

# ACTIVIDAD 3: Deep Learning para Clasificación de Texto

Desarrollado por:

- Nicolás Felipe Trujillo Montero
- Jesús Carlos Avecilla de la Herrán

En esta actividad vamos a trabajar en clasificar textos. Se recorrerá todo el proceso desde traer el dataset hasta proceder a dicha clasificación. Durante la actividad se llevarán a cabo muchos procesos como la creación de un vocabulario, el uso de embeddings y la creación de modelos.

Las cuestiones presentes en esta actividad están basadas en un Notebook creado por François Chollet, uno de los creadores de Keras y autor del libro "Deep Learning with Python".

En este Notebook se trabaja con el dataset "Newsgroup20" que contiene aproximadamente 20000 mensajes que pertenecen a 20 categorías diferentes.

El objetivo es entender los conceptos que se trabajan y ser capaz de hacer pequeñas experimentaciones para mejorar el Notebook creado.

In [2]:

```
# Basado en:  
# https://keras.io/examples/nlp/pretrained_word_embeddings/
```

#Librerías

In [3]:

```
import numpy as np  
import tensorflow as tf  
from tensorflow import keras
```

## Descarga de Datos

In [4]:

```
data_path = keras.utils.get_file(  
    "news20.tar.gz",  
    "http://www.cs.cmu.edu/afs/cs.cmu.edu/project/theo-20/www/data/news20.tar.gz",  
    untar=True,  
)
```

```
Downloading data from http://www.cs.cmu.edu/afs/cs.cmu.edu/project/theo-2  
0/www/data/news20.tar.gz (http://www.cs.cmu.edu/afs/cs.cmu.edu/project/the  
o-20/www/data/news20.tar.gz)  
17329808/17329808 [=====] - 18s 1us/step
```

In [5]:

```
import os
import pathlib

#Estructura de directorios del dataset
data_dir = pathlib.Path(data_path).parent / "20_newsgroup"
dirnames = os.listdir(data_dir)
print("Number of directories:", len(dirnames))
print("Directory names:", dirnames)
```

```
Number of directories: 20
Directory names: ['alt.atheism', 'comp.graphics', 'comp.os.ms-windows.mis
c', 'comp.sys.ibm.pc.hardware', 'comp.sys.mac.hardware', 'comp.windows.x',
'misc.forsale', 'rec.autos', 'rec.motorcycles', 'rec.sport.baseball', 're
c.sport.hockey', 'sci.crypt', 'sci.electronics', 'sci.med', 'sci.space',
'soc.religion.christian', 'talk.politics.guns', 'talk.politics.mideast',
'talk.politics.misc', 'talk.religion.misc']
```

In [6]:

```
#Algunos archivos de la categoria "com.graphics"
fnames = os.listdir(data_dir / "comp.graphics")
print("Number of files in comp.graphics:", len(fnames))
print("Some example filenames:", fnames[:5])
```

```
Number of files in comp.graphics: 1000
Some example filenames: ['37261', '37913', '37914', '37915', '37916']
```

In [7]:

```
#Ejemplo de un texto de la categoría "com.graphics"
print(open(data_dir / "comp.graphics" / "39625").read())
```

Path: cantaloupe.srv.cs.cmu.edu!rochester!udel!gatech!howland.reston.ans.net!newserver.jvnc.net!castor.hahnemann.edu!hal.hahnemann.edu!brennan  
From: brennan@hal.hahnemann.edu  
Newsgroups: comp.graphics  
Subject: .GIFs on a Tek401x ??  
Date: 15 MAY 93 14:29:54 EST  
Organization: Hahnemann University  
Lines: 14  
Message-ID: <15MAY93.14295461@hal.hahnemann.edu>  
NNTP-Posting-Host: hal.hahnemann.edu

I was skimming through a few gophers and bumped into one at NIH with a database that included images in .GIF format. While I have not yet worked out the kinks of getting the gopher client to call an X viewer, I figure that the majority of the users here are not in an X11 environment - instead using DOS and MS-Kermit.

With Kermit supporting Tek4010 emulation for graphics display, does anyone know of a package that would allow a Tek to display a .GIF image? It would be of more use to the local population to plug something of this sort in as the 'picture' command instead of XView or XLoadImage ...

andrew. (brennan@hal.hahnemann.edu)

**Pregunta 1 (0.5 puntos): Escribe un texto de la categoría 'sci.electronics'**

In [8]:

```
# Consultamos aquellos ficheros en la carpeta sci.electronics
# y recogemos algunos ejemplos
fnames2 = os.listdir(data_dir / "sci.electronics")
print("Number of files in sci.electronics:", len(fnames2))
print("Some example filenames:", fnames2[:5])
```

Number of files in sci.electronics: 1000  
Some example filenames: ['52434', '52446', '52464', '52717', '52718']

In [9]:

```
# Ejemplo de un texto de la categoría "sci.electronics"
print(open(data_dir / "sci.electronics" / "52717").read())
```

Newsgroups: sci.electronics  
Path: cantaloupe.srv.cs.cmu.edu!crabapple.srv.cs.cmu.edu!news  
From: jml@norman.vi.ri.cmu.edu  
Subject: Re: Radar Jammers And Stealth Cars  
Message-ID: <C518wH.H9x.1@cs.cmu.edu>  
Sender: news@cs.cmu.edu (Usenet News System)  
Nntp-Posting-Host: westend.vi.ri.cmu.edu  
Reply-To: jml@visus.com  
Organization: School of Computer Science, Carnegie Mellon  
References: <C4rwDH.BuI@csn.org>  
Distribution: usa  
Date: Mon, 5 Apr 1993 22:53:02 GMT  
Lines: 6

Eric H. Taylor writes  
> ... If you are determined  
> to go faster, get an airplane. They dont have speed limits.

Just don't make a habit of buzzing your local airport at >200 knots  
(250 knots if you're flying a jet). :-)

In [10]:

```
samples = []
labels = []
class_names = []
class_index = 0
for dirname in sorted(os.listdir(data_dir)):
    class_names.append(dirname)
    dirpath = data_dir / dirname
    fnames = os.listdir(dirpath)
    print("Processing %s, %d files found" % (dirname, len(fnames)))
    for fname in fnames:
        fpath = dirpath / fname
        f = open(fpath, encoding="latin-1")
        content = f.read()
        lines = content.split("\n")
        lines = lines[10:]
        content = "\n".join(lines)
        samples.append(content)
        labels.append(class_index)
    class_index += 1

print("Classes:", class_names)
print("Number of samples:", len(samples))
```

```
Processing alt.atheism, 1000 files found
Processing comp.graphics, 1000 files found
Processing comp.os.ms-windows.misc, 1000 files found
Processing comp.sys.ibm.pc.hardware, 1000 files found
Processing comp.sys.mac.hardware, 1000 files found
Processing comp.windows.x, 1000 files found
Processing misc.forsale, 1000 files found
Processing rec.autos, 1000 files found
Processing rec.motorcycles, 1000 files found
Processing rec.sport.baseball, 1000 files found
Processing rec.sport.hockey, 1000 files found
Processing sci.crypt, 1000 files found
Processing sci.electronics, 1000 files found
Processing sci.med, 1000 files found
Processing sci.space, 1000 files found
Processing soc.religion.christian, 997 files found
Processing talk.politics.guns, 1000 files found
Processing talk.politics.mideast, 1000 files found
Processing talk.politics.misc, 1000 files found
Processing talk.religion.misc, 1000 files found
Classes: ['alt.atheism', 'comp.graphics', 'comp.os.ms-windows.misc', 'comp.sys.ibm.pc.hardware', 'comp.sys.mac.hardware', 'comp.windows.x', 'misc.forsale', 'rec.autos', 'rec.motorcycles', 'rec.sport.baseball', 'rec.sport.hockey', 'sci.crypt', 'sci.electronics', 'sci.med', 'sci.space', 'soc.religion.christian', 'talk.politics.guns', 'talk.politics.mideast', 'talk.politics.misc', 'talk.religion.misc']
Number of samples: 19997
```

## Mezclando los datos para separarlos en Training y Test

In [11]:

```
# Shuffle the data
seed = 1337
rng = np.random.RandomState(seed)
rng.shuffle(samples)
rng = np.random.RandomState(seed)
rng.shuffle(labels)

# Extract a training & validation split
validation_split = 0.2
num_validation_samples = int(validation_split * len(samples))
train_samples = samples[:num_validation_samples]
val_samples = samples[-num_validation_samples:]
train_labels = labels[:num_validation_samples]
val_labels = labels[-num_validation_samples:]
```

**Pregunta 2 (0.5 puntos): ¿Por qué mezclamos los datos antes de separarlos en entrenamiento y validación?**

Mezclamos los datos ya que es una forma de poder evitar el overfitting en los modelos de Machine Learning, ya que si no mezclaramos todas las muestras estarían clusterizadas ya que hemos ido agregando al conjunto de las muestras por tipo, de forma que:

- De 0 a 999 son muestras de alt.atheism.
- De 1000 a 1999 son muestras de comp.graphics.
- ...

**Pregunta 3 (1 punto): ¿Por qué estás seguro que en la división aleatoria cada muestra coincide con su etiqueta correcta?**

Porque esto se controla mediante la semilla (seed), de forma que, si mezclamos las 19997 muestras de una forma, las etiquetas de esas 19997 muestras van a mezclarse de la misma forma, quedando en las posiciones nuevas, tanto en samples, como en labels, la equivalencia correspondiente.

## Tokenización de las palabras con TextVectorization

In [12]:

```
from tensorflow.keras.layers import TextVectorization
vectorizer = TextVectorization(max_tokens=20000, output_sequence_length=200)
text_ds = tf.data.Dataset.from_tensor_slices(train_samples).batch(128)
vectorizer.adapt(text_ds)
```

In [13]:

```
vectorizer.get_vocabulary()[:5]
```

Out[13]:

```
['', '[UNK]', 'the', 'to', 'of']
```

In [14]:

```
len(vectorizer.get_vocabulary())
```

Out[14]:

```
20000
```

**Pregunta 4 (2 puntos): En la construcción del vocabulario hemos limitado el número de tokens a 20.000 ¿Podrías indicar el número de token diferentes o tamaño del vocabulario sin limitar el número de tokens? Es decir, ¿Cuántas palabras diferentes existen en los documentos procesados como instancias?**

In [15]:

```
# En este caso, no hemos limitado el maximo de tokens, pero trabajaremos con La  
# misma cantidad para los batch  
vectorizer_noLimits = TextVectorization(max_tokens=None, output_sequence_length=200)  
vectorizer_noLimits.adapt(text_ds)
```

In [21]:

```
# Para comprobar que se obtienen valores únicos (tokens), se ha utilizado  
# los conjuntos que eliminan los valores duplicados obteniéndose el mismo numero  
# Verificamos la propiedad de que los tokens sean únicos  
tokens = vectorizer_noLimits.get_vocabulary()  
tokens_filtered = set(tokens)  
print(len(tokens), len(tokens_filtered))
```

```
158191 158191
```

## Viendo la salida de Vectorizer

In [22]:

```
output = vectorizer([["the cat sat on the mat"]])  
output.numpy()[0, :6]
```

Out[22]:

```
array([    2, 3457, 1682,    15,      2, 5776], dtype=int64)
```

In [23]:

output

Out[23]:

In [24]:

```
voc = vectorizer.get_vocabulary()  
word_index = dict(zip(voc, range(len(voc))))
```

In [27]:

```
test = ["the", "cat", "sat", "on", "the", "mat"]  
[word_index[w] for w in test]
```

Out[27]:

[2, 3457, 1682, 15, 2, 5776]

**Pregunta 5 (1 punto):** Arriba tenemos la codificación de la frase "the cat sat on the mat". Imagina la siguiente situación. La salida de `vectorizer()` para codificar los tokens ["El", "gato", "está", "sobre", "el", "tejado"] es la siguiente [1, 121, 405, 1, 45, 4561]. Si cada uno de los valores indica el índice en el que se encuentra cada palabra en el array creado para codificarla. ¿Podría ser correcta esta salida?

Los indices [1, 121, 405, 1, 45, 4561] codificados para los tokens ["El", "gato", "está", "sobre", "el", "tejado"] son erróneos ya que el indice 1 debería corresponderse con un único valor, pero en este caso se corresponde tanto con el token "El", como el token "sobre", por lo que es incorrecto.

También, un apartado a tener en cuenta es que el token "El" y el token "el" son distintos. El vectorizer es case sensitive, por lo que no considera que sean el mismo (En este caso no se encuentra dicho error)

# Tokenización de los datos de entrenamiento y validación

In [28]:

```
x_train = vectorizer(np.array([[s] for s in train_samples])).numpy()
x_val = vectorizer(np.array([[s] for s in val_samples])).numpy()

y_train = np.array(train_labels)
y_val = np.array(val_labels)
```

## Creación del modelo con un embedding hecho a mano y con redes neuronales convolucionales

Aunque las Redes Neuronales Convolucionales se desarrollaron originalmente para procesamiento de imágenes, se pueden aplicar a problemas de procesamiento de lenguaje natural gracias a su capacidad para capturar patrones locales y globales en los datos de entrada.

Previa a la alimentación de las capas convolucionales, vamos a incluir una capa de Embedding. Como hemos visto en clase, un embedding de palabras se refiere a una representación vectorial densa de una palabra en un espacio de características continuo de alta dimensión. Como resultado del embedding vamos a capturar la semántica de las palabras de manera que palabras similares tengan representaciones vectoriales similares.

A continuación se incluye la creación de un modelo con un embedding hecho a mano y usando Redes Neuronales Convolucionales

In [29]:

```
modeloEmbeddingManualConvolucionales = keras.models.Sequential()
modeloEmbeddingManualConvolucionales.add(keras.layers.Embedding(20000, 10, input_length=2))
modeloEmbeddingManualConvolucionales.add(keras.layers.Conv1D(128, 5, activation="relu"))
modeloEmbeddingManualConvolucionales.add(keras.layers.MaxPooling1D(5))
modeloEmbeddingManualConvolucionales.add(keras.layers.Conv1D(128, 5, activation="relu"))
modeloEmbeddingManualConvolucionales.add(keras.layers.MaxPooling1D(5))
modeloEmbeddingManualConvolucionales.add(keras.layers.Conv1D(128, 5, activation="relu"))
modeloEmbeddingManualConvolucionales.add(keras.layers.GlobalMaxPooling1D())
modeloEmbeddingManualConvolucionales.add(keras.layers.Dense(128, activation='relu'))
modeloEmbeddingManualConvolucionales.add(keras.layers.Dropout(0.3))
modeloEmbeddingManualConvolucionales.add(keras.layers.Dense(20, activation='softmax'))
```

In [30]:

```
modeloEmbeddingManualConvolucionales.compile(optimizer='adam', loss='binary_crossentropy'  
modeloEmbeddingManualConvolucionales.compile(loss="sparse_categorical_crossentropy", opti  
modeloEmbeddingManualConvolucionales.fit(x_train, y_train, batch_size=128, epochs=20, val
```

Epoch 1/20  
125/125 [=====] - 5s 34ms/step - loss: 2.9176 - a  
cc: 0.0764 - val\_loss: 2.6499 - val\_acc: 0.1138  
Epoch 2/20  
125/125 [=====] - 4s 33ms/step - loss: 2.5168 - a  
cc: 0.1257 - val\_loss: 2.3660 - val\_acc: 0.1438  
Epoch 3/20  
125/125 [=====] - 4s 32ms/step - loss: 2.3031 - a  
cc: 0.1551 - val\_loss: 2.1712 - val\_acc: 0.1960  
Epoch 4/20  
125/125 [=====] - 4s 31ms/step - loss: 2.1085 - a  
cc: 0.2054 - val\_loss: 2.1520 - val\_acc: 0.1975  
Epoch 5/20  
125/125 [=====] - 4s 31ms/step - loss: 1.9434 - a  
cc: 0.2583 - val\_loss: 1.9478 - val\_acc: 0.2856  
Epoch 6/20  
125/125 [=====] - 4s 31ms/step - loss: 1.7674 - a  
cc: 0.3214 - val\_loss: 1.7746 - val\_acc: 0.3548  
Epoch 7/20  
125/125 [=====] - 4s 31ms/step - loss: 1.5538 - a  
cc: 0.4034 - val\_loss: 1.9604 - val\_acc: 0.3261  
Epoch 8/20  
125/125 [=====] - 4s 31ms/step - loss: 1.3653 - a  
cc: 0.4727 - val\_loss: 1.5161 - val\_acc: 0.4699  
Epoch 9/20  
125/125 [=====] - 4s 31ms/step - loss: 1.2111 - a  
cc: 0.5363 - val\_loss: 1.4676 - val\_acc: 0.4994  
Epoch 10/20  
125/125 [=====] - 4s 31ms/step - loss: 1.0888 - a  
cc: 0.5979 - val\_loss: 1.5899 - val\_acc: 0.4826  
Epoch 11/20  
125/125 [=====] - 4s 31ms/step - loss: 0.9696 - a  
cc: 0.6480 - val\_loss: 1.4149 - val\_acc: 0.5551  
Epoch 12/20  
125/125 [=====] - 4s 31ms/step - loss: 0.8911 - a  
cc: 0.6841 - val\_loss: 1.5465 - val\_acc: 0.5524  
Epoch 13/20  
125/125 [=====] - 4s 31ms/step - loss: 0.8067 - a  
cc: 0.7163 - val\_loss: 1.5695 - val\_acc: 0.5424  
Epoch 14/20  
125/125 [=====] - 4s 31ms/step - loss: 0.7491 - a  
cc: 0.7395 - val\_loss: 1.5226 - val\_acc: 0.5746  
Epoch 15/20  
125/125 [=====] - 4s 31ms/step - loss: 0.6960 - a  
cc: 0.7634 - val\_loss: 1.4834 - val\_acc: 0.5836  
Epoch 16/20  
125/125 [=====] - 4s 31ms/step - loss: 0.6289 - a  
cc: 0.7868 - val\_loss: 1.7316 - val\_acc: 0.5624  
Epoch 17/20  
125/125 [=====] - 4s 31ms/step - loss: 0.5858 - a  
cc: 0.8017 - val\_loss: 1.5374 - val\_acc: 0.5989  
Epoch 18/20  
125/125 [=====] - 4s 31ms/step - loss: 0.5397 - a  
cc: 0.8180 - val\_loss: 1.6914 - val\_acc: 0.5961  
Epoch 19/20  
125/125 [=====] - 4s 31ms/step - loss: 0.4938 - a  
cc: 0.8309 - val\_loss: 1.7946 - val\_acc: 0.5839  
Epoch 20/20  
125/125 [=====] - 4s 31ms/step - loss: 0.4541 - a  
cc: 0.8476 - val\_loss: 1.7327 - val\_acc: 0.5991

Out[30]:

```
<keras.callbacks.History at 0x204078f5c50>
```

**Pregunta 6 (1 punto): ¿Por qué usamos convoluciones de 1 dimensión para procesar texto?**

Son usadas debido a las ventajas que nos ofrecen al trabajar de esa forma, por ejemplo, la mejora en su capacidad de poder analizar y capturar información del contexto del embedding y posibles patrones en los datos analizando cada palabra junto a la de sus vecinos.

Dicha mejora es gracias al uso de las rolling windows, que son el componente base de los kernels/filtros.

Si usaramos convoluciones de más de una dimensión, se estaría considerando más dimensiones, como podrían ser la estructura contextual o la estructura espacial de las palabras en el texto, por lo que, en el campo del procesamiento de texto no es de utilidad.

**Pregunta 7 (2 puntos): Crea un nuevo modelo partiendo del model anterior pero en lugar de Redes Neuronales Convolucionales vamos a utilizar Redes Neuronales Clásicas. Para ello, tras la capa Embedding añade una capa Flatten, una capa densa de 512 neuronas con función de activación relu y una capa Dropout de regulización al 30%. Por último, no olvides incluir la capa de salida con tantas neuronas como clases haya y la función de activación softmax.**

In [38]:

```
# Tamaño de cada imagen
sz_image = [28,28]

# Numero de neuronas en la capa oculta
sz_hid_layer = 512

# Funcion de activacion de la capa oculta
fun_hid_layer = "relu"

# Tipos distintos de tags que clasifican
tags = len(class_names)

# Training del modelo
modeloEmbeddingManualClasicas = keras.models.Sequential()
modeloEmbeddingManualClasicas.add(keras.layers.Embedding(20000, 10, input_length=200))
modeloEmbeddingManualClasicas.add(keras.layers.Flatten(input_shape = [1,200]))
modeloEmbeddingManualClasicas.add(keras.layers.Dense(sz_hid_layer, activation = fun_hid_layer))
modeloEmbeddingManualClasicas.add(keras.layers.Dropout(0.3))
modeloEmbeddingManualClasicas.add(keras.layers.Dense(tags, activation='softmax'))
```

In [39]:

```
modeloEmbeddingManualClasicas.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
modeloEmbeddingManualClasicas.compile(loss="sparse_categorical_crossentropy", optimizer="adam")
modeloEmbeddingManualClasicas.fit(x_train, y_train, batch_size=128, epochs=20, validation_split=0.2)
print(modeloEmbeddingManualClasicas.summary())
```

Epoch 1/20  
125/125 [=====] - 2s 12ms/step - loss: 2.8864 - a  
cc: 0.1010 - val\_loss: 2.6500 - val\_acc: 0.1498  
Epoch 2/20  
125/125 [=====] - 1s 10ms/step - loss: 2.3006 - a  
cc: 0.2540 - val\_loss: 2.1216 - val\_acc: 0.2848  
Epoch 3/20  
125/125 [=====] - 1s 11ms/step - loss: 1.6806 - a  
cc: 0.4710 - val\_loss: 1.6118 - val\_acc: 0.4544  
Epoch 4/20  
125/125 [=====] - 1s 11ms/step - loss: 1.1304 - a  
cc: 0.6555 - val\_loss: 1.2847 - val\_acc: 0.5656  
Epoch 5/20  
125/125 [=====] - 1s 10ms/step - loss: 0.7623 - a  
cc: 0.7738 - val\_loss: 1.0971 - val\_acc: 0.6189  
Epoch 6/20  
125/125 [=====] - 1s 11ms/step - loss: 0.5266 - a  
cc: 0.8500 - val\_loss: 1.0217 - val\_acc: 0.6549  
Epoch 7/20  
125/125 [=====] - 1s 10ms/step - loss: 0.3812 - a  
cc: 0.8895 - val\_loss: 0.9926 - val\_acc: 0.6644  
Epoch 8/20  
125/125 [=====] - 1s 11ms/step - loss: 0.2842 - a  
cc: 0.9166 - val\_loss: 0.9838 - val\_acc: 0.6837  
Epoch 9/20  
125/125 [=====] - 1s 11ms/step - loss: 0.2209 - a  
cc: 0.9337 - val\_loss: 1.0085 - val\_acc: 0.6847  
Epoch 10/20  
125/125 [=====] - 1s 10ms/step - loss: 0.1798 - a  
cc: 0.9466 - val\_loss: 1.0330 - val\_acc: 0.6894  
Epoch 11/20  
125/125 [=====] - 1s 10ms/step - loss: 0.1521 - a  
cc: 0.9520 - val\_loss: 1.0334 - val\_acc: 0.6949  
Epoch 12/20  
125/125 [=====] - 1s 11ms/step - loss: 0.1315 - a  
cc: 0.9567 - val\_loss: 1.0916 - val\_acc: 0.6947  
Epoch 13/20  
125/125 [=====] - 1s 10ms/step - loss: 0.1172 - a  
cc: 0.9597 - val\_loss: 1.0904 - val\_acc: 0.6977  
Epoch 14/20  
125/125 [=====] - 1s 10ms/step - loss: 0.1071 - a  
cc: 0.9610 - val\_loss: 1.0942 - val\_acc: 0.7057  
Epoch 15/20  
125/125 [=====] - 1s 11ms/step - loss: 0.1015 - a  
cc: 0.9619 - val\_loss: 1.1130 - val\_acc: 0.7042  
Epoch 16/20  
125/125 [=====] - 1s 11ms/step - loss: 0.0929 - a  
cc: 0.9644 - val\_loss: 1.1268 - val\_acc: 0.7082  
Epoch 17/20  
125/125 [=====] - 1s 11ms/step - loss: 0.0909 - a  
cc: 0.9635 - val\_loss: 1.1402 - val\_acc: 0.7072  
Epoch 18/20  
125/125 [=====] - 1s 11ms/step - loss: 0.0855 - a  
cc: 0.9649 - val\_loss: 1.1529 - val\_acc: 0.7114  
Epoch 19/20  
125/125 [=====] - 1s 10ms/step - loss: 0.0813 - a  
cc: 0.9647 - val\_loss: 1.1828 - val\_acc: 0.7054  
Epoch 20/20  
125/125 [=====] - 1s 10ms/step - loss: 0.0793 - a  
cc: 0.9656 - val\_loss: 1.1937 - val\_acc: 0.7082  
Model: "sequential\_5"

Layer (type)	Output Shape	Param #
embedding_5 (Embedding)	(None, 200, 10)	200000
flatten_2 (Flatten)	(None, 2000)	0
dense_4 (Dense)	(None, 512)	1024512
dropout_2 (Dropout)	(None, 512)	0
dense_5 (Dense)	(None, 20)	10260

Total params: 1,234,772  
Trainable params: 1,234,772  
Non-trainable params: 0

None

Tras hacer este experimento, es posible que pase una cosa curiosa y es que la red totalmente conectada tenga un mejor comportamiento que la convolucional. En general, las redes neuronales convolucionales tienden a superar a las totalmente conectadas en la mayoría de los problemas de este tipo debido a su capacidad para capturar patrones espaciales y posicionales en los datos de entrada. Dicho esto, en algunos casos, una red neuronal clásica podría superar a una convolucional en un problema de procesamiento de lenguaje natural, especialmente si el conjunto de datos es pequeño y las características relevantes son relativamente simples. En este caso, la mayoría de las categorías puede ser que tengan un vocabulario específico y esa pueda ser una de las razones.

#Creación del modelo con el embedding GloVe y redes neuronales convolucionales

Esta vez, en vez de crear nosotros mismos el embedding, vamos a utilizar Glove. Los vectores de palabras generados por GloVe capturan las relaciones semánticas y sintácticas entre las palabras en un corpus de texto mucho más grande que el nuestro.

In [40]:

```
# !wget http://nlp.stanford.edu/data/glove.6B.zip
# !unzip -q glove.6B.zip
```

"wget" no se reconoce como un comando interno o externo,  
programa o archivo por lotes ejecutable.  
"unzip" no se reconoce como un comando interno o externo,  
programa o archivo por lotes ejecutable.

**Pregunta 8 (1 puntos): El embedding que usa la actividad es el glove.6B. ¿De dónde se han obtenido los textos que se han usado para entrenarlo? ¿Cántos WordVectors lo componen? ¿De cuántas dimensiones tiene versiones? Puedes ayudarte del link <https://nlp.stanford.edu/projects/glove/> (<https://nlp.stanford.edu/projects/glove/>) para buscar la información**

La fuentes de texto que usa glove.6B es una mezcla de obtención de datos entre Wikipedia 2014 + Gigaword (Corpus de textos periodísticos), ambos en inglés.

Se han encontrado 400.000 word vectors, es decir, 400.000 tokens.

Pues existen cuatro ficheros que varian la dimension del vector asociado estando la version de token = vector de tamaño de (1xn). donde n puede ser 50. 100. 200 v 300.

In [46]:

```
# path_to_glove_file = os.path.join(
#     os.path.expanduser("~/"), ".keras/datasets/glove.6B.100d.txt"
# )
path_to_glove_file = "glove.6B/glove.6B.100d.txt"

embeddings_index = {}
with open(path_to_glove_file, encoding="utf-8") as f:
    for line in f:
        word, coefs = line.split(maxsplit=1)
        coefs = np.fromstring(coefs, "f", sep=" ")
        embeddings_index[word] = coefs

print("Found %s word vectors." % len(embeddings_index))
```

Found 400000 word vectors.

Creamos la capa Embedding de keras. Se trata de una matrix numpy donde el valor de cada posición corresponde con el vector pre-entrenado del vocabulario tras el vectorizer

In [51]:

```
num_tokens = len(voc) + 2
embedding_dim = 100
hits = 0
misses = 0

# Prepare embedding matrix
embedding_matrix = np.zeros((num_tokens, embedding_dim))
for word, i in word_index.items():
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
        # Words not found in embedding index will be all-zeros.
        # This includes the representation for "padding" and "OOV"
        embedding_matrix[i] = embedding_vector
        hits += 1
    else:
        misses += 1
print("Converted %d words (%d misses)" % (hits, misses))
```

Converted 18019 words (1981 misses)

Cargamos el embedding como una capa keras. Al poner trainable=False nos quedamos con los valores del modelo pre-entrenado, es decir, no actualizamos estos valores a lo largo del entrenamiento

In [52]:

```
from tensorflow.keras.layers import Embedding

embedding_layer = Embedding(
    num_tokens,
    embedding_dim,
    embeddings_initializer=keras.initializers.Constant(embedding_matrix),
    trainable=False,
)
```

In [53]:

```
from tensorflow.keras import layers

int_sequences_input = keras.Input(shape=(None,), dtype="int64")
embedded_sequences = embedding_layer(int_sequences_input)
x = layers.Conv1D(128, 5, activation="relu")(embedded_sequences)
x = layers.MaxPooling1D(5)(x)
x = layers.Conv1D(128, 5, activation="relu")(x)
x = layers.MaxPooling1D(5)(x)
x = layers.Conv1D(128, 5, activation="relu")(x)
x = layers.GlobalMaxPooling1D()(x)
x = layers.Dense(128, activation="relu")(x)
x = layers.Dropout(0.5)(x)
preds = layers.Dense(len(class_names), activation="softmax")(x)
modelEmbeddingGloveConvolucionales = keras.Model(int_sequences_input, preds)
modelEmbeddingGloveConvolucionales.summary()
```

Model: "model"

Layer (type)	Output Shape	Param #
=====		
input_1 (InputLayer)	[(None, None)]	0
embedding_6 (Embedding)	(None, None, 100)	2000200
conv1d_3 (Conv1D)	(None, None, 128)	64128
max_pooling1d_2 (MaxPooling 1D)	(None, None, 128)	0
conv1d_4 (Conv1D)	(None, None, 128)	82048
max_pooling1d_3 (MaxPooling 1D)	(None, None, 128)	0
conv1d_5 (Conv1D)	(None, None, 128)	82048
global_max_pooling1d_1 (GlobalMaxPooling1D)	(None, 128)	0
dense_6 (Dense)	(None, 128)	16512
dropout_3 (Dropout)	(None, 128)	0
dense_7 (Dense)	(None, 20)	2580
=====		
Total params: 2,247,516		
Trainable params: 247,316		
Non-trainable params: 2,000,200		

In [54]:

```
modelEmbeddingGloveConvolucionales.compile(    loss="sparse_categorical_crossentropy", opti
modelEmbeddingGloveConvolucionales.fit(x_train, y_train, batch_size=128, epochs=20, validation_d
predictions = modelEmbeddingGloveConvolucionales.predict(x_val)
```

Epoch 1/20  
125/125 [=====] - 7s 52ms/step - loss: 2.7041 - a  
cc: 0.1299 - val\_loss: 2.2550 - val\_acc: 0.2028  
Epoch 2/20  
125/125 [=====] - 6s 50ms/step - loss: 1.9749 - a  
cc: 0.3152 - val\_loss: 1.6767 - val\_acc: 0.4169  
Epoch 3/20  
125/125 [=====] - 6s 51ms/step - loss: 1.5574 - a  
cc: 0.4640 - val\_loss: 1.3134 - val\_acc: 0.5406  
Epoch 4/20  
125/125 [=====] - 6s 50ms/step - loss: 1.3141 - a  
cc: 0.5446 - val\_loss: 1.1611 - val\_acc: 0.6017  
Epoch 5/20  
125/125 [=====] - 6s 52ms/step - loss: 1.1290 - a  
cc: 0.6113 - val\_loss: 1.0822 - val\_acc: 0.6349  
Epoch 6/20  
125/125 [=====] - 6s 50ms/step - loss: 1.0038 - a  
cc: 0.6526 - val\_loss: 1.0541 - val\_acc: 0.6417  
Epoch 7/20  
125/125 [=====] - 7s 53ms/step - loss: 0.8938 - a  
cc: 0.6873 - val\_loss: 1.0626 - val\_acc: 0.6419  
Epoch 8/20  
125/125 [=====] - 6s 52ms/step - loss: 0.7929 - a  
cc: 0.7241 - val\_loss: 0.9937 - val\_acc: 0.6734  
Epoch 9/20  
125/125 [=====] - 6s 51ms/step - loss: 0.7043 - a  
cc: 0.7535 - val\_loss: 1.0021 - val\_acc: 0.6799  
Epoch 10/20  
125/125 [=====] - 6s 52ms/step - loss: 0.6197 - a  
cc: 0.7820 - val\_loss: 1.0484 - val\_acc: 0.6749  
Epoch 11/20  
125/125 [=====] - 6s 51ms/step - loss: 0.5525 - a  
cc: 0.8066 - val\_loss: 0.9607 - val\_acc: 0.7094  
Epoch 12/20  
125/125 [=====] - 6s 50ms/step - loss: 0.4910 - a  
cc: 0.8297 - val\_loss: 0.9344 - val\_acc: 0.7184  
Epoch 13/20  
125/125 [=====] - 6s 50ms/step - loss: 0.4119 - a  
cc: 0.8557 - val\_loss: 1.0458 - val\_acc: 0.7004  
Epoch 14/20  
125/125 [=====] - 6s 51ms/step - loss: 0.3656 - a  
cc: 0.8714 - val\_loss: 1.0464 - val\_acc: 0.7114  
Epoch 15/20  
125/125 [=====] - 7s 54ms/step - loss: 0.3343 - a  
cc: 0.8858 - val\_loss: 1.1506 - val\_acc: 0.7049  
Epoch 16/20  
125/125 [=====] - 8s 61ms/step - loss: 0.2848 - a  
cc: 0.9032 - val\_loss: 1.1690 - val\_acc: 0.6964  
Epoch 17/20  
125/125 [=====] - 7s 56ms/step - loss: 0.2567 - a  
cc: 0.9139 - val\_loss: 1.2911 - val\_acc: 0.6972  
Epoch 18/20  
125/125 [=====] - 7s 57ms/step - loss: 0.2393 - a  
cc: 0.9205 - val\_loss: 1.2865 - val\_acc: 0.7082  
Epoch 19/20  
125/125 [=====] - 6s 52ms/step - loss: 0.2158 - a  
cc: 0.9297 - val\_loss: 1.2998 - val\_acc: 0.7032  
Epoch 20/20  
125/125 [=====] - 7s 53ms/step - loss: 0.1949 - a

```
cc: 0.9365 - val_loss: 1.3598 - val_acc: 0.6989
```

```
125/125 [=====] - 1s 5ms/step
```

Pregunta 9 (1 puntos): A estas alturas ya te habrás dado cuenta que los Transformer que has utilizado en la asignatura de Procesamiento de Lenguaje Natural son, probablemente, los que mejores resultados obtienen. Sin utilizar Transformer, crea un modelo tú, bien ajustando los hiperparámetros de los modelos anteriores o uno completamente nuevo, que llegue al menos al 72% accuracy

In [75]:

```
sz_hid_layer = 128
fun_hid_layer = "relu"

modeloFinal = keras.models.Sequential()
modeloFinal.add(keras.layers.Embedding(20000, 10, input_length=200))
modeloFinal.add(keras.layers.Flatten(input_shape = [1,200]))
modeloFinal.add(keras.layers.Dense(sz_hid_layer, activation = fun_hid_layer))
modeloFinal.add(keras.layers.Dropout(0.5))
modeloFinal.add(keras.layers.Dense(tags, activation='softmax'))
```

In [78]:

```
modeloFinal.compile(optimizer='RMSprop', loss='binary_crossentropy', metrics=['accuracy'])
modeloFinal.compile(loss="sparse_categorical_crossentropy", optimizer="rmsprop", metrics=
modeloFinal.fit(x_train, y_train, batch_size=256, epochs=25, validation_data=(x_val, y_val))
print(modeloFinal.summary())
```

Epoch 1/25  
63/63 [=====] - 1s 8ms/step - loss: 0.0660 - acc: 0.9699 - val\_loss: 1.2202 - val\_acc: 0.7497  
Epoch 2/25  
63/63 [=====] - 0s 7ms/step - loss: 0.0633 - acc: 0.9724 - val\_loss: 1.2207 - val\_acc: 0.7494  
Epoch 3/25  
63/63 [=====] - 0s 7ms/step - loss: 0.0643 - acc: 0.9709 - val\_loss: 1.2260 - val\_acc: 0.7489  
Epoch 4/25  
63/63 [=====] - 0s 7ms/step - loss: 0.0633 - acc: 0.9712 - val\_loss: 1.2384 - val\_acc: 0.7482  
Epoch 5/25  
63/63 [=====] - 0s 7ms/step - loss: 0.0595 - acc: 0.9729 - val\_loss: 1.2556 - val\_acc: 0.7487  
Epoch 6/25  
63/63 [=====] - 0s 7ms/step - loss: 0.0595 - acc: 0.9726 - val\_loss: 1.2574 - val\_acc: 0.7474  
Epoch 7/25  
63/63 [=====] - 0s 7ms/step - loss: 0.0630 - acc: 0.9721 - val\_loss: 1.2555 - val\_acc: 0.7469  
Epoch 8/25  
63/63 [=====] - 0s 7ms/step - loss: 0.0603 - acc: 0.9716 - val\_loss: 1.2617 - val\_acc: 0.7482  
Epoch 9/25  
63/63 [=====] - 0s 7ms/step - loss: 0.0611 - acc: 0.9715 - val\_loss: 1.2748 - val\_acc: 0.7449  
Epoch 10/25  
63/63 [=====] - 0s 7ms/step - loss: 0.0592 - acc: 0.9716 - val\_loss: 1.3107 - val\_acc: 0.7477  
Epoch 11/25  
63/63 [=====] - 0s 8ms/step - loss: 0.0583 - acc: 0.9726 - val\_loss: 1.2972 - val\_acc: 0.7472  
Epoch 12/25  
63/63 [=====] - 0s 7ms/step - loss: 0.0581 - acc: 0.9717 - val\_loss: 1.2882 - val\_acc: 0.7487  
Epoch 13/25  
63/63 [=====] - 0s 7ms/step - loss: 0.0583 - acc: 0.9723 - val\_loss: 1.3039 - val\_acc: 0.7467  
Epoch 14/25  
63/63 [=====] - 0s 7ms/step - loss: 0.0600 - acc: 0.9707 - val\_loss: 1.2995 - val\_acc: 0.7479  
Epoch 15/25  
63/63 [=====] - 0s 7ms/step - loss: 0.0572 - acc: 0.9726 - val\_loss: 1.3056 - val\_acc: 0.7484  
Epoch 16/25  
63/63 [=====] - 0s 7ms/step - loss: 0.0566 - acc: 0.9714 - val\_loss: 1.3297 - val\_acc: 0.7439  
Epoch 17/25  
63/63 [=====] - 0s 7ms/step - loss: 0.0551 - acc: 0.9724 - val\_loss: 1.3164 - val\_acc: 0.7482  
Epoch 18/25  
63/63 [=====] - 0s 7ms/step - loss: 0.0554 - acc: 0.9719 - val\_loss: 1.3275 - val\_acc: 0.7442  
Epoch 19/25  
63/63 [=====] - 0s 7ms/step - loss: 0.0540 - acc: 0.9726 - val\_loss: 1.3480 - val\_acc: 0.7432  
Epoch 20/25  
63/63 [=====] - 0s 7ms/step - loss: 0.0550 - acc: 0.9721 - val\_loss: 1.3576 - val\_acc: 0.7452  
Epoch 21/25

```

63/63 [=====] - 0s 7ms/step - loss: 0.0529 - acc: 0.9727 - val_loss: 1.3689 - val_acc: 0.7424
Epoch 22/25
63/63 [=====] - 0s 7ms/step - loss: 0.0547 - acc: 0.9721 - val_loss: 1.3547 - val_acc: 0.7487
Epoch 23/25
63/63 [=====] - 0s 7ms/step - loss: 0.0525 - acc: 0.9741 - val_loss: 1.3605 - val_acc: 0.7474
Epoch 24/25
63/63 [=====] - 0s 7ms/step - loss: 0.0533 - acc: 0.9731 - val_loss: 1.3806 - val_acc: 0.7507
Epoch 25/25
63/63 [=====] - 0s 7ms/step - loss: 0.0529 - acc: 0.9729 - val_loss: 1.3657 - val_acc: 0.7512
Model: "sequential_13"

```

Layer (type)	Output Shape	Param #
<hr/>		
embedding_14 (Embedding)	(None, 200, 10)	200000
flatten_10 (Flatten)	(None, 2000)	0
dense_30 (Dense)	(None, 128)	256128
dropout_11 (Dropout)	(None, 128)	0
dense_31 (Dense)	(None, 20)	2580
<hr/>		
Total params: 458,708		
Trainable params: 458,708		
Non-trainable params: 0		

---

None

Hemos utilizado el modelo de red neuronal clásico utilizando el mismo número de capas que la estructura original de ANN, pero hemos cambiado el optimizador y el tamaño del batch. De esta forma, podemos conseguir un mayor accuracy.

El optimizador utiliza una optimización del método de descenso del gradiente basado en el algoritmo de RMSprop que itera en cada época utilizando el descenso del gradiente.