

Laboratorio 3

Francis Aguilar #22243

Cesar Lopez #22535

github: https://github.com/Czar272/Lab3_Deep.git

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

```
In [2]: 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 [3]: `# Seeds`
`seed_ = 2023`
`np.random.seed(seed_)`

In [4]: `# Celda escondida para utilidades 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 declaración 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 declaración que este trabajo es propio (es decir, no hay plagio)

In [5]: `carne_1 = "22535"`
`firma_mecanografiada_1 = "Cesar Lopez"`
`carne_2 = "22243"`
`firma_mecanografiada_2 = "Francis Aguilar"`
`# YOUR CODE HERE`
`# raise NotImplementedError()`

In [6]: `# 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 carácter 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 carácter 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} .

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 [8]: 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 únicos 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 [9]: 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 estamso haciendo predicción 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 imágenes 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 [10]: 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 targetes 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 targets para cada subgroupo
    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 [11]: def one_hot_encode(idx, vocab_size):
    """
    Encoda 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 largo "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):
    """
    Encoda 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}")

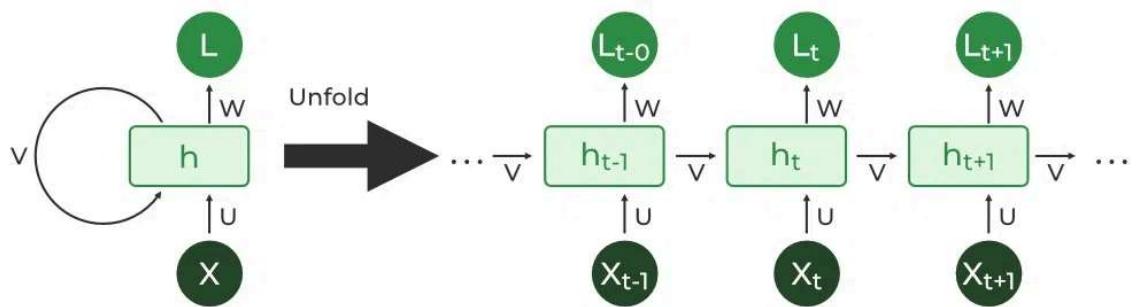
test_sentence = one_hot_encode_sequence(['a', 'b'], vocab_size)
print(f"Encodeado de la secuencia 'a b' con forma {test_sentence.shape}.")
```

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 activacion
- $L_t = softmax(Wh_t)$

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 [12]: 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):
    """
```

```

Inicia la RNN

Args:
    hidden_size: Dimensiones del hidden state
    vocab_size: Dimensión del vocabulario
    """
    # 1. Inicialización de matrices con las formas correctas
    # U: input -> hidden
    U = np.zeros((hidden_size, vocab_size))

    # V: hidden (prev) -> hidden (next)
    V = np.zeros((hidden_size, hidden_size))

    # W: hidden -> output
    W = np.zeros((vocab_size, hidden_size))

    # Bias del hidden state
    b_hidden = np.zeros((hidden_size, 1))

    # Bias de salida
    b_out = np.zeros((vocab_size, 1))

    # 2. Inicialización ortogonal de las matrices de pesos
    U = init_orthogonal(U)
    V = init_orthogonal(V)
    W = init_orthogonal(W)

    # 3. Devolver los parámetros como una tupla
    return U, V, W, b_hidden, b_out

# Ejecutamos la inicialización
params = init_rnn(hidden_size=hidden_size, vocab_size=vocab_size)

```

```

In [13]: 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]

Funciones de Activación

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

```
In [14]: 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))

    if derivative:
        return f * (1 - f)
    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.tanh(x_safe)

    if derivative:
        return 1 - f ** 2
```

```

    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 - np.max(x) # Para estabilidad numérica
    exp_x = np.exp(x_safe)
    f = exp_x / np.sum(exp_x, axis=0, keepdims=True)

    if derivative:
        pass # No se necesita explícitamente para backpropagation con cross-entrop
    else:
        return f

```

```
In [15]: 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 [16]: def forward_pass(inputs, hidden_state, params):
    """
    Calcula el paso forward de RNN
```

Args:

```

inputs: Secuencia de input a ser procesada (lista de vectores one-hot o índice)
hidden_state: Un estado oculto inicial (array de shape [hidden_size, 1])
params: Parámetros de la RNN
"""

# Obtener los parámetros
U, V, W, b_hidden, b_out = params

# Crear listas 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].reshape(-1, 1) # Asegura que sea columna

    # 1. Calcular el hidden state
    hidden_state = tanh(np.dot(U, x_t) + np.dot(V, hidden_state) + b_hidden)

    # 2. Calcular la salida (distribución de probabilidad)
    out = softmax(np.dot(W, hidden_state) + b_out)

    # Guardar resultados
    outputs.append(out)
    hidden_states.append(hidden_state.copy())

return outputs, hidden_states

```

```

In [17]: 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', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b']
```

Secuencia Target:

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

Secuencia Predicha:

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

✓ [5 marks]

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 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 [18]: 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))):

    loss += -np.sum(targets[t] * np.log(outputs[t] + 1e-12)) / len(targets[t])

    d_o = outputs[t].copy()
    d_o[np.argmax(targets[t])] -= 1

    d_W += np.dot(d_o, hidden_states[t].T)
    d_b_out += d_o

    d_h = np.dot(W.T, d_o) + d_h_next
    d_f = d_h * tanh(hidden_states[t], derivative=True)

    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)

    # 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

```

```

In [19]: 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 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

```
In [20]: 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).

```
In [21]: # Hyper parametro
# Se coloca como "respuesta" 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 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 (solo para obtener perdida, no actualizamos)
        loss, _ = backward_pass(inputs_one_hot, outputs, hidden_states, targets_one_hot)

        # 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)

# 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)

# Validar si la pérdida es nan → 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'Epoca {i}, training loss: {training_loss[-1]}, validation loss: {va

```

Epoca 0, training loss: 4.05046509496538, validation loss: 4.801971835967156
 Epoca 100, training loss: 2.729834076574944, validation loss: 3.2320576163982673
 Epoca 200, training loss: 2.109414655736732, validation loss: 2.498052632884415
 Epoca 300, training loss: 1.8235746981413405, validation loss: 2.198677070984531
 Epoca 400, training loss: 1.6884087861997366, validation loss: 2.077078608023496
 Epoca 500, training loss: 1.6129170568126507, validation loss: 2.0163543941716577
 Epoca 600, training loss: 1.5624028954062001, validation loss: 1.9780311638492243
 Epoca 700, training loss: 1.523501919791708, validation loss: 1.9496130467843358
 Epoca 800, training loss: 1.4895828031292173, validation loss: 1.9248315278145824
 Epoca 900, training loss: 1.4558865884071515, validation loss: 1.897822091215436
 Epoca 1000, training loss: 1.4173709332614925, validation loss: 1.860079817655523
 Epoca 1100, training loss: 1.3681783634403946, validation loss: 1.7993697026413986
 Epoca 1200, training loss: 1.3051122158818886, validation loss: 1.7081695076503571
 Epoca 1300, training loss: 1.2330985128125036, validation loss: 1.5999314734390082
 Epoca 1400, training loss: 1.1619900522538607, validation loss: 1.4998577602386731
 Epoca 1500, training loss: 1.1035554777966454, validation loss: 1.4282638416110445
 Epoca 1600, training loss: 1.0680633416284242, validation loss: 1.3958745915871205
 Epoca 1700, training loss: 1.0550402179563658, validation loss: 1.3963674481755954
 Epoca 1800, training loss: 1.0570111001893736, validation loss: 1.4185760443851865
 Epoca 1900, training loss: 1.0640880623573372, validation loss: 1.4524183517051121

In [22]: # 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()

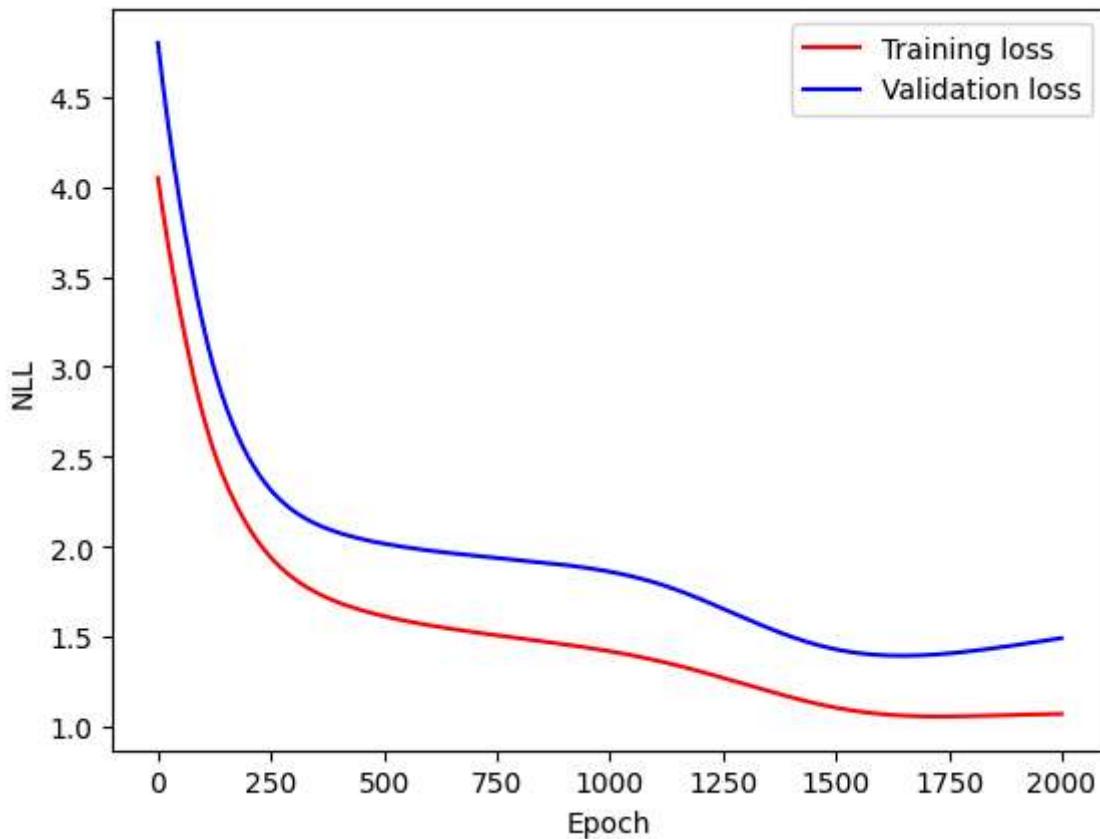
with tick.marks(10):
    assert compare_lists_by_percentage(targets, [idx_to_word[np.argmax(output)]] for

```

Secuencia Input:

Secuencia Target:

Secuencia Predicha:



✓ [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?
- ¿Cree que es un buen modelo basado solamente en el loss?
- ¿Cómo deberían de verse esas gráficas en un modelo ideal?

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:

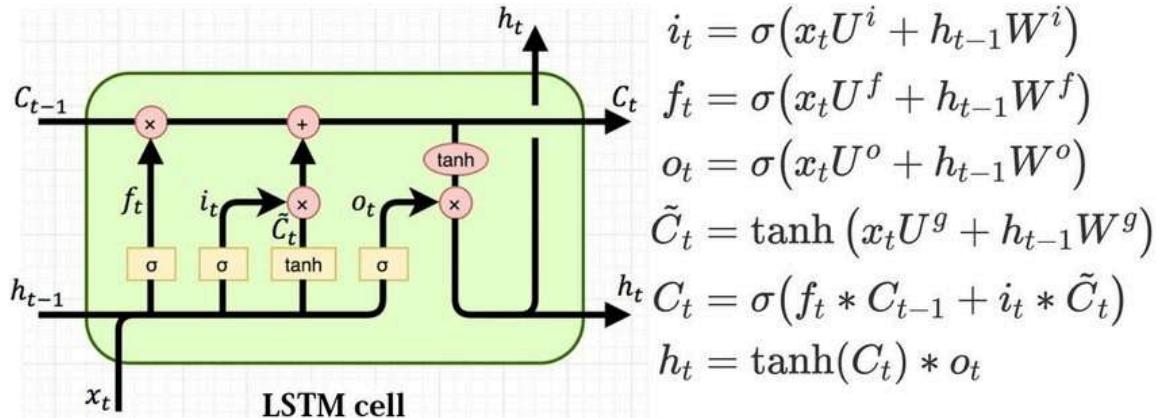
<http://colah.github.io/posts/2015-08-Understanding-LSTMs/>

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 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 "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 $\sigma = \text{softmax}$. 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 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

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

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 calcular 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 [23]: 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
    raise NotImplementedError()

    # 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
    raise NotImplementedError()

    # 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
    raise NotImplementedError()

    # 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
raise NotImplementedError()

# 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
raise NotImplementedError()

# 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)

```

```

-----
NotImplementedError                                     Traceback (most recent call last)
Cell In[23], line 66
      62     W_v = init_orthogonal(W_v)
      64     return W_f, W_i, W_g, W_o, W_v, b_f, b_i, b_g, b_o, b_v
--> 66 params = init_lstm(hidden_size=hidden_size, vocab_size=vocab_size, z_size=z_
size)

Cell In[23], line 21, in init_lstm(hidden_size, vocab_size, z_size)
    7 """
    8     Initializes our LSTM network.
    9     Init LSTM
   10     (...) 14     z_size: Dimensiones del input concatenado
   11     15 """
   16     # Aprox 1 linea para empezar la matriz de pesos de la forget gate
   17     # Recuerden que esta debe empezar con numeros aleatorios
   18     # W_f = np.random.randn
   19     # YOUR CODE HERE
--> 21 raise NotImplementedError()
   22 # Bias del forget gate
   23 b_f = np.zeros((hidden_size, 1))

NotImplementedError:

```

```

In [ ]: 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))

```

Forward

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

```

In [ ]: 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_a, n_v)
        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
    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
raise NotImplementedError()
z_s.append(z)

# Aprox 1 Linea para calcular el forget gate
# Hint: recuerde usar sigmoid
# f =
# YOUR CODE HERE
raise NotImplementedError()
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
raise NotImplementedError()
C_s.append(C_prev)

# Aprox 1 linea para el calculo de la output gate
# Hint: recuerde usar sigmoid
# o =
# YOUR CODE HERE
raise NotImplementedError()
o_s.append(o)

# Calculate hidden state
# Aprox 1 linea para el calculo del hidden state
# h_prev =
# YOUR CODE HERE
raise NotImplementedError()
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 [ ]: # 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))

```

Backward

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

```

In [ ]: 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
        b_v: Bias que relaciona el hidden state con el output, shape (n_a, 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))):

    # Aprox 1 linea para calcular la perdida con cross entropy
    # Loss += ...
    # YOUR CODE HERE
    raise NotImplementedError()

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

    # Compute the derivative of the relation of the hidden-state to the output
    # Calculo de las derivadas en relacion del hidden state al output gate
    dv = np.copy(outputs[t])
    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
    raise NotImplementedError()
    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])
    # Aprox 1 linea para calcular la derivada del output
    # do = ..
    # Hint: Recuerde multiplicar por el valor previo de do (el de arriba)

```

```

# YOUR CODE HERE
raise NotImplementedError()

# Actualización 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]
# Aprox 1 linea de codigo para terminar el calculo de dg
# YOUR CODE HERE
raise NotImplementedError()

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

# Compute the derivative of the input gate and update its gradients
# Calculo de la derivada del input gate y la actualización de sus gradientes
di = dC * g[t]
di = sigmoid(i[t], True) * di
# Aprox 2 lineas para el calculo de los pesos y bias del input gate
# W_i_d +=
# b_i_d +=
# YOUR CODE HERE
raise NotImplementedError()

# Calculo de las derivadas del forget gate y actualización de sus gradientes
df = dC * C_prev
df = sigmoid(f[t]) * df
# Aprox 2 lineas para el calculo de los pesos y bias de la forget gate
# W_f_d +=
# b_f_d +=
# YOUR CODE HERE
raise NotImplementedError()

# Calculo de las derivadas del input y la actualización de gradientes del h
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

```

In []: # 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'))
```

Training

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

```
In [ ]: # 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)

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

        # 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)

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

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

# Actualización 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 épocas
if i % 10 == 0:
    print(f'Epoch {i}, training loss: {training_loss[-1]}, validation loss: {validation_loss[-1]}')

```

```

In [ ]: # 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()
```

Preguntas

Responda lo siguiente dentro de esta celda

- ¿Qué modelo funcionó mejor? ¿RNN tradicional o el basado en LSTM? ¿Por qué?
- Observen la gráfica obtenida arriba, ¿en qué es diferente a la obtenida a RNN? ¿Es esto mejor o peor? ¿Por qué?
- ¿Por qué LSTM puede funcionar mejor con secuencias largas?

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 [ ]: 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 =
        # 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
        # YOUR CODE HERE
        raise NotImplementedError()

        # 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)
```

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

# Init una nueva red
net = Net()

# Aprox 2 Lineas para definir la función de perdida y el optimizador
# criterion = # Use CrossEntropy
# optimizer = # Use Adam con Lr=3e-4
# YOUR CODE HERE
raise NotImplementedError()

# 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)

        # Aprox 1 Linea para el Forward
        # outputs =
        # YOUR CODE HERE
        raise NotImplementedError()

        # Aprox 1 Linea para calcular la perdida
        # Loss =
        # Hint: Use el criterion definido arriba
        # YOUR CODE HERE
```

```

        raise NotImplementedError()

        # 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)

            # Aprox 1 Linea para el Forward
            # outputs =
            # YOUR CODE HERE
            raise NotImplementedError()

            # Aprox 1 Linea para calcular la perdida
            # Loss =
            # Hint: Use el criterion definido arriba
            # YOUR CODE HERE
            raise NotImplementedError()

            # Aprox 3 lineas para definir el backward
            # optimizer.
            # Loss.
            # optimizer.
            # YOUR CODE HERE
            raise NotImplementedError()

            # 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 epochas
            if i % 10 == 0:
                print(f'Epoch {i}, training loss: {training_loss[-1]}, validation loss: {va

```

In []: 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

In [ ]: # 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)

# Aprox 1 linea para el Forward
# outputs =
# YOUR CODE HERE
raise NotImplementedError()

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()

```

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?
- Compare la secuencia target y la predicha de esta parte, ¿en qué parte falló el modelo?
- ¿Qué sucede en el código donde se señala "NOTA 1" y "NOTA 2"? ¿Para qué son necesarias estas líneas?

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.

```
In [ ]: # 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 [ ]: import pandas as pd
```

```
url_data = "https://raw.githubusercontent.com/jbrownlee/Datasets/master/airline-passenger.csv"
dataset = pd.read_csv(url_data)
dataset.head(10)
```

```
In [ ]: # 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 [ ]: # En esta ocasión solo usaremos train y test, validation lo omitiremos para simplicidad
# 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 "características" como el "objetivo" tengan la misma forma. Por ejemplo, para una ventana de tres pasos de tiempo, la "características" 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 [ ]: 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)
```

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.

```
In [ ]: 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
raise NotImplementedError()

class CustomModelLSTM(nn.Module):
    def __init__(self):
        # YOUR CODE HERE
        raise NotImplementedError()
    def forward(self, x):
        # YOUR CODE HERE
        raise NotImplementedError()
        return x
```

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 [ ]: 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 sencilla
loader = data.DataLoader(data.TensorDataset(X_train, y_train), shuffle=False, batch_size=100)

# Perdidas
loss_train = []
loss_test = []

# Iteramos sobre cada época
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 predicción
        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 parámetros del modelo
        optimizer.step()

    # Validación cada 100 épocas
    if epoch % 100 == 0:
        loss_val = 0
        for X_val, y_val in loader:
            y_val_pred = model(X_val)
            loss_val += loss_fn(y_val_pred, y_val).item()
        loss_val /= len(loader)
        print(f'Epoch {epoch}: Validation Loss = {loss_val:.4f}')


# Guardar el modelo
torch.save(model.state_dict(), 'CustomModelLSTM.pth')
```

```

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))

```

In []: # 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 []: # 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 [ ]: 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
```

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

```
In [ ]:
```