrain[i,:,1]=Datos[i*600:(i+1)*600,1].T
  Xtrain[i,:,2]=Datos[i*600:(i+1)*600,2].T

print(Xtrain.shape)
```

```
(90, 600, 3)
```

Fig. 44. Unión de datos en 3 arreglos unidimensionales.

2. *Se define la arquitectura del modelo RNN*

```
Model: "sequential_10"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 simple_rnn_26 (SimpleRNN)   (None, 600, 5)            45

 simple_rnn_27 (SimpleRNN)   (None, 600, 10)           160

 simple_rnn_28 (SimpleRNN)   (None, 15)                390

 dense_9 (Dense)             (None, 4)                 64

=================================================================
Total params: 659 (2.57 KB)
Trainable params: 659 (2.57 KB)
Non-trainable params: 0 (0.00 Byte)
```

Fig. 45. Arquitectura del modelo RNN.

3. *Graficación y diagramación del modelo*

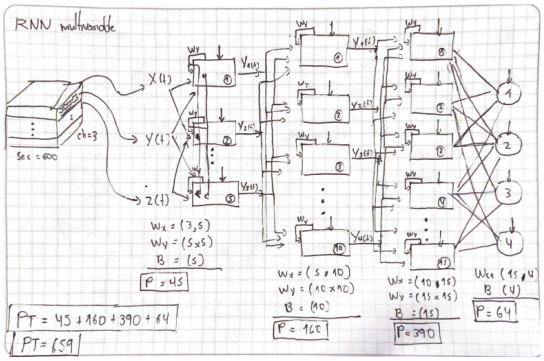


Fig. 46. Diagrama del modelo RNN.

4. *Entrenamiento del modelo*

```
modelo.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
historia = modelo.fit(Xtrain,Ytrain, epochs=1000,batch_size=None,verbose=0)
```

Fig. 47. Entrenamiento del modelo.

5. *Gráfica de la función de pérdida del entrenamiento*

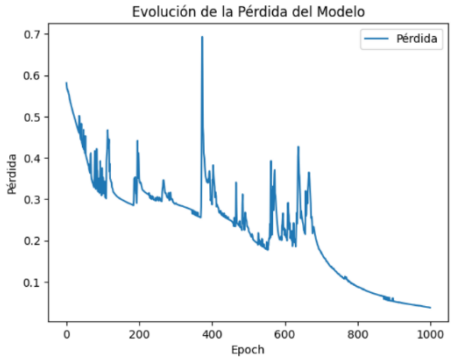


Fig. 48. Función de pérdida del entrenamiento.

6. *Se carga y procesa el conjunto de datos para testeo de la misma forma*
7. *Evaluación del modelo con el dataset de testeo*

```
modelo.evaluate(XVal, YVal)
```

```
1/1 [==============================] - 0s 487ms/step
[0.8691443204879761, 0.7096773982048035]
```

Fig. 49. Evaluación con datos de testeo.

8. *Resultados:* Dentro del contexto a evaluar y la naturaleza de sus datos, las RNN se presentan como una técnica adecuada para realizar el entrenamiento del modelo, el cual, si bien tiene puede mejorar su

precisión global en la clasificación (lo cual ciertamente se puede solucionar mediante una mejor y mayor cantidad de datos recolectados, así como técnicas de regularización), la sección más crítica determinada por la clasificación entre un correcto e incorrecto lavado de dientes tiene un buen resultado debido a que, luego de analizar la matriz de confusión, la categoría dos (lavado incorrecto) presenta una precisión y un recall de 88%, con una baja existencia de falsos negativos y falsos positivos. Se adjunta la matriz de confusión para más detalle.

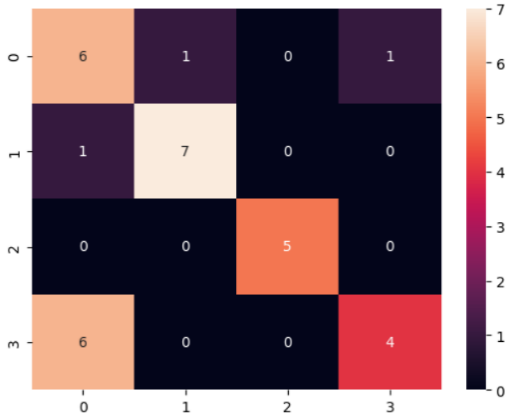


Fig. 50. Matriz de confusión del modelo RNN.

C *Modelo 2: Red transformers con normalización*

1. *Se divide los canales del archivo en 3 vectores de igual longitud tanto para el conjunto de entrenamiento, como de testeo*

```
for i in range(90):
  Xtrain[i,:,0]=Datos[i*600:(i+1)*600,0].T
  Xtrain[i,:,1]=Datos[i*600:(i+1)*600,1].T
  Xtrain[i,:,2]=Datos[i*600:(i+1)*600,2].T

print(Xtrain.shape)
```

```
(90, 600, 3)
```

Fig. 37. Unión de datos en 3 vectores según canal.

2. *Se define la arquitectura del modelo Transformer con capa de normalización en el encoder*

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	[(None, 600, 3)]	0	[]
layer_normalization (LayerNormalization)	(None, 600, 3)	6	['input_1[0][0]']
multi_head_attention (MultiHeadAttention)	(None, 600, 3)	3072	['layer_normalization[0][0]', 'layer_normalization[0][0]']
dropout (Dropout)	(None, 600, 3)	0	['multi_head_attention[0][0]']
conv1d (Conv1D)	(None, 600, 4)	16	['dropout[0][0]']
dropout_1 (Dropout)	(None, 600, 4)	0	['conv1d[0][0]']
conv1d_1 (Conv1D)	(None, 600, 3)	15	['dropout_1[0][0]']
tf.__operators__.add (TFOpLambda)	(None, 600, 3)	0	['dropout[0][0]', 'input_1[0][0]']
tf.__operators__.add_1 (TFOpLambda)	(None, 600, 3)	0	['conv1d_1[0][0]', 'tf.__operators__.add[0][0]']
global_average_pooling1d (GlobalAveragePooling1D)	(None, 600)	0	['tf.__operators__.add_1[0][0]']
dense (Dense)	(None, 128)	76928	['global_average_pooling1d[0][0]']
dropout_2 (Dropout)	(None, 128)	0	['dense[0][0]']
dense_1 (Dense)	(None, 4)	516	['dropout_2[0][0]']

Total params: 80553 (314.66 KB)

Fig. 38. Arquitectura del modelo Transformer normalizado.

3. *Graficación y diagramación del modelo*

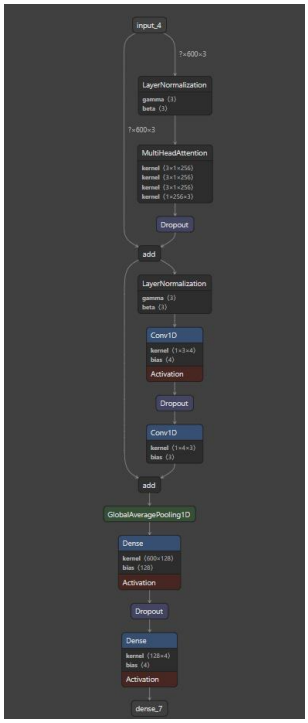


Fig. 39. Diagrama del modelo Transformer normalizado.

4. *Entrenamiento del modelo*

```
historia=modelo.fit(
    x= Xtrain,
    y= Ytrain,
    validation_data=(XVal,YVal),
    #validation_split=0.2,
    epochs=100,
#    batch_size=64,
    callbacks=callback,
)

modelo.evaluate(XVal, YVal, verbose=1)
```

Fig. 40. Entrenamiento del modelo.

5. *Gráfica de la función de pérdida del entrenamiento y testeo*

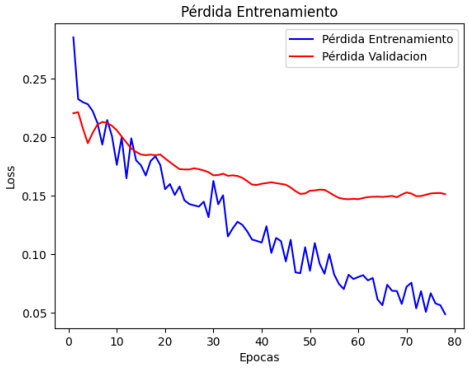


Fig. 41. Función de pérdida del entrenamiento y testeo.

6. *MSE de evaluación del modelo con el dataset de testeo*

```
# Se calcula el MSE del modelo con los datos de testeo
MSE = modelo.evaluate(XVal, YVal)
print("Loss:", MSE)
```

```
1/1 [==============================] - 0s 354ms/step - loss: 0.1515
Loss: 0.15145187079906464
```

Fig. 42. MSE evaluación del modelo basado en datos de testeo.

7. *Resultados:* Como segundo modelo se escogió la nueva arquitectura del contexto llamada Transformer, la cual se estableció y entrenó de manera estandarizada con una capa de normalización dentro del bloque codificador y previo a la capa de multi atención. Esto en un principio con el fin de mejorar el aprendizaje de los datos.

   Pero para sorpresa posterior, los resultados indicaban que dicha capa de normalización, aun estando de manera oficial dentro de la arquitectura generaba un deterioro en el patrón de reconocimiento de los datos por categoría al igual que de haberse normalizado en el preprocesamiento. Si bien esta anomalía es característica de contextos donde la diferenciación de los datos depende de la magnitud de sus valores, también cabe resaltar que a diferencia de otras arquitecturas entrenadas con el mismo dataset y normalizadas de igual manera, esta debido a su característica de extraer, retener y correlacionar información en el largo plazo y mediante diferentes puntos de atención, permitio que aun con un dataset normalizado (menor calidad) pueda obtener resultados por encima del promedio de otras arquitecturas de DL, aunque no menos mediocres e inútiles. Obteniendo un 50% de precisión en la clasificación. Cabe resaltar que en esta ocasión la categoría con mayor recall fue la categoría 3 (inactividad). A continuación, se presenta la matriz de confusión correspondiente.

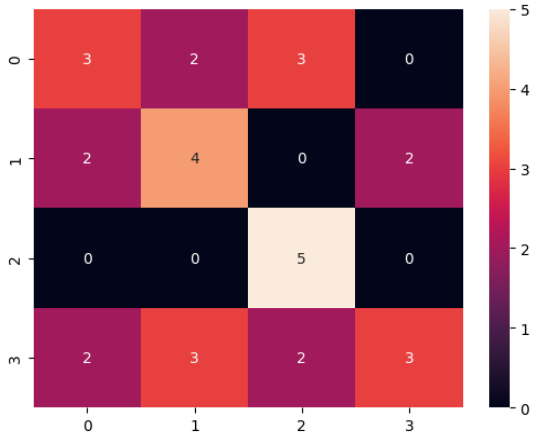


Fig. 43. Matriz de confusión del modelo Transformer normalizado.

D Modelo 3: Red transformer sin normalización

1. *Se divide los canales del archivo en 3 vectores de igual longitud*

```
for i in range(90):
  Xtrain[i,:,0]=Datos[i*600:(i+1)*600,0].T
  Xtrain[i,:,1]=Datos[i*600:(i+1)*600,1].T
  Xtrain[i,:,2]=Datos[i*600:(i+1)*600,2].T

print(Xtrain.shape)
```

```
(90, 600, 3)
```

Fig. 44. Unión de datos en 3 arreglos unidimensionales.

2. *Se define la arquitectura del modelo Transformer sin ningún tipo de normalización*

Layer (type)	Output Shape	Param #	Connected to
input_11 (InputLayer)	[(None, 600, 3)]	0	[]
multi_head_attention_11 (MultiHeadAttention)	(None, 600, 3)	3072	['input_11[0][0]', 'input_11[0][0]']
dropout_23 (Dropout)	(None, 600, 3)	0	['multi_head_attention_11[0][0]']
tf.__operators__.add_16 (TFOpLambda)	(None, 600, 3)	0	['dropout_23[0][0]', 'input_11[0][0]']
conv1d_16 (Conv1D)	(None, 600, 4)	16	['tf.__operators__.add_16[0][0]']
dropout_24 (Dropout)	(None, 600, 4)	0	['conv1d_16[0][0]']
conv1d_17 (Conv1D)	(None, 600, 3)	15	['dropout_24[0][0]']
tf.__operators__.add_17 (TFOpLambda)	(None, 600, 3)	0	['conv1d_17[0][0]', 'tf.__operators__.add_16[0][0]']
global_average_pooling1d_7 (GlobalAveragePooling1D)	(None, 600)	0	['tf.__operators__.add_17[0][0]']
dense_14 (Dense)	(None, 128)	76928	['global_average_pooling1d_7[0][0]']
dropout_25 (Dropout)	(None, 128)	0	['dense_14[0][0]']
dense_15 (Dense)	(None, 4)	516	['dropout_25[0][0]']

Total params: 80547 (314.64 KB)

Fig. 45. Arquitectura del modelo transformer sin normalizar.

3. *Graficación y diagramación del modelo*

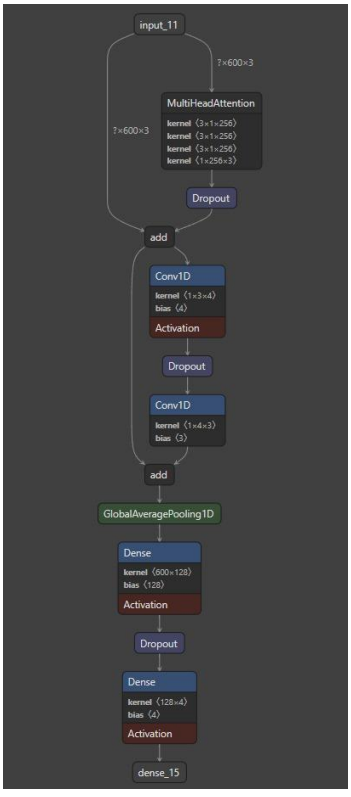


Fig. 46. Diagrama del modelo transformer sin normalizar.

4. *Entrenamiento del modelo*

```
historia=modelo.fit(
    x= Xtrain,
    y= Ytrain,
    validation_data=(XVal,YVal),
    #validation_split=0.2,
    epochs=100,
#    batch_size=64,
    callbacks=callback,
)
```

Fig. 47. Entrenamiento del modelo.

5. *Gráfica de la función de pérdida del entrenamiento*

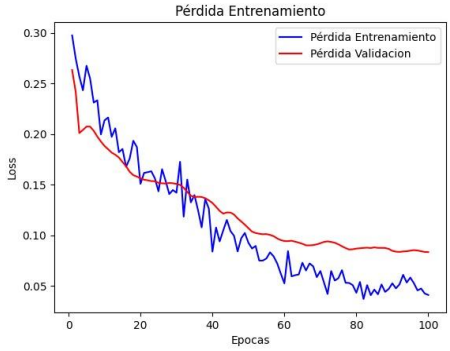


Fig. 48. Función de pérdida del entrenamiento.

6. *MSE de evaluación del modelo con el dataset de testeo*

```
# Se calcula el MSE del modelo con los datos de testeo
MSE = modelo.evaluate(XVal, YVal)
print("Loss:", MSE)
```

```
1/1 [==============================] - 0s 398ms/step - loss: 0.0802
Loss: 0.0801735669374466
```

Fig. 49. MSE evaluación del modelo basado en datos de testeo.

7. *Resultados:* Al igual que el anterior modelo, la arquitectura basada para este entrenamiento se enfocó en el uso de Transformers, pero como se ha mencionado anteriormente, y es un apartado clave para este análisis. La normalización en este modelo específico no se realizó, con el fin de analizar, evaluar y comprobar la importancia en la elección de incorporar o no este tipo de procesamiento en función del contexto y dataset escogidos. Permitiendo obtener unos resultados que contundentemente apoyan la necesidad de una mejor evaluación en la implementación indiscriminada de la normalización en modelos de DL.

   Específicamente, hablando en números, los resultados para este contexto se mostraron plenamente

favorables a razón de no realizar normalización al dataset, ni siquiera en las capas propias de la arquitectura transformer. Logrando así, gracias a ello y a los beneficios que trae consigo esta arquitectura, una precisión global del 77%, y del 89% para la clasificación de la categoría 2, la cual es clave al tratarse de la identificación de patrones correspondientes al mal cepillado de dientes. Esto junto con un recall del 100%, permitiendo eliminar por completo la existencia de falsos negativos, aunque con un pequeño porcentaje de falsos positivos, que probablemente podrían mejorar con un aumento en el volumen de datos del dataset. A continuación se adjunta la matriz de confusión para más detalle.

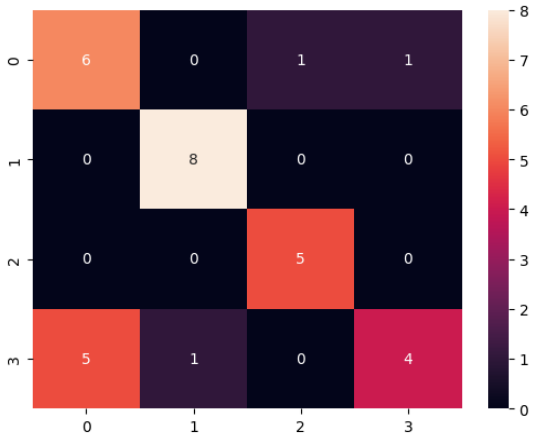


Fig. 50. Matriz de confusión del modelo Transformer sin normalizar.

II. PUNTO 2

1. Problemática

En el contexto de una crisis global, como una pandemia o un evento climático extremo, las redes sociales se convierten en un barómetro vital de la respuesta pública. Monitorear, clasificar y comprender los sentimientos expresados puede proporcionar a los organismos de respuesta a emergencias, gobiernos y organizaciones de salud pública insights valiosos sobre la percepción del público, permitiéndoles ajustar sus estrategias de comunicación y respuesta en tiempo real. Este proyecto utiliza algoritmos avanzados de PLN para analizar tweets relacionados con la crisis, clasificando los sentimientos en múltiples categorías para obtener una comprensión matizada de la respuesta emocional del público.

2. Introducción

En la era de la información digital, la proliferación de redes sociales como Twitter ha proporcionado una plataforma sin precedentes para la expresión pública de opiniones y emociones. Durante eventos de gran impacto social, como una pandemia global, los datos generados por los usuarios en estas plataformas ofrecen una oportunidad única para analizar la respuesta emocional y las tendencias de sentimiento de la población en tiempo real. Este análisis no solo tiene implicaciones sociológicas significativas sino que también puede ser instrumental para los tomadores de decisiones y líderes de opinión en la formulación de estrategias de comunicación y políticas públicas.

Este trabajo presenta un enfoque computacional para la clasificación de sentimientos en tweets, implementando y comparando dos arquitecturas de modelos de procesamiento de lenguaje natural: Redes Neuronales Recurrentes (RNN) y el modelo Transformer, específicamente BERT (Bidirectional Encoder Representations from Transformers). Se seleccionó un dataset que comprende una variedad de tweets etiquetados con sentimientos, que representan un espectro de reacciones humanas en contextos de crisis. Se entrenaron ambos modelos en este conjunto de datos, con el objetivo de identificar cuál estructura de modelo es más efectiva en la clasificación precisa de los sentimientos.

A lo largo del desarrollo, se documentaron las fases de preprocesamiento de datos, entrenamiento, validación y análisis comparativo de los modelos. Se presta especial atención a la interpretación de los resultados de cada modelo, evaluando no solo su precisión sino también su capacidad de generalización y su aplicabilidad práctica en escenarios del mundo real. El resultado es un marco de trabajo que puede ser desplegado para monitorear dinámicas sociales en tiempos de cambio y que puede ser extendido o adaptado para otras aplicaciones de análisis de sentimiento en redes sociales.

3. Desarrollo

En primera medida, se determina el uso de Google Colab para realizar el desarrollo de ambos modelos. Primero, se realizan las importaciones necesarias, incluyendo la conexión a Google Colab.

```python
# Montar Google Drive y cargar el dataset
from google.colab import drive
drive.mount('/content/drive')

import pandas as pd
import numpy as np
import tensorflow as tf
from tensorflow import keras
from matplotlib import pyplot as plt
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.utils import plot_model
from tensorflow.keras.layers import Embedding, LSTM, Dense
from tensorflow.keras.models import Sequential
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
```

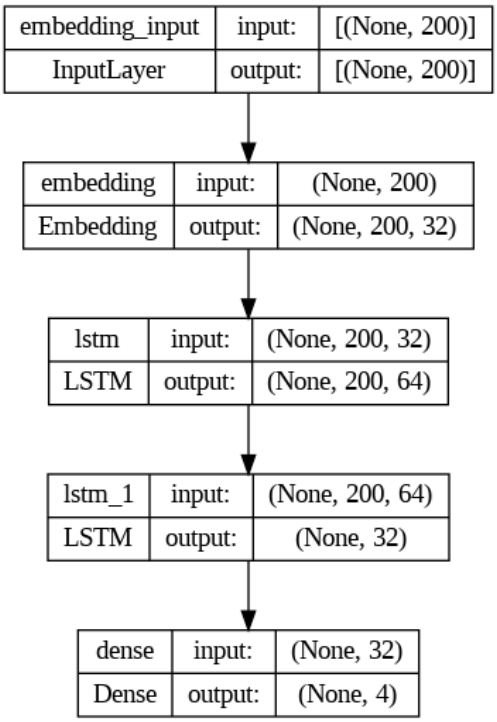
Posterior a esto, se define la ruta al dataset y se carga para su uso de ahora en adelante.

```python
# Ruta al dataset
dataset_path = '/content/drive/MyDrive/dataSecurncial/Punto2/twitter_training.csv.zip'

# Cargar el dataset
dataset = pd.read_csv(dataset_path)
print(dataset)
```

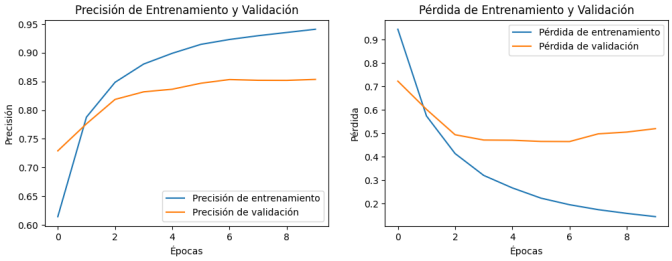
En el siguiente código, se comienza con la extracción de las columnas de texto y etiquetas de sentimiento del dataset, convirtiendo todos los tweets a cadenas de texto para procesamiento uniforme. Se usa LabelEncoder para convertir las etiquetas categóricas de sentimiento en valores numéricos, permitiendo que el modelo las interprete de manera eficiente. Acto seguido, se tokeniza los tweets para descomponerlos en palabras individuales y los transformo en secuencias de números, aplicando pad_sequences para normalizar su longitud. Esto asegura que todos los datos de entrada tengan la misma forma, una etapa esencial para el entrenamiento de modelos de aprendizaje automático. Finalmente, las etiquetas numéricas son convertidas en vectores binarios one-hot y divido el conjunto de datos en dos: uno para entrenar el modelo y otro para validar su rendimiento y capacidad de generalización.

```python
# Utilizar la columna de texto y la de etiqueta de sentimiento
tweets = dataset.iloc[:, 3].astype(str)  # Convertir a string
etiquetas = dataset.iloc[:, 2].values

# Codificar las etiquetas
label_encoder = LabelEncoder()
encoded_etiquetas = label_encoder.fit_transform(etiquetas)

# Preprocesamiento de los datos
# Tokenización
tokenizer = Tokenizer(num_words=10000, oov_token="<OOV>")
tokenizer.fit_on_texts(tweets)
word_index = tokenizer.word_index

# Convertir textos a secuencias
sequences = tokenizer.texts_to_sequences(tweets)

# Padding
padded = pad_sequences(sequences, maxlen=200)

# Convertir etiquetas a categóricas
etiquetas_categoricas = to_categorical(encoded_etiquetas)

# Dividir los datos en conjuntos de entrenamiento y prueba
train_padded, test_padded, train_etiquetas, test_etiquetas = train_test_split(padded, etiquetas_categoricas, test_size=0.2)
```

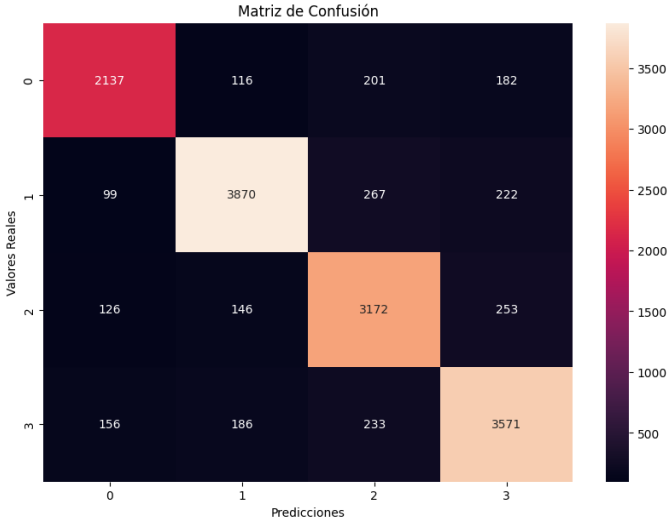
Posteriormente, se define el modelo. La estructura se inicia con una capa de entrada que recibe secuencias de longitud fija de 200 tokens. A continuación, una capa de embedding transforma cada token en un vector denso de 32 dimensiones, lo que permite que el modelo capte la semántica de las palabras en un espacio vectorial reducido. Dos capas de LSTM siguen al embedding, donde la primera procesa la secuencia con 64 unidades y la segunda refina la salida a 32 unidades, permitiendo al modelo capturar dependencias a largo plazo y patrones temporales en el texto. Finalmente, una capa densa conecta la salida de la LSTM con 4 unidades de salida, cada una correspondiente a una categoría de sentimiento, aplicando la función de activación softmax para obtener una distribución de probabilidad sobre las posibles clases.



Las gráficas presentadas a continuación reflejan la evolución de la precisión y la pérdida durante el proceso de entrenamiento y validación del modelo de clasificación de sentimientos. En la gráfica de la izquierda, se observa que el modelo está aprendiendo efectivamente de los datos proporcionados. La precisión de validación también mejora, aunque a un ritmo más lento, lo que sugiere que el modelo está generalizando bien a nuevos datos no vistos durante el entrenamiento. En la gráfica de la derecha, se muestra la pérdida de entrenamiento y de validación, descendiendo ambas conforme el modelo se ajusta mejor a los datos. Sin embargo, se debe estar atento al espacio entre las líneas de entrenamiento y validación, ya que una brecha grande puede ser indicativa de sobreajuste, donde el modelo aprende a memorizar los datos de entrenamiento en lugar de generalizar a partir de ellos.



La matriz de confusión visualizada proporciona una representación detallada del rendimiento del modelo de clasificación. Cada celda muestra el número de predicciones hechas por el modelo versus las etiquetas reales del conjunto de datos de prueba. Las diagonales destacan las clasificaciones correctas, mientras que los otros valores indican la cantidad de confusiones entre clases. Esta matriz es una herramienta crucial para identificar las fortalezas y debilidades del modelo en la clasificación de diferentes sentimientos.



Ahora, a continuación, vemos un ejemplo de predicción. El resultado es óptimo.

```
# Función para predecir la etiqueta de un tweet
def predecir_tweet(texto):
    secuencia = tokenizer.texts_to_sequences([texto])
    padded = pad_sequences(secuencia, maxlen=200)
    prediccion = model.predict(padded)
    clase_predicha = label_encoder.inverse_transform([np.argmax(prediccion)])
    print("Predicción:", prediccion, "\nClase Predicha:", clase_predicha[0])

predecir_tweet("I wanna eat some sushi because I love it!")
```

```
1/1 [==============================] - 0s 61ms/step
Predicción: [[0.14957088 0.00481288 0.14384465 0.7017716 ]]
Clase Predicha: Positive
```

Ahora se probará el siguiente modelo, el cual consiste en un Transformer usando Bert. Se realiza la importación de las librerías necesarias para la realización de esta parte del ejercicio.

```
# Importar las bibliotecas necesarias
from google.colab import drive
from transformers import BertTokenizer, TFBertForSequenceClassification
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
import tensorflow as tf
import numpy as np
import pandas as pd

# Montar Google Drive
drive.mount('/content/drive')

# Cargar el dataset desde Drive
dataset_path = '/content/drive/MyDrive/dataSecurncial/Punto2/twitter_training.csv.zip'
df = pd.read_csv(dataset_path)
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

El siguiente código detalla el proceso de preparación de datos para su uso con BERT, un modelo de Transformer preentrenado para tareas de PLN. Comienza seleccionando y procesando las columnas relevantes del dataframe: los tweets se convierten a texto y las etiquetas de sentimiento se extraen directamente. Se utiliza LabelEncoder para transformar las etiquetas a una forma numérica, lo que es esencial para el tratamiento computacional posterior. La cantidad de clases únicas se determina, lo que es crucial para configurar la capa de salida del modelo.

Posteriormente, se inicializa el tokenizer de BERT y se preparan los tweets. El método encode_plus se emplea para tokenizar los tweets, agregar tokens especiales, definir una longitud máxima y generar las máscaras de atención necesarias para el modelo BERT. Estos pasos convierten el texto en una forma que BERT puede procesar, resultando en listas de identificadores y máscaras de atención que luego se convierten en arrays de NumPy, preparando así el conjunto de datos para el entrenamiento y la evaluación del modelo.

```
# Seleccionar las columnas de texto y etiqueta
tweets = df.iloc[:, 3].astype(str).values  # Texto del tweet
etiquetas = df.iloc[:, 2].values  # Etiquetas de sentimiento

# Codificar las etiquetas en formato numérico
label_encoder = LabelEncoder()
etiquetas_codificadas = label_encoder.fit_transform(etiquetas)
num_clases = len(np.unique(etiquetas_codificadas))

# Inicializar el tokenizer de BERT
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')

# Preprocesar los tweets para BERT
input_ids = []
attention_masks = []

for tweet in tweets:
    encoded_dict = tokenizer.encode_plus(
        tweet,
        add_special_tokens=True,  # Agregar '[CLS]' y '[SEP]'
        max_length=64,  # Definir la longitud máxima de los tweets
        truncation=True,
        padding='max_length',  # Rellenar con ceros
        return_attention_mask=True,  # Construir la máscara de atención
        return_tensors='np',  # Retornar tensores de NumPy
    )
    input_ids.append(encoded_dict['input_ids'][0])
    attention_masks.append(encoded_dict['attention_mask'][0])

# Convertir las listas a arrays de NumPy
input_ids = np.array(input_ids)
attention_masks = np.array(attention_masks)
etiquetas_codificadas = np.array(etiquetas_codificadas)
```

El código a continuación, se encarga de establecer el flujo de trabajo para el entrenamiento de un modelo Transformer en una tarea de clasificación de secuencias. Tras dividir los datos en subconjuntos de entrenamiento y validación, se inicializa un modelo TFBertForSequenceClassification con la arquitectura

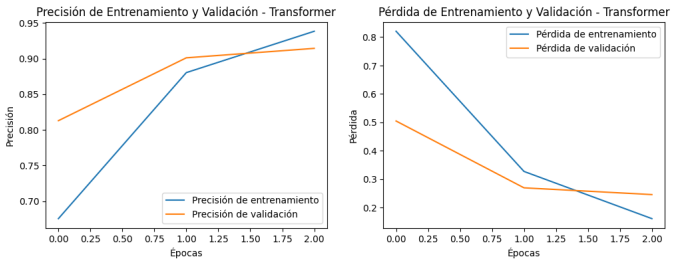
'bert-base-uncased' y un número de etiquetas que refleja las clases del dataset. Se compila el modelo con el optimizador Adam y una función de pérdida específica para tratar con etiquetas dispersas y logits, una configuración común en tareas de clasificación con múltiples clases. Finalmente, se entrena el modelo usando estos datos, proporcionando la entrada tokenizada y las máscaras de atención, y se registra el historial de entrenamiento para su posterior análisis. Este enfoque permite aprovechar el potencial de BERT para capturar contextos complejos y relaciones entre palabras, lo cual es fundamental para comprender la naturaleza intrincada del lenguaje natural en las redes sociales.

```
# Dividir los datos en conjuntos de entrenamiento y validación
train_inputs, validation_inputs, train_labels, validation_labels = train_test_spl
    input_ids, etiquetas_codificadas, random_state=2018, test_size=0.1
)
train_masks, validation_masks, _, _ = train_test_split(
    attention_masks, etiquetas_codificadas, random_state=2018, test_size=0.1
)

# Definir el modelo Transformer para clasificación
model = TFBertForSequenceClassification.from_pretrained(
    'bert-base-uncased', num_labels=num_clases
)

# Compilar el modelo
optimizer = tf.keras.optimizers.Adam(learning_rate=2e-5, epsilon=1e-08)
loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
model.compile(optimizer=optimizer, loss=loss, metrics=['accuracy'])

# Entrenar el modelo
history = model.fit(
    [train_inputs, train_masks],
    train_labels,
    validation_data=([validation_inputs, validation_masks], validation_labels),
    batch_size=32,
    epochs=3
)
```

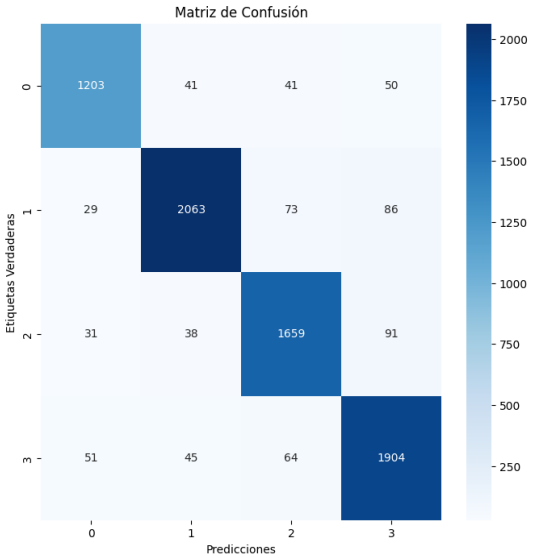
El modelo se compone principalmente de la capa TFBertMainLayer, que es el núcleo del modelo BERT con más de 109 millones de parámetros, reflejando la profundidad y complejidad de la red. Además, incluye una capa de Dropout para la regularización, ayudando a prevenir el sobreajuste, y una capa Dense que funciona como clasificador, ajustando la salida de BERT a las categorías de etiquetas específicas del conjunto de datos. Con más de 109 millones de parámetros entrenables, este modelo es capaz de captar y procesar las sutilezas del lenguaje humano, lo que lo hace excepcionalmente potente para tareas de PLN como la clasificación de sentimiento. Es posible visualizar una imágen de la estructura del modelo a través de Google Drive. El archivo se llama “model_transformer.png”

```
Model: "tf_bert_for_sequence_classification"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 bert (TFBertMainLayer)      multiple                  109482240

 dropout_37 (Dropout)        multiple                  0

 classifier (Dense)          multiple                  3076

=================================================================
Total params: 109485316 (417.65 MB)
Trainable params: 109485316 (417.65 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Se guarda el modelo en Google Drive, y se analizan las gráficas que explican el resultado del entrenamiento.Se observa un aumento constante en la precisión de entrenamiento y validación, lo que indica que el modelo está aprendiendo adecuadamente y mejorando su capacidad predictiva con cada época. Por otro lado, la pérdida disminuye notablemente tanto en entrenamiento como en validación, sugiriendo que el modelo se está ajustando bien a los datos sin signos evidentes de sobreajuste, ya que las líneas de entrenamiento y validación se mantienen próximas entre sí. Estas tendencias positivas en las métricas son indicativas de un modelo robusto y eficaz en la tarea de discernir los sentimientos expresados en los datos.



La matriz de confusión mostrada refleja la capacidad del modelo Transformer para clasificar correctamente los sentimientos expresados en tweets. Las celdas a lo largo de la diagonal principal representan las instancias donde las predicciones coinciden con las etiquetas verdaderas, indicando aciertos del modelo. Las celdas fuera de la diagonal principal señalan las ocasiones en que el modelo ha confundido una etiqueta por otra. Aunque la mayoría de las predicciones son correctas, como se evidencia en la concentración de valores más altos en la diagonal, también hay errores notables que sugieren áreas de mejora, particularmente en la diferenciación entre algunas clases específicas.



Finalmente, se realiza una predicción real del modelo, como se ve a continuación. Se realizaron varias, y el equipo de trabajo notó la superioridad de este modelo con respecto al anterior.

```
[ ]  # Ejemplo de uso: predecir el sentimiento de un nuevo tweet
     tweet = "I don't need to buy that, are you trying to scam me!?"
     print(predecir_sentimiento(tweet))

     1/1 [==============================] - 5s 5s/step
     Negative
```

En conclusión, la comparativa entre los modelos RNN y Transformer para la clasificación de sentimientos en tweets ha demostrado la superioridad del Transformer, específicamente BERT, en términos de precisión y generalización. Los resultados obtenidos reflejan que BERT no solo aprende con mayor eficacia las complejidades del lenguaje natural contenidas en los datos de entrenamiento, sino que también mantiene una notable capacidad de adaptación a nuevos datos, como lo evidencia la consistencia en la precisión de validación. La menor brecha entre las métricas de entrenamiento y validación en BERT frente al RNN respalda su robustez y menor susceptibilidad al sobreajuste, destacando su idoneidad para el análisis en tiempo real de sentimientos en plataformas dinámicas como Twitter.

Además, la matriz de confusión del modelo BERT subraya su capacidad para discernir adecuadamente entre distintas categorías de sentimiento, incluso en casos sutiles donde el contexto y la intención pueden ser difíciles de capturar. Este nivel de precisión en la clasificación es crítico en aplicaciones del mundo real, donde comprender correctamente la intención del usuario puede tener implicaciones significativas para la toma de decisiones empresariales y la gestión de la comunicación en situaciones de crisis. Por tanto, BERT emerge como la elección preferente para aplicaciones que requieren no solo un alto rendimiento, sino también una interpretación detallada y matizada del lenguaje humano.

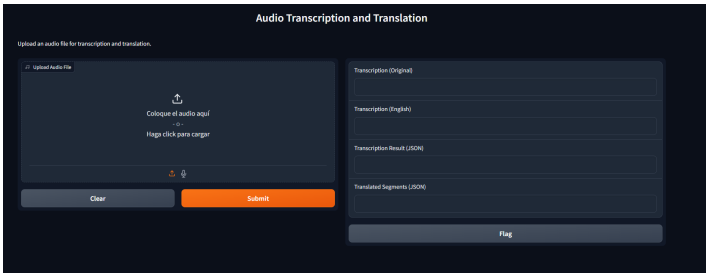
III. PUNTO 3

Para trabajar el punto número 3 se determinó en realizar una herramienta que sirviera en poner subtítulos en un podcast o audio de español , en el que le pasaras un audio en español con extensión .wav y este te devuelve la transcripción del audio , la traducción de ese audio a ingles , dos archivos .json de la línea de tiempo en español y en inglés .

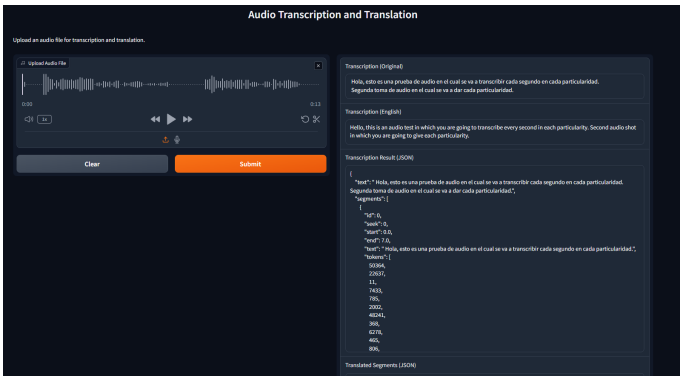
Esto se realizó para la transcripción del audio usando whisper y para la traducción se uso transformers usando pipelines usando el modelo traduccion de huggin face de español a ingles “Helsinki-NLP/opus-mt-es-en” ver en :https://huggingface.co/Helsinki-NLP/opus-mt-es-en

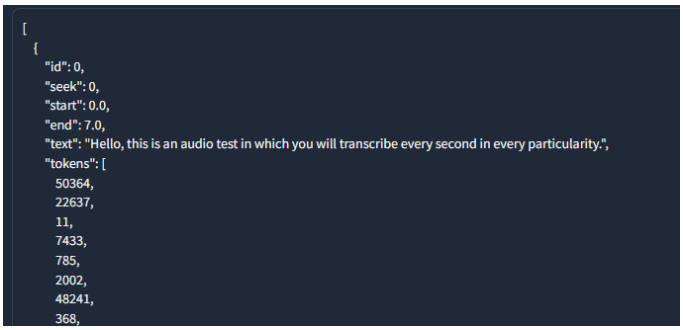
La integración fue realizada con Gradio en donde se sube los archivo de audio .wav en español y este a lado derecho se visualizará la transcripción de texto de español y su traduccion al ingles , ademas tambien se visualizará el timeline de la transcripción en español y en inglés, este “timeline” se visualiza el tiempo donde empieza y termina la frase , esto es útil para la creación de subtítulos por lo que esta herramienta ayudará a la creación de subtítulos en español e inglés.

Se muestra cual sería la interfaz , en donde se debe de cargar el archivo de audio . wav

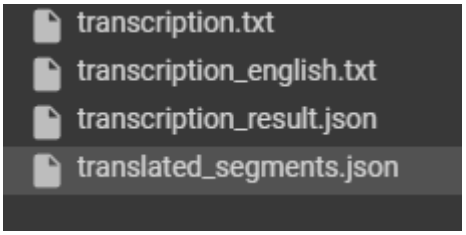


Después de cargado y subido, se espera hasta que finalice:

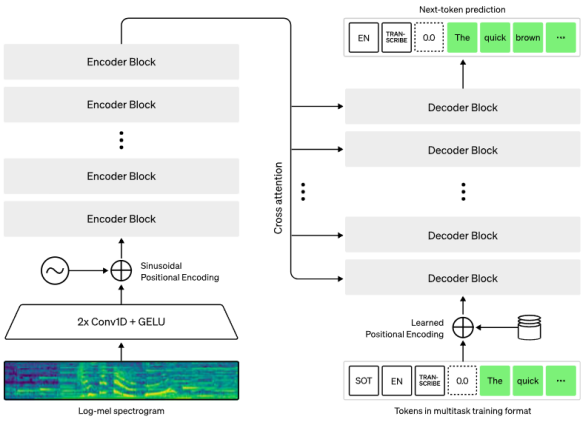




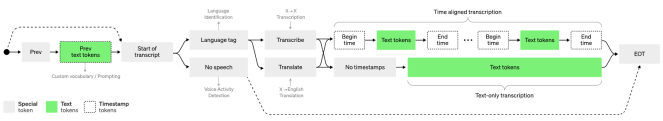
en donde se muestra cuatro apartados anteriormente mencionados , estos archivos se guardan en el entorno local.



Como se observa la transcripción y el timeline es gracias a Whisper. Este funciona debido a que Whisper es un sistema de reconocimiento automático de voz (ASR) entrenado en 680.000 horas de datos supervisados multilingües y multitarea recopilados de la web. Mostramos que el uso de un conjunto de datos tan grande y diverso conduce a una mayor solidez ante los acentos, el ruido de fondo y el lenguaje técnico. Además, permite la transcripción en varios idiomas, así como la traducción de esos idiomas al inglés. Son modelos de código abierto y código de inferencia que sirven como base para crear aplicaciones útiles y para futuras investigaciones sobre el procesamiento sólido del habla.



La arquitectura Whisper es un enfoque simple de extremo a extremo, implementado como un transformador codificador-decodificador. El audio de entrada se divide en fragmentos de 30 segundos, se convierte en un espectrograma log-Mel y luego se pasa a un codificador. Se entrena a un decodificador para predecir el título de texto correspondiente, entremezclado con tokens especiales que dirigen al modelo único a realizar tareas como identificación de idioma, marcas de tiempo a nivel de frase, transcripción de voz multilingüe y traducción de voz al inglés.[10]



Combinando el modelo de traducción de español a ingles fue éxito porque este modelo “Helsinki-NLP/opus-mt-es-en” se evidencia en sus benchmarks

Benchmarks

testset	BLEU	chr-F
newssyscomb2009-spaeng.spa.eng	30.6	0.570
news-test2008-spaeng.spa.eng	27.9	0.553
newstest2009-spaeng.spa.eng	30.4	0.572
newstest2010-spaeng.spa.eng	36.1	0.614
newstest2011-spaeng.spa.eng	34.2	0.599
newstest2012-spaeng.spa.eng	37.9	0.624
newstest2013-spaeng.spa.eng	35.3	0.609
Tatoeba-test.spa.eng	59.6	0.739

IV. CONCLUSIONES

El desarrollo y la creación de diferentes librerías ya consolidadas en torno al mundo de la inteligencia artificial, específicamente del aprendizaje profundo, permite acercar este tipo de enfoques complicados a simple vista, a cada vez más personas con una facilidad técnica que se reduce a aprender únicamente la teoría base detrás de ellas, permitiendo dirigir mayor atención a la recolección, diseño y análisis de datos sin limitaciones.

Si bien es sabido que el costo computacional de un método de aprendizaje profundo (DL) frente a uno de aprendizaje automático (ML) es más alto, la historia ha demostrado y el desarrollo tecnológico exponencial ha permitido reducir paulatinamente esta desventaja y consolidar cada vez más al campo del DL como uno de los enfoques con mayor potencial de crecimiento a futuro gracias a su funcionamiento que se potencia en la correlación del entorno; su integración con diferentes campos como IoT, visión computacional, Big Data; la confiabilidad que lo precede; así como el crecimiento y amplio margen de mejora en el largo plazo.

RECONOCIMIENTOS

Este documento tiene un enfoque práctico y académico que pretende ser de fácil entendimiento y cuenta con un nivel de profundidad relativamente superficial, pero no hubiese sido posible sin las enseñanzas y material impartido por el profesor Jesús A. López y el programa de Especialización en Inteligencia Artificial de la Universidad Autónoma de Occidente.

REFERENCIAS

[1] “La OMS destaca que el descuido de la salud bucodental afecta a casi la mitad de la población mundial”. World Health Organization (WHO). Accedido el 3 de noviembre de 2023. [En línea]. Disponible: https://www.who.int/es/news/item/18-11-2022-who-highlights-oral-health-neglect-affecting-nearly-half-of-the-world-s-population

[2] N. Pallarés Martínez, “Relación de la deficiente higiene bucal y la consecuente aparición de patologías bucales con las enfermedades cardiovasculares.”, Trabajo de grado, Univ. Jaume I, Castellón de la Plana, 2019.

[3] “Ni nos lavamos los dientes lo suficiente ni sabemos cuidar del cepillo”. El País. Accedido el 3 de noviembre de 2023. [En línea]. Disponible: https://elpais.com/elpais/2020/02/19/buenavida/1582130938_406935.html

[4] *Caras de los dientes | Dentistry student, Dental, Dentistry*. Accedido el 3 de noviembre de 2023. [Imagen]. Disponible: https://co.pinterest.com/pin/550987954360150 98/

[5] W. You, A. Hao, S. Li, Y. Wang y Bin Xia. “Deep learning-based dental plaque detection on primary teeth: a comparison with clinical assessments - BMC Oral Health”. BioMed Central. Accedido el 3 de noviembre de 2023. [En línea]. Disponible: https://bmcoralhealth.biomedcentral.com/articles/10.1186/s12903-020-01114-6

[6] Jesus A. Lopez "Procesamiento de datos secuenciales", códigos de clase 521376-EEA2, Facultad de ingeniería, Universidad Autónoma de Occidente, Octubre 2023.

[7] Edge Impulse Inc. *Edge Impulse*. (Data collect and processing).

[8] Google. *Google Colaboratory*. (Model training powered by Python).

[9] “Fine-tuning a model for music classification - Hugging Face Audio Course”. Hugging Face – The AI community building the future. Accedido el 2 de noviembre de 2023. [En línea]. Disponible: https://huggingface.co/learn/audio-course/chapter4/fine-tuning

[10]Introducing Whisper. (s.f.). OpenAI. https://openai.com/research/whisper