Universidad de Sevilla

Escuela Técnica Superior de Ingeniería Informática

Transformer como clasificador de textos

Marina Márquez Macías
Abril 2024

Índice general

Capítulo 1

Introducción

En 2017, a través del artículo *"Attention is all you need"* escrito por Ashish Vaswani, se presentó por primera vez la estructura de red neuronal de tipo "transformer". Esto se dio debido a que las redes neuronales recurrentes usadas anteriormente presentaban ciertos problemas a la hora de abordar diferentes tareas en el procesamiento de secuencias. El principal inconveniente que presentan estas redes es que no recuerdan las palabras que están a larga distancia, puesto que al ser secuenciales tienen un paralelismo muy limitado, y en los lenguajes naturales la relación entre palabras es a larga distancia [3].

Por ello y otros probelmas, se introdujo el modelo **Transformer**, un modelo de aprendizaje profundo con arquitectura de red neuronal codificador-decodificador diseñado para manejar datos secuenciales principalmente en el área del procesamiento del lenguaje natural (PLN), gracias a su capacidad para introducir los textos en paralelo y a su mecanismo de atención (*self-attention*), lo cual se explica con más detalle en esta memoria. Son capaces de capturar dependencias a largo plazo en datos secuenciales, mejorando así los modelos de redes neuronales recurrentes [1].

En este trabajo se va a explicar cómo funciona el modelo Transformer y se va a realizar una clasificación de textos utilizando este modelo a través del lenguaje de programación Python y con ayuda del dataset Reuters de keras.

Capítulo 2

Clasificación de textos con Transformer

En este capítulo se va a tratar de describir el funcionamiento del modelo Transformer (la arquitectura *encoder-decoder* y el mecanismo de *self-attention*), y se creará un clasificador de textos a partir de este modelo en Python.

2.1. Descripción del modelo Transformer

Como se ha comentado anteriormente, el modelo Transformer permite pasar los textos en paralelo, es decir, en vez de relacionar una palabra con la anterior únicamente, va a aprender a relacionar palabras independientemente de la distancia a la que estén. Esto hace que se pierda la información posicional de la palabra. Para solucionar este problema, se introduce el *Positional Encoding*, esto es, añadir a la información de la palabra un identificador de su posición.

2.1.1. Positional Encoding

Para empezar, se asocian valores numéricos a las palabras, los cuales se denominan **tokens**, pueden ser uno o más de uno para cada palabra. A esos tokens se le asignan **embeddings** por cada palabra (de la misma longitud prefijada) usando Word2Vec, lo que logra agrupar las palabras de manera automática en un espacio multidimensional. Una vez que tenemos esos vectores (embeddings) representativos de cada palabra, se debe añadir la información posicional a cada una de ellas, y para ello sumaremos a los embeddings otro vector que será el **positional encoder** de cada palabra, es decir, marcará su posición en la frase.

Este positional encoder debe tener la misma longitud prefijada que se usó para los embeddings, y además debe cumplir una serie de propiedades, que son las siguientes:

- Debe asignar el mismo vector para representar la misma posición en una frase.
- La distancia entre vectores debe ser consistente independientemente de la longitud de las frases.
- El modelo debe aprender a generalizar a frases más largas que no se han visto anteriormente sin ningún esfuerzo.
- Deben ser calculados de manera determinista.

La función que nos proporciona el positional encoding de cada palabra es la siguiente:

$$\vec{p_t}^{(i)} = f(t)^{(i)} := \begin{cases} sen(w_k \cdot t) & si \quad i = 2k \\ \\ cos(w_k \cdot t) & si \quad i = 2k+1 \end{cases}$$

donde $\vec{p_t}^{(i)}$ es el positional encoding, i representa cada coordenada del vector representante de la palabra, d es la dimensión del encoding (que tiene que ser par), y

$$w_k = \frac{1}{10000^{2k/d}}$$

En resumen, lo que se hace es tokenizar todas las palabras para que la máquina pueda operar con ellas, seguidamente a cada palabra tokenizada se le asigna un embedding (vector representante de la palabra) de una longitud prefijada (la misma para todos los embeddings) y a ese embedding se le suma un vector con la información posicional de la palabra, el cual es el positional encoding. Una vez que se tenga esa suma, se introducen todas las palabras en paralelo siguiendo la arquitectura encoder-decoder, la cual se explica con detalle a continuación.

2.1.2. Encoder

El encoder logra procesar en paralelo y está formado por varias capas.

Primero, hay una capa de Self-Attention, un mecanismo de atención. Este mecanismo va a modificar la representación vectorial de una palabra en función de las palabras que la acompañan. Agrupa las palabras en función de su contexto. El mecanismo se denomina self-attention porque también se aplica a una palabra sobre sí misma.

Para poder calcular la atención, se utilizan 3 matrices llamadas "Query-Key-Value" (Q, K, V) que van a operar siguiendo una fórmula que nos devuelve un "score" o puntaje de atención, un escalar por cada par de tokens.

En la matriz Q de Query tendremos los tokens (su embedding) que estamos evaluando. En la matriz K de Key tendremos los tokens nuevamente, como claves del diccionario. En la matriz V de Value tendremos todos los tokens (su embedding) "de salida". El resultado de la atención será aplicar la siguiente fórmula:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

Esta matriz resultante presenta un escalar por cada par de palabras. Dará un valor positivo y mayor cuanto más relacionadas estén las palabras en un contexto.

En el modelo de Transformer se va a utilizar un mecanismo de atención Multi-Cabeza (Multi-head Self-Attention Mechanism). Después de calcular la atención en cada cabeza, se concatenan las matrices de salida y se multiplican por una matriz de pesos adicional W_0. La matriz final captura información sobre todas las cabezas de atemción.

Después de la capa de atención, le sigue una capa de normalización para que el rango de valores entre capas no cambie bruscamente.

A continuación, se añade una capa de red feed forward que aprende a aplicar procesos no lineales, y, por último, se añade una capa más de normalización que realiza una conexión residual [2].

La conexión residual es una técnica que permite que la entrada de una capa se combine con la salida de esa misma capa en lugar de que cada capa transforme directamente su entrada en su salida. Esto facilita el entrenamiento de redes neuronales profundas. La conexión residual en Transformers se aplica típicamente en la capa de atención o en las capas de feedforward dentro de cada bloque de atención. Esto ayuda a mantener la información relevante de la entrada original a medida que pasa a través de múltiples capas de atención y feedforward.

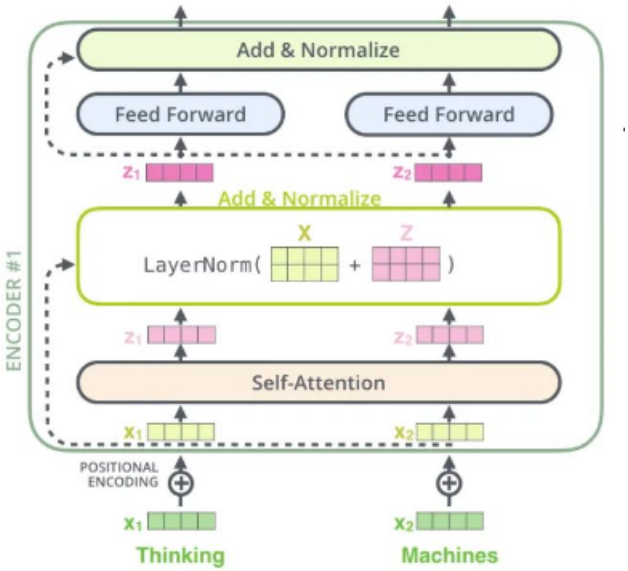

Figura 2.1: Encoder

2.1.3. Decoder

En el decodificador se usa como entrada también la salida del codificador. Por ejemplo, si nos encontramos en una tarea de traducción, y la entrada del encoder es "Hola amigos", la salida será "Hello friends", y estas dos serán la entrada del decodificador.

Para empezar, encontramos una capa de Masked Multi-Head Attention, la cual consiste en enmascarar parte triangular superior de la matriz de atención al calcularla para no adelantar información futura que el output no podría tener en su momento. Más claramente, en el encoder, cuando se hace self-attention se tiene un valor de atención para cada par de tokens. Sin embargo, al calcular las salidas de una frase por ejemplo, se va calculando palabra a palabra. Por tanto, no se debería tener la correlación entre la primera palabra y la última por ejempo, pues aún no se ha calculado esa palabra como salida. Por ello se usa la capa de Masked Multi-Head Attention. Después de esta se aplica de nuevo una capa de normalización.

A continuación, aplicamos una capa de Multi-Head Attention seguida de otra capa de normalización, luego una capa de red feed forward y la conexión residual.

Por último, se aplica una capa lineal seguida de una capa softmax, lo que da una distribución de probabilidad entre todas las palabras, y la palabra con más

probabilidad será la salida. Se puede ver una representación del encoder-decoder en la Figura 2.2.

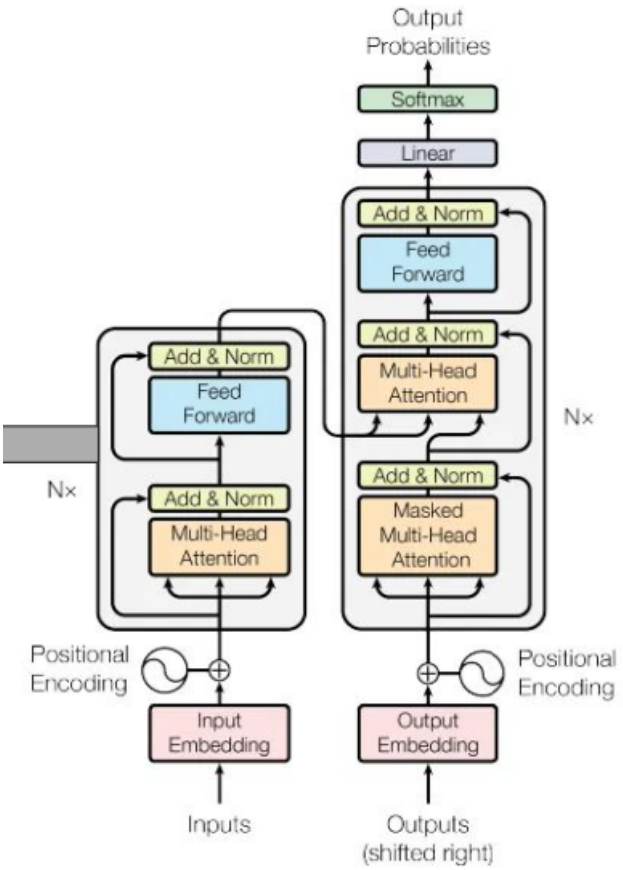

Figura 2.2: Encoder y decoder

2.2. Transformer como clasificador de textos

En este apartado se va a trabajar con el lenguaje de programación Python. Se va a importar el dataset Reuters de Keras, que consta de 11228 documentos de noticias y artículos, y se etiquetan en una de 46 categorías posibles. Por tanto, estamos ante una tarea de clasificación multiclase.

Los 11228 documentos han sido preprocesados y transformados en una representación numérica como vectores. La distribución de las clases no es uniforme, es decir, algunas categorías pueden tener más instacias que otras.

2.2.1. Configuraciones

Para empezar, importamos en Python las bibliotecas y módulos que se van a utilizar para llevar a cabo esta tarea:

```
!pip install keras --upgrade
import keras
from keras import ops
from keras import layers
import numpy as np
from tensorflow.keras.utils import to_categorical
```

La biblioteca *Keras* es una biblioteca de aprendizaje profundo de código abierto usada para construir y entrenar modelos de redes neuronales. El módulo *ops* de *Keras* consta de operaciones y funciones adicionales para trabajar con tensores y operaciones de bajo nivel. El módulo *layers* de *Keras* proporciona una amplia gama de capas predefinidas que se pueden usar para construir redes neuronales (capas densas, convolucionales, recurrentes...). NumPy es una librería de Python especializada en el cálculo numérico y el análisis de datos, especialmente para un gran volumen de datos. Por último se importa la función *to_categorical* del submódulo *utils* de *Keras* en *TensorFlow*. Esta función se utiliza para convertir etiquetas de clases enteras en una representación de matriz categórica (one-hot encoding), que es buena para entrenar modelos de clasificación multiclase.

2.2.2. Implementación del encoder

Se va a implementar ahora un bloque transformer como una capa, que va a representar el encoder del tranformer. El código para realizar esto es el siguiente:

```
class TransformerBlock(layers.Layer):
    def __init__(self, embed_dim, num_heads, ff_dim, rate=0.1):
        super().__init__()
        self.att = layers.MultiHeadAttention(num_heads=num_heads,key_dim=embed_dim)
     #capa de atecion multicabeza
        self.ffn = keras.Sequential([layers.Dense(ff_dim, activation="relu"),
    layers.Dense(embed_dim),]) #red feedforward
        self.layernorm1 = layers.LayerNormalization(epsilon=1e-6) #normalizacion
        self.layernorm2 = layers.LayerNormalization(epsilon=1e-6) #normalizacion
        self.dropout1 = layers.Dropout(rate) #dropout
        self.dropout2 = layers.Dropout(rate) #dropout

    def call(self, inputs):
        attn_output = self.att(inputs, inputs) #mecanismo de atencion
        attn_output = self.dropout1(attn_output) #dropout
        out1 = self.layernorm1(inputs + attn_output) #conexion residual
        ffn_output = self.ffn(out1) #red feedforward
        ffn_output = self.dropout2(ffn_output) #dropout
        return self.layernorm2(out1 + ffn_output) #conexion residual
```

Para empezar, se define una clase llamada `'TransformerBlock'` que hereda de las capas de Keras. Para definir esta clase, hace falta definir el constructor y la llamada.

Constructor de la clase `'TransformerBlock'`

El método `__init__` es el constructor de la clase, y toma los siguientes cuatro parámetros:

- `embed_dim`: dimensión de los embeddings (representaciones latentes de las palabras).
- `num_heads`: número de cabezas en la capa de atención multi-cabeza.
- `ff_dim`: dimensión de la red neuronal feedforward del encoder.
- `rate`: indica el porcentaje de neuronas que no van a cambiar sus pesos durante el entrenamiento. Su valor por defecto es de 0,1.

Lo que hace este constructor es, en primer lugar, crear la capa de atención multi-cabeza y guardarla en el atributo `att`. Esta capa tendrá tantas cabezas como `num_heads` y la dimensión de clave especificada será `embed_dim`.

En segundo lugar, instancia la red neuronal feedforward usando el módulo *keras.Sequential*, que define capas de manera secuencial. La red está compuesta por

dos capas densas, la primera con función de activación ReLu.

A continuación se presentan dos capas de normalización, la primera se aplicará a los vectores después de pasar por el mecanismo de atención, y la segunda se aplicará después de aplicar la red feedforward.

Por último, se definen dos capas de dropout, una para regularizar la salida de la capa de atención y otra para la salida de la red neuronal. El dropout impide el sobreajuste, al dejar algunas neuronas sin entrenar sus pesos.

Llamada de la clase `'TransformerBlock'`

El método `__call__` especifica el comportamiento del bloque Transformer cuando se llama con un conjunto de datos de entrada.

Primero realiza el mecanismo de atención multi-cabeza con las entradas proporcionadas, luego aplica la primera capa de dropout definida en el constructor a la salida de este mecanismo y aplica una capa de conexión residual. A continuación, aplica la capa de la red neuronal feedforward, vuelve a aplicar una capa de dropout y por último la otra capa de conexión residual. Esta es la salida final del bloque Transformer.

2.2.3. Implementación de los embeddings

En este apartado se van a implementar las representaciones latentes de las palabras de entrada (embeddings) mediante el siguiente código de Python. Se crearán dos capas de embeddings, una para representar vectorialmente los tokens de las palabras, y otra para determinar el vector de posición (positional encoding) de cada una de ellas.

```
1    class TokenAndPositionEmbedding(layers.Layer):
2    def __init__(self, maxlen, vocab_size, embed_dim):
3        super().__init__()
4        self.token_emb=layers.Embedding(input_dim=vocab_size,output_dim=embed_dim)
     #embedding para los tokens
5        self.pos_emb=layers.Embedding(input_dim=maxlen,output_dim=embed_dim) #
     positional encoding
6
7    def call(self, x):
8        maxlen = ops.shape(x)[-1]
9        positions = ops.arange(start=0, stop=maxlen, step=1)
10       positions = self.pos_emb(positions)
11       x = self.token_emb(x)
12       return x + positions
```

Se ha definido una nueva clase llamada `'TokenAndPositionEmbedding'`, que también hereda de las capas de Keras. De nuevo se tiene el constructor de la clase y la llamada.

Constructor de la clase `TokenAndPositionEmbedding`

Este constructor recibe tres parámetros que son los siguientes:

- `maxlen`: indica la longitud máxima de las secuencias de entrada.
- `vocab_size`: es el tamaño del vocabulario. Es decir, el número total de palabras diferentes que pueden aparecer como entradas.
- `embed_dim`: indica la dimensión de las representaciones latentes (vectores/embeddings) que se van a crear.

En primer lugar, se crea una capa de embedding que representa a los tokens (palabras) como vectores de dimensión `embed_dim`, y, a continuación, se crea otra capa de embedding también para asignar la información posicional a los anteriores (los positional encodings).

Llamada de la clase `TokenAndPositionEmbedding`

La llamada toma un vector o matriz `x` como entrada. Primero, se calcula el parámetro `maxlen` como la anchura de `x`, es decir, calcula de entre todas las frases de entrada cuál es la que tiene una mayor longitud y asigna el valor de dicha longitud al parámetro `maxlen`.

En segundo lugar se crea un tensor de posiciones que representa las posiciones de las palabras en las frases, desde 0 hasta `maxlen` − 1 y a continuación aplica la capa de embedding creada en el constructor para crear los positional encodings de las palabras.

Luego, transforma los tokens para obtener su representación en forma de vector aplicando la primera capa de embedding definida en el constructor.

Por último, devuelve la suma de las salidas de las dos capas aplicadas, es decir, de las representaciones de los tokens y los positional encodings.

2.2.4. Preparación del conjunto de datos

Como se ha mencionado anteriormente, el conjunto de datos que se va a usar en este trabajo es el dataset Reuters de Keras, el cual contiene 11228 documentos de textos que se clasifican en 46 categorías diferentes. En este apartado se descarga la conjunto de datos y se realiza un pequeño análisis exploratorio de él:

```
    vocab_size=40000
    maxlen=400
    (x_train, y_train), (x_val, y_val) = keras.datasets.reuters.load_data(num_words
    = vocab_size )
print(len(x_train), "Numero de secuencias de entrenamiento")
print(len(x_val), "Numero de secuencias de validacion")
x_train = keras.utils.pad_sequences(x_train, maxlen=maxlen)
x_val = keras.utils.pad_sequences(x_val, maxlen=maxlen)
y_train_onehot= to_categorical(y_train, 46)
y_val_onehot = to_categorical(y_val,46)
print(np.shape(x_train[10]))
print(np.shape(x_val[0]))
print(set(y_train.tolist()))
print(set(y_train_onehot[:,0].tolist()))
```

```
>8982 Numero de secuencias de entrenamiento
>2246 Numero de secuencias de validacion
>(400,)
>(400,)
>{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45}
>{0.0, 1.0}
```

Se va a importar el dataset teniendo en cuenta solo las 40000 primeras palabras del vocabulario de este conjunto de datos, y la longitud máxima de las frases de entrada será de 400.

`x_train` representa las frases tomadas para el conjunto de entrenamiento, y sus respectivas etiquetas están guardadas en el conjunto `y_train`. Lo mismo se tiene

para el conjunto de validación: `x_val` e `y_val`.

Como estamos ante un problema de clasificación multiclase, lo mejor es transformar las etiquetas a su representación one-hot. Para ello se usa la función `to_categorical`.

Se puede observar que cualquier frase de `x_train` o de `x_val` contiene 400 palabras, ya que es el valor que hemos asignado a `maxlen`. Por otro lado, los conjuntos de test `y_train` e `y_val` tienen como posibles valores los números enteros desde el 0 hasta el 45, ya que hay 46 etiquetas posibles de clasificación. Sin embargo, se puede notar que los posibles valores de `y_train_onehot` y de `y_val_onehot` son 0 y 1, ya que están representados en forma one-hot.

2.2.5. Modelo de clasificación

En este apartado se va a crear el modelo de clasificación de los textos usando la capa de transformer mediante el siguiente código:

```
    embed_dim = 50
    num_heads = 4
    ff_dim = 50
    inputs = layers.Input(shape=(maxlen,))
    embedding_layer = TokenAndPositionEmbedding(maxlen, vocab_size, embed_dim)
    x = embedding_layer(inputs)
    transformer_block = TransformerBlock(embed_dim, num_heads, ff_dim)
    x = transformer_block(x)
    x = layers.GlobalAveragePooling1D()(x)
    x = layers.Dropout(0.05)(x)
    x = layers.Dense(200, activation="sigmoid")(x)
    x = layers.Dropout(0.05)(x)
    outputs = layers.Dense(46, activation="softmax")(x)
    model = keras.Model(inputs=inputs, outputs=outputs)
```

En primer lugar, se asignan los valores de los parámetros `embed_dim`, `num_heads` y `ff_dim` ya explicados anteriormente.

A continuación, se implementan los embeddings de las palabras junto con su información posicional, y se hacen pasar por el bloque transformer (encoder) que se definió en el apartado 2.2.2.

Luego, se pasa por una capa de pooling y se aplica un dropout con un porcentaje de pesos no entrenados de un 0,5 por ciento. Después, se aplica una capa densa con función de activación sigmoide y 200 neuronas.

Se vuelve a aplicar dropout de nuevo con un porcentaje del $0,5$ por ciento de pesos sin entrenar y para finalizar se aplica una capa softmax con 46 neuronas, para poder clasificar el texto mediante una distribución de probabilidad.

2.2.6. Entrenamiento y evaluación del modelo

Para evaluar el rendimiento del modelo y las pérdidas (valores de *accuracy* y *loss*), primero se entrena el modelo asignando como función de pérdidas la función `categorical_crossentropy` ya que estamos en un problema de clasificación multiclase, y después se evalúa el modelo en el caso de entrenamiento y validación con 5 pasos. Esto se puede observar en el siguiente código:

```
    model.compile(optimizer= keras.optimizers.Adam(learning_rate=0.0001), loss='
    categorical_crossentropy', metrics=["accuracy"])
history = model.fit(x_train, y_train_onehot, batch_size = 32 , epochs=5,
    validation_data=(x_val, y_val_onehot))
```

```
Epoch 1/5
281/281 ━━━━━━━━━━━━━━━━━━━━ 30s 66ms/step - accuracy: 0.1766 - loss: 3.3621 - val_accuracy: 0.3865 - val_loss: 2.3345
Epoch 2/5
281/281 ━━━━━━━━━━━━━━━━━━━━ 19s 16ms/step - accuracy: 0.3741 - loss: 2.3127 - val_accuracy: 0.4003 - val_loss: 2.1949
Epoch 3/5
281/281 ━━━━━━━━━━━━━━━━━━━━ 5s 17ms/step - accuracy: 0.4353 - loss: 2.1606 - val_accuracy: 0.5383 - val_loss: 1.8652
Epoch 4/5
281/281 ━━━━━━━━━━━━━━━━━━━━ 4s 16ms/step - accuracy: 0.5521 - loss: 1.8100 - val_accuracy: 0.6238 - val_loss: 1.6341
Epoch 5/5
281/281 ━━━━━━━━━━━━━━━━━━━━ 5s 17ms/step - accuracy: 0.6364 - loss: 1.5681 - val_accuracy: 0.6331 - val_loss: 1.5468
```

Figura 2.3: Evaluación del clasificador

El resultado de aplicar este código se puede observar en la Figura 2.3.

Nótese que las medidas de rendimiento son bastante bajas, por lo que se va a probar a modificar ciertos parámetros para buscar una mejoría.

2.2.7. Modificaciones

Ahora se van a modificar algunos valores de los parámetros usados para intentar mejorar el rendimiento del modelo, ya que el que hemos obtenido no es muy bueno.

Primera modificación

El primer cambio que vamos a realizar es cambiar las funciones de activación tanto de la primera capa de feedforward del bloque transformer como de la capa de feedforward del modelo clasificador que hemos creado. En el bloque transformer, cambiamos la función de activación ReLu por una sigmoide, por lo que el bloque queda de la siguiente manera:

```
class TransformerBlock(layers.Layer):
    def __init__(self, embed_dim, num_heads, ff_dim, rate=0.1):
        super().__init__()
        self.att = layers.MultiHeadAttention(num_heads=num_heads,key_dim=embed_dim)
        self.ffn = keras.Sequential([layers.Dense(ff_dim, activation="sigmoid"),
    layers.Dense(embed_dim),]) #red feedforward, cambio relu por sigmoid
        self.layernorm1 = layers.LayerNormalization(epsilon=1e-6)
        self.layernorm2 = layers.LayerNormalization(epsilon=1e-6)
        self.dropout1 = layers.Dropout(rate)
        self.dropout2 = layers.Dropout(rate)

    def call(self, inputs):
        attn_output = self.att(inputs, inputs)
        attn_output = self.dropout1(attn_output)
        out1 = self.layernorm1(inputs + attn_output)
        ffn_output = self.ffn(out1)
        ffn_output = self.dropout2(ffn_output)
        return self.layernorm2(out1 + ffn_output)
```

En el modelo de clasificación, cambiamos la función sigmoide por una Relu en la capa feedforward, y la implementación sería la siguiente:

```
    embed_dim = 50
num_heads = 4
ff_dim = 50

inputs = layers.Input(shape=(maxlen,))
embedding_layer = TokenAndPositionEmbedding(maxlen, vocab_size, embed_dim)
x = embedding_layer(inputs)
transformer_block = TransformerBlock(embed_dim, num_heads, ff_dim)
x = transformer_block(x)
x = layers.GlobalAveragePooling1D()(x)
x = layers.Dropout(0.05)(x)
x = layers.Dense(200, activation="relu")(x) #cambio sigmoid por relu
x = layers.Dropout(0.05)(x)
outputs = layers.Dense(46, activation="softmax")(x)

model = keras.Model(inputs=inputs, outputs=outputs)
```

```
Epoch 1/5
281/281 ━━━━━━━━━━━━━━━━━━━━ 30s 61ms/step - accuracy: 0.3262 - loss: 2.7806 - val_accuracy: 0.3940 - val_loss: 2.2313
Epoch 2/5
281/281 ━━━━━━━━━━━━━━━━━━━━ 5s 18ms/step - accuracy: 0.4176 - loss: 2.1931 - val_accuracy: 0.5169 - val_loss: 1.9955
Epoch 3/5
281/281 ━━━━━━━━━━━━━━━━━━━━ 5s 17ms/step - accuracy: 0.5330 - loss: 1.8978 - val_accuracy: 0.5739 - val_loss: 1.7491
Epoch 4/5
281/281 ━━━━━━━━━━━━━━━━━━━━ 5s 17ms/step - accuracy: 0.6097 - loss: 1.6349 - val_accuracy: 0.6514 - val_loss: 1.5493
Epoch 5/5
281/281 ━━━━━━━━━━━━━━━━━━━━ 5s 17ms/step - accuracy: 0.6716 - loss: 1.4214 - val_accuracy: 0.6754 - val_loss: 1.3777
```

Figura 2.4: Evaluación del modelo tras las primeras modificaciones

El rendimiento final al hacer estos cambios se puede observar en la Figura 2.4. Nótese que el rendimiento ha aumentado pero no considerablemente.

Segunda modificación

Vamos a cambiar los valores de algunos parámetros. Los nuevos valores serán:

- `ff_dim`=30
- `embed_dim`=60
- `rate`=0.01

Además, en la capa de feedforward del modelo clasificador, vamos a añadir 50 neuronas, es decir, habrá 250 neuronas. Al reducir el valor de `rate` se corre el riesgo de sobreajuste. La implementación del modelo queda de la siguiente manera:

```
    embed_dim = 60
num_heads = 4
ff_dim = 30

inputs = layers.Input(shape=(maxlen,))
embedding_layer = TokenAndPositionEmbedding(maxlen, vocab_size, embed_dim)
x = embedding_layer(inputs)
transformer_block = TransformerBlock(embed_dim, num_heads, ff_dim)
x = transformer_block(x)
x = layers.GlobalAveragePooling1D()(x)
x = layers.Dropout(0.01)(x) #rate=0.01
x = layers.Dense(250, activation="sigmoid")(x) #250 neuronas
x = layers.Dropout(0.01)(x) #rate=0.01
outputs = layers.Dense(46, activation="softmax")(x)

model = keras.Model(inputs=inputs, outputs=outputs)
```

El rendimiento del modelo se observa en la Figura 2.5.

```
Epoch 1/5
281/281 ━━━━━━━━━━━━━━━━━━━━ 27s 55ms/step - accuracy: 0.3340 - loss: 2.7886 - val_accuracy: 0.4550 - val_loss: 2.1985
Epoch 2/5
281/281 ━━━━━━━━━━━━━━━━━━━━ 5s 18ms/step - accuracy: 0.4477 - loss: 2.1235 - val_accuracy: 0.5525 - val_loss: 1.8984
Epoch 3/5
281/281 ━━━━━━━━━━━━━━━━━━━━ 5s 17ms/step - accuracy: 0.5697 - loss: 1.7619 - val_accuracy: 0.6318 - val_loss: 1.6247
Epoch 4/5
281/281 ━━━━━━━━━━━━━━━━━━━━ 5s 17ms/step - accuracy: 0.6708 - loss: 1.4537 - val_accuracy: 0.6759 - val_loss: 1.4333
Epoch 5/5
281/281 ━━━━━━━━━━━━━━━━━━━━ 5s 18ms/step - accuracy: 0.7057 - loss: 1.2912 - val_accuracy: 0.6906 - val_loss: 1.3095
```

Figura 2.5: Evaluación del modelo tras la segunda modificación

Se puede notar que en el conjunto de entrenamiento el rendimiento ya es del 70 por ciento, pero aun así sigue siendo pobre.

Tercera modificación

Por último, vamos a modificar el learning rate (factor de aprendizaje). Va a pasar de ser 0.0001 a tener un valor de 0.00011. El problema de aumentar este factor es que hay riesgo de divergencia hacia una solución errónea. El rendimiento del modelo tras hacer esta modificación se puede observar en la Figura 2.6.

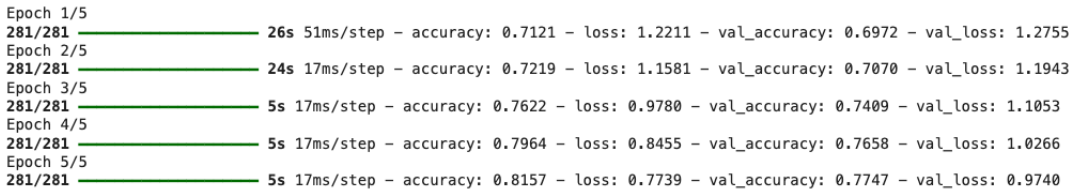

Figura 2.6: Evaluación del modelo tras la última modificación

Se puede comprobar que el rendimiento ha aumentado, y las pérdidas se han reducido considerablemente.

Bibliografía

[1] "¿Qúe es Transformer?" En: (2023). URL: https://gamco.es/glosario/transformers/.

[2] Juan Ignacio Bagnato. *¿Cómo funcionan los transformers?* 2023. URL: https://www.aprendemachinelearning.com/como-funcionan-los-transformers-espanol-nlp-gpt-bert/.

[3] Cristian Santander. "Transformers: Redes Neuronales". En: (2021). URL: https://www.linkedin.com/pulse/transformers-redes-neuronales-cristian-santander/.