Tema 4: Redes Neuronales

Redes Neuronales Recurrentes

Prof. Wladimir Rodríguez

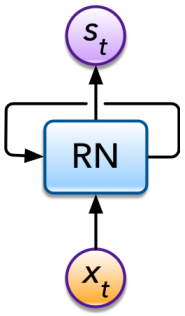
wladimir@ula.ve

Departamento de Computación

Los seres humanos no comienzan su pensamiento desde cero cada segundo. Al leer este ensayo, entiendes cada palabra según tu comprensión de las palabras anteriores. No tiras todo y comienzas a pensar de nuevo. Tus pensamientos tienen persistencia.

Las redes neuronales tradicionales no pueden hacer esto, y parece una deficiencia importante. Por ejemplo, imagine que desea clasificar qué tipo de evento está sucediendo en cada punto de una película. No está claro cómo una red neuronal tradicional podría usar su razonamiento sobre eventos previos en la película para informar a los posteriores.

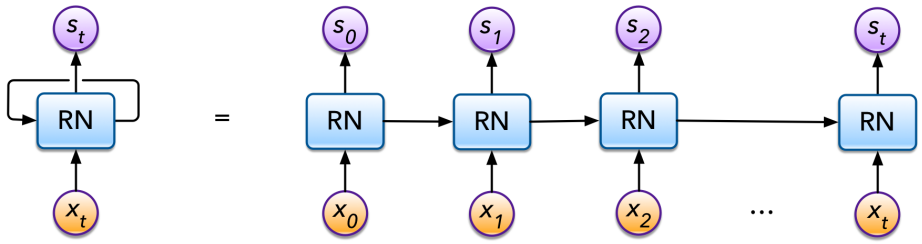
Las redes neuronales recurrentes abordan este problema. Son redes con bucles que permiten que la información persista.



En la figura de arriba, un trozo de la red neuronal RN, observa una entrada x_t y genera un valor s_t. Un ciclo permite que la información pase de un paso de la red al siguiente.

Estos bucles hacen que las redes neuronales recurrentes parezcan misteriosas. Sin embargo, si piensa un poco más, resulta que no son tan diferentes a una red neuronal normal. Una red neuronal recurrente se puede considerar como copias múltiples de la

misma red, cada una pasando un mensaje a un sucesor. Considere lo que sucede si desenrollamos el ciclo:



Una red neuronal recurrente desenrollada

Esta naturaleza de cadena revela que las redes neuronales recurrentes están íntimamente relacionadas con secuencias y listas. Son la arquitectura natural de una red neuronal para usar para tales datos.

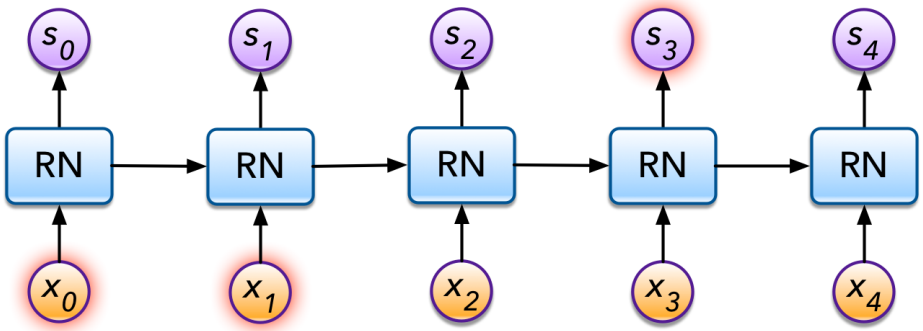
En los últimos años, ha habido un éxito increíble aplicando RNR a una variedad de problemas: reconocimiento de voz, modelado de lenguaje, traducción, subtitulado de imágenes ...

Esencial para estos éxitos es el uso de LSTM (Long Short-Time Memory), un tipo muy especial de red neuronal recurrente que funciona, para muchas tareas, mucho mejor que la versión estándar. Casi todos los mejores resultados basados en redes neuronales recurrentes se logran con ellas. Son estas LSTM las que se explorará a continuación.

El Problema de las Dependencias a Largo Plazo

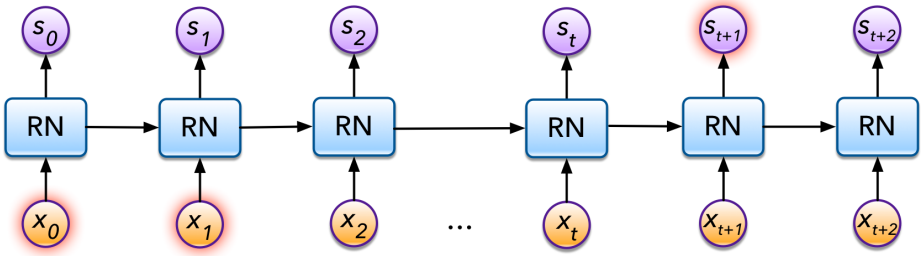
Uno de los atractivos de las RNR es la idea de que podrían ser capaces de conectar información previa a la tarea actual, por ejemplo, el uso de los cuadros de video previos podría ayudar a comprender el cuadro actual. Si las RNR pudieran hacer esto, serían extremadamente útiles.

Algunas veces, solo necesitamos ver información reciente para realizar la tarea actual. Por ejemplo, considere un modelo de lenguaje que intente predecir la siguiente palabra en función de las anteriores. Si estamos tratando de predecir la última palabra en "las nubes están en el *cielo*", no necesitamos ningún contexto adicional; es bastante obvio que la próxima palabra será cielo. En tales casos, cuando la brecha entre la información relevante y el lugar que se necesita es pequeña, las RNR pueden aprender a usar la información pasada.



Pero también hay casos en los que necesitamos más contexto. Considere tratar de predecir la última palabra en el texto "Crecí en Francia ... Hablo *francés* con fluidez". La información reciente sugiere que la siguiente palabra es probablemente el nombre de un idioma, pero si queremos reducir el idioma, necesitamos el contexto de Francia, desde atrás. Es completamente posible que exista una brecha entre la información relevante y el punto en el que se necesita que sea muy grande.

Desafortunadamente, a medida que crece esa brecha, las RNN no pueden aprender a conectar la información.



En teoría, las RNR son absolutamente capaces de manejar tales "dependencias a largo plazo". Un humano podría elegir cuidadosamente los parámetros para que resuelvan problemas de juguete de esta forma. Tristemente, en la práctica, las RNR no parecen ser capaces de aprenderlas. El problema fue explorado en profundidad por Hochreiter (1991) [alemán] y Bengio, et al. (1994), quienes encontraron algunas razones bastante fundamentales por las cuales podría ser difícil.

Este problema se puede resolver con las LSTM.

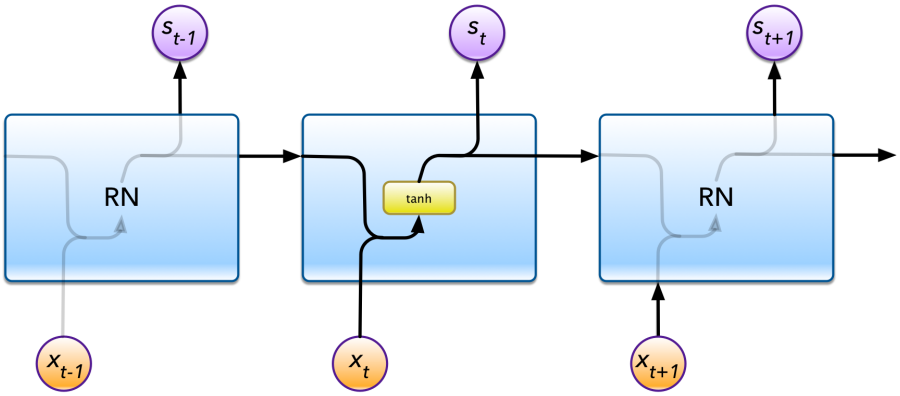
Redes LSTM

Las redes de memoria de corto y largo plazo, generalmente llamadas "LSTM" del ingles (Long Short Term Memory), son un tipo especial de RNR, capaz de aprender dependencias a largo plazo. Fueron introducidos por Hochreiter y Schmidhuber (1997), y

fueron refinadas y popularizadas por muchas personas en trabajos posteriores. Funcionan muy bien en una gran variedad de problemas, y ahora son ampliamente utilizadas.

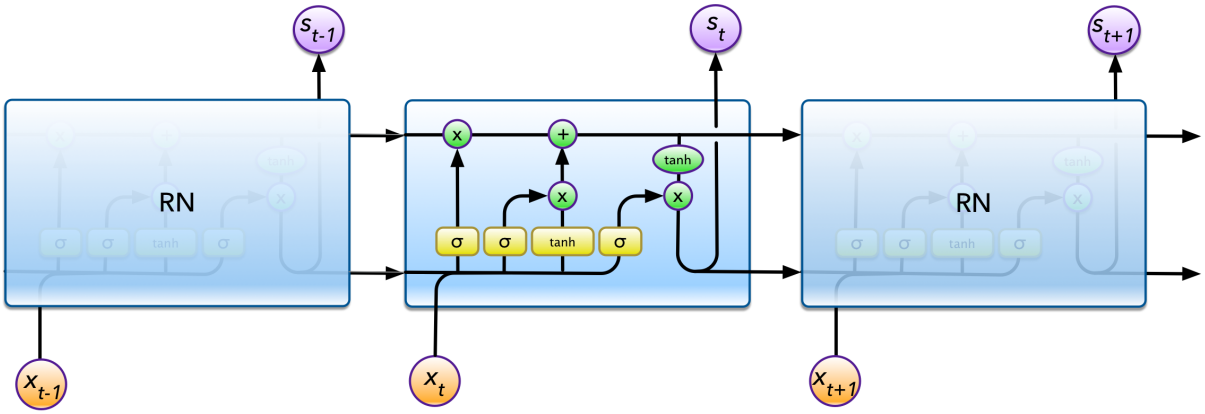
Las LSTM están diseñados explícitamente para evitar el problema de dependencia a largo plazo. Recordar la información durante largos períodos de tiempo es prácticamente su comportamiento predeterminado.

Todas las redes neuronales recurrentes tienen la forma de una cadena de módulos repetitivos de red neuronal. En RNN estándar, este módulo de repetición tendrá una estructura muy simple, como una sola capa de tanh.



El módulo de repetición en una RNR estándar contiene una sola capa

Los LSTM también tienen esta estructura tipo cadena, pero el módulo de repetición tiene una estructura diferente. En lugar de tener una sola capa de red neuronal, hay cuatro, interactuando de una manera muy especial.



El módulo de repetición en una LSTM contiene cuatro capas que interactúan

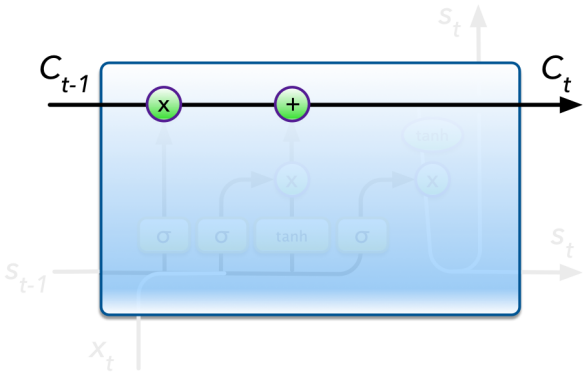
Notación:

En la figura de arriba, cada línea lleva un vector completo, desde la salida de un nodo hasta las entradas de otros. Los círculos verdes representan operaciones puntuales, como la adición de vectores, mientras que los cuadros amarillos son capas entrenadas de redes neuronales. Las líneas que se unen denotan concatenación, mientras que una línea de bifurcación denota que su contenido se copia y las copias van a diferentes ubicaciones.

La idea central detrás de los LSTM

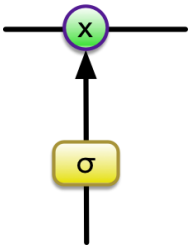
La clave para las LSTM es el estado de la celda, la línea horizontal que se extiende por la parte superior de la figura.

El estado de la celda es como una cinta transportadora. Corre directamente por toda la cadena, con solo algunas interacciones lineales menores. Es muy fácil que la información fluya sin cambios



La LSTM tiene la capacidad de eliminar o agregar información al estado de la celda, cuidadosamente regulado por estructuras llamadas compuertas.

Las compuertas son una forma de dejar pasar la información. Se componen de una capa de red neuronal con activación sigmoide y una operación de multiplicación puntual.



La salida de capa con activación sigmoide son números entre cero y uno, que describe la cantidad que se debe dejar pasar cada componente. Un valor de cero significa "no dejar pasar nada", mientras que un valor de uno significa "dejar pasar todo"

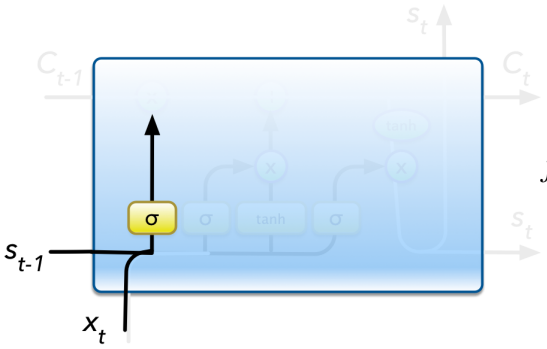
Una LSTM tiene tres de estas compuertas, para proteger y controlar el estado de la celda.

Recorrido paso a paso de una LSTM

El primer paso en nuestra LSTM es decidir qué información vamos a olvidar/recordar del estado de la celda. Esta decisión es tomada por una capa con activación sigmoide

llamada "capa de compuerta de olvido". Basado en los valores de s_{t-1} y x_t, genera un número entre 0 y 1 para cada número del estado de la celda C_{t-1}. Un 1 representa "mantener esto por completo", mientras que un 0 representa "deshacerse por completo de esto".

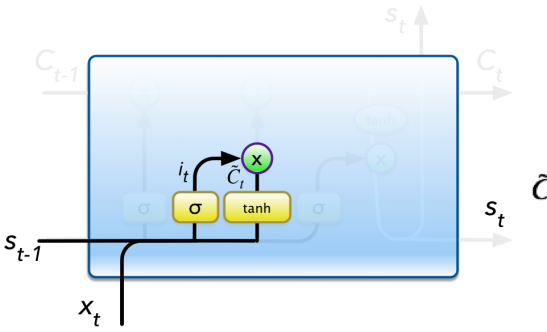
Regresemos a nuestro ejemplo de un modelo de lenguaje que intenta predecir la siguiente palabra en base a todas las anteriores. En tal problema, el estado de la celda puede incluir el género del sujeto presente, de modo que puedan usarse los pronombres correctos. Cuando vemos un sujeto nuevo, queremos olvidar el género del sujeto anterior.



$$f_t = \sigma(W_f \cdot [s_{t-1}, x_t] + b_f)$$

El siguiente paso es decidir qué nueva información vamos a almacenar en el estado de la celda. Esto tiene dos partes. Primero, una capa sigmoide llamada "compuerta capa de entrada" decide qué valores actualizaremos. A continuación, una capa $tanh$ crea un vector de nuevos valores candidatos, $\tilde{C}_t$, que podrían agregarse al estado. En el siguiente paso, combinaremos estos dos para crear la actualización del estado.

En el ejemplo de nuestro modelo de lenguaje, nos gustaría agregar el género del nuevo sujeto al estado de la celda, para reemplazar el anterior que estamos olvidando.



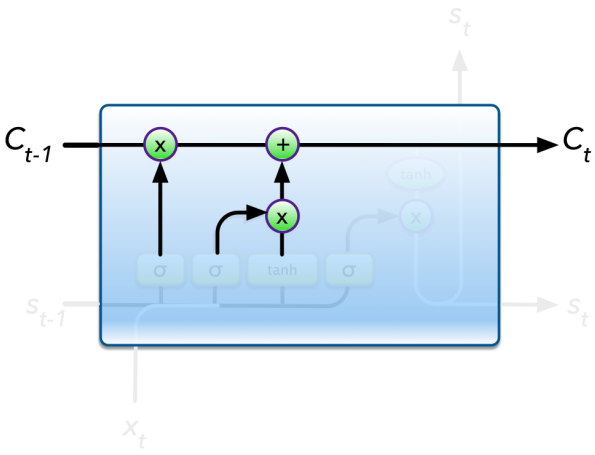
$$i_t = \sigma(W_i \cdot [s_{t-1}, x_t] + b_i)$$

$$\tilde{C}_t = tanh(W_C[s_{t-1}, x_t] + b_C)$$

Ahora es el momento de actualizar el estado de celda anterior, C_{t-1}, en el nuevo estado de celda C_t. Los pasos anteriores ya decidieron qué hacer, solo tenemos que hacerlo.

Multiplicamos el estado anterior por f_t, olvidando las cosas que decidimos olvidar antes. Luego le sumamos $i_t * \tilde{C}_t$. Estos son los nuevos valores candidatos, ajustados por cuánto decidimos actualizar cada valor de estado.

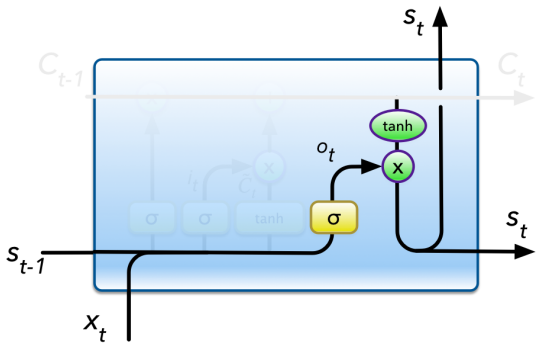
En el caso del modelo de lenguaje, aquí es donde dejamos caer la información sobre el género del sujeto anterior y agregamos la nueva información, como decidimos en los pasos anteriores.



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Finalmente, tenemos que decidir qué vamos a generar. Este resultado se basará en el estado de celda, pero será una versión filtrada. Primero, ejecutamos una capa con activación sigmoide que decide qué partes del estado de la celda vamos a generar. Luego, ponemos el estado de la celda a través de una función `tanh` (para ajustar los valores entre -1 y 1) y posteriormente se multiplican por la salida de la compuerta sigmoide, de modo que solo se genererarán las partes que decidamos.

Para el ejemplo del modelo de lenguaje, dado que acaba de ver un tema, es posible que desee generar información relevante para un verbo, en caso de que eso sea lo que viene a continuación. Por ejemplo, podría mostrar si el sujeto es singular o plural, de modo que sabemos en qué forma debe conjugarse un verbo si eso es lo que sigue a continuación.



$$o_t = \sigma(W_0[h_{t-1}, x_t] + b_0)$$

$$s_t = o_t * tanh(C_t)$$

Ejemplo de una Red Neuronal Recurrente

Para este ejemplo, utilizaremos el conjunto de datos de reseñas de clientes de Amazon que se puede encontrar en Kaggle. El conjunto de datos contiene un total de 4 millones de reseñas y cada reseña está etiquetada como de sentimiento positivo o negativo.

Nuestro objetivo al momento de esta implementación será crear un modelo LSTM que pueda clasificar y distinguir con precisión el sentimiento de una reseña. Para hacerlo, tendremos que comenzar con un preprocesamiento de datos, definir y entrenar el modelo, seguido de una evaluación del modelo.

Preprocesamiento de los datos

Para nuestros pasos de preprocesamiento de datos, usaremos expresiones regulares `re`, `numpy` y la librería `NLTK` (Natural Language Toolkit) para algunas funciones simples de ayuda de Procesamiento de Lenguaje Natural (PLN). Como los datos están comprimidos en el formato *bz2*, usaremos el módulo `bz2` de Python para leer los datos.

In [1]:
```
import bz2
from collections import Counter
import re
import nltk
import numpy as np
nltk.download('punkt')
```

```
[nltk_data] Downloading package punkt to /Users/wladimir/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
```

Out[1]:
```
True
```

El paquete `punkt` es una parte importante del proceso de *tokenización*. Por lo tanto, puede usar el administrador de descargas nltk para obtenerlo o usar la función de `download('punkt')` en la biblioteca `nltk` para obtenerlo a través del código.

In [2]:
```
archivo_entrenamiento = bz2.BZ2File('../datos/amazon_reviews/train.ft.txt.bz
archivo_prueba = bz2.BZ2File('../datos/amazon_reviews/test.ft.txt.bz2')
```

In [3]:
```
archivo_entrenamiento = archivo_entrenamiento.readlines()
archivo_prueba = archivo_prueba.readlines()
```

In [4]:
```
print("Número de reseñas de entrenamiento: " + str(len(archivo_entrenamientc
print("Número de reseñas de prueba: " + str(len(archivo_prueba)))
```

```
Número de reseñas de entrenamiento: 3600000
Número de reseñas de prueba: 400000
```

Este conjunto de datos contiene un total de 4 millones de reseñas: 3,6 millones de entrenamiento y 0,4 millones de prueba. No usaremos todo el conjunto de datos para ahorrar tiempo. Sin embargo, si tiene el poder y la capacidad de cómputo, continúe y entrene el modelo en una porción más grande de datos.

In [5]:
```
numero_entrenamiento = 800000 #Entrenaremos con las primeras 800.000 reseñas
numero_prueba = 200000 #Usaremos las primeras 200.000 reseñas

archivo_entrenamiento = [x.decode('utf-8') for x in archivo_entrenamiento[:n
archivo_prueba = [x.decode('utf-8') for x in archivo_prueba[:numero_prueba]]
```

In [6]:
```
print(archivo_entrenamiento[0])
```

```
__label__2 Stuning even for the non-gamer: This sound track was beautiful! I
t paints the senery in your mind so well I would recomend it even to people
who hate vid. game music! I have played the game Chrono Cross but out of all
of the games I have ever played it has the best music! It backs away from cr
ude keyboarding and takes a fresher step with grate guitars and soulful orch
estras. It would impress anyone who cares to listen! ^_^
```

A continuación, tendremos que extraer las etiquetas de las oraciones. Los datos tienen el formato `__label__1/2 <oración>`, por lo que podemos dividirlos fácilmente en consecuencia. Las etiquetas de opiniones positivas se almacenan como 1 y las negativas como 0.

También cambiaremos todas las URL a una "" estándar, ya que la URL exacta es irrelevante para el sentimiento en la mayoría de los casos.

In [7]:
```
# Extrayendo etiquetas de las oraciones

etiquetas_entrenamiento = [0 if x.split(' ')[0] == '__label__1' else 1 for x
oraciones_entrenamiento = [x.split(' ', 1)[1][:-1].lower() for x in archivo_


etiquetas_prueba = [0 if x.split(' ')[0] == '__label__1' else 1 for x in arc
oraciones_prueba = [x.split(' ', 1)[1][:-1].lower() for x in archivo_prueba]

# Una limpeza de datos sencilla

for i in range(len(oraciones_entrenamiento)):
    oraciones_entrenamiento[i] = re.sub('\d','0',oraciones_entrenamiento[i])

for i in range(len(oraciones_prueba)):
    oraciones_prueba[i] = re.sub('\d','0',oraciones_prueba[i])

# Modificar las URL a <url>

for i in range(len(oraciones_entrenamiento)):
    if 'www.' in oraciones_entrenamiento[i] or 'http:' in oraciones_entrenam
        oraciones_entrenamiento[i] = re.sub(r"([^ ]+(?<=\.[a-z]{3}))", "<url

for i in range(len(oraciones_prueba)):
    if 'www.' in oraciones_prueba[i] or 'http:' in oraciones_prueba[i] or 'h
        oraciones_prueba[i] = re.sub(r"([^ ]+(?<=\.[a-z]{3}))", "<url>", ora
```

In [8]:
```
del archivo_entrenamiento, archivo_prueba
```

Después de limpiar rápidamente los datos, haremos la tokenización de las oraciones, que es una tarea estándar de PNL. La tokenización es la tarea de dividir una oración en tokens individuales, que pueden ser palabras o puntuación, etc. Hay muchas bibliotecas NLP que pueden hacer esto, como *spaCy* o *Scikit-learn*, pero aquí usaremos *NLTK* ya que tiene uno de los tokenizadores más rápidos.

Luego, las palabras se almacenarán en un diccionario asignando la palabra a su número de apariciones. Estas palabras se convertirán en nuestro **vocabulario**.

In [9]:
```
palabras = Counter() #Diccionario que mapea una palabra a el número de veces
for i, oracion in enumerate(oraciones_entrenamiento):
    #Las oraciones se almacenarán en una lista de palabras/tokens
    oraciones_entrenamiento[i] = []
    for palabra in nltk.word_tokenize(oracion): #Tokenización de palabras
        palabras.update([palabra.lower()]) #Convertir todas las palabras a m
        oraciones_entrenamiento[i].append(palabra)
    if i%20000 == 0:
        print(str((i*100)/numero_entrenamiento) + "% hecho")
print("100% hecho")
```

```
0.0% hecho
2.5% hecho
5.0% hecho
7.5% hecho
10.0% hecho
12.5% hecho
15.0% hecho
17.5% hecho
20.0% hecho
22.5% hecho
25.0% hecho
27.5% hecho
30.0% hecho
32.5% hecho
35.0% hecho
37.5% hecho
40.0% hecho
42.5% hecho
45.0% hecho
47.5% hecho
50.0% hecho
52.5% hecho
55.0% hecho
57.5% hecho
60.0% hecho
62.5% hecho
65.0% hecho
67.5% hecho
70.0% hecho
72.5% hecho
75.0% hecho
77.5% hecho
80.0% hecho
82.5% hecho
85.0% hecho
87.5% hecho
90.0% hecho
92.5% hecho
95.0% hecho
97.5% hecho
100% hecho
```

Para eliminar errores tipográficos y palabras que probablemente no existan, eliminaremos todas las palabras del vocabulario que solo aparecen una vez. Para tener en cuenta las palabras **desconocidas** y el **relleno**, también tendremos que agregarlas a nuestro vocabulario. A cada palabra del vocabulario se le asignará un índice de número entero y, a partir de entonces, se asignará a este número entero.

In [10]:
```
# Remover las palabras que solo aparecen una vez
palabras = {k:v for k,v in palabras.items() if v>1}
# Ordenar las palabras de acuerdo al número de apariciones
palabras = sorted(palabras, key=palabras.get, reverse=True) #De mayor a menc
# Agregar padding and unknown a nuestro vocabulario para asignarles un indic
palabras = ['_PAD','_UNK'] + palabras
# Dictionaries to store the word to index mappings and vice versa
palabra2idx = {o:i for i,o in enumerate(palabras)}
idx2palabra = {i:o for i,o in enumerate(palabras)}
```

Con las asignaciones, convertiremos las palabras de las oraciones a sus índices correspondientes.

In [11]:
```
for i, oracion in enumerate(oraciones_entrenamiento):
    # Asignar el indice alas palabras respectivas
    oraciones_entrenamiento[i] = [palabra2idx[palabra] if palabra in palabra

for i, oracion in enumerate(oraciones_prueba):
    # Para las oraciones de prueba, tenemos que tokenizar las oraciones tamb
    oraciones_prueba[i] = [palabra2idx[palabra.lower()] if palabra.lower() i
```

En el último paso de preprocesamiento, rellenaremos las oraciones con 0 y acortaremos las oraciones largas para que los datos se puedan entrenar en lotes para acelerar las cosas.

In [12]:
```
# Definir una función que o acorte la sentencia o agregue 0 (padding) a un l

def pad_entrada(oraciones, largo_secuencia):
    atributos = np.zeros((len(oraciones), largo_secuencia),dtype=int)
    for ii, review in enumerate(oraciones):
        if len(review) != 0:
            atributos[ii, -len(review):] = np.array(review)[:largo_secuencia
    return atributos
```

In [13]:
```
largo_secuencia = 200

oraciones_entrenamiento = pad_entrada(oraciones_entrenamiento, largo_secuenc
oraciones_prueba = pad_entrada(oraciones_prueba, largo_secuencia)
```

In [14]:
```
# Convertir nuestras etiquetas a arreglos numpy
etiquetas_entrenamiento = np.array(etiquetas_entrenamiento)
etiquetas_prueba = np.array(etiquetas_prueba)
```

Una oración con relleno se verá así, donde 0 representa el relleno:

In [15]:
```
oraciones_prueba[0]
```

Out[15]:
```
array([    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,     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,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,    40,    99,    13,    28,  1447,  4272,    57,
          31,    10,     3,    40,  1781,    10,    85,  1730,     2,
           5,    27,   907,     8,    11,    99,    16,   151,     6,
           5,   140,    90,     9,     2,    68,     5,   122,    14,
           7,    42,  1847,     9,   210,    58,   243,   108,     2,
           7,   133,  1847,    46, 29316,    38,  2642,    14,     3,
        2379,     2,    11,    99,    46, 18846,   160,     2,   934,
          30,     1,     1,     6,   560,    46,  1285,     2,    31,
          10,   160,    21,  2336,  4156,     2,    11,    12,     7,
        3570, 14981,    99,    14,    28,    24,     2,   182,   102,
         130,   147,     9,   239,    12,    46,   827,    58,     2,
        2587,     5,   263,    11,     4,    72,   601,   444,     4,
         579,     4,   416,     4,   153,     4,  1689,     4,  1255,
        1816,   521,    31,   179,    33,    80,    18,    17,   829,
          61,    32])
```

Nuestro conjunto de datos ya está dividido en datos de *entrenamiento* y *pruebas*. Sin embargo, todavía necesitamos un conjunto de datos para la validación durante el entrenamiento. Por lo tanto, dividiremos nuestros datos de prueba a la mitad en un conjunto de validación y un conjunto de prueba. Se puede encontrar una explicación detallada sobre las divisiones de conjuntos de datos aquí.

In [16]:
```
frac_division = 0.5
id_division = int(frac_division * len(oraciones_prueba))
oraciones_validacion, oraciones_prueba = oraciones_prueba[:id_division], ora
etiquetas_validacion, etiquetas_prueba = etiquetas_prueba[:id_division], eti
```

Convertir los datos a `Datasets` y `DataLoader` de PyTorch

A continuación, comenzaremos a trabajar con la librería PyTorch. Primero definiremos los conjuntos de datos de las oraciones y las etiquetas, y luego los cargaremos en un cargador de datos. Establecemos el tamaño del lote en 256. Esto se puede modificar según sus necesidades.

In [17]:
```
import torch
from torch.utils.data import TensorDataset, DataLoader

datos_entrenamiento = TensorDataset(torch.from_numpy(oraciones_entrenamientc
datos_validacion = TensorDataset(torch.from_numpy(oraciones_validacion), tor
datos_prueba = TensorDataset(torch.from_numpy(oraciones_prueba), torch.from_

tamaño_lote = 400

cargador_entrenamiento = DataLoader(datos_entrenamiento, shuffle=True, batch
cargador_validacion = DataLoader(datos_validacion, shuffle=True, batch_size=
cargador_prueba = DataLoader(datos_prueba, shuffle=True, batch_size=tamaño_l
```

También podemos verificar si tenemos alguna GPU para acelerar nuestro tiempo de entrenamiento en muchos pliegues.

In [18]:
```
if torch.backends.mps.is_available():
    dispositivo = 'mps'
elif torch.cuda.is_available():
    dispositivo = "cuda"
else: "cpu"
dispositivo
```

Out[18]:
```
'mps'
```

In [19]:
```
iterador_datos = iter(cargador_entrenamiento)
ejemplo_x, ejemplo_y = iterador_datos.next()

print(ejemplo_x.shape, ejemplo_y.shape)
```

```
torch.Size([400, 200]) torch.Size([400])
```

Construir el Modelo

En este punto, definiremos la arquitectura del modelo. En esta etapa, podemos crear redes neuronales que tengan capas profundas o una gran cantidad de capas LSTM apiladas una encima de la otra. Sin embargo, un modelo simple como el que se muestra a continuación funciona bastante bien y requiere mucho menos tiempo de entrenamiento. Estaremos entrenando nuestras propios *words embedings* de palabras en la primera capa antes de que las oraciones se introduzcan en la capa LSTM.

La capa final es una capa totalmente conectada con una función sigmoidea para clasificar si la reseña es de opinión positiva o negativa.

In [20]:
```
import torch.nn as nn

class RedAnalisisSentimientos(nn.Module):
    def __init__(self, tamaño_vocabulario, tamaño_salida, dim_embedding, dim
        super(RedAnalisisSentimientos, self).__init__()
        self.tamaño_salida = tamaño_salida
        self.numero_capas = numero_capas
        self.dim_oculta = dim_oculta

        self.embedding = nn.Embedding(tamaño_vocabulario, dim_embedding)
        self.lstm = nn.LSTM(dim_embedding, dim_oculta, numero_capas, dropout
        self.dropout = nn.Dropout(0.2)
        self.fc = nn.Linear(dim_oculta, tamaño_salida)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x, oculta):
        tamaño_lote = x.size(0)
        x = x.long()
        embeds = self.embedding(x)
        lstm_salida, oculta = self.lstm(embeds, oculta)
        lstm_salida = lstm_salida.contiguous().view(-1, self.dim_oculta)

        salida = self.dropout(lstm_salida)
        salida = self.fc(salida)
        salida = self.sigmoid(salida)

        salida = salida.view(tamaño_lote, -1)
        salida = salida[:,-1]
        return salida, oculta

    def inicializar_oculta(self, tamaño_lote):
        pesos = next(self.parameters()).data
        oculta = (pesos.new(self.numero_capas, tamaño_lote, self.dim_oculta)
                  pesos.new(self.numero_capas, tamaño_lote, self.dim_ocu
        return oculta
```

Tenga en cuenta que en realidad podemos cargar *word embeddings* de palabras previamente entrenadas, como GloVe o fastText, que pueden aumentar la precisión del modelo y disminuir el tiempo de entrenamiento.

Con esto, podemos instanciar nuestro modelo después de definir los argumentos. La dimensión de salida solo será 1, ya que solo necesita generar 1 o 0. La tasa de aprendizaje, la función de pérdida y el optimizador también serán definidos.

In [21]:
```
tamaño_vocabulario = len(palabra2idx) + 1
tamaño_salida = 1
dim_embedding = 400
dim_oculta = 512
numero_capas = 2

modelo = RedAnalisisSentimientos(tamaño_vocabulario, tamaño_salida, dim_embe
modelo.to(dispositivo)
print(modelo)
```

```
RedAnalisisSentimientos(
  (embedding): Embedding(221606, 400)
  (lstm): LSTM(400, 512, num_layers=2, batch_first=True, dropout=0.5)
  (dropout): Dropout(p=0.2, inplace=False)
  (fc): Linear(in_features=512, out_features=1, bias=True)
  (sigmoid): Sigmoid()
)
```

In [22]:
```
taza_aprendizaje = 0.005
fn_perdida = nn.BCELoss()
optimizador = torch.optim.Adam(modelo.parameters(), lr=taza_aprendizaje)
```

Finalmente, podemos comenzar a entrenar el modelo. Por cada 1000 pasos, verificaremos el resultado de nuestro modelo con el conjunto de datos de validación y guardaremos el modelo si se desempeñó mejor que la vez anterior. El `state_dict` son los pesos del modelo en PyTorch y se puede posteriormente cargar en un modelo con la misma arquitectura.

In [ ]:
```
epocas = 2
contador = 0
imprimir_cada = 1000
clip = 5
perdida_min_validacion = np.Inf

modelo.train()
for i in range(epocas):
    h = modelo.inicializar_oculta(tamaño_lote)

    for entradas, etiquetas in cargador_entrenamiento:
        contador += 1
        h = tuple([e.data for e in h])
        entradas, etiquetas = entradas.to(dispositivo), etiquetas.to(disposi
        modelo.zero_grad()
        salida, h = modelo(entradas, h)
        perdida = fn_perdida(salida.squeeze(), etiquetas.float())
        perdida.backward()
        nn.utils.clip_grad_norm_(modelo.parameters(), clip)
        optimizador.step()

        if contador%imprimir_cada == 0:
            h_val = modelo.inicializar_oculta(tamaño_lote)
            perdidas_val = []
            modelo.eval()
            for entrada, etiqueta in cargador_validacion:
                h_val = tuple([cada.data for cada in h_val])
                entrada, etiqueta = entrada.to(dispositivo), etiqueta.to(dis
                salida, h_val = modelo(entrada, h_val)
                perdida_val = fn_perdida(salida.squeeze(), etiqueta.float())
                perdidas_val.append(perdida_val.item())

            modelo.train()
            print(f'Epoca: {i+1}/{epocas}...',
                  f'Paso: {contador}...',
                  f'Perdida: {perdida.item():.6f}...',
                  f'Perdida Val: {np.mean(perdidas_val):.6f}')
            if np.mean(perdidas_val) <= perdida_min_validacion:
                torch.save(modelo.state_dict(), '../modelos/RedAnalisisSenti
                print(f'Validation loss decreased ({perdida_min_validacion:.
                perdida_min_validacion = np.mean(perdidas_val)
```

Una vez que hayamos terminado el entrenamiento, es hora de probar nuestro modelo en un conjunto de datos que nunca antes había visto: nuestro conjunto de datos de prueba. Primero cargaremos los pesos del modelo desde el punto donde la pérdida de validación es la más baja. Podemos calcular la precisión del modelo para ver qué tan precisas son las predicciones de nuestro modelo.

In [31]:
```
#Cargar el mejor modelo
modelo.load_state_dict(torch.load('../modelos/RedAnalisisSentimientos.pt', m
```

Out[31]:
```
<All keys matched successfully>
```

In [34]:
```
perdidas_prueba = []
numero_correctos = 0
h = modelo.inicializar_oculta(tamaño_lote)

modelo.eval()
for entradas, etiquetas in cargador_prueba:
    h = tuple([cada.data for cada in h])
    entradas, etiquetas = entradas.to(dispositivo), etiquetas.to(dispositivo
    salida, h = modelo(entradas, h)
    perdida_prueba = fn_perdida(salida.squeeze(), etiquetas.float())
    perdidas_prueba.append(perdida_prueba.item())
    prediccion = torch.round(salida.squeeze()) #rounds the output to 0/1
    tensor_correcto = prediccion.eq(etiquetas.float().view_as(prediccion))
    correcto = np.squeeze(tensor_correcto.cpu().numpy())
    numero_correctos += np.sum(correcto)

print(f'Perdida prueba: {np.mean(perdidas_prueba):.3f}')
exactitud_prueba = numero_correctos/len(cargador_prueba.dataset)
print(f'Exactitud prueba: {exactitud_prueba*100:.3f}%')
```

```
Perdida prueba: 0.737
Exactitud prueba: 47.654%
```

In [ ]: