MINIPROYECTO #3

Traductor de secuencia a secuencia con transformadores

Melina Paula Gonzalez Rubiño- 2236748
melina.gonzalez@uao.edu.co

Juan Camilo Caicedo Cortes - 2190732
juan_c.caicedo@uao.edu.co

Juan Camilo León - 2190612
juan_camilo.leon@uao.edu.co

Redes neuronales artificiales y deep learning
Universidad Autónoma de Occidente
Facultad de Ingeniería
Noviembre - 2023

***Resumen*— El siguiente informe, corresponde al tercer miniproyecto de la asignatura de Redes neuronales artificiales y deep learning, en donde se detalla la implementación de un traductor automático basado en la arquitectura de Transformadores para la tarea de traducción secuencia a secuencia. La implementación se realizó utilizando librerías y herramientas modernas de deep learning.**

***Palabras claves*— Arquitectura, transformadores, librerías, herramientas, implementación, deep learning.**

***Abstract*— The following report corresponds to the third mini-project of the Artificial Neural Networks and Deep Learning course, which details the implementation of an automatic translator based on the Transformer architecture for the sequence-to-sequence translation task. The implementation was carried out using modern deep learning libraries and tools.**

***Keywords*— Architecture, transformers, libraries, tools, implementation, deep learning.**

I. Introducción

En el panorama dinámico y desafiante de la inteligencia artificial y el procesamiento de lenguaje natural, la traducción automática ha experimentado múltiples avances significativos gracias a la evolución de las arquitecturas de modelos. En este contexto, el presente informe aborda la implementación exitosa de un Traductor de secuencia a secuencia basado en la revolucionaria arquitectura de Transformadores. Esta ejecución no sólo marca un logro significativo en la eficiencia de las traducciones automáticas, sino también resalta la capacidad y flexibilidad relacionada a los modelos de deep learning en el ámbito de la comprensión y generación de lenguaje natural.

En cuanto más crece la demanda global de comunicación transcultural, de manera exponencial, la necesidad de sistemas de traducción automática precisos y ágiles se ha vuelto imperativa. Los modelos de traducción automática basados en transformadores han surgido como líderes en este campo, superando las limitaciones de las arquitecturas anteriores al capturar dependencias a largo plazo de manera más efectiva y permitir el procesamiento paralelo, lo que acelera significativamente el entrenamiento y la deducción o conclusión.

De acuerdo con lo anterior, este informe proporcionará un análisis detallado de la implementación de un traductor de secuencia a secuencia utilizando la arquitectura de Transformadores. Se explorarán aspectos claves, desde el preprocesamiento de datos hasta la evaluación del modelo, detallando decisiones claves de diseño y estrategias adoptadas para mejorar la calidad de las traducciones.

A continuación, se presentarán cada uno de los conceptos relacionados con la implementación de un Traductor de secuencia a secuencia basado en la arquitectura de Transformadores, se conocerá el contexto en el que fue basada dicha implementación, el desarrollo de la implementación y los resultados obtenidos.

II. Marco teórico

A. Modelos Seq2Seq

Son especialmente efectivos en tareas de traducción, donde la sucesión de palabras en un idioma se convierte en una sucesión de palabras diferentes en otro idioma.

Constan de un codificador y un decodificador. El codificador toma la secuencia de entrada y la asigna a un espacio de dimensiones superiores (vector n-dimensional).

B. Transformadores

La estructura de los Transformadores en el campo de la inteligencia artificial consta de capas de codificación y descodificación, las cuales desempeñan una función esencial en el procesamiento y la creación de datos.

El componente de codificación en los Transformadores asume la responsabilidad de comprender y procesar la información de entrada. Para este propósito, emplea la auto-atención, un mecanismo que identifica y establece conexiones entre los diversos elementos de los datos en

secuencia. Además, integra múltiples cabezales y una red de retroalimentación para potenciar la comprensión contextual.

C. Embedding

Un embedding se define como una representación de dimensiones reducidas de un vector que originalmente tiene dimensiones más elevadas. Estos embeddings simplifican el proceso de aprendizaje automático al trabajar con entradas extensas, como vectores dispersos que representan palabras. En un escenario ideal, la función de embedding logra capturar aspectos semánticos de la entrada al situar de manera cercana en el espacio de embedding aquellas entradas que son semánticamente similares.

Es posible adquirir y aplicar un embedding en distintos modelos.

Dentro del ámbito del Procesamiento del Lenguaje Natural (NLP), los embeddings se emplean para expresar palabras, frases o documentos en un espacio vectorial continuo. Esta práctica facilita que los modelos de aprendizaje automático manejen datos textuales con mayor eficacia y precisión.

D. NLP (Natural Language Processing)

El Procesamiento del Lenguaje Natural (NLP) constituye un campo de estudio dedicado a la comprensión computacional del lenguaje humano. Este ámbito se sitúa en la intersección de disciplinas como la Ciencia de Datos, la Inteligencia Artificial (específicamente el Aprendizaje Automático) y la lingüística.

En NLP, las computadoras se encargan de analizar el lenguaje humano, interpretarlo y otorgarle significado para su aplicación práctica. A través de NLP, se llevan a cabo tareas diversas como la síntesis automática de textos, traducción de idiomas, extracción de relaciones, análisis de sentimiento, reconocimiento del habla y clasificación temática de artículos.

E. Multi Head Attention

La atención multicabezal representa un componente de los mecanismos de atención que atraviesa de manera simultánea un mecanismo de atención varias veces en paralelo. Posteriormente, las salidas de atención individuales se combinan concatenándolas y se transforman linealmente para ajustarse a la dimensión deseada. En términos intuitivos, la presencia de múltiples cabezas de atención posibilita enfocarse en diferentes partes de la secuencia de manera distintiva, como por ejemplo, atender a dependencias a largo plazo en contraste con dependencias a corto plazo.

III. Planteamiento

En un mundo cada vez más interconectado, la comunicación transcultural se ha vuelto esencial para el comercio, la colaboración y la comprensión global. Sin embargo, a pesar de los avances significativos en la traducción automática en la última década, persisten desafíos significativos que limitan el óptimo rendimiento de estos sistemas. La necesidad de superar estas limitaciones se presenta como un imperativo, y es en este contexto que se sitúa la implementación de un Traductor de Secuencia a Secuencia basado en la arquitectura de Transformadores.

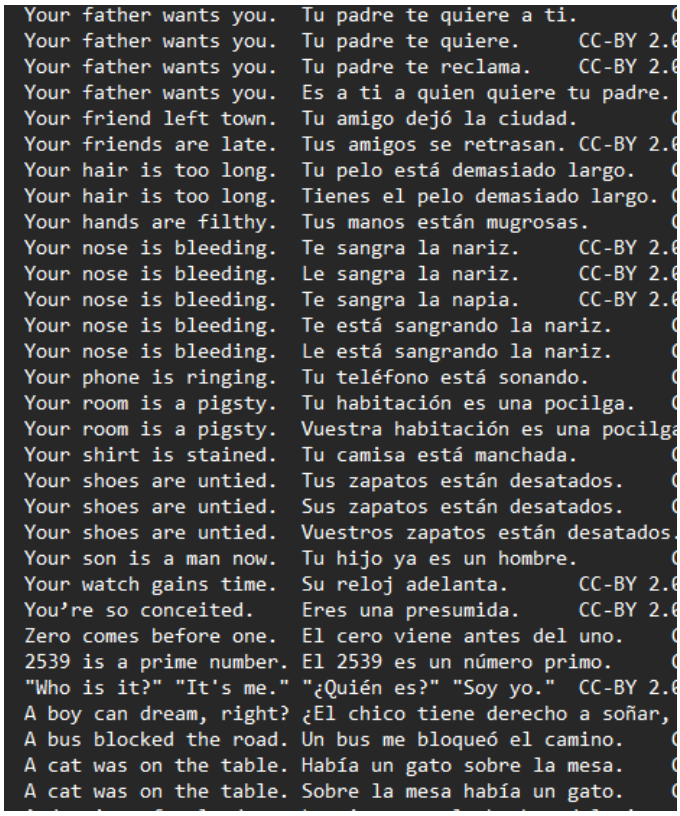
La traducción automática, aunque valiosa, a menudo se ve obstaculizada por la dificultad de capturar matices contextuales, tonos y giros idiomáticos específicos de cada idioma. Las arquitecturas de modelos más antiguas han mostrado limitaciones en la gestión de dependencias a largo plazo y en la adaptación a la variabilidad lingüística. Este problema se manifiesta especialmente en la generación de traducciones que pueden carecer de fluidez y coherencia, lo que compromete la calidad de la comunicación final.

Además, las demandas crecientes de traducción automática en tiempo real, en entornos que van desde aplicaciones móviles hasta plataformas web, requieren soluciones más eficientes y rápidas. La necesidad de procesar grandes volúmenes de datos lingüísticos de manera rápida y precisa plantea un desafío adicional que debe abordarse para mejorar la practicidad y la utilidad de las herramientas de traducción automática.

En este escenario o contexto bastante desafiante, la ejecución de un Traductor de Secuencia a Secuencia basado en la arquitectura de Transformadores se presenta como una estrategia clave para enfrentar los problemas persistentes en la traducción automática. Este informe explorará no sólo cómo los Transformadores enfrentan las limitaciones previamente identificadas, sino también cómo tienen el potencial revolucionario para transformar la manera en que comprendemos y superamos las barreras lingüísticas en el entorno digital actual.

IV. Implementación

Para la implementación de este traductor descrito anteriormente se utilizó la base de datos de Anki (disponible en https://www.manythings.org/anki/), la cual dispone de numerosas opciones de pares de idiomas. Particularmente en este caso se trabajó con la traducción de inglés a español. Al descargar el dataset se cuenta con dos archivos de texto, uno con las referencias del mismo y otro con los datos propiamente dichos, que se ven de la siguiente manera:



```
Your father wants you.  Tu padre te quiere a ti.        (
Your father wants you.  Tu padre te quiere.     CC-BY 2.(
Your father wants you.  Tu padre te reclama.    CC-BY 2.(
Your father wants you.  Es a ti a quien quiere tu padre.
Your friend left town.  Tu amigo dejó la ciudad.        (
Your friends are late.  Tus amigos se retrasan. CC-BY 2.(
Your hair is too long.  Tu pelo está demasiado largo.   (
Your hair is too long.  Tienes el pelo demasiado largo. (
Your hands are filthy.  Tus manos están mugrosas.       (
Your nose is bleeding.  Te sangra la nariz.     CC-BY 2.(
Your nose is bleeding.  Le sangra la nariz.     CC-BY 2.(
Your nose is bleeding.  Te sangra la napia.     CC-BY 2.(
Your nose is bleeding.  Te está sangrando la nariz.     (
Your nose is bleeding.  Le está sangrando la nariz.     (
Your phone is ringing.  Tu teléfono está sonando.       (
Your room is a pigsty.  Tu habitación es una pocilga.   (
Your room is a pigsty.  Vuestra habitación es una pocilga
Your shirt is stained.  Tu camisa está manchada.        (
Your shoes are untied.  Tus zapatos están desatados.    (
Your shoes are untied.  Sus zapatos están desatados.    (
Your shoes are untied.  Vuestros zapatos están desatados.
Your son is a man now.  Tu hijo ya es un hombre.        (
Your watch gains time.  Su reloj adelanta.      CC-BY 2.(
You're so conceited.    Eres una presumida.     CC-BY 2.(
Zero comes before one.  El cero viene antes del uno.    (
2539 is a prime number. El 2539 es un número primo.     (
"Who is it?" "It's me." "¿Quién es?" "Soy yo."  CC-BY 2.(
A boy can dream, right? ¿El chico tiene derecho a soñar,
A bus blocked the road. Un bus me bloqueó el camino.    (
A cat was on the table. Había un gato sobre la mesa.    (
A cat was on the table. Sobre la mesa había un gato.    (
```

Fig. 1. Visualización del dataset

En esta imagen se visualiza cada una de las oraciones en inglés y a continuación su correspondiente traducción. Las oraciones se encuentran ordenadas por cantidad de caracteres (en inglés) como primer criterio y por orden alfabético como segundo criterio (en inglés). Además se proponen varias traducciones para una misma oración.
Una vez escogido el dataset se procede a descargar el mismo en el entorno de programación:

```
text_file = keras.utils.get_file(
    fname="spa-eng.zip",
    origin="http://storage.googleapis.com/download.tensorflow.org/data/spa-eng.zip",
    extract=True,
)
text_file = pathlib.Path(text_file).parent / "spa-eng" / "spa.txt"
```

Fig. 2. Importación del dataset

Se leen cada una de las líneas en el archivo y se les agrega una bandera de inicio y fin para conocer cuándo comienza y finaliza la traducción, y luego se guarda cada una de dichas líneas en una lista llamada text_pairs.

```
with open(text_file) as f:
    lines = f.read().split("\n")[:-1]
text_pairs = []
for line in lines:
    eng, spa = line.split("\t")
    #le agrega las banderas inicio y fin a la oración en español para detectarla
    spa = "[start] " + spa + " [end]"
    #arma una lista con los pares de oraciones en inglés y español
    text_pairs.append((eng, spa))
```

Fig. 3. Lectura del dataset

Para detectar si los datos se obtuvieron correctamente se imprimen 5 oraciones escogidas aleatoriamente:

```
#Imprime 5 oraciones aleatoriamente con su respectiva traducción
for _ in range(5):
    print(random.choice(text_pairs))
```

```
('Wait!', '[start] ¡Espera! [end]')
('Tom returned to Australia.', '[start] Tom regresó a Australia. [end]')
('Why are you following me?', '[start] ¿Por qué me sigue usted? [end]')
("I don't like studying in this heat.", '[start] No me gusta estudiar con este calor. [end]')
('Are you waiting for something?', '[start] ¿Estás esperando algo? [end]')
```

Fig. 4. Comprobación de los datos del dataset

Como los datos fueron cargados satisfactoriamente a la lista se procede a mezclar el orden de las oraciones y a separarlas en conjuntos de entrenamiento, validación y testeo:

```
#se separa el dataset en sets de entrenamiento, validación y testeo
random.shuffle(text_pairs)
num_val_samples = int(0.15 * len(text_pairs))
num_train_samples = len(text_pairs) - 2 * num_val_samples
train_pairs = text_pairs[:num_train_samples]
val_pairs = text_pairs[num_train_samples : num_train_samples + num_val_samples]
test_pairs = text_pairs[num_train_samples + num_val_samples :]

print(f"{len(text_pairs)} total pairs")
print(f"{len(train_pairs)} training pairs")
print(f"{len(val_pairs)} validation pairs")
print(f"{len(test_pairs)} test pairs")
```

```
118964 total pairs
83276 training pairs
17844 validation pairs
17844 test pairs
```

Fig. 5. Mezclado de los datos y separación en conjuntos de entrenamiento, validación y testeo.

Se obtienen 83276 pares (inglés - español) para entrenamiento, 17844 para validación y la misma cantidad para testeo.
Para poder continuar el procesamiento se usa dos veces TextVectorization, una con los datos en inglés y otra en español, para convertir los strings originales en secuencias de valores enteros y poder trabajar con ellos. Además agrega el signo de apertura de interrogación en las preguntas en español y se hace gracias a las herramientas proporcionadas por regex (REGular EXpression).

```
strip_chars = string.punctuation + "¿"
strip_chars = strip_chars.replace("[", "")
strip_chars = strip_chars.replace("]", "")

vocab_size = 15000
#todas las secuencias tienen 20 pasos, o el número que se defina acá
sequence_length = 20
batch_size = 64

def custom_standardization(input_string):
    lowercase = tf.strings.lower(input_string)
    return tf.strings.regex_replace(lowercase,
                                    "[%s]" % re.escape(strip_chars), "")

#Vectorización para inglés
eng_vectorization = TextVectorization(
    max_tokens=vocab_size,
    output_mode="int",
    output_sequence_length=sequence_length)

#vectorización para español
spa_vectorization = TextVectorization(
    max_tokens=vocab_size,
    output_mode="int",
    output_sequence_length=sequence_length + 1,
    standardize=custom_standardization)

train_eng_texts = [pair[0] for pair in train_pairs]
train_spa_texts = [pair[1] for pair in train_pairs]
eng_vectorization.adapt(train_eng_texts)
spa_vectorization.adapt(train_spa_texts)
```

Fig. 6. Vectorización de los datos

Una vez formateado el conjunto de datos aplicando la vectorización y recreando la base de datos con esto, se continúa definiendo finalmente los conjuntos de entrenamiento y validación con sus respectivas dimensiones y luego construyendo el modelo de Transformers, que a su vez utiliza los conceptos de Autoencoders (Decoder y Encoder) y Embedding.

Para el Encoder se tiene:

```
class TransformerEncoder(layers.Layer):
    def __init__(self, embed_dim, dense_dim, num_heads, **kwargs):
        super().__init__(**kwargs)
        self.embed_dim = embed_dim
        self.dense_dim = dense_dim
        self.num_heads = num_heads
        self.attention = layers.MultiHeadAttention(
            num_heads=num_heads, key_dim=embed_dim
        )
        self.dense_proj = keras.Sequential(
            [
                layers.Dense(dense_dim, activation="relu"),
                layers.Dense(embed_dim),
            ]
        )
        self.layernorm_1 = layers.LayerNormalization()
        self.layernorm_2 = layers.LayerNormalization()
        self.supports_masking = True

    def call(self, inputs, mask=None):
        attention_output = self.attention(query=inputs, value=inputs, key=inputs)
        proj_input = self.layernorm_1(inputs + attention_output)
        proj_output = self.dense_proj(proj_input)
        return self.layernorm_2(proj_input + proj_output)

    def get_config(self):
        config = super().get_config()
        config.update(
            {
                "embed_dim": self.embed_dim,
                "dense_dim": self.dense_dim,
                "num_heads": self.num_heads,
            }
        )
        return config
```

Fig. 7. Arquitectura del Encoder

Para definir la arquitectura del Encoder se define la dimensión de la entrada así como las dimensiones de las capas que lo conforman. Luego se crea la arquitectura propiamente dicha y se trabaja con la función de activación ReLU. Se aplica una normalización de las capas y luego se trabaja el bloque de atención y las conexiones residuales. Éstas son:

- proj_input = self.layernorm_1(inputs + attention_output):

En esta línea, la salida de la capa de atención (attention_output) se suma a la entrada original (inputs). Luego, el resultado se pasa a través de una capa de normalización (self.layernorm_1). Esto representa la primera conexión residual.

- return self.layernorm_2(proj_input + proj_output):

En esta línea, la salida de la capa de proyección (proj_output) se suma a la entrada proyectada (proj_input). Nuevamente, el resultado se pasa a través de una capa de normalización (self.layernorm_2). Esto representa la segunda conexión residual.

En resumen, estas conexiones residuales se implementan mediante la suma de la entrada original a la salida de ciertas capas (capa de atención y capa de proyección) antes de pasar a través de capas de normalización. Estas conexiones residuales son fundamentales en la arquitectura de Transformer y ayudan a mitigar el problema de la desaparición del gradiente en redes profundas.

El Embedding:

```
class PositionalEmbedding(layers.Layer):
    def __init__(self, sequence_length, vocab_size, embed_dim, **kwargs):
        super().__init__(**kwargs)
        self.token_embeddings = layers.Embedding(
            input_dim=vocab_size, output_dim=embed_dim
        )
        self.position_embeddings = layers.Embedding(
            input_dim=sequence_length, output_dim=embed_dim
        )
        self.sequence_length = sequence_length
        self.vocab_size = vocab_size
        self.embed_dim = embed_dim

    def call(self, inputs):
        length = tf.shape(inputs)[-1]
        positions = tf.range(start=0, limit=length, delta=1)
        embedded_tokens = self.token_embeddings(inputs)
        embedded_positions = self.position_embeddings(positions)
        return embedded_tokens + embedded_positions

    def compute_mask(self, inputs, mask=None):
        return tf.math.not_equal(inputs, 0)

    def get_config(self):
        config = super().get_config()
        config.update(
            {
                "sequence_length": self.sequence_length,
                "vocab_size": self.vocab_size,
                "embed_dim": self.embed_dim,
            }
        )
        return config
```

Fig. 8. Arquitectura del Embedding

El código define una capa de incrustación posicional (PositionalEmbedding) en el contexto de un modelo Transformer. La incrustación posicional se utiliza en los modelos Transformer para dotar al modelo de información sobre la posición relativa de las palabras en una secuencia y así poder armar o trabajar con oraciones o frases coherentes. La capa PositionalEmbedding se inicializa con la longitud de la secuencia (sequence_length), el tamaño del vocabulario (vocab_size), y la dimensión de la incrustación (embed_dim). Utiliza dos capas de incrustación (Embedding): una para representar las incrustaciones de los tokens y otra para representar las incrustaciones posicionales.
El método call toma las entradas (inputs), que son secuencias de tokens, y calcula las incrustaciones posicionales. Utiliza la capa de incrustación de tokens (token_embeddings) para obtener las incrustaciones de los tokens y la capa de incrustación posicional (position_embeddings) para obtener las incrustaciones posicionales. Luego, suma estas dos incrustaciones para obtener las incrustaciones finales que incluyen información sobre la posición relativa de los tokens.
Este método devuelve una máscara que es True para las posiciones donde los valores de entrada no son iguales a cero y False donde son cero. Esto se utiliza para ignorar el cero en el cálculo de atención, ya que el valor cero podría representar relleno en secuencias de longitud variable.

El Decoder:

```
class TransformerDecoder(layers.Layer):
    def __init__(self, embed_dim, latent_dim, num_heads, **kwargs):
        super().__init__(**kwargs)
        self.embed_dim = embed_dim
        self.latent_dim = latent_dim
        self.num_heads = num_heads
        self.attention_1 = layers.MultiHeadAttention(
            num_heads=num_heads, key_dim=embed_dim
        )
        self.attention_2 = layers.MultiHeadAttention(
            num_heads=num_heads, key_dim=embed_dim
        )
        self.dense_proj = keras.Sequential(
            [
                layers.Dense(latent_dim, activation="relu"),
                layers.Dense(embed_dim),
            ]
        )
        self.layernorm_1 = layers.LayerNormalization()
        self.layernorm_2 = layers.LayerNormalization()
        self.layernorm_3 = layers.LayerNormalization()
        self.add = layers.Add()  # instead of `+` to preserve mask
        self.supports_masking = True
```

```
def call(self, inputs, encoder_outputs, mask=None):
    attention_output_1 = self.attention_1(
        query=inputs, value=inputs, key=inputs, use_causal_mask=True
    )
    out_1 = self.layernorm_1(self.add([inputs, attention_output_1]))

    attention_output_2 = self.attention_2(
        query=out_1,
        value=encoder_outputs,
        key=encoder_outputs,
    )
    out_2 = self.layernorm_2(self.add([out_1, attention_output_2]))

    proj_output = self.dense_proj(out_2)
    return self.layernorm_3(self.add([out_2, proj_output]))

def get_config(self):
    config = super().get_config()
    config.update(
        {
            "embed_dim": self.embed_dim,
            "latent_dim": self.latent_dim,
            "num_heads": self.num_heads,
        }
    )
    return config
```

Fig. 9. Arquitectura del Decoder

La capa del decoder recibe la nueva representación de la secuencia entregada por el encoder junto con la secuencia target hasta el momento. El decoder busca predecir las siguientes palabras en la secuencia del momento siguiente. Se utiliza un enmascaramiento causal (use_causal_mask=True en la primera capa de atención del TransformerDecoder) dado que el TransformerDecoder ve las secuencias completas a la vez y, por lo tanto, debe asegurarse de que sólo use información de los tokens objetivo 0 a N al predecir el token N+1 (de lo contrario, podría usar información del futuro, lo que daría como resultado un modelo que no puede utilizarse en el momento de la inferencia o con datos desconocidos).
Las conexiones residuales son las siguientes:

- out_1 = self.layernorm_1(self.add([inputs, attention_output_1]))

En esta línea, la salida de la primera capa de atención (attention_output_1) se suma a la entrada original (inputs). Luego, el resultado se pasa a través de una capa de normalización (self.layernorm_1). Esto representa la primera conexión residual.

- out_2 = self.layernorm_2(self.add([out_1, attention_output_2]))

En esta línea, la salida de la segunda capa de atención (attention_output_2) se suma a la salida intermedia de la primera capa de atención (out_1). Nuevamente, el resultado se pasa a través de una capa de normalización (self.layernorm_2). Esto representa la segunda conexión residual.

- return self.layernorm_3(self.add([out_2, proj_output]))

En esta línea, la salida de la capa de proyección (proj_output) se suma a la salida intermedia de la segunda capa de atención (out_2). Luego, el resultado se pasa a través de una capa de normalización (self.layernorm_3). Esto representa la tercera conexión residual.

Estas conexiones residuales se implementan mediante la suma de la salida de ciertas capas a la entrada original antes de pasar a través de capas de normalización. Estas conexiones son esenciales en la arquitectura Transformer para facilitar el entrenamiento de redes profundas en tareas de procesamiento del lenguaje natural.

Luego debe ensamblarse el modelo para tener la arquitectura completa, lo que se hace de la siguiente manera:

```
embed_dim = 256
latent_dim = 2048
num_heads = 8

#las entradas al encoder
encoder_inputs = keras.Input(shape=(None,), dtype="int64", name="encoder_inputs")
x = PositionalEmbedding(sequence_length, vocab_size, embed_dim)(encoder_inputs)
#las salidas del encoder
encoder_outputs = TransformerEncoder(embed_dim, latent_dim, num_heads)(x)
encoder = keras.Model(encoder_inputs, encoder_outputs)

#las entradas al decoder
decoder_inputs = keras.Input(shape=(None,), dtype="int64", name="decoder_inputs")
encoded_seq_inputs = keras.Input(shape=(None, embed_dim), name="decoder_state_inputs")
x = PositionalEmbedding(sequence_length, vocab_size, embed_dim)(decoder_inputs)
x = TransformerDecoder(embed_dim, latent_dim, num_heads)(x, encoded_seq_inputs)
x = layers.Dropout(0.5)(x)
#las salidas del decoder
decoder_outputs = layers.Dense(vocab_size, activation="softmax")(x)
decoder = keras.Model([decoder_inputs, encoded_seq_inputs], decoder_outputs)

decoder_outputs = decoder([decoder_inputs, encoder_outputs])
#modelo del transformer
transformer = keras.Model(
    [encoder_inputs, decoder_inputs], decoder_outputs, name="transformer"
)
```

Fig. 10. Armado de la arquitectura completa

Con lo anterior se define el modelo completo de la red.

Al momento del entrenamiento del modelo se trabaja con lo siguiente:

```
epochs = 40  # This should be at least 30 for convergence

transformer.summary()
plot_model(transformer, to_file='Transformer.jpg', show_shapes=True)
transformer.compile(
    "rmsprop", loss="sparse_categorical_crossentropy", metrics=["accuracy"]
)
transformer_train=transformer.fit(train_ds, epochs=epochs, validation_data=val_ds)
```

Fig. 11. Summary, compilación y entrenamiento del modelo

La arquitectura final del modelo es la que se presenta en el esquema y cuadro a continuación:

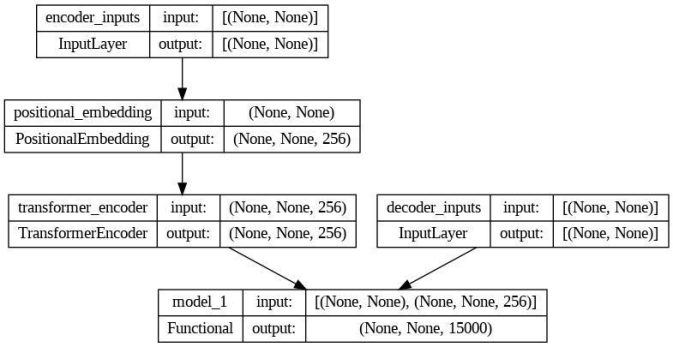


Fig. 12. Arquitectura completa del modelo

```
Model: "transformer"
__________________________________________________________________________________________________
 Layer (type)                Output Shape              Param #   Connected to
==================================================================================================
 encoder_inputs (InputLayer  [(None, None)]            0         []
 )

 positional_embedding (Posi  (None, None, 256)         3845120   ['encoder_inputs[0][0]']
 tionalEmbedding)

 decoder_inputs (InputLayer  [(None, None)]            0         []
 )

 transformer_encoder (Trans  (None, None, 256)         3155456   ['positional_embedding[0][0]']
 formerEncoder)

 model_1 (Functional)        (None, None, 15000)       1295964   ['decoder_inputs[0][0]',
                                                       0          'transformer_encoder[0][0]']

==================================================================================================
Total params: 19960216 (76.14 MB)
Trainable params: 19960216 (76.14 MB)
Non-trainable params: 0 (0.00 Byte)
```

Fig. 13. Arquitectura y cantidad de parámetros del modelo

V. Resultados.

Luego de 40 épocas de entrenamiento se obtienen las siguientrs gráficas para la función de accuracy y la función de pérdida del modelo.

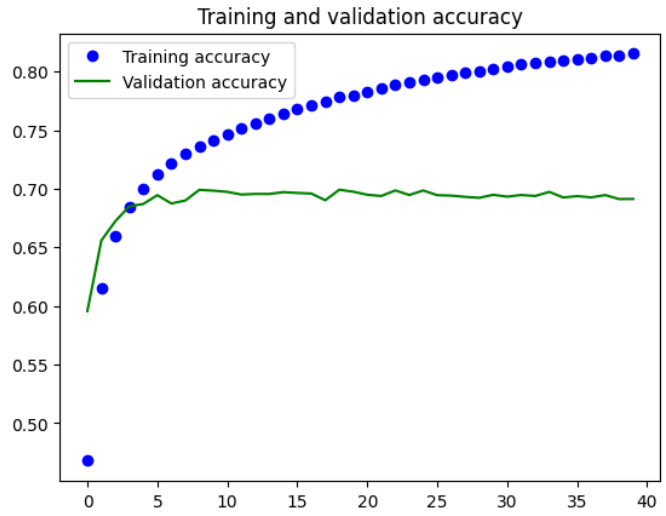


Fig. 14. Evolución del accuracy del modelo a lo largo de las 40 épocas del entrenamiento.

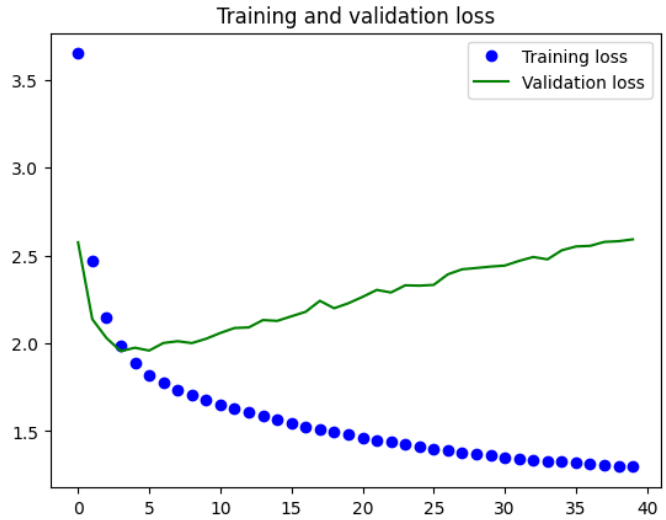


Fig. 15. Evolución de la función de pérdida del modelo a lo largo de las 40 épocas del entrenamiento.

Es de valor destacar que luego de 40 épocas el modelo fue capaz de traducir relativamente bien una abreviación utilizada en el idioma inglés como lo es “ASAP” (as soon as possible), mientras que cuando el modelo se entrenó con 3 épocas para validar su funcionamiento la traducción no fue correcta.

```
print(decode_sequence("I want to finish this asap"))
```

```
[start] quiero terminar esto tan pronto como sea como sea [end]
```

Fig. 16. Traducción de una frase con una sigla

De igual manera, al probar con otras frases, el modelo no reconoce determinadas palabras, por lo cual aún son necesarias más épocas de entrenamiento para poder seguir potenciando su funcionamiento.

```
print(decode_sequence("Once you choose hope, anything's possible"))
```

```
[start] una vez [UNK] que está [UNK] de algo es posible [end]
```

Fig. 17. Primer ejemplo de traducción con error

```
print(decode_sequence("Doubt that the stars are fire,\
                       doubt that the sun moves, doubt truth to be a lie,\
                       but never doubt I love you"))
```

```
[start] si duda las estrellas son [UNK] que el sol [UNK] al [UNK] y nunca se [UNK] pero nunca [end]
```

Fig. 18. Segundo ejemplo de traducción con error.

En ambos casos se necesita reentrenar el modelo para obtener una traducción adecuada.

Link de acceso al código: https://colab.research.google.com/drive/1tZYIjtrbPmKcvEy50jOa_iPmfMeIwhdx?usp=sharing

VI. Referencias

[1] Transformador: https://medium.com/inside-machine-learning/what-is-a-transformer-d07dd1fbec04

[2] Modelo Transformer: https://la.blogs.nvidia.com/2022/04/19/que-es-un-modelo-transformer/

[3] Modelos Seq2Seq: https://medium.com/inside-machine-learning/what-is-a-transformer-d07dd1fbec04

[4] Los transformadores en IA: https://blog.dinobrain.ai/transformadores-en-inteligencia-artificial/

[5] Embedding y NLP: https://es.linkedin.com/pulse/embedding-y-nlp-roxana-brites#:~:text=Los%20embeddings%20facilitan%20el%20aprendizaje,en%20el%20espacio%20de%20embedding.

[6] Natural Language Processing (NLP): https://www.aprendemachinelearning.com/procesamiento-del-lenguaje-natural-nlp/#:~:text=%C2%BFQu%C3%A9%20es%20Natural%20Language%20Processing,Aprendizaje%20Autom%C3%A1tico)%20y%20la%20ling%C3%BC%C3%ADstica. .

[7] Multi Head Attention: https://paperswithcode.com/method/multi-head-attention