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 asegurándo que ustedes tengan una nota pronto sino también mostrandoles 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 dependecia que se necesita instalar. Ver la celda de abajo por favor

```
# 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/zipball/master

import numpy as np
import copy
import matplotlib.pyplot as plt
import scipy
from PIL import Image
import os
from collections import defaultdict

# 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, array_hash, _check_scalar
import jhwutils.image_audio as ia
import jhwutils.tick as tick
from lautils.gradeutils import new_representation, hex_to_float,
compare_numbers, compare_lists_by_percentage,
calculate_coincidences_percentage

###
tick.reset_marks()

%matplotlib inline

# Seeds
seed_ = 2023
np.random.seed(seed_)

# 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)

```
carne_1 = "21752"
firma_mecanografiada_1 = "Diego Leiva"
carne_2 = "21970"
firma_mecanografiada_2 = "Pablo Orellana"

# Deberia poder ver dos checkmarks verdes [0 marks], que indican que
su información básica está OK

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)
```

```
<IPython.core.display.HTML object>

<IPython.core.display.HTML object>
```

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.

Representanción 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 enconding, 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 $t_1, \ldots, t_{n-1}$.

```
# 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 "indice" 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.

```
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 unicas
    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):
        word_to_idx[word] = idx
        idx_to_word[idx] = word


    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 incluyendo UNK")
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
```

```
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'))
```

```
<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>
```

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 estamso 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 eficienciente, al ser este un generador/iterador. Además, este nos sirve para obtener la información en batches.

```
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 secuencia de largo L
        # pero los targetes están movidos a la derecha por uno, para
que podamos predecir la siguiente palabra
        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.

```
def one_hot_encode(idx, vocab_size):
    """
    Encodea una sola palabra dado su indice y el tamaño del
vocabulario

    Args:
     idx: indice 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 memroia 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 facil referirse a un paso t. Tenemos los siguientes calculos en la red

- $h_t = f$ ¿ donde f es la función de activacion
- $L_t = softmax(W h_t)$

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.

```
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
    """
    # Definir la matriz de pesos (input del hidden state)
```

```
    U = np.zeros((hidden_size, vocab_size))
    # Definir la matriz de pesos de los calculos recurrentes
    V = np.zeros((hidden_size, hidden_size))
    # Definir la matriz de pesos del hidden state a la salida
    W = np.zeros((vocab_size, hidden_size))
    # Bias del hidden state
    b_hidden = np.zeros((hidden_size, 1))
    # Bias de la salida
    b_out = np.zeros((vocab_size, 1))
    # Para estas use np.zeros y asegurese de darle las dimensiones
correcta a cada elemento

    # Inicializar los pesos de forma ortogonal usando la
    # funcion init_orthogonal
    U = init_orthogonal(U)
    V = init_orthogonal(V)
    W = init_orthogonal(W)


    # Return parameters as a tuple
    return U, V, W, b_hidden, b_out


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

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))
```

```
<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>
```

Funciones de Activación

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

```
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

    f = 1 / (1 + np.exp(-x_safe))

    # 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

    f = (np.exp(x_safe) - np.exp(-x_safe)) / (np.exp(x_safe) +
np.exp(-x_safe))

    # 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

    f = np.exp(x_safe) / np.sum(np.exp(x_safe))

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

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))
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

Implementación del paso Forward

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

```
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)):

        # Calculo del nuevo hidden state usando tanh
        # Recuerden que al ser el hidden state tienen que usar los
pesos del input multiplicado por el input
        #  a esto sumarle los pesos recurrentes por el hidden state y
finalmente sumarle b
        hidden_state = tanh(np.dot(U, inputs[t]) + np.dot(V,
hidden_state) + b_hidden)


        # Ccalculo del output
        # Al ser la salida, deben usar softmax sobre la multiplicación
de pesos de salida con el hidden_state actual
        #   es decir el calculado en el paso anterior y siempre
sumarle su bias correspondiente
        out = softmax(np.dot(W, hidden_state) + b_out)

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

    return outputs, hidden_states

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', 'b', 'b', 'b', 'b', 'b', 'b', 'b',
'b', 'EOS']
Secuencia Predicha:
['a', 'b', 'a', 'a', 'a', 'EOS', 'EOS', 'EOS', 'EOS', 'EOS', 'EOS',
'EOS', 'b', 'b', 'b', 'b']

<IPython.core.display.HTML object>
```

Implementación del paso Backward

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

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

Understanding Gradient Clipping (and How It Can Fix Exploding Gradients Problem)

What exactly happens in gradient clipping by norm?

```
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 norma total
    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 gradiente
    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 bias)
    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))):

        # Aprox 1 linea para calcular la perdida cross-entry (un
escalar)
        # Hint: Sumen +1e-12 a cada output_t
        # Hint2: Recuerden que la perdida es el promedio de
multiplicar el logaritmo de los output con los targets
        loss += -np.mean(np.log(outputs[t] + 1e-12) * targets[t])


        d_o = outputs[t].copy()
        # Aprox 1 linea para backpropagate en los output (derivada del
cross-entropy)
        # Si se sienten perdidos refieran a esta lectura:
```

```
http://cs231n.github.io/neural-networks-case-study/#grad
        d_o[np.argmax(targets[t])] -= 1

        # Backpropagation de W
        d_W += np.dot(d_o, hidden_states[t].T)

        d_b_out += d_o

        # Backpropagation de h
        d_h = np.dot(W.T, d_o) + d_h_next
        # Hint: Probablemente necesiten sacar la transpuesta de W
        # Hint2: Recuerden sumar el bias correcto!

        # Calcular el backprop en la funcion de activacion tanh
        d_f = d_h * tanh(hidden_states[t], derivative=True)
        # Hint: Recuerden pasar el parametro derivate=True a la
funcion que definimos
        # Hint2: Deben multiplicar con d_h

        d_b_hidden += d_f

        # Backpropagation en U
        d_U += np.dot(d_f, inputs[t].T)


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

        d_h_next = np.dot(V.T, d_f)

    # Empaquetar las gradientes
    grads = d_U, d_V, d_W, d_b_hidden, d_b_out

    # Corte de gradientes
    grads = clip_gradient_norm(grads)

    return loss, grads

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))
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

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 descediente. Recuerden la regla de optimizacion

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

- 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

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

```
# Hyper parametro
# Se coloca como "repsuesta" para que la herramienta no modifique el
numero de iteraciones que colocaron
num_epochs = 2000


# Init una nueva RNN
```

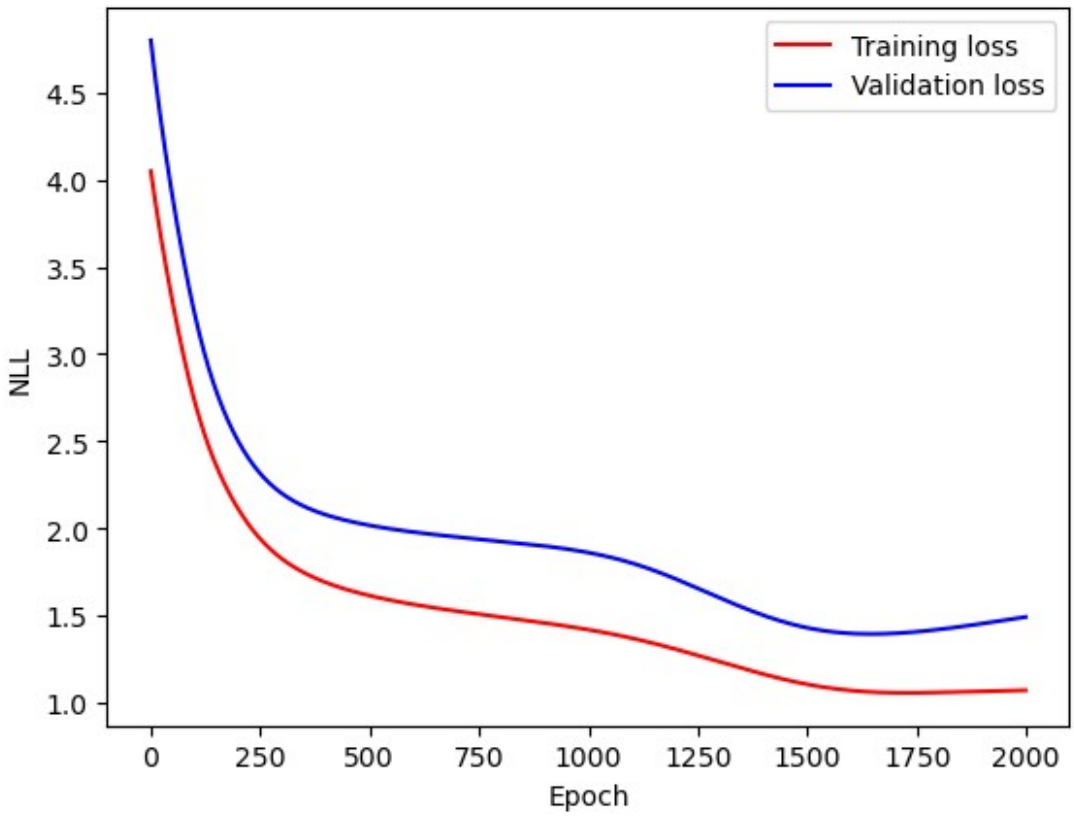
```
params = init_rnn(hidden_size=hidden_size, vocab_size=vocab_size)

# Init hiddent 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 zero
    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)

        # Paso forward
        outputs, hidden_states = forward_pass(inputs_one_hot,
hidden_state, params)

        # Paso backward
        loss, _ = backward_pass(inputs_one_hot, outputs,
hidden_states, targets_one_hot, params)

        # Actualización de perdida
        epoch_validation_loss += loss

    # For each sentence in 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_one_hot = one_hot_encode_sequence(targets, vocab_size)

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

        # Paso forward
        outputs, hidden_states = forward_pass(inputs_one_hot,
hidden_state, params)
```

```
        # Paso backward
        loss, grads = backward_pass(inputs_one_hot, outputs,
hidden_states, targets_one_hot, params)

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

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

        # Actualización de perdida
        epoch_training_loss += loss

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

    # Mostrar la perdida cada 100 epocas
    if i % 100 == 0:
        print(f'Epoca {i}, training loss: {training_loss[-1]},
validation loss: {validation_loss[-1]}')
```

```
Epoca 0, training loss: 4.05046509496538, validation loss:
4.801971835967156
Epoca 100, training loss: 2.729834076574944, validation loss:
3.2320576163982673
Epoca 200, training loss: 2.1094146557367317, validation loss:
2.498052632884415
Epoca 300, training loss: 1.8235746981413405, validation loss:
2.198677070984531
Epoca 400, training loss: 1.6884087861997368, validation loss:
2.077078608023497
Epoca 500, training loss: 1.6129170568126512, validation loss:
2.016354394171658
Epoca 600, training loss: 1.5624028954062006, validation loss:
1.9780311638492247
Epoca 700, training loss: 1.5235019197917083, validation loss:
1.9496130467843362
Epoca 800, training loss: 1.489582803129218, validation loss:
1.9248315278145836
Epoca 900, training loss: 1.4558865884071523, validation loss:
1.897822091215437
Epoca 1000, training loss: 1.4173709332614934, validation loss:
1.8600798176555249
Epoca 1100, training loss: 1.368178363440396, validation loss:
1.799369702641401
Epoca 1200, training loss: 1.3051122158818909, validation loss:
```

```
1.7081695076503602
Epoca 1300, training loss: 1.2330985128125058, validation loss:
1.5999314734390115
Epoca 1400, training loss: 1.1619900522538624, validation loss:
1.4998577602386756
Epoca 1500, training loss: 1.1035554777966472, validation loss:
1.428263841611046
Epoca 1600, training loss: 1.0680633416284255, validation loss:
1.3958745915871216
Epoca 1700, training loss: 1.0550402179563663, validation loss:
1.3963674481755957
Epoca 1800, training loss: 1.0570111001893738, validation loss:
1.4185760443851878
Epoca 1900, training loss: 1.0640880623573372, validation loss:
1.4524183517051117

# 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 evalular 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 output in outputs], 65)

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', 'b', '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']
```




```
<IPython.core.display.HTML object>
```

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 entre las gráficas de pérdida de entrenamiento (training loss) y validación (validation loss) indica cómo está generalizando el modelo a datos no vistos durante el

entrenamiento. Una pequeña separación entre estas gráficas generalmente sugiere que el modelo está generalizando bien y no está sobreajustado. Sin embargo, si la separación es grande, podría indicar que el modelo está sobreajustado, es decir, ha aprendido demasiado bien el conjunto de entrenamiento y no se desempeña bien en datos nuevos.

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

- No, evaluar un modelo basado únicamente en la pérdida (loss) no es suficiente. La pérdida puede proporcionar una buena indicación del rendimiento del modelo, pero es crucial considerar otras métricas como la precisión (accuracy), la precisión y el recall, la F1-score, y otras métricas específicas del dominio del problema.

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

- En un modelo ideal, las gráficas de pérdida de entrenamiento y de validación deberían converger hacia valores bajos y mantenerse cercanas entre sí. Esto indica que el modelo está aprendiendo de manera efectiva sin sobreajustar. La pérdida de entrenamiento debería disminuir constantemente, y la pérdida de validación debería disminuir en paralelo hasta estabilizarse, lo que sugiere que el modelo está generalizando bien a datos no vistos.

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 Frellsen (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 imagenes 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 "celula" 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 $\sigma = softmax$. Entonces tenemos la input gate i, la forget gate f y la output gate o

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

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

Donde W^i, W^f, W^o son las matrices de pesos aplicada a cada aplicadas a una matriz contatenada 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

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

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

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

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.

```
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
    """

    # Empezar la matriz de pesos de la forget gate
    # Recuerden que esta debe empezar con numeros aleatorios
    W_f = np.random.randn(hidden_size, z_size)
```

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

    # Empezar la matriz de pesos de la input gate
    # Recuerden que esta debe empezar con numeros aleatorios
    W_i = np.random.randn(hidden_size, z_size)

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

    # Empezar la matriz de pesos para la memoria candidata
    # Recuerden que esta debe empezar con numeros aleatorios
    W_g = np.random.randn(hidden_size, z_size)

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

    # Empezar la matriz de pesos para la output gate
    W_o = np.random.randn(hidden_size, z_size)

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


    # Empezar la matriz que relaciona el hidden state con el output
    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)

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))
```

```
<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>
```

Forward

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

```
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 el output, shape (n_v, n_a)
                        b_v: Bias que relaciona el hidden state con el
output, shape (n_v, 1)
    Returns:
    z_s, f_s, i_s, g_s, C_s, o_s, h_s, v_s: Lista de tamaño m
conteniendo los calculos de cada paso forward
    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:

        # Concatenar el input y el hidden state
        z = np.row_stack((h_prev, x))

        z_s.append(z)

        # Calcular el forget gate
        # Hint: recuerde usar sigmoid
        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)

        # Calcular el estado de la memoria
        C_prev = f * C_prev + i * g
        C_s.append(C_prev)

        # Calculo de la output gate
        # Hint: recuerde usar sigmoid
        o = sigmoid(np.dot(W_o, z) + b_o)
        o_s.append(o)

        # Calculo del hidden state
        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

# 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, 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])


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', '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', 'EOS', 'EOS',
'EOS', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b']

<IPython.core.display.HTML object>
```

Backward

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

```
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 el output, shape (n_v, n_a)
                        b_v: Bias que relaciona el hidden state con el
output, shape (n_v, 1)
    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))):

        # Calcular la perdida con cross entropy
        loss += -np.mean(np.log(outputs[t] + 1e-12) * targets[t])

        # Obtener el hidden state del estado previo
        C_prev= C[t-1]

        # Calculo de las derivadas en relacion del hidden state al
output gate
        dv = np.copy(outputs[t])
        dv[np.argmax(targets[t])] -= 1

        # Actualizar la gradiente de la relacion del hidden-state al
output gate
        W_v_d += np.dot(dv, h[t].T)
        b_v_d += dv

        # Calculo de la derivada del hidden state y el output gate
        dh = np.dot(W_v.T, dv)
        dh += dh_next
        do = dh * tanh(C[t])

        # Calcular la derivada del output
        do = sigmoid(o[t], True) * do
        # Hint: Recuerde multiplicar por el valor previo de do (el de
arriba)
```

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

        # Calculo de las derivadas del estado y la memoria candidata g
        dC = np.copy(dC_next)
        dC += dh * o[t] * tanh(tanh(C[t]), derivative=True)
        dg = dC * i[t]

        # Terminar el calculo de dg
        dg = tanh(g[t], True) * dg

        # Actualización de las gradientes con respecto de la mem
candidata
        W_g_d += np.dot(dg, z[t].T)
        b_g_d += dg

        # Calculo de la derivada del input gate y la actualización de
sus gradientes
        di = dC * g[t]
        di = sigmoid(i[t], True) * di

        # Calculo de los pesos y bias del input gate
        W_i_d += np.dot(di, z[t].T)
        b_i_d += di

        # Calculo de las derivadas del forget gate y actualización de
sus gradientes
        df = dC * C_prev
        df = sigmoid(f[t]) * df

        # Calculo de los pesos y bias de la forget gate
        W_f_d += np.dot(df, z[t].T)
        b_f_d += df

        # Calculo de las derivadas del input y la actualizacion de
gradientes del hidden state previo
        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

    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

# 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_hot, params)

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

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

```
Perdida obtenida:7.637217940741176
```

```
<IPython.core.display.HTML object>
```

Training

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

```
# 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, c, params)

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

        # 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, c, params)

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

        # 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: {validation_loss[-1]}')
```

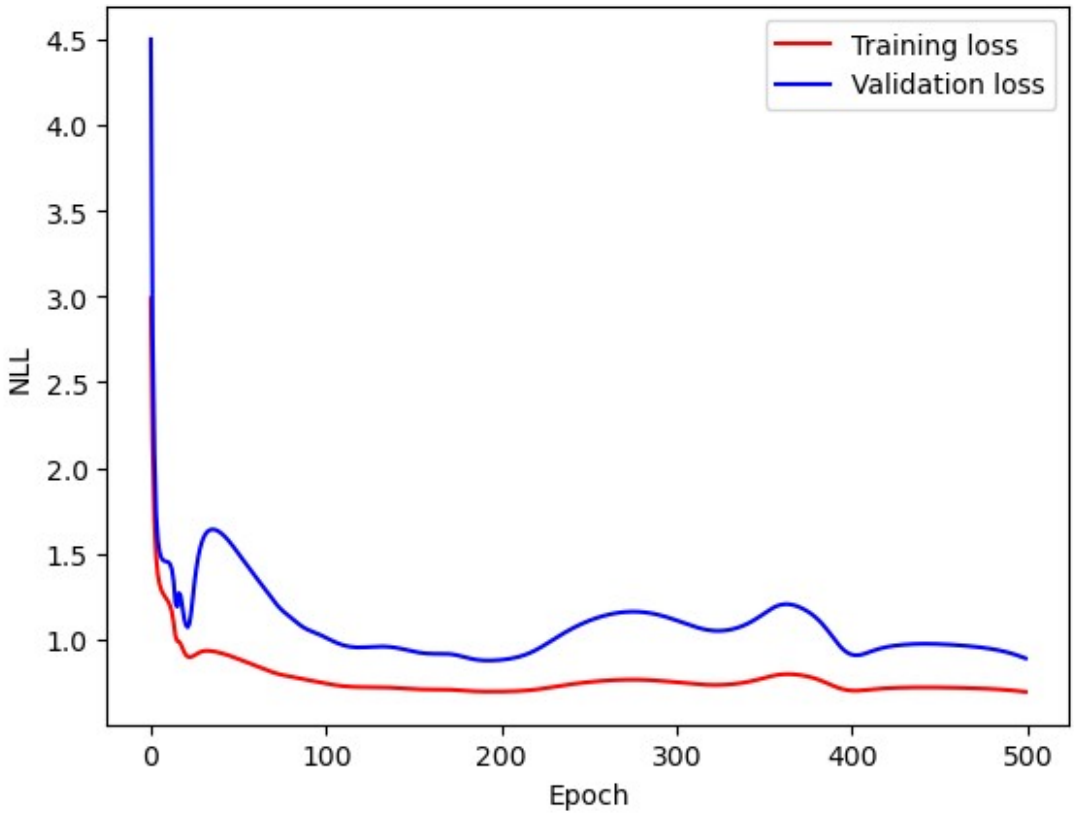
```
Epoch 0, training loss: 2.9885565716555442, validation loss:
4.499707061158503
Epoch 10, training loss: 1.2170995637192898, validation loss:
1.4488214228788996
Epoch 20, training loss: 0.9073644447149842, validation loss:
1.0815213281697804
Epoch 30, training loss: 0.9303750511190998, validation loss:
1.5909496801342096
Epoch 40, training loss: 0.9187082336869414, validation loss:
1.619079602026829
Epoch 50, training loss: 0.883855860160882, validation loss:
1.4990399685803517
Epoch 60, training loss: 0.8430567008469587, validation loss:
1.3609169235891538
Epoch 70, training loss: 0.8050372301526325, validation loss:
1.2239162533423624
Epoch 80, training loss: 0.7809193343593629, validation loss:
1.1246054751717889
Epoch 90, training loss: 0.7600330437761145, validation loss:
1.0526780777870859
Epoch 100, training loss: 0.7412121295737716, validation loss:
1.0079189357647842
Epoch 110, training loss: 0.7254067392295772, validation loss:
0.9637842708236972
Epoch 120, training loss: 0.7202030582604173, validation loss:
0.9518831884921604
Epoch 130, training loss: 0.7194397177517484, validation loss:
0.9567955068790919
Epoch 140, training loss: 0.7154953247105261, validation loss:
0.949474862408524
Epoch 150, training loss: 0.7088698499088, validation loss:
0.9273671799306765
Epoch 160, training loss: 0.7059253382773187, validation loss:
0.914889165825681
Epoch 170, training loss: 0.7052924506315916, validation loss:
0.9135747466313312
Epoch 180, training loss: 0.6985401734887426, validation loss:
0.8918252303975756
Epoch 190, training loss: 0.6941816345113876, validation loss:
0.8758085908794631
Epoch 200, training loss: 0.6942684719391353, validation loss:
0.8791512120619107
Epoch 210, training loss: 0.6974052532416868, validation loss:
0.8973331347104428
Epoch 220, training loss: 0.7062945988245934, validation loss:
0.938425796453282
Epoch 230, training loss: 0.7216060934773494, validation loss:
0.9999963361081032
Epoch 240, training loss: 0.7373005079847916, validation loss:
1.0602555584405837
```

```
Epoch 250, training loss: 0.7497432590038592, validation loss:
1.107605977845815
Epoch 260, training loss: 0.7584753648510025, validation loss:
1.14056200048906
Epoch 270, training loss: 0.7630605106472166, validation loss:
1.1579921733401533
Epoch 280, training loss: 0.7628045874396161, validation loss:
1.1582162982001603
Epoch 290, training loss: 0.7574272018989955, validation loss:
1.1407415819479314
Epoch 300, training loss: 0.7480212844117256, validation loss:
1.108893619305438
Epoch 310, training loss: 0.7379643670440981, validation loss:
1.07296599508396
Epoch 320, training loss: 0.7325371067411459, validation loss:
1.0502167720713307
Epoch 330, training loss: 0.7358520224631776, validation loss:
1.0543313729411607
Epoch 340, training loss: 0.7495176769266053, validation loss:
1.0883205913716707
Epoch 350, training loss: 0.7740690948110396, validation loss:
1.1510570053509208
Epoch 360, training loss: 0.7943872549167433, validation loss:
1.2019354729266538
Epoch 370, training loss: 0.7910334176807681, validation loss:
1.1866317936970396
Epoch 380, training loss: 0.7667557816086469, validation loss:
1.1200050465897564
Epoch 390, training loss: 0.7242461959698461, validation loss:
0.9978593449096957
Epoch 400, training loss: 0.6994646748025939, validation loss:
0.9090963464592002
Epoch 410, training loss: 0.705335025002721, validation loss:
0.9251312017282656
Epoch 420, training loss: 0.7137177022923847, validation loss:
0.9546990913951188
Epoch 430, training loss: 0.7174796634941857, validation loss:
0.9680080894752636
Epoch 440, training loss: 0.7187181648061347, validation loss:
0.9725416272546239
Epoch 450, training loss: 0.7182025550352744, validation loss:
0.9709316019956553
Epoch 460, training loss: 0.7164488026916438, validation loss:
0.9651077128220743
Epoch 470, training loss: 0.713620883780454, validation loss:
0.9558919777498074
Epoch 480, training loss: 0.7092569990186645, validation loss:
0.9420090293356791
```

```
Epoch 490, training loss: 0.702062240862966, validation loss:
0.9193046946379606
```

```
# 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, params)

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', 'b']
Secuencia Target:
['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:
['a', 'a', 'a', 'a', 'a', 'a', 'b', 'b', '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 funcionó mejor que la RNN tradicional. Esto se debe a que las LSTM están diseñadas para manejar dependencias a largo plazo en las secuencias de datos. Las RNN tradicionales tienden a sufrir de desvanecimiento y explosión de gradientes, lo que dificulta la capacidad del modelo para aprender y retener información a través de largas secuencias. Las LSTM superan este problema utilizando puertas de entrada, olvido y salida, lo que les permite controlar el flujo de información y mantener estados de memoria durante períodos más largos.

Observen la gráfica obtenida arriba, ¿en qué es diferente a la obtenida a RNN? ¿Es esto mejor o peor? ¿Por qué?

- La gráfica obtenida con la LSTM muestra una pérdida de validación que sigue más de cerca a la pérdida de entrenamiento y se estabiliza en valores más bajos en comparación con la RNN. Esto es mejor porque indica que el modelo LSTM está generalizando mejor a datos no vistos durante el entrenamiento. La menor pérdida de validación sugiere que el modelo no solo ha aprendido los datos de entrenamiento, sino que también es capaz de predecir correctamente en el conjunto de validación, lo que es un indicador de un mejor rendimiento general del modelo.

¿Por qué LSTM puede funcionar mejor con secuencias largas?

- Las LSTM pueden funcionar mejor con secuencias largas debido a su capacidad para mantener y actualizar un estado de memoria a través de muchas etapas de tiempo. Utilizan puertas de entrada, olvido y salida para controlar el flujo de información, lo que les permite recordar información importante durante largos períodos y olvidar información irrelevante. Esto mitiga los problemas de desvanecimiento y explosión de gradientes que suelen afectar a las RNN tradicionales.

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

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

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

        # Declarar una capa LSTM
        self.lstm = nn.LSTM(input_size = vocab_size,
                            hidden_size = 50,
                            num_layers = 1,
                            bidirectional = False)
        # Hint: Esta tiene que tener el input_size del tamaño del
vocabulario,
        #     debe tener 50 hidden states (hidden_size)
        #     una layer
        #     y NO (False) debe ser bidireccional


        # 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)
)
```

```
# Hyper parametros
num_epochs = 500

# Init una nueva red
net = Net()

# Definir la función de perdida y el optimizador
criterion = nn.CrossEntropyLoss() # Use CrossEntropy
optimizer = optim.Adam(net.parameters(), lr=3e-4) # Use Adam con
lr=3e-4

# 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 inpyt 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)

        # Paso Forward
```

```
        outputs = net(inputs_one_hot)

        # Calcular la perdida
        loss = criterion(outputs, targets_idx)
        # Hint: Use el criterion definido arriba

        # 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 inpyt 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)

        # Paso Forward
        outputs = net(inputs_one_hot)

        # Calcular la perdida
        loss = criterion(outputs, targets_idx)
        # Hint: Use el criterion definido arriba

        # Definir el backward
        optimizer.zero_grad()   # Reiniciar los gradientes
        loss.backward()         # Calcular los gradientes
        optimizer.step()        # Actualizar los pesos

        # 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: {validation_loss[-1]}')
```

```
Epoch 0, training loss: 1.3063723415136337, validation loss:
1.3868818879127502
Epoch 10, training loss: 0.5637635737657547, validation loss:
0.532869154214859
Epoch 20, training loss: 0.4247903060168028, validation loss:
0.3874460577964783
Epoch 30, training loss: 0.3668004373088479, validation loss:
0.3227157711982727
Epoch 40, training loss: 0.33821501173079016, validation loss:
0.29906695485115053
Epoch 50, training loss: 0.32130545992404225, validation loss:
0.2884485274553299
Epoch 60, training loss: 0.31303826849907634, validation loss:
0.29442181140184404
Epoch 70, training loss: 0.30476649552583696, validation loss:
0.27418791204690934
Epoch 80, training loss: 0.3008359640836716, validation loss:
0.27088542729616166
Epoch 90, training loss: 0.29801501650363205, validation loss:
0.2686256095767021
Epoch 100, training loss: 0.29595681801438334, validation loss:
0.26708156019449236
Epoch 110, training loss: 0.2944457795470953, validation loss:
0.2661609321832657
Epoch 120, training loss: 0.36146851237863303, validation loss:
0.26604084968566893
Epoch 130, training loss: 0.292103561013937, validation loss:
0.2674125641584396
Epoch 140, training loss: 0.29186920281499623, validation loss:
0.26690469682216644
Epoch 150, training loss: 0.29162680320441725, validation loss:
0.2669309049844742
Epoch 160, training loss: 0.2913685435429215, validation loss:
0.2673835828900337
Epoch 170, training loss: 0.29110332410782574, validation loss:
0.2681694239377975
Epoch 180, training loss: 0.2908589828759432, validation loss:
0.26915804743766786
Epoch 190, training loss: 0.2906112980097532, validation loss:
0.2703138068318367
Epoch 200, training loss: 0.289594485424459, validation loss:
0.27337429523468015
Epoch 210, training loss: 0.28963610958307984, validation loss:
0.2729730561375618
Epoch 220, training loss: 0.2897568840533495, validation loss:
0.27325090765953064
Epoch 230, training loss: 0.289841597341001, validation loss:
0.27365351617336275
Epoch 240, training loss: 0.28988458346575496, validation loss:
0.27409162521362307
```

```
Epoch 250, training loss: 0.2898931039497256, validation loss:
0.2745038241147995
Epoch 260, training loss: 0.2898803574964404, validation loss:
0.2748549982905388
Epoch 270, training loss: 0.3646071722730994, validation loss:
0.2755950212478638
Epoch 280, training loss: 0.2890165474265814, validation loss:
0.27642074078321455
Epoch 290, training loss: 0.28919076174497604, validation loss:
0.27599202692508695
Epoch 300, training loss: 0.2893316922709346, validation loss:
0.27576281130313873
Epoch 310, training loss: 0.28943439535796645, validation loss:
0.2756440624594688
Epoch 320, training loss: 0.28949890211224555, validation loss:
0.2756026864051819
Epoch 330, training loss: 0.2895283615216613, validation loss:
0.2756101742386818
Epoch 340, training loss: 0.28952716160565617, validation loss:
0.2756371319293976
Epoch 350, training loss: 0.28950339313596485, validation loss:
0.2756565690040588
Epoch 360, training loss: 0.28946761433035134, validation loss:
0.27563548982143404
Epoch 370, training loss: 0.2886753806844354, validation loss:
0.27709122002124786
Epoch 380, training loss: 0.28871708586812017, validation loss:
0.2759151592850685
Epoch 390, training loss: 0.28886076621711254, validation loss:
0.27577117532491685
Epoch 400, training loss: 0.28897461425513027, validation loss:
0.27567042857408525
Epoch 410, training loss: 0.2890548471361399, validation loss:
0.2756019040942192
Epoch 420, training loss: 0.28910430800169706, validation loss:
0.27556965351104734
Epoch 430, training loss: 0.289126967638731, validation loss:
0.2755678594112396
Epoch 440, training loss: 0.2891259633004665, validation loss:
0.27558509111404417
Epoch 450, training loss: 0.2891049986705184, validation loss:
0.27560639530420306
Epoch 460, training loss: 0.2890695910900831, validation loss:
0.27562153786420823
Epoch 470, training loss: 0.289024512656033, validation loss:
0.27563034147024157
Epoch 480, training loss: 0.28897376023232935, validation loss:
0.27563616931438445
```

```
Epoch 490, training loss: 0.2889199798926711, validation loss:
0.2756426602602005
```

```
with tick.marks(5):
    assert compare_numbers(new_representation(training_loss[-1]),
"3c3d", '0x1.28f5c28f5c28fp-2')

with tick.marks(5):
    assert compare_numbers(new_representation(validation_loss[-1]),
"3c3d", '0x1.28f5c28f5c28fp-2')
```

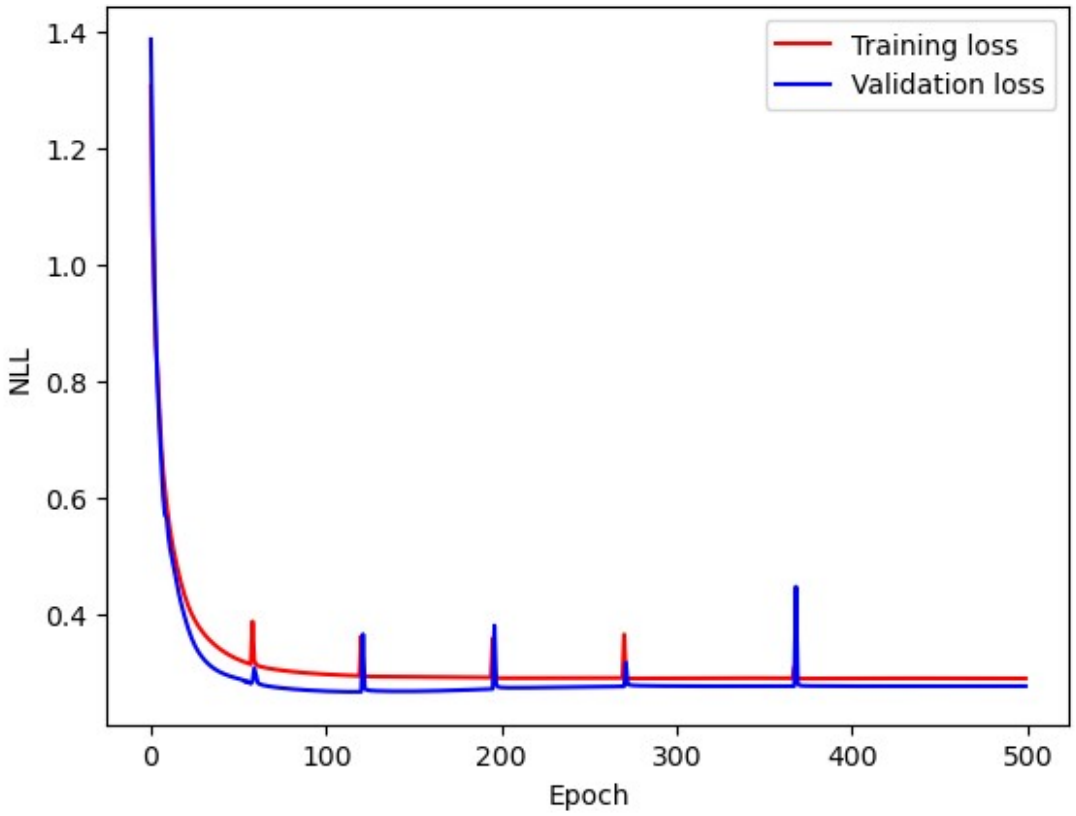
```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
# 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_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)

# Convertir outputs a NumPy para el procesamiento
outputs_np = outputs.detach().numpy()

# Paso Forward
outputs = net(inputs_one_hot)

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

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

# Convertir las predicciones a índices de palabras
predicted_indices = [np.argmax(output_np, axis=-1) for output_np in
outputs_np]

# Convertir los índices a palabras
predicted_words = [idx_to_word[idx] for idx in predicted_indices]

print("Secuencia Predicha:")
print(predicted_words)
```

```
# 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', 'b']
Secuencia Target:
['a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'b', 'b', 'b', 'b',
'b', 'b', 'b', 'b', 'b', 'b', 'b', 'EOS']
Secuencia Predicha:
['a', 'a', 'a', 'a', 'a', 'a', 'b', 'b', 'b', 'b', 'b', 'EOS']
```



Preguntas

Responda 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?

- Al comparar las gráficas obtenidas para el LSTM "a mano" y el LSTM utilizando PyTorch, podemos observar que la gráfica de PyTorch muestra una convergencia más rápida y estable. Esto es mejor porque indica que PyTorch maneja de manera más eficiente la optimización y el cálculo de gradientes, gracias a sus optimizadores avanzados (como Adam) y su manejo interno de las operaciones tensoriales. Además, PyTorch proporciona mayor estabilidad y precisión numérica, lo cual contribuye a una mejor convergencia del modelo.

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

- Secuencia Target: ['a', 'a', '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: ['a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'EOS'] El modelo predijo correctamente toda la secuencia target en esta iteración. Sin embargo, en casos donde haya diferencias, fallos pueden ocurrir principalmente en:
- La transición entre secuencias de 'a' a 'b' o de 'b' a 'EOS'.
- La longitud de la secuencia predicha puede ser incorrecta si el modelo no maneja bien los tokens de inicio y fin de secuencia.

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

- **En nota1:** Cambia el modelo a modo de evaluación. En modo de evaluación, ciertas capas como Dropout y BatchNorm se comportan de manera diferente. Esto es crucial para obtener una evaluación precisa del rendimiento del modelo en los datos de validación.
- **En nota2:** Cambia el modelo de nuevo a modo de entrenamiento. En modo de entrenamiento, se habilitan las características como Dropout y BatchNorm para mejorar la generalización del modelo durante el entrenamiento.

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 este 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.

```
# Seed all
import torch
import random
import numpy as np


random.seed(seed_)
```

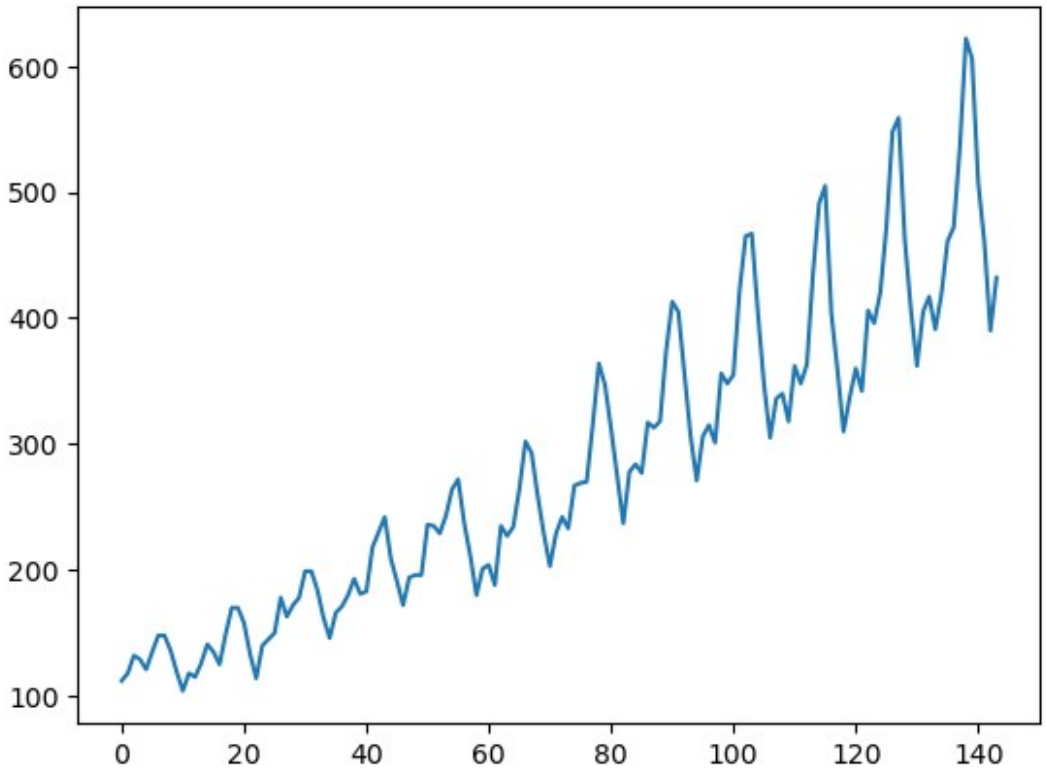
```
np.random.seed(seed_)
torch.manual_seed(seed_)
if torch.cuda.is_available():
    torch.cuda.manual_seed(seed_)
    torch.cuda.manual_seed_all(seed_)  # Multi-GPU.
    print("Using CUDA")
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False
```

```
Using CUDA
```

```
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)
```

```
     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
```

```
# 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

```
# En esta ocasion solo usaremos train y test, validation lo omitiremos
para simpleza del ejercicio
# 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 t1 al t2, y su tarea es predecir para el tiempo t3 (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 - 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.

```
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)

    # Para evitar UserWarning: Creating a tensor from a list of
numpy.ndarrays is extremely slow.
    # Convertir listas a numpy arrays y luego a tensores
    X = np.array(X)
    y = np.array(y)
    return torch.tensor(X, dtype=torch.float32), torch.tensor(y,
```

```
dtype=torch.float32)


# EL VALOR DE LB SÍ LO PUEDEN CAMBIAR SI LO CONSIDERAN NECESARIO
lb = 10
X_train, y_train = create_timeseries_dataset(train, lookback=lb)
X_test, y_test = create_timeseries_dataset(test, lookback=lb)

print(X_train.shape, y_train.shape)
print(X_test.shape, y_test.shape)
```

```
torch.Size([105, 10, 1]) torch.Size([105, 10, 1])
torch.Size([19, 10, 1]) torch.Size([19, 10, 1])
```

Ahora necesitamos crear una clase que definirá nuestro modelo de red neuronal con LSTM. Noten que acá solo se dejaran 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.

```
import torch.nn as nn

# NOTA: Moví el numero de iteraciones para que no se borre al ser
evaluado
# Pueden cambiar el número de epocas en esta ocasión con tal de llegar
al valor de la metrica de desempeño
n_epochs = 2500

class CustomModelLSTM(nn.Module):
    def __init__(self, input_size, hidden_size, output_size,
num_layers):
        super(CustomModelLSTM, self).__init__()

        # Definir la capa LSTM
        self.lstm = nn.LSTM(input_size = input_size,
                            hidden_size = hidden_size,
                            num_layers = num_layers,
                            batch_first = True)

        # Definir la capa de salida
        self.linear = nn.Linear(hidden_size,
                                output_size)

        # Agregar una capa de activación ReLU
        self.relu = nn.ReLU()

        self.hidden_size = hidden_size  # Numero de hidden states
        self.num_layers = num_layers    # Numero de capas


    def forward(self, x):
```

```
        # Inicializar los hidden states
        h0 = torch.zeros(self.lstm.num_layers, x.size(0),
self.lstm.hidden_size)
        c0 = torch.zeros(self.lstm.num_layers, x.size(0),
self.lstm.hidden_size)

        # Forward pass a la capa LSTM
        out, _ = self.lstm(x, (h0, c0))

        # Aplicar la capa de activación
        out = self.relu(out)

        # Solo se necesita el último hidden state
        out = self.linear(out)

        return 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.

```
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.
    print("Using CUDA")
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False
############

input_size = vocab_size

# Definir hiperparametros del modelo
input_size = 1 # Tamaño de la secuencia de entrada
```

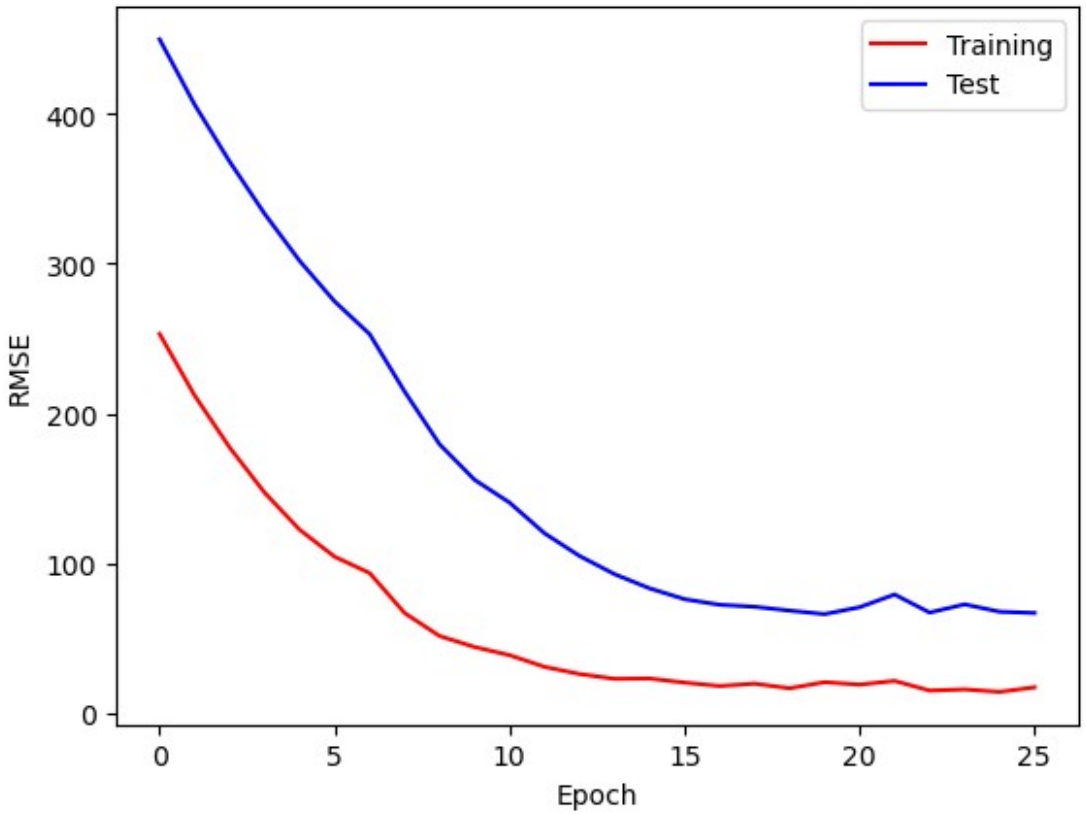
```
hidden_size = 100 # Tamaño del hidden state
num_layers = 2 # Numero de capas LSTM
output_size = 1 # Tamaño de la secuencia de salida

# Inicializar el modelo
model = CustomModelLSTM(input_size, hidden_size, output_size,
num_layers)

# 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_size=8)


# 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_rmse))
```

```
Using CUDA
Epoch 0: train RMSE 253.1210, test RMSE 450.0649
Epoch 100: train RMSE 211.9711, test RMSE 406.4285
Epoch 200: train RMSE 177.1982, test RMSE 368.3467
Epoch 300: train RMSE 146.9849, test RMSE 333.4351
Epoch 400: train RMSE 122.2205, test RMSE 301.9798
Epoch 500: train RMSE 104.1527, test RMSE 274.8271
Epoch 600: train RMSE 93.2511, test RMSE 253.0059
Epoch 700: train RMSE 66.5112, test RMSE 214.5958
Epoch 800: train RMSE 51.0976, test RMSE 179.2880
Epoch 900: train RMSE 43.7838, test RMSE 155.5613
Epoch 1000: train RMSE 38.3152, test RMSE 140.3109
Epoch 1100: train RMSE 30.4900, test RMSE 119.7686
Epoch 1200: train RMSE 25.7066, test RMSE 104.6337
Epoch 1300: train RMSE 22.5012, test RMSE 92.4723
Epoch 1400: train RMSE 22.6997, test RMSE 82.9891
Epoch 1500: train RMSE 19.9896, test RMSE 75.7528
Epoch 1600: train RMSE 17.7032, test RMSE 72.0248
Epoch 1700: train RMSE 19.2737, test RMSE 70.6863
Epoch 1800: train RMSE 16.1867, test RMSE 68.1198
Epoch 1900: train RMSE 20.2639, test RMSE 65.7832
Epoch 2000: train RMSE 18.6453, test RMSE 70.3675
Epoch 2100: train RMSE 21.1314, test RMSE 78.8986
Epoch 2200: train RMSE 14.6561, test RMSE 66.8102
Epoch 2300: train RMSE 15.4514, test RMSE 72.2968
Epoch 2400: train RMSE 13.7791, test RMSE 67.3734
Epoch 2499: train RMSE 16.8427, test RMSE 66.5504
```

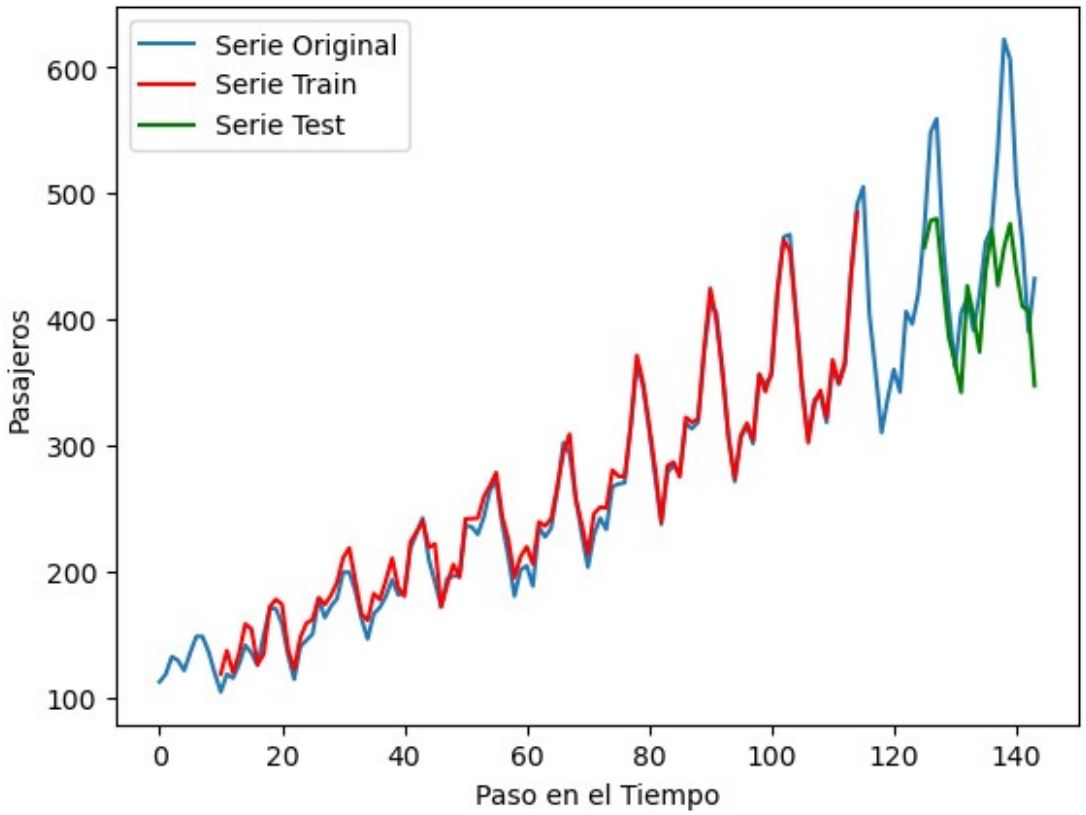
```
# Visualización del rendimiento
epoch = np.arange(len(loss_train))
plt.figure()
plt.plot(epoch, loss_train, 'r', label='Training',)
plt.plot(epoch, loss_test, 'b', label='Test')
plt.legend()
```

```
plt.xlabel('Epoch'), plt.ylabel('RMSE')
plt.show()
```




```
# 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**.

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

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


<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>
```

```
print()
print("La fraccion de abajo muestra su rendimiento basado en las
partes visibles de este laboratorio")
tick.summarise_marks() #
```

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

<IPython.core.display.HTML object>
```