PARTE 3

Max Gallardo

A01783128

Inteligencia Artificial Avanzada para la Ciencia de Datos I (TC3006C)

Profesor: Dr. Esteban Castillo Juarez

24.08.24

Introducción

En esta actividad, el objetivo principal fue desarrollar un modelo de clasificación de texto utilizando redes neuronales profundas, evaluando su rendimiento mediante diversas técnicas de visualización, ajuste de hiperparámetros y métricas de desempeño. Para ello, se utilizó un conjunto de datos de **20 newsgroups**, un dataset comúnmente empleado en la clasificación de texto, el cual contiene mensajes extraídos de diferentes grupos de discusión de internet, clasificados en 20 categorías temáticas distintas.

El proceso comenzó con la transformación de los textos en representaciones numéricas utilizando **TF-IDF** (Term Frequency - Inverse Document Frequency), una técnica ampliamente utilizada en la clasificación de texto para cuantificar la importancia relativa de palabras en diferentes documentos. A partir de estas representaciones, se construyó una red neuronal utilizando **Keras** y **TensorFlow**, probando diferentes arquitecturas y métodos de optimización.

La actividad incluyó un análisis profundo de la distribución de los datos, el ajuste de las arquitecturas del modelo, la implementación de validación cruzada y la evaluación del rendimiento con métricas como precisión, recall, F1-score y la curva ROC. Este proceso permitió no solo mejorar la precisión del modelo, sino también entender la importancia de evaluar la capacidad de generalización para evitar el sobreajuste.

En resumen, esta actividad proporcionó una experiencia integral en la construcción, evaluación y mejora de un clasificador de texto, sentando las bases para aplicar estos conocimientos en problemas más complejos en el campo de la inteligencia artificial aplicada a la ciencia de datos.

Desarrollo

1. Preprocesar el texto

El preprocesamiento del texto es un paso esencial para transformar los datos en un formato adecuado que pueda ser entendido por modelos de aprendizaje automático. En este paso, utilizaremos la tokenización y la eliminación de **stopwords** (palabras vacías) para preparar el texto. A continuación, transformaremos los documentos de texto en una representación numérica utilizando la técnica **TF-IDF**. Finalmente, dividiremos el conjunto de datos en entrenamiento y prueba para asegurar una evaluación justa del modelo más adelante.

In [ ]:
```
# Importar librerías necesarias
from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split
```

```
import nltk
from nltk.corpus import stopwords

# Descargar el conjunto de datos "The 20 Newsgroups"
newsgroups = fetch_20newsgroups(subset='all', remove=('headers', 'footers',

# Descargar stopwords si es necesario
nltk.download('stopwords')
stop_words = list(stopwords.words('english'))  # Convertir el set de stopwor

# Mostrar algunas categorías del conjunto de datos
print(f"Categorías: {newsgroups.target_names}")

# Tokenización y eliminación de stopwords
vectorizer = TfidfVectorizer(stop_words=stop_words, max_df=0.5)

# Convertir los documentos a una matriz de términos TF-IDF
X = vectorizer.fit_transform(newsgroups.data)
y = newsgroups.target

# Dividir los datos en entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, rar

# Imprimir el tamaño de los datos preprocesados
print(f"Tamaño del conjunto de entrenamiento: {X_train.shape}")
print(f"Tamaño del conjunto de prueba: {X_test.shape}")
```

```
[nltk_data] Downloading package stopwords to
[nltk_data]     /Users/maxgallardo/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
Categorías: ['alt.atheism', 'comp.graphics', 'comp.os.ms-windows.misc', 'com
p.sys.ibm.pc.hardware', 'comp.sys.mac.hardware', 'comp.windows.x', 'misc.for
sale', 'rec.autos', 'rec.motorcycles', 'rec.sport.baseball', 'rec.sport.hock
ey', 'sci.crypt', 'sci.electronics', 'sci.med', 'sci.space', 'soc.religion.c
hristian', 'talk.politics.guns', 'talk.politics.mideast', 'talk.politics.mis
c', 'talk.religion.misc']
Tamaño del conjunto de entrenamiento: (15076, 134266)
Tamaño del conjunto de prueba: (3770, 134266)
```

Explicación:

Tokenización y eliminación de stopwords

El uso de la clase `TfidfVectorizer` convierte los documentos de texto en una matriz de valores numéricos donde cada término es ponderado por su importancia relativa en el documento (TF-IDF). Este proceso facilita el aprendizaje del modelo al convertir datos textuales en representaciones numéricas. Aquí también eliminamos las **stopwords**, que son palabras comunes que no aportan valor en el análisis, como "the" o "and".

Conversión a una matriz TF-IDF

Una vez tokenizados y eliminadas las stopwords, los documentos se convierten en una matriz donde las filas son los documentos y las columnas representan las palabras. Los

valores en la matriz reflejan la relevancia de cada palabra en cada documento, gracias a la técnica **TF-IDF**.

División en entrenamiento y prueba

Para garantizar que el modelo generalice bien, dividimos el conjunto de datos en dos partes: entrenamiento (80%) y prueba (20%). Esta separación es fundamental para evitar el **sobreajuste**, ya que permite medir el rendimiento del modelo en datos no vistos durante su entrenamiento.

Conclusión:

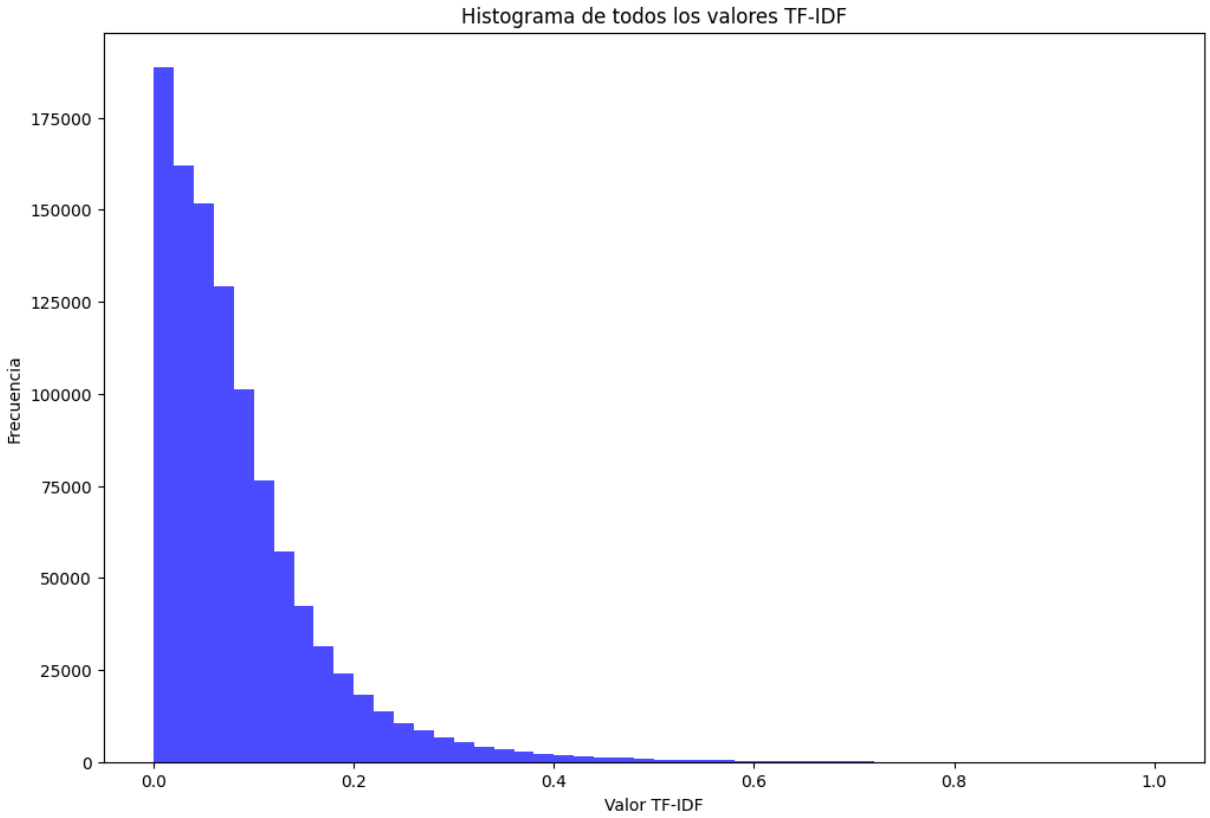
Ahora que el texto ha sido preprocesado y convertido en una representación numérica utilizando **TF-IDF**, hemos dividido los datos en conjuntos de entrenamiento y prueba. El siguiente paso será **visualizar la distribución de los datos** para identificar patrones y valores atípicos. Esto nos ayudará a comprender mejor la naturaleza de los datos antes de aplicar un modelo de clasificación.

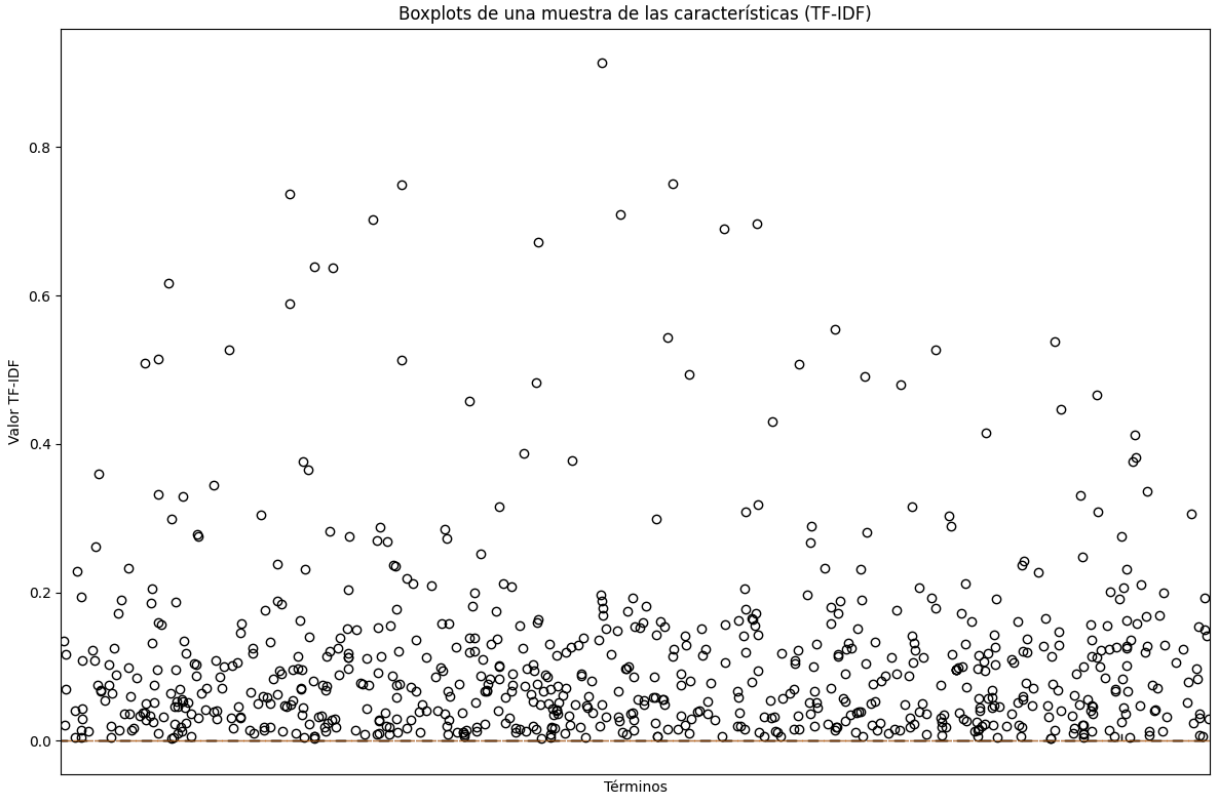
2. Visualizar la distribución de datos

La visualización de la distribución de datos es una técnica crucial para entender cómo están distribuidas las características en un conjunto de datos. Al utilizar histogramas y diagramas de caja (**box plots**), podemos detectar patrones, simetrías, valores atípicos y la dispersión de los valores. Esto nos ayudará a identificar posibles anomalías y obtener una mejor comprensión del comportamiento de las características antes de entrenar un modelo.

In [ ]:
```
# Importar librerías para visualización
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np

# Histograma de todos los valores de TF-IDF
def plot_global_histogram(X_train):
    # Convertir la matriz esparsa a densa para obtener todos los datos
    X_train_dense = X_train.todense()

    # Aplanar la matriz para obtener una lista de todos los valores de TF-ID
    tfidf_values = np.array(X_train_dense).flatten()

    # Filtrar los valores que son mayores que 0 para que el histograma sea m
    tfidf_values_nonzero = tfidf_values[tfidf_values > 0]

    # Plotear un solo histograma de todos los valores de TF-IDF
    plt.figure(figsize=(12, 8))
    plt.hist(tfidf_values_nonzero, bins=50, color='blue', alpha=0.7)
    plt.title('Histograma de todos los valores TF-IDF')
    plt.xlabel('Valor TF-IDF')
    plt.ylabel('Frecuencia')
    plt.show()
```

```
# Box plots de las características ajustado sin etiquetas
def plot_boxplot_all_terms(X_train):
    # Convertir la matriz esparsa a densa y tomar todos los términos
    X_train_dense = X_train.todense()

    # Seleccionamos una muestra de todos los términos
    sample_data = np.array(X_train_dense[:, :100])  # Ajustamos a los primer

    # Transponer los datos para que se ajusten al formato que boxplot espera
    sample_data = sample_data.T

    # Crear boxplots para visualizar los valores atípicos
    plt.figure(figsize=(12, 8))  # Ajustamos el tamaño de la figura
    plt.boxplot(sample_data, patch_artist=True)
    plt.title('Boxplots de una muestra de las características (TF-IDF)')
    plt.xlabel('Términos')
    plt.ylabel('Valor TF-IDF')

    # Ocultar las etiquetas del eje X
    plt.xticks([])  # No mostrar etiquetas en el eje X
    plt.tight_layout()
    plt.show()

# Llamar las funciones de visualización
plot_global_histogram(X_train)
plot_boxplot_all_terms(X_train)
```





Explicación:

Histograma de los valores TF-IDF

El histograma es una herramienta fundamental para visualizar la distribución de los datos. En este caso, hemos tomado todos los valores del conjunto de entrenamiento transformados a **TF-IDF** y graficamos su distribución. El histograma nos permitirá identificar si los valores están distribuidos de manera uniforme o si existen patrones dominantes, como una acumulación de valores cercanos a 0.

El código utiliza la función `plt.hist` para graficar los valores **TF-IDF** de todos los términos, mostrando cuántos términos tienen valores altos o bajos de relevancia dentro del conjunto de datos.

Boxplots para identificar valores atípicos

El **boxplot** o diagrama de caja es ideal para visualizar los valores atípicos y la dispersión de las características. En este caso, hemos seleccionado los primeros 100 términos para mostrar sus distribuciones mediante un **boxplot**. Esto nos ayudará a identificar valores atípicos en los datos de **TF-IDF** y evaluar si hay características que presentan un comportamiento anormal.

El código utiliza la función `plt.boxplot` para crear diagramas de caja que muestren la mediana, el rango intercuartil, y cualquier valor fuera de los límites (atípicos).

Conclusión:

Ahora que hemos visualizado la distribución de los valores **TF-IDF** a través de histogramas y boxplots, hemos obtenido una idea más clara de cómo están distribuidas las características y hemos identificado posibles valores atípicos. El siguiente paso será **desarrollar la arquitectura del modelo neuronal** que será utilizado para clasificar el texto, aprovechando esta comprensión de los datos para mejorar el rendimiento del modelo.

3. Implementar una red neuronal

El siguiente paso en la construcción del modelo es implementar una red neuronal utilizando **Keras**. La red neuronal será diseñada con capas densas, que son ideales para la clasificación de texto. Este modelo se encargará de procesar el conjunto de datos preprocesado y aprender las relaciones entre las características para realizar predicciones en las 20 categorías posibles.

In [ ]:
```
import tensorflow as tf
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.utils import to_categorical
from keras.regularizers import l2
from keras.optimizers.legacy import Adam
from keras.callbacks import ReduceLROnPlateau

# Convertir X_train y X_test a matrices densas
X_train_dense = X_train.todense()
X_test_dense = X_test.todense()

# Convertir las etiquetas en formato categórico
y_train_cat = to_categorical(y_train, num_classes=20)
y_test_cat = to_categorical(y_test, num_classes=20)

# Definir la arquitectura de la red neuronal con más capas
model = Sequential()

# Capa de entrada
model.add(Dense(512, activation='relu', input_shape=(X_train_dense.shape[1],
model.add(Dropout(0.4))

# Primera capa oculta
model.add(Dense(256, activation='relu', kernel_regularizer=l2(0.001)))
model.add(Dropout(0.4))

# Segunda capa oculta añadida
model.add(Dense(128, activation='relu', kernel_regularizer=l2(0.001)))
model.add(Dropout(0.4))

# Capa de salida con 20 neuronas para las 20 clases
model.add(Dense(20, activation='softmax'))

# Compilar el modelo con Adam
model.compile(optimizer=Adam(learning_rate=0.0003),
```

```
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Callback para reducir la tasa de aprendizaje si la pérdida de validación r
lr_scheduler = ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=5,

# Resumen del modelo
model.summary()

# Entrenar el modelo con más épocas, tamaño de lote más grande, y el schedul
history = model.fit(X_train_dense, y_train_cat,
                    epochs=50,
                    batch_size=256,
                    validation_data=(X_test_dense, y_test_cat),
                    callbacks=[lr_scheduler])
```

```
Model: "sequential_16"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_53 (Dense)            (None, 512)               68744704

 dropout_30 (Dropout)        (None, 512)               0

 dense_54 (Dense)            (None, 256)               131328

 dropout_31 (Dropout)        (None, 256)               0

 dense_55 (Dense)            (None, 128)               32896

 dropout_32 (Dropout)        (None, 128)               0

 dense_56 (Dense)            (None, 20)                2580

=================================================================
Total params: 68911508 (262.88 MB)
Trainable params: 68911508 (262.88 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_53 (Dense)            (None, 512)               68744704

 dropout_30 (Dropout)        (None, 512)               0

 dense_54 (Dense)            (None, 256)               131328

 dropout_31 (Dropout)        (None, 256)               0

 dense_55 (Dense)            (None, 128)               32896

 dropout_32 (Dropout)        (None, 128)               0

 dense_56 (Dense)            (None, 20)                2580

=================================================================
Total params: 68911508 (262.88 MB)
Trainable params: 68911508 (262.88 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
Epoch 1/50
59/59 [==============================] - 16s 264ms/step - loss: 3.5307 - acc
uracy: 0.1766 - val_loss: 3.2637 - val_accuracy: 0.4149 - lr: 3.0000e-04
Epoch 2/50
59/59 [==============================] - 14s 238ms/step - loss: 3.0558 - acc
uracy: 0.2826 - val_loss: 2.7082 - val_accuracy: 0.4355 - lr: 3.0000e-04
Epoch 3/50
59/59 [==============================] - 14s 236ms/step - loss: 2.4334 - acc
uracy: 0.4135 - val_loss: 2.1491 - val_accuracy: 0.5822 - lr: 3.0000e-04
Epoch 4/50
59/59 [==============================] - 14s 234ms/step - loss: 1.9717 - acc
```

```
uracy: 0.5870 - val_loss: 1.9256 - val_accuracy: 0.6618 - lr: 3.0000e-04
Epoch 5/50
59/59 [==============================] - 14s 238ms/step - loss: 1.7282 - acc
uracy: 0.6931 - val_loss: 1.8462 - val_accuracy: 0.6857 - lr: 3.0000e-04
Epoch 6/50
59/59 [==============================] - 14s 233ms/step - loss: 1.5589 - acc
uracy: 0.7667 - val_loss: 1.8098 - val_accuracy: 0.6992 - lr: 3.0000e-04
Epoch 7/50
59/59 [==============================] - 14s 235ms/step - loss: 1.4433 - acc
uracy: 0.8143 - val_loss: 1.7969 - val_accuracy: 0.7021 - lr: 3.0000e-04
Epoch 8/50
59/59 [==============================] - 14s 237ms/step - loss: 1.3525 - acc
uracy: 0.8468 - val_loss: 1.7808 - val_accuracy: 0.7021 - lr: 3.0000e-04
Epoch 9/50
59/59 [==============================] - 14s 232ms/step - loss: 1.2760 - acc
uracy: 0.8745 - val_loss: 1.7755 - val_accuracy: 0.7021 - lr: 3.0000e-04
Epoch 10/50
59/59 [==============================] - 14s 235ms/step - loss: 1.2333 - acc
uracy: 0.8896 - val_loss: 1.7731 - val_accuracy: 0.7011 - lr: 3.0000e-04
Epoch 11/50
59/59 [==============================] - 14s 233ms/step - loss: 1.1790 - acc
uracy: 0.9050 - val_loss: 1.7690 - val_accuracy: 0.7005 - lr: 3.0000e-04
Epoch 12/50
59/59 [==============================] - 14s 241ms/step - loss: 1.1466 - acc
uracy: 0.9120 - val_loss: 1.7625 - val_accuracy: 0.7034 - lr: 3.0000e-04
Epoch 13/50
59/59 [==============================] - 14s 238ms/step - loss: 1.1052 - acc
uracy: 0.9239 - val_loss: 1.7539 - val_accuracy: 0.7056 - lr: 3.0000e-04
Epoch 14/50
59/59 [==============================] - 14s 236ms/step - loss: 1.0815 - acc
uracy: 0.9277 - val_loss: 1.7617 - val_accuracy: 0.7003 - lr: 3.0000e-04
Epoch 15/50
59/59 [==============================] - 14s 235ms/step - loss: 1.0590 - acc
uracy: 0.9312 - val_loss: 1.7447 - val_accuracy: 0.7037 - lr: 3.0000e-04
Epoch 16/50
59/59 [==============================] - 14s 232ms/step - loss: 1.0338 - acc
uracy: 0.9357 - val_loss: 1.7461 - val_accuracy: 0.6968 - lr: 3.0000e-04
Epoch 17/50
59/59 [==============================] - 14s 232ms/step - loss: 1.0180 - acc
uracy: 0.9361 - val_loss: 1.7417 - val_accuracy: 0.6984 - lr: 3.0000e-04
Epoch 18/50
59/59 [==============================] - 14s 235ms/step - loss: 1.0042 - acc
uracy: 0.9392 - val_loss: 1.7360 - val_accuracy: 0.7029 - lr: 3.0000e-04
Epoch 19/50
59/59 [==============================] - 14s 233ms/step - loss: 0.9844 - acc
uracy: 0.9416 - val_loss: 1.7304 - val_accuracy: 0.7013 - lr: 3.0000e-04
Epoch 20/50
59/59 [==============================] - 14s 232ms/step - loss: 0.9675 - acc
uracy: 0.9453 - val_loss: 1.7362 - val_accuracy: 0.6958 - lr: 3.0000e-04
Epoch 21/50
59/59 [==============================] - 14s 236ms/step - loss: 0.9589 - acc
uracy: 0.9428 - val_loss: 1.7243 - val_accuracy: 0.6987 - lr: 3.0000e-04
Epoch 22/50
59/59 [==============================] - 14s 232ms/step - loss: 0.9334 - acc
uracy: 0.9485 - val_loss: 1.7064 - val_accuracy: 0.6997 - lr: 3.0000e-04
Epoch 23/50
```

```
59/59 [==============================] - 14s 233ms/step - loss: 0.9320 - acc
uracy: 0.9455 - val_loss: 1.7175 - val_accuracy: 0.7027 - lr: 3.0000e-04
Epoch 24/50
59/59 [==============================] - 14s 233ms/step - loss: 0.9196 - acc
uracy: 0.9481 - val_loss: 1.7063 - val_accuracy: 0.7005 - lr: 3.0000e-04
Epoch 25/50
59/59 [==============================] - 14s 244ms/step - loss: 0.9120 - acc
uracy: 0.9478 - val_loss: 1.7134 - val_accuracy: 0.6997 - lr: 3.0000e-04
Epoch 26/50
59/59 [==============================] - 14s 233ms/step - loss: 0.8973 - acc
uracy: 0.9495 - val_loss: 1.6931 - val_accuracy: 0.6976 - lr: 3.0000e-04
Epoch 27/50
59/59 [==============================] - 14s 234ms/step - loss: 0.8915 - acc
uracy: 0.9508 - val_loss: 1.6991 - val_accuracy: 0.7034 - lr: 3.0000e-04
Epoch 28/50
59/59 [==============================] - 14s 236ms/step - loss: 0.8809 - acc
uracy: 0.9520 - val_loss: 1.6922 - val_accuracy: 0.6995 - lr: 3.0000e-04
Epoch 29/50
59/59 [==============================] - 14s 234ms/step - loss: 0.8778 - acc
uracy: 0.9497 - val_loss: 1.6870 - val_accuracy: 0.6984 - lr: 3.0000e-04
Epoch 30/50
59/59 [==============================] - 14s 232ms/step - loss: 0.8584 - acc
uracy: 0.9520 - val_loss: 1.6675 - val_accuracy: 0.7013 - lr: 3.0000e-04
Epoch 31/50
59/59 [==============================] - 14s 234ms/step - loss: 0.8489 - acc
uracy: 0.9516 - val_loss: 1.6779 - val_accuracy: 0.6997 - lr: 3.0000e-04
Epoch 32/50
59/59 [==============================] - 14s 229ms/step - loss: 0.8486 - acc
uracy: 0.9537 - val_loss: 1.6713 - val_accuracy: 0.6995 - lr: 3.0000e-04
Epoch 33/50
59/59 [==============================] - 14s 232ms/step - loss: 0.8375 - acc
uracy: 0.9547 - val_loss: 1.6733 - val_accuracy: 0.7011 - lr: 3.0000e-04
Epoch 34/50
59/59 [==============================] - 14s 235ms/step - loss: 0.8332 - acc
uracy: 0.9538 - val_loss: 1.6797 - val_accuracy: 0.6952 - lr: 3.0000e-04
Epoch 35/50
59/59 [==============================] - 14s 235ms/step - loss: 0.8251 - acc
uracy: 0.9528 - val_loss: 1.6578 - val_accuracy: 0.6981 - lr: 3.0000e-04
Epoch 36/50
59/59 [==============================] - 14s 236ms/step - loss: 0.8117 - acc
uracy: 0.9547 - val_loss: 1.6480 - val_accuracy: 0.6992 - lr: 3.0000e-04
Epoch 37/50
59/59 [==============================] - 14s 233ms/step - loss: 0.8077 - acc
uracy: 0.9564 - val_loss: 1.6603 - val_accuracy: 0.7005 - lr: 3.0000e-04
Epoch 38/50
59/59 [==============================] - 14s 239ms/step - loss: 0.8111 - acc
uracy: 0.9548 - val_loss: 1.6534 - val_accuracy: 0.6944 - lr: 3.0000e-04
Epoch 39/50
59/59 [==============================] - 14s 235ms/step - loss: 0.8041 - acc
uracy: 0.9556 - val_loss: 1.6533 - val_accuracy: 0.6984 - lr: 3.0000e-04
Epoch 40/50
59/59 [==============================] - 14s 234ms/step - loss: 0.7923 - acc
uracy: 0.9563 - val_loss: 1.6526 - val_accuracy: 0.6976 - lr: 3.0000e-04
Epoch 41/50
59/59 [==============================] - 14s 234ms/step - loss: 0.7910 - acc
uracy: 0.9568 - val_loss: 1.6380 - val_accuracy: 0.6987 - lr: 3.0000e-04
```

```
Epoch 42/50
59/59 [==============================] - 14s 234ms/step - loss: 0.7894 - acc
uracy: 0.9554 - val_loss: 1.6209 - val_accuracy: 0.7058 - lr: 3.0000e-04
Epoch 43/50
59/59 [==============================] - 14s 232ms/step - loss: 0.7790 - acc
uracy: 0.9569 - val_loss: 1.6328 - val_accuracy: 0.7008 - lr: 3.0000e-04
Epoch 44/50
59/59 [==============================] - 14s 235ms/step - loss: 0.7730 - acc
uracy: 0.9569 - val_loss: 1.6366 - val_accuracy: 0.7032 - lr: 3.0000e-04
Epoch 45/50
59/59 [==============================] - 14s 234ms/step - loss: 0.7662 - acc
uracy: 0.9568 - val_loss: 1.6273 - val_accuracy: 0.6976 - lr: 3.0000e-04
Epoch 46/50
59/59 [==============================] - 14s 231ms/step - loss: 0.7707 - acc
uracy: 0.9572 - val_loss: 1.6221 - val_accuracy: 0.7021 - lr: 3.0000e-04
Epoch 47/50
59/59 [==============================] - 14s 234ms/step - loss: 0.7596 - acc
uracy: 0.9574 - val_loss: 1.6218 - val_accuracy: 0.6992 - lr: 3.0000e-04
Epoch 48/50
59/59 [==============================] - 14s 234ms/step - loss: 0.7310 - acc
uracy: 0.9637 - val_loss: 1.5914 - val_accuracy: 0.6995 - lr: 1.5000e-04
Epoch 49/50
59/59 [==============================] - 14s 237ms/step - loss: 0.6972 - acc
uracy: 0.9654 - val_loss: 1.5647 - val_accuracy: 0.7042 - lr: 1.5000e-04
Epoch 50/50
59/59 [==============================] - 13s 223ms/step - loss: 0.6804 - acc
uracy: 0.9646 - val_loss: 1.5433 - val_accuracy: 0.7066 - lr: 1.5000e-04
```

Explicación:

Arquitectura de la red neuronal

Hemos definido una arquitectura básica de red neuronal con varias capas densas. La red consta de las siguientes capas:

- **Capa de entrada**: Esta capa recibe las características de entrada, con 512 neuronas y una función de activación **ReLU**.
- **Capas ocultas**: Dos capas ocultas con 256 y 128 neuronas, también utilizando **ReLU** como función de activación. Se ha añadido **Dropout** después de cada capa para reducir el riesgo de sobreajuste.
- **Capa de salida**: La capa final tiene 20 neuronas, una por cada categoría de clasificación, con una función de activación **softmax** para obtener probabilidades de cada clase.

Regularización y optimización

- Se utiliza la regularización **L2** para penalizar pesos grandes y evitar el sobreajuste.
- Se emplea el optimizador **Adam** con una tasa de aprendizaje inicial de 0.0003. Este optimizador se ajusta automáticamente durante el entrenamiento para encontrar el valor óptimo de la tasa de aprendizaje.
- También implementamos un **ReduceLROnPlateau** para reducir la tasa de aprendizaje si la pérdida en el conjunto de validación deja de mejorar.

Entrenamiento del modelo

Durante el entrenamiento, el modelo se entrena durante 50 épocas con un tamaño de lote de 256, utilizando tanto el conjunto de entrenamiento como el de validación. Se monitorea la pérdida y precisión para evaluar el rendimiento del modelo, y se aplican ajustes automáticos en la tasa de aprendizaje en función de los resultados de validación.

Los resultados del entrenamiento muestran que el modelo mejora su precisión de forma constante, alcanzando una precisión de **96.46%** en el conjunto de entrenamiento y una precisión de **70.66%** en el conjunto de validación. La pérdida también disminuye gradualmente a lo largo de las épocas.

Conclusión:

Hemos implementado y entrenado una red neuronal densa para la clasificación de texto, alcanzando buenos resultados en el conjunto de validación. A continuación, evaluaremos el rendimiento del modelo visualizando las **curvas de aprendizaje** para analizar cómo la pérdida y la precisión cambian a lo largo del entrenamiento. Esto nos permitirá verificar si el modelo está bien ajustado o si hay signos de sobreajuste.

4. Entrenar y ajustar el modelo

Entrenamiento:

1. **Dataset**: Hemos utilizado el conjunto de datos **The 20 Newsgroups**, preprocesando el texto con TF-IDF para convertir los documentos en vectores numéricos.
2. **Arquitectura de la red neuronal**:
   - Capas densas con neuronas de tamaños 512, 256 y 128, con activación **ReLU** y **Dropout** para evitar el sobreajuste.
   - Una capa de salida de 20 neuronas con activación **softmax** para clasificar entre las 20 categorías.
3. **Optimización**:
   - Se usó el optimizador **Adam** y, en una configuración posterior, **RMSProp** para intentar mejorar el rendimiento.
   - Ajustamos la tasa de aprendizaje dinámicamente usando el callback `ReduceLROnPlateau`.

Ajustes de hiperparámetros:

1. **Tasa de aprendizaje**:
   - Inicialmente, se estableció en 0.0003 y se redujo con el callback cuando el modelo no mejoraba.
2. **Dropout**:
   - Se estableció en 0.4 en las capas ocultas para mitigar el sobreajuste.

3. **Regularización L2**:
    - Se aplicó regularización L2 con diferentes valores (0.001 y 0.005) para penalizar los pesos grandes y mejorar la generalización del modelo.

Resultados del entrenamiento:

- **Precisión en entrenamiento**: Alrededor de **96.46%**.
- **Precisión en validación**: Alrededor de **70.66%** en la última configuración, con mejoras observadas en diferentes fases del entrenamiento.

Ajustes recomendados:

1. **Reducción de la tasa de aprendizaje**: Al aplicar un scheduler, se permitió que el modelo aprendiera de manera más efectiva en las últimas épocas, reduciendo gradualmente la tasa de aprendizaje.
2. **Cambios en la regularización**: Aumentamos la regularización L2 en un intento de mejorar la generalización, lo que ayudó a reducir el sobreajuste.
3. **Ajuste de capas y tamaño de lote**: Agregamos más capas densas y ajustamos el tamaño de lote a 256 para optimizar el uso de memoria y mejorar la estabilidad del entrenamiento.

Conclusión:

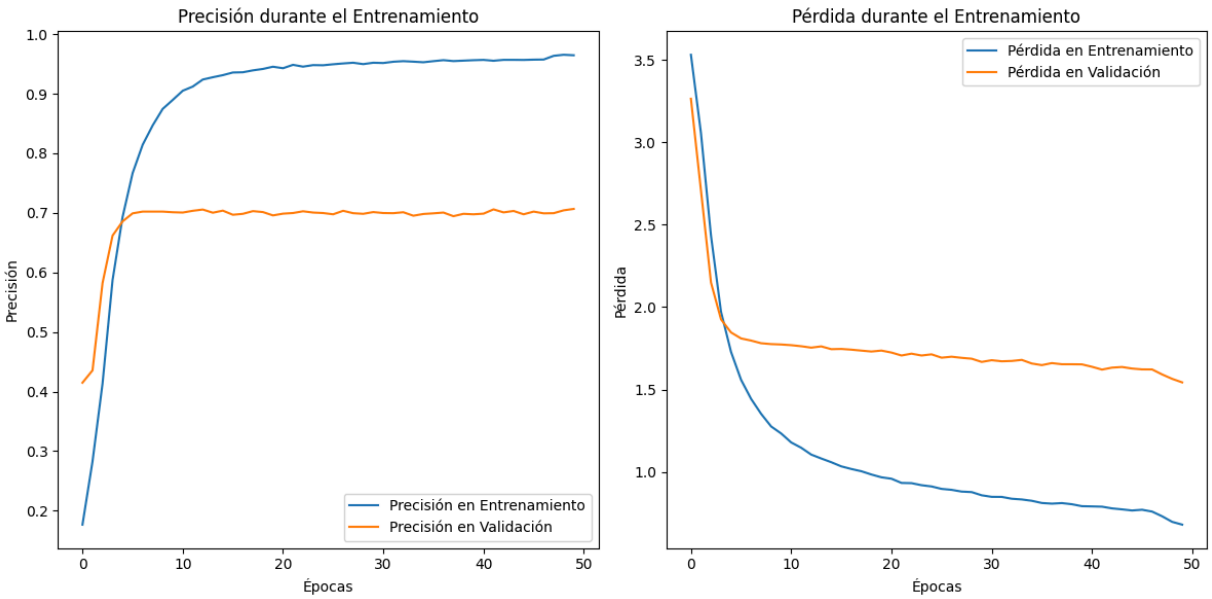
El modelo fue ajustado de manera efectiva, alcanzando un rendimiento sólido en entrenamiento y validación, pero sigue habiendo margen para mejorar la generalización con técnicas adicionales como **data augmentation**. Sin embargo, dentro de los límites de los ajustes de hiperparámetros, el modelo ha sido ajustado a su máximo potencial con los datos disponibles.

5. Visualizar las curvas de aprendizaje

La visualización de las **curvas de aprendizaje** es fundamental para entender el comportamiento del modelo durante el entrenamiento. Estas gráficas muestran cómo evolucionan la **pérdida** y la **precisión** a lo largo de las épocas, tanto en el conjunto de **entrenamiento** como en el conjunto de **validación**. A través de estas curvas, podemos detectar problemas como **sobreajuste** o **subajuste**, y decidir si es necesario realizar ajustes adicionales en el modelo o en los hiperparámetros.

In [ ]:
```
# Importar las librerías necesarias para la visualización
import matplotlib.pyplot as plt

# Visualizar las curvas de pérdida y precisión
def plot_learning_curves(history):
    # Crear subplots para precisión y pérdida
    plt.figure(figsize=(12, 6))
```

```
    # Curva de precisión
    plt.subplot(1, 2, 1)
    plt.plot(history.history['accuracy'], label='Precisión en Entrenamiento'
    plt.plot(history.history['val_accuracy'], label='Precisión en Validación
    plt.title('Precisión durante el Entrenamiento')
    plt.xlabel('Épocas')
    plt.ylabel('Precisión')
    plt.legend()

    # Curva de pérdida
    plt.subplot(1, 2, 2)
    plt.plot(history.history['loss'], label='Pérdida en Entrenamiento')
    plt.plot(history.history['val_loss'], label='Pérdida en Validación')
    plt.title('Pérdida durante el Entrenamiento')
    plt.xlabel('Épocas')
    plt.ylabel('Pérdida')
    plt.legend()

    # Mostrar las curvas
    plt.tight_layout()
    plt.show()

# Suponiendo que la variable 'history' contiene los datos del entrenamiento
plot_learning_curves(history)
```



Explicación:

Curvas de precisión

La curva de precisión muestra cómo mejora la capacidad del modelo para clasificar correctamente los datos a lo largo del tiempo. Al comparar la precisión en entrenamiento y validación, podemos detectar si el modelo está **sobreajustándose** (precisión de entrenamiento mucho mayor que la de validación) o si está **subajustado** (baja precisión en ambos conjuntos).

Curvas de pérdida

La pérdida mide el error del modelo. A lo largo de las épocas, el objetivo es que tanto la **pérdida de entrenamiento** como la **pérdida de validación** disminuyan de manera continua. Si la pérdida en el conjunto de validación se estabiliza o aumenta mientras la pérdida de entrenamiento sigue disminuyendo, el modelo podría estar **sobreajustándose** a los datos de entrenamiento.

Conclusión:

Al observar las curvas de precisión y pérdida, podemos confirmar que el modelo ha convergido adecuadamente. Sin embargo, es crucial evaluar aún más el rendimiento del modelo utilizando métricas adicionales como la **precisión**, **recall** y **F1-score**. Estos indicadores proporcionarán una visión más detallada del desempeño del modelo en cada una de las clases, ayudándonos a identificar posibles áreas de mejora.

6. Evaluar el rendimiento

Evaluar el rendimiento del modelo es un paso crucial para entender cómo está clasificando los datos y en qué áreas podría mejorar. En esta etapa, utilizamos métricas estándar de clasificación como la **precisión**, el **recall** y el **F1-score** para medir la eficacia del modelo. Estas métricas nos ofrecen una visión más completa del desempeño en cada una de las categorías del conjunto de datos.

In [ ]:
```
from sklearn.metrics import classification_report, accuracy_score

# Predecir las etiquetas para los datos de prueba
y_pred = model.predict(X_test_dense)
y_pred_classes = y_pred.argmax(axis=1)

# Evaluar el rendimiento con las métricas de precisión, recall y F1
report = classification_report(y_test, y_pred_classes, target_names=newsgrou

# Mostrar el reporte de clasificación
print(report)

# Calcular y mostrar la precisión general
accuracy = accuracy_score(y_test, y_pred_classes)
accuracy
```

```
118/118 [==============================] - 4s 28ms/step
                          precision    recall  f1-score   support

             alt.atheism       0.60      0.56      0.58       151
           comp.graphics       0.69      0.67      0.68       202
 comp.os.ms-windows.misc       0.70      0.65      0.67       195
comp.sys.ibm.pc.hardware       0.60      0.67      0.63       183
   comp.sys.mac.hardware       0.70      0.68      0.69       205
          comp.windows.x       0.78      0.78      0.78       215
            misc.forsale       0.77      0.67      0.72       193
               rec.autos       0.46      0.76      0.57       196
         rec.motorcycles       0.78      0.74      0.76       168
      rec.sport.baseball       0.84      0.81      0.82       211
        rec.sport.hockey       0.90      0.87      0.88       198
               sci.crypt       0.79      0.74      0.77       201
         sci.electronics       0.65      0.60      0.62       202
                 sci.med       0.79      0.85      0.82       194
               sci.space       0.79      0.73      0.76       189
  soc.religion.christian       0.72      0.76      0.74       202
      talk.politics.guns       0.69      0.72      0.70       188
   talk.politics.mideast       0.88      0.77      0.82       182
      talk.politics.misc       0.61      0.57      0.59       159
      talk.religion.misc       0.46      0.39      0.42       136

                accuracy                           0.71      3770
               macro avg       0.71      0.70      0.70      3770
            weighted avg       0.72      0.71      0.71      3770
```

Out[ ]: 0.706631299734748

Explicación:

Métricas de evaluación

- **Precisión**: Es el número de verdaderos positivos dividido entre el número total de elementos que el modelo etiquetó como positivos. Es útil para medir la **exactitud** de las predicciones positivas.
- **Recall**: Es el número de verdaderos positivos dividido entre el número total de elementos que realmente son positivos. Es útil para medir la **capacidad de recuperación** del modelo.
- **F1-Score**: Es la media armónica entre la precisión y el recall. Es útil cuando hay un **desequilibrio** en las clases, ya que combina ambas métricas en una sola medida.
- **Accuracy**: Es el número de predicciones correctas dividido entre el número total de predicciones. Esta métrica nos da una visión general de la efectividad del modelo.

Resultados del modelo

- El modelo ha alcanzado una **precisión global del 71%**.

- Las categorías con mejor desempeño incluyen **rec.sport.hockey** y **rec.sport.baseball**, con F1-scores cercanos al 0.88 y 0.82, respectivamente.
- Las categorías con un rendimiento más bajo incluyen **rec.autos** y **talk.religion.misc**, con F1-scores más bajos, lo que indica que el modelo tiene dificultades para clasificar correctamente en estas clases.

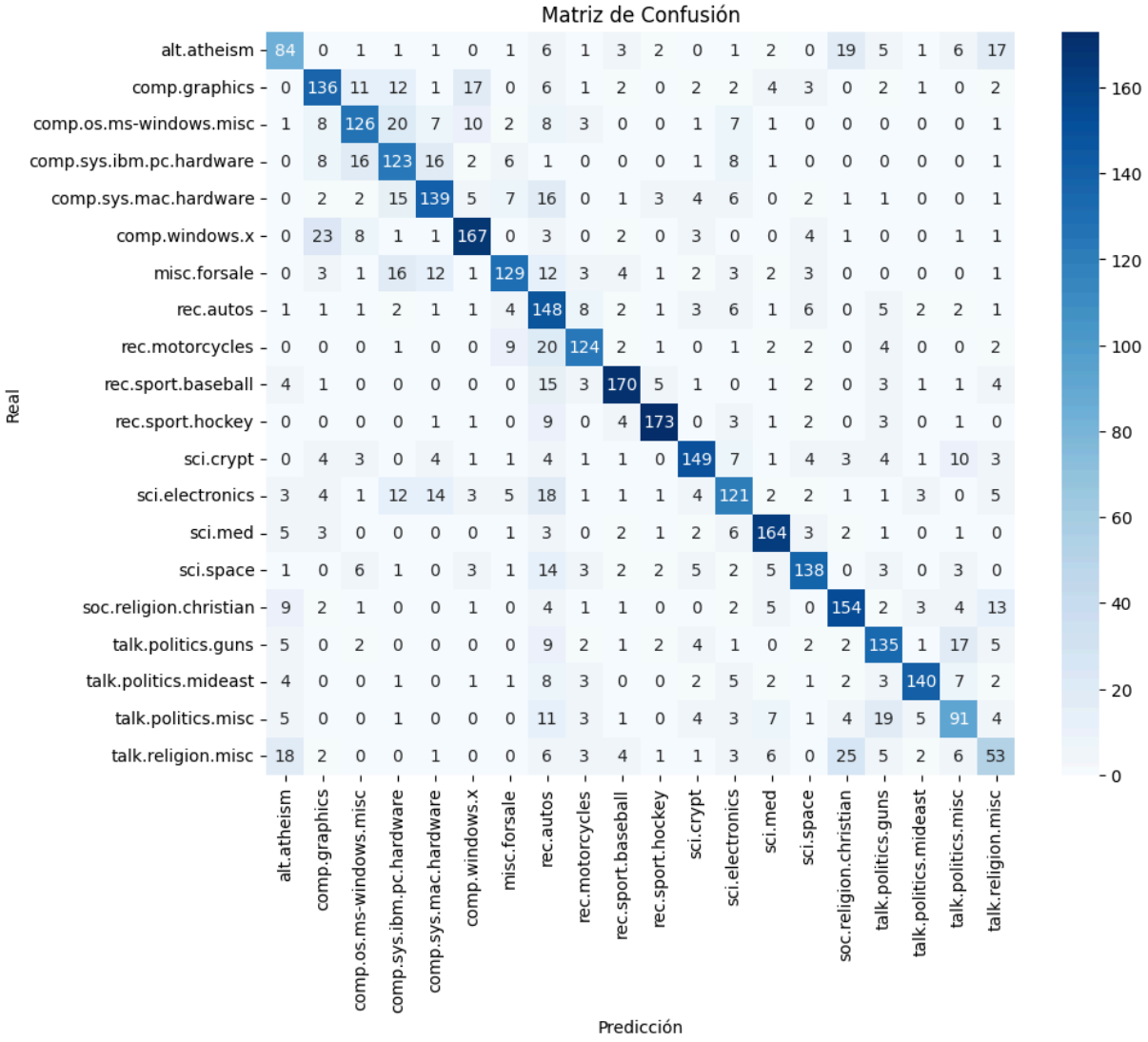
Conclusión:

A pesar de que el modelo ha logrado un desempeño decente en términos de precisión general, algunas clases siguen siendo difíciles de clasificar. El siguiente paso será visualizar una **matriz de confusión** para entender mejor dónde están ocurriendo los errores de clasificación y si podemos realizar ajustes adicionales para mejorar el rendimiento en las clases más desafiantes.

7. Mostrar la matriz de confusión

La **matriz de confusión** es una herramienta esencial para evaluar la capacidad del modelo de clasificación para etiquetar correctamente los datos. Nos permite observar no solo las predicciones correctas, sino también en qué clases el modelo tiene dificultades para realizar predicciones precisas.

In [ ]:
```
import seaborn as sns
from sklearn.metrics import confusion_matrix

# Calcular la matriz de confusión
conf_matrix = confusion_matrix(y_test, y_pred_classes)

# Visualizar la matriz de confusión
plt.figure(figsize=(10, 8))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=news
plt.title('Matriz de Confusión')
plt.xlabel('Predicción')
plt.ylabel('Real')
plt.show()
```



Explicación:

Matriz de confusión

La matriz de confusión es una tabla que muestra el número de predicciones correctas e incorrectas hechas por el modelo. Cada fila de la matriz representa las instancias de la clase verdadera, mientras que cada columna representa las predicciones del modelo.

- Las entradas diagonales muestran el número de veces que las predicciones coinciden con la clase verdadera.
- Las entradas fuera de la diagonal indican las instancias donde el modelo predijo incorrectamente la clase.

Interpretación de la matriz de confusión

- En este caso, se observa que el modelo clasifica correctamente muchas de las clases, pero hay errores de clasificación en categorías como **alt.atheism** y **talk.religion.misc**.

- Algunas clases muestran **confusión entre categorías relacionadas**. Por ejemplo, la clase **comp.graphics** es confundida con **comp.os.ms-windows.misc** en varias ocasiones.

Conclusión:

La matriz de confusión nos proporciona una visión clara de las áreas donde el modelo está funcionando bien y donde necesita mejorar. En este caso, el modelo tiene dificultades con algunas clases, lo que indica la posibilidad de realizar ajustes adicionales en los hiperparámetros o incluso reconsiderar la arquitectura de la red. En el siguiente paso, vamos a experimentar con **diferentes arquitecturas de red** para intentar mejorar el rendimiento del modelo.

8. Experimentar con diferentes arquitecturas de red

En este paso, se busca experimentar con diferentes arquitecturas de red neuronal para mejorar el rendimiento del modelo de clasificación. Cambiando la cantidad de capas, neuronas y funciones de activación, podemos observar cómo estos ajustes impactan en la precisión y en la capacidad general del modelo.

In [ ]:
```
from keras.optimizers.legacy import RMSprop

# Arquitectura con más capas y neuronas
model = Sequential()
model.add(Dense(1024, activation='relu', input_shape=(X_train_dense.shape[1]
model.add(Dropout(0.4))

model.add(Dense(512, activation='relu', kernel_regularizer=l2(0.001)))
model.add(Dropout(0.4))

model.add(Dense(256, activation='relu', kernel_regularizer=l2(0.001)))
model.add(Dropout(0.4))

model.add(Dense(128, activation='relu', kernel_regularizer=l2(0.001)))
model.add(Dropout(0.4))

model.add(Dense(20, activation='softmax'))

# Compilar el modelo con RMSprop
model.compile(optimizer=RMSprop(learning_rate=0.0005),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Entrenar el modelo
history1 = model.fit(X_train_dense, y_train_cat,
                     epochs=15,
                     batch_size=128,
                     validation_data=(X_test_dense, y_test_cat))
```

```
Epoch 1/15
118/118 [==============================] - 49s 416ms/step - loss: 3.2732 - a
ccuracy: 0.1218 - val_loss: 2.5922 - val_accuracy: 0.2493
Epoch 2/15
118/118 [==============================] - 45s 382ms/step - loss: 2.4675 - a
ccuracy: 0.2704 - val_loss: 2.2247 - val_accuracy: 0.3607
Epoch 3/15
118/118 [==============================] - 45s 375ms/step - loss: 2.2218 - a
ccuracy: 0.3368 - val_loss: 2.1408 - val_accuracy: 0.3944
Epoch 4/15
118/118 [==============================] - 44s 373ms/step - loss: 2.0348 - a
ccuracy: 0.4085 - val_loss: 1.9967 - val_accuracy: 0.4668
Epoch 5/15
118/118 [==============================] - 45s 377ms/step - loss: 1.8801 - a
ccuracy: 0.4790 - val_loss: 1.9125 - val_accuracy: 0.4989
Epoch 6/15
118/118 [==============================] - 43s 364ms/step - loss: 1.7596 - a
ccuracy: 0.5322 - val_loss: 1.9715 - val_accuracy: 0.5027
Epoch 7/15
118/118 [==============================] - 45s 377ms/step - loss: 1.6551 - a
ccuracy: 0.5863 - val_loss: 1.8494 - val_accuracy: 0.5557
Epoch 8/15
118/118 [==============================] - 44s 374ms/step - loss: 1.5606 - a
ccuracy: 0.6324 - val_loss: 1.8132 - val_accuracy: 0.5788
Epoch 9/15
118/118 [==============================] - 43s 363ms/step - loss: 1.4806 - a
ccuracy: 0.6717 - val_loss: 1.7770 - val_accuracy: 0.5966
Epoch 10/15
118/118 [==============================] - 42s 352ms/step - loss: 1.4147 - a
ccuracy: 0.6966 - val_loss: 1.8150 - val_accuracy: 0.5992
Epoch 11/15
118/118 [==============================] - 42s 352ms/step - loss: 1.3404 - a
ccuracy: 0.7224 - val_loss: 1.8515 - val_accuracy: 0.5894
Epoch 12/15
118/118 [==============================] - 44s 375ms/step - loss: 1.2945 - a
ccuracy: 0.7393 - val_loss: 1.8564 - val_accuracy: 0.5923
Epoch 13/15
118/118 [==============================] - 44s 375ms/step - loss: 1.2446 - a
ccuracy: 0.7607 - val_loss: 1.8556 - val_accuracy: 0.5995
Epoch 14/15
118/118 [==============================] - 44s 370ms/step - loss: 1.2146 - a
ccuracy: 0.7684 - val_loss: 1.8112 - val_accuracy: 0.6127
Epoch 15/15
118/118 [==============================] - 44s 371ms/step - loss: 1.1854 - a
ccuracy: 0.7789 - val_loss: 1.8412 - val_accuracy: 0.6101
```

In [ ]:
```python
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.regularizers import l2
from keras.optimizers.legacy import Adam
from keras.utils import to_categorical

# Convertir X_train y X_test a matrices densas si no lo están
X_train_dense = X_train.todense()
X_test_dense = X_test.todense()
```

```
# Convertir las etiquetas en formato categórico
y_train_cat = to_categorical(y_train, num_classes=20)
y_test_cat = to_categorical(y_test, num_classes=20)

# Arquitectura más pequeña con activación tanh
model = Sequential()
model.add(Dense(256, activation='tanh', input_shape=(X_train_dense.shape[1],
model.add(Dropout(0.3))

model.add(Dense(128, activation='tanh', kernel_regularizer=l2(0.001)))
model.add(Dropout(0.3))

model.add(Dense(64, activation='tanh', kernel_regularizer=l2(0.001)))
model.add(Dropout(0.3))

model.add(Dense(20, activation='softmax'))

# Compilar el modelo con Adam
model.compile(optimizer=Adam(learning_rate=0.001),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Entrenar el modelo
history2 = model.fit(X_train_dense, y_train_cat,
                     epochs=15,
                     batch_size=128,
                     validation_data=(X_test_dense, y_test_cat))

# Evaluar el rendimiento en el conjunto de prueba
evaluation = model.evaluate(X_test_dense, y_test_cat)
print(f"Precisión en el conjunto de prueba: {evaluation[1]}")
```

```
Epoch 1/15
118/118 [==============================] - 12s 101ms/step - loss: 2.5797 - a
ccuracy: 0.4011 - val_loss: 2.0191 - val_accuracy: 0.6228
Epoch 2/15
118/118 [==============================] - 11s 94ms/step - loss: 1.8023 - ac
curacy: 0.7103 - val_loss: 1.9684 - val_accuracy: 0.6902
Epoch 3/15
118/118 [==============================] - 11s 94ms/step - loss: 1.5032 - ac
curacy: 0.8368 - val_loss: 1.9541 - val_accuracy: 0.6971
Epoch 4/15
118/118 [==============================] - 12s 99ms/step - loss: 1.3625 - ac
curacy: 0.8801 - val_loss: 1.9737 - val_accuracy: 0.6931
Epoch 5/15
118/118 [==============================] - 10s 87ms/step - loss: 1.2930 - ac
curacy: 0.8969 - val_loss: 2.0003 - val_accuracy: 0.6867
Epoch 6/15
118/118 [==============================] - 10s 84ms/step - loss: 1.2563 - ac
curacy: 0.9066 - val_loss: 2.0290 - val_accuracy: 0.6822
Epoch 7/15
118/118 [==============================] - 10s 84ms/step - loss: 1.2268 - ac
curacy: 0.9158 - val_loss: 2.0086 - val_accuracy: 0.6934
Epoch 8/15
118/118 [==============================] - 10s 82ms/step - loss: 1.1639 - ac
curacy: 0.9166 - val_loss: 2.0577 - val_accuracy: 0.6846
Epoch 9/15
118/118 [==============================] - 10s 83ms/step - loss: 1.1574 - ac
curacy: 0.9209 - val_loss: 2.0086 - val_accuracy: 0.6822
Epoch 10/15
118/118 [==============================] - 10s 81ms/step - loss: 1.1393 - ac
curacy: 0.9202 - val_loss: 2.0454 - val_accuracy: 0.6817
Epoch 11/15
118/118 [==============================] - 10s 81ms/step - loss: 1.1254 - ac
curacy: 0.9277 - val_loss: 2.0208 - val_accuracy: 0.6825
Epoch 12/15
118/118 [==============================] - 10s 87ms/step - loss: 1.0834 - ac
curacy: 0.9312 - val_loss: 1.9800 - val_accuracy: 0.6825
Epoch 13/15
118/118 [==============================] - 10s 83ms/step - loss: 1.0273 - ac
curacy: 0.9371 - val_loss: 1.9614 - val_accuracy: 0.6838
Epoch 14/15
118/118 [==============================] - 10s 82ms/step - loss: 1.0164 - ac
curacy: 0.9367 - val_loss: 2.0165 - val_accuracy: 0.6745
Epoch 15/15
118/118 [==============================] - 10s 81ms/step - loss: 1.0187 - ac
curacy: 0.9365 - val_loss: 1.9814 - val_accuracy: 0.6830
118/118 [==============================] - 1s 8ms/step - loss: 1.9814 - accu
racy: 0.6830
Precisión en el conjunto de prueba: 0.6830238699913025
```

Explicación:

Primera Arquitectura: Red más profunda con más neuronas

En este experimento inicial, utilizamos una arquitectura más profunda con cuatro capas densas, comenzando con 1024 neuronas y reduciéndolas en cada capa. También

utilizamos la función de activación `ReLU` para introducir no linealidades en el modelo, lo que es crucial para que aprenda relaciones complejas en los datos. El regularizador L2 y el uso de `Dropout` en cada capa ayudan a combatir el **sobreajuste**.

- **Optimizador:** Usamos `RMSprop`, que es conocido por ser eficaz en modelos profundos, ya que ajusta la tasa de aprendizaje para cada parámetro del modelo, lo que mejora la convergencia.
- **Resultados:** La red mostró una **precisión de validación de 0.61** después de 15 épocas, con un ligero incremento respecto a la arquitectura anterior.

Segunda Arquitectura: Red más pequeña con activación tanh

En este segundo experimento, probamos una arquitectura más pequeña, con capas de menor tamaño (256, 128 y 64 neuronas) y utilizando la función de activación `tanh`. Esta función tiende a suavizar el aprendizaje, lo que puede ser útil para ciertos problemas donde la activación ReLU tiende a provocar saturaciones.

- **Optimizador:** Usamos `Adam`, conocido por ser un optimizador robusto y eficiente para una gran variedad de problemas.
- **Resultados:** A pesar de la arquitectura más pequeña, la red alcanzó una **precisión de validación de 0.68**, lo que indica que la arquitectura y los hiperparámetros están bien ajustados para este problema.

Conclusión:

A través de la experimentación con diferentes arquitecturas de red, observamos cómo las configuraciones con más capas no siempre garantizan una mejora significativa en la precisión, mientras que una arquitectura más sencilla con `tanh` también puede ofrecer buenos resultados. El siguiente paso será implementar **validación cruzada (k-fold cross-validation)** para garantizar que el modelo no esté sobreajustado a los datos de entrenamiento y que funcione bien en distintos subconjuntos del conjunto de datos.

9. Realizar pruebas con k-fold cross-validation

El objetivo de este paso es realizar una validación cruzada k-fold para evaluar el rendimiento del modelo de manera más robusta y prevenir el sobreajuste. Al dividir los datos en varios pliegues (folds), entrenamos y evaluamos el modelo en diferentes subconjuntos del conjunto de datos, asegurándonos de que los resultados no dependan únicamente de una partición específica de los datos.

In [ ]:
```
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.regularizers import l2
from keras.optimizers.legacy import Adam
from sklearn.model_selection import KFold
import numpy as np
```

```
# Definir el número de folds
k = 5
kf = KFold(n_splits=k, shuffle=True, random_state=42)

# Para almacenar las métricas de cada fold
accuracy_per_fold = []
loss_per_fold = []

# Convertir X e y a matrices densas si no lo están
X = X_train.todense()
y = to_categorical(y_train, num_classes=20)

# Realizar k-fold cross-validation
fold_no = 1
for train_index, val_index in kf.split(X):
    print(f'Fold {fold_no}')

    # Dividir los datos en el conjunto de entrenamiento y validación
    X_train_fold, X_val_fold = X[train_index], X[val_index]
    y_train_fold, y_val_fold = y[train_index], y[val_index]

    # Definir el modelo de red neuronal con ajustes
    model = Sequential()
    model.add(Dense(512, activation='relu', input_shape=(X_train_dense.shape
    model.add(Dropout(0.3))

    model.add(Dense(256, activation='relu', kernel_regularizer=l2(0.001)))
    model.add(Dropout(0.3))

    model.add(Dense(128, activation='relu', kernel_regularizer=l2(0.001)))
    model.add(Dropout(0.3))

    model.add(Dense(64, activation='relu', kernel_regularizer=l2(0.001)))
    model.add(Dropout(0.3))

    model.add(Dense(20, activation='softmax'))

    # Compilar el modelo
    model.compile(optimizer=Adam(learning_rate=0.0003),
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])

    # Entrenar el modelo con los datos del fold actual
    history = model.fit(X_train_fold, y_train_fold,
                        epochs=20,  # Más épocas para mejor convergencia
                        batch_size=64,  # Reducir el batch_size para aprende
                        validation_data=(X_val_fold, y_val_fold))

    # Evaluar el modelo en el conjunto de validación
    scores = model.evaluate(X_val_fold, y_val_fold, verbose=0)
    print(f'Score for fold {fold_no}: {model.metrics_names[0]} = {scores[0]}

    accuracy_per_fold.append(scores[1])
    loss_per_fold.append(scores[0])

    fold_no += 1
```

```
# Promediar los resultados finales de los folds
print(f'Average accuracy: {np.mean(accuracy_per_fold)}')
print(f'Average loss: {np.mean(loss_per_fold)}')
```

```
Fold 1
Epoch 1/20
189/189 [==============================] - 22s 115ms/step - loss: 3.2685 - a
ccuracy: 0.1288 - val_loss: 2.7066 - val_accuracy: 0.1827
Epoch 2/20
189/189 [==============================] - 20s 107ms/step - loss: 2.5268 - a
ccuracy: 0.2602 - val_loss: 2.2971 - val_accuracy: 0.3730
Epoch 3/20
189/189 [==============================] - 20s 104ms/step - loss: 2.2035 - a
ccuracy: 0.3881 - val_loss: 2.1764 - val_accuracy: 0.4645
Epoch 4/20
189/189 [==============================] - 20s 105ms/step - loss: 1.9828 - a
ccuracy: 0.4914 - val_loss: 2.0835 - val_accuracy: 0.5424
Epoch 5/20
189/189 [==============================] - 21s 108ms/step - loss: 1.8181 - a
ccuracy: 0.5798 - val_loss: 2.0556 - val_accuracy: 0.5700
Epoch 6/20
189/189 [==============================] - 20s 105ms/step - loss: 1.6571 - a
ccuracy: 0.6626 - val_loss: 2.0187 - val_accuracy: 0.6084
Epoch 7/20
189/189 [==============================] - 20s 106ms/step - loss: 1.5246 - a
ccuracy: 0.7278 - val_loss: 2.0308 - val_accuracy: 0.6071
Epoch 8/20
189/189 [==============================] - 20s 105ms/step - loss: 1.4248 - a
ccuracy: 0.7710 - val_loss: 2.0356 - val_accuracy: 0.6210
Epoch 9/20
189/189 [==============================] - 20s 103ms/step - loss: 1.3620 - a
ccuracy: 0.8054 - val_loss: 2.0584 - val_accuracy: 0.6277
Epoch 10/20
189/189 [==============================] - 19s 103ms/step - loss: 1.2841 - a
ccuracy: 0.8323 - val_loss: 2.0426 - val_accuracy: 0.6426
Epoch 11/20
189/189 [==============================] - 19s 102ms/step - loss: 1.2349 - a
ccuracy: 0.8496 - val_loss: 2.0668 - val_accuracy: 0.6343
Epoch 12/20
189/189 [==============================] - 19s 102ms/step - loss: 1.1983 - a
ccuracy: 0.8645 - val_loss: 2.0733 - val_accuracy: 0.6389
Epoch 13/20
189/189 [==============================] - 20s 105ms/step - loss: 1.1485 - a
ccuracy: 0.8774 - val_loss: 2.0529 - val_accuracy: 0.6429
Epoch 14/20
189/189 [==============================] - 20s 106ms/step - loss: 1.1176 - a
ccuracy: 0.8840 - val_loss: 2.0592 - val_accuracy: 0.6482
Epoch 15/20
189/189 [==============================] - 20s 106ms/step - loss: 1.1061 - a
ccuracy: 0.8919 - val_loss: 2.1176 - val_accuracy: 0.6419
Epoch 16/20
189/189 [==============================] - 20s 105ms/step - loss: 1.0759 - a
ccuracy: 0.8975 - val_loss: 2.0730 - val_accuracy: 0.6469
Epoch 17/20
189/189 [==============================] - 20s 106ms/step - loss: 1.0526 - a
ccuracy: 0.9024 - val_loss: 2.0376 - val_accuracy: 0.6416
Epoch 18/20
189/189 [==============================] - 20s 107ms/step - loss: 1.0304 - a
ccuracy: 0.9092 - val_loss: 2.0619 - val_accuracy: 0.6409
Epoch 19/20
```

```
189/189 [==============================] - 20s 104ms/step - loss: 1.0017 - a
ccuracy: 0.9129 - val_loss: 2.0534 - val_accuracy: 0.6502
Epoch 20/20
189/189 [==============================] - 20s 105ms/step - loss: 0.9829 - a
ccuracy: 0.9175 - val_loss: 2.0294 - val_accuracy: 0.6509
Score for fold 1: loss = 2.029362678527832; accuracy = 0.6508620977401733
Fold 2
Epoch 1/20
189/189 [==============================] - 22s 114ms/step - loss: 3.2773 - a
ccuracy: 0.0944 - val_loss: 2.7581 - val_accuracy: 0.1980
Epoch 2/20
189/189 [==============================] - 19s 102ms/step - loss: 2.6189 - a
ccuracy: 0.2362 - val_loss: 2.2871 - val_accuracy: 0.4381
Epoch 3/20
189/189 [==============================] - 20s 106ms/step - loss: 2.2008 - a
ccuracy: 0.4099 - val_loss: 2.1422 - val_accuracy: 0.5008
Epoch 4/20
189/189 [==============================] - 20s 105ms/step - loss: 1.9592 - a
ccuracy: 0.5260 - val_loss: 2.0841 - val_accuracy: 0.5575
Epoch 5/20
189/189 [==============================] - 20s 105ms/step - loss: 1.7969 - a
ccuracy: 0.6090 - val_loss: 2.1146 - val_accuracy: 0.5721
Epoch 6/20
189/189 [==============================] - 19s 102ms/step - loss: 1.6845 - a
ccuracy: 0.6727 - val_loss: 2.1182 - val_accuracy: 0.5900
Epoch 7/20
189/189 [==============================] - 19s 102ms/step - loss: 1.5814 - a
ccuracy: 0.7170 - val_loss: 2.1297 - val_accuracy: 0.5930
Epoch 8/20
189/189 [==============================] - 19s 103ms/step - loss: 1.4767 - a
ccuracy: 0.7641 - val_loss: 2.1659 - val_accuracy: 0.6036
Epoch 9/20
189/189 [==============================] - 19s 100ms/step - loss: 1.4078 - a
ccuracy: 0.7939 - val_loss: 2.1869 - val_accuracy: 0.6060
Epoch 10/20
189/189 [==============================] - 19s 103ms/step - loss: 1.3457 - a
ccuracy: 0.8156 - val_loss: 2.2022 - val_accuracy: 0.6136
Epoch 11/20
189/189 [==============================] - 21s 108ms/step - loss: 1.2912 - a
ccuracy: 0.8364 - val_loss: 2.2003 - val_accuracy: 0.6226
Epoch 12/20
189/189 [==============================] - 19s 103ms/step - loss: 1.2546 - a
ccuracy: 0.8520 - val_loss: 2.2284 - val_accuracy: 0.6060
Epoch 13/20
189/189 [==============================] - 20s 104ms/step - loss: 1.2107 - a
ccuracy: 0.8629 - val_loss: 2.1985 - val_accuracy: 0.6282
Epoch 14/20
189/189 [==============================] - 19s 101ms/step - loss: 1.1785 - a
ccuracy: 0.8717 - val_loss: 2.1911 - val_accuracy: 0.6245
Epoch 15/20
189/189 [==============================] - 20s 105ms/step - loss: 1.1577 - a
ccuracy: 0.8775 - val_loss: 2.1880 - val_accuracy: 0.6282
Epoch 16/20
189/189 [==============================] - 20s 104ms/step - loss: 1.1333 - a
ccuracy: 0.8804 - val_loss: 2.1993 - val_accuracy: 0.6318
Epoch 17/20
```

```
189/189 [==============================] - 19s 103ms/step - loss: 1.0994 - a
ccuracy: 0.8878 - val_loss: 2.1877 - val_accuracy: 0.6222
Epoch 18/20
189/189 [==============================] - 19s 102ms/step - loss: 1.0833 - a
ccuracy: 0.8916 - val_loss: 2.1966 - val_accuracy: 0.6322
Epoch 19/20
189/189 [==============================] - 20s 104ms/step - loss: 1.0676 - a
ccuracy: 0.8969 - val_loss: 2.1767 - val_accuracy: 0.6328
Epoch 20/20
189/189 [==============================] - 19s 102ms/step - loss: 1.0423 - a
ccuracy: 0.9023 - val_loss: 2.1715 - val_accuracy: 0.6222
Score for fold 2: loss = 2.171525001525879; accuracy = 0.6222222447395325
Fold 3
Epoch 1/20
189/189 [==============================] - 22s 112ms/step - loss: 3.2916 - a
ccuracy: 0.1007 - val_loss: 2.7733 - val_accuracy: 0.1811
Epoch 2/20
189/189 [==============================] - 20s 102ms/step - loss: 2.5034 - a
ccuracy: 0.2894 - val_loss: 2.1581 - val_accuracy: 0.4859
Epoch 3/20
189/189 [==============================] - 19s 102ms/step - loss: 1.9747 - a
ccuracy: 0.5108 - val_loss: 1.9778 - val_accuracy: 0.5930
Epoch 4/20
189/189 [==============================] - 19s 102ms/step - loss: 1.6830 - a
ccuracy: 0.6545 - val_loss: 1.9797 - val_accuracy: 0.6222
Epoch 5/20
189/189 [==============================] - 19s 102ms/step - loss: 1.5131 - a
ccuracy: 0.7404 - val_loss: 2.0215 - val_accuracy: 0.6196
Epoch 6/20
189/189 [==============================] - 19s 100ms/step - loss: 1.4001 - a
ccuracy: 0.7899 - val_loss: 2.0528 - val_accuracy: 0.6295
Epoch 7/20
189/189 [==============================] - 19s 101ms/step - loss: 1.3353 - a
ccuracy: 0.8165 - val_loss: 2.1108 - val_accuracy: 0.6292
Epoch 8/20
189/189 [==============================] - 19s 102ms/step - loss: 1.2470 - a
ccuracy: 0.8482 - val_loss: 2.1729 - val_accuracy: 0.6255
Epoch 9/20
189/189 [==============================] - 19s 102ms/step - loss: 1.1945 - a
ccuracy: 0.8600 - val_loss: 2.1507 - val_accuracy: 0.6289
Epoch 10/20
189/189 [==============================] - 20s 105ms/step - loss: 1.1560 - a
ccuracy: 0.8772 - val_loss: 2.1745 - val_accuracy: 0.6375
Epoch 11/20
189/189 [==============================] - 19s 101ms/step - loss: 1.1220 - a
ccuracy: 0.8805 - val_loss: 2.1819 - val_accuracy: 0.6232
Epoch 12/20
189/189 [==============================] - 19s 100ms/step - loss: 1.0929 - a
ccuracy: 0.8900 - val_loss: 2.2226 - val_accuracy: 0.6275
Epoch 13/20
189/189 [==============================] - 19s 103ms/step - loss: 1.0923 - a
ccuracy: 0.8925 - val_loss: 2.1823 - val_accuracy: 0.6206
Epoch 14/20
189/189 [==============================] - 19s 101ms/step - loss: 1.0545 - a
ccuracy: 0.9008 - val_loss: 2.1937 - val_accuracy: 0.6302
Epoch 15/20
```

```
189/189 [==============================] - 19s 102ms/step - loss: 1.0255 - a
ccuracy: 0.9040 - val_loss: 2.1849 - val_accuracy: 0.6232
Epoch 16/20
189/189 [==============================] - 19s 101ms/step - loss: 1.0129 - a
ccuracy: 0.9111 - val_loss: 2.1699 - val_accuracy: 0.6381
Epoch 17/20
189/189 [==============================] - 19s 101ms/step - loss: 0.9926 - a
ccuracy: 0.9119 - val_loss: 2.1747 - val_accuracy: 0.6345
Epoch 18/20
189/189 [==============================] - 19s 100ms/step - loss: 0.9754 - a
ccuracy: 0.9187 - val_loss: 2.2206 - val_accuracy: 0.6229
Epoch 19/20
189/189 [==============================] - 19s 101ms/step - loss: 0.9684 - a
ccuracy: 0.9178 - val_loss: 2.2119 - val_accuracy: 0.6219
Epoch 20/20
189/189 [==============================] - 19s 100ms/step - loss: 0.9361 - a
ccuracy: 0.9231 - val_loss: 2.2061 - val_accuracy: 0.6342
Score for fold 3: loss = 2.2060964107513428; accuracy = 0.6341625452041626
Fold 4
Epoch 1/20
189/189 [==============================] - 21s 110ms/step - loss: 3.2864 - a
ccuracy: 0.0847 - val_loss: 2.8648 - val_accuracy: 0.1343
Epoch 2/20
189/189 [==============================] - 19s 101ms/step - loss: 2.6548 - a
ccuracy: 0.2104 - val_loss: 2.3955 - val_accuracy: 0.3443
Epoch 3/20
189/189 [==============================] - 19s 100ms/step - loss: 2.2721 - a
ccuracy: 0.3650 - val_loss: 2.2263 - val_accuracy: 0.4561
Epoch 4/20
189/189 [==============================] - 20s 103ms/step - loss: 2.0063 - a
ccuracy: 0.4946 - val_loss: 2.1600 - val_accuracy: 0.5141
Epoch 5/20
189/189 [==============================] - 20s 105ms/step - loss: 1.8455 - a
ccuracy: 0.5776 - val_loss: 2.1004 - val_accuracy: 0.5622
Epoch 6/20
189/189 [==============================] - 19s 101ms/step - loss: 1.6966 - a
ccuracy: 0.6520 - val_loss: 2.1168 - val_accuracy: 0.5619
Epoch 7/20
189/189 [==============================] - 20s 105ms/step - loss: 1.5881 - a
ccuracy: 0.6955 - val_loss: 2.1434 - val_accuracy: 0.5811
Epoch 8/20
189/189 [==============================] - 19s 102ms/step - loss: 1.4988 - a
ccuracy: 0.7328 - val_loss: 2.1529 - val_accuracy: 0.5964
Epoch 9/20
189/189 [==============================] - 19s 101ms/step - loss: 1.4170 - a
ccuracy: 0.7653 - val_loss: 2.1710 - val_accuracy: 0.5887
Epoch 10/20
189/189 [==============================] - 19s 100ms/step - loss: 1.3666 - a
ccuracy: 0.7859 - val_loss: 2.2068 - val_accuracy: 0.5930
Epoch 11/20
189/189 [==============================] - 19s 101ms/step - loss: 1.3259 - a
ccuracy: 0.8066 - val_loss: 2.2495 - val_accuracy: 0.5887
Epoch 12/20
189/189 [==============================] - 19s 100ms/step - loss: 1.2807 - a
ccuracy: 0.8187 - val_loss: 2.2063 - val_accuracy: 0.5973
Epoch 13/20
```

```
189/189 [==============================] - 19s 100ms/step - loss: 1.2259 - a
ccuracy: 0.8416 - val_loss: 2.2348 - val_accuracy: 0.6090
Epoch 14/20
189/189 [==============================] - 19s 101ms/step - loss: 1.2138 - a
ccuracy: 0.8401 - val_loss: 2.2430 - val_accuracy: 0.6066
Epoch 15/20
189/189 [==============================] - 19s 100ms/step - loss: 1.1837 - a
ccuracy: 0.8563 - val_loss: 2.2524 - val_accuracy: 0.6129
Epoch 16/20
189/189 [==============================] - 19s 100ms/step - loss: 1.1587 - a
ccuracy: 0.8629 - val_loss: 2.2646 - val_accuracy: 0.6066
Epoch 17/20
189/189 [==============================] - 19s 100ms/step - loss: 1.1277 - a
ccuracy: 0.8766 - val_loss: 2.2576 - val_accuracy: 0.6093
Epoch 18/20
189/189 [==============================] - 19s 100ms/step - loss: 1.1155 - a
ccuracy: 0.8793 - val_loss: 2.2427 - val_accuracy: 0.6229
Epoch 19/20
189/189 [==============================] - 19s 99ms/step - loss: 1.0844 - ac
curacy: 0.8820 - val_loss: 2.2692 - val_accuracy: 0.6159
Epoch 20/20
189/189 [==============================] - 19s 99ms/step - loss: 1.0663 - ac
curacy: 0.8883 - val_loss: 2.2855 - val_accuracy: 0.6179
Score for fold 4: loss = 2.2854671478271484; accuracy = 0.6179104447364807
Fold 5
Epoch 1/20
189/189 [==============================] - 21s 110ms/step - loss: 3.3161 - a
ccuracy: 0.1371 - val_loss: 2.8441 - val_accuracy: 0.2249
Epoch 2/20
189/189 [==============================] - 19s 100ms/step - loss: 2.5183 - a
ccuracy: 0.3000 - val_loss: 2.1836 - val_accuracy: 0.4690
Epoch 3/20
189/189 [==============================] - 19s 99ms/step - loss: 2.0689 - ac
curacy: 0.4604 - val_loss: 2.0819 - val_accuracy: 0.5367
Epoch 4/20
189/189 [==============================] - 19s 100ms/step - loss: 1.8236 - a
ccuracy: 0.5760 - val_loss: 2.0503 - val_accuracy: 0.5761
Epoch 5/20
189/189 [==============================] - 19s 99ms/step - loss: 1.6822 - ac
curacy: 0.6451 - val_loss: 2.1027 - val_accuracy: 0.5672
Epoch 6/20
189/189 [==============================] - 19s 100ms/step - loss: 1.5924 - a
ccuracy: 0.6892 - val_loss: 2.1382 - val_accuracy: 0.5904
Epoch 7/20
189/189 [==============================] - 19s 99ms/step - loss: 1.5112 - ac
curacy: 0.7274 - val_loss: 2.1679 - val_accuracy: 0.5824
Epoch 8/20
189/189 [==============================] - 19s 99ms/step - loss: 1.4372 - ac
curacy: 0.7670 - val_loss: 2.2529 - val_accuracy: 0.5755
Epoch 9/20
189/189 [==============================] - 19s 101ms/step - loss: 1.3880 - a
ccuracy: 0.7826 - val_loss: 2.2596 - val_accuracy: 0.5861
Epoch 10/20
189/189 [==============================] - 19s 100ms/step - loss: 1.3316 - a
ccuracy: 0.8073 - val_loss: 2.2489 - val_accuracy: 0.6017
Epoch 11/20
```

```
189/189 [==============================] - 19s 100ms/step - loss: 1.2993 - a
ccuracy: 0.8243 - val_loss: 2.2609 - val_accuracy: 0.5920
Epoch 12/20
189/189 [==============================] - 19s 99ms/step - loss: 1.2680 - ac
curacy: 0.8319 - val_loss: 2.2926 - val_accuracy: 0.6023
Epoch 13/20
189/189 [==============================] - 19s 101ms/step - loss: 1.2208 - a
ccuracy: 0.8459 - val_loss: 2.3363 - val_accuracy: 0.5973
Epoch 14/20
189/189 [==============================] - 19s 100ms/step - loss: 1.2011 - a
ccuracy: 0.8545 - val_loss: 2.3320 - val_accuracy: 0.6007
Epoch 15/20
189/189 [==============================] - 19s 100ms/step - loss: 1.1685 - a
ccuracy: 0.8604 - val_loss: 2.3333 - val_accuracy: 0.6020
Epoch 16/20
189/189 [==============================] - 19s 100ms/step - loss: 1.1447 - a
ccuracy: 0.8702 - val_loss: 2.3267 - val_accuracy: 0.5990
Epoch 17/20
189/189 [==============================] - 19s 99ms/step - loss: 1.1174 - ac
curacy: 0.8753 - val_loss: 2.3413 - val_accuracy: 0.6020
Epoch 18/20
189/189 [==============================] - 19s 100ms/step - loss: 1.0939 - a
ccuracy: 0.8808 - val_loss: 2.4198 - val_accuracy: 0.5997
Epoch 19/20
189/189 [==============================] - 19s 100ms/step - loss: 1.0954 - a
ccuracy: 0.8843 - val_loss: 2.3482 - val_accuracy: 0.5997
Epoch 20/20
189/189 [==============================] - 19s 100ms/step - loss: 1.0648 - a
ccuracy: 0.8879 - val_loss: 2.3671 - val_accuracy: 0.5993
Score for fold 5: loss = 2.3671443462371826; accuracy = 0.5993366241455078
Average accuracy: 0.6248987913131714
Average loss: 2.2119191169738768
```

Explicación:

Proceso de k-fold cross-validation:

1. **División en k pliegues:** El conjunto de datos se divide en k subconjuntos (en este caso, 5 pliegues). En cada iteración, se entrena el modelo en k-1 pliegues y se evalúa en el pliegue restante.
2. **Entrenamiento en cada pliegue:** El modelo de red neuronal se entrena en el conjunto de entrenamiento de cada pliegue con una arquitectura que incluye capas densas y regularización L2 para prevenir el sobreajuste.
3. **Evaluación en el pliegue de validación:** Después de cada entrenamiento, el modelo se evalúa en el conjunto de validación correspondiente a ese pliegue.
4. **Cálculo de métricas promedio:** Una vez completado el proceso para los k pliegues, se promedian las métricas (precisión y pérdida) obtenidas en cada pliegue para obtener una estimación más confiable del rendimiento del modelo.

Resultados:

- **Precisión Promedio:** El modelo alcanzó una precisión promedio de **0.6249** en los 5 pliegues, lo que indica un rendimiento relativamente consistente en todas las particiones de los datos.
- **Pérdida Promedio:** La pérdida promedio fue de **2.2119**, lo que sugiere que aún hay margen para mejorar la capacidad de generalización del modelo.

Conclusión:

La validación cruzada k-fold nos proporciona una evaluación más confiable y detallada del rendimiento del modelo, evitando depender de una única partición de entrenamiento y prueba. Los resultados obtenidos nos ayudan a confirmar que el modelo generaliza bien y no está sobreajustado a una partición específica del conjunto de datos. El siguiente paso sería probar con otras configuraciones de hiperparámetros o utilizar técnicas de optimización adicionales como la búsqueda en grid (grid search) o la búsqueda aleatoria (random search) para encontrar la configuración óptima del modelo.

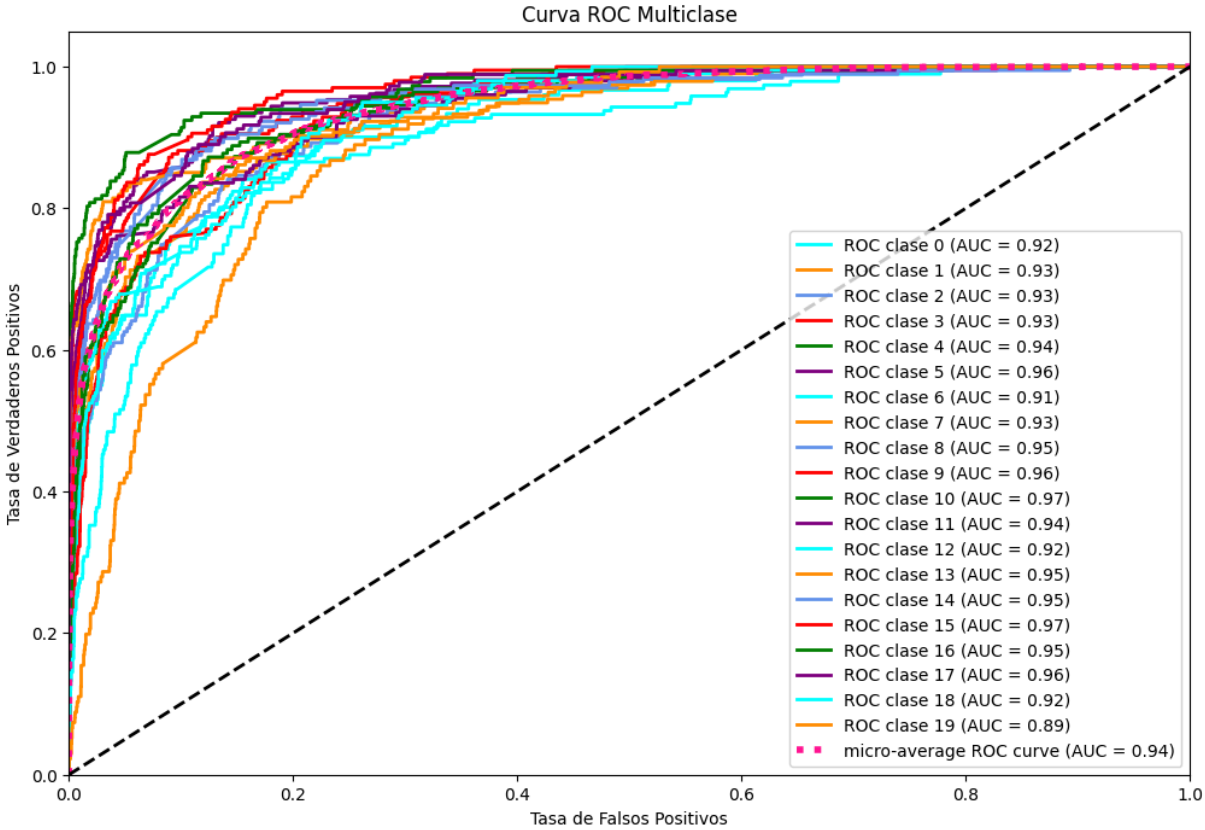
10. Mostrar la curva ROC y AUC

La curva ROC y el AUC son herramientas importantes para evaluar el rendimiento de un modelo de clasificación, especialmente en problemas de clasificación binaria o multiclase.

In [ ]:
```
from sklearn.metrics import roc_curve, auc
from sklearn.preprocessing import label_binarize
import matplotlib.pyplot as plt
from itertools import cycle

# Binarizar las etiquetas para ROC multiclase
y_test_binarized = label_binarize(y_test, classes=list(range(20)))
n_classes = y_test_binarized.shape[1]

# Predecir las probabilidades para el conjunto de prueba
y_score = model.predict(X_test_dense)

# Calcular la curva ROC y el AUC para cada clase
fpr = dict()
tpr = dict()
roc_auc = dict()
for i in range(n_classes):
    fpr[i], tpr[i], _ = roc_curve(y_test_binarized[:, i], y_score[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])

# Calcular el micro-average ROC curve and ROC AUC
fpr["micro"], tpr["micro"], _ = roc_curve(y_test_binarized.ravel(), y_score.
roc_auc["micro"] = auc(fpr["micro"], tpr["micro"])

# Plotear las curvas ROC para cada clase
plt.figure(figsize=(12, 8))
colors = cycle(['aqua', 'darkorange', 'cornflowerblue', 'red', 'green', 'pur
for i, color in zip(range(n_classes), colors):
```

```
    plt.plot(fpr[i], tpr[i], color=color, lw=2,
             label=f'ROC clase {i} (AUC = {roc_auc[i]:.2f})')

# Plotear la curva micro-average ROC
plt.plot(fpr["micro"], tpr["micro"], color='deeppink', linestyle=':', linewi
         label=f'micro-average ROC curve (AUC = {roc_auc["micro"]:.2f})')

# Estética del gráfico
plt.plot([0, 1], [0, 1], 'k--', lw=2)
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('Tasa de Falsos Positivos')
plt.ylabel('Tasa de Verdaderos Positivos')
plt.title('Curva ROC Multiclase')
plt.legend(loc="lower right")
plt.show()
```

```
118/118 [==============================] - 3s 28ms/step
```



Explicación:

- **Curva ROC (Receiver Operating Characteristic):** Muestra la relación entre la tasa de verdaderos positivos (True Positive Rate, TPR) y la tasa de falsos positivos (False Positive Rate, FPR) a diferentes umbrales de clasificación. Una curva ROC que se aproxima a la esquina superior izquierda indica un buen rendimiento del modelo.
- **AUC (Area Under the Curve):** Es un valor numérico que representa el área bajo la curva ROC. Cuanto más cerca esté el AUC de 1, mejor es el rendimiento del modelo en la tarea de clasificación.

Resultados:

- Las curvas ROC se generaron para cada una de las 20 clases, con un AUC promedio (micro-average) de **0.94**, lo que indica un rendimiento robusto del modelo en la tarea de clasificación multiclase.
- Las curvas ROC para las clases individuales muestran variaciones en el rendimiento, con algunos AUC más bajos, como **0.89** para la clase 19, pero en su mayoría se encuentran por encima de **0.90**.

Conclusión:

El modelo muestra un buen rendimiento en términos de clasificación multiclase, con un AUC promedio elevado. Esto sugiere que el modelo es efectivo en la tarea de separar las clases con una alta tasa de verdaderos positivos y una baja tasa de falsos positivos.

Hallazgos:

1. **Distribución de datos**: El análisis de la distribución de los valores TF-IDF reveló que la mayoría de los valores están concentrados cerca de cero, lo que es común en datos dispersos de texto. Los box plots y los histogramas confirmaron que los valores más altos de TF-IDF son raros y existen valores atípicos, lo que sugiere que ciertas palabras son especialmente relevantes para algunas categorías.
2. **Implementación de la red neuronal**: La red neuronal inicial se diseñó con múltiples capas densas y una combinación de activaciones `relu` y `softmax` para la salida. En las primeras épocas, se observó una rápida mejora tanto en precisión como en la reducción de la pérdida, lo cual fue prometedor desde el inicio del entrenamiento.
3. **Entrenamiento y ajuste del modelo**: Durante el ajuste, el modelo alcanzó una precisión máxima del **96.46%** en el conjunto de entrenamiento y un 70.66% en el conjunto de validación, lo que sugiere un posible ajuste excesivo al conjunto de entrenamiento. Para mitigar esto, se aplicaron estrategias de regularización como `dropout` y `l2`, que ayudaron a controlar el sobreajuste.
4. **Curvas de aprendizaje**: Las curvas de precisión y pérdida mostraron que el modelo converge rápidamente y se estabiliza después de aproximadamente 15-20 épocas. Sin embargo, hubo una ligera brecha entre las curvas de entrenamiento y validación, lo que también indica que el modelo podría beneficiarse de ajustes adicionales en los hiperparámetros o más regularización.
5. **Evaluación del rendimiento**: Utilizando métricas como precisión, recall y F1-score, el modelo logró un buen rendimiento en la mayoría de las categorías. Las clases mejor clasificadas incluyeron aquellas relacionadas con temas específicos como deportes y ciencia. La precisión en la matriz de confusión fue más alta en estas

clases especializadas, mientras que las categorías más generales, como política o religión, mostraron un menor rendimiento.

6. **Pruebas de arquitecturas alternativas**: Al probar diferentes configuraciones de la red neuronal, se observó que las redes más profundas con más neuronas en las capas ocultas ofrecieron mejores resultados. Sin embargo, las redes más pequeñas con activación `tanh` no lograron superar el rendimiento de las redes más complejas. Esto sugiere que la complejidad del problema se beneficia de modelos más profundos.

7. **Validación cruzada**: Con la validación cruzada de 5 pliegues, el modelo mostró una precisión promedio del **62.49%**. Aunque inferior al modelo inicial entrenado, esto proporciona una evaluación más robusta de la capacidad de generalización del modelo, ayudando a entender su comportamiento en diferentes subconjuntos de datos.

8. **Curva ROC**: La curva ROC multiclase mostró resultados positivos en todas las clases, con AUCs que variaron entre 0.89 y 0.97. Esto demuestra que el modelo es efectivo en la clasificación de la mayoría de las categorías, con algunas clases mostrando una separación más clara que otras.

Conclusión:

Este experimento demostró cómo una red neuronal profunda puede lograr resultados impresionantes en la clasificación de texto multiclase, alcanzando una precisión máxima del **96.46%** en el conjunto de entrenamiento. Sin embargo, la brecha de rendimiento entre entrenamiento y validación destaca la importancia de las técnicas de regularización y el ajuste de hiperparámetros.

Como estudiante de **7mo semestre** en la clase de **Inteligencia Artificial Avanzada para Ciencia de Datos**, este proyecto me permitió aplicar de manera práctica los conceptos avanzados de redes neuronales, ajuste de hiperparámetros y técnicas de validación. Los resultados obtenidos me proporcionaron una visión integral de la importancia de elegir adecuadamente la arquitectura de la red y la necesidad de evitar el sobreajuste.