

Natural Language Processing in TensorFlow



DeepLearning.AI

Víctor Díaz Bustos

12 de junio de 2023

Índice

1. Introducción	2
2. Sentimiento en textos	3
2.1. Uso de APIs	3
2.2. Texto a secuencia	5
2.3. Padding	7
2.4. Sarcasmo	8
3. Incrustaciones de palabras	10
3.1. Conjunto de datos IMDB	11
3.2. Vectores	11
3.3. Función de pérdida	15
3.4. Sub-palabras preentrenadas	16
4. Modelos de secuencia	19
4.1. LSTM	19
4.1.1. Implementar LSTM	20
4.1.2. LSTM vs. non-LSTM	23
4.2. Otros tipos de RNN	25
4.2.1. GRU	25
4.2.2. Diferencias entre LSTM y GRU	25
4.2.3. Convolución	26
4.3. Evitar sobreajuste	27
5. Modelos de secuencia y literatura	29
5.1. Preparar el entrenamiento	29
5.2. Construir NN	32
5.3. Predecir una palabra	34
6. Resumen	36

1. Introducción

En este curso, aprenderás a usar TensorFlow para procesar textos para el procesamiento de lenguaje natural. A diferencia de las imágenes, el texto es más desordenado, hay frases largas y frases cortas. Así que en este curso, aprendes a lidiar con todo eso.

Vamos a echar un vistazo a lo que se necesita para procesar texto porque las redes neuronales generalmente tratan con números. Las funciones cero, el cálculo de pesos y sesgos, todos son números. Entonces, ¿cómo vamos a convertir nuestros textos en números de una manera sensata? Dada una palabra, ¿cómo se convierte eso en un conjunto de números que se pueden alimentar en la red neuronal? Frases de longitud múltiple también. ¿Cómo lidiamos con rellenarlos? O si tienes como un conjunto de palabras que usas para entrenar, y luego tienes otro conjunto de palabras que realmente quieres predecir, pero vas a tener algunas palabras en este conjunto que no están en ese. ¿Cómo lidias con los tokens de vocabulario y ese tipo de cosas?

En este curso vamos a volver a construir modelos, pero nos centraremos en el texto y cómo construir el clasificador se basa en modelos de texto. Empezaremos por ver el sentimiento en el texto y aprenderemos a crear modelos que entiendan el texto que está entrenado en el texto etiquetado y, a continuación, podemos clasificar el texto nuevo en función de lo que hayan visto.

Cuando tratábamos con imágenes, era relativamente fácil para nosotros alimentarlas en una red neuronal, ya que los valores de píxel ya eran números. Y la red podría aprender parámetros de funciones que podrían usarse para ajustar clases a etiquetas. Pero, ¿qué pasa con el texto? ¿Cómo podemos hacer eso con frases y palabras?

2. Sentimiento en textos

Podríamos tomar codificaciones de caracteres para cada carácter de un conjunto. Por ejemplo, los valores ASCII. ¿Pero eso nos ayudará a entender el significado de una palabra? Entrenar una red neuronal con sólo las letras podría ser una tarea desalentadora. Entonces, ¿qué tal si tenemos en cuenta las palabras? ¿Qué pasaría si pudiéramos dar un valor a las palabras y utilizar esos valores en la formación de una red? Ahora podríamos estar llegando a alguna parte.

No importa el valor que sea, es solo que tenemos un valor por palabra, y el valor es el mismo para la misma palabra cada vez.

Esto es al menos un comienzo y cómo podemos empezar a entrenar una red neuronal basada en palabras. Afortunadamente, TensorFlow y Keras nos dan algunas API que hacen que sea muy sencillo hacer esto.

2.1. Uso de APIs

```
1 import tensorflow as tf
2 from tensorflow import keras
3 from tensorflow.keras.preprocessing.text import Tokenizer
4
5 sentences = [
6     'I love my dog',
7     'I love my cat'
8 ]
9
10 tokenizer = Tokenizer(num_words=100)
11 tokenizer.fit_on_texts(sentences)
12 word_index = tokenizer.word_index
13 print(word_index)
14
15 # {'i':1, 'my':3, 'dog':4, 'cat':5, 'love':2}
```

Aquí está el código para codificar dos frases. Vamos a analizarlo línea por línea:

[3] TensorFlow y Keras nos dan varias maneras de codificar palabras, pero en la que me voy a centrar es el **tokenizer**. Esto manejará el trabajo pesado para nosotros, generando el diccionario de codificaciones de palabras y creando vectores fuera de las oraciones.

[5] Pondré las **oraciones** en una lista.

[10] Creo una **instancia** del tokenizer. En este caso, estoy usando 100 tokens, que es demasiado grande, ya que solo hay cinco palabras distintas en estos datos. Si estás creando un conjunto de entrenamiento basado en un montón de texto, normalmente no sabes cuántas palabras únicas hay en ese texto. Entonces, al establecer este hiperparámetro, lo que hará el tokenizador es

tomar las 100 palabras principales por volumen y simplemente codificarlas. Es un método abreviado práctico cuando se trata de muchos datos, y vale la pena experimentar cuando se entrena con datos reales más adelante en este curso. A veces, el impacto de menos palabras puede ser mínimo en precisión de entrenamiento, pero enorme en tiempo de entrenamiento, pero utilízalo con cuidado.

- [11] El método de ajuste en los textos del tokenizador toma los datos y los codifica.
- [12] El tokenizer proporciona una propiedad de índice de palabras que devuelve un diccionario que contiene pares de valores clave, donde la clave es la palabra, y el valor es el token de esa palabra, que puede inspeccionar simplemente imprimiéndola.

Recuerda la palabra que estaba en mayúsculas ('I'), tenga en cuenta que está en minúsculas aquí. Eso es otra cosa que el tokenizador hace por ti. Elimina la puntuación. Esto es realmente útil si considera este caso.

```
1 sentences = [  
2     'I love my dog',  
3     'I love my cat',  
4     'You love my dog!'  
5 ]
```

Aquí, he añadido otra frase, pero hay algo muy diferente en ella. He añadido una exclamación. Ahora, ¿debería tratarse esto como una palabra diferente a sólo 'dog'? Por supuesto que no. Así que los resultados del código que vimos anteriormente con este nuevo corpus de datos, se verán así.

```
1 # {'i':3, 'my':2, 'you'=6, 'love'=1, 'cat':5, 'dog':4}
```

Observa que todavía sólo tenemos 'dog' como clave. Que la exclamación no afectó a esto, y por supuesto, tenemos una nueva clave para la palabra 'you' que fue detectada.

El valor de *num_words* tiene un impacto en la secuencia generada por el Tokenizer, pero no en la generación del diccionario *word_index*. La idea detrás de limitar el número de palabras es que muchas veces los modelos de NLP funcionan mejor con un número reducido de tokens y además puede ayudar a reducir la complejidad computacional del modelo.

Cuando se usa el método *texts_to_sequences()* de la clase Tokenizer, se genera una secuencia de enteros para cada oración en el corpus. Cada entero representa una palabra en el diccionario *word_index*. Si se especifica un valor para *num_words*, se incluirán solo las *num_words* palabras más frecuentes en el corpus, y las palabras menos frecuentes se asignarán a un token especial (0 por defecto) que representa "palabra desconocida".

Por lo tanto, en este caso, el valor de `num_words` limita la cantidad de palabras diferentes que se incluyen en la secuencia generada, pero no afecta la generación del diccionario `word_index`.

2.2. Texto a secuencia

El siguiente paso será convertir sus oraciones en listas de valores basadas en estos tokens. Una vez que los tengas, es probable que también necesites manipular estas listas, no menos importante para hacer que cada frase tenga la misma longitud, de lo contrario, puede ser difícil entrenar una red neuronal con ellas.

Recuerde que cuando estábamos haciendo imágenes, definimos una capa de entrada con el tamaño de la imagen que estamos alimentando en la red neuronal. En los casos en que las imágenes tenían un tamaño diferente, las cambiaríamos para que se ajustaran. Bueno, vas a enfrentarte a lo mismo con el texto.

Afortunadamente, TensorFlow incluye una API para manejar estos problemas. Comencemos con la creación de una lista de secuencias, las oraciones codificadas con los tokens que generamos:

```
1 import tensorflow as tf
2 from tensorflow import keras
3 from tensorflow.keras.preprocessing.text import Tokenizer
4
5 sentences = [
6     'I love my dog',
7     'I love my cat',
8     'You love my dog!',
9     'Do you think my dog is amazing?'
10 ]
11
12 tokenizer = Tokenizer(num_words=100)
13 tokenizer.fit_on_texts(sentences)
14 word_index = tokenizer.word_index
15
16 sequences = tokenizer.texts_to_sequences(sentences)
17
18 print(word_index)
19 print(sequences)
```

[5] Se ha añadido **otra frase** al final de la lista de oraciones. Tenga en cuenta que todas las oraciones anteriores tenían cuatro palabras en ellas. Así que este es un poco más largo. Usaremos eso para demostrar el relleno en un momento.

[16] Se llama al tokenizador para obtener textos a secuencias, y los convertirá en un conjunto de secuencias para mí.

Una cosa muy útil sobre esto que usará más adelante es el hecho de que el texto a las secuencias llamadas puede tomar cualquier conjunto de oraciones,

por lo que puede codificarlas en función del conjunto de palabras que aprendió del que se pasó en forma en los textos. Esto es muy significativo si piensas un poco más adelante. Si entrenas una red neuronal en un corpus de textos, y el texto tiene un índice de palabras generado a partir de ella, entonces cuando quieras hacer inferencia con el modelo de entrenamiento, tendrás que codificar el texto que quieres inferir con el mismo índice de palabras, de lo contrario no tendría sentido.

```
1 test_data = [  
2     'i really love my dog',  
3     'my dog loves my manatee'  
4 ]  
5  
6 test_seq = tokenizer.texts_to_sequences(test_data)  
7 print(test_seq)  
8  
9 # [[4,2,1,3], [1,3,1]]
```

Entonces, si considera este código, ¿cuál espera que sea el resultado? Hay algunas palabras familiares aquí, como 'love', 'my' y 'dog', pero también algunas no vistas anteriormente. Si ejecuto este código, esto es lo que obtendría. Así que 'i really love my dog' todavía estaría codificado como [4, 2, 1, 3], que es 'I love my dog', perdiendo 'really' ya que la palabra no está en el índice de palabras, y 'my dog loves my manatee' se codificaría en [1, 3, 1], que es sólo 'my dog my'.

Realmente necesitamos una gran cantidad de datos de entrenamiento para obtener un vocabulario amplio. En muchos casos, es una buena idea, en lugar de ignorar palabras invisibles, poner un valor especial cuando se encuentra una palabra invisible. Puede hacer esto con una propiedad en el tokenizer.

```
1 from tensorflow.keras.preprocessing.text import Tokenizer  
2  
3 sentences = [  
4     'I love my dog',  
5     'I love my cat',  
6     'You love my dog!',  
7     'Do you think my dog is amazing?'  
8 ]  
9  
10 tokenizer = Tokenizer(num_words=100, oov_token="<OOV>")  
11 tokenizer.fit_on_texts(sentences)  
12 word_index = tokenizer.word_index  
13  
14 test_data = [  
15     'i really love my dog',  
16     'my dog loves my manatee'  
17 ]  
18  
19 test_seq = tokenizer.texts_to_sequences(test_data)  
20 print(test_seq)
```

Aquí está el código completo que muestra tanto las oraciones originales como los datos de prueba. Lo que he cambiado es agregar un token "<OOV>" (Out Of Vocabulary) de propiedad al constructor tokenizer. Ahora puede ver que he especificado que quiero que el token "<OOV>" para el vocabulario externo se use para palabras que no están en el índice de palabras. Puedes usar lo que quieras aquí, pero recuerda que debe ser algo **único y distinto** que no se confunda con una palabra real. Así que ahora, si ejecuto este código, conseguiré que mis secuencias de prueba se vean así.

```
1 # [[5,1,3,2,4], [2,4,1,2,1]]
2
3 {'think':9, 'amazing':11, 'dog':4, 'do':8, 'i':5, 'cat':7, 'you':6,
  'love':3, '<OOV>':1, 'my':2, 'is':10}
```

La primera frase será 'i <OOV> love my dog'. El segundo será 'my dog <OOV> my <OOV>'. Todavía no sintácticamente bien, pero lo está haciendo mejor. A medida que el corpus crece y más palabras están en el índice, es de esperar que las oraciones nunca vistas tengan una mejor cobertura.

Cuando estábamos construyendo redes neuronales para manejar imágenes, cuando los alimentamos en la red para el entrenamiento, necesitábamos que fueran uniformes en tamaño. A menudo, usamos los generadores para cambiar el tamaño de la imagen para que se ajuste. Con los textos te enfrentarás a un requisito similar antes de poder entrenar con textos, necesitábamos tener algún nivel de uniformidad de tamaño, por lo que el relleno (*padding*) es tu amigo ahí.

2.3. Padding

```
1 from tensorflow.keras.preprocessing.text import Tokenizer
2 from tensorflow.keras.preprocessing.sequence import pad_sequences
3
4 sentences = [
5     'I love my dog',
6     'I love my cat',
7     'You love my dog!',
8     'Do you think my dog is amazing?'
9 ]
10
11 tokenizer = Tokenizer(num_words=100, oov_token="<OOV>")
12 tokenizer.fit_on_texts(sentences)
13 word_index = tokenizer.word_index
14
15 sequences = tokenizer.texts_to_sequences(sentences)
16 padded = pad_sequences(sequences)
17
18 print(word_index)
19 print(sequences)
20 print(padded)
```


[2] Para usar las funciones de relleno, tendrá que importar `'pad_sequences'`.

[16] Una vez que el tokenizer ha creado las secuencias, estas secuencias se pueden pasar a `pad_sequences` para tenerlas acolchadas así.

```
{'do': 8, 'you': 6, 'love': 3, 'i': 5, 'amazing': 11, 'my': 2, 'is': 10, 'think': 9, 'dog': 4, '<OOV>': 1, 'cat': 7}

[[5, 3, 2, 4], [5, 3, 2, 7], [6, 3, 2, 4], [8, 6, 9, 2, 4, 10, 11]]

[[ 0 0 0 5 3 2 4]
 [ 0 0 0 5 3 2 7]
 [ 0 0 0 6 3 2 4]
 [ 8 6 9 2 4 10 11]]
```

Ahora puede ver que la lista de oraciones se ha rellenado en una matriz y que cada fila de la matriz tiene la misma longitud. Esto se logró colocando el número apropiado de 0's antes de la oración. A menudo verá ejemplos en los que el relleno está después de la oración y no antes como acabas de ver. Esto podemos configurarlo:

```
1 padded = pad_sequences(sequences, padding='post')
```

Es posible que haya notado que el ancho de la matriz era el mismo que la frase más larga. Pero puede anular eso con el parámetro `'maxlen'`.

```
1 padded = pad_sequences(sequences, padding='post', maxlen=5)
```

Esto, por supuesto, llevará a la pregunta. Si tengo oraciones más largas que la longitud máxima, entonces **perderé información**, pero de dónde. Al igual que con el relleno, el valor predeterminado es `pre`, lo que significa que perderás desde el principio de la oración. Si desea anular esto para que pierda desde el final en su lugar, puede hacerlo con el parámetro de truncamiento.

```
1 padded = pad_sequences(sequences, padding='post', maxlen=5,
    truncating='post')
```

2.4. Sarcasmo

Si descargas los datos de ese sitio de Kaggle, verás un conjunto de entradas de lista con pares nombre-valor donde el nombre es enlace del artículo, título e `'is_sarcástico'`.

```

1 from tensorflow.keras.preprocessing.text import Tokenizer
2 from tensorflow.keras.preprocessing.sequence import pad_sequences
3
4 tokenizer = Tokenizer(oov_token="<OOV>")
5 tokenizer.fit_on_texts(sentences)
6 word_index = tokenizer.word_index
7
8 sequences = tokenizer.texts_to_sequences(sentences)
9 padded = pad_sequences(sequences, padding='post')
10
11 print(padded[0])
12 # Figura 1
13
14 print(padded.shape)
15 # (26709, 40)

```

```

[ 308 15115  679 3337 2298  48 382 2576 15116  6 2577 8434
  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0]

```

Figura 1: padded[0]

Este código es muy similar a lo que hemos visto hasta ahora.

- [5] Acabamos de crear frases desde los titulares, en el conjunto de datos del sarcasmo. Entonces, al llamar a **tokenizer.fit_on_texts**, generará el índice de palabras e inicializaremos el tokenizer.
- [8] y [9] Ahora crearemos las secuencias a partir del texto, así como rellenarlas.
- [12] Podemos ver que ha sido codificado con los valores de las claves que son la palabra correspondiente en la frase.
- [15] Este es el tamaño de la matriz acolchada. Teníamos 26.709 frases, y estaban codificadas con relleno, para conseguir hasta 40 palabras de largo que era la longitud de la palabra más larga. Podemos truncar esto si queremos.

3. Incrustaciones de palabras

En esta sección, aprenderás a usar la capa Keras en TensorFlow para implementar incrustaciones de palabras (*'embedding'*), que es una de las ideas más importantes en Procesamiento de Lenguaje Natural.

Si tenemos un tamaño de vocabulario de, por ejemplo, 10.000 palabras, en lugar de usar los números del 1-10.000 para representar estas palabras, ¿podemos tener una mejor manera de representar esos números?

Es realmente interesante cómo funcionan las incrustaciones para representar la semántica de una palabra. Así que ahora en lugar de que la palabra sea solo un número, es como un vector en el espacio n-dimensional.

Así, por ejemplo, la palabra 'perro', podría ser un vector apuntando en una dirección particular y luego la palabra 'canino', podría aprenderse como un vector apuntando en una dirección muy similar, y sabemos que tienen un significado semántico muy similar fuera de eso.

Puedes descargar la incrustación de palabras preentrenada que tal vez alguien más haya entrenado.

Esto le da a su algoritmo de aprendizaje una pista sobre el significado de estas palabras. Así que cuando ves otra palabra de vocabulario, tal vez veas 'canino' por primera vez, es como si pudieras hacerle saber al algoritmo que 'canino' significa algo un relacionado con perro, incluso si nunca has visto la palabra 'canino' en tu entrenamiento específico. Las inserciones de palabras han demostrado ser una de las ideas más poderosas y útiles para ayudar a los equipos a obtener un buen rendimiento.

En la sección anterior, vimos como tokenizar palabras y convertir frases en un vector de números y luego convertir las cadenas en matrices de números. Este es el comienzo de **sacar el sentimiento** de tus oraciones. Pero ahora mismo, sigue siendo sólo una cadena de números que representan palabras. Así que a partir de ahí, ¿cómo se sentiría uno realmente?

Eso es algo que se puede aprender de un conjunto de palabras de la misma manera que las características fueron extraídas de las imágenes. Este proceso se denomina **incrustación** - *embedding* -, con la idea de que las palabras asociadas se agrupan como vectores en un espacio multidimensional.

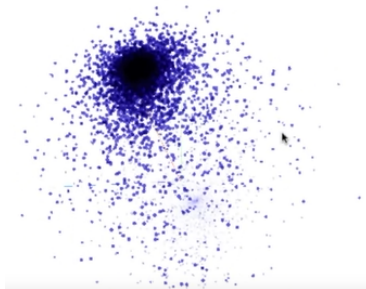


Figura 2: Embedding

3.1. Conjunto de datos IMDB

Parte de la visión de TensorFlow para que el aprendizaje automático y el aprendizaje profundo sean más fáciles de aprender y de usar, es el concepto de tener conjuntos de datos integrados. Hay una biblioteca llamada '*TensorFlow Data Services*' o *TFTS* para abreviar, y que contiene muchos conjuntos de datos y muchas categorías diferentes. Este conjunto de datos es ideal porque contiene un gran cuerpo de textos, 50.000 reseñas de películas que se clasifican como positivas o negativas.

```
1 vocab_size = 10000
2 embedding_dim = 16
3 max_length = 120
4 trunc_type = 'post'
5 oov_tok = "<OOV>"
6
7 from tensorflow.keras.preprocessing.text import Tokenizer
8 from tensorflow.keras.preprocessing.sequence import pad_sequences
9
10 tokenizer = Tokenizer(num_words=vocab_size, oov_token=oov_tok)
11 tokenizer.fit_on_texts(training_sentences)
12 word_index = tokenizer.word_index
13 sequences = tokenizer.texts_to_sequences(training_sentences)
14 padded = pad_sequences(sequences, maxlen=max_length, truncating=
    trunc_type)
15
16 testing_sequences = tokenizer.texts_to_sequences(testing_sentences)
17 testing_padded = pad_sequences(testing_sequences, maxlen=max_length
    )
```

3.2. Vectores

¿Qué pasaría si pudieras elegir un vector en un espacio de dimensiones, y las palabras que se encuentran juntas reciben vectores similares? Con el tiempo, las palabras pueden comenzar a agruparse. El significado de las palabras puede provenir del etiquetado del conjunto de datos. A medida que la red neuronal se entrena, puede aprender estos vectores asociándolos con las etiquetas para crear lo que se llama embedding, es decir, los vectores de cada palabra con su sentimiento asociado.

```
1 model = tf.keras.Sequential([
2     tf.keras.layers.Embedding(vocab_size, embedding_dim,
3         input_length=max_length),
4     tf.keras.layers.Flatten(),
5     tf.keras.layers.Dense(6, activation='relu'),
6     tf.keras.layers.Dense(1, activation='sigmoid')
7 ])
```

- [2] Los resultados de la incrustación serán una matriz 2D con la longitud de la frase y la dimensión de embedding.
- [3] Así que necesitamos aplanarlo de la misma manera que necesitábamos para aplanar nuestras imágenes.
- [4] y [5] Luego alimentamos eso en una red neuronal densa para hacer la clasificación.

Layer (type)	Output Shape	Param #
embedding_9 (Embedding)	(None, 120, 16)	160000
flatten_3 (Flatten)	(None, 1920)	0
dense_14 (Dense)	(None, 6)	11526
dense_15 (Dense)	(None, 1)	7
Total params: 171,533		
Trainable params: 171,533		
Non-trainable params: 0		

A menudo, en el procesamiento de lenguaje natural, se utiliza un tipo de capa diferente al de un aplanado, y este es un promedio global de agrupación 1D. La razón de esto es el tamaño del vector de salida que se alimenta en la danza.

```

1 model = tf.keras.Sequential([
2     tf.keras.layers.Embedding(vocab_size, embedding_dim,
3                               input_length=max_length),
4     tf.keras.layers.GlobalAveragePooling1D(),
5     tf.keras.layers.Dense(6, activation='relu'),
6     tf.keras.layers.Dense(1, activation='sigmoid')
7 ])

```

- [3] Alternativamente, puede usar una agrupación media global 1D, que promedia a través del vector para aplanarlo.

Layer (type)	Output Shape	Param #
embedding_11 (Embedding)	(None, 120, 16)	160000
global_average_pooling1d_3 (GlobalAveragePooling1D)	(None, 16)	0
dense_16 (Dense)	(None, 6)	102
dense_17 (Dense)	(None, 1)	7
Total params: 160,109		
Trainable params: 160,109		
Non-trainable params: 0		

Tras más de 10 épocas con la **GlobalAveragePooling1D()**, obtuve una precisión de 0.9664 en el entrenamiento y 0.8187 en la prueba, tomando alrededor de 6.2 segundos por época. Con **Flatten()**, mi precisión fue 1.0 y mi validación de aproximadamente 0.83 tomando alrededor de 6.5 segundos por época. Así que fue un poco más lento, pero un poco más preciso.

Hay una buena posibilidad de que estemos sobreajustando. Vamos a ver algunas estrategias para evitar esto más adelante, pero usted debe esperar resultados un poco como este.

Ahora visualizamos el embedding. Comenzaremos por obtener los resultados de la capa de incrustaciones, que es la capa cero.

```

1 e = model.layers[0]
2 weights = e.get_weights()[0]
3 print(weights.shape)
4 # (10000, 16)
5 # (<vocab_size>, <embedding_dim>)

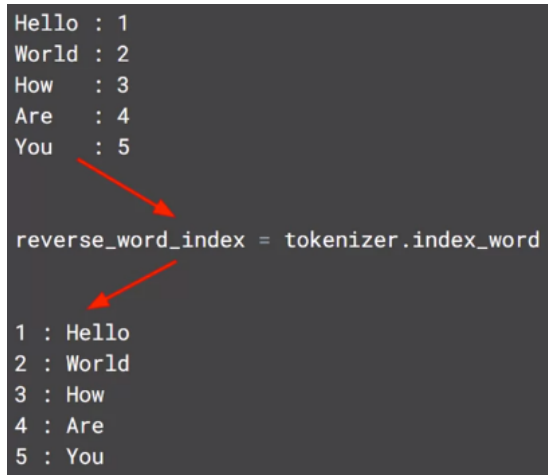
```

Podemos ver que esta es una matriz de 10.000 por 16. Tenemos 10.000 palabras en nuestro conjunto, y estamos trabajando en una matriz de 16 dimensiones, por lo que nuestro embedding tendrá esa forma.

```
Hello : 1
World : 2
How : 3
Are : 4
You : 5

reverse_word_index = tokenizer.index_word

1 : Hello
2 : World
3 : How
4 : Are
5 : You
```



Para poder trazarlo, necesitamos una función de ayuda para revertir nuestro índice de palabras. Tal como está actualmente, nuestro índice de palabras tiene la clave siendo la palabra, y el valor es el token de la palabra. Tendremos que voltear esto, mirar a través de la lista acolchada para decodificar los tokens de nuevo en las palabras.

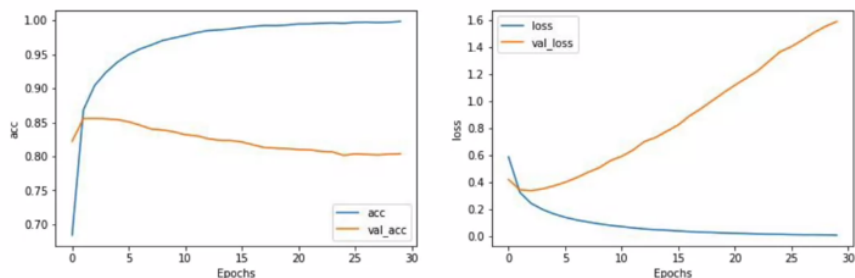
Ahora es el momento de escribir los vectores y sus archivos automáticos de metadatos. El proyector TensorFlow lee este tipo de archivo y lo utiliza para trazar los vectores en el espacio 3D para que podamos visualizarlos.

```
1 import io
2
3 out_v = io.open('vecs.tsv', 'w', encoding='utf-8')
4 out_m = io.open('meta.tsv', 'w', encoding='utf-8')
5
6 for word_num in range(1, vocab_size):
7     word = reverse_word_index[word_num]
8     embeddings = weights[word_num]
9
10
11     out_m.write(word + "\n")
12     out_v.write("\t".join([str(x) for x in embeddings]) + "\n")
13
14 out_v.close()
15 out_m.close()
```

[11] Para la matriz de metadatos, solo escribimos las palabras.

[12] Para el archivo vectores, simplemente escribimos el valor de cada uno de los elementos en la matriz de embeddings, es decir, el coeficiente de cada dimensión en el vector para esta palabra.

3.3. Función de pérdida



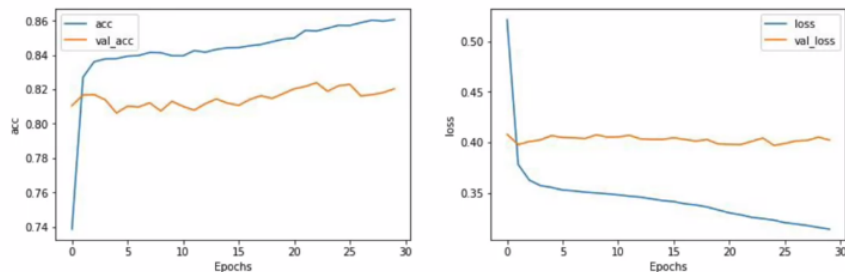
Podemos ver que la precisión aumenta muy bien a medida que entrenamos y la precisión de validación estaba bien, pero no muy buena. Lo interesante es que los valores de pérdida de la derecha, la pérdida de entrenamiento caen, pero la pérdida de validación aumentó. Bueno, ¿por qué podría ser eso?

Piense en la pérdida en este contexto, como una confianza en la predicción. Así que mientras que el número de predicciones precisas aumentó con el tiempo, lo interesante fue que la confianza por predicción disminuyó efectivamente. Puede que esto ocurra mucho con los datos de texto. Así que es muy importante vigilarlo.

Una forma de hacerlo es explorar las diferencias a medida que modifica los hiperparámetros.

```
vocab_size = 1000    (was 10,000)
embedding_dim = 16
max_length = 16      (was 32)
trunc_type='post'
padding_type='post'
oov_tok = "<OOV>"
training_size = 20000
```

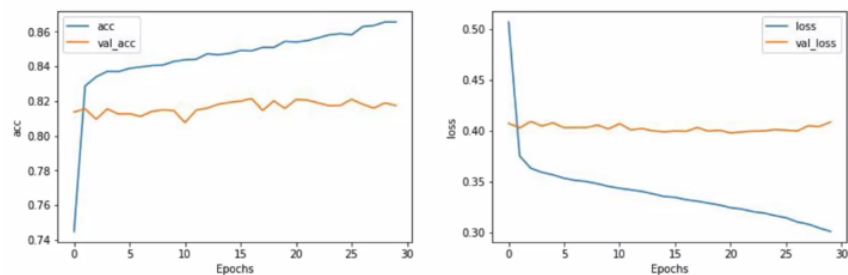
Así, por ejemplo, si considera estos cambios, una disminución en el tamaño del vocabulario, y tomar oraciones más cortas, reducir la probabilidad de relleno, y luego volver a ejecutar, es posible que vea resultados como este.



Aquí, se puede ver que la pérdida se ha aplanado lo que se ve bien, pero por supuesto, su precisión no es tan alta.

```
vocab_size = 1000 (was 10,000)
embedding_dim = 32 (was 16)
max_length = 16 (was 32)
trunc_type='post'
padding_type='post'
oov_tok = "<OOV>"
training_size = 20000
```

Otro retoque. También se intentó cambiar el número de dimensiones utilizando la incrustación. Aquí, podemos ver que eso tenía muy poca diferencia.



Poner los hiperparámetros como variables separadas como este es un ejercicio de programación útil, por lo que es mucho más fácil para usted ajustar y explorar su impacto en el entrenamiento.

3.4. Sub-palabras preentrenadas

Ahora tenemos un tokenizador de sub-palabras pre-entrenado, por lo que podemos inspeccionar su vocabulario mirando su propiedad de sub-palabras. Si queremos ver cómo codifica o decodifica cadenas, podemos hacerlo con este código.

```
1 sample_string = 'TensorFlow, from basics to mastery'
2
3 tokenized_string = tokenizer_subwords.encode(sample_string)
4 print('Tokenized string is {}'.format(tokenized_string))
5 # Tokenized string is [6307, 2327, 4043, 2120, 2, 48, 4249, 4429,
6   7, 2652, 8050]
7 original_string = tokenizer_subwords.decode(tokenized_string)
8 print('The original string is {}'.format(original_string))
9 # The original string is TensorFlow, from basics to mastery
```

[3] Podemos codificar simplemente llamando al método encode pasándole la cadena.

[7] De manera similar, decodificar llamando al método `decode`.

[5] y [9] Podemos ver los resultados de la tokenización cuando imprimimos las cadenas codificadas y decodificadas.

Si queremos ver los tokens en sí mismos, podemos tomar cada elemento y decodificarlo, mostrando el valor al token.

```
1 for ts in tokenized_string:
2     print('{ } ----> { }'.format(ts, tokenizer_subwords.decode([ts]))
```

```
6307 ----> Ten
2327 ----> sor
4043 ----> Fl
2120 ----> OW
2 ----> ,
48 ----> from
4249 ----> basi
4429 ----> cs
7 ----> to
2652 ----> master
```

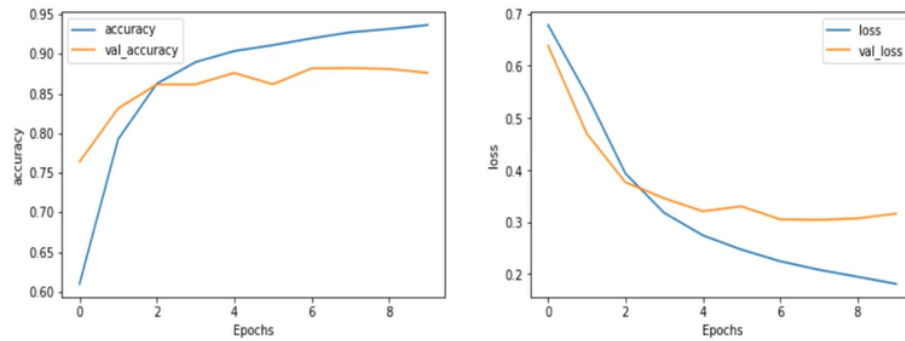
Tenga en cuenta que esto es sensible a mayúsculas y minúsculas y la puntuación se mantiene a diferencia del tokenizador que vimos en el último apartado. Todavía no necesita hacer nada con ellos, solo quería mostrar cómo funciona la tokenización de sub-palabras.

Así que ahora, echemos un vistazo a cómo clasificar IMDB con esto. ¿Cuáles serán los resultados? Aquí está el modelo.

```
1 embedding_dim = 64
2 model = tf.keras.Sequential([
3     tf.keras.layers.Embedding(tokenizer_subwords.vocab_size, embedding_dim),
4     tf.keras.layers.GlobalAveragePooling1D(),
5     tf.keras.layers.Dense(6, activation='relu'),
6     tf.keras.layers.Dense(1, activation='sigmoid')
7 ])
8
9 model.summary()
```

[4] Una cosa a tener en cuenta es la forma de los vectores que vienen del tokenizador a través del embedding, y no se pueden aplanar fácilmente. Por lo tanto, usaremos Global Average Pooling 1D en su lugar.

Podemos graficar los resultados.



Los significados de las sub-palabras a menudo no tienen sentido y solo cuando los combinamos en secuencias tienen semántica significativa. Por lo tanto, alguna forma de **aprendizaje de secuencias** sería un gran avance, y eso es exactamente lo que hará la en próxima sección con las redes neuronales recurrentes.

4. Modelos de secuencia

En esta sección veremos cómo implementar modelos de secuencia, como RNN (Recurrent Neural Network).

En una red neuronal, para tener en cuenta el orden de las palabras, se utilizan Arquitecturas de Red Neural Especializadas, como un RNN, GIO, o LSTM,

El contexto de las palabras es difícil de seguir cuando las palabras se dividen en subpalabras y la secuencia en la que aparecen los tokens para las subpalabras se vuelve muy importante para entender su significado.

La red neuronal es como una función que cuando la alimentas en datos y etiquetas, deduce las reglas de estas, y luego puedes usar esas reglas.



Una RNN a menudo se dibuja un poco así. Tiene su x como en la entrada y su y como salida. Pero también hay un elemento que se alimenta en función de una función anterior.

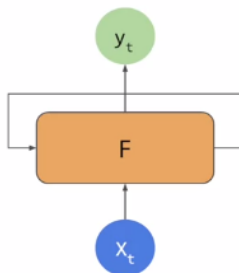
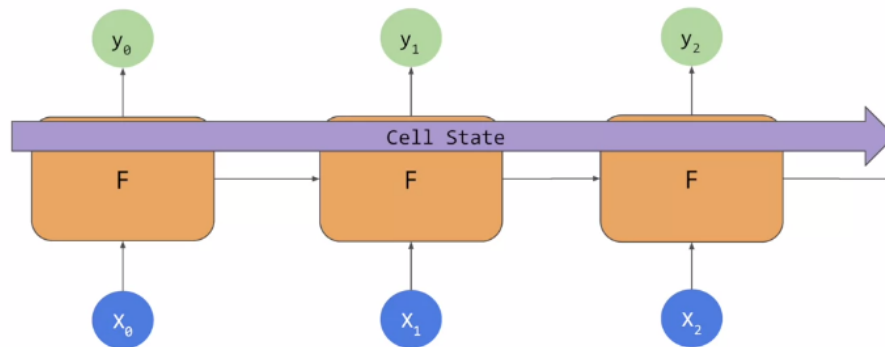


Figura 3: RNN

4.1. LSTM

Long Short Term Memory es una actualización de RNN que ha creado una **memoria a corto y largo plazo**. LSTM tiene una canalización adicional de contextos llamados **estado de celda**. Esto puede pasar a través de la red para impactarla. Esto ayuda a mantener el contexto de la relevancia de los tokens anteriores en los posteriores para que se puedan **evitar** problemas como la **pérdida de contexto**.



Los estados de celda también pueden ser **bidireccionales**. Por lo tanto, los contextos posteriores pueden afectar a los anteriores.

4.1.1. Implementar LSTM

```
1 model = tf.keras.Sequential([
2     tf.keras.layers.Embedding(tokenizer.vocab_size, 64),
3     tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64)),
4     tf.keras.layers.Dense(64, activation='relu'),
5     tf.keras.layers.Dense(1, activation='sigmoid')
6 ])
```

- [3] Se agrega una capa LSTM utilizando **tf.keras.layers.LSTM**. El parámetro pasado es el número de salidas que deseo de esa capa, en este caso es 64. Si lo envuelvo con **tf.keras.layers.Bidireccional**, hará que mi estado de celda vaya en ambas direcciones.

Layer (type)	Output Shape	Param #
embedding_2 (Embedding)	(None, None, 64)	523840
bidirectional_1 (Bidirection	(None, 128)	66048
dense_4 (Dense)	(None, 64)	8256
dense_5 (Dense)	(None, 1)	65
Total params: 598,209		
Trainable params: 598,209		
Non-trainable params: 0		

Tenemos nuestra incrustación y nuestra bidireccional que contiene el LSTM, seguido de las dos capas densas. Como podemos ver, la salida de la bidireccional

es ahora un 128, a pesar de que le dijimos a nuestro LSTM que queríamos 64, el bidireccional duplica esto hasta un 128.

```
1 model = tf.keras.Sequential([
2     tf.keras.layers.Embedding(tokenizer.vocab_size, 64),
3     tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64,
4         return_sequences=True)),
5     tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(32)),
6     tf.keras.layers.Dense(64, activation='relu'),
7     tf.keras.layers.Dense(1, activation='sigmoid')
8 ])
```

También podemos **apilar LSTM** como cualquier otra capa keras. Cuando un LSTM alimenta a otro, tiene que establecer *return_sequences=True* en el primero. Esto asegura que las salidas del LSTM coincidan con las entradas deseadas del siguiente. El resumen del modelo se verá así.

Layer (type)	Output Shape	Param #
embedding_3 (Embedding)	(None, None, 64)	523840
bidirectional_2 (Bidirectional)	(None, None, 128)	66048
bidirectional_3 (Bidirectional)	(None, 64)	41216
dense_6 (Dense)	(None, 64)	4160
dense_7 (Dense)	(None, 1)	65
Total params: 635,329		
Trainable params: 635,329		
Non-trainable params: 0		

Aquí está la comparación de las precisiones entre el LSTM de una capa y la de dos capas una en 10 épocas.

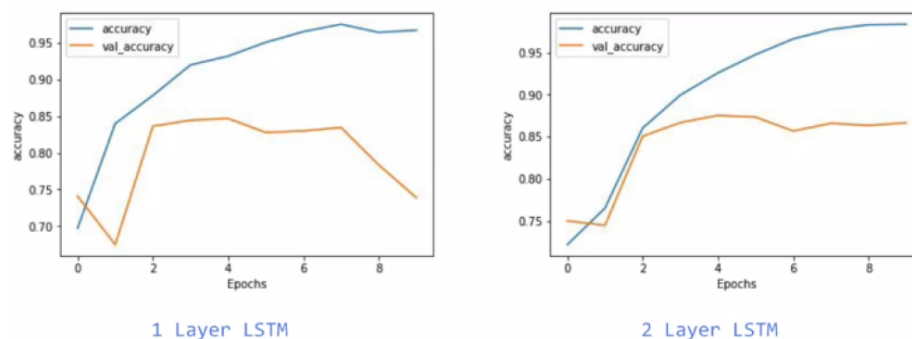


Figura 4: 10 Epochs: Accuracy Measurement

No hay mucha diferencia excepto la caída en picado y la precisión de la

validación. Pero observe cómo la curva de entrenamiento es más suave. La **irregularidad** puede ser una indicación de que su modelo necesita mejora, y el único LSTM que puede ver aquí no es el más suave.

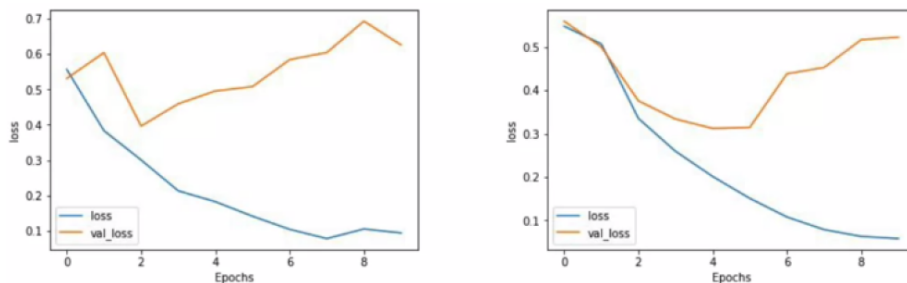


Figura 5: 10 Epochs: Loss Measurement

Si nos fijamos en la pérdida, en las primeras 10 épocas, podemos ver resultados similares.

Pero mira lo que sucede cuando aumentamos a 50 épocas de entrenamiento.

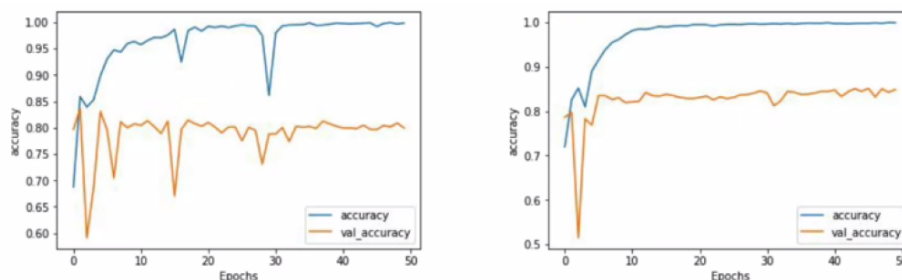


Figura 6: 50 Epochs: Accuracy Measurement

Nuestro LSTM de una capa, mientras sube con precisión, también es propenso a algunas caídas bastante agudas. El resultado final podría ser bueno, pero esas caídas me hacen sospechar sobre la precisión general del modelo.

El LSTM de dos capas se ve mucho más suave, y como tal me hace mucho más seguro en sus resultados.

Tenga en cuenta también la precisión de la validación. Teniendo en cuenta que los niveles se sitúan en alrededor del 80 por ciento, no está mal dado que el conjunto de entrenamiento y el conjunto de pruebas fueron 25.000 revisiones. Pero estamos usando 8.000 sub-palabras tomadas sólo del conjunto de entrenamiento. Así que habría muchos tokens en los conjuntos de pruebas que estarían fuera de vocabulario. Sin embargo, a pesar de eso, todavía estamos en cerca del 80 por ciento de precisión.

Nuestros resultados de pérdida son similares con las dos capas que tienen una curva mucho más suave. La pérdida está aumentando época por época. Así

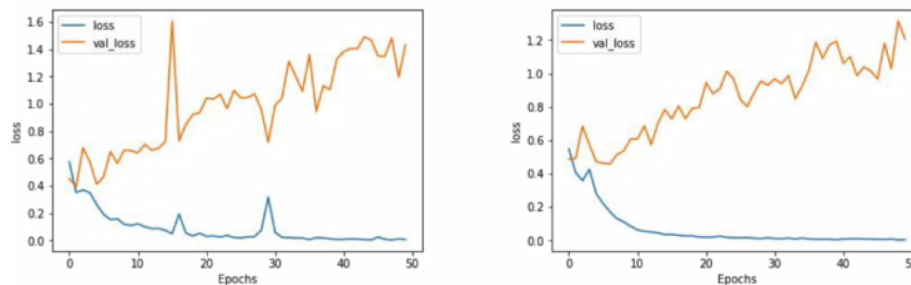


Figura 7: 50 Epochs: Loss Measurement

que vale la pena monitorear para ver si se aplanan en épocas posteriores como se desearía.

4.1.2. LSTM vs. non-LSTM

```
1 model = tf.keras.Sequential([
2     tf.keras.layers.Embedding(vocab_size, embedding_dim,
3                               input_length=max_length),
4     tf.keras.layers.GlobalAveragePooling1D(),
5     tf.keras.layers.Dense(64, activation='relu'),
6     tf.keras.layers.Dense(1, activation='sigmoid')
7 ])
```

Esta básica red neuronal tiene una incrustación que toma el tamaño de mi vocabulario, dimensiones de incrustación y longitud de entrada. La salida de la incrustación es promediada y luego alimenta a una red neuronal densa.

Podemos experimentar con las capas que unen la incrustación y la densa eliminando el aplanamiento (`GlobalAveragePooling1D`) y reemplazándolo con un LSTM como este.

```
1 model = tf.keras.Sequential([
2     tf.keras.layers.Embedding(vocab_size, embedding_dim,
3                               input_length=max_length),
4     tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(32)),
5     tf.keras.layers.Dense(64, activation='relu'),
6     tf.keras.layers.Dense(1, activation='sigmoid')
7 ])
```

Si entrenamos con el conjunto de datos de sarcasmo, cuando solo usamos la agrupación y el aplanamiento, rápidamente obtenemos cerca del 85 % de precisión y luego se aplanó. El conjunto de validación era un poco menos preciso, pero las curvas estamos bastante sincronizados.

Por otro lado, al usar LSTM, alcancé una precisión del 85 % muy rápidamente y continué subiendo hacia un 97,5 % de precisión dentro de 50 épocas.

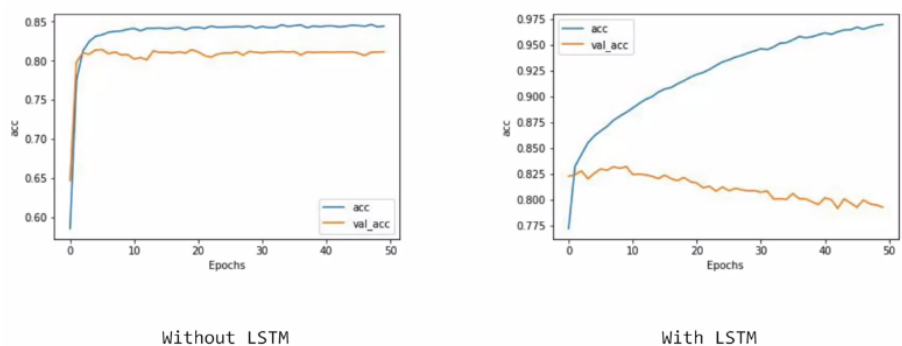


Figura 8: Accuracy non-LSMT vs. LSMT

El conjunto de validación cayó lentamente, pero todavía estaba cerca del mismo valor que la versión no LSTM. Aún así, la caída indica que hay algo de sobreajuste pasando aquí. Así que un poco de afinado al LSTM debería ayudar a solucionarlo.

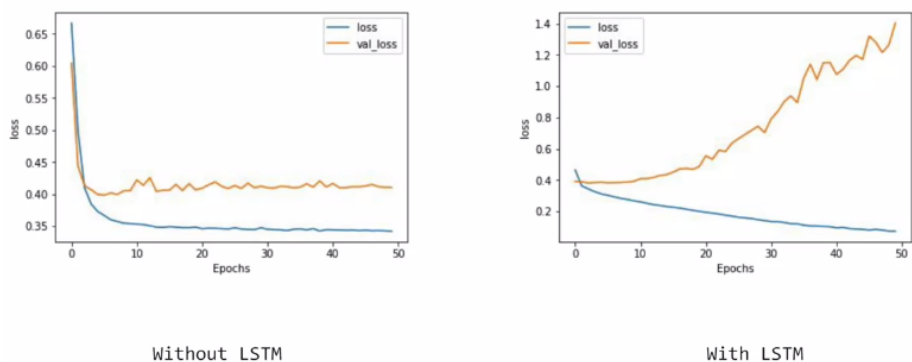


Figura 9: Loss non-LSMT vs. LSMT

Del mismo modo, los valores de pérdida sin LSTM alcanzaron un buen resultado con bastante rapidez y luego se aplanó. Mientras que con el LSTM, la pérdida de entrenamiento disminuye muy bien, pero la pérdida de validación aumentó a medida que continúo entrenando.

Una vez más, esto muestra algunos ajustes excesivos en la red LSTM. Mientras que la precisión de la predicción aumentó, la confianza en ella disminuyó. Por lo tanto, debe tener cuidado de ajustar sus parámetros de entrenamiento cuando use diferentes tipos de red.

4.2. Otros tipos de RNN

Las palabras que no son vecinos inmediatos pueden afectar el contexto del otro.

En este apartado veremos algunas otras opciones de RNN incluyendo **convoluciones**, **GRU** (*Gated Recurrent Units - Unidades Recurrentes con Puertas*), y más sobre cómo escribir el código para ellos.

4.2.1. GRU

Los GRU (Gated Recurrent Units) son un tipo de células de memoria utilizadas en redes neuronales recurrentes. Estas unidades tienen la capacidad de **retener información** en el tiempo y **adaptarse a patrones cambiantes** en los datos de entrada.

Una de las principales características de los GRU es su **estructura de puerta**, que les permite **controlar el flujo de información** a través de la unidad. Esto les permite **recordar información importante** y **descartar información no relevante**.

Además, los GRU son eficientes en términos de memoria, lo que los hace adecuados para aplicaciones de procesamiento de lenguaje natural y series de tiempo. También se pueden utilizar en aplicaciones de visión por computadora, reconocimiento de voz y otros campos relacionados con la inteligencia artificial.

```
1 model = tf.keras.Sequential([
2     tf.keras.layers.Embedding(vocab_size, embedding_dim,
3                               input_length=max_length),
4     tf.keras.layers.Bidirectional(tf.keras.layers.GRU(32)),
5     tf.keras.layers.Dense(64, activation='relu'),
6     tf.keras.layers.Dense(1, activation='sigmoid')
7 ])
```

4.2.2. Diferencias entre LSTM y GRU

- **Complejidad:** Las LSTM son generalmente más complejas que las GRU, ya que tienen más unidades de memoria y más puertas que controlan el flujo de información a través de la unidad.
- **Puertas:** Las LSTM tienen tres tipos de puertas diferentes que controlan el flujo de información a través de la unidad (puerta de entrada, puerta de salida y puerta de olvido), mientras que las GRU tienen dos puertas (puerta de actualización y puerta de reinicio).
- **Células de memoria:** Las LSTM utilizan células de memoria separadas para almacenar información a corto y largo plazo, mientras que las GRU tienen una sola unidad de memoria que cumple ambas funciones.

- **Eficiencia:** Las GRU son generalmente más eficientes que las LSTM en términos de computación y memoria, lo que las hace adecuadas para aplicaciones con recursos limitados.

En términos de rendimiento, no hay una clara ventaja de una sobre la otra, ya que ambas redes neuronales recurrentes han demostrado ser efectivas en diversas tareas. La elección de una u otra dependerá del problema específico que se esté abordando y de las características de los datos de entrada.

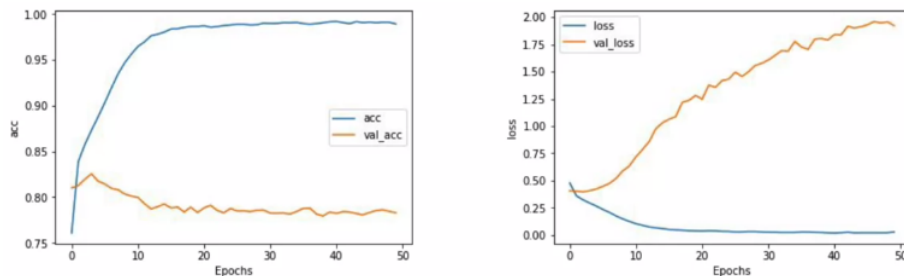
- En general, los datos que se ajustan mejor a las **LSTM** son aquellos que tienen dependencias a largo plazo y requieren una memoria a largo plazo. Por ejemplo, en el procesamiento de lenguaje natural, las oraciones y los párrafos que tienen una estructura compleja y dependencias a largo plazo son buenos candidatos para ser modelados por una LSTM. Las LSTM también son útiles en aplicaciones de visión por computadora, donde es necesario seguir objetos en el tiempo y mantener una memoria a largo plazo de su posición y movimiento.
- Por otro lado, los datos que se ajustan mejor a las **GRU** son aquellos que no requieren una memoria tan a largo plazo como LSTM. Las GRU son particularmente útiles en aplicaciones donde los recursos de computación y memoria son limitados, ya que son más eficientes que las LSTM en términos de memoria y computación. Las GRU son adecuadas para tareas de clasificación y predicción en series de tiempo, así como en la generación de texto y habla.

4.2.3. Convolución

Otro tipo de capa que puedes usar es una convolución, de una manera muy similar a la que hicimos con las imágenes.

```
1 model = tf.keras.Sequential([
2     tf.keras.layers.Embedding(vocab_size, embedding_dim,
3                               input_length=max_length),
4     tf.keras.layers.Conv1D(128, 5, activation='relu'),
5     tf.keras.layers.GlobalMaxPooling1D(),
6     tf.keras.layers.Dense(64, activation='relu'),
7     tf.keras.layers.Dense(1, activation='sigmoid')
8 ])
```

- [3] y [4] Especificamos el número de circunvoluciones que deseamos aprender, su tamaño y su función de activación. Ahora las palabras se agruparán en el tamaño del filtro en este caso 5. Y las circunvoluciones aprenderán que pueden asignar la clasificación de palabras a la salida deseada.



Si entrenamos con las convoluciones ahora, veremos que nuestra precisión funciona incluso mejor que antes con cerca del *100 %* en entrenamiento y alrededor del **80 %** en validación. Pero como antes, nuestra pérdida aumenta en el conjunto de validación, lo que indica un posible sobreajuste.

```
max_length = 120

tf.keras.layers.Conv1D(128, 5, activation='relu'),
```

Layer (type)	Output Shape	Param #
embedding (Embedding)	(None, 120, 16)	16000
conv1d (Conv1D)	(None, 116, 128)	10368
global_max_pooling1d (Global)	(None, 128)	0
dense (Dense)	(None, 24)	3096
dense_1 (Dense)	(None, 1)	25

```
Total params: 29,489
Trainable params: 29,489
Non-trainable params: 0
```

Si volvemos al modelo y exploramos los parámetros, veremos que tenemos 128 filtros cada uno para 5 palabras. Y una exploración del modelo mostrará estas dimensiones. Como el tamaño de la entrada fue de 120 palabras (max_length), y un filtro que es de 5 palabras de largo recortará 2 palabras de la parte delantera y trasera, dejándonos con 116.

4.3. Evitar sobreajuste

¿Cuál es la mejor manera de evitar el overfitting en conjuntos de datos de procesamiento de lenguaje natural (NLP)?

- Utilizar LSTMs
- Utilizar GRUs
- Utilizar Conv1D
- Ninguna de las anteriores**

La mejor manera de evitar el overfitting en conjuntos de datos de NLP es utilizar técnicas apropiadas de regularización y validación. Algunas de las técnicas comúnmente utilizadas son:

- **Dropout:** es una técnica de regularización que elimina aleatoriamente algunos de las neuronas durante el entrenamiento. Esto ayuda a prevenir que el modelo se ajuste demasiado a los datos de entrenamiento.
- **Detención temprana:** esta técnica implica detener el proceso de entrenamiento cuando el error de validación deja de mejorar. Esto ayuda a prevenir el overfitting evitando más entrenamiento una vez que el modelo ha alcanzado su rendimiento óptimo en el conjunto de validación.
- **Regularización L1 y L2:** son técnicas de regularización que agregan un término de penalización a la función de pérdida para evitar que el modelo se ajuste demasiado.
- **Validación cruzada:** esto implica dividir los datos en conjuntos de entrenamiento y validación múltiples veces y usar diferentes subconjuntos para entrenamiento y validación. Esto ayuda a obtener una mejor estimación del rendimiento del modelo en datos no vistos.

Usar solo LSTMs, GRUs o Conv1D no puede prevenir el overfitting en conjuntos de datos de NLP. La elección de la arquitectura del modelo debe basarse en los requisitos específicos de la tarea en cuestión y las características del conjunto de datos. Sin embargo, las técnicas mencionadas anteriormente deben usarse en conjunto con la arquitectura del modelo elegida para prevenir el overfitting y garantizar una mejor generalización.

5. Modelos de secuencia y literatura

Hasta ahora hemos visto la clasificación del texto. Pero ¿qué pasa si queremos **generar nuevo texto**? En lugar de generar nuevo texto, ¿qué tal pensar en él como un problema de predicción?

Podemos obtener un conjunto de textos, extraer el vocabulario completo de él, y luego crear conjuntos de datos a partir de eso, donde hacemos que la frase sea Xs y la siguiente palabra en esa frase sea Ys .

```
1 tokenizer = Tokenizer()
2
3 data = "In the town of Athy one Jeremy Lanigan \n Battered away ...
4 corpus = data.lower().split("\n")
5
6 tokenizer.fit_on_texts(corpus)
7 total_words = len(tokenizer.word_index) + 1
```

- [3] En este caso, para mantener las cosas simples, ponemos la oración en una sola cadena.
- [4] Luego, llamando a la función *split()* en "\n", podemos crear una lista de oraciones a partir de los datos y convertiremos todo eso a minúsculas con la función *lower()*.
- [6] Usando el tokenizer, podemos llamar *fit_on_texts()* a este corpus de trabajo y creará el diccionario de palabras y el corpus general. Este es un par de valores y clave, con la clave siendo la palabra y el valor es el token de esa palabra.
- [7] Podemos encontrar el número total de palabras en el corpus, obteniendo la longitud de su índice de palabras. Vamos a añadir uno a esto, para considerar palabras de vocabulario externo (*'oov_token'*).

5.1. Preparar el entrenamiento

Vamos el código para tomar este corpus y convertirlo en datos de entrenamiento.

```
1 input_sequences = []
2 for line in corpus:
3     token_list = tokenizer.texts_to_sequence([line])[0]
4     for i in range(1, len(token_list)):
5         n_gram_sequence = token_list[:i+1]
6         input_sequences.append(n_gram_sequence)
```

- [1] *input_sequences* será nuestras X de entrenamiento.

- [3] Para cada línea en el corpus, generaremos una lista de tokens usando el método `tokenizer.texts_to_sequence()`. Esto convertirá una línea de texto en una lista de los tokens que representan las palabras
- [5] Luego vamos a iterar sobre esta lista de tokens y crear un número de secuencias de `n_grams`, es decir, las 2 primeras palabras de la oración o una secuencia, luego las 3 primeras son otra secuencia, etc.

El resultado de esto será, para la primera línea de la canción, la siguiente entrada secuencias que se generarán.

Line:	Input Sequences:
[4 2 66 8 67 68 69 70]	[4 2]
	[4 2 66]
	[4 2 66 8]
	[4 2 66 8 67]
	[4 2 66 8 67 68]
	[4 2 66 8 67 68 69]
	[4 2 66 8 67 68 69 70]

A continuación, necesitamos encontrar la longitud de la frase más larga en el corpus.

```
1 max_sequence_len = max([len(x) for x in input_sequences])
```

Una vez que tengamos nuestra longitud de secuencia más larga, lo siguiente que hacer es rellenar todas las secuencias para que tengan la misma longitud. Vamos a hacer pre-padding con ceros para que sea más fácil extraer la etiqueta.

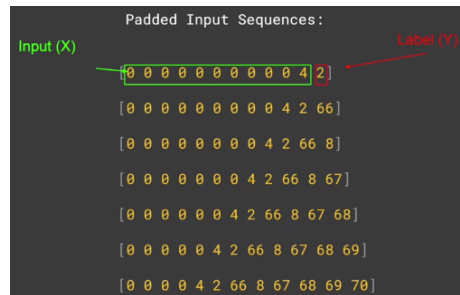
```
1 input_sequences = np.array(pad_sequences(input_sequences, maxlen=
    max_sequence_len, padding='pre'))
```

Así que ahora, nuestra línea estará representada por un conjunto de secuencias de entrada acolchadas que se ven así.

Line:	Padded Input Sequences:
[4 2 66 8 67 68 69 70]	[0 0 0 0 0 0 0 0 4 2]
	[0 0 0 0 0 0 0 0 4 2 66]
	[0 0 0 0 0 0 0 0 4 2 66 8]
	[0 0 0 0 0 0 0 0 4 2 66 8 67]
	[0 0 0 0 0 0 0 0 4 2 66 8 67 68]
	[0 0 0 0 0 0 0 0 4 2 66 8 67 68 69]
	[0 0 0 0 0 0 0 0 4 2 66 8 67 68 69 70]

Ahora, que tenemos nuestras secuencias, lo siguiente que tenemos que hacer es **convertirlas en X e Y** , nuestros valores de entrada y sus etiquetas. Todo lo que tenemos que hacer es tomar **todo menos el último carácter como la X** y luego usar el **último carácter como la Y** en nuestra etiqueta.

```
1 xs = input_sequences[:, :-1]
2 labels = input_sequences[:, -1]
```

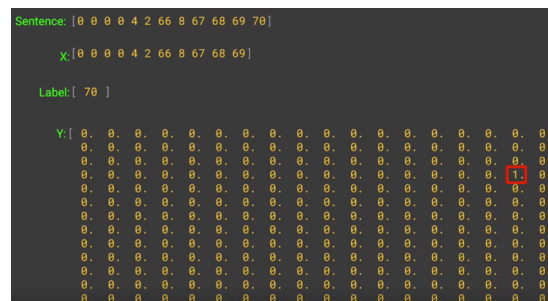


En este punto, debería quedar claro por qué hicimos pre-relleno, porque nos hace mucho más fácil obtener la etiqueta simplemente agarrando el último token. Cuando se le dé una secuencia de palabras, puedo clasificar desde el corpus, cuál sería la siguiente palabra.

Ahora, debería **codificar** mis etiquetas, ya que esto realmente es un **problema de clasificación**.

```
1 ys = tf.keras.utils.to_categorical(labels, num_classes=total_words)
```

[1] `tf.keras.utils.to_categorical()` convierte una lista en una categórica. Simplemente le doy la lista de etiquetas y el número de clases que es mi número de palabras, y creará una **codificación unificada** (*one-hot encoding*) de las etiquetas.



Así, por ejemplo, si consideramos esta lista de tokens como una frase, entonces la X es la lista hasta el último valor, y la *etiqueta* es el último valor que en este caso es 70. La Y es una matriz codificada de una sola vez si la longitud es el tamaño del corpus de palabras y el valor que se establece en **1** es el que está en el **índice de la etiqueta** que en este caso es el elemento 70.

5.2. Construir NN

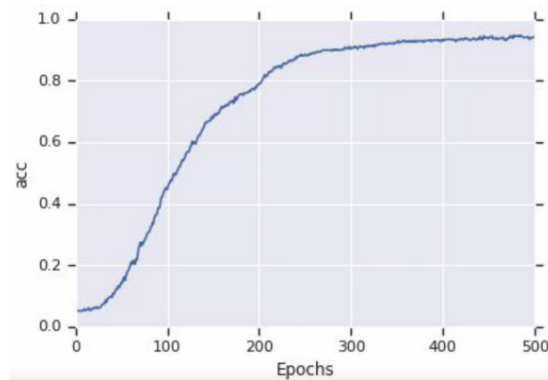
Ahora que tenemos nuestros datos como xs y ys , es relativamente simple para nosotros crear una red neuronal para clasificar cuál debería ser la siguiente palabra, dado un conjunto de palabras.

```
1 model = Sequential()
2 model.add(Embedding(total_words, 64, input_length=max_sequence_len
3                     - 1))
4 model.add(LSTM(20))
5 model.add(Dense(total_words, activation='softmax'))
6 model.compile(loss='categorical_crossentropy', optimizer='adam',
7               metrics=['accuracy'])
8 model.fit(xs, ys, epochs=500, verbose=1)
```

- [2] En la capa de incrustación (*Embedding*) vamos a querer que maneje todas nuestras palabras, así que lo establecemos en el primer parámetro (*total_words*). El segundo parámetro es el número de dimensiones que se van a utilizar para trazar el vector de una palabra (**64**). Siéntase libre de modificar esto para ver cuál sería su impacto en los resultados. Finalmente, el tamaño de las dimensiones de entrada será alimentado, y esta es la **longitud de la secuencia más larga menos 1**. Restamos uno porque recortamos la última palabra de cada secuencia para obtener la etiqueta, por lo que nuestras secuencias serán una menor que la longitud máxima de secuencia.
- [3] A continuación agregaremos un *LSTM*. Como vimos anteriormente, su *estado celular* significa que llevan contexto junto con ellos, por lo que no son solo palabras vecinas las que tienen un impacto. Vamos a especificar **20** unidades aquí, pero de nuevo, podemos sentirnos libres de experimentar.
- [4] Finalmente hay una capa *densa* de tamaño como el **total de palabras**, que es el mismo tamaño que usamos para la codificación *one-hot*. Así, esta capa tendrá **una neurona por palabra**, y esa neurona debería iluminarse cuando predecimos una palabra dada.
- [6] Estamos haciendo una clasificación categórica, así que estableceremos las pérdidas como *categorical_crossentropy*. Y usaremos el optimizador *adam* que parece funcionar particularmente bien para tareas como esta.

- [7] Finalmente, vamos a entrenar para una gran cantidad de épocas, digamos alrededor de 500, ya que un modelo como este tarda un tiempo en converger, sobre todo porque tiene muy pocos datos.

Si entrenamos el modelo para 500 épocas, se verá así.



Aquí hay algunas frases que se generaron cuando le di a la red neuronal la frase "Lawrence fue a Dublín", y le pedí que predijera las siguientes 10 palabras.

```
Laurence went to dublin round the plenty as red wall me for wall wall  
Laurence went to dublin odaly of the nice of lanigans ball ball ball hall  
Laurence went to dublin he hadnt a minute both relations hall new relations youd
```

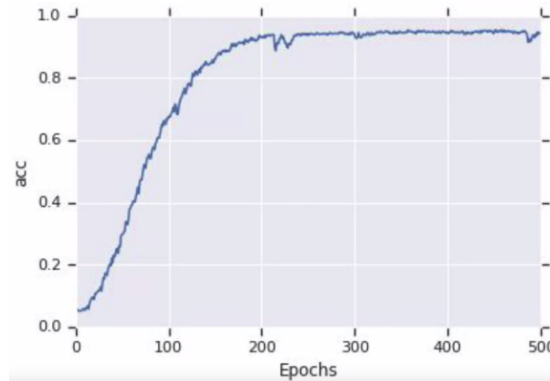
En esta frase, 3 de las últimas 5 palabras son pared (wall), 3 de las 4 últimas son bola (ball), e incluso la palabra relaciones (relations) se repite. Esto se debe a que **nuestro LSTM solo llevaba el contexto hacia adelante**.

Echemos un vistazo a lo que sucede si cambiamos el código para que sea **bidireccional**.

```
1 model = Sequential()  
2 model.add(Embedding(total_words, 64, input_length=max_sequence_len  
   - 1))  
3 model.add(Bidirectional(LSTM(20)))  
4 model.add(Dense(total_words, activation='softmax'))  
5  
6 model.compile(loss='categorical_crossentropy', optimizer='adam',  
   metrics=['accuracy'])  
7 model.fit(xs, ys, epochs=500, verbose=1)
```

- [3] Al agregar esta línea simplemente se define el LSTM como *bidireccional*.

Al volver a entrenar, puedo ver que converjo un poco más rápido.



Después de entrenar y probar, ahora recibo estas frases.

```
Laurence went to dublin think and wine for lanigans ball entangled in nonsense me  
Laurence went to dublin his pipes bellows chanter and all all entangled all kinds  
Laurence went to dublin how the room a whirligig ructions long at brooks fainted
```

Tienen un poco más de sentido, pero todavía hay algo de repetición. Dicho esto, recuerden que esta es una canción donde las palabras riman como bola (*ball*), todo (*all*) y pared (*wall*), etcétera, y como tales muchas de ellas van a aparecer.

5.3. Predecir una palabra

Ahora, echemos un vistazo a cómo obtener una predicción para una palabra y cómo generar nuevo texto basado en esas predicciones. Así que empecemos con una sola frase. Por ejemplo, "Lawrence fue a Dublín". Llamemos a esta frase la semilla (*seed_text*).

Si quiero predecir las siguientes 10 palabras en la oración para seguir esto, entonces este código lo tokenizará usando el método de `texts_to_sequences`.

```
1 token_list = tokenizer.texts_to_sequences([seed_text])[0]
```

Como no tenemos una palabra de vocabulario externa, ignorará 'Lawrence,' que no está en el corpus y obtendrá la siguiente secuencia.

```
Laurence went to dublin  
[134, 13, 59]
```

Este código rellenará la secuencia para que coincida con los del conjunto de entrenamiento.

```
1 token_list = pad_sequences([token_list], maxlen=max_sequence_len -  
2   1, padding='pre')  
3 # [0  0  0  0  0  0  0 134 13 59]
```

Así que terminamos con algo como esto que podemos pasar al modelo para obtener una predicción de vuelta.

```
1 predicted = model.predict(token_list)  
2 predicted = np.argmax(probabilities, axis= - 1)[0]
```

Esto nos dará el símbolo de la palabra más probable que sea la siguiente en la secuencia. Así que ahora, podemos hacer una búsqueda inversa en los elementos del índice de palabras para convertir el token de nuevo en una palabra y agregar eso a nuestros textos iniciales.

```
1 output_word = tokenizer.index_word[predicted]  
2 seed_text += " " + output_word
```

Este es el código completo para hacerlo 10 veces.

```
1 seed_text = "Lawrence weny to dublin"  
2 next_words = 10  
3 for _ in range(next_words):  
4     token_list = tokenizer.texts_to_sequences([seed_text])[0]  
5     token_list = pad_sequences([token_list], maxlen=  
6         max_sequence_len - 1, padding='pre')  
7     predicted = model.predict_classes(token_list, verbose=0)  
8     output_word = tokenizer.index_word[predicted]  
9     seed_text += " " + output_word  
10 print(seed_text)
```

Este enfoque funciona muy bien hasta que tiene grandes cuerpos de texto con muchas palabras. Así, por ejemplo, podría probar las obras completas de Shakespeare y probablemente golpeará errores de memoria, como asignar las codificaciones de una sola vez en caliente de las etiquetas a matrices que tienen más de 31.477 elementos, que es el número de palabras únicas en la colección, y hay más de 15 millones de secuencias generadas usando el algoritmo que mostramos aquí. Por lo tanto, las etiquetas por sí solas requerirían el almacenamiento de muchos terabytes de RAM.

6. Resumen

```
1 # Paso 1: Tokenizar
2 # Paso 2: Transformacion n_grams (hacer subconjuntos de tokens)
3 # Paso 3: pad sequences (añadir tokens para tener misma longitud)
4 # Paso 4: Separar características y etiquetas
5 # Paso 5: Crear modelo
6     # Embedding: transforma texto a secuencia numerica
7     # Memoria corto/largo plazo, uni/bidireccional
8     # Capas intermedias
9     # Capa salida 'softmax'
10 # Paso 6: Entrenar modelo
```