

# Clasificación de opiniones en tuits en español. Grupo: Equipo B.

David López Pretel - derwey@correo.ugr.es

Rshad Zhran - rshadzhran@correo.ugr.es

Míriam Mengíbar Rodríguez - mirismr@correo.ugr.es

Néstor Rodríguez Vico - nrv23@correo.ugr.es

25 de abril de 2019

# Índice

<b>1. Introducción.</b>	<b>4</b>
<b>2. Metodología de trabajo.</b>	<b>4</b>
<b>3. Estudio y preparación de los datos.</b>	<b>5</b>
3.1. Lectura de datos. . . . .	5
3.2. Preprocesamiento. . . . .	5
3.3. Oversampling. . . . .	8
<b>4. Modelos de redes.</b>	<b>8</b>
4.1. tfidf_rnn. . . . .	9
4.2. calculated_embeddings_rnn. . . . .	10
4.3. pretrain_embeddings_rnn. . . . .	12
4.4. sigmoid_pretrain_embeddings_rnn. . . . .	13
4.5. stacked_lstm_rnn. . . . .	15
4.6. adadelta_rnn. . . . .	17
4.7. calculated_embeddings_LSTM_CONV. . . . .	18
4.8. pretrain_embeddings_LSTM_CONV. . . . .	20
4.9. big_LSTM_CONV_rnn. . . . .	22
4.10. dropout_LSTM_CONV_rnn. . . . .	23
4.11. bidirectional_lstm_rnn. . . . .	25
4.12. pretrained_embeddings_cnn_bidirectional_LSTM. . . . .	26
<b>5. Experimentos.</b>	<b>28</b>
5.1. tfidf_rnn. . . . .	28
5.2. calculated_embeddings_rnn. . . . .	29
5.3. pretrain_embeddings_rnn. . . . .	31
5.4. sigmoid_pretrain_embeddings_rnn. . . . .	32
5.5. epochs100_pretrain_embeddings_rnn. . . . .	33
5.6. stacked_lstm_rnn. . . . .	34
5.7. adadelta_rnn. . . . .	35
5.8. adam_lr_0005_rnn. . . . .	36
5.9. pretrain_embeddings_LSTM_CONV. . . . .	37
5.10. preprocess_tfidf_rnn. . . . .	38
5.11. preprocess_calculated_embeddings_rnn. . . . .	39
5.12. preprocess_pretrain_embeddings_rnn. . . . .	40
5.13. preprocess_calculated_embeddings_LSTM_CONV. . . . .	41
5.14. epochs50_preprocess_calculated_embeddings_LSTM_CONV. . . . .	43
5.15. preprocess_pretrain_embeddings_LSTM_CONV. . . . .	44
5.16. big_LSTM_CONV_rnn. . . . .	45
5.17. dropout_LSTM_CONV_rnn. . . . .	46
5.18. pretrain_embeddings_LSTM_CONV_OVERSAMPLING. . . . .	48
5.19. bidirectional_lstm_rnn. . . . .	49
5.20. fasttext_sbwc_bidirectional_lstm_rnn. . . . .	50
5.21. glove_sbwc_i25_bidirectional_lstm_rnn. . . . .	51
5.22. SBW_vectors_300_min5_bidirectional_lstm_rnn. . . . .	53
5.23. wiki_es_bidirectional_lstm_rnn. . . . .	54
5.24. pretrained_embeddings_cnn_bidirectional_LSTM. . . . .	55

<b>6. Otros experimentos.</b>	<b>56</b>
6.1. Variables calculadas. . . . .	56
6.2. GRU. . . . .	57
<b>7. Resultados.</b>	<b>57</b>
<b>8. Bibliografía.</b>	<b>59</b>

## 1. Introducción.

En la actualidad, el auge de las redes sociales y su uso continuado provocan una cantidad enorme de datos de los que podemos sacar información muy valiosa. Es por ello que la minería de medios sociales es una rama de la ciencia de datos que trata de hacer esto.

En concreto, este trabajo trata de clasificar un *tuit* en base a la opinión que el autor expresa, pudiendo clasificar un *tuit* en alguna de las siguientes clases:

- *N*: *tuit* con opinión negativa. En nuestro conjunto de entrenamiento hay 418 *tuits* asociados a esta clase.
- *P*: *tuit* con opinión positiva. En nuestro conjunto de entrenamiento hay 318 *tuits* asociados a esta clase.
- *NEU*: *tuit* con opinión neutra. En nuestro conjunto de entrenamiento hay 133 *tuits* asociados a esta clase.
- *NONE*: *tuit* sin opinión. En nuestro conjunto de entrenamiento hay 139 *tuits* asociados a esta clase.

Para esta tarea de clasificación hemos hecho uso de redes neuronales usando la librería *keras* y *tensorflow*. Además, se detallará el preprocesamiento realizado a los *tuits*, las distintas arquitecturas, los experimentos realizados y, por último, los resultados obtenidos.

## 2. Metodología de trabajo.

Para la elección de los modelos en la competición, se ha usado la estrategia *K-Fold Cross Validation* con  $K=5$ . En nuestro caso, hemos unido el conjunto de *train* y *validation* y, sobre este nuevo conjunto, hemos realizado las distintas particiones.

Para cada modelo usado, se ha pintado una gráfica donde se muestra el como evoluciona el valor de la función *loss* en el conjunto de entrenamiento y el conjunto de validación para cada partición del *K-Fold Cross Validation* a lo largo de las épocas de aprendizaje. También, se ha pintado la media de todas las particiones.

A la hora de comparar las clases predichas con las clases reales se ha usado la métrica *F1 score*, la cual se define como:

$$2 * \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} \quad (1)$$

Dicha métrica se ha calculado en dos versiones:

- *micro*: Calcula las métricas globalmente contando el total de los verdaderos positivos, falsos negativos y falsos positivos.
- *macro*: Calcula las métricas para cada etiqueta y encuentre su media no ponderada. Esta no tiene en cuenta el desequilibrio de la etiqueta.

Al igual que se ha hecho con la gráfica de la función *loss*, se ha calculado el valor de las métricas *micro-f1* y *macro-f1* para cada partición y se han usado como métricas finales del modelo la media de las mismas.

### 3. Estudio y preparación de los datos.

En esta sección se describe cómo se han leído los datos, el procesamiento realizado a los *tuits* y cómo se ha balanceado el conjunto de datos mediante *oversampling*.

#### 3.1. Lectura de datos.

Para la lectura de datos se ha usado la función *parse* de la biblioteca *xml.etree.ElementTree*, la cual es estándar de *Python*. A continuación, se han creado objetos de una clase *Tweet* (creada por nosotros para la lectura de datos) para los conjuntos *train* y *validation*. A continuación se ha extraído el texto del *tuit*, el identificador y la clase del mismo.

#### 3.2. Preprocesamiento.

El hecho de que los datos vengan de una fuente tan diversa como una red social, requiere un preprocesamiento exhaustivo y específico. Cada usuario expresará sus opiniones de formas muy diversas, por lo que en este apartado intentaremos ser capaces de transformar los datos de forma que los *tuits* que pertenezcan a una misma clase sean lo más similares posible. Describamos una a una las distintas etapas que se han seguido para lograr este objetivo.

En primer lugar, lo que haremos será convertir todo el texto de cada *tuit* a minúscula y sin acentos. Es evidente pensar que esto hará que los textos de los diferentes *tuits* se asemejen más sin cambiar su significado. Además, se eliminarán signos de puntuación, caracteres como & o y aquellas letras que aparezcan repetidas más de dos veces. Por ejemplo, si tenemos la palabra *hoooootel* la transformaremos a *hotel*. A continuación, para los *hashtag*, eliminaremos el carácter # aunque si dejaremos el *hashtag* en cuestión, pues este puede estar relacionado con algún tipo de opinión (p.e, el #felicidad podría estar normalmente asociado a un *tuit* positivo. Así mismo, eliminaremos la cadena *rt* que no aporta nada de información al contenido. Por último, terminaremos esta primera etapa reduciendo espacios múltiples a un solo espacio.

Como segunda etapa de preprocesamiento, haremos un tratamiento específico de los *emojis*. Hemos agrupado los *emojis* según la emoción que expresen. Una vez hecho esto, reemplazaremos el *emoji* por *POSITIVE* en caso de expresar una emoción positiva (p. e, :) ) o por *NEGATIVE* en caso de expresar una opinión negativa (p. e, : ( ). Así mismo, las risas se han reemplazado por *POSITIVE*. Sobre esta idea, hemos aplicado un proceso simple para premiar a los *tuits* que contenga *emojis*: parece lógico pensar que si un *tuit* tiene un *emoji* es más probable que la clasificación de este *tuit* sea a la emoción que expresa dicho *emoji*. Es por ello, que cada uno de ellos será reemplazado por cuatro repeticiones de *POSITIVE* o de *NEGATIVE* según corresponda.

En la tercera etapa, tratamos de hacer un procedimiento similar a la anterior pero, en este caso, con palabras malsonantes. Una palabra malsonante puede ser, por ejemplo, un insulto. Para ello, se ha construido una lista de forma manual de algunas palabras mal sonantes, ya que, en español, no hemos encontrado una lista completa de dichas palabras. Cada palabra malsonante será reemplazada por cuatro repeticiones de *NEGATIVE*, debido a que parece lógico que este *tuit* exprese un sentimiento negativo.

Como cuarta etapa, se ha procedido a convertir todos los números (escritos numéricamente) a texto. Esto permite que si el texto de dos *tuits* distintos fuesen “12” y “doce”, tengan el mismo contenido.

Como etapa opcional, podemos indicar si queremos quedarnos con la raíz de cada palabra. Esto permite que, por ejemplo, conjugaciones del mismo verbo o palabras con la misma raíz sean lo mismo: *gustar*, *gusto*, *gustaría*... Sin embargo, esto puede hacernos perder información al eliminar parte de las palabras.

Por último, si analizamos las palabras más frecuentes de todos los *tuits* mediante *wordclouds* nos encontramos que palabras con apenas significado semántico, son las que más aparecen (el tamaño de la palabra es proporcional a la veces que aparece, es decir, cuánto más grande sea su tamaño, más veces aparecerá). Se muestran en las siguientes imágenes las *wordclouds* para todos los *tuits* de cada clase en los conjuntos de *train* y *validation*.



Figura 1: *Wordcloud* del conjunto de *train* para cada clase.



Figura 2: *Wordcloud* del conjunto de *validation* para cada clase.

Vemos que en casi todas las clases se repiten las mismas palabras aproximadamente para los dos con-

juntos de datos. Dichas palabras se conocen como *stopwords*. Este conjunto de palabras está formado por preposiciones, artículos, conjunciones, etc. Por tanto, en esta última etapa de preprocesamiento trataremos de eliminarlas. Para ver el efecto que esto supone, se vuelven a mostrar las *wordclouds* de ambos conjuntos (*train* y *validation*) para las distintas clases.

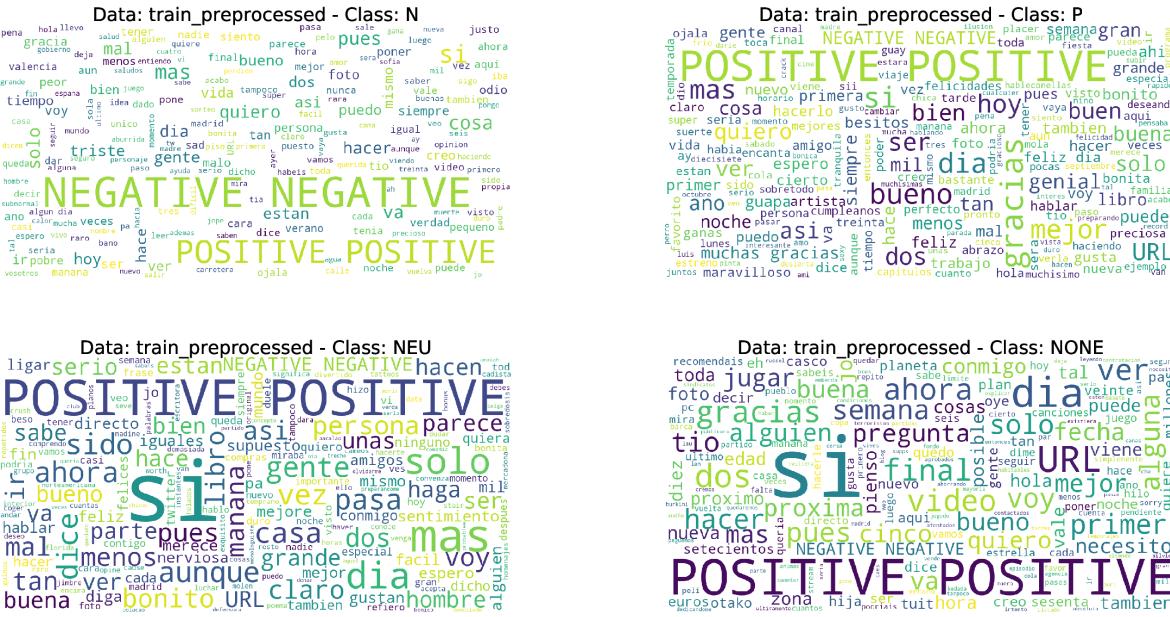


Figura 3: *Wordcloud* del conjunto de *train* preprocesado para cada clase.

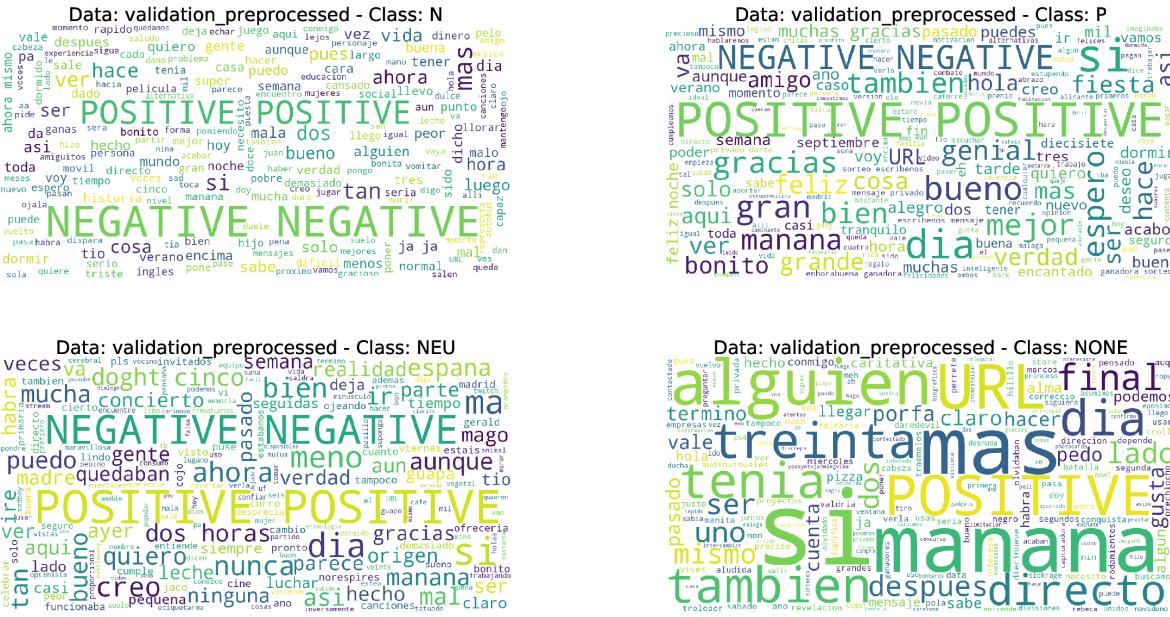


Figura 4: Wordcloud del conjunto de validation preprocesado para cada clase.

Si observamos las *wordclouds* preprocesadas, vemos como en la clase positiva la palabra *POSITIVE* se repite bastante, lo cual puede indicar que nuestras suposiciones a la hora de reemplazar los *emojis* es

buenas. Así mismo, vemos palabras como *gracias*, *genial*, *bien* y *bueno* entran en las positivas y palabras como *triste*, *dinero*, *malo* y *llorar* aparecen en las negativas. Sin embargo, vemos que para la clase *NONE* no hay ninguna palabra que identifique sentimientos positivos o negativos. En la clase *NEU*, se mezclan bastantes palabras sin nada que destacar aparentemente.

### 3.3. Oversampling.

Dado que hay una clara desigualdad entre las clases de los datos (hay 318 elementos de la clase *P*, hay 418 elementos de la clase *N*, hay 139 elementos de la clase *NONE* y 133 elementos de la clase *NEU*) creemos que aplicar técnicas que afronten los problemas de clasificación no balanceada puede aportar buenos resultados. Para ello se ha elegido el método de *oversampling SMOTE*. Dicho método se encarga de crear muestras sintéticas interpolando las muestras de la clase minoritaria con un vecino aleatorio de sus *k* vecinos más cercanos.

Para aplicar *SMOTE* a los datos de los que disponemos ha sido necesario crear una *bolsa de palabras* ya que será la representación en forma de vector que permita al algoritmo interpolar nuevas instancias y, por tanto, nuevos *tuits*. Para la creación de la bolsa de palabras se ha creado un vocabulario que contiene todos los token pertenecientes a todos los *tuits* y a partir de ese vocabulario se ha representado cada *tuit* en forma de vector de 0's y 1's, añadiendo un 1 en el índice correspondiente a la palabra del vocabulario. Una vez obtenida la representación basta con aplicar *SMOTE* y revertir el proceso, es decir, pasar de la *bolsa de palabras* a los *tuits*. A la hora de crear el vocabulario se han obviado las comas, puntos y demás caracteres especiales ya que se generarían *tuits* sintéticos con dichos caracteres en posiciones no comunes y tampoco marcan una diferencia a la hora de clasificar la etiqueta.

**the dog is on the table**



Figura 5: Ejemplo de bolsa de palabras

## 4. Modelos de redes.

En esta sección se describirán las partes comunes a los distintos modelos y modelos de redes que se han probado. Para ello, se mostrará el código *Python* junto a una breve explicación de los razonamientos seguidos para su construcción.

Todas las redes mostradas a continuación tienen los siguientes puntos en común:

- Reciben como entrada el conjunto de entrenamiento junto con sus etiquetadas y el conjunto de test con sus etiquetas. En el proceso de elección del modelo, el conjunto de test será el conjunto *validation*, del cual disponemos de las etiquetas y pueden ser pasadas al modelo. Estas etiquetas son las usadas para validar el modelo internamente durante el proceso de aprendizaje y, de esta manera, poder pintar la gráfica comentada anteriormente. En la fase de predicción, el conjunto de test que recibe la red es el conjunto de test disponible, del cual no disponemos las etiquetadas y por lo tanto no se generará la gráfica para dicha ejecución.
- Un argumento *verbose*, para indicar si se pueden informar del proceso de aprendizaje o no.

- Los argumentos necesarios para la propia red, como puede ser el número de épocas o el valor de *learning rate* deseado.
- Lo primero que haces es llamar a la función *own\_set\_seed()* para “setear” las semillas de todos las librerías usadas para poder reproducir los resultados.

A parte de lo comentado anteriormente, hay una serie de funciones y variables declaradas y usadas por las redes:

- *\_\_CLASSES\_\_*: Variable usada para almacenar las clases disponibles.
- *\_\_CLASSES\_TO\_NUM\_DIC\_\_*: Diccionario usado para mapear cada clase a un valor numérico.
- *\_\_NUM\_TO\_CLASSES\_DIC\_\_*: Diccionario usado para mapear un valor numérico a cada clase.
- *tokenize*: función usada para tokenizar un *tuit*.
- *fit\_transform\_vocabulary*: Función para crear el vocabulario a partir del *corpus* de entrenamiento.
- *fit\_transform\_vocabulary\_pretrain\_embeddings*: Función para crear el vocabulario a partir del *corpus* de entrenamiento y un conjunto preentrenado de *embeddings*.
- *read\_embeddings*: Función para leer un fichero de *embeddings*.

#### 4.1. *tfidf\_rnn*.

La primera red que hemos probado es una red básica que aplica el algoritmo *tfidf*. Esta red no usa *embeddings*, sino el concepto *frecuencia de término - frecuencia inversa de documento*. Esa idea es una medida numérica que expresa cómo de relevante es una palabra para un documento en una colección. El proceso de montar la información necesaria para *tfidf* lo podemos ver en las líneas 15-29. La arquitectura usada en esta red es una arquitectura estándar:

- Capa de entrada *LSTM* con 64 neuronas.
- Capa oculta *Dense* con 32 neuronas y *tanh* como función de activación.
- Capa oculta *Flatten*, para “aplanar” la salida.
- Capa de salida *Dense* con *numero de clases* neuronas y *softmax* como función de activación, para poder obtener una distribución de probabilidad que nos permita decidir a qué clase pertenece la entrada.

Una vez tenemos la información necesaria para *tfidf* y la red definida, convertimos la entrada a sus “valores” *tfidf* correspondientes (líneas 45-55) y entrenamos la red (líneas 58-63), para predecir la salida en la línea 65.

```

1 from keras.layers.core import Dense
2 from keras.layers.core import Flatten
3 from keras.layers.recurrent import LSTM
4 from keras.models import Sequential
5 from keras.preprocessing import sequence
6 from sklearn.feature_extraction.text import TfidfVectorizer
7
8 from src.util.utilities import *
9
10
11 def tfidf_rnn(train_xs, train_ys, test_xs, test_ys=None, verbose=1):
12     own_set_seed()
13

```

```

14 # Representation
15 tfidf_parser = TfidfVectorizer(tokenizer=tokenize, lowercase=False, analyzer='word')
16
17 train_sparse_matrix_features_tfidf = tfidf_parser.fit_transform(train_xs)
18 test_sparse_matrix_features_tfidf = tfidf_parser.transform(test_xs)
19
20 train_features_tfidf = []
21 own_train_features_tfidf_append = train_features_tfidf.append
22 lengths_tweets = []
23 own_lengths_tweets_append = lengths_tweets.append
24
25 for tweet in train_sparse_matrix_features_tfidf:
26     own_train_features_tfidf_append(tweet.data)
27     own_lengths_tweets_append(len(tweet.data))
28
29 test_features_tfidf = [tweet.data for tweet in test_sparse_matrix_features_tfidf]
# Average length
30 max_len_input = int(np.max(lengths_tweets, 0))
31 # lstm_output_dim = int(2**np.log2(max_len_input))
32
33 # NN model
34 nn_model = Sequential()
35 nn_model.add(LSTM(64, input_shape=(max_len_input, 1), return_sequences=True))
36 nn_model.add(Dense(32, activation='tanh'))
37 nn_model.add(Flatten())
38 nn_model.add(Dense(len(src.util.global_vars.__CLASSES__), activation='softmax'))
39 nn_model.compile(optimizer="adam", loss="sparse_categorical_crossentropy")
40
41 if verbose == 1:
42     print(nn_model.summary())
43
44 train_features_tfidf_pad = sequence.pad_sequences(train_features_tfidf, maxlen=
45 max_len_input,
46                                         padding="post", truncating="post",
47                                         dtype=train_sparse_matrix_features_tfidf.
48                                         dtype)
48 train_features_tfidf_pad = np.expand_dims(train_features_tfidf_pad, axis=-1)
49
50 np_labels_train = np.array(train_ys)
51
52 test_features_tfidf_pad = sequence.pad_sequences(test_features_tfidf, maxlen=max_len_input,
53 ,
54                                         padding="post", truncating="post",
55                                         dtype=test_sparse_matrix_features_tfidf.
56                                         dtype)
55 test_features_tfidf_pad = np.expand_dims(test_features_tfidf_pad, axis=-1)
56
57 history = None
58 if test_ys is None:
59     nn_model.fit(train_features_tfidf_pad, np_labels_train, batch_size=32, epochs=10,
60     verbose=verbose)
61 else:
62     history = nn_model.fit(train_features_tfidf_pad, np_labels_train,
63     validation_data=(test_features_tfidf_pad, test_ys), batch_size
64     =32, epochs=10,
65     verbose=verbose)
66
65 y_labels = nn_model.predict_classes(test_features_tfidf_pad, batch_size=32, verbose=
66 verbose)
67
67 return y_labels, history

```

## 4.2. calculated\_embeddings\_rnn.

Esta es la primera red en la que vamos a usar el concepto de *Embedding*. Un *embedding* es la representación de una palabra o frase del vocabulario en forma de un vector de número reales. Conceptualmente, nos permite representar las palabras en un espacio  $N$ -dimensional sobre el cual podemos aplicar operaciones de vectores para determinar las relaciones entre las palabras de nuestro vocabulario. En este caso, vamos

a calcular los *embeddings* asociados a nuestro conjunto de entrenamiento en la propia red (líneas 14-24). Una vez tenemos los *embeddings* montamos una red con la siguiente arquitectura:

- Capa *Embedding* como capa de entrada.
  - Capa oculta *LSTM* con 64 neuronas.
  - Capa oculta *Dense* con 32 neuronas y *tanh* como función de activación.
  - Capa oculta *Flatten*, para “aplanar” la salida.
  - Capa de salida *Dense* con *numero de clases* neuronas y *softmax* como función de activación, para poder obtener una distribución de probabilidad que nos permita decidir a que clase pertenece la entrada.

Una vez tenemos los *embeddings* y la red definida, mapeamos la entrada a sus *embeddings* correspondientes (líneas 37-42) y entrenamos la red (líneas 44-49), para predecir la salida en la línea 51.

```

1 from keras.layers.core import Dense
2 from keras.layers.core import Flatten
3 from keras.layers.embeddings import Embedding
4 from keras.layers.recurrent import LSTM
5 from keras.models import Sequential
6 from keras.preprocessing import sequence
7
8 from src.util.utilities import *
9
10
11 def calculated_embeddings_rnn(train_xs, train_ys, test_xs, test_ys=None, verbose=1):
12     own_set_seed()
13
14     # Build vocabulary and corpus indexes
15     vocabulary_train, corpus_train_index = fit_transform_vocabulary(train_xs)
16
17     max_len_input = int(np.max([len(tweet_train) for tweet_train in corpus_train_index], 0))
18
19     corpus_test_index = []
20     own_corpus_test_index_append = corpus_test_index.append
21     for tweet_test in test_xs:
22         tokens_test = tokenize(tweet_test)
23         own_corpus_test_index_append([vocabulary_train.get(token_test, 1)
24                                       for token_test in tokens_test])
25
26     nn_model = Sequential()
27     nn_model.add(Embedding(len(vocabulary_train) + 2, 100, input_length=max_len_input,
28                           trainable=False))
29     nn_model.add(LSTM(64, return_sequences=True))
30     nn_model.add(Dense(32, activation='tanh'))
31     nn_model.add(Flatten())
32     nn_model.add(Dense(len(src.util.global_vars.__CLASSES__), activation='softmax'))
33     nn_model.compile(optimizer="adam", loss="sparse_categorical_crossentropy")
34
35     if verbose == 1:
36         print(nn_model.summary())
37
38     train_features_pad = sequence.pad_sequences(corpus_train_index, maxlen=max_len_input,
39                                                 padding="post",
40                                                 truncating="post", dtype=type(
41                                                 corpus_train_index[0][0]))
42     np_labels_train = np.array(train_ys)
43
44     test_features_pad = sequence.pad_sequences(corpus_test_index, maxlen=max_len_input,
45                                                 padding="post",
46                                                 truncating="post", dtype=type(corpus_test_index
47                                                 [0][0]))
48
49     history = None
50     if test_ys is None:
51
52         # Train the model
53         nn_model.fit(x=train_xs, y=train_ys, epochs=10, batch_size=32, validation_split=0.2)
54
55         # Evaluate the model
56         test_loss, test_accuracy = nn_model.evaluate(x=test_xs, y=test_ys)
57
58         print("Test Loss: ", test_loss)
59         print("Test Accuracy: ", test_accuracy)
60
61     else:
62
63         # Train the model
64         nn_model.fit(x=train_xs, y=train_ys, epochs=10, batch_size=32, validation_split=0.2)
65
66         # Evaluate the model
67         test_loss, test_accuracy = nn_model.evaluate(x=test_xs, y=test_ys)
68
69         print("Test Loss: ", test_loss)
70         print("Test Accuracy: ", test_accuracy)
71
72     return nn_model, history

```

```

46     nn_model.fit(train_features_pad, np_labels_train, batch_size=32, epochs=25, verbose=
47     verbose)
48     else:
49         history = nn_model.fit(train_features_pad, np_labels_train, validation_data=(test_
50         features_pad, test_ys),
51         batch_size=32, epochs=25, verbose=verbose)
52
53     y_labels = nn_model.predict_classes(test_features_pad, batch_size=32, verbose=verbose)
54
55     return y_labels, history

```

### 4.3. pretrain\_embeddings\_rnn.

Una vez hemos introducido el concepto de *embedding*, podemos ir un paso más allá. Construir un conjunto de *embeddings* usando sólo las palabras disponibles en nuestro conjunto de entrenamiento puede llevarnos a generar unos *embeddings* que no sean de calidad y que no representen realmente lo que queremos que representen. Por lo tanto, surgen técnicas e ideas de generar un conjunto de *embeddings* a partir de un conjunto de entrenamiento lo suficientemente grande y descriptivo del lenguaje empleado en el problema a resolver. Esta red hace uso de un fichero de *embeddings* ya existente, el cual se especifica en el argumento *embeddings\_path*. A parte de los *embeddings* definimos dos nuevas entradas para poder lidiar con palabras que no están en el conjunto de *embeddings* y otra para llenar la entrada de la red si esta no es lo suficientemente grande como para cubrir todas las neuronas de entrada. Con estos *embeddings* montamos nuestro vocabulario de entrenamiento (líneas 17-22). A continuación montamos una red con la siguiente arquitectura:

- Capa *Embedding* como capa de entrada.
- Capa oculta *LSTM* con 64 neuronas.
- Capa oculta *Dense* con 32 neuronas y *tanh* como función de activación.
- Capa oculta *Flatten*, para “aplanar” la salida.
- Capa de salida *Dense* con *numero de clases* neuronas y *softmax* como función de activación, para poder obtener una distribución de probabilidad que nos permita decidir a que clase pertenece la entrada.

Finalmente, mapeamos la entrada a sus *embeddings* correspondientes (líneas 51-55) y entrenamos la red (líneas 58-62), para predecir la salida en la línea 64.

```

1 from keras.layers.core import Dense
2 from keras.layers.core import Flatten
3 from keras.layers.embeddings import Embedding
4 from keras.layers.recurrent import LSTM
5 from keras.models import Sequential
6 from keras.optimizers import Adam
7 from keras.preprocessing import sequence
8
9 from src.util.utilities import *
10
11
12 def pretrain_embeddings_rnn(embeddings_path, train_xs, train_ys, test_xs, test_ys=None, epochs
13     =25, learning_rate=0.001,
14     verbose=1):
15     own_set_seed()
16
17     # Offset = 2; Padding and OOV.
18     word_embeddings, word_emb_indexes = read_embeddings(embeddings_path, 2)
19     word_embeddings[0] = 2 * 0.1 * np.random.rand(len(word_embeddings[2])) - 1
20     word_embeddings[1] = 2 * 0.1 * np.random.rand(len(word_embeddings[2])) - 1
21
22     # Build vocabulary and corpus indexes

```

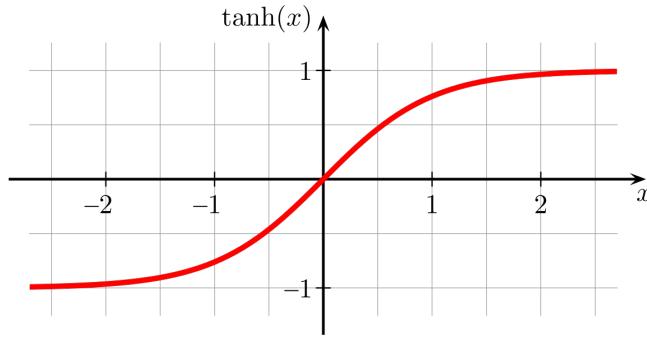
```

22 vocabulary_train, corpus_train_index = fit_transform_vocabulary_pretrain_embeddings(
23 train_xs, word_emb_indexes)
24 max_len_input = int(np.average([len(tweet_train) for tweet_train in corpus_train_index], 0))
25
26 corpus_test_index = []
27 own_corpus_test_index_append = corpus_test_index.append
28 own_lowercase = str.lower
29 for tweet_test in test_xs:
30     tokens_test = tokenize(own_lowercase(tweet_test))
31     doc_test_index = []
32     for token_test in tokens_test:
33         if RE_TOKEN_USER.fullmatch(token_test) is not None:
34             token_test = "@user"
35         doc_test_index.append(word_emb_indexes.get(token_test, 1))
36     own_corpus_test_index_append(doc_test_index)
37
38 nn_model = Sequential()
39 nn_model.add(Embedding(len(word_embeddings), len(word_embeddings[0]), weights=[np_array(
40 word_embeddings)], input_length=max_len_input, trainable=False))
41 nn_model.add(LSTM(64, return_sequences=True))
42 nn_model.add(Dense(32, activation='tanh'))
43 nn_model.add(Flatten())
44 nn_model.add(Dense(len(src.util.global_vars.__CLASSES__), activation='softmax'))
45 adam_optimizer = Adam(lr=learning_rate)
46 nn_model.compile(optimizer=adam_optimizer, loss="sparse_categorical_crossentropy")
47
48 if verbose == 1:
49     print(nn_model.summary())
50
51 train_features_pad = sequence.pad_sequences(corpus_train_index, maxlen=max_len_input,
52 padding="post",
53                                         truncating="post", dtype=type(
54 corpus_train_index[0][0]))
55 np_labels_train = np.array(train_ys)
56 test_features_pad = sequence.pad_sequences(corpus_test_index, maxlen=max_len_input,
57 padding="post",
58                                         truncating="post", dtype=type(corpus_test_index
59 [0][0]))
60
61 history = None
62 if test_ys is None:
63     nn_model.fit(train_features_pad, np_labels_train, batch_size=32, epochs=epochs,
64 verbose=verbose)
65 else:
66     history = nn_model.fit(train_features_pad, np_labels_train, validation_data=(test_
67 features_pad, test_ys),
68                             batch_size=32, epochs=epochs, verbose=verbose)
69
70 y_labels = nn_model.predict_classes(test_features_pad, batch_size=32, verbose=verbose)
71 return y_labels, history

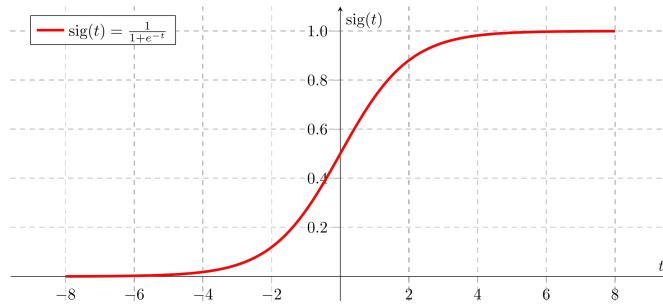
```

#### 4.4. sigmoid\_pretrain\_embeddings\_rnn.

Una vez tenemos claro que lo mejor es usar un conjunto de *embeddings* obtenido sobre un conjunto de entrenamiento grande, veamos que más podemos hacer en una red. Una de las cosas que hemos hecho hasta ahora es usar la función de tangente hiperbólica como la función de activación de la primera capa *Dense* de nuestras redes. Dicha función tiene la siguiente forma:



Sin embargo, otra de las funciones clásicas para Redes Neuronales es la función *sigmoide*, la cual tiene la siguiente forma:



Como podemos ver, la función *tanh* es una función que toma valores de -1 a 1 mientras que la función *sigmoide* toma valores de 0 a 1. En esta red hemos querido probar la función *sigmoide* (línea 40) para ver cómo influye la función de activación elegida.

```
1 from keras.layers.core import Dense
2 from keras.layers.core import Flatten
3 from keras.layers.embeddings import Embedding
4 from keras.layers.recurrent import LSTM
5 from keras.models import Sequential
6 from keras.preprocessing import sequence
7
8 from src.util.utilities import *
9
10
11 def sigmoid_pretrain_embeddings_rnn(embeddings_path, train_xs, train_ys, test_xs, test_ys=None,
12 , verbose=1):
13     own_set_seed()
14
15     # Offset = 2; Padding and OOV.
16     word_embeddings, word_emb_indexes = read_embeddings(embeddings_path, 2)
17     word_embeddings[0] = 2 * 0.1 * np.random.rand(len(word_embeddings[2])) - 1
18     word_embeddings[1] = 2 * 0.1 * np.random.rand(len(word_embeddings[2])) - 1
19
20     # Build vocabulary and corpus indexes
21     vocabulary_train, corpus_train_index = fit_transform_vocabulary_pretrain_embeddings(
22     train_xs, word_emb_indexes)
23
24     max_len_input = int(np.average([len(tweet_train) for tweet_train in corpus_train_index]),
25 0))
26
27     corpus_test_index = []
28     own_corpus_test_index_append = corpus_test_index.append
29     own_lowercase = str.lower
30     for tweet_test in test_xs:
31         tokens_test = tokenize(own_lowercase(tweet_test))
32         doc_test_index = []
33         for token_test in tokens_test:
```

```

31         if RE_TOKEN_USER.fullmatch(token_test) is not None:
32             token_test = "@user"
33             doc_test_index.append(word_emb_indexes.get(token_test, 1))
34             own_corpus_test_index.append(doc_test_index)
35
36 nn_model = Sequential()
37 nn_model.add(Embedding(len(word_embeddings), len(word_embeddings[0])), weights=[np_array(
38     word_embeddings)], input_length=max_len_input, trainable=False))
39 nn_model.add(LSTM(64, return_sequences=True))
40 nn_model.add(Dense(32, activation='sigmoid'))
41 nn_model.add(Flatten())
42 nn_model.add(Dense(len(src.util.global_vars.__CLASSES__), activation='softmax'))
43 nn_model.compile(optimizer="adam", loss="sparse_categorical_crossentropy")
44
45 if verbose == 1:
46     print(nn_model.summary())
47
48 train_features_pad = sequence.pad_sequences(corpus_train_index, maxlen=max_len_input,
49 padding="post",
50                                         truncating="post", dtype=type(
51     corpus_train_index[0][0]))
52 np_labels_train = np.array(train_ys)
53
54 test_features_pad = sequence.pad_sequences(corpus_test_index, maxlen=max_len_input,
55 padding="post",
56                                         truncating="post", dtype=type(corpus_test_index
57 [0][0]))
58 history = None
59 if test_ys is None:
60     nn_model.fit(train_features_pad, np_labels_train, batch_size=32, epochs=25, verbose=
61     verbose)
62 else:
63     history = nn_model.fit(train_features_pad, np_labels_train, validation_data=(test_
64 features_pad, test_ys),
65     batch_size=32, epochs=25, verbose=verbose)
66
67 y_labels = nn_model.predict_classes(test_features_pad, batch_size=32, verbose=verbose)
68
69 return y_labels, history

```

Tras los experimentos realizados, hemos visto que *tanh* ofrece mejores resultados. Buscando información hemos aprendido que las *LSTMs* tienen un vector interno para representar que pesos pueden crecer o decrecer en el proceso de aprendizaje. La función *sigmoide* genera siempre valores positivos y, por lo tanto, los valores de los pesos sólo pueden crecer. Sin embargo, *tanh* genera valores positivos y negativos, permitiendo así que los pesos crezcan y decrezcan, lo cual da más flexibilidad en el proceso de aprendizaje ya que da más libertad para que los pesos se ajusten.

#### 4.5. stacked\_lstm\_rnn.

Tal y como hemos estado viendo, lo mejor es trabajar con la función *tanh* y *embeddings* pre-entrenados. La siguiente prueba que hemos hecho es aumentar el número de capas de la red, para tener así una red más compleja. La idea detrás del uso de redes neuronales en *Deep Learning* es precisamente, tener redes con varias capas ocultas que permitan un aprendizaje más detallado del problema. Por ello, la arquitectura usada para esta red es la siguiente:

- Capa *Embedding* como capa de entrada.
- Cuatro capas ocultas *LSTM* con 64 neuronas cada una.
- Capa oculta *Dense* con 32 neuronas y *tanh* como función de activación.
- Capa oculta *Flatten*, para “aplanar” la salida.

- Capa de salida *Dense* con *numero de clases* neuronas y *softmax* como función de activación, para poder obtener una distribución de probabilidad que nos permita decidir a qué clase pertenece la entrada.

```

1 from keras.layers.core import Dense
2 from keras.layers.embeddings import Embedding
3 from keras.layers.recurrent import LSTM
4 from keras.models import Sequential
5 from keras.preprocessing import sequence
6
7 from src.util.utilities import *
8
9
10 def stacked_lstm_rnn(embeddings_path, train_xs, train_ys, test_xs, test_ys=None, epochs=25,
11   verbose=1):
12   own_set_seed()
13
14   # Offset = 2; Padding and OOV.
15   word_embeddings, word_emb_indexes = read_embeddings(embeddings_path, 2)
16   word_embeddings[0] = 2 * 0.1 * np.random.rand(len(word_embeddings[2])) - 1
17   word_embeddings[1] = 2 * 0.1 * np.random.rand(len(word_embeddings[2])) - 1
18
19   # Build vocabulary and corpus indexes
20   vocabulary_train, corpus_train_index = fit_transform_vocabulary_pretrain_embeddings(
21     train_xs, word_emb_indexes)
22
23   max_len_input = int(np.average([len(tweet_train) for tweet_train in corpus_train_index], 0))
24
25   corpus_test_index = []
26   own_corpus_test_index_append = corpus_test_index.append
27   own_lowercase = str.lower
28   for tweet_test in test_xs:
29     tokens_test = tokenize(own_lowercase(tweet_test))
30     doc_test_index = []
31     for token_test in tokens_test:
32       if RE_TOKEN_USER.fullmatch(token_test) is not None:
33         token_test = "@user"
34       doc_test_index.append(word_emb_indexes.get(token_test, 1))
35     own_corpus_test_index_append(doc_test_index)
36
37   nn_model = Sequential()
38   nn_model.add(Embedding(len(word_embeddings), len(word_embeddings[0]), weights=[np_array(
39     word_embeddings)], input_length=max_len_input, trainable=False))
40   nn_model.add(LSTM(64, return_sequences=True))
41   nn_model.add(LSTM(64, return_sequences=True))
42   nn_model.add(LSTM(64, return_sequences=True))
43   nn_model.add(LSTM(64))
44   nn_model.add(Dense(len(src.util.global_vars.__CLASSES__), activation='softmax'))
45   nn_model.compile(optimizer="adam", loss="sparse_categorical_crossentropy")
46
47   if verbose == 1:
48     print(nn_model.summary())
49
50   train_features_pad = sequence.pad_sequences(corpus_train_index, maxlen=max_len_input,
51   padding="post", truncating="post", dtype=type(
52     corpus_train_index[0][0]))
53   np_labels_train = np.array(train_ys)
54   test_features_pad = sequence.pad_sequences(corpus_test_index, maxlen=max_len_input,
55   padding="post", truncating="post", dtype=type(corpus_test_index
56     [0][0]))
57
58   history = None
59   if test_ys is None:
60     nn_model.fit(train_features_pad, np_labels_train, batch_size=32, epochs=epochs,
61     verbose=verbose)
62   else:

```

```

58     history = nn_model.fit(train_features_pad, np_labels_train, validation_data=(  

59         test_features_pad, test_ys),  

60             batch_size=32, epochs=epochs, verbose=verbose)  

61     y_labels = nn_model.predict_classes(test_features_pad, batch_size=32, verbose=verbose)  

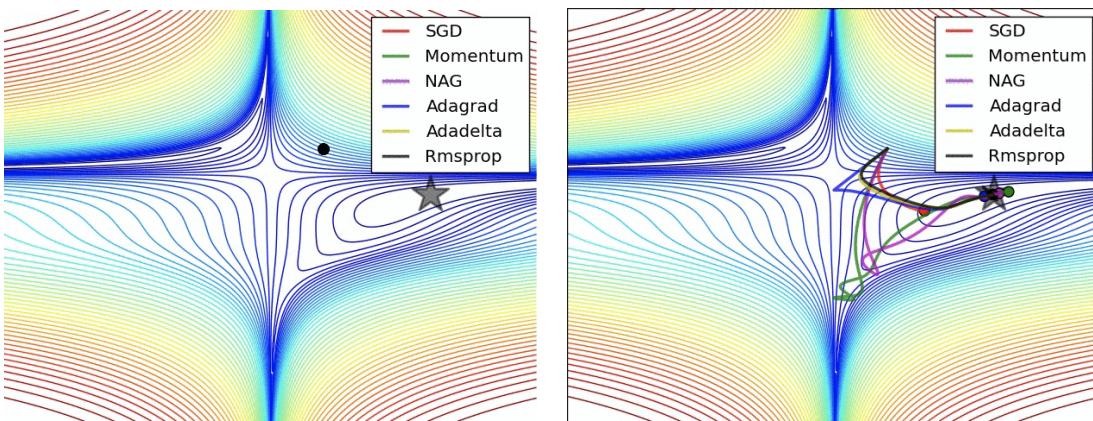
62     return y_labels, history

```

En nuestro caso particular, aumentar el número de capas ocultas no ha mejorado los resultados obtenidos.

#### 4.6. adadelta\_rnn.

Otra de las cosas que podemos modificar de una red es el optimizador usada por la misma. En todos los ejemplos mostrados hasta ahora el optimizador usado ha sido *adam* (*Adam: A Method for Stochastic Optimization*). En <http://cs231n.github.io/neural-networks-3/#ada> podemos ver un *gif* que nos muestra cómo varía el comportamiento de una red en base al optimizador usado. Si nos quedamos con el momento inicial y el momento final del *gif* tenemos lo siguiente:



A pesar de que todos parten del mismo punto (figura de la izquierda), el camino recorrido para llegar a la solución, representada como una estrella, es totalmente distinto. Si observamos el *gif*, también podremos apreciar la diferencia de velocidad para encontrar dicha solución. Si nos fijamos, el que mejores resultados da es *adadelta*, y por lo tanto lo hemos probado. La red usada es la misma que la explicada en la sección *pretrain\_embeddings\_rnn* (4.3) pero usando *adadelta* (línea 43) como optimizador.

```

1 from keras.layers.core import Dense  

2 from keras.layers.core import Flatten  

3 from keras.layers.embeddings import Embedding  

4 from keras.layers.recurrent import LSTM  

5 from keras.models import Sequential  

6 from keras.preprocessing import sequence  

7  

8 from src.util.utilities import *  

9  

10  

11 def adadelta_rnn(embeddings_path, train_xs, train_ys, test_xs, test_ys=None, epochs=25,  

12     verbose=1):  

13     own_set_seed()  

14  

15     # Offset = 2; Padding and OOV.  

16     word_embeddings, word_emb_indexes = read_embeddings(embeddings_path, 2)  

17     word_embeddings[0] = 2 * 0.1 * np.random.rand(len(word_embeddings[2])) - 1  

18     word_embeddings[1] = 2 * 0.1 * np.random.rand(len(word_embeddings[2])) - 1  

19  

20     # Build vocabulary and corpus indexes  

21     vocabulary_train, corpus_train_index = fit_transform_vocabulary_pretrain_embeddings(  

        train_xs, word_emb_indexes)

```

```

22 max_len_input = int(np.average([len(tweet_train) for tweet_train in corpus_train_index], 0))
23
24 corpus_test_index = []
25 own_corpus_test_index.append = corpus_test_index.append
26 own_lowercase = str.lower
27 for tweet_test in test_xs:
28     tokens_test = tokenize(own_lowercase(tweet_test))
29     doc_test_index = []
30     for token_test in tokens_test:
31         if RE_TOKEN_USER.fullmatch(token_test) is not None:
32             token_test = "@user"
33             doc_test_index.append(word_emb_indexes.get(token_test, 1))
34     own_corpus_test_index.append(doc_test_index)
35
36 nn_model = Sequential()
37 nn_model.add(Embedding(len(word_embeddings), len(word_embeddings[0]), weights=[np_array(word_embeddings)], input_length=max_len_input, trainable=False))
38 nn_model.add(LSTM(64, return_sequences=True))
39 nn_model.add(Dense(32, activation='tanh'))
40 nn_model.add(Flatten())
41 nn_model.add(Dense(len(src.util.global_vars.__CLASSES__), activation='softmax'))
42 nn_model.compile(optimizer="adadelta", loss="sparse_categorical_crossentropy")
43
44 if verbose == 1:
45     print(nn_model.summary())
46
47 train_features_pad = sequence.pad_sequences(corpus_train_index, maxlen=max_len_input, padding="post",
48                                              truncating="post", dtype=type(
49          corpus_train_index[0][0]))
50 np_labels_train = np.array(train_ys)
51 test_features_pad = sequence.pad_sequences(corpus_test_index, maxlen=max_len_input, padding="post",
52                                              truncating="post", dtype=type(corpus_test_index[0][0]))
53
54 history = None
55 if test_ys is None:
56     nn_model.fit(train_features_pad, np_labels_train, batch_size=32, epochs=epochs, verbose=verbose)
57 else:
58     history = nn_model.fit(train_features_pad, np_labels_train, validation_data=(test_features_pad, test_ys),
59                            batch_size=32, epochs=epochs, verbose=verbose)
60
61 y_labels = nn_model.predict_classes(test_features_pad, batch_size=32, verbose=verbose)
62 return y_labels, history

```

## 4.7. calculated\_embeddings\_LSTM\_CONV.

Hasta ahora, nos hemos centrado en capas *LSTM* pero queremos probar también cómo funcionan las capas convolucionales en procesamiento de texto. Para ello, hemos diseñado una red que combina ambas neuronas. La arquitectura usada para esta red es la siguiente:

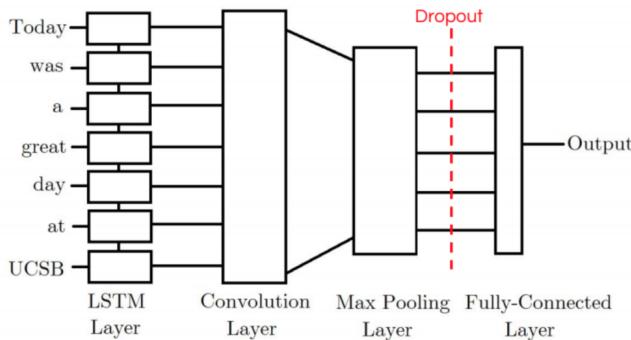
- Capa *Embedding* como capa de entrada.
- Capa *LSTM*, con 64 neuronas y con un *Dropout* al 50 % de las neuronas.
- Capa oculta de *pooling*, mediante la función *Maxpooling1D*.
- Capa oculta de convolución con *Conv1D*, con 128 neuronas, y con un *Dropout* al 50 % de las neuronas.
- Capa oculta de *pooling* con la función *AveragePooling1D()*.

- Capa oculta *Dense* con 128 neuronas y *relu* como función de activación.
- Capa oculta *Flatten*, para “aplanar” la salida.
- Capa de salida *Dense* con *numero de clases* neuronas y *softmax* como función de activación, para poder obtener una distribución de probabilidad que nos permita decidir a que clase pertenece la entrada.

La idea de usar un modelo que combine *LSTM* y *CNN* es que *LSTM* permite almacenar información no solo del token actual, sino también de cualquier token anterior. La salida de la capa *LSTM* se introduce en una capa de convolucional para extraer las características locales.

Más particularmente, si hablamos del uso de *CNN* y dónde puede ayudar, en nuestro caso particular, podría capturar una frase negativa como “no me gusta”, independientemente de dónde suceda en el *tuit*, como por ejemplo:

- No me gusta ver ese tipo de películas.
- Esa es la única cosa que realmente no me gusta.
- Vi la película, y no me gusta cómo terminó.



```

1 from keras.layers import MaxPooling1D, Conv1D, AveragePooling1D
2 from keras.layers.core import Dense, Dropout
3 from keras.layers.core import Flatten
4 from keras.layers.embeddings import Embedding
5 from keras.layers.recurrent import LSTM
6 from keras.models import Sequential
7 from keras.preprocessing import sequence
8
9 from src.util.utilities import *
10
11
12 def calculated_embeddings_LSTM_CONV(train_xs, train_ys, test_xs, test_ys=None, epochs=25,
13     verbose=1):
14     own_set_seed()
15
16     # Build vocabulary and corpus indexes
17     vocabulary_train, corpus_train_index = fit_transform_vocabulary(train_xs)
18
19     max_len_input = int(np.max([len(tweet_train) for tweet_train in corpus_train_index], 0))
20
21     corpus_test_index = []
22     own_corpus_test_index_append = corpus_test_index.append
23     for tweet_test in test_xs:
24         tokens_test = tokenize(tweet_test)
25         own_corpus_test_index_append([vocabulary_train.get(token_test, 1)
26                                       for token_test in tokens_test])

```

```

26
27 nn_model = Sequential()
28 nn_model.add(Embedding(len(vocabulary_train) + 2, 100, input_length=max_len_input,
29 trainable=False))
30 nn_model.add(LSTM(64, return_sequences=True))
31 nn_model.add(Dropout(0.5))
32 nn_model.add(MaxPooling1D())
33
34 nn_model.add(Conv1D(128, 5, activation='relu', padding='same'))
35 nn_model.add(Dropout(0.5))
36 nn_model.add(AveragePooling1D())
37
38 nn_model.add(Dense(128, activation='relu'))
39 nn_model.add(Dropout(0.5))
40 nn_model.add(AveragePooling1D())
41
42 nn_model.add(Flatten())
43 nn_model.add(Dense(len(src.util.global_vars.__CLASSES__), activation='softmax'))
44
45 nn_model.compile(loss='sparse_categorical_crossentropy', optimizer='adam')
46
47 if verbose == 1:
48     print(nn_model.summary())
49
50 train_features_pad = sequence.pad_sequences(corpus_train_index, maxlen=max_len_input,
51 padding="post",
52                                         truncating="post", dtype=type(
53 corpus_train_index[0][0]))
54 np_labels_train = np.array(train_ys)
55
56 test_features_pad = sequence.pad_sequences(corpus_test_index, maxlen=max_len_input,
57 padding="post",
58                                         truncating="post", dtype=type(corpus_test_index
59 [0][0]))
60
61 history = None
62 if test_ys is None:
63     nn_model.fit(train_features_pad, np_labels_train, batch_size=32, epochs=epochs,
64 verbose=verbose)
65 else:
66     history = nn_model.fit(train_features_pad, np_labels_train, validation_data=(
67 test_features_pad, test_ys),
68                             batch_size=32, epochs=epochs, verbose=verbose)
69
70 y_labels = nn_model.predict_classes(test_features_pad, batch_size=32, verbose=verbose)
71
72 return y_labels, history

```

## 4.8. pretrain\_embeddings\_LSTM\_CONV.

Al igual que hemos propuesto usar *embeddings* para redes *LSTM*, hemos pensado que obtendríamos una mejora si usamos *embeddings* pre-entrenados con esa nueva arquitectura de red.

```

1 from keras.layers import MaxPooling1D, Conv1D, AveragePooling1D
2 from keras.layers.core import Dense, Dropout
3 from keras.layers.core import Flatten
4 from keras.layers.embeddings import Embedding
5 from keras.layers.recurrent import LSTM
6 from keras.models import Sequential
7 from keras.preprocessing import sequence
8
9 from src.util.utilities import *
10
11
12 def pretrain_embeddings_LSTM_CONV(embeddings_path, train_xs, train_ys, test_xs, test_ys=None,
13 epochs=25, verbose=1):
14     own_set_seed()
15
16     # Offset = 2; Padding and OOV.
17     word_embeddings, word_emb_indexes = read_embeddings(embeddings_path, 2)

```

```

17 word_embeddings[0] = 2 * 0.1 * np.random.rand(len(word_embeddings[2])) - 1
18 word_embeddings[1] = 2 * 0.1 * np.random.rand(len(word_embeddings[2])) - 1
19
20 # Build vocabulary and corpus indexes
21 vocabulary_train, corpus_train_index = fit_transform_vocabulary_pretrain_embeddings(
22 train_xs, word_emb_indexes)
23 max_len_input = int(np.average([len(tweet_train) for tweet_train in corpus_train_index], 0))
24
25 corpus_test_index = []
26 own_corpus_test_index_append = corpus_test_index.append
27 own_lowercase = str.lower
28 for tweet_test in test_xs:
29     tokens_test = tokenize(own_lowercase(tweet_test))
30     doc_test_index = []
31     for token_test in tokens_test:
32         if RE_TOKEN_USER.fullmatch(token_test) is not None:
33             token_test = "@user"
34         doc_test_index.append(word_emb_indexes.get(token_test, 1))
35     own_corpus_test_index_append(doc_test_index)
36
37 nn_model = Sequential()
38 nn_model.add(Embedding(len(word_embeddings), len(word_embeddings[0]), weights=[np_array(
39 word_embeddings)], input_length=max_len_input, trainable=False))
40
41 nn_model.add(LSTM(64, return_sequences=True))
42 nn_model.add(Dropout(0.5))
43 nn_model.add(MaxPooling1D())
44
45 nn_model.add(Conv1D(128, 5, activation='relu', padding='same'))
46 nn_model.add(Dropout(0.5))
47 nn_model.add(AveragePooling1D())
48
49 nn_model.add(Dense(128, activation='relu'))
50 nn_model.add(Dropout(0.5))
51 nn_model.add(AveragePooling1D())
52
53 nn_model.add(Flatten())
54 nn_model.add(Dense(len(src.util.global_vars.__CLASSES__), activation='softmax'))
55
56 nn_model.compile(loss='sparse_categorical_crossentropy', optimizer='adam')
57
58 if verbose == 1:
59     print(nn_model.summary())
60
61 train_features_pad = sequence.pad_sequences(corpus_train_index, maxlen=max_len_input,
62 padding="post",
63                                         truncating="post", dtype=type(
64 corpus_train_index[0][0]))
65 np_labels_train = np.array(train_ys)
66 test_features_pad = sequence.pad_sequences(corpus_test_index, maxlen=max_len_input,
67 padding="post",
68                                         truncating="post", dtype=type(corpus_test_index
69 [0][0]))
70
71 history = None
72 if test_ys is None:
73     nn_model.fit(train_features_pad, np_labels_train, batch_size=32, epochs=epochs,
74 verbose=verbose)
75 else:
76     history = nn_model.fit(train_features_pad, np_labels_train, validation_data=(test_
77 features_pad, test_ys),
78                             batch_size=32, epochs=epochs, verbose=verbose)
79
80 y_labels = nn_model.predict_classes(test_features_pad, batch_size=32, verbose=verbose)
81
82 return y_labels, history

```

#### 4.9. big\_LSTM\_CONV\_rnn.

En este modelo, se ha seguido el concepto de *stacked models*, en otras palabras, considerar un modelo de los explicados anteriormente como una capa. En este caso, hemos utilizado el modelo *LSTM-CNN* varias veces en forma de pila. En los experimentos realizados, hemos podido observar que este modelo tiene un gran sobreaprendizaje. Este sobreaprendizaje se produce por la manera de aprendizaje que se implementa y la que se puede resumir como lo siguiente: Una capa *LSTM* sirve para memorizar y capturar el cambio de sentimientos en un *tuit*, dando más peso al sinónimo que refiere a un sentimiento pero que aparece al final del texto. Por ejemplo si tenemos el siguiente *tuit*: *Al principio me encantó, pero luego terminé odiándolo*. En este caso aparecen 2 sentimientos diferentes en la frase que son totalmente opuestos, pero claramente este *tuit* nos dice que al final la persona acabó con odio y no con gusto. Y como ya hemos mencionado antes una capa *CNN* ayuda a capturar la negatividad en un texto. Juntar varias capas para analizar el sentimiento que aparece en un texto que consideramos corto, no es oportuno, porqué aumentar el número de capas ocultas puede aumentar la precisión en clasificación pero hacer que el modelo aprendido sea demasiado específico. Analizar *tuits* de unas 10 o 20 palabras no es una tarea *compleja* y, por lo tanto, no necesitamos ese sobreajuste en el proceso de aprendizaje.

```
1 from keras.layers import Conv1D
2 from keras.layers.core import Dense, Dropout
3 from keras.layers.core import Flatten
4 from keras.layers.embeddings import Embedding
5 from keras.layers.recurrent import LSTM
6 from keras.models import Sequential
7 from keras.preprocessing import sequence
8
9 from src.util.utilities import *
10
11
12 def big_LSTM_CONV_rnn(embeddings_path, train_xs, train_ys, test_xs, test_ys=None, epochs=25,
13     verbose=1):
14     own_set_seed()
15
16     # Offset = 2; Padding and OOV.
17     word_embeddings, word_emb_indexes = read_embeddings(embeddings_path, 2)
18     word_embeddings[0] = 2 * 0.1 * np.random.rand(len(word_embeddings[2])) - 1
19     word_embeddings[1] = 2 * 0.1 * np.random.rand(len(word_embeddings[2])) - 1
20
21     # Build vocabulary and corpus indexes
22     vocabulary_train, corpus_train_index = fit_transform_vocabulary_pretrain_embeddings(
23         train_xs, word_emb_indexes)
24
25     max_len_input = int(np.average([len(tweet_train) for tweet_train in corpus_train_index], 0))
26
27     corpus_test_index = []
28     own_corpus_test_index_append = corpus_test_index.append
29     own_lowercase = str.lower
30
31     for tweet_test in test_xs:
32         tokens_test = tokenize(own_lowercase(tweet_test))
33         doc_test_index = []
34         for token_test in tokens_test:
35             if RE_TOKEN_USER.fullmatch(token_test) is not None:
36                 token_test = "@user"
37             doc_test_index.append(word_emb_indexes.get(token_test, 1))
38         own_corpus_test_index_append(doc_test_index)
39
40     nn_model = Sequential()
41     nn_model.add(Embedding(len(word_embeddings), len(word_embeddings[0]), weights=[np_array(
42         word_embeddings)], input_length=max_len_input, trainable=False))
43
44     nn_model.add(LSTM(64, return_sequences=True))
45     nn_model.add(Dropout(0.5))
46
47     nn_model.add(Conv1D(128, 5, activation='relu', padding='same'))
48     nn_model.add(Dropout(0.5))
```

```

46 nn_model.add(LSTM(64, return_sequences=True))
47 nn_model.add(Dropout(0.5))
48
49 nn_model.add(Conv1D(128, 5, activation='relu', padding='same'))
50 nn_model.add(Dropout(0.5))
51
52 nn_model.add(LSTM(64, return_sequences=True))
53 nn_model.add(Dropout(0.5))
54
55 nn_model.add(Conv1D(128, 5, activation='relu', padding='same'))
56 nn_model.add(Dropout(0.5))
57
58 nn_model.add(Dense(128, activation='relu'))
59 nn_model.add(Dropout(0.5))
60
61 nn_model.add(Flatten())
62 nn_model.add(Dense(len(src.util.global_vars.__CLASSES__), activation='softmax'))
63
64 nn_model.compile(loss='sparse_categorical_crossentropy', optimizer='adam')
65
66 if verbose == 1:
67     print(nn_model.summary())
68
69 train_features_pad = sequence.pad_sequences(corpus_train_index, maxlen=max_len_input,
70 padding="post",
71                                         truncating="post", dtype=type(
72 corpus_train_index[0][0]))
73 np_labels_train = np.array(train_ys)
74 test_features_pad = sequence.pad_sequences(corpus_test_index, maxlen=max_len_input,
75 padding="post",
76                                         truncating="post", dtype=type(corpus_test_index
77 [0][0]))
78
79 history = None
80 if test_ys is None:
81     nn_model.fit(train_features_pad, np_labels_train, batch_size=32, epochs=epochs,
82 verbose=verbose)
83 else:
84     history = nn_model.fit(train_features_pad, np_labels_train, validation_data=(test_
85 features_pad, test_ys),
86                             batch_size=32, epochs=epochs, verbose=verbose)
87
88 y_labels = nn_model.predict_classes(test_features_pad, batch_size=32, verbose=verbose)
89
90 return y_labels, history

```

#### 4.10. dropout\_LSTM\_CONV\_rnn.

Este modelo es una versión modificada de *pretrain\_embeddings\_LSTM\_CONV* en la que aumentamos el valor de *Dropout* de 0.5 a 0.7, es decir, aplicamos la técnica de *Dropout* al 70% de las neuronas de la capa. La motivación de esta red ha sido intentar reducir el sobreabrendizaje. A nivel práctico, este modelo ha producido una mejora en la puntuación de *Kaggle*, pasando de un *0.51845* con *dropout = 0.5* a un *0.54305* *dropout = 0.7*.

```

1 from keras.layers import MaxPooling1D, Conv1D, AveragePooling1D
2 from keras.layers.core import Dense, Dropout
3 from keras.layers.core import Flatten
4 from keras.layers.embeddings import Embedding
5 from keras.layers.recurrent import LSTM
6 from keras.models import Sequential
7 from keras.preprocessing import sequence
8
9 from src.util.utilities import *
10
11
12 def dropout_LSMT_CONV_rnn(embeddings_path, train_xs, train_ys, test_xs, test_ys=None, epochs
13 =25, verbose=1):
14     own_set_seed()

```

```

14
15 # Offset = 2; Padding and OOV.
16 word_embeddings, word_emb_indexes = read_embeddings(embeddings_path, 2)
17 word_embeddings[0] = 2 * 0.1 * np.random.rand(len(word_embeddings[2])) - 1
18 word_embeddings[1] = 2 * 0.1 * np.random.rand(len(word_embeddings[2])) - 1
19
20 # Build vocabulary and corpus indexes
21 vocabulary_train, corpus_train_index = fit_transform_vocabulary_pretrain_embeddings(
22 train_xs, word_emb_indexes)
23
24 max_len_input = int(np.average([len(tweet_train) for tweet_train in corpus_train_index], 0))
25
26 corpus_test_index = []
27 own_corpus_test_index_append = corpus_test_index.append
28 own_lowercase = str.lower
29 for tweet_test in test_xs:
30     tokens_test = tokenize(own_lowercase(tweet_test))
31     doc_test_index = []
32     for token_test in tokens_test:
33         if RE_TOKEN_USER.fullmatch(token_test) is not None:
34             token_test = "@user"
35         doc_test_index.append(word_emb_indexes.get(token_test, 1))
36     own_corpus_test_index_append(doc_test_index)
37
38 nn_model = Sequential()
39 nn_model.add(Embedding(len(word_embeddings), len(word_embeddings[0]), weights=[np_array(
40 word_embeddings)], input_length=max_len_input, trainable=False))
41
42 nn_model.add(LSTM(64, return_sequences=True))
43 nn_model.add(Dropout(0.7))
44 nn_model.add(MaxPooling1D())
45
46 nn_model.add(Conv1D(128, 5, activation='relu', padding='same'))
47 nn_model.add(Dropout(0.7))
48 nn_model.add(AveragePooling1D())
49
50 nn_model.add(Dense(128, activation='relu'))
51 nn_model.add(Dropout(0.7))
52 nn_model.add(AveragePooling1D())
53
54 nn_model.add(Flatten())
55 nn_model.add(Dense(len(src.util.global_vars.__CLASSES__), activation='softmax'))
56
57 nn_model.compile(loss='sparse_categorical_crossentropy', optimizer='adam')
58
59 if verbose == 1:
60     print(nn_model.summary())
61
62 train_features_pad = sequence.pad_sequences(corpus_train_index, maxlen=max_len_input,
63 padding="post",
64                                         truncating="post", dtype=type(
65 corpus_train_index[0][0]))
66 np_labels_train = np.array(train_ys)
67 test_features_pad = sequence.pad_sequences(corpus_test_index, maxlen=max_len_input,
68 padding="post",
69                                         truncating="post", dtype=type(corpus_test_index
70 [0][0]))
71
72 history = None
73 if test_ys is None:
74     nn_model.fit(train_features_pad, np_labels_train, batch_size=32, epochs=epochs,
75 verbose=verbose)
76 else:
77     history = nn_model.fit(train_features_pad, np_labels_train, validation_data=(test_
78 features_pad, test_ys),
79                             batch_size=32, epochs=epochs, verbose=verbose)
80
81 y_labels = nn_model.predict_classes(test_features_pad, batch_size=32, verbose=verbose)
82 return y_labels, history

```

#### 4.11. bidirectional\_lstm\_rnn.

Las neuronas *LSTM* usadas hasta ahora son unidireccionales, esto quiere decir que, en su núcleo, preservan la información de las entradas que ya han pasado por dicha neurona. Es decir, las *LSTM* unidireccionales sólo conservan información del pasado porque las únicas entradas que han visto son del pasado. Si usamos *LSTM* bidireccionales, la entrada será pasada por la neurona de dos maneras, una primera vez de *pasado al futuro* y otra del *futuro al pasado*. Este enfoque permite conservar información del pasado y del futuro en dos estados dentro de la neurona. De esta manera, si los combinamos, somos capaces de, en cualquier momento, tener información tanto del pasado como del futuro. Veamos esto con un ejemplo. Supongamos que estamos en un problema de predicción de la siguiente palabra cuando estamos escribiendo, tal y como puede hacer el auto-corrección que usamos todos los días en los móviles. A muy alto nivel, una capa formada por neuronas *LSTM* unidireccionales podría ver *Ellos fueron....* En ese momento, intentamos predecir la siguiente palabra. Si usamos una red con capas formadas por neuronas *LSTM* bidireccionales podemos ver la información con más detalle, teniendo una información *desde el pasado al futuro*, como podría ser *Ellos fueron a comer...* e información *desde el futuro al pasado*, como ... *y luego a jugar*. Si permitimos que la red aprenda el contexto en ambos sentidos.

La red usada es la misma que la explicada en la sección *pretrain\_embeddings\_rnn* (4.3) pero con neuronas *LSTM* bidireccionales tal y como podemos ver en la línea 41.

```

1 from keras.layers import Bidirectional, GlobalMaxPool1D, Dropout
2 from keras.layers.core import Dense
3 from keras.layers.embeddings import Embedding
4 from keras.layers.recurrent import LSTM
5 from keras.models import Sequential
6 from keras.optimizers import Adam
7 from keras.preprocessing import sequence
8
9 from src.util.utilities import *
10
11
12 def bidirectional_lstm_rnn(embeddings_path, train_xs, train_ys, test_xs, test_ys=None, verbose=1):
13     own_set_seed()
14
15     # Offset = 2; Padding and OOV.
16     word_embeddings, word_emb_indexes = read_embeddings(embeddings_path, 2)
17     word_embeddings[0] = 2 * 0.1 * np.random.rand(len(word_embeddings[2])) - 1
18     word_embeddings[1] = 2 * 0.1 * np.random.rand(len(word_embeddings[2])) - 1
19
20     # Build vocabulary and corpus indexes
21     vocabulary_train, corpus_train_index = fit_transform_vocabulary_pretrain_embeddings(
22         train_xs, word_emb_indexes)
23
24     max_len_input = int(np.average([len(tweet_train) for tweet_train in corpus_train_index], 0))
25
26     corpus_test_index = []
27     own_corpus_test_index_append = corpus_test_index.append
28     own_lowercase = str.lower
29     for tweet_test in test_xs:
30         tokens_test = tokenize(own_lowercase(tweet_test))
31         doc_test_index = []
32         for token_test in tokens_test:
33             if RE_TOKEN_USER.fullmatch(token_test) is not None:
34                 token_test = "@user"
35             doc_test_index.append(word_emb_indexes.get(token_test, 1))
36             own_corpus_test_index_append(doc_test_index)
37
38     nn_model = Sequential()
39     nn_model.add(Embedding(len(word_embeddings), len(word_embeddings[0]), weights=[np_array(
40         word_embeddings)], input_length=max_len_input, trainable=False))
```

```

41 nn_model.add(Bidirectional(LSTM(100, return_sequences=True, dropout=0.25,
42 recurrent_dropout=0.1)))
43 nn_model.add(GlobalMaxPool1D())
44 nn_model.add(Dense(100, activation="relu"))
45 nn_model.add(Dropout(0.25))
46 nn_model.add(Dense(len(src.util.global_vars.__CLASSES__), activation='softmax'))
47 adam_optimizer = Adam(lr=0.001)
48 nn_model.compile(optimizer=adam_optimizer, loss="sparse_categorical_crossentropy")
49
50 if verbose == 1:
51     print(nn_model.summary())
52
53 train_features_pad = sequence.pad_sequences(corpus_train_index, maxlen=max_len_input,
54 padding="post",
55                                         truncating="post", dtype=type(
56 corpus_train_index[0][0]))
57 np_labels_train = np.array(train_ys)
58 test_features_pad = sequence.pad_sequences(corpus_test_index, maxlen=max_len_input,
59 padding="post",
60                                         truncating="post", dtype=type(corpus_test_index
61 [0][0]))
62
63 history = None
64 if test_ys is None:
65     nn_model.fit(train_features_pad, np_labels_train, batch_size=32, epochs=10, verbose=
66 verbose)
67 else:
68     history = nn_model.fit(train_features_pad, np_labels_train, validation_data=(test_
69 features_pad, test_ys),
70                             batch_size=32, epochs=10, verbose=verbose)
71
72 y_labels = nn_model.predict_classes(test_features_pad, batch_size=32, verbose=verbose)
73 return y_labels, history

```

#### 4.12. pretrained\_embeddings\_cnn\_bidirectional\_LSTM.

Al igual que hemos probado a combinar capas *LSTM* con capas convolucionales, hemos combinado la versión bidireccional de estas últimas, teniendo así una red que mezcla neuronas convolucionales (*línea 47*) y neuronas de *pooling* (*línea 49*), ambas características de redes *CNN*, con neuronas *LSTM* bidireccionales (*línea 54*).

```

1 from keras.layers import Bidirectional, Dropout, Conv1D, AveragePooling1D, MaxPooling1D,
2     Flatten
3 from keras.layers.core import Dense
4 from keras.layers.embeddings import Embedding
5 from keras.layers.recurrent import LSTM
6 from keras.models import Sequential
7 from keras.optimizers import Adam
8 from keras.preprocessing import sequence
9
10 from src.util.utilities import *
11
12 def pretrained_embeddings_cnn_bidirectional_LSTM(embeddings_path, train_xs, train_ys, test_xs,
13                                                 test_ys=None,
14                                                 verbose=1):
15     own_set_seed()
16
17     # Offset = 2; Padding and OOV.
18     word_embeddings, word_emb_indexes = read_embeddings(embeddings_path, 2)
19     word_embeddings[0] = 2 * 0.1 * np.random.rand(len(word_embeddings[2])) - 1
20     word_embeddings[1] = 2 * 0.1 * np.random.rand(len(word_embeddings[2])) - 1
21
22     # Build vocabulary and corpus indexes
23     vocabulary_train, corpus_train_index = fit_transform_vocabulary_pretrain_embeddings(
24         train_xs, word_emb_indexes)

```

```

24 max_len_input = int(np.average([len(tweet_train) for tweet_train in corpus_train_index], 0))
25
26 corpus_test_index = []
27 own_corpus_test_index.append = corpus_test_index.append
28 own_lowercase = str.lower
29 for tweet_test in test_xs:
30     tokens_test = tokenize(own_lowercase(tweet_test))
31     doc_test_index = []
32     for token_test in tokens_test:
33         if RE_TOKEN_USER.fullmatch(token_test) is not None:
34             token_test = "@user"
35         doc_test_index.append(word_emb_indexes.get(token_test, 1))
36     own_corpus_test_index.append(doc_test_index)
37
38 # Initializing the model
39 nn_model = Sequential()
40
41 # Embedding layer
42 nn_model.add(Embedding(len(word_embeddings), len(word_embeddings[0]), weights=[np_array(
43     word_embeddings)], input_length=max_len_input, trainable=False))
44
45 # Setting a CNN layer in order to extract features in each epoch. 64 neurons. this number
46 # was selected after many
47 # experiments.
48 nn_model.add(Conv1D(64, 5, activation='relu', padding='same'))
49 nn_model.add(Dropout(0.25)) # Dropout in order to avoid 'over fitting'.
50 nn_model.add(AveragePooling1D()) # Pooling layer, used to reduce variance, reduce
51 # computation complexity and
52 # extract low level features.
53
54 # Bidirectional LSTM, used to introduce an adaptive gating mechanism, which decides the
55 # degree to keep the previous
56 # state and memorize the extracted features of the current data input
57 nn_model.add(Bidirectional(LSTM(300, return_sequences=True, dropout=0.3, recurrent_dropout
58 =0.1)))
59 nn_model.add(MaxPooling1D()) # Pooling layer
60
61 # Dense layer, to densely connect all the neurons of the previous layer.
62 nn_model.add(Dense(100, activation="sigmoid"))
63 nn_model.add(Dropout(0.25))
64 nn_model.add(AveragePooling1D())
65
66 nn_model.add(Flatten())
67 nn_model.add(Dense(len(src.util.global_vars.__CLASSES__), activation='softmax'))
68
69 adam_optimizer = Adam(lr=0.001)
70 nn_model.compile(optimizer=adam_optimizer, loss="sparse_categorical_crossentropy")
71
72 if verbose == 1:
73     print(nn_model.summary())
74
75 train_features_pad = sequence.pad_sequences(corpus_train_index, maxlen=max_len_input,
76 padding="post",
77                                         truncating="post", dtype=type(
78     corpus_train_index[0][0]))
79 np_labels_train = np.array(train_ys)
80 test_features_pad = sequence.pad_sequences(corpus_test_index, maxlen=max_len_input,
81 padding="post",
82                                         truncating="post", dtype=type(corpus_test_index
83 [0][0]))
84
85 history = None
86 if test_ys is None:
87     nn_model.fit(train_features_pad, np_labels_train, batch_size=32, epochs=10, verbose=
88 verbose)
89 else:
90     history = nn_model.fit(train_features_pad, np_labels_train, validation_data=(test_
91 features_pad, test_ys),
92                             batch_size=32, epochs=10, verbose=verbose)

```

```

83
84     y_labels = nn_model.predict_classes(test_features_pad, batch_size=32, verbose=verbose)
85
86     return y_labels, history

```

## 5. Experimentos.

En esta sección se describirán los distintos experimentos realizados, combinando las arquitecturas de las redes con distintos parámetros y métodos de preprocesamiento. Para cada experimento se va a mostrar la gráfica de la función *loss* comentada anteriormente, un resumen de la arquitectura de la red obtenida con la función *summary* de *keras* aplicada sobre el modelo de red y el valor de *macro\_f1*, *micro\_f1* y el valor obtenido en la competición de *kaggle*.

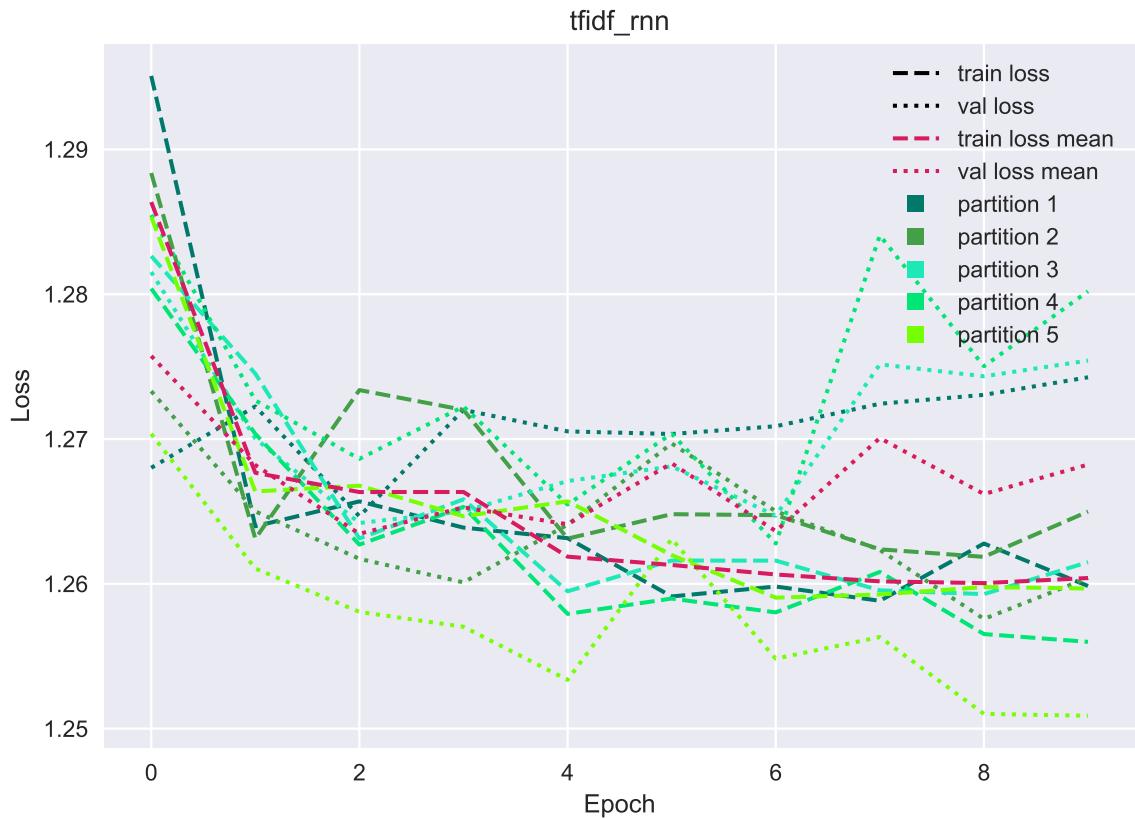
Cuando se muestre el resumen de la arquitectura, no se va a hacer en gran detalle porque se ha comentado en la sección 4. En esta sección comentaremos más el número de parámetros que tiene la red, ya que esto define la complejidad de la misma y el número de parámetros entrenables frente al número de parámetros no entrenables. Un parámetro entrenable es un parámetro cuyo valor varía durante el aprendizaje. En *keras* podemos tener varios tipos de parámetros no entrenables:

- Parámetros que han sido marcados como constantes. Esto significa que *keras* no los actualizará durante el entrenamiento.
- Parámetros estadísticos en las capas *BatchNormalization*. Estos se actualizan con la media y la varianza, pero no están con el proceso de *backpropagation*.
- Parámetros de entrada de la propia red, como pueden ser los *embeddings*.

De no indicarse lo contrario, los *embeddings* usados en todas las redes han sido los proporcionados por el profesorado de asignatura en las clases de prácticas, es decir, *fasttext\_spanish\_twitter\_100d.vec*.

### 5.1. tfidf\_rnn.

Experimento generado usando la red descrita en la sección *tfidf\_rnn* (4.1) usando los parámetros por defecto indicados en la misma.



Como podemos ver, el valor de la función *loss* decrece según van pasando las épocas, tanto para el conjunto de *train* como para el conjunto de *validation*. Esta es la situación que nos interesa ver en este tipo de gráficas. Veamos ahora la arquitectura de la red:

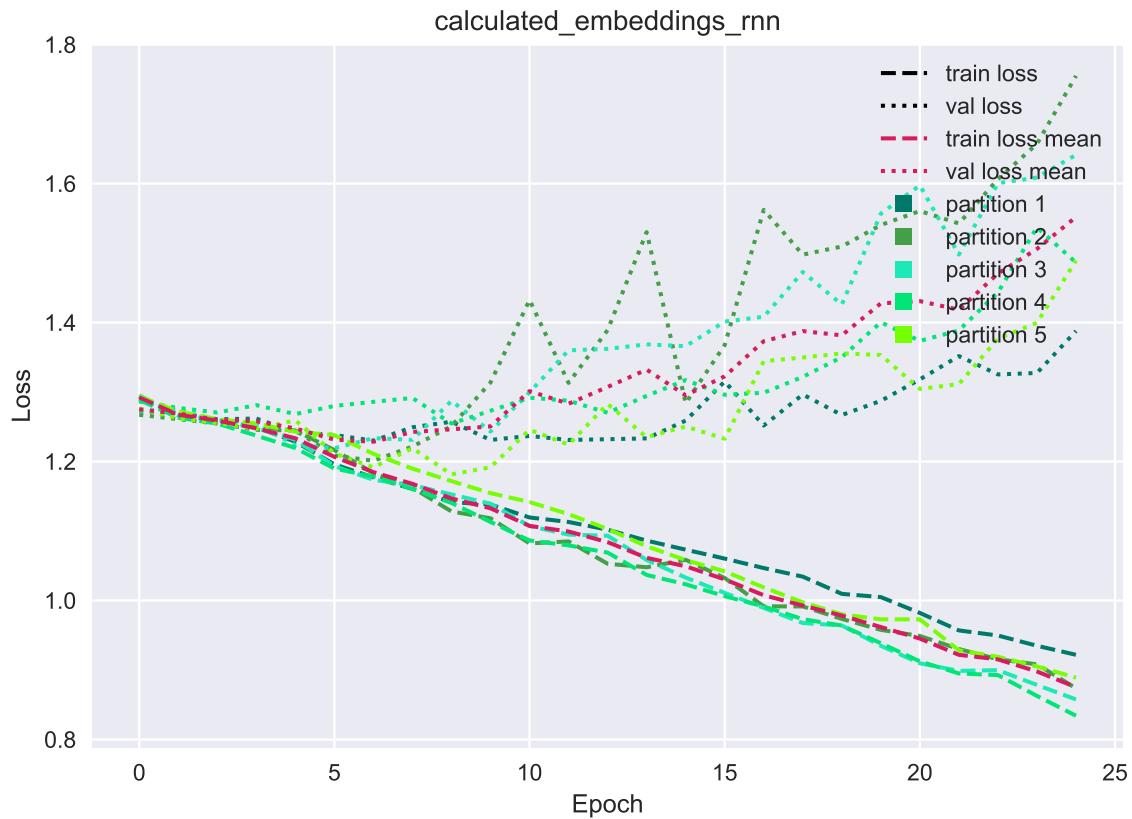
Layer (type)	Output Shape	Param #
=====		
lstm_1 (LSTM)	(None, 29, 64)	16896
dense_1 (Dense)	(None, 29, 32)	2080
flatten_1 (Flatten)	(None, 928)	0
dense_2 (Dense)	(None, 4)	3716
=====		
Total params:	22,692	
Trainable params:	22,692	
Non-trainable params:	0	

Como podemos ver, todos los parámetros de la red son entrenables. Los resultados de este experimento son:

**macro\_f1:** 0.159393 | **micro\_f1:** 0.414133 | **kaggle:** 0.42179

## 5.2. calculated\_embeddings\_rnn.

Experimento generado usando la red descrita en la sección *calculated\_embeddings\_rnn* (4.2) usando los parámetros por defecto indicados en la misma.



En este caso, el valor de la función *loss* decrece según van pasando las épocas hasta la época cinco, pero luego se separa, sigue decreciendo para el conjunto de *train* pero empieza a crecer para el conjunto de *validation*. Esta situación nos indica que la red está sobreaprendiendo, tal y como hemos explicado anteriormente. Veamos ahora la arquitectura de la red:

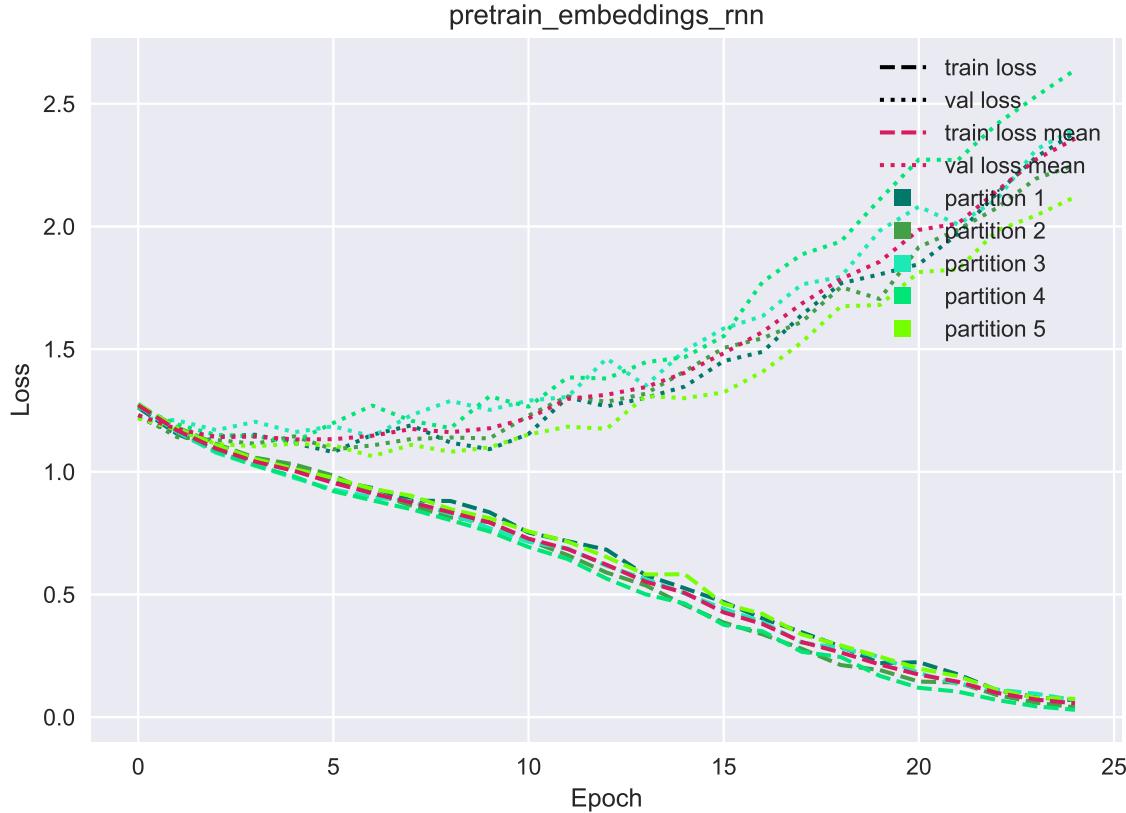
Layer (type)	Output Shape	Param #
<hr/>		
embedding_1 (Embedding)	(None, 35, 100)	499000
lstm_2 (LSTM)	(None, 35, 64)	42240
dense_3 (Dense)	(None, 35, 32)	2080
flatten_2 (Flatten)	(None, 1120)	0
dense_4 (Dense)	(None, 4)	4484
<hr/>		
Total params:	547,804	
Trainable params:	48,804	
Non-trainable params:	499,000	

En este caso, no todos los parámetros de la red son entrenables. Si nos fijamos en el *summary* de la arquitectura de la red, los parámetros no entrenables se encuentran en la capa de *Embedding*, tal y como se ha comentado en la introducción a esta sección. Los resultados de este experimento son:

**macro\_f1: 0.283373 | micro\_f1: 0.433367 | kaggle: 0.45342**

### 5.3. pretrain\_embeddings\_rnn.

Experimento generado usando la red descrita en la sección *pretrain\_embeddings\_rnn* (4.3) usando los parámetros por defecto indicados en la misma.



En este caso, el valor de la función *loss* decrece según van pasando las épocas hasta la época cinco, pero luego se separa, sigue decreciendo para el conjunto de *train* pero empieza a crecer para el conjunto de *validation*. Esta situación nos indica que la red está sobreaprendiendo, es decir, se hace muy buena en el conjunto de *train* pero, cuando la enfrentas a datos nuevos, el resultado es bastante peor. Veamos ahora la arquitectura de la red:

Layer (type)	Output Shape	Param #
<hr/>		
embedding_2 (Embedding)	(None, 16, 100)	51334200
lstm_3 (LSTM)	(None, 16, 64)	42240
dense_5 (Dense)	(None, 16, 32)	2080
flatten_3 (Flatten)	(None, 512)	0
dense_6 (Dense)	(None, 4)	2052
<hr/>		
Total params: 51,380,572		
Trainable params: 46,372		
Non-trainable params: 51,334,200		

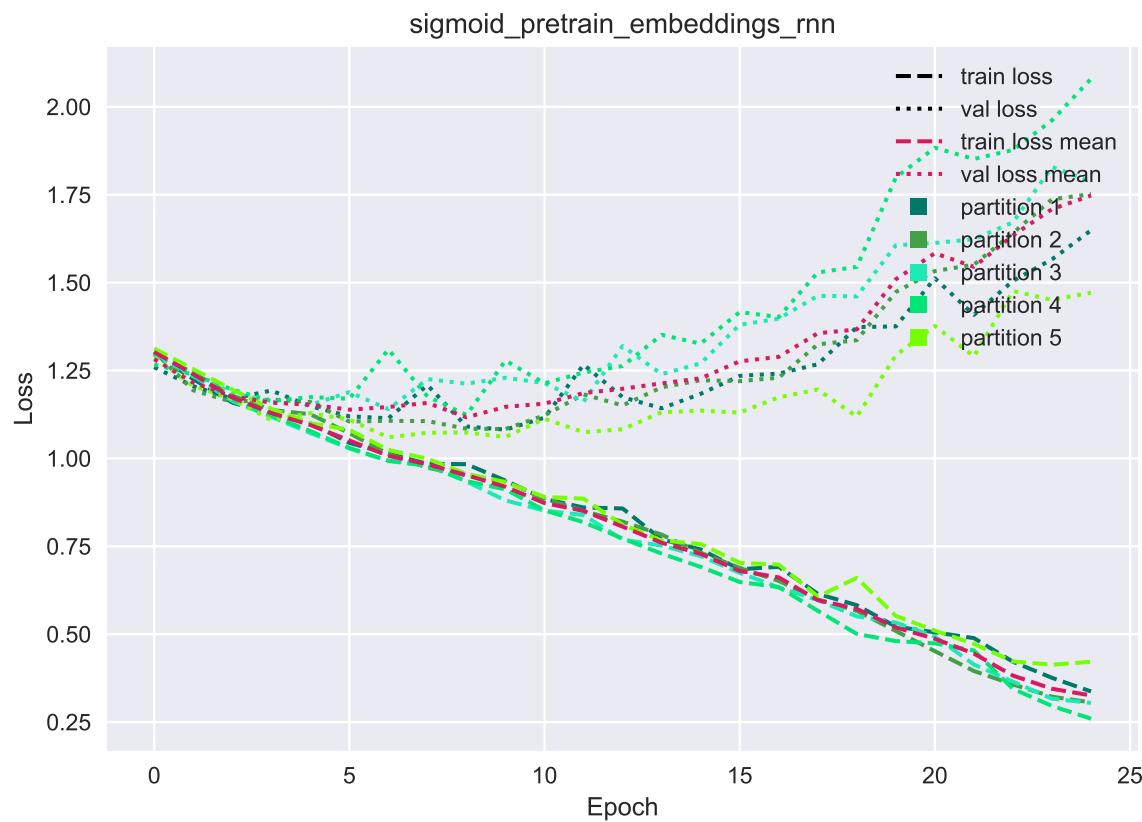
En este caso, no todos los parámetros de la red son entrenables. Si nos fijamos en el *summary* de la arquitectura de la red, los parámetros no entrenables se encuentran en la capa de *Embedding*, tal y como

se ha comentado en la introducción a esta sección. En este caso, el número de parámetros no entrenados es mucho mayor ya que el conjunto de *embeddings* se le proporciona a la red y es bastante mayor del calculado por la red en el experimento anterior. Los resultados de este experimento son:

**macro\_f1:** 0.417279 | **micro\_f1:** 0.507906 | **kaggle:** 0.49384

#### 5.4. sigmoid\_pretrain\_embeddings\_rnn.

Experimento generado usando la red descrita en la sección *sigmoid\_pretrain\_embeddings\_rnn* (4.4) usando los parámetros por defecto indicados en la misma.



Al igual que en el experimento anterior, el valor de la función *loss* decrece según van pasando las épocas hasta la época cinco, pero luego se separa, sigue decreciendo para el conjunto de *train* pero empieza a crecer para el conjunto de *validation*, lo que nos indica que la red está sobreaprendiendo. Sin embargo, si nos fijamos en la escala sobre la que se mueven los valores de la función *loss*, podemos ver que es menor, lo cual nos indica que la red está sobreaprendiendo menos que en el experimento anterior. Veamos ahora la arquitectura de la red:

Layer (type)	Output Shape	Param #
<hr/>		
embedding_3 (Embedding)	(None, 16, 100)	51334200
lstm_4 (LSTM)	(None, 16, 64)	42240
dense_7 (Dense)	(None, 16, 32)	2080
flatten_4 (Flatten)	(None, 512)	0

```

dense_8 (Dense)           (None, 4)          2052
=====
Total params: 51,380,572
Trainable params: 46,372
Non-trainable params: 51,334,200

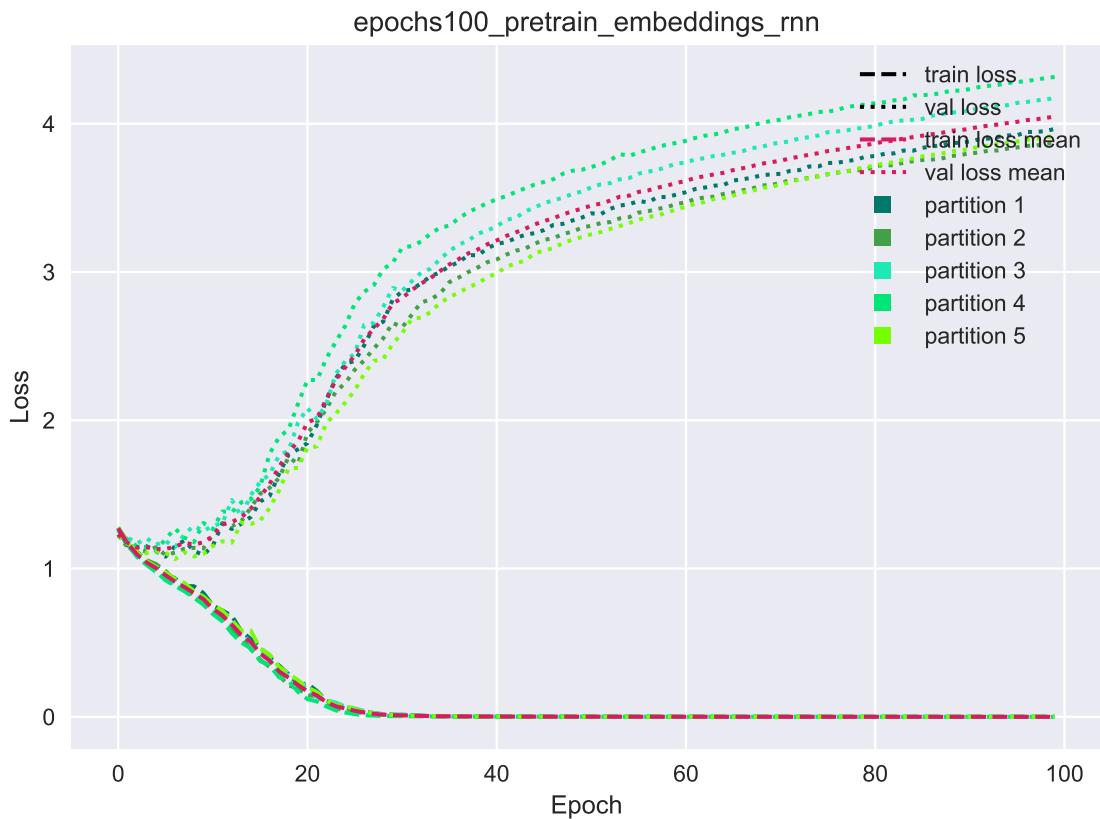
```

Una vez más, no todos los parámetros de la red son entrenables pero los parámetros no entrenables se encuentran en la capa de *Embedding*. Los resultados de este experimento son:

**macro\_f1: 0.408407 | micro\_f1: 0.506601 | kaggle: 0.49560**

### 5.5. epochs100\_pretrain\_embeddings\_rnn.

Experimento generado usando la red descrita en la sección *pretrain\_embeddings\_rnn* (4.3) usando los parámetros por defecto indicados en la misma. La diferencia con el experimento *pretrain\_embeddings\_rnn* (5.3) es que el número de épocas se ha incrementado hasta 100.



Si la red ya sobreaprendía con 25 épocas, podemos ver que con 100 épocas el sobreaprendizaje es bastante mayor. Veamos ahora la arquitectura de la red:

```

Layer (type)          Output Shape         Param #
=====
embedding_4 (Embedding)    (None, 16, 100)      51334200
lstm_5 (LSTM)            (None, 16, 64)       42240

```

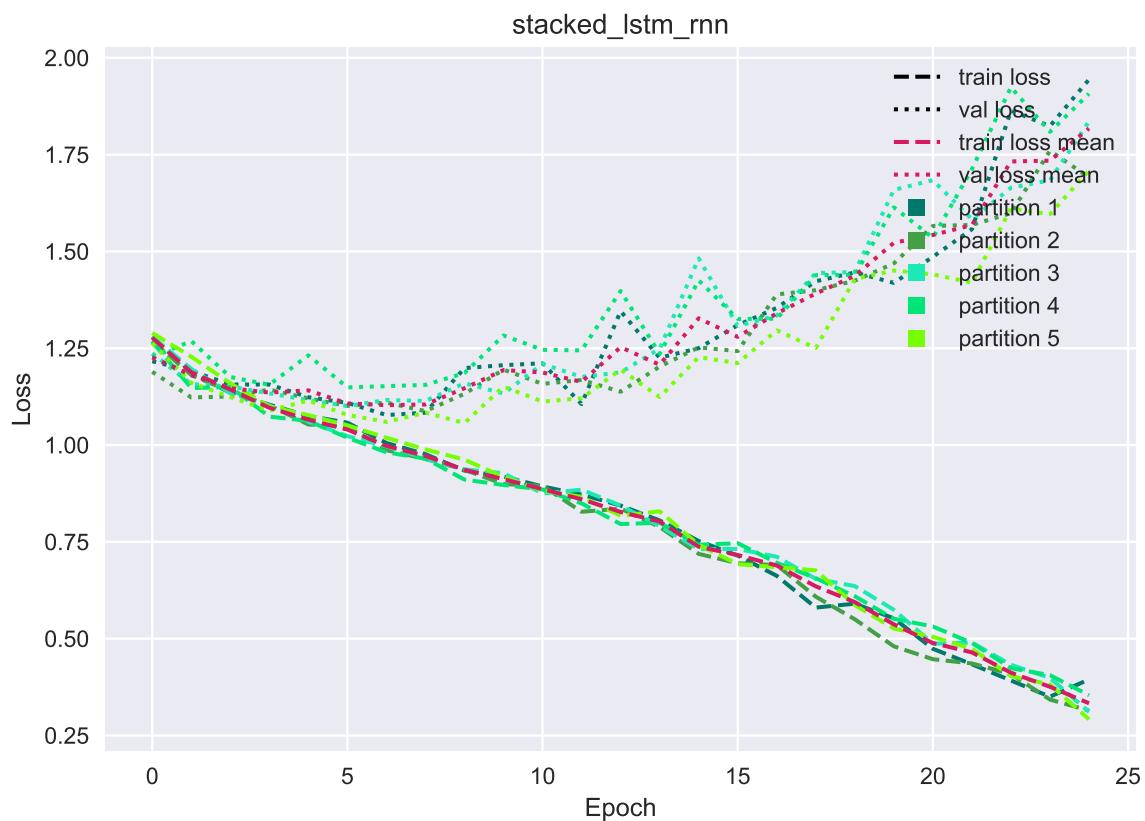
dense_9 (Dense)	(None, 16, 32)	2080
flatten_5 (Flatten)	(None, 512)	0
dense_10 (Dense)	(None, 4)	2052
=====		
Total params:		51,380,572
Trainable params:		46,372
Non-trainable params:		51,334,200

Como en los experimentos anteriores, no todos los parámetros de la red son entrenables y esto se debe a los *embeddings*. Los resultados de este experimento son:

**macro\_f1:** 0.430359 | **micro\_f1:** 0.516541 | **kaggle:** 0.49560

## 5.6. stacked\_lstm\_rnn.

Experimento generado usando la red descrita en la sección *stacked\_lstm\_rnn* (4.5), usando los parámetros por defecto indicados en la misma.



Al igual que en los experimentos anteriores, podemos apreciar que la red sobreaprende. Veamos ahora la arquitectura de la red:

Layer (type)	Output Shape	Param #
=====		
embedding_5 (Embedding)	(None, 16, 100)	51334200

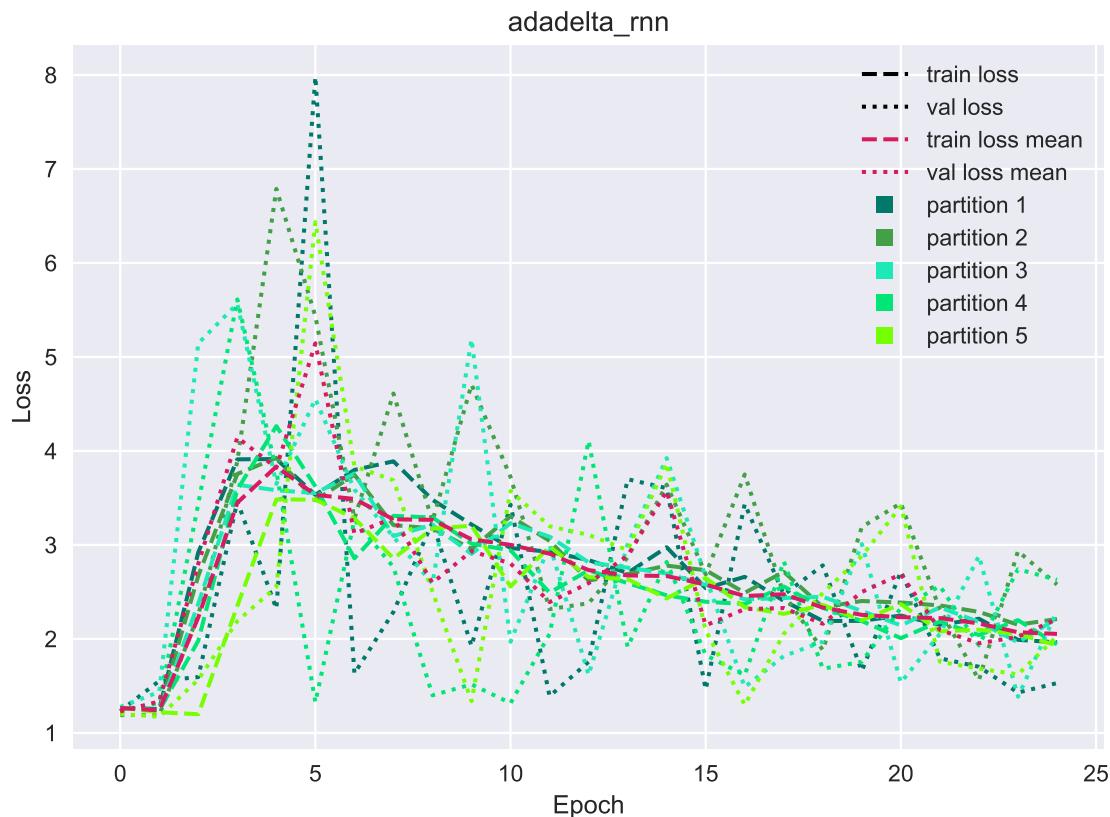
lstm_6 (LSTM)	(None, 16, 64)	42240
lstm_7 (LSTM)	(None, 16, 64)	33024
lstm_8 (LSTM)	(None, 16, 64)	33024
lstm_9 (LSTM)	(None, 64)	33024
dense_11 (Dense)	(None, 4)	260
<hr/>		
Total params:		51,475,772
Trainable params:		141,572
Non-trainable params:		51,334,200

Como en los experimentos anteriores, no todos los parámetros de la red son entrenables y esto se debe a los *embeddings*. Los resultados de este experimento son:

**macro\_f1:** 0.434354 | **micro\_f1:** 0.531007 | **kaggle:** 0.52372

## 5.7. adadelta\_rnn.

Experimento generado usando la red descrita en la sección *adadelta\_rnn* (4.6), usando los parámetros por defecto indicados en la misma.



En este caso, no podemos decir que la red sobreaprenda ya que los valores de la función *loss* para el conjunto de *train* y el conjunto de *validation* evolucionan de forma similar con el paso de las épocas. Veamos ahora la arquitectura de la red:

Layer (type)	Output Shape	Param #
--------------	--------------	---------

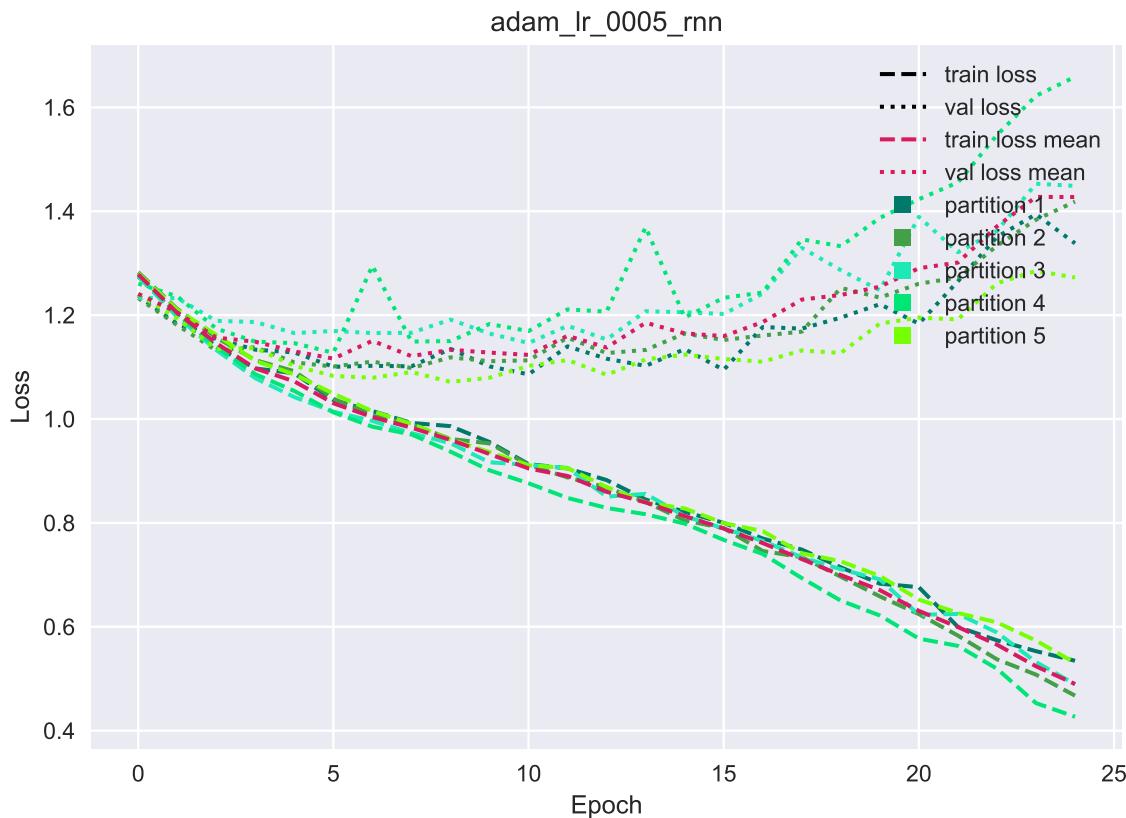
```
=====
embedding_1 (Embedding)      (None, 16, 100)      51334200
lstm_1 (LSTM)                (None, 16, 64)       42240
dense_1 (Dense)              (None, 16, 32)       2080
flatten_1 (Flatten)          (None, 512)          0
dense_2 (Dense)              (None, 4)            2052
=====
Total params: 51,380,572
Trainable params: 46,372
Non-trainable params: 51,334,200
```

Como en los experimentos anteriores, no todos los parámetros de la red son entrenables y esto se debe a los *embeddings*. Los resultados de este experimento son:

**macro\_f1: 0.100923 | micro\_f1: 0.262983 | kaggle: No entregado**

### 5.8. adam\_lr\_0005\_rnn.

Experimento generado usando la red descrita en la sección *pretrain\_embeddings\_rnn* (4.3). En este caso, se ha cambiado el *learning rate* para que sea de *0.005*.



El *learning rate* indica como de rápido aprende la red. Ya que hemos indicado un valor más alto, la red aprende más lento y, por lo tanto, no sobreaprende tanto como en los ejemplos anteriores, ya que no tiene tiempo (épocas) para hacerlo. Veamos ahora la arquitectura de la red:

Layer (type)	Output Shape	Param #
embedding_7 (Embedding)	(None, 16, 100)	51334200
lstm_14 (LSTM)	(None, 16, 64)	42240
dense_13 (Dense)	(None, 16, 32)	2080
flatten_6 (Flatten)	(None, 512)	0
dense_14 (Dense)	(None, 4)	2052

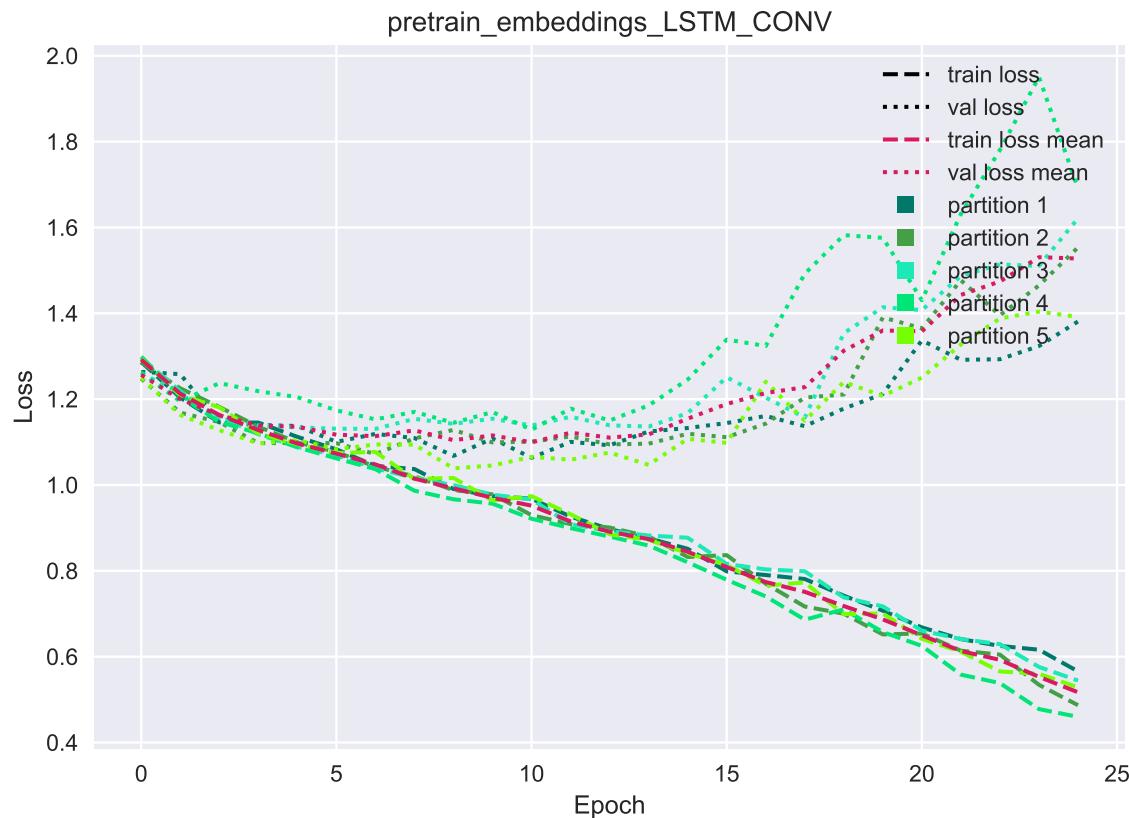
Total params: 51,380,572  
Trainable params: 46,372  
Non-trainable params: 51,334,200

Como en los experimentos anteriores, no todos los parámetros de la red son entrenables y esto se debe a los *embeddings*. Los resultados de este experimento son:

**macro\_f1:** 0.424711 | **micro\_f1:** 0.526395 | **kaggle:** 0.50439

## 5.9. pretrain\_embeddings\_LSTM\_CONV.

Experimento generado usando la red descrita en la sección *pretrain\_embeddings\_LSTM\_CONV* (4.8), usando los parámetros por defecto indicados en la misma.



Al igual que en los experimentos anteriores, podemos apreciar que la red sobreaprende. Veamos ahora la arquitectura de la red:

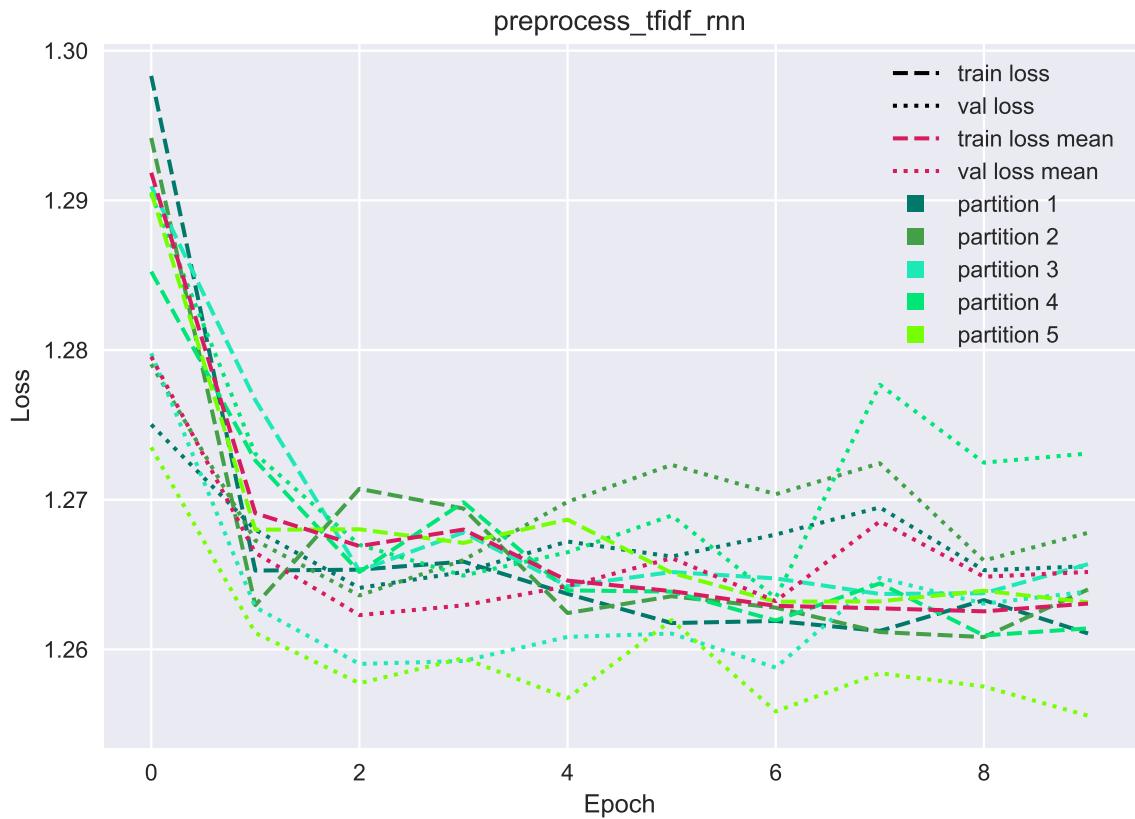
Layer (type)	Output Shape	Param #
<hr/>		
embedding_8 (Embedding)	(None, 16, 100)	51334200
lstm_15 (LSTM)	(None, 16, 64)	42240
dropout_1 (Dropout)	(None, 16, 64)	0
max_pooling1d_1 (MaxPooling1D)	(None, 8, 64)	0
conv1d_1 (Conv1D)	(None, 8, 128)	41088
dropout_2 (Dropout)	(None, 8, 128)	0
average_pooling1d_1 (AveragePooling1D)	(None, 4, 128)	0
dense_15 (Dense)	(None, 4, 128)	16512
dropout_3 (Dropout)	(None, 4, 128)	0
average_pooling1d_2 (AveragePooling1D)	(None, 2, 128)	0
flatten_7 (Flatten)	(None, 256)	0
dense_16 (Dense)	(None, 4)	1028
<hr/>		
Total params: 51,435,068		
Trainable params: 100,868		
Non-trainable params: 51,334,200		

Como en los experimentos anteriores, no todos los parámetros de la red son entrenables y esto se debe a los *embeddings*. Los resultados de este experimento son:

**macro\_f1:** 0.443507 | **micro\_f1:** 0.526316 | **kaggle:** 0.51845

## 5.10. preprocess\_tfidf\_rnn.

Experimento generado usando la red descrita en la sección *tfidf\_rnn* (4.1), usando los parámetros por defecto indicados en la misma. La diferencia es que se ha aplicado el preprocesamiento explicado en la sección *Preprocesamiento* (3.2), sin contar *oversampling*.



Al igual que sucedía en el experimento *tfidf\_rnn* (5.1), esta red no sobreaprende, ya que el valor de la función *loss* evoluciona de manera similar para el conjunto de *train* y el conjunto *validation*. Veamos ahora la arquitectura de la red:

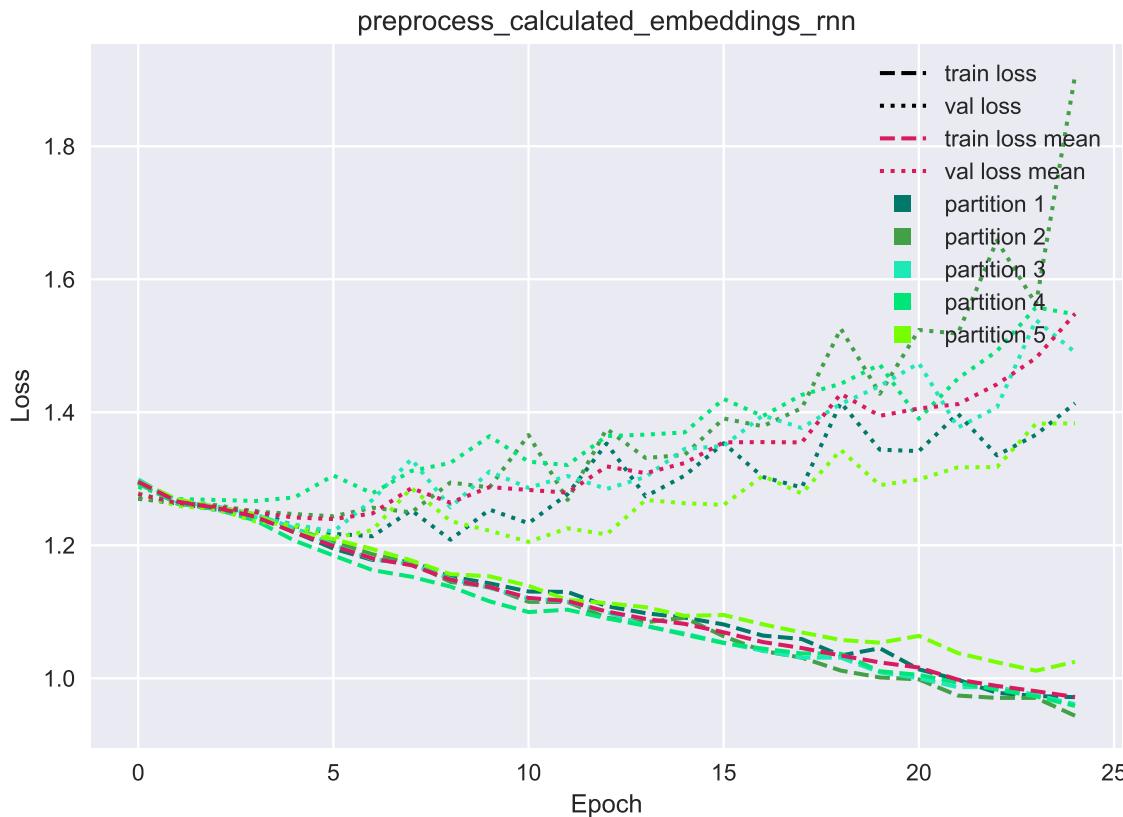
Layer (type)	Output Shape	Param #
=====		
lstm_16 (LSTM)	(None, 20, 64)	16896
dense_17 (Dense)	(None, 20, 32)	2080
flatten_8 (Flatten)	(None, 640)	0
dense_18 (Dense)	(None, 4)	2564
=====		
Total params:	21,540	
Trainable params:	21,540	
Non-trainable params:	0	

Como en los experimentos anteriores, no todos los parámetros de la red son entrenables y esto se debe a los *embeddings*. Los resultados de este experimento son:

**macro\_f1: 0.148071 | micro\_f1: 0.420742 | kaggle: 0.42179**

### 5.11. preprocess\_calculated\_embeddings\_rnn.

Experimento generado usando la red descrita en la sección *calculated\_embeddings\_rnn* (4.2), usando los parámetros por defecto indicados en la misma. La diferencia es que se ha aplicado el preprocesamiento explicado en la sección *Preprocesamiento* (3.2), sin contar *oversampling*.



Al igual que sucedía en el experimento *calculated\_embeddings\_rnn* (5.2), esta red sobreaprende. Veamos ahora la arquitectura de la red:

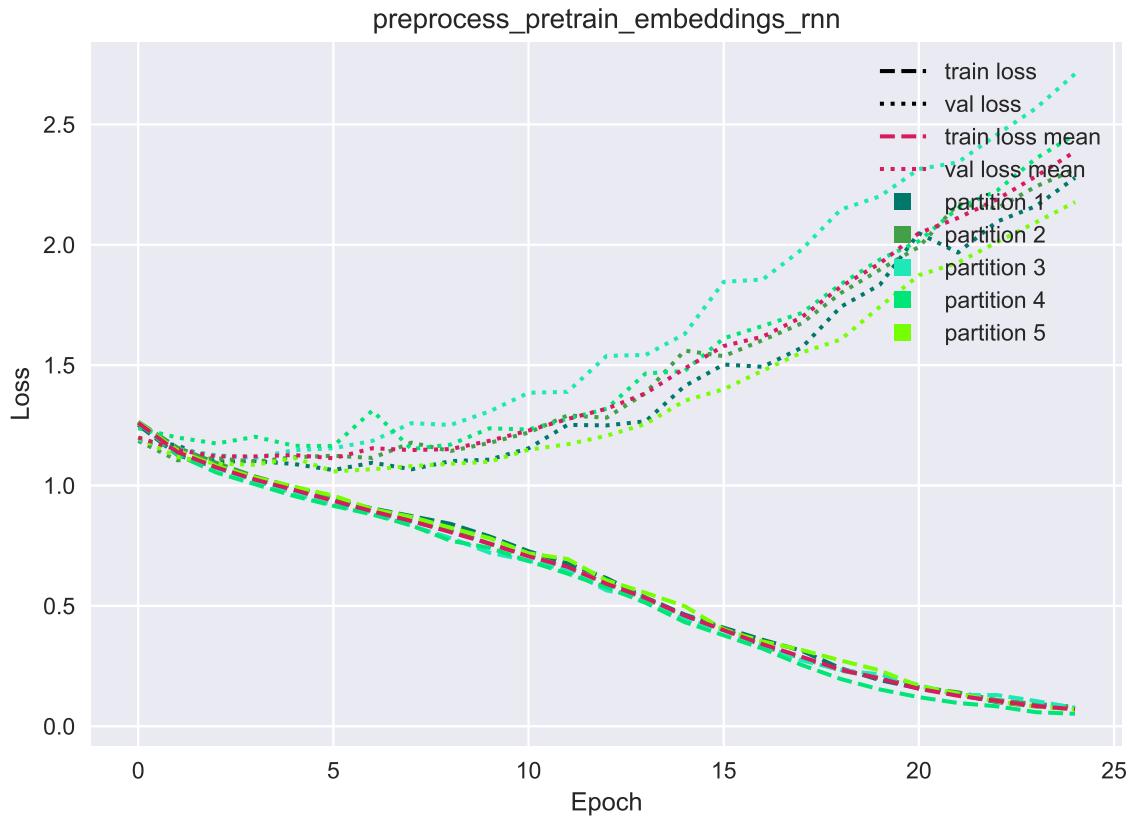
Layer (type)	Output Shape	Param #
=====		
embedding_9 (Embedding)	(None, 27, 100)	379000
lstm_17 (LSTM)	(None, 27, 64)	42240
dense_19 (Dense)	(None, 27, 32)	2080
flatten_9 (Flatten)	(None, 864)	0
dense_20 (Dense)	(None, 4)	3460
=====		
Total params: 426,780		
Trainable params: 47,780		
Non-trainable params: 379,000		

Como en los experimentos anteriores, no todos los parámetros de la red son entrenables y esto se debe a los *embeddings*. Los resultados de este experimento son:

**macro\_f1: 0.256807 | micro\_f1: 0.372415 | kaggle: 0.41827**

## 5.12. preprocess\_pretrain\_embeddings\_rnn.

Experimento generado usando la red descrita en la sección *pretrain\_embeddings\_rnn* (4.3), usando los parámetros por defecto indicados en la misma. La diferencia es que se ha aplicado el preprocesamiento explicado en la sección *Preprocesamiento* (3.2), sin contar *oversampling*.



Al igual que sucedía en el experimento *pretrain\_embeddings\_rnn* (5.3), esta red sobreaprende. Veamos ahora la arquitectura de la red:

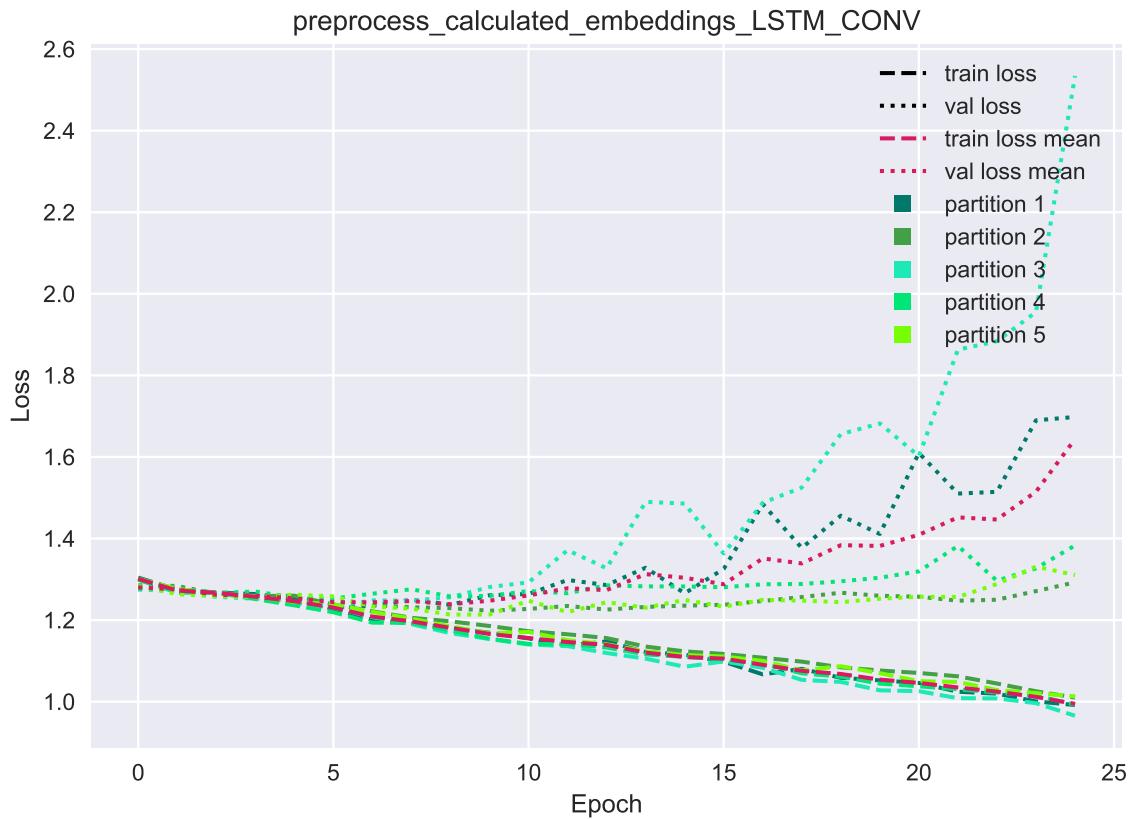
Layer (type)	Output Shape	Param #
=====		
embedding_10 (Embedding)	(None, 9, 100)	51334200
lstm_18 (LSTM)	(None, 9, 64)	42240
dense_21 (Dense)	(None, 9, 32)	2080
flatten_10 (Flatten)	(None, 288)	0
dense_22 (Dense)	(None, 4)	1156
=====		
Total params:	51,379,676	
Trainable params:	45,476	
Non-trainable params:	51,334,200	

Como en los experimentos anteriores, no todos los parámetros de la red son entrenables y esto se debe a los *embeddings*. Los resultados de este experimento son:

**macro\_f1: 0.416798 | micro\_f1: 0.50527 | kaggle: 0.50615**

### 5.13. preprocess\_calculated\_embeddings\_LSTM\_CONV.

Experimento generado usando la red descrita en la sección *calculated\_embeddings\_LSTM\_CONV* (4.7), usando los parámetros por defecto indicados en la misma. La diferencia es que se ha aplicado el preprocesamiento explicado en la sección *Preprocesamiento* (3.2), sin contar *oversampling*.



Como podemos ver, esta red sobreaprende, ya que el valor de la función *loss* evoluciona de manera distinta para el conjunto de *train* y el conjunto *validation*, despuntándose incluso en una partición (la 3) a unos valores bastante más altos. Veamos ahora la arquitectura de la red:

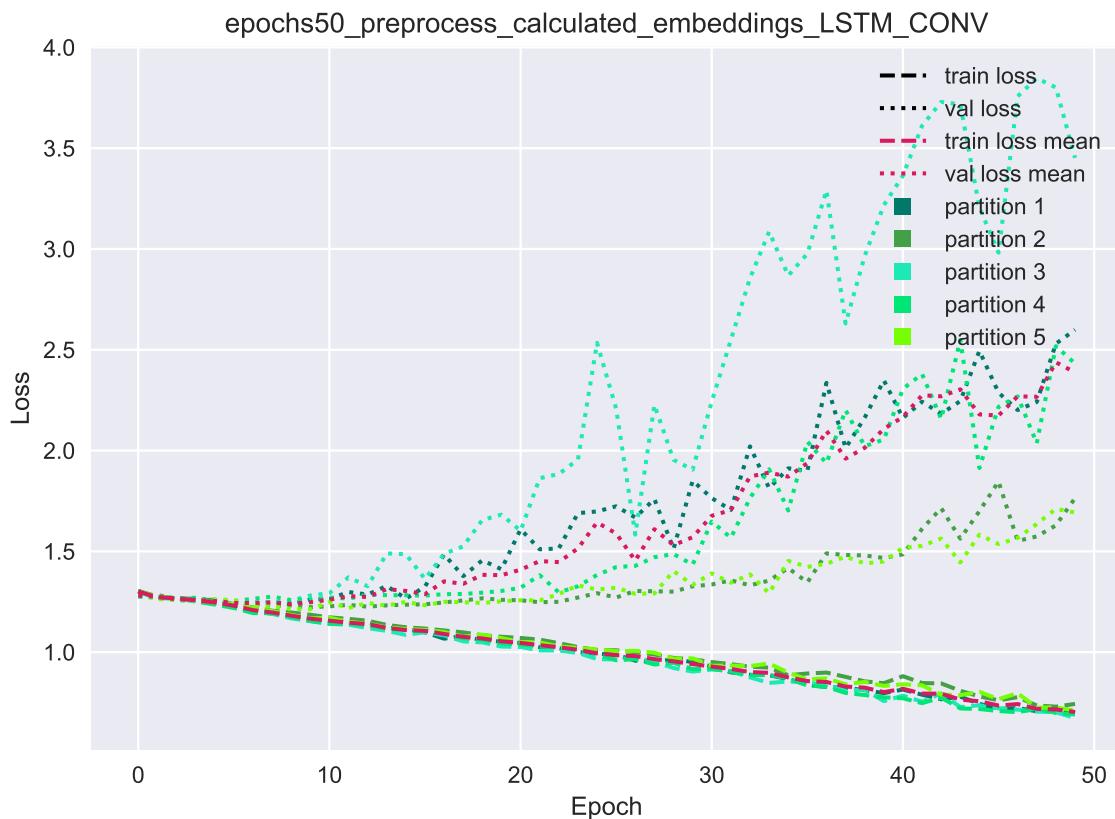
Layer (type)	Output Shape	Param #
=====		
embedding_12 (Embedding)	(None, 27, 100)	379000
lstm_20 (LSTM)	(None, 27, 64)	42240
dropout_7 (Dropout)	(None, 27, 64)	0
max_pooling1d_3 (MaxPooling1D)	(None, 13, 64)	0
conv1d_3 (Conv1D)	(None, 13, 128)	41088
dropout_8 (Dropout)	(None, 13, 128)	0
average_pooling1d_5 (Average)	(None, 6, 128)	0
dense_25 (Dense)	(None, 6, 128)	16512
dropout_9 (Dropout)	(None, 6, 128)	0
average_pooling1d_6 (Average)	(None, 3, 128)	0
flatten_12 (Flatten)	(None, 384)	0
dense_26 (Dense)	(None, 4)	1540
=====		
Total params: 480,380		
Trainable params: 101,380		
Non-trainable params: 379,000		

Como en los experimentos anteriores, no todos los parámetros de la red son entrenables y esto se debe a los *embeddings*. Los resultados de este experimento son:

**macro\_f1:** 0.260481 | **micro\_f1:** 0.450435 | **kaggle:** 0.36731

### 5.14. epochs50\_preprocess\_calculated\_embeddings\_LSTM\_CONV.

Este experimento es el mismo que el mostrado en la sección *preprocess\_calculated\_embeddings\_LSTM\_CONV*. 5.13 pero con más épocas, 50 en vez de 25. Hemos hecho este experimento para ver que sucedía con esa partición que despuntaba en las últimas épocas de aprendizaje.



Como podemos ver, el valor de la función *loss* sigue creciendo conforme pasan las épocas y, por ello, aumenta el sobreaprendizaje de la red. Veamos ahora la arquitectura de la red:

Layer (type)	Output Shape	Param #
<hr/>		
embedding_13 (Embedding)	(None, 27, 100)	379000
lstm_21 (LSTM)	(None, 27, 64)	42240
dropout_10 (Dropout)	(None, 27, 64)	0
max_pooling1d_4 (MaxPooling1D)	(None, 13, 64)	0
conv1d_4 (Conv1D)	(None, 13, 128)	41088
dropout_11 (Dropout)	(None, 13, 128)	0
average_pooling1d_7 (Average)	(None, 6, 128)	0
dense_27 (Dense)	(None, 6, 128)	16512
dropout_12 (Dropout)	(None, 6, 128)	0
average_pooling1d_8 (Average)	(None, 3, 128)	0
flatten_13 (Flatten)	(None, 384)	0

```

dense_28 (Dense)           (None, 4)          1540
=====
Total params: 480,380
Trainable params: 101,380
Non-trainable params: 379,000

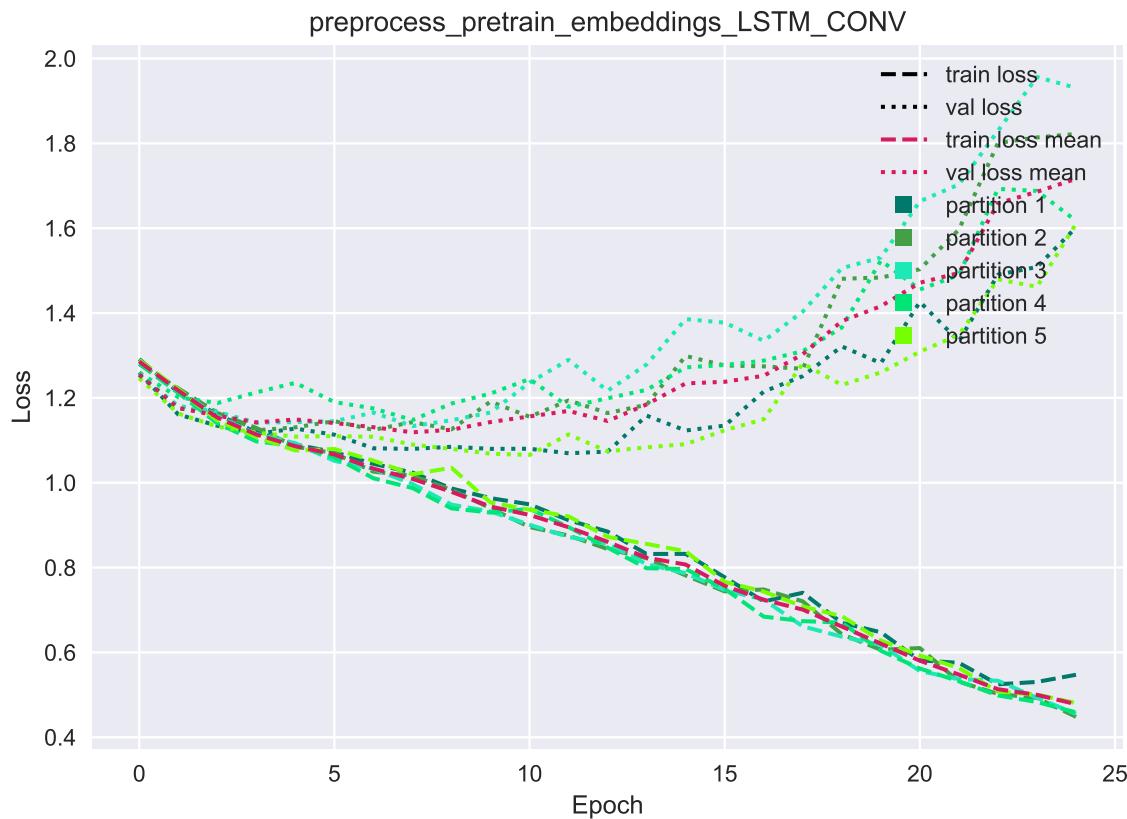
```

Como en los experimentos anteriores, no todos los parámetros de la red son entrenables y esto se debe a los *embeddings*. Los resultados de este experimento son:

**macro\_f1: 0.297376 | micro\_f1: 0.418076 | kaggle: 0.36203**

### 5.15. preprocess\_pretrain\_embeddings\_LSTM\_CONV.

Experimento generado usando la red descrita en la sección *pretrain\_embeddings\_LSTM\_CONV* (4.8), usando los parámetros por defecto indicados en la misma. La diferencia es que se ha aplicado el preprocesamiento explicado en la sección *Preprocesamiento* (3.2), sin contar *oversampling*.



Al igual que sucedía en el experimento *pretrain\_embeddings\_LSTM\_CONV* (5.9), esta red sobreaprende. Veamos ahora la arquitectura de la red:

```

Layer (type)          Output Shape       Param #
=====
embedding_11 (Embedding)    (None, 9, 100)      51334200
lstm_19 (LSTM)            (None, 9, 64)        42240

```

```

dropout_4 (Dropout)           (None, 9, 64)          0
max_pooling1d_2 (MaxPooling1) (None, 4, 64)          0
conv1d_2 (Conv1D)             (None, 4, 128)         41088
dropout_5 (Dropout)           (None, 4, 128)          0
average_pooling1d_3 (Average) (None, 2, 128)          0
dense_23 (Dense)              (None, 2, 128)         16512
dropout_6 (Dropout)           (None, 2, 128)          0
average_pooling1d_4 (Average) (None, 1, 128)          0
flatten_11 (Flatten)          (None, 128)            0
dense_24 (Dense)              (None, 4)               516
=====
Total params: 51,434,556
Trainable params: 100,356
Non-trainable params: 51,334,200

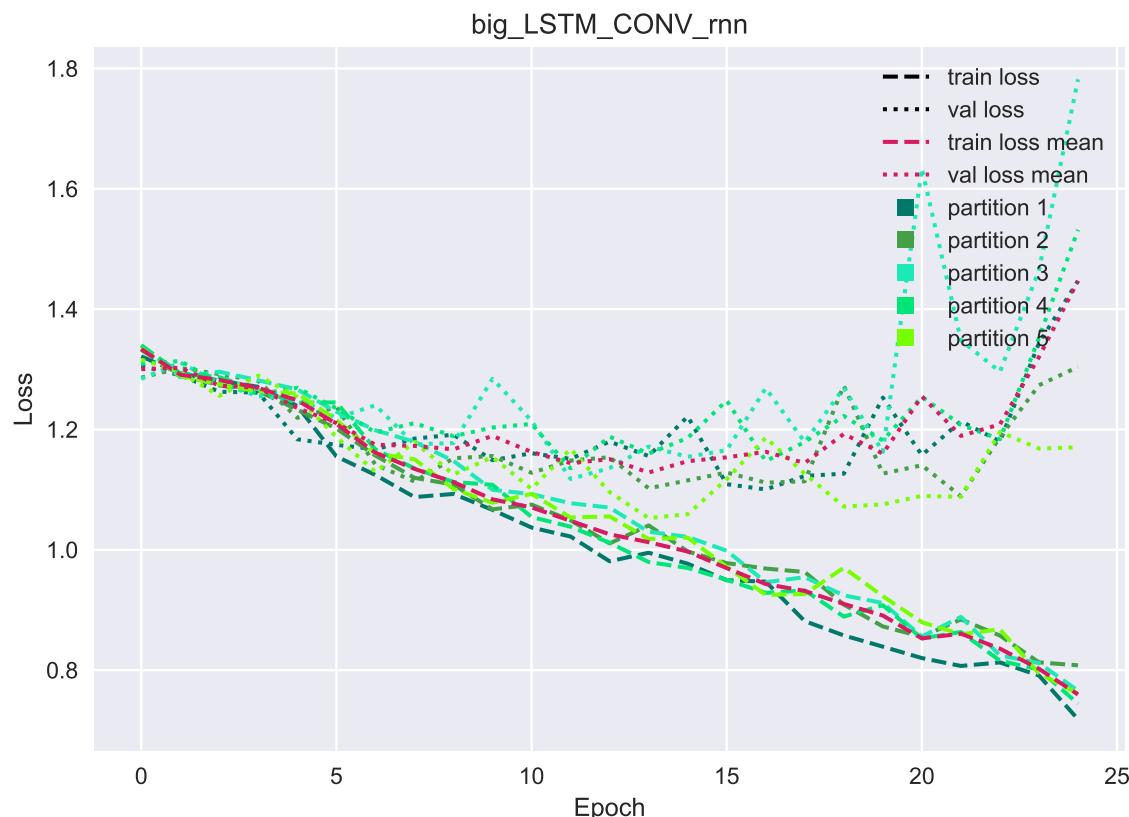
```

Como en los experimentos anteriores, no todos los parámetros de la red son entrenables y esto se debe a los *embeddings*. Los resultados de este experimento son:

**macro\_f1: 0.418226 | micro\_f1: 0.48079 | kaggle: 0.50790**

### 5.16. big\_LSTM\_CONV\_rnn.

Experimento generado usando la red descrita en la sección *big\_LSTM\_CONV\_rnn* (4.9), usando los parámetros por defecto indicados en la misma.



Viendo la gráfica podemos ver que esta red sobreaprende, lo cual tiene sentido, ya que al aumentar el número de capas, la red es capaz de extraer conocimiento más específico y, por lo tanto, perder esa capacidad de generalización necesaria para clasificar correctamente nuevas instancias. Veamos ahora la arquitectura de la red:

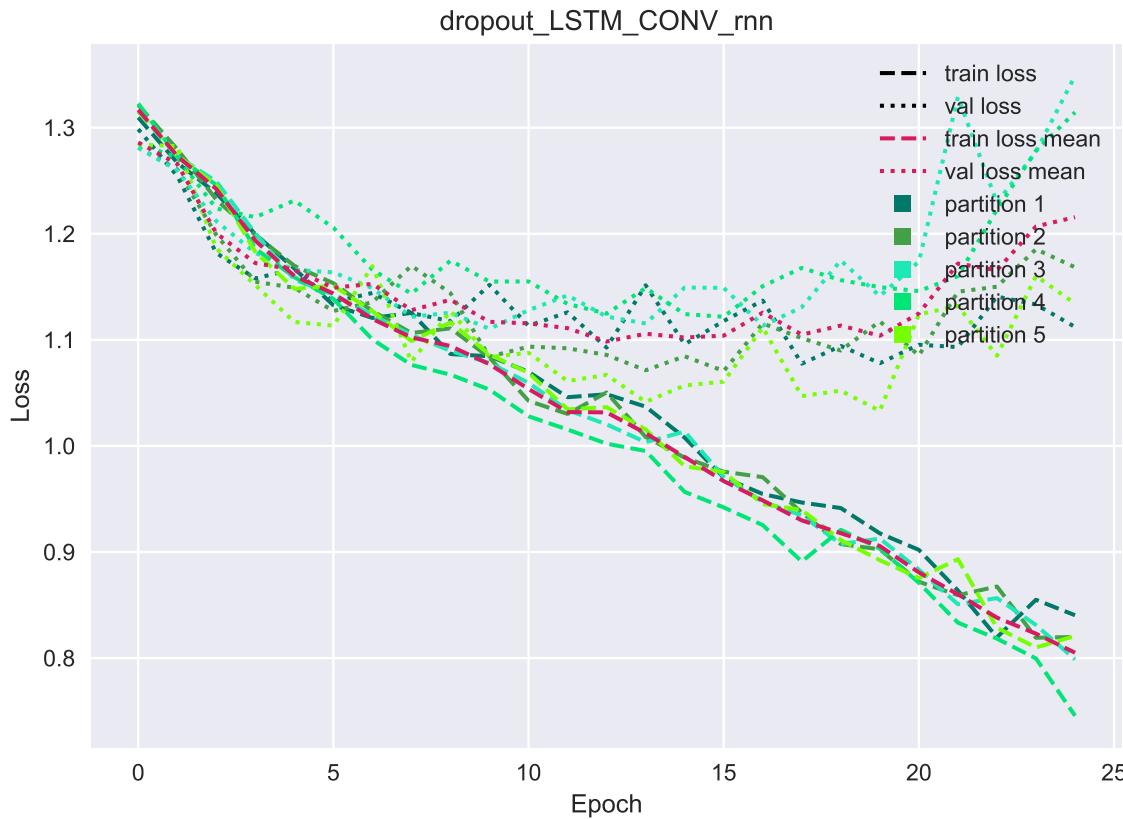
Layer (type)	Output Shape	Param #
<hr/>		
embedding_14 (Embedding)	(None, 9, 100)	51334200
lstm_22 (LSTM)	(None, 9, 64)	42240
dropout_13 (Dropout)	(None, 9, 64)	0
conv1d_5 (Conv1D)	(None, 9, 128)	41088
dropout_14 (Dropout)	(None, 9, 128)	0
lstm_23 (LSTM)	(None, 9, 64)	49408
dropout_15 (Dropout)	(None, 9, 64)	0
conv1d_6 (Conv1D)	(None, 9, 128)	41088
dropout_16 (Dropout)	(None, 9, 128)	0
lstm_24 (LSTM)	(None, 9, 64)	49408
dropout_17 (Dropout)	(None, 9, 64)	0
conv1d_7 (Conv1D)	(None, 9, 128)	41088
dropout_18 (Dropout)	(None, 9, 128)	0
dense_29 (Dense)	(None, 9, 128)	16512
dropout_19 (Dropout)	(None, 9, 128)	0
flatten_14 (Flatten)	(None, 1152)	0
dense_30 (Dense)	(None, 4)	4612
<hr/>		
Total params:	51,619,644	
Trainable params:	285,444	
Non-trainable params:	51,334,200	

Como en los experimentos anteriores, no todos los parámetros de la red son entrenables y esto se debe a los *embeddings*. Los resultados de este experimento son:

**macro\_f1:** 0.413876 | **micro\_f1:** 0.553534 | **kaggle:** 0.30052

### 5.17. dropout\_LSTM\_CONV\_rnn.

Experimento generado usando la red descrita en la sección *dropout\_LSTM\_CONV\_rnn* (4.10), usando los parámetros por defecto indicados en la misma.



Si nos fijamos con detalle en esta gráfica podemos ver cierto parecido con la gráfica obtenida en el experimento *adam\_lr\_0005* (5.8), ya que la red sobreaprende pero no tanto como en otros experimentos. Esto se debe a que, al haber introducido la técnica *dropout*, hay neuronas que no aprenden en ciertas épocas, lo cual supone un aprendizaje de la red más lento, que es precisamente lo que se ha buscado en el experimento *adam\_lr\_0005* (5.8) aumentando el *learning\_rate*. Veamos ahora la arquitectura de la red:

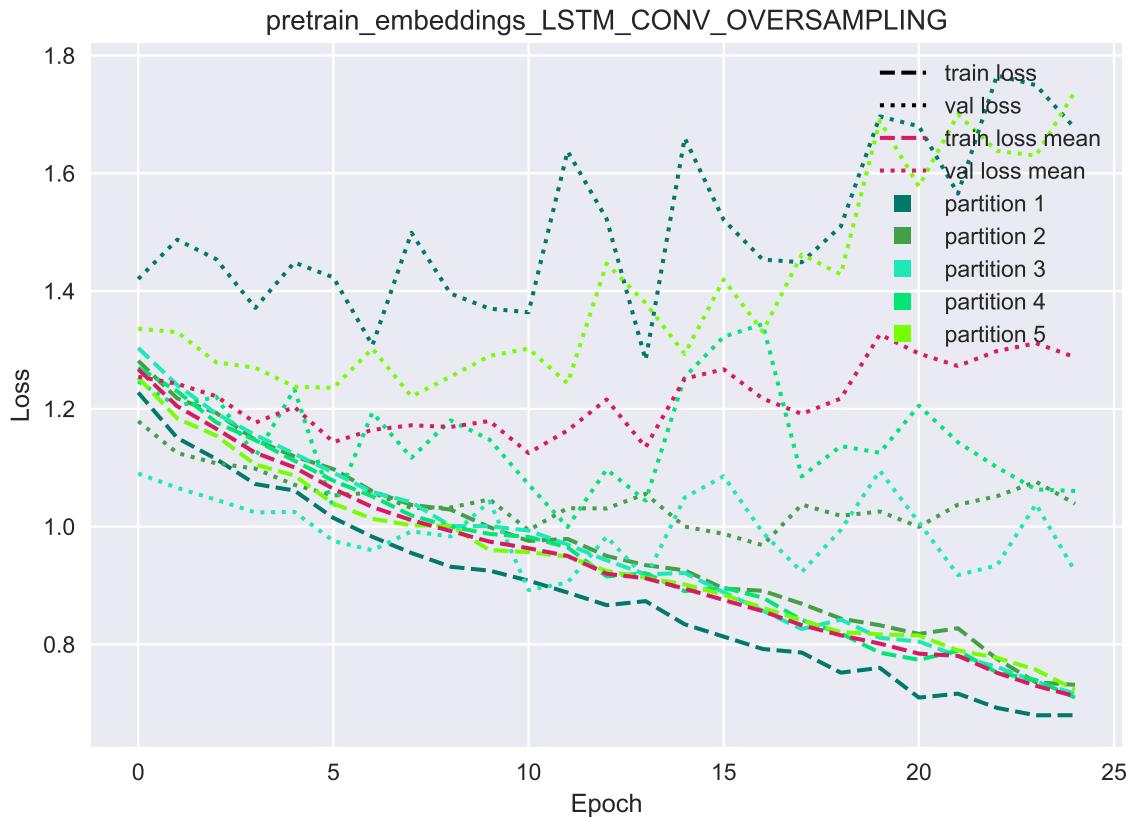
Layer (type)	Output Shape	Param #
<hr/>		
embedding_15 (Embedding)	(None, 16, 100)	51334200
lstm_25 (LSTM)	(None, 16, 64)	42240
dropout_20 (Dropout)	(None, 16, 64)	0
max_pooling1d_5 (MaxPooling1D)	(None, 8, 64)	0
conv1d_8 (Conv1D)	(None, 8, 128)	41088
dropout_21 (Dropout)	(None, 8, 128)	0
average_pooling1d_9 (AveragePooling1D)	(None, 4, 128)	0
dense_31 (Dense)	(None, 4, 128)	16512
dropout_22 (Dropout)	(None, 4, 128)	0
average_pooling1d_10 (AveragePooling1D)	(None, 2, 128)	0
flatten_15 (Flatten)	(None, 256)	0
dense_32 (Dense)	(None, 4)	1028
<hr/>		
Total params: 51,435,068		
Trainable params: 100,868		
Non-trainable params: 51,334,200		

Como en los experimentos anteriores, no todos los parámetros de la red son entrenables y esto se debe a los *embeddings*. Los resultados de este experimento son:

**macro\_f1:** 0.432329 | **micro\_f1:** 0.554804 | **kaggle:** 0.54305

### 5.18. pretrain\_embeddings\_LSTM\_CONV\_OVERSAMPLING.

Experimento generado usando la red descrita en la sección *pretrain\_embeddings\_LSTM\_CONV* (4.8), usando los parámetros por defecto indicados en la misma. La diferencia es que se ha aplicado *oversampling*, tal y como se ha explicado en la sección *Preprocesamiento* (3.2).



Al igual que sucedía en el experimento *pretrain\_embeddings\_LSTM\_CONV* (5.9), esta red sobreaprende, pero de una manera más irregular (obsérvese los valores de la función *loss* para las particiones del conjunto *validation* -las líneas de puntos-). Veamos ahora la arquitectura de la red:

Layer (type)	Output Shape	Param #
<hr/>		
embedding_1 (Embedding)	(None, 11, 100)	51334200
lstm_1 (LSTM)	(None, 11, 64)	42240
dropout_1 (Dropout)	(None, 11, 64)	0
max_pooling1d_1 (MaxPooling1D)	(None, 5, 64)	0
conv1d_1 (Conv1D)	(None, 5, 128)	41088
dropout_2 (Dropout)	(None, 5, 128)	0
average_pooling1d_1 (Average)	(None, 2, 128)	0

```

dense_1 (Dense)           (None, 2, 128)      16512
dropout_3 (Dropout)       (None, 2, 128)      0
average_pooling1d_2 (Average (None, 1, 128)    0
flatten_1 (Flatten)       (None, 128)        0
dense_2 (Dense)           (None, 4)          516
=====
Total params: 51,434,556
Trainable params: 100,356
Non-trainable params: 51,334,200

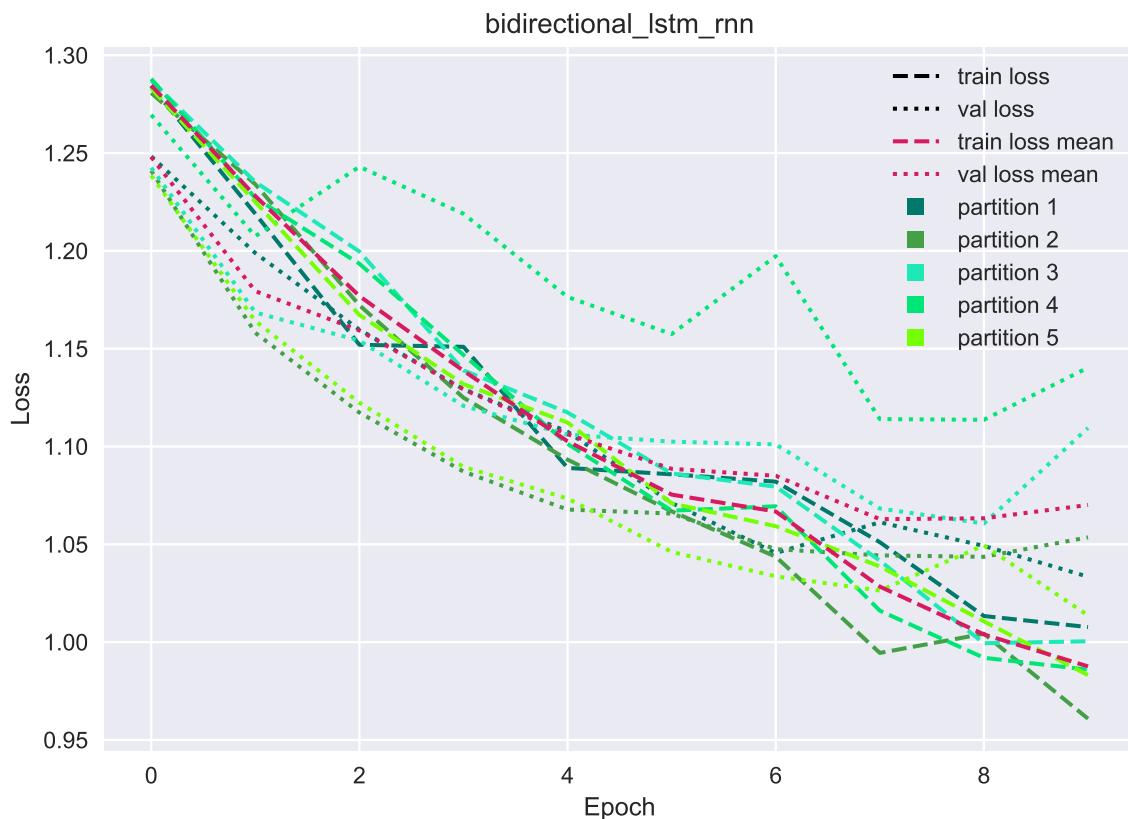
```

Como en los experimentos anteriores, no todos los parámetros de la red son entrenables y esto se debe a los *embeddings*. Los resultados de este experimento son:

**macro\_f1: 0.515203 | micro\_f1: 0.542741 | kaggle: 0.49912**

### 5.19. bidirectional\_lstm\_rnn.

Experimento generado usando la red descrita en la sección *bidirectional\_lstm\_rnn* (4.11), usando los parámetros por defecto indicados en la misma.



En este caso, los valores de la función *loss* decrecen tanto en el conjunto de *train* como en el conjunto de *validation*, lo cual es la situación deseada, ya que nos indica que la red no sobreaprende. Veamos ahora la arquitectura de la red:

Layer (type)	Output Shape	Param #
--------------	--------------	---------

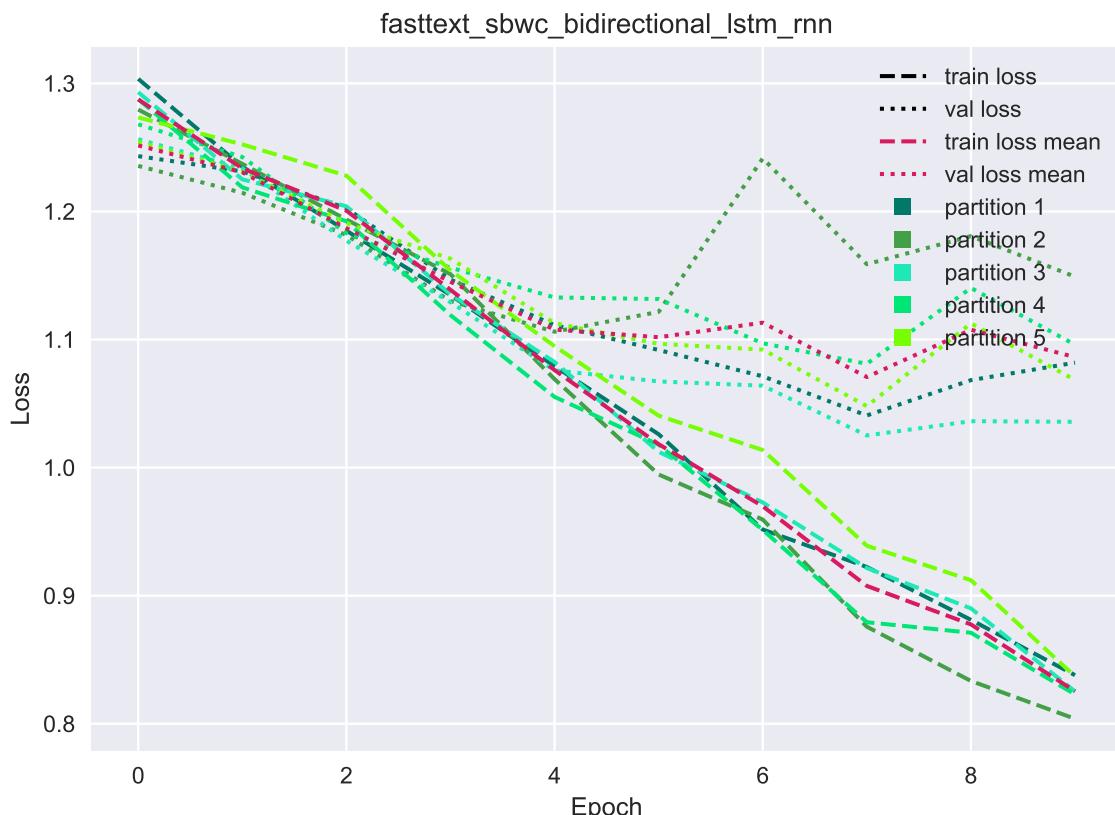
```
=====
embedding_16 (Embedding)      (None, 16, 100)      51334200
bidirectional_1 (Bidirection (None, 16, 200)      160800
global_max_pooling1d_1 (Glob (None, 200)          0
dense_33 (Dense)             (None, 100)          20100
dropout_23 (Dropout)         (None, 100)          0
dense_34 (Dense)             (None, 4)            404
=====
Total params: 51,515,504
Trainable params: 181,304
Non-trainable params: 51,334,200
```

Como en los experimentos anteriores, no todos los parámetros de la red son entrenables y esto se debe a los *embeddings*. Los resultados de este experimento son:

**macro\_f1: 0.392873 | micro\_f1: 0.561425 | kaggle: 0.57996**

## 5.20. fasttext\_sbwc\_bidirectional\_lstm\_rnn.

Este experimento es el mismo que el mostrado en la sección *bidirectional\_lstm\_rnn* (5.19) pero usando un conjunto distinto de *embeddings*. En este primer ejemplo se han usado los *embeddings* generados con *FastText* (<https://github.com/facebookresearch/fastText>) usando el *corpus Spanish Billion Word Corpus* (<http://crscardellino.github.io/SBWCE/>).



Como podemos ver, en este caso si se produce sobreaprendizaje, lo cual nos hace ver la importancia de seleccionar unos buenos *embeddings* para el problema que queremos resolver. Veamos ahora la arquitectura de la red:

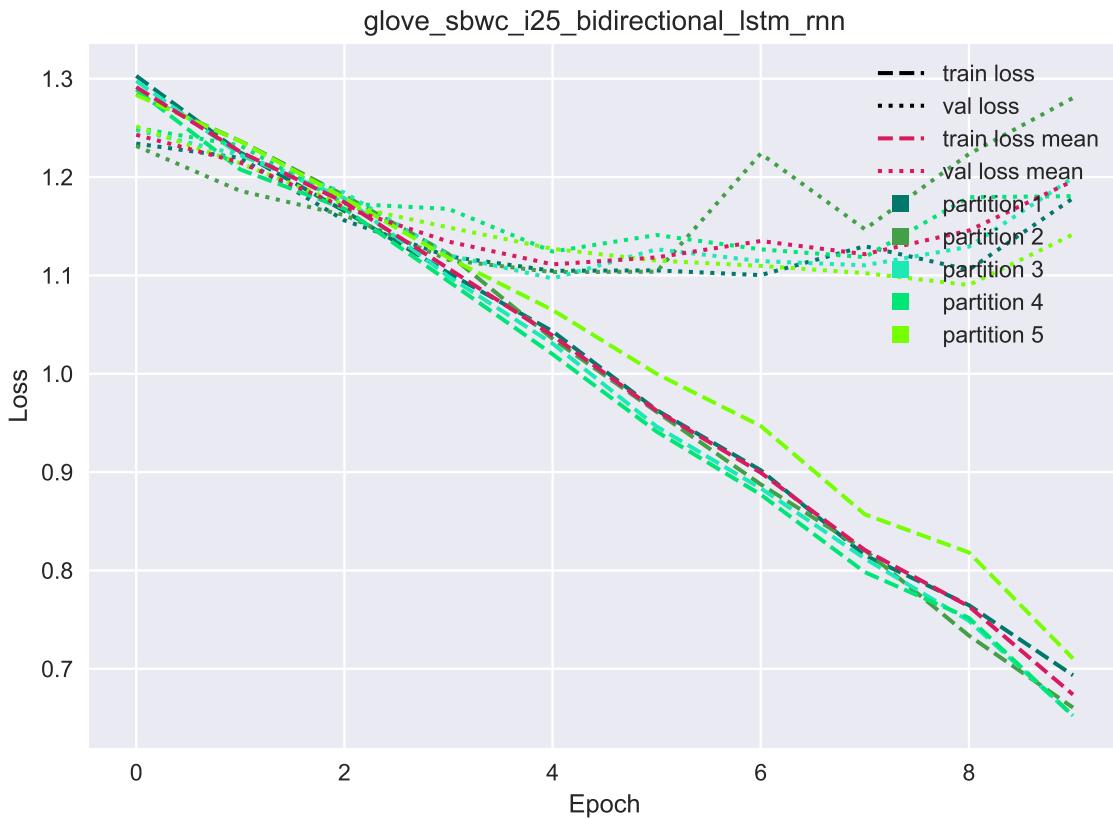
Layer (type)	Output Shape	Param #
<hr/>		
embedding_1 (Embedding)	(None, 16, 300)	256614600
bidirectional_1 (Bidirection	(None, 16, 200)	320800
global_max_pooling1d_1 (Glob	(None, 200)	0
dense_1 (Dense)	(None, 100)	20100
dropout_1 (Dropout)	(None, 100)	0
dense_2 (Dense)	(None, 4)	404
<hr/>		
Total params:	256,955,904	
Trainable params:	341,304	
Non-trainable params:	256,614,600	

Como en los experimentos anteriores, no todos los parámetros de la red son entrenables y esto se debe a los *embeddings*. Sin embargo, podemos ver que el número de parámetros no entrenables en la capa *Embedding* es mayor en comparación con el experimento original, ya que el tamaño del fichero de *embeddings* es mayor. Los resultados de este experimento son:

```
macro_f1: 0.419309 | micro_f1: 0.559381 | kaggle: 0.56414
```

## 5.21. glove\_sbwc\_i25\_bidirectional\_lstm\_rnn.

Este experimento es el mismo que el mostrado en la sección *bidirectional\_lstm\_rnn* (5.19) pero usando un conjunto distinto de *embeddings*. En este primer ejemplo se han usado los *embeddings* generados con *GloVe* (<https://github.com/stanfordnlp/GloVe>) usando el *corpus Spanish Billion Word Corpus* (<http://crscardellino.github.io/SBWCE/>).



Como podemos ver, en este caso si se produce sobreaprendizaje, lo cual nos hace ver la importancia de seleccionar unos buenos *embeddings* para el problema que queremos resolver. Veamos ahora la arquitectura de la red:

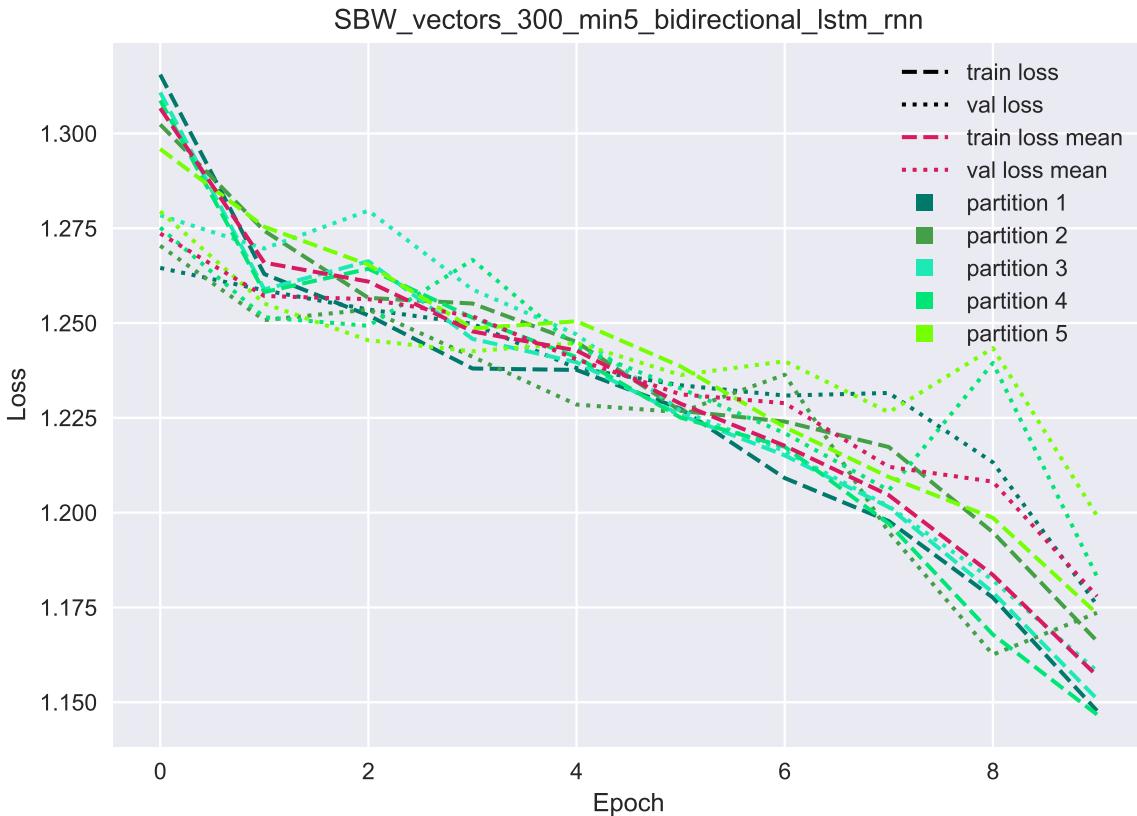
Layer (type)	Output Shape	Param #
<hr/>		
embedding_1 (Embedding)	(None, 16, 300)	256614600
bidirectional_1 (Bidirection)	(None, 16, 200)	320800
global_max_pooling1d_1 (Glob)	(None, 200)	0
dense_1 (Dense)	(None, 100)	20100
dropout_1 (Dropout)	(None, 100)	0
dense_2 (Dense)	(None, 4)	404
<hr/>		
Total params:	256,955,904	
Trainable params:	341,304	
Non-trainable params:	256,614,600	

Como en los experimentos anteriores, no todos los parámetros de la red son entrenables y esto se debe a los *embeddings*. Sin embargo, podemos ver que el número de parámetros no entrenables en la capa *Embedding* es mayor en comparación con el experimento original, ya que el tamaño del fichero de *embeddings* es mayor. Los resultados de este experimento son:

**macro\_f1: 0.435177 | micro\_f1: 0.534398 | kaggle: 0.55360**

## 5.22. SBW\_vectors\_300\_min5\_bidirectional\_lstm\_rnn.

Este experimento es el mismo que el mostrado en la sección *bidirectional\_lstm\_rnn* (5.19) pero usando un conjunto distinto de *embeddings*. En este primer ejemplo se han usado los *embeddings* generados con *Word2Vec* (<https://radimrehurek.com/gensim/models/word2vec.html>) usando el *corpus Spanish Billion Word Corpus* (<http://crscardellino.github.io/SBWCE/>).



En este caso, los valores de la función *loss* decrecen tanto en el conjunto de *train* como en el conjunto de *validation*, lo cual es la situación deseada, ya que nos indica que la red no sobreaprende. Veamos ahora la arquitectura de la red:

Layer (type)	Output Shape	Param #
<hr/>		
embedding_1 (Embedding)	(None, 16, 300)	300196500
bidirectional_1 (Bidirection)	(None, 16, 200)	320800
global_max_pooling1d_1 (Glob)	(None, 200)	0
dense_1 (Dense)	(None, 100)	20100
dropout_1 (Dropout)	(None, 100)	0
dense_2 (Dense)	(None, 4)	404
<hr/>		
Total params: 300,537,804		
Trainable params: 341,304		
Non-trainable params: 300,196,500		

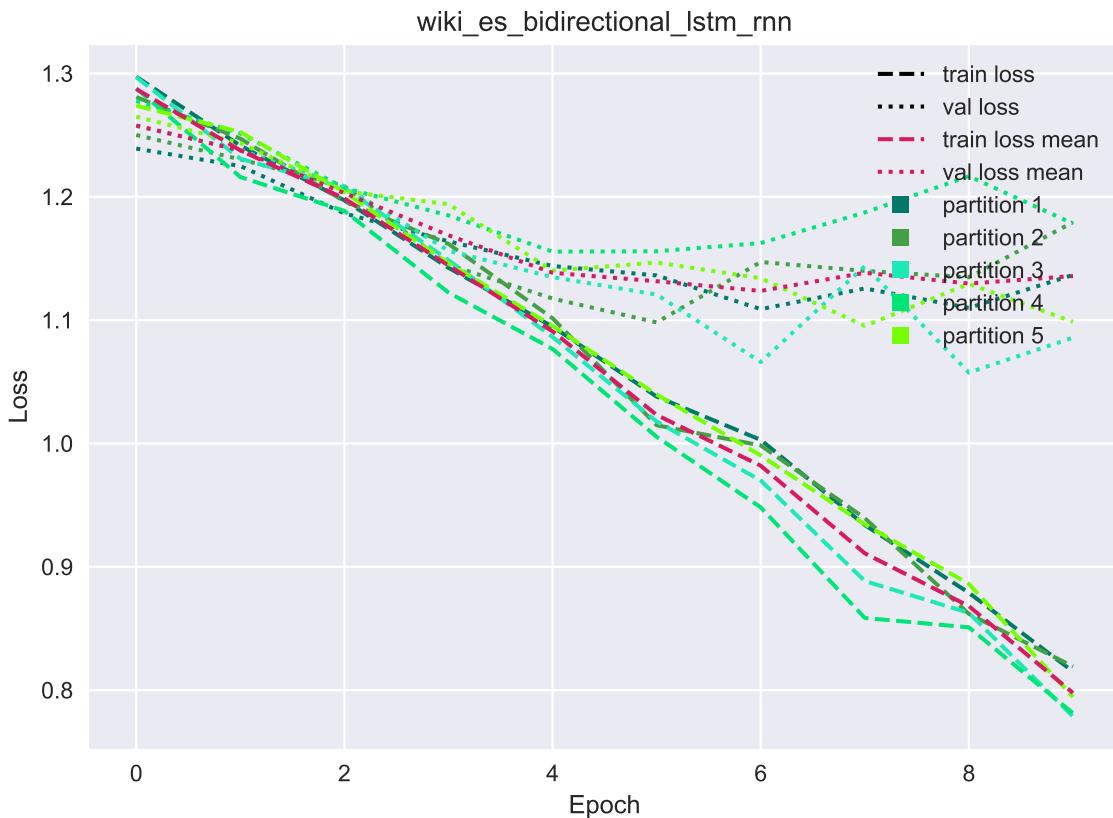
Como en los experimentos anteriores, no todos los parámetros de la red son entrenables y esto se

debe a los *embeddings*. Sin embargo, podemos ver que el número de parámetros no entrenables en la capa *Embedding* es mayor en comparación con el experimento original, ya que el tamaño del fichero de *embeddings* es mayor. Los resultados de este experimento son:

**macro\_f1:** 0.297016 | **micro\_f1:** 0.518501 | **kaggle:** 0.46572

### 5.23. wiki\_es\_bidirectional\_lstm\_rnn.

Este experimento es el mismo que el mostrado en la sección *bidirectional\_lstm\_rnn* (5.19) pero usando un conjunto distinto de *embeddings*. En este primer ejemplo se han usado los *embeddings* generados con *FastText* (<https://github.com/facebookresearch/fastText>) usando el *corpus Wikipedia Spanish Dump on January 05, 2015* (<https://archive.org/details/eswiki-20150105>).



Como podemos ver, en este caso si se produce sobreaprendizaje, lo cual nos hace ver la importancia de seleccionar unos buenos *embeddings* para el problema que queremos resolver. Veamos ahora la arquitectura de la red:

Layer (type)	Output Shape	Param #
<hr/>		
embedding_7 (Embedding)	(None, 16, 300)	295700700
bidirectional_7 (Bidirection)	(None, 16, 200)	320800
global_max_pooling1d_7 (Glob)	(None, 200)	0
dense_13 (Dense)	(None, 100)	20100
dropout_7 (Dropout)	(None, 100)	0

```

dense_14 (Dense)           (None, 4)          404
=====
Total params: 296,042,004
Trainable params: 341,304
Non-trainable params: 295,700,700

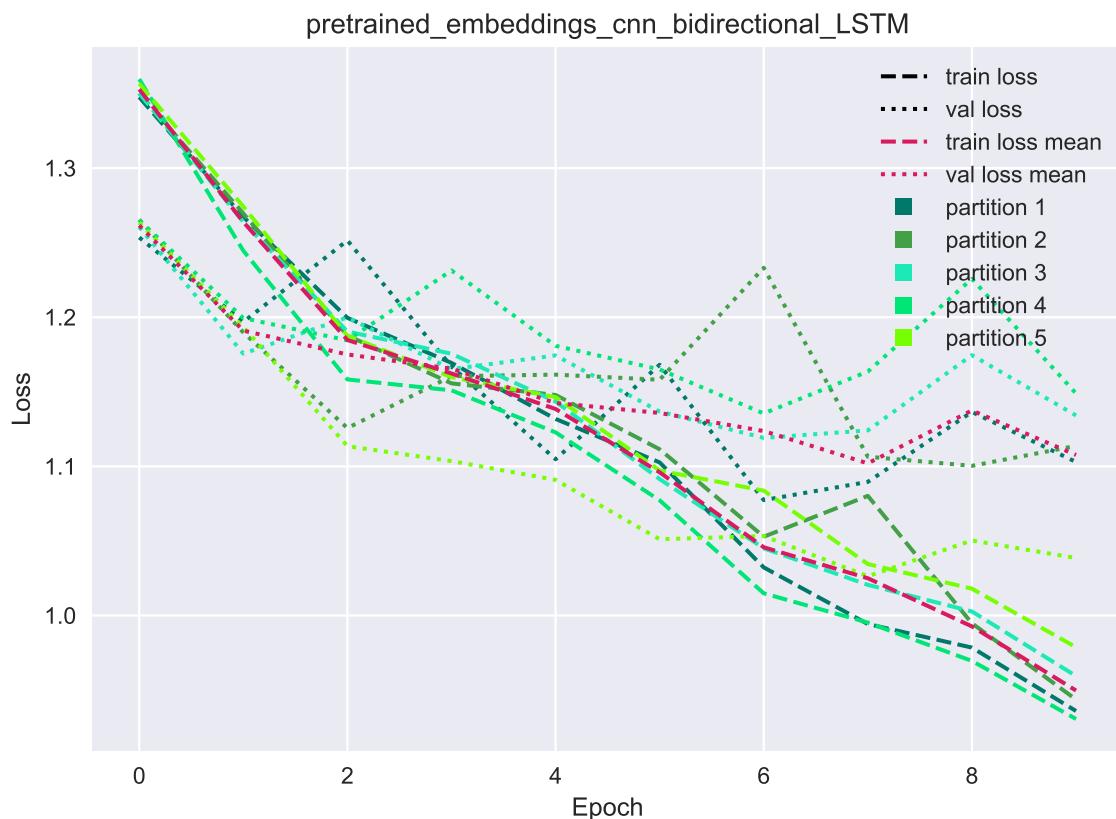
```

Como en los experimentos anteriores, no todos los parámetros de la red son entrenables y esto se debe a los *embeddings*. Sin embargo, podemos ver que el número de parámetros no entrenables en la capa *Embedding* es mayor en comparación con el experimento original, ya que el tamaño del fichero de *embeddings* es mayor. Los resultados de este experimento son:

**macro\_f1: 0.427138 | micro\_f1: 0.530431 | kaggle: 0.56414**

#### 5.24. pretrained\_embeddings\_cnn\_bidirectional\_LSTM.

Experimento generado usando la red descrita en la sección *pretrained\_embeddings\_cnn\_bidirectional\_LSTM* (4.12), usando los parámetros por defecto indicados en la misma.



Como podemos ver esta red no sobreaprende demasiado, aunque parece que lo va a hacer si la hubiésemos dejado más épocas, ya que el valor de la función *loss* empieza a separarse para el conjunto de *validation* con respecto al conjunto de *train*. Veamos ahora la arquitectura de la red:

```
***** pretrained_embeddings_cnn_bidirectional_LSTM *****
Layer (type)          Output Shape         Param #

```

```
=====
embedding_1 (Embedding)      (None, 16, 100)      51334200
conv1d_1 (Conv1D)           (None, 16, 64)       32064
dropout_1 (Dropout)         (None, 16, 64)       0
average_pooling1d_1 (Average) (None, 8, 64)       0
bidirectional_1 (Bidirection) (None, 8, 600)     876000
max_pooling1d_1 (MaxPooling1) (None, 4, 600)     0
dense_1 (Dense)             (None, 4, 100)      60100
dropout_2 (Dropout)         (None, 4, 100)      0
average_pooling1d_2 (Average) (None, 2, 100)     0
flatten_1 (Flatten)         (None, 200)        0
dense_2 (Dense)             (None, 4)          804
=====
Total params: 52,303,168
Trainable params: 968,968
Non-trainable params: 51,334,200
```

Como en los experimentos anteriores, no todos los parámetros de la red son entrenables y esto se debe a los *embeddings*. Sin embargo, podemos ver, esta es la red con el mayor número de parámetros entrenables de los experimentos realizados. Los resultados de este experimento son:

**macro\_f1:** 0.391742 | **micro\_f1:** 0.557514 | **kaggle:** 0.58347

## 6. Otros experimentos.

### 6.1. Variables calculadas.

Después de haber probado el enfoque conocido para el análisis de texto y que se basa en la conversión de las palabras de un texto en números utilizando maneras diferentes como vimos en nuestros experimentos ya explicados antes, hemos intentado transformar el conjunto de *tuits* en un dataset que tiene un concepto y arquitectura totalmente diferente al original. Para ello hemos estudiado las características que creemos que pueden representar a un *tuit*, en nuestro caso estas características son:

- Longitud de texto.
- Número de palabras que refieren a duda o interrogación.
- Número de *hashtags*.
- Número de palabras con letras repetidas más de 2 veces.
- Si el texto tiene o no signos de interrogación y/o signos de exclamación.
- Número de palabras que refieren a insultos o sentimientos negativos. Para ello, hemos definido una lista de los mismos.
- Cálculo de un factor de negatividad que se relaciona con el factor comentado en el punto anterior y el número total de palabras del texto.
- Número de palabras que refieren a sentimiento positivo.
- Cálculo de un factor de positividad que se relaciona con el factor comentado en el punto anterior y el número total de palabras del texto.

Pero al contrario de lo esperado, éste modelo de dataset no nos guiaba a modelos potentes, sino a modelos con una precisión muy baja en comparación a los obtenidos anteriormente.

## 6.2. GRU.

Los experimentos que se han hecho con *GRU*, no daban una mejora sobre *LSTM* y esto se puede entender por la demostración de *Gail Weiss & Yoav Goldberg & Eran Yahav* (<https://arxiv.org/abs/1805.04908>), la cual indica que *LSTM* es “estrictamente más fuerte” que *GRU*, ya que puede realizar fácilmente el conteo ilimitado, mientras que *GRU* no puede. Es por eso que *GRU* no aprende idiomas simples que se pueden aprender con *LSTM*.

La unidad *GRU* controla el flujo de información como la unidad *LSTM*, pero sin tener que usar una unidad de memoria. Simplemente expone el contenido oculto completo sin ningún control. *GRU* es relativamente nuevo, y la única ventaja que puede tener sobre *LSTM* es en tema de eficiencia computacional.

## 7. Resultados.

A continuación, mostramos una tabla resumen de todos los resultados mostrados en esta memoria. Dicha tabla tiene los modelos ordenados en base a su puntuación obtenida en *Kaggle*. Los tres modelos seleccionados para la entrega han sido los tres primeros de esta tabla.

		Número de parámetros						¿Entrenables?
		macro	f1	micro	f1	kaggle	Total	
	pretrained_embeddings_cnn_bidirectional_LSTM	0.391742	0.557514	0.58347	0.52303	168	51,334,200	968,968
	bidirectional_lstm_rnn_results	0.392873	0.561425	0.57996	51,515,504	51,334,200	181,304	
	fasttext_sbwc_bidirectional_lstm_rnn	0.419309	0.559381	0.56414	256,955,904	256,614,600	341,304	
	wiki_es_bidirectional_lstm_rnn	0.427138	0.530431	0.56414	296,042,004	295,700,700	341,304	
	glove_sbwc_i25_bidirectional_lstm_rnn	0.435177	0.534398	0.5536	256,955,904	256,614,600	341,304	
	dropout_LSTM_CONV_rnn	0.432329	0.554804	0.54305	51,334,200	51,435,068	100,868	
	stacked_lstm_rnn	0.434354	0.531007	0.52372	51,475,772	51,334,200	141,572	
	pretrain_embeddings_LSTM_CONV	0.443507	0.526316	0.51845	51,435,068	51,334,200	100,868	
	pretrain_embeddings_LSTM_CONV	0.418226	0.480790	0.50790	51,434,556	51,334,200	100,356	
	preprocess_pretrain_embeddings_rnn	0.416798	0.505270	0.50615	51,379,676	51,334,200	45,476	
58	adam_lr_0005_rnn	0.424711	0.526395	0.50439	51,380,572	51,334,200	46,372	
	pretrain_embeddings_LSTM_CONV_OVERAMPLING	0.515203	0.542741	0.49912	51,434,556	51,334,200	100,356	
	epoch100_pretrain_embeddings_rnn	0.430359	0.516541	0.49560	51,380,572	51,334,200	46,372	
	sigmoid_pretrain_embeddings_rnn	0.408407	0.506601	0.49560	51,380,572	51,334,200	46,372	
	pretrain_embeddings_rnn	0.417279	0.507906	0.49384	51,380,572	51,334,200	46,372	
	SBW_vectors_300_min5_bidirectional_lstm_rnn	0.297016	0.518501	0.46572	300,537,804	300,196,500	341,304	
	calculated_embeddings_rnn	0.283373	0.433367	0.45342	499,000	547,804	48,804	
	preprocess_tfidf_rnn	0.148071	0.420742	0.42179	21,540	0	21,540	
	tfidf_rnn	0.159393	0.414133	0.42179	22,692	0	22,692	
	preprocess_calculated_embeddings_rnn	0.256807	0.372415	0.41827	426,780	379,000	47,780	
	preprocess_calculated_embeddings_LSTM_CONV	0.260481	0.450435	0.36731	480,380	379,000	101,380	
	epoch50_preprocess_calculated_embeddings_LSTM_CONV	0.297376	0.418076	0.36203	480,380	379,000	101,380	
	big_LSTM_CONV_rnn	0.413876	0.553534	0.30052	51,619,644	51,334,200	285,444	
	adadelta_rnn	0.100923	0.262983	—	51,380,572	51,334,200	46,372	

## 8. Bibliografía.

- Keras: <https://keras.io/>
- Tensorflow: [https://www.tensorflow.org/api\\_docs/python/tf](https://www.tensorflow.org/api_docs/python/tf)
- Stackoverflow: <https://stackoverflow.com/>
- FastText: <https://github.com/facebookresearch/fastText>
- Corpus: Spanish Billion Word Corpus: <http://crscardellino.github.io/SBWCE/>
- GloVe: <https://github.com/stanfordnlp/GloVe>
- Word2Vec: <https://radimrehurek.com/gensim/models/word2vec.html>)
- Corpus: Wikipedia Spanish Dump on January 05, 2015 <https://archive.org/details/eswiki-20150105>
- Modelo LSTM-CONV: Pedro M Sosa, Twitter Sentiment Analysis using combined LSTM-CNN Models
- GRU: Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, Yoshua Bengio. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling.