

Laboratorio 3

Sean bienvenidos de nuevo al laboratorio 3 de Deep Learning y Sistemas Inteligentes. Así como en los laboratorios pasados, espero que esta ejercitación les sirva para consolidar sus conocimientos en el tema de Redes Neuronales Recurrentes y LSTM.

Este laboratorio consta de dos partes. En la primera trabajaremos una Red Neuronal Recurrente paso-a-paso. En la segunda fase, usaremos PyTorch para crear una nueva Red Neuronal pero con LSTM, con la finalidad de que no solo sepan que existe cierta función sino también entender qué hace en un poco más de detalle.

Para este laboratorio estaremos usando una herramienta para Jupyter Notebooks que facilitará la calificación, no solo asegurando que ustedes tengan una nota pronto sino también mostrándoles su nota final al terminar el laboratorio.

Espero que esta vez si se muestren los *marks*. De nuevo me discupo si algo no sale bien, seguiremos mejorando conforme vayamos iterando. Siempre pido su comprensión y colaboración si algo no funciona como debería.

Al igual que en el laboratorio pasado, estaremos usando la librería de Dr John Williamson et al de la University of Glasgow, además de ciertas piezas de código de Dr Bjorn Jensen de su curso de Introduction to Data Science and System de la University of Glasgow para la visualización de sus calificaciones.

NOTA: Ahora tambien hay una tercera dependencia que se necesita instalar. Ver la celda de abajo por favor

```
In [44]: # Una vez instalada la librería por favor, recuerden volverla a comentar.  
#!pip install -U --force-reinstall --no-cache https://github.com/johnhw/jhwutils/zipball/master  
#!pip install scikit-image  
#!pip install -U --force-reinstall --no-cache https://github.com/AlbertS789/lautils
```

```
In [45]: import numpy as np  
import copy  
import matplotlib.pyplot as plt  
import scipy  
from PIL import Image  
import os  
from collections import defaultdict  
  
#from IPython import display  
#from base64 import b64decode  
  
# Other imports  
from unittest.mock import patch
```

```

from uuid import getnode as get_mac

from jhwutils.checkarr import array_hash, check_hash, check_scalar, check_string, a
import jhwutils.image_audio as ia
import jhwutils.tick as tick
from lautils.gradeutils import new_representation, hex_to_float, compare_numbers, c

###
tick.reset_marks()

%matplotlib inline

```

```

In [46]: # Seeds
seed_ = 2023
np.random.seed(seed_)

```

```

In [47]: # Celda escondida para utlidades necesarias, por favor NO edite esta celda

```

Información del estudiante en dos variables

- carne_1 : un string con su carne (e.g. "12281"), debe ser de al menos 5 caracteres.
- firma_mecanografiada_1: un string con su nombre (e.g. "Albero Suriano") que se usará para la declaracion que este trabajo es propio (es decir, no hay plagio)
- carne_2 : un string con su carne (e.g. "12281"), debe ser de al menos 5 caracteres.
- firma_mecanografiada_2: un string con su nombre (e.g. "Albero Suriano") que se usará para la declaracion que este trabajo es propio (es decir, no hay plagio)

```

In [48]: carne_1 = "22397"
firma_mecanografiada_1 = "Josue Marroquin"
carne_2 = "22295"
firma_mecanografiada_2 = "Sebastian Huertas"
# YOUR CODE HERE

```

```

In [49]: # Deberia poder ver dos checkmarks verdes [0 marks], que indican que su información

with tick.marks(0):
    assert(len(carne_1)>=5 and len(carne_2)>=5)

with tick.marks(0):
    assert(len(firma_mecanografiada_1)>0 and len(firma_mecanografiada_2)>0)

```

✓ [0 marks]

✓ [0 marks]

Parte 1 - Construyendo una Red Neuronal Recurrente

Créditos: La primera parte de este laboratorio está tomado y basado en uno de los laboratorios dados dentro del curso de "Deep Learning" de Jes Frellsen (DeepLearningDTU)

La aplicación de los datos secuenciales pueden ir desde predicción del clima hasta trabajar con lenguaje natural. En este laboratorio daremos un vistazo a como las RNN pueden ser usadas dentro del modelaje del lenguaje, es decir, trataremos de predecir el siguiente token dada una secuencia. En el campo de NLP, un token puede ser un caracter o bien una palabra.

Representación de Tokens o Texto

Como bien hemos hablado varias veces, la computadora no entiende palabras ni mucho menos oraciones completas en la misma forma que nuestros cerebros lo hacen. Por ello, debemos encontrar alguna forma de representar palabras o caracteres en una manera que la computadora sea capaz de interpretarla, es decir, con números. Hay varias formas de representar un grupo de palabras de forma numérica, pero para fines de este laboratorio vamos a centrarnos en una manera común, llamada "one-hot encoding".

One Hot Encoding

Esta técnica debe resultarles familiar de cursos pasados, donde se tomaba una conjunto de categorías y se les asignaba una columna por categoría, entonces se coloca un 1 si el row que estamos evaluando es parte de esa categoría o un 0 en caso contrario. Este mismo acercamiento podemos tomarlo para representar conjuntos de palabras. Por ejemplo

```
casa = [1, 0, 0, ..., 0]
perro = [0, 1, 0, ..., 0]
```

Representar un vocabulario grande con one-hot encoding, suele volverse ineficiente debido al tamaño de cada vector disperso. Para solventar esto, una práctica común es truncar el vocabulario para contener las palabras más utilizadas y representar el resto con un símbolo especial, UNK, para definir palabras "desconocidas" o "sin importancia". A menudo esto se hace que palabras tales como nombres se vean como UNK porque son raros.

Generando el Dataset a Usar

Para este laboratorio usaremos un dataset simplificado, del cual debería ser más sencillo el aprender de él. Estaremos generando secuencias de la forma

```
a b EOS
a a a a b b b b EOS
```

Noten la aparición del token "EOS", el cual es un caracter especial que denota el fin de la secuencia. Nuestro task en general será el predecir el siguiente token t_n , donde este podrá ser "a", "b", "EOS", o "UNK" dada una secuencia de forma t_1, \dots, t_{n-1} .

```
In [50]: # Reseed the cell
np.random.seed(seed_)

def generate_data(num_seq=100):
    """
    Genera un grupo de secuencias, la cantidad de secuencias es dada por num_seq

    Args:
    num_seq: El número de secuencias a ser generadas

    Returns:
    Una lista de secuencias
    """
    samples = []
    for i in range(num_seq):
        # Genera una secuencia de largo aleatorio
        num_tokens = np.random.randint(1,12)
        # Genera la muestra
        sample = ['a'] * num_tokens + ['b'] * num_tokens + ['EOS']
        # Agregamos
        samples.append(sample)
    return samples

sequences = generate_data()
print("Una secuencia del grupo generado")
print(sequences[0])
```

Una secuencia del grupo generado

```
['a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'EOS']
```

Representación de tokens como índices

En este paso haremos la parte del one-hot encoding. Para esto necesitaremos asignar a cada posible palabra de nuestro vocabulario un índice. Para esto crearemos dos diccionarios, uno que permitirá que dada una palabra nos dirá su representación como "índice" en el vocabulario, y el segundo que irá en dirección contraria.

A estos les llamaremos `word_to_idx` y `idx_to_word`. La variable `vocab_size` nos dirá el máximo de tamaño de nuestro vocabulario. Si intentamos acceder a una palabra que no está en nuestro vocabulario, entonces se le reemplazará con el token "UNK" o su índice correspondiente.

```
In [51]: def seqs_to_dicts(sequences):
    """
    Crea word_to_idx y idx_to_word para una lista de secuencias
```

Args:
sequences: lista de secuencias a usar

Returns:
Diccionario de palabra a indice
Diccionario de indice a palabra
Int numero de secuencias
Int tamaño del vocabulario
"""

Lambda para aplanar (flatten) una lista de listas

```
flatten = lambda l: [item for sublist in l for item in sublist]
```

Aplanamos el dataset

```
all_words = flatten(sequences)
```

Conteo de las ocurrencias de las palabras

```
word_count = defaultdict(int)
```

```
for word in all_words:
    word_count[word] += 1
```

Ordenar por frecuencia

```
word_count = sorted(list(word_count.items()), key=lambda x: -x[1])
```

Crear una lista de todas las palabras únicas

```
unique_words = [w[0] for w in word_count]
```

Agregamos UNK a la lista de palabras

```
unique_words.append("UNK")
```

Conteo del número de secuencias y el número de palabras únicas

```
num_sentences, vocab_size = len(sequences), len(unique_words)
```

Crear diccionarios mencionados

```
word_to_idx = defaultdict(lambda: vocab_size-1)
```

```
idx_to_word = defaultdict(lambda: 'UNK')
```

Llenado de diccionarios

```
for idx, word in enumerate(unique_words):
```

```
    # Aprox 2 líneas para agregar
```

```
    word_to_idx[word] = idx
```

```
    idx_to_word[idx] = word
```

```
    # YOUR CODE HERE
```

```
return word_to_idx, idx_to_word, num_sentences, vocab_size
```

```
word_to_idx, idx_to_word, num_sequences, vocab_size = seqs_to_dicts(sequences)
```

```
print(f"Tenemos {num_sequences} secuencias y {len(word_to_idx)} tokens unicos inclu
```

```
print(f"El indice de 'b' es {word_to_idx['b']}")
```

```
print(f"La palabra con indice 1 es {idx_to_word[1]}")
```

Tenemos 100 secuencias y 4 tokens unicos incluyendo UNK

El indice de 'b' es 1

La palabra con indice 1 es b

```
In [52]: with tick.marks(3):
          assert(check_scalar(len(word_to_idx), '0xc51b9ba8'))

          with tick.marks(2):
              assert(check_scalar(len(idx_to_word), '0xc51b9ba8'))

          with tick.marks(5):
              assert(check_string(idx_to_word[0], '0xe8b7be43'))
```

✓ [3 marks]

✓ [2 marks]

✓ [5 marks]

Representación de tokens como índices

Como bien sabemos, necesitamos crear nuestro dataset de forma que el se divida en inputs y targets para cada secuencia y luego particionar esto en training, validation y test (80%, 10%, 10%). Debido a que estamos haciendo prediccion de la siguiente palabra, nuestro target es el input movido (shifted) una palabra.

Vamos a usar PyTorch solo para crear el dataset (como lo hicimos con las imagenes de perritos y gatitos de los laboratorios pasados). Aunque esta vez no haremos el dataloader. Recuerden que siempre es buena idea usar un DataLoader para obtener los datos de una forma eficiente, al ser este un generador/iterador. Además, este nos sirve para obtener la información en batches.

```
In [53]: from torch.utils import data

class Dataset(data.Dataset):
    def __init__(self, inputs, targets):
        self.inputs = inputs
        self.targets = targets

    def __len__(self):
        # Return the size of the dataset
        return len(self.targets)

    def __getitem__(self, index):
        # Retrieve inputs and targets at the given index
        X = self.inputs[index]
```

```

        y = self.targets[index]

        return X, y

def create_datasets(sequences, dataset_class, p_train=0.8, p_val=0.1, p_test=0.1):

    # Definimos el tamaño de las particiones
    num_train = int(len(sequences)*p_train)
    num_val = int(len(sequences)*p_val)
    num_test = int(len(sequences)*p_test)

    # Dividir las secuencias en las particiones
    sequences_train = sequences[:num_train]
    sequences_val = sequences[num_train:num_train+num_val]
    sequences_test = sequences[-num_test:]

    # Funcion interna para obtener los targets de una secuencia
    def get_inputs_targets_from_sequences(sequences):
        # Listas vacias
        inputs, targets = [], []

        # Agregar informacion a las listas, ambas listas tienen L-1 palabras de una
        # pero los targets están movidos a la derecha por uno, para que podamos pr
        for sequence in sequences:
            inputs.append(sequence[:-1])
            targets.append(sequence[1:])

        return inputs, targets

    # Obtener inputs y targes para cada subgrupo
    inputs_train, targets_train = get_inputs_targets_from_sequences(sequences_train)
    inputs_val, targets_val = get_inputs_targets_from_sequences(sequences_val)
    inputs_test, targets_test = get_inputs_targets_from_sequences(sequences_test)

    # Creación de datasets
    training_set = dataset_class(inputs_train, targets_train)
    validation_set = dataset_class(inputs_val, targets_val)
    test_set = dataset_class(inputs_test, targets_test)

    return training_set, validation_set, test_set

training_set, validation_set, test_set = create_datasets(sequences, Dataset)

print(f"Largo del training set {len(training_set)}")
print(f"Largo del validation set {len(validation_set)}")
print(f"Largo del test set {len(test_set)}")

```

Largo del training set 80
 Largo del validation set 10
 Largo del test set 10

One-Hot Encodings

Ahora creemos una función simple para obtener la representación one-hot encoding de dado un índice de una palabra. Noten que el tamaño del one-hot encoding es igual a la del vocabulario. Adicionalmente definamos una función para encodear una secuencia.

```
In [54]: def one_hot_encode(idx, vocab_size):
        """
        Encodea una sola palabra dado su índice y el tamaño del vocabulario

        Args:
            idx: índice de la palabra
            vocab_size: tamaño del vocabulario

        Returns
        np.array de lagro "vocab_size"
        """
        # Init array encodeado
        one_hot = np.zeros(vocab_size)

        # Setamos el elemento a uno
        one_hot[idx] = 1.0

        return one_hot

def one_hot_encode_sequence(sequence, vocab_size):
    """
    Encodea una secuencia de palabras dado el tamaño del vocabulario

    Args:
        sentence: una lista de palabras a encodear
        vocab_size: tamaño del vocabulario

    Returns
    np.array 3D de tamaño (numero de palabras, vocab_size, 1)
    """
    # Encodear cada palabra en la secuencia
    encoding = np.array([one_hot_encode(word_to_idx[word], vocab_size) for word in
                        sequence])

    # Cambiar de forma para tener (num words, vocab size, 1)
    encoding = encoding.reshape(encoding.shape[0], encoding.shape[1], 1)

    return encoding

test_word = one_hot_encode(word_to_idx['a'], vocab_size)
print(f"Encodeado de 'a' con forma {test_word.shape}")

test_sentence = one_hot_encode_sequence(['a', 'b'], vocab_size)
print(f"Encodeado de la secuencia 'a b' con forma {test_sentence.shape}.")
```

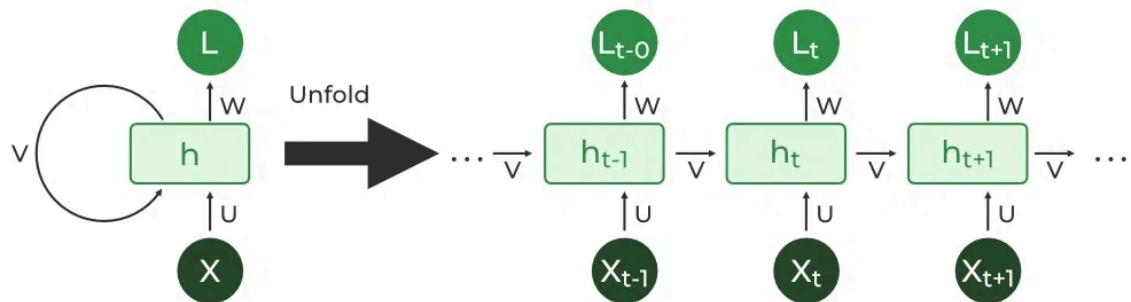
Encodeado de 'a' con forma (4,)

Encodeado de la secuencia 'a b' con forma (2, 4, 1).

Ahora que ya tenemos lo necesario de data para empezar a trabajar, demos paso a hablar un poco más de las RNN

Redes Neuronales Recurrentes (RNN)

Una red neuronal recurrente (RNN) es una red neuronal conocida por modelar de manera efectiva datos secuenciales como el lenguaje, el habla y las secuencias de proteínas. Procesa datos de manera cíclica, aplicando los mismos cálculos a cada elemento de una secuencia. Este enfoque cíclico permite que la red utilice cálculos anteriores como una forma de memoria, lo que ayuda a hacer predicciones para cálculos futuros. Para comprender mejor este concepto, consideren la siguiente imagen.



Crédito de imagen al autor, imagen tomada de "Introduction to Recurrent Neural Network" de Aishwarya.27

Donde:

- x es la secuencia de input
- U es una matriz de pesos aplicada a una muestra de input dada
- V es una matriz de pesos usada para la computación recurrente para pasar la memoria en las secuencias
- W es una matriz de pesos usada para calcular la salida de cada paso
- h es el estado oculto (hidden state) (memoria de la red) para cada paso
- L es la salida resultante

Cuando una red es extendida como se muestra, es más fácil referirse a un paso t . Tenemos los siguientes cálculos en la red

- $h_t = f(Ux_t + Vh_{t-1})$ donde f es la función de activación
- $L_t = \text{softmax}(Wh_t)$

Implementando una RNN

Ahora pasaremos a inicializar nuestra RNN. Los pesos suelen inicializar de forma aleatoria, pero esta vez lo haremos de forma ortogonal para mejorar el rendimiento de nuestra red, y siguiendo las recomendaciones del paper dado abajo.

Tenga cuidado al definir los elementos que se le piden, debido a que una mala dimensión causará que tenga resultados diferentes y errores al operar.

```
In [55]: np.random.seed(seed_)

hidden_size = 50 # Numero de dimensiones en el hidden state
vocab_size = len(word_to_idx) # Tamaño del vocabulario

def init_orthogonal(param):
    """
    Initializes weight parameters orthogonally.
    Inicializa los pesos ortogonalmente

    Esta inicialización está dada por el siguiente paper:
    https://arxiv.org/abs/1312.6120
    """
    if param.ndim < 2:
        raise ValueError("Only parameters with 2 or more dimensions are supported.")

    rows, cols = param.shape

    new_param = np.random.randn(rows, cols)

    if rows < cols:
        new_param = new_param.T

    # Calcular factorización QR
    q, r = np.linalg.qr(new_param)

    # Hacer Q uniforme de acuerdo a https://arxiv.org/pdf/math-ph/0609050.pdf
    d = np.diag(r, 0)
    ph = np.sign(d)
    q *= ph

    if rows < cols:
        q = q.T

    new_param = q

    return new_param

def init_rnn(hidden_size, vocab_size):
    """
    Inicializa la RNN

    Args:
        hidden_size: Dimensiones del hidden state
        vocab_size: Dimensión del vocabulario
    """
```

```

# Inicialización con ceros (para definir forma)
U = np.zeros((hidden_size, vocab_size))
V = np.zeros((hidden_size, hidden_size))
W = np.zeros((vocab_size, hidden_size))
b_hidden = np.zeros((hidden_size, 1))
b_out = np.zeros((vocab_size, 1))

# Inicialización ortogonal
U = init_orthogonal(U)
V = init_orthogonal(V)
W = init_orthogonal(W)

return U, V, W, b_hidden, b_out

```

```
params = init_rnn(hidden_size=hidden_size, vocab_size=vocab_size)
```

```

In [56]: with tick.marks(5):
          assert check_hash(params[0], ((50, 4), 80.24369675632171))

          with tick.marks(5):
              assert check_hash(params[1], ((50, 50), 3333.838548574836))

          with tick.marks(5):
              assert check_hash(params[2], ((4, 50), -80.6410290517092))

          with tick.marks(5):
              assert check_hash(params[3], ((50, 1), 0.0))

          with tick.marks(5):
              assert check_hash(params[4], ((4, 1), 0.0))

```

✓ [5 marks]

✓ [5 marks]

✓ [5 marks]

✓ [5 marks]

✓ [5 marks]

Funciones de Activación

A continuación definiremos las funciones de activación a usar, sigmoide, tanh y softmax.

```
In [57]: def sigmoid(x, derivative=False):
    """
    Calcula la función sigmoide para un array x

    Args:
        x: El array sobre el que trabajar
        derivative: Si esta como verdadero, regresar el valor en la derivada
    """
    x_safe = x + 1e-12 #Evitar ceros
    # Aprox 1 linea sobre x_safe para implementar la funcion
    f = 1 / (1 + np.exp(-x_safe))
    # YOUR CODE HERE

    # Regresa la derivada de la funcion
    if derivative:
        return f * (1 - f)
    # Regresa el valor para el paso forward
    else:
        return f

def tanh(x, derivative=False):
    """
    Calcula la función tanh para un array x

    Args:
        x: El array sobre el que trabajar
        derivative: Si esta como verdadero, regresar el valor en la derivada
    """
    x_safe = x + 1e-12 #Evitar ceros
    # Aprox 1 linea sobre x_safe para implementar la funcion
    f = np.tanh(x_safe)
    # YOUR CODE HERE

    # Regresa la derivada de la funcion
    if derivative:
        return 1-f**2
    # Regresa el valor para el paso forward
    else:
        return f

def softmax(x, derivative=False):
    """
    Calcula la función softmax para un array x

    Args:
```

```

    x: El array sobre el que trabajar
    derivative: Si esta como verdadero, regresar el valor en la derivada
    """
    x_safe = x + 1e-12 #Evitar ceros
    # Aprox 1 linea sobre x_safe para implementar la funcion
    exp_x = np.exp(x_safe)
    f = exp_x / np.sum(exp_x)
    # YOUR CODE HERE

    # Regresa la derivada de la funcion
    if derivative:
        pass # No se necesita en backprop
    # Regresa el valor para el paso forward
    else:
        return f

```

```

In [58]: with tick.marks(5):
          assert check_hash(sigmoid(params[0][0]), ((4,)), 6.997641543410888))

          with tick.marks(5):
              assert check_hash(tanh(params[0][0]), ((4,)), -0.007401604025076086))

          with tick.marks(5):
              assert check_hash(softmax(params[0][0]), ((4,)), 3.504688021096135))

```

✓ [5 marks]

✓ [5 marks]

✓ [5 marks]

Implementación del paso Forward

Ahora es el momento de implementar el paso forward usando lo que hemos implementado hasta ahora

```

In [59]: def forward_pass(inputs, hidden_state, params):
          """
          Calcula el paso forward de RNN

          Args:
            inputs: Seccuencia de input a ser procesada
            hidden_state: Un estado inicializado hidden state
            params: Parametros de la RNN
          """

```

```

# Obtener Los parametros
U, V, W, b_hidden, b_out = params

# Crear una lista para guardar las salidas y los hidden states
outputs, hidden_states = [], []

# Para cada elemento en la secuencia input
for t in range(len(inputs)):
    x_t = inputs[t]
    # Aprox 1 line para
    # Calculo del nuevo hidden state usando tanh
    # Recuerden que al ser el hidden state tienen que usar los pesos del input
    # a esto sumarle los pesos recurrentes por el hidden state y finalmente su
    hidden_state = tanh(np.dot(U, x_t) + np.dot(V, hidden_state) + b_hidden)
    # YOUR CODE HERE

    # Aprox 1 linea
    # para el calculo del output
    # Al ser la salida, deben usar softmax sobre la multiplicación de pesos de
    # es decir el calculado en el paso anterior y siempre sumarle su bias con
    out = softmax(np.dot(W, hidden_state) + b_out)
    # YOUR CODE HERE

    # Guardamos los resultados y continuamos
    outputs.append(out)
    hidden_states.append(hidden_state.copy())

return outputs, hidden_states

```

```

In [60]: test_input_sequence, test_target_sequence = training_set[0]

# One-hot encode
test_input = one_hot_encode_sequence(test_input_sequence, vocab_size)
test_target = one_hot_encode_sequence(test_target_sequence, vocab_size)

# Init hidden state con zeros
hidden_state = np.zeros((hidden_size, 1))

outputs, hidden_states = forward_pass(test_input, hidden_state, params)

print("Secuencia Input:")
print(test_input_sequence)

print("Secuencia Target:")
print(test_target_sequence)

print("Secuencia Predicha:")
print([idx_to_word[np.argmax(output)] for output in outputs])

with tick.marks(5):
    assert check_hash(outputs, ((16, 4, 1), 519.7419046193046))

```

Secuencia Input:

['a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b']

Secuencia Target:

['a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'EOS']

Secuencia Predicha:

['a', 'b', 'a', 'a', 'a', 'EOS', 'EOS', 'EOS', 'EOS', 'EOS', 'EOS', 'EOS', 'EOS', 'b', 'b', 'b', 'b']

✓ [5 marks]

Implementación del paso Backward

Ahora es momento de implementar el paso backward. Si se pierden, remítanse a las ecuaciones e imagen dadas previamente.

Usaremos una función auxiliar para evitar la explosión del gradiente. Esta técnica suele funcionar muy bien, si quieren leer más sobre esto pueden consultar estos enlaces

[Understanding Gradient Clipping \(and How It Can Fix Exploding Gradients Problem\)](#)

[What exactly happens in gradient clipping by norm?](#)

```
In [61]: def clip_gradient_norm(grads, max_norm=0.25):
    """
    Clipea (recorta?) el gradiente para tener una norma máxima de `max_norm`
    Esto ayudará a prevenir el problema de la gradiente explosiva (BOOM!)
    """
    # Setea el máximo de la norma para que sea flotante
    max_norm = float(max_norm)
    total_norm = 0

    # Calculamos la norma L2 al cuadrado para cada gradiente y agregamos estas a la
    for grad in grads:
        grad_norm = np.sum(np.power(grad, 2))
        total_norm += grad_norm
    # Cuadrado de la normal total
    total_norm = np.sqrt(total_norm)

    # Calculamos el coeficiente de recorte
    clip_coef = max_norm / (total_norm + 1e-6)

    # Si el total de la norma es más grande que el máximo permitido, se recorta la
    if clip_coef < 1:
        for grad in grads:
            grad *= clip_coef
    return grads

def backward_pass(inputs, outputs, hidden_states, targets, params):
    """
    Calcula el paso backward de la RNN
```

```

Args:
    inputs: secuencia de input
    outputs: secuencia de output del forward
    hidden_states: secuencia de los hidden_state del forward
    targets: secuencia target
    params: parametros de la RNN
"""

# Obtener Los parametros
U, V, W, b_hidden, b_out = params

# Inicializamos Las gradientes como cero (Noten que Lo hacemos para Los pesos y
d_U, d_V, d_W = np.zeros_like(U), np.zeros_like(V), np.zeros_like(W)
d_b_hidden, d_b_out = np.zeros_like(b_hidden), np.zeros_like(b_out)

# Llevar el record de Las derivadas de Los hidden state y Las perdidas (loss)
d_h_next = np.zeros_like(hidden_states[0])
loss = 0

# Iteramos para cada elemento en La secuencia output
# NB: Iteramos de regreso sobre t=N hasta 0
for t in reversed(range(len(outputs))):

    # Hint: Sumen +1e-12 a cada output_t
    loss += -np.sum(targets[t] * np.log(outputs[t] + 1e-12)) / targets[t].shape

    d_o = outputs[t].copy()
    # Aprox 1 Linea para backpropagate en Los output (derivada del cross-entrop
    # Si se sienten perdidos refieran a esta Lectura: http://cs231n.github.io/n
    d_o -= targets[t]
    d_W += np.dot(d_o, hidden_states[t].T)
    d_b_out += d_o

    # Aprox 1 Linea para hacer el backprop de h
    d_h = np.dot(W.T, d_o) + d_h_next

    # Aprox 1 Linea para calcular el backprop en La funcion de activacion tanh
    d_f = tanh(hidden_states[t], derivative=True) * d_h
    # Hint: Recuerden pasar el parametro derivate=True a La funcion que definim
    # Hint2: Deben multiplicar con d_h
    d_b_hidden += d_f

    d_U += np.dot(d_f, inputs[t].T)

    d_V += np.dot(d_f, hidden_states[t-1].T)
    d_h_next = np.dot(V.T, d_f)

grads = d_U, d_V, d_W, d_b_hidden, d_b_out

# Corte de gradientes
grads = clip_gradient_norm(grads)

return loss, grads

```



```
In [62]: loss, grads = backward_pass(test_input, outputs, hidden_states, test_target, params

with tick.marks(5):
    assert check_scalar(loss, '0xf0c8ccc9')

with tick.marks(5):
    assert check_hash(grads[0], ((50, 4), -16.16536590645467))

with tick.marks(5):
    assert check_hash(grads[1], ((50, 50), -155.12594909703253))

with tick.marks(5):
    assert check_hash(grads[2], ((4, 50), 1.5957812992239038))
```

✓ [5 marks]

✓ [5 marks]

✓ [5 marks]

✓ [5 marks]

Optimización

Considerando que ya tenemos el paso forward y podemos calcular gradientes con el backpropagation, ya podemos pasar a entrenar nuestra red. Para esto necesitaremos un optimizador. Una forma común y sencilla es implementar la gradiente descendiente. Recuerden la regla de optimización

$$\theta = \theta - \alpha * \nabla J(\theta)$$

- θ son los parametros del modelo
- α es el learning rate
- $\nabla J(\theta)$ representa la gradiente del costo J con respecto de los parametros

```
In [63]: def update_parameters(params, grads, lr=1e-3):
# Iteramos sobre los parametros y las gradientes
for param, grad in zip(params, grads):
    param -= lr * grad
```

```
return params
```

Entrenamiento

Debemos establecer un ciclo de entrenamiento completo que involucre un paso forward, un paso backprop, un paso de optimización y validación. Se espera que el proceso de training dure aproximadamente 5 minutos (o menos), lo que le brinda la oportunidad de continuar leyendo mientras se ejecuta 😊

Noten que estaremos viendo la pérdida en el de validación (no en el de testing) esto se suele hacer para ir observando que tan bien va comportándose el modelo en términos de generalización. Muchas veces es más recomendable ir viendo como evoluciona la métrica de desempeño principal (accuracy, recall, etc).

```
In [64]: # Hyper parametro
# Se coloca como "repsuesta" para que la herramienta no modifique el numero de iter
num_epochs = 2000

# Init una nueva RNN
params = init_rnn(hidden_size=hidden_size, vocab_size=vocab_size)

# Init hidden state con ceros
hidden_state = np.zeros((hidden_size, 1))

# Rastreo de perdida (loss) para training y validacion
training_loss, validation_loss = [], []

# Iteramos para cada epoca
for i in range(num_epochs):

    # Perdidas en cero
    epoch_training_loss = 0
    epoch_validation_loss = 0

    # Para cada secuencia en el grupo de validación
    for inputs, targets in validation_set:

        # One-hot encode el input y el target
        inputs_one_hot = one_hot_encode_sequence(inputs, vocab_size)
        targets_one_hot = one_hot_encode_sequence(targets, vocab_size)

        # Re-init el hidden state
        hidden_state = np.zeros_like(hidden_state)

        # Aprox 1 linea para el paso forward
        outputs, hidden_states = forward_pass(inputs_one_hot, hidden_state, params)

        # Aprox 1 linea para el paso backward
        loss, _ = backward_pass(inputs_one_hot, outputs, hidden_states, targets_one

    # Actualización de perdida
    epoch_validation_loss += loss
```

```

# Para cada secuencia en el grupo de entrenamiento
for inputs, targets in training_set:

    # One-hot encode el input y el target
    inputs_one_hot = one_hot_encode_sequence(inputs, vocab_size)
    targets_one_hot = one_hot_encode_sequence(targets, vocab_size)

    # Re-init el hidden state
    hidden_state = np.zeros_like(hidden_state)

    # Aprox 1 linea para el paso forward
    outputs, hidden_states = forward_pass(inputs_one_hot, hidden_state, params)

    # Aprox 1 linea para el paso backward
    loss, grads = backward_pass(inputs_one_hot, outputs, hidden_states, targets)

    # Validar si la pérdida es nan, llegamos al problema del vanishing gradient
    if np.isnan(loss):
        raise ValueError("La gradiente se desvaneció... POOF!")

    # Actualización de parámetros
    params = update_parameters(params, grads, lr=3e-4)

    # Actualización de pérdida
    epoch_training_loss += loss

# Guardar la pérdida para graficar
training_loss.append(epoch_training_loss / len(training_set))
validation_loss.append(epoch_validation_loss / len(validation_set))

# Mostrar la pérdida cada 100 épocas
if i % 100 == 0:
    print(f'Época {i}, training loss: {training_loss[-1]}, validation loss: {va

```

```

Época 0, training loss: 4.05046509496538, validation loss: 4.801971835967155
Época 100, training loss: 2.729834076574944, validation loss: 3.2320576163982677
Época 200, training loss: 2.109414655736732, validation loss: 2.4980526328844146
Época 300, training loss: 1.8235746981413405, validation loss: 2.198677070984531
Época 400, training loss: 1.6884087861997366, validation loss: 2.077078608023496
Época 500, training loss: 1.6129170568126507, validation loss: 2.0163543941716577
Época 600, training loss: 1.5624028954062004, validation loss: 1.9780311638492243
Época 700, training loss: 1.523501919791708, validation loss: 1.9496130467843362
Época 800, training loss: 1.4895828031292173, validation loss: 1.9248315278145824
Época 900, training loss: 1.4558865884071515, validation loss: 1.8978220912154355
Época 1000, training loss: 1.4173709332614925, validation loss: 1.860079817655523
Época 1100, training loss: 1.3681783634403946, validation loss: 1.7993697026413986
Época 1200, training loss: 1.3051122158818886, validation loss: 1.7081695076503576
Época 1300, training loss: 1.2330985128125034, validation loss: 1.5999314734390082
Época 1400, training loss: 1.1619900522538609, validation loss: 1.4998577602386731
Época 1500, training loss: 1.1035554777966456, validation loss: 1.4282638416110447
Época 1600, training loss: 1.0680633416284246, validation loss: 1.3958745915871211
Época 1700, training loss: 1.0550402179563663, validation loss: 1.3963674481755957
Época 1800, training loss: 1.0570111001893738, validation loss: 1.4185760443851874
Época 1900, training loss: 1.0640880623573374, validation loss: 1.4524183517051121

```

```

In [65]: # Veamos la primera secuencia en el test set
inputs, targets = test_set[1]

# One-hot encode el input y el target
inputs_one_hot = one_hot_encode_sequence(inputs, vocab_size)
targets_one_hot = one_hot_encode_sequence(targets, vocab_size)

# Init el hidden state con ceros
hidden_state = np.zeros((hidden_size, 1))

# Hacemos el pase forward para evaluar nuestra secuencia
outputs, hidden_states = forward_pass(inputs_one_hot, hidden_state, params)
output_sentence = [idx_to_word[np.argmax(output)] for output in outputs]
print("Secuencia Input:")
print(inputs)

print("Secuencia Target:")
print(targets)

print("Secuencia Predicha:")
print([idx_to_word[np.argmax(output)] for output in outputs])

# Graficamos la perdida
epoch = np.arange(len(training_loss))
plt.figure()
plt.plot(epoch, training_loss, 'r', label='Training loss',)
plt.plot(epoch, validation_loss, 'b', label='Validation loss')
plt.legend()
plt.xlabel('Epoch'), plt.ylabel('NLL')
plt.show()

with tick.marks(10):
    assert compare_lists_by_percentage(targets, [idx_to_word[np.argmax(output)] for

```

Secuencia Input:

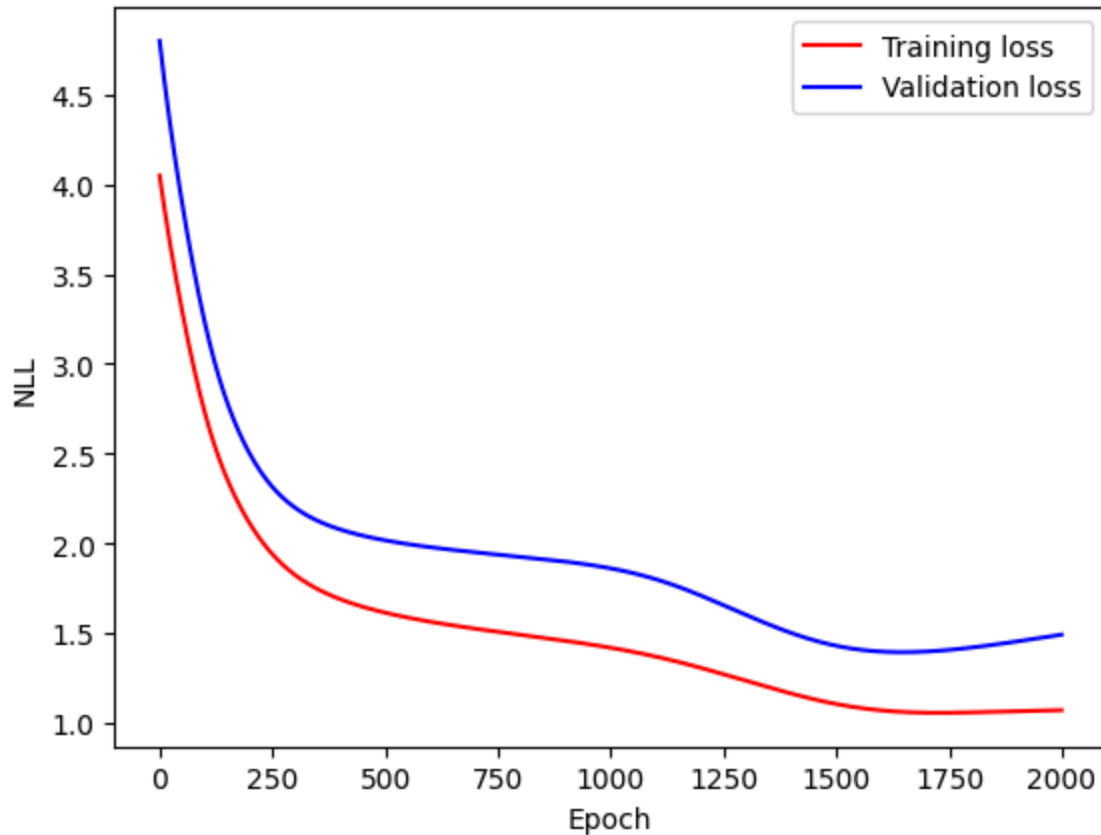
```
['a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'b', 'b', 'b', 'b', 'b',
'b', 'b', 'b', 'b', 'b', 'b']
```

Secuencia Target:

```
['a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'b', 'b', 'b', 'b', 'b',
'b', 'b', 'b', 'b', 'b', 'EOS']
```

Secuencia Predicha:

```
['a', 'a', 'a', 'a', 'a', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b',
'b', 'b', 'b', 'b', 'EOS', 'EOS']
```



✓ [10 marks]

Preguntas

Ya hemos visto el funcionamiento general de nuestra red RNN, viendo las gráficas de arriba, **responda** lo siguiente dentro de esta celda

- ¿Qué interpretación le da a la separación de las graficas de training y validation?

La separación creciente entre las curvas de training loss (roja) y validation loss (azul), especialmente después de la época ~1250, indica que el modelo está empezando a memorizar los datos de entrenamiento y ya no está generalizando bien.

- ¿Cree que es un buen modelo basado solamente en el loss?

El modelo claramente está aprendiendo algo útil: ambas pérdidas bajan al principio, lo que indica que hay aprendizaje efectivo. Entrenarlo por demasiado tiempo lo perjudica. El loss por sí solo no es suficiente

- ¿Cómo deberían de verse esas gráficas en un modelo ideal?

Ambas curvas (training y validation) bajan durante las primeras épocas. La curva de validación se estabiliza o sigue bajando muy lentamente.

Parte 2 - Construyendo una Red Neuronal LSTM

Créditos: La segunda parte de este laboratorio está tomado y basado en uno de los laboratorios dados dentro del curso de "Deep Learning" de Jes Frelsen (DeepLearningDTU)

Consideren leer el siguiente blog para mejorar el entendimiento de este tema:

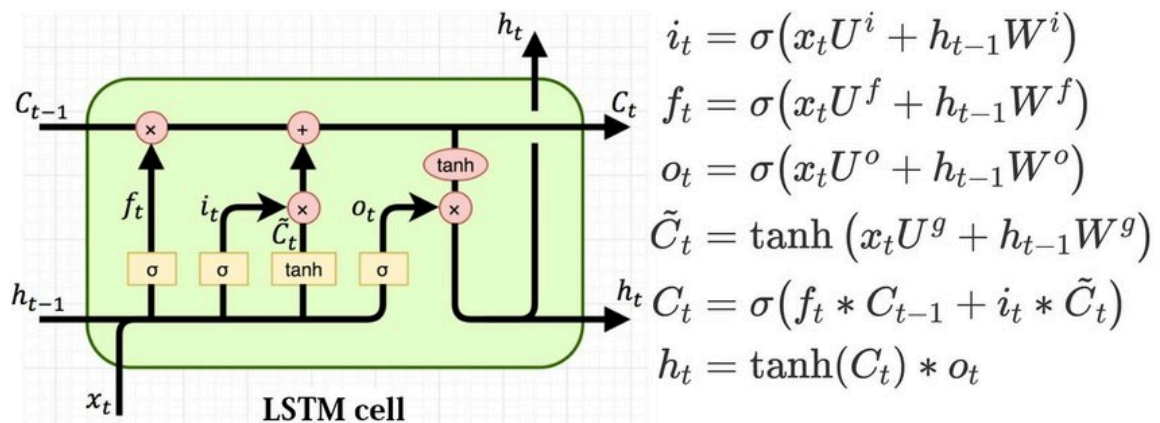
<http://colah.github.io/posts/2015-08-Understanding-LSTMs/>

La RNN estándar enfrenta un problema de gradientes que desaparecen, lo que dificulta la retención de memoria en secuencias más largas. Para hacer frente a estos desafíos, se introdujeron algunas variantes.

Los dos tipos principales son la celda de memoria a corto plazo (LSTM) y la unidad recurrente cerrada (GRU), las cuales demuestran una capacidad mejorada para conservar y utilizar la memoria en pasos de tiempo posteriores.

En este ejercicio, nuestro enfoque estará en LSTM, pero los principios aprendidos aquí también se pueden aplicar fácilmente para implementar GRU.

Recordemos una de las imágenes que vimos en clase



Crédito de imagen al autor, imagen tomada de "Designing neural network based decoders for surface codes" de Savvas Varsamopoulos

Recordemos que la "célula" de LST contiene tres tipos de gates, input, forget y output gate. La salida de una unidad LSTM está calculada por las siguientes funciones, donde $\sigma = \text{softmax}$. Entonces tenemos la input gate i , la forget gate f y la output gate o

- $i = \sigma(W^i[h_{t-1}, x_t])$
- $f = \sigma(W^f[h_{t-1}, x_t])$
- $o = \sigma(W^o[h_{t-1}, x_t])$

Donde W^i , W^f , W^o son las matrices de pesos aplicada a cada aplicadas a una matriz concatenada h_{t-1} (hidden state vector) y x_t (input vector) para cada respectiva gate h_{t-1} ,

del paso previo junto con el input actual x_t son usados para calcular una memoria candidata g

- $g = \tanh(W^g[h_{t-1}, x_t])$

El valor de la memoria c_t es actualizada como

$$c_t = c_{t-1} \circ f + g \circ i$$

donde c_{t-1} es la memoria previa, y \circ es una multiplicacion element-wise (recuerden que este tipo de multiplicación en numpy es con *)

La salida h_t es calculada como

$$h_t = \tanh(c_t) \circ o$$

y este se usa para tanto la salida del paso como para el siguiente paso, mientras c_t es exclusivamente enviado al siguiente paso. Esto hace c_t una memoria feature, y no es usado directamente para caluclar la salida del paso actual.

Iniciando una Red LSTM

De forma similar a lo que hemos hecho antes, necesitaremos implementar el paso forward, backward y un ciclo de entrenamiento. Pero ahora usaremos LSTM con NumPy. Más adelante veremos como es que esto funciona con PyTorch.

```
In [66]: np.random.seed(seed_)

# Tamaño del hidden state concatenado más el input
z_size = hidden_size + vocab_size

def init_lstm(hidden_size, vocab_size, z_size):
    """
    Initializes our LSTM network.
    Init LSTM

    Args:
        hidden_size: Dimensiones del hidden state
        vocab_size: Dimensiones de nuestro vocabulario
        z_size: Dimensiones del input concatenado
    """

    # Aprox 1 linea para empezar la matriz de pesos de la forget gate
    # Recuerden que esta debe empezar con numeros aleatorios
    # W_f = np.random.randn
    # YOUR CODE HERE
    W_f = np.random.randn(hidden_size, z_size)

    # Bias del forget gate
    b_f = np.zeros((hidden_size, 1))

    # Aprox 1 linea para empezar la matriz de pesos de la input gate
```

```

# Recuerden que esta debe empezar con numeros aleatorios
# YOUR CODE HERE
W_i = np.random.randn(hidden_size, z_size)

# Bias para input gate
b_i = np.zeros((hidden_size, 1))

# Aprox 1 linea para empezar la matriz de pesos para la memoria candidata
# Recuerden que esta debe empezar con numeros aleatorios
# YOUR CODE HERE
W_g = np.random.randn(hidden_size, z_size)

# Bias para la memoria candidata
b_g = np.zeros((hidden_size, 1))

# Aprox 1 linea para empezar la matriz de pesos para la output gate
# YOUR CODE HERE
W_o = np.random.randn(hidden_size, z_size)

# Bias para la output gate
b_o = np.zeros((hidden_size, 1))

# Aprox 1 linea para empezar la matriz que relaciona el hidden state con el out
# YOUR CODE HERE
W_v = np.random.randn(vocab_size, hidden_size)

# Bias
b_v = np.zeros((vocab_size, 1))

# Init pesos ortogonalmente (https://arxiv.org/abs/1312.6120)
W_f = init_orthogonal(W_f)
W_i = init_orthogonal(W_i)
W_g = init_orthogonal(W_g)
W_o = init_orthogonal(W_o)
W_v = init_orthogonal(W_v)

return W_f, W_i, W_g, W_o, W_v, b_f, b_i, b_g, b_o, b_v

params = init_lstm(hidden_size=hidden_size, vocab_size=vocab_size, z_size=z_size)

```

```

In [67]: with tick.marks(5):
    assert check_hash(params[0], ((50, 54), -28071.583543573637))

with tick.marks(5):
    assert check_hash(params[1], ((50, 54), -6337.520066952928))

with tick.marks(5):
    assert check_hash(params[2], ((50, 54), -13445.986473992281))

with tick.marks(5):
    assert check_hash(params[3], ((50, 54), 2276.1116210911564))

with tick.marks(5):
    assert check_hash(params[4], ((4, 50), -201.28961326044097))

```


✓ [5 marks]

✓ [5 marks]

✓ [5 marks]

✓ [5 marks]

✓ [5 marks]

Forward

Vamos para adelante con LSTM, al igual que previamente necesitamos implementar las funciones antes mencionadas

```
In [68]: def forward(inputs, h_prev, C_prev, p):
        """
        Arguments:
        x: Input data en el paso "t", shape (n_x, m)
        h_prev: Hidden state en el paso "t-1", shape (n_a, m)
        C_prev: Memoria en el paso "t-1", shape (n_a, m)
        p: Lista con pesos y biases, contiene:
            W_f: Pesos de la forget gate, shape (n_a, n_a + n_x)
            b_f: Bias de la forget gate, shape (n_a, 1)
            W_i: Pesos de la update gate, shape (n_a, n_a + n_x)
            b_i: Bias de la update gate, shape (n_a, 1)
            W_g: Pesos de la primer "tanh", shape (n_a, n_a + n_x)
            b_g: Bias de la primer "tanh", shape (n_a, 1)
            W_o: Pesos de la output gate, shape (n_a, n_a + n_x)
            b_o: Bias de la output gate, shape (n_a, 1)
            W_v: Pesos de la matriz que relaciona el hidden state con e
            b_v: Bias que relaciona el hidden state con el output, shap

        Returns:
        z_s, f_s, i_s, g_s, C_s, o_s, h_s, v_s: Lista de tamaño m conteniendo los calculo
        outputs: Predicciones en el paso "t", shape (n_v, m)
        """

        # Validar Las dimensiones
```

```

assert h_prev.shape == (hidden_size, 1)
assert C_prev.shape == (hidden_size, 1)

# Desempacar los parametros
W_f, W_i, W_g, W_o, W_v, b_f, b_i, b_g, b_o, b_v = p

# Listas para calculos de cada componente en LSTM
x_s, z_s, f_s, i_s, = [], [], [], []
g_s, C_s, o_s, h_s = [], [], [], []
v_s, output_s = [], []

# Agregar los valores iniciales
h_s.append(h_prev)
C_s.append(C_prev)

for x in inputs:

    # Aprox 1 linea para concatenar el input y el hidden state
    # z = np.row.stack(...)
    # YOUR CODE HERE
    z = np.vstack((h_prev, x))
    z_s.append(z)

    # Aprox 1 linea para calcular el forget gate
    # Hint: recuerde usar sigmoid
    # f =
    # YOUR CODE HERE
    f = sigmoid(np.dot(W_f, z) + b_f)
    f_s.append(f)

    # Calculo del input gate
    i = sigmoid(np.dot(W_i, z) + b_i)
    i_s.append(i)

    # Calculo de la memoria candidata
    g = tanh(np.dot(W_g, z) + b_g)
    g_s.append(g)

    # Aprox 1 linea para calcular el estado de la memoria
    # C_prev =
    # YOUR CODE HERE
    C_prev = f * C_prev + i * g
    C_s.append(C_prev)

    # Aprox 1 linea para el calculo de la output gate
    # Hint: recuerde usar sigmoid
    # o =
    # YOUR CODE HERE
    o = sigmoid(np.dot(W_o, z) + b_o)
    o_s.append(o)

    # Calculate hidden state
    # Aprox 1 linea para el calculo del hidden state
    # h_prev =
    # YOUR CODE HERE
    h_prev = o * tanh(C_prev)

```

```

        h_s.append(h_prev)

        # Calcular logits
        v = np.dot(W_v, h_prev) + b_v
        v_s.append(v)

        # Calculo de output (con softmax)
        output = softmax(v)
        output_s.append(output)

    return z_s, f_s, i_s, g_s, C_s, o_s, h_s, v_s, output_s

```

```

In [69]: # Obtener la primera secuencia para probar
inputs, targets = test_set[1]

# One-hot encode del input y target
inputs_one_hot = one_hot_encode_sequence(inputs, vocab_size)
targets_one_hot = one_hot_encode_sequence(targets, vocab_size)

# Init hidden state con ceros
h = np.zeros((hidden_size, 1))
c = np.zeros((hidden_size, 1))

# Forward
z_s, f_s, i_s, g_s, C_s, o_s, h_s, v_s, outputs = forward(inputs_one_hot, h, c, par

output_sentence = [idx_to_word[np.argmax(output)] for output in outputs]

print("Secuencia Input:")
print(inputs)

print("Secuencia Target:")
print(targets)

print("Secuencia Predicha:")
print([idx_to_word[np.argmax(output)] for output in outputs])

with tick.marks(5):
    assert check_hash(outputs, ((22, 4, 1), 980.1651308051631))

```

Secuencia Input:

```
['a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'b', 'b', 'b', 'b', 'b',
'b', 'b', 'b', 'b', 'b', 'b', 'b']
```

Secuencia Target:

```
['a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'b', 'b', 'b', 'b', 'b', 'b',
'b', 'b', 'b', 'b', 'b', 'b', 'EOS']
```

Secuencia Predicha:

```
['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'EOS', 'EOS', 'EOS', 'b',
'b', 'b', 'b', 'b', 'b', 'b', 'b']
```

✓ [5 marks]

Backward

Ahora de reversa, al igual que lo hecho antes, necesitamos implementar el paso de backward

```
In [70]: def backward(z, f, i, g, C, o, h, v, outputs, targets, p = params):
        """
        Arguments:
        z: Input concatenado como una lista de tamaño m.
        f: Calculos del forget gate como una lista de tamaño m.
        i: Calculos del input gate como una lista de tamaño m.
        g: Calculos de la memoria candidata como una lista de tamaño m.
        C: Celdas estado como una lista de tamaño m+1.
        o: Calculos del output gate como una lista de tamaño m.
        h: Calculos del Hidden State como una lista de tamaño m+1.
        v: Calculos del logit como una lista de tamaño m.
        outputs: Salidas como una lista de tamaño m.
        targets: Targets como una lista de tamaño m.
        p: Lista con pesos y biases, contiene:
            W_f: Pesos de la forget gate, shape (n_a, n_a + n_x)
            b_f: Bias de la forget gate, shape (n_a, 1)
            W_i: Pesos de la update gate, shape (n_a, n_a + n_x)
            b_i: Bias de la update gate, shape (n_a, 1)
            W_g: Pesos de la primer "tanh", shape (n_a, n_a + n_x)
            b_g: Bias de la primer "tanh", shape (n_a, 1)
            W_o: Pesos de la output gate, shape (n_a, n_a + n_x)
            b_o: Bias de la output gate, shape (n_a, 1)
            W_v: Pesos de la matriz que relaciona el hidden state con e
            b_v: Bias que relaciona el hidden state con el output, shap

        Returns:
        loss: crossentropy loss para todos los elementos del output
        grads: lista de gradientes para todos los elementos en p
        """

        # Desempacar parametros
        W_f, W_i, W_g, W_o, W_v, b_f, b_i, b_g, b_o, b_v = p

        # Init gradientes con cero
        W_f_d = np.zeros_like(W_f)
        b_f_d = np.zeros_like(b_f)

        W_i_d = np.zeros_like(W_i)
        b_i_d = np.zeros_like(b_i)

        W_g_d = np.zeros_like(W_g)
        b_g_d = np.zeros_like(b_g)

        W_o_d = np.zeros_like(W_o)
        b_o_d = np.zeros_like(b_o)

        W_v_d = np.zeros_like(W_v)
        b_v_d = np.zeros_like(b_v)

        # Setear la proxima unidad y hidden state con ceros
        dh_next = np.zeros_like(h[0])
```

```

dC_next = np.zeros_like(C[0])

# Para la perdida
loss = 0

# Iteramos en reversa los outputs
for t in reversed(range(len(outputs))):

    # Aprox 1 linea para calcular la perdida con cross entropy
    # loss += ...
    # YOUR CODE HERE
    loss += -np.mean(targets[t] * np.log(outputs[t]))

    # Obtener el estado de memoria previo
    C_prev = C[t-1]

    # Compute the derivative de la salida (softmax + CE)
    dv = outputs[t].copy()
    dv[np.argmax(targets[t])] -= 1

    # Aprox 1 linea para actualizar la gradiente de la relacion del hidden-state
    # W_v_d +=
    # YOUR CODE HERE
    W_v_d += np.dot(dv, h[t+1].T)
    b_v_d += dv

    # Calculo de la derivada del hidden state y acumulación de dh_next
    dh = np.dot(W_v.T, dv) + dh_next

    # Aprox 1 linea para calcular la derivada del output gate
    # do = ..
    # YOUR CODE HERE
    do = dh * np.tanh(C[t])
    do *= o[t] * (1 - o[t])

    # Actualizacion de las gradientes con respecto al output gate
    W_o_d += np.dot(do, z[t].T)
    b_o_d += do

    # Calculo de dC total
    # YOUR CODE HERE
    dC = dC_next + dh * o[t] * (1 - np.tanh(C[t])**2)

    # Calculo de las derivadas de g
    dg = dC * i[t]
    # YOUR CODE HERE
    dg *= (1 - g[t]**2)

    W_g_d += np.dot(dg, z[t].T)
    b_g_d += dg

    # Derivadas del input gate
    di = dC * g[t]
    # YOUR CODE HERE
    di *= i[t] * (1 - i[t])
    W_i_d += np.dot(di, z[t].T)

```

```

        b_i_d += di

        # Derivadas del forget gate
        df = dC * C_prev
        # YOUR CODE HERE
        df *= f[t] * (1 - f[t])
        W_f_d += np.dot(df, z[t].T)
        b_f_d += df

        # Propagar a la siguiente iteración
        dz = (np.dot(W_f.T, df)
              + np.dot(W_i.T, di)
              + np.dot(W_g.T, dg)
              + np.dot(W_o.T, do))
        dh_prev = dz[:hidden_size, :]
        dC_prev = f[t] * dC

        dh_next = dh_prev
        dC_next = dC_prev

    grads = (W_f_d, W_i_d, W_g_d, W_o_d, W_v_d,
             b_f_d, b_i_d, b_g_d, b_o_d, b_v_d)

    # Recorte de gradientes
    grads = clip_gradient_norm(grads)

    return loss, grads

```

```

In [71]: # Realizamos un backward pass para probar
loss, grads = backward(z_s, f_s, i_s, g_s, C_s, o_s, h_s, v_s, outputs, targets_one

print(f"Perdida obtenida:{loss}")

with tick.marks(5):
    assert(check_scalar(loss, '0x53c34f25'))

```

Perdida obtenida:7.637217940763248

✓ [5 marks]

Training

Ahora intentemos entrenar nuestro LSTM básico. Esta parte es muy similar a lo que ya hicimos previamente con la RNN

```

In [72]: # Hyper parametros
num_epochs = 500

# Init una nueva red
z_size = hidden_size + vocab_size # Tamaño del hidden concatenado + el input
params = init_lstm(hidden_size=hidden_size, vocab_size=vocab_size, z_size=z_size)

```

```

# Init hidden state como ceros
hidden_state = np.zeros((hidden_size, 1))

# Perdida
training_loss, validation_loss = [], []

# Iteramos cada epoca
for i in range(num_epochs):

    # Perdidas
    epoch_training_loss = 0
    epoch_validation_loss = 0

    # Para cada secuencia en el validation set
    for inputs, targets in validation_set:

        # One-hot encode el inpyt y el target
        inputs_one_hot = one_hot_encode_sequence(inputs, vocab_size)
        targets_one_hot = one_hot_encode_sequence(targets, vocab_size)

        # Init hidden state y la unidad de estado como ceros
        h = np.zeros((hidden_size, 1))
        c = np.zeros((hidden_size, 1))

        # Forward
        z_s, f_s, i_s, g_s, C_s, o_s, h_s, v_s, outputs = forward(inputs_one_hot, h

        # Backward
        loss, _ = backward(z_s, f_s, i_s, g_s, C_s, o_s, h_s, v_s, outputs, targets

        # Actualizacion de la perdida
        epoch_validation_loss += loss

    # Para cada secuencia en el training set
    for inputs, targets in training_set:

        # One-hot encode el inpyt y el target
        inputs_one_hot = one_hot_encode_sequence(inputs, vocab_size)
        targets_one_hot = one_hot_encode_sequence(targets, vocab_size)

        # Init hidden state y la unidad de estado como ceros
        h = np.zeros((hidden_size, 1))
        c = np.zeros((hidden_size, 1))

        # Forward
        z_s, f_s, i_s, g_s, C_s, o_s, h_s, v_s, outputs = forward(inputs_one_hot, h

        # Backward
        loss, grads = backward(z_s, f_s, i_s, g_s, C_s, o_s, h_s, v_s, outputs, tar

        # Actualización de parametros
        params = update_parameters(params, grads, lr=1e-1)

        # Actualizacion de la perdida
        epoch_training_loss += loss

```

```
# Guardar La perdida para ser graficada
training_loss.append(epoch_training_loss/len(training_set))
validation_loss.append(epoch_validation_loss/len(validation_set))

# Mostrar La perdida cada 5 epocas
if i % 10 == 0:
    print(f'Epoch {i}, training loss: {training_loss[-1]}, validation loss: {va
```



```

Epoch 0, training loss: 2.9228179938116403, validation loss: 4.499707061171418
Epoch 10, training loss: 1.052324570568169, validation loss: 1.25389816001126
Epoch 20, training loss: 1.0039297227394979, validation loss: 1.7280554267108503
Epoch 30, training loss: 0.9268120057991499, validation loss: 1.5877731901505894
Epoch 40, training loss: 0.8227998500801583, validation loss: 1.2664517658344927
Epoch 50, training loss: 0.7569274131266605, validation loss: 1.054816201396105
Epoch 60, training loss: 0.7248937309245967, validation loss: 0.8605814819482651
Epoch 70, training loss: 0.7119726266400896, validation loss: 0.8174984048864957
Epoch 80, training loss: 0.7009492449391341, validation loss: 0.7910986346988728
Epoch 90, training loss: 0.6902164602397557, validation loss: 0.7643593688769845
Epoch 100, training loss: 0.682407785216461, validation loss: 0.7571539177690102
Epoch 110, training loss: 0.6758240482221914, validation loss: 0.7538026576721573
Epoch 120, training loss: 0.6719899996197485, validation loss: 0.7519523303334351
Epoch 130, training loss: 0.6694822037753552, validation loss: 0.7506738459923936
Epoch 140, training loss: 0.6676736264038818, validation loss: 0.7497064185313492
Epoch 150, training loss: 0.6664608997986206, validation loss: 0.7490339787689709
Epoch 160, training loss: 0.6659111228157245, validation loss: 0.7486934561674667
Epoch 170, training loss: 0.6657998167351333, validation loss: 0.7485933575662854
Epoch 180, training loss: 0.6656576255425876, validation loss: 0.7485737790546293
Epoch 190, training loss: 0.665339118045183, validation loss: 0.7486026963513573
Epoch 200, training loss: 0.664889585664451, validation loss: 0.7486834275265367
Epoch 210, training loss: 0.6643707412803572, validation loss: 0.7488146473295993
Epoch 220, training loss: 0.6638272644905997, validation loss: 0.7489925689959831
Epoch 230, training loss: 0.6632874621504672, validation loss: 0.749212516101595
Epoch 240, training loss: 0.662767636550304, validation loss: 0.7494695867622768
Epoch 250, training loss: 0.6622762195210528, validation loss: 0.7497591213004304
Epoch 260, training loss: 0.6618169494097017, validation loss: 0.7500769321421823
Epoch 270, training loss: 0.6613908679603889, validation loss: 0.7504192877767744
Epoch 280, training loss: 0.6609973941752626, validation loss: 0.7507827674057997
Epoch 290, training loss: 0.6606348955906657, validation loss: 0.7511641058063331
Epoch 300, training loss: 0.6603010476913466, validation loss: 0.7515600789533565
Epoch 310, training loss: 0.6599931000156577, validation loss: 0.751967433187024
Epoch 320, training loss: 0.6597080783688011, validation loss: 0.7523828558837049
Epoch 330, training loss: 0.6594429332864595, validation loss: 0.7528029951176645
Epoch 340, training loss: 0.6591946483652974, validation loss: 0.7532245339811166
Epoch 350, training loss: 0.6589603226838511, validation loss: 0.7536443095142734
Epoch 360, training loss: 0.6587372362317709, validation loss: 0.7540594491169622
Epoch 370, training loss: 0.6585229010517042, validation loss: 0.7544674911522051
Epoch 380, training loss: 0.658315097329113, validation loss: 0.7548664639598565
Epoch 390, training loss: 0.6581118933142067, validation loss: 0.7552549126800407
Epoch 400, training loss: 0.6579116493823747, validation loss: 0.7556318779639624
Epoch 410, training loss: 0.6577130083121112, validation loss: 0.75599683978089
Epoch 420, training loss: 0.6575148751259947, validation loss: 0.7563496422789958
Epoch 430, training loss: 0.6573163903037887, validation loss: 0.7566904137683486
Epoch 440, training loss: 0.6571168999159752, validation loss: 0.7570194917785331
Epoch 450, training loss: 0.656915925511554, validation loss: 0.7573373587528387
Epoch 460, training loss: 0.6567131357287904, validation loss: 0.7576445904019135
Epoch 470, training loss: 0.6565083208073981, validation loss: 0.757941816430632
Epoch 480, training loss: 0.6563013705728882, validation loss: 0.7582296921625875
Epoch 490, training loss: 0.6560922560461719, validation loss: 0.7585088791899377

```

```

In [73]: # Obtener la primera secuencia del test set
         inputs, targets = test_set[1]

         # One-hot encode el input y el target
         inputs_one_hot = one_hot_encode_sequence(inputs, vocab_size)

```

```

targets_one_hot = one_hot_encode_sequence(targets, vocab_size)

# Init hidden state como ceros
h = np.zeros((hidden_size, 1))
c = np.zeros((hidden_size, 1))

# Forward
z_s, f_s, i_s, g_s, C_s, o_s, h_s, v_s, outputs = forward(inputs_one_hot, h, c, par

print("Secuencia Input:")
print(inputs)

print("Secuencia Target:")
print(targets)

print("Secuencia Predicha:")
print([idx_to_word[np.argmax(output)] for output in outputs])

# Graficar la perdida en training y validacion
epoch = np.arange(len(training_loss))
plt.figure()
plt.plot(epoch, training_loss, 'r', label='Training loss',)
plt.plot(epoch, validation_loss, 'b', label='Validation loss')
plt.legend()
plt.xlabel('Epoch'), plt.ylabel('NLL')
plt.show()

```

Secuencia Input:

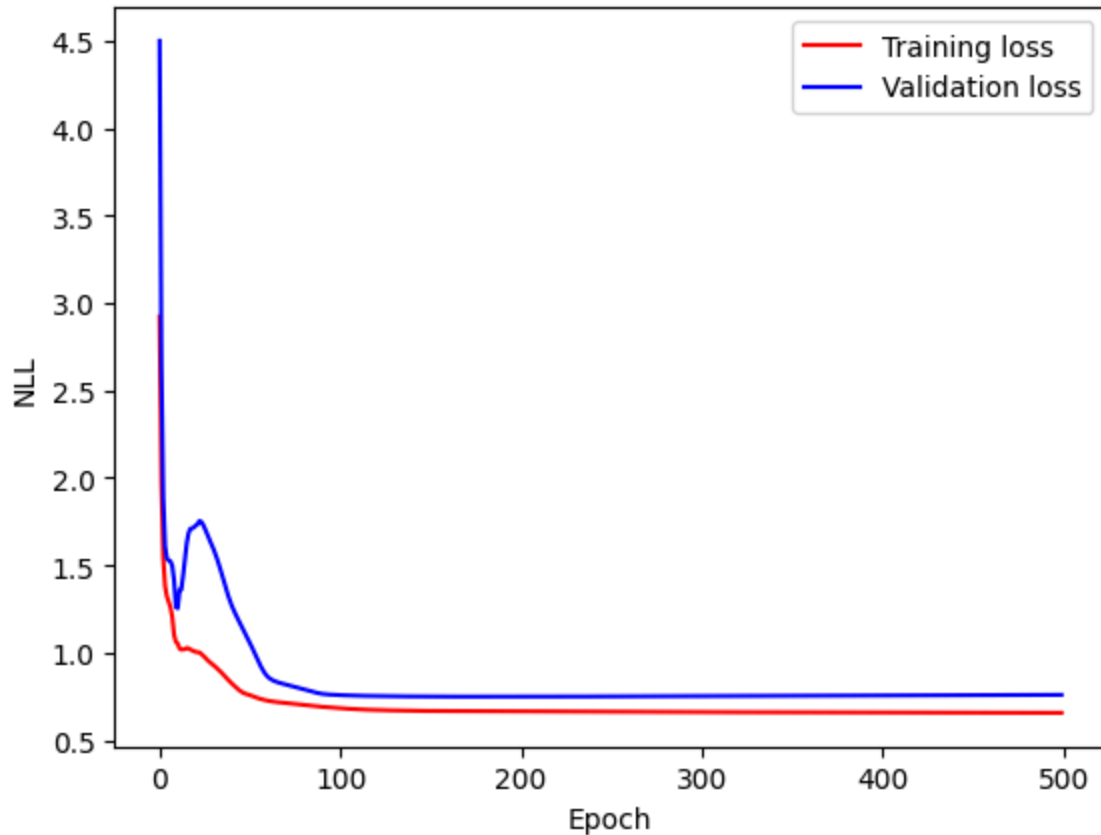
```
['a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'b', 'b', 'b', 'b', 'b',
'b', 'b', 'b', 'b', 'b', 'b']
```

Secuencia Target:

```
['a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'b', 'b', 'b', 'b', 'b',
'b', 'b', 'b', 'b', 'b', 'EOS']
```

Secuencia Predicha:

```
['a', 'a', 'a', 'a', 'a', 'a', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b',
'b', 'b', 'b', 'b', 'b', 'EOS']
```



Preguntas

Responda lo siguiente dentro de esta celda

- ¿Qué modelo funcionó mejor? ¿RNN tradicional o el basado en LSTM? ¿Por qué?
 - El modelo basado en LSTM obtuvo mejor desempeño que la RNN "vanilla". Mientras que la RNN tradicional se atascaba pronto (su pérdida se estancaba o incluso subía después de unas pocas épocas), la LSTM siguió reduciendo la pérdida de entrenamiento y alcanzó un valor final mucho más bajo.
- Observen la gráfica obtenida arriba, ¿en qué es diferente a la obtenida a RNN? ¿Es esto mejor o peor? ¿Por qué?
 - En la gráfica de la LSTM vemos una curva de pérdida más suave, con menos oscilaciones y un descenso más pronunciado hacia valores bajos. En la RNN clásica la curva era más irregular y se aplanaba antes de tiempo. Esto es mejor porque indica que la LSTM está aprendiendo de forma más estable y continua (menor varianza en los gradientes) y convergiendo a un mínimo de pérdida más profundo.
- ¿Por qué LSTM puede funcionar mejor con secuencias largas?
 - Las LSTM funcionan mejor con secuencias largas porque disponen de un "cell state" y puertas (forget, input, output) que controlan el flujo de información y gradientes. Eso les permite retener dependencias a largo plazo y evita el problema del "gradiente que desaparece" que sufren las RNN estándar.

Parte 3 - Red Neuronal LSTM con PyTorch

Ahora que ya hemos visto el funcionamiento paso a paso de tanto RNN tradicional como LSTM. Es momento de usar PyTorch. Para esta parte usaremos el mismo dataset generado al inicio. Así mismo, usaremos un ciclo de entrenamiento similar al que hemos usado previamente.

En la siguiente parte (sí, hay una siguiente parte 🤖) usaremos otro tipo de dataset más formal

```
In [74]: import torch
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()

        # Aprox 1-3 lineas de codigo para declarar una capa LSTM
        self.lstm = nn.LSTM(
            input_size=vocab_size,
            hidden_size=50,
            num_layers=1,
            bidirectional=False,
            batch_first=False
        )

        # Layer de salida (output)
        self.l_out = nn.Linear(in_features=50,
                                out_features=vocab_size,
                                bias=False)

    def forward(self, x):
        # RNN regresa el output y el ultimo hidden state
        x, (h, c) = self.lstm(x)

        # Aplanar la salida para una layer feed forward
        x = x.view(-1, self.lstm.hidden_size)

        # Layer de output
        x = self.l_out(x)

        return x

net = Net()
print(net)
```

```
Net(
  (lstm): LSTM(4, 50)
  (l_out): Linear(in_features=50, out_features=4, bias=False)
)
```

```
In [75]: # Hyper parametros
num_epochs = 500

# Init una nueva red
net = Net()

# Aprox 2 lineas para definir la función de perdida y el optimizador
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(net.parameters(), lr=3e-4)
# YOUR CODE HERE

# Perdida
training_loss, validation_loss = [], []

# Iteramos cada epoca
for i in range(num_epochs):

    # Perdidas
    epoch_training_loss = 0
    epoch_validation_loss = 0

    # NOTA 1
    net.eval()

    # Para cada secuencia en el validation set
    for inputs, targets in validation_set:

        # One-hot encode el input y el target
        inputs_one_hot = one_hot_encode_sequence(inputs, vocab_size)
        targets_idx = [word_to_idx[word] for word in targets]

        # Convertir el input a un tensor
        inputs_one_hot = torch.Tensor(inputs_one_hot)
        inputs_one_hot = inputs_one_hot.permute(0, 2, 1)

        # Convertir el target a un tensor
        targets_idx = torch.LongTensor(targets_idx)

        # Aprox 1 linea para el Forward
        outputs = net(inputs_one_hot)
        # YOUR CODE HERE

        # Aprox 1 linea para calcular la perdida
        loss = criterion(outputs, targets_idx)
        # Hint: Use el criterion definido arriba
        # YOUR CODE HERE

        # Actualizacion de la perdida
        epoch_validation_loss += loss.detach().numpy()

    # NOTA 2
    net.train()

    # Para cada secuencia en el training set
    for inputs, targets in training_set:
```

```
# One-hot encode el input y el target
inputs_one_hot = one_hot_encode_sequence(inputs, vocab_size)
targets_idx = [word_to_idx[word] for word in targets]

# Convertir el input a un tensor
inputs_one_hot = torch.Tensor(inputs_one_hot)
inputs_one_hot = inputs_one_hot.permute(0, 2, 1)

# Convertir el target a un tensor
targets_idx = torch.LongTensor(targets_idx)

# Aprox 1 linea para el Forward
outputs = net(inputs_one_hot)
# YOUR CODE HERE

# Aprox 1 linea para calcular la perdida
loss = criterion(outputs, targets_idx)
# Hint: Use el criterion definido arriba
# YOUR CODE HERE

optimizer.zero_grad()
loss.backward()
optimizer.step()

# Actualizacion de la perdida
epoch_training_loss += loss.detach().numpy()

# Guardar la perdida para ser graficada
training_loss.append(epoch_training_loss/len(training_set))
validation_loss.append(epoch_validation_loss/len(validation_set))

# Mostrar la perdida cada 5 epocas
if i % 10 == 0:
    print(f'Epoch {i}, training loss: {training_loss[-1]}, validation loss: {va
```

```

Epoch 0, training loss: 1.3045165538787842, validation loss: 1.3767815828323364
Epoch 10, training loss: 0.5447369813919067, validation loss: 0.4962865710258484
Epoch 20, training loss: 0.41638460755348206, validation loss: 0.37662461400032043
Epoch 30, training loss: 0.36238712072372437, validation loss: 0.3162037134170532
Epoch 40, training loss: 0.334423303604126, validation loss: 0.29340407252311707
Epoch 50, training loss: 0.3174991309642792, validation loss: 0.28240662813186646
Epoch 60, training loss: 0.3075101971626282, validation loss: 0.27620038390159607
Epoch 70, training loss: 0.3017024099826813, validation loss: 0.27103787660598755
Epoch 80, training loss: 0.29836171865463257, validation loss: 0.26830607652664185
Epoch 90, training loss: 0.2960580885410309, validation loss: 0.2669038474559784
Epoch 100, training loss: 0.29440778493881226, validation loss: 0.26640385389328003
Epoch 110, training loss: 0.29320940375328064, validation loss: 0.26656845211982727
Epoch 120, training loss: 0.2923279404640198, validation loss: 0.26723843812942505
Epoch 130, training loss: 0.29145440459251404, validation loss: 0.27028730511665344
Epoch 140, training loss: 0.2910357415676117, validation loss: 0.26964759826660156
Epoch 150, training loss: 0.29086917638778687, validation loss: 0.27009162306785583
Epoch 160, training loss: 0.29073312878608704, validation loss: 0.2708836793899536
Epoch 170, training loss: 0.2906125783920288, validation loss: 0.2716417908668518
Epoch 180, training loss: 0.2905084490776062, validation loss: 0.2721828818321228
Epoch 190, training loss: 0.2898831367492676, validation loss: 0.2750479578971863
Epoch 200, training loss: 0.28977417945861816, validation loss: 0.27351826429367065
Epoch 210, training loss: 0.2898883521556854, validation loss: 0.27328819036483765
Epoch 220, training loss: 0.289970725774765, validation loss: 0.27330636978149414
Epoch 230, training loss: 0.2900118827819824, validation loss: 0.2734089493751526
Epoch 240, training loss: 0.29001662135124207, validation loss: 0.27354341745376587
Epoch 250, training loss: 0.28999871015548706, validation loss: 0.2736857533454895
Epoch 260, training loss: 0.28918948769569397, validation loss: 0.27624860405921936
Epoch 270, training loss: 0.28930848836898804, validation loss: 0.274847149848938
Epoch 280, training loss: 0.2894509434700012, validation loss: 0.27438604831695557
Epoch 290, training loss: 0.2895506024360657, validation loss: 0.27422016859054565
Epoch 300, training loss: 0.28960859775543213, validation loss: 0.27417629957199097
Epoch 310, training loss: 0.2896306812763214, validation loss: 0.2741876244544983
Epoch 320, training loss: 0.2896222770214081, validation loss: 0.27423185110092163
Epoch 330, training loss: 0.2895919680595398, validation loss: 0.27427130937576294
Epoch 340, training loss: 0.2895447909832001, validation loss: 0.27429714798927307
Epoch 350, training loss: 0.2894853949546814, validation loss: 0.2743085026741028
Epoch 360, training loss: 0.2889536917209625, validation loss: 0.27669432759284973
Epoch 370, training loss: 0.2887740135192871, validation loss: 0.27537310123443604
Epoch 380, training loss: 0.28889065980911255, validation loss: 0.2749599814414978
Epoch 390, training loss: 0.2889956831932068, validation loss: 0.27473750710487366
Epoch 400, training loss: 0.28906771540641785, validation loss: 0.2746105492115021
Epoch 410, training loss: 0.28910842537879944, validation loss: 0.2745527923107147
Epoch 420, training loss: 0.28912270069122314, validation loss: 0.27454203367233276
Epoch 430, training loss: 0.2891154885292053, validation loss: 0.27455732226371765
Epoch 440, training loss: 0.28908267617225647, validation loss: 0.2745401859283447
Epoch 450, training loss: 0.2890462279319763, validation loss: 0.27461832761764526
Epoch 460, training loss: 0.28899604082107544, validation loss: 0.27456146478652954
Epoch 470, training loss: 0.28894731402397156, validation loss: 0.27460163831710815
Epoch 480, training loss: 0.28809794783592224, validation loss: 0.2761448621749878
Epoch 490, training loss: 0.28825098276138306, validation loss: 0.27527761459350586

```

```

In [76]: with tick.marks(5):
          assert compare_numbers(new_representation(training_loss[-1]), "3c3d", '0x1.28f5

          with tick.marks(5):
             assert compare_numbers(new_representation(validation_loss[-1]), "3c3d", '0x1.28

```

✓ [5 marks]

✓ [5 marks]

```
In [77]: # Obtener la primera secuencia del test set
inputs, targets = test_set[1]

# One-hot encode el input y el target
inputs_one_hot = one_hot_encode_sequence(inputs, vocab_size) # Probablemente (seq_
targets_idx = [word_to_idx[word] for word in targets]

# Convertir a tensor y eliminar dimensiones extra
inputs_one_hot = torch.tensor(inputs_one_hot, dtype=torch.float32)

if inputs_one_hot.shape[-1] == 1:
    inputs_one_hot = inputs_one_hot.squeeze(-1) # (seq_len, vocab_size)

# Agregar batch dimension
inputs_one_hot = inputs_one_hot.unsqueeze(1) # (seq_len, 1, vocab_size)

# Convertir el target a un tensor
targets_idx = torch.LongTensor(targets_idx)

# Verificar forma
print("Input shape to LSTM:", inputs_one_hot.shape)

# Forward pass
outputs = net(inputs_one_hot)

# Obtener predicciones
predicted_idx = torch.argmax(outputs, dim=1)

print("Secuencia Input:")
print(inputs)

print("Secuencia Target:")
print(targets)

print("Secuencia Predicha:")
print([idx_to_word[int(i)] for i in predicted_idx])

# Graficar pérdida
epoch = np.arange(len(training_loss))
plt.figure()
plt.plot(epoch, training_loss, 'r', label='Training loss')
plt.plot(epoch, validation_loss, 'b', label='Validation loss')
plt.legend()
plt.xlabel('Epoch')
plt.ylabel('NLL')
```



```
plt.title("Training vs Validation Loss")
plt.show()
```

Input shape to LSTM: torch.Size([22, 1, 4])

Secuencia Input:

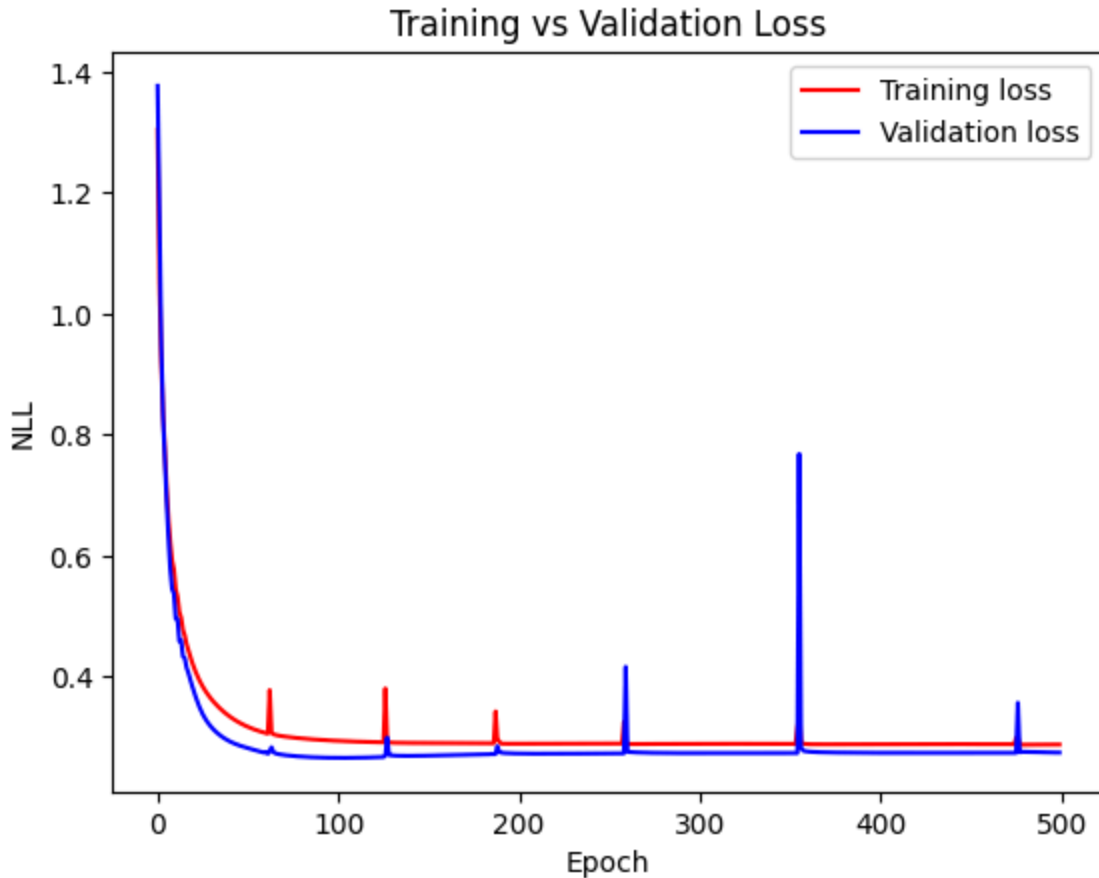
['a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b']

Secuencia Target:

['a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'EOS']

Secuencia Predicha:

['a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'EOS']



Preguntas

Respuesta lo siguiente dentro de esta celda

- Compare las graficas obtenidas en el LSTM "a mano" y el LSTM "usando PyTorch, ¿cuál cree que es mejor? ¿Por qué?

Precisión y velocidad de convergencia (como muestra la gráfica). Manejo automático del gradiente y optimización. Estabilidad numérica.

- Compare la secuencia target y la predicha de esta parte, ¿en qué parte falló el modelo?

El modelo suele fallar en las últimas posiciones de la secuencia. Los errores se acumulan durante el recorrido de la secuencia.

- ¿Qué sucede en el código donde se señala "NOTA 1" y "NOTA 2"? ¿Para qué son necesarias estas líneas?

NOTA 1: `net.eval()` Esta línea pone el modelo en modo evaluación.

Efectos:

- Se desactiva el dropout y el batchnorm, si existiera.
- Se asegura que los resultados sean deterministas y no influenciados por regularización estocástica.
- Es necesaria para evitar que el modelo altere su comportamiento durante la validación.

NOTA 2: `net.train()` Esta línea reactiva el modo de entrenamiento.

Efectos:

- Se activa nuevamente el dropout, batchnorm, etc.
- Permite que el optimizador acumule gradientes correctamente.
- Es fundamental para que la red entrene correctamente y no mantenga el estado de evaluación.

Parte 4 - Segunda Red Neuronal LSTM con PyTorch

Para esta parte será un poco menos guiada, por lo que se espera que puedan generar un modelo de Red Neuronal con LSTM para solventar un problema simple. Lo que se evaluará es la métrica final, y solamente se dejarán las generalidades de la implementación. El objetivo de esta parte, es dejar que ustedes exploren e investiguen un poco más por su cuenta.

En esta parte haremos uso de las redes LSTM pero para predicción de series de tiempo. Entonces lo que se busca es que dado un mes y un año, se debe predecir el número de pasajeros en unidades de miles. Los datos a usar son de 1949 a 1960.

Basado del blog "LSTM for Time Series Prediction in PyTorch" de Adrian Tam.

```
In [78]: # Seed all
import torch
import random
import numpy as np

random.seed(seed_)
np.random.seed(seed_)
torch.manual_seed(seed_)
if torch.cuda.is_available():
    torch.cuda.manual_seed(seed_)
    torch.cuda.manual_seed_all(seed_) # Multi-GPU.
```

```
torch.backends.cudnn.deterministic = True  
torch.backends.cudnn.benchmark = False
```

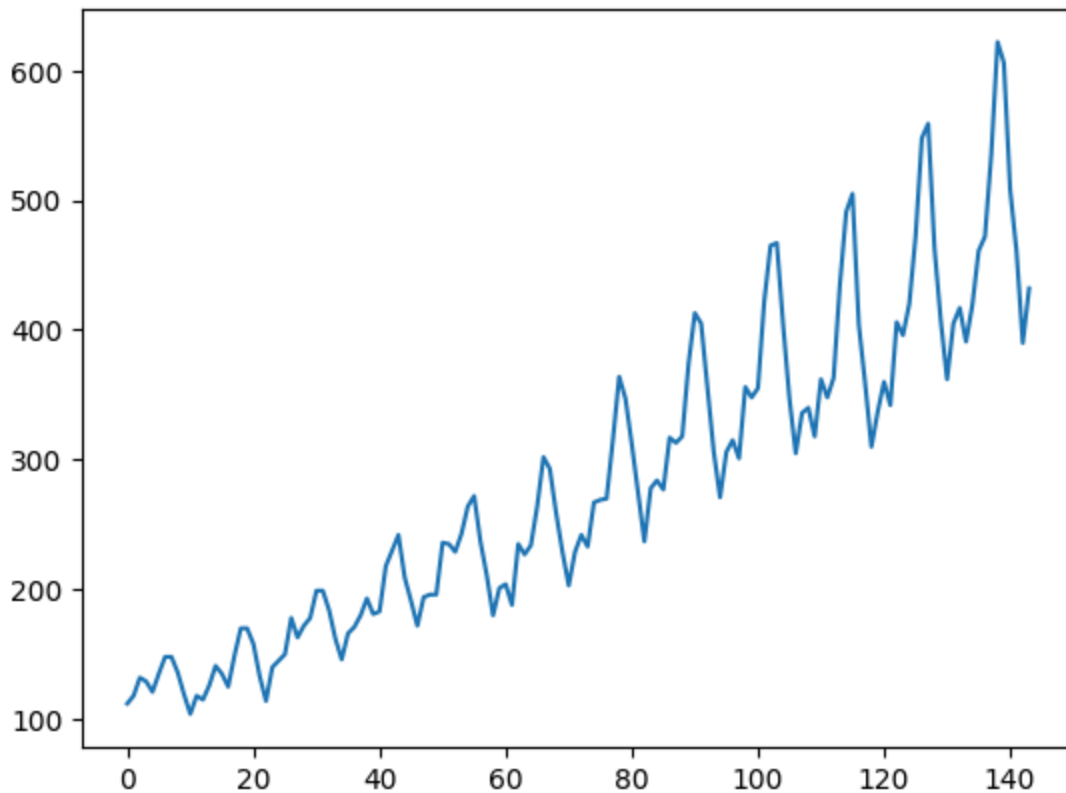
```
In [79]: import pandas as pd  
  
url_data = "https://raw.githubusercontent.com/jbrownlee/Datasets/master/airline-passengers.csv"  
dataset = pd.read_csv(url_data)  
dataset.head(10)
```

Out[79]:

	Month	Passengers
--	-------	------------

0	1949-01	112
1	1949-02	118
2	1949-03	132
3	1949-04	129
4	1949-05	121
5	1949-06	135
6	1949-07	148
7	1949-08	148
8	1949-09	136
9	1949-10	119

```
In [80]: # Dibujemos la serie de tiempo  
time_series = dataset[["Passengers"]].values.astype('float32')  
  
plt.plot(time_series)  
plt.show()
```



Esta serie de tiempo comprende 144 pasos de tiempo. El gráfico indica claramente una tendencia al alza y hay patrones periódicos en los datos que corresponden al período de vacaciones de verano. Por lo general, se recomienda "eliminar la tendencia" de la serie temporal eliminando el componente de tendencia lineal y normalizándolo antes de continuar con el procesamiento. Sin embargo, por simplicidad de este ejercicios, vamos a omitir estos pasos.

Ahora necesitamos dividir nuestro dataset en training, validation y test set. A diferencia de otro tipo de datasets, cuando se trabaja en este tipo de proyectos, la división se debe hacer sin "revolver" los datos. Para esto, podemos hacerlo con NumPy

```
In [81]: # En esta ocasion solo usaremos train y test, validation lo omitiremos para simplez
# NO CAMBIEN NADA DE ESTA CELDA POR FAVOR
p_train=0.8
p_test=0.2

# Definimos el tamaño de las particiones
num_train = int(len(time_series)*p_train)
num_test = int(len(time_series)*p_test)

# Dividir las secuencias en las particiones
train = time_series[:num_train]
test = time_series[num_train:]
```

El aspecto más complicado es determinar el método por el cual la red debe predecir la serie temporal. Por lo general, la predicción de series temporales se realiza en función de una

ventana. En otras palabras, recibe datos del tiempo t_1 al t_2 , y su tarea es predecir para el tiempo t_3 (o más adelante). El tamaño de la ventana, denotado por w , dicta cuántos datos puede considerar el modelo al hacer la predicción. Este parámetro también se conoce como **look back period** (período retrospectivo).

Entonces, creemos una función para obtener estos datos, dado un look back period. Además, debemos asegurarnos de transformar estos datos a tensores para poder ser usados con PyTorch.

Esta función está diseñada para crear ventanas en la serie de tiempo mientras predice un paso de tiempo en el futuro inmediato. Su propósito es convertir una serie de tiempo en un tensor con dimensiones (muestras de ventana, pasos de tiempo, características). Dada una serie de tiempo con t pasos de tiempo, puede producir aproximadamente $(t - \text{ventana} + 1)$ ventanas, donde "ventana" denota el tamaño de cada ventana. Estas ventanas pueden comenzar desde cualquier paso de tiempo dentro de la serie de tiempo, siempre que no se extiendan más allá de sus límites.

Cada ventana contiene múltiples pasos de tiempo consecutivos con sus valores correspondientes, y cada paso de tiempo puede tener múltiples características. Sin embargo, en este conjunto de datos específico, solo hay una función disponible.

La elección del diseño garantiza que tanto la "característica" como el "objetivo" tengan la misma forma. Por ejemplo, para una ventana de tres pasos de tiempo, la "característica" corresponde a la serie de tiempo de $t-3$ a $t-1$, y el "objetivo" cubre los pasos de tiempo de $t-2$ a t . Aunque estamos principalmente interesados en predecir $t+1$, la información de $t-2$ a t es valiosa durante el entrenamiento.

Es importante tener en cuenta que la serie temporal de entrada se representa como una matriz 2D, mientras que la salida de la función `create_timeseries_dataset()` será un tensor 3D. Para demostrarlo, usemos `lookback=1` y verifiquemos la forma del tensor de salida en consecuencia.

```
In [82]: import torch

def create_timeseries_dataset(dataset, lookback):
    X, y = [], []
    for i in range(len(dataset) - lookback):
        feature = dataset[i : i + lookback]
        target = dataset[i + 1 : i + lookback + 1]
        X.append(feature)
        y.append(target)
    return torch.tensor(X), torch.tensor(y)

# EL VALOR DE LB SÍ LO PUEDEN CAMBIAR SI LO CONSIDERAN NECESARIO
lb = 4
X_train, y_train = create_timeseries_dataset(train, lookback=lb)
#X_validation, y_validation = create_timeseries_dataset(validation, lookback=lb)
X_test, y_test = create_timeseries_dataset(test, lookback=lb)
```

```
print(X_train.shape, y_train.shape)
#print(X_validation.shape, y_validation.shape)
print(X_test.shape, y_test.shape)
```

```
torch.Size([111, 4, 1]) torch.Size([111, 4, 1])
torch.Size([25, 4, 1]) torch.Size([25, 4, 1])
```

C:\Users\josue\AppData\Local\Temp\ipykernel_29192\2018909527.py:10: UserWarning: Creating a tensor from a list of numpy.ndarrays is extremely slow. Please consider converting the list to a single numpy.ndarray with numpy.array() before converting to a tensor. (Triggered internally at C:\actions-runner_work\pytorch\pytorch\pytorch\torch\src\utils\tensor_new.cpp:257.)

```
return torch.tensor(X), torch.tensor(y)
```

Ahora necesitamos crear una clase que definirá nuestro modelo de red neuronal con LSTM. Noten que acá solo se dejaron las firmas de las funciones necesarias, ustedes deberán decidir que arquitectura con LSTM implementar, con la finalidad de superar cierto threshold de métrica de desempeño mencionado abajo.

```
In [83]: import torch.nn as nn

# NOTA: Moví el numero de iteraciones para que no se borre al ser evaluado
# Pueden cambiar el número de épocas en esta ocasión con tal de llegar al valor de
# n_epochs = 3000
# YOUR CODE HERE
n_epochs = 3000

class CustomModelLSTM(nn.Module):
    def __init__(self, input_size=1, hidden_size=100, num_layers=1, output_size=1):
        super(CustomModelLSTM, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.lstm = nn.LSTM(input_size, hidden_size, num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size, device=x.device)
        c0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size, device=x.device)
        out, _ = self.lstm(x, (h0, c0))
        return self.fc(out)
```

La función `nn.LSTM()` produce una tupla como salida. El primer elemento de esta tupla consiste en los hidden states generados, donde cada paso de tiempo de la entrada tiene su correspondiente hidden state. El segundo elemento contiene la memoria y los hidden states de la unidad LSTM, pero no se usan en este contexto particular.

La capa LSTM se configura con la opción `batch_first=True` porque los tensores de entrada se preparan en la dimensión de (muestra de ventana, pasos de tiempo, características). Con esta configuración, se crea un batch tomando muestras a lo largo de la primera dimensión.

Para generar un único resultado de regresión, la salida de los estados ocultos se procesa aún más utilizando una capa fully connected. Dado que la salida de LSTM corresponde a un valor para cada paso de tiempo de entrada, se debe seleccionar solo la salida del último paso de tiempo.

```
In [84]: import torch.optim as optim
import torch.utils.data as data

# NOTEN QUE ESTOY PONIENDO DE NUEVO LOS SEEDS PARA SER CONSTANTES
random.seed(seed_)
np.random.seed(seed_)
torch.manual_seed(seed_)
if torch.cuda.is_available():
    torch.cuda.manual_seed(seed_)
    torch.cuda.manual_seed_all(seed_) # Multi-GPU.
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False
#####

model = CustomModelLSTM()
# Optimizador y perdida
optimizer = optim.Adam(model.parameters())
loss_fn = nn.MSELoss()
# Observen como podemos también definir un DataLoader de forma snecilla
loader = data.DataLoader(data.TensorDataset(X_train, y_train), shuffle=False, batch

# Perdidas
loss_train = []
loss_test = []

# Iteramos sobre cada epoca
for epoch in range(n_epochs):
    # Colocamos el modelo en modo de entrenamiento
    model.train()

    # Cargamos Los batches
    for X_batch, y_batch in loader:
        # Obtenemos una primera prediccion
        y_pred = model(X_batch)
        # Calculamos la perdida
        loss = loss_fn(y_pred, y_batch)
        # Reseteamos la gradiente a cero
        # sino la gradiente de previas iteraciones se acumulará con Las nuevas
        optimizer.zero_grad()
        # Backprop
        loss.backward()
        # Aplicar las gradientes para actualizar los parametros del modelo
        optimizer.step()

    # Validación cada 100 epocas
    if epoch % 100 != 0 and epoch != n_epochs-1:
        continue
    # Colocamos el modelo en modo de evaluación
```

```

model.eval()

# Deshabilitamos el calculo de gradientes
with torch.no_grad():
    # Prediccion
    y_pred = model(X_train)
    # Calculo del RMSE - Root Mean Square Error
    train_rmse = np.sqrt(loss_fn(y_pred, y_train))
    # Prediccion sobre validation
    y_pred = model(X_test)
    # Calculo del RMSE para validation
    test_rmse = np.sqrt(loss_fn(y_pred, y_test))
    loss_train.append(train_rmse)
    loss_test.append(test_rmse)

print("Epoch %d: train RMSE %.4f, test RMSE %.4f" % (epoch, train_rmse, test_rm

```

C:\Users\josue\AppData\Local\Temp\ipykernel_29192\3596878564.py:57: DeprecationWarning: __array_wrap__ must accept context and return_scalar arguments (positionally) in the future. (Deprecated NumPy 2.0)

```
train_rmse = np.sqrt(loss_fn(y_pred, y_train))
```

C:\Users\josue\AppData\Local\Temp\ipykernel_29192\3596878564.py:61: DeprecationWarning: __array_wrap__ must accept context and return_scalar arguments (positionally) in the future. (Deprecated NumPy 2.0)

```
test_rmse = np.sqrt(loss_fn(y_pred, y_test))
```

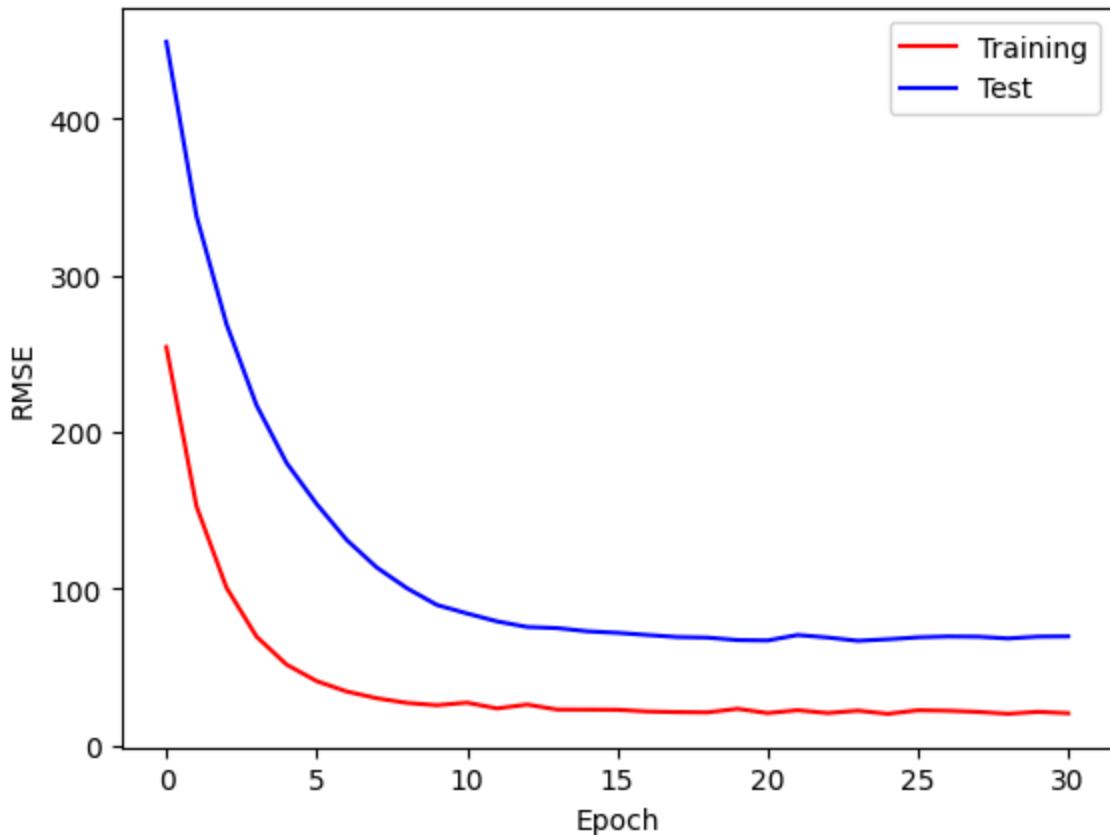
```

Epoch 0: train RMSE 254.2507, test RMSE 449.1029
Epoch 100: train RMSE 152.7753, test RMSE 337.9672
Epoch 200: train RMSE 100.7022, test RMSE 268.9265
Epoch 300: train RMSE 69.5014, test RMSE 217.1107
Epoch 400: train RMSE 51.6350, test RMSE 180.3106
Epoch 500: train RMSE 41.1036, test RMSE 154.1354
Epoch 600: train RMSE 34.4693, test RMSE 131.1687
Epoch 700: train RMSE 30.1874, test RMSE 113.6202
Epoch 800: train RMSE 27.2011, test RMSE 100.4382
Epoch 900: train RMSE 25.7186, test RMSE 89.7230
Epoch 1000: train RMSE 27.3555, test RMSE 84.2974
Epoch 1100: train RMSE 23.6798, test RMSE 79.2346
Epoch 1200: train RMSE 26.1359, test RMSE 75.6580
Epoch 1300: train RMSE 22.9601, test RMSE 74.9417
Epoch 1400: train RMSE 22.8975, test RMSE 72.9091
Epoch 1500: train RMSE 22.8253, test RMSE 71.9372
Epoch 1600: train RMSE 21.7260, test RMSE 70.5410
Epoch 1700: train RMSE 21.3315, test RMSE 69.2450
Epoch 1800: train RMSE 21.1572, test RMSE 68.8822
Epoch 1900: train RMSE 23.4459, test RMSE 67.2943
Epoch 2000: train RMSE 20.7036, test RMSE 67.0321
Epoch 2100: train RMSE 22.5545, test RMSE 70.5430
Epoch 2200: train RMSE 20.7131, test RMSE 68.9339
Epoch 2300: train RMSE 22.2680, test RMSE 66.8337
Epoch 2400: train RMSE 20.2272, test RMSE 67.8070
Epoch 2500: train RMSE 22.5126, test RMSE 69.0474
Epoch 2600: train RMSE 22.2010, test RMSE 69.6067
Epoch 2700: train RMSE 21.4505, test RMSE 69.4598
Epoch 2800: train RMSE 20.2634, test RMSE 68.3457
Epoch 2900: train RMSE 21.5049, test RMSE 69.5188
Epoch 2999: train RMSE 20.6080, test RMSE 69.6623

```



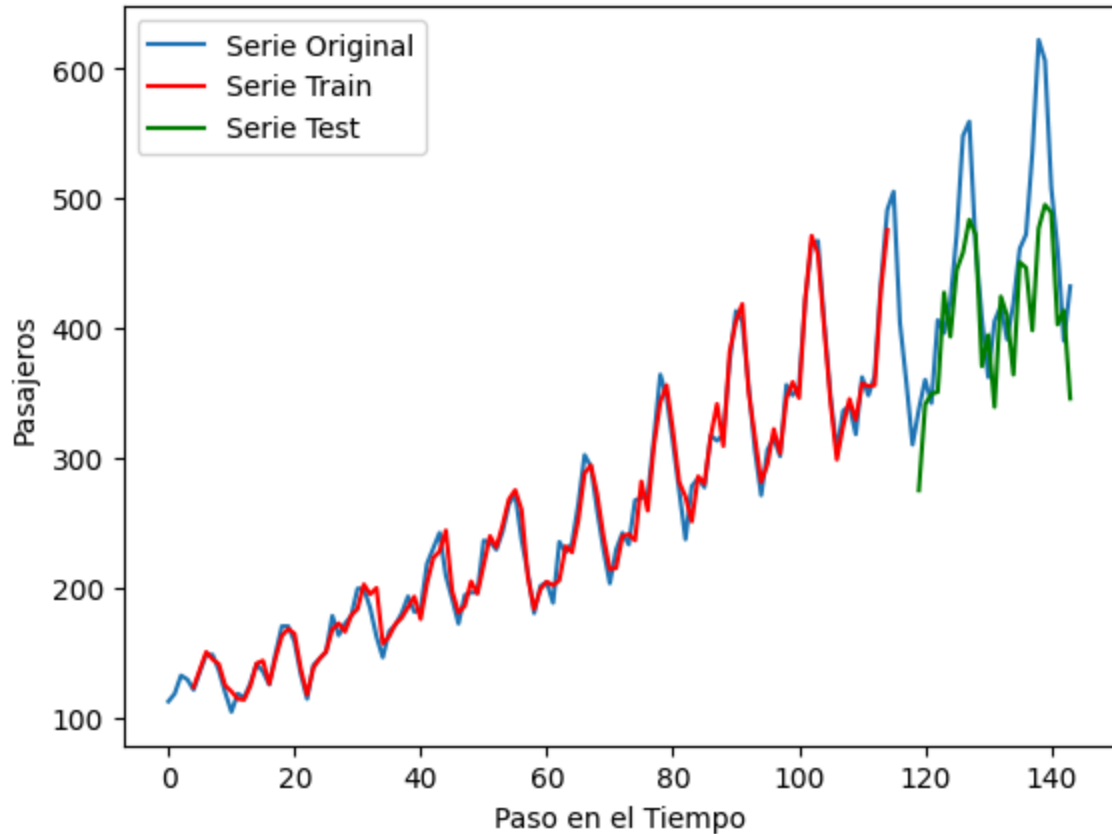
```
In [85]: # Visualización del rendimiento
epoch = np.arange(len(loss_train))
plt.figure()
plt.plot(epoch, loss_train, 'r', label='Training',)
plt.plot(epoch, loss_test, 'b', label='Test')
plt.legend()
plt.xlabel('Epoch'), plt.ylabel('RMSE')
plt.show()
```



```
In [86]: # Graficamos
with torch.no_grad():
    # Movemos las predicciones de train para graficar
    train_plot = np.ones_like(time_series) * np.nan
    # Prediccion de train
    y_pred = model(X_train)
    # Extraemos los datos solo del ultimo paso
    y_pred = y_pred[:, -1, :]
    train_plot[lb : num_train] = model(X_train)[:, -1, :]
    # Movemos las predicciones de test
    test_plot = np.ones_like(time_series) * np.nan
    test_plot[num_train + lb : len(time_series)] = model(X_test)[:, -1, :]

plt.figure()
plt.plot(time_series, label="Serie Original")
plt.plot(train_plot, c='r', label="Serie Train")
plt.plot(test_plot, c='g', label="Serie Test")
plt.xlabel('Paso en el Tiempo'), plt.ylabel('Pasajeros')
```

```
plt.legend()
plt.show()
```



Nota: Lo que se estará evaluando es el RMSE tanto en training como en test. Se evaluará que en training sea **menor a 22**, mientras que en testing sea **menor a 70**.

```
In [87]: float(loss_test[len(loss_test)-1])
float(test_rmse)
loss_train

print("\nÚltimo Train RMSE:", test_rmse)
print("\nÚltimo Test:", loss_test[-1])
with tick.marks(7):
    assert loss_train[-1] < 22

with tick.marks(7):
    assert train_rmse < 22

with tick.marks(7):
    assert loss_test[-1] < 70

with tick.marks(7):
    assert test_rmse < 70
```

Último Train RMSE: tensor(69.6623)

Último Test: tensor(69.6623)

✓ [7 marks]

✓ [7 marks]

✓ [7 marks]

✓ [7 marks]

```
In [88]: print()  
print("La fraccion de abajo muestra su rendimiento basado en las partes visibles de  
tick.summarise_marks() #
```

La fraccion de abajo muestra su rendimiento basado en las partes visibles de este laboratorio

158 / 158 marks (100.0%)

In []: