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.
!pip3 install -U --force-reinstall --no-cache
        https://github.com/johnhw/jhwutils/zipball/master
!pip3 install scikit-image
!pip3 install -U --force-reinstall --no-cache
        https://github.com/AlbertS789/lautils/zipball/master
```

```
Collecting https://github.com/johnhw/jhwutils/zipball/master
  Downloading https://github.com/johnhw/jhwutils/zipball/master
     - 0 bytes ? 0:00:00
     - 11.6 kB ? 0:00:00
```

```
     \ 38.1 kB 369.0 kB/s 0:00:00
  Preparing metadata (setup.py): started
  Preparing metadata (setup.py): finished with status 'done'
Building wheels for collected packages: jhwutils
  Building wheel for jhwutils (setup.py): started
  Building wheel for jhwutils (setup.py): finished with status 'done'
  Created wheel for jhwutils: filename=jhwutils-1.0-py3-none-any.whl
size=33803
sha256=66c465f45e8161dd31f692f63b4f8549eea46f620e84122ff79801e1e4bb97d
  Stored in directory: C:\Users\charl\AppData\Local\Temp\pip-ephem-
wheel-cache-
pdot5pqt\wheels\27\3c\cb\eb7b3c6ea36b5b54e5746751443be9bb0d73352919033
Successfully built jhwutils
Installing collected packages: jhwutils
  Attempting uninstall: jhwutils
    Found existing installation: jhwutils 1.0
    Uninstalling jhwutils-1.0:
      Successfully uninstalled jhwutils-1.0
Successfully installed jhwutils-1.0



[notice] A new release of pip is available: 23.0.1 -> 23.2.1
[notice] To update, run:
C:\Users\charl\AppData\Local\Microsoft\WindowsApps\PythonSoftwareFounda
-m pip install --upgrade pip
```

```
Requirement already satisfied: scikit-image in
c:\users\charl\appdata\local\packages\pythonsoftwarefoundation.python.
packages\python310\site-packages (0.21.0)
Requirement already satisfied: networkx>=2.8 in
c:\users\charl\appdata\local\packages\pythonsoftwarefoundation.python.
packages\python310\site-packages (from scikit-image) (3.1)
Requirement already satisfied: pillow>=9.0.1 in
c:\users\charl\appdata\local\packages\pythonsoftwarefoundation.python.
packages\python310\site-packages (from scikit-image) (10.0.0)
Requirement already satisfied: packaging>=21 in
c:\users\charl\appdata\local\packages\pythonsoftwarefoundation.python.
packages\python310\site-packages (from scikit-image) (23.1)
Requirement already satisfied: PyWavelets>=1.1.1 in
c:\users\charl\appdata\local\packages\pythonsoftwarefoundation.python.
```

```
packages\python310\site-packages (from scikit-image) (1.4.1)
Requirement already satisfied: tifffile>=2022.8.12 in
c:\users\charl\appdata\local\packages\pythonsoftwarefoundation.python.3
packages\python310\site-packages (from scikit-image) (2023.7.18)
Requirement already satisfied: numpy>=1.21.1 in
c:\users\charl\appdata\local\packages\pythonsoftwarefoundation.python.3
packages\python310\site-packages (from scikit-image) (1.24.3)
Requirement already satisfied: lazy_loader>=0.2 in
c:\users\charl\appdata\local\packages\pythonsoftwarefoundation.python.3
packages\python310\site-packages (from scikit-image) (0.3)
Requirement already satisfied: imageio>=2.27 in
c:\users\charl\appdata\local\packages\pythonsoftwarefoundation.python.3
packages\python310\site-packages (from scikit-image) (2.31.1)
Requirement already satisfied: scipy>=1.8 in
c:\users\charl\appdata\local\packages\pythonsoftwarefoundation.python.3
packages\python310\site-packages (from scikit-image) (1.11.1)



[notice] A new release of pip is available: 23.0.1 -> 23.2.1
[notice] To update, run:
C:\Users\charl\AppData\Local\Microsoft\WindowsApps\PythonSoftwareFounda
-m pip install --upgrade pip
```

```
Collecting https://github.com/AlbertS789/lautils/zipball/master
  Downloading https://github.com/AlbertS789/lautils/zipball/master
     - 0 bytes ? 0:00:00
     - 4.1 kB ? 0:00:00
  Preparing metadata (setup.py): started
  Preparing metadata (setup.py): finished with status 'done'
Building wheels for collected packages: lautils
  Building wheel for lautils (setup.py): started
  Building wheel for lautils (setup.py): finished with status 'done'
  Created wheel for lautils: filename=lautils-1.0-py3-none-any.whl
size=2704
sha256=c8143484d632e8fb6c5776ca6495c6fd8174e9d7bf68b0f2539bb07946b97ae0
  Stored in directory: C:\Users\charl\AppData\Local\Temp\pip-ephem-
wheel-cache-
pojrt_xd\wheels\16\3a\a0\5fbae86e17ef6bb8ed057aa04b591584005d1212c72d69
Successfully built lautils
Installing collected packages: lautils
```

```
  Attempting uninstall: lautils
    Found existing installation: lautils 1.0
    Uninstalling lautils-1.0:
      Successfully uninstalled lautils-1.0
Successfully installed lautils-1.0



[notice] A new release of pip is available: 23.0.1 -> 23.2.1
[notice] To update, run:
C:\Users\charl\AppData\Local\Microsoft\WindowsApps\PythonSoftwareFounda
-m pip install --upgrade pip
```

```
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, 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 = "20347"
firma_mecanografiada_1 = "Alejandro Gómez"
carne_2 =  "20498"
firma_mecanografiada_2 = "Gabriel Vicente"
# YOUR CODE HERE
#raise NotImplementedError()

# 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)
```

✓ [0 marks]

✓ [0 marks]

Parte 1 - Construyendo una Red Neuronal Recurrente

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

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

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 $t_1, ..., t_{n-1}$.

```
# 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):
        # Aprox 2 lineas para agregar
        # word_to_idx[word] =
        # idx_to_word[idx] =
        # YOUR CODE HERE
        word_to_idx[word] = idx
        idx_to_word[idx] = word
        #raise NotImplementedError()

    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'))
```

✓ [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 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
```

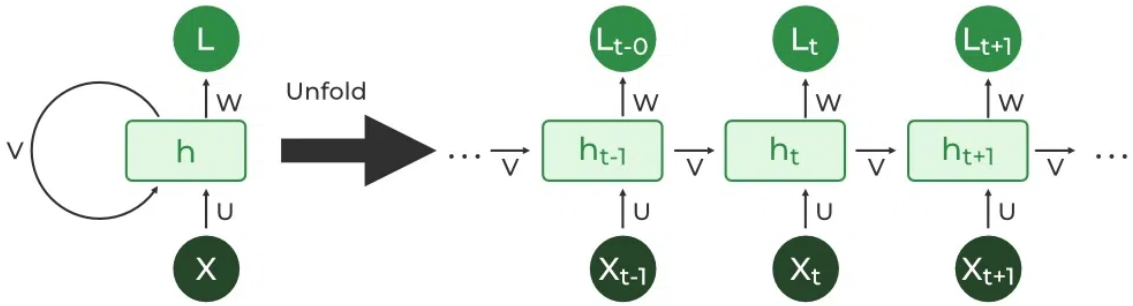
```
    Returns
    np.array de lagro "vocab_size"
    """
    # Init array encodeado
    one_hot = np.zeros(vocab_size)

    # 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

- $h_t = f(Ux_t + Vh_{t-1}$ 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.

```
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))

        # Aprox 3 lineas para inicializar los pesos de forma ortogonal
            usando la
        # funcion init_orthogonal
        U = init_orthogonal(U)
        V = init_orthogonal(V)
        W = init_orthogonal(W)
```

```
    return U, V, W, b_hidden, b_out


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

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

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

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

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

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

✓ [5 marks]

✓ [5 marks]

✓ [5 marks]

✓ [5 marks]

✓ [5 marks]

Funciones de Activación

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

```
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.tanh(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
    e_x = np.exp(x_safe - np.max(x_safe)) # Prevents numerical
        instability
    f = e_x / e_x.sum(axis=0)

    # Regresa la derivada de la funcion
    if derivative:
        pass # No se necesita en backpropagation
    # 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))
```

✓ [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

```
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
        hidden_state = tanh(U @ inputs[t] + V @ hidden_state +
        b_hidden)

        # Para el calculo del output
        # Al ser la salida, deben usar softmax sobre la multiplicación
        de pesos de salida con el hidden_state actual
        out = softmax(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']
```

✓ [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 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
    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
    for t in reversed(range(len(outputs))):
        loss += -np.mean(np.log(outputs[t]+1e-12) * 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 = tanh(hidden_states[t], derivative=True) * d_h
        d_b_hidden += d_f
        d_U += np.dot(d_f, inputs[t].T)
        d_V += np.dot(d_f, hidden_states[t-1].T)
        d_h_next = np.dot(V.T, d_f)

    # 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))
```

✓ [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 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
# YOUR CODE HERE
#raise NotImplementedError()

# 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)

            # Aprox 1 line para el paso forward
            # outputs, hidden_states =
            # YOUR CODE HERE
            outputs, hidden_states = forward_pass(inputs_one_hot,
            hidden_state, params)
            #raise NotImplementedError()

            # Aprox 1 line para el paso backward
            # loss, _ =
            # YOUR CODE HERE
            loss, _ = backward_pass(inputs_one_hot, outputs,
            hidden_states, targets_one_hot, params)
            #raise NotImplementedError()

            # 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)

            # Aprox 1 line para el paso forward
            # outputs, hidden_states =
            # YOUR CODE HERE
            outputs, hidden_states = forward_pass(inputs_one_hot,
            hidden_state, params)
```

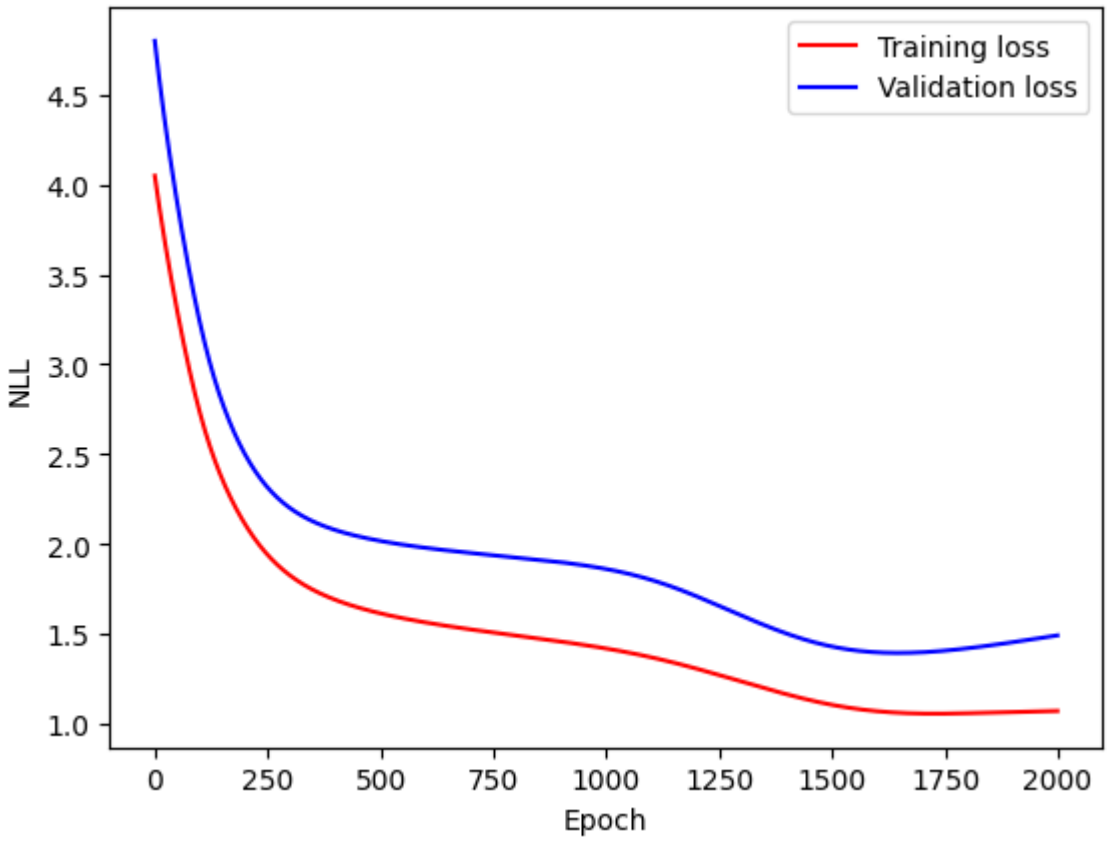
```
            #raise NotImplementedError()


            # Aprox 1 line para el paso backward
            # loss, grads =
            # YOUR CODE HERE
            loss, grads = backward_pass(inputs_one_hot, outputs,
            hidden_states, targets_one_hot, params)
            #raise NotImplementedError()


            # 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: 2.109414655736732, validation loss:
2.4980526328844146
Epoca 300, training loss: 1.823574698141341, validation loss:
2.1986770709845316
Epoca 400, training loss: 1.6884087861997372, validation loss:
2.077078608023497
Epoca 500, training loss: 1.6129170568126512, validation loss:
2.0163543941716586
Epoca 600, training loss: 1.5624028954062004, validation loss:
```

```
1.9780311638492247
Epoca 700, training loss: 1.5235019197917083, validation loss:
1.949613046784337
Epoca 800, training loss: 1.489582803129218, validation loss:
1.9248315278145838
Epoca 900, training loss: 1.4558865884071521, validation loss:
1.8978220912154378
Epoca 1000, training loss: 1.4173709332614932, validation loss:
1.860079817655525
Epoca 1100, training loss: 1.3681783634403957, validation loss:
1.7993697026414015
Epoca 1200, training loss: 1.3051122158818906, validation loss:
1.7081695076503602
Epoca 1300, training loss: 1.2330985128125058, validation loss:
1.5999314734390115
Epoca 1400, training loss: 1.1619900522538622, validation loss:
1.499857760238676
Epoca 1500, training loss: 1.1035554777966472, validation loss:
1.4282638416110474
Epoca 1600, training loss: 1.0680633416284258, validation loss:
1.395874591587123
Epoca 1700, training loss: 1.0550402179563676, validation loss:
1.3963674481755979
Epoca 1800, training loss: 1.0570111001893752, validation loss:
1.41857604438519
Epoca 1900, training loss: 1.064088062357339, validation loss:
1.4524183517051152

# 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', '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', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b',
'b', 'b', 'b', 'b', 'b', 'b', 'b', 'EOS', 'EOS']
```



✓ [10 marks]

Preguntas

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

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

La distancia entre las gráficas de training y validation brindan un entendimiento del comportamiento de este modelo. Una amplia separación entre estas curvas sugiere la posibilidad de que nuestro modelo, ha memorizado las peculiaridades de los datos de entrenamiento, olvidando la esencia subyacente. En términos más coloquiales, podría estar haciendo overfitting. Por otro lado, si las curvas se acercan, tenemos un indicativo de que nuestro modelo no solo ha comprendido los patrones en el conjunto de entrenamiento, sino que también está preparado para enfrentarse con gracia a datos desconocidos.

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

Confundir la métrica de pérdida (o 'loss') como la vara única para medir la calidad de un modelo es un error tan común como juzgar un libro únicamente por su cubierta. Sin embargo, el loss nos da una estimación del error y es un indicativo de cómo el modelo está comprendiendo los datos, pero no es el "oráculo" definitivo del desempeño. Es esencial que no olvidemos considerar otras métricas y, muy importante, exponer el modelo a situaciones inexploradas para realmente comprender su capacidad.

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

En un mundo, donde código y datos se fusionan perfectamente, las curvas de entrenamiento y validación descenderían juntas, en paralelo, hacia los valores bajos. Mstrando que nuestro modelo no solo ha capturado la esencia de los datos de entrenamiento, sino que también está listo para generalizar y aplicar ese conocimiento a escenarios desconocidos. El overfitting sería una rareza, pues el modelo se centraría en aprender la melodía subyacente, en lugar de las peculiaridades de cada nota.

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)

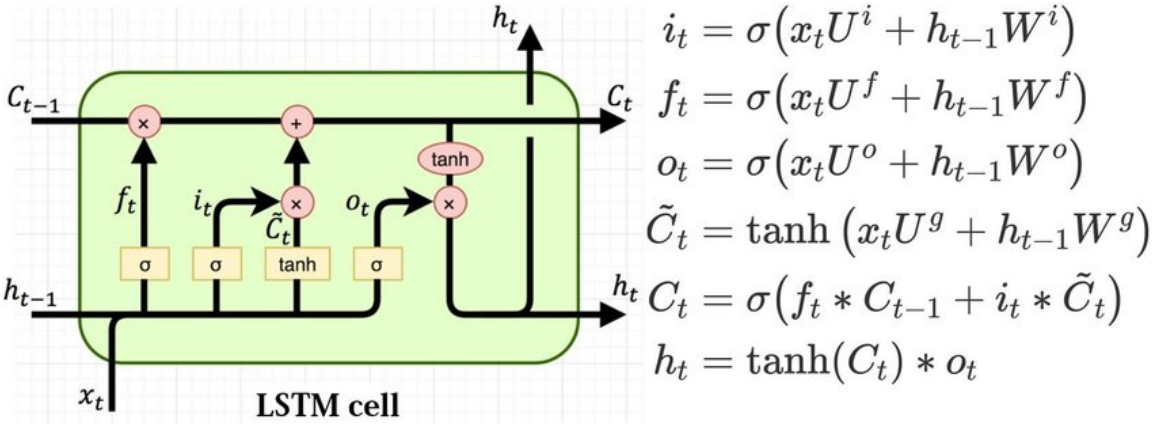
Consideren leer el siguiente blog para mejorar el entendimiento de este tema: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

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

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

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 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
    """

    # 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
    W_f = np.random.randn(hidden_size, z_size)
    #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
    W_i = np.random.randn(hidden_size, z_size)
    #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
    W_g = np.random.randn(hidden_size, z_size)
    #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
    W_o = np.random.randn(hidden_size, z_size)

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


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

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

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

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

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

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

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

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

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

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

✓ [5 marks]

✓ [5 marks]

✓ [5 marks]

✓ [5 marks]

✓ [5 marks]

Forward

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

```
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:
```

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

# Aprox 1 linea para calcular el forget gate
# Hint: recuerde usar sigmoid
# f =
f = sigmoid(np.dot(W_f, z) + b_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 =
C_prev = f * C_prev + i * g# 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 =
o = sigmoid(np.dot(W_o, z) + b_o)# YOUR CODE HERE
#raise NotImplementedError()
o_s.append(o)

# Calculate hidden state
# Aprox 1 linea para el calculo del hidden state
# h_prev =
h_prev = o * tanh(C_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

# 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']
```

✓ [5 marks]

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))):

    # Aprox 1 linea para calcular la perdida con cross entropy
```

```
# loss += ...
loss += -np.mean(np.log(outputs[t]) * targets[t])# 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 gate
# 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 al output gate
# W_v_d +=
W_v_d += np.dot(dv, h[t].T)# 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)
do = do * o[t] * (1 - o[t])# YOUR CODE HERE
#raise NotImplementedError()

# 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]
# Aprox 1 linea de codigo para terminar el calculo de dg
dg = dg * (1 - g[t] ** 2)# 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 += np.dot(di, z[t].T) # W_i_d +=
        b_i_d += di # 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 += np.dot(df, z[t].T) # W_f_d +=
        b_f_d += df # b_f_d +=
        # YOUR CODE HERE
        #raise NotImplementedError()

        # 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:7.637217940763248
```

✓ [5 marks]

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: 2.988603772853675, validation loss:
4.499707061171418
Epoch 10, training loss: 1.2138346782420837, validation loss:
1.443017449173577
Epoch 20, training loss: 0.9205397027246413, validation loss:
1.089850853930022
Epoch 30, training loss: 0.9461880893217114, validation loss:
1.5162333116430573
Epoch 40, training loss: 0.8447097453708239, validation loss:
1.2637189800574518
Epoch 50, training loss: 0.8210537143874097, validation loss:
1.2153374399092132
Epoch 60, training loss: 0.8234789823165087, validation loss:
1.262060603937272
Epoch 70, training loss: 0.8193493781664719, validation loss:
1.2509081934594906
Epoch 80, training loss: 0.825162211305458, validation loss:
1.2643167682476868
Epoch 90, training loss: 0.8335299528338794, validation loss:
1.311726663041251
Epoch 100, training loss: 0.8452056669790936, validation loss:
1.346602278942647
Epoch 110, training loss: 0.8795634139668811, validation loss:
1.5241101691996608
```

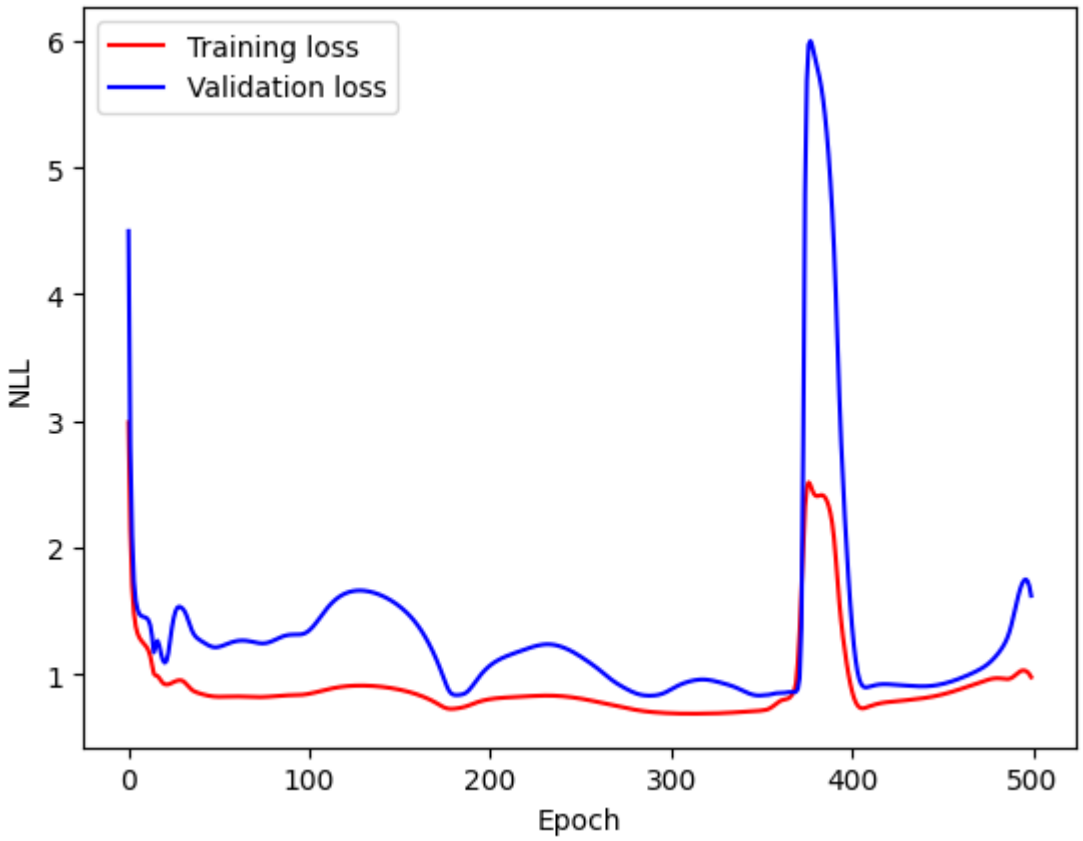
```
Epoch 120, training loss: 0.9038403083892776, validation loss:
1.638356073613571
Epoch 130, training loss: 0.909552503439043, validation loss:
1.6592155285015693
Epoch 140, training loss: 0.900108174017553, validation loss:
1.6201020554944634
Epoch 150, training loss: 0.8771005280496279, validation loss:
1.5335961914126635
Epoch 160, training loss: 0.8373765510273241, validation loss:
1.3828966640202993
Epoch 170, training loss: 0.7731721265634427, validation loss:
1.1227207896063398
Epoch 180, training loss: 0.7266228290665203, validation loss:
0.8363381912487409
Epoch 190, training loss: 0.7630666507434529, validation loss:
0.904597018744143
Epoch 200, training loss: 0.8029831921164708, validation loss:
1.0697653390659076
Epoch 210, training loss: 0.8164207500870427, validation loss:
1.1478987519878623
Epoch 220, training loss: 0.8244645342521512, validation loss:
1.1993745369194524
Epoch 230, training loss: 0.8304826014661837, validation loss:
1.234657318722853
Epoch 240, training loss: 0.8252078588269504, validation loss:
1.2127946099788862
Epoch 250, training loss: 0.8067015254913514, validation loss:
1.1383059179072952
Epoch 260, training loss: 0.7781123570830506, validation loss:
1.0372930086607786
Epoch 270, training loss: 0.7461939570242271, validation loss:
0.9262763135396928
Epoch 280, training loss: 0.7187021141135842, validation loss:
0.8508436113147969
Epoch 290, training loss: 0.6997819849284078, validation loss:
0.8307801697088356
Epoch 300, training loss: 0.6907102716780675, validation loss:
0.8799567484797273
Epoch 310, training loss: 0.6877451077905203, validation loss:
0.9422632200439682
```

```
Epoch 320, training loss: 0.6889083658215852, validation loss:
0.9554895878817234
Epoch 330, training loss: 0.693447642038852, validation loss:
0.922634385839048
Epoch 340, training loss: 0.7019862410717361, validation loss:
0.8662937051050277
Epoch 350, training loss: 0.7126839521404047, validation loss:
0.8321971399701308
Epoch 360, training loss: 0.7858620441269694, validation loss:
0.849475717569258
Epoch 370, training loss: 1.0918171878664036, validation loss:
0.8829595678548301
Epoch 380, training loss: 2.4077346452513635, validation loss:
5.839149321690478
Epoch 390, training loss: 2.0775448816049567, validation loss:
4.370072562113064
Epoch 400, training loss: 0.8646637056092811, validation loss:
1.3785094125607047
Epoch 410, training loss: 0.7473683433698743, validation loss:
0.9021029248653468
Epoch 420, training loss: 0.7794666745794128, validation loss:
0.9199545600421114
Epoch 430, training loss: 0.7940313795381526, validation loss:
0.9099175189065454
Epoch 440, training loss: 0.8114174147364199, validation loss:
0.9039386008160957
Epoch 450, training loss: 0.8397053687697229, validation loss:
0.9224015505689047
Epoch 460, training loss: 0.8822009456520965, validation loss:
0.9680250473404859
Epoch 470, training loss: 0.9320670386445622, validation loss:
1.0327259928213954
Epoch 480, training loss: 0.9676229660412737, validation loss:
1.1501898224017304
Epoch 490, training loss: 0.989126307908189, validation loss:
1.5033645288897612
```

```
# 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', 'EOS', 'EOS', 'EOS']
```



Preguntas

Responda lo siguiente dentro de esta celda

- ¿Qué modelo funcionó mejor? ¿RNN tradicional o el basado en LSTM? ¿Por qué?

El modelo RNN tradicional ha demostrado ser superior en este escenario particular. Una de las principales ventajas de un RNN sobre su contraparte LSTM es su simplicidad inherente. Esta simplicidad no solo reduce la carga computacional, sino que también minimiza el riesgo de sobreajuste, especialmente cuando el conjunto de datos no es extremadamente complejo o extenso.

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

Al examinar las gráficas, se puede observar que el desempeño del LSTM varía más a lo largo de las épocas en comparación con el RNN tradicional. Mientras que el LSTM podría tener el potencial de superar al RNN en ciertos escenarios, en esta instancia específica, la consistencia y simplicidad del RNN lo convierten en la elección óptima.

- ¿Por qué LSTM puede funcionar mejor con secuencias largas?

El diseño subyacente de los LSTM se creó para abordar las deficiencias de los RNNs en el manejo de secuencias más largas, en particular, el problema del desvanecimiento del gradiente. Los LSTM cuentan con una estructura de puertas y celdas de memoria que les permite decidir qué información retener y cuál olvidar en cada paso temporal. Esta capacidad le da una ventaja significativa en tareas que requieren considerar dependencias a largo plazo, como el procesamiento del lenguaje natural o la predicción de series temporales. Sin embargo, su eficacia dependerá en última instancia del contexto y de la naturaleza de los datos con los que se esté trabajando.

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

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
        self.lstm = nn.LSTM(input_size=vocab_size,
                            hidden_size=50,
                            num_layers=1,
```

```
                                bidirectional=False)
        #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)
```

```
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()

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

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

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

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

        # Aprox 3 lineas para definir el backward
        # optimizer.
        # loss.
        # optimizer.
        # YOUR CODE HERE
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

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

```
Epoch 0, training loss: 1.348198813199997, validation loss:
1.4144882917404176
Epoch 10, training loss: 0.5598714828491211, validation loss:
0.5109817296266556
Epoch 20, training loss: 0.41817260831594466, validation loss:
0.3748889058828354
Epoch 30, training loss: 0.36341856271028516, validation loss:
0.3150088548660278
Epoch 40, training loss: 0.33668678179383277, validation loss:
0.293888683617115
Epoch 50, training loss: 0.3205028915777802, validation loss:
0.28407980799674987
Epoch 60, training loss: 0.31017738599330186, validation loss:
0.274866783618927
Epoch 70, training loss: 0.304358615167439, validation loss:
0.2709150478243828
Epoch 80, training loss: 0.3286062298342586, validation loss:
0.3390388160943985
Epoch 90, training loss: 0.29831806998699906, validation loss:
0.2679189458489418
Epoch 100, training loss: 0.29660330787301065, validation loss:
0.2667516052722931
Epoch 110, training loss: 0.29525244552642105, validation loss:
0.26602641940116883
Epoch 120, training loss: 0.2942402884364128, validation loss:
0.26556051671504977
Epoch 130, training loss: 0.29346368424594405, validation loss:
```

```
0.26726687103509905
Epoch 140, training loss: 0.2929681332781911, validation loss:
0.26610356420278547
Epoch 150, training loss: 0.29263029955327513, validation loss:
0.26574755311012266
Epoch 160, training loss: 0.29232770670205355, validation loss:
0.26576957702636717
Epoch 170, training loss: 0.292026955075562, validation loss:
0.26603140532970426
Epoch 180, training loss: 0.29174363501369954, validation loss:
0.2663534492254257
Epoch 190, training loss: 0.29085528254508974, validation loss:
0.2681730806827545
Epoch 200, training loss: 0.29083405788987876, validation loss:
0.2672805920243263
Epoch 210, training loss: 0.2908763142302632, validation loss:
0.2671742781996727
Epoch 220, training loss: 0.29087852481752635, validation loss:
0.26734518110752103
Epoch 230, training loss: 0.2908420303836465, validation loss:
0.2676637753844261
Epoch 240, training loss: 0.2907707057893276, validation loss:
0.26804947555065156
Epoch 250, training loss: 0.33269670959562064, validation loss:
0.2687272742390633
Epoch 260, training loss: 0.2899562789127231, validation loss:
0.2693792715668678
Epoch 270, training loss: 0.29008453730493783, validation loss:
0.2691534891724586
Epoch 280, training loss: 0.29016607981175185, validation loss:
0.26927988678216935
Epoch 290, training loss: 0.2902008067816496, validation loss:
0.2695646956562996
Epoch 300, training loss: 0.2901948308572173, validation loss:
0.2699388787150383
Epoch 310, training loss: 0.2901601163670421, validation loss:
0.27037187069654467
Epoch 320, training loss: 0.29010991640388967, validation loss:
0.2707863196730614
Epoch 330, training loss: 0.2911234933882952, validation loss:
```

```
0.2768925681710243
Epoch 340, training loss: 0.28946690894663335, validation loss:
0.2725307419896126
Epoch 350, training loss: 0.28951781447976827, validation loss:
0.27186423540115356
Epoch 360, training loss: 0.2895985659211874, validation loss:
0.2718219250440598
Epoch 370, training loss: 0.2896580878645182, validation loss:
0.2720210999250412
Epoch 380, training loss: 0.2896872444078326, validation loss:
0.27232115119695666
Epoch 390, training loss: 0.28969427235424516, validation loss:
0.27265151143074035
Epoch 400, training loss: 0.2896821966394782, validation loss:
0.2729654982686043
Epoch 410, training loss: 0.2896499678492546, validation loss:
0.27324529737234116
Epoch 420, training loss: 0.2895962970331311, validation loss:
0.2735102534294128
Epoch 430, training loss: 0.2887963453307748, validation loss:
0.27459167689085007
Epoch 440, training loss: 0.28891611453145744, validation loss:
0.27437804043293
Epoch 450, training loss: 0.28903115522116424, validation loss:
0.27429051101207735
Epoch 460, training loss: 0.28911731000989677, validation loss:
0.27426301389932634
Epoch 470, training loss: 0.2891741365194321, validation loss:
0.27427901774644853
Epoch 480, training loss: 0.2892046818509698, validation loss:
0.27432120144367217
Epoch 490, training loss: 0.28921155761927364, validation loss:
0.27437656074762345
```

```
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')
```

✓ [5 marks]

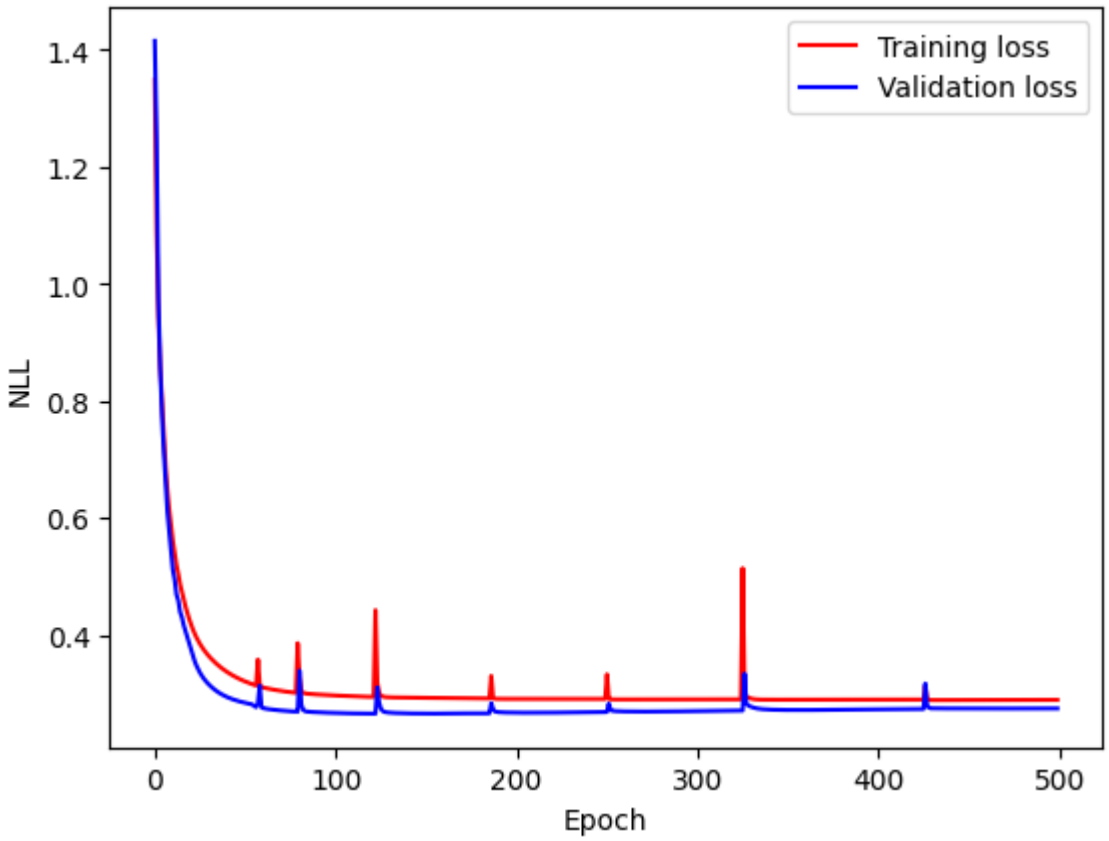
✓ [5 marks]

```
# 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
outputs = net(inputs_one_hot)
outputs = F.softmax(outputs, dim=1)
#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])
```

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

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

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

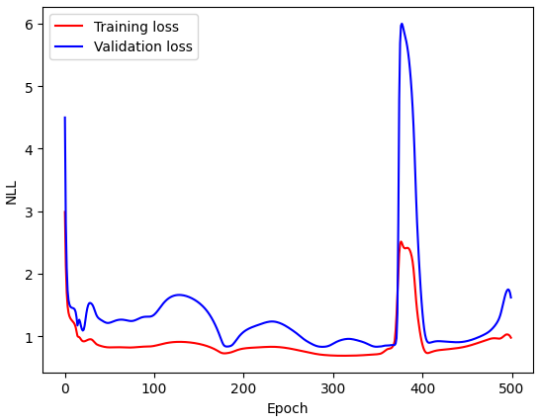
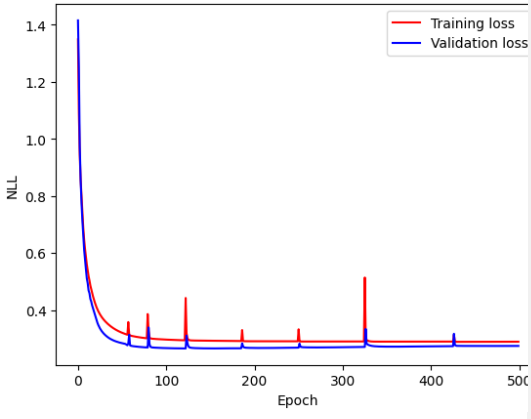


Preguntas

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?

R/. Para poder comparalas de manera más cómoda, se agregan a continuación con el fin de tener fácil acceso a estas

LSTM "a mano"	LSTM "usando pytorch"
	

Inmediatamente se puede ver una diferencia considerable en lo que a corelacion entre "Training loss" y "Validation loss" se refiere, por lo que se puede argumentar que LSTM "usando pytorch" es mejor.

Esto se puede deber a los niveles de precisión que maneja la liberia a comparación de los calculos en python, además de que en ciertas partes de los procesos a mano se puede perder detalle.

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

  R/. En los outputs de la celda anterior, se puede apreciar que esta falla en lo que corresponde al punto en que las 'a's se convierten en 'b's

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

  R/. Mientras que "NOTA 1" corresponde al proceso de recibir el input y target y pasarlo a un tensor y de ahi recibir la actualización de perdida para cada secuencia en el validation set, "NOTA 2" corresponde al mismo proceso pero para cada secuencia dentro del training set

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.

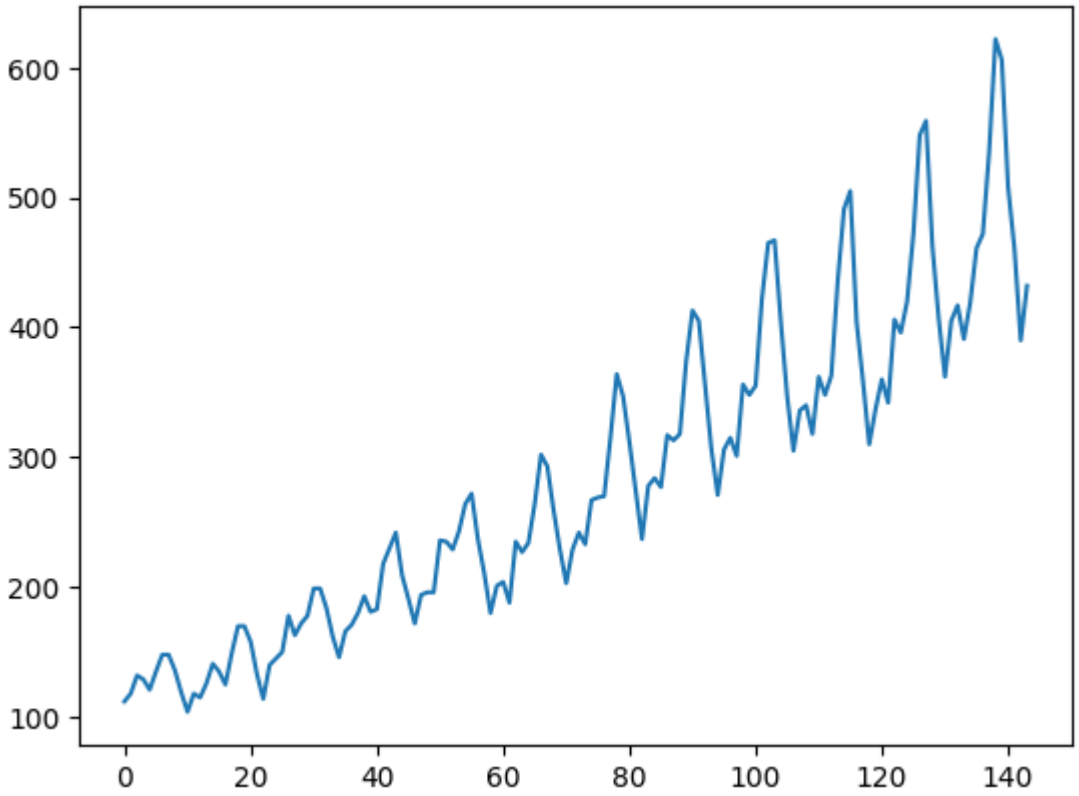
En este parte haremos uso de las redes LSTM pero para predicción de series de tiempo. Entonces lo que se busca es que dado un mes y un año, se debe predecir el número de pasajeros en unidades de miles. Los datos a usar son de 1949 a 1960.

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

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

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

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

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

C:\Users\charl\AppData\Local\Temp\ipykernel_43384\2018909527.py:10:
UserWarning: Creating a tensor from a list of numpy.ndarrays is
extremely slow. Please consider converting the list to a single
numpy.ndarray with numpy.array() before converting to a tensor.
(Triggered internally at ..\torch\csrc\utils\tensor_new.cpp:248.)
  return torch.tensor(X), torch.tensor(y)
```

Ahora necesitamos crear una clase que definirá nuestro modelo de red neuronal con LSTM. Noten que acá solo se 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 = 4000
# YOUR CODE HERE
# raise NotImplementedError()


# class CustomModelLSTM(nn.Module):
#     def __init__(self):
#         # YOUR CODE HERE
#         # raise NotImplementedError()
#         super().__init__()
#         self.lstm = nn.LSTM(input_size=1, hidden_size=50,
        num_layers=1, batch_first=True)
#         self.linear = nn.Linear(50, 1)
#     def forward(self, x):
#         # YOUR CODE HERE
#         # raise NotImplementedError()
#         # return x
#         x, _ = self.lstm(x)
#         x = self.linear(x)
#         return x
class CustomModelLSTM(nn.Module):
    def __init__(self):
        super().__init__()
        input_size = 1
        hidden_size = 50
        num_layers = 2
        bidirectional = True

        self.lstm = nn.LSTM(input_size=input_size,
        hidden_size=hidden_size, num_layers=num_layers,
        batch_first=True, bidirectional=bidirectional)
        self.dropout = nn.Dropout(0.2)
        self.linear = nn.Linear(hidden_size * 2 if bidirectional else
        hidden_size, 1)


    def forward(self, x):
        x, _ = self.lstm(x)
        x = self.dropout(x)
```

```
        x = self.linear(x)
        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.

```
import torch.optim as optim
import torch.utils.data as data

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

model = CustomModelLSTM()
# Optimizador y perdida
optimizer = optim.Adam(model.parameters())
loss_fn = nn.MSELoss()
# Observen como podemos también definir un DataLoader de forma
        snecilla
loader = data.DataLoader(data.TensorDataset(X_train, y_train),
        shuffle=False, batch_size=8)
```

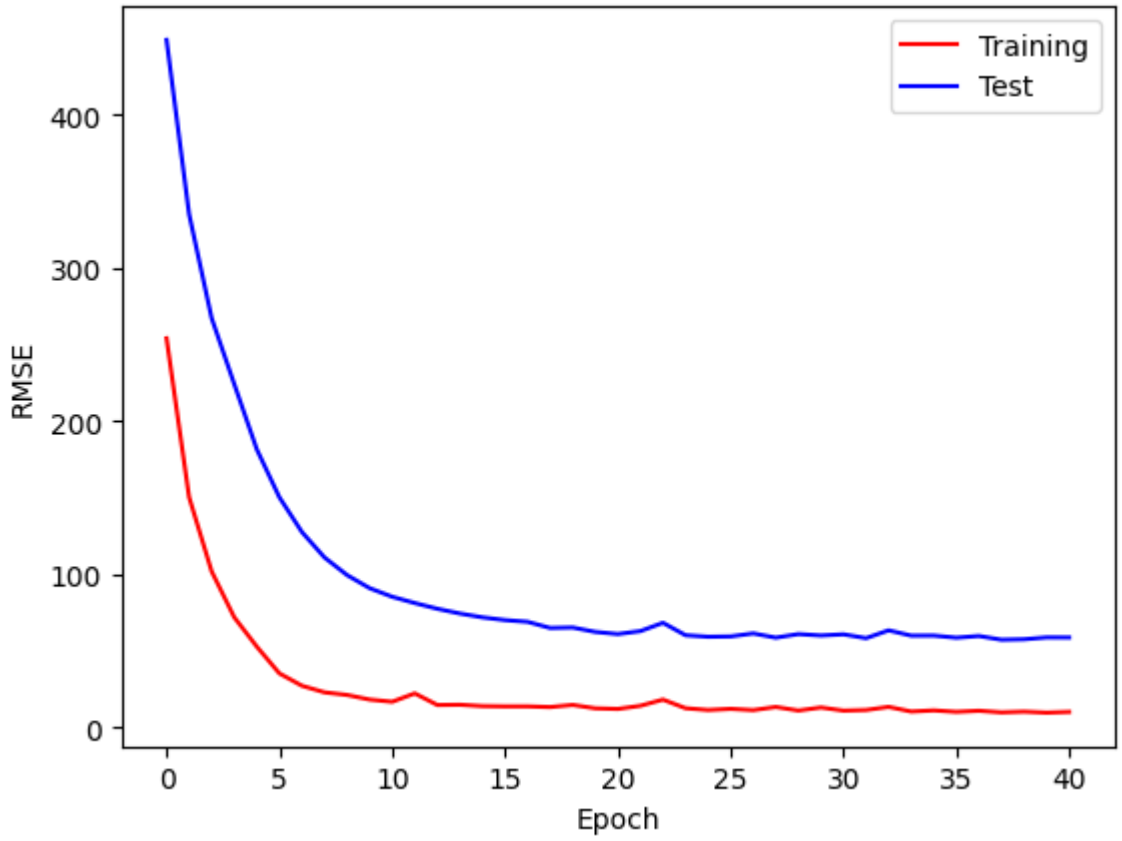
```
# Perdidas
loss_train = []
loss_test = []

# 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))
```

```
Epoch 0: train RMSE 254.1170, test RMSE 448.9859
Epoch 100: train RMSE 150.3338, test RMSE 335.1707
Epoch 200: train RMSE 101.8262, test RMSE 267.1939
Epoch 300: train RMSE 71.8909, test RMSE 224.2183
Epoch 400: train RMSE 52.7294, test RMSE 181.5784
Epoch 500: train RMSE 35.2587, test RMSE 150.2749
Epoch 600: train RMSE 27.1761, test RMSE 127.5523
Epoch 700: train RMSE 22.8643, test RMSE 110.9470
Epoch 800: train RMSE 21.1688, test RMSE 99.4444
Epoch 900: train RMSE 18.1533, test RMSE 90.9114
Epoch 1000: train RMSE 16.7192, test RMSE 85.2862
Epoch 1100: train RMSE 22.1860, test RMSE 81.2006
Epoch 1200: train RMSE 14.6047, test RMSE 77.4486
Epoch 1300: train RMSE 14.7741, test RMSE 74.3524
Epoch 1400: train RMSE 13.8799, test RMSE 71.8381
Epoch 1500: train RMSE 13.7004, test RMSE 70.0551
Epoch 1600: train RMSE 13.7066, test RMSE 68.9328
Epoch 1700: train RMSE 13.2695, test RMSE 64.9038
Epoch 1800: train RMSE 14.6497, test RMSE 65.2097
Epoch 1900: train RMSE 12.4337, test RMSE 62.3382
Epoch 2000: train RMSE 12.0386, test RMSE 60.9419
Epoch 2100: train RMSE 14.0973, test RMSE 62.8782
Epoch 2200: train RMSE 18.1339, test RMSE 68.2894
Epoch 2300: train RMSE 12.5044, test RMSE 60.1656
Epoch 2400: train RMSE 11.3381, test RMSE 59.1912
Epoch 2500: train RMSE 12.0881, test RMSE 59.3898
Epoch 2600: train RMSE 11.2627, test RMSE 61.3556
Epoch 2700: train RMSE 13.3345, test RMSE 58.5820
Epoch 2800: train RMSE 10.9882, test RMSE 60.9149
Epoch 2900: train RMSE 12.9894, test RMSE 60.0054
Epoch 3000: train RMSE 10.9358, test RMSE 60.7575
Epoch 3100: train RMSE 11.3355, test RMSE 58.2408
Epoch 3200: train RMSE 13.3795, test RMSE 63.3641
Epoch 3300: train RMSE 10.3484, test RMSE 59.9070
Epoch 3400: train RMSE 11.0857, test RMSE 59.9422
```

```
Epoch 3500: train RMSE 10.1330, test RMSE 58.5551
Epoch 3600: train RMSE 10.8447, test RMSE 59.6736
Epoch 3700: train RMSE 9.7848, test RMSE 57.2224
Epoch 3800: train RMSE 10.2627, test RMSE 57.5288
Epoch 3900: train RMSE 9.6011, test RMSE 58.7403
Epoch 3999: train RMSE 10.0920, test RMSE 58.6863
```

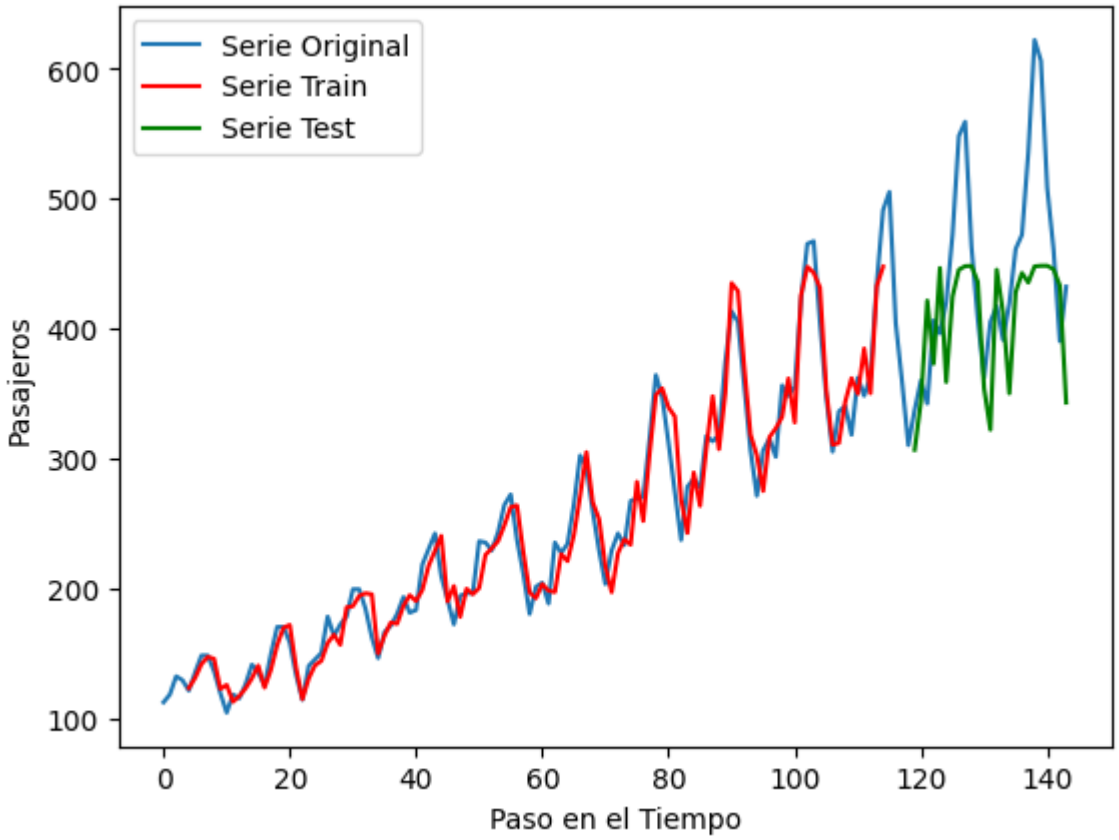
```python
# Visualización del rendimiento
epoch = np.arange(len(loss_train))
plt.figure()
plt.plot(epoch, loss_train, 'r', label='Training',)
plt.plot(epoch, loss_test, 'b', label='Test')
plt.legend()
plt.xlabel('Epoch'), plt.ylabel('RMSE')
plt.show()
```




```python
# 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
```

✓ [7 marks]

✓ [7 marks]

✓ [7 marks]

✓ [7 marks]

```
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
```

158 / 158 marks (100.0%)